به نام پروردگار

مقدمه‌ای بر مسابقات برنامه‌نویسی

نویسندگان: احمد یوسفان (عضو هیئت علمی دانشگاه کاشان)، محسن بیگلری، فائزه میرزائی، امین بابادی

نشر شاسوسا

عنوان و نام پدیدآور:	مقدمه‌ای بر مسابقات برنامه‌نویسی/ نویسندگان احمد یوسفان ... [و دیگران].
مشخصات نشر:	کاشان: شاسوسا، ۱۳۹۵.
مشخصات ظاهری:	۴۷۷ ص.: مصور، جدول، نمودار.
شابک:	۹۷۸-۶۰۰-۶۴۴۰-۸۷-۳
وضعیت فهرست نویسی:	فیپا
یادداشت:	نویسندگان احمد یوسفان ، محسن بیگلری ، فائزه میرزائی ، امین بابادی .
یادداشت:	واژه‌نامه .
موضوع:	برنامه‌نویسی -- مسابقه‌ها
موضوع:	Computer programming -- Competitions
شناسه افزوده:	یوسفان نجف‌آبادی، احمد، ۱۳۵۴ -
رده بندی کنگره:	QA ۷۶/۶/م۷ ۱۳۹۵
رده بندی دیویی:	۰۰۱/۶۴۲
شماره کتابشناسی ملی:	۴۳۳۰۵۹۰

مقدمه‌ای بر مسابقات برنامه‌نویسی

چاپ یکم

شمارگان : ۱۰۰۰

بها: ۱۵۰۰۰ تومان

نشر شاسوسا: کاشان ـ میدان امام حسین(ع) ـ مجتمع الماس شهر ـ واحد ۱۰۲؛ صندوق پستی: ۳۱۳۵ ـ ۸۷۱۳۵

تلفکس: ۵۵۳۱۹۷۷۰ ـ ۰۳۱؛ همراه: ۰۹۳۶۵۱۱۱۳۶۵ پست الکترونیکی: shasoosapub@yahoo.com

Website: http://yoosofan.kashanu.ac.ir/acm-book.html
Website: http://yoosofan.github.io/acm-book.html
Email: yoosofan@kashanu.ac.ir , yoosofan@mail.com, yoosofan@gmail.com
Website: http://mbt925.ir
Email: mbt925@gmail.com
Email: mirzaeifaezeh@gmail.com
Email: amin.babadi@yahoo.com

پیشگفتار

این کتاب مجموعه‌ای کامل از ابزارهای موردِ نیاز برای تبدیل شدن به یک برنامه‌نویس کارآمد و حرفه‌ای برای حل مسأله‌های گوناگون الگوریتمی است. همچنین به نوعی کامل کننده‌ی درس‌های برنامه‌نویسی، ساختمان داده و طراحی الگوریتم است و دربردارنده‌ی نکته‌های ساده و همچنین دشواری است که اغلب در این درس‌ها به آنها کمتر پرداخته می‌شود ولی برنامه‌نویس به آنها نیاز دارد. کتاب حاضر خواننده را برای مسابقه‌های برنامه‌نویسی مانند ای–سی–ام آماده می‌سازد. این کتاب دربردارنده‌ی هفت فصل است. در فصل‌های آغازین به معرفی مفاهیم اصلی و همچنین آشنایی با ورودی و خروجی استاندارد، کار با پرونده، میانگیرها و چگونگی به کارگیری آن‌ها پرداخته می‌شود. سپس به کتابخانه‌های استاندارد سی++ و ساختمان‌داده‌های کاربردی پرداخته می شود. در ادامه مجموعه‌ای متنوع از الگوریتم‌ها بررسی شده و به کمک ساختمان‌داده‌های معرفی شده، پیاده‌سازی آن‌ها انجام می شود؛ در این میان، به بررسی نکات ظریفی که در هر یک نهفته است نیز پرداخته می‌شود. ترتیب فصل‌های کتاب به شکلی برگزیده شده است که خواننده همراه با کتاب، سطح خود را بهبود بخشیده و به طور کامل با کتاب همراه شود. در این کتاب، دسته‌ای گسترده از الگوریتم‌ها پیاده‌سازی و بررسی می‌شود.

پیشگفتار نشر الکترونیکی

با هماهنگی میان نویسندگان و ناشر، این کتاب افزون بر نسخه‌ی چاپی به صورت الکترونیکی نیز منتشر شده است. امید است بتواند کمکی برای بهبود سطح برنامه‌نویسان و آشنایی بهتر با مسابقات برنامه‌نویسی باشد.

اگر کتاب را ارزشمند و سودمند یافتید می‌توانید برای تشویقِ نویسندگان برای انتشار آثار بیشتر و بهتری به این شکل، کمکی کنید یا نسخه‌ی چاپی کتاب را خریداری نمایید.

پخش الکترونیکی این کتاب به شرطِ حفظ یکپارچی و عدم تغییر در آن (به هر شکلی) کاملا آزاد است؛ روشن است که برداشتن مطالبی از این کتاب بدون ارجاع به آن و پخش کردنِ آن به نام خود، کاری ناپسند و غیرقانونی است.

کارت بانک ملی از محسن بیگلری و فائزه میرزائی

6037997123309537

حساب paypal از امین بابادی

http://paypal.me/AminBabadi

شماره‌های بیت‌کوین، ایتریوم و monero از احمد یوسفان

https://blockexplorer.com/address/1yQUAqDRftfKEH6opaN5hhPiyabmgbPsY
https://etherscan.io/address/0x25EF958A163fe8bee6a321BC6839Ea072119790B
43hRo59AGvT1xueyaNyPgx56Co1BGmn4zgMUXzN5tc9vJCRMBnXcxkGgJpwedFNYpA41vGv5HepoN53JVhPc
faE6GrvBj7W

فهرست مطالب

فصل ۱- مقدمه

۱-۱- مسابقات برنامه‌نویسی

برنامه‌نویسی یک مهارت بسیار لذت‌بخش و سودمند است و به همین دلیل یکی از جذاب‌ترین فعالیت‌ها در حوزه‌ی رایانه محسوب می‌شود. سال‌ها پیش گمان می‌رفت که با پیشرفت فناوری، برنامه‌نویسی به طور کامل خودکار شده و به این ترتیب برنامه‌نویسی توسط انسان کنار گذاشته خواهد شد و مفاهیم سطح بالاتر در مهندسی نرم‌افزار اهمیت پیدا خواهند کرد. با این حال این رخداد پیش نیامد و برعکس هر روز بر میزان اهمیت و ارزش برنامه‌نویسی افزوده شد. کد برنامه به نوعی نشان‌دهنده‌ی نیازمندی‌های برنامه است که به شکل مناسبی بیان شده است تا بتواند به خوبی نیازهای کاربر را برآورده کند. ممکن است که با پیشرفت فناوری بتوانیم زبان‌هایی طراحی کنیم که به نیازهای کاربر نزدیک‌تر باشند. ولی حتی در آن صورت، همچنان نیاز به برنامه‌نویسی خواهیم داشت. به همین دلیل جای تعجب نیست که امروزه به طور پیاپی شاهد برگزاری مسابقات برنامه‌نویسی مختلف از جمله «مسابقه‌ی اِی-سی-ام[۱]» هستیم.

در مسابقات برنامه‌نویسی معمولا تعدادی مسأله در اختیار شرکت‌کنندگان قرار داده می‌شود و از آنها خواسته می‌شود که در کوتاه‌ترین زمان ممکن آن مسائل را حل کنند. شرکت‌کنندگان باید جواب هر سوال را در قالب یک قطعه کد برنامه‌نویسی برای داوران ارسال کنند تا داوران بتوانند به ارزیابی آن بپردازند. در نهایت تیم‌ها بر اساس تعداد سوالات حل شده و زمانی که برای حل آنها صرف کرده‌اند رده‌بندی شده و برنده مشخص می‌شود. فصل هفتم به تفصیل به جزییات مسابقات اِی-سی-ام می‌پردازد.

۱-۱-۱- یک مسأله‌ی ساده

مسائلی که در مسابقات برنامه‌نویسی مطرح می‌شوند، عموما ریاضیاتی یا منطقی هستند و حل این مسائل نیاز به داشتن اطلاعات کافی در زمینه‌های مختلفی همچون ساختمان داده، الگوریتم، ترکیبیات، نظریه گراف و هندسه دارد. در ادامه یک مثال بسیار ساده از این مسائل آمده است.

---

[۱] ACM Contest

به عنوان مثال، دنباله فیبوناچی را در نظر بگیرید. دنباله فیبوناچی یک دنباله بسیار معروف در ریاضیات است که به این صورت تعریف می‌شود: دو عدد اول دنباله برابر ۱ هستند. اعداد بعدی هم از جمع دو عدد پیشین خود ساخته می شوند. به این ترتیب، ده عدد اول دنباله فیبوناچی از راست به چپ عبارتند از:

۱، ۱، ۲، ۳، ۵، ۸، ۱۳، ۲۱، ۳۴، ۵۵

حال فرض کنید در صورت مسأله از شما خواسته می‌شود تا برنامه‌ای بنویسید که بتواند *n* عدد نخست دنباله فیبوناچی را تولید کند؛ به طوری که *n* یک عدد دلخواه باشد که به عنوان ورودی به برنامه شما داده می‌شود. در ادامه کد مربوط به پاسخ این سوال نشان داده شده است.

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;
    int a, b, c;
    a = 1;
    b = 0;
    for(int i = 0; i < n; i++) {
        c = a + b;
        a = b;
        b = c;
        if(i > 0)
            cout << " ";
        cout << c;
    }
    cout << endl;
    return 0;
}
```

از دستور cin و cout به ترتیب برای خواندن ورودی و چاپ کردن در خروجی استفاده می‌شود (برای جزییات بیشتر در مورد این دستورات به فصل دوم مراجعه کنید). همانطور که می‌بینید، کد نشان داده شده ساختار بسیار ساده‌ای دارد. در این کد سه متغیر به نام‌های *a*، *b* و *c* تعریف شده است که از آنها برای نگهداری دو عدد پیشین دنباله (متغیرهای *a* و *b*) و محاسبه عدد بعدی (متغیر *c*) استفاده می‌شود. نکته جالب توجه این است که مقدار اولیه متغیر *b* در این کد برابر صفر قرار داده شده است در حالی که طبق تعریف، دو عدد نخست در دنباله فیبوناچی برابر یک هستند! این مقداردهی با این که در نگاه نخست ممکن است یک اشتباه ساده به نظر برسد، کاملا درست بوده و به منظور ساده‌سازی کد به این صورت انجام شده است (تحلیل این نکته به عنوان یک تمرین ذهنی ساده به خواننده واگذار می‌شود).

همانطور که گفته شد، مثال صفحه قبل یک نمونه بسیار ساده از مسائلی است که ممکن است در یک مسابقه برنامه نویسی مطرح شود. در واقع، معمولا بیش از ۹۰٪ مسائلی که در یک مسابقه برنامه‌نویسی مطرح می‌شوند، بسیار دشوارتر از این مثال هستند و حل آنها نیازمند تسلط بر تکنیک‌های حل مسأله و برنامه‌نویسی است. به همین دلیل، در این کتاب تکنیک‌های پرکاربرد مورد نیاز برای شرکت در مسابقات برنامه‌نویسی پوشش داده می‌شود.

۱-۱-۲- کاربردها

یکی از مهمترین سوالاتی که ممکن است هنگام آشنایی با مسابقات برنامه‌نویسی برای شخص پیش بیاید، این است که کاربرد این مسابقات چیست؟ به عنوان مثال، یک برنامه‌نویس در چند درصد مواقع نیاز به نوشتن برنامه‌ای برای تولید دنباله فیبوناچی دارد؟! در این قسمت سعی می‌کنیم پاسخی کوتاه به این سوالات بدهیم.

حقیقت این است که با پیشرفت فناوری، امروزه ابزارهای زیادی برای تولید نرم‌افزار به صورت خودکار (حتی به رایگان) در اختیار برنامه‌نویسان و عموم مردم قرار دارد. به همین دلیل، ممکن است در بسیاری از موارد شخص بتواند حتی بدون داشتن دانش پایه‌ی برنامه‌نویسی، نرم‌افزارهای مختلفی تولید کند. یکی از بهترین مثال‌ها از این دسته ابزارها، نرم‌افزارهای تولید خودکار وب سایت‌های اینترنتی هستند. با استفاده از این نرم‌افزارها، شخص می‌تواند تنها با وارد کردن محتوای مورد نظر خود (که به هیچ دانشی از برنامه‌نویسی نیاز ندارد) یک وب سایت اینترنتی کامل ایجاد کند. ولی ناامید نشوید! خبر خوب این است حتی با وجود این ابزارها هم در دنیای نرم‌افزار موقعیت‌های بیشماری به وجود می‌آید که نیاز شدیدی به برنامه‌نویسان احساس می‌شود؛ زیرا دنیای نرم‌افزار به سرعت در حال تغییر است و همواره نیاز به برنامه‌نویسانی که این تغییرات را ممکن کنند، وجود دارد. به عنوان مثال، سایت گوگل[۲] را در نظر بگیرید و سعی کنید به این سوال پاسخ دهید که این سایت با این که در اصل به عنوان یک موتور جستجوگر اینترنتی شناخته می‌شود، در سال‌های اخیر چه سرویس‌هایی را برای نخستین بار در دنیای اینترنت ارائه داده است؟ به طور حتم ارائه این سرویس‌ها بدون برنامه‌نویسان حرفه‌ای امکان‌پذیر نیست.

یکی از مهمترین عوامل در تعیین سطح یک برنامه‌نویس، توانایی حل مسأله اوست. مهمترین فایده‌ای که مسابقات برنامه‌نویسی می‌توانند داشته باشند، تقویت و محک توانایی حل مسأله برنامه‌نویسان است. به همین دلیل، معمولا حامیان اصلی این مسابقات شرکت‌های تولید نرم‌افزار هستند؛ زیرا از این طریق می‌توانند به ترتیب و شناسایی نیروهای کاری آینده خود بپردازند. به عنوان مثال، حامی اصلی مسابقات اِی-سی-ام شرکت آی بی اِم[۳] است. بنابراین تجربه در مسابقات برنامه‌نویسی می‌تواند در شکل‌گیری آینده کاری دانشجویان نقش مهمی داشته باشد.

از دیگر فواید مسابقات برنامه‌نویسی، می‌توان به افزایش روحیه کار گروهی و همچنین افزایش انگیزه برای حل مسأله اشاره کرد. در این مسابقات شرکت‌کنندگان اغلب به صورت گروه‌های دو یا سه نفره با یکدیگر به رقابت می‌پردازند. واضح است که تمرکز همزمان تمامی اعضای گروه روی یک مسأله تا زمان حل کامل آن، باعث به هدر رفتن زمان می‌شود. به همین دلیل بهترین استراتژی در این مسابقات این است که افراد به طور موازی به حل مسائل مختلف بپردازند. در واقع بسیاری از گروه‌های موفق در مسابقات برای افراد نقش‌های مختلفی مانند برنامه‌نویس و تحلیل‌گر الگوریتم در نظر می‌گیرند تا بتوانند در مدت زمان کمتر تعداد سوالات بیشتری را حل کنند. به این ترتیب زمانی که برنامه‌نویس گروه در حال کدزنی جواب یک مسأله است، تحلیل‌گر الگوریتم می‌تواند به سوالات باقیمانده بپردازد. به همین دلیل، در این مسابقات تقسیم وظایف و همکاری با یکدیگر عامل مهمی در موفقیت تیم‌ها محسوب می‌شود.

---

[۲] Google

[۳] IBM

۱-۲- زبان سی++

همانطور که احتمالا تا به حال متوجه شده‌اید، تاکید اصلی این کتاب بر کار با «زبان برنامه‌نویسی سی++[۴]» است. به همین دلیل، بهتر است برای شروع کمی با این زبان برنامه‌نویسی محبوب آشنا شوید.

رایانه هیچ کاری نمی‌تواند انجام دهد مگر این که کسی آن کار را صراحتا و با تمام جزییات مورد نیاز برای رایانه بیان کند. به این بیان کامل یک برنامه یا کد گفته می‌شود. برنامه‌نویسی یا کدنویسی هم فرآیند تولید و ارزیابی برنامه‌ها است.

همه‌ی ما تا به حال به نوعی در زندگی برنامه‌نویسی کرده‌ایم. حتما تا به حال برای شما پیش آمده است که برای انجام کار خاصی به یکی از دوستان خود دستورالعمل‌های لازم را بدهید. به عنوان مثال، اگر یکی از دوستانتان از شما آدرس نزدیک‌ترین رستوران را بپرسد، یکی از جواب‌های احتمالی شما می‌تواند این باشد: «از دانشگاه که خارج می‌شوی، یک مقدار به سمت راست برو تا رستوران را پیدا کنی». تفاوت بین یک برنامه و چنین دستورالعملی در میزان دقت به کار رفته است. مثلا با این که دستورالعمل مذکور به نظر راهکار مناسبی برای پیدا کردن نزدیکترین رستوران به نظر می‌رسد، اگر با دقت بیشتری به آن نگاه کنید ابهامات زیادی در آن مشاهده می‌کنید و حتی ممکن است جملات آن از نظر دستور زبان نیز ایراد داشته باشند. مثلا در این دستورالعمل بیان نشده است که چگونه باید از دانشگاه خارج شد، یا این که علامت مشخص‌کننده رستوران برای پیدا کردن آن چیست. با این حال، با توجه به این که انسان‌ها می‌توانند این کاستی‌ها را توسط قدرت تفکر خود جبران کنند، چنین دستورالعمل‌هایی برای آنها کافیست. در طرف مقابل، یک رایانه به علت نداشتن قدرت تفکر توانایی درک چنین دستورالعمل‌هایی را ندارد. به طور مثال، در مواجهه با دستورالعمل پیدا کردن نزدیک‌ترین رستوران، سوالات زیادی برای رایانه پیش می‌آید. دانشگاه چیست؟ خارج شدن یعنی چه؟ چگونه باید از دانشگاه خارج شد؟ یک مقدار یعنی چقدر؟ سمت راست چیست؟ رستوران چیست؟ برای از بین بردن این ابهامات، نیاز به یک زبان دقیق با یک دستور زبان مشخص و یک فرهنگ واژگان خوش تعریف داریم که توانایی بیان دستورالعمل‌های مورد نیاز ما را داشته باشد. به چنین زبانی یک زبان برنامه‌نویسی گفته می‌شود.

زبان سی++ یکی از قدرتمندترین و پرطرفدارترین زبان‌های برنامه‌نویسی موجود است. علاوه بر این، سی++ زبان برنامه‌نویسی مورد استفاده در بسیاری از مسابقات برنامه‌نویسی از جمله اِی-سی-ام نیز است. به همین علت، این زبان ابزار اصلی برنامه‌نویسی در این کتاب خواهد بود. در ادامه این فصل به بیان پیش‌نیازهای مورد نیاز برای برنامه‌نویسی به زبان سی++ پرداخته می‌شود.

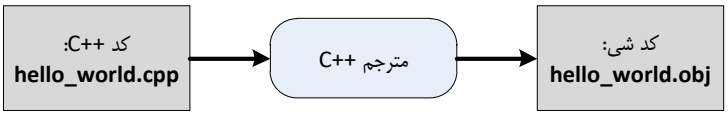
۱-۳- فرآیند اجرای یک برنامه

برای اجرای یک برنامه به زبان سی++، ابتدا باید کد برنامه را از حالتی که برای انسان قابل فهم است به حالتی تبدیل کرد که برای رایانه قابل فهم باشد. این تبدیل حالت توسط برنامه‌ای به نام مترجم انجام می‌شود. خروجی برنامه مترجم، کد شی نام دارد (شکل ۱,۱). کدهای نوشته شده با زبان سی++، با پسوند cpp. (مانند

---

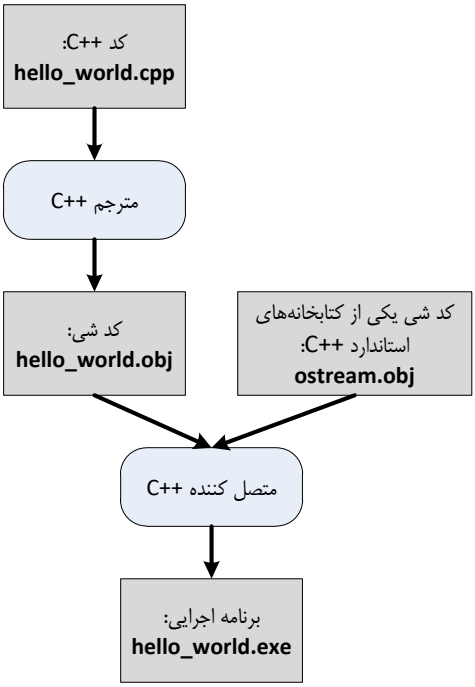
[۴] C++ Programming language

hello_world.cpp) یا h. (مانند std_lib_facilities.h)، و کدهای شی هم در سیستم عامل ویندوز[۵] با پسوند obj. و در سیستم عامل یونیکس[۶] با پسوند o. ذخیره می‌شوند.



شکل ۱,۱. نحوه کار مترجم سی++

فرآیند ترجمه کد فرآیند پیچیده‌ای است که طی آن مترجم ابتدا صحت کد برنامه را از نظر دستور زبان بررسی می کند و مطمئن می‌شود که تمامی کلمات استفاده شده در کد جزو کلمات مجاز هستند. در حالت کلی وظیفه مترجم این است که پیش از اجرای برنامه، تا حد ممکن تمامی ایرادهای برنامه را پیدا کرده و به برنامه‌نویس گزارش دهد.
پس از اتمام فرآیند ترجمه، فرآیند دیگری به نام اتصال باید انجام شود تا پرونده اجرایی برنامه ایجاد شود (شکل ۲,۱). در ادامه به شرح مختصری از این فرآیند پرداخته می‌شود.



شکل ۲,۱. نحوه کار مترجم و متصل‌کننده سی++

[۵] Windows

[۶] Unix

یک برنامه معمولا از قسمت‌های جداگانه‌ای تشکیل می‌شود که توسط افراد مختلف ایجاد شده‌اند. برخی از این قسمت‌ها می توانند در قالب کتابخانه در دسترس برنامه‌نویسان قرار داشته باشند. یک کتابخانه در واقع مجموعه‌ای از کدهای از پیش ترجمه شده است که یک برنامه‌نویس می‌تواند برای سهولت در کدنویسی از آنها استفاده کند. قسمت‌های مختلف یک برنامه باید بعد از ترجمه به همدیگر متصل شوند تا یک برنامه قابل اجرا تولید شود. به برنامه‌ای که کار اتصال قسمت‌های مختلف یک برنامه را انجام می‌دهد، متصل‌کننده گفته می‌شود (شکل ۲,۱). لازم به ذکر است که امروزه اغلب برنامه‌های مترجم هر دو عملیات ترجمه و اتصال را روی کدها انجام می‌دهند؛ بنابراین از این قسمت کتاب به بعد، به این دو عملیات به اختصار عملیات ترجمه گفته شده و منظور از برنامه مترجم برنامه‌ای است که این دو عملیات را روی کد انجام می‌دهد.

به علت این که در اکثر مثال‌های این کتاب، نیاز به استفاده از کتابخانه استاندارد سی++ وجود دارد، در اینجا کمی بیشتر در مورد این مفهوم صحبت می کنیم. همانطور که گفته شد، یک کتابخانه معمولا یک قطعه کد است که معمولا توسط دیگران نوشته شده است و برنامه‌نویس می‌تواند با استفاده از کلمه کلیدی #include در کد خود، از آنها استفاده کند. به عنوان مثال، در قطعه کد زیر از کتابخانه vector استفاده شده است. به چنین دستوراتی که با اضافه شدن به ابتدای کد برنامه امکان استفاده از قابلیت‌های کدهای از پیش نوشته شده دیگر را به برنامه‌نویس می‌دهند، سرآیند گفته می‌شود.

```
#include<vector>
int main() {
    //…
    return 0;
}
```

به خطاهایی که در زمان ترجمه مشاهده می‌شوند، خطاهای زمان ترجمه و به خطاهایی که در زمان اتصال مشاهده می‌شوند، خطاهای زمان اتصال گفته می‌شود. خطاهایی هم که تا زمان اجرای برنامه مشخص نمی‌شوند، خطاهای زمان اجرا یا خطاهای منطقی نامیده می‌شوند. معمولا خطاهای زمان ترجمه ساده‌تر از خطاهای زمان اتصال، و خطاهای زمان اتصال ساده‌تر از خطاهای زمان اجرا برطرف می‌شوند.

۱-٤- محیط‌های برنامه‌نویسی

تا اینجا مشخص شده است که برای برنامه‌نویسی به یک زبان برنامه‌نویسی، یک مترجم و یک متصل‌کننده نیاز است. علاوه بر این‌ها، به ابزاری نیاز داریم که کدهای خود را درون آن نوشته و ویرایش کنیم. به چنین ابزارهایی محیط‌های برنامه‌نویسی یا محیط‌های توسعه برنامه گفته می‌شود.

با این که هدف اصلی محیط‌های توسعه برنامه تنها ویرایش کد است، برخی از این محیط‌ها به طور خودکار عملیات ترجمه و اتصال کد را نیز انجام می‌دهند. به چنین محیط‌هایی، «محیط‌های توسعه مجتمع[۷]» یا به اختصار «آی-

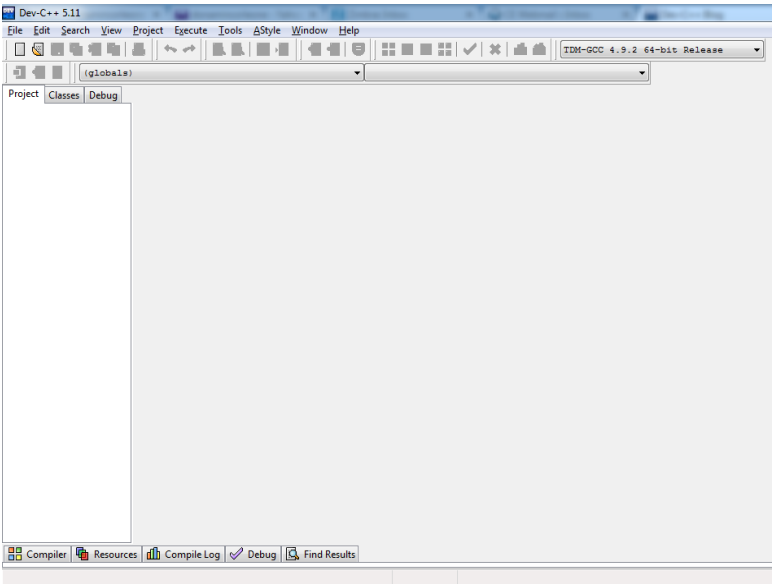
[۷] Integrated development environment

دی-ای[۸]» گفته می‌شود. اگر برنامه‌نویس برای کدنویسی از آی-دی-ای استفاده نکند، ناچار است که از طریق پنجره خط دستور به صورت دستی عملیات ترجمه و اتصال را اجرا کند. با توجه به این که برنامه‌های امروزی نیاز به کدنویسی زیادی دارند و به همین خاطر برنامه‌نویس در حین برنامه‌نویسی باید به دفعات بسیار زیاد کد برنامه خود را اجرا کند، اجرای دستی عملیات ترجمه و اتصال می‌تواند بسیار وقت‌گیر و خسته‌کننده باشد. به همین دلیل امروزه اکثر برنامه‌نویسان برای کدنویسی از محیط‌های توسعه مجتمع استفاده می‌کنند.

برای کدزنی به زبان سی++، محیط‌های برنامه‌نویسی بسیار زیادی وجود دارد. با توجه به این که هدف این کتاب آماده‌سازی خواننده برای شرکت در مسابقات برنامه‌نویسی است، در این بخش به معرفی سه محیط پرکاربرد در این مسابقات پرداخته می‌شود. تمامی کدهای آورده شده در این کتاب، در تمامی محیط‌های برنامه‌نویسی معرفی شده در این بخش آزمایش شده و قابل اجرا هستند.

۱-٤-۱- محیط برنامه‌نویسی دِو-سی++

برنامه «دِو-سی++[۹]» یک محیط برنامه‌نویسی رایگان است که برای برنامه‌نویسی به زبان‌های سی و سی++ استفاده می‌شود. این برنامه که با زبان دلفی[۱۰] پیاده‌سازی شده است، در حال حاضر فقط قابل اجرا بر روی سیستم عامل ویندوز است. محیط این برنامه در شکل ۳,۱ نشان داده شده است.



شکل ۳,۱. محیط برنامه دِو-سی++

[۸] IDE

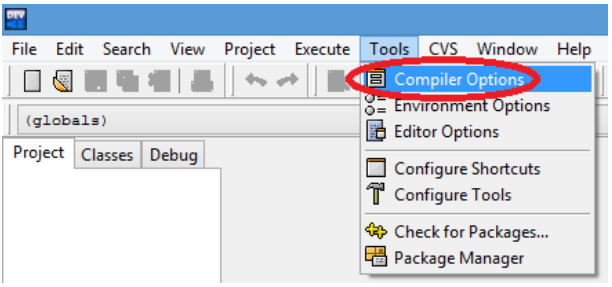
[۹] Dev-C++

[۱۰] Delphi

محیط برنامه‌نویسی دِو-سی++، از آدرس اینترنتی http://sourceforge.net/projects/orwelldevcpp/ قابل بارگیری است. نسخه به کار رفته از این برنامه در این کتاب، Dev-C++ 5.11[۱۱] است. بنابراین استفاده از نسخه‌های پیشین این نسخه به هیچ عنوان توصیه نمی‌گردد.

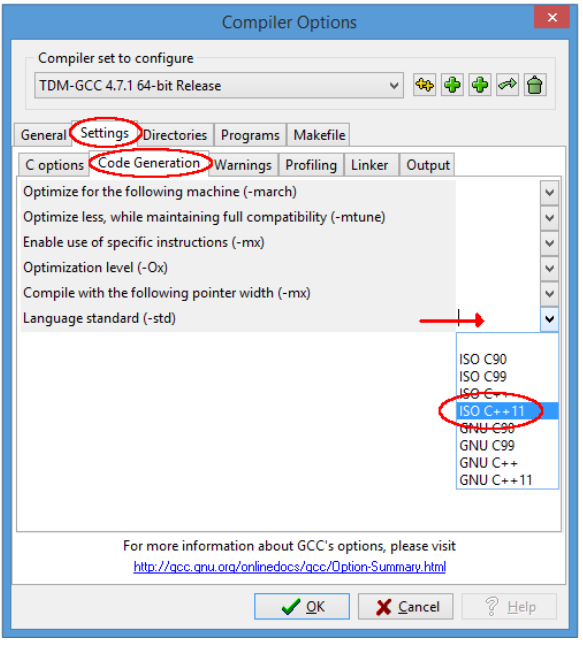
لازم به ذکر است که دِو-سی++ به صورت پیشفرض از نسخه ۱۱ زبان سی++ پشتیبانی نمی‌کند. برای حل این مشکل کافیست این مراحل را دنبال کنید:

- از منوی Tools گزینه Compiler Options را انتخاب کنید (شکل ۴,۱).



شکل ٤,١. نحوه فعالسازی پشتیبانی دِو-سی++ از سی++ ۱۱

- در پنجره باز شده، ابتدا زبانه Settings و سپس زبانه Code Generation را انتخاب کنید (شکل ۵,۱).
- در نهایت در قسمت Language standard (-std) گزینه ISO C++ 11 را انتخاب کرده و دکمه OK را بزنید (شکل ۵,۱).



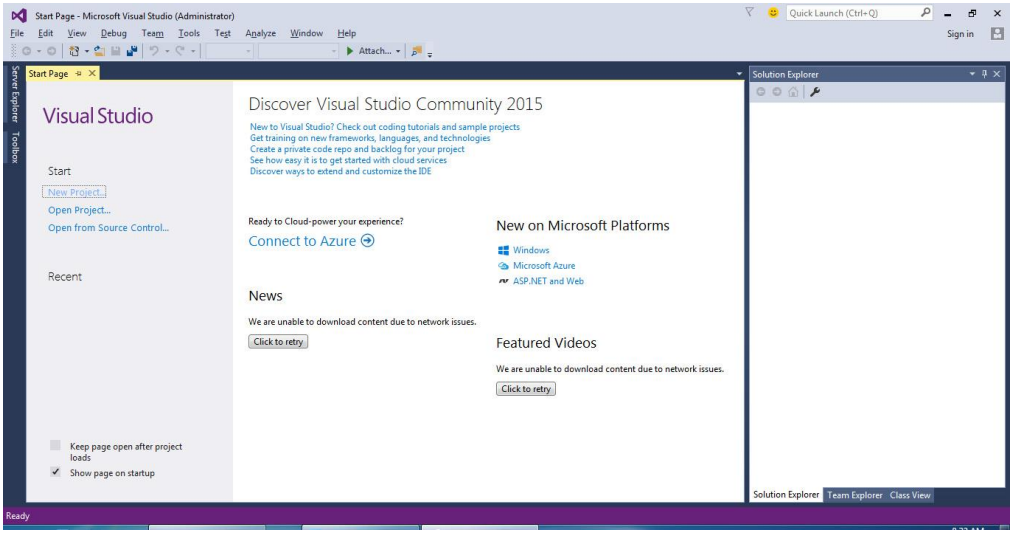
شکل ۵,۱. نحوه فعالسازی پشتیبانی دِو-سی++ از سی++ ۱۱

[۱۱] Dev-Cpp 5.11 TDM-GCC 4.9.2 (32bit and 64bit)

۱-٤-۲- محیط برنامه‌نویسی ویژوال استودیو

برنامه «ویژوال استودیو[۱۲]» محیط برنامه‌نویسی ارائه شده توسط شرکت مایکروسافت[۱۳] است که به زبان سی++ و سی‌شارپ[۱۴] پیاده‌سازی شده است. این برنامه که قابل اجرا بر نسخه‌های مختلف سیستم عامل ویندوز است؛ به علت امکانات بسیاری که برای ساده‌سازی برنامه‌نویسی ارائه می‌دهد، در حال حاضر یکی از پرطرفدارترین محیط‌های توسعه میان برنامه‌نویسان (به ویژه آنهایی که با زبان‌های سی++ و سی‌شارپ کار می‌کنند) است.

در حال حاضر برای برنامه ویژوال استودیو نسخه‌های زیادی وجود دارد که برنامه‌نویسان (یا گروه‌های برنامه‌نویسی) برای کار با این محیط لازم است با توجه به نیازهای خود، مناسبترین نسخه را برای پیاده‌سازی برنامه‌های خود انتخاب کنند. در این کتاب از یکی از نسخه‌های این برنامه به نام «ویژوال استودیو کامیونیتی[۱۵]» که به رایگان در اختیار برنامه‌نویسان قرار دارد، استفاده شده است. این برنامه در سال ۲۰۱۴ توسط شرکت مایکروسافت معرفی شده و برای توسعه‌دهندگان مستقل و گروه‌های برنامه‌نویسی کوچک گزینه مناسبی محسوب می‌شود. این برنامه از آدرس https://www.visualstudio.com/products/visual-studio-community-vs قابل بارگیری است. محیط این برنامه که از این قسمت به اختصار ویژوال استودیو نامیده می‌شود، در شکل ۶,۱ نشان داده شده است.



شکل ۶,۱ محیط برنامه ویژوال استودیو

---

[۱۲] Visual Studio

[۱۳] Microsoft

[۱۴] C#

[۱۵] Visual Studio Community

۱-٤-۳- محیط لینوکس

ترجمه کد برنامه‌ها در سیستم عامل لینوکس[۱۶] به کمک دو مترجم مهم «جی–سی–سی[۱۷]» و «کلانگ[۱۸]» می‌تواند انجام شود. نصب این دو برنامه در لینوکس به سادگی به کمک نرم‌افزار پیش‌افزار نصب برنامه‌ها در نسخه‌های گوناگون لینوکس قابل انجام است. همچنین این کار به سادگی می‌تواند در خط فرمان[۱۹] نیز انجام شود برای نمونه در نسخه دبیان[۲۰] و اوبونتو[۲۱] در خط فرمان می‌توان دستور زیر را برای نصب جی++[۲۲] وارد کرد:

```
sudo apt-get install g++
```

برنامه پس از وارد کردن رمز کاربر با مجوز نصب برنامه از اینترنت بارگیری و سپس نصب خواهد شد.

نصب کلانگ نیز توسط دستور زیر در خط فرمان امکان‌پذیر است:

```
sudo apt-get install clang
```

پس از وارد کردن رمز برنامه نصب می‌شود.

بدون وارد شدن به جزییات زیاد و نیاز به یادگیری گزینه‌های انتخابی گوناگونی که این دو کامپایلر در خط فرمان دارند، به سادگی می‌توان برنامه را با هر کدام از این دو مترجم در خط فرمان ترجمه کرد. به عنوان مثال، دو دستور زیر برای ترجمه پرونده‌ی حاوی کد سی++ به نام sample01.cpp به کار می‌روند:

```
g++ sample01.cpp
clang++ sample01.cpp
```

برای ترجمه کافی است فقط یکی از این دو مترجم (از طریق یکی از دو دستور گفته شده) اجرا شود. همه‌ی نمونه‌های معرفی شده در این کتاب با هر کدام از این دو دستور ترجمه می‌شوند مگر این که به روشنی توضیح داده شود که باید به شکل دیگری عملیات ترجمه را انجام داد.

برنامه اجرایی با نام a.out در کنار کد برنامه ساخته می‌شود و می‌توان به صورت زیر آن را در خط فرمان اجرا کرد.

```
./a.out
```

برای به کارگیری ویژگی‌های نسخه ۲۰۱۱ زبان سی++ در نسخه‌های قدیمی‌تر این مترجم‌ها (پیش از نسخه‌ی جی–سی–سی ۵,۲) باید گزینه‌ی –std=gnu++11 نیز برای آنها گذاشته شود و برای کامپایل برنامه با ویژگی‌های جدید یکی از دستورهای زیر به کار گرفته شود.

```
g++ -std=gnu++11  sample01.cpp
clang++ -std=gnu++11  sample01.cpp
```

برای نسخه‌های جدید این دو مترجم نیازی به گذاشتن این گزینه‌ها نیست و به صورت پیش‌فرض آخرین نسخه‌ی زبان به کار برده می‌شود. برای به کارگیری ویژگی‌های ۲۰۱۴ زبان در جی–سی–سی ۵,۲ می‌توان هر کدام از دو دستور زیر را به کار برد:

```
g++ -std=gnu++14  sample01.cpp
```

---

[۱۶] Linux
[۱۷] gcc
[۱۸] clang
[۱۹] Console, Terminal
[۲۰] Debian
[۲۱] Ubuntu
[۲۲] g++

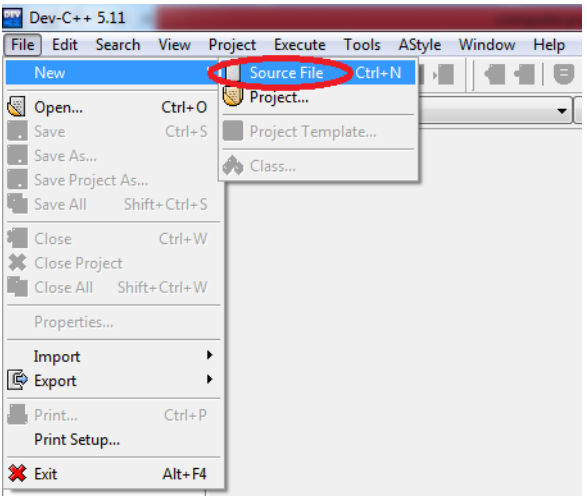
```
clang++ -std=gnu++14  sample01.cpp
```

۱-۵- سلام دنیا!

در این بخش برای آشنایی با نحوه کار با محیط‌های معرفی شده در بخش قبل، نحوه برنامه‌نویسی یک برنامه ساده را که به نام سلام دنیا! شناخته می‌شود، توضیح می‌دهیم.

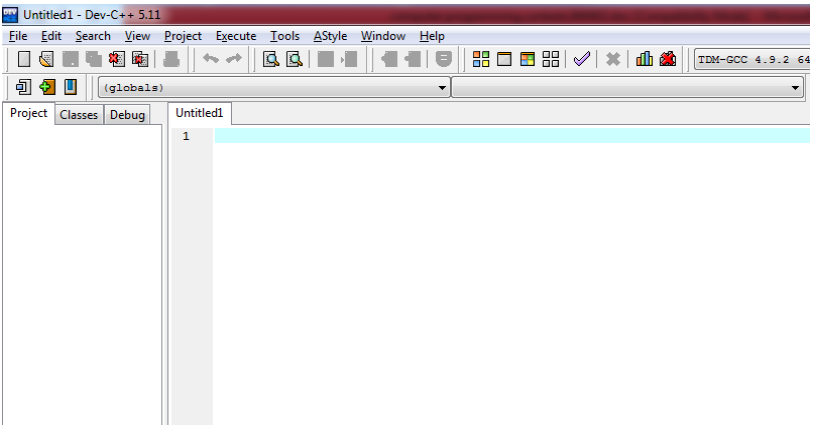
۱-۵-۱- محیط دِو-سی++

برای شروع، برنامه دِو-سی++ را اجرا کنید. سپس از منوی File به قسمت New رفته و گزینه Source File را انتخاب کنید (می‌توانید این کار را با فشردن همزمان کلیدهای Ctrl و N نیز انجام دهید).



شکل ٦,١. نحوه ایجاد یک پرونده جدید در دِو-سی++

با این کار یک پرونده جدید باز می‌شود (در این مثال دِو-سی++ نام Untitled1 را برای پرونده انتخاب کرده است). شما می‌توانید در این پرونده برنامه‌نویسی کنید.



شکل ۷٫۱. پرونده جدید ایجاد شده در محیط دِو-سی++

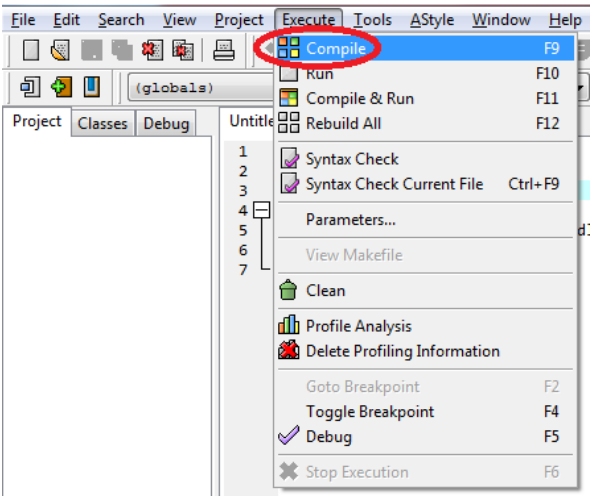
بهتر است پیش از شروع کار، پرونده را ذخیره کنید. هنگام ذخیره‌سازی، دِو-سی++ به صورت خودکار پسوند cpp. را برای پرونده انتخاب می‌کند. پس از ذخیره‌سازی پرونده، این قطعه کد را درون پرونده وارد کنید:

```
#include<iostream>
using namespace std;

int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

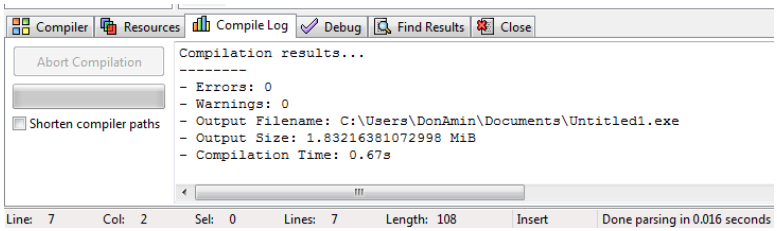
در اینجا قصد نداریم به شرح جزییات کد نشان داده شده بپردازیم. تنها کاری که این کد انجام می‌دهد این است که یک پیغام حاوی متن Hello World! را در صفحه نمایش چاپ کرده و سپس خاتمه می‌یابد.

حال بیایید این برنامه ساده را اجرا کنیم. ابتدا باید عملیات ترجمه و اتصال را انجام دهیم. برای این کار، از منوی Execute گزینه Compile را انتخاب کنید (یا کلید میانبر F9 را یک بار فشار دهید).



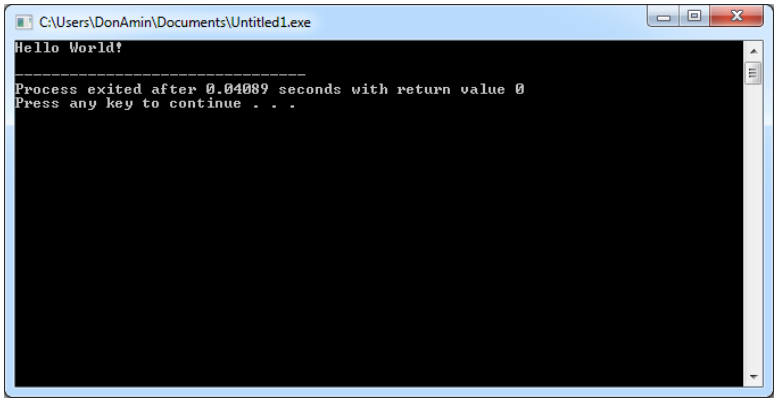
شکل ۸٫۱. نحوه اجرای عملیات ترجمه و اتصال در محیط دِو-سی++

با این کار، در پایین پنجره برنامه دِو-سی++، قسمت کوچکی نشان داده می‌شود که نتایج عملیات ترجمه و اتصال را نشان می‌دهد.



شکل ۹,۱. نتیجه اجرای عملیات ترجمه و اتصال در محیط دِو-سی++

همانطور که در تصویر مشاهده می‌شود، کد نوشته شده هیچ خطایی نداشته و عملیات ترجمه و اتصال با موفقیت به اتمام رسیده است. بنابراین حالا می‌توان این برنامه را اجرا کرد. برای اجرای برنامه، از منوی Execute گزینه Run را انتخاب کنید (یا کلید میانبر F10 را یک بار فشار دهید). با این کار پنجره جدیدی باز می‌شود که نتیجه اجرای برنامه‌ای که نوشته‌اید در آن نشان داده می‌شود.



شکل ۱۰,۱. نتیجه اجرای برنامه سلام دنیا! در محیط دِو-سی++

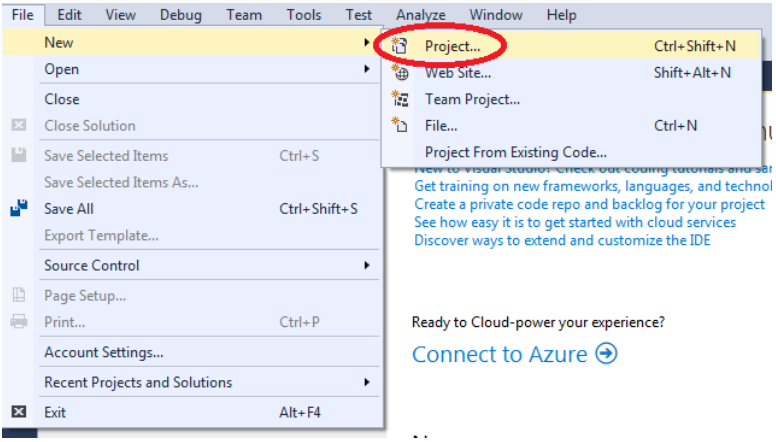
همانطور که می‌بینید برنامه به طور کامل اجرا شده است و پیغام Hello World! در پنجره اجرای برنامه نشان داده شده است. برای اجرای کدهایی در این کتاب معرفی می‌شوند، می‌توانید از همین روال معرفی شده در برنامه سلام دنیا! استفاده کنید.

۱-۵-۲- محیط ویژوال استودیو

نحوه برنامه‌نویسی در محیط ویژوال استودیو بسیار مشابه محیط دِو-سی++ است. تفاوت اصلی میان این دو محیط این است که در محیط ویژوال استودیو ابتدا باید یک پروژه جدید بسازید تا بتوانید یک پرونده خالی برای کدنویسی ایجاد کنید؛ در حالی که در محیط دِو-سی++ برای ایجاد یک پرونده جدید نیازی به ساخت یک پروژه نیست.

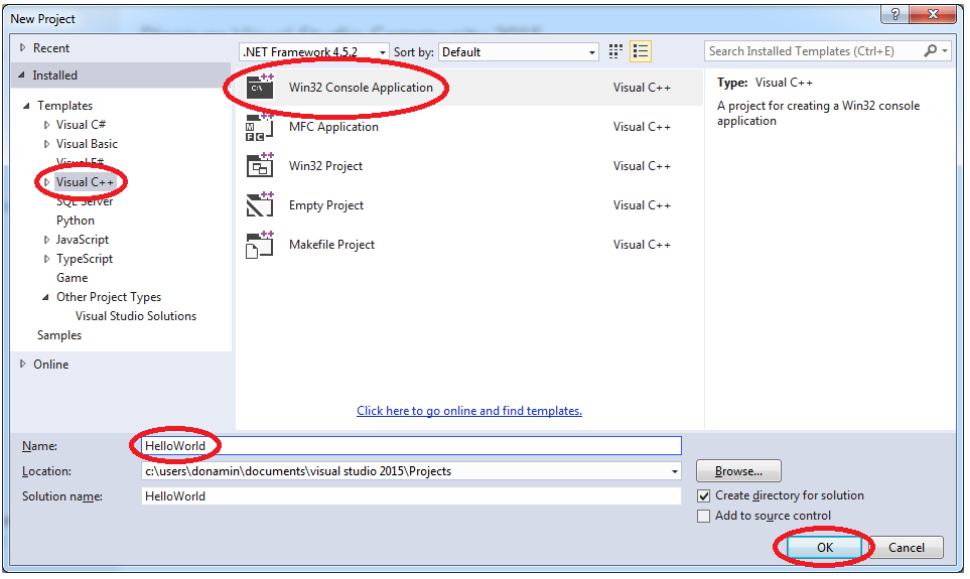
برای ساخت یک پروژه جدید در ویژوال استودیو این مراحل را دنبال کنید:

از منوی File به قسمت New رفته و گزینه Project را انتخاب کنید (شکل ۱۱,۱).



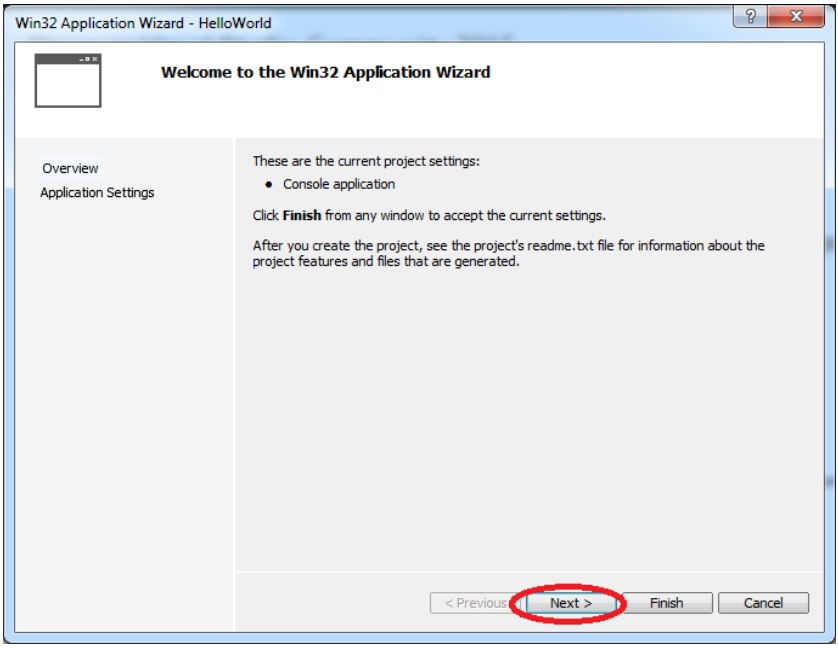
شکل ۱۱,۱. نحوه ایجاد یک پروژه جدید در ویژوال استودیو

به این ترتیب پنجره New Project باز می‌شود. در این پنجره در قسمت Visual C++ گزینه Win32 Console Application را انتخاب کنید. همچنین در این مرحله باید نام پروژه را وارد کنید. در اینجا نام HelloWorld انتخاب شده است (شکل ۱۲,۱).



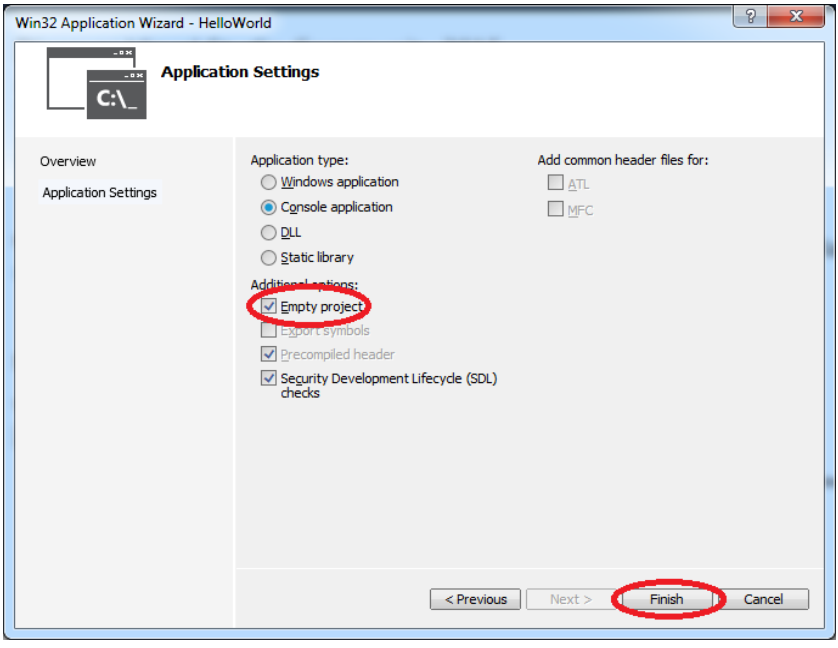
شکل ۱۲,۱. نحوه ایجاد یک پروژه جدید در ویژوال استودیو

با این کار پنجره جدیدی برای مشخص کردن تنظیمات پروژه جدید باز می‌شود. در صفحه اول این پنجره گزینه Next را انتخاب کنید (شکل ۱۳,۱).



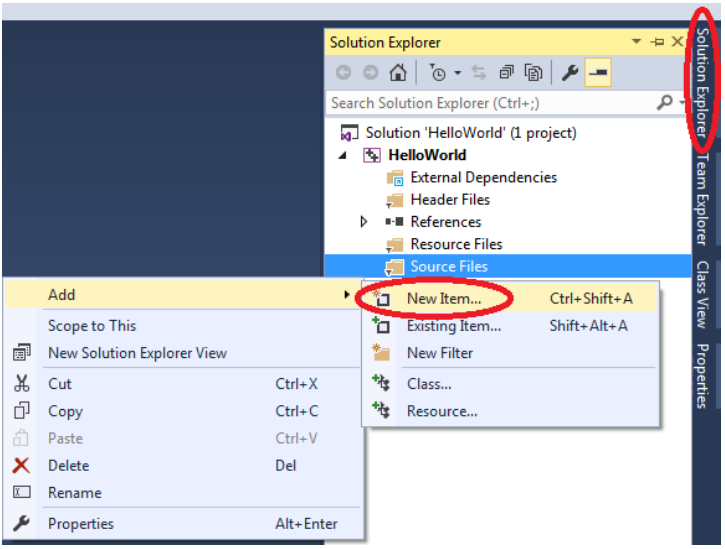
شکل ۱۳,۱. پنجره تنظیمات پروژه جدید در ویژوال استودیو

در صفحه بعد گزینه Empty Project را فعال کرده و دکمه Next را بزنید (شکل ۱۴,۱).



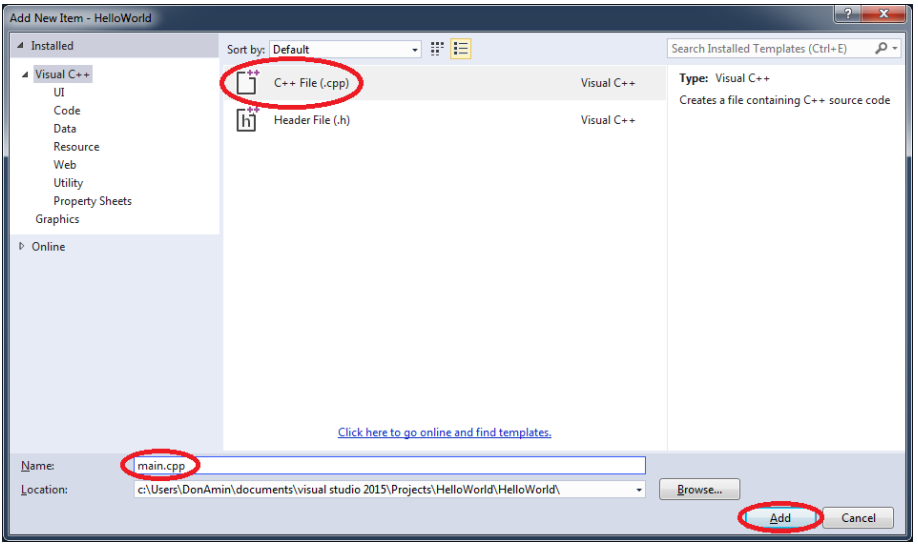
شکل ۱۴,۱. پنجره تنظیمات پروژه جدید در ویژوال استودیو

به این ترتیب پروژه‌ی جدید ساخته شده است. همانطور که پیش از این گفته شد، حال باید یک پرونده جدید برای کدنویسی ایجاد کنیم. برای این کار، در قسمت Solution Explorer، روی پوشه Source Files کلیک راست کرده و از قسمت Add گزینه New Item را انتخاب کنید (شکل ۱۵,۱).



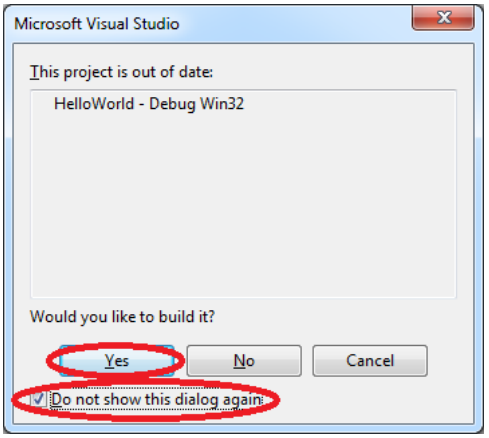
شکل ۱۵,۱. نحوه افزودن یک پرونده جدید به پروژه در ویژوال استودیو

در پنجره‌ی باز شده گزینه C++ File (.cpp) را انتخاب کرده و یک نام برای پرونده‌ی جدید وارد کنید (در اینجا main.cpp وارد شده است). سپس دکمه Add را بزنید (شکل ۱۶,۱).



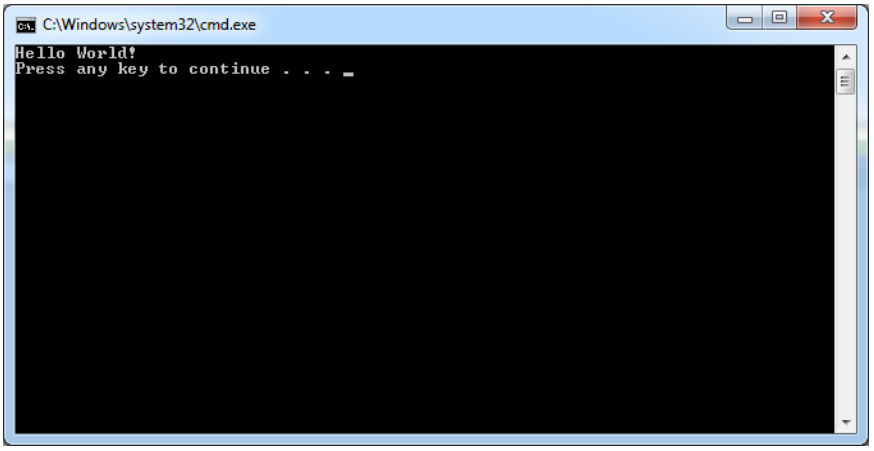
شکل ۱۶,۱. نحوه افزودن یک پرونده جدید به پروژه در ویژوال استودیو

به این ترتیب یک پرونده جدید به نام main.cpp به پروژه افزوده شده است. حال کد برنامه سلام دنیا! را که در قسمت قبل برای دِو–سی++ معرفی شد، در این پرونده نوشته و پرونده را ذخیره کنید. حال می‌توانید این برنامه را با فشردن همزمان کلیدهای Ctrl و F5 ترجمه و اجرای کنید. اگر با این کار با پنجره نشان داده شده در شکل ۱۷٫۱ مواجه شدید، گزینه Do not show this dialog again را فعال کرده و دکمه Yes را فشار دهید.



شکل ۱۷٫۱. پیغامی که در ویژوال استودیو پیش از اولین اجرای برنامه دیده می‌شود

با اجرای برنامه پنجره جدیدی باز شده و پیغام Hello World! در این پنجره نمایش داده می‌شود که به این معنی است که برنامه به درستی اجرا شده است (شکل ۱۸٫۱).



شکل ۱۸٫۱. نتیجه اجرای برنامه سلام دنیا! در محیط ویژوال استودیو

۱-۵-۳- محیط لینوکس

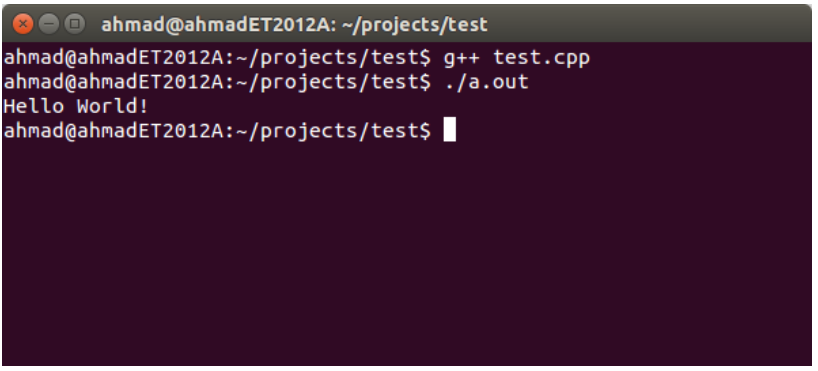
همانطور که پیش از این گفته شد، برای برنامه‌نویسی در محیط لینوکس باید یک پرونده جدید ایجاد کرده و پس از نوشتن کد مورد نظر خود پرونده را ذخیره کنید. سپس از طریق یکی از دو دستور زیر (با توجه به مترجم مورد استفاده) می توان عملیات ترجمه کد را انجام داد:

```
g++ sample01.cpp
clang++ sample01.cpp
```

در نهایت برای اجرای برنامه ترجمه شده می توان از این دستور استفاده نمود:

```
./a.out
```

نحوه اجرای این برنامه در شکل ۱۹,۱ نشان داده شده است.



شکل ۱۹,۱. نحوه اجرای برنامه سلام دنیا! در محیط لینوکس

۱-۶- ساختار کتاب

ادامه این کتاب به این صورت سازماندهی شده است: در فصل دوم ابزارهای اصلی در زبان سی++ از جمله نحوه کار با ورودی و خروجی معرفی می‌شود. فصل سوم به معرفی کتابخانه الگوی استاندارد در زبان سی++ می‌پردازد که حاوی پیاده‌سازی تمامی ساختمان داده‌های مهم بوده و برای کسانی که می‌خواهند به زبان سی++ برنامه‌نویسی کنند، ابزاری بسیار کاربردی محسوب می‌شود. در فصل چهارم الگوریتم‌های پرکاربرد در مسابقات برنامه‌نویسی به طور کامل شرح داده شده و پیاده‌سازی می‌شوند. با توجه به این که مسائل ریاضی یکی از سخت‌ترین انواع سوالات در مسابقات برنامه‌نویسی هستند، فصل پنجم به معرفی مفاهیم ریاضی مورد نیاز می‌پردازد. فصل ششم به معرفی مسائل معروف و پیچیده الگوریتمی و پاسخ هرکدام از آنها اختصاص داده شده است. فصل ششم از این جهت حائز اهمیت است که معمولا بسیاری از مسائل در مسابقات برنامه‌نویسی قابل تبدیل به همین مسائل معروف هستند؛ بنابراین آشنایی با این گونه مسائل و همچنین پاسخ‌های آنان ابزاری بسیار مهم در تقویت توانایی حل مسأله و موفقیت در مسابقات برنامه‌نویسی محسوب می‌شود. در نهایت فصل هفتم به معرفی مسابقه اِی-سی-ام می‌پردازد. مسابقه اِی-سی-ام یکی از مهمترین و چالش برانگیزترین مسابقات برنامه‌نویسی است؛ به همین دلیل در فصل هفتم قوانین این مسابقه و تکنیک‌های مورد نیاز برای شرکت در آن شرح داده می‌شود.

مراجع

Arefin, A.S., 2006. *Art of Programming Contest*, Reviewed by Dr. M. Lutfar Rahman and Forworded by Professor Miguel Revilla, University de Valladolid, Spain, Gyankosh Prokashoni, Dhaka, Bangladesh, First Ed., ISBN: 984-32-3382-4.

Martin, R.C., 2009. *Clean code: a handbook of agile software craftsmanship*, Pearson Education.

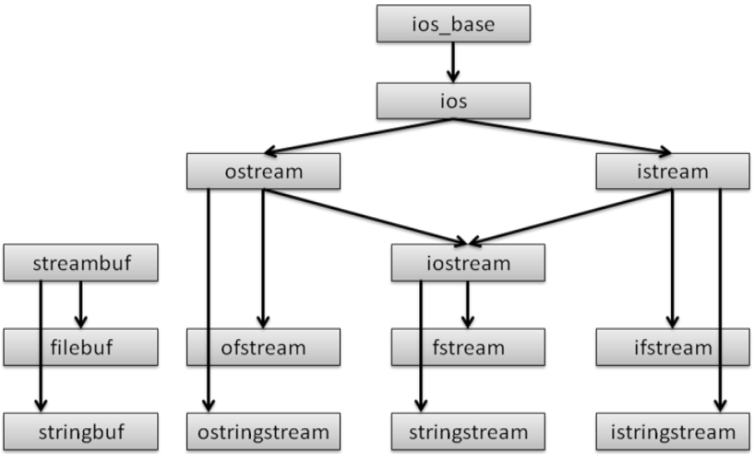
Powers, L. & Snell, M., 2015. *Microsoft Visual Studio 2015 unleashed* 3rd ed., Sams Publishing.

Skiena, S.S. & Revilla, M.A., 2006. *Programming challenges: The programming contest training manual*, Springer Science & Business Media.

Stroustrup, B., 2014. *Programming: principles and practice using C++* 2nd ed., Pearson Education.

فصل ۲- کار با ورودی و خروجی

در این فصل به معرفی انواع ورودی‌ها و خروجی‌ها پرداخته می‌شود. دستورهای ورودی و خروجی مربوط به زبان سی که در سی++ نیز قابل استفاده هستند، توضیح داده می‌شود. کلاس‌های مربوط به زبان سی++ دارای یک سلسله مراتب ارث‌بری هستند که در شکل ۱,۲ نشان داده شده است. در اینجا به همه‌ی کلاس‌های موجود در این سلسله مراتب، پرداخته نمی‌شود و به توضیح کلاس‌های کاربردی‌تر بسنده می‌شود.



شکل ۱,۲. سلسله مراتب کلاس‌های پایه ورودی و خروجی در سی++

در ادامه فصل به معرفی میانگیرها پرداخته می‌شود. میانگیرها، کلاس‌هایی هستند که امکان خواندن/نوشتن از/در مجموعه‌ای از نویسه‌ها را فراهم می‌کنند. همانگونه که می‌توان از یک رشته یا پرونده مقداری را خواند و یا در آن نوشت، می‌توان در میانگیرها نیز این کارها را انجام داد. استفاده از میانگیرها در بسیاری از موارد می تواند کار کردن با ورودی و خروجی را برای برنامه‌نویس ساده‌تر کند. علاوه بر این، در مواقعی ممکن است با مسأله‌ای مواجه شوید که ساختار ورودی یا خروجی آن شما را وادار به استفاده از میانگیرها کند.

۲-۱- ورودی و خروجی استاندارد

برای کار با ورودی و خروجی استاندارد در زبان سی++، هم می‌توان کتابخانه‌های ویژه‌ی سی++ و هم کتابخانه‌های سی را به کار برد. برای نمونه دو تابع printf و scanf مربوط به زبان سی هستند. این دو تابع ثابت‌های زیادی دارند که کار با آن‌ها را بسیار انعطاف‌پذیر کرده است و در این بخش با بیشتر ثابت‌های آن‌ها آشنا خواهیم شد.

۲-۱-۱- کار با ورودی و خروجی استاندارد در زبان سی

برای به‌کارگیری دستورات ورودی و خروجی مربوط به زبان سی، به سرآیند cstdio[۲۳] نیاز است. همه‌ی تابع‌هایی که در این بخش توضیح داده می‌شوند، به این سرآیند نیاز دارند.

۲-۱-۱-۱- تابع printf

این تابع برای کار با خروجی استاندارد به کار برده می‌شود. قالب کلی کار با این تابع به این صورت است:

```
#include <cstdio>
int main() {
    printf("Characters: %c %c \n", 'a', 65);
    return 0;
}
```

پارامتر اول، رشته‌ای است که می‌تواند دربردارنده تعدادی قالب (مثلا %d یا %c) باشد؛ اگر در پارامتر اول، تعـدادی قالب به کاربرده شده باشد، باید به تعداد قالب‌های نوشته شده، پس از پارامتر اول، پارامتر داشته باشیم (پارامتر دوم تا تعداد قالب‌ها به اضافه‌ی ۱). به عنوان مثال، در مثال بالا دو قالب (%c و %c) استفاده شده است؛ به همین دلیل دو پارامتر اضافی ('a' و 65) پس از پارامتر اول آمده است. قالب‌ها با نویسه % آغاز شده و با یک یا چند نویسه معنی‌دار ادامه می‌یابند. به جز بخش‌هایی که با % مشخص می‌شوند، بقیه رشته‌ها بدون تغییر در خروجی چاپ خواهند شـد. برای نمونه نویسه c پس از علامت % به این معنی است که به جای این قالب (%c) یک نویسه قرار خواهد گرفت؛ نویسه مربوطه باید در پارامترهای بعدی آورده شود. خروجی کد مذکور به این صورت است:

Output
Characters: a A

به جای %c اول، نویسه 'a' که در پارامتر دوم نوشته شده و به جای %c دوم، پارامتر سوم که عدد 65 (کـد اسـکی نویسه 'A')است، قرار می‌گیرد؛ از آنجا که %c به معنی یک نویسه است، بنابراین هر عددی (در اینجا 65) به نویسـه معادل با آن (در اینجا 'A') تبدیل خواهد شد.
فهرست بعضی از قالب‌های تابع printf در جدول ۱،۲ آورده شده است.

[۲۳] C Standard Input And Output

جدول ۱,۲. برخی از قالب‌های قابل استفاده در تابع printf

ثابت	توضیح
c	نویسه
d یا i	عدد صحیح علامت‌دار
e	نمایش عدد به صورت نماد علمی با حرف e
E	نمایش عدد به صورت نماد علمی با حرف E
f	عدد اعشاری
g	عدد اعشاری؛ اگر عدد بزرگتر از حد ممکن باشد، به صورت نماد علمی با حرف e نمایش داده می‌شود.
G	عدد اعشاری، اگر عدد بزرگتر از حد ممکن باشد، به صورت نماد علمی با حرف E نمایش داده می‌شود.
o	عدد در مبنای هشت به صورت علامت‌دار
s	آرایه‌ای از نویسه‌ها
u	عدد صحیح بدون علامت
x	عدد در مبنای شانزده بدون علامت
X	عدد در مبنای شانزده بدون علامت و با حروف بزرگ
p	آدرس اشاره‌گر (یک اشاره‌گر دریافت کرده و آدرس آن را چاپ می‌کند)
%	برای چاپ خود علامت % باید دو بار آن را بنویسید (%%)

برای روشن شدن قالب‌های جدول ۱,۲ به این مثال توجه کنید (سرآیند climits به خاطر اجرا شدن در محیط لینوکس در این کد include شده است):

```
#include<cstdio>
#include<iostream>
#include<climits>
int main() {
    int num = 12;
    int* p = &num;
    printf("Characters: %c %c \n", 'a', 65);
    printf("Decimals: %ld %ld \n", LONG_MIN, LONG_MAX);
    printf("Some different radixes: %o %x \n", 100, 100);
    printf("Floats: %f %e %E %g \n", 3.141697, 3.141697, 3.141697,
3.141697);
    printf("%s \n", "A string");
    printf("Pointer: %p \n", p);
    printf("%% : %% \n");
    return 0;
}
```

خروجی کد مذکور به این صورت است:

```
Output
Characters:  a A
Decimals: -2147483648 2147483647
Some different radixes: 144 64
Floats: 3.141697 3.141697e+000 3.141697E+000 3.1417
A string
Pointer: 0012FF60
% : %
```

قالب‌هایی که تا اینجا معرفی شدند مربوط به نوع داده‌ها هستند. قالب‌های دیگری نیز وجود دارند که برای تعیین قالب و شکل‌دهی به خروجی به کار می‌روند. در جدول ۲,۲ فهرست این قالب‌ها آورده شده است.

جدول ۲,۲. فهرست قالب‌های تابع printf برای شکل‌دهی به خروجی

ثابت	توضیح
-	اگر برای خروجی فضا تعیین شده باشد، خروجی را در فضای مربوطه از سمت چپ چاپ خواهد کرد.
+	علامت اعداد را در خروجی نمایش می‌دهد. حتی برای اعداد مثبت نیز علامت + قرار خواهد داد.
فاصله	یک فاصله پیش از عدد چاپ می‌کند (تعداد فاصله‌ها مهم نیست)؛ البته اگر علامت عدد چاپ نشود.
#	این نویسه می‌تواند پیش از نویسه‌های o، x و X قرار گرفته و موجب می‌شود پیش از آن‌ها علامت 0، 0x و 0X چاپ شود. اگر این نویسه پیش از نویسه‌های e، E و f به کار برده شود، باعث می‌شود در پایان ارقام به تعداد باقیمانده رقم صفر قرار گیرد؛ در حالت پیش‌فرض صفرهای اضافی حذف می‌شوند. امکان استفاده با نویسه g و G نیز وجود دارد.
0	فضای اضافی عدد را با رقم صفر پر می‌کند؛ تعیین کردن فضای اضافی در ادامه توضیح داده شده است.

یکی دیگر از نکات جالب در مورد تابع printf این است که می‌توان فضای مشخصی برای چاپ نویسه‌ها در خروجی تعیین کرد. به این شکل که نویسه‌ها در یک فضای معین چاپ شوند و اگر اندازه اشغال شده توسط آن‌ها از فضای تعیین شده کوچکتر باشد، بقیه فضا با فضای خالی و یا به شکل‌های دیگر پر خواهد شد. تنظیمات مربوط به تعیین فضا در جدول ۳,۲ آورده شده است.

جدول ۳,۲. تنظیمات مربوط به تعیین فضا در تابع printf

ثابت	توضیح
عدد	پیش از نویسه‌های قالبی (d، c، f و ...) می‌تواند یک عدد قرار گیرد. این عدد نشان‌دهنده فضای چاپ خروجی مربوطه است. اگر طول خروجی بزرگتر از فضای تعیین شده باشد، کل خروجی چاپ خواهد شد و در صورتی که تعداد نویسه‌های خروجی از فضای تعیین شده کمتر باشد، بقیه فضا با نویسه فاصله پر خواهد شد.
*	اگر بخواهید مقدار فضا را در زمان اجرا تعیین کنید، می‌توانید این نویسه را پیش از قالب مربوطه به کار ببرید. طول فضا به شکل یکی از پارامترها فرستاده می‌شود.

برای درک بیشتر این قالب‌ها به نمونه بعدی توجه کنید:

```
#include<cstdio>
#include<iostream>
int main() {
    int DW = 5;
    printf("Width 5: %05d \n", 123);
    printf("Dynamic Width:  %*d \n", DW, 123);
    printf("Jusfity: %-5d \n", 123);
    printf("Fill with 0: %05d \n", 123);
    printf("Dynamic Fill: %0*5d \n", DW, 123);
    printf("One Space: %  d \n", 123);
    printf("Sign: %+d \n", 123);
    printf("0, 0x, 0X: %#o ,%#x ,%#X \n", 20, 20, 20);
    return 0;
}
```

خروجی کد مذکور به این صورت است:

```
Output
Width 5:  123
Dynamic Width:    123
Jusfity: 123
Fill with 0: 00123
Dynamic Fill: 00123
One Space:  123
Sign: +123
0, 0x, 0X: 024 ,0x14 ,0X14
```

نحوه تعیین فضا در جدول ۳,۲ و کد قبلی معرفی شد. نویسه {'.' + عدد} نیز برای تعیین فضا به کار برده می‌شود؛ عملکرد این قالب با قالب پیشین کمی متفاوت است. جزییات این قالب در جدول ۴,۲ آمده است.

جدول ٤,٢. تنظیمات بیشتر مربوط به تنظیم فضا در تابع printf

ثابت	توضیح
عدد.	اگر پیش از نویسه‌های {d، i ، o ، u ، x ، X} به کار برده شود، برای تعیین دقت خواهد بود؛ اگر تعداد ارقام خروجی از فضای تعیین شده کمتر باشد، بقیه فضا با عدد صفر پر می‌شود. اگر پیش از نویسه‌های {f، e، E} به کار برده شود، تعیین‌کننده تعداد ارقام اعشاری خواهد بود. برای نویسه {s} تعیین‌کننده بیشترین تعداد نویسه‌هایی است که باید چاپ شوند. در حالت پیش‌فرض، چاپ نویسه‌ها تا وقتی که به نویسه null (‘\n’) نرسیده باشیم، ادامه می‌یابد. برای نویسه {c} این ثابت بی‌معنی است.
*.	این ثابت دقیقا مثل بالا عمل می‌کند؛ با این تفاوت که طول فضا به صورت پویا تعیین می‌شود.

در مثال بعدی نحوه کار با این قالب نشان داده شده است.

```
#include<cstdio>
#include<iostream>
int main() {
    int DW = 5;
    printf("%.d , %.5f , %.5x \n", 123, 123.456, 20);
    printf("%.5e  \n", 123.456);
    printf("%.5s  \n", "A String");
    printf("%.*s  \n", DW, "A String");
    printf("%5.15s  \n", "Str");
    printf("%5.15s  \n", "A String");
    return 0;
}
```

خروجی برنامه مذکور به این صورت است:

```
Output
00123 , 123.45600 , 00014
1.23456e+002
A Str
A Str
   Str
A String
```

سه قالب دیگر نیز وجود دارند که برای تعیین انواع عددی به کار برده می‌شوند (جدول ۵,۲).

جدول ۵٫۲. قالب‌های تابع printf برای تعیین انواع عددی

ثابت	توضیح
H	پارامتر ورودی را از نوع اعداد صحیح ۲ بایتی (short int) علامت‌دار/ بدون علامت تفسیر می‌کند؛ این ثابت برای قالب‌های مربوط به اعداد صحیح به کار می‌رود: {i، d، o، u، x و X}
l	پارامتر ورودی را از نوع اعداد صحیح ۴ بایتی (long int) علامت‌دار/ بدون علامت تفسیر می‌کند؛ این ثابت برای قالب‌های مربوط به اعداد صحیح به کار می‌رود: {i، d، o، u، x و X}
L	پارامتر ورودی را از نوع اعداد اعشاری ۴ بایتی (long double) تفسیر می‌کند؛ این ثابت برای قالب‌های مربوط به اعداد اعشاری به کار می‌رود: {e، E، f، g و G}

در این مثال از قالب‌های معرفی شده در جدول ۵٫۲ استفاده شده است.

```
#include<cstdio>
#include<iostream>
#include<climits>
int main() {
    printf("Short Int: %hd , %hd \n", SHRT_MIN, SHRT_MAX);
    printf("Long Int: %ld , %ld \n", LONG_MIN, LONG_MAX);
    printf("Long Double: %Lf , %Lf \n", -2547483647.2133,
2647483647.2133);
    return 0;
}
```

خروجی کد مذکور به این صورت است:

```
Output
Short Int: -32768 , 32767
Long Int: -2147483648 , 2147483647
Long Double: -2547483647.213300 , 2647483647.213300
```

قالب‌های متنوع بالا، انعطاف‌پذیری بالای تابع printf را نشان می‌دهند.

۲-۱-۱-۲- تابع scanf

تابع scanf برای کار با ورودی استاندارد به کار برده می‌شود (در برخی از نسخه‌های جدید ویژوال استودیو این تابع با نام scanf_s تعریف شده است). قالب کلی کار با این تابع به این صورت است:

```
#include <cstdio>
int main() {
    char c;
    printf("Enter a Character: ");
    scanf("%c", &c);
    return 0;
}
```

پارامتر اول این تابع مانند تابع printf رشته‌ای است که می‌تواند دربردارنده قالب‌های مختلفی باشد؛ به ازای هر قالب نوشته شده در پارامتر اول، این تابع مقداری را از ورودی خوانده و در متغیر یاد شده در پارامترهای بعدی قرار می‌دهد. نمونه بالا، نویسه‌ای را از ورودی خوانده و در متغیر *c* قرار می‌دهد. توجه داشته باشید که برای قرار دادن متغیر مربوطه باید آدرس آن را به کار ببرید؛ به این صورت که برای متغیرهای معمولی پیش از آن‌ها علامت & قرار دهید؛ برای آرایه‌ها، نوشتن نام آن‌ها کافی است. شکل کلی این تابع به این صورت است (هر یک از این بخش‌ها در ادامه توضیح داده شده‌اند):

[	%	[	*	]	[	پهنای فضا	]	[	اصلاح کننده	]	[	نوع	]

جزییات مربوط به نحوه‌ی استفاده از تابع scanf در جدول ۶,۲ آمده است.

جدول ٦,٢. قسمت‌های مختلف تابع scanf

بخش	توضیح
*	اگر پس از نویسه % این علامت به کار برده شود، ورودی بعدی تابع نادیده گرفته شده و در مکانی ذخیره نخواهد شد.
پهنای فضا	این عدد تعیین‌کننده حداکثر تعداد نویسه‌هایی است که تابع دریافت می‌کند.
اصلاح‌کننده	مقادیر مربوط به این بخش در جدول ۷,۲ توضیح داده شده‌اند.
نوع	مقادیر مربوط به این بخش در جدول ۸,۲ توضیح داده شده‌اند.

نوع‌های داده‌ای قابل استفاده در تابع scanf در جدول ۷,۲ معرفی شده‌اند.

جدول ۷,۲. نوع‌های داده‌ای قابل استفاده در تابع scanf

نوع	توضیح
c	یک نویسه؛ اگر پیش از آن، ثابت تعداد (یک عدد) به کار برده شود، به تعداد یاد شده از ورودی نویسه خوانده و در متغیری که در پارامترهای بعدی معرفی شده، قرار می‌دهد.
d	اعداد صحیح علامت‌دار؛ علامت اعداد نیز به درستی تاثیر داده می‌شود.
G ،g ،f ،E ،e	انواع اعداد اعشاری؛ علامت‌ها و نماد علمی نیز به درستی خوانده می‌شوند.
o	اعداد در مبنای هشت
s	دنباله‌ای از نویسه‌ها تا جایی که به نویسه‌های فضای خالی (فاصله، تب و آخر خط) نرسد، خوانده شده و در یک آرایه (که در پارامترهای بعدی آورده شده است) قرار داده می‌شود.
u	اعداد صحیح بدون علامت
X ،x	اعداد در مبنای شانزده

چند مثال برای استفاده از تابع scanf در مثال بعدی قابل مشاهده است.

```
#include<cstdio>
#include<iostream>
int main() {
    char arr[6], c;
    int num_int;
    float num_float1, num_float2, num_float3;
    int oct, hex;
    char string[100];
    printf("Enter a char, 6 char: ");
    scanf("%c", &c);
    scanf("%6c", arr);
    printf("\nEnter an int, float, float with e, float with E: ");
    scanf("%d %f %e %E", &num_int, &num_float1, &num_float2,
&num_float3);
    printf("\nEnter an oct, hex: ");
    scanf("%o %x", &oct, &hex);
    printf("\nEnter a string: ");
    scanf("%s", string);
    return 0;
}
```

ورودی زیر یک ورودی معتبر برای کد مذکور است:

Input
A 5Char 10 20.1 1e10 1E10 12 A Astring

بخش اصلاح‌کننده، می‌تواند مقدارهای متفاوتی بپذیرد. این مقدارها در جدول ۸,۲ شرح داده شده‌اند.

جدول ۸,۲. اصلاح‌کننده‌های قابل استفاده در تابع scanf

اصلاح‌کننده	توضیح
h	عددی از نوع اعداد صحیح ۲ بایتی (short int) علامت‌دار/بدون علامت از ورودی می‌خواند؛ این اصلاح‌کننده برای قالب‌های مربوط به اعداد صحیح به کار می‌رود: {i، d، o، u، x و X}
l	عددی از نوع اعداد صحیح ۴ بایتی (long int) علامت‌دار/بدون علامت از ورودی می‌خواند؛ این اصلاح‌کننده برای قالب‌های مربوط به اعداد صحیح به کار می‌رود: {i، d، o، u، x و X}
L	عددی از نوع اعداد اعشاری ۴ بایتی (long double) از ورودی می‌خواند؛ این اصلاح‌کننده برای قالب‌های مربوط به اعداد اعشاری به کار می‌رود: {e، f و g}

این تابع افزون بر توضیحات بالا، می‌تواند به شکلی جالب نیز ورودی دریافت کند؛ به نمونه‌ی بعدی توجه کنید:

```
#include <cstdio>
int main() {
    char string[100];
    scanf("%[^\n]", string);
    printf("You Entered: %s\n", string);
    scanf("%*d");
    scanf("%[abcd]", string);
    printf("Filter Result: %s\n", string);
    return 0;
}
```

تابع scanf اجازه می‌دهد که برای آن، محدوده‌ای از نویسه‌های مجاز یا غیرمجاز تعیین شود؛ به این صورت کـه آن ها را در میان علامت [] قرار می‌دهیم؛ برای نمونه، این قطعه کد به کار رفته در نمونه قبلی به این معنی اسـت کـه تا وقتی نویسه‌های a، b، c و d را در ورودی دیده نشده است، به دریافت نویسه از ورودی ادامه داده و سپس نتیجـه را در آرایه string قرار بده:

```
scanf("%[abcd]", string);
```

اگر در ابتدای نویسه‌های تعیین شده، علامت ^ قرار داده شود، عملکرد تابع scanf وارون می‌شود؛ برای نمونه ایـن خط که در نمونه کد قبلی به کار برده شد، به این معنی است که تا وقتی به نویسه \n (آخر خط) نرسیدی، به دریافت نویسه از ورودی ادامه بده؛ بنابراین دستور زیر یک خط را بطور کامل همراه با فاصله‌های وارد شده می‌خواند.

```
scanf("%[^\n]", string);
```

حال تنها بخش مجهول کد قبلی، این دستور است:

```
scanf("%*d");
```

این دستور، ورودی بعدی را دریافت کرده، ولی در مکانی ذخیره نمی‌کند (آن را نادیـده می‌گیـرد). بـه‌کـارگیری ایـن دستور، به دلیل وجود تابع scanf قبل از آن است؛ زیرا تابع scanf پیش از آن، خوانـدن از ورودی را تـا پایـان خـط ادامه می‌دهد، ولی نویسه آخر خط را نمی‌خواند؛ بنابراین باید به نحوی این نویسه‌ی اضافه را خوانده و دور بریزیم.

۲-۱-۳- تابع getchar

تابع getchar برای خواندن یک نویسه از ورودی استاندارد به کار برده می‌شود. این تابع نویسـه بعـدی را از ورودی دریافت کرده و کد اسکی آن را به صورت یک عدد صحیح از نوع int برمی‌گرداند؛ این تابع نویسه‌هایی مانند فاصله، آخر خط و ... را نیز می‌خواند. نحوه استفاده از این تابع در ادامه آمده است.

```
#include<cstdio>
int main() {
    char c;
    printf("Enter a char: ");
    c = getchar();
    printf("Your entered: %c\n", c);
    return 0;
}
```

۲-۱-۱-٤- تابع‌های putchar و puts

تابع putchar یک نویسه و تابع puts، آرایه‌ای از نویسه‌ها را دریافت کرده و در خروجی نمایش می‌دهند. نمونه زیر برای آشنایی با این دو تابع کافی است:

```
#include<cstdio>
int main() {
    char c = 'H';
    char string[] = "\nHello world!";
    putchar(c);
    puts(string);
    return 0;
}
```

۲-۱-۱-۵- تابع perror

این تابع برای چاپ یک پیام در جریان خروجی خطای استاندارد به کار می‌رود. در حالت معمولی، این پیام در خروجی استاندارد نیز نمایش داده می‌شود؛ مگر اینکه این دو خروجی از هم جدا شده باشند. ساختار این تابع به این صورت است:

```
void perror(const char * str);
```

پارامتر ورودی این تابع، آرایه‌ای از نویسه‌ها است که در میانگیر خروجی قرار می‌گیرد. پس از چاپ رشته مورد نظر، آخرین خطایی که از فراخوانی توابع ایجاد شده نیز چاپ خواهد شد. میان پارامتر ورودی و خطای چاپ شده توسط تابع، علامت ":" و یک فاصله چاپ می‌شود. به نمونه زیر و توضیح آن توجه کنید:

```
#include <cstdio>
int main() {
    FILE * pFile;
    pFile = fopen("unexist.txt", "r");
    if (pFile == NULL)
        perror("The following error occurred");
    else
        fclose(pFile);
    return 0;
}
```

پس از فراخوانی تابع fopen، به دلیل عدم وجود پرونده "unexist.txt" خطای نبود پرونده رخ داده و پیام خطای "No such file or directory" در میانگیر خطای خروجی قرار می‌گیرد. بنابراین تابع perror پس از چاپ رشته‌ی موجود در پارامتر ورودی آن، خطای یاد شده را نیز چاپ خواهد کرد. بنابراین خروجی این برنامه به این صورت خواهد بود:

Output
The following error occurred: No such file or directory

لازم به ذکر است که تابع fopen در نسخه‌های جدید ویژوال استودیو با نام fopen_s قابل استفاده است. توابع دیگری نیز برای مدیریت خطاها وجود دارند که در سایر بخش‌ها توضیح داده می‌شوند.

۲-۱-۲- کار با ورودی و خروجی استاندارد در زبان سی++

برای به‌کارگیری دستورات مربوط به ورودی و خروجی استاندارد در زبان سی++، سرآیند iostream[۲۴] را به کار می بریم. این کلاس از دو کلاس istream و ostream به ارث برده است؛ بنابراین هر دو عمل ورودی و خروجی را انجام می‌دهد. این کلاس در پس‌زمینه دو میانگیر را برای ورودی و خروجی به کار می‌برد؛ به همین دلیل سرعت آن کندتر از توابع مربوط به ورودی و خروجی در زبان سی است که در بخش پیشین معرفی شد؛ البته به‌کارگیری میانگیر، سودهایی نیز درپی دارد که در ادامه گفته می‌شود.

۲-۱-۲-۱- کلاس‌های cin و cout

دو شیء cin و cout برای کار با ورودی و خروجی به کار برده می‌شوند. دو عملگر اصلی << و >> معروف به عملگرهای استخراج‌کننده هستند. به وسیله این دو عملگر، خواندن انواع داده‌ها از ورودی و یا چاپ آن‌ها در خروجی امکان‌پذیر است. داده‌های موجود در ورودی در واقع فقط مجموعه‌ای از نویسه‌ها هستند، ولی عملگر استخراج‌کننده ورودی، آن‌ها را بسته به نوع داده‌ای که از آن خواسته‌اید، استخراج کرده و در متغیر مربوطه قرار می‌دهد. انواع داده‌ای که این عملگر از آن‌ها پشتیبانی می‌کند، به این قرار هستند:

bool	short	unsigned short	int
unsigned int	long	unsigned long	float
double	long double	void *	

نمونه کد بعدی نحوه کار با cin و cout را نشان می‌دهد (این عملگر به شکل‌های دیگری نیز می‌تواند با ورودی کار کند که به علت گستردگی، در اینجا از توضیح آن پرهیز شده است):

[۲۴] Input/Output Stream

```cpp
#include <iostream>
using namespace std;
int main() {
    char c;
    int i;
    long l;
    cin >> c;
    cin >> i >> l;
    cout << c << " " << i << " " << l << endl;
    return 0;
}
```

شیءهای cin و cout تنظیماتی دارند که نوع عملکرد آن‌ها را در دریافت ورودی و چاپ خروجی کاملا سفارشی می‌سازد. در ادامه به شرح این تنظیمات می‌پردازیم.

جدول ۹,۲. پرچم‌های مربوط به تنظیمات شیءهای cin و cout

گروه	پرچم	توضیح
پرچم‌های مستقل	boolalpha noboolalpha	ورودی: عبارات true و false را جایگزین صفر و یک می‌کند؛ در این حالت باید به جای وارد کردن صفر و یک، عبارات true و false را وارد کنید؛ noboolalpha اثر آن را از میان می‌برد. خروجی: مقدارهای boolean را به صورت true یا false در خروجی چاپ می‌کند (در حالت پیش‌فرض صفر یا یک چاپ می‌شود)؛ noboolalgha اثر آن را از میان می‌برد.
	skipws noskipws	هنگام استخراج داده‌ها از ورودی، نویسه‌های فضای خالی را نیز به عنوان نویسه درنظر گرفته و آن‌ها را می‌خواند؛ skipws اثر آن را از میان می‌برد.
	showbase	مبنای اعداد نیز در خروجی نمایش داده می‌شود؛ برای اعداد مبنای شانزده عبارت 0X و برای اعداد مبنای هشت عدد صفر در سمت چپ آن‌ها نمایش داده خواهد شد.
	uppercase	نویسه‌ها (a-z) به شکل بزرگ خود چاپ خواهند شد.
	showpos	علامت اعداد مثبت (+) نیز چاپ می‌شود.
مبنای عددی	dec hex oct	ورودی: مقدار ورودی را به عنوان عددی در مبنای ده، شانزده و هشت در نظر گرفته و می‌خواند. خروجی: مقدار خروجی را با قالب عددی در مبنای ده، شانزده و هشت در نظر گرفته و می‌نویسد.
قالب اعشاری	fixed	اعشار اعداد نیز همراه با آن چاپ می‌شود؛ تا آخرین دقت ممکن؛ حتی اگر رقم اعشاری وجود نداشته باشد.
	scientific	نماد علمی اعداد نمایش داده خواهد شد.
موقعیت	right left	خروجی را در فضای تعیین شده از سمت راست/چپ تنظیم می‌کند.

در مثال بعدی کاربرد پرچم‌های معرفی شده در جدول ۹,۲ نشان داده شده است.

```
#include <iostream>
using namespace std;
int main() {
    bool b1, b2, b3, b4;
    cin >> b1 >> b2;
    cin >> boolalpha >> b3 >> b4;
    cout << endl;
    cout << boolalpha << b1 << " " << b2 << endl;
    cout << noboolalpha << b3 << " " << b4 << endl;
    return 0;
}
```

یک ورودی معتبر به همراه خروجی متناظر با آن برای مثال گفته شده به این صورت است:

Input	Output
0 1 true false	false true 1 0

مثال بعدی برای آشنایی با نحوه کار با پرچم‌های skipws و noskipws نوشته شده است.

```
#include <iostream>
using namespace std;
int main() {
    char a, b, c;
    cin >> skipws >> a >> b >> c;
    cout << a << b << c << endl;
    cin >> noskipws >> a >> b >> c;
    cout << a << b << c << endl;
    cout << (int)a << (int)b << (int)c << endl;
    return 0;
}
```

حال به یک ورودی و خروجی خاص از برنامه مذکور توجه کنید:

Input	Output
1 2 3 1 2 3	123 1 10 49 32

همانطور که مشاهده می‌کنید، در این ورودی دو ورودی یکسان 1 2 3 (سه عدد که با نویسه فاصله از هم جدا شده‌اند) به برنامه داده شده است. برنامه پس از دریافت خط اول به علت این که با پرچم skipws ورودی را

می‌خواند، دو نویسه فاصله را نادیده گرفته و مقادیر ۱، ۲ و ۳ را به ترتیب در متغیرهای *a*، *b* و *c* ذخیره می‌کند. ولی پس از دریافت خط دوم به علت نبود پرچم noskipws مقداری وضعیت تغییر می‌کند. توجیه خروجی برنامه در این حالت (از خط دوم تا چهارم خروجی) به عنوان تمرین به خواننده واگذار می‌شود. برای راهنمایی بیشتر، خط آخر برنامه مقادیر عددی نویسه‌های ذخیره شده درون سه متغیر را چاپ می‌کند. این مقادیر نیز در خط آخر خروجی قابل مشاهده است.

مثال بعدی تاثیر پرچم‌های hex، oct و dec را بر دو شیء cin و cout نشان می‌دهد.

```
#include <iostream>
using namespace std;
int main() {
    int i1, i2, i3;
    cin >> hex >> i1 >> oct >> i2 >> dec >> i3;
    cout << dec << i1 << " " << i2 << " " << i3 << endl;
    return 0;
}
```

نمونه‌ای از ورودی و خروجی مثال قبل به این صورت است:

Input	Output
20 20 20	32 16 20

به عنوان یک تمرین، سعی کنید نتیجه حالت‌های دیگر استفاده از پرچم‌های hex، oct و dec را بدون کدنویسی به دست آورید.

در مثال بعدی به طرز کار پرچم‌های fixed و scientific می پردازیم.

```
#include <iostream>
using namespace std;
int main() {
    double s = 1.0;
    cout << fixed << s << endl;
    cout << scientific << s << endl;
    return 0;
}
```

خروجی این برنامه ساده به این صورت است:

Output
1.000000 1.000000e+000

برای تعیین فضا در چاپ خروجی می‌توان تابع width را به کار برد. ثابت‌های تعیین موقعیت در هنگامی که فضای تعیین شده برای چاپ خروجی از طول خروجی بیشتر باشد، کاربرد دارند و موقعیت خروجی را در فضای تعیین شده، تنظیم می‌کنند. نمونه‌ای از کاربرد پرچم‌های موقعیت و تابع width در مثال بعد آمده است.

```
#include <iostream>
using namespace std;
int main() {
    int num = 123;
    cout.width(10);
    cout << right << num << endl;
    cout << left << num << endl;
    return 0;
}
```

در این مثال فراخوانی width(10) باعث می‌شود که به طور پیشفرض یک فضا به اندازه ۱۰ نویسه برای هربار چاپ خروجی توسط cout در نظر گرفته شود. به همین دلیل، چاپ عدد ۱۲۳ توسط پرچم right باعث می‌شود که ابتدا هفت نویسه فاصله و سپس عدد ۱۲۳ در خروجی حاضر شود. بنابراین خروجی این برنامه به این صورت خواهد بود:

```
Output
     123
123
```

در مثال قبل لازم به ذکر است که اگر طول خروجی بیشتر از فضای تعیین شده باشد، دیگر عملا تابع width تاثیری نخواهد داشت و برنامه بدون توجه به این تابع، فضای مورد نیاز برای نشان دادن خروجی را مصرف می‌کند.

تابع دیگری که کاربرد فراوانی برای کار با اعداد اعشاری دارد، تابع precision است؛ این تابع تعداد ارقام اعشاری را که باید در خروجی نمایش داده شوند، تعیین می‌کند. اگر تعداد ارقام اعشاری عدد مورد نظر، از دقت تعیین شده‌ی این تابع بیشتر باشد، اعشار عدد مورد نظر بسته به دقت تعیین شده، گرد خواهد شد. اگر مایل به تعیین دقت ثابتی هستید، به شکلی که در ارقام اعشار باقیمانده از عدد، رقم صفر چاپ شود، باید ثابت fixed را به کار ببرید. برای روشن‌تر شدن توضیح، به این نمونه کد توجه کنید:

```
#include <iostream>
using namespace std;
int main() {
    double f = 3.14159;
    cout.precision(5);
    cout << f << endl;
    cout.precision(10);
    cout << f << endl;
    cout << fixed;
    cout << f << endl;
    return 0;
}
```

خروجی این کد به این صورت خواهد بود:

```
Output
3.1416
3.14159
3.1415900000
```

تا اینجا، عملگرهای استخراج‌کننده را معرفی کرده و دیدیم که این عملگرها چگونه به استخراج داده‌ها می‌پردازند. همچنین به کار بستن بعضی تنظیمات بر روی آن‌ها را نیز آموختیم. حال کمی با توابعی کار می‌کنیم که به صورت بدون قالب به داده‌ها نگاه کرده و فقط به شکل مجموعه‌ای از نویسه‌ها با آن‌ها برخورد می‌کنند. دسته‌ای از این تابع‌ها از کلاس istream و دسته‌ای دیگر از کلاس ostream به ارث برده شده‌اند؛ در نتیجه، آشنایی با توابع این کلاس منجر به آشنایی با دو کلاس یاد شده نیز خواهد شد. پس از توضیح کوتاه این تابع‌ها، در ادامه برای هرکدام نمونه‌ای آورده می‌شود.

در جدول ۱۰,۲ فهرست توابع ارث برده شده از کلاس istream آمده است.

جدول ۱۰,۲. فهرست توابع ارث برده شده از کلاس istream

نام تابع	توضیح
get	خواندن یک یا چند نویسه از ورودی
getline	خواندن چند نویسه از ورودی
read	خواندن چند نویسه از ورودی
putback	قرار دادن یک نویسه در مکان آخرین نویسه خوانده شده از میانگیر و برگرداندن اشاره‌گر میانگیر به عقب (در اینجا منظور از اشاره‌گر، اشاره‌گر خواندن است).
unget	اشاره‌گر میانگیر را یکی به عقب حرکت می‌دهد؛ بنابراین امکان دوباره خواندن نویسه‌ی پایانی خوانده شده فراهم می‌شود.
tellg	مکان کنونی اشاره‌گر میانگیر را برمی‌گرداند.
seekg	اشاره‌گر میانگیر را به مکانی دلخواه جابجا می‌کند.
sync	نویسه‌های کنونی موجود در میانگیر را دور می‌ریزد و آن را با شرایط کنونی همگام می‌کند.

در جدول ۱۱,۲ فهرست توابع ارث برده شده از کلاس ostream نشان داده شده است.

جدول ۱۱,۲. فهرست توابع ارث برده شده از کلاس ostream

نام تابع	توضیح
put	نوشتن یک نویسه در خروجی
write	نوشتن چند نویسه در خروجی
tellp	مکان کنونی اشاره‌گر میانگیر را برمی‌گرداند (در اینجا منظور از اشاره‌گر، اشاره‌گر نوشتن است).
seekp	اشاره‌گر میانگیر را به مکانی دلخواه جابجا می‌کند.
flush	نویسه‌های موجود در میانگیر که هنوز در خروجی به کار بسته نشده‌اند را به کار بسته و در خروجی نمایش می‌دهد.

توابع به ارث برده شده از کلاس istream

توابع get، getline و read عملی مشابه با تفاوتی جزئی انجام می‌دهند. تابع get دارای چهار تعریف کاربردی زیر است (دارای تعاریف دیگری نیز است):

```
int get();
get(char& c);
get(char* s, streamsize n);
get(char* s, streamsize n, char delim);
```

دو تعریف نخست، یک نویسه را از ورودی می‌خوانند. تعریف اول، کد اسکی نویسه خوانده شده را به صورت یک عدد صحیح برمی‌گرداند و تعریف دوم، نویسه خوانده شده را در متغیر *c* قرار می‌دهد. تعریف سوم، به تعداد *n-1* نویسه از ورودی خوانده و در آرایه *s* قرار خواهد داد. تعریف آخر مانند تعریف سوم عمل می‌کند، با این تفاوت که پارامتر سوم آن، نویسه‌ی جداکننده[۲۵] دریافت می‌کند؛ در این حالت خواندن از ورودی تا جایی ادامه پیدا می‌کند که

۱- *n-1* نویسه خوانده شود، یا

۲- به نویسه جداکننده برسیم، یا

۳- به نویسه آخر خط برسیم (اگر نویسه جداکننده گذاشته نشده باشد).

نحوه استفاده از تابع get در این کد نشان داده شده است:

```
#include <iostream>
using namespace std;
int main() {
    char c1, c2;
    char arr1[10], arr2[10];
    char delim = 't';
    c1 = cin.get();
    cin.get(c2);
    cin.get(arr1, 5);
    cin.get(arr2, 10, delim);
    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;
    cout << "arr1: " << arr1 << endl;
    cout << "arr2: " << arr2 << endl;
    return 0;
}
```

نمونه‌ای از ورودی و خروجی مثال مذکور در ادامه آمده است:

Input	Output
SampleString	c1: S c2: a arr1: mple arr2: S

[۲۵] Delimiter

تابع getline می‌تواند عملیات مشابه دو تعریف پایانی تابع get را انجام دهد.

```
getline(char* s, streamsize n);
getline(char* s, streamsize n, char delim);
```

تابع read تنها می‌تواند عملیات مشابه تعریف سوم تابع get را انجام دهد.

```
read(char* s, streamsize n);
```

هر بار فراخوانی تابع getline فقط بر روی یک خط از ورودی عمل می‌کند (مگر اینکه نویسه آخر خط گذاشته شده باشد؛ در این صورت این نویسه نادیده گرفته می‌شود)؛ به این معنی که حتی اگر تعداد نویسه درخواست شده از تابع نیز خوانده شده باشد، فراخوانی بعدی این تابع از خط بعد، آغاز به خواندن خواهد کرد. این دو تابع به صورت خودکار در پایان آرایه، نویسه پایان رشته ('\0') را می‌گذارند.

تابع read مانند تابع get عمل می‌کند؛ با این تفاوت که در پایان رشته، نویسه‌ای قرار نخواهد داد. این تابع خواندن را تا جایی ادامه می‌دهد که تعداد نویسه‌های درخواست شده از آن بطور کامل خوانده شود. تنها نویسه‌ای که باعث توقف این تابع می‌شود، نویسه‌ی EOF است.

```
#include <iostream>
using namespace std;
int main() {
    char fname[256], lname[256], cname[7];
    cin.getline(fname, 256);
    cin.getline(lname, 256);
    cin.read(cname, 4);
    cin.read(cname + 4, 2);
    cname[6] = 0;
    cout << "First name: " << fname << endl;
    cout << "Last name: " << lname << endl;
    cout << "Country name: " << cname << endl;
    return 0;
}
```

نمونه‌ای از ورودی و خروجی مثال قبل در ادامه آمده است:

Input	Output
Amin Babadi IranIR	First name: Amin Last name: Babadi Country name: IranIR

همانطور که در کد قبلی مشاهده شد، پس از گرفتن همه ورودی‌ها، نویسه هفتم متغیر *cname* برابر نویسه صفر قرار داده شده است. به عنوان یک تمرین، سعی کنید تاثیر این دستور را در کد قبلی شرح دهید.

تابع putback و unget تا حدودی مانند هم عمل می‌کنند. هر دو تابع اشاره‌گر میانگیر را یکی به عقب برمی‌گردانند؛ با این تفاوت که تابع putback نویسه‌ای را به عنوان ورودی دریافت کرده و در مکانی که اشاره‌گر به عقب برگشته اشاره می‌کند، قرار می‌دهد. در نمونه زیر، هر دو تابع یک عمل را انجام می‌دهند:

```
#include <iostream>
using namespace std;
int main() {
    char c;
    int n;
    char str[256];
    cout << "Enter a number or a word: ";
    c = cin.get();
    if ((c >= '0') && (c <= '9')) {
        cin.putback(c); //or cin.unget()
        cin >> n;
        cout << "You have entered number " << n << endl;
    }
    else {
        cin.unget(); //or cin.putback (c);
        cin >> str;
        cout << " You have entered word " << str << endl;
    }
    return 0;
}
```

نمونه بالا، وارد شدن عدد یا رشته‌ای غیر عددی در ورودی را شناسایی می‌کند و به درستی آن را خوانده و در خروجی نمایش می‌دهد (فرض شده است که نویسه اول یک عدد یکی از نویسه‌های عددی '0' تا '9' است).

دو تابع tellg و seekg برای مدیریت اشاره‌گر میانگیر در نظر گرفته شده‌اند (شرح کاملی از مفهوم میانگیرها و نحوه کار با آنها در بخش‌های بعد آمده است). تابع seekg برای تنظیم محل نویسه بعدی برای خوانده شدن و تابع tellg برای آگاهی از محل این نویسه به کار می‌رود. نمونه‌ای از نحوه استفاده از این دو تابع در مثال بعدی آمده است.

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    std::ifstream fin("test.txt", std::ifstream::binary);
    if (fin) {
        fin.seekg(0, fin.end);
        int length = fin.tellg();
        cout << "Length of file is: " << length << endl;
    }
    return 0;
}
```

همانطور که گفته شد، جزییات نحوه کار با مفاهیم مربوط به میانگیرها در بخش‌های بعدی می‌آید. به همین دلیل در اینجا از ذکر جزییات مربوط به این مفاهیم خودداری می‌شود. کاری که نمونه کد مذکور انجام می‌دهد این است که یک میانگیر به نام fin برای کار با یک پرونده متنی به نام test.txt ایجاد می‌کند. سپس با استفاده از دستور seekg به انتهای میانگیر رفته و با استفاده از دستور tellg شماره مربوط به مکان نویسه بعدی (که در اینجا آخرین نویسه درون پرونده است) را محاسبه می‌کند. به این ترتیب، متغیر length در نهایت تعداد نویسه‌های درون پرونده متنی مذکور را درون خود خواهد داشت. لازم به ذکر است که برای اجرا شدن کامل این کد، ابتدا باید یک پرونده متنی به نام test.txt در کنار کد منبع برنامه ایجاد شود؛ در غیر این صورت میانگیر خروجی fin تولید نشده و قسمت مربوط به if در کد مذکور اجرا نخواهد شد.

تابع ignore برای همگام‌سازی میانگیر به کار می‌رود. اگر می‌خواهید محتوای کنونی میانگیر (ورودی) را دور ریخته و محتوای جدیدی در آن قرار داده (و بخوانید) این تابع را به شکل زیر به کار ببرید. البته در این حالت باید پس از وارد کردن هر کلمه حتما دکمه Enter را فشار دهید. این نمونه کد، پس از دریافت نخستین حرف از کلمه‌ی اول وارد شده، بقیه حروف آن را دور ریخته و آماده دریافت کلمه دوم می‌شود.

```
#include <iostream>
using namespace std;
int main() {
    char first, second;
    first = cin.get();
    cin.ignore(1000, '\n');
    second = cin.get();
    cout << "The first word began by " << first << endl;
    cout << "The second word began by " << second << endl;
    return 0;
}
```

توابع به ارث برده شده از کلاس ostream

تابع put و write یک یا چند نویسه را در خروجی می‌نویسند. نحوه استفاده از این دو تابع در مثال بعدی آمده است. به علت سادگی طرز کار تابع‌های put و write، تحلیل خروجی این مثال به خواننده واگذار می‌شود.

```
#include <iostream>
using namespace std;
int main() {
    char c = 'C';
    char eol = '\n';
    char arr[12] = "Write Array";
    arr[11] = '\n';
    cout.put(c);
    cout.put(eol);
    cout.write(arr, 12);
    return 0;
}
```

توابع tellp و seekp در نقطه مقابل تابع‌های tellg و seekg قرار دارند. تابع flush وظیفه به‌روزرسانی خروجی را بر عهده دارد؛ این تابع نویسه‌هایی را که در میانگیر خروجی وجود دارند، ولی هنوز در خروجی نشان داده نشده‌اند، در خروجی نمایش می‌دهد.

۲-۲-۱-۲- کلاس‌های cerr و clog

کلاس cerr معادل تابع perror برای کار با میانگیر خطای خروجی استاندارد و کلاس clog نیز برای کار با میانگیر ثبت جزییات در خروجی استاندارد به کار می‌رود. میانگیرهای این دو کلاس مستقل از یکدیگر هستند، ولی معمولا هر دو در خروجی استاندارد نمایش داده می‌شوند. به این نمونه توجه کنید:

```
#include <iostream>
using namespace std;
int main() {
    cout << "print in standard output" << endl;
    cerr << "print in error output" << endl;
    clog << "print in log output" << endl;
    return 0;
}
```

در نمونه مذکور، با اجرای عادی برنامه هر سه متن را در خروجی استاندارد مشاهده خواهید کرد. ولی در صورتی که برنامه را از طریق کنسول اجرا کرده و خروجی استاندارد را به یک فایل هدایت کنید، خواهید دید که تنها حاصل دو دستور cerr و clog چاپ می‌شود. در نمونه زیر، فرض شده است که فایل اجرای برنامه به نام a.exe در درایو D:\ قرار دارد.

```
D:\a.exe 1>output.txt
print in error output
print in log output
```

با اجرای دستور گفته شده در خط فرمان، یک پرونده متنی به نام output.txt در کنار فایل اجرایی برنامه (در اینجا a.exe) تولید شده و عبارت print in standard output در آن چاپ می‌شود. همانطور که می‌بینید، دو دستور بعدی به علت این که اجازه چاپ شدن در خروجی استاندارد (پرونده output.txt) را ندارند، در همان خط فرمان قرار گرفته‌اند.

۲-۲- کار ساده با پرونده‌های متنی

برای حل بسیاری از مسأله‌ها، کار با پرونده‌های متنی در ورودی و خروجی مورد نیاز است. در حل مسائلی که زمان اجرای آن‌ها مهم است، روشی که برای خواندن و نوشتن در پرونده‌ها به کار برده می‌شود، در حل مسأله بسیار تاثیرگذار است. در این بخش با کتابخانه fstream در سطحی ساده آشنا می‌شویم. در بخش‌های بعد، به طور کامل به کار با پرونده‌ها پرداخته می‌شود. با به‌کارگیری کتابخانه fstream می‌توان نوشتن و خواندن در پرونده‌ها را کاملا مشابه با ورودی و خروجی استاندارد انجام داد. تنها کاری که لازم است انجام دهید این است که ابتدا جریان ورودی

و خروجی را تعریف کنید. پس از آن بیشتر کارهای لازم را می‌توانید مشابه بخش قبل به سادگی انجام دهید. این کد نمونه ساده‌ای از این عملیات را نشان می‌دهد. برای اجرای این کد باید یک پرونده متنی به نام Input.txt که احتمالا حاوی حداقل یک عدد در ابتدای آن است، در پوشه اصلی برنامه وجود داشته باشد.

```
#include <fstream>
using namespace std;
int main() {
    ifstream fin("Input.txt");
    ofstream fout("Output.txt");
    fout << "Write In Output File" << endl;
    int ReadedNum;
    fin >> ReadedNum;
    return 0;
}
```

کلاس ifstream برای خواندن از یک پرونده و کلاس ofstream برای نوشتن در یک پرونده به کار می‌رود. این دو کلاس مسیر پرونده ورودی/خروجی را در سازنده خود دریافت می‌کنند. اگر فقط نام پرونده ورودی/خروجی نوشته شود، این پرونده‌ها در پوشه‌ای که کد برنامه قرار دارد، جستجو خواهند شد و اگر در این مسیر وجود نداشته باشند، اطلاعاتی خوانده/نوشته نخواهد شد. برای خواندن و نوشتن کافی است نام کلاس مربوطه همراه با دو عملگر << و >> (به ترتیب) به کار برده شود. در نمونه‌ی بالا، >> fout برای نوشتن در پرونده‌ی Output.txt و << fin برای خواندن از پرونده‌ی Input.txt به کار برده شده است. پس از اجرای این کد، پرونده Output.txt ساخته می‌شود (اگر وجود نداشته باشد) و محتوای زیر را خواهد داشت. اگر پرونده وجود داشته باشد، محتوای جدید جایگزین محتوای پیشین آن خواهد شد.

Output
Write In Output File

۲-۳- کار با رشته‌ها

بسیاری از ورودی‌ها باید به صورت رشته دریافت شده و پردازش شوند و گاهی به خروجی‌هایی در قالب رشته نیاز می‌شود. برای کار با رشته می‌توان کتابخانه‌ی string را به کار برد. با include کردن این کتابخانه می‌توان رشته‌ها را با هم مقایسه کرد، دو رشته را در هم ادغام کرد، در رشته‌ای به دنبال رشته‌ای دیگر گشت و ... .
نمونه کد بعدی نحوه مقایسه و ترکیب دو رشته را نشان می‌دهد:

```
#include <iostream>
#include <string>
using namespace std;
string str = "String P1";
int main() {
    cout << "The result of comparison of two strings is: " << (str <
"s") << endl;
    str += "  String P2";
    cout << str << endl;
    return 0;
}
```

خروجی این کد به این صورت خواهد بود:

Output
The result of comparison of two strings is: 1 String P1 String P2

به کمک اشیائی که از نوع کلاس string تعریف می‌شوند، می‌توان به تابع‌هایی(در این کتاب، از نام تابع برای تابع‌های درون یا بیرون کلاس استفاده می‌شود) دسترسی داشت که کار با رشته‌ها را ساده خواهند کرد.
تابع compare برای مقایسه‌ی دو رشته به کار می‌رود. این تابع اگر رشته اول کوچکتر از رشته دوم باشد، مقدار ۱- و اگر رشته دوم کوچکتر باشد مقدار ۱ و در صورت تساوی دو رشته، مقدار ۰ را بر می‌گرداند. نمونه کد بعدی طرز کار این تابع را نشان می‌دهد:

```
#include <iostream>
#include <string>
using namespace std;
string str = "String P1";
int main() {
    string str1 = "ABC";
    string str2 = "DEF";
    cout << "Result of comparing Str1 with Str2: " << str1.compare(str2)
<< endl;
    return 0;
}
```

خروجی این برنامه به این صورت است:

Output
Result of comparing Str1 with Str2: -1

تابع append رشته‌ای را به پایان رشته‌ای دیگر می‌چسباند.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str1 = "ABC";
    string str2 = "DEF";
    cout << "Str2 Concat to Str1: " << str1.append(str2) << endl;
    return 0;
}
```

خروجی این برنامه به این صورت است:

```
Output
Str2 Concat to Str1: ABCDEF
```

تابع find وجود یک زیررشته در رشته‌ای دیگر را بررسی می‌کند. اگر زیررشته‌ی مورد نظر، در رشته‌ی مقصد وجود داشته باشد، مکان آغاز آن در رشته بازگردانده می‌شود وگرنه عددی تصادفی در بازه‌ای نامتعارف برگردانده می‌شود که روشی برای بررسی و شناسایی آن در ادامه معرفی می‌شود.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str1 = "ABCDEF";
    cout << "Find a SubStr In Str1: " << str1.find("DEF", 0) << endl;
    return 0;
}
```

خروجی این کد به این صورت است:

```
Output
Find a SubStr In Str1: 3
```

برای بررسی اینکه آیا رشته‌ی جستجو شده، در رشته مقصد وجود دارد یا خیر، می‌توان ثابت string::npos را به کار برد. مقایسه مقدار خروجی تابع find با این ثابت نشان‌دهنده‌ی بودن یا نبودن رشته‌ی جستجو شده خواهد بود.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str1 = "ABCDEF";
    cout << "SubStr Searching Result: " << ((str1.find("DEF", 0) ==
string::npos) ? "False" : "True") << endl;
    return 0;
}
```

خروجی این کد به این صورت است:

Output
SubStr Searching Result: True

تابع insert زیررشته‌ای را در مکان مشخصی از رشته مقصد قرار می‌دهد. به این نمونه توجه کنید:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str1 = "ABCDEF";
    cout << "Insert SubStr in Str1: " << str1.insert(3, "SubStr") <<
endl;
    return 0;
}
```

در این مثال، زیررشته "SubStr" در مکان سوم از رشته "ABCDEF" قرار خواهد گرفت. خروجی این برنامه به ایـن صورت است:

Output
Insert SubStr in Str1: ABCSubStrDEF

تابع length طول کنونی رشته را برمی‌گرداند. به این مثال توجه کنید:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str1 = "ABCDEF";
    cout << "Length of Str1: " << str1.length() << endl;
    return 0;
}
```

خروجی این برنامه به این صورت خواهد بود:

Output
Length of Str1: 6

کلاس Streambuf کلاسی انتزاعی است که توابعی را برای کار با میانگیرها و مدیریت آن‌ها فراهم می‌کند. کلاس انتزاعی، کلاسی است که درون خود دست کم یک تابع انتزاعی داشته باشد. تابع انتزاعی، تابعی مجازی است که به جای گذاشتن بدنه برای آن، روبروی اعلان آن در کلاس، عملگر انتساب، سپس صفر و ; گذاشته می‌شود. از آنجا که این کلاس، کلاسی انتزاعی است، ساخت شیء از آن ممکن نیست؛ بنابراین دو کلاس filebuf و stringbuf از آن به ارث برده شده و آن را گسترش داده‌اند. کد زیر تعریف یک تابع انتزاعی را نشان می‌دهد. دقت کنید که این تعریف باید در داخل یک کلاس (یا ساختار) قرار گیرد.

```
virtual void func() = 0;
```

۴-۲- میانگیر پرونده

میانگیر پرونده، به پرونده‌ای ویژه متصل شده و کار با آن را به صورت آرایه‌ای از نویسه‌ها شبیه‌سازی می‌کند. بنابراین می‌توان از آن خواند و در آن نوشت؛ به مکانی ویژه پرش کرد و یا نویسه‌ی برداشته شده را به مکان پیشین خود برگرداند.

این کلاس در سرآیند fstream قرار دارد. از آنجا که این کلاس و کلاس stringbuf از کلاس streambuf ارث می‌برند، خیلی از توابع این دو کلاس مربوط به کلاس streambuf است که در این بخش به توضیح آن نیز پرداخته می‌شود.

برای آغازِ به‌کارگیری میانگیر پرونده، باید جریان پرونده‌ای را به آن متصل کرد. اگر پرونده باز باشد، می‌توان از میانگیر پرونده خواند و در آن نوشت؛ تغییرات انجام شده در پرونده نیز اعمال خواهند شد. برای اتصال میانگیر به یک پرونده‌ی از پیش ساخته شده به این صورت عمل می‌کنیم:

```
#include <fstream>
using namespace std;
int main() {
    fstream files("file.txt");
    filebuf* fb;
    fb = files.rdbuf();
    //…
    files.close();
    return 0;
}
```

می‌توان اتصال میانگیر به پرونده را به این صورت نیز انجام داد:

```
#include <fstream>
using namespace std;
int main() {
    ifstream is("file.txt");
    filebuf* fb;
    fb = is.rdbuf();
    //...
    fb->close();
    return 0;
}
```

به جای باز کردن پرونده در هنگام اعلان پرونده (در اینجا هنگام اعلان is)، می‌توان filebuf وصل شده به آن را مانند این کد باز نمود:

```
#include <fstream>
using namespace std;
int main() {
    ifstream is;
    filebuf* fb;
    fb = is.rdbuf();
    fb->open("file.txt", ios::in);
    //...
    fb->close();
    return 0;
}
```

برای ساخت میانگیر پرونده، اشاره‌گری از نوع filebuf تعریف می‌کنیم. جریان‌های پرونده (fstream، ifstream و ofstream) همگی دارای تابع rdbuf هستند؛ این تابع اشاره‌گری از نوع میانگیر پرونده برمی‌گرداند.

در نمونه کد اول، می‌توان هم در میانگیر نوشت و هم از آن خواند. ولی در نمونه کد دوم و سوم فقط می‌توان از میانگیر خواند، زیرا پرونده در حالت فقط خواندنی (ios::in) باز شده است.

تابع open پرونده‌ای را باز کرده و اشاره‌گری از نوع میانگیر برای آن می‌سازد؛ پارامتر اول این تابع، مسیر پرونده و پارامتر دوم آن، ثابتی است که حالت باز کردن پرونده را نشان می‌دهد. این حالت‌ها شش دسته دارند که در جدول ۱۲,۲ نشان داده شده‌اند.

جدول ۱۲,۲. حالت‌های مختلف باز کردن پرونده توسط تابع open

ثابت	حالت باز کردن پرونده
app	append: در هر عمل نوشتن در پرونده، اشاره‌گر موقعیت به پایان پرونده رفته و مقدار در آنجا نوشته خواهد شد.
ate	at end: هنگام باز شدن پرونده، اشاره‌گر موقعیت به پایان پرونده اشاره می‌کند.
binary	binary: پرونده از نوع دودویی باز می‌شود.
in	input: پرونده از نوع خواندنی باز می‌شود.
out	output: پرونده از نوع نوشتنی باز می‌شود.
trunc	truncate: محتوای کنونی پرونده نادیده گرفته شده و فرض می‌شود که پرونده خالی است. محتوای پیشین از میان نمی‌رود، مگر اینکه همراه با out به کار رود. یعنی ios::trunc | ios::out محتوای پرونده را پاک می‌کند.

پس از اتصال میانگیر به پرونده، می‌توان کارهایی را که روی پرونده انجام می‌شود، به کمک میانگیر شبیه‌سازی کرد. از آنجا که هنگام به‌کارگیری میانگیر، پرونده مقصد باید باز باشد، تابع is_open برای بررسی باز بودن پرونده

به کار برده می‌شود. این تابع در صورت باز بودن پرونده مقدار true را برخواهد گرداند. تابع close نیز برای بستن پرونده‌ی باز شده به کار می‌رود.

تابع pubseekoff برای پرش به مکانی مشخص از میانگیر به کار برده می‌شود. این پرش نسبی است و می‌تواند نسبت به آغاز، مکان کنونی و پایان میانگیر صورت گیرد. پارامترهای مربوط به این سه حالت در جدول ۱۳,۲ نشان داده شده است.

جدول ۱۳,۲. پارامترهای مربوط به مبدا پرش در تابع pubseekoff

ثابت	توضیح
ios_base::beg	پرش نسبت به آغاز میانگیر
ios_base::cur	پرش نسبت به مکان کنونی اشاره‌گر موقعیت میانگیر
ios_base::end	پرش نسبت به پایان میانگیر

این تابع دارای دو پارامتر است. نخستین پارامتر مقدار پرش را تعیین می‌کند (مقداری مثبت) و دومین پارامتر، مکانی که نسبت به آن باید پرش صورت گیرد. خروجی این تابع مقدار واقعی پرش صورت گرفته خواهد بود (از نوع long). به این مثال توجه کنید:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    long size;
    filebuf* pbuf;
    fstream filestr("file.txt");
    pbuf = filestr.rdbuf();
    size = pbuf->pubseekoff(0, ios_base::end);
    filestr.close();
    cout << "size of file is " << size << endl;
    return 0;
}
```

در مثال مذکور، برنامه ابتدا به پایان پرونده پرش می‌کند. از آنجا که میانگیر به صورت آرایه‌ای از نویسه‌ها با پرونده برخورد می‌کند، بنابراین مقدار خروجی تابع pubseekoff، تعداد نویسه‌هایی است که پرش از روی آن‌ها صورت گرفته است. اگر از آغاز میانگیر به پایان آن پرش انجام شود، در واقع تعداد همه‌ی نویسه‌های موجود در میانگیر به دست می‌آید که همان اندازه پرونده است.

تابع pubseekpos نیز عملکردی مشابه با تابع pubseekoff دارد؛ با این تفاوت که این تابع به صورت مطلق پرش خواهد کرد. یعنی مقدار پارامتر ورودی این تابع، مکان نسبت به آغاز پرونده (میانگیر) است. میانگیر دارای دو اشاره‌گر موقعیت، یکی برای نوشتن و یکی برای خواندن است. این تابع می‌تواند هر یک از این اشاره‌گرها را به عقب یا جلو

حرکت دهد. اگر پارامتر دوم این تابع، خالی رها شود، هر دو اشاره‌گر موقعیت، بـه مکـان تعیـین شـده حرکـت داده می‌شوند وگرنه یکی از آن‌ها بر اساس مقادیر نشان داده شده در جدول ۱۴,۲ حرکت داده خواهد شد.

جدول ۱٤,۲. اشاره‌گرهایی که تابع pubseekpos قادر به حرکت دادن آنها است.

ثابت	توضیح
ios_base::in	جابجایی اشاره‌گر خواندن
ios_base::out	جابجایی اشاره‌گر نوشتن

این نمونه کد از نویسه دهم پرونده، نه نویسه را خوانده و در آرایه buffer قرار می‌دهد:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    int n;
    filebuf* pbuf;
    char buffer[10];
    fstream filestr("file.txt");
    pbuf = filestr.rdbuf();
    pbuf->pubseekpos(10);
    pbuf->sgetn(buffer, 9);
    buffer[9] = '\0';
    filestr.close();
    cout << buffer << endl;
    return 0;
}
```

تابع sgetn() برای خواندن از میانگیر به کار می‌رود؛ در نمونه کد قبلی، ده نویسه از میانگیر، توسط این تابع خوانـده شده (از مکانی که اشاره‌گر موقعیت اشـاره می‌کنـد) و در آرایـه buffer قـرار می‌گیـرد. در نهایـت آرایـه buffer در خروجی استاندارد چاپ می‌شود. سعی کنید با تغییر در پرونده ورودی (در اینجا file.txt) رفتار کد قبل را تحلیل کنید. هنگامی که مقدار جدیدی در میانگیر نوشته می‌شود و یا در آن تغییر می‌کند، این تغییرها در زمان بسته شدن پرونده اعمال خواهند شد. برای اینکه تغییرات به صورت آنی صورت گیرند، می‌توان تابع pubsync را به کار برد. اگـر ایـن تابع کار خود را با موفقیت به انجام برساند، مقدار ۰ وگرنه مقدار ۱- را برمی‌گرداند. در نمونه کد بعد، پـس از نوشـته شدن رشته نخست اگر پرونده‌ی مقصد باز شود، تغییرات دیده می‌شود.

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    filebuf* pbuf;
    ofstream ostr("File.txt");
    pbuf = ostr.rdbuf();
    pbuf->sputn("First sentence\n", 15);
    pbuf->pubsync();
    pbuf->sputn("Second sentence\n", 16);
    ostr.close();
    return 0;
}
```

در ادامه به دستورات خواندن از میانگیر پرداخته می‌شود. تابع sgetc برای خواندن نویسه‌ی کنونی (نویسه‌ای که اشاره‌گر موقعیت خواندن به آن اشاره می‌کند) به کار می‌رود؛ دو تابع دیگر نیز برای این کار وجود دارند که افزون بر خواندن نویسه، اشاره‌گر موقعیت را به جلو می‌برند؛ تابع sbumpc نویسه کنونی را خوانده و اشاره‌گر را یکی زیاد می‌کند و تابع snextc ابتدا اشاره‌گر را یکی زیاد کرده، سپس نویسه مورد نظر را می‌خواند؛ در واقع این تابع نویسه بعدی را خوانده و برمی‌گرداند. نمونه کد بعدی به این دو روش محتوای پرونده (میانگیر) را خوانده و در خروجی استاندارد نشان می‌دهد:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    char ch;
    filebuf* pbuf;
    ifstream istr("file.txt");
    pbuf = istr.rdbuf();
    //Read 1
    do {
        ch = pbuf->sgetc();
        cout << ch;
    }while (pbuf->snextc() != EOF);
    cout << endl << endl;
    //Read 2
    pbuf->pubseekoff(0, ios_base::beg);
    while ((ch = pbuf->sbumpc()) != EOF)
        cout << ch;
    istr.close();
    return 0;
}
```

تابع sgetn برای خواندن چند نویسه از میانگیر به کاربرده می‌شود؛ پارامتر نخست این تابع، اشاره‌گر به آرایه‌ای از نویسه‌ها و پارامتر دوم آن، تعداد نویسه‌هایی است که می‌خواهیم آن‌ها را بخوانیم. به این مثال توجه کنید:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    char ch;
    filebuf* pbuf;
    ifstream istr("file.txt");
    pbuf = istr.rdbuf();
    char content[11];
    pbuf->sgetn(content, 10);
    content[10] = '\n';
    cout.write(content, 11);
    istr.close();
    return 0;
}
```

حال به توابع خواندن در میانگیر می‌رسیم. دو تابع sputc و sputn به ترتیب برای نوشتن یک و چند نویسه در میانگیر به کار می‌روند. این دو تابع پس از هر بار نوشتن یک نویسه، اشاره‌گر موقعیت نوشتن را یکی به جلو می‌برند. به مثال بعد توجه کنید:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    char temp[] = "sample text";
    filebuf* pbuf;
    ofstream ostr("test.txt");
    pbuf = ostr.rdbuf();
    pbuf->sputc('\n');
    pbuf->sputn(temp, sizeof(temp) - 1);
    ostr.close();
    return 0;
}
```

با اجرای کد قبلی پرونده ای به نام test.txt در کنار کد منبع ایجاد شده و محتوای آن به این صورت خواهد بود:

Output

```
sample text
```

تابع sungetc برای برگرداندن آخرین نویسه خوانده شده به مکان پیشین خود به کار گرفته می‌شود؛ تابع sputbackc نیز مانند تابع قبل عمل می‌کند؛ با این تفاوت که این تابع، یک نویسه را به عنوان پارامتر ورودی دریافت کرده و اگر آخرین نویسه (که باید برگردانده شود) با این نویسه ورودی برابر باشد، آن را به محل پیشین خود

برمی‌گرداند؛ اگر نویسه‌ها برابر نباشند، هیچ نویسه‌ای به محل پیشین برگردانده نمی‌شود؛ در این حالت، مقدار خروجی تابع EOF خواهد بود. این مثال چگونگی کار این دو تابع را به روشنی نشان می‌دهد:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    char temp[] = "sample text";
    int i = 0;
    filebuf* pbuf;
    ofstream ostr("test.txt");
    pbuf = ostr.rdbuf();
    //Write 1
    for (i = 0; i < sizeof(temp) - 1; i++)
        pbuf->sputc(temp[i]);
    //Write 2
    pbuf->sputn(temp, sizeof(temp) - 1);
    ostr.close();
    char buffer[20];
    ifstream istr("test.txt");
    pbuf = istr.rdbuf();
    for (i = 0; i < 3; i++)
        buffer[i] = pbuf->sbumpc();
    buffer[i] = 0;
    cout << buffer << '\t';
    pbuf->sungetc();
    cout << char(pbuf->sbumpc()) << '\t';
    cout << char(pbuf->sbumpc()) << '\t';
    // unget last char (x)
    pbuf->sputbackc('p');
    cout << char(pbuf->sbumpc()) << '\t';
    // fail to unget last char (e)
    pbuf->sputbackc('t');
    cout << char(pbuf->sbumpc()) << '\t' << endl;
    istr.close();
    return 0;
}
```

خروجی این کد به این صورت است:

```
Output
sam    m    p    p    t
```

برای کار با ورودی و خروجی از پایانه نیز می‌توان همین روش را به کار برد و نویسه‌ی خوانده شده را به ورودی برگرداند. کد زیر ابتدا تعدادی حرف و سپس یک عدد را خوانده و نمایش می‌دهد. در این کد، حرف‌ها نادیده گرفته

می‌شوند تا به نخستین رقم از یک عدد برسیم؛ آن‌گاه این رقم خوانده شده به ورودی برگردانده می‌شود تا به سادگی بتوان عدد را یکجا خواند.

```
#include <iostream>
#include <cstdio>
using namespace std;
int main() {
    char ch;
    long n;
    streambuf * pbuf;
    pbuf = cin.rdbuf();
    do {
        ch = pbuf->sbumpc();
        if ((ch >= '0') && (ch <= '9')) {
            pbuf->sungetc();
            cin >> n;
            cout << "You entered number " << n << endl;
            break;
        }
    } while (ch != EOF);
    return 0;
}
```

مثالی از ورودی و خروجی کد قبلی به این صورت است:

Input	Output
asdf1234	You entered number 1234

۲-۵- میانگیر رشته

میانگیر stringbuf برای مدیریت جریان‌های رشته‌ای طراحی شده است. این میانگیر نیز به دلیل به ارث بردن از کلاس streambuf، توابع مشابه زیادی با کلاس filebuf دارد که در بخش پیشین به آن پرداخته شد. این کلاس در سرآیند sstream قرار دارد. در واقع جریان‌های رشته‌ای در پس‌زمینه، یک میانگیر رشته را به کار می‌برند. میانگیر رشته دارای همه توابع گفته شده برای کلاس filebuf است.

میانگیر رشته‌ای دارای تابعی به نام str است که محتوای میانگیر را به صورت یک رشته برمی‌گرداند. این نمونه چگونگی کار میانگیر را به روشنی نشان می‌دهد:

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main() {
    char temp[] = "sample text";
    stringbuf sbuf;
    sbuf.sputn(temp, sizeof(temp) - 1);
    sbuf.sputn("\nadd new text in new line\n", 26);
    string strall = sbuf.str();
    cout << strall;
    return 0;
}
```

خروجی این کد به این صورت خواهد بود:

```
Output
sample text
add new text in new line
```

از آنجا که در این نمونه، میانگیر رشته به هیچ جریانی متصل نیست، به صورت اشاره‌گر تعریف نشده است؛ پس از مطالعه جریان‌های رشته‌ای، می‌توانید میانگیر رشته‌ای را به جریان رشته‌ای متصل کرده و آن را به کار ببرید.

۲-۶- ورودی و خروجی پرونده

برای کار با ورودی و خروجی پرونده در زبان سی++، هم می‌توان کتابخانه‌های سی++ و هم کتابخانه‌های مربوط به زبان سی را به کار برد. کتابخانه‌های سی++ تا حدودی کار با ورودی و خروجی پرونده را آسان کرده‌اند. دو تابع fprintf و fscanf از توابع مربوط به زبان سی هستند که کار با پرونده را کمی سریع‌تر از توابع زبان سی++ انجام می‌دهند.

۲-۶-۱- دستورات مربوط به زبان سی

برای به‌کارگیری دستورهای ورودی و خروجی پرونده در سی ، به سرآیند cstdio نیاز است.

۲-۶-۱-۱- توابع fprintf و fscanf

از آنجا که این دو تابع عملکردی کاملا مشابه با دو تابع printf و scanf که در ابتدای فصل توضیح داده شدند، دارند؛ از توضیح بخش‌های مشترک آن‌ها خودداری کرده و به توضیح بخش‌های دیگر این دو تابع می‌پردازیم.

برای آغاز کار با پرونده، باید ابتدا، شیءای از جنس اشاره‌گر به پرونده (FILE*) تعریف کنید. سپس، باید پرونده را باز کرده و اعمال ورودی و خروجی مورد نظرتان را بر روی آن انجام داده و در پایان آن را ببندید. باز و بسته کردن پرونده توسط دو تابع fopen و fclose صورت می‌گیرد.

اعلان تابع fopen به این صورت است:

```
FILE * fopen(const char* filename, const char* mode);
```

تابع fopen دو پارامتر ورودی می‌پذیرد که ورودی اول، آدرس پرونده مورد نظر و ورودی دوم، حالت باز کردن پرونده (خواندن، نوشتن و ...) است. ورودی دوم از نوع رشته‌ای بوده و دارای قالبی بر پایه‌ی جدول ۱۵,۲ است.

جدول ۱۵,۲. حالت‌های مختلف باز کردن پرونده توسط تابع fopen

قالب رشته	حالت باز کردن پرونده
r	پرونده را برای خواندن باز می‌کند؛ پرونده مورد نظر باید وجود داشته باشد، وگرنه اشاره‌گر پرونده دربردارنده مقدار NULL و خطایی در میانگیر خطا نوشته خواهد شد.
w	پرونده برای نوشتن ساخته خواهد شد؛ اگر پرونده پیش از این موجود باشد، محتوای پیشین آن پاک می‌شود.
a	محتوای پیشین پرونده حفظ شده و داده‌های جدید به پایان پرونده افزوده می‌شوند؛ اگر پرونده وجود نداشته باشد، ایجاد می‌شود.
r+	پرونده برای خواندن و نوشتن باز خواهد شد؛ پرونده باید وجود داشته باشد.
w+	پرونده‌ای جدید برای خواندن و نوشتن ساخته می‌شود؛ اگر پرونده پیش از این موجود باشد، محتوای پیشین آن پاک می‌شود.
a+	پرونده برای خواندن و نوشتن (فقط در پایان پرونده) باز خواهد شد. برای خواندن از هر نقطه‌ای از پرونده می‌توانید اشاره‌گر خواندن را با به‌کارگیری توابعی چون fseek و rewind به محل مورد نظر انتقال دهید؛ ولی نوشتن فقط در پایان پرونده انجام می‌شود؛ اگر پرونده وجود نداشته باشد، ایجاد خواهد شد.

همه‌ی قالب‌های رشته‌ای بالا، پرونده را در حالت متنی باز خواهند کرد؛ برای باز کردن پرونده به صورت دودویی می‌توانید به پایان (یا میان) قالب‌های رشته‌ای معرفی شده، نویسه 'b' را بیفزایید ("rb"، "wb"، "ab"، "r+b"، "w+b"، "a+b"، "rb+"، "wb+" یا "ab+").

البته در نسخه‌های جدید ویژوال استودیو تابع fopen با نام fopen_s تعریف شده است و نحوه اعلان آن به این صورت است:

```
errno_t fopen_s(FILE **_Stream, const char *_FileName, const char
*_Mode);
```

تابع fclose اشاره‌گری به پرونده باز شده را دریافت کرده و آن را می‌بندد.

```
int fclose(FILE* stream);
```

همه‌ی توضیحات داده شده برای دو تابع printf و scanf، درباره‌ی توابع fprintf و fscanf نیز صدق می‌کند. از این رو برای توضیح چهار تابع اشاره شده، به این نمونه بسنده می‌شود (در برخی نسخه‌های جدید ویژوال استودیو تابع fscanf با نام fscanf_s تعریف شده است):

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main() {
    FILE* FileIn;
    FILE* FileOut;
    char name[100];
    FileIn = fopen("in.txt", "r"); //fopen_s(&FileIn, "in.txt", "r");
    FileOut = fopen("out.txt", "w"); //fopen_s(&FileOut, "out.txt","r");
    for (int n = 0; n < 3; n++) {
        fscanf(FileIn, "%s", name);
        fprintf(FileOut, "Name %d [%-10.10s]\n", n, name);
    }
    fclose(FileIn);
    fclose(FileOut);
    return 0;
}
```

نمونه کد قبلی، سه رشته نخست موجود در پرونده ورودی را خوانده و در پرونده خروجی با قالب ویژه‌ای می‌نویسد. عبارت "10.10" کمترین و بیشترین طول رشته را ده قرار می‌دهد؛ بنابراین اگر طول رشته کمتر از ده باشد، بقیه فضا با نویسه فاصله پر خواهد شد و اگر طول رشته بیشتر از ده باشد، بقیه نویسه‌های آن نادیده گرفته خواهد شد. نویسه '-' رشته را در فضای مربوط به آن از سمت چپ تنظیم خواهد کرد.

نمونه از محتوای پرونده ورودی و خروجی (پس از اجرای برنامه) به این صورت است:

Input	Output
Yoosofan Mohsen-Biglari Mirzaei Ali	Name 1 [Yoosofan] Name 2 [Mohsen-Big] Name 3 [Mirzaei]

تابع feof رسیدن به پایان پرونده را بررسی می‌کند. اگر اشاره‌گر خواندن یا نوشتن (بسته به حالت باز شدن پرونده) به پایان پرونده رسیده باشد، این تابع مقداری غیر صفر (true) وگرنه مقدار صفر (false) را برمی‌گرداند. به مثال بعد توجه کنید:

```
#include <iostream>
using namespace std;
int main() {
    FILE* pFile;
    pFile = fopen("File.txt", "rb"); //fopen_s(&pFile, "File.txt","rb");
    long n = 0;
    while (!feof(pFile)) {
        fgetc(pFile);
        n++;
    }
    fclose(pFile);
    printf("Total number of bytes: %d\n", n - 1);
    return 0;
}
```

نمونه‌ی بالا، پرونده را در حالت دودویی باز کرده، تعداد بایت‌های موجود در آن را شمرده و چاپ می‌کند؛ شمارش بایت‌ها تا زمانی که به پایان پرونده نرسیده‌ایم، ادامه پیدا می‌کند.

۲-۶-۱-۲- تابع‌های fgetc و fgets

تابع fgetc اشاره‌گر پرونده‌ی باز شده را به عنوان ورودی دریافت کرده و نویسه‌ای که اشاره‌گر درونی پرونده به آن اشاره می‌کند، خوانده و بر می‌گرداند. اعلان این تابع به این صورت است:

```
int fgetc(FILE* stream);
```

تابع fgets سه پارامتر ورودی زیر را دریافت کرده، تعداد نویسه‌های درخواست شده را در صورت امکان خوانده و در آرایه‌ای از نویسه‌ها قرار می‌دهد. پارامترهای ورودی این تابع عبارتند از:

۱- آرایه‌ای از نویسه‌ها،

۲- تعداد نویسه‌هایی که باید از پرونده خوانده شود، و

۳- اشاره‌گری به پرونده باز شده.

```
char* fgets(char* str, int num, FILE* stream);
```

این تابع خواندن نویسه‌ها از پرونده را تا جایی ادامه می‌دهد که اشاره‌گر پرونده به نویسه آخر خط یا آخر پرونده برسد؛ با این تفاوت که نویسه آخر خط، خود نیز خوانده شده و در آرایه قرار داده می‌شود. مثال بعدی چگونگی کار با این دو تابع را نشان می‌دهد:

```
#include <iostream>
using namespace std;
int main() {
    char str1[5], str2[5];
    FILE* pFile;
    pFile = fopen("File.txt", "r"); //fopen_s(&pFile, "File.txt", "r");
    fgets(str1, 5, pFile);
    fgetc(pFile);
    fgets(str2, 5, pFile);
    fclose(pFile);
    printf("1: %s\n", str1, str2);
    printf("2: %s\n", str1, str2);
    return 0;
}
```

نمونه‌ای از محتوای پرونده ورودی و خروجی متناظر با آن به این صورت است:

Input	Output
Amin Babadi	1: Amin 2: Amin

۲-٦-۱-۳- تابع‌های fputc و fputs

عملکرد این دو تابع، برعکس دو تابع fgetc و fgets است. نمونه زیر گویا است. همان‌طور که در این نمونه مشخص است، این توابع قادر به خواندن از ورودی استاندارد نیز هستند. به علت سادگی این مثال، تحلیل آن به خواننده واگذار می‌شود.

```
#include <iostream>
using namespace std;
int main() {
    FILE* pFile;
    char sentence[256];
    pFile = fopen("file.txt", "a"); //fopen_s(&pFile, "file.txt", "a");
    printf("Enter sentence to append: ");
    fgets(sentence, 255, stdin);
    fputc('\0', pFile);
    fputs(sentence, pFile);
    fclose(pFile);
    return 0;
}
```

۲-٦-۱-٤- تابع‌های fseek، rewind و ftell

توابعی که در این بخش به آن‌ها پرداخته می‌شود برای کنترل اشاره‌گر درونی پرونده به کار می‌روند. تابع ftell مکان کنونی اشاره‌گر درونی پرونده را برمی‌گرداند. تابع rewind اشاره‌گر پرونده را به آغاز پرونده جابجا می‌کند. تابع fseek

برای پرش نسبی در پرونده به کار می‌رود؛ این تابع نسبت به یک مبدا که در واقع پارامتر سوم آن است، پرش می‌کند. ساختار این تابع به این صورت است:

```
int fseek(FILE* stream, long int offset, int origin);
```

پارامتر اول، اشاره‌گری به پرونده باز شده است؛ پارامتر دوم مقدار پرش است که نسبت به مبدا (پارامتر سوم) محاسبه می‌شود. پارامتر سوم، یکی از سه مقدار جدول ۱۶٫۲ را می‌پذیرد.

جدول ۱۶٫۲. سه حالت مختلف ممکن برای تعیین مبدا پرش در تابع fseek

قالب مبدا	توضیح
SEEK_SET	آغاز پرونده
SEEK_CUR	موقعیت کنونی اشاره‌گر درونی پرونده
SEEK_END	پایان پرونده

تابع rewind یک حالت ویژه از تابع fseek به این صورت است:

```
fseek(stream, 0, SEEK_SET);
```

نمونه بعدی چگونگی کار سه تابع معرفی شده را توضیح می‌دهد:

```
#include <iostream>
using namespace std;
int main() {
    FILE* pFile;
    pFile = fopen("File.txt", "w"); //fopen_s(&pFile, "File.txt", "w");
    fputs("  This is an apple.", pFile);
    rewind(pFile);
    fputs("1.", pFile);
    fseek(pFile, 11, SEEK_SET);
    fputs(" sam", pFile);
    fseek(pFile, 0, SEEK_END);
    fprintf(pFile, "\nSize of me without current line: %ld Bytes",
ftell(pFile));
    fclose(pFile);
    return 0;
}
```

نمونه بالا در آغاز، جمله‌ای را در پرونده می‌نویسد؛ سپس به ابتدای پرونده پرش کرده و مقداری (رشته‌ای) را به آن می‌افزاید؛ در پایان، به میانه‌ی رشته پرش کرده و رشته را کامل می‌کند. در ادامه، با به‌کارگیری تابع ftell حجم

پرونده را محاسبه، به پایان پرونده پرش کرده و رشته‌ای را در آنجا می‌نویسد. محتوای پرونده خروجی در پایان به صورت زیر خواهد بود:

۲-۶-۱-۵- تابع ferror

تابع ferror برای بررسی بروز خطا در کار با پرونده در نظر گرفته شده است. این تابع اگر خطایی هنگام کار با پرونده رخ دهد، مقداری غیر صفر و در غیر این صورت، مقدار صفر را برمی‌گرداند.
تابع دیگری که مرتبط با این بخش است، تابع clearerr است؛ این تابع خطاهای ایجاد شده را پاک می‌کند؛ بنابراین پس از فراخوانی این تابع، تابع ferror همیشه مقدار صفر را برخواهد گرداند. به این نمونه توجه کنید:

```
#include <iostream>
using namespace std;
int main() {
    FILE* pFile;
    pFile = fopen("File.txt", "r"); //fopen_s(&pFile, "File.txt", "r");
    fputc('x', pFile);
    if (ferror(pFile))
        perror("Error Writing to File.txt");
    clearerr(pFile);
    if (ferror(pFile))
        perror("Error Writing to File.txt");
    fclose(pFile);
    return 0;
}
```

خروجی این نمونه به این صورت است:

Output
Error Writing to File.txt: Bad file descriptor

۲-۶-۱-۶- تابع fflush

تغییراتی که در محتوای پرونده‌ها ایجاد می‌شود، معمولا تا زمان بسته شدن آن، اعمال نمی‌شود. تابع fflush محتوای پرونده را بدون بسته شدن، بروز می‌کند و داده‌هایی که در میانگیر قرار دارند و هنوز در پرونده نوشته نشده‌اند، از میانگیر حذف کرده و در پرونده قرار می‌دهد. دستور fflush(NULL) این عمل را بر روی همه پرونده‌های باز برنامه‌ی کنونی انجام می‌دهد. اگر در حالی که برنامه در حال اجراست، محتوای پرونده مورد نظر را بررسی کنید، متوجه عملکرد این تابع خواهید شد. طرز استفاده از این تابع در مثال بعدی مشخص است.

```
#include <iostream>
using namespace std;
int main() {
    FILE *pFile1, *pFile2;
    pFile1 = fopen("File1.txt", "w");//fopen_s(&pFile1,"File1.txt","w");
    pFile2 = fopen("File2.txt", "w");//fopen_s(&pFile2,"File2.txt","w");
    fputs("string1", pFile1);
    fputs("string2", pFile2);
    fflush(pFile1);
    fflush(NULL);
    fclose(pFile1);
    fclose(pFile2);
    return 0;
}
```

۲-۶-۲- دستورات مربوط به زبان سی++

برای به‌کارگیری دستورهای مربوط به ورودی و خروجی پرونده در زبان سی++، سرآیند fstream[۲۶] را به کار می‌بریم. این کلاس، خود از کلاس iostream به ارث برده که در ابتدای این فصل بطور کامل با آن آشنا شدیم؛ در این بخش بطور کوتاه با چگونگی کار این کلاس آشنا می‌شویم و مثال‌هایی از کاربردهای آن را مرور می‌کنیم.

کلاس fstream قادر به هر دو عمل ورودی و خروجی است. برای باز کردن یک پرونده توسط این کلاس می‌توانید تابع open را به کار ببرید؛ البته سازنده این کلاس نیز قادر به انجام این عمل است. این تابع دارای دو پارامتر است که یکی آدرس پرونده مورد نظر و دیگری حالت باز کردن پرونده را مشخص می‌کند؛ پارامتر دوم این تابع پیش از این توضیح داده شده است. به این مثال توجه کنید:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
    fstream file_r, file_w;
    file_r.open("filein.txt", fstream::in);
    file_w.open("fileout.txt", fstream::out);
    string strin = "";
    string strout = "this is a sample";
    file_r >> strin;
    file_w << strout << endl;
    file_r.close();
    file_w.close();
    return 0;
}
```

[۲۶] Input/Output File Stream

تابع close پرونده باز شده را می‌بندد. تابع is_open برای بررسی باز یا بسته بودن پرونده به کار می‌رود.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
    fstream filestr("File.txt");
    if (filestr.is_open())
        cout << "File successfully opened" << endl;
    else
        cout << "Error opening file" << endl;
    return 0;
}
```

دیگر توابع این کلاس کاملا مشابه کلاس iostream است؛ بنابراین در ادامه تنها به برخی از آنها که پیش از این توضیح داده نشده‌اند، پرداخته می‌شود. با این حال، به علت شباهت بسیار زیادی که این توابع به توابع معرفی شده در بخش‌های قبل، تحلیل کدها به عهده خواننده گذاشته شده است.

۲-٦-۲-۱- تابع tellg و seekg

این دو تابع پیش از این توضیح داده شدند. در این بخش فقط نمونه کدی از آن‌ها نوشته می‌شود. در نمونه زیر با به کارگیری این دو تابع همه‌ی پرونده را یکجا خوانده و در خروجی چاپ می‌کنیم؛ برای این منظور، ابتدا تعداد نویسه‌های موجود در پرونده را شمرده و آرایه‌ای به آن طول می‌سازیم.

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    int length;
    char* buffer;
    ifstream is;
    is.open("File.txt", fstream::binary);
    is.seekg(0, ios::end);
    length = is.tellg();
    is.seekg(0, ios::beg);
    buffer = new char[length];
    is.read(buffer, length);
    is.close();
    cout.write(buffer, length);
    cout << endl;
    return 0;
}
```

۲-۶-۲-۲- تابع flush

این تابع مشابه تابع fflush بخش قبل است و برای به‌روزرسانی محتویات میانگیر در خروجی به کار می‌رود. با به کارگیری این تابع، به صورت همزمان در یک پرونده، رشته‌ای را نوشته و همان رشته را می‌خوانیم. به این نمونه توجه کنید:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
    string strin = "";
    string strout = "this is a sample";
    fstream file_rw("File.txt", fstream::out | fstream::in);
    file_rw << strout;
    file_rw.flush();
    file_rw.seekg(0, fstream::beg);
    file_rw >> strin;
    file_rw.close();
    return 0;
}
```

همان‌طور که می‌بینید، برای باز کردن پرونده در چند حالت (خواندن و نوشتن به صورت همزمان)، می‌توان قالب‌های مورد نظر را با هم OR (منطقی) کرد.

در نمونه بالا، پس از نوشتن رشته this is a sample در پرونده، با به‌کارگیری تابع flush محتوای آن را بروز و سپس به آغاز پرونده پرش کردیم؛ سپس از آغاز پرونده یک کلمه را می‌خوانیم. به این نکته توجه داشته باشید که فراخوانی تابع seekg، خود باعث همگام‌سازی محتوای میانگیر و پرونده خواهد شد؛ بنابراین می‌توان فراخوانی تابع flush را از نمونه بالا حذف کرد.

۲-۷- جریان‌های ورودی و خروجی رشته‌ای

تا اینجا با ورودی و خروجی استاندارد و پرونده آشنا شدیم. دیدیم که در ورودی و خروجی استاندارد، عملگرهای استخراجی، رشته‌ای از نویسه‌ها را دریافت کرده و از آن‌ها، انواع مورد نظر ما را استخراج می‌کردند. در این بخش با جریان‌های ورودی و خروجی رشته‌ای آشنا می‌شویم. پیش از این، از ورودی و خروجی استاندارد، انواع مورد نظرمان را دریافت می‌کردیم و یا این کار را با پرونده‌ها انجام می‌دادیم. در این بخش این عمل را بر روی رشته‌ها انجام می‌دهیم؛ به این شکل که رشته‌ای دربردارنده‌ی نویسه‌ها را در یک جریان رشته‌ای قرار داده و با به‌کارگیری عملگرهای استخراج و دیگر توابع کاربردی این کلاس، انواع مورد نظرمان را می‌خوانیم و می‌نویسیم.

کلاسی که در این بخش با آن آشنا خواهیم شد، کلاس stringstream[۲۷] نام دارد. این کلاس از کلاس iostream به ارث برده است؛ بنابراین تا اندازه‌ای با توابع آن آشنا هستید. در این بخش فقط چگونگی کار با این کلاس توضیح داده می‌شود. برای به‌کارگیری این کلاس نیاز به سرآیند sstream داریم. این کلاس در پس‌زمینه، از میانگیر stringbuf استفاده می‌کند.

برای قرار دادن آرایه‌ای از نویسه‌ها در شیءای از این کلاس، می‌توان به یکی از این دو روش عمل کرد:

۱- به‌کارگیری سازنده کلاس

۲- به‌کارگیری تابع str

نحوه به کارگیری هر دو روش در این مثال نشان داده شده است:

```
#include <sstream>
using namespace std;
int main() {
    stringstream ss1("string1 stream");
    stringstream ss2;
    ss2.str("string2 stream");
    return 0;
}
```

تابع str افزون بر کاربردی که گفته شد، دارای یک پیاده‌سازی دیگر نیز است؛ تعریف دوم این تابع پارامتر ورودی نداشته و آرایه‌ای از نویسه‌ها که در واقع، در میانگیر آن قرار دارند را برمی‌گرداند. این کلاس برای اجرای پیاده‌سازی دوم، در پس‌زمینه به این صورت عمل می‌کند:

```
rdbuf()->str();
```

برای به‌کارگیری این کلاس، ورودی مورد نظرمان را در قالب یک رشته به روش‌های گفته شده، به جریان معرفی می‌کنیم. سپس می‌توانیم با به‌کارگیری عملگرهای استخراجی >> و << رشته را به دید یک ورودی/خروجی بنگریم و آن را به سادگی پردازش کنیم.

برای نمونه فرض کنید، رشته‌ای شامل چند عدد بوده و قصد جدا کردن اعداد و انجام عملیاتی بر روی هر یک را دارید؛ چه روشی برای این کار پیشنهاد می‌کنید؟ تک‌تک نویسه‌های رشته را بررسی کرده و اعداد را رقم به رقم جدا کنید؟ با به‌کارگیری این کلاس، به راحتی قادر به این کار خواهید بود:

[۲۷] Input/Output String Stream

```
#include <iostream>
#include <sstream>
using namespace std;
int main() {
    stringstream ssnums("123+45*9/2678");
    int num = 0;
    char c;
    while (ssnums.good()) {
        c = ssnums.get();
        if (c >= '0' && c <= '9') {
            ssnums.unget();
            ssnums >> num;
            cout << "number: " << num << endl;
        }
        else
            cout << "operator: " << c << endl;
    }
    return 0;
}
```

تابع good اگر هنوز نویسه‌ای برای خواندن وجود داشته باشد، مقدار true را برمی‌گرداند. تابع get نویسه بعدی را که اشاره‌گر درونی میانگیر به آن اشاره می‌کند، خوانده و برمی‌گرداند. در واقع در نمونه بالا، در آغاز یک نویسه خوانده می‌شود، اگر آن نویسه عدد باشد، پس با یک عدد روبه‌رو هستیم؛ در نتیجه از عملگر استخراجی >> می‌خواهیم که یک عدد صحیح برایمان بخواند؛ در غیر این صورت یک عملگر (در واقع نویسه‌ی غیرعددی) خواهیم داشت. خروجی این کد به این صورت است:

Output

```
number: 123
operator : +
number: 45
operator : *
number: 9
operator : /
number: 2678
```

اگر نیاز به تشخیص عدد یا غیر عدد نداشته باشیم، می‌توانیم به جای به‌کارگیری تابع good خود عملگرهای استخراجی را به کار بریم. مقدار استخراج شده توسط این عملگرها برگردانده نمی‌شود؛ بلکه بطور مستقیم در متغیر به کار رفته، قرار می‌گیرد. مقدار خروجی این تابع، اشاره‌گری به خودش است و اگر موفق به انجام عمل درخواستی نشود، مقدار صفر را برمی‌گرداند. در مثال بعد، نتیجه خود عملگر استخراجی را در شرط حلقه while به کار برده‌ایم:

```
#include <iostream>
#include <sstream>
using namespace std;
int main() {
    stringstream ssnums("123 45 9 2678");
    int num = 0, counter = 1;
    while (ssnums >> num)
        cout << "number " << counter++ << ": " << num << endl;
    return 0;
}
```

خروجی این کد به صورت زیر است:

```
Output
number 1: 123
number 2: 45
number 3: 9
number 4: 2678
```

توضیحات این بخش، درباره‌ی عملگرهای استخراجی مربوط به کلاس‌های پیشین نیز درست است. اگر عملگر استخراجی به هر دلیلی با مشکلی روبه‌رو شود، یکی از پرچم‌های درونی خود را مقداردهی می‌کند. هریک از این پرچم‌ها نشان‌دهنده یک نوع از خطای ایجاد شده هستند؛ جدول ۱۷,۲ این پرچم‌ها را توضیح می‌دهد.

جدول ۱۷,۲. پرچم‌های خطای عملگرهای استخراجی

پرچم	توضیح
eofbit	در هنگام انجام یک استخراج، به پایان میانگیر رسیده است.
failbit	عملگر استخراجی در هنگام استخراج با شکست روبرو شده است؛ یکی از دلیل‌ها می‌تواند عدم هم‌خوانی نوع درخواستی و نوع موجود در ورودی باشد.
badbit	خطایی غیر از دو نوع بالا رخ داده است.
goodbit	بدون خطا؛ نبود هر یک از سه خطای بالا را اعلام می‌کند.

برای تشخیص محتوای هر یک از پرچم‌های بالا، یک تابع در نظر گرفته شده است که خروجی همه‌ی آن‌ها از نوع Boolean است. توابع در نظر گرفته شده در ادامه آمده‌اند:

پرچم	تابع مربوطه
eofbit	eof()
failbit and badbit	fail()
badbit	bad()
goodbit	good()

به جای به‌کارگیری تابع good می‌توان تابع eof را به این شکل به کار برد (خروجی این مثال دقیقا مانند مثال قبلی است):

```
#include <iostream>
#include <sstream>
using namespace std;
int main() {
    stringstream ssnums("123 45 9 2678");
    int num = 0, counter = 1;
    while (!ssnums.eof()) {
        ssnums >> num;
        cout << "number " << counter++ << ": " << num << endl;
    }
    return 0;
}
```

اگر نوع داده‌ی نادرستی را درخواست کنید، مقدار پرچم fail به true تغییر پیدا خواهد کرد. به مثال بعدی توجه کنید:

```
#include <iostream>
#include <sstream>
using namespace std;
int main() {
    stringstream ssnums("abc");
    int num = 0;
    ssnums >> num;
    if (ssnums.fail())
        cout << "bad variable type requested!" << endl;
    return 0;
}
```

به جای به‌کارگیری تابع fail می‌توان عملگر ! و نام جریان را به کار برد؛ این دو مانند یکدیگرند. بنابراین قسمت if نمونه کد پیشین می‌تواند به این صورت تغییر پیدا کند:

```
if (!ssnums)
    cout << "bad variable type requested!" << endl;
```

تمرین‌ها

۱- برنامه‌ای بنویسید که همه‌ی نویسه‌های وارد شده توسط کاربر را به همان شکلی که وارد شده‌اند (شامل فاصله و خط جدید)، ذخیره کند و پس از وارد کردن کلمه "PRINTALL"، همه‌ی متن‌های ذخیره شده تا این لحظه را در خروجی چاپ کند.

Input	Output
This is a sample text Previous line left empty	This is a sample text Previous line left empty

۲- در یک برنامه متغیر INT به کار برده شده است ولی برنامه‌نویس آن، متوجه شده که این نوع برای هدف مورد نظرش کافی نیست؛ او قصد دارد برنامه‌ای ساده بنویسد که همه‌ی متغیرهای INT به کار برده شده در برنامه را به LONG تغییر دهد. در این کار به او کمک کنید. در خروجی برنامه، دقیقا کد دریافت شده در ورودی را چاپ کنید؛ با این تفاوت که همه‌ی رشته‌های INT در کد خروجی با LONG جایگزین شده باشند. دقت کنید که رشته‌ی INT نباید جزئی از رشته‌ی دیگر باشد؛ برای نمونه زیررشته‌ی INT در رشته‌ی INTERANET نباید به LONGERANET تبدیل شود. پایان کد با رشته‌ی END مشخص می‌شود (روشن است که در بدنه‌ی کد، رشته‌ی END ظاهر نخواهد شد).

Input	Output

INT a=0, b=0;
// this line is a comment for INT vars
for (INT i=0; i<10; i++) {
  a = a + 1;
  b *= 2 + a;
}
cout << "integers: " << endl;
cout << a << " " << b << endl;
	LONG a=0, b=0;
	// this line is a comment for LONG vars
	for (LONG i=0; i<10; i++) {
	  a = a + 1;
	  b *= 2 + a;
	}
	cout << "integers: " << endl;
	cout << a << " " << b << endl;

۳- برنامه‌ای بنویسید که یک ماشین‌حساب را به این صورت پیاده‌سازی کند:

- ورودی این ماشین حساب یک رشته‌ی محاسباتی است که به صورت پیوسته و در یک خط وارد می‌شود.
- این رشته شامل اعداد صحیح و چند عملگر است که هر یک با فاصله از هم جدا شده‌اند.
- عملگرهای پشتیبانی شونده توسط این ماشین‌حساب، جمع، تفریق و ضرب هستند.
- تقدم همه‌ی عملگرها برابر است و ترتیب به کار برده شدن آن‌ها از راست به چپ است.
- خروجی برنامه، نتیجه محاسبه رشته ورودی به همراه توضیح نحوه محاسبه آن است.

Input	Output
12 + 11 * 2 – 4	-10 2-4→ -2 11-2 → -22 12+(-22) → -10

۴- برنامه‌ای بنویسید که نتیجه‌ی بازی‌های فوتبال را دریافت کرده و جدول لیگ را برای آن‌ها تشکیل دهد.

- در خط اول ورودی، n تعداد بازی‌های لیگ آورده می‌شود.
- در n خط بعد، در هر خط یک بازی آورده می‌شود. با ساختار «تیم اول، تعداد گل زده، تیم دوم، تعداد گل زده». تعداد بازی‌های انجام شده توسط تیم‌ها، لزوما برابر نیست.
- هر برد ۳ امتیاز، مساوی ۱ امتیاز و باخت امتیازی ندارد.
- جدول خروجی دارای ستون‌های «نام تیم، امتیاز، گل زده، گل خورده، تفاضل (گل زده – گل خورده)، رتبه، تعداد بازی» است و تیم‌ها به ترتیب رتبه فهرست می‌شوند.

- دقت کنید که ستون‌های خروجی باید چپ‌چین باشند و همه‌ی اعداد و رشته‌های مورد نمایش در آن ستون، نباید از محدوده‌ی ستون خارج شوند.
- در صورت برابر بودن امتیاز دو تیم، تعداد بازی کمتر و تفاضل گل بیشتر موجب برتری می‌شود.

Input	Output
5 A 5 B 1 A 4 C 3 B 1 C 2 B 2 D 1 D 1 C 3	NAME SCORE SS RR MM RANK PC A 6 9 4 5 1 2 C 6 8 6 2 2 3 B 3 4 8 -4 3 3 D 0 2 5 -3 4 2

۵- برنامه‌ای بنویسید که مسیر یک پرونده را دریافت کرده و اطلاعات زیر را در خروجی چاپ کند:

- نام پرونده
- تعداد خط‌های پرونده
- تعداد نویسه‌های موجود در پرونده
- حجم پرونده بر حسب بایت

۶- برنامه‌ای بنویسید که تعدادی عدد را از ورودی خوانده و جذر آن‌ها را از عدد آخر به اول در خروجی چاپ کند. اعداد همگی مثبت و کوچکتر از 10^{18} هستند. تعداد اعداد مشخص نیست ولی مقداری کمتر از 10^5 می‌باشد. اعداد خروجی باید با چهار رقم ثابت اعشار چاپ شوند.

Input	Output
5 1427 0 876652098643267843 5276538	2297.0716 936297014.1164 0.0000 37.7757

۷- برنامه‌ای بنویسید که عدد صحیح n بین ۱ تا ۹ را از ورودی دریافت کرده و مثلث زیر را به ازای آن چاپ کند. برای چاپ فاصله از عملگر تعیین فضای متغیر (d*%) استفاده کنید.

Input	Output
5	1 1 1 2 2 4 4 8 8

۸- برنامه‌ای بنویسید که عدد صحیح n بین ۴ تا ۹۰ را از ورودی دریافت کرده و یک مستطیل به صورت نشان داده شده را به ازای آن چاپ کند. واضح است که برای n_های زوج، مستطیل به مربع تبدیل خواهد شد. برای پیاده‌سازی این برنامه می‌توانید از آرایه‌ها بهره ببرید. دقت کنید که برای نمایش درست مستطیل در خروجی، باید عملگر تعیین فضا را به درستی به کار بگیرید.

Input	Output
20	1 2 3 4 5 6 20 7 19 8 18 9 17 10 16 15 14 13 12 11

مراجع

Stroustrup, B., 2014. *Programming: principles and practice using C++* 2nd ed., Pearson Education.

فصل ۳- کتابخانه الگوی استاندارد

در این فصل مجموعه‌ای از کتابخانه‌های کاربردی سی++ را که همگی از اعضای کتابخانه الگوی استاندارد هستند، معرفی خواهیم کرد. این کلاس‌ها اغلب از ساختمان‌های داده‌ای معمول بوده و باقی آن‌ها از کتابخانه‌های بسیار کاربردی برای پیاده‌سازی برنامه‌های مختلف هستند.

۳-۱- لیست

لیست در واقع در دسته‌ی ساختمان‌های داده قرار نمی‌گیرد، ولی از آنجا که از کلاس‌های کتابخانه استاندارد سی++ است، در اینجا به توضیح آن می‌پردازیم. این ساختمان‌داده همان‌طور که از نامش برمی‌آید، برای مدیریت فهرستی از عنصرها به کار می‌رود. حال این عنصرها می‌توانند از هر نوعی مانند عدد صحیح، عدد اعشاری، رشته و ... باشند. این کلاس دارای متدهایی برای حذف و افزودن عنصر، دسترسی به عنصرهای موجود در آن، مرتب کردن عنصرها، تعیین و تغییر ظرفیت لیست و ... فراهم آورده که به‌کارگیری آن را بسیار کاربردی کرده است.

۳-۱-۱- معرفی ابزار معادل در کتابخانه استاندارد سی++

از آنجا که این کلاس، نخستین موردی است که آن را بررسی می‌کنیم، کمی بیشتر وارد جزییات خواهید شد؛ ولی برای کلاس‌های بعدی کوتاه‌تر عمل خواهیم کرد؛ زیرا توابع و خصیصه‌های بیشتر کتابخانه‌های استاندارد مشابه یکدیگر هستند.

این کلاس در سرآیند list قرار دارد و برای به‌کارگیری آن باید این سرآیند را include کنیم. چند سازنده برای این کلاس درنظر گرفته شده که در ادامه توضیح داده شده‌اند. به این مثال توجه کنید:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> first;
    list<int> second(4, 100);
    list<int> third(second.begin(), second.end());
    list<int> fourth(third);
    int myints[] = {16, 2, 77, 29};
```

```
    list<int> fifth(myints, myints + sizeof(myints) / sizeof(int));
    cout << "The contents of fifth list are: ";
    for (list<int>::iterator it = fifth.begin(); it != fifth.end();it++)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

کد بالا لیستی از نوع int (عنصرهای این لیست اعداد صحیح خواهند بود) و بدون هیچ عنصری می‌سازد. برای این کار از مفهومی به نام الگو استفاده شده است. به کمک الگوها می‌توان خصیصه‌های درون کلاس‌ها را به گونه‌ای تعریف کرد که نوع آنها در هنگام فراخوانی سازنده کلاس به صورت یک پارامتر تعیین شود. به این ترتیب می‌توان از یک کلاس در موقعیت‌های مختلف مجددا و بدون نیاز به برنامه‌نویسی اضافی استفاده نمود. کلاس لیست نیز از این قابلیت استفاده می‌کند. به عنوان مثال، دستور اول در کد قبلی لیستی از اعداد صحیح به نام first تولید می‌کند.

```
list<int> first;
```

دستور دوم لیست دیگری از اعداد صحیح به نام second ایجاد می‌کند؛ با این تفاوت که از همان ابتدا در این آرایه چهار عدد صحیح با مقدار ۱۰۰ قرار می‌دهد.

```
list<int> second(4, 100);
```

اگر در دستور قبلی پارامتر دوم نوشته نشود، مقدار ۰ در لیست قرار خواهد گرفت (چون محتوای لیست از نوع عدد صحیح است).

دستور سوم در واقع محتویات لیست پیشین (second) را در این لیست کپی می‌کند. با به‌کارگیری این روش می‌توان بخشی از یک لیست را نیز کپی کرد.

```
list<int> third(second.begin(), second.end());
```

در دستور چهارم محتویات لیست third را در لیست fourth قرار داده می‌شود.

```
list<int> fourth(third);
```

در کد قبلی آرایه fifth به شکلی متفاوت تولید شده است. به این صورت که ابتدا یک آرایه ایستا به نام myints که حاوی چهار عدد صحیح است تشکیل شده و در مرحله بعد محتویات این آرایه درون لیست جدیدی به نام fifth قرار داده می‌شود.

```
int myints[] = {16, 2, 77, 29};
list<int> fifth(myints, myints + sizeof(myints) / sizeof(int));
```

در این کد sizeof(myints)/sizeof(int) طول آرایه myints را محاسبه می‌کند. در این نمونه شما می‌توانید به جای این عبارت عدد چهار هم قرار دهید.

با به‌کارگیری این روش می‌توانید محتویات یک آرایه‌ی هم نوع با لیست را در آن کپی کنید. کپی از هر بخشی از آرایه امکان‌پذیر است. در این نمونه از آغاز تا پایان آرایه در لیست کپی خواهد شد.

تا اینجا با سازنده‌های کلاس list آشنا شدید. اغلب کلاس‌های استاندارد دارای سازنده‌های بیشمار و کاربردی هستند. برای پیمایش این کلاس و کلاس‌های مشابه می‌توانید از پیمایشگر استفاده کنید. هر کلاسی پیمایشگر مختص به خود را دارد. برای نمونه پیمایشگر کلاس list به این صورت به کار می‌رود:

```
for (list<int>::iterator it = fifth.begin(); it != fifth.end();it++)
    cout << *it << " ";
```

عبارت list<int>::iterator به معنی پیمایشگر مربوط به کلاس list است. در کد قبلی یک شیء پیمایشگر ایجاد شده که مقدار اولیه آن برابر با ابتدای لیست قرار داده شده است. حلقه‌ی for هر بار یک عنصر از لیست را چاپ کرده و پیمایشگر را در لیست حرکت می‌دهد (it++). حلقه تا وقتی که پیمایشگر به پایان لیست نرسیده، ادامه پیدا خواهد کرد (it!=fifth.end()).

دو تابع برای قرار دادن عنصرهای جدید در لیست تعبیه شده است. تابع push_back برای قرار دادن عنصر در پایان لیست و تابع push_front برای قرار دادن عنصر در آغاز لیست به کار می‌روند. نحوه استفاده از این دو تابع در این کد نشان داده شده است:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> mylist(1, 50);
    mylist.push_back(40);
    mylist.push_front(60);
    for(list<int>::iterator it= mylist.begin(); it!= mylist.end(); it++)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

خروجی کد قبل به این صورت خواهد بود:

Output
60 50 40

برای خارج کردن عنصرها از لیست نیز دو تابع درنظر گرفته شده که همانند دو تابع قبلی، یکی از آغاز لیست (pop_front) و دیگری از پایان لیست (pop_back) عناصر را حذف می‌کنند. نحوه استفاده از این دو تابع در این کد نشان داده شده است (یافتن خروجی این کد به عهده خواننده واگذار می‌شود):

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> mylist(1, 50);
    mylist.push_back(40);
    mylist.push_front(60);
    mylist.pop_front();
    mylist.pop_back();
    for(list<int>::iterator it= mylist.begin(); it!= mylist.end(); it++)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

برای حذف عنصرها دو تابع remove و remove_if نیز درنظر گرفته شده‌اند. تابع remove مقدار یک عنصر را به عنوان ورودی دریافته کرده و آن را از لیست حذف می‌کند. به این مثال توجه کنید:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> mylist(1, 50);
    mylist.push_back(40);
    mylist.push_front(60);
    mylist.remove(40);
    for(list<int>::iterator it= mylist.begin(); it!= mylist.end(); it++)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

تابع remove_if کاربرد جالبی دارد. این تابع عنصرهایی که دارای شرطی ویژه هستند را حذف می‌کند. این شرط می‌تواند توسط یک تابع یا یک کلاس که عملگر () را پیاده‌سازی کرده است، تعیین شود. ورودی تابع یا عملگر مربوطه ارجاعی ثابت (const) از نوع عنصرهای موجود در لیست است. برای نمونه این لیست را در نظر بگیرید:

```
int myints[] = {15, 36, 7, 17, 20, 39, 4, 1};
list<int> mylist(myints, myints + 8);
```

فرض کنید می خواهیم اعدادی که مقداری کمتر از ۱۰ دارند را از لیست حذف کنیم. برای این منظور تابعی به این صورت تعریف می‌کنیم. تابع remove_if برای هر عنصر موجود در لیست، این تابع را فراخوانی کرده و عنصر مربوطه را در پارامتر ورودی value قرار می‌دهد؛ اگر تابع مقدار true را برگرداند، عنصر مربوطه از لیست حذف خواهد شد.

```
bool single_digit(const int& value) {
    return (value < 10);
}
```

حال با این دستور می توان عناصری که مقداری کمتر از ۱۰ دارند را از لیست حذف کرد:

```
mylist.remove_if(single_digit);
```

به جای به‌کارگیری تابع، می‌توانید یک کلاس به اضافه پیاده‌سازی عملگر () را به کار ببرید. به عنوان مثال، با استفاده از این کلاس می توان اعداد فرد را از لیست حذف کرد:

```
class is_odd {
public:
    bool operator() (const int& value) {
        return (value % 2) == 1;
    }
};
```

در این حالت با استفاده از این دستور مقادیر فرد از لیست حذف می‌شود:

```
mylist.remove_if(is_odd());
```

تابع empty خالی بودن یا نبودن لیست را بررسی می‌کند. نمونه کد بعدی مجموع اعداد موجود در لیست را به دست می‌آورد:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> mylist;
    int sum(0);
    for (int i = 1; i <= 10; i++)
        mylist.push_back(i);
    while (!mylist.empty()) {
        sum += mylist.front();
        mylist.pop_front();
```

```
    }
    cout << "total: " << sum << endl;
    return 0;
}
```

تابع clear همه عناصر موجود در لیست را حذف می‌کند.

```
mylist.clear();
```

تابع erase عنصری را از موقعیتی مشخص حذف می‌کند. همچنین قابلیت حذف عنصرها از یک مکان تا مکانی دیگر از لیست را نیز دارد. به این کد توجه کنید:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> mylist;
    list<int>::iterator it1, it2;
    for (int i = 1; i < 10; i++)
        mylist.push_back(i * 10);
    it1 = it2 = mylist.begin();
    advance(it2, 6);
    ++it1;
    it1 = mylist.erase(it1);
    it2 = mylist.erase(it2);
    ++it1;
    --it2;
    mylist.erase(it1, it2);
    return 0;
}
```

تابع advance در کد بالا پیمایشگر it2 را ۶ خانه به جلو می‌برد.
کد قبلی عنصری که در موقعیت ۲ (عدد ۲۰) است (it1 به عنصر دوم اشاره می‌کند) را حذف می‌کند. به این ترتیب عدد ۳۰ در موقعیت دوم قرار می‌گیرد و پیمایشگر it1 به این عدد اشاره خواهد کرد.
پیش از اجرای کد قبلی پیمایشگر it1 مقدار ۳ و it2 مقدار ۵ را دارند. دستور erase در نمونه بالا، عنصرهایی که در مکان ۳، ۴ و ۵ هستند را از لیست حذف می‌کند (عنصرهای موجود در مکان‌های ۳ تا ۵).
تابع resize برای تغییر اندازه لیست به کار می‌رود. به این کد توجه کنید:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> mylist;
```

```
    for (int i = 1; i < 10; i++)
        mylist.push_back(i);      // 1 2 3 4 5 6 7 8 9
    mylist.resize(5);             // 1 2 3 4 5
    mylist.resize(8, 100);        // 1 2 3 4 5 100 100 100
    mylist.resize(12);            // 1 2 3 4 5 100 100 100 0 0 0 0
    return 0;
}
```

کد قبلی در آغاز اعداد ۱ تا ۹ را در لیست قرار می‌دهد (اندازه لیست برابر ۹ می‌شود). سپس اندازه آن را بـه ۵ تغییـر می‌دهد (۴ عنصر آخر لیست به صورت خودکار حذف خواهند شد). در ادامه اندازه لیست به ۸ تغییر داده شده و مقدار پیش‌فرض خانه‌هایی که مقداردهی نشده‌اند، ۱۰۰ قرار داده می‌شود. در پایان اندازه لیست به ۱۲ تغییر پیدا کرده و ۴ عنصر جدید که مقداردهی نشده‌اند، مقدار ۰ را خواهند داشت.

تابع reverse ترتیب عناصر موجود در لیست را عکس می‌کند:

```
#include <list>
using namespace std;
int main() {
    list<int> mylist;
    for (int i = 1; i < 10; i++)
        mylist.push_back(i);      // 1 2 3 4 5 6 7 8 9
    mylist.reverse();             // 9 8 7 6 5 4 3 2 1
    return 0;
}
```

تابع swap عنصرهای دو لیست را با هم عوض می‌کند:

```
#include <list>
using namespace std;
int main() {
    list<int> first(3, 100);    // 100 100 100
    list<int> second(5, 200);   // 200 200 200 200 200
    list<int>::iterator it;
    first.swap(second);         // first:  200 200 200 200 200
                                // second: 100 100 100
    return 0;
}
```

تابع sort عنصرهای موجود در لیست را به صورت صعودی مرتب می‌کند:

```
#include <list>
using namespace std;
int main() {
    list<string> mylist;
    mylist.sort();
```

```
    return 0;
}
```

همچنین می‌توانید تابعی سفارشی برای مرتب‌کردن لیست نوشته و بطور دلخواه عنصرهای موجود در لیست را مرتب کنید. برای نمونه می‌توان لیستی از نوع رشته ایجاد کرده و عنصرهای موجود در آن را به صورت نزولی مرتب نمـود. نوع خروجی تابع مربوطه باید از نوع bool بوده و دارای ۲ پارامتر ورودی از نوع عنصرهای موجود در لیست باشد.

```
#include <iostream>
#include <list>
#include <string>
using namespace std;
bool compare_items(string first, string second) {
    if (first.compare(second) < 0)
        return false;
    else
        return true;
}
int main() {
    list<string> mylist;
    mylist.push_back("A");
    mylist.push_back("D");
    mylist.push_back("B");
    mylist.push_back("C");
    mylist.sort(compare_items);
    return 0;
}
```

توجه کنید که برای به‌کارگیری کلاس string، باید سرآیند string را include کنید.

تابع unique عنصرهای تکراری موجود در لیست را حذف می‌کند:

```
mylist.unique();
```

این تابع همچنین می‌تواند به صورت سفارشی عنصرهای تکراری را حذف کند. در این حالت، باید تابع یا کلاسی بـا پیاده‌سازی عملگر ()، پیاده‌سازی شود. برای نمونه در مثال بعدی لیستی از اعداد اعشاری ساخته و اعدادی که فاصله نزدیکی با هم دارند را از لیست حذف می‌کنیم:

```
#include <iostream>
#include <list>
using namespace std;
class is_near {
public:
    bool operator() (double first, double second) {
        return (fabs(first - second)<5.0);
```

```
    }
};
int main() {
    double mydoubles[] = {12.15, 2.72, 73.0, 12.77, 3.14,
        12.77, 73.35, 72.25, 15.3, 72.25};
    list<double> mylist(mydoubles, mydoubles + 10);
    mylist.sort();            //2.72,  3.14,  12.15, 12.77, 12.77,
                              //15.3,  72.25, 72.25, 73.0,  73.35
    mylist.unique(is_near());//2.72, 12.15, 72.25
    return 0;
}
```

تابع fabs قدرمطلق اعداد اعشاری را محاسبه می‌کند. اگر اختلاف دو عدد موجود در لیست کمتر از ۰٫۵ باشد، عدد دوم از لیست حذف خواهد شد.

تابع‌های begin و back به ترتیب ارجاعی از عنصر اول و آخر لیست برمی‌گردانند.

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> mylist;
    mylist.push_back(10);
    while (mylist.back() != 0)
        mylist.push_back(mylist.back() - 1);
    //10 9 8 7 6 5 4 3 2 1 0
    return 0;
}
```

مورد پایانی که برای این کلاس توضیح می‌دهیم، عملگر = است. توسط این عملگر می‌توانید محتویات یک لیست را در دیگری قرار دهید.

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> first;
    first.push_back(1000);
    first.push_back(100);
    first.push_back(10);
    list<int> second;
    second = first;          //  second = {1000, 100, 10}
    first = list<int>();     //  first = {}
    return 0;
}
```

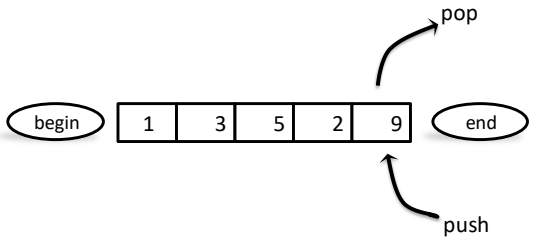
خط آخر در کد قبلی، برای لیست first از نو حافظه تخصیص می‌دهد (با اندازه صفر). بنابراین پس از اجرای این کد لیست second حاوی سه عدد بوده و لیست first خالی خواهد بود.

کلاس list حاوی چند تابع دیگر نیز هست که به علت اهمیت پایین در این فصل به آنها پرداخته نمی‌شود.

از آنجا که توابع به کار رفته در این کلاس، در بسیاری از کلاس‌های استاندارد دیگر نیز وجود داشته و کاربرد یکسانی دارند؛ در این بخش بطور مفصل به توضیح این توابع پرداختیم. در بخش‌های بعد، از پرداختن به جزییات می‌پرهیزیم.

۳-۲- پشته

پشته، ساختمان‌داده‌ای بسیار پرکاربرد و در عین حال ساده است که در پیاده‌سازی بسیاری از برنامه‌ها، نقشی اساسی دارد. عنصرهای جدید به پایان پشته اضافه شده و از پایان آن نیز حذف می‌شوند (LIFO: Last In First Out). شکل ۱,۳ چگونگی عملکرد پشته را نمایش می‌دهد.



شکل ۱,۳. نحوه عملکرد پشته

کلاس پشته به نام stack در کتابخانه الگوی استاندارد پیاده‌سازی شده است. برای به‌کارگیری این کلاس به سرآیند stack نیاز خواهیم داشت. کلاس stack مانند سایر کلاس‌های استاندارد، با استفاده از الگو پیاده‌سازی شده است؛ بنابراین باید هنگام ساختن شیء از آن، نوع عنصرهای موجود در آن را مشخص کرد. به این کد توجه کنید:

```
#include <stack>
using namespace std;
int main() {
    stack<int> st;
    return 0;
}
```

سازنده این کلاس می‌تواند با به‌کارگیری کلاس‌های dequeue و vector (صف دوطرفه و بردار) نیز مقداردهی شود. این کلاس دارای ۵ تابع به نام‌های empty، push، pop، top و size است که در ادامه شرح داده می‌شوند.

تابع empty خالی بودن پشته را بررسی کرده و در صورت خالی بودن، مقدار true را برمی‌گرداند. تابع push مقداری از جنس پشته (در این مثال int) را در انتهای آن قرار داده و تابع pop مقدار روی پشته را حذف می‌کند.

برای دسترسی به عنصر روی پشته، تابع top به کار برده می‌شود. تابع size نیز اندازه پشته را برمی‌گرداند. به این مثال توجه کنید:

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> st;
    st.push(100);
    st.push(200);
    st.push(300);
    cout << st.size() << endl;
    while (!st.empty()) {
        cout << st.top() << endl;
        st.pop();
    }
    return 0;
}
```

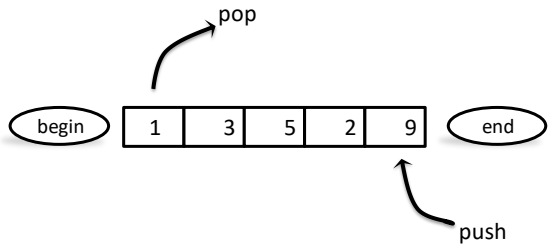
خروجی این کد به این صورت خواهد بود:

```
Output
3
300
200
100
```

۳-۳- صف

ساختمان‌داده‌ی صف دقیقا از صف در دنیای واقعی الگوبرداری شده است. این ساختمان‌داده مانند پشته از ساختمان داده‌های ساده و پایه‌ای در علم رایانه است.

عنصرهای جدید به پایان صف اضافه شده و از آغاز آن حذف می‌شوند (FIFO: First In First Out). شکل ۲,۳ عملکرد صف را به شکلی ساده نمایش می‌دهد.



شکل ۲,۳. نحوه عملکرد صف

ساختمان داده صف در کتابخانه الگوی استاندارد توسط کلاس queue که کاملا مشابه با کلاس stack است، پیاده‌سازی شده است. برای به‌کارگیری این کلاس باید سرآیند queue را include کنید.

کلاس queue دارای توابع empty، size، front، back، push و pop است که در ادامه شرح داده می‌شود.

تابع‌های empty و size کاملا مشابه توابع هم‌نام خود در کلاس stack عمل می‌کنند. تابع push عنصری را در پایان صف قرار داده و تابع pop عنصری را از آغاز صف حذف می‌کند. دو تابع front و back برای دسترسی به عنصر آغاز و پایان صف تعبیه شده‌اند. به مثال بعدی توجه کنید:

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    queue<int> que;
    que.push(100);
    que.push(200);
    que.push(300);
    cout << que.size() << endl;
    while (!que.empty()) {
        cout << que.front() << endl;
        que.pop();
    }
    return 0;
}
```

خروجی این کد به این صورت خواهد بود:

```
Output
3
100
200
300
```

۳-٤- مجموعه

مجموعه‌ها، نوعی از آرایه‌های انجمنی هستند که عنصرهای یکتا را درون خود نگه می‌دارند. به عبارت دیگر عنصرهای درون این نوع از آرایه‌ها، خود نقش شناسه را بازی می‌کنند.

ویژگی اصلی آرایه‌های انجمنی این است که به کمک شناسه‌ی هر داده، می‌توان به آن دسترسی داشت؛ برخلاف آرایه‌های معمولی که برای دسترسی به عنصرهای درون آن‌ها، باید از نمایه استفاده کرد.

اعضای درون مجموعه به صورت پیش‌فرض به ترتیب صعودی مرتب می‌شوند. مجموعه‌ها در واقع به کمک درخت های جستجوی دودویی که در همین فصل توضیح داده خواهند شد، پیاده‌سازی شده‌اند. بنابراین ویژگی‌های اصلی مجموعه‌ها عبارتند از:

۱- مقادیر عنصرهای موجود در آن یکتا هستند و هیچ دو عنصری در مجموعه نمی‌توانند با هم برابر باشند.
آرایه‌ی انجمنی مشابهی به نام مجموعه چندگانه وجود دارد که می توان درون آن اطلاعات تکراری نیز قرار داد.

۲- مقدار عنصرها، خود نقش شناسه را بازی می‌کنند. آرایه‌ی انجمنی مشابهی به نام نگاشت نیز وجود دارد که در آن، مقدار عنصرها از شناسه‌ها جدا هستند (در بخش بعد به معرفی این آرایه انجمنی خواهیم پرداخت).

ساختمان داده‌ای مجموعه توسط کلاس set پیاده‌سازی شده است که در سرآیند set قرار دارد. این نوع از آرایه‌های انجمنی، پیمایشگرهای دوسویه را پشتیبانی می‌کند. به شکلی که شما می‌توانید با به‌کارگیری دو نوع پیمایشگر متفاوت، مجموعه را از آغاز و پایان پیمایش کرده و به عنصرهای موجود در آن دسترسی داشته باشید. در ادامه به توضیح بعضی از توابع کاربردی مجموعه‌ها می‌پردازیم.

سازنده‌ی این کلاس می‌تواند به چند شکل متفاوت به کار برده شود که هر یک کاربرد ویژه‌ی خود را دارند. هر یک از این شکل‌ها را کمی توضیح خواهیم داد. برای ایجاد یک مجموعه به این صورت می توان عمل کرد:

```
#include <set>
using namespace std;
int main() {
    set<int> first;
    return 0;
}
```

کد قبلی مجموعه‌ای خالی ایجاد خواهد کرد. در این نمونه عنصرهای مجموعه از نوع عدد صحیح تعیین شده‌اند. سازنده این کلاس می‌تواند مجموعه‌ای دیگر را به عنوان پارامتر ورودی بگیرد. در این حالت یک کپی از عنصرهای مجموعه مورد نظر در مجموعه جدید قرار داده می‌شوند. به کد بعدی توجه کنید:

```
#include <set>
using namespace std;
int main() {
    int myints[] = {10, 20, 30, 40, 50};
    set<int> second(myints, myints + 5);
    set<int> third(second);
    return 0;
}
```

با اجرای کد قبلی، مجموعه third دقیقا یک کپی از مجموعه second خواهد بود. مشاهده می‌کنید که عنصرهای یک آرایه از نوع int در مجموعه second قرار داده شده‌اند. توجه کنید که نوع آرایه مورد نظر باید با نوع عنصرهای مجموعه یکسان باشد. در واقع سازنده می‌تواند دو پیمایشگر بپذیرد و عنصرهای میان دو پیمایشگر را در مجموعه قرار دهد. حال این دو پیمایشگر می‌توانند نام آرایه‌ای هم جنس با عنصرهای مجموعه بوده (کد قبلی) و یا اینکه از جنس پیماشگر مجموعه باشند (به مثال بعدی توجه کنید).

```
#include <set>
using namespace std;
int main() {
    int myints[] = {10, 20, 30, 40, 50};
    set<int> second(myints, myints + 5);      // 10 20 30 40 50
    set<int>::iterator it1 = second.begin();
    set<int>::iterator it2 = second.end();
    advance(it1, 1);
    advance(it2, -1);
    set<int> fourth(it1, it2);                // 20 30 40
    return 0;
}
```

این کد عنصرهای ۲۰، ۳۰ و ۴۰ را در مجموعه third قرار خواهد داد. تابع advance پیمایشگر it1 را یک خانه به جلو (اشاره به عنصر ۲۰) و پیمایشگر it2 را یک خانه به عقب (اشاره به عنصر ۴۰) حرکت می‌دهد.

در مثال‌هایی که تا به حال زده شد، نوع عنصرهای مجموعه از انواع ساده‌ای مانند عدد صحیح بود. حال فرض کنید قصد دارید یک نوع جدید (یک ساختار) ایجاد کرده و مجموعه‌ای از آن بسازید. این امر، بدون مقدمه‌ای از کلاس set امکان‌پذیر نیست. زیرا این کلاس عنصرها را مرتب نگه می‌دارد ولی بر چه اساسی باید نوع جدید شما را مرتب نگه دارد؟ بنابراین امکان ایجاد کردن مجموعه‌ای از جنس یک ساختار با روش‌هایی که گفته شد، وجود ندارد. برای این منظور باید روشی برای مرتب کردن ساختار جدید به مجموعه معرفی کنید. این معرفی می‌تواند به دو شکل صورت پذیرد:

۱- تعریف یک کلاس جدید که عملگر () را پیاده‌سازی کرده باشد.

۲- تعریف یک تابع مقایسه‌کننده و معرفی آن به مجموعه توسط یک اشاره‌گر به تابع.

هر دو روش به شکلی مشابه، روش مقایسه عنصرهای موجود در مجموعه را تعریف می‌کنند؛ استفاده از هر دو روش در ادامه توضیح داده شده است.

برای ایجاد یک کلاس و پیاده‌سازی عملگر () به این صورت عمل می‌کنیم: پارامترهای ورودی عملگر () (روش اول) و تابع مقایسه‌کننده (روش دوم) و نوع خروجی آن‌ها مشترک است.

```
struct classcomp {
    bool operator() (const TYPE& lhs, const TYPE& rhs) const {
        return true OR false;
    }
};
```

در ساختار بالا به جای کلمه‌های TYPE نوع عناصر مجموعه را به کار می‌بریم. برای روشن‌تر شدن موضوع، فرض کنید ساختاری دو بخشی به این صورت داریم و می‌خواهیم مجموعه‌ای از آن بسازیم:

```
struct stsample {
    int num;
    string str;
};
```

برای ایجاد مجموعه‌ای از این ساختار باید ابتدا کلاس مقایسه‌کننده را تعریف کنیم. این تعریف می‌تواند به این صورت انجام شود:

```
struct classcomp {
    bool operator() (const stsample& lhs, const stsample& rhs) const {
        return lhs.num < rhs.num;
    }
};
```

توجه کنید که نوع پارامترهای ورودی عملگر () از نوع ساختار ایجاد شده است. پارامترهای ورودی از نوع ارجاعی و ثابت هستند. می‌خواهیم ساختارمان را بر اساس بخش عددی آن مرتب کنیم؛ بنابراین فقط بخش‌های عددی را با هم مقایسه کردیم. در حالت کنونی عنصرها به صورت صعودی مرتب خواهند شد. برای نزولی مرتب کردن عنصرها، عملگر > را به < تغییر دهید. حال می‌توانیم مجموعه مورد نظرمان را بسازیم:

```
set<stsample, classcomp> stcontent;
```

تنها تفاوت این نمونه با نمونه‌های پیشین، در پارامتر دوم الگوی مربوط به مجموعه است. این پارامتر نام کلاس مقایسه‌کننده است.

حال به کد بعدی توجه کنید. در این کد آرایه‌ای به طول سه، از ساختارمان ایجاد کرده و آن را مقداردهی می‌کنیم. سپس عنصرهای آن را در مجموعه قرار می‌دهیم.

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
struct stsample {
    int num;
    string str;
};
struct classcomp {
    bool operator() (const stsample& lhs, const stsample& rhs) const {
        return lhs.num < rhs.num;
    }
};
int main() {
    stsample mystruct[3];
    mystruct[0].num = 10;
    mystruct[0].str = "str2";
    mystruct[1].num = 5;
    mystruct[1].str = "str1";
    mystruct[2].num = 15;
    mystruct[2].str = "str3";
    set<stsample, classcomp> stcontent;
```

```
    for (int i = 0; i < 3; i++)
        stcontent.insert(mystruct[i]);
    for (set<stsample, classcomp>::iterator it = stcontent.begin();
        it != stcontent.end(); it++)
        cout << (*it).num << ": " << (*it).str << endl;
    return 0;
}
```

تابع insert عنصری را در مجموعه قرار می‌دهد. پس از اجرا شدن این برنامه، عنصرها بر اساس مقدار خصیصه num مرتب خواهند شد. شما به عنوان تمرین می‌توانید عنصرها را بر اساس خصیصه str مرتب کنید. خروجی کد قبلی به این صورت خواهد بود:

```
Output
5: str1
10: str2
15: str3
```

به‌کارگیری تابع مقایسه‌کننده کمی دشوارتر از روش پیشین است. ساختار تابع مقایسه‌کننده کاملا مشابه با روش پیشین است. پارامترهای ورودی می‌توانند از نوع ارجاعی و ثابت نباشند. به این کد توجه کنید:

```
bool fncomp(const stsample& lhs, const stsample& rhs) {
    return lhs.num < rhs.num;
}
```

می‌بینید که تفاوتی در نوع مقایسه و پارامترها دیده نمی‌شود. تنها تفاوت در چگونگی به‌کارگیری و معرفی آن به مجموعه است که در ادامه نشان داده شده است:

```
bool(*fn)(const stsample&, const stsample&) = fncomp;
set<stsample, bool(*)(const stsample&, const stsample&)> stcontent(fn);
```

ابتدا اشاره‌گری به تابع مقایسه‌کننده می‌سازیم. برای تعریف اشاره‌گر به تابع از دستوری به این صورت استفاده می‌شود:

```
(نوع پارامتر ورودی دوم , نوع پارامتر ورودی اول) (نامی دلخواه *)  نوع خروجی
```

پس از تعریف اشاره‌گر به تابع، تابع مقایسه‌کننده را در این اشاره‌گر قرار می‌دهیم. حال در پارامتر دوم قالب مربوط به مجموعه، اعلان تابع را که مشابه با ساختار بالاست، قرار می‌دهیم. در اعلان تابع نیازی به ذکر نام نیست.

```
bool(*)(const stsample&, const stsample&)
```

حال کافی است نام اشاره‌گر به تابع تعریف شده را در پارامتر ورودی مجموعه قرار دهیم.

```
set<stsample, bool(*)(const stsample&, const stsample&)> stcontent(fn);
```

تابع begin پیمایشگری به نخستین خانه از مجموعه برمی‌گرداند. تابع end نیز پیمایشگری به خانه‌ی پس از خانه‌ی پایانی در مجموعه برمی‌گرداند. با به‌کارگیری این دو تابع می‌توانید به راحتی مجموعه را پیمایش کرده و به عنصرهای موجود در آن دسترسی داشته باشید. برای درک بیشتر به این کد توجه کنید:

```
#include <iostream>
#include <set>
using namespace std;
int main() {
    int myints[] = {75, 23, 65, 42, 13};
    set<int> myset(myints, myints + 5);
    set<int>::iterator it;
    cout << "myset contains:";
    for (it = myset.begin(); it != myset.end(); it++)
        cout << " " << *it;
    cout << endl;
    return 0;
}
```

خروجی کد قبل به این صورت خواهد بود:

```
Output
myset contains: 13 23 42 65 75
```

توابع rbegin و rend به ترتیب پیمایشگرهایی به پایان و آغاز لیست برمی‌گردانند (دقیقا برعکس توابع begin و end). تابع clear همه‌ی عنصرهای موجود در مجموعه را حذف می‌کند. تابع erase برای حذف عنصر یا عنصرهایی خاص از مجموعه به کار می‌رود. حذف عنصرها به اشکال زیر (پارامترهای ورودی متفاوت) امکان‌پذیر است:

۱- position: این پارامتر، پیمایشگری به عنصری است که قصد حذف آن را داریم.

۲- x: این پارامتر حاوی مقداری است که می‌خواهیم آن را از مجموعه حذف کنیم.

۳- first و last: به کمک این دو پارامتر (که هر دو پیمایشگر هستند) محدوده‌ی مورد نظر مشخص می‌شود. بنابراین همه‌ی عنصرهایی که درون این محدوده وجود دارند، به همراه خود first (عنصری که به آن اشاره می‌کند) از مجموعه حذف خواهند شد.

تابع empty خالی بودن/نبودن مجموعه را بررسی می‌کند. اگر مجموعه خالی باشد مقدار یک وگرنه مقدار صفر را برمی‌گرداند. تابع find وجود یا عدم وجود یک عنصر در مجموعه را بررسی می‌کند. اگر عنصر مورد نظر موجود باشد، پیمایشگری به آن عنصر برگردانده شده و در غیر این صورت پیمایشگری به پایان مجموعه (معادل استفاده از تابع end) برگردانده می‌شود. به مثال بعدی توجه کنید:

```
#include <iostream>
#include <set>
using namespace std;
int main() {
    set<int> myset;
    for (int i = 1; i <= 5; i++)
        myset.insert(i * 10);
    if (myset.find(20) != myset.end())
        cout << "20 exists in set." << endl;
    if (myset.find(60) == myset.end())
        cout << "60 does not exist in set." << endl;
    return 0;
}
```

تابع ()count یک عنصر را به عنوان پارامتر ورودی دریافته کرده و تعداد تکرار آن را در مجموعه برمی‌گرداند. از آنجا که عنصر تکراری در مجموعه وجود ندارد، این تابع نیز می‌تواند برای تشخیص وجود یا عدم وجود یک عنصر در مجموعه استفاده شود. به این ترتیب که در صورت وجود عنصر مورد نظر، مقدار یک را برمی‌گرداند. در واقع این تابع کارکردی مشابه با تابع find دارد. به این ترتیب کد قبلی به این صورت قابل بازنویسی خواهد بود:

```
#include <iostream>
#include <set>
using namespace std;
int main() {
    set<int> myset;
    for (int i = 1; i <= 5; i++)
        myset.insert(i * 10);
    if (myset.count(20) == 1)
        cout << "20 exists in set." << endl;
    if (myset.count(60) == 0)
        cout << "60 does not exist in set." << endl;
    return 0;
}
```

تابع insert برای قرار دادن عنصر یا مجموعه‌ای از عنصرها در مجموعه به کار می‌رود. البته طی این فرآیند مقادیر تکراری به صورت خودکار حذف خواهند شد. این تابع به چند شکل و با پارامترهای ورودی متفاوت عمل می‌کند:

۱- x: مقداری که قصد داریم آن را اضافه کنیم. این مقدار باید از جنس عنصرهای مجموعه باشد. در این حالت، مقدار بازگشتی تابع، یک نمونه از جنس کلاس pair است (این کلاس در بخش بعدی توضیح

داده خواهد شد) که بخش اول آن (first) پیمایشگری به عنصر تازه اضافه شده یا به عنصری که پیش از این موجود بوده، است؛ و بخش دوم آن (second) در صورتی کـه عنصـر وارد شـده در مجموعـه وجـود نداشته باشد، برابر true و در غیر این صورت برابر false خواهد بود.

۲- position: به کمک این پیمایشگر می‌توان مشخص کرد که برای تعیین محل دقیق عنصر جدیـد از کـدام عنصر آغاز به مقایسه کند. با کمک این متغیر فقط می‌توان کارایی تابع insert را بالا برد. در صورتی کـه لازم باشد، عنصر جدید پیش از عنصر مشخص شده با متغیر position قرار گیرد، این پیمایشـگر نادیـده گرفته می‌شود.

۳- first و last: با کمک این دو پیمایشگر، محدوده‌ای از اعداد که قصد درج آن‌ها در مجموعه را داریم، تعیین می‌کنیم. لازم به ذکر است که این محدوده از همه‌ی عنصرهای بین این دو پیمایشگر به اضـافه‌ی first تشکیل می‌شود.

کد بعدی انواع روش‌های به‌کارگیری تابع insert را نشان می‌دهد:

```
#include <set>
using namespace std;
int main() {
    set<int> myset;
    set<int>::iterator it;
    pair<set<int>::iterator, bool> ret;
    for (int i = 1; i <= 5; i++)
        myset.insert(i * 10);          //10 20 30 40 50
    ret = myset.insert(20);            //no new element inserted
    if (ret.second == false)
        it = ret.first;                //"it" now points to element 20
    myset.insert(it, 25);              //max efficiency inserting
    myset.insert(it, 24);              //max efficiency inserting
    myset.insert(it, 26);              //no max efficiency inserting
    int myints[] = { 5,10,15 };        //10 already in set, not inserted
    myset.insert(myints, myints + 3); //5 10 15 20 24 25 26 30 40 50
    return 0;
}
```

تابع max_size بیشترین تعداد عنصرهایی را که می‌تواند در مجموعه قرار گیرد برمی‌گرداند.

```
#include <iostream>
#include <set>
using namespace std;
int main() {
    set<int> myset;
    if (myset.max_size() > 1000) {
        for (int i = 0; i < 1000; i++)
            myset.insert(i);
        cout << "The set contains 1000 elements.\n";
    }
```

```
    else
        cout << "The set could not hold 1000 elements.\n";
    return 0;
}
```

۳-۵- نگاشت

ساختمان داده نگاشت مشابه ساختمان‌داده‌ی جدول درهم‌سازی است که در همین فصل توضیح داده خواهد شد؛ با این تفاوت که نگاشت به صورت درونی مانند جدول درهم‌سازی عمل نمی‌کند. نگاشت دربرگیرنده‌ی ترکیبی از کلید ها و داده‌های مربوط به آن‌ها است. هر کلید برای مشخص کردن یک عضو یکتا به کار برده می‌شود. یعنی هر عنصر توسط کلید خود شناخته می‌شود. برای نمونه، روشن‌ترین کاربرد نگاشت را می‌توان در مسائلی شبیه به نگهداری اطلاعات یک دفترچه تلفن دانست که هر شماره تلفن مختص به یک شخص است. بنابراین، شماره تلفن را می‌توان کلید هر شخص در نظر گرفت.

در نتیجه دو ویژگی مهم در نگاشت عبارتست از:

۱- کلیدهای یکتا: هیچ دو عضوی از نگاشت دارای کلید یکسان نیستند.

۲- هر عضو دارای یک کلید و یک داده است.

برای به‌کارگیری کلاس نگاشت باید سرآیند map را include کنید. عنصرهای موجود در نگاشت به صورت پیش‌فرض برحسب کلیدشان به صورت صعودی مرتب می‌شوند. می‌دانیم که هر عضو نگاشت از یک جفت عنصر تشکیل می‌شود (کلید و داده‌ی مربوط به آن). کلاس pair برای این منظور در نظر گرفته شده است. به این معنی که هر عضو نگاشت، یک جفت است. برای به‌کارگیری این کلاس به این نمونه ساده توجه کنید:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    pair<char, int> p1;
    p1.first = 'a';
    p1.second = (int) 'a';
    cout << "Key:" << p1.first << ", Value:" << p1.second << endl;
    return 0;
}
```

برای ایجاد شیءای از این کلاس، کافی است نوع کلید و داده مشخص شود؛ در کد نشان داده شده، کلید از نوع char و داده از نوع int تعیین شده است. برای دسترسی به کلید (مقداردهی و خواندن مقدار آن) از خصیصه first و برای دسترسی به داده، خصیصه second را به کار می‌بریم. این کلاس تنها دارای یک تابع به نام swap است که مقادیر دو نمونه از کلاس را باهم جابجا می‌کند. به این نمونه توجه کنید:

```
#include <iostream>
#include <map>
using namespace std;
```

```
int main() {
    pair<char, int> p1;
    pair<char, int> p2;
    p1.first = 'a';
    p1.second = (int) 'a';
    p2.first = 'A';
    p2.second = (int) 'A';
    cout << "1. " << "Key:" << p1.first << ", Value:" << p1.second <<
endl;
    cout << "2. " << "Key:" << p2.first << ", Value:" << p2.second <<
endl;
    swap(p1, p2);
    cout << "After swapping: " << endl;
    cout << "1. " << "Key:" << p1.first << ", Value:" << p1.second <<
endl;
    cout << "2. " << "Key:" << p2.first << ", Value:" << p2.second <<
endl;
    return 0;
}
```

خروجی کد نشان داده شده به این صورت خواهد بود:

```
Output
1. Key:a, Value:97
2. Key:A, Value:65
After swapping:
1. Key:A, Value:65
2. Key:a, Value:97
```

حال که با کلاس pair آشنا شدید، به سراغ کلاس map می‌رویم. این کلاس دارای چند سازنده است که در ادامه تعدادی از پرکاربردترین آن‌ها را توضیح می‌دهیم. این کلاس نیز با استفاده از الگو پیاده‌سازی شده است، بنابراین می‌توانید انواع دلخواهتان را برای کلید و داده‌ها استفاده کنید.

ساده‌ترین شکل سازنده این کلاس تنها دربرگیرنده نوع کلید و داده است و خود سازنده هیچ پارامتری ندارد:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<char, int> first;
    first['a'] = 10;
    first['b'] = 30;
    first['c'] = 50;
    first['d'] = 70;
    return 0;
}
```

توسط کد نشان داده شده نگاشتی ساخته می‌شود که دارای چهار عضو هست. با توجه به مطالب گفته شده، سعی کنید اطلاعات درون این چهار عضو را مشخص کنید.

سازنده می‌تواند دربرگیرنده‌ی نام شیءای دیگر (از همین نوع) نیز باشد که در این صورت، محتوای آن شیء را در شیء جدید رونویسی می‌کند. افزون بر این، می‌توان دو پیمایشگر را که به دو نقطه از نگاشتی دیگر اشاره می‌کنند نیز به کار برد؛ که عنصرهای میان دو پیمایشگر در شیء جدید قرار خواهند گرفت. به مثال بعد توجه کنید:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<char, int> first;
    first['a'] = 10;
    first['b'] = 30;
    first['c'] = 50;
    first['d'] = 70;
    map<char, int> second(first.begin(), first.end());
    map<char, int> third(second);
    return 0;
}
```

با اجرای کد نشان داده شده، نگاشت‌های second و third دقیقا محتوای نگاشت first را درون خود خواهند داشت.

الگوی کلاس map نیز مانند بعضی از کلاس‌های پیشین می‌تواند تابعی مقایسه‌کننده دریافت کند. مقایسه‌کننده می تواند به دو شکل کلاس (با پیاده‌سازی عملگر ()) و تابعی با دو پارامتر ورودی باشد. همان‌طور که در بخش‌های پیشین توضیح داده شد، نوع پارامتر و چگونگی کار تابع در هر دو روش یکسان است. تنها تفاوت در نگاشت‌ها این است که مقایسه فقط برروی کلیدها صورت می‌گیرد. بنابراین نوع پارامترهای ورودی باید از نوع کلیدها باشد. به علت شباهت این قسمت به قسمت مربوط به پیاده‌سازی مقایسه‌کننده مجموعه‌ها، در اینجا از توضیح اضافی خودداری شده و تنها به نشان دادن قطعه کد بعدی که هر دو روش را شامل می‌شود، بسنده می‌کنیم:

```
#include <iostream>
#include <map>
using namespace std;
bool fncomp(char lhs, char rhs) {
    return lhs < rhs;
}
struct classcomp {
    bool operator() (const char& lhs, const char& rhs) const {
        return lhs<rhs;
    }
};
int main() {
    map<char, int, classcomp> fourth;
    bool(*fn_pt)(char, char) = fncomp;
    map<char, int, bool(*)(char, char)> fifth(fn_pt);
    return 0;
}
```

عملگر انتساب نیز برای کلاس نگاشت تعریف شده است و با اعمال این عملگر به دو شیء از این کلاس، عنصرهای موجود در شیء سمت راست در دیگری رونویسی می‌شوند. دو تابع begin و end دو پیمایشگر به آغاز (نخستین عنصر) و پایان (پس از عنصر پایانی) برمی‌گردانند. این دو تابع برای پیمایش شیء map مناسب هستند. برای درک بیشتر طرز کار این دو تابع به این مثال توجه کنید:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<char, int> mymap;
    map<char, int>::iterator it;
    mymap['b'] = 100;
    mymap['a'] = 200;
    mymap['c'] = 300;
    for (it = mymap.begin(); it != mymap.end(); it++)
        cout << (*it).first << " => " << (*it).second << endl;
    return 0;
}
```

همانطور که می‌بینید، شباهت زیادی در کارکرد کلاس‌های کتابخانه استاندارد وجود دارد. عملکرد متدهایی مانند empty، size، max_size، erase، swap، clear، count مشابه همتاهای خود در کلاس set که در بخش پیشین توضیح داده شد، کار می‌کنند.

ویژگی بارز کلاس map که آن را بسیار کاربردی ساخته، پیاده‌سازی عملگر [] است. با به‌کارگیری این عملگر می‌توانید به راحتی به داده‌ی مربوط به کلید مورد نظر دست پیدا کنید. برای قرار دادن کلیدی جدید و مقدار مربوط به آن نیز می‌توانید همین عملگر را به کار ببرید به این کد توجه کنید:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<char, int> mymap;
    mymap['a'] = 100;
    cout << mymap['a'] << endl;
    mymap['a'] = mymap['a'] + 200;
    cout << mymap['a'] << endl;
    return 0;
}
```

در کد نشان داده شده ، با انتساب اول کلید 'a' ایجاد شده و عدد ۱۰۰ به عنوان داده‌ی آن قرار می‌شود. انتساب دوم مقدار داده‌ی مربوط به کلید 'a' را تغییر می‌دهد. برای قرار دادن عنصرهای جدید در نگاشت، علاوه بر روش ساده‌ی بالا، می‌توان تابع insert را نیز به کار برد. پارامترهای این تابع کاملا مشابه با کلاس set است، با این تفاوت که عنصر مورد قبول برای این تابع از نوع کلاس pair می‌باشد. نمونه زیر طرز کار این تابع را به سادگی نشان می‌دهد:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<char, int> mymap;
    map<char, int>::iterator it;
    pair<map<char, int>::iterator, bool> ret;
    mymap.insert(pair<char, int>('a', 100));
    mymap.insert(pair<char, int>('z', 200));
    ret = mymap.insert(pair<char, int>('z', 500));
    if (ret.second == false) {
        cout << "element 'z' already existed";
        cout << " with a value of " << ret.first->second << endl;
    }
    it = mymap.begin();
    mymap.insert(it, pair<char, int>('b', 300));
    mymap.insert(it, pair<char, int>('c', 400));
    map<char, int> anothermap;
    anothermap.insert(mymap.begin(), mymap.find('c'));
    cout << "mymap contains:\n";
    for (it = mymap.begin(); it != mymap.end(); it++)
        cout << (*it).first << " => " << (*it).second << endl;
    cout << "anothermap contains:\n";
    for (it = anothermap.begin(); it != anothermap.end(); it++)
        cout << (*it).first << " => " << (*it).second << endl;
    return 0;
}
```

خروجی کد نشان داده شده به این صورت خواهد بود:

```
Output
element 'z' already existed with a value of 200
mymap contains:
a => 100
b => 300
c => 400
z => 200
anothermap contains:
a => 100
b => 300
```

تابع find آخرین تابعی است که در این بخش برای کلاس map توضیح می‌دهیم. برای تشخیص وجود یا عدم وجود یک کلید در نگاشت، این تابع را به کار می‌بریم. اگر کلید مورد نظر در نگاشت وجود داشته باشد، یک پیمایشگر که به آن عنصر اشاره می‌کند، برگردانده می‌شود وگرنه پیمایشگری به پایان نگاشت (end) برگردانده خواهد شد:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<char, int> mymap;
    mymap['a'] = 50;
    mymap['b'] = 100;
    if (mymap.find('b') == mymap.end())
        mymap['b'] = 250;
    cout << "a => " << mymap.find('a')->second << endl;
    cout << "b => " << mymap.find('b')->second << endl;
    return 0;
}
```

۳-٦- بردار

بردار کلاسی برای نگهداری مجموعه‌ای از عنصرها است که تا حدودی مشابه لیست عمل می‌کند. عنصرهای موجود در بردار در حافظه‌ای پیوسته ذخیره شده که این امر موجب می‌شود علاوه بر قابل پیمایش بودن بردار، قابلیت دستیابی به عنصرها توسط نمایه نیز وجود داشته باشد.

حافظه مورد نیاز در بردارها به صورت خودکار مدیریت می‌شود و امکان افزایش و کاهش آن نیز وجود دارد. بردارها دارای ویژگی‌های بارز زیر هستند:

۱- دسترسی به عنصرهای موجود در بردار توسط نمایه‌ی آن‌ها در زمان ثابت

۲- پیمایش عنصرهای بردار در هر دو جهت در زمان خطی

۳- افزودن و حذف عنصرها از پایان آن در زمان ثابت

از آنجا که بردارها امکان تغییر اندازه خودکار دارند، حافظه‌ی بیشتری از ساختمان‌داده‌های مشابه اشغال می‌کنند؛ زیرا باید مقداری از حافظه را رزرو کرده تا بتوانند حافظه را به صورت پیوسته افزایش دهند. در مقایسه با دیگر ساختمان داده‌های مشابه، بردارها در کل موثرتر و سریع‌تر عمل می‌کنند. در ادامه به نحوه استفاده از این کلاس در کتابخانه الگوی استاندارد پرداخته می‌شود.

برای به‌کارگیری بردارها باید سرآیند vector را به کار ببرید. این کلاس دارای سازنده‌هایی مشابه با کلاس list است که نمونه کد بعدی انواع آن را به روشنی نشان می‌دهد:

```
#include <vector>
using namespace std;
int main() {
    vector<int> first;
    vector<int> second(4, 100);
    vector<int> third(second.begin(), second.end());
    vector<int> fourth(third);
    int myints[] = {16, 2, 77, 29};
    vector<int> fifth(myints, myints + sizeof(myints) / sizeof(int));
    return 0;
```

```
}
```

دو تابع begin و end به ترتیب به عنصر نخست و پس از عنصر آخر بردار اشاره می‌کنند؛ با این توابع و دو تابع ebegin و rend در بخش‌های پیش آشنا شدیم. برای دستیابی به عنصرهای بردار، می‌توانیم از پیمایشگرها و یا نمایه استفاده کنیم. در نمونه کد بعدی عنصرهای بردار پیمایش شده و به هر دو روش چاپ می‌شوند:

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> myvector;
    for (int i = 1; i <= 5; i++)
        myvector.push_back(i);
    vector<int>::iterator it;
    int i = 0;
    cout << "myvector contains:";
    for (i = 0, it = myvector.begin(); it < myvector.end(); it++, i++)
        cout << " " << *it << " " << myvector[i];
    cout << endl;
    return 0;
}
```

خروجی کد قبلی به این صورت خواهد بود:

```
Output
myvector contains: 1 1 2 2 3 3 4 4 5 5
```

مانند سایر ساختمان‌داده‌هایی که تا به حال با آن‌ها آشنا شدیم، کلاس vector نیز دارای تابع size است که تعداد عنصرهای موجود در آن را مشخص می‌کند؛ ولی بردارها علاوه بر اندازه دارای ظرفیت نیز هستند؛ ظرفیت مقدار حافظه‌ی تخصیص داده شده به بردار است. این حافظه می‌توانند مساوی یا بیشتر از اندازه‌ی بردار باشد و برای تغییر اندازه‌ی بردار موثر است. اگر به سرعت بالایی نیاز دارید، ظرفیت بالایی برای بردار تعیین کنید، تا در هنگام تغییر اندازه نیاز به تخصیص حافظه‌ی جدید پیدا نکرده و تغییر اندازه سریعتر صورت گیرد.

همان‌طور که گفته شد، تابع size عملکردی مشابه با ساختمان‌داده‌های پیشین دارد. تابع جدیدی که مختص به بردارهاست، تابع capacity است که مقدار حافظه اشغال شده توسط بردار را برمی‌گرداند. این مقدار لزوما برابر با تعداد عنصرهای موجود در بردار نیست. تابع resize برای تغییر اندازه‌ی بردار و تابع reserve برای تغییر ظرفیت بردار به کار می‌روند. در هنگام تغییر اندازه، اگر اندازه‌ی جدید کمتر از اندازه‌ی کنونی بردار باشد، عنصرهای اضافی پایان بردار (به تعداد اختلاف میان دو اندازه) حذف می‌شوند. اگر اندازه‌ی جدید بردار، بزرگتر از اندازه‌ی کنونی آن باشد، اندازه بردار افزایش پیدا کرده و مقدار عنصرهای جدید افزوده شده برابر با مقدار پیش‌فرض (پارامتر دوم تابع resize) قرار داده خواهد شد. بنابراین تابع resize دارای دو پارامتر است که اولی اندازه جدید بردار و دومی مقدار پیش‌فرض عنصرهای جدید است. در ادامه نمونه کدی برای آشنایی با چهار تابع معرفی شده خواهیم دید:

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> myvector;
    for (int i = 1; i <= 5; i++)
        myvector.push_back(i);
    cout << myvector.size() << " " << myvector.capacity() << endl;
    myvector.resize(myvector.size() + 3, 10);
    myvector.reserve(myvector.size() + 10);
    cout << myvector.size() << " " << myvector.capacity() << endl;
    return 0;
}
```

تابع at مشابه با عملگر [] عمل می‌کند و ارجاعی به عنصر موجود در نمایه‌ی مشخص شده برمی‌گرداند؛ تنها تفاوت میان این دو روش در این است که تابع at وقتی نمایه‌ای خارج از محدوده‌ی اندازه‌ی بردار دریافت کند، استثنایی با محتوای «خارج از محدوده بودن» ایجاد می‌کند.

دو تابع front و back به ترتیب برای دسترسی به عنصر اول و آخر بردار به کار برده می‌شوند. تابع push_back و pop_back نیز همانطور که از نامشان پیداست، به ترتیب برای قرار دادن یک عنصر در پایان بردار و خارج کردن عنصری از پایان بردار به کار برده می‌شوند. طرز کار این چهار تابع در نمونه کد بعدی نشان داده شده است:

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> myvector;
    for (int i = 1; i <= 5; i++)
        myvector.push_back(i);
    while (!myvector.empty()) {
        cout << myvector.front() << " " << myvector.back() << endl;
        myvector.pop_back();
    }
    return 0;
}
```

تابع assign برای انتساب مجموعه عنصرهایی جدید به بردار به کار می‌رود؛ به شکلی که محتوای پیشین بردار حذف خواهد شد؛ این تابع به دو شکل قابل به‌کارگیری است. شکل اول دارای دو پارامتر است که اولی تعداد تکرار عنصر جدید و دومی مقدار عنصر مورد نظر است؛ که عنصر جدید را به تعداد تکرار گفته شده در بردار قرار می‌دهد. در شکل دوم، پارامتر اول و دوم دو پیمایشگر به یک بردار (یا یک آرایه هم‌نوع با بردار) هستند؛ این شکل فراخوانی

باعث حذف محتوای پیشین بردار و قرار گرفتن عنصرهای موجود در میان دو پیمایشگر داده شده در بردار می‌شود. طرز کار تابع assign در این کد نشان داده شده است:

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> first;
    vector<int> second;
    vector<int> third;
    first.assign(7, 100);
    vector<int>::iterator it;
    it = first.begin() + 1;
    second.assign(it, first.end() - 1);
    int myints[] = {23, 7, 41};
    third.assign(myints, myints + 3);
    return 0;
}
```

سایر توابع این کلاس کاملا مشابه با ساختمان‌داده‌های پیشین بوده و از توضیح آن‌ها خودداری می‌کنیم.

۳-۷- درخت جستجوی دودویی

درخت جستجوی دودویی از ساختمان‌داده‌های کاربردی است که اهمیت ویژه‌ای در الگوریتم‌های مرتب‌سازی دارد. علاوه‌براین در پیاده‌سازی ساختمان‌داده‌های دیگری مانند مجموعه‌ها نیز (که در بخش‌های گذشته معرفی شد) کاربرد دارد. درخت جستجوی دودویی دارای کلاس مستقلی در کتابخانه‌ی الگوی استاندارد سی++ نیست ولی کوشش می‌کنیم توابع مورد نیاز این ساختمان‌داده را توضیح دهیم. درخت جستجوی دودویی، درختی دودویی با این قوانین است:

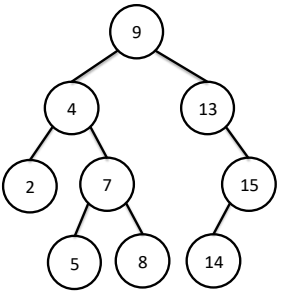
۱- هر گره‌ای در درخت دارای یک مقدار یکتا است (مقادیر تکراری مجاز نیست).

۲- زیردرخت راست و چپ خود درخت جستجوی دودویی هستند.

۳- فرزندان سمت چپ هر گره، فقط می‌توانند مقادیر کمتر از آن گره را داشته باشند.

۴- فرزندان سمت راست هر گره، فقط می‌توانند مقادیر بیشتر از آن گره را داشته باشند.

نمونه‌ای از یک درخت جستجوی دودویی در شکل ۳,۳ نشان داده شده است.



شکل ۳,۳. نمونه‌ای از یک درخت جستجوی دودویی

برای جلوگیری از قرار گرفتن مقادیر تکراری در درخت، باید این امر را در پیاده‌سازی مدنظر قرار دهیم. درخت جستجوی دودویی دارای توابع زیر است:

۱- جستجوی یک گره در درخت

۲- افزودن یک گره به درخت

۳- حذف یک گره از درخت

۴- پیمایش درخت

پیش از توضیح توابع مذکور، ساختمان داده‌ای ساده برای درخت جستجوی دودویی به زبان سی++ به شکل زیر معرفی می‌کنیم:

```
struct BST {
    int value;
    BST* left;
    BST* right;
};
```

ساختار نشان داده شده، ساختمان هر گره از درخت را توصیف می‌کند. هر گره دارای این سه بخش است:

۱- مقدار

۲- اشاره‌گری به فرزند چپ

۳- اشاره‌گری به فرزند راست

در ادامه با به‌کارگیری ساختمان‌داده‌ی معرفی شده به پیاده‌سازی توابع مورد نیاز برای یک درخت جستجوی دودویی می‌پردازیم.

۳-۷-۱- جستجوی یک گره در درخت

برای یافتن یک گره در درخت، کار ساده‌ای پیش‌رو داریم. برای این منظور یک تابع بازگشتی می‌نویسیم که به صورت زیر عمل می‌کند:

۱- اگر مقدار گره کنونی با مقدار مورد جستجو برابر باشد، گره مورد نظر، گره هدف است. پس جستجو را خاتمه داده و گره کنونی را برمی‌گردانیم.

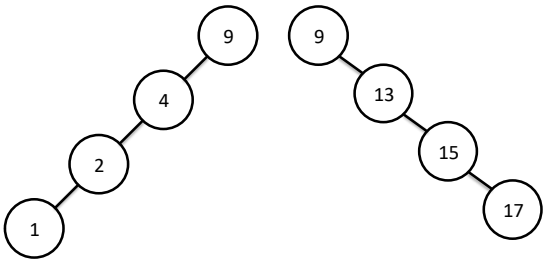
۲- اگر مقدار گره کنونی از مقدار مورد جستجو بزرگتر باشد، مطمئنا مقدار مورد نظر در صورت وجود در زیردرخت چپ گره کنونی وجود دارد. بنابراین تابع را برای زیردرخت چپ فراخوانی می‌کنیم.

۳- اگر مقدار گره کنونی از مقدار مورد جستجو کوچکتر باشد، مطمئنا مقدار مورد نظر (در صورت وجود) در زیردرخت راست گره کنونی وجود دارد. بنابراین تابع را برای زیردرخت راست فراخوانی می‌کنیم.

کد بعدی بدنه تابع جستجو را نشان می‌دهد. اگر گره مورد نظر در درخت یافت شود، مقدار آن گره وگرنه مقدار ۱- بازگردانده خواهد شد.

```
int search_bst(BST* node, int key) {
    if (node == NULL)
        return -1;
    if (key < node->value)
        return search_bst(node->left, key);
    if (key > node->value)
        return search_bst(node->right, key);
    else
        return node->value;
}
```

پیچیدگی زمانی جستجو در درخت جستجوی دودویی در حالت میانگین $O(\log n)$ است. زیرا طی کردن یک ارتفاع از درخت برای یافتن گره مورد نظر یا تشخیص عدم وجود آن کافی است. البته در بدترین حالت به $O(n)$ نیز خواهد رسید. بدترین حالت زمانی رخ می‌دهد که همه‌ی گره‌ها در یک راستا (چپ یا راست) قرارگیرند (شکل ۴,۳).



شکل ٤,٣. نمایی از دو درخت جستجوی دودویی که در آنها تمامی گره‌ها در یک راستا قرار گرفته‌اند.

۳-۷-۲- افزودن یک گره به درخت

برای افزودن یک گره به درخت باید ابتدا مکان مناسبی را برای آن بیابیم. برای این منظور با دو قانون زیر در عمق درخت پیش می‌رویم:

۱- اگر مقدار گره کنونی از گره مورد نظر کوچکتر بود، جستجو را در زیردرخت سمت راست گره کنونی ادامه می‌دهیم.

۲- اگر مقدار گره کنونی از گره مورد نظر بزرگتر بود، جستجو را در زیردرخت سمت چپ گره کنونی ادامه می دهیم.

اگر مقدار گره کنونی برابر با گره مورد نظر باشد، جستجو خاتمه یافته و هیچ مقدار جدیدی به درخت افزوده نخواهد شد. کد بعدی یک پیاده‌سازی برای تابع افزودن گره جدید به درخت نوشته شده است. این تابع نیز دارای پیچیدگی زمانی $O(\log n)$ در حالت میانگین و پیچیدگی $O(n)$ در بدترین حالت است.

```
void insert_bst(BST* &treeNode, BST* newNode) {
    if (treeNode == NULL)
        treeNode = newNode;
    else if (newNode->value < treeNode->value)
        insert_bst(treeNode->left, newNode);
    else
        insert_bst(treeNode->right, newNode);
}
```

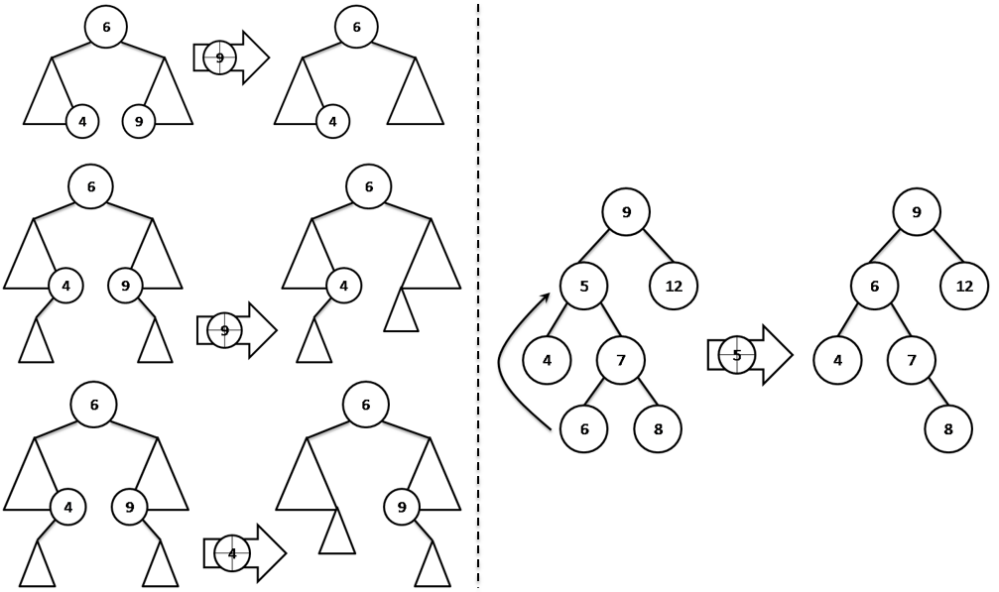
۳-۷-۳- حذف یک گره از درخت

برای حذف یک گره از درخت یکی از سه حالت زیر رخ می‌دهد. این حالت‌ها در شکل ۵,۳ نشان داده شده‌اند.

۱- حذف گره‌ای بدون فرزند (برگ): در این حالت گره‌ی مورد نظر به سادگی حذف می‌شود.

۲- حذف گره‌ای با یک فرزند: در این حالت پس از حذف گره، فرزندش در مکان آن قرار خواهد گرفت.

۳- حذف گره‌ای با دو فرزند: در این حالت پس از حذف گره، چپ‌ترین فرزند زیردرخت راست گره و یا راست ترین فرزند زیردرخت چپ گره، جایگزین آن خواهد شد. زیرا مقدار این گره، از همه گره‌های زیردرخت راست کوچکتر و از همه گره‌های زیردرخت چپ بزرگتر است.



شکل ۵,۳. حالت‌های مختلفی که ممکن است هنگام حذف یک گره از یک درخت جستجوی دودویی رخ دهد.

تابع نشان داده شده در نمونه کد بعدی گره ورودی را در صورت وجود از درخت جستجوی دودویی حذف می‌کند. اگر حذف با موفقیت صورت گیرد، مقدار true و درغیر این صورت مقدار false برگردانده می‌شود. پیچیدگی زمانی حذف یک گره از درخت در حالت میانگین $O(\log n)$ و در بدترین حالت $O(n)$ است.

```
bool delete_bst(BST* node, int val) {
    BST* successor;
    BST* node_delete;
    if (node) {
        if (node->value > val)
            return delete_bst(node->left, val);
        else if (node->value < val)
            return delete_bst(node->right, val);
        else {
            if (node->left == NULL) {
                node_delete = node;
                node = node->right;
                free(node_delete);
            }
            else if (node->right == NULL) {
                node_delete = node;
                node = node->left;
                free(node_delete);
            }
            else {
```

```
                successor = leftmost_child(node->right);
                node->value = successor->value;
                return delete_bst(node->right, successor->value);
            }
            return true;
        }
    }
    else
        return false;
}
```

۳-۷-٤- پیمایش درخت

درخت‌های دودویی به سه شکل قابل پیمایش هستند. ترتیب ملاقات گره‌ها برای هر یک توضیح داده شده است.

پیشوندی: الف) ملاقات ریشه ب) ملاقات فرزند چپ ج) ملاقات فرزند راست

میانوندی: الف) ملاقات فرزند چپ ب) ملاقات ریشه ج) ملاقات فرزند راست

پسوندی: الف) ملاقات فرزند چپ ب) ملاقات فرزند راست ج)ملاقات ریشه

در پیمایش میانوندی، درخت جستجوی دودویی اعداد را به ترتیب صعودی ملاقات خواهد کرد (چرا؟). بنابراین می توان با به‌کارگیری این پیمایش، اعداد را به صورت مرتب شده در خروجی داشته باشیم. در این بخش فقط پیمایش میانوندی را پیاده‌سازی کرده و دو روش دیگر را به خواننده واگذار می‌کنیم.

تابع بازگشتی نشان داده شده به سادگی پیمایش میانوندی را پیاده‌سازی می‌کند. با کمی تغییر در کد زیر می‌توانید روش‌های دیگر پیمایش درخت دودویی را پیاده‌سازی کنید.

```
void inorder_bst(BST* tree) {
    if (tree == NULL)
        return;
    inorder_bst(tree->left);
    cout << tree->value << " ";
    inorder_bst(tree->right);
}
```

پیچیدگی زمانی پیمایش درخت $O(n)$ است؛ زیرا همه گره‌های درخت باید ملاقات شوند. برای نمونه، پیمایش میانوندی درخت شکل ۳,۳ به صورت زیر خواهد بود (از چپ به راست):

Output
2 4 5 7 8 9 13 14 15

جدول ۱,۳ پیچیدگی زمانی عملیات‌های توضیح داده شده را در حالت میانگین و بدترین حالت نشان می‌دهد.

جدول ۱,۳. پیچیدگی زمانی عملیات مختلف درخت جستجوی دودویی

عملیات	حالت میانگین	بدترین حالت
افزودن	$O(\log n)$	$O(n)$
حذف	$O(\log n)$	$O(n)$
جستجو	$O(\log n)$	$O(n)$
پیمایش	$O(n)$	$O(n)$

۳-۷-۵- معرفی ابزار معادل در کتابخانه استاندارد سی++

کتابخانه استاندارد سی++ برای جستجوی دودویی تابعی در نظر گرفته است که در ادامه آن را معرفی می‌کنیم. نام این تابع binary_search است و در سرآیند algorithm قرار دارد. این تابع یک لیست مرتب از اعداد و مقدار مورد نظر شما را دریافت کرده و به دنبال مقدار مورد نظر می‌گردد. اگر مقدار مورد نظر در لیست وجود داشته باشد، true وگرنه مقدار false بازگردانده می‌شود. این تابع را می‌توان به دو شکل زیر به کار برد:

```
bool binary_search(مورد نظر مقدار ,لیست پایان پیمایشگر ,لیست آغاز پیمایشگر);
bool binary_search(کننده مقایسه تابع ,مورد نظر مقدار ,لیست پایان پیمایشگر ,لیست آغاز پیمایشگر);
```

برای جستجو در درخت جستجوی دودویی، ابزار اصلی مقایسه است. از این رو در پیاده‌سازی دوم این تابع، می‌توانید تابع مقایسه‌کننده مورد نظرتان را نیز به آن معرفی کنید. تابع مقایسه‌کننده باید دارای دو ورودی از نوع مقادیر آرایه و خروجی از نوع bool باشد. این تابع مشخص می‌کند که آیا پارامتر اول کوچکتر از پارامتر دوم است یا خیر؟ در ادامه نمونه‌ای از هر دو پیاده‌سازی این تابع آمده است:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
bool compare(int a, int b) {
    return (a < b);
}
int main() {
    int a[] = {1, 4, 7, 2, 3, 6};
    vector<int> v(a, a + sizeof(a) / sizeof(int));
    vector<int>::iterator it;
    sort(v.begin(), v.end());
    cout << "search for 6:";
    if (binary_search(v.begin(), v.end(), 6))
        cout << " found" << endl;
    else
        cout << " not found" << endl;
    cout << "search for 10:";
    if (binary_search(v.begin(), v.end(), 10, compare))
        cout << " found" << endl;
    else
```

```
        cout << " not found" << endl;
    return 0;
}
```

خروجی کد نشان داده شده به این صورت خواهد بود:

Output
search for 6: found search for 10: not found

۳-۸- درخت اِی-وی-ال

درخت اِی-وی-ال یک درخت جستجوی دودویی متوازن است. متوازن بـه ایـن معنـی اسـت کـه اخـتلاف ارتفـاع زیردرخت چپ و راست هیچ گره‌ای از درخت، بیشتر از یک نیست. این ویژگی موجب می‌شود که پیچیـدگی زمـانی افزودن و حذف یک گره به/از درخت اِی-وی-ال و همچنین جستجوی یک گره در حالت میانگین و بدترین حالـت از مرتبه لگاریتمی باشد. درخت اِی-وی-ال در مقایسه با درخت جستجوی دودویـی معمـولی، سـرعت بیشـتری در جستجوی یک گره دارد زیرا درخت همیشه متوازن است؛ در عوض هزینه‌ی افزودن و حذف یک گره در درخت اِی-وی-ال افزایش یافته است، زیرا گاهی پس از افزودن و حذف یک گـره، تـوازن درخـت بـر هـم خـورده و نیـاز بـه متوازن‌سازی دارد. برای سنجش توازن درخت، مفهومی به نام عامل توازن به صورت زیر تعریف می‌شود (ارتفاع یک گره بدون فرزند، صفر در نظر گرفته می‌شود):

عامل توازن برابر است با قدرمطلق اختلاف ارتفاع زیر درخت راست و زیر درخـت چـپ (تفاضـل ارتفـاع زیردرخـت راست و اندازه ارتفاع زیردرخت چپ).

بر روی هر یک از گره‌ها در درخت اِی-وی-ال، عامل توازن آن گره نوشته می‌شود؛ هر گره‌ای کـه عامـل تـوازنش یکی از مقدارهای $\{۰ ، ۱\}$ باشد، متوازن وگرنه نامتوازن نامیده می‌شود. با توجه به این تعاریف، درخـت متـوازن بـه این صورت تعریف می‌شود: درختی که هیچ گره‌ی نامتوازنی در آن یافت نشود، یک درخت متوازن است.

بر اساس تعریف گفته شده اگر درختی نامتوازن باشد، دارای حداقل یک گره نامتوازن است. هنگامی که توازن درخت اِی-وی-ال بر هم می‌خورد، به متوازن‌سازی احتیاج خواهد داشت. متوازن‌سازی به معنی چرخش (جابجایی) گره‌هـا حول گره نامتوازن است به شکلی که توازن گره را برقرار کند. متوازن‌سازی به دو شکل زیر صورت می‌گیرد. همـراه با توضیح افزودن و حذف گره به/از درخت اِی-وی-ال به شکل‌های مختلف متوازن‌سازی نیز خواهیم پرداخت.

۱- چرخش یگانه

۲- چرخش دوگانه

۳-۸-۱- افزودن یک گره به درخت اِی-وی-ال

افزودن گره به درخت اِی-وی-ال، مشابه درخت جستجوی دودویی است؛ با این تفاوت که در هنگام افزودن گره به درخت اِی-وی-ال باید توازن گره‌ها نیز بررسی شود. اگر توازن گره‌ای بیشتر از یک شود، آن گره نـامتوازن بـوده و درخت در آن گره نیاز به متوازن‌سازی دارد. برای راحتی کار، در ادامه، گره نامتوازن را P، فرزند راسـت را R و فرزنـد چپ را L می‌نامیم. به صورت کلی چهار حالت به متوازن‌سازی احتیاج دارند که دو حالـت آن قرینـه دو حالـت دیگـر

هستند. از این چهار حالت، دو حالت به متوازن‌سازی یگانه و دو حالت دیگر به متوازن‌سازی دوگانه احتیاج دارنـد. در همه‌ی این حالت‌ها، مقدار عامل توازن گره P برابر با دو است. بسته بـه حالت‌هـایی کـه فرزنـدان ایـن گـره دارنـد، متوازن‌سازی به شکلی متفاوت صورت می‌گیرد. حالت‌های مختلف به شکل زیـر دسـته‌بندی مـی‌شـوند. شـکل ۶,۳ چرخش‌های یگانه و شکل ۷,۳ چرخش‌های دوگانه را نمایش می‌دهند.

۱- زیردرخت چپ ارتفاع بیشتری نسبت به زیردرخت راست دارد.

۱-۱- گره جدید به سمت چپ گره L اضافه شده است: چرخش یگانه به راست

۱-۲- گره جدید به سمت راست گره L اضافه شده است: چرخش دوگانه به چپ-راست

۲- زیردرخت راست ارتفاع بیشتری نسبت به زیردرخت چپ دارد.

۱-۳- گره جدید به سمت چپ گره R اضافه شده است: چرخش دوگانه به راست-چپ

۱-۴- گره جدید به سمت راست گره R اضافه شده است: چرخش یگانه به چپ

مراحل چرخش یگانه به راست به این صورت است:

۱- گره P به سمت راست می‌چرخد.

۲- گره L به ریشه منتقل می‌شود.

۳- گره C (فرزند چپ گره L) به مکان گره L منتقل می‌شود.

مراحل چرخش یگانه به چپ عبارت است از:

۱- گره P به سمت چپ می‌چرخد.

۲- گره R به ریشه منتقل می‌شود.

۳- گره C (فرزند راست گره R) به مکان گره R منتقل می‌شود.

مراحل چرخش دوگانه به چپ-راست به این صورت است:

۱- گره L به سمت چپ می‌چرخد و گره C جایگزین آن می‌شود.

۲- گره P به سمت راست می‌چرخد.

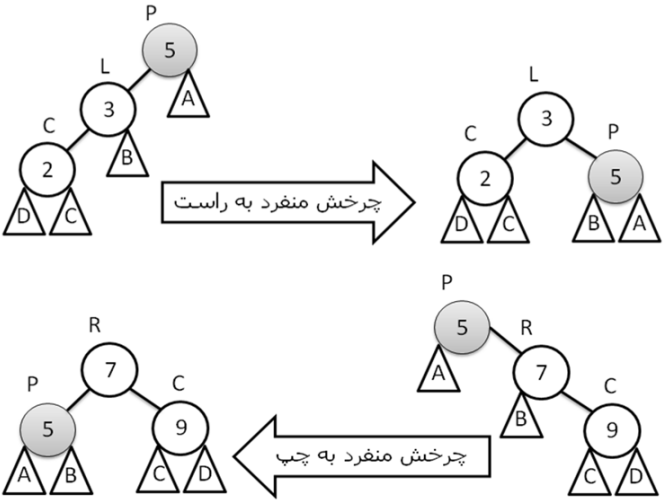
۳- گره C جایگزین ریشه می‌شود و در نتیجه گره L نیز جایگزین گره C خواهد شد.

مراحل چرخش دوگانه به راست-چپ عبارت است از:

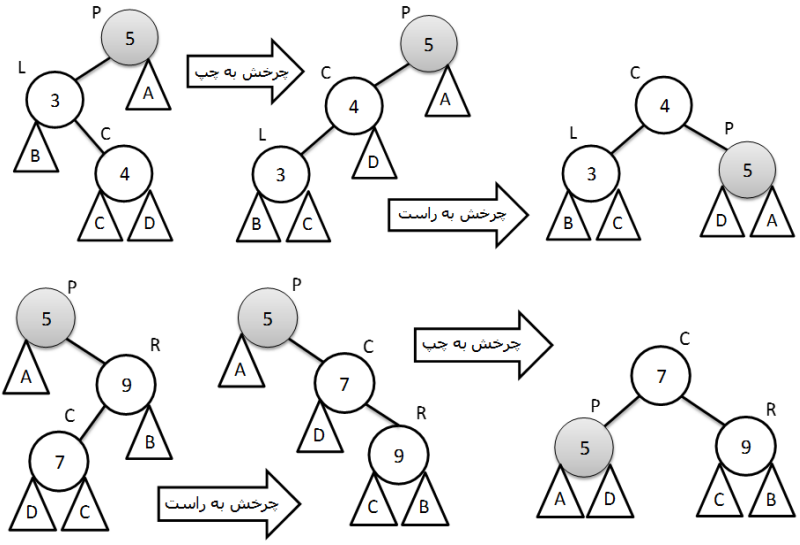
۱- گره R به سمت راست می‌چرخد و گره C جایگزین آن می‌شود.

۲- گره P به سمت چپ می‌چرخد.

۳- گره C جایگزین ریشه می‌شود و در نتیجه گره R نیز جایگزین گره C خواهد شد.



شکل ٦,٣. چرخش‌های یگانه در درخت اِی-وی-ال



شکل ۷,۳. چرخش‌های دوگانه در درخت اِی-وی-ال

برای افزودن یک گره، حداکثر به یک متوازن‌سازی نیاز خواهیم داشت که در زمان ثابت صورت می‌گیرد؛ بنابراین پیچیدگی زمانی برابر با $O(\log n)$ خواهد بود.

۳-۸-۲- حذف یک گره از درخت اِی-وی-ال

حذف گره از درخت اِی-وی-ال نیز مشابه با حذف گره از درخت جستجوی دودویی صورت می‌گیرد؛ به اضافه‌ی اینکه پس از حذف گره باید توازن درخت را بررسی کنیم. بخاطر دارید که برای حذف گره از درخت جستجوی دودویی با حالت‌های زیر روبرو بودیم:

۱- حذف گره‌ای بدون فرزند (برگ): در این حالت گره‌ی مورد نظر به سادگی حذف می‌شود.

۲- حذف گره‌ای با یک فرزند: در این حالت پس از حذف گره، فرزندش در مکان آن قرار خواهد گرفت.
۳- حذف گره‌ای با دو فرزند: در این حالت پس از حذف گره، چپی‌ترین فرزند زیردرخت راست گره و یا راستی‌ترین فرزند زیردرخت چپ گره، جایگزین آن خواهد شد.

پس از حذف گره برگ، توازن گره پدر این گره تا گره ریشه درخت اِی-وی-ال بررسی شده و عامل توازن همه‌ی گره‌هایی که در این مسیر قرار دارند، بروز می‌شوند. اگر عامل انشعاب یکی از گره‌ها تغییر نکرده باشد، پیمایش متوقف شده و نیازی به ادامه نیست؛ در غیر این صورت بررسی تا ریشه درخت ادامه پیدا می‌کند.
پیچیدگی زمانی حذف یک گره از درخت $O(\log n)$ و متوازن‌سازی کردن درخت در بدترین حالت $O(\log n)$ خواهد بود. در نتیجه پیچیدگی زمانی کل حذف یک گره از درخت، همان $O(\log n)$ است. جدول زیر پیچیدگی زمانی عملیات‌های اصلی برای درخت اِی-وی-ال را نشان می‌دهد.

جدول ۲,۳. پیچیدگی زمانی عملیات اصلی درخت اِی-وی-ال

عملیات	حالت میانگین	بدترین حالت
افزودن	$O(\log n)$	$O(\log n)$
حذف	$O(\log n)$	$O(\log n)$
جستجو	$O(\log n)$	$O(\log n)$
پیمایش	$O(n)$	$O(n)$

۳-۹- هرم

هرم یکی از ساختمان‌های داده‌ای مبتنی بر درخت است که در کاربردهای متفاوتی مانند مرتب‌سازی اعداد ابزاری مناسب محسوب می‌شود. برای تعریف این ساختمان داده‌ای ابتدا باید درخت کامل را تعریف کنیم.
درخت کامل درختی است که دارای قوانین زیر باشد:

۱- تا وقتی سطح کنونی درخت پر نشده باشد، قرار گرفتن گره در سطح‌های بعدی امکان‌پذیر نیست.
۲- گره‌ها از سمت چپ به راست به درخت افزوده می‌شوند. بنابراین در درخت کامل، وجود گره‌ای که فقط دارای فرزند راست باشد، ممکن نیست.

هرم یک درخت کامل است که دارای یکی از دو قانون زیر است:

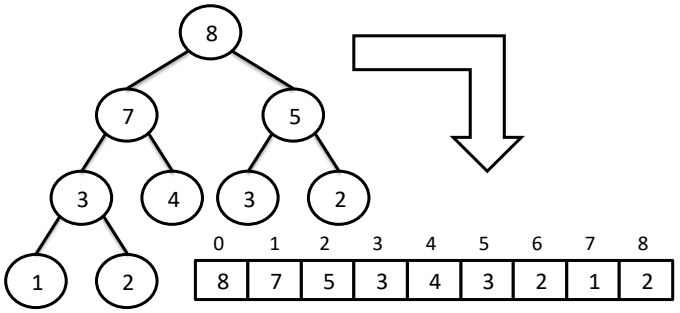
۱- مقدار درون هر گره از مقدار درون گره‌های فرزندانش بیشتر است.
۲- مقدار درون هر گره از مقدار درون گره‌های فرزندانش کمتر است.

اگر هرم دارای قانون نخست باشد، آن را هرم بیشینه و اگر از قانون دوم پیروی کند آن را هرم کمینه می‌نامند. هرم دارای کاربردهای بسیاری است که بعضی از آن‌ها عبارتند از:

۱- مرتب‌سازی هرمی که یکی از بهترین روش‌های مرتب‌سازی است.
۲- یافتن سریع کمترین/بیشترین مقدار در مجموعه‌ای از اعداد بدون نیاز به مرتب‌سازی کامل مجموعه.
۳- صف اولویت‌دار: در هرم بیشترین/کمترین مقدار در ریشه قرار دارد که در واقع، نشان‌دهنده‌ی بالاترین اولویت است. پس از حذف ریشه، فرزندی که بیشترین/کمترین مقدار را دارد، جایگزین ریشه خواهد شد.

در صف اولویت‌دار، تنها اولویت بعدی است که اهمیت دارد، نه وضعیت بقیه گره‌ها و این امر دقیقا وضعیتی است که در هرم با آن روبرو هستیم. الگوریتم‌های پرایم و دایجسترا با به‌کارگیری صف اولویت دار به خوبی قابل پیاده‌سازی هستند.

برای پیاده‌سازی هرم باید ابتدا چگونگی نگهداری عنصرهای درخت در یک آرایه‌ی یک‌بعدی را توضیح دهیم. برای این منظور، ریشه در خانه‌ی اول آرایه قرار می‌گیرد. دو خانه‌ی بعدی آرایه به فرزندان گره ریشه اختصاص می‌یابند؛ و به همین ترتیب مقادیر گره‌های درخت در آرایه قرار خواهند گرفت. با این تفسیر، اگر گره پدر در مکان i_ام از آرایه قرار گرفته باشد، فرزند چپ آن در مکان $2i$ و فرزند راست آن در مکان $2i+1$ قرار خواهد گرفت؛ البته این رابطه در صورتی درست است که نخستین خانه از آرایه دارای نمایه‌ی یک باشد. در حالتی که آرایه از نمایه‌ی صفرم آغاز شود، فرزند چپ در مکان $2i+1$ و فرزند راست در مکان $2i+2$ قرار خواهد داشت. برای نمونه، شکل ۸,۳ یک هرم و آرایه معادل با آن را نمایش می‌دهد. درخت نمایش داده شده در این شکل، یک هرم بیشینه است.



شکل ۸,۳. یک هرم بیشینه و آرایه معادل با آن

دو عمل اصلی برروی هرم صورت می‌گیرد:

۱- حذف بیشترین/کمترین مقدار از درخت (که در ریشه قرار دارد)

۲- افزودن یک گره جدید به درخت

پس از هر بار انجام اعمال اصلی، باید هرم بودن درخت را بررسی کنیم و اگر قوانین هرم برقرار نباشد، با انجام روندی مشخص درخت را مجددا به هرم تبدیل کنیم. این روند را در اصطلاح هرم‌سازی می‌نامند. هرم‌سازی به دو دسته تقسیم می‌شود که یکی در هنگام حذف گره از هرم و دیگری هنگام افزودن یک گره جدید به هرم به کار می‌رود. همه‌ی توضیحات این بخش درباره‌ی هرم بیشینه ارائه شده‌اند که به راحتی برای هرم کمینه نیز قابل تعمیم هستند.

۱- بالا هرمی: این فرآیند در هنگام افزودن گره جدید به هرم صورت می‌گیرد. چگونگی کار این فرآیند به این شکل است که مقدار گره پدرِ گره افزوده شده را با مقدار خود گره مقایسه کرده و اگر مقدار گره کنونی از مقدار گره پدر خود بیشتر باشد، مکان دو گره را عوض می‌کند. این فرآیند تا ریشه ادامه می‌یابد، تا وقتی که مقدار گره کنونی از مقدار گره پدر خود کمتر باشد. به علت این که گره جدید ابتدا به پایین هرم اضافه شده و تا جایی که نیاز باشد به سمت بالای هرم حرکت می‌کند، به این فرآیند بالا هرمی گفته می‌شود.

۲- پایین هرمی: این فرآیند عکس فرآیند بالا هرمی است و هنگام حذف گره از هرم به کار می‌رود؛ به این شکل که مقدار گره ریشه را با مقدار فرزندان آن مقایسه کرده و اگر مقدار آن از مقدار فرزندانش کمتر باشد، با فرزندی که دارای مقدار بیشتری است، جابه‌جا می‌شود. این روند تا زمانی که مقدار گره پدر از مقدار فرزندانش بیشتر باشد، ادامه پیدا خواهد کرد.

در اینجا دو فرآیند مذکور را با به‌کارگیری کلاس vector پیاده‌سازی می‌کنیم. البته با استفاده از آرایه نیز می‌توانیم پیاده‌سازی کنونی را انجام دهیم ولی به‌کارگیری بردار، کار را ساده‌تر خواهد کرد. پیاده‌سازی فرآیند بالا هرمی می‌تواند به این صورت انجام شود:

```
void upheap(vector<int>* heap, int index) {
    int child = (*heap)[index];
    int parent = 0;
    while (index > 0) {
        parent = (*heap)[(index - 1) / 2];
        if (child > parent)
            swap((*heap)[index], (*heap)[(index - 1) / 2]);
        else
            break;
        index /= 2;
    }
}
```

حال به پیاده‌سازی فرآیند پایین هرمی توجه کنید:

```
void downheap(vector<int>* heap) {
    int left_child = 0, right_child = 0;
    int index = 0;
    int parent = (*heap)[index];
    while (heap->size() > (index * 2 + 1)) {
        right_child = parent - 1;
        left_child = (*heap)[index * 2 + 1];
        if (heap->size()>(index * 2 + 2))
            right_child = (*heap)[index * 2 + 2];
        if (parent < left_child && parent < right_child) {
            if (left_child > right_child) {
                swap((*heap)[index], (*heap)[index * 2 + 1]);
                index = index * 2 + 1;
            }
            else {
                swap((*heap)[index], (*heap)[index * 2 + 2]);
                index = index * 2 + 2;
            }
        }
        else
            break;
    }
```

```
}
```

پیچیدگی زمانی دو فرآیند بالا هرمی و پایین هرمی در بدترین حالت $O(\log n)$ خواهد بود؛ زیرا در این دو تابع، عملیات نهایتا تا برگ درخت (برای فرآیند پایین هرمی) و یا تا ریشه درخت (برای فرآیند بالا هرمی) ادامه می‌یابد و عمق درخت حداکثر یک بار به طور کامل طی می‌شود.

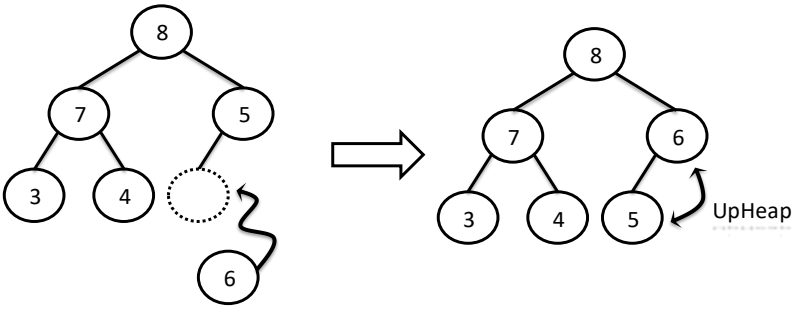
۳-۹-۱- افزودن یک گره به هرم

برای افزودن گره جدید به هرم، به این صورت عمل می‌کنیم.

۱- گره جدید را در نخستین مکان اشغال نشده قرار می‌دهیم (از چپ به راست).

۲- فرآیند upheap را برای گره افزوده شده فراخوانی می‌کنیم.

شکل ۹,۳ افزودن یک گره جدید به هرم و فرآیند بالا هرمی را نمایش می‌دهد.



شکل ۹,۳. نحوه افزودن یک گره جدید به یک هرم بیشینه و انجام فرآیند بالا هرمی

همان‌طور که در شکل ۹,۳ می‌بینید، فرآیند بالا هرمی فقط تا یک مرحله انجام شده، زیرا مقدار ۶ از مقدار گره پدر خود، یعنی ۸ کمتر است. کد بعدی افزودن یک گره به هرم را پیاده‌سازی می‌کند. پیچیدگی زمانی افزودن یک گره به هرم به علت انجام فرآیند بالاهرمی پس از افزودن گره، $O(\log n)$ خواهد بود.

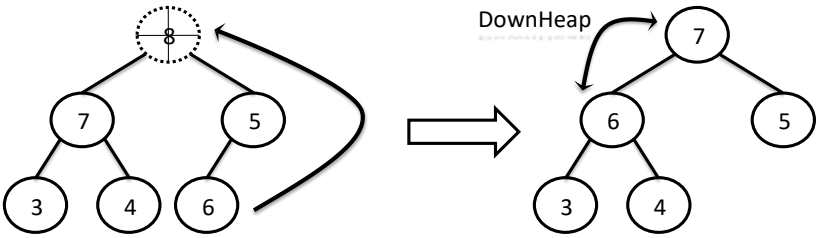
```
void insert_heap(vector<int>* heap, int element) {
    heap->push_back(element);
    upheap(heap, heap->size() - 1);
}
```

۳-۹-۲- حذف بیشترین مقدار از هرم

در یک هرم بیشینه بیشترین مقدار در ریشه درخت قرار دارد (چرا؟). برای حذف این مقدار این مراحل را دنبال می‌کنیم:

۱- گره ریشه را حذف می‌کنیم.

۲- سمت راست‌ترین گره از آخرین سطح هرم را جایگزین ریشه می‌کنیم.
۳- فرآیند پایین هرمی را برای ریشه فراخوانی می‌کنیم.
شکل ۱۰,۳ این مراحل را برای یک هرم بیشینه به تصویر کشیده است.



شکل ۱۰,۳. نحوه حذف گره ریشه از یک هرم بیشینه و انجام فرآیند پایین هرمی

در شکل ۱۰,۳، فرآیند پایین هرمی فقط تا یک مرحله انجام شده، زیرا مقدار ۶ از مقدار هر دو فرزندش بیشتر است.
کد بعدی گره ریشه را از هرم حذف کرده و مقدار آن را بازمی‌گرداند:

```
int remove_heap(vector<int>* heap) {
    if (heap->size() <= 0)
        return -1;
    int max_heap = (*heap)[0];
    (*heap)[0] = (*heap)[heap->size() - 1];
    heap->pop_back();
    if (heap->size() > 0) downheap(heap);
    return max_heap;
}
```

پیچیدگی زمانی حذف یک گره از هرم به علت انجام فرآیند پایین هرمی پس از حذف گره، برابر با $O(\log n)$ خواهد بود. جدول ۳,۳ پیچیدگی زمانی عملیات اصلی مربوط به هرم را نشان می‌دهد.

جدول ۳,۳. پیچیدگی زمانی عملیات اصلی مربوط به هرم

عملیات	پیچیدگی زمانی
یافتن بیشترین/کمترین مقدار در هرم بیشینه/هرم کمینه	$O(1)$
افزودن یک گره	$O(\log n)$
حذف یک گره	$O(\log n)$
ایجاد یک heap از n عنصر	$O(n)$

در جدول ۳,۳ ملاحظه می‌کنیم که پیچیدگی ایجاد یک هرم از n عنصر برابر با $O(n)$ ذکر شده است. در اینجا این سوال مطرح می‌شود که مگر برای اضافه شدن هر عنصر یک فرآیند بالا هرمی نیاز نیست؟ بنابراین باید پیچیدگی ایجاد یک هرم برابر با $O(n\log n)$ باشد! در پاسخ باید گفت که درست است که برای اضافه کردن هر عنصر باید یک بار فرآیند بالا هرمی انجام شود؛ ولی از آنجا که درخت در حال ساخته شدن است، هر بار کل ارتفاع درخت پیمایش نمی‌شود. در نتیجه، پیچیدگی بهتر از $O(n\log n)$ خواهد بود.

۳-۹-۳- ابزار معادل در کتابخانه الگوی استاندارد سی++

سه تابع برای کار با هرم در سرآیند algorithm قرار داده شده است.

۱- تابع make_heap: لیستی از اعداد را دریافت کرده و آن‌ها را به Heap تبدیل می‌کند.

۲- تابع push_heap: مقداری را به Heap می‌افزاید.

۳- تابع pop_heap: مقداری را از heap حذف می‌کند.

نمونه کد بعدی به روشنی عملکرد سه تابع معرفی شده را نشان می‌دهد. تابع pop_heap مقدار حذف شده را در پایان بردار قرار می‌دهد و باید برای حذف آن به صورت دستی عمل کرد.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main() {
    int myints[] = {10, 20, 30, 5, 15};
    vector<int> v(myints, myints + 5);
    vector<int>::iterator it;
    make_heap(v.begin(), v.end());
    cout << "initial max heap:";
    for (int i = 0; i < v.size(); i++)
        cout << " " << v[i];
    cout << endl;
    pop_heap(v.begin(), v.end());
    v.pop_back();
    cout << "max heap after pop:";
    for (int i = 0; i < v.size(); i++)
        cout << " " << v[i];
    cout << endl;
    v.push_back(99); push_heap(v.begin(), v.end());
    cout << "max heap after push:";
    for (int i = 0; i < v.size(); i++)
        cout << " " << v[i];
    cout << endl;
    return 0;
}
```

خروجی نمونه کد نشان داده شده به این صورت است:

```
Output
initial max heap   : 30 20 10 5 15
max heap after pop :  20 15 10 5
max heap after push :  99 20 10 5 15
```

۳-۱۰- صف اولویت

صف اولویت نوع خاصی از صف است؛ با این تفاوت که عضو سر صف همیشه طبق یک معیار مرتب‌سازی از سایر اعضای صف ارزشمندتر است. شباهت اصلی این ساختمان داده‌ای با صف این است که در صورت برابر بودن دو عنصر درون صف اولویت طبق معیار مرتب سازی، ترتیب ورود عناصر به صف مورد توجه قرار خواهد گرفت.

این ساختمان داده‌ای شباهت بسیاری به هرم دارد؛ زیرا در هر زمانی می‌توان عنصری به آن افزود؛ ولی در هر لحظه فقط عنصر بیشینه‌ی آن در دسترس است. در واقع بهترین ابزار برای پیاده‌سازی صف اولویت، ساختمان داده‌ای هرم است.

ساختمان داده‌ای صف اولویت در کتابخانه الگوی استاندارد توسط کلاس priority_queue پیاده‌سازی شده است. برای به کارگیری این کلاس باید سرآیند queue را include کنید.

کلاس priority_queue دارای توابع empty، size، top، push و pop است که در ادامه شرح داده می‌شود.

تابع‌های empty و size کاملا مشابه توابع هم‌نام خود در کلاس queue عمل می‌کنند. تابع push عنصری را به صف اولویت اضافه کرده و تابع pop عنصر سر صف را حذف می‌کند. لازم به ذکر است که تابع push پس از اضافه کردن عنصر جدید به صف اولویت، محل قرارگیری آن را با توجه به اولویتی که دارد مشخص می‌کند. تابع top نیز برای دسترسی به عنصر سر صف استفاده می‌شود. به مثال بعدی توجه کنید:

```cpp
#include <iostream>
#include <queue>
using namespace std;
int main() {
    priority_queue<int> mypq;
    mypq.push(30);
    mypq.push(100);
    mypq.push(25);
    mypq.push(40);
    cout << "Popping out elements..." << endl;
    while (!mypq.empty()) {
        cout << mypq.top() << endl;
        mypq.pop();
    }
    return 0;
}
```

خروجی این کد به این صورت خواهد بود:

```
Output
```

```
Popping out elements...
100
40
30
25
```

۳-۱۱- جدول درهم‌سازی

جدول درهم‌سازی ساختمان‌داده‌ای است که توسط یک تابع درهم‌سازی، هر کلید یکتایی را به یک مقدار یکتا نسبت می‌دهد. این ویژگی باعث می‌شود تا در در زمان ثابت ($O(1)$) امکان دسترسی به داده‌های جدول فراهم شود. این خصیصه، امتیاز بزرگی برای جدول درهم‌سازی نسبت به ساختمان‌داده‌های دیگری مانند درخت جستجوی دودویی و موارد مشابه آن است. جدول درهم‌سازی در واقع از یک بردار یا آرایه‌ای از داده‌ها به شکلی کاملا معمولی تشکیل می‌شود. نقش اصلی به عهده تابع درهم‌سازی است که یک کلید را به عنوان ورودی دریافت کرده و نمایه‌ای از آرایه را برمی‌گرداند؛ تا مقدار مربوط به آن کلید در آن نمایه از آرایه قرار گیرد.

بنابراین یک جدول درهم‌سازی به شکلی ساده از دو بخش زیر تشکیل می‌شود:

۱- آرایه‌ای از داده‌ها (معمولا با طول مشخص)

۲- یک تابع درهم‌سازی که هر کلید را به یک نمایه وابسته می‌کند.

بر اساس توضیحات داده شده، ساختار کلی یک تابع درهم‌سازی می‌تواند به این صورت باشد:

```
index = hash_function(key, arraylength);
```

طول آرایه نقش مهمی در عملکرد تابع درهم‌سازی ایفا می‌کند. برای نمونه این تابع درهم‌سازی ساده را درنظر بگیرید:

```
#define arraylength 8999
int hash_function(string key) {
    long sum = 0;
    for (int i = 0; i < key.length(); i++)
        sum = (sum << 3) + key[i];
    return sum % arraylength;
}
```

این تابع، هر کلید ورودی را به یک نمایه از ۰ تا ۸۹۹۸ نسبت می‌دهد. نتیجه‌ی به‌کارگیری این تابع برای ۲۰ کلمه از آغاز یک فرهنگ لغت در این فهرست نشان داده شده است:

```
AARON       8511
ABBIE       7942
ABBOTT      1131
ABBRA       8010
ABBY        2125
ABE         4757
ABEL        2136
ABENI       8178
ABIA        2157
```

ABIBA	8330
ABIE	2161
ABIGAIL	5399
ABNER	8691
ABRAHAM	3580
ABRAM	8910
ABRIANNA	4735
ABRIENDA	6703
ABRIL	8973
ABSOLOM	6213
ABU	4773

همانطور که مشخص است، نمایه‌ها از توزیع خوبی برخوردارند. در عمل معمولا یافتن یک تابع درهم‌سازی که هر کلید را فقط به یک نمایه نسبت دهد، بسیار دشوار است. برای نمونه، اگر ۲۵۰۰ کلید به همراه آرایه‌ای به طول ۱ میلیون در اختیار داشته باشیم، حتی با توزیع یکنواخت تصادفی این کلیدها در آرایه (با یک تابع درهم‌سازی مناسب)، بازهم به احتمال ۹۵٪، دست کم دو کلید وجود خواهند داشت که به یک نمایه نسبت داده می‌شوند (بر اساس نظریه تضاد روز تولد). تابعی که در اینجا معرفی شد نیز برای بعضی کلمه‌ها، نمایه‌ی یکسان تولید می‌کند. در بخش‌های بعد، توضیحات بیشتری در این رابطه ارائه خواهیم داد.

به یک جدول درهم‌سازی که دقیقا هر کلید را به یک نمایه نسبت دهد، جدول درهم‌سازی کامل گفته می‌شود. پس طبیعی است که بیشتر توابع درهم‌سازی چند کلید را به یک نمایه نسبت دهند؛ این عمل برخورد نامیده می‌شود. همیشه کوشش بریافتن تابع درهم‌سازی است که کمترین برخورد را تولید کند. در هر صورت، هر چقدر هم که تعداد برخوردهای تولید شده کم باشند، باید راه حلی برای مقابله با برخوردها درنظر گرفت. روش‌هایی برای این منظور ارائه شده است که روش‌های رفع برخورد نامیده می‌شوند و در ادامه به آن‌ها خواهیم پرداخت.

۳-۱۱-۱- عامل بارگیری

کارایی بیشتر روش‌های رفع برخورد به تنهایی به تعداد کلیدهای ذخیره شده وابسته نیست؛ بلکه ارتباط تنگاتنگی با عامل بارگیری جدول دارد. عامل بارگیری برابر است با نسبت تعداد کلیدهای ذخیره شده به طول آرایه:

$$LoadFactor = \frac{n}{s}$$

با داشتن یک تابع درهم‌سازی مناسب و عامل بارگیری میان ۰ تا ۰,۷، زمان دسترسی به داده‌ها ثابت ($O(1)$) خواهد بود. با افزایش عامل بارگیری، احتمال ایجاد برخورد بیشتر و در نتیجه سرعت دسترسی به داده‌ها کمتر و کمتر می‌شود. از سوی دیگر برای کاهش عامل بارگیری باید طول آرایه (s) را افزایش دهیم که این امر خود موجب هدر رفتن حافظه خواهد شد.

۳-۱۱-۲- روش‌های رفع برخورد

روش‌های رفع برخورد در واقع برخوردها را مدیریت می‌کنند؛ به شکلی که بتوان با وجود این برخوردها، بازهم به داده‌ها دستیابی پیدا کرد. در ادامه بعضی از روش‌های معمول رفع برخورد را می‌بینید که فقط تعدادی از آن‌ها را بطور کوتاه توضیح خواهیم داد. این روش‌ها عبارتند از:

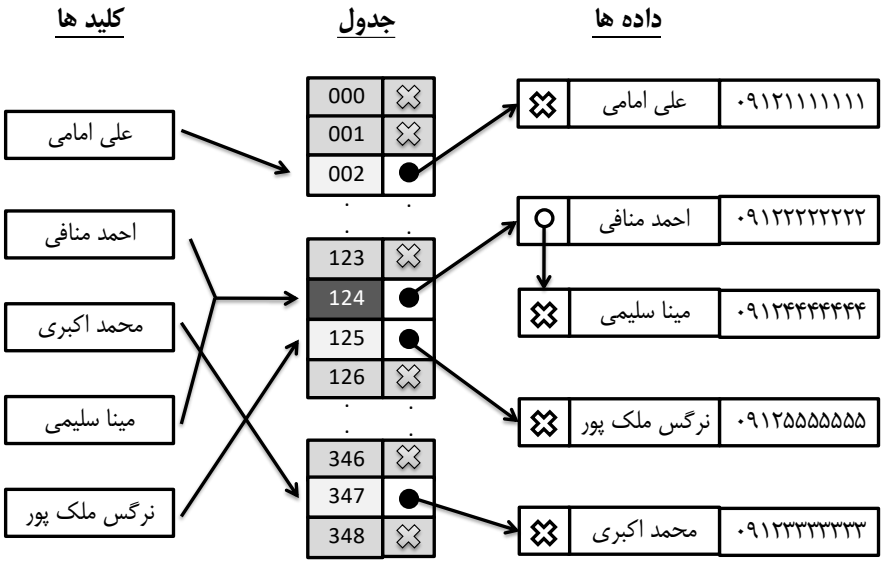
۱- زنجیرسازی
۲- آدرس‌دهی باز
۳- درهم‌سازی ترکیبی
۴- درهم‌سازی رابین‌هود
۵- درهم‌سازی فاخته

۳-۱۱-۲-۱- زنجیرسازی

در این روش معمولا لیست پیوندی برای نگهداری برخوردها به کار برده می‌شود. هر خانه از آرایه به آغاز لیست پیوندی اشاره می‌کند. برای یافتن داده‌ی مربوط به کلید مورد نظر باید لیست پیوندی را پیمایش کرده تا به داده‌ی مورد نظر دست یابیم. این روش، درهم‌سازی باز یا آدرس‌دهی بسته نیز نامیده می‌شود. اگر تعداد برخوردها در حد قابل قبولی باشد، سرعت دستیابی همچنان ثابت خواهد ماند؛ هرچند اگر تعداد کلیدها نسبت به طول آرایه خیلی بیشتر باشد، بازهم به‌کارگیری این روش موثر است. کارایی این روش با رشد عامل بارگیری به شکل قابل قبولی کاهش می‌یابد. برای نمونه جدول درهم‌سازی با آرایه‌ای به طول ۱۰۰۰ و ۱۰۰۰۰ کلید که زنجیرسازی را برای رفع برخورد به کار می‌برد (عامل بارگیری = ۱۰)، ۵ تا ۱۰ برابر کندتر از جدول درهم‌سازی با آرایه‌ای به طول ۱۰۰۰۰ (عامل بارگیری = ۱) است ولی باز هم ۱۰۰۰ برابر سریع‌تر از یک لیست ترتیبی معمولی خواهد بود؛ و حتی ممکن است از یک درخت جستجوی دودویی نیز سریع‌تر عمل کند. سرعت میانگین دستیابی به داده‌ها به تعداد میانگین کلیدها در یک خانه از آرایه بستگی دارد، که عامل بارگیری ارتباط مستقیمی با آن دارد.

در این روش، افزودن و حذف داده از جدول و جستجوی یک داده‌ی مشخص در جدول نیاز به پیمایش لیست پیوندی خواهد داشت.

شکل ۱۱,۳ نمایی از زنجیرسازی را به نمایش گذاشته است. معمولا داده‌ها در لیست پیوندی به ترتیب ورود نگهداری می‌شوند؛ البته اگر از پیش بدانیم که بعضی از داده‌ها بیشتر از سایر داده‌ها درخواست می‌شوند، به‌کارگیری لیست اولویت‌دار، سرعت را نسبت به لیست معمولی بیشتر خواهد کرد.



شکل ۱۱,۳. نمونه ای از درهم‌سازی با استفاده از زنجیرسازی

به‌کارگیری لیست پیوندی در این روش دو ایراد دارد که عبارتند از:

۱- اشاره‌گر به عنصر بعدی در لیست پیوندی خود باعث مصرف فضای اضافی می‌شود.

۲- پیمایش لیست پیوندی کارایی حافظه نهان را کاهش داده و پردازنده نمی‌تواند از حافظه نهان به شکل موثری در این رابطه بهره گیرد. زیرا عنصرهای لیست پیوندی در حافظه‌ی پیوسته‌ای قرار ندارند؛ بنابراین تعداد عدم اصابت در حافظه‌ی نهان افزایش پیدا خواهد کرد.

بدترین حالت برای این روش، زمانی است که همه کلیدها به یک نمایه نسبت داده شوند؛ در این حالت سرعت دستیابی به داده‌ها $O(n)$ خواهد بود که n تعداد کلیدهاست.

برای پیاده‌سازی زنجیرسازی، در آغاز یک ساختار برای لیست پیوندی مورد نیازمان معرفی می‌کنیم. در واقع جدول درهم‌سازی، آرایه‌ای از این ساختار خواهد بود. تعریف این ساختار و آرایه مورد نظر می‌تواند به این صورت باشد:

```
#define arraylength 8999
struct selement {
    selement* pointer;
    string key;
    int value;
}hash_table[hash_table[arraylength]];
```

حال دو تابع جدید به نام‌های hash_insert و hash_lookup را به ترتیب برای قرار دادن یک کلید در جدول و بازیابی آن از جدول به این صورت پیاده‌سازی می‌کنیم:

```
void hash_insert(string key, int value) {
    selement* element = new selement();
    int hash = hash_function(key);
    element->pointer = hash_table[hash].pointer;
    element->key = key;
    element->value = value;
    hash_table[hash].pointer = element;
}
```

```
int hash_lookup(string key) {
    int hash = hash_function(key);
    selement* next_element = hash_table[hash].pointer;
    while (next_element) {
        if (next_element->key == key)
            return (next_element->value);
        next_element = next_element->pointer;
    }
    return -1;
}
```

برای آزمایش جدول درهم‌سازی بالا به همراه توابعی که برای آن معرفی کردیم، به هر یک از کلمه‌های یک فرهنگ لغت ۴۶۱۸ کلمه‌ای، یک مقدار تصادفی به عنوان مقدار آن کلمه (خصیصه value) انتساب دادیم. همه‌ی کلمه‌ها را در جدول قرار داده و همه را یک‌به‌یک بازیابی کردیم. به صورت میانگین مقدار هر کلمه فقط با یک مرحله جستجو بازیابی شد. از این توضیحات می‌توان نتیجه گرفت که برای عامل‌های بارگیری زیر ۰٫۷ (عامل بارگیری این جدول برابر است با $\frac{4618}{8999} = 0.51$) سرعت دستیابی به اطلاعات جدول درهم‌سازی ثابت ($O(1)$) بوده و به‌کارگیری آن نسبت به سایر ساختمان‌داده‌ها موثرتر خواهد بود.

به جای به‌کارگیری لیست پیوندی می‌توان از ساختمان‌داده‌های دیگری نیز در این رابطه بهره برد. برای نمونه با به کارگیری درخت جستجوی دودویی می‌توان سرعت دستیابی به داده‌ها را در بدترین حالت به $O(\log n)$ کاهش داد؛ و یا می‌توان یک آرایه‌ی پویا را برای این منظور به کار برد. به این شکل که برای هر خانه از جدول، یک آرایه‌ی پویا در نظر گرفته و داده‌ها را به پایان آن می‌افزاییم. این روش باعث افزایش کارایی نسبت به استفاده از لیست پیوندی می‌شود زیرا حافظه‌ی پیوسته در آرایه‌ی پویا، به‌کارگیری بهینه‌تر پردازنده از حافظه نهان و در نتیجه افزایش سرعت دستیابی به داده‌ها را در برخواهد داشت.

۳-۱۱-۲-۲- آدرس‌دهی باز

این روش برای رفع برخورد، با یک روش جستجو از مکان کنونی (که توسط تابع درهم‌سازی تعیین شده است) آغاز به جستجو کرده و داده را در نخستین مکان خالی یافت شده قرار می‌دهد. برای جستجوی یک داده نیز باید به شکلی مشابه عمل کرده و از مکان تعیین شده توسط تابع درهم‌سازی آغاز به جستجو کنیم؛ تا زمانی که داده‌ی مورد نظر را یافته و یا به یک خانه‌ی خالی از جدول برسیم که عدم وجود داده‌ی مورد نظر را تایید می‌کند.

از آنجا که تابع درهم‌سازی مکان دقیق داده در جدول را مشخص نمی‌کند، این روش را آدرس‌دهی باز می‌نامند. در ادامه به روش‌های جستجوی مطرح که برای یافتن نخستین خانه‌ی خالی در جدول به کار برده می‌شوند، اشاره خواهیم کرد.

۱- کاوش خطی: در این روش یک گام ثابت برای جستجو به کار برده می‌شود (معمولا ۱).

۲- کاوش درجه دوم: گام جستجو در این روش، در هر مرحله به اندازه ثابتی افزایش می‌یابد (معمولا ۱).

۳- درهم‌سازی دوباره: گام جستجو توسط یک تابع درهم‌سازی دیگر محاسبه می‌شود.

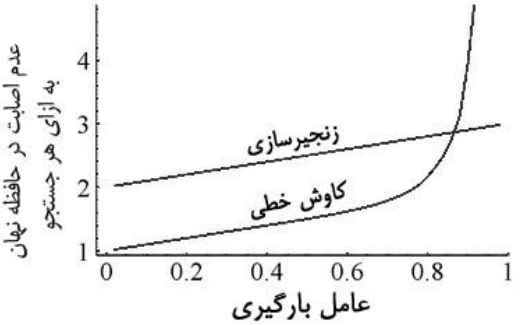
از توضیحات داده شده به روشنی می‌توان نتیجه گرفت که تعداد داده‌ها نمی‌تواند از تعداد خانه‌های جدول بیشتر شود. این امر باعث می‌شود که حتی با به‌کارگیری یک تابع درهم‌سازی مناسب نیز کارایی برای عاملهای بارگیری بیشتر از ۰,۷ بطور قابل توجهی کاهش یابد. این محدودیت موجب می‌شود تا بسیاری از نرم‌افزارها از روش تغییر اندازه‌ی پویا استفاده کنند.

آدرس‌دهی باز دارای این مزایا است:

۱- از حافظه‌ی اضافی استفاده نکرده و نیازی به تخصیص‌دهنده‌ی حافظه ندارد.

۲- موجب افزایش کارایی پردازنده در به‌کارگیری حافظه‌ی نهان شده و سرعت را بهبود می‌بخشد.

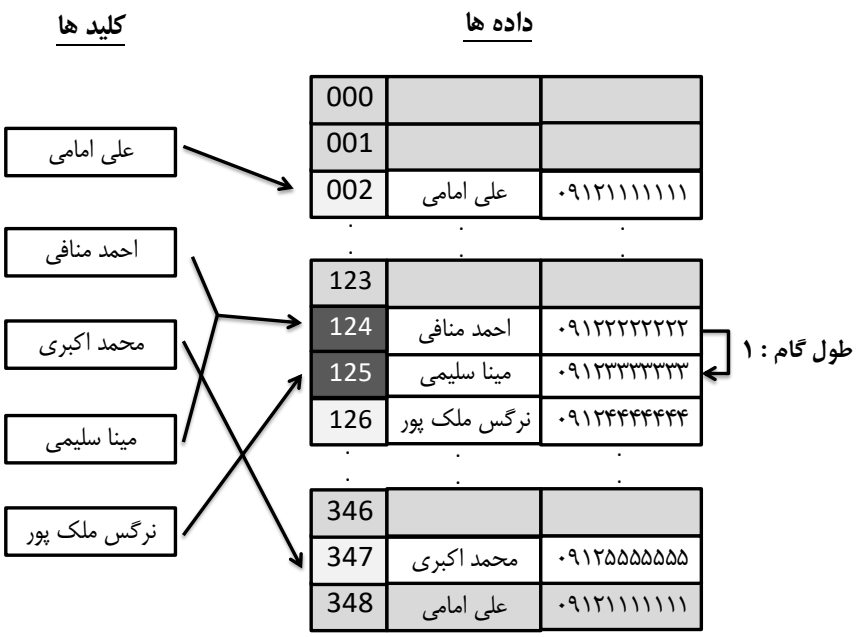
اگر تعداد برخوردها افزایش یابد، کارایی این روش شدیدا کاهش خواهد یافت؛ زیرا تعداد عدم اصابت در حافظه‌ی نهان افزایش پیدا خواهد کرد. مقایسه‌ای بین عملکرد روش‌های زنجیرسازی و کاوش خطی در شکل ۱۲,۳ نشان داده شده است.



شکل ۱۲,۳. مقایسه عملکرد روش‌های زنجیرسازی و کاوش خطی

با توجه به مطالب گفته شده می‌توان نتیجه گرفت که به‌کارگیری روش آدرس‌دهی باز برای عامل بارگیری زیر ۰,۷ مناسب است. برای بازدهی بیشتر، بهتر است این روش را برای داده‌هایی با اندازه‌ی کوچکتر به کار گرفت؛ زیرا در این صورت عدم اصابت در حافظه نهان کمتر اتفاق می‌افتد. برای داده‌های بزرگتر، زنجیرسازی گزینه‌ی مناسب‌تری است.

سرعت دستیابی به داده‌ها برای این روش نیز $O(n)$ است. شکل ۱۳,۳ نمونه‌ای از رفع برخورد به روش کاوش خطی (طول گام=۱) در آدرس‌دهی باز را نشان می‌دهد.



شکل ۱۳,۳. نمونه‌ای از رفع برخورد به روش کاوش خطی (طول گام=۱) در آدرس‌دهی باز

در شکل ۱۳,۳ دقت داشته باشید که کلید «نرگس ملک پور» به نمایه‌ی ۱۲۵ از جدول نگاشت می‌شود ولی چون این خانه از پیش اشغال شده است، با به‌کارگیری روش کاوش خطی و با طول گام ۱، به دنبال نخستین خانه‌ی خالی بعدی گشته و آن را در نمایه‌ی ۱۲۶ جدول قرار می‌دهیم.

۳-۱۱-۲-۳- درهم‌سازی ترکیبی

این روش، ترکیب زنجیرسازی و آدرس‌دهی باز است. مانند روش آدرس‌دهی باز، داده‌ها در خانه‌های جدول قرار می‌گیرند و هنگام بروز برخورد، داده در نخستین خانه‌ی بعدی قرار داده می‌شود؛ با این تفاوت که خانه‌ی اصلی به این خانه اشاره خواهد کرد. به‌کارگیری این روش باعث می‌شود تا زمان جستجوی داده‌ها در زمانی که عامل بارگیری بیش از حد معمول است، سریع‌تر انجام شود. مزیت‌های این روش، ترکیبی از مزایای هر دو روش زنجیرسازی و آدرس‌دهی باز است که عبارتند از:

۱- به‌کارگیری مناسب فضای جدول

۲- استفاده از مزایای حافظه نهان

۳- پر شدن مناسب و پرتراکم جدول

از سوی دیگر، مانند روش آدرس‌دهی باز، نمی‌توان تعداد داده‌هایی بیشتر از تعداد خانه‌های جدول را ذخیره کرد. سرعت دستیابی در بدترین حالت برای این روش مانند دو روش پیشین خواهد بود.

۳-۱۱-۲-٤- درهم‌سازی رابین‌هود

این روش یکی از انواع درهم‌سازی دوباره (آدرس‌دهی باز) است که اساس آن، تعداد دفعات جستجو شدن هر داده است. به این شکل که برای هر داده یک شمارنده در نظر گرفته و هنگام بروز برخورد، داده‌ای که دارای تعداد دفعات جستجوی کمتری است به مکان دیگری منتقل می‌شود. این معیار باعث می‌شود زمان دستیابی در بدترین حالت بهبود یافته و بطور کلی تعداد گام‌هایی که برای جستجو برداشته می‌شوند، کاهش یابد؛ زمان دستیابی به داده‌ها در بدترین حالت $O(n)$ خواهد بود.

۳-۱۱-۲-٥- درهم‌سازی فاخته

فاخته نام پرنده‌ای است که بعضی از انواع آن رفتار عجیبی دارند. برای نمونه جوجه‌های فاخته، پس از تولد، بقیه تخم‌ها را هل داده و از خود دور می‌کنند و یا پس از تولد از آشیانه خارج می‌شوند. ایده‌ی اصلی در این روش، به کارگیری دو تابع درهم‌سازی است. بنابراین برای هر کلید، دو مکان وجود داشته و در زمانی که برخورد به وجود می‌آید، یکی از داده‌ها به مکان دوم خود انتقال داده می‌شود. برای دستیابی به یک داده‌ی مشخص، حداکثر باید دو خانه بررسی شوند. این موضوع، مزیت اصلی این روش نسبت به سایر روش‌هاست؛ زیرا موجب می‌شود سرعت دستیابی در بدترین حالت برای درهم‌سازی فاخته ثابت ($O(1)$) باشد. بطور کلی مزایای این روش را می‌توان این موارد دانست:

۱- زمان دستیابی ثابت در بدترین حالت

۲- به‌کارگیری بهینه‌تر فضای جدول با به‌کارگیری دو تابع درهم‌ساز

جدول ۴,۳ سرعت دستیابی همه‌ی روش‌های رفع برخورد را در بدترین حالت نمایش می‌دهد.

جدول ٤,۳. مقایسه سرعت دستیابی روش‌های مختلف رفع برخورد در بدترین حالت

روش رفع برخورد	سرعت دستیابی به داده‌ها در بدترین حالت
زنجیره‌ای	$O(n)$
آدرس‌دهی باز	$O(n)$
درهم‌سازی ترکیبی	$O(n)$
درهم‌سازی رابین‌هود	$O(n)$
درهم‌سازی فاخته	$O(1)$

۳-۱۱-۳- تغییر اندازه‌ی پویا

تغییر اندازه‌ی پویای جدول، راه‌حلی است که برای پایین نگه داشتن عامل بارگیری به کار برده می‌شود. عامل بارگیری میان ۰,۲۵ تا ۰,۷۵ محدوده‌ی مناسبی برای یک جدول درهم‌سازی است. در بعضی از پیاده‌سازی‌های جدول درهم‌سازی، وقتی عامل بارگیری از یک آستانه‌ی مشخص عبور می‌کند، طول جدول به صورت پویا افزایش یافته تا عامل بارگیری را دوباره به محدوده‌ی قابل قبولی بازگرداند؛ و با کاهش داده‌ها، دوباره به صورت پویا اندازه‌ی خود را کاهش می‌دهد. این روش را تغییر اندازه‌ی پویا می‌نامند. برای نمونه کلاس HashMap در زبان جاوا دارای آستانه ۰,۷۵ برای گسترش طول جدول است. برای گسترش جدول معمولا دو روش به کار برده می‌شود:

۱- تغییر اندازه‌ی کامل
۲- تغییر اندازه‌ی تدریجی

۳-۱۱-۳-۱- تغییر اندازه‌ی کامل

در روش تغییر اندازه‌ی کامل هنگامی که عامل بارگیری از آستانه‌ی درنظر گرفته شده تجاوز کرد، درخواست حافظه‌ای با طول بیشتر کرده و همه‌ی داده‌های جدول پیشین را به جدول جدید منتقل می‌کنیم؛ سپس حافظه‌ی جدول پیشین را آزاد خواهیم کرد. مراحل انجام این کار در ادامه فهرست شده است:

۱- درخواست حافظه‌ای با طول بیشتر برای جدول جدید
۲- انتقال داده‌های جدول پیشین به جدول جدید
۳- آزادسازی فضای اشغالی توسط جدول پیشین

۳-۱۱-۳-۲- تغییر اندازه‌ی تدریجی

در بعضی از پیاده‌سازی‌ها، مخصوصا در سیستم‌های بلادرنگ، زمان لازم برای انتقال داده‌ها به صورت یکجا را در اختیار نداریم. در این مواقع می‌توانیم به تدریج داده‌ها را منتقل کنیم. مراحل کار در این روش به صورت زیر خواهد بود:

۱- درخواست حافظه‌ای با طول بیشتر برای جدول جدید
۲- انتقال تدریجی داده‌ها در هر بار مراجعه به جدول پیشین. در هر مراجعه، k داده منتقل می‌شوند.
۳- داده‌های جدید فقط به جدول جدید افزوده می‌شوند.
۴- درخواست‌های حذف یا دستیابی به داده‌ها در هر دو جدول بررسی می‌شود.
۵- زمانی که همه‌ی داده‌های جدول پیشین حذف و به جدول جدید انتقال پیدا کردند، فضای مربوط به جدول پیشین آزادسازی می‌شود.

۳-۱۱-٤- برگزیدن تابع درهم‌سازی مناسب

برگزیدن یک تابع درهم‌سازی مناسب به تنهایی برای داشتن یک جدول درهم‌سازی ایده‌آل کافی است. برگزیدن بی‌دقت تابع درهم‌سازی و تمرکز روی روش‌های رفع برخورد نمی‌تواند راه‌حل مناسبی برای دستیابی به جدول درهم‌سازی ایده‌آل باشد.

یکی از مهم‌ترین ویژگی‌های یک تابع درهم‌سازی مناسب، توزیع یکنواخت کلیدها در جدول است. توزیع غیریکنواخت کلیدها، موجب افزایش برخورد در جدول و در نتیجه افزایش هزینه برای رفع برخورد خواهد شد.

به‌کارگیری جدول درهم‌سازی مزایایی دارد که باید با بررسی شرایط مسأله، زمان مناسب به‌کارگیری آن را تشخیص داد. برخی از مزایای این ساختمان‌داده به قرار زیر می‌باشند:

۱- دسترسی به داده‌ها در زمان ثابت، که در هیچ ساختمان‌داده‌ی دیگری بطور کامل قابل حصول نیست.
۲- به‌کارگیری در پیاده‌سازی آرایه‌های انجمنی، شاخص‌گذاری پایگاه داده‌ای، حافظه‌ی نهان و مجموعه‌ها.
۳- جلوگیری از ورود داده‌های تکراری از کاربردهای دیگری است که تابع درهم‌سازی فراهم می‌آورد.

۳-۱۱-۵- ابزار معادل در سی++

کلاس hash_map در زبان سی++ جدول درهم‌سازی را پیاده‌سازی می‌کند. این کلاس، از مجموعه‌ی کتابخانه‌ی استاندارد حذف شده است؛ بنابراین فقط به اشاره‌ای کلی درباره‌ی آن بسنده می‌کنیم.
به‌کارگیری این کلاس کاملا مشابه با کلاس map است. به اضافه‌ی اینکه، امکان دریافت یک تابع درهم‌سازی را نیز دارد. بطور کلی، ساختار تعریف آن به این صورت است:

```
hash_map<Key, Data, HashFcn, EqualKey, Alloc>;
```

پارامترهای الگوی این کلاس در ادامه توضیح داده شده‌اند.

۱- Key: نوع کلیدهای جدول را مشخص می‌کند.

۲- Data: نوع داده‌هایی که در جدول قرار خواهند گرفت را مشخص می‌کند.

۳- HashFcn: یک تابع درهم‌سازی که پارامتر ورودی آن از نوع کلیدهای جدول و خروجی آن نمایه‌ای متناسب با کلید ورودی است.

۴- EqualKey: کلاسی است که برای مشخص کردن یکسانی دو کلید به کار می‌رود. معمولا باید عملگر () را با دو پارامتر ورودی از نوع کلیدهای جدول و خروجی از نوع bool پیاده‌سازی کرد.

۵- Alloc: کلاسی که نحوه‌ی مدیریت حافظه‌ی درونی را مشخص می‌کند.

تنها تفاوتی که در ساختار تعریف این کلاس و کلاس map وجود دارد، امکان معرفی تابع درهم‌سازی است. برای به‌کارگیری این کلاس به سرآیند hash_map نیاز خواهید داشت.

۳-۱۲- مجموعه‌های از هم جدا

ساختمان‌داده‌ی مجموعه‌های جدا، برای نگهداری عنصرها در قالب مجموعه‌های متفاوت است. برای نمونه فرض کنید در برنامه‌ای تعدادی درخت دارید که هریک شامل چند گره است و در این برنامه نیازمند به انجام این کارها هستید:

۱- تشخیص درخت مربوط به هر گره.

۲- یکسان بودن درخت‌های مربوط به دو گره.

۳- اجتماع دو درخت با یکدیگر.

توسط مجموعه‌های از هم جدا می‌توانید به راحتی این کارها را انجام دهید. اساس کار این ساختمان‌داده نگهداری درخت در قالب لیست‌پیوندی یا آرایه‌ها است. یکی کاربردهای این ساختمان‌داده در پیاده‌سازی الگوریتم کروسکال است (فصل ۴).
دو عمل اصلی در این ساختمان‌داده عبارتند از:

۱- یافتن ریشه‌ی مربوط به هر عنصر.

۲- اجتماع دو مجموعه؛ برای این منظور، ابتدا ریشه‌های دو مجموعه جستجو شده و سپس یکی از ریشه‌ها به دیگری متصل می‌شود.

در آغاز کار، هر عنصر یک مجموعه‌ی مستقل است. عمل ایجاد مجموعه یکی دیگر از تابع‌های این ساختمان‌داده است که یک عنصر جدید را در قالب یک مجموعه به ساختمان‌داده اضافه می‌کند. در ادامه واژه‌ی درخت را برای اشاره به هر مجموعه به کار می‌بریم. ریشه‌ی هر درخت نماینده آن بوده و برای مقایسه دو درخت به کار گرفته می‌شود. هر عنصر دارای یک والد است و در آغاز والد هر عنصری خودش خواهد بود. با آغاز از هر گره و پیمایش والدهای آن در پایان به ریشه‌ی درخت می‌رسیم.
شبه‌کدهای زیر روش کار این سه عمل را نشان می‌دهند:

```
makeset(x) {
    x.parent = x
}
```

```
find(x) {
    if (x.parent == x)
        return x
    else
        return find(x.parent)
}
```

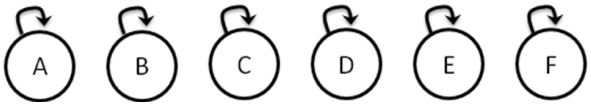
تابع find مقدار ریشه‌ی درخت یافت شده را برمی‌گرداند. تابع union اگر دو گره از دو درخت متفاوت باشند، آن‌ها را به هم متصل کرده و مقدار true را برمی‌گرداند و در غیراین صورت مقدار false برگردانده می‌شود.

```
union(x, y) {
    xRoot = find(x)
    yRoot = find(y)
    if (xRoot != yRoot) {
        xRoot.parent = yRoot
        return true
    }
    return false
}
```

برای به‌کارگیری این ساختمان‌داده، ابتدا عناصر مورد نظر را توسط تابع makeset به آن اضافه می‌کنیم. سپس با به کارگیری دو تابع find و union کارهای مورد نظرمان را انجام می‌دهیم. برای نمونه عنصرهای نشان داده شده را در نظر بگیرید:

$$\{A, B, C, D, E, F\}$$

پس از فراخوانی تابع makeset برای هریک از عنصرهای نشان داده شده، ساختمان داده حاصل مانند شکل ۱۴,۳ خواهد بود:



شکل ۱۴,۳. ساختمان داده حاصل پس از فراخوانی تابع makeset برای عنصرهای $\{A, B, C, D, E, F\}$

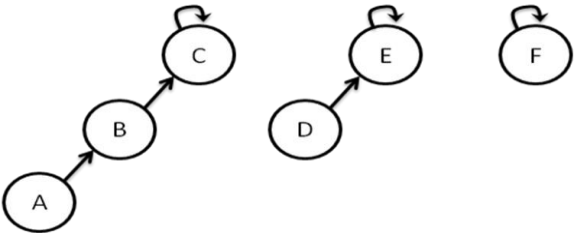
فرض کنید در مرحله بعد تابع union را به این ترتیب فراخوانی کنیم:

$union(A, B)$

$union(D, E)$

$union(B, C)$

نتیجه این فراخوانی‌ها در شکل ۱۵,۳ نشان داده شده است. در این حالت، مقدار بازگشتی فراخوانی find(B) مقدار C و مقدار بازگشتی فراخوانی find(D) مقدار E است (چرا؟).



شکل ۱۵,۳. مجموعه‌های به دست آمده پس از انجام چند عملیات اجتماع بر مجموعه‌های شکل ۱۴,۳

۳-۱۲-۱- پیاده‌سازی مجموعه‌های از هم جدا

برای پیاده‌سازی مجموعه‌های از هم جدا، از آرایه‌ها استفاده می‌کنیم. برای آغاز، آرایه‌ای به طول مشخص درنظر گرفته و فرض می‌کنیم هر خانه‌ی آن عنصری مستقل یا در واقع درختی مستقل است. در خانه‌ی i_ام آرایه، نمایه‌ی والد عنصر i_ام قرار می‌گیرد. این نمونه کد نحوه پیاده‌سازی این ساختمان داده را به سادگی نشان می‌دهد:

```
#define size 100
int fu_set[size];
void makeset(int index) {
    fu_set[index] = index;
}
int find(int index) {
    while (fu_set[index] != index)
        index = fu_set[index];
    return index;
}
bool union_set(int index1, int index2) {
    int root1 = find(index1);
    int root2 = find(index2);
    if (root1 != root2) {
        fu_set[root1] = root2;
        return true;
    }
    return false;
}
```

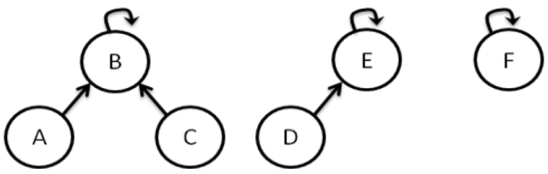
برای آزمایش کد نشان داده شده، می توانید این کد را به کار ببرید:

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 10; i++)
        makeset(i);
    union_set(1, 2);
    union_set(2, 3);
    union_set(7, 8);
    union_set(8, 9);
    union_set(1, 8);
    cout << find(2) << " - " << find(8) << endl;
    return 0;
}
```

خروجی این کد به این صورت خواهد بود:

```
Output
9-9
```

در بدترین حالت، درخت در یک جهت رشد کرده و این امر موجب می‌شود پیچیدگی زمانی تابع find در بدترین حالت از درجه‌ی خطی $O(n)$ باشد. برای بهبود بخشیدن به سرعت اجرای این تابع، می‌توانیم عملکرد تابع union_set را کمی بهینه‌تر کنیم. برای این منظور، درختی که دارای عمق کمتری است را به درخت با عمق بیشتر متصل می‌کنیم؛ این بهینه‌سازی موجب می‌شود عمق درخت حاصل تغییر نکند. در این حالت مجموعه‌های شکل ۱۵,۳ به صورت نشان داده شده در شکل ۱۶,۳ در خواهند آمد.



شکل ۱۶,۳. مجموعه‌های به دست آمده پس از انجام چند عملیات اجتماع بر مجموعه‌های شکل ۱۴,۳ با بهبود عملیات اجتماع

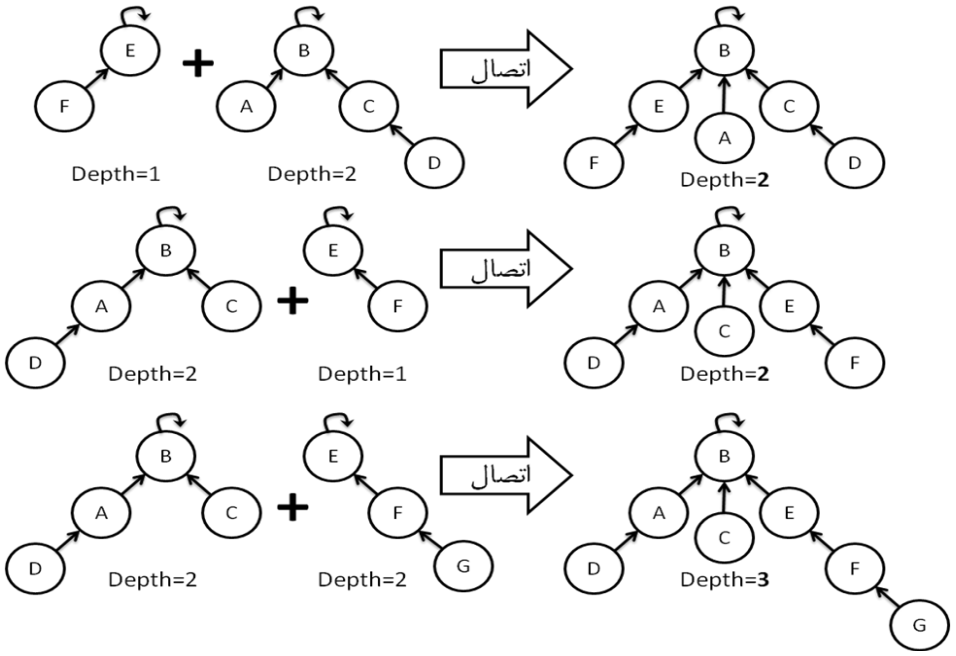
در روش پیشنهادی، برای اجتماع دو درخت سه حالت زیر پیش می‌آید.

۱- عمق درخت اول بیشتر از درخت دوم است.

۲- عمق درخت اول کمتر از درخت دوم است.

۳- عمق درخت اول برابر با درخت دوم است.

در حالت اول و دوم، درختی که دارای عمق کمتر است به دیگری متصل می‌شود؛ در این صورت عمق درخت حاصل ثابت می‌ماند؛ در حالت سوم، پس از اتصال یکی از درخت‌ها (به صورت دلخواه) به دیگری، عمق درخت حاصل یکی افزایش پیدا می‌کند. این سه حالت در شکل ۱۷٫۳ نشان داده شده‌اند.



شکل ۱۷٫۳. حالت‌های مختلفی که ممکن است موقع اجتماع دو مجموعه به وجود بیاید.

برای پیاده‌سازی راه‌حل پیشنهادی، این ساختار را برای هر گره به کار می‌بریم. ساختار جدید از دو بخش والد و عمق تشکیل شده است.

```
struct node {
    int parent;
    int depth;
};
```

حال سه تابع معرفی شده را می‌توانیم به صورت نشان داده شده در مثال بعدی تغییر دهیم. تابع makeset عمق درخت‌های تک عنصری را صفر قرار می‌دهد. در این حالت پیچیدگی زمانی تابع find به درجه‌ی لگاریتمی $O(\log n)$ کاهش پیدا می‌کند. حالت بهبود یافته‌ی این ساختمان‌داده را در فصل ۴ برای پیاده‌سازی الگوریتم کروسکال به کار خواهیم برد.

```
#define size 100
node fu_set[size];
void makeset(int index) {
    fu_set[index].parent = index;
    fu_set[index].depth = 0;
}
int find(int index) {
    while (fu_set[index].parent != index)
        index = fu_set[index].parent;
    return index;
}
bool union_set(int index1, int index2) {
    int root1 = find(index1);
    int root2 = find(index2);
    if (root1 != root2) {
        if (fu_set[root1].depth < fu_set[root2].depth)
            fu_set[root1].parent = root2;
        else if (fu_set[root1].depth > fu_set[root2].depth)
            fu_set[root2].parent = root1;
        else {
            fu_set[root1].parent = root2;
            fu_set[root2].depth++;
        }
        return true;
    }
    return false;
}
```

تمرین‌ها

۱- برنامه‌ای بنویسید که یک عدد از ورودی دریافت کرده و بسته به مقدار آن یکی از عمل‌های زیر را انجام دهد:

- عددی غیر از صفر و ۱-: عدد در حافظه‌ی مشخصی (باید کلاسی مناسب را برای ذخیره‌سازی اعداد انتخاب کنید) ذخیره شود و دوباره منتظر دریافت عددی دیگر شود.
- عدد صفر: اعداد ذخیره شده تا این لحظه به ترتیب عکسِ وارد شدن، در خروجی چاپ و از حافظه‌ی مورد نظر حذف شوند.
- عدد ۱-: پایان برنامه

Input	Output
1 2 3 0 4 6 0 -1	3 2 1 6 4

۲- برنامه‌ای بنویسید که دو مجموعه از اعداد را به صورت زیر دریافت کند. سپس آن‌ها را به شکلی در هم ادغام کند که مجموعه‌ی حاصل دارای هیچ عضو تکراری نباشد. آنگاه مجموعه‌ی بدست آمده را به صورت صعودی چاپ کنید.

- برای هر مجموعه، در ابتدا یک عدد n در ورودی می‌آید که مشخص‌کننده‌ی طول مجموعه است و سپس n عدد آورده می‌شود.

Input	Output
5 1 2 3 2 5 6 5 2 1 1 6 3	1 2 3 5 6

۳- برنامه‌ای بنویسید که بازی زیر را پیاده‌سازی کند:

- این بازی دارای قواعد زیادی است که بازیکن به ازای داشتن یک یا چند کارت مشخص، یک یا چند کارت جدید جایزه می‌گیرد. برای نمونه یک قاعده می‌تواند این باشد که اگر شما کارت شماره یک و سه را داشته باشید، کارت‌های شماره پنج و هفت به شما داده می‌شود.
- دقت کنید که داشتن چند کارت با شماره یکسان با یک کارت از آن شماره تفاوتی نمی‌کند.
- بازی وقتی به پایان می‌رسد که بازیکن دیگر نتواند با قاعده‌های بازی، کارت جدیدی جایزه بگیرد.
- در سطر اول ورودی، عدد n که نشان‌دهنده‌ی تعداد قاعده‌های بازی است، می‌آید. در $2n$ سطر بعد، هر یک از n قانون بازی در دو سطر آورده شده‌اند. به این ترتیب که در سطر اول یک عدد x و سپس x عدد می‌آید و در سطر دوم یک عدد y و سپس y عدد آورده می‌شود که نشان می‌دهد اگر x کارت اول را داشته باشیم، y کارت دوم به ما جایزه داده می‌شود. در ادامه یک عدد k و سپس k عدد می‌آید که کارت‌های آغازین بازیکن برای شروع بازی است. شماره‌ی همه‌ی کارت‌ها، یک عدد در بازه ۱ تا ۵۰ است.
- خروجی برنامه شامل دو سطر است: در سطر اول، بیشترین تعداد کارت متفاوتی که بازیکن می‌تواند به دست آورد و در سطر دوم شماره‌ی آن کارت‌ها به ترتیب صعودی چاپ می‌شود.

Input	Output
3 1 2 2 3 4 1 3 1 1 2 1 4 2 5 6 1 2	6 1 2 3 4 5 6

۴- برنامه‌ای بنویسید که مسیر یک پرونده ورودی را دریافت کرده و تعداد کلمه‌های موجود در آن را بشمارد. سپس در هر خط از خروجی، کلمه‌ها و تعداد تکرار آن‌ها در پرونده ورودی را چاپ کند. دقت کنید که عملیات شمارش باید با یک بار پیمایش پرونده انجام شود.

مراجع

Arefin, A.S., 2006. *Art of Programming Contest*, Reviewed by Dr. M. Lutfar Rahman and Forworded by Professor Miguel Revilla, University de Valladolid, Spain, Gyankosh Prokashoni, Dhaka, Bangladesh, First Ed., ISBN: 984-32-3382-4.

Cormen, T.H. et al., 2009. *Introduction to algorithms* 3rd ed., MIT Press and McGraw-Hill.

Horowitz, E., Sahni, S. & Mehta, D., 2006. *Fundamentals of data structures in C++* 2nd ed., Silicon Pr.

Skiena, S.S. & Revilla, M.A., 2006. *Programming challenges: The programming contest training manual*, Springer Science & Business Media.

فصل ٤- الگوریتم‌های کاربردی

در فصل‌های گذشته با ساختمان‌داده‌ها و ابزارهای ورودی و خروجی به تفصیل آشنا شدیم. در این فصل با به کارگیری آموخته‌های کسب شده در فصل‌های پیشین، به تحلیل و پیاده‌سازی الگوریتم‌های مختلف پرداخته می‌شود. الگوریتم‌های ارائه شده به صورت کامل توضیح داده شده و با شبه‌کدهای سی++ پیاده‌سازی خواهند شد. در موارد لزوم، بهینه‌سازی‌هایی را نیز بر روی برخی الگوریتم‌ها به کار بسته و نتیجه آن را بررسی می‌کنیم.

در آغاز تعریفی ساده از الگوریتم ارائه می‌دهیم:

ارائه یک روش که به صورت گام به گام و مرحله به مرحله به حل مسأله می‌پردازد.

در این فصل، پیاده‌سازی الگوریتم‌ها را با به‌کارگیری شبه‌کدها انجام می‌دهیم ولی در فصل بعد در قالب پرسش‌های متنوع، به زبان سی++، آن‌ها را پیاده‌سازی خواهیم کرد. شبه‌کدهای به کار رفته نیز تا آنجا که بشود با زبان سی++ مطابقت خواهند داشت؛ با این تفاوت که فقط بدنه اصلی الگوریتم توضیح داده شده و از برخی از جزییات نسبت به کد واقعی صرف نظر می‌شود.
پیش از ارائه شبه‌کد برای هر مسأله، صورت مسأله، ورودی و خروجی آن را تعیین می‌کنیم تا کاربرد الگوریتم مربوطه کاملا مشخص شود.

٤-۱- تحلیل مرتبه الگوریتم‌ها

برای مسائل مختلف و حتی یک مسأله، الگوریتم‌های متفاوتی ارائه می‌شود و ما در پایان به مقایسه الگوریتم‌ها و سرعت و کارایی آن‌ها نیاز خواهیم داشت؛ بنابراین باید روشی فراگیر و اصولی را برای مقایسه الگوریتم‌ها به کار ببریم. روشی که از آن صحبت می‌کنیم نباید وابسته به رایانه‌ای که الگوریتم بر روی آن اجرا می‌شود، زبان برنامه نویسی که برای پیاده‌سازی به کار گرفته می‌شود، خود برنامه‌نویس و در کل هر جزییاتی که عادلانه بودن مقایسه را زیر سوال می‌برد، باشد. برای نمونه نمی‌توانیم تعداد دستورهای برنامه را به عنوان یک معیار به‌کار ببریم، زیرا دستورها در زبان‌های مختلف متفاوت بوده و در نتیجه تعداد خطوط یک برنامه یکسان با استفاده از هر زبان نیز متفاوت خواهد بود؛ و یا به‌کارگیری تعداد چرخه‌های واحد پردازنده مرکزی برای اجرای برنامه نمی‌تواند معیار مناسبی باشد، زیرا این مقدار در رایانه‌های مختلف بسته به قدرت پردازنده، متفاوت است.

مناسب‌ترین معیاری که می‌توان به کار برد، توجه به ساختار الگوریتم‌ها است. الگوریتم‌ها معمولا دارای مجموعه‌ای از دستورات اصلی هستند که در یک یا چند بخش از بدنه الگوریتم قرار دارند؛ کارایی الگوریتم وابسته به این بخش‌هاست. این بخش‌ها، به عنوان بخش/بدنه اصلی الگوریتم شناخته می‌شوند. برای تحلیل الگوریتم‌ها معمولا به تحلیل دو جنبه از آن‌ها پرداخته می‌شود که عبارتند از:

۱- تحلیل پیچیدگی زمانی

۲- تحلیل پیچیدگی فضایی

در ادامه به توضیح این دو جنبه می‌پردازیم.

٤-۱-۱- تحلیل پیچیدگی زمانی

معیاری مناسب برای تحلیل پیچیدگی زمانی، توجه به حلقه‌ها در الگوریتم است. فرض می‌کنیم، ریزعمل‌هایی همچون مقداردهی اولیه، افزایش نمایه و ... در کوچکترین واحد زمانی انجام شوند و به تنهایی تاثیری در پیچیدگی زمانی الگوریتم نداشته باشند؛ بنابراین فقط به تعداد اجرای بدنه اصلی توجه می‌کنیم. تشخیص بدنه‌ی اصلی الگوریتم روش مشخصی نداشته و معمولا به صورت تجربی صورت می‌گیرد. تعداد دفعات تکرار انجام عمل اصلی به ازای اندازه ورودی n با $T(n)$ نشان داده شده و به عنوان پیچیدگی زمانی الگوریتم شناخته می‌شود. حلقه‌ها فقط محدود به ساختارهایی مانند for و while نمی‌شود بلکه توابع بازگشتی، انواع پرش‌ها و فراخوانی‌ها نیز در این دسته قرار می‌گیرند. برای روشن‌تر شدن مفاهیم گفته شده به نمونه‌های ساده آورده شده در ادامه توجه کنید.

نمونه اول

```
int n;
cin >> n;
for (int i = 1; i <= n; i++)
    cout << i << endl;
```

برای این نمونه $T(n) = n$ است که n از ورودی خوانده می‌شود. حلقه for به کار رفته در نمونه بالا، بدنه اصلی برنامه را تشکیل داده و به تعداد n دریافت شده از ورودی تکرار می‌شود.

نمونه دوم

```
int arr[n] = {1, 2, 5, 4, 3, ...};
for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n; j++) {
        if (arr[i] > arr[j])
            swap(arr[i], arr[j]);
    }
```

برای این نمونه $T(n)=\frac{n(n-1)}{2}$ است که n طول آرایه arr است. این نمونه عنصرهای موجود در آرایه arr را به صورت صعودی مرتب می‌کند. در دور اول از حلقه بیرونی، حلقه درونی $n-1$ بار اجرا می‌شود؛ در دور دوم، $n-2$ بار و همین طور به ترتیب کاهش یافته تا اینکه در پایان به ۱ بار می‌رسد؛ بنابراین خواهیم داشت:

$$T(n)=(n-1)+(n-2)+...+1=\frac{n(n-1)}{2}$$

نمونه سوم

```
int n;
void print_next(int i) {
    if (i > n)
        return;
    cout << i << endl;
    print_next(i + 1);
}
int main() {
    cin >> n;
    print_next(1);
    return 0;
}
```

تابع بازگشتی به کار رفته در این نمونه، در واقع عملی مشابه با نمونه اول انجام می‌دهد. تعداد تکرار (فراخوانی تابع) یکی بیشتر از اندازه ورودی n است، زیرا آخرین فراخوانی print_next(i:n+1) خواهد بود؛ بنابراین برای این نمونه داریم:

$$T(n)=n+1$$

٤-١-٢- تحلیل پیچیدگی فضایی

معیار دومی که برای مقایسه الگوریتم‌ها دارای اهمیت است، پیچیدگی فضایی آن‌ها است. پیچیدگی فضایی به مقدار فضای اشغال شده توسط یک الگوریتم در زمان اجرا اطلاق می‌شود. وقتی پیچیدگی فضایی الگوریتم‌ها در حدی پایین (نسبت به حافظه‌های معمول امروزی) قرار داشته باشد، معمولا فقط به مقایسه پیچیدگی زمانی آن‌ها پرداخته می‌شود؛ در غیر این صورت هر دو پیچیدگی ذکر خواهد شد. برای نمایش پیچیدگی فضایی از $M(n)$ استفاده می شود.

٤-١-٣- مرتبه الگوریتم‌ها

پیچیدگی بعضی الگوریتم‌ها به ازای مقادیر مختلف ورودی، یکسان نیست. برای مثال یک جستجوی خطی ساده را در نظر بگیرید. ممکن است عنصر مورد جستجو در خانه‌ی اول آرایه قرار داشته باشد و یا در خانه‌ی آخر! نه می‌توان گفت که پیچیدگی زمانی جستجو خطی همیشه برابر با $T(n)=1$ است و نه می‌توان گفت $T(n)=n$. به صورت کلی، الگوریتم‌ها دارای بدترین حالت، حالت میانگین و بهترین حالت هستند. در صورتی که هر سه‌ی این حالات برای الگوریتمی یکسان باشد، تنها از $T(n)$ برای بیان پیچیدگی آن استفاده می‌کنیم که اشاره به پیچیدگی

زمانی معمول دارد. برای بیان بدترین حالت از نماد $W(n)$، حالت میانگین از نماد $A(n)$ و بهترین حالت از نماد $B(n)$ استفاده می‌شود.

برای بدست آوردن پیچیدگی زمانی جستجوی خطی در حالت میانگین، باید وجود عنصر مورد جستجو در هر یک از خانه‌های آرایه را بررسی کرده و سپس میانگین آن را محاسبه کنیم. اگر عنصر در خانه‌ی اول باشد، یک بررسی برای یافتن آن کافی است؛ اگر در خانه‌ی دوم باشد، دو بررسی و به همین ترتیب برای خانه‌ی آخر، n بررسی مورد نیاز است. بنابراین پیچیدگی زمانی جستجوی خطی برای حالت میانگین به صورت زیر محاسبه خواهد شد:

$$A(n) = \frac{\sum_{i=1}^{n} i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

تا این لحظه آموختیم که چگونه پیچیدگی زمانی تقریبی یک الگوریتم را بدست آوریم. اما همیشه محاسبه اینچنینِ پیچیدگی زمانی امکان‌پذیر نبوده و در مواردی اصلا مورد نیاز نیست. معمولا پیچیدگی زمانی و فضایی به صورت کلی بیان می‌شود؛ به شکلی که تنها گروه یا تابع کلی آن گزارش می‌شود. جدول ۱,۴، توابع پیچیدگی معمول را فهرست کرده است.

جدول ۱,٤. توابع پیچیدگی پرکاربرد

تابع پیچیدگی	گروه
1	ثابت
$\log n$	لگاریتمی
n	خطی
$n \log n$	شبه‌خطی یا خطی–لگاریتمی
n^2	درجه دوم
n^c	چند جمله‌ای
c^n	نمایی
$n!$	فاکتوریل

برای مثال اگر پیچیدگی زمانی الگوریتمی برابر با $T(n) = n^2 + 5n + 6$ باشد، با رشد n به مقادیر بسیار بزرگ، می‌توان از بخش $5n + 6$ صرفنظر کرد؛ زیرا مقدار این بخش در مقایسه با بخش درجه دوم (n^2) بسیار کوچک خواهد بود.

برای مقایسه پیچیدگی الگوریتم‌ها، یک روش معمول، استفاده از روش مجانبی است. در این روش، سقف یا کف تابع پیچیدگی الگوریتم بیان می‌شود (و یا هر دو). سه نماد O، θ و Ω برای این منظور ارائه شده‌اند که به آنها نمادهای مجانبی گفته می‌شود. نماد O (اُی بزرگ) برای تعیین سقف پیچیدگی به کار می‌رود و بیشتر از سایر نمادها به کار گرفته می‌شود. نماد θ (تتای بزرگ) برای بیان سقف و کف پیچیدگی الگوریتم مورد استفاده قرار

می‌گیرد. و نماد Ω (اُمگای بزرگ) برای تعیین کف تابع پیچیدگی به کار گرفته می‌شود. این سه نماد هم برای پیچیدگی زمانی و هم فضایی مورد استفاده قرار می‌گیرند.

وقتی می‌گوییم پیچیدگی جستجوی خطی برابر با $O(n)$ است، یعنی جستجوی خطی در بدترین حالت، از مرتبه خطی است. این امر به صورت $T(n) \in O(n)$ بیان می‌شود. پیچیدگی زمانی الگوریتم، رشد زمان اجرای تابع با افزایش اندازه‌ی ورودی را توصیف می‌کند. وقتی گفته می‌شود الگوریتمی دارای پیچیدگی زمانی $O(n)$ است، یعنی زمان اجرای الگوریتم نسبت به اندازه ورودی (n) به صورت خطی تغییر می‌کند؛ برای نمونه، با دو برابر شدن اندازه ورودی، زمان اجرای الگوریتم نیز دو برابر خواهد شد. پیچیدگی زمانی $O(n^2)$ به معنی چهار برابر شدن زمان اجرای الگوریتم در مقابل دو برابر شدن اندازه ورودی آن است. در ادامه به تعریف ریاضی این سه نماد خواهیم پرداخت و کاربرد آن‌ها را دقیق‌تر مورد بررسی قرار می‌دهیم.

نماد O

$f(n) = O(g(n))$ یعنی مقادیر مثبتی مانند c و k وجود دارند، به قسمی که نامساوی $0 \le f(n) \le cg(n)$ برای همه‌ی $n \ge k$ برقرار است. مقادیر c و k ثابت بوده و به مقدار n وابسته نیستند (شکل ۱,۴).

$$O(g(n)) = \{ f(n) : \exists c, k > 0,\ \forall n \ge k :\ 0 \le f(n) \le cg(n) \}$$

نماد θ

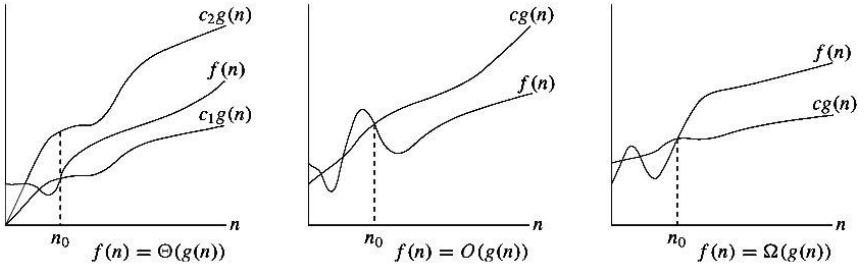
$f(n) = \theta(g(n))$ یعنی مقادیر مثبتی مانند c_1، c_2 و k وجود دارند، به قسمی که نامساوی $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ برای همه‌ی $n \ge k$ برقرار است. مقادیر c_1، c_2 و k ثابت بوده و به مقدار n وابسته نیستند (شکل ۱,۴).

$$\theta(g(n)) = \{ f(n) : \exists c_1, c_2, k > 0,\ \forall n \ge k :\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \}$$

نماد Ω

$f(n) = \Omega(g(n))$ یعنی مقادیر مثبتی مانند c و k وجود دارند، به قسمی که نامساوی $0 \le cg(n) \le f(n)$ برای همه‌ی $n \ge k$ برقرار است. مقادیر c و k ثابت بوده و به مقدار n وابسته نیستند (شکل ۱,۴).

$$\Omega(g(n)) = \{ f(n) : \exists c, k > 0,\ \forall n \ge k :\ 0 \le cg(n) \le f(n) \}$$



شکل ۱,٤. مقایسه نمادهای مجانبی پرکاربرد

به عنوان مثال، نشان می‌دهیم که $3n^2+15n=O(n^2)$. برای این منظور، باید دو ثابت مثبت c و k بیابیم که به ازای آن‌ها نامساوی زیر برقرار باشد. با قرار دادن $c=4$ و $k=15$ نامساوی زیر برقرار خواهد بود.

$$3n^2+15n \le cn^2$$

به عنوان مثال دوم، نشان می‌دهیم $\frac{n(n+1)}{3}=O(n^2)$. به ازای n_های مثبت داریم:

$$\frac{n(n+1)}{3}=\frac{n^2+n}{3}\le\frac{n^2+n^2}{3}=\frac{2}{3}n^2$$

بنابراین، با قرار $c=\frac{2}{3}$ و $k=2$ نامساوی زیر برقرار خواهد بود.

$$\frac{n(n+1)}{3}\le cn^2$$

با افزایش مرتبه‌ی پیچیدگی الگوریتم، سرعت اجرای الگوریتم (و یا فضای اشغالی توسط آن) به نسبت افزایش اندازه ورودی، کاهش (فضا ← افزایش) می‌یابد. پیچیدگی‌های معمول در ادامه به ترتیب رشد چیده شده‌اند. جدول ۲,۴ رشد برخی از این توابع را بر حسب اندازه ورودی نشان می‌دهد.

$$O(1)\to O(\log n)\to O(n)\to O(n\log n)\to O(n^2)\to O(n^3)\to O(2^n)\to O(3^n)\to O(n!)\to O(n^n)$$

جدول ۲,٤. رشد توابع پیچیدگی نسبت به اندازه ورودی

اندازه ورودی n	$O(1)$	$O(\log n)$	$O(n)$	$O(n\log n)$	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1,024	1	10	1,024	10,240	1,048,576	1,073,741,824
1,048,576	1	20	1,048,576	20,971,520	10^{12}	10^{16}

تحلیل پیچیدگی الگوریتم می‌تواند بسیار مفید و موثر باشد ولی نه همیشه؛ زیرا مشکلاتی هم در این میان وجود دارد؛ برای نمونه:

۱- **سخت بودن تحلیل:** بررسی بعضی از برنامه‌ها از نظر ریاضی بسیار دشوار است.

۲- **مجهول بودن عملکرد میانگین:** میانگین عملکرد الگوریتم‌ها، معیار بسیار مهمی برای مقایسه آن‌هاست؛ ولی اُی بزرگ اطلاعاتی درباره‌ی آن نمی‌دهد.

۳- **نادیده گرفتن ثابت‌ها:** همان‌طور که گفته شد، اُی بزرگ بعضی از سربارهای ثابت را برای محاسبه پیچیدگی نادیده می‌گیرد؛ در صورتی که برای مقایسه‌ی بعضی برنامه‌ها، این موضوع نیز حائز اهمیت است.

۴- **مجموعه داده‌های کوچک:** برای این چنین برنامه‌هایی، تحلیل پیچیدگی اهمیتی ندارد.

٤-۱-٤- توابع بازگشتی

شبه‌کد تابع بازگشتی نشان داده شده را در نظر بگیرید. a و b ثابت و n اندازه‌ی ورودی مسئله است. این تابع بازگشتی، در هر بار فراخوانی، یک مرتبه تابع f را فراخوانی کرده و سپس به تعداد a مرتبه خود را فرا می‌خواند. در هر بار فراخوانی خود، اندازه‌ی ورودی بر b تقسیم می‌شود.

```
function T(n: size of problem)
{
    if n < 1 then exit
        call f(n)
    for a times:
        call T(n / b)
}
```

پیچیدگی توابع بازگشتی با چنین ساختاری را می‌توان با این رابطه نمایش داد:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

با استفاده از قضیه مستر[۲۸] می‌توان پیچیدگی زمانی چنین توابعی را به سادگی محاسبه کرد. شکل کلی توابع بازگشتی مورد اشاره به صورت زیر است:

$$T(n) = aT(\frac{n}{b}) + f(n) \quad a \geq 1,\ b > 1$$

- n اندازه مسئله است.
- a تعداد فراخوانی‌های صورت گرفته به صورت بازگشتی است.
- $\frac{n}{b}$ اندازه‌ی ورودی جدید در فراخوانی بازگشتی است.

[۲۸] Master Theorem

- $f(n)$ به شکل $\theta(n^k)$ می‌باشد. این تابع، هزینه‌ای است که قبل و بعد از فراخوانی بازگشتی پرداخت می‌شود. این هزینه شامل تقسیم مسئله به زیرمسئله‌ها، ادغام نتیجه‌ها و سایر اعمال مرتبط است.

$$T(n) = aT(\frac{n}{b}) + \theta(n^k) \quad a \geq 1,\ b > 1$$

با توجه به رابطه‌ی بالا، پیچیدگی تابع بازگشتی با استفاده از روابط زیر محاسبه خواهد شد.

$$\begin{cases} \theta(n^k) & a < b^k \\ \theta(n^k \log n) & a = b^k \\ \theta(n^{\log_b^a}) & a > b^k \end{cases}$$

به عنوان مثال، رابطه بازگشتی بعدی را در نظر بگیرید:

$$T(n) = 9T(\frac{n}{3}) + n$$

در این رابطه، $a = 9$، $b = 3$ و $k = 1$ می‌باشد. در نتیجه پیچیدگی زمانی برابر خواهد بود با $\theta(n^2)$.
حال به رابطه‌ی بازگشتی بعدی توجه کنید:

$$T(n) = 16T(\frac{n}{4}) + n^2$$

در این رابطه، $a = 16$، $b = 4$ و $k = 2$ می‌باشد. در نتیجه پیچیدگی زمانی برابر خواهد بود با $\theta(n^2 \log n)$.
در صورتی که رابطه توابع بازگشتی به شکل کلی زیر باشد، راه‌حل کوتاه شده‌ی دیگری برای آن ارائه می‌کنیم.

$$T(n) = aT(n-b) + c \quad a > 0,\ \ b, c \geq 0$$

با توجه به رابطه‌ی گفته شده، پیچیدگی تابع بازگشتی با استفاده از این روابط محاسبه خواهد شد:

$$\begin{cases} \theta(a^{\frac{n}{b}}) & a \neq 1 \\ \theta(n) & a = 1 \end{cases}$$

به عنوان مثال، این رابطه بازگشتی را در نظر بگیرید:

$$T(n) = 5T(n-4) + 6$$

در این رابطه، $a = 5$، $b = 4$ و $c = 6$ می‌باشد. در نتیجه پیچیدگی زمانی رابطه بازگشتی برابر خواهد بود با $\theta(5^{\frac{n}{4}})$.

٤-۲- تکنیک‌های جستجو

در این بخش با تکنیک‌های جستجوی درخت‌ها آشنا می‌شویم. البته اغلب این تکنیک‌ها در واقع مربوط به گراف‌ها هستند که آن‌ها را برای درخت‌ها به کار می‌بریم؛ پیش از آغاز بحث مربوط به درخت‌ها، به گراف‌ها نیز اشاره‌ای خواهیم کرد.

منظور از جستجو، ملاقات گره‌های درخت (گراف) به ترتیبی ویژه است. هر یک از روش‌ها، به ترتیبی متفاوت و با پیچیدگی متفاوت به جستجو درون درخت‌ها می‌پردازند. در بعضی از مسأله‌ها، برای یافتن هدف (حل مسأله)، باید همه‌ی مسیرهای ممکن را بررسی کنیم که درخت در واقع تجسمی از حالت‌ها و مسیرهای ممکن مسأله است؛ در این حالت‌ها به درخت مربوطه، درخت فضای حالت یا درخت جستجو گفته می‌شود.

همه روش‌های پیمایش درخت یک هدف را دنبال می‌کنند که مشخصا دیدن همه گره‌های درخت است. بنابراین ساختار مسأله را یک بار بیان می‌کنیم و از تکرار آن در بخش‌های بعد می‌پرهیزیم.

الگوریتم تکنیک‌های جستجوی (پیمایش) درخت/گراف

۱- **صورت مسأله:** دیدن همه گره‌های درخت/گراف

۲- **ورودی:** درخت جستجوی مربوطه

۳- **خروجی:** ترتیب دیدن گره‌ها

٤-۲-۱- جستجوی اول‌عمق

در این روش، ابتدا ریشه ملاقات شده و سپس به سمت عمق درخت پیش می‌رویم؛ در هر مرحله یکی از فرزندان برگزیده شده و همین طور در جهت عمق درخت، فرزندان بعدی را ملاقات می‌کنیم. برگزیدن فرزند چپ یا راست بستگی به مسأله مورد نظر دارد؛ معمولا یا سمت چپ‌ترین فرزند هر گره برگزیده می‌شود و یا سمت راست‌ترین آن‌ها. عملکرد این روش تا حدود زیادی شبیه به پیمایش پیشوندی درخت‌های دودویی است؛ با این تفاوت که جستجوی اول‌عمق را می‌توان برای درخت‌های غیر دودویی نیز به کار برد.

این الگوریتم برای گراف‌ها نیز به شکلی مشابه عمل کرده و از میان همسایه‌های گره کنونی یکی را برگزیده و در عمق پیش می‌رود. با این تفاوت که در گراف امکان دیدن گره‌های تکراری وجود دارد، بنابراین پیش از باز کردن فرزندان هر گره، بررسی می‌کنیم که آیا این گره پیش از این دیده شده است یا خیر؟ و اگر دیده نشده بود، فرزندان آن را باز می‌کنیم.

چگونگی کار این الگوریتم در این شبه‌کد مشخص است:

```
void Graph_DFS(node v) {
    push(S, v)
    node current
    while (!empty(S)) {
        current = pop(S)
        visit current
```

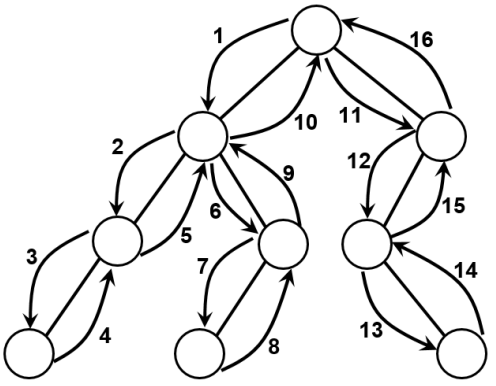
```
        node u
        for (each neighbor u of current)
            if (u not visited)
                push(S, u)
    }
}
```

پشته S برای پیشروی در عمق به کار می‌رود. آخرین فرزندی که باز می‌شود، بر روی پشته قرار می‌گیرد، در نتیجه همیشه عمق مقدم بر سطح خواهد بود (چرا؟).

تابع pop گره بالای پشته را برمی‌گرداند. تابع push گره u را در بالای پشته قرار می‌دهد. تابع empty بررسی خالی/پر بودن پشته را به عهده دارد. گره v گره‌ای است که جستجو از آن آغاز می‌شود؛ بنابراین در ابتدای کار، بر روی پشته قرار می‌گیرد.

از آنجا که هدف اصلی ما، جستجو در درخت‌ها است، به همین اشاره درباره‌ی گراف‌ها بسنده می‌کنیم. جستجو در درخت‌ها ساده‌تر خواهد بود، زیرا هیچگاه به گره‌های تکراری (که پیش از این ملاقات شده‌اند) بر نخواهیم خورد (چرا؟)

درخت شکل ۲,۴ مراحل جستجوی اول‌عمق را نشان می‌دهد؛ در این شکل ابتدا فرزند سمت چپ را برمی‌گزینیم. گره‌ها به ترتیبی که شماره‌ها نشان می‌دهند، ملاقات می‌شوند.



شکل ۲,٤. ترتیب ملاقات گره‌های یک درخت در جستجوی اول عمق

شبه‌کد بعدی، عملکرد این روش را با به‌کارگیری یک تابع بازگشتی شبیه‌سازی می‌کند. روش غیربازگشتی برای پیاده‌سازی این الگوریتم، به وسیله‌ی پشته صورت می‌گیرد که مشابه با شبه‌کد ارائه شده برای گراف خواهد بود.

```
void Tree_DFS(node v) {
    visit v
    node u
    for (each child u of v)
        Tree_DFS(u)
}
```

ورودی تابع، گره‌ای است که قصد بررسی آن را داریم. تابع پس از دیدن گره کنونی، با استفاده از حلقه for، همه فرزندان آن گره را بررسی کرده و تابع Tree_DFS را برای هر یک از آن‌ها فراخوانی می‌کند. برای شروع، تابع به این صورت فراخوانی می‌شود:

```
Tree_DFS(root);
```

متغیر root ریشه درخت است. کاربرد عملی این روش را در فصل بعد خواهید دید. حال به بررسی پیچیدگی این شبه‌کد می‌پردازیم. از آنجا که با یک تابع بازگشتی روبه‌رو هستیم، به دو عامل یاد شده برای آن توجه می‌کنیم:

۱- تعداد تکرار تابع بازگشتی برنامه‌ی بالا (عمق) برابر با عمق درخت است که آن را d فرض می‌کنیم.

۲- عامل انشعاب آن را برابر با b فرض می‌کنیم که در واقع تعداد فرزندان هر گره است.

بر طبق این دو عامل، پیچیدگی زمانی این الگوریتم $O(b^d)$ است. از آنجا که فقط یک عمق از درخت در زمان اجرا ذخیره می‌شود (d مرتبه فراخوانی تودرتوی تابع توسط خودش در پشته زمان اجرا ذخیره می‌گردد)، بنابراین پیچیدگی فضایی برابر $O(d)$ خواهد بود؛ یعنی به صورت خطی، نسبت به عمق درخت رشد می‌کند که پیچیدگی فضایی بسیار مناسبی است.

بر اساس تعداد گره‌ها و یال‌ها، پیچیدگی زمانی این الگوریتم برابر با $O(|V|+|E|)$ خواهد بود. پیچیدگی فضایی آن نیز $O(|V|)$ خواهد بود، زیرا در بدترین حالت، با درخت مورب مواجه هستیم. برای گراف‌ها، این مقدار به $O(2|V|)$ افزایش می‌یابد، زیرا باید دیده شدن/نشدن گره‌ها ذخیره گردد. $|V|$ تعداد گره‌ها و $|E|$ تعداد یال‌ها است. این روش برای درخت‌ها، تنها یک عمق را ذخیره می‌کند ($O(\log|V|)$).

٤-۲-۲- جستجوی اول‌سطح

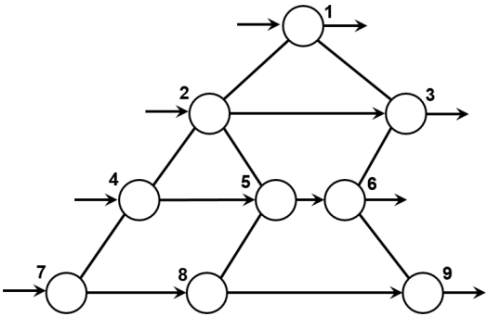
این روش همان‌طور که از نامش برمی‌آید، گره‌های هر سطح از درخت را بطور کامل دیده و سپس به سطح بعد می‌رود. در این روش، ابتدا گره ریشه را ملاقات می‌کنیم؛ سپس تک‌تک فرزندان آن را ملاقات خواهیم کرد؛ پس از ملاقات هر گره، فرزندان آن گره را به یک صف می‌افزاییم، تا بتوانیم در مرحله بعد، به ترتیب آن‌ها را ببینیم.

در آغاز، الگوریتم مربوط به گراف‌ها را ارائه کرده و سپس کار را با درخت‌ها ادامه می‌دهیم. شبه‌کد بعدی، بر خلاف شبه‌کد مربوط به جستجوی اول‌عمق، به جای پشته، یک صف را به کار می‌برد:

```
void Graph_BFS(node v) {
    enquque(Q, v)
    node current
    while (!empty(Q)) {
        current = dequeue(Q)
        visit current
        node u
        for (each neighbor u of current)
            if (u not visited)
                enqueue(Q, u)
    }
}
```

صف Q دارای سه تابع enqueue، dequeue و empty است که به ترتیب عنصری را در پایان صف قرار داده، عنصر آغاز صف را خارج کرده و خالی/پر بودن صف را بررسی می‌کنند. از آنجا که صف، تقدم را رعایت می‌کند و فرزندان در پایان صف قرار می‌گیرند، پس مشخصا، فرزندان در پایان کار و پس از اینکه همه گره‌های سطح کنونی دیده شدند، دیده خواهند شد.

حال به سراغ درخت‌ها می‌رویم. شکل ۳,۴ ترتیب جستجو را به روش اول‌سطح نشان می‌دهد. شماره‌های نشان داده شده ترتیب ملاقات گره‌ها را مشخص می‌کند.



شکل ۳,٤. ترتیب ملاقات گره‌های یک درخت در جستجوی اول سطح

شبه‌کد بعدی چگونگی کار این روش را توصیف می‌کند:

```
void Tree_BFS(node root) {
    enquque(Q, root)
    node current
    while (!empty(Q)) {
        current = dequeue(Q)
        visit current
        for (each child u of current)
            enqueue(Q, u)
    }
}
```

صف Q شامل گره‌هایی است که هنوز جستجو نشده‌اند و باید به ترتیب قرارگیری در صف دیده شوند. پس از جستجوی هر گره، آن را از ابتدای صف حذف می‌کنیم. برای شروع جستجو، گره ریشه به صف افزوده می‌شود. همان‌طور که ملاحظه می‌کنید، این شبه‌کد شباهت زیادی به شبه‌کد ارائه شده برای گراف‌ها دارد، با این تفاوت که در درخت‌ها نیازی به بررسی دیده شدن/نشدن گره نیست و مطمئنا گره‌های تکراری بررسی نمی‌شوند.

در شبه‌کد نشان داده شده، جستجو تا نهایت عمق درخت صورت می‌گیرد ولی در عمل نیازی به جستجوی همه‌ی درخت نیست (مگر اینکه فقط قصد پیمایش داشته باشیم) و در عمق مشخصی هدف را خواهیم یافت. بنابراین پیچیدگی زمانی و فضایی روش بالا بر اساس عمقی که هدف در آن قرار دارد، بیان می‌شود.

پیچیدگی فضایی این روش نسبت به عامل انشعاب درخت، نمایی است. اگر عامل انشعابی برابر با b و عمقی که هدف مورد نظرمان را در آن خواهیم یافت، k فرض کنیم، پیچیدگی فضایی برابر با $O(b^k)$ خواهد بود؛ زیرا در مرحله اول b گره به صف افزوده می‌شود، در مرحله دوم $O(b^2)$، در مرحله سوم $O(b^3)$ و در مرحله آخر $O(b^k)$. پیچیدگی فضایی برابر است با:

$$1+b+b^2+b^3+...+b^k \in O(b^k)$$

البته بیشترین تعداد گره‌ای که در یک زمان در صف وجود خواهد داشت، از مرتبه $O(b^k)$ است. پیچیدگی زمانی این روش برابر با پیچیدگی فضایی آن است زیرا تعداد بررسی‌ها برابر با تعداد گره‌هایی است که به صف افزوده می شوند.

بر اساس تعداد گره‌ها و یال‌ها، پیچیدگی زمانی مانند روش اول‌عمق برابر با $O(|V|+|E|)$ خواهد بود. پیچیدگی فضایی این روش $O(|V|)$ خواهد بود. این حالت وقتی اتفاق می‌افتد که درخت کاملا پهن باشد (همه‌ی گره‌های آن در یک سطح قرار داشته باشند). برای درخت‌ها، این روش تنها یک سطح را ذخیره می‌کند ($O(\frac{|V|}{2})$). به صورت کلی، پیچیدگی زمانی روش اول‌عمق بهتر از روش اول‌سطح است.

در هر دو روش اول‌عمق و اول‌سطح باید فضای مورد نیاز برای نگهداری گراف/درخت را نیز لحاظ کنیم. در صورتی که گراف با استفاده از ماتریس مجاورت نگهداری شود، به $O(|V|^2)$ فضا نیاز است. و در حالتی که از لیست پیوندی برای نگهداری گراف استفاده می‌شود، به $O(|V|+|E|)$ فضا نیاز است.

٤-۲-۳- جستجوی اول‌عمق عمیق‌شونده تکراری

اگر به پیچیدگی زمانی و فضایی دو روش پیشین دقت کرده باشید، حتما دریافته‌اید که روش اول‌عمق، پیچیدگی فضایی بهتر و روش اول‌سطح، پیچیدگی زمانی بهتری دارد. بنابراین هر کدام از این روش‌ها دارای نقطه ضعفی از نظر زمانی یا فضایی هستند. روش جستجوی اول‌عمقِ عمیق‌شونده تکراری، ترکیبی از دو روش پیشین است. کاربرد این روش در مواقعی است که به جستجوی اول‌سطح نیاز داشته باشید ولی فضای محدودی در اختیار دارید؛ در چنین حالتی فضا نسبت به زمان از اهمیت بیشتری برای شما برخوردار است.

این روش به صورت تکراری، روش اول‌عمق را به کار می‌برد ولی هر بار با یک محدودیت عمق L؛ یعنی در ابتدا، محدودیت عمق را یک قرار می‌دهد، پس از پایان جستجو، محدودیت عمق را دو قرار داده و دوباره به جستجوی اول‌عمق تا این عمق می‌پردازد و همین طور محدودیت عمق را افزایش می‌دهد؛ تا جایی که به نهایت عمق درخت برسد. شبه‌کد بعدی، الگوریتم این روش را توضیح می‌دهد. تابع Tree_DFS همان شبه‌کد ارائه شده برای روش اول‌عمق است به اضافه محدودیت عمقی که به آن افزوده شده است.

```
#define MAX_DEPTH d
int LIMIT_DEPTH
void Tree_DFS(node v, int depth) {
    visit u
    if (depth == LIMIT_DEPTH)
        return
```

```
    node u
    for (each child u of v)
        Tree_DFS(u, depth + 1)
}
void Tree_ID_DFS() {
    for (int L = 1; L <= MAX_DEPTH; i++) {
        LIMIT_DEPTH = L;
        Tree_DFS(root, 0);
    }
}
```

پیچیدگی فضایی این الگوریتم $O(k)$ است که k همان عمق هدف است. پیچیدگی زمانی این روش کمی بیشتر از پیچیدگی زمانی روش اول‌سطح است؛ البته بازهم در پایان دارای اُی بزرگ یکسانی خواهند بود. در این روش گره‌های آخرین سطح فقط یک بار بررسی می‌شوند (همان عمق هدف یعنی k)، گره‌های سطح پیش از آن دو بار و ... ؛ در پایان تعداد بررسی‌ها به صورت زیر خواهد بود. می‌بینید که این روش دارای پیچیدگی زمانی روش اول‌سطح و پیچیدگی فضایی روش اول‌عمق است.

$$(k)b + (k-1)b^2 + (k-2)b^3 + ... + (2)b^{k-1} + (1)b^k \in O(b^k)$$

بسته به صورت مسأله و محدودیت‌های آن، هر یک از این روش‌ها می‌تواند کارا باشد؛ در جدول ۳,۴ یک جمع‌بندی از پیچیدگی این سه روش و موارد کاربرد هر یک ارائه داده است. در این جدول همچون قبل b و d به ترتیب عامل انشعاب و عمق درخت، و k عمق گره هدف را مشخص می‌کند.

جدول ٣,٤. مقایسه عملکرد سه روش جستجوی اول عمق، اول سطح و اول عمق عمیق شونده تکراری

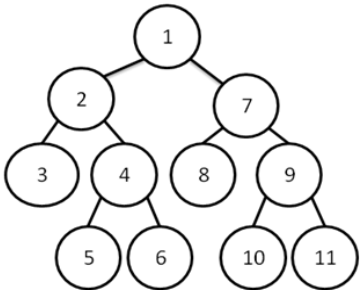
روش جستجو	پیچیدگی زمانی	پیچیدگی فضایی	کاربرد
اول‌عمق	$O(b^d)$	$O(d)$	می‌دانید که باید کل درخت جستجو شود از عمق هدف مطلع هستید به دنبال کوتاه‌ترین مسیر نیستید
اول‌سطح	$O(b^k)$	$O(b^k)$	به دنبال کوتاه‌ترین مسیر نسبت به ریشه هستید می‌دانید که هدف در نزدیکی ریشه قرار دارد
عمیق‌شونده تکراری	$O(b^k)$	$O(k)$	قصد به‌کارگیری روش اول‌سطح را دارید ولی با محدودیت فضا روبه‌رو هستید و از طرفی زمان برایتان اهمیت کمتری دارد

٤-۳- جستجوی عقب‌گرد

در بعضی از مسأله‌ها، هدف یافتن دنباله‌ای از مقادیر است؛ برای نمونه در مسأله معروف هشت‌وزیر، هدف یافتن یک یا همه‌ی دنباله‌های به طول هشت است که هر کدام از عناصر آن بیانگر موقعیت یکی از وزیرها در سطرهای ۱ تا ۸ صفحه‌ی شطرنج می‌باشند. یافتن هر یک از مقدارهای این دنباله را یک مرحله از حل مسأله در نظر می‌گیریم.

جستجوی عقب‌گرد برای درخت‌ها به کار می‌رود؛ در واقع این روش حالت اصلاح شده‌ای از جستجوی اول‌عمق است که در بخش پیش توضیح داده شد. در جستجوی اول‌عمق به صورت کورکورانه در عمق پیش می‌رفتیم تا به انتها رسیده و سپس به سمت ریشه باز می‌گشتیم تا مسیر دیگری را دنبال کنیم. روش عقب‌گرد کوشش می‌کند مسیرهایی را که در پایان به بن‌بست منتهی می‌شود، تشخیص داده و از پیشروی در آن مسیرها خودداری کند. به همین دلیل نام این روش، عقب‌گرد قرار داده شده است، زیرا مسیرهای نادرست را تشخیص داده، به عقب (ریشه) بازگشته و مسیر دیگری را دنبال می‌کند. این روش خود از روش اول‌عمق برای یافتن هدف بهره می‌برد؛ به اضافه‌ی اینکه در هر مرحله محدودیت‌های مسأله و شرایط کنونی، بررسی شده تا درستی مسیر تشخیص داده شود. کاربرد اصلی جستجوی عقب‌گرد برای مسأله‌های ارضای محدودیت است. مسائل ارضای محدودیت به مسائلی که دارای یک محدودیت مشخص هستند اطلاق می‌شود؛ برای نمونه مسأله هشت‌وزیر در این دسته قرار می‌گیرد و محدودیت آن، عدم تهدید وزیرها توسط یکدیگر است.

ترتیب نشان داده شده در شکل ۴,۴ می‌تواند یک نتیجه از دیدن گره‌ها با استفاده از جستجوی عقب‌گرد باشد. ترتیب برگزیده شده کاملا تصادفی است. همان‌طور که ملاحظه می‌کنید، در گره‌های ۳، ۵، ۶، ۸ و ۱۰ عقب‌گرد صورت گرفته است. اگر بیشترین عمق درخت ۳ باشد، گره ۱۱ گره پایانی درخت و آخرین گره‌ای است که بررسی می‌شود؛ بنابراین می‌تواند نقطه‌ی یافتن پاسخ مسأله و یا تشخیص عدم وجود راه‌حل برای مسأله باشد.



شکل ٤,٤. نمونه‌ای از ترتیب ملاقات گره‌های یک درخت جستجو در روش جستجوی عقبگرد

مسأله‌های ارضای محدودیت مانند هشت‌وزیر، رنگ‌آمیزی گراف و سودوکو با به‌کارگیری روش عقب‌گرد قابل حل هستند. در این بخش به ارائه الگوریتم برای مسأله هشت‌وزیر و رنگ‌آمیزی گراف پرداخته و در فصل ۶ هر سه مسأله را پیاده‌سازی می‌کنیم.

الگوریتم روش عقب‌گرد

۱- صورت مسأله: یافتن مسیری از ریشه تا برگ درخت جستجو که محدودیت‌های مسأله را ارضا کند.
۲- ورودی: درخت جستجوی مربوطه و عمق آن.
۳- خروجی: آرایه‌ی path که شامل مسیری از ریشه تا برگ درخت است.

این شبه‌کد الگوریتم این روش را توصیف می‌کند:

```
#define MAX_DEPTH d
int path[MAX_DEPTH]
void BackTracking(node v, int depth) {
    if not_valid(v)
        return
    if (depth == MAX_DEPTH) {
        print solution
        return
    }
    for (each child u of v) {
        visit u
        path[depth] = u
        BackTracking(u, depth + 1)
    }
}
```

تابع not_valid وظیفه بررسی گره کنونی را برعهده دارد و اگر گره کنونی با محدودیت مسأله سازگار نباشد، مقدار false را برمی‌گرداند. عملکرد درونی این تابع بسته به مسأله مورد نظر متغیر است؛ مثلا برای مسأله هشت‌وزیر، باید ستون‌ها و قطرهای اصلی و فرعی مربوط به وزیر کنونی بررسی شده و امکان قرار گرفتن آن در مکان فعلی سنجیده شود.

اگر به عمق MAX_DEPTH (برگ‌های درخت) برسیم، یعنی مسیری از ریشه تا برگ را یافته‌ایم که محدودیت‌های مسأله را ارضا کرده‌اند. مسیر یافت شده که در آرایه Path قرار دارد، در واقع یک پاسخ مسأله است. بنابراین در داخل شرط if، اقدام به چاپ راه حل می‌نماییم. فراخوانی آغازین به این شکل خواهد بود (متغیر root ریشه‌ی درخت جستجو و عمق ریشه صفر فرض شده است):

```
BackTracking(root, 0)
```

زمان اجرای الگوریتم‌های مبتنی بر روش عقب‌گرد بسیار بهتر از الگوریتم‌های مبتنی بر روش اول‌عمق بدون هیچ استراتژی ویژه‌ای است؛ برای نمونه، مسأله n وزیر را با روش معرفی شده (الگوریتم اصلی در بخش بعد معرفی می‌شود) و همچنین روش اول‌عمق معمولی پیاده‌سازی کردیم و زمان اجرای آن‌ها را به ازای تعداد وزیرهای ۶، ۸، ۱۰ و ۱۲ در جدول زیر می‌بینید. تفاوت عملکرد دو الگوریتم، با افزایش تعداد وزیرها، چشمگیر می‌شود.

جدول ٤,٤. زمان حل مسأله n وزیر (بر حسب ثانیه) با استفاده از روش‌های جستجوی اول عمق و جستجوی عقب‌گرد

تعداد وزیر	اول‌عمق معمولی	عقب‌گرد
6	0.07	0.01
8	4	0.02
10	316	0.025
12	24014	0.03

٤-۳-۱- مسأله هشت‌وزیر

برای حل مسأله‌ی هشت‌وزیر، از سطر اول صفحه شطرنج آغاز کرده و در هر سطح یک وزیر را قرار می‌دهیم. پیش از قرار دادن هر وزیر، بررسی می‌کنیم که وزیر دیگری در ستون، قطر اصلی و قطر فرعی مربوط به آن وجود نداشته باشد؛ اگر وزیر دیگری وجود داشت، مکان وزیر کنونی را تغییر می‌دهیم تا مکانی مناسب برای آن بیابیم؛ در پایان اگر مکان مناسبی برای آن یافت نشد، یک مرحله به عقب بازگشته و مکان وزیر پیشین را تغییر می‌دهیم. فرآیند پیش‌روی و عقب‌گرد، به همین ترتیب ادامه می‌یابد تا جایی که همه وزیرها در مکانی مناسب قرار بگیرند.

الگوریتمی که در بیشتر مراجع ارائه می‌شود، در عمل الگوریتم بهینه‌ای نیست زیرا برای بررسی مناسب بودن مکان کنونی وزیر فعلی، از حلقه استفاده کرده (برای پیمایش ستون، قطر اصلی و فرعی) و تک‌تک خانه‌های مشکوک به تهدید شدن را بررسی می‌کند؛ این امر سرعت الگوریتم را در تعداد وزیرهای بالا بسیار کاهش می‌دهد. در روش پیشنهادی، برای بررسی ستون و قطرهای مربوط به هر مکان، از سه آرایه بهره می‌بریم که وجود یا عدم وجود وزیر در آن راستا را مشخص می‌کنند؛ یعنی هر مسیر به یک خانه از آرایه نگاشت می‌شود و با مراجعه به همان یک خانه، می‌توان تهدید شدن توسط وزیرهای دیگر را تشخیص داد.

الگوریتمی که در ادامه ارائه می‌کنیم، قادر به حل مسأله n وزیر است؛ با به‌کارگیری ثابت QUEENS می‌توانید تعداد وزیرها را مشخص کنید. البته می‌توان بهینه‌سازی‌هایی برای این راه‌حل ارائه کرد که سرعت اجرای الگوریتم را تا حدودی بهبود می‌بخشد. برای نمونه می‌توانید وزیر اول را در نیمه اول از سطر اول حرکت داده و دنباله‌های بدست آمده را نسبت به نیمه‌ی صفحه شطرنج معکوس کنید تا نیمی دیگر از پاسخ‌ها نیز تولید شوند؛ در این حالت، از نیمی از پردازش‌ها صرفه‌جویی شده و زمان اجرای الگوریتم نصف خواهد شد. البته باید برای تعداد فرد وزیرها، کمی متفاوت عمل کنید (برای ستون میانی).

الگوریتم مسأله هشت‌وزیر

۱- صورت مسأله: قرار دادن هشت‌وزیر در صفحه شطرنج به شکلی که هیچیک از آن‌ها، یکدیگر را تهدید نکنند.

۲- ورودی: تعداد وزیرها که همان ابعاد صفحه شطرنج می‌باشد (ثابت QUEENS).

۳- خروجی: آرایه‌ای به طول تعداد سطرهای صفحه شطرنج که خانه‌ی i_ام آن، دربردارنده مکان وزیر در سطر i_ام است.

```
#define QUEENS 8
bool col[QUEENS];
bool diagonal1[QUEENS];
bool diagonal2[QUEENS];
int answer[QUEENS];
void NQueen(int r) {
    if (r > QUEENS) {
        print answer;
        return;
    }
```

```
    for (int c = 1; c <= QUEENS; c++) {
        if (col[c] || diagonal1[c - r + 1] || diagonal2[c + r])
            continue;
        answer[r] = c;
        col[c] = diagonal1[c - r + QUEENS] = diagonal2[c + r] = true;
        NQueen(r + 1);
        col[c] = diagonal1[c - r + QUEENS] = diagonal2[c + r] = false;
    }
}
```

این الگوریتم همه‌ی راه‌حل‌های ممکن را به دست آورده و چاپ می‌کند. با کمی تغییر می‌توانید الگوریتم را پس از یافتن نخستین راه حل، متوقف کنید.

سه آرایه col، diagonal1 و diagonal2 برای بررسی وجود وزیر در ستون، قطر اصلی و فرعی مربوط به مکان کنونی در نظر گرفته شده‌اند. ترکیبات c+r و c-r+QUEENS، هر کدام از قطرهای فرعی و اصلی را به یک خانه از آرایه‌های diagonal2 و diagonal1 نگاشت می‌کنند. برای نمونه مختصات بزرگترین قطر فرعی صفحه شطرنج هشت در هشت را در نظر بگیرید؛ از بالایی‌ترین خانه‌ی سمت راست به سمت پایینی‌ترین خانه‌ی سمت چپ:

$$(1,8),(2,7),(3,6),(4,5),(5,4),(6,3),(7,2),(8,1)$$
$$(1+8)=(2+7)=\ldots=(7+2)=(8+1)=9$$

ترکیب c+r، همه‌ی این هشت خانه را به مکان نهم آرایه diagonal2 نگاشت می‌کند؛ بنابراین می‌توان به سادگی بررسی کرد که آیا وزیری در این قطر قرار دارد یا نه. ترکیب c-r+QUEENS نیز به همین شکل برای قطرهای اصلی عمل می‌کند. برای نمونه بزرگترین قطر اصلی صفحه شطرنج هشت در هشت را به خانه هشتم از آرایه diagonal1 نگاشت خواهد کرد.

$$(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9)$$
$$(1-1+8)=(2-2+8)=\ldots=(8-8+8)=(9-9+8)=8$$

سه دنباله‌ی اولی که این الگوریتم برای مسأله‌ی هشت‌وزیر می‌یابد، در ادامه آورده شده‌اند. اعداد از سمت چپ، موقعیت وزیرها در سطرهای اول تا هشتم را نشان می‌دهند.

```
1 5 8 6 3 7 2 4
1 6 8 3 7 4 2 5
1 7 4 6 8 2 5 3
```

٤-۳-۲- رنگ‌آمیزی گراف

گراف ساختمان‌داده‌ای بسیار کاربردی است که پیمایش و جستجو در آن، همیشه مورد بحث بوده است. در بخش های پیشین روش‌هایی را برای پیمایش و جستجو در گراف‌ها معرفی کردیم. در این بخش به مسأله رنگ‌آمیزی

گراف می‌پردازیم. اما پیش از ارائه صورت مسأله و راه‌حل‌های آن به تعاریفی نیاز داریم که باید ابتدا به آن‌ها بپردازیم.

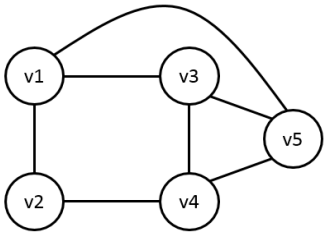
٤-۳-۲-۱- رنگ‌آمیزی m

رنگ‌آمیزی m به معنی رنگ کردن یک گراف تنها با به‌کارگیری m رنگ متمایز است، به شکلی که هیچ دو گره ی مجاوری دارای رنگ‌های یکسان نباشند.

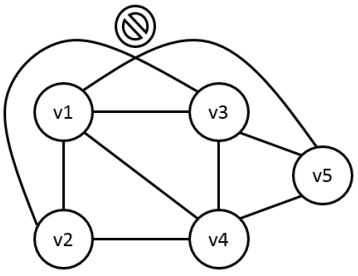
به ازای هر مقداری از m با مسأله‌ای متفاوت روبه‌رو هستیم. برای نمونه شاید گرافی با دو رنگ ($m = 2$) قابل رنگ‌آمیزی نباشد ولی با سه رنگ باشد.

٤-۳-۲-۲- گراف مسطح

گراف مسطح به گرافی گفته می‌شود که بتوان آن را روی صفحه رسم کرد به شکلی که هیچیک از دو یال آن یکدیگر را قطع نکنند. برای نمونه گراف شکل ۵,۴ گرافی مسطح بوده ولی گراف شکل ۶,۴ مسطح نیست.



شکل ٥,٤. نمونه‌ای از یک گراف مسطح



شکل ٦,٤. نمونه‌ای از یک گراف غیرمسطح

رنگ‌آمیزی گراف کاربردهای زیادی دارد که یکی از مهم‌ترین آن‌ها، رنگ‌آمیزی نقشه است. هر نقشه‌ای را می‌توان به یک گراف مسطح تبدیل کرد. رنگ‌آمیزی گراف نیز از مسأله‌های ارضای محدودیت است که محدودیت آن به شکل زیر بیان می‌شود:

هیچ دو گره مجاوری دارای رنگ یکسان نباشند.

یک راه حل ساده برای این مسأله به‌کارگیری روش عقب‌گرد است. هر چقدر استراتژی روش عقب‌گردمان را برای پیش‌بینی حالت‌های منجر به شکست در درخت فضای حالت بهبود ببخشیم، کارایی آن را بالاتر برده‌ایم. ساده‌ترین

درخت فضای حالت برای این مسأله، درختی است که دارای عامل انشعاب m برای m رنگ و دارای عمق n برای n گره است. بنابراین در هر سطح یک گره رنگ می‌شود، به صورتی که یکی از m رنگ موجود در آن سطح برگزیده شده و به سطح بعدی می‌رویم و به همین شکل همه‌ی گره‌ها را رنگ می‌کنیم؛ پیش از برگزیدن هر رنگ، امکان برگزیدن آن بررسی خواهد شد. اگر امکان برگزیدن هیچ رنگی برای گره کنونی فراهم نباشد، به عقب بازگشته و مسیر دیگری را دنبال می‌کنیم.

برای پیاده‌سازی این الگوریتم، یک آرایه دو بعدی (مثلا به نام graph) را برای تعیین وجود یال میان دو گره به کار می‌بریم؛ اگر میان دو گره‌ی i و j یال وجود داشته باشد، مقدار graph[i][j]، true خواهد بود. برای نگهداری رنگ برگزیده شده برای هر گره، از آرایه vcolor به طول n (تعداد گره‌های گراف) استفاده می‌کنیم.

الگوریتم مسأله رنگ‌آمیزی گراف

۱- صورت مسأله: رنگ‌آمیزی گراف توسط m رنگ متمایز به شکلی که هیچ دو گره مجاوری دارای رنگ‌های یکسان نباشند.

۲- ورودی: گراف مورد نظر و تعداد رنگ‌ها (m)

۳- خروجی: آرایه‌ای شامل رنگ‌های برگزیده شده برای هر گره و یا تعداد حالت‌هایی که می‌توان گراف ورودی را با تعداد رنگ‌های داده شده رنگ‌آمیزی کرد.

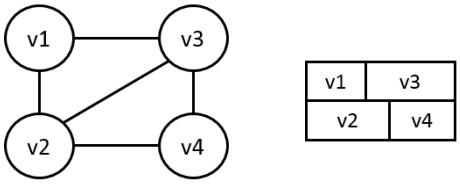
شبه‌کد ارائه شده همه‌ی راه‌حل‌های ممکن را برای مسأله رنگ آمیزی گراف یافته و چاپ می‌کند.

```
#define MAX_COLOR m
void m_coloring(int v) {
    if (v == n) {
        print solution
        return
    }
    for (int color = 0; color<MAX_COLOR; color++) {
        vcolor[v] = color
        if (valid(v))
            m_coloring(v + 1)
    }
}
bool valid(int v) {
    for (int i = 0; i<v; i++)
        if (graph[v][i] && vcolor[v] == vcolor[i])
            return false
    return true
}
```

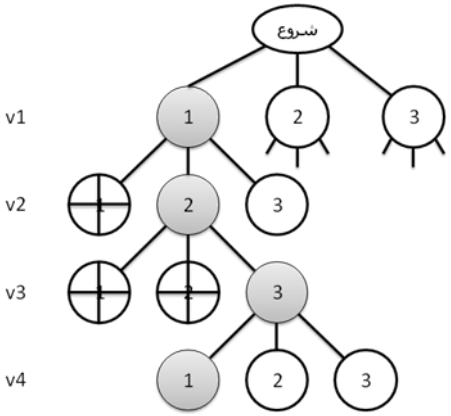
فراخوانی اولیه به این صورت انجام می‌شود:

```
m_coloring(0);
```

تابع valid رنگ برگزیده شده برای گره v را بررسی می‌کند؛ به این شکل که رنگ همه گره‌های رنگ‌آمیزی شده‌ی (نمایه‌ی آن‌ها کوچکتر از نمایه‌ی گره v است) مجاور با گره v را بررسی کرده و اگر یکی از آن‌ها رنگ یکسانی با گره v داشته باشد، مقدار false را برمی‌گرداند. بخشی از درخت جستجوی الگوریتم ارائه شده، برای گراف شکل ۷,۴ در شکل ۸,۴ نمایش داده شده است.



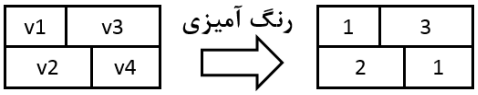
شکل ۷,٤. یک گراف مسطح و نقشه معادل با آن



شکل ۸,٤. بخشی از درخت جستجوی ایجاد شده توسط الگوریتم عقب گرد برای رنگ‌آمیزی گراف شکل ۷,٤

شکل ۹,۴ یکی از راه‌حل‌های رنگ‌آمیزی گراف را نشان می‌دهد که به این صورت است:

$$1, 2, 3, 1$$



شکل ۹,٤. یک رنگ آمیزی معتبر برای گراف شکل ۷,٤

الگوریتم ارائه شده می‌تواند به روش‌های گوناگونی بهینه‌سازی شود. برای نمونه این روش‌ها می‌توانند کارا باشند:

۱- وارسی پیش رو

۱-۱- معمولی
۱-۲- کمترین مقدار باقیمانده
۲- عقب‌گرد هوشمندانه

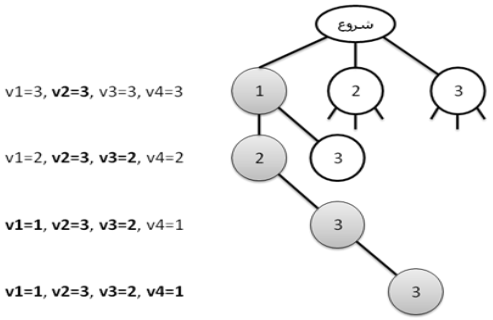
٤-۳-۲-۳- وارسی پیش رو

۱- معمولی

در این روش بهینه‌سازی، برای هر یک از گره‌ها یک دامنه در نظر می‌گیریم که رنگ‌های مجاز برای آن گره را مشخص می‌کند. هر بار که رنگی به یکی از گره‌ها انتساب داده شد، دامنه‌ی رنگ‌های مجاز برای همه گره‌های مجاور آن اصلاح می‌شود. با به‌کارگیری این روش، دیگر نیاز به استفاده از تابع valid برای بررسی مجاز بودن رنگ کنونی نیست، زیرا رنگ‌های غیرمجاز از دامنه‌ی گره حذف شده‌اند. به‌کارگیری این روش، تاثیر چشم‌گیری در سرعت اجرای الگوریتم خواهد داشت.

۲- کمترین مقدار باقیمانده

کمترین مقدار باقیمانده در واقع یک تابع اکتشافی است که برای ترتیب برگزیدن گره‌ها ارزش قائل می‌شود. در روش‌هایی که تا به حال ارائه شد، گره‌ها به ترتیب برگزیده می‌شدند؛ ولی با به‌کارگیری این تابع اکتشافی، گره‌ای را برمی‌گزینیم که دامنه رنگ‌های ممکن برای آن کمتر از بقیه گره‌های موجود در آن سطح باشد. اگر تعداد رنگ‌های ممکن برای دو یا چند گره یکسان بود، گره‌ای که دارای بیشترین تعداد همسایگی است، برگزیده می‌شود. اگر تعداد همسایه‌ها نیز برابر باشد، نمایه‌ی گره‌ها تعیین‌کننده خواهد بود. شکل ۱۰,۴ بخشی از درخت جستجوی گراف شکل ۷,۴ را نمایش می‌دهد.



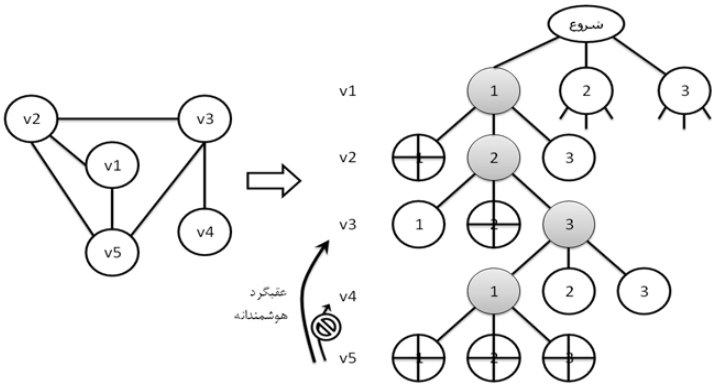
شکل ۱۰,٤. بخشی از درخت جستجوی گراف شکل ۷,٤ با استفاده از روش کمترین باقیمانده

مشاهده می‌کنیم که در هر مرحله، فقط رنگ‌های مجاز قرار دارند. V1=3 یعنی دامنه مقادیر مجاز گره v1 دارای سه عضو است؛ البته در عمل نگهداری تعداد اعضای دامنه، به تنهایی کافی نیست و باید خود مقدارها نیز نگهداری شوند. به‌کارگیری این روش موجب شناسایی زود هنگام حالت‌های بن‌بست شده و زمان اجرای الگوریتم را کاهش می‌دهد.

٤-۳-۲-٤- عقب‌گرد هوشمندانه

در الگوریتم عقب‌گرد، زمانی که به بن‌بست برمی‌خوریم، یک مرحله به عقب بازگشته و کار را از آنجا دنبال می‌کنیم. در مسائلی مانند رنگ‌آمیزی گراف، این عمل می‌توانند بسیار تعیین‌کننده باشد. برای نمونه فرض کنید، دو گره غیرمجاور از گراف (مثلا v4 و v5) به ترتیب در حال رنگ‌آمیزی هستند. فرض کنید هنگامی که نوبت به رنگ آمیزی گره v5 می‌رسد، بن‌بست ایجاد شده و باید به عقب بازگردیم؛ در چنین حالتی اگر یک مرحله به عقب برگردیم، به گره v4 می‌رسیم که هیچ ارتباطی با گره v5 ندارد (v4 و v5 غیرمجاورند) و تغییر رنگ آن تاثیری در رنگ برگزیده شده برای گره v5 نخواهد داشت؛ این امر باعث می‌شود چند انتقال بیهوده از گره v4 به v5 و بالعکس صورت گیرد و در پایان به گره قبل از v4 عقب‌گرد کنیم.

روش عقب‌گرد هوشمندانه پیشنهاد می‌کند: زمانی که بن‌بست رخ می‌دهد، به گره‌ای عقب‌گرد کنیم که در مجاورت با گره ایجاد کننده‌ی بن‌بست قرار دارد. این امر باعث صرفه‌جویی در تعداد زیادی از انتقال‌های بیهوده میان گره‌های غیرمجاور خواهد شد. برای نمونه، شکل ۱۱,۴ حالتی را نشان می‌دهد که به‌کارگیری عقب‌گرد هوشمندانه موثر است. البته این یک حالت بسیار ساده است؛ حالتی را تصور کنید که تعداد زیادی گره غیر مجاور وجود دارد و با یک عقب‌گرد هوشمندانه، از همه‌ی آن‌ها عبور خواهیم کرد.



شکل ۱۱,٤. حالتی از به بن بست رسیدن در رنگ‌آمیزی گراف که در آن به کارگیری عقب‌گرد هوشمندانه موثر است

٤-٤- مرتب‌سازی

در این بخش، روش‌های مرتب‌سازی را به صورت کامل بررسی کرده و الگوریتم‌هایی برای هریک ارائه می‌کنیم. سپس عامل‌هایی را در هر یک بررسی کرده که برای مقایسه‌ی آن‌ها با یکدیگر ضروری به نظر می‌رسند. ابتدا لازم است الگوریتم مرتب‌سازی را تعریف کنیم.

الگوریتم مرتب‌سازی، الگوریتمی است که مجموعه‌ای از عنصرها را دریافت کرده و آن‌ها را به ترتیب ویژه‌ای نسبت به یکدیگر قرار می‌دهد. روش‌هایی که در این بخش مورد بررسی قرار می‌گیرند، روش‌های **مرتب‌سازی مقایسه‌ای** هستند. روش‌هایی که فقط با مقایسه‌ی کلیدهای موجود در مجموعه، آن را مرتب می‌کنند، **روش‌های مرتب‌سازی مقایسه‌ای** نامیده می‌شوند. این روش‌ها یک عملگر مقایسه‌ای را به کار برده و هر بار

فقط دو کلید از مجموعه را با هم مقایسه می‌کنند. منظور از کلید، بخشی از عنصرها است که با یکدیگر مقایسه می‌شوند. در روش‌های توضیح داده شده، ترتیب صعودی مورد نظرمان خواهد بود.
روش‌های مرتب‌سازی معمولا بر اساس عاملهای زیر دسته‌بندی می‌شوند. هر یک از عاملهای زیر را برای همه‌ی روش‌های توضیح داده شده، بررسی خواهیم کرد.

۱- **پیچیدگی زمانی در حالت میانگین و بدترین حالت:** در روش‌های مرتب‌سازی مقایسه‌ای، پیچیدگی زمانی بر اساس تعداد مقایسه‌های انجام شده محاسبه می‌شود. برای روش‌های مرتب‌سازی، پیچیدگی زمانی بد، $O(n^2)$ و پیچیدگی زمانی خوب، $O(n\log n)$ است. البته، پیچیدگی زمانی ایده‌آل $O(n)$ است که تا به حال کسی نتوانسته به آن دست یابد.
۲- **پیچیدگی فضایی:** بعضی از روش‌های مرتب‌سازی از فضای اضافی استفاده نمی‌کنند؛ به این روش مرتب‌سازی، **مرتب‌سازی درجا** گفته می‌شود. روش‌های مرتب‌سازی درجا از پیچیدگی فضایی $O(1)$ و یا $O(\log n)$ برخوردارند. روش‌های دیگر، از لیست‌های کمکی برای مرتب‌سازی بهره می‌گیرند که این کار، پیچیدگی فضایی آن‌ها را به $O(n)$ افزایش می‌دهد. بنابراین منظور از پیچیدگی فضایی، فضای **اضافی** به کار رفته توسط روش مرتب‌سازی مورد نظر است.
۳- **بازگشت:** بعضی از روش‌های مرتب‌سازی به صورت بازگشتی و بعضی دیگر به صورت غیربازگشتی عمل می‌کنند. تعدادی از روش‌ها نیز، ترکیبی از هر دو را به کار می‌گیرند (مرتب‌سازی ادغامی). روش‌هایی نیز وجود دارند که به هر دو روش بازگشتی و غیربازگشتی قابل پیاده‌سازی هستند.
۴- **پایداری:** روش‌های مرتب‌سازی پایدار، ترتیب پیشین عنصرهایی با کلید یکسان را حفظ می‌کنند. پایداری یک روش بستگی به روش مقایسه‌ی عنصرها دارد.
۵- **انواع مختلف:** بعضی از روش‌ها، دارای انواع دیگری نیز هستند که فقط در موارد جزئی با یکدیگر تفاوت دارند. برای هر یک از روش‌های توضیح داده شده، به سایر نسخه‌های ارائه شده از آن‌ها نیز اشاره‌ی کوچکی خواهیم کرد.
۶- **مقایسه با روش‌های مرتب‌سازی دیگر:** برای تحلیل کارایی روش‌های مرتب‌سازی، بعضی از آن‌ها را با روش‌های هم‌خانواده مقایسه کرده و به مزایا و معایب هر یک می‌پردازیم.

برای درک بهتر مفهوم پایداری در روش‌های مرتب‌سازی، به این توضیحات توجه کنید:
مجموعه‌ای را در نظر بگیرید که شامل دو عنصر S و R با کلیدهای یکسان بوده و مکان عنصر S پیش از عنصر R است. اگر در مجموعه‌ی مرتب شده نیز عنصر S پیش از عنصر R قرار داشته باشد، روش مرتب‌سازی به‌کارگرفته شده، روشی پایدار نامیده می‌شود. برای نمونه، مجموعه‌ی زوج مرتب‌های زیر را درنظر بگیرید.
(4, 2) (3, 7) (3, 1) (5, 6)
فرض کنید این مجموعه را به این دو روش مرتب می‌کنیم:
۱- صعودی بر اساس جزء اول و سپس صعودی بر اساس جزء دوم
۲- صعودی بر اساس جزء اول و سپس نزولی بر اساس جزء دوم

مرتب‌سازی به روش اول:

(3, 1) (3, 7) (4, 2) (5, 6)

مرتب‌سازی به روش دوم:

(3, 7) (3, 1) (4, 2) (5, 6)

در صورتی مرتب‌سازی بر اساس جزء دوم اتفاق می‌افتد که جزء اول دو عنصر با هم برابر باشند. در مثال گفته شده، مرتب‌سازی بر اساس جزء دوم فقط به زوج مرتب‌های (3,7) و (3,1) اعمال می‌شود.
روش اول، ناپایدار و روش دوم پایدار است. زیرا در روش دوم، ترتیب دو زوج مرتب (3,7) و (3,1) نسبت به یکدیگر حفظ شده است.

الگوریتم‌های ارائه شده در این بخش از ساختار مشترک زیر برخوردارند؛ علاوه بر این همگی جزء دسته‌ی روش‌های مرتب‌سازی مقایسه‌ای قرار می‌گیرند. بهترین پیچیدگی زمانی برای این روش‌ها در بدترین حالت $O(n \log n)$ است؛ بنابراین، روش‌هایی را که از این درجه‌ی پیچیدگی زمانی برخوردارند، روش‌های بهینه می‌نامند.

الگوریتم‌های مرتب‌سازی

۱- **صورت مسأله:** مرتب کردن مجموعه‌ای از عنصرها
۲- **ورودی:** مجموعه‌ی عنصرها
۳- **خروجی:** مجموعه‌ی عنصرهای ورودی به صورت مرتب

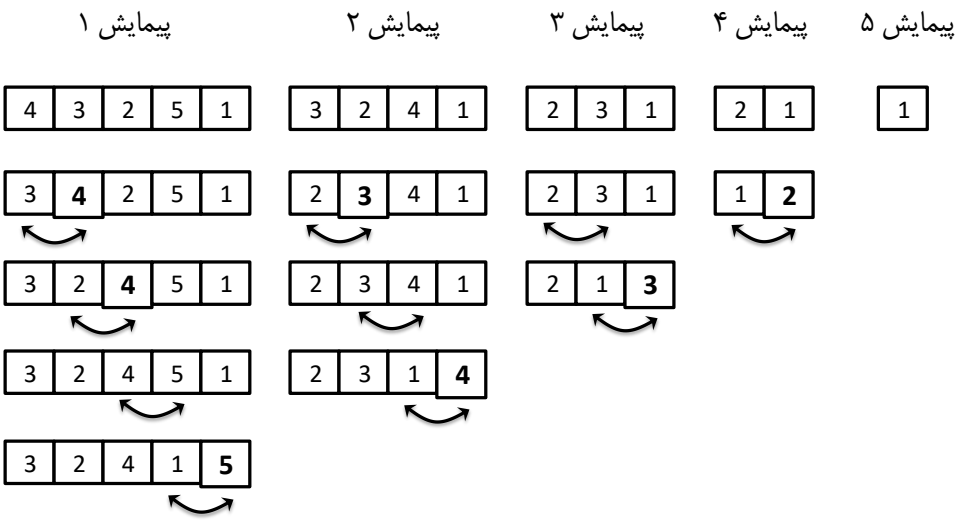
٤-٤-١- مرتب‌سازی حبابی

مرتب‌سازی حبابی روشی بسیار ساده است که فقط با جابه‌جایی عنصرهای یک مجموعه، آن را مرتب می‌کند. مرتب‌سازی حبابی از آغاز مجموعه آغاز کرده و هر بار دو عنصر را با هم مقایسه می‌کند؛ اگر عنصر اول از دوم بزرگتر باشد، دو عنصر را باهم جابه‌جا می‌کند. مقایسه‌ی عنصرهای کناری باهم، تا پایان مجموعه ادامه می‌یابد. در پیمایش اول، بزرگترین عنصر به مکان آخر مجموعه منتقل می‌شود (چرا؟). سپس دوباره از آغاز مجموعه آغاز کرده و روال پیش را تکرار می‌کنیم ولی این‌بار تا یکی مانده به پایان مجموعه پیش می‌رویم. این روال تا زمانی ادامه می‌یابد که یکی از دو حالت زیر اتفاق بیفتد:

۱- همه‌ی زوج عنصرها با هم مقایسه شده باشند و دیگر عنصری برای مقایسه وجود نداشته باشد (بدترین حالت ممکن)؛ این حالت زمانی اتفاق می‌افتد که مجموعه به صورت نزولی مرتب شده و قصد مرتب کردن صعودی آن را داشته باشیم (یا برعکس).
۲- در یک پیمایش کامل، هیچ دو عنصری باهم جابه‌جا نشده باشند؛ اگر این حالت اتفاق بیفتد، مطمئنا مجموعه مرتب شده است.

انتقال عنصر بزرگتر به پایان مجموعه مانند حرکت یک حباب است؛ به همین دلیل، این روش را روش حبابی می‌نامند. شکل ۱۲,۴ مراحل انجام شده برای مرتب کردن این مجموعه‌ی ۵ عضوی را نشان می‌دهد:

4 3 2 5 1



شکل ۱۲,٤. مرتب‌سازی مجموعه {4, 3, 2, 5, 1} به روش مرتب‌سازی حبابی

٤-٤-۱-۱- پیاده‌سازی مرتب‌سازی حبابی

فرض کنید a مجموعه‌ی اعداد مورد نظر و length تعداد اعداد مجموعه است. این کد پیاده‌سازی روش مرتب‌سازی حبابی را نشان می‌دهد (این کد را می‌توانید برای انواع داده‌ای دیگر نیز گسترش دهید):

```
void bubble_sort(int a[], int length) {
    bool swapped;
    do {
        swapped = false;
        length--;
        for (int i = 0; i<length; i++) {
            if (a[i]>a[i + 1]) {
                swap(a[i], a[i + 1]);
                swapped = true;
            }
        }
    } while (swapped);
}
```

٤-٤-١-٢- پیچیدگی زمانی مرتب‌سازی حبابی

تعداد مقایسه‌های انجام شده توسط روش مرتب‌سازی حبابی به صورت زیر محاسبه می‌شود:

$$(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}$$

با توجه به این رابطه، پیچیدگی زمانی این روش در حالت میانگین و بدترین حالت $O(n^2)$ است. بدترین حالت برای این روش حالتی است که مجموعه در جهت عکس مرتب شده باشد.

٤-٤-١-٣- پیچیدگی فضایی مرتب‌سازی حبابی

مرتب‌سازی حبابی روشی درجا است و از فضای اضافی برای مرتب کردن استفاده نمی‌کند. بنابراین پیچیدگی فضایی آن $O(1)$ خواهد بود.

٤-٤-١-٤- بازگشت

پیاده‌سازی بازگشتی برای این روش امکان‌پذیر است ولی معمولا زمانی روش بازگشتی به کار برده می‌شود که پیاده‌سازی و فهم روش را ساده‌تر کند. راه‌حل غیربازگشتی برای این روش معمول‌تر است.

٤-٤-١-٥- پایداری

راه‌حل ارائه شده، راه‌حلی پایدار است زیرا ترتیب اعداد یکسان را تغییر نمی‌دهد. اگر مقایسه‌ی اعداد را به این شکل انجام می‌دادیم، مرتب‌سازی غیرپایدار صورت می‌گرفت:

```
if (a[i] >= a[i + 1])
```

٤-٤-١-٦- انواع مختلف

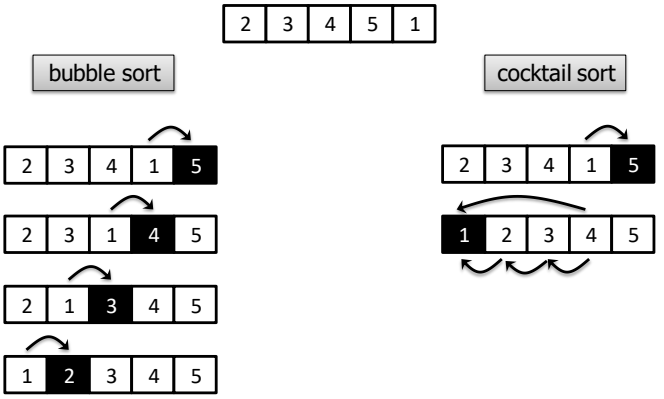
مرتب‌سازی حبابی دارای یک نوع دیگر به نام مرتب‌سازی ترکیبی نیز است که عملکرد آن را تا حدودی بهبود بخشیده است. در ادامه به شرح این روش می‌پردازیم.

٤-٤-١-٦-١- مرتب‌سازی ترکیبی

نام دیگر این روش، مرتب‌سازی حبابی دوطرفه است؛ زیرا در هر مرحله دوبار مجموعه را پیمایش می‌کند؛ یکبار از آغاز به پایان و بار دیگر در جهت عکس. در پیمایش اول، بزرگترین عنصر به پایان مجموعه منتقل شده و در پیمایش دوم کوچکترین عنصر به آغاز مجموعه منتقل می‌شود. این روش تقریبا دو برابر سریع‌تر از روش حبابی عمل می‌کند. برای نمونه این مجموعه‌ی ۵ عضوی را درنظر بگیرید:

2, 3, 4, 5, 1

شکل ۱۳,۴ تفاوت عملکرد این دو روش را بر روی این مجموعه نشان می‌دهد.



شکل ۱۳,٤. مرتب‌سازی مجموعه {2, 3, 4, 5, 1} به روش مرتب‌سازی حبابی ساده و مرتب‌سازی ترکیبی

می‌بینیم که روش ترکیبی دو مرحله سریع‌تر موفق به مرتب کردن مجموعه‌ی مورد نظر می‌شود. پیچیدگی زمانی روش ترکیبی در بدترین حالت و حالت میانگین برابر با $O(n^2)$ است. البته بازدهی این روش برای مجموعه‌های حدودا مرتب بهتر از درجه‌ی دوم است. سایر ویژگی‌های این روش مشابه با روش حبابی است (به جدول ۵,۴ توجه کنید).

جدول ٥,٤. ویژگی‌های روش مرتب‌سازی ترکیبی

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n^2)$
پیچیدگی زمانی در بدترین حالت	$O(n^2)$
پیچیدگی فضایی	$O(1)$
پایداری	بلی

٤-٤-١-٧- مقایسه با روش‌های مرتب‌سازی مرتبط

اگرچه مرتب‌سازی حبابی روشی بسیار ساده بوده و فهم آن بسیار آسان است؛ ولی پیچیدگی زمانی نامناسب آن موجب می‌شود تا برای مجموعه‌های بزرگ کاربرد چندانی نداشته باشد. از همین رو معمولا این روش در آموزش و معرفی الگوریتم‌های مرتب‌سازی به کار گرفته می‌شود.

این روش معمولا با روش‌های مرتب‌سازی درجی و مرتب‌سازی انتخابی مقایسه می‌شود، زیرا دارای پیچیدگی زمانی یکسانی هستند. مرتب‌سازی درجی اگرچه پیچیدگی زمانی یکسانی با این روش دارد ولی تفاوت عمده‌ای در تعداد جابجایی‌ها با آن دارد. بنابراین در بیشتر کتاب‌های امروزی، این روش نسبت به روش درجی کمتر به کار برده می‌شود.

روش حبابی دست کم دو برابر روش درجی عمل نوشتن در حافظه را انجام می‌دهد و در نتیجه دو برابر آن عدم اصابت در حافظه‌ی نهان اتفاق خواهد افتاد. براساس مقایسه‌های صورت گرفته، روش درجی حدودا ۵ مرتبه سریع‌تر و روش انتخابی ۱٫۴ مرتبه سریع‌تر از روش حبابی عمل می‌کند.

٤-٤-١-٨- جمع‌بندی

جدول ۶٫۴، ویژگی‌های اصلی روش مرتب‌سازی حبابی را نشان می‌دهد.

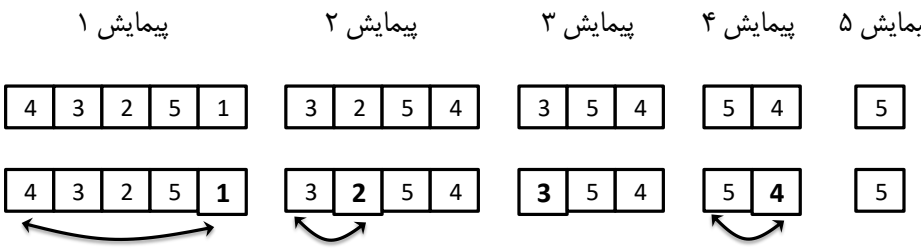
جدول ٦,٤. ویژگی‌های اصلی روش مرتب‌سازی حبابی

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n^2)$
پیچیدگی زمانی در بدترین حالت	$O(n^2)$
پیچیدگی فضایی	$O(1)$
پایداری	بلی

٤-٤-٢- مرتب‌سازی انتخابی

مرتب‌سازی انتخابی نیز مانند مرتب‌سازی حبابی روشی بسیار ساده را برای مرتب کردن مجموعه‌ای از عنصرها به کار می‌برد. این روش در نخستین پیمایش، کوچکترین عنصر مجموعه را یافته و با عنصر مکان اول جابه‌جا می‌کند؛ سپس دومین عنصر کوچکتر را یافته و با عنصر مکان دوم جابه‌جا می‌کند؛ این کار تا مرتب شدن همه‌ی اعضای مجموعه ادامه می‌یابد. در هر پیمایش، کوچکترین عنصر انتخاب می‌شود؛ به همین دلیل، این روش را روش انتخابی می‌نامند. شکل ۱۴٫۴ مراحل انجام شده برای مرتب کردن مجموعه‌ی این ۵ عضوی را نشان می‌دهد:

4 3 2 5 1



شکل ١٤,٤. مرتب‌سازی مجموعه {4, 3, 2, 5, 1} به روش مرتب‌سازی انتخابی

٤-٤-٢-١- پیاده‌سازی مرتب‌سازی انتخابی

نمونه‌ی کد بعدی دقیقا مراحل توضیح داده شده در مرتب‌سازی انتخابی را پیاده‌سازی می‌کند:

```
void selection_sort(int a[], int length) {
    int min_index;
    for (int i = 0; i<length - 1; i++) {
        min_index = i;
        for (int j = i + 1; j<length; j++) {
            if (a[j]<a[min_index])
                min_index = j;
        }
        if (i != min_index)
            swap(a[i], a[min_index]);
    }
}
```

٤-٤-٢-٢- پیچیدگی زمانی مرتب‌سازی انتخابی

تحلیل این روش در مقایسه با روش‌های دیگر ساده‌تر است؛ زیرا عملکرد آن به مقدار اعضای مجموعه بستگی ندارد؛ بنابراین، پیچیدگی زمانی در حالت میانگین و بدترین حالت برای این الگوریتم یکسان است.

تعداد مقایسه‌های صورت گرفته به این صورت محاسبه می‌شود:

$$(n-1)+(n-2)+...+2+1=\frac{n(n-1)}{2}$$

بنابراین پیچیدگی زمانی این روش در هر دو حالت میانگین و بدترین حالت برابر با $O(n^2)$ خواهد بود. تعداد جابجایی‌های این روش در بدترین حالت، از درجه‌ی یک است ($O(n)$).

٤-٤-٢-٣- پیچیدگی فضایی مرتب‌سازی انتخابی

مرتب‌سازی انتخابی نیز مانند مرتب‌سازی حبابی، روشی درجا است و از هیچ فضای اضافی برای مرتب کردن بهره نمی‌برد. بنابراین پیچیدگی فضایی آن $O(1)$ است.

٤-٤-٢-٤- بازگشت

روش معمولی برای مرتب‌سازی انتخابی معمول‌تر و بهینه‌تر است؛ بنابراین پیاده‌سازی اصلی این روش به صورت غیربازگشتی است. برای روش‌های مرتب‌سازی بعدی، تنها اگر پیاده‌سازی بازگشتی بهینه و یا معمول باشد، به توضیح آن خواهیم پرداخت.

٤-٤-٢-٥- پایداری

روش پیاده‌سازی شده، راه‌حلی پایدار برای این روش ارائه می‌کند. ترتیب مقادیر یکسان در پیاده‌سازی بالا تغییر نخواهد کرد.

٤-٤-٢-٦- انواع مختلف

مرتب‌سازی انتخابی به دو شکل دیگر، توسعه داده شده است. در ادامه به این دو اشاره شده است.

٤-٤-٢-٦-١- مرتب‌سازی بینگو

این روش در هر مرحله، دو بار مجموعه را پیمایش می‌کند:

۱- در پیمایش اول، کوچکترین عنصر یافت می‌شود.
۲- در پیمایش دوم، همه‌ی عنصرهای برابر با مقدار یافت شده به آغاز مجموعه و در مکان نهایی خود قرار داده می‌شوند.

زمانی که تعداد عنصرهای تکراری در مجموعه زیاد باشند، به‌کارگیری این روش مناسب بوده و کارایی آن بهتر از روش انتخابی خواهد بود. نام این روش به معنی کلمه‌ی bingo اشاره دارد که در واقع بانگ شادی برای پیدا کردن مکان درست همه‌ی مقدارهای مشابه است. بدترین حالت برای روش بینگو زمانی رخ می‌دهد که همه‌ی عنصرها یکتا بوده و هیچ دو عنصری دارای مقادیر یکسان نباشد. در این حالت پیچیدگی زمانی برابر با $O(n^2)$ است. پیچیدگی زمانی در حالت میانگین برابر با $O(nm)$ است که m تعداد عنصرهای یکتای مجموعه‌ی ورودی است (جدول ۷,۴). این روش را می‌توان به صورت پایدار نیز پیاده‌سازی کرد.

جدول ۷,٤. ویژگی‌های اصلی روش مرتب‌سازی بینگو

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(nm)$
پیچیدگی زمانی در بدترین حالت	$O(n^2)$
پیچیدگی فضایی	$O(1)$
پایداری	بلی

٤-٤-۲-٦-۲- مرتب‌سازی هرمی

مرتب‌سازی هرمی در واقع نوعی مرتب‌سازی انتخابی است؛ با این تفاوت که برای کاهش زمان یافتن کوچکترین عنصر، از ساختمان داده‌ای هرم استفاده می‌کند. این امر موجب می‌شود، زمان یافتن کوچکترین عنصر به $O(\log n)$ کاهش یابد. در نتیجه، پیچیدگی زمانی مرتب‌سازی هرمی $O(n \log n)$ خواهد بود. جزییات این روش در ادامه توضیح داده خواهد شد.

٤-٤-۲-۷- مقایسه با روش‌های مرتب‌سازی مرتبط

روش انتخابی معمولا با روش‌های حبابی و درجی مقایسه می‌شود. تعداد جابجایی‌های این روش در بدترین حالت از درجه‌ی یک است؛ در صورتی که تعداد جابه‌جایی‌های روش حبابی در بدترین حالت، از درجه‌ی دوم است.

یکی از تفاوت‌های اساسی مرتب‌سازی انتخابی با دیگر روش‌های مرتب‌سازی درجه دوم، مستقل از مقدار بودن آن است؛ عملکرد مرتب‌سازی انتخابی از نظر تعداد مقایسه‌های انجام شده، مستقل از مقدار عنصرهای مجموعه‌ی مورد نظر است؛ این ویژگی موجب می‌شود کارایی این روش همیشه ثابت بوده و برای همه‌ی اشکال ورودی‌ها، پیچیدگی زمانی ثابتی داشته باشد. به صورت کلی هیچکدام از این سه روش برای مرتب‌سازی مجموعه‌های بزرگ مناسب نیستند ولی با این حال، روش انتخابی کارایی بهتری نسبت به روش حبابی و کارایی کمتری نسبت به روش درجی دارد.

٤-٤-٢-٨- جمع‌بندی

جدول ۸,۴، موضوعات اصلی را برای مرتب‌سازی انتخابی دسته‌بندی می‌کند.

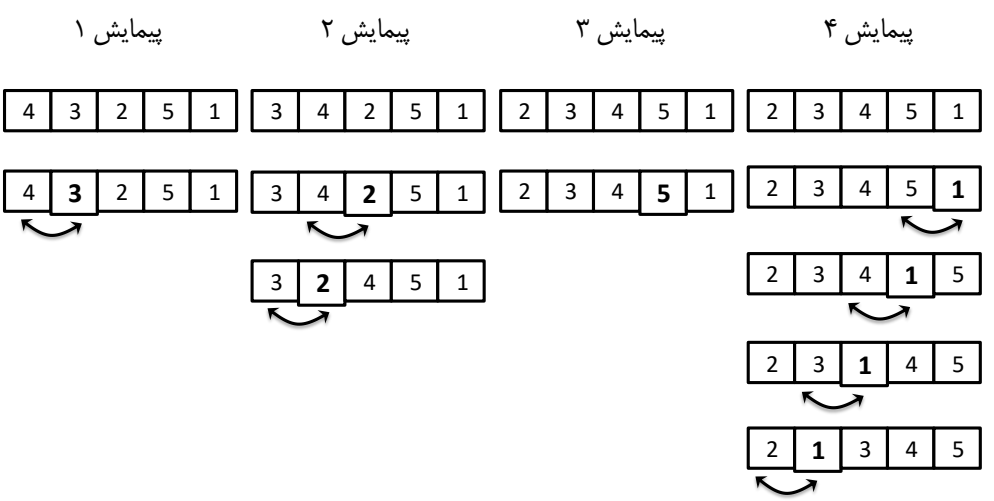
جدول ۷,٤. ویژگی‌های اصلی روش مرتب‌سازی انتخابی

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n^2)$
پیچیدگی زمانی در بدترین حالت	$O(n^2)$
پیچیدگی فضایی	$O(1)$
پایداری	بلی

٤-٤-٣- مرتب‌سازی درجی

مرتب‌سازی درجی از روش‌های ساده‌ی مرتب‌سازی است که با درج هر عنصر در مکانی مناسب به مرتب کردن مجموعه می‌پردازد. این روش، اعداد را از ورودی دریافت کرده و سپس به دنبال مکان مناسبی برای آن در مجموعه می‌گردد؛ به این صورت که از پایان مجموعه آغاز کرده و عناصری که مقدار آن‌ها از عنصر کنونی بیشتر است به سمت راست شیفت می‌دهد تا به عنصری برسد که مقدار آن برابر یا کوچکتر از عنصر کنونی است؛ آنگاه عنصر کنونی را در این مکان قرار می‌دهد. این توضیح برای زمانی است که عنصرها از ورودی دریافت می‌شوند؛ ولی در عمل ممکن است مجموعه به‌صورت کامل داده شود. در این حالت از عنصر دوم مجموعه آغاز کرده و به دنبال مکان مناسبی برای آن در بخش چپ مجموعه می‌گردیم؛ سپس این عمل را برای عنصر سوم تکرار می‌کنیم و همین‌طور به سمت پایان مجموعه پیش می‌رویم. شکل ۱۵,۴ مراحل کار این روش را برای این مجموعه نشان می‌دهد:

4 3 2 5 1



شکل ۱۵,٤. مرتب‌سازی مجموعه {4, 3, 2, 5, 1} به روش مرتب‌سازی درجی

این روش برای مجموعه‌های بزرگ نامناسب است ولی نسبت به سایر روش‌های مرتب‌سازی درجه دوم کارایی بهتری دارد. در کل این روش دارای مزایای زیر است:

۱- پیاده‌سازی آسان

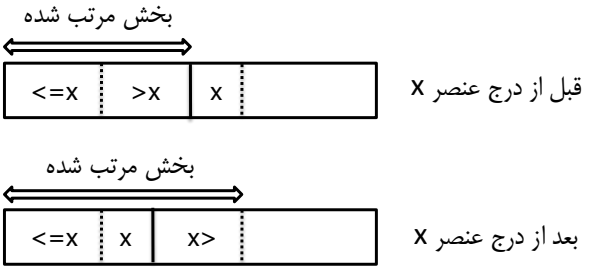
۲- بهینه برای مجموعه‌های کوچک

۳- پیچیدگی زمانی بهتر نسبت به سایر روش‌های مرتب‌سازی درجه دوم

۴- قابلیت مرتب‌سازی برخط: دریافت اعداد از ورودی و مرتب کردن آن‌ها

٤-٤-۳-۱- پیاده‌سازی مرتب‌سازی درجی

برای پیاده‌سازی مرتب‌سازی درجی، روشی را که توضیح داده شد، به کار می‌بریم. در این روش، از سمت چپ مجموعه آغاز به مرتب‌سازی می‌کنیم. هر عنصر نسبت به مجموعه‌ی سمت چپ خود سنجیده شده و در مکانی مناسب نسبت به آن قرار می‌گیرد. بنابراین پس از k پیمایش، $k+1$ عنصر سمت چپ مجموعه مرتب خواهد بود. شکل ۱۶,۴ این توضیح را به تصویر کشیده است.



شکل ١٦,٤. نحوه کار روش مرتب‌سازی درجی در پردازش عنصر x

پیاده‌سازی مرتب‌سازی درجی در این کد انجام شده است:

```
void insertion_sort(int a[], int length) {
    int j;
    for (int i = 1; i < length; i++) {
        int item = a[i];
        for (j = i - 1; j >= 0 && a[j] > item; j--)
            a[j + 1] = a[j];
        a[j + 1] = item;
    }
}
```

٤-٤-۳-۲- پیچیدگی زمانی مرتب‌سازی درجی

پیچیدگی زمانی مرتب‌سازی درجی در حالت میانگین برابر با $o(\frac{n^2}{4})$ است که از دو روش حبابی و انتخابی بهتر است. در بدترین حالت، این روش نیز مانند دو روش دیگر از پیچیدگی زمانی $O(n^2)$ برخوردار است. بدترین حالت

زمانی رخ می‌دهد که مجموعه در جهت عکس مرتب شده باشد؛ زیرا در این حالت برای یافتن مکان هر عنصر، باید کل عنصرهای سمت چپ آن را پیمایش کنیم. در این حالت تعداد مقایسه‌های لازم از این رابطه قابل محاسبه است:

$$1+2+...+(n-2)+(n-1)=\frac{n(n-1)}{2}$$

نکته‌ی دیگری که درباره‌ی این روش می‌توان گفت، پیچیدگی زمانی آن در بهترین حالت است. اگر مجموعه‌ی ورودی در جهت مورد نظر ما مرتب شده باشد، پیچیدگی زمانی $O(n)$ خواهد بود که بسیار مناسب است؛ در این حالت برای قرار دادن هر عنصر در مکان مناسب آن، فقط یک مقایسه کافی است.

مرتب‌سازی درجی برای مجموعه‌های ۱۰ عنصری و کمتر از آن بسیار سریع عمل می‌کند؛ بنابراین برای تعداد عنصرهای کم، این روش معمولا به روش‌های دیگر که حتی بعضی از آن‌ها از درجه‌ی خطی–لگاریتمی برخوردارند، ترجیح داده می‌شود.

٤-٤-٣-٣- پیچیدگی فضایی مرتب‌سازی درجی

این روش به صورت درجا عمل می‌کند. از این رو، پیچیدگی فضایی آن $O(1)$ است.

٤-٤-٣-٤- پایداری

پیاده‌سازی صورت گرفته، پایدار است. برای مرتب‌سازی درجی می‌توان به سادگی پیاده‌سازی پایدار ارائه کرد.

٤-٤-٣-٥- انواع مختلف

یکی از انواع مرتب‌سازی درجی، روش مرتب‌سازی شِل است. نام این روش از ابداع‌کننده‌ی آن دونالد شِل[۲۹] گرفته شده است. روشی دیگری که از روش درجی الهام گرفته، مرتب‌سازی کتابخانه‌ای است. در ادامه این دو روش شرح داده می‌شوند.

٤-٤-٣-٥-١- مرتب‌سازی شِل

در مرتب‌سازی درجی، برای یافتن مکان مناسب برای هر عنصر، آن را با طول گام ۱، حرکت می‌دادیم. برای نمونه اگر کوچکترین عنصر مجموعه در پایان آن قرار داشت، این عنصر باید با همه‌ی عناصر مجموعه جابه‌جا می‌شد تا به آغاز مجموعه که مکان مناسبی برای آن بود می‌رسید.

آقای شِل طول گام‌های متفاوتی را به جای طول گام ۱ پیشنهاد کرد. برای نمونه در توضیح قبلی، اگر کوچکترین عنصر را با طول گام ۵ حرکت می‌دادیم، زمان رسیدن به آغاز مجموعه و تعداد جابجایی‌های مورد نیاز کمتر می‌شد.

مرتب‌سازی شِل در چند مرحله و هر بار با طول گام متفاوتی صورت می‌گیرد. در هر مرحله طول گام کمتر شده و در پایان به طول گام یک که در واقع همان روش درجی است، خواهد رسید. در آخرین مرحله (طول گام یک)، مرتب‌سازی نهایی صورت خواهد گرفت؛ با انجام مراحل پیش از مرحله‌ی آخر (طول گام‌های بیشتر از یک) سرعت مرحله‌ی آخر بیشتر خواهد شد؛ زیرا عناصر کوچکتر به آغاز مجموعه و عناصر بزرگتر به پایان مجموعه نزدیکتر شده‌اند.

---

[۲۹] Donald Shell

برای نمونه، این مجموعه اعداد را در نظر بگیرید:

13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10

برای مرتب‌سازی این مجموعه، در مرحله‌ی اول طول گام ۵ را به کار می‌بریم. یعنی عناصری که در مکان‌های زیر قرار دارند را به صورت مستقل مرتب می‌کنیم (به روش درجی).

- ۱، ۶، ۱۱ و ۱۶
- ۲، ۷ و ۱۲
- ۳، ۸ و ۱۳
- ۴، ۹ و ۱۴
- ۵، ۱۰ و ۱۵

برای نمونه به صورت زیر اعداد مرتبط را در یک ستون قرار داده و هر ستون را مرتب می‌کنیم:

13 14 94 33 82
25 59 94 65 23
45 27 73 25 39
10

نتیجه‌ی این مرحله از مرتب‌سازی به شکل مجموعه‌ی زیر خواهد بود:

10 14 73 25 23
13 27 94 33 39
25 59 94 65 82
45

اگر در همین مرحله وضعیت مجموعه را بررسی کنیم، خواهیم دید که عنصر ۱۰ که در پایان مجموعه قرار داشت، با تعداد جابجایی بسیار کمی به آغاز مجموعه انتقال یافت؛ در صورتی که در مرتب‌سازی درجی معمولی به ۱۵ جابجایی نیاز بود. برای مرحله‌ی دوم، طول گام سه را به کار می‌گیریم:

10 14 73
25 23 13
27 94 33
39 25 59
94 65 82
45

مجموعه‌ی مرتب‌شده‌ی حاصل از این مرحله به شکل زیر است:

10 14 13
25 23 33
27 25 59
39 65 73
45 94 82
94

در مرحله‌ی پایانی، مرتب‌سازی درجی (طول گام یک) را به کار برده و به مجموعه‌ی مرتب مورد نظرمان دست پیدا می‌کنیم. در این حالت همه‌ی اعداد در یک ستون قرار داده شده و مرتب می‌شوند. با انجام دو مرحله‌ی پیش، تعداد جابجایی‌ها کاهش پیدا کرده و در نتیجه سرعت مرتب‌سازی افزایش خواهد یافت.

بخشی از این روش، انتخاب طول گام مناسب است. آقای شِل برای نخستین مرحله، طول گام $\frac{n}{2}$ را پیشنهاد کرد. به این شکل که در هر مرحله، طول گام نصف شده تا به طول گام یک برسد.

محاسبه‌ی پیچیدگی فضایی این روش دشوار بوده و بستگی به طول گام‌های برگزیده شده دارد؛ ولی در بدترین حالت، پیچیدگی اجرایی آن در حدود $O(n\log^2 n)$ خواهد بود. پیاده‌سازی این روش به فضای اضافی $O(n)$ نیاز دارد (جدول ۸,۴). علاوه بر این، ساختار این روش به شکلی است که نمی‌تواند به صورت پایدار عمل کند.

جدول ۸,٤. ویژگی‌های اصلی روش مرتب‌سازی شِل

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	وابسته به طول گام
پیچیدگی زمانی در بدترین حالت	$O(n\log^2 n)$
پیچیدگی فضایی	$O(n)$
پایداری	خیر

٤-٤-۳-٥-۲- مرتب‌سازی کتابخانه‌ای

نام روش مرتب‌سازی کتابخانه‌ای از قضیه‌ی زیر گرفته شده است:

کتابخانه‌داری را در نظر بگیرید که کتاب‌هایش را به ترتیب حروف الفبا در قفسه‌ی کتاب‌ها چیده است. همه‌ی کتاب‌ها از «الف» تا «ی» در کنار هم قرار داده شده‌اند و هیچ فاصله‌ای میان آن‌ها وجود ندارد. اگر کتابخانه‌دار بخواهد کتابی را به قفسه اضافه کند، باید کلیه کتاب‌های پس از آن را کمی به جلو حرکت دهد، تا جا برای کتاب جدید باز شود؛ این عمل در واقع همان روشی است که مرتب‌سازی درجی به کار می‌برد. حال فرض کنید که کتابخانه‌دار در پایان هر حرف کمی فضای خالی درنظر گرفته باشد؛ در این صورت نیازی به حرکت دادن همه‌ی کتاب‌ها نیست و فقط باید کتاب‌هایی که دارای حرف مشابه هستند به جلو حرکت داده شوند؛ این روش، ایده‌ی اصلی مرتب‌سازی کتابخانه‌های است. این روش در حالت میانگین عملکردی شبیه به خطی-لگاریتمی دارد و در بدترین حالت مانند روش درجی عمل می‌کند. علاوه‌بر این، مانند روش درجی روشی پایدار است. جدول ۹,۴ ویژگی‌های اصلی این الگوریتم را نشان می‌دهد.

جدول ۹,٤. ویژگی‌های اصلی روش مرتب‌سازی کتابخانه‌ای

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$\approx O(n \log n)$
پیچیدگی زمانی در بدترین حالت	$O(n^2)$
پیچیدگی فضایی	$O(1)$
پایداری	بلی

٤-٤-٣-٦- مقایسه با روش‌های مرتب‌سازی مرتبط

مرتب‌سازی درجی شباهت زیادی به مرتب‌سازی انتخابی دارد. برای نمونه در هر دو روش، پس از پایان k مرحله، k عنصر آغاز مجموعه، مرتب خواهند بود. هر یک از این دو روش مزیتی نسبت به دیگری دارد:

۱- روش درجی، برای یافتن مکان یک عنصر، فقط به تعداد مورد نیاز، مقایسه انجام می‌دهد؛ در صورتی که روش انتخابی همه‌ی عنصرها را جستجو می‌کند.

۲- تعداد نوشتن در حافظه، در روش انتخابی کمتر از روش درجی است؛ زیرا روش درجی برای یافتن مکان هر عنصر، هر بار آن را با عنصر کناری‌اش جابه‌جا می‌کند؛ این امر موجب می‌شود تا تعداد نوشتن‌ها در این روش از درجه‌ی دو باشد؛ در صورتی که، تعداد نوشتن در روش انتخابی حداکثر از درجه‌ی یک خواهد بود. بنابراین اگر هزینه‌ی نوشتن در حافظه بیشتر از هزینه‌ی خواندن باشد، روش انتخابی ترجیح داده می‌شود (مانند حافظه‌های EEPROM و flash memory).

روش درجی برای مجموعه‌های مرتب بسیار سریع عمل می‌کند؛ در حالت‌هایی که مجموعه‌ی ورودی به حالت مرتب نزدیک باشد، این روش نسبت به روش‌های درجه‌ی دوم دیگر، انتخاب مناسب‌تری است.

٤-٤-٣-٧- جمع‌بندی

جدول ۱۰,۴، موضوعات اصلی را برای این روش دسته‌بندی می‌کند.

جدول ۷,٤. ویژگی‌های اصلی روش مرتب‌سازی درجی

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n^2)$
پیچیدگی زمانی در بدترین حالت	$O(n^2)$
پیچیدگی فضایی	$O(1)$
پایداری	بلی

٤-٤-٤- مرتب‌سازی ادغامی

مرتب‌سازی ادغامی یکی از سریع‌ترین روش‌های مرتب‌سازی است. این روش، مجموعه‌ی ورودی را به مجموعه‌های کوچکتر شکسته و آن‌ها را مرتب کرده و سپس مجموعه‌های کوچک مرتب شده را در هم ادغام

می‌کند. ادغام دو زیرمجموعه‌ی مرتب شده به شکلی انجام می‌شود که مجموعه‌ی حاصل نیز مرتب خواهد بود. اساس کار این روش، ادغام کردن دو زیرمجموعه‌ی مرتب است؛ به این دلیل، این روش را روش ادغامی نامند. مراحل کلی این روش به صورت زیر است:

۱- اگر طول مجموعه صفر یا یک بود، پس نیاز به مرتب‌سازی ندارد. در غیر این صورت به مرحله‌ی ۲ می‌رویم.

۲- مجموعه را به دو زیر مجموعه با طول برابر تقسیم کرده و برای هر کدام از زیرمجموعه‌ها، مراحل گفته شده را تکرار می‌کنیم.

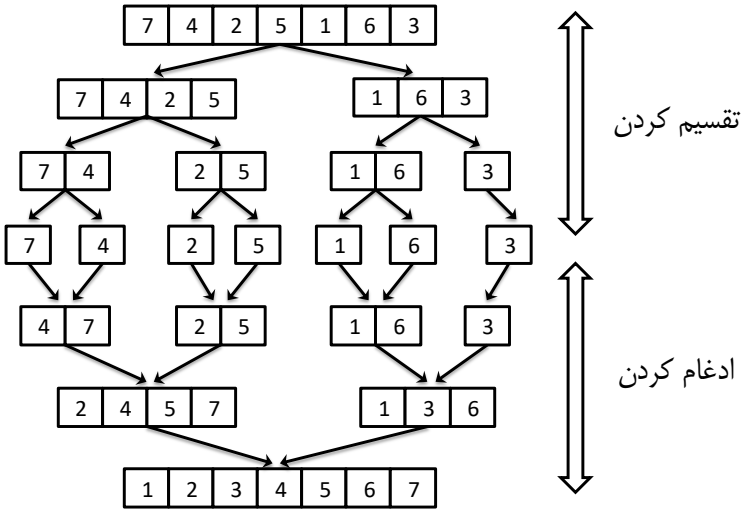
۳- دو زیرمجموعه‌ی مرتب را ادغام می‌کنیم.

اساس کار بر دو قاعده‌ی زیر استوار است:

۱- مرتب کردن مجموعه‌ی کوچک سریع‌تر و ساده‌تر از مرتب کردن مجموعه‌ی بزرگ است.

۲- با ادغام دو مجموعه‌ی مرتب، سریع‌تر می‌توان به یک مجموعه‌ی مرتب رسید (در مقایسه با ادغام دو مجموعه‌ی نامرتب).

شکل ۱۷,۴ نمایی از چگونگی کار مرتب‌سازی ادغامی را نشان می‌دهد. البته مراحل نمایش داده شده در تصویر، روش کار انسان برای مرتب‌سازی ادغامی است.



شکل ۱۷,٤. مراحل مرتب‌سازی مجموعه {7, 4, 2, 5, 1, 6, 3} به روش مرتب‌سازی ادغامی

٤-٤-٤-١- پیاده‌سازی مرتب‌سازی ادغامی

آرایه‌ای با n عنصر را در نظر بگیرید. آرایه را از وسط به دو نیمه تقسیم می‌کنیم. تابع مرتب‌کننده را برای نیمه‌ی اول و دوم این آرایه فراخوانی می‌کنیم؛ سپس این دو نیمه‌ی مرتب شده را توسط تابع ادغام کننده در هم ادغام می‌کنیم. برای سادگی در فهم عملکرد این روش، در آغاز شبه‌کدی برای آن ارائه کرده و سپس با به‌کارگیری

کلاس بردار به پیاده‌سازی عملی آن می‌پردازیم. دو شبه‌کد بعدی، دو تابع مرتب کننده (merge_sort) و ادغام‌کننده (merge) را نشان می‌دهند:

```
array merge_sort(array a) {
    array left, right, result
    if (length(a) <= 1)
        return a
    int half = length(a) / 2
    for (int i = 0; i < half; i++)
        add a[i] to left
    for (int i = half; i < length(a); i++)
        add a[i] to right
    left = merge_sort(left)
    right = merge_sort(right)
    result = merge(left, right)
    return result
}
```

نوع array یک نوع فرضی است که مشخص کننده‌ی نوع اعضای مجموعه‌ی مورد نظر است.

```
array merge(array left, array right) {
    array result
    while (length(left) > 0 && length(right) > 0) {
        if (first(left) <= first(right)) {
            add first(left) to result
                remove first(left)      // remove first element from left
        }
        else {
            add first(right) to result
                remove first(right)     // remove first element from right
        }
    }
    if (length(left) > 0)
        append left to result
    else
        append right to result
    return result
}
```

برای پیاده‌سازی الگوریتم مرتب‌سازی ادغامی، آرایه‌های معمولی سی++ را به کار نمی‌بریم؛ زیرا سی++ بـه آرایـه‌ها به صورت ارجاعی رفتار می‌کند. گزینه‌ای مناسب برای این منظور، کلاس بردار است و از طرفی به صورت مقـداری برای تابع ارسال می‌شود. البته با استفاده از آرایه‌های معمولی، می‌توان پیاده‌سازی سریع‌تری را ارائه کرد. کـد بعـدی این دو تابع را پیاده‌سازی می‌کند. برای آشنایی با توابع کلاس بردار می‌توانید به فصل سوم مراجعه کنید.

```
vector<int> merge_sort(vector<int> a) {
    int half = (a.size()) / 2;
    vector<int> left, right, result;
    if (a.size() <= 1)
        return a;
    for (int i = 0; i<half; i++)
        left.push_back(a[i]);
    for (int i = half; i<a.size(); i++)
        right.push_back(a[i]);
    left = merge_sort(left);
    right = merge_sort(right);
    result = merge(left, right);
    return result;
}
vector<int> merge(vector<int> left, vector<int> right) {
    vector<int> result;
    while (left.size() > 0 && right.size()>0) {
        if (left[0] <= right[0]) {
            result.push_back(left[0]);
            left.erase(left.begin());      // remove first(left)
        }
        else {
            result.push_back(right[0]);
            right.erase(right.begin());    // remove first(right)
        }
    }
    if (left.size() > 0) // append left to result
        result.insert(result.end(), left.begin(), left.end());
    if (right.size() > 0)   // append right to result
        result.insert(result.end(), right.begin(), right.end());
    return result;
}
```

٤-٤-٢- پیچیدگی زمانی مرتب‌سازی ادغامی

برای محاسبه‌ی پیچیدگی زمانی الگوریتم مرتب‌سازی ادغامی، به بررسی تابع بازگشتی ارائه شده می‌پردازیم. اگر زمان اجرای تابع merge_sort را $T(n)$ در نظر بگیریم، ساختار زمانی این تابع به صورت زیر خواهد بود:

$$T(n) = 2T(\frac{n}{2}) + (n-1)$$

$2T(\frac{n}{2})$ برای دو فراخوانی تابع merge_sort توسط خودش که برای مرتب کردن دو نیمه‌ی چپ و راست صورت می‌گیرد. برای ادغام دو مجموعه‌ی مرتب به طول $\frac{n}{2}$ و ایجاد یک مجموعه مرتب به طول n، حداکثر به $n-1$ مقایسه نیاز خواهیم داشت. با استفاده از قضیه مستر، پیچیدگی زمانی مرتب‌سازی ادغامی در حالت میانگین $O(n \log n)$ خواهد بود. بدترین حالت برای این روش زمانی رخ می‌دهد که طول مجموعه‌ی ورودی توانی از ۲ باشد؛ در این حالت، پیچیدگی زمانی مشابه با حالت میانگین است.

٤-٤-٤-٣- پیچیدگی فضایی مرتب‌سازی ادغامی

همان‌طور که از کدهای ارائه شده می‌توان تشخیص داد، مرتب‌سازی ادغامی روشی درجا نیست؛ زیرا مجموعه‌ی مرتب شده جدا از مجموعه‌ی ورودی و اندازه آن وابسته به طول آرایه‌ی ورودی است. در نتیجه، این روش به فضای اضافی به اندازه‌ی طول مجموعه نیاز داشته و پیچیدگی فضایی آن $O(n)$ است. البته می‌توان این روش را به صورت درجا نیز پیاده‌سازی کرد که کمی دشوار بوده و پایداری خود را از دست می‌دهد. روش مرتب‌سازی ادغامی درجا دقیقا شبیه به مرتب‌سازی ادغامی عمل می‌کند با این تفاوت که پیچیدگی فضایی آن به $O(1)$ کاهش یافته و پایدار نخواهد بود.

می‌توان به روش‌هایی این پیچیدگی فضایی ($O(n)$) را کاهش داد؛ برای نمونه می‌توان عنصرها را در یک لیست پیوندی قرار داد و برای مرتب کردن عنصرها، پیوند میان آن‌ها را تغییر داد. این امر دو مزیت زیر را به همراه دارد:

۱- نیازی به جابجایی عنصرها نیست؛ این موضوع به ویژه برای مجموعه‌های بزرگ موثر است.

۲- نیازی به استفاده از فضای اضافی نیست؛ تنها فضای اضافی به کار برده شده، فضای لازم برای نگهداری پیوندها است.

٤-٤-٤-٤- بازگشت

معمول‌ترین راه‌حل برای روش ادغامی، پیاده‌سازی بازگشتی آن است. این نوع پیاده‌سازی، عملکرد الگوریتم را به خوبی نشان داده و فهم آن بسیار ساده است؛ ولی از طرفی دارای مشکلاتی مانند سربار فراخوانی و اشغال فضای اضافی است. پیاده‌سازی این روش با به‌کارگیری راه‌حل‌های غیربازگشتی نیز امکان‌پذیر است که چندان معمول نیستند.

٤-٤-٤-٥- پایداری

مرتب‌سازی ادغامی می‌تواند به صورت پایدار پیاده‌سازی شود. این ویژگی، بستگی به تابع ادغام کننده دارد. تابع ادغام کننده ارائه شده در کد قبلی، به صورت پایدار عمل می‌کند.

٤-٤-٤-٦- مقایسه با روش‌های مرتب‌سازی مرتبط

مرتب‌سازی ادغامی معمولا با دو روش مرتب‌سازی هرمی و سریع مقایسه می‌شود؛ زیرا هر دوی این روش‌ها مانند مرتب‌سازی ادغامی، دارای پیچیدگی زمانی خطی-لگاریتمی هستند.

روش ادغامی حدودا ٣٩٪ تعداد مقایسه‌های کمتری نسبت به روش سریع در حالت میانگین انجام می‌دهد؛ در حالت کلی نیز، تعداد مقایسه‌های روش ادغامی کمتر از روش سریع است؛ مگر موارد محدود، مانند زمانی که بدترین حالت روش ادغامی برابر با بهترین حالت روش سریع قرارگیرد. از طرفی پیاده‌سازی بازگشتی روش ادغامی دارای سربار $1+2+4+...+n=2n-1$ فراخوانی تابع، در بدترین حالت (n توانی از ۲ باشد) است؛ این مقدار دو برابر تعداد فراخوانی‌های تابع در روش سریع (n) است؛ برای حذف این سربار می‌توان، پیاده‌سازی غیربازگشتی را برای روش ادغامی به کار برد که چندان هم دشوار نیست.

مقایسه‌ای خلاصه‌وار میان روش ادغامی و دو روش هرمی و سریع در ادامه ارائه شده است:

۱- فضای اضافی به کار برده شده توسط روش‌های هرمی و سریع کمتر از مرتب‌سازی ادغامی است.

۲- روش ادغامی روشی پایدار است، در صورتی که دو روش دیگر، پایدار نیست.

۳- روش ادغامی به راحتی می‌تواند به صورت موازی پیاده‌سازی شود.
۴- روش ادغامی برای مرتب‌سازی لیست‌های پیوندی و سایر مجموعه‌هایی که فقط دارای دسترسی ترتیبی هستند بسیار مناسب است؛ در صورتی که روش سریع در این موارد بسیار ضعیف عمل کرده و روش هرمی اصلا قادر به مرتب‌سازی در این حالت‌ها نیست.

افزون بر این، روش ادغامی برای مرتب‌سازی برخط نیز به کار برده می‌شود؛ به این شکل که مجموعه‌ی ورودی را دریافت کرده و پس از مرتب‌سازی، آن را در مجموعه‌ی مرتب کنونی ادغام می‌کند؛ این عمل توسط تابع merge صورت می‌گیرد. اگر مجموعه‌ی ورودی کوچک باشد و یا فقط هر بار یک عنصر از ورودی دریافت شود، این روش از نظر هزینه‌ی زمانی و فضایی مناسب نیست؛ در این حالت، راه‌حلی بهتر، به‌کارگیری یک درخت جستجوی دودویی متوازن است.

٤-٤-٤-۷- جمع‌بندی

جدول ۸,۴ موضوعات اصلی را برای این روش دسته‌بندی می‌کند.

جدول ۸,٤. ویژگی‌های اصلی روش مرتب‌سازی ادغامی

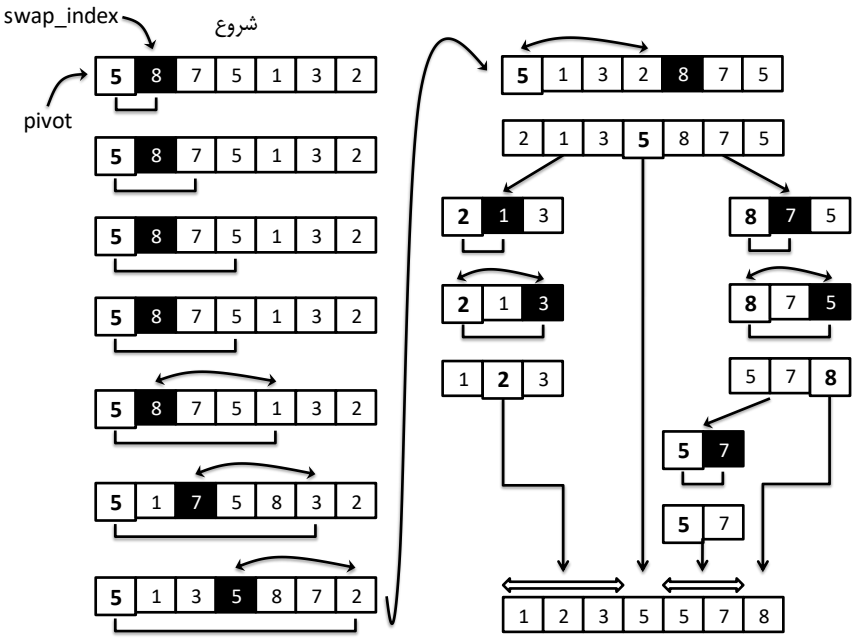
موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n \log n)$
پیچیدگی زمانی در بدترین حالت	$O(n \log n)$
پیچیدگی فضایی	$O(n)$
پایداری	بلی

٤-٤-۵- مرتب‌سازی سریع

مرتب‌سازی سریع از شناخته‌شده‌ترین روش‌های مرتب‌سازی است. این روش مرتب‌سازی، در عمل از دیگر روش‌های مرتب‌سازی که از درجه‌ی خطی-لگاریتمی برخوردارند، سریع‌تر عمل می‌کند؛ به همین دلیل، آن را مرتب‌سازی سریع می‌نامند. مرتب‌سازی این روش مانند روش ادغامی از طریق تقسیم و غلبه انجام می‌شود؛ به این صورت که مجموعه‌ای اصلی را به دو مجموعه‌ی کوچک‌تر تقسیم کرده و به روش بازگشتی هریک از آن‌ها را مرتب می‌کند.
روش سریع بر پایه‌ی انتخاب عنصر محوری عمل می‌کند. عنصر محوری می‌تواند هر یک از اعضای مجموعه ورودی باشد که معمولا عنصر اول به عنوان عنصر محوری در نظر گرفته می‌شود. البته انتخاب عنصر محوری به این شکل، کارایی روش سریع را برای مجموعه‌های مرتب و نزدیک به مرتب کاهش خواهد داد.
اساس کار روش سریع به مراحل زیر تقسیم می‌شود:
۱- انتخاب عنصر محوری
۲- انتقال همه‌ی عناصر کوچک‌تر از عنصر محوری به سمت چپ آن و انتقال همه‌ی عناصر بزرگ‌تر از آن به سمت راست

۳- مرتب‌کردن بازگشتی مجموعه‌ی سمت چپ عنصر محوری و مجموعه‌ی سمت راست آن

اصلی که مرتب‌سازی سریع بر آن استوار است، مرتب بودن مجموعه‌های با طول یک و صفر است (مانند مرتب‌سازی ادغامی). شکل ۱۸,۴ مراحل انجام شده توسط این روش را برای یک مجموعه‌ی هفت عضوی نمایش می‌دهد.



شکل ۱۸,٤. مراحل مرتب‌سازی مجموعه {5, 8, 7, 5, 1, 3, 2} به روش مرتب‌سازی سریع

٤-٤-٥-١- پیاده‌سازی مرتب‌سازی سریع

برای فهم ساده‌تر این الگوریتم، ابتدا شبه‌کدی را برای آن ارائه کرده و سپس به پیاده‌سازی آن می‌پردازیم. در این شبه‌کد، مراحل زیر را دنبال خواهیم کرد:

۱- عنصر محوری را انتخاب و از آرایه حذف می‌کنیم.

۲- عناصر کوچک‌تر از عنصر محوری را در آرایه‌ی smaller و عناصر بزرگ‌تر از عنصر محوری را در آرایه‌ی greater قرار می‌دهیم.

۳- به صورت بازگشتی دو آرایه‌ی smaller و greater را مرتب می‌کنیم.

۴- در پایان، آرایه‌ای به طول آرایه‌ی ورودی در نظر گرفته و آرایه‌ی smaller را در آغاز و آرایه‌ی greater را در پایان آن قرار می‌دهیم. عنصر محوری نیز در میان این دو آرایه قرار می‌گیرد. آرایه‌ی مرتب شده را برمی‌گردانیم.

این مراحل در شبه کد زیر نشان داده شده‌اند:

```
void quick_sort(array a) {
    array smaller, greater
    if (length(a) <= 1)
        return a

    for (int i = 0; i < length(a); i++) {
        if (a[i] <= pivot)
            append a[i] to smaller
        else
            append a[i] to greater
    }
    return concatenate(quicksort(smaller), pivot, quicksort(greater))
}
```

شبه‌کد بالا مانند روش ادغامی، دارای پیچدگی فضایی $O(n)$ است؛ زیرا مجموعه‌ی خروجی جدا از مجموعه‌ی ورودی است. البته فضای اشغال شده توسط مجموعه‌های smaller و greater را نادیده می‌گیریم. ولی پیاده‌سازی اصلی که در ادامه ارائه می‌کنیم، این فضای اضافی را به کار نمی‌برد.

برای پیاده‌سازی الگوریتم مرتب‌سازی سریع، دو تابع در نظر گرفته‌ایم:

۱- تابع partition: برای انتقال عنصرهای کوچک‌تر از عنصر محوری به سمت چپ آن و عنصرهای بزرگ‌تر از عنصر محوری به سمت راست آن.

۲- تابع quick_sort: برای مرتب‌کردن بازگشتی آرایه‌ی ورودی.

کد زیر پیاده‌سازی تابع partition را نشان می‌دهد:

```
int partition(int left, int right) {
    int pivot = a[left];
    int swap_index = left + 1;
    for (int i = left + 1; i<right; i++)
        if (a[i]<pivot)
            swap(a[i], a[swap_index++]);
    // insert pivot in it's correct position
    swap(a[swap_index - 1], a[left]);
    return (swap_index - 1);// return pivot's correct position
}
```

آرایه‌ی ورودی (a) را به صورت سراسری در نظر گرفته‌ایم. تابع partition ابتدا عناصر کوچک‌تر از عنصر محوری را به سمت چپ آن انتقال می‌دهد؛ سپس عنصر محوری را در مکان مناسب خود (پس از عنصرهای کوچک‌تر از آن) قرار داده و نمایه‌ی این مکان را برمی‌گرداند.

در شکل ۱۸,۴ اعداد پررنگ، نشان‌دهنده‌ی عنصر محوری هستند؛ نمایه‌ی خانه‌های مشکی رنگ نیز در واقع معادل متغیر swap_index در کد بالا هستند.

پیاده‌سازی تابع quick_sort در این کد انجام شده است:

```
void quick_sort(int left, int right) {
    // if length(array) <= 1 then it is already sorted
    if (left >= right)
        return;
    int pivotindex = partition(left, right); // partion current array
    quick_sort(left, pivotindex);            // quick sort left array
    quick_sort(pivotindex + 1, right);       // quick sort right array
}
```

٤-٤-٥-٢- پیچیدگی زمانی مرتب‌سازی سریع

تابع partition حداکثر در زمان $O(n)$ کار خود را به پایان می‌رساند. کارایی تابع quick_sort بر اساس چیدمان عناصری که در مجموعه‌ی ورودی قرار دارند، تعیین می‌شود. چیدمان عنصرها، می‌تواند منجر به بهترین حالت یا بدترین حالت مرتب‌سازی سریع شود. در بهترین حالت، تابع quick_sort در هر مرحله، مجموعه را به دو بخش تقریبا برابر تقسیم می‌کند. اگر زمان اجرای تابع quick_sort را $T(n)$ درنظر بگیریم، ساختار زمانی الگوریتم در بهترین حالت به این صورت خواهد بود:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

بر اساس این رابطه، عمق فراخوانی بازگشتی $\log n$ است؛ در نتیجه، پیچیدگی زمانی مرتب‌سازی سریع در بهترین حالت $O(n \log n)$ خواهد بود. بدترین حالت زمانی رخ می‌دهد که مجموعه‌ی ورودی در جهت عکس مرتب شده باشد. در این حالت تابع quick_sort مجموعه‌ی ورودی را هر بار به دو زیرمجموعه با طول صفر و $n-1$ تقسیم می‌کند. بر این اساس، ساختار زمانی زیر را خواهیم داشت:

$$T(n) = T(0) + T(n-1) + O(n) = T(n-1) + O(n)$$

در این فرمول، عمق فراخوانی بازگشتی n است و در نتیجه پیچیدگی زمانی مرتب‌سازی سریع در بدترین حالت $O(n^2)$ خواهد بود. پیچیدگی زمانی در حالت میانگین، میان بهترین و بدترین حالت متغیر بوده و برابر با $O(n \log n)$ است. برای کاهش احتمال رخ دادن بدترین حالت، باید در انتخاب عنصر محوری دقت کرد. در این راستا، روش‌هایی برای انتخاب بهتر عنصر محوری ارائه شده است. برای نمونه آقای وِرث[۳۰] انتخاب عنصر میانی را به عنوان عنصر محوری پیشنهاد کرده است. یکی از روش‌های دیگر ارائه شده، انتخاب عنصر میانه از سه عنصر مجموعه است؛ در این روش، میانه‌ی سه عنصر اول، وسط و آخر مجموعه به عنوان عنصر محوری برگزیده می‌شود.

[۳۰] Niklaus Wirth

٤-٤-٥-٣- پیچیدگی فضایی مرتب‌سازی سریع

شبه‌کد ارائه شده دارای پیچیدگی فضایی $O(n)$ است. در پیاده‌سازی اصلی به صورت درجا عمل کرده و پیچیدگی فضایی را با حذف فضای اضافی به $O(1)$ کاهش دادیم. البته اگر سربار فراخوانی بازگشتی را نیز در نظر بگیریم، پیچیدگی فضایی در بهترین حالت $O(\log n)$ خواهد بود که در بدترین حالت به $O(n)$ نیز می‌رسد.

٤-٤-٥-٤- بازگشت

راه‌حل بازگشتی، روشی بسیار معمول برای پیاده‌سازی مرتب‌سازی سریع است. به شکلی که معمولا روش غیربازگشتی برای پیاده‌سازی این روش به کار برده نمی‌شود. ساختار بازگشتی این روش، اجازه‌ی اجرای موازی آن را می‌دهد؛ در ضمن، برای اجرای موازی نیاز به همگام‌سازی نخواهیم داشت، زیرا زیرمجموعه‌ها به یکدیگر وابسته نیستند. در نتیجه هر نخ می‌تواند به صورت جداگانه اجرا و به پایان برسد. هنگام پایان یافتن همه‌ی نخ‌ها، مجموعه‌های مرتب شده را از آغاز به هم متصل می‌کنیم. در این حالت پیچیدگی زمانی از درجه‌ی خطی ($O(n)$) خواهد بود که بسیار عالی است.

٤-٤-٥-٥- پایداری

اغلب پیاده‌سازی‌های مرتب‌سازی سریع پایدار نبوده و ترتیب عنصرهای یکسان را حفظ نمی‌کنند. پیاده‌سازی ارائه شده نیز ناپایدار است. این موضوع یک ضعف نسبت به روش‌های مرتب‌سازی خطی-لگاریتمی پایدار محسوب می‌شود.

٤-٤-٥-٦- انواع مختلف

روشی برای به‌کارگیری مزایای هر دو روش هرمی و سریع ارائه شده است که مرتب‌سازی درون‌گرا نام دارد. اساس کار این روش، به‌کارگیری پیچیدگی زمانی روش هرمی در بدترین حالت و پیچیدگی زمانی روش سریع در حالت میانگین است. هر چند این روش با هر دو روش نامبرده در ارتباط است ولی در این بخش به توضیح آن پرداخته‌ایم.

٤-٤-٥-٦-١- مرتب‌سازی درون‌گرا

مرتب‌سازی درون‌گرا در واقع یک روش جدید نیست، بلکه ترکیبی از دو روش هرمی و سریع است. این روش در حالت پیش‌فرض مرتب‌سازی سریع را به کار می‌برد؛ اگر در هنگام مرتب‌سازی، عمق بازگشت از $x \log n$ تجاوز کند، روش مرتب‌سازی تعویض شده و به مرتب‌سازی هرمی تبدیل می‌گردد. x مقداری ثابت است که بسته به کارایی مورد نیاز برنامه‌نویس تعیین می‌شود.

این روش و روش عنصر میانه از سه عنصر، برای یک مجموعه با ۱۰۰۰۰۰ هزار عضو، آزمایش شده‌اند؛ براساس این آزمایش، سرعت اجرای روش درون‌گرا، ۱٫۲ برابر بهتر از روش دیگر بود. این روش مانند دو روش هرمی و سریع ناپایدار است. ویژگی‌های اصلی این روش در جدول ۹٫۴ نشان داده شده است.

جدول ٩,٤. ویژگی‌های اصلی روش مرتب‌سازی درون‌گرا

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n \log n)$
پیچیدگی زمانی در بدترین حالت	$O(n \log n)$
پیچیدگی فضایی	$O(\log n)$
پایداری	خیر

٤-٤-٥-٧- مقایسه با روش‌های مرتب‌سازی مرتبط

مرتب‌سازی هرمی سخت‌ترین رقیب مرتب‌سازی سریع است. در مجموع، روش سریع زمان اجرای بهتری نسبت به روش هرمی دارد ولی پیچیدگی زمانی درجه‌ی دوم روش سریع در بدترین حالت، موجب برتری روش هرمی در این مورد می‌گردد.

مرتب‌سازی ادغامی نیز از روش‌هایی است که با این دو روش مقایسه می‌شود. این روش نیز در بدترین حالت زمان اجرای بهتری نسبت به مرتب‌سازی سریع دارد. از طرف دیگر، پایداری روش ادغامی و کارایی بالای آن در مرتب کردن لیست‌های پیوندی و مجموعه‌هایی با دستیابی ترتیبی، موجب برتری نسبی آن نسبت به روش سریع می‌شود. البته روش سریع را هم می‌توان برای لیست‌های پیوندی به کار گرفت ولی به دلیل دستیابی ترتیبی در لیست‌های پیوندی، هزینه‌ی انتخاب عنصر محوری مناسب در این حالت، افزایش پیدا کرده و در نتیجه کارایی کاهش می‌یابد.

٤-٤-٥-٨- جمع‌بندی

جدول ۱۰,۴ موضوعات اصلی را برای این روش دسته‌بندی می‌کند.

جدول ١٠,٤. ویژگی‌های اصلی روش مرتب‌سازی سریع

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n \log n)$
پیچیدگی زمانی در بدترین حالت	$O(n^2)$
پیچیدگی فضایی	$O(\log n)$
پایداری	خیر

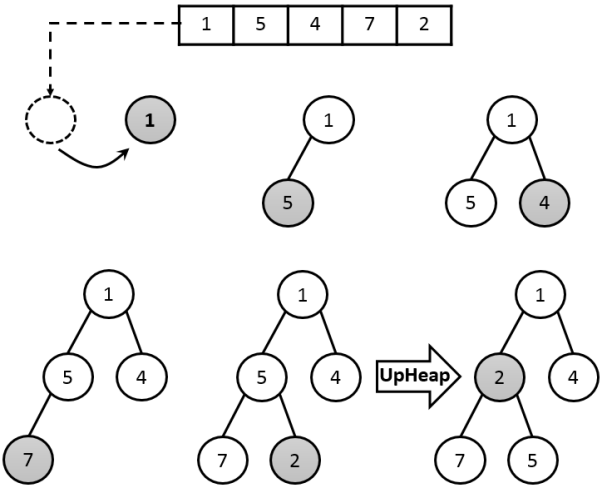
٤-٤-٦- مرتب‌سازی هرمی

روش هرمی که از روش‌های بسیار کاربردی است، یکی از انواع مرتب‌سازی انتخابی به حساب می‌آید. این روش مانند روش انتخابی عمل می‌کند؛ با این تفاوت که برای انتخاب بزرگترین یا کوچکترین عنصر، از هرم بهره می‌برد. این روش ابتدا ورودی را به شکل یک هرم بیشینه یا هرم کمینه درآورده و سپس در هر مرحله عنصر ریشه (بیشترین/کمترین عنصر) را از آن حذف می‌کند؛ این فرآیند را تا زمانی که همه‌ی عنصرهای مجموعه مرتب شوند، ادامه می‌دهد. از آنجا که هدف ما، مرتب‌سازی صعودی است، بنابراین باید داده‌های ورودی را به کمینه تبدیل کنیم (چرا؟).

این مجموعه‌ی ۵ عضوی را در نظر بگیرید:

1, 5, 4, 7, 2

برای مرتب‌کردن این مجموعه، ابتدا آن را به صورت زیر به یک هرم کمینه تبدیل می‌کنیم (شکل ۱۹,۴). عمل بالا هرمی که در شکل دیده می‌شود، در واقع جزئی از فرآیند هرم‌سازی است که در بخش مربوط به هـرم توضـیح داده شد.



شکل ۱۹,٤. بخشی از عملیات مرتب‌سازی مجموعه {1, 5, 4, 7, 2} با روش مرتب‌سازی هرمی

٤-٤-٦-١- پیاده‌سازی مرتب‌سازی هرمی

برای مرتب‌سازی هرمی، این مراحل را به ترتیب بر روی آرایه‌ی ورودی انجام می‌دهیم. پیاده‌سازی درخت با استفاده از آرایه در فصل پیش توضیح داده شد. منظور از ریشه، خانه‌ی اول آرایه است.

۱- آرایه‌ی ورودی را به هرم کمینه تبدیل می‌کنیم.
۲- عنصر ریشه را حذف کرده و در آرایه‌ی خروجی قرار می‌دهیم.
۳- عنصر پایانی آرایه را در ریشه قرار می‌دهیم.
۴- آرایه‌ی باقیمانده را توسط فرآیند پایین هرمی دوباره به هرم تبدیل می‌کنیم.
۵- مراحل ۲ تا ۴ را تا زمانی که طول آرایه به صفر برسد، ادامه می‌دهیم.

مراحل توضیح داده شده، آرایه‌ی ورودی را به صورت صعودی مرتب می‌کند. روش توضیح داده شـده درجـا نیسـت. برای ارائه‌ی راه‌حلی درجا، تنها از خود آرایه‌ی ورودی بهره می‌بریم؛ برای این منظور، عنصر ریشه را پس از حذف در خانه‌ی پایانی آرایه قرار می‌دهیم؛ ریشه‌ی حذف شده‌ی بعدی را در خانه‌ی یکی قبل از آخر قرار مـی‌دهیم؛ اگـر بـه همین صورت ادامه دهیم، در پایان آرایه‌ای مرتب شده به صورت نزولی خواهیم داشت.

توابع upheap، downheap و remove_heap در فصل پیش توضیح داده شدند. سه تابع معرفـی شـده را نیـز بـا تغییری جزئی دوباره می‌نویسیم. برای پیاده‌سازی ساده‌تر، از بردار به جای آرایه‌ی معمولی بهره می‌بریم. آرایه ورودی را به صورت عمومی به این صورت تعریف می‌کنیم:

```
vector<int> heap;
```

سه تابع upheap، downheap و remove_heap را به این صورت بازنویسی می‌کنیم:

```
void downheap(int length) {
    int left_child, right_child;
    int index = 0;
    int parent = heap[index];
    while (length>(index * 2 + 1)) {
        right_child = parent - 1;
        left_child = heap[index * 2 + 1];
        if (length>(index * 2 + 2))
            right_child = heap[index * 2 + 2];
        if (parent<max(left_child, right_child)) {
            if (left_child>right_child) {
                swap(heap[index], heap[index * 2 + 1]);
                index = index * 2 + 1;
            }
            else {
                swap(heap[index], heap[index * 2 + 2]);
                index = index * 2 + 2;
            }
        }
        else
            break;
    }
}
```

```
void upheap(int index) {
    int child = heap[index];
    int parent = 0;
    while (index > 0) {
        parent = heap[(index - 1) / 2];
        if (child > parent)
            swap(heap[index], heap[(index - 1) / 2]);
        else
            break;
        index /= 2;
    }
}
```

کاربرد پارامتر ورودی length برای مرتب‌کردن درجا است. زیرا عناصری که در پایان آرایه قرار گرفته‌اند، در واقع حذف شده در نظر گرفته می‌شوند؛ بنابراین با کم کردن مقدار این پارامتر، تابع downheap را بر روی بقیه‌ی آرایه اعمال می‌کنیم.

```
void remove_heap(int length) {
    int max_heap = heap[0];
    heap[0] = heap[length];
    heap[length] = max_heap;
}
```

این تابع عنصر ریشه را که در نمایه‌ی صفر قرار دارد، حذف کرده و در خانه‌ای با نمایه‌ی length قرار می‌دهد. پیش از مرتب کردن آرایه‌ی ورودی باید، ابتدا آن را هرم‌سازی کنیم. به این منظور تابع heapify را با استفاده از تابع upheap به این شکل پیاده‌سازی می‌کنیم:

```
void heapify() {
    for (int i = 1; i < heap.size(); i++)
        upheap(i);
}
```

پس از تبدیل آرایه‌ی ورودی به heap و با به‌کارگیری توابع معرفی شده، به شکل زیر آن را مرتب می‌کنیم:

```
void heap_sort() {
    heapify();
    for (int i = heap.size() - 1; i > 0; i--) {
        remove_heap(i);
        downheap(i);
    }
}
```

تابع heap_sort پس از فراخوانی تابع heapify، در هر دور از حلقه، عنصر ریشه را از heap حذف کرده و در پایان آرایه قرار می‌دهد. پس از $n-1$ مرتبه تکرار این چرخه، آرایه مرتب خواهد شد. برای نمونه، کد بعدی آرایه‌ی ۵ عضوی معرفی شده را مرتب کرده و در خروجی چاپ می‌کند:

```
#include <iostream>
#include <vector>
using namespace std;
int arr[] = {1, 5, 4, 7, 2};
vector<int> heap(arr, arr + 5);
int main() {
    heap_sort();
    for (int i = 0; i < heap.size(); i++)
        cout << heap[i] << " ";
    return 0;
}
```

خروجی حاصل از اجرای کد قبل، به این صورت خواهد بود:

Output
1 2 4 5 7

٤-٤-٦-٢- پیچیدگی زمانی مرتب‌سازی هرمی

برای تحلیل پیچیدگی زمانی روش مرتب‌سازی هرمی به تحلیل سه بخش اصلی تابع heap_sort می‌پردازیم. پیچیدگی زمانی دو تابع upheap و downheap در بدترین حالت $O(\log n)$ است.

۱- تابع heapify: این تابع برای تبدیل آرایه‌ی ورودی به heap، به تعداد n مرتبه (طول آرایه‌ی ورودی) تابع upheap را فراخوانی می‌کند. بنابراین پیچیدگی زمانی اجرای آن $n \log n$ خواهد بود.

۲- تابع remove_heap: عنصر ریشه را حذف و در پایان آرایه قرار می‌دهد. زمان اجرای این تابع ثابت است ($O(1)$).

۳- تابع downheap: پیچیدگی اجرایی این تابع $\log n$ است؛ به علت وجود این تابع در یک حلقه که خود به تعداد n مرتبه تکرار می‌شود، پیچیدگی اجرایی این بخش از کد $n \times (O(1) + O(\log n)) = O(n \log n)$ خواهد بود.

بر پایه‌ی توضیحات داده شده پیچیدگی اجرایی روش مرتب‌سازی هرمی $O(n \log n) + O(n \log n) = O(2n \log n)$ است. از آنجا که پیچیدگی دو تابع downheap و upheap در بدترین حالت درنظر گرفته شدند، پیچیدگی ذکر شده مربوط به بدترین حالت مرتب‌سازی هرمی است که در واقع از درجه‌ی خطی-لگاریتمی و برابر با $O(n \log n)$ است. محاسبه‌ی حالت میانگین برای این روش از قاعده مشخصی پیروی نمی‌کند ولی با اطمینان می‌توان گفت که در بهترین حالت بهتر از $O(n \log n)$ نخواهد بود.

٤-٤-٦-٣- پیچیدگی فضایی مرتب‌سازی هرمی

پیاده‌سازی ارائه شده به صورت درجا عمل می‌کند. بنابراین پیچیدگی فضایی برابر با $O(1)$ است. این پیچیدگی فضایی برای روش هرمی که از درجه‌ی خطی-لگاریتمی است، یک امتیاز ویژه محسوب می‌شود.

٤-٤-٦-٤- پایداری

یکی از معایب مرتب‌سازی هرمی پایدار نبودن آن است. این روش به دلیل ساختار درونی هرم نمی‌تواند به صورت پایدار عمل کند.

٤-٤-٦-٥- مقایسه با روش‌های مرتب‌سازی مرتبط

مرتب‌سازی سریع و ادغامی از رقبای این روش محسوب می‌شوند. در حالت کلی روش سریع به دلیل به‌کارگیری بهتر از حافظه‌ی نهان و سایر عوامل مربوطه، بهتر از این روش عمل می‌کند؛ از طرف دیگر، پیچیدگی زمانی خطی-لگاریتمی این روش در بدترین حالت، بهتر از پیچیدگی زمانی روش سریع در بدترین حالت است. همین برتری موجب می‌شود در سیستم‌های بلادرنگ که سرعت حرف اول را می‌زند، روش هرمی به کار گرفته شود. روش هرمی

از جنبه‌ی پیچیدگی فضایی از هر دو روش دیگر بهتر است. ولی روش ادغامی کاربردهای ویژه‌ای دارد که هیچیک از دو روش دیگر دارا نیستند.

٤-٤-٦-٦- جمع‌بندی

جدول ۱۱,۴، موضوعات اصلی را برای این روش دسته‌بندی می‌کند.

جدول ۱۱,٤. ویژگی‌های اصلی روش مرتب‌سازی هرمی

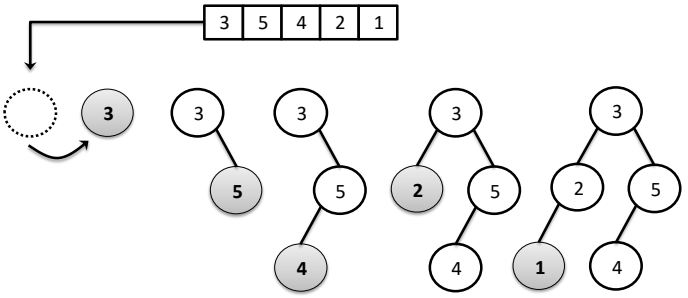
موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n \log n)$
پیچیدگی زمانی در بدترین حالت	$O(n \log n)$
پیچیدگی فضایی	$O(1)$
پایداری	خیر

٤-٤-٧- مرتب‌سازی درخت دودویی

مرتب‌سازی درخت دودویی روشی است که با به‌کارگیری درخت دودویی به مرتب‌سازی می‌پردازد. این روش ابتدا عنصرهای ورودی را در یک درخت دودویی قرار داده و سپس با یک پیمایش میانوندی، عنصرها را به ترتیب صعودی استخراج می‌کند. برای نمونه این مجموعه‌ی ۵ عضوی را درنظر بگیرید:

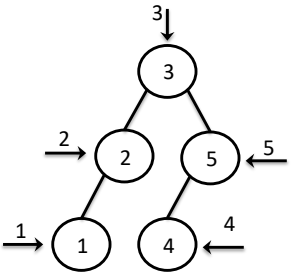
3, 5, 4, 2, 1

شکل ۲۰,۴ درخت جستجوی دودویی ساخته شده از این مجموعه را نشان می‌دهد.



شکل ۲۰,٤. درخت جستجوی دودویی ساخته شده از مجموعه {3, 5, 4, 2, 1}

حال اگر درخت جستجوی دودویی ساخته شده در شکل ۲۰,۴ را به روش میانوندی پیمایش می‌کنیم، نتیجه به صورت نشان داده شده در شکل ۲۱,۴ خواهد بود. شماره‌ها ترتیب دیدن گره‌ها را مشخص می‌کنند.



شکل ٤,٢١. نتیجه پیمایش درخت دودویی شکل ٤,٢٠ به روش میانوندی

٤-٤-٧-١- پیاده‌سازی مرتب‌سازی درخت دودویی

درخت جستجوی دودویی در فصل‌های قبل معرفی شد. توابع معرفی شده برای این ساختمان داده را در این بخش به کار خواهیم برد. برای پیاده‌سازی این الگوریتم، عنصرهای ورودی را دریافت کرده و در یک درخت جستجوی دودویی قرار می‌دهیم؛ سپس با یک پیمایش میانوندی عنصرها را به صورت مرتب در یک آرایه قرار می‌دهیم. از لیست‌پیوندی برای پیاده‌سازی درخت دودویی بهره می‌بریم. برای سادگی، آرایه‌ی ورودی و لیست پیوندی را به صورت عمومی تعریف کرده‌ایم.

```
int arr[] = {1, 5, 4, 7, 2, 6, 3, 9};
BST* binary_search_tree;
```

حال تابع binary_tree_sort را به این صورت پیاده‌سازی می‌کنیم:

```
void binary_tree_sort(int* a, int length) {
    BST* item;
    // insert items into binary search tree
    for (int i = 0; i < length; i++) {
        item = new BST();
        item->value = a[i];
        insert_bst(binary_search_tree, item);
    }
    // inorder traversal of items : SORT
    inorder_bst(binary_search_tree);
}
```

حلقه‌ی for، همه‌ی عنصرهای آرایه را در درخت جستجوی دودویی قرار می‌دهد و تابع inorder_bst عنصرها را به صورت میانوندی پیمایش کرده و در خروجی چاپ می‌کند. این تابع قبلا معرفی شده است. برای به‌کارگیری تابع بالا می‌توانیم به این صورت عمل کنیم:

```
int main() {
    binary_tree_sort(arr, sizeof(arr) / sizeof(int));
    return 0;
}
```

خروجی حاصل از اجرای کد بالا برای آرایه‌ی معرفی شده به این صورت خواهد بود:

Output
1 2 3 4 5 6 7 9

٤-٤-٧-٢- پیچیدگی زمانی مرتب‌سازی درخت دودویی

قرار دادن یک عنصر در درخت جستجوی دودویی در بدترین حالت برابر با $O(n)$است. این حالت زمانی رخ می‌دهد که مجموعه‌ی ورودی مرتب باشد. بنابراین در بدترین حالت، قرار دادن n عنصر در درخت جستجوی دودویی، از مرتبه $O(n^2)$ خواهد بود. پیچیدگی اجرایی پیمایش میانوندی درخت دودویی نیز $O(n)$ است. در نتیجه پیچیدگی کل مرتب‌سازی درخت دودویی در بدترین حالت از درجه‌ی دوم است.

$$O(n^2) + O(n) \in O(n^2)$$

پیچیدگی زمانی در حالت میانگین $O(n \log n)$ است. زیرا در این حالت، قرار دادن عنصرها در درخت جستجوی دودویی از درجه‌ی لگاریتمی ($O(\log n)$) است.

$$O(n \log n) + O(n) \in O(n \log n)$$

برای بهبود بخشیدن به پیچیدگی اجرایی در بدترین حالت، می‌توانیم درخت جستجوی دودویی متوازن را به کار ببریم. در این صورت قرار دادن عنصرها در درخت جستجوی سریع‌تر صورت گرفته و در بدترین حالت از درجه‌ی لگاریتمی خواهد بود.

٤-٤-٧-٣- پیچیدگی فضایی مرتب‌سازی درخت دودویی

پیاده‌سازی انجام شده در بخش پیش، درجا نبوده و از پیچیدگی فضایی $O(n)$ برخوردار است؛ زیرا فضای اشغال شده توسط لیست پیوندی برابر با طول مجموعه‌ی ورودی است. می‌توان راه‌حلی درجا نیز برای این الگوریتم ارائه کرد که پیاده‌سازی آن کمی دشوارتر است.

٤-٤-٧-٤- پایداری

پیاده‌سازی معمول درخت جستجوی دودویی، اجازه‌ی قرار دادن عنصرهای تکراری را نمی‌دهد. بنابراین در حالت معمول، این روش پایدار نیست. کلاس multiset در کتابخانه‌ی استاندارد سی++ توسط درخت جستجوی دودویی پیاده‌سازی شده و اجازه‌ی قرار دادن عنصرهای تکراری را نیز می‌دهد.

٤-٤-٧-٥- مقایسه با روش‌های مرتب‌سازی مرتبط

این روش مشابه روش هرمی بوده و کارایی هر دو روش مشابه یکدیگر است. به جستجوی درخت دودویی اصطلاحا روش مرتب‌سازی تند نیز گفته می‌شود زیرا سرعت بالایی در مرتب‌سازی داده‌ها دارد.

٤-٤-٧-٦- جمع‌بندی

جدول ۱۲,۴ ویژگی‌های اصلی این روش را نشان می‌دهد.

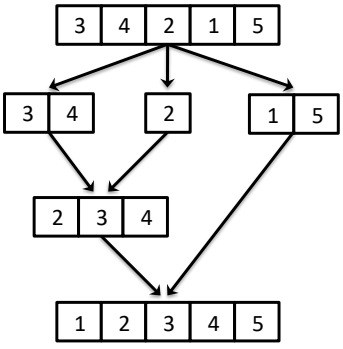
جدول ۱۲,٤. ویژگی‌های اصلی روش مرتب‌سازی درخت دودویی

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n \log n)$
پیچیدگی زمانی در بدترین حالت	$O(n^2)$
پیچیدگی فضایی	$O(n)$
پایداری	خیر

٤-٤-٨- مرتب‌سازی پیوندی

مرتب‌سازی پیوندی از روش‌های جالب و ساده است. اساس کار این روش، استخراج زیرمجموعه‌های مرتب از مجموعه‌ی اصلی و ادغام آن‌ها در یکدیگر است. نام این روش از لغت strand یعنی به‌هم بافتن و پیوند دادن گرفته شده، زیرا داده‌های مرتب شده را استخراج کرده و در کنار هم قرار می‌دهد؛ سپس همه‌ی زیرمجموعه‌های مرتب استخراج شده را به هم پیوند می‌دهد. مرتب‌سازی پیوندی مانند مرتب‌سازی ادغامی از ادغام مجموعه‌های مرتب بهره می‌گیرد. شکل ۲۲,۴ مراحل انجام شده توسط این روش را برای مجموعه‌ی ۵ عضوی زیر نشان می‌دهد:

3, 4, 2, 1, 5



شکل ۲۲,٤. مراحل مرتب‌سازی مجموعه {3, 4, 2, 1, 5} به روش مرتب‌سازی پیوندی

٤-٤-٨-١- پیاده‌سازی مرتب‌سازی پیوندی

الگوریتم مرتب‌سازی پیوندی از دو بخش زیر تشکیل شده است:

۱- استخراج زیرمجموعه‌های مرتب

۲- پیوند زیرمجموعه‌های مرتب با یکدیگر

زیرمجموعه‌های مرتب استخراج شده از مجموعه‌ی اصلی حذف خواهند شد. کار تا جایی ادامه پیدا می‌کند که مجموعه‌ی اصلی خالی شود. شبه‌کد بعدی مراحل مذکور را روشن‌تر می‌سازد:

```
array strand_sort(array a) {
    array sublist, result
    while (length(a) > 0) {
        sublist[0] = a[0]
        remove a[0]
        for (int i = 0; i < length(a); i++) {
            if (a[i] >= last(sublist)) {
                append a[i] to sublist
                    remove a[i]
            }
        }
        result = merge(result, sublist)
        clear sublist
    }
    return result
}
```

تابع merge دقیقا مشابه پیاده‌سازی انجام شده در بخش مرتب‌سازی ادغامی است. برای پیاده‌سازی اصلی این کلاس، از بردار استفاده کرده‌ایم. تابع strand_sort را به صورت زیر می‌نویسیم. قبلا با کارایی تابع advance آشنا شده‌ایم. تابع نوشته شده، آرایه‌ی مرتب شده را در پایان برمی‌گرداند.

```
vector<int> strand_sort(vector<int> a) {
    vector<int> result, sublist;
    vector<int>::iterator remove_index;
    while (a.size() > 0) {
        sublist.push_back(a[0]);
        a.erase(a.begin());
        for (int i = 0; i < a.size(); i++) {
            if (a[i] >= sublist[sublist.size() - 1]) {
                sublist.push_back(a[i]);// append a[i] to sublist
                                        // use this iterator for removing
item i
                remove_index = a.begin();
                advance(remove_index, i);   // remove_index +=i
                a.erase(remove_index);      // remove a[i]
            }
        }
        result = merge(result, sublist);
        sublist.clear();
    }
    return result;
}
```

٤-٤-٨-٢- پیچیدگی زمانی مرتب‌سازی پیوندی

بدترین حالت برای این روش، زمانی رخ می‌دهد که مجموعه در جهت عکس مرتب شده باشد؛ در این حالت پیچیدگی زمانی $O(n^2)$ خواهد بود. بهترین حالت نیز برای مجموعه‌ی مرتب رخ می‌دهد ولی در جهت درست. در

این حالت پیچیدگی زمانی $O(n)$ را خواهیم داشت. پیچیدگی زمانی در حالت میانگین از درجه‌ی خطی-لگاریتمی ($O(n\log n)$) است.

٤-٤-٨-٣- پیچیدگی فضایی مرتب‌سازی پیوندی

مرتب‌سازی پیوندی به صورت درجا عمل نمی‌کند. مجموعه‌ی خروجی جدا از مجموعه‌ی ورودی است؛ بنابراین پیچیدگی فضایی برابر با $O(n)$ است.

٤-٤-٨-٤- پایداری

این روش پایدار بوده و پیاده‌سازی ارائه شده نیز به صورت پایدار عمل می‌کند.

٤-٤-٨-٥- مقایسه با روش‌های مرتب‌سازی مرتبط

این روش را می‌توان با روش ادغامی مقایسه کرد، زیرا از نظر عملکرد تا حدودی شبیه به هم عمل کرده و هر دو جزء دسته‌ی روش‌های مرتب‌سازی ادغامی قرار می‌گیرند. هر دو روش کارایی بالایی در مرتب‌سازی لیست‌های پیوندی دارند. پیچیدگی زمانی درجه‌ی دوم در بدترین حالت برای روش پیوندی یک ضعف به حساب می‌آید. یکی از مزایای روش پیوندی در مرتب‌سازی مجموعه‌های نزدیک به مرتب است؛ زیرا در پیمایش نخست، بزرگترین زیرمجموعه‌ی مرتب را استخراج کرده و از مجموعه‌ی اصلی حذف می‌کند؛ این امر سرعت اجرای آن را افزایش می‌دهد.

٤-٤-٨-٦- جمع‌بندی

جدول ۱۳,۴ ویژگی‌های اصلی این روش را نشان می‌دهد.

جدول ۱۳,٤. ویژگی‌های اصلی روش مرتب‌سازی پیوندی

موضوع مورد بررسی	توضیح
پیچیدگی زمانی در حالت میانگین	$O(n\log n)$
پیچیدگی زمانی در بدترین حالت	$O(n^2)$
پیچیدگی فضایی	$O(n)$
پایداری	بلی

٤-٤-٩- مرتب‌سازی توپولوژیکی

مرتب‌سازی توپولوژیکی ارتباط مستقیمی با سایر روش‌های مرتب‌سازی که پیش از این بررسی کردیم، ندارد؛ ولی از آنجا که این روش نیز در گروه روش‌های مرتب‌سازی قرار می‌گیرد، در این بخش به آن پرداخته‌ایم.

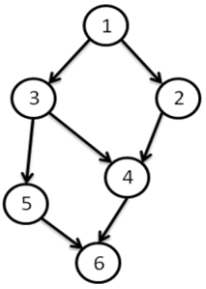
این روش مرتب‌سازی برای گراف‌های جهت‌دار و بدون دور مطرح می‌شود و کاربرد اصلی آن، زمان‌بندی کارها و وظایف اولویت‌دار است؛ به شکلی که وظایف به ترتیب اولویت انجام شوند.

هر گره از گراف دارای درجه ورودی و درجه خروجی است که تعداد یال‌های وارد شده به و خارج شده از آن گره را مشخص می‌کنند. مراحل این روش به صورت زیر است:

۱- در آغاز گره‌ای که دارای درجه ورودی صفر است، می‌بینیم (گره a). اگر گره‌ای با این شرط وجود ندارد (در حالتی که هنوز همه گره‌ها دیده نشده‌اند)، ترتیب توپولوژیکی برای این گراف وجود ندارد.
۲- از درجه‌ی ورودی همه گره‌هایی که گره a به آن‌ها متصل است (به تعداد درجه خروجی گره a)، یکی کم می‌کنیم.
۳- مراحل ۱ و ۲ را آنقدر تکرار می‌کنیم تا همه‌ی گره‌های گراف دیده شوند.

ترتیب توپولوژیکی یک گراف، لزوما یکتا نیست. برای نمونه گراف شکل ۲۳,۴ را درنظر بگیرد. ترتیب‌های توپولوژیکی ممکن برای این گراف به صورت زیر هستند. مثلا، برای دیدن گره ۴، باید ابتدا گره‌های ۲ و ۳ دیده شوند و برای دیدن این گره‌ها باید ابتدا گره ۱ دیده شود.

1 2 3 4 5 6
1 3 2 4 5 6
1 2 3 5 4 6
1 3 2 5 4 6



شکل ٤,۲۳. نمونه‌ای از یک گراف مورد بررسی برای مرتب‌سازی توپولوژیکی

شبه‌کد زیر مراحل مذکور را نشان می‌دهد:

```
L <- Empty list that will contain the sorted elements
S <- Set of all nodes with no incoming edges
while S is non - empty do {
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do {
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
    }
}
if graph has edges then
    output error message(graph has at least one cycle)
else
    output message(proposed topologically sorted order : L)
```

برای آغاز الگوریتم، گره‌هایی که دارای درجه ورودی صفر هستند را در لیست S قرار می‌دهیم. اگر گراف بـدون دور باشد، باید دست کم یک گره با درجه ورودی صفر وجود داشته باشد.

الگوریتم مرتب‌سازی توپولوژیکی

۱- **صورت مسأله:** یافتن ترتیب توپولوژیکی گراف

۲- **ورودی:** یک گراف جهت‌دار و بدون دور

۳- **خروجی:** ترتیب توپولوژیکی گره‌ها

یکی از پیاده‌سازی‌های ممکن برای این الگوریتم در ادامه قابل مشاهده است:

```
#include<fstream>
using namespace std;
ifstream fin("input.txt");
ofstream fout("output.txt");
#define MAXN 100
int graph[MAXN][MAXN];
int n;
void read_graph() {
    fin >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            fin >> graph[i][j];
    fin.close();
}
int incoming_edges(int node) {
    int counter = 0;
    for (int i = 1; i <= n; i++)
        if (graph[i][node] == 1)
            counter++;
    return counter;
}
void topological_sort(int node) {
    for (int i = 1; i <= n; i++) {
        if (graph[node][i] == 1) {
            graph[node][i] = 0;
            if (incoming_edges(i) == 0) {
                fout << " " << i;
                topological_sort(i);
            }
        }
    }
}
int main() {
    read_graph(); // read graph from input
                  // find edges with incoming edges = 0
```

```
    int no_incoming[MAXN];
    int noi_count = 0;
    for (int i = 1; i <= n; i++) {
        if (incoming_edges(i) == 0)
            no_incoming[noi_count++] = i;
    }
    // call "topological sort" function for all edges with incoming=0
    for (int i = 0; i < noi_count; i++) {
        fout << " " << no_incoming[i];
        topological_sort(no_incoming[i]);
    }
    fout.close();
    return 0;
}
```

تابع topological_sort به صورت بازگشتی پیاده‌سازی شده است؛ به صورتی که هر بار یک گره را به عنوان ورودی دریافت کرده و الگوریتم را برای آن اجرا می‌کند؛ یعنی از درجه‌ی ورودی همه‌ی گره‌های متصل به آن یکی کاسته و اگر درجه‌ی ورودی آن گره صفر شده باشد، خود را به صورت بازگشتی برای آن نیز فراخوانی می‌کند. در تابع main، ابتدا گره‌هایی که دارای درجه ورودی صفر هستند را یافته و تابع را برای آن‌ها فراخوانی می‌کنیم.

پیچیدگی زمانی روش پیاده‌سازی شده $O(n^3)$ است که نشان‌دهنده نامناسب بودن این روش است. زیرا برای محاسبه درجه‌ی ورودی هر گره، یک سطر کامل آرایه‌ی graph جستجو می‌شود. از طرف دیگر برای یافتن گره‌های متصل به گره node نیز یک ستون کامل جستجو می‌شود. با به‌کارگیری یک ساختمان داده مناسب مانند نمونه زیر، می‌توان پیچیدگی زمانی را به $O(|E|+|V|)$ کاهش داد.

```
struct node {
    int incoming_degree;
    vector<int> connected_edges;
};
```

با به‌کارگیری این ساختمان‌داده، هر دو مشکل گفته شده برطرف می‌شود. توجه می‌کنید که ساختمان‌داده می‌تواند چه تاثیر مهمی بر روی پیچیدگی داشته باشد. با به‌کارگیری این ساختمان داده‌ی جدید، روش پیشین را بهبود می‌بخشیم. واضح است که برای استفاده از این ساختمان داده باید سرآیند vector را نیز به سرآیندهای کد قبل اضافه کنید. به این ترتیب کد قبلی به این صورت تبدیل می‌شود:

```
struct node {
    int incoming_degree;
    vector<int> connected_edges;
};
node graph[MAXN];
int n;
void read_graph() {
    int temp = 0;
    fin >> n;
```

```
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
            fin >> temp;
            if (temp == 1) {
                graph[i].connected_edges.push_back(j);
                graph[j].incoming_degree++;
            }
        }
    fin.close();
}
void topological_sort(int node) {
    int destination = 0;
    for (int i = graph[node].connected_edges.size() - 1; i >= 0; i--) {
        destination = graph[node].connected_edges[i];  // get next
connected edge (i)
        graph[destination].incoming_degree--;   // decrease incoming
degree of i
        graph[node].connected_edges.pop_back();     // remove i from list
of connected edfe
        if (graph[destination].incoming_degree == 0) {
            fout << " " << destination;
            topological_sort(destination);
        }
    }
}
int main() {
    read_graph();     // read graph from input
                  // find edges with incoming edges = 0
    int no_incoming[MAXN];
    int noi_count = 0;
    for (int i = 1; i <= n; i++) {
        if (graph[i].incoming_degree == 0)
            no_incoming[noi_count++] = i;
    }
    // call "topological sort" function for all edges with incoming=0
    for (int i = 0; i < noi_count; i++) {
        fout << " " << no_incoming[i];
        topological_sort(no_incoming[i]);
    }
    fout.close();
    return 0;
}
```

٤-٤-١٠- نتیجه‌گیری

تا اینجا روش‌های مرتب‌سازی مختلف را بررسی کرده و ویژگی‌های هر یک را دیدیم. هر یک از روش‌ها کاربرد خاص خود را دارند. روش‌هایی که از درجه‌ی خطی–لگاریتمی برخوردارند، بهترین روش‌ها بوده و اگر پیاده‌سازی آن‌ها پرهزینه نباشد، بهترین انتخاب نیز خواهند بود. جدول ۱۴,۴ ویژگی‌های بررسی شده را برای همه‌ی روش‌ها دسته‌بندی می‌کند.

جدول ١٤,٤. مقایسه روش‌های مرتب‌سازی مختلف

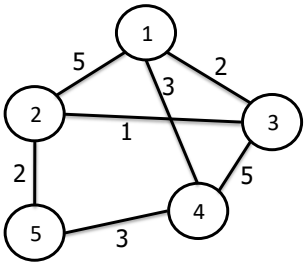
روش مرتب‌سازی	پیچیدگی زمانی در حالت میانگین	پیچیدگی زمانی در بدترین حالت	پیچیدگی فضایی	پایداری
حبابی	$O(n^2)$	$O(n^2)$	$O(1)$	بلی
انتخابی	$O(n^2)$	$O(n^2)$	$O(1)$	بلی
ترکیبی	$O(n^2)$	$O(n^2)$	$O(1)$	بلی
بینگو	$O(nm)$	$O(n^2)$	$O(1)$	بلی
درجی	$O(n^2)$	$O(n^2)$	$O(1)$	بلی
شِل	وابسته به طول گام	$O(n\log^2 n)$	$O(n)$	خیر
کتابخانه‌ای	$\approx O(n\log n)$	$O(n^2)$	$O(1)$	بلی
ادغامی	$O(n\log n)$	$O(n\log n)$	$O(n)$	بلی
سریع	$O(n\log n)$	$O(n^2)$	$O(\log n)$	خیر
درون‌گرا	$O(n\log n)$	$O(n\log n)$	$O(\log n)$	خیر
هرمی	$O(n\log n)$	$O(n\log n)$	$O(1)$	خیر
درخت دودویی	$O(n\log n)$	$O(n^2)$	$O(n)$	خیر
پیوندی	$O(n\log n)$	$O(n^2)$	$O(n)$	بلی

٤-٥- درخت پوشای کمینه

در این بخش، محاسبه‌ی درخت پوشای کمینه را مورد بررسی قرار می‌دهیم. درخت پوشای کمینه در گراف‌های بدون جهت و همبند مطرح می‌شود که هر یال در آن‌ها دارای وزن است. برای نمونه این صورت مسأله را در نظر بگیرید:

«چند شهر مرتبط با هم در یک استان وجود دارند که هنوز میان آن‌ها جاده‌سازی نشده است. می‌خواهیم این شهرها را به هم متصل کنیم، به شکلی که از هر کدام از شهرها بتوان به شهرهای دیگر سفر کرد. از طرفی ساختن جاده میان هر دو شهر، هزینه‌ی خاص خود را دارد. هدف یافتن کمترین تعداد جاده با کمترین مجموع هزینه‌ی ساخت است، به شکلی که از هر شهری بتوان به شهر دیگر رفت. برای رفتن از هر شهر به شهر دیگر حتما نیاز به جاده‌ی مستقیم میان آن دو شهر نیست؛ بلکه می‌توان به صورت غیرمستقیم هم سفر کرد.»

این مسأله دقیقا مسأله‌ای است که در گراف‌ها با آن مواجه هستیم. منظور از گراف بی‌جهت، گرافی است که یال‌های آن، جهت‌دار نیستند؛ مانند جاده‌هایی که شهرها را به هم متصل می‌کنند. گراف همبند به این معنی است که از هر گره‌ای در گراف به همه‌ی گره‌های دیگر مسیری یافت می‌شود (به صورت مستقیم یا غیرمستقیم)؛ برای نمونه در شکل ۲۴,۴ یک مسیر از گره ۱ به گره ۵ به صورت $5 \to 2 \to 1$ وجود دارد. گراف وزن‌دار به گرافی گفته می‌شود که هر یال در آن دارای وزن است. مانند جاده‌ها که برای ساخت هریک از آن‌ها باید هزینه‌ای پرداخت شود. شکل ۲۴,۴ یک گراف همبند، بدون جهت و وزن‌دار را نمایش می‌دهد.



شکل ۲٤,٤. نمونه‌ای از یک گراف همبند، بدون جهت و وزن‌دار

در شکل ۲۴,۴ اعداد روی یال‌ها، وزن و اعداد نوشته شده در گره‌ها، شماره گره هستند. وزن‌ها را مقادیر غیرمنفی فرض می‌کنیم. هر یال توسط زوج گره‌های مبدا و مقصدش، نشان داده می‌شود؛ برای نمونه $(1,4)$ مشخص کننده‌ی یالی است که دو گره‌ی ۱ و ۴ را به هم متصل می‌کند؛ از آنجا که گراف بی‌جهت است، این یال برابر با یال $(4,1)$ نیز هست.

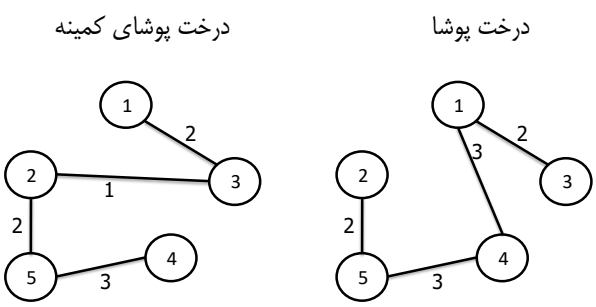
درخت یک گراف بی‌جهت، همبند و بدون چرخه است. مسیری که از یک گره آغاز و به خودش ختم می‌شود، چرخه نامیده می‌شود. منظور از درخت پوشای یک گراف، شکلی از آن گراف است که دارای شرایط زیر باشد:

۱- شامل همه‌ی گره‌های گراف باشد.

۲- هیچ چرخه‌ای در آن وجود نداشته باشد.

۳- از هر گره‌ای به همه‌ی گره‌های دیگر مسیری وجود داشته باشد.

درخت پوشای کمینه، درخت پوشایی است که هزینه‌ی آن کمینه باشد؛ منظور از هزینه، مجموع وزن یال‌های درخت است. شکل ۲۵,۴ یک درخت پوشا و یک درخت پوشای کمینه را برای گراف شکل ۲۴,۴ نمایش می‌دهد. درخت‌های پوشا و پوشای کمینه یکتا نیستند ولی به یقین، همه‌ی درخت‌های پوشای کمینه دارای کمترین هزینه هستند.



شکل ۲۵,٤. یک درخت پوشا و یک درخت پوشای کمینه برای گراف شکل ۲٤,٤

هر گراف را به صورت یک زوج از راس‌ها و یال‌های آن نشان می‌دهند.

$$G = (V, E)$$

V مجموعه‌ای شامل راس‌های گراف و E مجموعه‌ی یال‌های گراف است. برای نمونه گراف شکل ۲۴،۴ به این صورت نشان داده می‌شود:

$$V = \{1,2,3,4,5\}$$
$$E = \{(1,2),(1,3),(1,4),(2,3),(2,5),(3,4),(4,5)\}$$

برای یافتن درخت پوشای کمینه چند الگوریتم ارائه شده‌اند که به توضیح برخی از آن‌ها می‌پردازیم؛ از جمله‌ی این روش‌ها، روش پرایم[۳۱]، کروسکال[۳۲]، حذف معکوس و بروکاو[۳۳] هستند. همه‌ی این روش‌ها، در دسته‌ی مسأله‌های حریصانه قرار می‌گیرند. ساختار مسأله برای این الگوریتم‌ها به صورت زیر است. برای جلوگیری از تکرار فقط یک‌بار آن را بیان می‌کنیم.

الگوریتم‌های یافتن درخت پوشای کمینه

۱- **صورت مسأله:** یافتن درخت پوشای کمینه
۲- **ورودی:** یک گراف بی‌جهت، همبند و وزن‌دار
۳- **خروجی:** مجموعه‌ای از یال‌ها که درخت پوشای کمینه را مشخص می‌کنند.

٤-۵-۱- الگوریتم پرایم

الگوریتم پرایم دو مجموعه‌ی SV و SE را برای نگهداری راس‌ها و یال‌های انتخابی درنظر می‌گیرد که هر دو در ابتدا تهی هستند. برای آغاز الگوریتم، یک راس را به دلخواه برگزیده، آن را به مجموعه‌ی SV اضافه کرده و مرحله‌های زیر را دنبال می‌کنیم:

۱- از میان راس‌های خارج از مجموعه‌ی SV، نزدیک‌ترین راس به یکی از راس‌های داخل مجموعه را برمی‌گزینیم. در واقع در این حالت، از میان یال‌هایی که راس‌های داخل مجموعه را به راس‌های خارج مجموعه متصل می‌کنند، یالی که دارای کمترین وزن است، برگزیده می‌شود.
۲- راس برگزیده شده را به مجموعه‌ی SV اضافه می‌کنیم.
۳- یال مربوطه را به مجموعه‌ی SE اضافه می‌کنیم.
۴- اگر مجموعه‌ی SV شامل همه‌ی راس‌های گراف است، الگوریتم به پایان رسیده وگرنه به مرحله ۱ بازمی‌گردیم.

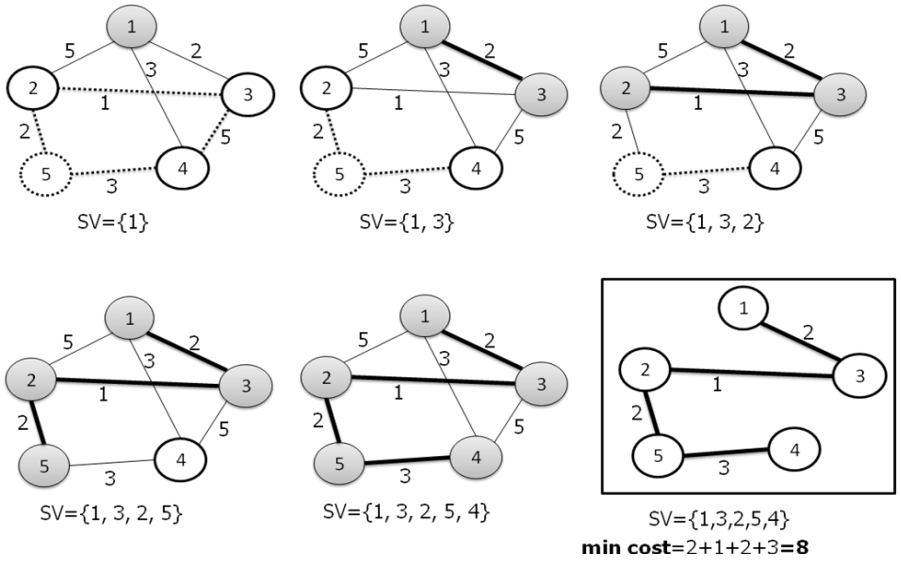
شبه‌کد بعد مراحل گفته شده را نشان می‌دهد. مجموعه‌ی V شامل همه‌ی راس‌های گراف است.

---

[۳۱] Prime
[۳۲] Kruskal
[۳۳] Borůvka

```
V = {all vertices: V1...Vn}
SV = {V1}
SE = {}
while (V - SV is not empty) {
    select a vertex in V - SV that nearest to SV
    add the vertex to SV
    add the edge to SE
}
```

بخش اصلی الگوریتم پرایم، مرحله‌ی اول است که نزدیکترین راس در آن برگزیده می‌شود. مزیت الگوریتم پرایم عدم نیاز به بررسی ایجاد چرخه است؛ برگزیدن راس از مجموعه‌ی $V - SV$ عدم ایجاد چرخه را تضمین می‌کند. براساس الگوریتم توضیح داده شده، درخت پوشای کمینه‌ی مربوط به گراف شکل ۲۴,۴ را بدست می‌آوریم (شکل ۲۶,۴). برای این گراف، دو درخت پوشای کمینه با هزینه‌ی هشت بدست می‌آید. اگر در مرحله‌ی آخر یال $(1,4)$ را انتخاب می‌کردیم، درخت پوشای دیگری بدست می‌آمد.



شکل ۲۶,٤. مراحل به دست آوردن درخت پوشای کمینه گراف شکل ۲٤,٤ با استفاده از روش پرایم

برای پیاده‌سازی الگوریتم پرایم، از ماتریس مجاورت برای نگهداری گراف استفاده می‌کنیم. اگر نام این ماتریس را a بگذاریم، هر خانه‌ی آن مشخص کننده‌ی وزن یک یال است. البته در حالت کلی خانه‌ی $a[i][j]$ وجود/عدم وجود یال میان دو راس i و j را تعیین می‌کند. اگر میان دو راس یالی وجود نداشته باشد، در خانه‌ی مربوطه مقدار ∞ را قرار می‌دهیم. برای پیاده‌سازی اصلی می‌توانیم مقداری بزرگ را درنظر بگیریم که بیشتر از بیشترین وزن ممکن باشد.

$$a[i][j] = \begin{cases} w & \text{وزن یالی که بین } i \text{ و } j \text{ دارد} \\ \infty & \text{یالی بین } i \text{ و } j \text{ وجود ندارد} \end{cases}$$

ماتریس مجاورت گراف شکل ۲۴,۴ را در ادامه مشاهده می‌کنید.

	1	2	3	4	5
1	∞	5	2	3	∞
2	5	∞	1	∞	2
3	2	1	∞	4	∞
4	∞	∞	5	∞	3
5	∞	2	∞	3	∞

برای یافتن درخت پوشای کمینه به دو آرایه تک‌بعدی به این صورت احتیاج داریم:

nearest[i] = ام_i نمایه‌ی نزدیک‌ترین راس موجود در SV به راس

distance[i]= کمترین فاصله‌ی راس i تا مجموعه‌ی SV

در واقع وزن یال میان راس ام_i و راس مشخص شده همان nearest[i] است.

پس از ارائه‌ی کد، بخش‌های مختلف آن را بررسی خواهیم کرد. برای سادگی بیشتر، بعضی از آرایه‌ها و متغیرها را به صورت عمومی تعریف کرده‌ایم. کد بعدی علاوه بر یافتن مجموعه‌ی SE که شامل یال‌های درخت پوشای کمینه است، هزینه‌ی آن را نیز محاسبه می‌کند. مجموعه‌ی SE در بردار mst_edges و هزینه‌ی آن در متغیر mst_cost نگهداری می‌شود. ثابت n تعداد راس‌ها را مشخص می‌کند که باز هم برای سادگی آن را ثابت در نظر گرفتیم. آرایه‌ی intree معادل مجموعه‌ی SV است که راس‌های اضافه شده به درخت پوشای کمینه را مشخص می‌کند. این آرایه هنگام جستجو برای راس جدید به کار می‌آید.

آغاز الگوریتم، با برگزیدن راس شماره ۱ و اضافه کردن آن به درخت پوشای کمینه آغاز می‌شود؛ سپس دو آرایه distance و nearest بروز می‌شوند. مجموعه‌ی $SV = \{1\}$ است، بنابراین در همه‌ی خانه‌های آرایه‌ی nearest مقدار ۱ قرار داده می‌شود. در ادامه distance[i] نیز برابر با وزن یال $(1, i)$ قرار می‌گیرد.

حلقه‌ی اصلی الگوریتم $n-1$ مرتبه تکرار شده و در هر مرحله یک راس که نزدیکترین راس به مجموعه‌ی SV است، برگزیده می‌شود. پس از $n-1$ مرحله درخت پوشا ساخته شده و الگوریتم به پایان می‌رسد.

```
#include<iostream>
#include<vector>
using namespace std;
const int n = 5;
struct edge {
    int v1;
    int v2;
```

```
};
int a[n + 1][n + 1]; // adjacency matrix
int nearest[n + 1], least_distance[n + 1];
int mst_cost = 0;    // minimum spanning tree cost
vector<edge> mst_edges; // SE={}
bool prime() {
    bool intree[n + 1] = {false};  // SV={}
    intree[1] = true;           // SV={1}
    for (int i = 2; i <= n; i++) {
        least_distance[i] = a[1][i];
        nearest[i] = 1;
    }
    for (int treesize = n - 1; treesize > 0; treesize--) {
        int near_index = -1;
        int min = INT_MAX;
        for (int i = 2; i <= n; i++)   // find nearest vertices
            if (least_distance[i] < min && intree[i] == false) {
                min = least_distance[i];
                near_index = i;
            }
        if (near_index == -1)
            return false; // graph is not connected
        intree[near_index] = true;
        mst_cost += least_distance[near_index];
        // add edge (nearest[near_index],near_index) to SE
        edge near_edge;
        near_edge.v1 = nearest[near_index];       // v1
        near_edge.v2 = near_index;              // v2
        mst_edges.push_back(near_edge);         // add (v1,v2) to SE
        for (int i = 2; i <= n; i++) {          // update distances
            if (least_distance[i] > a[near_index][i]) {
                least_distance[i] = a[near_index][i];
                nearest[i] = near_index;
            }
        }
    }
    return true;
}
```

بدنه‌ی حلقه‌ی اصلی از دو حلقه‌ی دیگر تشکیل شده است. حلقه‌ی اول به دنبال راسی مناسب گشته که از نظر وزن دارای کمترین مقدار باشد و از طرفی در مجموعه‌ی SV هم قرار نداشته باشد. اگر راسی یافت نشود، گراف همبند نبوده و برنامه خطایی را بازمی‌گرداند. در غیر این صورت یال مربوطه به مجموعه‌ی SE اضافه شده و وزن آن نیز به مجموع وزن‌ها اضافه می‌شود. حلقه‌ی دوم، راس‌های متصل به راس جدید را جستجو کرده و وزن آن‌ها را در متغیر distance بروز می‌کند.

حلقه‌ی اصلی و دو حلقه‌ی توضیح داده شده هر کدام $n-1$ مرتبه تکرار می‌شوند. بنابراین پیچیدگی زمانی این الگوریتم برابر با $O((n-1)*(2*(n-1))=O(n^2)$ خواهد بود. برای یافتن نزدیکترین راس می‌توان به

جای ماتریس مجاورت، ساختمان‌داده‌های دیگری مانند heap را نیز به کار برد؛ در این صورت پیچیدگی زمانی به درجه‌ی خطی–لگاریتمی کاهش پیدا خواهد کرد.

٤-٥-٢- الگوریتم کروسکال

روش پرایم هر بار یک راس به درخت اضافه می‌کرد و این چرخه را تا وقتی که به درخت پوشا برسد، ادامه می‌داد. روش کروسکال اساس را بر انتخاب یال گذاشته و هر بار یالی با کمترین وزن را به درخت اضافه می‌کند. این الگوریتم برای شروع، راس‌های گراف را بدون هیچ یالی درنظر می‌گیرد؛ سپس یال‌ها را به ترتیب غیرنزولیِ وزن مرتب کرده و در هر مرحله یالی با کمترین وزن را برمی‌گزیند؛ اگر اضافه شدن یال انتخاب شده به درخت، چرخه ایجاد نکند، آن را اضافه کرده و در غیراین صورت یال بعدی را انتخاب می‌کند. این روال تا وقتی که به یک درخت پوشا برسیم، ادامه پیدا خواهد کرد (انتخاب $n-1$ یال).

برای پیاده‌سازی این روش، ساختمان‌داده‌ی مجموعه‌های جدا را به کار می‌بریم. در آغاز همه‌ی راس‌ها را به مجموعه‌ی جدا اضافه می‌کنیم؛ بنابراین هر راس، یک درخت محسوب می‌شود. با برگزیدن هر یال $(v1, v2)$، دو درختی که دربردارنده‌ی گره‌های $v1$ و $v2$ هستند، به هم متصل می‌شوند. اگر این دو گره، پیش از این در یک درخت قرار داشته باشند، افزودن یال $(v1, v2)$ چرخه ایجاد کرده و نباید آن را اضافه کرد. مراحل زیر توضیحات داده شده را دقیق‌تر بیان می‌کند.

۱- راس‌های گراف را در مجموعه‌ی جدا قرار داده و مجموعه‌ی خالی F را برای نگهداری یال‌های اضافه شده به درخت درنظر می‌گیریم.

۲- یال‌ها را به ترتیب غیرنزولی مرتب می‌کنیم تا بتوانیم در هر مرحله یالی که دارای وزن کمتر است، انتخاب کنیم.

۳- یالی که دارای کمترین وزن است، برمی‌گزینیم.

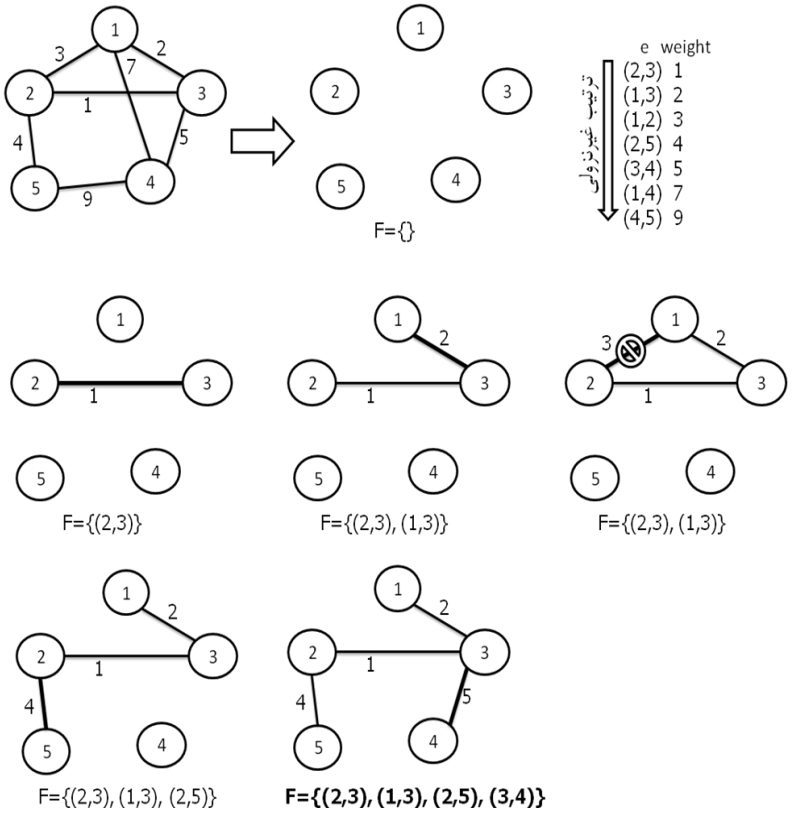
۴- اگر راس‌های یال برگزیده شده، متعلق به دو درخت مستقل هستند، آن را به مجموعه‌ی F اضافه کرده و دو درخت مربوطه را نیز به هم متصل می‌کنیم (با به‌کارگیری تابع union از بخش مجموعه‌های از هم-جدا)؛ در غیر این صورت، این یال را کنار می‌گذاریم.

۵- اگر درخت پوشا ساخته شده است، الگوریتم به پایان می‌رسد. در غیراین صورت به مرحله ۳ بازمی‌گردیم.

شبه‌کد زیر مراحل بالا را توصیف می‌کند:

```
F = {};
for (int i = 1; i <= n; i++)   // add vertics to disjoint_set
makeset(i)
sort all edges by weight in nondecreasing order
while (length(F) < n - 1) {
    edge e = select next edge that have minimum weight
    xRoot = find(e.v1)
    yRoot = find(e.v2)
    if (xRoot != yRoot) {
        add e to F
        union(xRoot, yRoot)
    }
}
```

درخت پوشای نهایی دارای $n-1$ یال است (n راس). بنابراین شرط پایان حلقه، وجود $n-1$ یال در مجموعه‌ی F است. برای روشن‌تر شدن موضوع، درخت پوشای کمینه‌ی گراف نشان داده شده در شکل ۲۷,۴ را به روش کروسکال بدست می‌آوریم.



شکل ۲۷,٤. مراحل به دست آوردن درخت پوشای کمینه یک گراف با استفاده از روش کروسکال

با به‌کارگیری مجموعه‌های جدا، کار ساده‌ای برای پیاده‌سازی این الگوریتم در پیش‌رو داریم. به نمونه کد زیر، توابع مربوط به مجموعه‌های جدا را نیز اضافه کنید:

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
#define max_n 100
struct edge {
    int v1;   int v2;   int weight;
}set_of_edges[max_n];
vector<edge> F;
int compare(const void* e1, const void* e2)
```

```
{
    return(((edge*)e1)->weight - ((edge*)e2)->weight);
}
void kruskal(int n, int e) {
    F.clear(); // F={}
               // sort edges in nondecreasing order
    qsort(set_of_edges, e, sizeof(edge), compare);
    // insert all vertics into disjoint_set
    for (int i = 1; i <= n; i++)
        makeset(i);
    int sel_edge_index = 0;
    edge sel_edge;
    while (F.size() < n - 1) {
        sel_edge = set_of_edges[sel_edge_index];
        if (union(sel_edge.v1, sel_edge.v2))
            F.push_back(sel_edge);  // add edge to F
        sel_edge_index++;       // select next edge
    }
}
```

برای آزمایش کد بالا، تعداد راس‌ها و یال‌های گراف را از ورودی دریافت کرده و سپس به تعداد یال‌های وارد شده، از ورودی یال دریافت می‌کنیم. برای دریافت یال، ابتدا راس مبدا، سپس راس مقصد و در پایان وزن آن با فاصله از هم دریافت شده و در آرایه set_of_edges قرار داده می‌شود. برای سادگی این آرایه و بردار F را به صورت سراسری تعریف کرده‌ایم.

```
int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; i++)        // get m edges from input
        cin >> set_of_edges[i].v1 >> set_of_edges[i].v2 >>
set_of_edges[i].weight;
    kruskal(n, m);                  // calculate mst
    for (int i = 0; i < F.size(); i++) // print F in output
        cout << "(" << F[i].v1 << "," << F[i].v2 << ")" << endl;
    return 0;
}
```

n تعداد راس‌ها و m تعداد یال‌ها است. کد توضیح داده شده را اجرا کرده و گراف شکل ۲۷,۴ را به عنوان ورودی به آن بدهید (در این نمونه، $n=5$ و $m=7$ است. اعداد 3 2 1 در خط دوم یعنی یال $(1,2)$ با وزن ۳. به این صورت ورودی و خروجی این گونه خواهد بود:

Input	Output
5 7 1 2 3 1 3 2 1 4 7 2 3 1 2 5 4 3 4 5 4 5 9	(2,3) (1,3) (2,5) (3,4)

تنها بخش مجهول کد ارائه شده، به‌کارگیری تابع qsort است. این تابع در سرآیند algorithm قرار داشته و برای مرتب کردن آرایه‌ها به کار گرفته می‌شود. اعلان تابع qsort به این صورت است:

```
qsort(array, array length, sizeof(element), compare method);
```

پارامترهای ورودی تابع qsort عبارتند از:

۱- array: نام آرایه

۲- array length: طول آرایه

۳- sizeof element: طول هر عنصر از آرایه. برای به دست آوردن فضای اشغالی عنصرهای آرایه تابع sizeof() را به کار ببرید. پارامتر ورودی این تابع نوع مورد نظر شماست. برای نمونه sizeof(int) مقدار فضای اشغالی توسط یک متغیر از نوع int را برمی‌گرداند.

۴- compare method: برای مرتب کردن آرایه باید یک تابع برای مقایسه عنصرها به تابع qsort معرفی کنیم. نوشتن تابع compare مشابه توضیحات داده شده برای توابع مقایسه‌کننده در فصل سوم است. اعلان تابع compare باید به این صورت باشد:

```
int method_name(const void*, const void*);
```

این تابع، دو ورودی از نوع *void دریافت کرده و مقداری از نوع int برمی‌گرداند. مقدار خروجی این تابع، یکی از سه مقدار ۰، ۱- و ۱ است. استفاده از تابع qsort به دلیل سادگی و سرعت بالایی که دارد، توصیه می‌شود. این تابع در واقع پیاده‌سازی روش مرتب‌سازی سریع است.

حال به بررسی پیچیدگی زمانی الگوریتم کروسکال می‌پردازیم. در ادامه پیچیدگی زمانی هر یک از بخش‌های تابع kruskal را محاسبه خواهیم کرد.

- مرتب‌کردن مجموعه‌ی شامل یال‌ها توسط تابع qsort از درجه‌ی خطی-لگاریتمی است؛ بنابراین پیچیدگی زمانی آن $O(e \log e)$ خواهد بود که e همان تعداد یال‌هاست.
- قرار دادن همه‌ی راس‌ها در مجموعه‌ی جدا در زمان $O(n)$ صورت می‌گیرد. زیرا پیچیدگی زمانی تابع makeset از مرتبه‌ی $O(1)$ است و به تعداد راس‌ها (n) فراخوانی می‌شود.

- حلقه‌ی while به تعداد $n-1$ مرتبه تکرار شده و در هر تکرار، یک بار تابع union را فراخوانی می‌کند. تابع union خود دو مرتبه تابع find را که دارای پیچیدگی زمانی $O(\log n)$ است، فراخوانی می‌کند. بنابراین پیچیدگی زمانی تابع union برابر با $2O(\log n) = O(\log n)$ بوده و در نتیجه پیچیدگی زمانی حلقه‌ی while برابر با $O(n \log n)$ خواهد بود.

از سه بخش بررسی شده، پیچیدگی زمانی الگوریتم کروسکال به این صورت محاسبه می‌شود:

$$O(e \log e) + O(n) + O(n \log n) \in O(e \log e)$$

حداقل تعداد یال‌ها برابر با یکی کمتر از تعداد راس‌ها است. بنابراین در رابطه‌ی بالا می‌توان از $O(n \log n)$ در برابر $O(e \log e)$ چشم‌پوشی کرد. از طرفی پیچیدگی زمانی بالا می‌تواند به این صورت نیز تفسیر شود:

$$O(e \log e) = O(e \log n) = O(n^2 \log n)$$

زیرا در بدترین حالت، تعداد یال‌ها از قاعده‌ی زیر پیروی می‌کند:

$$e = \frac{n(n-1)}{2} \in O(n^2)$$

اگر e را در بدترین حالت برابر با n^2 درنظر بگیریم (در واقع همیشه کمتر از این مقدار است)، خواهیم داشت:

$$e = n^2 \Rightarrow \log e = \log n^2 = 2 \log n \Rightarrow O(\log e) = O(2 \log n) \in O(\log n)$$

٤-٥-٣- روش‌های دیگر

الگوریتم حذف معکوس و الگوریتم بروکاو با پیچیدگی‌های زمانی $O(e \log e (\log \log e)^3)$ و $O(e \log n)$ دو روش دیگر برای به دست آوردن درخت پوشای کمینه هستند. روش حذف معکوس دقیقا وارون روش کروسکال عمل می‌کند؛ به این صورت که در آغاز گراف را با همه‌ی یال‌هایش در نظر گرفته و سپس در هر مرحله، یالی با بیشترین وزن را برمی‌گزیند؛ اگر با حذف یال مربوطه گراف همبند بماند، آن را حذف می‌کند؛ این روال تا وقتی که همه‌ی یال‌ها بررسی شوند، ادامه پیدا می‌کند. روش بروکاو شبیه به روش پرایم عمل می‌کند. با این تفاوت که روش بروکاو برای گرافی با وزن‌های متمایز به کار گرفته می‌شود و در برگزیدن یال بعدی، روشی متفاوت در پیش می‌گیرد. این الگوریتم یکی از روش‌های خوب برای به دست آوردن درخت پوشای کمینه است. از ترکیب این روش و روش پرایم می‌توان به روشی با پیچیدگی زمانی بهتر دست یافت.

٤-٥-٤- مقایسه‌ی روش‌های مختلف

جدول ۱۵,۴ پیچیدگی زمانی الگوریتم‌های معرفی شده را نشان می‌دهد.

جدول ۱۵,٤. مقایسه پیچیدگی زمانی الگوریتم‌های مختلف برای به دست آوردن درخت پوشای کمینه یک گراف

الگوریتم	پیچیدگی زمانی
پرایم	$O(n^2)$
کروسکال	$O(n\log n^2)$ یا $O(e\log e)$
حذف معکوس	$O(e\log e(\log\log e)^3)$
روش بروکاو	$O(e\log n)$

در یک گراف همبند قاعده‌ی زیر برقرار است:

$$n-1\le e\le \frac{n(n-1)}{2}$$

اگر تعداد یال‌ها به کرانه‌ی پایینی نزدیک‌تر باشد، الگوریتم کروسکال با پیچیدگی زمانی $O(e\log e)$ گزینه‌ی بهتری است؛ و اگر تعداد یال‌ها به کرانه‌ی بالایی نزدیک‌تر باشد، الگوریتم کروسکال با پیچیدگی زمانی $O(n^2\log n)$ ضعیف‌تر از الگوریتم پرایم با پیچیدگی زمانی $O(n^2)$ عمل می‌کند.

کارایی الگوریتم حذف معکوس مشابه الگوریتم کروسکال است؛ در حالت کلی، روش کروسکال بهتر از روش حذف معکوس عمل می‌کند. روش بروکاو مشابه روش پرایم و تا حدودی بهتر از آن عمل می‌کند. البته همان‌طور که توضیح داده شد، عملکرد دقیق با توجه به تعداد راس‌ها و یال‌ها مشخص می‌شود.

٤-٦- کوتاهترین مسیر در گراف

یافتن کوتاه‌ترین مسیر در گراف‌ها از مسأله‌های بسیار کاربردی است. منظور از گراف، گرافی جهت‌دار و وزن‌دار است که وزن‌های آن مقادیر غیرمنفی هستند. البته گراف غیرجهت‌دار هم نوعی گراف جهت‌دار محسوب می‌شود (همه‌ی یال‌های آن در هر دو جهت هستند). مسیریابی از مهم‌ترین کاربردهای این الگوریتم‌ها است. برای نمونه چند شهر که از یکدیگر فاصله‌های متفاوت و مشخصی دارد را فرض کنید؛ توسط الگوریتم‌های توضیح داده شده در این بخش می‌توانید کوتاهترین مسیر از یک شهر به شهر یا شهرهای دیگر را بیابید. در این بخش به توضیح دو الگوریتم دیکسترا[۳۴] و فلوید–وارشال[۳۵] می‌پردازیم. الگوریتم دیکسترا در دسته‌ی روش‌های حریصانه و الگوریتم فلوید–وارشال در دسته‌ی روش‌های پویا قرار می‌گیرد. الگوریتم فلوید–وارشال برای یافتن کوتاه‌ترین مسیر از هر راسی به راس‌های دیگر به کار گرفته می‌شود؛ ولی از الگوریتم دیکسترا برای یافتن کوتاه‌ترین مسیر از تنها یک راس به همه‌ی راس‌های دیگر بهره می‌بریم؛ از این رو، روش دیکسترا را الگوریتمی برای یافتن کوتاه‌ترین مسیر تک‌مبدا در گراف‌ها می‌شناسند.

---

[۳۴] Dijkstra

[۳۵] Floyd–Warshall

٤-٦-١- الگوریتم دیکسترا برای یافتن کوتاه‌ترین مسیر تک‌مبدا

همان‌طور که گفته شد این الگوریتم برای یافتن کوتاه‌ترین مسیر از یک راس خاص به دیگر راس‌ها در گراف به کار می‌رود. بنابراین برای آغاز الگوریتم باید ابتدا راس مبدا را مشخص کنیم. در این روش به هر راس یک مسافت نسبت داده می‌شود که فاصله آن راس تا راس مبدا است. در پایان الگوریتم دیکسترا، این مسافت به کوتاهترین فاصله بین راس مربوطه و راس مبدا تبدیل می‌شود. پس از بررسی هر راس، آن را در مجموعه‌ی visited (مجموعه رئوس ملاقات شده) قرار داده تا از بررسی مجدد آن جلوگیری کنیم. راس مبدا را source نام‌گذاری می‌کنیم. مراحل زیر، روش کار این الگوریتم را نشان می‌دهند:

۱- در آغاز مسافت همه‌ی راس‌ها غیر از source را بی‌نهایت قرار داده و مسافت راس source را صفر قرار می‌دهیم.

۲- راس source را انتخاب کرده و در مجموعه‌ی visited قرار می‌دهیم. سپس مسافت همه‌ی راس‌های مجاور با آن را به وزن یالی که میان source و آن راس وجود دارد، تغییر می‌دهیم.

۳- از میان راس‌های داخل مجموعه‌ی visited، راسی را که دارای کمترین مسافت است، برگزیده و آن را u می‌نامیم.

۴- راس u را در مجموعه‌ی visited قرار می‌دهیم. سپس مسافت همه‌ی راس‌های مجاور با u که در مجموعه‌ی visited قرار ندارند را بررسی می‌کنیم. اگه مسیری که از راس u به آن راس‌ها وجود دارد، از نظر مسافت بهتر از مسافت کنونی آن راس‌ها است، مسافت جدید را جایگزین مسافت پیشین آن راس خواهیم کرد.

۵- اگر همه‌ی راس‌ها در مجموعه‌ی visited قرار دارند، الگوریتم به پایان رسیده است؛ در غیر این صورت، الگوریتم را از مرحله ۳ تکرار می‌کنیم.

مرحله‌ی ۲ را می‌توان در مرحله‌ی ۳ ادغام کرد، زیرا در آغاز الگوریتم، مسافت راس source از دیگر راس‌ها کمتر است (۰ در مقابل ∞) و حتما این راس به عنوان نخستین راس برگزیده می‌شود. در شبه‌کد ارائه شده، این دو مرحله در هم ادغام شده‌اند. هدف الگوریتم، یافتن کوتاه‌ترین مسیر است ولی در مراحل ذکر شده، صحبتی از مسیر کوتاه‌تر نشده و فقط کوتاه‌ترین مسافت‌ها محاسبه شده‌اند. در شبه‌کد زیر برای نگهداری کوتاه‌ترین مسیرها، روشی ارائه شده است:

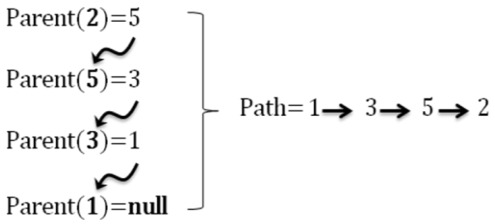
```
for each vertex in graph {
    distance(vertex) = ∞
    parent(vertex) = null
    visited(vertex) = false
}
distance(source) = 0
while (notvisited < n) {
    u = select unvisited vertex with min distance
    visited(u) = true
    // update distances of neighbors of u
    for all neighbors v of vertex u {
```

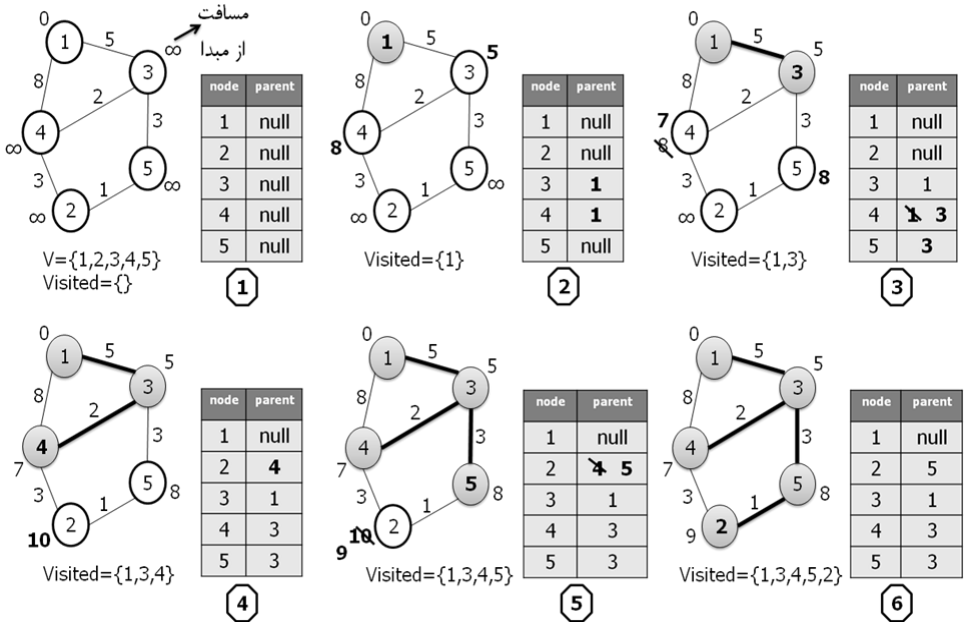
```
        if (distance(u) + weight(u,v) < distance(v)) {
            distance(v) = distance(u) + weight(u, v)
            parent(v) = u
        }
    }
}
```

حلقه while تا زمانی که تعداد راس‌های بازدید شده با n (تعداد راس‌های گراف) برابر نشود، ادامه می‌یابد. آرایه‌ی parant برای نگهداری کوتاه‌ترین مسیرها به کار گرفته شده است. طول این آرایه به تعداد راس‌های گراف است. parent(v)=u یعنی راس u در کوتاهترین مسیر به راس v وجود دارد و نزدیک‌ترین راس به راس v است. برای نمونه اگر مسیر p کوتاه‌ترین مسیر از source به راس v باشد، مطمئنا شکل کلی مسیر p به صورت زیر است:

$$source \rightarrow \ldots \rightarrow u \rightarrow v$$

برای روشن‌تر شدن موضوع، الگوریتم دیکسترا را به گراف پنج راسی شکل ۲۸,۴ اعمال می‌کنیم. با به‌کارگیری آرایه‌ی parent می‌توانیم کوتاه‌ترین مسیر از راس ۱ به راس‌های ۲، ۳، ۴ و ۵ را بدست بیاوریم. برای نمونه برای به دست آوردن کوتاه‌ترین مسیر از راس ۱ به راس ۲ به این صورت عمل می‌کنیم:





شکل ۲۸,٤. نحوه عملکرد روش دیکسترا بر یک گراف ساده (راس ۱ مبدا در نظر گرفته شده است)

حال به پیاده‌سازی این الگوریتم می‌پردازیم. پیاده‌سازی انجام شده کاملا مطابق با شبه‌کد ارائه شده در قبل است. آرایه‌های distance، parent، visited و آرایه‌ی دوبعدی weight که شامل وزن‌های گراف است، برای سادگی به صورت عمومی تعریف شده‌اند. پارامتر ورودی n تعداد راس‌های گراف است. پس از فراخوانی تابع dijkstra آرایه‌ی parent شامل کوتاه‌ترین مسیرها از راس ۱ (راس مبدا) به سایر راس‌ها است.

روش به دست آوردن مسیر از راس مبدا به هر راس دلخواه، قبلا توضیح داده شد؛ در ادامه برای پیاده‌سازی این روش، تابع get_shortest_path را ارائه می‌کنیم. این تابع راس مقصد را دریافت کرده و کوتاه‌ترین مسیر از راس ۱ به آن راس را در خروجی چاپ می‌کند. روش بازگشتی ساده‌ترین روش برای پیاده‌سازی این تابع است.

```
void get_shortest_path(int destination) {
    if (parent[destination] != 0)
        get_shortest_path(parent[destination]);
    cout << destination << " ";
}
```

الگوریتم دیکسترا

۴- **صورت مسأله:** یافتن کوتاه‌ترین مسیر تک‌مبدا در گراف جهت‌دار و وزن‌دار

۵- **ورودی:** یک گراف جهت‌دار و وزن‌دار

۶- **خروجی:** کوتاه‌ترین مسیرها از راس مبدا به دیگر راس‌های گراف

```
#define MAX_N 100
#define MAX_INT 100000
int distance[MAX_N], parent[MAX_N];
bool visited[MAX_N];
int weight[MAX_N][MAX_N];
void dijkstra(int n) {
    for (int i = 1; i <= n; i++) {
        distance[i] = MAX_INT;
        parent[i] = 0;    // NULL
        visited[i] = false;
    }
    distance[1] = 0; // assume vertex 1 as source
    for (int j = 1; j <= n; j++) {
        int u = -1, min = MAX_INT;
        // u = select unvisited vertex with min distance to source
        for (int i = 1; i <= n; i++)
            if (!visited[i] && distance[i] < min) {
                min = distance[i];
                u = i;
            }
        if (u == -1) return; // Graph is not connected
        visited[u] = true;
        // update distances of neighbors of u
        for (int v = 1; v <= n; v++) {
```

```
            if (!visited[v] && distance[u] + weight[u][v] < distance[v]){
                distance[v] = distance[u] + weight[u][v];
                parent[v] = u;
            }
        }
    }
}
```

در تابع بالا اگر گراف همبند نباشد، کار متوقف شده و خارج می‌شویم. برای تحلیل پیچیدگی زمانی این الگوریتم، به حلقه‌ها توجه می‌کنیم. حلقه‌ی نخست که برای مقداردهی اولیه به آرایه‌ها به کار رفته، از مرتبه‌ی $O(n)$ است. حلقه‌ی دوم که خود شامل دو حلقه‌ی دیگر است، n بار تکرار می‌شود. دو حلقه‌ی درونی نیز هر یک n مرتبه تکرار می‌شوند؛ بنابراین پیچیدگی زمانی این سه حلقه $O(n*2n)=O(2n^2)$ است. از این توضیحات نتیجه می‌گیریم که پیچیدگی زمانی الگوریتم دیکسترا برابر با $O(n)+O(2n^2)\in O(n^2)$ است. برای یافتن کوتاه‌ترین مسیر از همه‌ی راس‌ها به دیگر راس‌ها می‌توانیم این الگوریتم را n بار اجرا کرده و هر بار یکی از راس‌ها را به عنوان راس مبدا درنظر بگیریم. در این حالت پیچیدگی زمانی $O(n^3)$ خواهد بود.

می‌توان پیاده‌سازی‌های بهینه‌تری نیز برای این الگوریتم ارائه کرد؛ برای نمونه اگر یک هرم معمولی را برای برگزیدن راس بعدی با کمترین فاصله از مبدا به کار ببریم، پیچیدگی زمانی به $O(e\log n)$ کاهش پیدا خواهد کرد که e تعداد یال‌ها است. استفاده از هرم فیبوناچی[۳۶] می‌تواند این مقدار را حتی به $O(e+n\log n)$ نیز کاهش دهد. البته این دو پیاده‌سازی کمی دشوارتر از پیاده‌سازی معمولی الگوریتم دیکسترا خواهند بود.

٤-٦-٢- الگوریتم فلوید-وارشال برای یافتن همه‌ی کوتاه‌ترین مسیرها

الگوریتم فلوید-وارشال برای یافتن همه‌ی کوتاه‌ترین مسیرها از هر راس به همه‌ی راس‌های دیگر در گرافی جهت‌دار و وزن‌دار به کار گرفته می‌شود. این الگوریتم به روش پویا عمل کرده و پیاده‌سازی آن بسیار ساده است؛ بنابراین گاهی حتی برای به دست آوردن کوتاهترین مسیر از تنها یک راس به راس‌های دیگر، این روش به کار برده می‌شود. مراحل کار این الگوریتم به صورت زیر است:

۱- ابتدا مسافت میان هر دو راس را برابر با وزن یالی که میان آن دو وجود دارد، قرار داده و اگر یالی میان دو راس وجود نداشته باشد، مقدار ∞ قرار داده می‌شود.

۲- برای همه‌ی جفت راس‌های u و v، اگر راسی مانند k وجود داشته باشد که مسافت u تا k به اضافه‌ی k تا v بهتر از مسافت کنونی u تا v باشد، مسافت جدید را جایگزین می‌کند.

مراحل بالا اساس کار الگوریتم فلوید-وارشال است. البته مرحله‌ی دوم به صورت جزئی خود شامل n مرحله است که n تعداد راس‌های گراف است. این n مرحله را در ادامه بررسی می‌کنیم.

---

[۳۶] Fibonacci heap

مرحله ۱: راس کمکی $k1$ از گراف را در نظر بگیر. اگر مسافت u تا v با به‌کارگیری این راس کمتر است، پس این راس را برای رفتن از u به v به‌کار ببر.
مرحله ۲: مجموعه راس‌های کمکی $\{k_1, k_2\}$ را در نظر بگیر. اگر مسافت u تا v با به‌کارگیری این مجموعه راس‌ها کمتر است، پس آن‌ها را برای رفتن از u به v به‌کار ببر.
...
مرحله n: مجموعه راس‌های کمکی $\{k_1, k_2, ..., k_i\}$ را در نظر بگیر. اگر مسافت u تا v با به‌کارگیری این مجموعه راس‌ها کمتر است، پس آن‌ها را برای رفتن از u به v به‌کار ببر.

منظور از کمک گرفتن از مجموعه‌ی راس‌های $\{k_1, k_2, ..., k_i\}$، به‌کارگیری همه‌ی آن‌ها نیست؛ بلکه تعدادی از آن‌ها که به کوتاه‌تر شدن مسیر کمک می‌کنند، را به کار می‌بریم. برای فهم کامل این الگوریتم باید به الگوی پویای آن نیز دقت کنید. فرض کنید تابع shortestpath(u,v,k) کوتاه‌ترین مسیر از راس u به v را با استفاده از راس‌های کمکی 1 تا k بدست می‌آورد. در این صورت الگوی بازگشتی زیر، روش پویای به کار رفته در این الگوریتم را نشان می‌دهد.

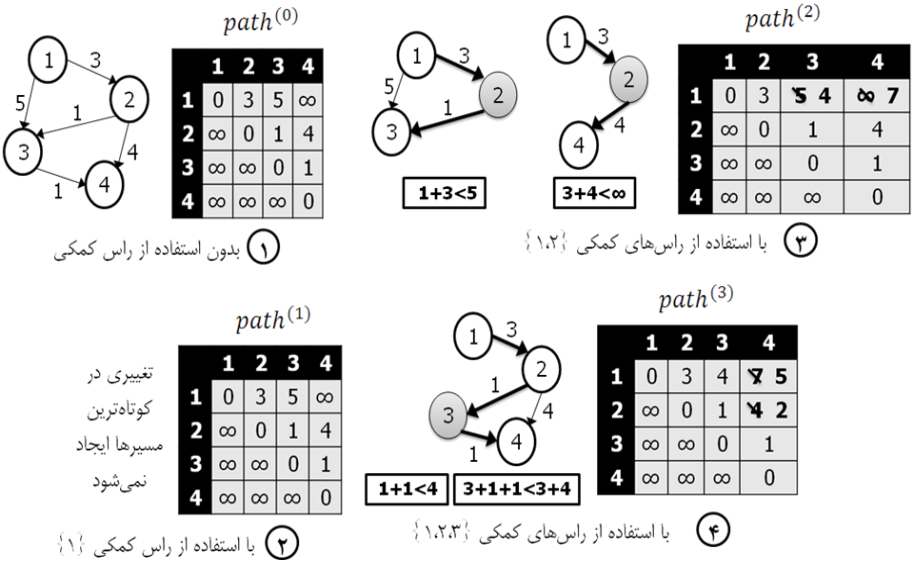
```
shortestpath(u, v, k) = min(shortestpath(u, v, k - 1), shortestpath(u,
k, k - 1) + shortestpath(k, v, k - 1))
shortestpath(u, v, 0) = weight(u, v)
```

الگوی پویای بالا را به صورت زیر معنی می‌کنیم (برای اکثر روش‌های پویا می‌توان به همین شکل عمل کرد). ابتدا دقیقا هدف را مشخص می‌کنیم؛ می‌خواهیم کوتاه‌ترین مسیر از u به v را با به‌کارگیری راس‌های کمکی 1 تا k بدست آوریم. برای یافتن الگوی پویا، باید به دنبال روشی بگردیم که از مرحله یا مرحله‌های پیشین، مرحله‌ی بعدی را بسازد. برای این منظور، فرض می‌کنیم کوتاه‌ترین مسیرها با استفاده از راس‌های کمکی 1 تا $k-1$ را به دست آورده‌ایم. حال راس k را به راس‌های کمکی اضافه کرده و دو حالت زیر را برای آن بررسی می‌کنیم:

۱- راس k در کوتاه‌ترین مسیر از راس u به v شرکت نکند. بنابراین کوتاه‌ترین مسیر از راس u به v همان مقدار پیشین و با به‌کارگیری همان $k-1$ راس کمکی پیشین خواهد بود؛ یعنی shortestpath(u,v,k-1).
۲- راس k در کوتاه‌ترین مسیر از راس u به v شرکت کند. در این حالت کوتاه‌ترین مسیر از راس u به v از راس k نیز عبور خواهد کرد و به صورت shortestpath(u,k,k-1)+shortestpath(k,v,k-1) محاسبه می‌شود.

بنابراین برای هر راس کمکی k هر دو حالت بالا را بررسی کرده و هر کدام که به صرفه‌تر است، برمی‌گزینیم. اگر وجود راس k منجر به کوتاه‌تر شدن مسیر شود، آن را به کار می‌بریم. در پایان، وقتی از همه‌ی n راس گراف به عنوان راس کمکی بهره بگیریم، کوتاه‌ترین مسیرها بدست خواهند آمد. زیرا در این حالت مطمئنا همه‌ی روش‌ها

برای یافتن کوتاه‌ترین مسیرها را آزمایش کرده‌ایم. شکل ۲۹,۴ مراحل توضیح داده شده را برای یک گراف نمونه نمایش می‌دهد.



شکل ۲۹,٤. جزییات طرز کار الگوریتم فلوید-وارشال برای یافتن همه کوتاهترین مسیرهای یک گراف

مرحله‌ی آخر شکل ۲۹,۴ مانند مرحله‌ی دوم اثری در کوتاه‌ترین مسیرها ندارد، بنابراین از شکل حذف شده‌اند. آرایه‌ی دوبعدی path شامل کوتاه‌ترین مسیرها است. $path[u][v]$ طول کوتاه‌ترین مسیر از راس u به v است و $path^{(i)}$ شامل کوتاه‌ترین مسیرها با به‌کارگیری مجموعه راس‌های کمکی $\{k_1, k_2, ..., k_i\}$ می‌باشد. $path^{(0)}[u][v]$ برابر با وزن یال میان راس u و v است؛ یعنی $weight(u,v)$.

براساس الگوی بازگشتی معرفی شده، مراحل جدید را از مرحله‌ی پیش از آن بدست می‌آوریم. برای پیاده‌سازی این الگوریتم در واقع به n آرایه‌ی دوبعدی نیاز داریم ($path^{(0)}..path^{(n)}$) که مقادیر آرایه‌ی $path^{(i)}$ را از روی آرایه‌ی $path^{(i-1)}$ بدست می‌آوریم. الگوی بازگشتی بالا بر پایه‌ی آرایه‌ی $path$ به شکل زیر بازنویسی می‌شود:

$$path^{(k)}[u][v] = \min(path^{(k-1)}[u][v], path^{(k-1)}[u][k] + path^{(k-1)}[k][v])$$
$$path^{(0)}[u][v] = weight(u,v)$$

با دقت به الگوی بالا، درخواهید یافت که به‌کارگیری تنها یک آرایه‌ی $path$ برای پیاده‌سازی این الگوریتم کافی است. زیرا در هر مرحله‌ی تنها به مقادیر مرحله‌ی قبل نیاز داریم. پیاده‌سازی زیر همین قاعده را به کار برده و تنها یک آرایه‌ی $path$ را به‌کار می‌گیرد.

الگوریتم فلوید–وارشال

۱- **صورت مسأله:** یافتن همه‌ی کوتاه‌ترین مسیرها در گراف جهت‌دار و وزن‌دار
۲- **ورودی:** یک گراف جهت‌دار و وزن‌دار
۳- **خروجی:** همه‌ی کوتاه‌ترین مسیرها از هر راس به همه‌ی راس‌های دیگر

```
#define MAX_N 100
int weight[MAX_N][MAX_N];
int path[MAX_N][MAX_N];
void floyd_warshall(int n) {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            path[i][j] = weight[i][j];
    for (int k = 1; k <= n; k++)   // path(k)
        for (int u = 1; u <= n; u++)
            for (int v = 1; v <= n; v++)
                path[u][v] = min(path[u][v], path[u][k] + path[k][v]);
}
```

دو حلقه‌ی تودرتوی آغازین، وزن یال‌های گراف را به عنوان کوتاه‌ترین مسیرهای اولیه (با بـه‌کـارگیری صـفر راس کمکی)، در آرایه‌ی path قرار می‌دهند ($path^{(0)}$). سه حلقه‌ی تودرتـوی بعـدی، بدنـه‌ی اصـلی تـابع را تشـکیل می‌دهنـد. در دور اول از حلقـه بیرونـی (شـمارنده‌ی k)، $path^{(1)}$ محاسـبه می‌شـود؛ در دور دوم $path^{(2)}$ و در دور k_ام، $path^{(k)}$ محاسبه می‌شود. در پایان، آرایه‌ی $path^{(n)}$ دربردارنده‌ی طول همه‌ی کوتـاه‌ترین مسـیرها خواهد بود.

تحلیل پیچیدگی زمانی این تابع بسیار ساده است؛ دو حلقه‌ی آغازین به همراه سه حلقه‌ی تودرتوی پس از آن منجر به پیچیدگی زمانی درجه سه برای الگوریتم فلوید–وارشال می‌شـوند. پیاده‌سـازی سـاده و سـریع ایـن الگـوریتم بـه برنامه‌نویسان این امکان را می‌دهد تا در مواقعی که پیچیدگی زمانی درجه‌ی سه قابل قبول اسـت، ایـن روش را بـه جای روش دیکسترا به کار ببرند.

$$O(n^2 + n^3) \in O(n^3)$$

پیاده‌سازی بالا فقط طول کوتاه‌ترین مسیرها را بدست می‌آورد. برای نگهداری خود مسـیرها می‌تـوان یـک آرایـه‌ی دوبعدی دیگر را به کار برد. به‌کارگیری این آرایـه مشـابه آرایـه تک‌بعـدی parent در روش دیکسـترا خواهـد بـود. پیاده‌سازی زیر کوتاه‌ترین مسیرها را نیز بدست می‌آورد:

```
#define MAX_N 100
int weight[MAX_N][MAX_N];
int path[MAX_N][MAX_N], shortest[MAX_N][MAX_N];
void floyd_warshall(int n) {
    for (int i = 1; i <= n; i++)
```

```
        for (int j = 1; j <= n; j++) {
            path[i][j] = weight[i][j];
            shortest[i][j] = 0;
        }
    for (int k = 1; k <= n; k++)
        for (int u = 1; u <= n; u++)
            for (int v = 1;v <= n; v++)
                if (path[u][k] + path[k][v]<path[u][v]) {
                    path[u][v] = path[u][k] + path[k][v];
                    shortest[u][v] = k;
                }
}
```

آرایه‌ی shortest شامل کوتاه‌ترین مسیرها است. مانند بخش پیشین، تابعی برای دریافت مسیر از هر راس u به راس دلخواه v به این صورت ارائه می‌کنیم:

```
void get_shortest_path(int source, int destination) {
    if (shortest[source][destination] != 0) {
        get_shortest_path(source, shortest[source][destination]);
        cout << shortest[source][destination] << " ";
    }
}
```

برای نمونه، خروجی تابع بالا با فراخوانی get_shortest_path(1,4) برای گراف پیشین به صورت زیر خواهد بود. راس مبدا و مقصد در خروجی چاپ نمی‌شوند.

Output
2 3

٤-۷- مسیر اویلری

مسیر اویلری[۳۷] یکی از مباحث بسیار کاربردی در نظریه گراف است. این موضوع از مسأله‌ی معروف پل‌های کونیگسبرگ[۳۸] نشأت گرفته است که توسط آقای اویلر[۳۹] حل شد. در شهر کونیگسبرگ آلمان، دو جزیره در درون رودخانه‌ای به نام پِرگِل[۴۰] قرار داشت که توسط هفت پل به یکدیگر و سواحل رودخانه متصل بودند. مسأله‌ی پل‌های

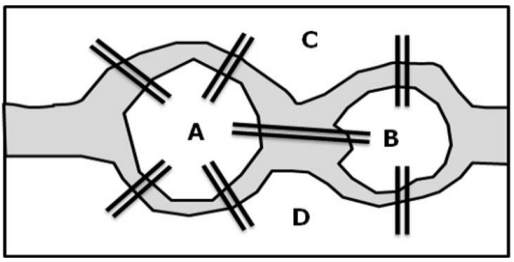
[۳۷] Eulerian path

[۳۸] Bridges of Königsberg

[۳۹] Leonhard Euler

[۴۰] Pregel

کونیگسبرگ به این صورت بود که آیا می‌توان مسیری را با شروع از یکی از مکان‌ها (A، B، C یا D) یافت که از همه‌ی پل‌ها تنها یک بار عبور کرده و به مکان آغاز بازگردد (شکل ۳۰,۴)؟



شکل ٤,۳۰. مسأله پل‌های کونیگسبرگ

مسیری که از همه‌ی یال‌های گراف یکبار عبور کند، مسیر اویلری نامیده می‌شود. اگر این مسیر از یک گره آغاز شده و به همان ختم شود، مدار اویلری یا چرخه‌ی اویلری نامیده می‌شود. گرافی که در آن یک مدار اویلری وجود داشته باشد، گراف اویلری نامیده می‌شود. اگر در گراف، مدار اویلری وجود نداشته باشد ولی مسیر اویلری یافت شود، گراف را نیمه‌اویلری یا قابل عبور می‌نامند.

یک گراف، اویلری است (دارای مدار اویلری است)، اگر و فقط اگر دارای شرایط زیر باشد:

۱- همبند باشد.

۲- درجه‌ی همه‌ی گره‌های آن زوج باشد.

یک گراف، نیمه‌اویلری است (دارای مسیر اویلری است)، اگر و فقط اگر دارای شرایط زیر باشد:

۱- همبند باشد.

۲- درجه‌ی همه‌ی گره‌های آن زوج باشد؛ در این حالت حداکثر دو گره می‌توانند درجه‌ی فرد داشته که باید حتما نقش گره آغازین و پایانی در مسیر اویلری را بازی کنند.

شرایط گفته شده برای گراف‌های بدون‌جهت است. برای گراف‌های جهت‌دار شرایط کمی متفاوت است. یک گراف جهت‌دار اویلری است، اگر و فقط اگر:

۱- قویاً همبند باشد. منظور از قویاً همبند، قابل دسترس بودن هر گره از همه‌ی گره‌های دیگر در گراف جهت‌دار است.

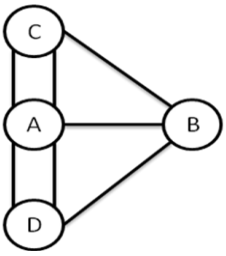
۲- درجه‌ی ورودی و خروجی هر گره برابر باشد. به بیانی دیگر اختلاف درجه‌ی ورودی و خروجی هر گره برابر با صفر باشد.

یک گراف جهت‌دار، نیمه‌اویلری است، اگر و فقط اگر:

۱- قویاً همبند باشد.

۲- اختلاف درجه‌ی ورودی و خروجی همه‌ی گره‌ها غیر از دو گره، صفر باشد. برای دو گره‌ی دیگر، درجه‌ی یکی از آن‌ها ۱ (گره مبدا) و درجه‌ی دیگری ۱- (گره مقصد) خواهد بود.

پل‌های نشان داده شده در شکل ۳۰,۴ را می‌توان به صورت گراف شکل ۳۱,۴ مدل کرد. براساس قوائد بالا، این گراف اویلری نبوده و در نتیجه مسأله پل‌های کونیگسبرگ بدون پاسخ است.



شکل ۳۱,٤. گراف متناظر با مسأله پل‌های کونیگسبرگ شکل ۳۰,٤

در این بخش، الگوریتمی برای یافتن مدار اویلری در گراف بی‌جهت ارائه می‌کنیم. هدف الگوریتم، یافتن مسیری از یک گره مشخص، پیمایش همه‌ی یال‌ها و بازگشت به گره آغازین است. این الگوریتم بر اساس این قاعده عمل می‌کند: با حذف یک چرخه از گراف اویلری، گراف باز هم اویلری باقی خواهد ماند.
بنابراین الگوریتم به دنبال چرخه‌های کوچک‌تر گشته و با حذف یال‌های آن، چرخه‌ی یافت شده را حذف می‌کند. این روال تا زمانی که همه‌ی یال‌های گراف حذف شوند، ادامه پیدا خواهد کرد. مراحل الگوریتم به صورت دقیق‌تر در ادامه ارائه شده است. منظور از مدار در این مراحل، یک پشته است که در پایان دربردارنده‌ی ترتیب پیمایش گره‌های گراف برای تشکیل یک مدار اویلری است.

۱- یک گره را در ابتدا برگزیده و الگوریتم را برای آن فراخوانی می‌کنیم.
۲- گره برگزیده شده را u می‌نامیم. اگر گره u هیچ گره‌ی مجاوری ندارد (هیچ یالی متصل به گره u نیست)، گره u را به مدار اضافه کن و برگرد.
۳- اگر گره u دارای گره‌های مجاور است، گره‌های مجاور را به ترتیب دلخواه برگزیده و هر یک را به صورت زیر پردازش کن. گره‌ی مجاور برگزیده شده را v می‌نامیم.
۳-۱- یال میان گره u و v را حذف کن.
۳-۲- الگوریتم را برای گره v فراخوانی کن (به صورت بازگشتی).
۴- گره v را به مدار اضافه کن.

مراحل گفته شده تا پیمایش همه‌ی یال‌ها ادامه خواهد یافت. شبه‌کد ساده‌ی زیر مراحل توضیح داده شده را بهتر نشان می‌دهد:

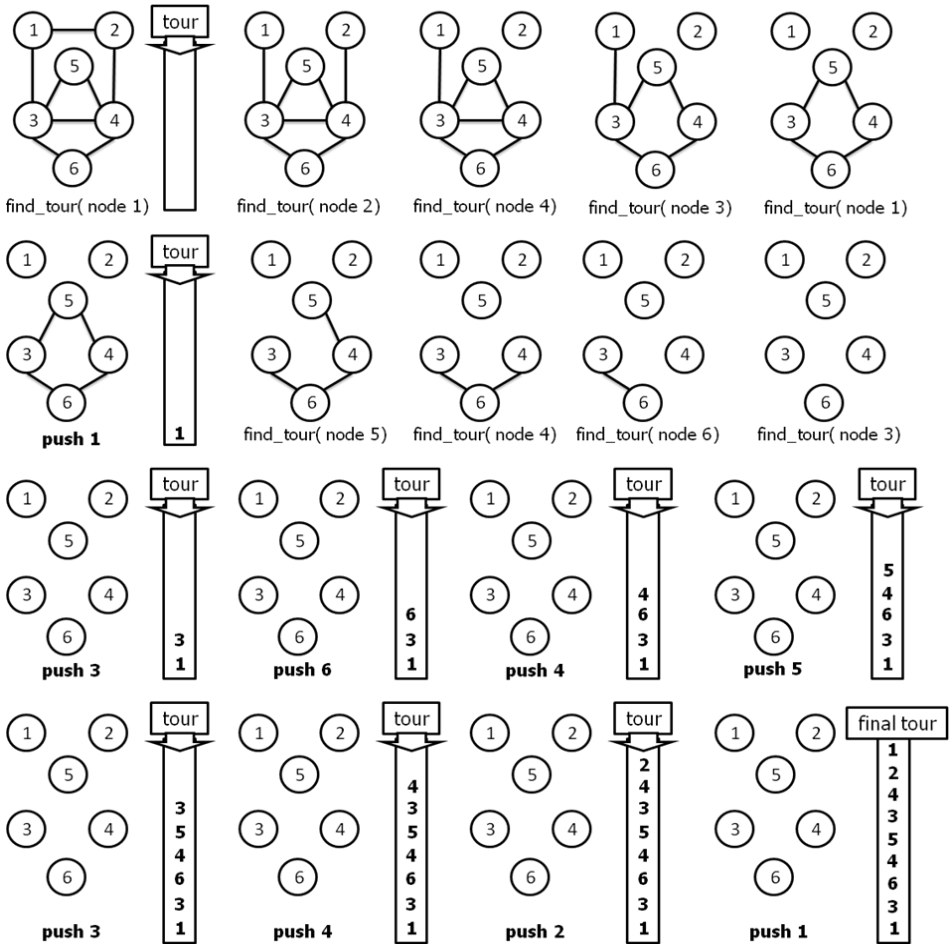
```
stack tour
void find_tour(node u) {
   for each edge e = (u, v) in E {
      remove e from E
```

```
        find_tour(v)
    }
    tour.push(u)
}
```

آغاز الگوریتم می‌تواند با فراخوانی زیر صورت گیرد:

```
find_tour(node 1);
```

برای روشن‌تر شدن توضیحات، الگوریتم را برای یک گراف نمونه، به کار می‌گیریم. شکل ۳۲,۴ مراحل انجام شده توسط الگوریتم را نشان می‌دهد.



شکل ٣٢,٤. مراحل طی شده برای یافتن مدار اویلری یک گراف ساده

ترتیب مرحله‌ها در شکل از چپ به راست و بالا به پایین است. در پایان، اگر مقادیر موجود در پشته را به ترتیب خارج کنید، ترتیب پیمایش گره‌ها را بدست خواهید آورد. اگر به ترتیب اعداد دقت کنید، متوجه خواهید شد که الگوریتم چند چرخه‌ی کوچکتر را برای رسیدن به چرخه‌ی اصلی یافته است. برای نمونه چرخه‌های زیر از جمله چرخه‌های یافت شده هستند:

4 3 5 4
3 5 4 6 3

برای پیاده‌سازی الگوریتم، یک ماتریس مجاورت را برای تشخیص وجود یال بین گره‌ها به کار می‌بریم. برای حذف یال (u,v) از گراف، مقدار خانه‌های $[u][v]$ و $[v][u]$ از ماتریس مجاورت را برابر با false قرار می‌دهیم.

الگوریتم یافتن مدار اویلری

۱- **صورت مسأله:** یافتن یک چرخه‌ی اویلری
۲- **ورودی:** یک گراف بی‌جهت
۳- **خروجی:** دنباله‌ای از گره‌های تشکیل دهنده‌ی مدار اویلری

پیاده‌سازی این الگوریتم می‌تواند به این صورت باشد:

```
#include<iostream>
#include<stack>
using namespace std;
#define MAX_N 100
bool a[MAX_N][MAX_N];   // adjacency matrix
stack<int> tour;
int n;              // number of nodes
void find_tour(int u) {
    for (int v = 1; v <= n; v++)       // search for neighbors
        if (a[u][v]) {                 // v is a neighbors
            a[u][v] = a[v][u] = false; // remove edge (u,v)
            find_tour(v);
        }
    tour.push(u);
}
```

پیچیدگی زمانی این الگوریتم برابر با $O(n+e)$ خواهد بود که n تعداد گره‌ها و e تعداد یال‌ها است. از آنجا که گراف همبند است ($e >= n-1$)، می‌توانیم پیچیدگی زمانی را از مرتبه‌ی $O(e)$ در نظر بگیریم. البته در پیاده‌سازی بالا، ماتریس مجاورت موجب می‌شود که در هر فراخوانی تابع، حلقه‌ی for کل سطر مربوط به گره u را جستجو کند؛ این قضیه موجب افزایش مرتبه‌ی زمانی الگوریتم به $O(ne+e)$ خواهد شد. برای بهبود روش پیاده‌سازی شده، می‌توانیم به جای ماتریس مجاورت، برای هر گره، یک لیست در نظر گرفته که گره‌های مجاور آن را در خود داشته باشد. این بهینه‌سازی موجب حذف پیمایش‌های اضافی خواهد شد.

مسأله‌ی گراف همیلتونی[۴۱] مشابه گراف اویلری است. گرافی که شامل دور همیلتونی[۴۲] باشد، گراف همیلتونی نامیده می‌شود. دور همیلتونی، دوری است که از هر گره‌ی گراف یکبار عبور کرده و به گره آغازین برگردد. مسأله‌ی شناخته شده‌ی فروشنده‌ی دوره‌گرد نیز توسط آقای همیلتون[۴۳] مطرح شده است. در این مسأله، یک فروشنده‌ی دوره‌گرد وجود دارد که قصد سفر به مجموعه‌ای از شهرها برای فروش اجناس خود را دارد. هدف، پیدا کردن مسیری از مبدا فروشنده به همه‌ی شهرهای مورد نظر است، به صورتی که هزینه‌ی سفر کمینه بوده و در پایان به شهر مبدا بازگردد. هنوز هیچ راه‌حل مناسبی از مرتبه‌ی چندجمله‌ای برای این مسأله ارائه نشده است و بهترین راه‌حل‌ها از درجه‌ی نمایی هستند. راه‌حل پویای ارائه شده برای این مسأله از مرتبه‌ی $O(n^2 2^n)$ است که برای تعداد شهر بیشتر از ۲۰ مناسب نیست. بهینه‌سازی‌های زیادی برای این مسأله ارائه شده که هر یک، تعداد شهرهای ممکن را کمی افزایش داده‌اند (تا ۴۰–۶۰ شهر و بعضی تا ۲۰۰ شهر)؛ بهترین پیاده‌سازی این مسأله توانسته یک نمونه با ۸۵۹۰۰ شهر را حل کند.

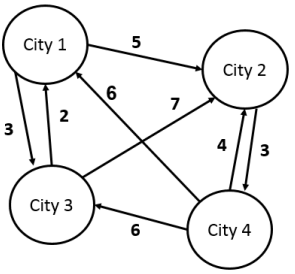
٤–۸– فروشنده دوره‌گرد

فروشنده‌ای را در نظر بگیرید که به شهرهای مختلف برای فروش اجناس خود سفر می‌کند. این فروشنده قصد دارد از شهر محل زندگی خود سفر را آغاز کرده و پس از طی کردن همه‌ی شهرهای موجود در مسیر مورد نظرش، به شهر مبدا بازگردد. نیاز او یافتن مسیری است که کمترین هزینه را داشته باشد.

این مسأله را می‌توان به یک گراف وزن‌دار و جهت‌دار نگاشت کرد. مسیری که از یک گره در گراف آغاز شده و پس از دیدن همه‌ی گره‌ها (دقیقا یک بار) به گره‌ی مبدا باز گردد، مدار همیلتونی نامیده می‌شود. گراف نمایش داده شده در شکل ۳۳,۴ را در نظر بگیرید. دو مدار همیلتونی موجود در آن و هزینه‌ی هر یک در ادامه آورده شده است. مدار اول، مسیر بهینه و راه‌حل مسأله محسوب می‌شود. هدف بدست آوردن این مسیر است.

$1 \to 2 \to 4 \to 3 \to 1$, $Cost = 16$

$1 \to 3 \to 2 \to 4 \to 1$, $Cost = 19$



[۴۱] Hamiltonian Graph

[۴۲] Hamiltonian Cycle

[۴۳] William Rowan Hamilton

شکل ۳۳,٤. یک گراف وزن‌دار و جهت‌دار نمونه برای مسأله فروشنده دوره‌گرد

گره آغازین مهم نیست و در مدار همیلتونی تاثیری ندارد. بنابراین همیشه گره اول را به عنوان نقطه شروع در نظر می‌گیریم. یک راه‌حل ساده، بررسی همه‌ی مسیرهای ممکن است. اگر گرافی با n گره را فرض کنیم. پس از انتخاب گره اول، برای انتخاب گره دوم، $n-1$ گزینه در اختیار خواهیم داشت. برای انتخاب گره سوم، $n-2$ و به همین ترتیب برای گره آخر، یک گزینه وجود خواهد داشت. بنابراین تعداد کل حالت‌هایی که مورد بررسی قرار می‌گیرد، برابر است با $(n-1)(n-2)(n-3)...1=(n-1)!$ که پیچیدگی زمانی از مرتبه فاکتوریل را به همراه دارد.

اگر یک مدار همیلتونی بهینه (با وزن کمینه) را در نظر بگیرید. هر زیرمجموعه‌ای از این مدار نیز دارای وزن کمینه است. بنابراین می‌توان برای حل این مسأله از روش برنامه‌نویسی پویا استفاده کرد. ماتریس مجاورت گراف شکل ۳۳,۴ را با A و به صورت زیر نمایش می‌دهیم. ∞ به معنی عدم وجود مسیر بین دو گره است. وزن‌ها مقادیری غیرمنفی هستند.

$$A=\begin{bmatrix}0 & 5 & 3 & \infty\\ \infty & 0 & \infty & 3\\ 2 & 7 & 0 & \infty\\ 6 & 4 & 6 & 0\end{bmatrix}$$

برای توضیح روش پویا، دو مجموعه و یک آرایه‌ی دوبعدی را به صورت زیر تعریف می‌نماییم. ابتدا زیرمسأله‌ها را حل کرده و سپس با استفاده از زیرمسأله، به حل مسأله‌ی اصلی می‌پردازیم. برای گراف شکل ۳۳,۴، نحوه‌ی استفاده از این سه را توضیح می‌دهیم.

- V: مجموعه‌ی همه گره‌های گراف (n گره)
- SV: مجموعه‌ی گره‌های کمکی: زیرمجموعه‌ای از V
- $D[n][2^{n-1}]$: آرایه‌ی دوبعدی برای نگهداری هزینه کوتاه‌ترین مسیرها از هر گره به گره مبدا

$V=\{1,2,3,4\}$

$D[4][SV]$ یعنی هزینه‌ی حرکت از گره شماره ۴ به گره مبدا (۱) با استفاده از گره‌های کمکی موجود در SV. اگر $SV=\{3\}$ باید، $D[4][\{3\}]=Cost(4,3,1)=8$ خواهد بود. در صورتی که $SV=\emptyset$ مقدار $D[i][\emptyset]=A[i][1]$ خواهد بود. برای مثال، $D[4][\emptyset]=6$.

در صورتی که $SV=\{2,3\}$ باید، برای محاسبه‌ی $D[4][SV]$ باید همه‌ی مسیرهایی که از گره ۴ آغاز شده و پس از گذر از گره‌های ۲ و ۳ به گره مبدا می‌رسند، در نظر بگیریم. برای این منظور، باید یک بار فرض کنیم که ابتدا از گره ۲ و سپس از گره ۳ عبور کرده و یک بار عکس این ترتیب در نظر گرفته شود (ابتدا گره ۳ و سپس گره ۲).

$$D[4][\{2,3\}]=\min(Cost(4,2,3,1)=\infty, Cost(4,3,2,1)=\infty)=\infty$$

به صورت کلی، هزینه کمینه‌ی هر مسیر از گره شماره i به گره مبدا با استفاده از گره‌های موجود در مجموعه SV برابر است با:

$$D[i][SV] = \begin{cases} \min(A[i][j] + D[j][SV - \{j\}]) \quad j = 2..n & SV \neq \emptyset \\ A[i][1] & SV = \emptyset \end{cases}$$

هزینه یک مسیر بهینه از گره مبدا که به خود آن برمی‌گردد ($SV = V - \{1\}$) برابر خواهد بود با:

$$Global\ Min\ Cost = D[1][V - \{1\}] = \min(A[1][j] + D[j][V - \{1, j\}]) \quad j = 2..n$$

برای بدست آوردن مدار هملیتونی کمینه در گراف شکل ۳۳,۴ به صورت نشان داده شده در شکل ۳۴,۴ عمل می‌کنیم.

$$A = \begin{bmatrix} 0 & 5 & 3 & \infty \\ \infty & 0 & \infty & 3 \\ 2 & 7 & 0 & \infty \\ 6 & 4 & 6 & 0 \end{bmatrix}$$

$SV =$ بدون عضو

$$\begin{bmatrix} D[2][\emptyset] = A[2][1] = \infty \\ D[3][\emptyset] = A[3][1] = 2 \\ D[4][\emptyset] = A[4][1] = 6 \end{bmatrix}$$

$SV =$ یک عضو

$$\begin{bmatrix} D[3][\{2\}] = A[3][2] + D[2][\emptyset] = 7 + \infty = \infty \\ D[4][\{2\}] = A[4][2] + D[2][\emptyset] = 4 + \infty = \infty \\ D[2][\{3\}] = A[2][3] + D[3][\emptyset] = \infty + 2 = \infty \\ D[4][\{3\}] = A[4][3] + D[3][\emptyset] = 6 + 2 = 8 \\ D[2][\{4\}] = A[2][4] + D[4][\emptyset] = 3 + 6 = 9 \\ D[3][\{4\}] = A[3][4] + D[4][\emptyset] = \infty + 6 = \infty \end{bmatrix}$$

$SV =$ دو عضو

$$\begin{bmatrix} D[4][\{2,3\}] \\ = \min(A[4][2] + D[2][\{3\}] = 4 + \infty, A[4][3] + D[3][\{2\}] = 6 + \infty) = \infty \\ D[3][\{2,4\}] \\ = \min(A[3][2] + D[2][\{4\}] = 7 + 9, A[3][4] + D[4][\{2\}] = \infty + \infty) = 16 \\ D[2][\{3,4\}] \\ = \min(A[2][3] + D[3][\{4\}] = \infty + \infty, A[2][4] + D[4][\{3\}] = 3 + 8) = 11 \end{bmatrix}$$

$SV =$ سه عضو مرحله آخر

$$\begin{bmatrix} D[1][\{2,3,4\}] = \min(\\ \quad A[1][2] + D[2][\{3,4\}] = 5 + 11, \\ \quad A[1][3] + D[3][\{2,4\}] = 3 + 16, \\ \quad A[1][4] + D[4][\{2,3\}] = \infty + \infty) = 16 \end{bmatrix}$$

شکل ٣٤,٤. حل مسأله فروشنده دوره‌گرد برای گراف شکل ٣٣,٤ به روش پویا

ابتدا در مجموعه گره‌های کمکی، صفر گره قرار داده و در انتها ، همه گره‌ها (غیر از گره مبدا) را در آن قرار می‌دهیم. همان‌طور در شکل می‌بینید، دو مدار نشان داده شده در ابتدای این بخش، در مرحله‌ی آخر بدست آمده و مسیر بهتر انتخاب شده است. در ادامه، شبه‌کدی برای این الگوریتم ارائه شده است.

الگوریتم یافتن مدار همیلتونی بهینه

۱- **صورت مسأله:** یافتن یک چرخه با هزینه کمینه در یک گراف وزن‌دار و جهت‌دار

۲- **ورودی:** یک گراف وزن‌دار و جهت‌دار

۳- **خروجی:** دنباله‌ای از گره‌های تشکیل دهنده‌ی مدار همیلتونی (و هزینه‌ی آن) به صورتی که هزینه‌ی آن کمینه باشد

شبه‌کد زیر، هزینه‌ی مدار بهینه و خود مدار را محاسبه می‌نماید. آرایه‌ی دوبعدی A، ماتریس مجاورت گراف ورودی است. آرایه P برای نگهداری مسیرها استفاده می‌شود. با استفاده از این آرایه، می‌توانیم در پایان مسیر بهینه را بدست آوریم. تابع ارائه شده، هزینه‌ی کمینه را باز می‌گرداند.

```
int TravellingSalesman(int n, int A[][], int P[][]) {
    int D[n][2^(n-1)]
    for (int i = 2; i <= n; i++) {
        D[i][ ∅] = A[i][1]
        P[i][ ∅] = 1
    }
    for (int len = 1; len <= n - 2; len++)
        for (all subsets SV ⊆ V – {v1} containing exactly len vertices)
            for (int i such that i ≠ 1 and vi is not in SV) {
                D[i][SV] = min(A[i][j] + D[j][SV - { vj}]    j: vj∈SV
                P[i][SV] = value of j that gave the minimum
            }
    D[1][V - {v1}] = min(A[1][j] + D[j][V - {v1, vj}]    j = 2..n
    P[1][V - {v1}] = value of j that gave the minimum
    return D[1][V - {v1}]
}
```

برای تحلیل پیچیدگی زمانی الگوریتم، هر چهار حلقه‌ی شبه‌کد را بررسی می‌کنیم. حلقه اول، $n-2$ مرتبه تکرار می‌شود. حلقه‌ی دوم به تعداد زیرمجموعه‌های SV با len عضو تکرار می‌شود که برابر است با $\binom{n-1}{len}$. حلقه ی سوم به تعداد گره‌هایی که خارج از مجموعه SV قرار دارند، تکرار می‌شود، یعنی $n-len-1$ مرتبه. حلقه چهارم به تعداد اعضای مجموعه SV تکرار می‌شود (len مرتبه). مجموع این تکرارها برابر است با:

$$T(n)=\sum_{len=1}^{n-2}(n-len-1)len\binom{n-1}{len}$$

از طرفی داریم:

$$(n-k-1)\binom{n-1}{k} = (n-k-1)\frac{(n-1)!}{k!\,(n-k-1)!}$$

$$= (n-1)\frac{(n-2)!}{k!\,(n-k-2)!} = (n-1)\binom{n-2}{k}$$

با جایگذاری عبارت بالا در $T(n)$ خواهیم داشت:

$$T(n) = (n-1)\sum_{len=1}^{n-2} len\binom{n-2}{len}$$

از آنجا که: ($k\binom{n}{k} = n\binom{n-1}{k-1}$)

$$\sum_{k=1}^{n} k\binom{n}{k} = n\sum_{k=1}^{n}\binom{n-1}{k-1} = n\sum_{k=1}^{n-1}\binom{n-1}{k} = n(2^{n-1}-1) \approx n2^{n-1}$$

برای روشن‌تر شدن فرمول بالا، عبارت $\sum_{k=0}^{n}\binom{n}{k}$ را برای چند n ابتدایی رسم کرده‌ایم. در شکل ۳۵,۴ ملاحظه می‌کنید که حاصل، مثلث خیام–پاسکال است. اگر به مجموع ارقام موجود در هر سطر توجه کنید، خواهید دید که مجموع ارقام سطر i_ام برابر با 2^i است.

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

شکل ٣٥,٤. رسم عبارت $\sum_{k=0}^{n}\binom{n}{k}$ را برای چند n ابتدایی

در نتیجه پیچیدگی زمانی این الگوریتم برابر خواهد بود با:

$$T(n) = (n-1)(n-2)2^{n-3} \in O(n^2 2^n)$$

برای تحلیل پیچیدگی فضایی شبه‌کد ارائه شده، سه آرایه‌ی A، P و D را مورد بررسی قرار می‌دهیم. فضای اشغالی توسط ماتریس مجاورت A برابر با $O(n^2)$ می‌باشد. اندازه‌ی دو آرایه‌ی دوبعدی P و D برابر است. این دو آرایه دارای n سطر و 2^{n-1} ستون هستند که 2^{n-1} تعداد اعضای مجموعه‌ی توانی مربوط به مجموعه V است. بنابراین، پیچیدگی فضایی این الگوریتم برابر خواهد بود با:

$$M(n) = n^2 + 2(n2^{n-1}) \in O(n2^n)$$

روش پویا منجر به پیچیدگی زمانی و فضایی از مرتبه‌ی نمایی شده است. این روش برای n_های بزرگتر از ۴۰ بسیار کند عمل خواهد کرد. هرچند در مقایسه با روش ابتدایی که همه‌ی راه‌حل‌های ممکن را بررسی می‌کرد، بسیار بهتر است. برای مثال، اگر بخواهیم مسأله‌ی فروشنده‌ی دوره‌گرد را با ۳۰ شهر حل کنیم، روش اول به $29!$ میکروثانیه زمان نیز دارد (بیش از چند صد هزار سال). در صورتی که روش پویا به $29 \times 28 \times 2^{27}$ میکروثانیه زمان نیاز دارد (۳۰ ساعت).

و اما نحوه استفاده‌ی از آرایه‌ی P برای بدست آوردن مسیر بهینه. خانه‌ی $P[i][SV]$ را به این صورت معنی می‌کنیم. برای حرکت از گره‌ی i_ام با استفاده از گره‌های موجود در مجموعه‌ی SV، باید ابتدا به کدام گره پرش کنیم؟ برای مثال اگر داشته باشیم، $P[4][\{2,3\}] = 3$، یعنی برای حرکت از گره ۴ به گره ۱ (مبدا) با استفاده از گره‌های ۲ و ۳، باید ابتدا به گره ۳ پرش کنیم. گام بعدی حرکت، از طریق گره ۳ مشخص می‌شود ($P[3][\{2\}] = 2$)؛ و به همین ترتیب تا رسیدن به گره‌ی مبدا ادامه می‌دهیم. برای گراف، خانه‌های اصلی آرایه ی P را در ادامه نشان داده‌ایم. می‌بینیم که خروجی سطر اول، به عنوان نمایه در سطر بعد به کار گرفته شده است و از مجموعه‌ی گره‌های کمکی نیز حذف شده است. خروجی آخرین سطر، گره مبدا است.

$$P[1][\{2,3,4\}] = 2$$
$$P[2][\{3,4\}] = 4$$
$$P[4][\{3\}] = 3$$
$$P[3][\emptyset] = 1$$

در ادامه، پیاده‌سازی انجام شده به زبان سی++ را ارائه کرده‌ایم. در این کد، ابتدا مجموعه‌ی توانی مربوط به مجموعه‌ی V ساخته شده (تابع makeallsubsets) و به هر کدام نمایه‌ای نسبت داده شده است. از این نمایه به عنوان نماینده‌ی آن مجموعه استفاده کرده‌ایم. آرایه‌ی $SV[i]$ شامل همه زیرمجموعه‌های به طول i است. خانه‌ی $SVlength[i]$، تعداد مجموعه‌های موجود در $SV[0]$ تا $SV[i]$ را در خود دارد. تابع findset، یک زیرمجموعه set و یک گره vj دریافت کرده و نمایه‌ی زیرمجموعه‌ای که برابر با $set - \{vj\}$ باشد را می‌یابد.

```
#include<iostream>
#include<vector>
using namespace std;
#define INF INT_MAX
struct subset {
    vector<int> vertices;
    int index;
public:
    subset() {
        index = 0;
    }
    bool exist(int v) {
        for (int i = 0; i < vertices.size(); i++)
```

```
            if (vertices[i] == v)
                return true;
        return false;
    }
};
const int N = 4;            // number of vertices
vector<subset> SV[N];           // SV[i]: all subsets of V with length i
int SVlength[N];            // cumulative sum of length of SV
int A[N + 1][N + 1];        // adjacency matrix
int findset(subset set, int notvj) {
    int len = set.vertices.size() - 1;
    for (int i = 0; i<SVlength[len] - SVlength[len - 1]; i++) {
        subset currset = SV[len][i];
        int first = 0, second = 0;
        bool found = true;
        while (first < set.vertices.size() && second <
currset.vertices.size()) {
            if (set.vertices[first] == notvj) {
                first++;
                continue;
            }
            if (currset.vertices[second] == notvj) {
                found = false;
                break;
            }
            if (set.vertices[first] == currset.vertices[second]) {
                first++;
                second++;
            }
            else {
                found = false;
                break;
            }
        }
        if (found)
            return currset.index;
    }
    return -1;
}
int travellingsalesman() {
    int *D[N + 1], *P[N + 1];
    for (int i = 1; i <= N; i++) {        // define D[n][2^(n-1)] &
P[n][2^(n-1)]
        D[i] = new int[SVlength[N - 1]];
        P[i] = new int[SVlength[N - 1]];
    }
    for (int i = 2; i <= N; i++) {
        D[i][0] = A[i][1];
        P[i][0] = 1;
    }
    for (int len = 1; len <= N - 2; len++) {
```

```
        for (int si = 0; si<SVlength[len] - SVlength[len - 1]; si++) {
            subset currSubset = SV[len][si];
            for (int i = 2; i <= N; i++) {
                if (currSubset.exist(i))
                    continue;
                D[i][currSubset.index] = INF;
                for (int j = 0; j<currSubset.vertices.size(); j++) {
                    int newval = A[i][currSubset.vertices[j]] +
                        D[currSubset.vertices[j]][findset(currSubset,
currSubset.vertices[j])];
                    newval = (newval<0) ? INF : newval;
                    if (newval<D[i][currSubset.index]) {
                        D[i][currSubset.index] = newval;
                        P[i][currSubset.index] = currSubset.vertices[j];
                    }
                }
            }
        }
    }
    subset currSubset = SV[N - 1][0];
    D[1][currSubset.index] = INF;
    for (int j = 0; j<currSubset.vertices.size(); j++) {
        int newval = A[1][currSubset.vertices[j]] +
            D[currSubset.vertices[j]][findset(currSubset,
currSubset.vertices[j])];
        newval = (newval<0) ? INF : newval;
        if (newval<D[1][currSubset.index]) {
            D[1][currSubset.index] = newval;
            P[1][currSubset.index] = currSubset.vertices[j];
        }
    }
    return D[1][currSubset.index];
}
void makeallsubsets(int len, int start, subset& currset) {
    if (currset.vertices.size() == len)   {
        if (SVlength[len] == 0)
            SVlength[len] += SVlength[len - 1];
        currset.index = SVlength[len];
        SV[len].push_back(currset);
        SVlength[len]++;
        return;
    }
    for (int i = start; i <= N; i++) {
        currset.vertices.push_back(i);
        makeallsubsets(len, i + 1, currset);
        currset.vertices.pop_back();
    }
}
int main() {
    // make all subsets of V
    subset set;
```

```
    SV[0].push_back(set);
    SVlength[0] = 1;
    for (int i = 1; i<N; i++) {
        subset set;
        makeallsubsets(i, 2, set);
    }
    cout << travellingsalesman() << endl;
    return 0;
}
```

با استفاده از آرایه P، می‌توانیم مسیر بهینه را بدست آوریم. تابع زیر، توضیحات داده شده در این رابطه را پیاده‌سازی کرده است:

```
void printpath(int *P[])
{
    int len = N - 1;
    int currV = 1, currIndex = SV[len][0].index;
    cout << "1 -> ";
    while (P[currV][currIndex] != 1)
    {
        cout << P[currV][currIndex] << " -> ";
        currV = P[currV][currIndex];
        currIndex = findset(SV[len][currIndex - SVlength[len - 1]],
currV);
        len--;
    }
    cout << "1" << endl;
}
```

با دادن گراف شکل ۳۳,۴ به این برنامه، مقدار ۱۶ در خروجی چاپ خواهد شد. با ارسال آرایه P برای تابع printpath، مسیر بهینه نیز به صورت زیر چاپ خواهد شد:

```
Output
1 -> 2 -> 4 -> 3 -> 1
```

٤-۹- کدگذاری هافمن

فشرده‌سازی از مباحثی است که حتی با رشد حافظه‌های جانبی امروزی نیز دارای اهمیت ویژه‌ای است. کدگذاری هافمن[۴۴] روشی برای کدگذاری داده‌ها است که توسط هافمن[۴۵] ارائه شده است. در این بخش به توضیح آن پرداخته و الگوریتمی برای آن طراحی خواهیم کرد.

[۴۴] Huffman Coding

یکی از روش‌های ذخیره‌ی اطلاعات، شکل دودویی آن است. در این روش برای هر نویسه، یک کد دودویی در نظر گرفته می‌شود. دو روش برای کد کردن داده‌ها به کار گرفته می‌شود:

۱- کد دودویی طول ثابت

۲- کد دودویی طول متغیر

در کد دودویی طول ثابت، طول کدهای اختصاص داده شده به همه‌ی نویسه‌ها، یکسان است. برای نمونه، برای کد کردن رشته‌ی زیر، به هر نویسه یک کد دودویی با طول یکسان نسبت می‌دهد.

abcbacaaabca

برای نمونه اگر کدهای دودویی زیر را به هر نویسه نسبت دهیم (طول همه‌ی کدها برابر با دو است):

$a:00$ $b:01$ $c:11$

کدگذاری رشته‌ی بالا به شکل زیر خواهد بود.

0001110100 1100000001 1100

در روش کد دودویی طول متغیر، کدی با طول متغیر به هر نویسه نسبت داده می‌شود. با به‌کارگیری این ویژگی، می‌توانیم کدگذاری را بهتر انجام دهیم؛ به صورتی که طول کد حاصل کوچکتر و بهینه‌تر شود. یکی از این روش‌ها که موجب کوچکتر شدن کد می‌شود، اختصاص دادن کد دودویی با طول کمتر، به نویسه‌هایی است که در رشته بیشتر تکرار می‌شوند. این نکته، پایه‌ی کدگذاری هافمن است. برای نمونه رشته‌ی بالا را به روش کد دودویی طول متغیر کدگذاری می‌کنیم. برای این منظور کدهای زیر را به سه نویسه‌ی موجود در رشته اختصاص می‌دهیم:

$a:0$ $b:10$ $c:11$

نویسه‌ی a بیشتر از بقیه‌ی نویسه‌ها در رشته تکرار شده است؛ بنابراین کد دودویی کوچکتری به آن اختصاص می‌دهیم. کدگذاری حاصل به صورت زیر خواهد بود. طول کد دودویی بدست آمده، برابر با ۱۸ است، در صورتی که طول کد دودویی حاصل از روش طول ثابت ۲۴ بود. می‌بینیم که روش طول متغیر فضای کمتری اشغال کرده و بهینه‌تر است.

0101110011 00010110

٤-۹-۱- کد پیشوندی

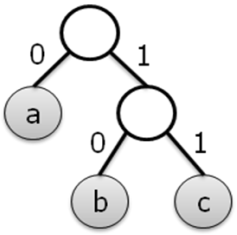
کد پیشوندی نوعی کد طول متغیر است که دارای ویژگی پیشوندی است. یعنی کد مربوط به هیچ نویسه‌ای آغاز کد نویسه‌ی دیگری نیست. برای نمونه اگر کد ۱۰ را به یک نویسه‌ی اختصاص دادید، نمی‌توانید کدی مانند ۱۰۱ و هر

---

[۴۵] David A. Huffman

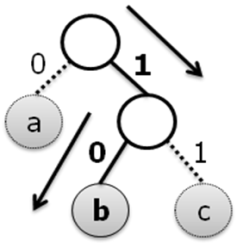
کد دیگری که با ۱۰ آغاز شده را به نویسه‌ی دیگری نسبت دهید. این ویژگی موجب می‌شود که به راحتی و فقط با یک پیمایش ساده، کد تولید شده را رمزگشایی کرده و رشته‌ی اصلی را بدست آوریم. همه‌ی کدهای پیشوندی را می‌توان در قالب یک درخت دودویی نمایش داد به شکلی که برگ‌های آن، نویسه‌هایی هستند که باید کدگذاری شوند و هر یال آن مشخص کننده یک بیت از کد است (۰ یا ۱). برای نمونه درخت دودویی شکل ۳۶,۴ کدهای زیر را نمایش می‌دهد.

$$a:0 \qquad b:10 \qquad c:11$$



شکل ٣٦,٤. نمونه‌ای از یک درخت دودویی برای کدگذاری نویسه‌های a، b و c

برای یافتن نویسه‌ی مربوط به هر کد، از ریشه‌ی درخت دودویی آغاز کرده و با دیدن هر ۰ به سمت چپ و هر ۱ به سمت راست درخت حرکت می‌کنیم. با رسیدن به برگ درخت، نویسه‌ی موجود در برگ، نویسه‌ی مورد نظر است. برای نمونه برای یافتن نویسه‌ی مربوط به کد ۱۰ به صورت نشان داده شده در شکل ۳۸,۴ عمل می‌کنیم.



شکل ٣٨,٤. نحوه یافتن نویسه متناظر با کد ۱۰ در درخت دودویی شکل ٣٦,٤

٤-۹-۲- کد هافمن

اساس روش کدگذاری هافمن، یافتن یک کد دودویی بهینه برای هر نویسه است. این روش با ایجاد یک درخت دودویی، کد دودویی بهینه‌ی مربوط به هر نویسه را می‌سازد. برای ساخت درخت، به ساختمان‌داده‌ای برای نگهداری هر گره و فرزندان آن نیاز داریم. علاوه بر این، فراوانی هر نویسه در رشته‌ی ورودی نیز مورد نیاز است. همان‌طور که پیش از این گفته شد، کدی با طول کمتر به نویسه‌ای با فراوانی بیشتر اختصاص می‌یابد تا طول کد حاصل بهینه شود. ساختمان‌داده‌ی مورد نیاز باید دارای چهار بخش زیر باشد:

۱- بخشی برای نگهداری نویسه‌ی ورودی

۲- بخشی برای نگهداری فراوانی نویسه در رشته‌ی ورودی

۳- اشاره‌گری به فرزند چپ
۴- اشاره‌گری به فرزند راست

برای هر نویسه، یک عنصر از ساختمان‌داده‌ی بالا ساخته و همه‌ی آن‌ها را در یک صف اولویت‌دار قرار می‌دهیم. هر عنصر را یک گره می‌نامیم. اولویت گره‌ها بر اساس فراوانی آن‌ها تعیین می‌شود. نویسه‌ای با فراوانی کمتر، اولویت بیشتری داشته و در آغاز صف قرار خواهد گرفت. مراحل الگوریتم هافمن به صورت زیر است:

۱- دو گره‌ی آغاز صف را برمی‌داریم.
۲- گره جدیدی ایجاد کرده و چهار بخش آن را به صورت زیر تعیین می‌کنیم:
۲-۱- بخش نویسه‌ی آن را بدون مقدار رها می‌کنیم.
۲-۲- فراوانی آن را برابر با مجموع فراوانی دو گره برگزیده شده قرار می‌دهیم.
۲-۳- اشاره‌گر چپ را به گره اول و اشاره‌گر راست را به گره دوم اشاره می‌دهیم.
۳- گره جدید را به صف اولویت‌دار اضافه می‌کنیم.
۴- مرحله‌های ۱ تا ۳ را تا زمانی که فقط یک گره در صف باقی بماند، ادامه می‌دهیم.

برای پیاده‌سازی الگوریتم، ساختمان‌داده‌ی زیر را در زبان سی++، معادل با توضیحات گفته شده طراحی می‌کنیم:

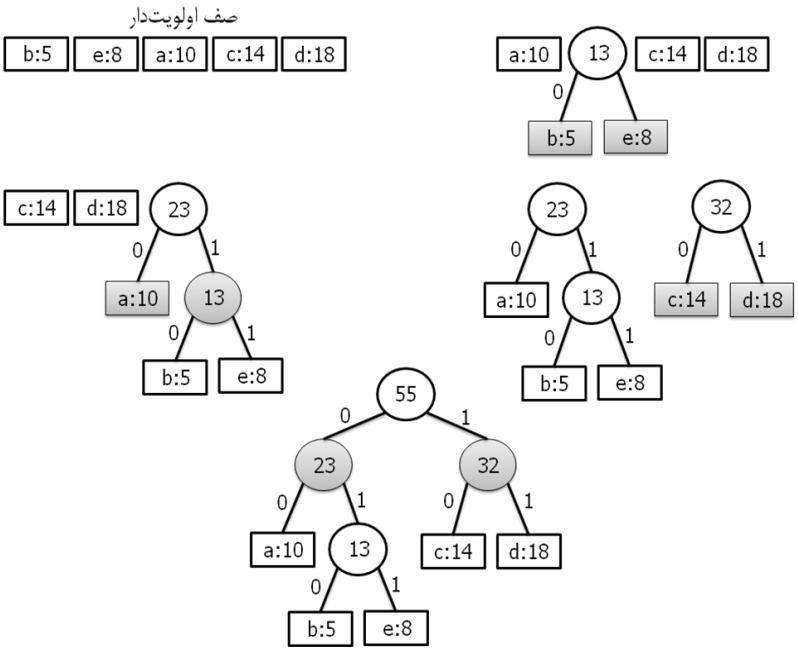
```
struct node {
    char symbol;  // the value of character
    int weight;   // the frequency of symbol in input string
    node* left;   // link to left child
    node* right;  // link to right child
};
```

شبه‌کد زیر مراحل گفته شده را پیاده‌سازی می‌کند:

```
assume pqueue as a priority queue
node huffman_coding() {
    while (pqueue is not empty) {
        pop(pqueue, n1)
        pop(pqueue, n2)
        create new_node
        new_node.weight = n1.weight + n2.weight
        new_node.left = n1
        new_node.right = n2
        push(pqueue, new_node)
    }
    pop(pqueue, rnode)
    return rnode
}
```

pqueue یک صف اولویت‌دار است که اولویت گره‌ها را بر اساس فراوانی کمتر تعیین می‌کند. کلاس معادل سی++ صف اولویت‌دار را برای پیاده‌سازی واقعی الگوریتم به کار خواهیم برد. برای روشن‌تر شدن توضیحات، نویسه‌های زیر و فراوانی تعیین شده برای آن‌ها را در نظر گرفته و الگوریتم هافمن را اجرا می‌کنیم. شکل ۳۸,۴ مراحل اجرای الگوریتم را نشان می‌دهد.

$$a:10 \qquad b:5 \qquad c:14 \qquad d:18 \qquad e:8$$



شکل ٣٨,٤. مراحل اجرای الگوریتم هافمن

ترتیب مرحله‌ها از چپ به راست و از بالا به پایین است. با توجه به درخت دودویی بدست آمده، کد مربوط به هر نویسه به صورت زیر خواهد بود:

$$a:00 \qquad b:010 \qquad c:10 \qquad d:11 \qquad e:011$$

همان‌طور که ملاحظه می‌کنید، نویسه‌هایی که فراوانی بیشتری دارند، در عمق‌های نزدیک به ریشه‌ی درخت و دیگر نویسه‌ها در عمق‌های بیشتر قرار می‌گیرند.

پیش از ارائه‌ی پیاده‌سازی، به معرفی کلاس priority_queue در سرآیند queue می‌پردازیم. این کلاس در پس‌زمینه، از ساختمان‌داده‌ی هرم استفاده می‌کند؛ بنابراین، قرار دادن گره‌های جدید در صف به سرعت صورت خواهد گرفت. برای مثال، نمونه کد زیر، یک صف اولویت‌دار از پنج عدد صحیح ایجاد کرده و سپس آن‌ها را به ترتیب اولویت در خروجی چاپ می‌کند:

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    int a[] = {1, 5, 8, 2, 6};
    priority_queue<int> pq(a, a + 5);
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
    cout << endl;
    return 0;
}
```

خروجی حاصل از اجرای کد بالا به صورت زیر خواهد بود:

Output
8 6 5 2 1

در حالت پیش‌فرض، اولویت مقادیر بزرگتر، بیشتر است. تعریف استفاده شده برای صف اولویت‌دار بالا، معادل با تعریف زیر است:

```
priority_queue<int, vector<int>, less<int>> pq(a, a + 5);
```

حال هر یک از سه بخش تعریف بالا (بخش‌های مربوط به الگو) را توضیح می‌دهیم:

- **بخش اول- int**: نوع عناصری که در صف قرار می‌گیرند.
- **بخش دوم- <vector<int>**: این تعریف، نوع (کلاس) بردار را به صف اولویت‌دار برای نگهداری عنصرها پیشنهاد می‌کند.
- **بخش سوم- <less<int>**: کلاسی برای مقایسه دو عنصر است. اگر عنصر اول کوچکتر باشد، مقدار true را برمی‌گرداند. صف اولویت‌دار با به‌کارگیری این کلاس، عنصرها را به صورت نزولی مرتب می‌کند (برای عنصرهایی با مقدار بیشتر، اولویت بیشتر قائل می‌شود).

برای تغییر اولویت به صورت صعودی، تعریف بالا را به صورت زیر تغییر می‌دهیم:

```
priority_queue<int, vector<int>, greater<int>> pq(a, a + 5);
```

برای تعیین اولویت سفارشی، می‌توانیم تابع مقایسه‌کننده‌ی مورد نظرمان را به کلاس معرفی کنیم. برای نمونه، صف اولویت‌داری از نوع ساختار node ایجاد کرده و تابع مقایسه‌کننده‌ی زیر را برای آن می‌نویسیم. این تابع،

اولویت گره‌هایی با فراوانی کمتر را بیشتر از بقیه‌ی گره‌ها قرار می‌دهد. حال که با عملکرد صف اولویت‌دار آشنا شدید، به پیاده‌سازی اصلی می‌پردازیم.

```
priority_queue<node, vector<node>, mycomparison> pq;
class mycomparison {
public:
    bool operator() (const node left, const node right) const {
        return (left.weight > right.weight);
    }
};
```

الگوریتم کد هافمن

۱- **صورت مسأله:** یافتن کد دودویی بهینه برای نویسه‌های موجود در یک رشته

۲- **ورودی:** رشته‌ی مورد نظر برای کدگذاری

۳- **خروجی:** کد دودویی اختصاص داده شده به هر نویسه از رشته

```
class mycomparison {
public:
    bool operator() (const node* left, const node* right) const {
        return (left->weight>right->weight);
    }
};
priority_queue<node*, vector<node*>, mycomparison> pq;
node* huffman_coding() {
    while (pq.size() > 1) {
        node* n1 = pq.top(); pq.pop();
        node* n2 = pq.top(); pq.pop();
        node* new_node = new node();
        new_node->weight = n1->weight + n2->weight;
        new_node->left = n1;
        new_node->right = n2;
        pq.push(new_node);
    }
    node* rnode = pq.top(); pq.pop();
    return rnode;
}
```

می‌توانیم برای به دست آوردن نویسه‌ی مربوط به هر کد دودویی، تابع زیر را به کار بریم:

```
char get_symbol(string strcode, node* root) {
    for (int i = 0; i < strcode.length(); i++) {
        if (root == NULL)
            break;
        if (strcode[i] == '0')
```

```
            root = root->left;
        else
            root = root->right;
    }
    return root->symbol;
}
```

اگر نویسه‌ها و فراوانی‌های مربوط به مثال قبل را در نظر بگیریم، حاصل فراخوانی تابع get_symbol() به صورت زیر، نویسه‌ی 'b' خواهد بود:

```
node* root = huffman_coding();
cout << get_symbol("010", root);
```

پیچیدگی زمانی قرار دادن اولیه‌ی نویسه‌ها در صف اولویت‌دار، برابر با $O(n)$ است. به صورت کلی، اضافه کردن یک گره به صف اولویت‌دار (هرم)، در زمان $O(\log m)$ صورت می‌گیرد که m تعداد گره‌های درخت دودویی است. در درخت هافمن n برگ داریم، در نتیجه تعداد گره‌های درخت، حداکثر برابر با $2n-1$ خواهد بود. بنابراین پیچیدگی زمانی اضافه کردن یک گره به صف بر حسب n، برابر با $O(\log n)$ است. حلقه‌ی اصلی الگوریتم $n-1$ مرتبه اجرا می‌شود (برای n نویسه)، که در هر مرتبه، یک گره به درخت اضافه و دو گره از آن حذف می‌شوند. پیچیدگی حذف یک گره از صف اولویت‌دار، برابر با $O(1)$ است. بنابراین پیچیدگی زمانی حلقه، برابر با $O(n(1+1+\log n)) = O(n\log n)$ خواهد بود. پیچیدگی زمانی کل الگوریتم هافمن به صورت زیر محاسبه می‌شود:

$$O(n) + O(n\log n) \in O(n\log n)$$

٤-۱۰- مسأله کوله‌پشتی

مسأله کوله‌پشتی[۴۶] در دسته‌ی مسأله‌های بهینه‌سازی قرار می‌گیرد. در این دسته از مسأله‌ها، هدف بیشینه کردن سود (یا کمینه کردن ضرر) است. هر مسأله، تعریف ویژه‌ی خود را برای سود دارد. نام کوله‌پشتی به مسأله‌ی معروف پر کردن کوله‌پشتی اشاره می‌کند:

شخصی دارای یک کوله‌پشتی با ظرفیت w بوده و با مجموعه‌ای از اشیاء روبه‌رو است: هر یک از اشیاء دارای ارزش و وزن مختص به خود هستند. هدف برداشتن اشیائی است که مجموع ارزش آن‌ها بیشینه بوده و مجموع وزن آن‌ها از وزن کوله‌پشتی تجاوز نکند.

این مسأله به سه دسته‌ی کلی زیر تقسیم می‌شود. در ادامه، مسأله‌ی کوله‌پشتی کسری و صفر و یک را بررسی کرده و برای هر یک الگوریتمی ارائه خواهیم کرد.

۱- کوله‌پشتی کسری[۴۷]: در این مسائل که ساده‌ترین مسأله‌های کوله‌پشتی هستند، برگزیدن بخشی از اشیاء نیز امکان‌پذیر است. یعنی می‌توان کسری از آن‌ها را نیز برگزید.

---

[۴۶] Knapsack Problem

۲- کوله‌پشتی صحیح یا صفر و یک[۴۸]: در این مسائل، هر شیء یا باید بطور کامل برگزیده شود و یا نشود؛ امکان انتخاب کسری از اشیاء وجود ندارد.

۳- کوله‌پشتی چندگانه[۴۹]: این مسائل در واقع دسته‌ی جدیدی نبوده و ترکیبی از دو دسته‌ی پیشین هستند. در این مسأله‌ها باید چند کوله‌پشتی پر شوند. اگر امکان برگزیدن کسری از اشیاء وجود داشته باشد، می‌توان مسأله را به مسأله‌ای با یک کوله‌پشتی با مجموع ظرفیت کوله‌پشتی‌های چندگانه تبدیل کرد.

٤-۱۰-۱- کوله‌پشتی کسری

برای این دسته از مسأله‌ها، راه‌حل حریصانه به خوبی کار می‌کند. راه‌حل حریصانه در هر مرحله به صورت حریصانه عمل کرده و شیء‌ای را برمی‌گزیند که دارای بیشترین ارزش و در عین حال کمترین وزن باشد. برای ارائه‌ی الگوریتمی درست، باید هر دو عامل ارزش و وزن در نظر گرفته شوند و نادیده گرفتن یکی از این دو منجر به شکست در حل درست مسأله می‌گردد. برای برگزیدن ارزش بیشتر و وزن کمتر، نسبت ارزش به وزن اشیاء را در نظر می‌گیریم، یعنی $d = \frac{value}{weight}$. الگوریتم را به صورت مرحله به مرحله ارائه می‌کنیم:

۱- اشیاء را بر اساس مقدار d (نسبت ارزش به وزن) به صورت نزولی مرتب کرده و در یک صف قرار می‌دهیم.

۲- شیء‌ای را از آغاز صف برداشته و آن را s می‌نامیم.

۳- اگر وزن s از وزن کوله‌پشتی کمتر باشد، همه‌ی s را در کوله‌پشتی قرار داده و الگوریتم را از مرحله ۲ تکرار می‌کنیم.

۴- اگر وزن s بیشتر از وزن کوله‌پشتی باشد (یا مساوی)، بیشترین مقدار ممکن از s که کوله‌پشتی را پر می‌کند، برداشته و در کوله‌پشتی قرار می‌دهیم؛ در این حالت الگوریتم به پایان می‌رسد.

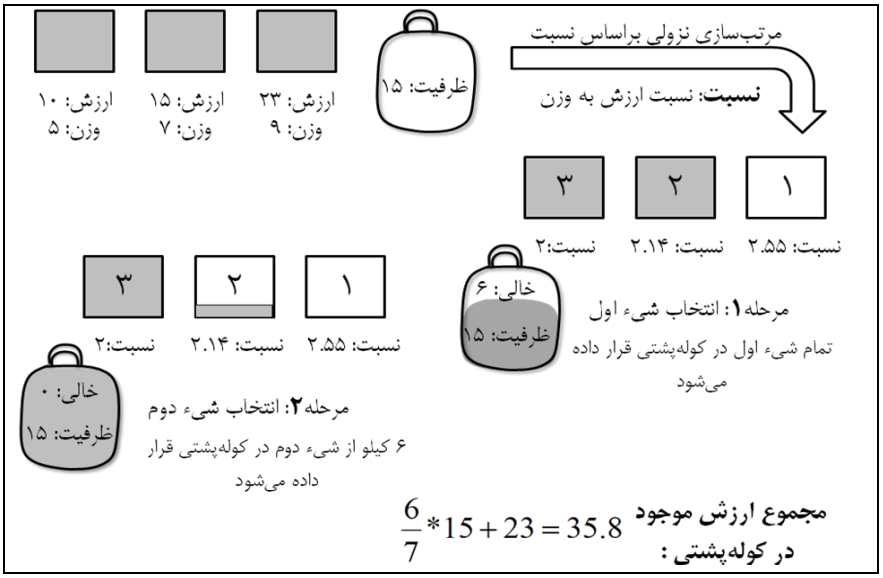
شکل ۳۹,۴ مراحل اجرای الگوریتم را برای سه شیء نمونه، با ارزش و وزن نمایش داده شده در شکل و یک کوله‌پشتی با ظرفیت ۱۵ نشان می‌دهد. پس از قرار دادن شیء اول و بخشی از شیء دوم، کوله‌پشتی پر شده و نوبت به شیء سوم نمی‌رسد. هنگام محاسبه ارزش کل اشیاء موجود در کوله‌پشتی، فقط ارزش بخشی از شیء دوم که در کوله‌پشتی قرار داده شده را محاسبه می‌کنیم.

---

[۴۷] Fractional Knapsack

[۴۸] Integer Knapsack / 0-1 knapsack

[۴۹] Multiple Knapsack



شکل ۳۹,٤. مراحل اجرای الگوریتم کوله‌پشتی برای سه شی نمونه

کد ارائه شده در ادامه‌ی این بخش، مراحل ذکر شده را پیاده‌سازی می‌کند. چگونگی به‌کارگیری تابع qsort پیش از این توضیح داده شده است. ساختار object برای توصیف اشیاء تعریف شده و دارای دو بخش ارزش و وزن است. پارامتر ورودی n تعداد اشیاء را مشخص می‌کند. ظرفیت کوله‌پشتی (w) به صورت سراسری تعریف شده است. پیچیدگی زمانی الگوریتم با بررسی دو بخش اصلی کد مشخص می‌شود.

۱- مرتب کردن مجموعه‌ی اشیاء: پیچیدگی زمانی مرتب‌سازی با روش سریع $O(n\log n)$ است.

۲- حلقه‌ی for: این حلقه در بدترین حالت همه‌ی اشیاء را بررسی کرده و در صورت برقراری شرط مورد نظر، آن‌ها در کوله‌پشتی قرار می‌دهد. پیچیدگی زمانی این بخش $O(n)$ خواهد بود.

پیچیدگی زمانی کل الگوریتم از مرتبه‌ی خطی-لگاریتمی خواهد بود.

$$O(n\log n)+O(n)\in O(n\log n)$$

الگوریتم کوله‌پشتی کسری

۱- **صورت مسأله:** پر کردن کوله‌پشتی با ظرفیت w توسط مجموعه‌ای از اشیاء (که هر یک دارای ارزش و وزن مشخص هستند)، به شکلی که مجموع ارزش اشیاء موجود در کوله‌پشتی بیشینه و مجموع وزن آن‌ها از ظرفیت کوله‌پشتی تجاوز نکند. امکان انتخاب کسری از یک شیء نیز وجود دارد.

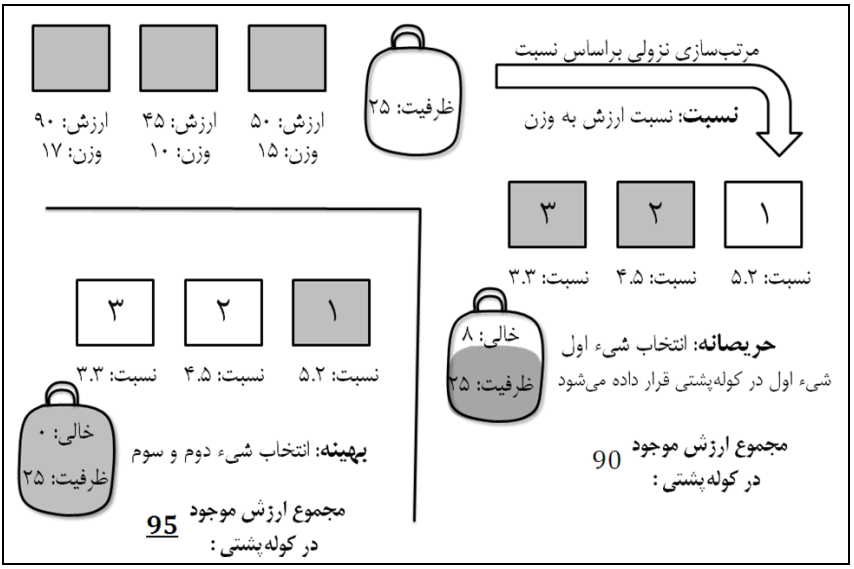
۲- **ورودی:** ظرفیت کوله‌پشتی و مجموعه‌ی ارزش و وزن هر یک از اشیاء.

۳- **خروجی:** مجموع ارزش بیشینه ممکن که می‌تواند در کوله‌پشتی قرار گیرد.

```
#define MAX_N 100
struct object {
    int value;
    int weight;
}objects[MAX_N];
int w;     // knapsack capacity
int compare_by_density(const void* a, const void* b) {
    object* obj1 = (object*)a;
    object* obj2 = (object*)b;
    return (obj2->value / obj2->weight) - (obj1->value / obj1->weight);
}
float fractional_knapsack(int n) {
    float sum_value = 0; // sum of selected objects values
    int sum_weights = 0; // sum of selected objects weights
                         // sort objects by density=value/weight in desc
order;
    qsort(objects, n, sizeof(object), compare_by_density);
    for (int i = 0; i <= n; i++) {
        if (objects[i].weight >= (w - sum_weights)) {
            float selected_part = (w - sum_weights) /
(float)objects[i].weight;
            sum_value += selected_part*objects[i].value;
            break;
        }
        else {
            sum_value += objects[i].value;
            sum_weights += objects[i].weight;
        }
    }
    return sum_value;
}
```

٤-١٠-٢- کوله‌پشتی صفر و یک

در مسأله‌ی کوله‌پشتی صفر و یک، مجاز به برگزیدن بخشی از اشیاء نیستیم و فقط دو گزینه برای هر شیء وجود دارد: انتخاب یا عدم انتخاب. راه‌حل حریصانه‌ی کوله‌پشتی کسری برای این مسأله قابل استفاده نیست. از این رو باید در جستجوی راه‌حل دیگری باشیم. شکل ۴۰,۴، شکست راه‌حل حریصانه را در حل یک نمونه مسأله کوله‌پشتی صفر و یک به روشنی نشان می‌دهد.



شکل ٤٠,٤. عدم موفقیت رویکرد حریصانه در حل مسأله کوله‌پشتی صفر و یک

روش پویا قادر به حل مسأله‌ی کوله‌پشتی صفر و یک است. اساس روش پویا، استفاده از مرحله یا مراحل پیشین برای رسیدن به مرحله‌ی بعدی است. برای این مسأله، یک قاعده‌ی پویا به صورت زیر تعریف می‌کنیم.

یک کوله‌پشتی با ظرفیت مشخص w در اختیار داریم و باید آن را از اشیاء موجود به گونه‌ای پر کنیم که مجموع ارزش آن‌ها بیشینه شود. برای شروع، فرض می‌کنیم ظرفیت کوله‌پشتی یک واحد است و فقط یک شیء برای برگزیدن وجود دارد. اگر شیء بتواند در کوله‌پشتی قرارگیرد، آن را برمی‌گزینیم. در مرحله‌ی بعد یک واحد به ظرفیت کوله‌پشتی اضافه می‌کنیم و کار مرحله‌ی قبل را تکرار می‌کنیم. این روال را تا زمانی که ظرفیت کوله‌پشتی به مقدار w برسد، ادامه می‌دهیم. سپس دوباره ظرفیت کوله‌پشتی را یک واحد فرض می‌کنیم و این بار دو شیء اول را در نظر می‌گیریم. اگر اضافه شدن شیء دوم به کوله‌پشتی پر شده در مرحله‌ی پیشین، ارزش آن را افزایش می‌دهد (و وزن آن نیز شرایط لازم را دارد)، پس آن را نیز برمی‌گزینیم، در غیر این صورت به وضعیت پیشین آن اکتفا می‌کنیم. در این مرحله نیز، ظرفیت کوله‌پشتی را تا مقدار w افزایش می‌دهیم. این روال تا زمانی که ظرفیت کوله‌پشتی به w و تعداد اشیاء درنظر گرفته شده به n (تعداد کل اشیاء ممکن) برسد، ادامه پیدا می‌کند؛ در پایان، پاسخ مسأله، ارزش بیشینه‌ی ممکنی خواهد بود که می‌توان در کوله‌پشتی قرار دارد. این توضیحات به صورت ساده در جمله‌ی زیر خلاصه می‌شود:

برای رسیدن به ارزش بیشینه نهایی، کوشش می‌کنیم کوله‌پشتی را در هر مرحله، با هر ظرفیتی از ۱ تا w و هر تعداد شیء ممکنی از ۱ تا n، دارای ارزش بیشینه نگه داریم. این امر با انجام مراحل بالا ممکن می‌شود.

حال به ارائه‌ی ساختاری برای پیاده‌سازی قاعده‌ی پویای توضیح داده شده می‌پردازیم. آرایه‌ی دوبعدی با بعدهای w و n در نظر می‌گیریم؛ در پایان $m[n][w]$ شامل ارزش بیشینه‌ی ممکن موجود در کوله‌پشتی خواهد بود. بر اساس توضیحات بالا، وزن‌های ممکن کوله‌پشتی را مقادیری مثبت و به صورت $w_1, w_2, \ldots, w_n$ فرض می‌کنیم که $w_n = w$ است. بنابراین $m[j][y]$ یعنی پر کردن کوله‌پشتی با ظرفیت y و با به‌کارگیری اشیاء ۱ تا j به

شکلی که مجموع ارزش آن‌ها بیشینه و مجموع وزن آن‌ها از مقدار y تجاوز نکند. فرض کنید ارزش بیشینه‌ی ممکن با $n-1$ شیء نخست و ظرفیت w را محاسبه کرده‌ایم و در این مرحله می‌خواهیم شیء n_ام را نیز به مجموعه‌ی اشیاء اضافه کرده و مجموع ارزش را نیز بیشینه نگه داریم؛ در این شرایط، قاعده‌ی پویا دو حالت زیر را در نظر گرفته و گزینه‌ی بهتر را برمی‌گزینند:

۱. اضافه کردن شیء n_ام به کوله‌پشتی: در این حالت ارزش، کوله‌پشتی برابر با مقدار $value_n + m[n-1][w-weight_n]$ خواهد شد. اضافه کردن شیء n_ام، به اندازه‌ی وزن این شیء($weight_n$) از ظرفیت کوله‌پشتی می‌کاهد. بنابراین از ارزش بیشینه‌ی $n-1$ شیء باقیمانده در بقیه‌ی فضای کوله‌پشتی بهره می‌گیریم ($m[n-1][w-weight_n]$).

۲. اضافه نکردن شیء n_ام به کوله‌پشتی: به کار نبردن شیء n_ام، یعنی پر کردن کوله‌پشتی از $n-1$ شیء دیگر به شکلی که مجموع ارزش آن‌ها بیشینه شود؛ این توضیح معادل است با مقدار $m[n-1][w]$.

روش پویا، ارزش کوله‌پشتی را در دو حالت بالا مقایسه کرده و مقدار بیشتر را برمی‌گزینند. اگر وزن شیء n_ام بیشتر از ظرفیت کوله‌پشتی باشد، به ناچار آن را کنار گذاشته و فقط از $n-1$ شیء باقیمانده بهره می‌گیریم. در ادامه به پیاده‌سازی الگوریتم بر اساس این توضیحات خواهیم پرداخت.

$$m[n][w] = \begin{cases} \max(m[n-1][w], value_n + m[n-1][w-weight_n] & weight_n \le w \\ m[n][w] = m[n-1][w] & weight_n > w \end{cases}$$

الگوریتم کوله‌پشتی صفر و یک

۱- **صورت مسأله:** پر کردن کوله‌پشتی با ظرفیت w توسط مجموعه‌ای از اشیاء (که هر یک دارای ارزش و وزن مشخص هستند)، به شکلی که مجموع ارزش اشیاء موجود در کوله‌پشتی بیشینه و مجموع وزن آن‌ها از ظرفیت کوله‌پشتی تجاوز نکند. برای برگزیدن اشیاء فقط دو حالت وجود دارد: انتخاب یا عدم انتخاب.

۲- **ورودی:** ظرفیت کوله‌پشتی و مجموعه‌ی ارزش و وزن هر یک از اشیاء.

۳- **خروجی:** مجموع ارزش بیشینه ممکن که می‌تواند در کوله‌پشتی قرار گیرد.

```
#define MAX_N 100
#define MAX_W 1000
struct object {
    int value;
    int weight;
}objects[MAX_N];
int n;
int w;        // knapsack capacity
int integer_knapsack(int n) {
    int m[MAX_N][MAX_W] = {0};
    for (int j = 1; j <= n; j++)
```

```
        for (int y = 1; y <= w; y++) {
            if (objects[j].weight > y)
                m[j][y] = m[j - 1][y];
            else
                m[j][y] = max(m[j - 1][y],
                    objects[j].value + m[j - 1][w - objects[j].weight]);
        }
    return m[n][w];
}
```

پیاده‌سازی ارائه شده بسیار ساده بوده و نیاز به توضیح اضافی ندارد. پیچیدگی زمانی و فضایی الگوریتم کوله‌پشتی صفر و یک برابر با $O(nw)$ است. پیچیدگی فضایی را می‌توان با به‌کارگیری یک آرایه‌ی تک‌بعدی به طول w به جای استفاده از آرایه‌ی دوبعدی، به $O(w)$ کاهش داد.

سرعت اجرای این روش برای ظرفیت‌های (w) کم، بسیار مناسب است ولی با رشد w، فاصله‌ی زیاد وزن اشیاء نسبت به ظرفیت کوله‌پشتی، موجب کند شدن الگوریتم می‌شود؛ در این حالت، پیچیدگی زمانی حتی می‌تواند به مرتبه‌ی فاکتوریلی هم برسد (w را $n!$ در نظر بگیرید). بهترین بهینه‌سازی‌های ارائه شده برای این الگوریتم، پیچیدگی زمانی آن را بهتر از نمایی نکرده‌اند. در ادامه، به یک بهینه‌سازی که می‌تواند پیچیدگی زمانی الگوریتم را در بدترین حالت به مرتبه‌ی نمایی کاهش دهد، اشاره می‌کنیم. اگر عملکرد الگوریتم را از پایان به آغاز بررسی کنید، متوجه می‌شوید که در هر مرحله، فقط چند مقدار از مرحله‌ی پیشین به کار برده می‌شود؛ بنابراین می‌توان به جای تولید همه‌ی مقدارها در هر مرحله، فقط مقادیری که در مرحله‌ی بعد به کار می‌روند را تولید کرد. البته پیاده‌سازی این روش کمی دشوارتر است.

٤-۱۱- ضرب زنجیره‌ای ماتریس‌ها

برای ضرب استاندارد دو ماتریس با ابعاد $i \times j$ و $j \times k$، به $i \times j \times k$ عمل ضرب نیاز است. برای مثال دو ماتریس زیر را در نظر بگیرید. برای ضرب هر سطر از ماتریس اول در یک ستون از ماتریس دوم به سه عمل ضرب نیاز است. در نتیجه برای ضرب دو سطر در چهار ستون، به $2 \times 4 \times 3 = 24$ نیاز خواهیم داشت.

$$\begin{bmatrix} 2 & 1 & 3 \\ 1 & 4 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 3 & 4 & 1 \\ 2 & 1 & 2 & 5 \\ 3 & 2 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 13 & 13 & 13 & 13 \\ 15 & 11 & 14 & 25 \end{bmatrix}$$

برای ضرب بیش از دو ماتریس، باید ابتدا آن‌ها را پرانتزبندی کرد و سپس عملیات ضرب را آغاز کرد. از آنجا که ضرب ماتریس‌ها شرکت‌پذیر است، ترتیب پرانتزبندی در نتیجه تاثیرگذار نیست. یعنی اگر چهار ماتریس A، B، C و D در اختیار داشته باشیم، همه‌ی ترتیب‌های زیر، نتیجه‌ی یکسانی تولید خواهند کرد.

$$A(B(CD)) = (AB)(CD) = (A(BC))D = \ldots$$

هرچند ترتیب پرانتزبندی، تاثیری در نتیجه‌ی نهایی نخواهد داشت ولی رابطه‌ی مستقیمی با تعداد ضرب‌ها دارد. از آنجا که عملیات ضرب یک عملیات زمان‌بر برای پردازنده‌ها به حساب می‌آید، هدف ما، کاهش هر چه بیشتر آن

است. برای مثال، دو ترتیب متفاوت برای ضرب سه ماتریس $A_{5\times15}$ ، $B_{15\times8}$ ، $C_{8\times14}$ را در نظر گرفته و تعداد ضرب‌های مورد نیاز برای محاسبه‌ی آن‌ها را نشان داده‌ایم. می‌بینیم که ترتیب دوم، کمتر از نصف تعداد ضرب‌های ترتیب اول را انجام می‌دهد. هر عقل سلیمی، استفاده از ترتیب دوم را به ترتیب اول ترجیح می‌دهد.

$$A_{5\times15}\times(B_{15\times8}\times C_{8\times14}) = (15\times8\times14)+(5\times15\times14) = 1680+1050 = 2730$$
$$(A_{5\times15}\times B_{15\times8})\times C_{8\times14} = (5\times15\times8)+(5\times8\times14) = 600+560 = 1160$$

تا اینجا، هدف را تعیین کرده‌ایم که یافتن ترتیب بهینه‌ی پرانتزبندی است. برای این منظور می‌توانیم همه‌ی ترتیب های ممکن را بررسی کنیم؛ ولی این روش از مرتبه‌ی نمایی نسبت به تعداد ماتریس‌ها (n) است که برای n_های بزرگ بسیار زمان‌بر است. یک راه‌حل مناسب، شکستن مسأله به تعدادی زیرمسأله و حل آن‌ها و سپس استفاده‌ی مجدد از زیرمسأله‌ها است. این راه‌حل همان برنامه‌نویسی پویا است. فرض کنید ترتیب $A(B(CD))$ ، یک ترتیب بهینه برای ضرب ماتریس‌های A ، B ، C و D باشد. در این حالت، ترتیب $B(CD)$ نیز مطمئنا یک ترتیب بهینه برای ضرب ماتریس‌های B ، C و D است. بنابراین حل زیرمسأله‌ها، قابل استفاده‌ی مجدد هستند.

برای ارائه‌ی راه‌حل پویا از یک ماتریس دو بعدی بالا مثلثی به صورت زیر استفاده می‌کنیم. هدف یافتن ترتیب بهینه‌ی ضرب ماتریس‌های $A_1\times A_2\times\ldots\times A_n$ است. سطر i_ام و ستون j _ام از ماتریس M ، دربردارنده‌ی حداقل تعداد ضرب‌های مورد نیاز برای ضرب‌های ماتریس‌های A_i تا A_j است.

$$M[i][j] = \min \ \ number\ of\ \ products\ to\ multiply\ A_i\times\ldots\times A_j \qquad i<j$$

پنج ماتریس A_1 تا A_5 را در نظر بگیرید. ترتیب بهینه‌ی کلی یکی از حالت‌های زیر خواهد بود. فرض می‌کنیم ترتیب اعمال شده در داخل هر پرانتز بهینه باشد.

$$A_1(A_2A_3A_4A_5)$$
$$(A_1A_2)(A_3A_4A_5)$$
$$(A_1A_2A_3)(A_4A_5)$$
$$(A_1A_2A_3A_4)A_5$$

تعداد ضرب‌های هر حالت را در ادامه نشان داده‌ایم. A_{25} یعنی حاصلضرب ماتریس‌های $A_2\times A_5$ ؛ و $A_1\times A_{25}$ نشان‌دهنده‌ی تعداد ضرب‌های مورد نیاز برای محاسبه‌ی حاصلضرب دو ماتریس است. از بین این پنج حالت، حالتی که منجر به کمترین تعداد ضرب‌ها می‌شود، ترتیب بهینه است.

$$M[1][1]+M[2][5]+A_1\times A_{25}$$
$$M[1][2]+M[3][5]+A_{12}\times A_{35}$$
$$M[1][3]+M[4][5]+A_{13}\times A_{45}$$
$$M[1][4]+M[5][5]+A_{14}\times A_5$$

بنابراین می‌توانیم فرمول زیر را برای بدست آوردن تعداد ضرب‌های کمینه ماتریس‌های اول تا پنجم ارائه دهیم. حال اگر از کوچکترین حالت‌ها شروع به حل مسأله کنیم، به سادگی می‌توانیم به پاسخ بهینه دست یابیم.

$$M[1][5] = \min(M[1][k] + M[k+1][5] + A_{1k} \times A_{(k+1)5} \quad k = 1,2,...,5$$

در شکل ۴۱,۴ روش پویا را مرحله به مرحله بر روی پنج ماتریس نمونه اعمال کرده‌ایم. ابتدا حالت‌های تک ماتریسه، سپس دو ماتریسه و در انتها پنج ماتریسه را حل می‌کنیم. از راه‌حل حالت‌های قبلی در حالت‌های بعدی بهره می‌بریم.

$$A^1_{5\times10} \times A^2_{10\times4} \times A^3_{4\times8} \times A^4_{8\times6} \times A^5_{6\times10}$$

مرحله اول

$$\begin{cases} M[1][2] = M[1][1] + M[2][2] + 5 \times 10 \times 4 = 200 \\ M[2][3] = M[2][2] + M[3][3] + 10 \times 4 \times 8 = 320 \\ M[3][4] = M[3][3] + M[4][4] + 4 \times 8 \times 6 = 192 \\ M[4][5] = M[4][4] + M[5][5] + 8 \times 6 \times 10 = 480 \end{cases}$$

مرحله دوم

$$\begin{cases} M[1][3] = \min \begin{cases} M[1][1] + M[2][3] + 5 \times 10 \times 8 = 720 \\ M[1][2] + M[3][3] + 5 \times 4 \times 8 = 360 \end{cases} \\ M[2][4] = \min \begin{cases} M[2][2] + M[3][4] + 10 \times 4 \times 6 = 432 \\ M[2][3] + M[4][4] + 10 \times 8 \times 6 = 800 \end{cases} \\ M[3][5] = \min \begin{cases} M[3][3] + M[4][5] + 4 \times 8 \times 10 = 800 \\ M[3][4] + M[5][5] + 4 \times 6 \times 10 = 432 \end{cases} \end{cases}$$

مرحله سوم

$$\begin{cases} M[1][4] = \min \begin{cases} M[1][1] + M[2][4] + 5 \times 10 \times 6 = 732 \\ M[1][2] + M[3][4] + 5 \times 4 \times 6 = 512 \\ M[1][3] + M[4][4] + 5 \times 6 \times 10 = 660 \end{cases} \\ M[2][5] = \min \begin{cases} M[2][2] + M[3][5] + 10 \times 4 \times 10 = 832 \\ M[2][3] + M[4][5] + 10 \times 8 \times 10 = 1600 \\ M[2][4] + M[5][5] + 10 \times 6 \times 10 = 1032 \end{cases} \end{cases}$$

مرحله آخر

$$M[1][5] = min \begin{cases} M[1][1] + M[2][5] + 5 \times 10 \times 10 = 1332 \\ M[1][2] + M[3][5] + 5 \times 4 \times 10 = 832 \\ M[1][3] + M[4][5] + 5 \times 8 \times 10 = 1240 \\ M[1][4] + M[5][5] + 5 \times 6 \times 10 = 812 \end{cases}$$

M

	1	2	3	4	5
1	0	200	360	512	812
2	0	0	320	432	832
3	0	0	0	192	432
4	0	0	0	0	480
5	0	0	0	0	0

مرحله اول — مرحله دوم — مرحله سوم — پاسخ نهایی

شکل ٤١,٤. محاسبه کمترین تعداد ضرب مورد نیاز برای پنج ماتریس نمونه

در ادامه شبه کد نحوه‌ی پیاده‌سازی روش پویا برای این مسأله آمده است.

الگوریتم ضرب زنجیره‌ای ماتریس‌ها

۱- **صورت مسأله:** دریافت n ماتریس و محاسبه‌ی کمترین تعداد ضرب مورد نیاز (و پرانتزبندی بهینه‌ای که منجر به این کمترین می‌شود) برای محاسبه حاصلضرب آن‌ها.

۲- **ورودی:** n ماتریس به همراه ابعاد هر یک

۳- **خروجی:** پرانتزبندی بهینه ماتریس‌ها و کمترین تعداد ضرب مورد نیاز برای محاسبه‌ی حاصلضرب آن‌ها

```
chainorder(int p[]) {
    int n = p.length - 1
    for (int i = 1; i <= n; i++)
        M[i][i] = 0
    for (int chainlen = 2; chainlen <= n; chainlen++)
        for (int i = 1; i <= n - chainlen + 1; i++) {
            int j = i + chainlen - 1// chain end
            M[i][j] = ∞
            for (int k = i; k <= j - 1; k++) {
                int cost = M[i][k] + M[k + 1][j] + p[i - 1] * p[k] * p[j]
                if (cost<M[i][j]) {
                    M[i][j] = cost       // new cost
                    S[i][j] = k          // save chain order
                }
            }
        }
}
```

ورودی شبه‌کد ارائه شده، آرایه‌ی p است. این آرایه ابعاد ماتریس‌ها را در خود دارد. برای مثال، محتویات این آرایه برای ماتریس‌های شکل ۴۱,۴ در ادامه نمایش داده شده است. ابعاد ماتریس اول در نمایه‌های صفر و یک، ابعاد ماتریس دوم در نمایه‌های یک و دو و به همین ترتیب، ابعاد ماتریس i_ام در نمایه‌های $i-1$ و i قرار دارد. تعداد ماتریس‌ها یکی کمتر از طول این آرایه است. در شبه‌کد، علاوه‌بر آرایه دوبعدی M از آرایه‌ی دوبعدی S نیز استفاده کرده‌ایم. مقدار خانه‌ی S[i][j]، نقطه‌ای را مشخص می‌کند که برای ضرب بهینه‌ی ماتریس‌های i تا j باید در آن نقطه پرانتز قرار داد. روش استفاده از این آرایه را بعد از ارائه‌ی کد به زبان سی++، توضیح خواهیم داد.

p	5	10	4	8	6	10

```
#include<iostream>
using namespace std;
const int MAXN = 1000;
int M[MAXN][MAXN], S[MAXN][MAXN];
int chainorder(int p[], int n) {
    for (int i = 1; i <= n; i++)
        M[i][i] = 0;
    for (int chainlen = 2; chainlen <= n; chainlen++)
        for (int i = 1; i <= n - chainlen + 1; i++) {
            int j = i + chainlen - 1;
```

```
            M[i][j] = INT_MAX;
            for (int k = i; k <= j - 1; k++) {
                int cost = M[i][k] + M[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (cost<M[i][j]) {
                    M[i][j] = cost;      // new cost
                    S[i][j] = k;     // save chain order
                }
            }
        }
    return M[1][n];
}
int main() {
    int p[] = {5, 10, 4, 8, 6, 10};
    int n = (sizeof(p) / sizeof(int)) - 1;
    cout << chainorder(p, n) << endl;
    return 0;
}
```

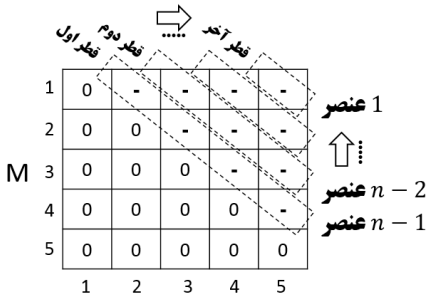
کد ارائه شده شبیه به شبه‌کد است و نیاز به توضیح اضافی ندارد. برای بدست آوردن ترتیب پرانتزبندی، از آرایه S به صورت زیر بهره می‌بریم. تابع بازگشتی printchainorder، نمایه‌ی ماتریس اول و آخر را دریافت کرده و ترتیب بهینه‌ی پرانتزبندی آن‌ها را در خروجی چاپ می‌کند.

```
void printchainorder(int i, int j) {
    if (i == j) {
        cout << i;
        return;
    }
    cout << "(";
    printchainorder(i, S[i][j]);
    printchainorder(S[i][j] + 1, j);
    cout << ") ";
}
```

برای مثال، اگر تابع بالا را برای کد قبل و به صورت printchainorder(1,5) فراخوانی نماییم، خروجی زیر چاپ خواهد شد.

Output
(((12) (34)) 5)

برای محاسبه پیچیدگی زمانی الگوریتم برای n ماتریس، از ماتریس M با ابعاد $n \times n$ استفاده می‌کنیم. همان‌طور که در شکل ۴۲,۴ نشان داده شده است، تعداد $n-1$ خانه در اولین قطر غیرصفر وجود دارد. $n-2$ در دومین قطر و به همین ترتیب تا آخرین قطر که دارای 1 خانه بوده و پاسخ نهایی مسأله را در خود دارد.



شکل ٤٢,٤. تعداد عناصر هر خانه از ماتریس M

در شکل ۴۱,۴ دیدیم که برای محاسبه هر یک از خانه‌های موجود در قطر اول، به تنها یک مرحله نیاز است. برای محاسبه‌ی خانه‌های موجود در قطر دوم به دو مرحله و برای خانه‌های موجود در قطر آخر به $n-1$ مرحله نیاز است. تعداد مراحل مورد نیاز برای همه قطرها را به صورت رابطه‌ی زیر می‌نویسیم.

$$(n-1)*1+(n-2)*2+...+1*(n-1)=\sum_{i=1}^{n-1}(n-i)*i$$

در صورتی که بتوانیم این رابطه را ساده کنیم، به پیچیدگی زمانی این الگوریتم دست پیدا خواهیم کرد. رابطه‌ی بالا را به صورت زیر ساده می‌کنیم. پیچیدگی زمانی این الگوریتم از مرتبه $O(n^3)$ می‌باشد که بسیار بهتر از مرتبه‌ی نمایی اولیه است.

$$\sum_{i=1}^{n-1}(n-i)i=\sum_{i=1}^{n-1}ni-i^2=n\sum_{i=1}^{n-1}i-\sum_{i=1}^{n-1}i^2$$

$$=n(\frac{n(n-1)}{2})-(\frac{n(n-1)(2n-1)}{6})=\frac{n(n-1)(n+1)}{6}\in O(n^3)$$

٤-۱۲- زمان‌بندی فعالیت‌ها

زمان‌بندی در دسته‌ی مسائل بهینه‌سازی ریاضی قرار می‌گیرد. در این بخش دو دسته از مسائل زمان‌بندی را بررسی خواهیم کرد:

- مسئله زمان‌بندی بیشترین تعداد فعالیت‌های نامتداخل
- مسئله زمان‌بندی فعالیت‌ها با سود و مهلت معین

هر دو دسته از مسائل نامبرده با استفاده از روش حریصانه حل می‌شوند که راه‌حل آن‌ها را در ادامه‌ی بخش خواهیم دید. در صورتی یک مسئله به روش حریصانه قابل حل است که ویژگی زیر برای آن برقرار باشد:
در هر مرحله، انتخابی که در آن زمان بهترین است را برگزینیم؛ و با این وجود به راه‌حل بهینه مسئله برسیم.

٤-۱۲-۱- زمان‌بندی بیشترین تعداد فعالیت‌های نامتداخل

فرض کنید تعدادی کلاس درس وجود دارد که هر کدام دارای یک زمان شروع و یک زمان پایان هستند. برای برگزاری کلاس‌های درس تنها یک کلاس خالی وجود دارد. هدف انتخاب بیشترین تعداد کلاس‌های درس نامتداخل است که قابل برگزاری در اتاق خالی موجود باشند.

در این مسأله، به صورت کلی تعدادی فعالیت با زمان شروع (s_i) و پایان (f_i) مثبت وجود دارند ($s_i \leq f_i$). الگوریتمی حریصانه ارائه خواهیم کرد که بیشترین تعداد فعالیت‌های نامتداخل را برگزینند. برای این منظور، ابتدا فعالیت‌ها را بر اساس زمان پایانشان به صورت صعودی مرتب می‌کنیم. سپس از ابتدای فهرست، فعالیت‌های نامتداخل را انتخاب می‌کنیم. شبه‌کد زیر، روش پیشنهادی را نشان می‌دهد:

```
sort activities by finishing time(f)
S = {1}
lastf = f[1]
for i = 2 to n {
   if s[i] >= lastf {
      S = S U i
      lastf = f[i]
   }
}
```

در شبه‌کد بالا، S مجموعه فعالیت‌های انتخاب شده‌ی نامتداخل است. lastf نیز زمان پایان آخرین فعالیت انتخاب شده می‌باشد. به مثالی که در ادامه آمده است، توجه کنید.

پنج فعالیت نشان داده شده در جدول ۱۶٫۴ را در نظر بگیرید. الگوریتم ارائه شده را به آن اعمال کرده و حداکثر تعداد فعالیت ممکن را بدست می‌آوریم.

جدول ١٦,٤. نمونه‌ای از چند فعالیت به همراه زمان شروع و زمان پایان

زمان شروع	زمان پایان
۵	۶
۴	۶
۱	۳
۳	۵
۱	۴

ابتدا فعالیت‌ها را بر اساس زمان پایانشان مرتب می‌کنیم. سپس از آغاز جدول شروع کرده و فعالیت‌های نامتداخل را انتخاب می‌کنیم (فعالیت‌های اول، سوم و چهارم). این فعالیت‌ها در جدول ۱۷٫۴ با رنگ تیره‌تر مشخص شده‌اند. ملاحظه می‌شود که حداکثر تعداد فعالیت‌های نامتداخل برابر با سه است. ترتیب‌های دیگری نیز می‌توان بدست آورد

ولی مطمئنا تعداد آن‌ها نیز از سه عدد تجاوز نخواهد کرد (مثال: فعالیت‌های اول، سوم و پنجم). در ادامه راه‌حل پیاده سازی شده به زبان سی++ ارائه گردیده است.

جدول ۱۷,٤. فعالیت‌های نامتداخل جدول ۱٦,٤

زمان شروع	زمان پایان
۱	۳
۱	۴
۳	۵
۵	۶
۴	۶

الگوریتم زمان‌بندی بیشترین تعداد فعالیت‌های نامتداخل

۴- **صورت مسأله:** تعدادی فعالیت در اختیار داریم که هر یک دارای زمان شروع ($start$) و پایان (end) می‌باشند. هدف، یافتن بیشترین تعداد فعالیت‌های نامتداخل است.

۵- **ورودی:** زمان شروع و پایان فعالیت‌ها.

۶- **خروجی:** بیشترین تعداد فعالیت‌های نامتداخل

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
#define MAX_N 100
struct job {
    int start;
    int end;
}jobs[MAX_N];
int n;            // number of activities
vector<int> S;        // selected activities
int compare(const void* a, const void* b) {
    return ((job*)a)->end - ((job*)b)->end;
}
int main() {
    qsort(jobs, n, sizeof(job), compare);
    S.push_back(0);
    int lastf = jobs[0].end;
    for (int i = 1; i < n; i++) {
        if (jobs[i].start >= lastf) {
            S.push_back(i);
            lastf = jobs[i].end;
        }
    }
    return 0;
}
```

در کد ارائه شده، ابتدا فعالیت‌ها بر اساس زمان پایانشان به صورت صعودی مرتب می‌شوند. سپس اولین فعالیت انتخاب شده و نمایه‌ی آن در مجموعه‌ی S قرار داده می‌شود. متغیر lastf زمان پایان آخرین فعالیت انتخاب شده را نگه می‌دارد. حلقه for سایر فعالیت‌ها را به ترتیب زمان پایانشان پیمایش کرده و آن‌هایی که زمان شروع‌شان بزرگتر یا مساوی lastf باشد را به مجموعه S اضافه خواهد کرد. در پایان، مجموعه‌ی S شامل نمایه‌ی فعالیت‌های انتخاب شده خواهد بود.

پیچیدگی زمانی راه‌حل ارائه شده برابر خواهد بود با $O(n\log n + n) \in O(n\log n)$:

- مرتب کردن فعالیت‌ها توسط بهترین روش‌های مرتب‌سازی از مرتبه $O(n\log n)$ خواهد بود.
- پیمایش فعالیت‌ها توسط یک حلقه‌ی ساده نیز از مرتبه $O(n)$ می‌باشد.

٤-۱۲-۲- زمان‌بندی فعالیت‌ها با سود و مهلت معین

این دسته از مسائل زمان‌بندی دشوارتر از دسته‌ی پیشین است. در این مسأله، طول همه فعالیت‌ها برابر با یک واحد زمانی است. هر فعالیت دارای یک سود و مهلت مشخص است و در صورتی که فعالیت مربوطه قبل از مهلت تعیین شده انجام شود، سودش به ما تعلق می‌گیرد. هدف، دست یافتن به حداکثر سود ممکن است. هیچ اجباری در انجام همه‌ی فعالیت‌ها وجود ندارد. در صورتی که مهلت یک فعالیت به پایان برسد، انجام گرفتن یا نگرفتن آن فعالیت، تفاوتی نخواهد داشت. در نتیجه آن را نادیده می‌گیریم.

به عنوان نمونه، مجموعه‌ی فعالیت‌های زیر به همراه سود و مهلت هر یک را در نظر بگیرید. فعالیت اول دارای مهلت ۲ است؛ اگر این فعالیت در واحد زمانی ۱ یا ۲ انجام شود، سود ۲۰ را به همراه خواهد داشت. به صورت مشابه، اگر فعالیت سوم در واحد زمانی ۱، ۲ و یا ۳ انجام شود، سود ۳۰ را به همراه خواهد داشت. در صورتی که دو فعالیت دارای مهلت یکسان باشند، انتخاب موردی که دارای سود بیشتر است، منطقی به نظر می‌رسد. هدف یافتن ترتیبی از فعالیت‌ها است که انجام آن‌ها، ما را به سود بیشینه می‌رساند. بر مثال ترتیب $\{1,2\}$ یعنی انجام فعالیت اول در واحد زمانی ۱ و انجام فعالیت دوم در واحد زمانی ۲ که سود ۵۰ را به همراه خواهد داشت. ترتیب $\{3,4,5\}$ یک ترتیب غیرممکن است زیرا انجام هر دو فعالیت چهارم و پنجم در واحد زمانی ۱ امکان‌پذیر نیست.

راه‌حل حریصانه‌ی این مسأله بر اساس سود عمل می‌کند. ابتدا همه‌ی فعالیت‌ها بر اساس سود (به صورت نزولی) مرتب می‌شوند. سپس از ابتدای فهرست مرتب شده، فعالیت‌ها را پیمایش کرده و آن‌ها را به مجموعه S اضافه می‌کنیم. اگر اضافه شدن فعالیت i_ام موجب غیرممکن شدن ترتیب موجود در S شود، آن را از مجموعه حذف می‌کنیم. در ادامه، این روال را برای فعالیت‌های نشان داده شده در جدول ۱۸,۴ پیاده خواهیم کرد.

جدول ۱۸,٤. نمونه‌ای از چند فعالیت به همراه سود و مهلت معین

مهلت	سود
۲	۲۰
۳	۳۰
۲	۲۵
۱	۱۵
۱	۱۹

ابتدا فعالیت‌ها را بر اساس سود و به صورت نزولی مرتب می‌کنیم. فعالیت اول به مجموعه S اضافه می‌شود. برای بررسی ممکن بودن ترتیب موجود در S، باید به ازای همه فعالیت‌های موجود در آن، قاعده زیر را بررسی کنیم.

- فعالیت i_ام در مجموعه S باید در مهلت i قابل انجام باشد. به بیانی دیگر:

$$if\ s[i] = a\ \ then\ \ a.deadline \geq i$$

با اضافه شدن فعالیت‌های دوم و سوم، باز هم ترتیب موجود در S ممکن خواهد بود. ولی اضافه شدن فعالیت‌های چهارم یا پنجم، آن را غیرممکن خواهد کرد؛ در نتیجه آن‌ها را از مجموعه حذف می‌کنیم. در پایان، فعالیت‌های اول، دوم و سوم انتخاب خواهند شد. ترتیب ممکن از آن‌ها، انجام فعالیت دوم در واحد زمانی ۱، فعالیت سوم در واحد زمانی ۲ و فعالیت اول در واحد زمانی ۳ است.

جدول ۱۹,٤. اطلاعات فعالیت‌های جدول ۱۸,٤ پس از مرتب‌سازی بر اساس سود به صورت نزولی

مهلت	سود
۳	۳۰
۲	۲۵
۲	۲۰
۱	۱۹
۱	۱۵

$$S = \{\}$$
$$S = \{1\} : possible$$
$$S = \{1,2\} : possible$$
$$S = \{1,2,3\} : possible \rightarrow \{2,3,1\}$$
$$S = \{1,2,3,4\} : impossible$$
$$S = \{1,2,3,5\} : impossible$$
$$final\, S = \{1,2,3\}$$

در ادامه شبه کد روش پیشنهادی آمده است. متغیر maxprofit در پایان این الگوریتم، سود بیشینه را در خود خواهد داشت. بررسی ممکن بودن ترتیب موجود در مجموعه S از بخش‌های کلیدی این مسأله است. در راه‌حل ارائه شده به زبان سی++، از روشی بهینه برای این منظور استفاده کرده‌ایم.

```
sort activities by profits in nonincreasing order
S = {}
maxprofit = 0
for i = 1 to n {
    S = S U activities[i]
    maxprofit = maxprofit + activities[i].profit
    if S is not feasible with this new ativity {
        remove activities[i] from S
        maxprofit = maxprofit - activities[i].profit
    }
}
```

الگوریتم زمان‌بندی فعالیت‌ها با سود و مهلت معین

۱- **صورت مسأله:** تعدادی فعالیت در اختیار داریم که هر یک دارای سود (*profit*) و مهلت (*deadline*) مشخص هستند. هدف، انتخاب فعالیت‌ها به گونه‌ای است که بیشترین سود را به همراه داشته باشد.

۲- **ورودی:** سود و مهلت هر یک از فعالیت‌ها.

۳- **خروجی:** حداکثر سود قابل کسب.

در کد بعدی، n تعداد فعالیت‌ها و S مجموعه‌ی فعالیت‌های انتخاب شده است. تابع compare برای مرتب‌سازی فعالیت‌ها به صورت غیرنزولی و بر اساس سود به کار رفته است. تابع addjob یک فعالیت را به گونه‌ای به مجموعه‌ی فعالیت‌ها اضافه می‌کند که بر اساس مهلت‌شان و به صورت صعودی در حالت مرتب قرار گیرند (مانند روش مرتب‌سازی درجی). مرتب بودن این مجموعه، بررسی ممکن بودن آن را با یک پیمایش میسر می‌سازد (تابع feasible).

```
#include<iostream>
#include<vector>
using namespace std;
#define MAX_N 100
struct job {
    int profit;
    int deadline;
}jobs[MAX_N];
int n;
vector<job> S;
int compare(const void* a, const void* b) {
    return ((job*)b)->profit - ((job*)a)->profit;
}
int addjob(int index) {
    vector<job>::iterator it = S.begin();
    for (int i = 0; i < S.size(); i++)
        if (jobs[index].deadline < S[i].deadline) {
            advance(it, i);
            S.insert(it, jobs[index]);
            return i;
        }
    S.push_back(jobs[index]);
    return S.size() - 1;
}
bool feasible() {
    for (int i = 0; i < S.size(); i++)
        if (S[i].deadline < i + 1)
            return false;
    return true;
}
```

```
int main() {
    qsort(jobs, n, sizeof(job), compare);
    int maxprofit = 0;
    for (int i = 0; i < n; i++) {
        int index = addjob(i);
        maxprofit += jobs[i].profit;
        if (!feasible()) {
            vector<job>::iterator it = S.begin();
            advance(it, index);
            S.erase(it);
            maxprofit -= jobs[i].profit;
        }
    }
    return 0;
}
```

تابع feasible با فرض مرتب بودن مجموعه S و استفاده از قاعده زیر، ممکن بودن آن را بررسی می‌کند. در صورتی که ترتیب موجود در S ممکن باشد، مقدار true و در غیراین صورت مقدار false را برمی‌گرداند.

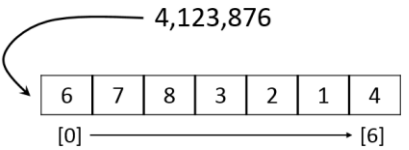
$$if\ s[i]=a\ \ then\ \ a.deadline \geq i$$

پیچیدگی زمانی هر یک از بخش‌های الگوریتم ارائه شده در ادامه توضیح داده شده‌اند. بر این اساس، پیچیدگی زمانی کل الگوریتم برابر خواهد بود با $O(n\log n+n(n+n+1)) \in O(n^2)$:

- مرتب‌سازی فعالیت‌ها بر اساس سود از مرتبه $O(n\log n)$ می‌باشد.
- حلقه‌ی for که به تعداد n مرتبه تکرار می‌شود، دارای بخش‌های زیر است.
  - اضافه کردن یک فعالیت به مجموعه S در بدترین حالت از مرتبه $O(n)$ خواهد بود.
  - بررسی ممکن بودن ترتیب موجود در S نیز در بدترین حالت از مرتبه $O(n)$ می‌باشد.
  - حذف فعالیت از مجموعه S در زمان ثابت انجام می‌شود ($O(1)$).

٤-۱۳- ضرب اعداد بزرگ

اعداد صحیح بزرگ معمول با استفاده از آرایه‌ها پیاده‌سازی می‌شود (شکل ۴۳,۴). پیاده‌سازی اعمالی همچون جمع، تفریق، ضرب و تقسیم بر روی اعداد صحیح بزرگ معمول است. جمع و تفریق اعداد صحیح بزرگ به طول n در زمان $O(n)$ قابل پیاده‌سازی است. الگوریتم‌های استاندارد برای ضرب و تقسیم اعداد صحیح بزرگ از مرتبه $O(n^2)$ هستند. در این بخش تنها در رابطه با ضرب این اعداد بحث خواهیم کرد و سایر عملیات‌های مشابه را به خواننده واگذار می‌کنیم. یکی از روش‌های ضرب اعداد بزرگ، روش تقسیم و غلبه است که ابتدا اعداد را به دو قسمت شکسته و سپس آن‌ها را در هم ضرب می‌کند. این الگوریتم را می‌توان طوری تغییر داد که پیچیدگی زمانی آن بهتر از درجه‌ی دوم شود. عدد x را با n رقم در نظر بگیرید. این عدد را می‌توان به شکل زیر نوشت. این شکل برای هر عددی با طول $m \leq n$ نیز صادق است.



شکل ٤٣,٤. نمایش عدد صحیح در آرایه

$$x = x_1 \times 10^m + x_2 \qquad m = \left\lfloor \frac{n}{2} \right\rfloor$$

بخش x_1 دارای $\left\lceil \frac{n}{2} \right\rceil$ رقم و بخش x_2 دارای $\left\lfloor \frac{n}{2} \right\rfloor$ رقم است. برای مثال، عدد ۱۲۳۴۵۶ را می‌توان بـه صـورت زیر نوشت:

$$123,456 = 123 \times 10^3 + 456$$

برای ضرب دو عدد x و y می‌توان به صورت زیر عمل کرد:

$$x = x_1 \times 10^m + x_2$$
$$y = y_1 \times 10^m + y_2$$
$$x \times y = (x_1 \times 10^m + x_2) \times (y_1 \times 10^m + y_2) = x_1 y_1 \times 10^{2m} + (x_1 y_2 + x_2 y_1) \times 10^m + x_2 y_2$$

در روش ارائه شده، تقسیم مسأله به بخش‌های کوچکتر انقدر ادامه می‌یابد تا طول اعداد به اندازه‌ای برسد که بتوان آن‌ها را با ابزارهای موجود در زبان ضرب کرد. برای دو نیم کردن یک عدد بزرگ به دو عملگر زیر نیاز است که هـر دوی آن‌ها در زمان خطی قابل انجامند.

- تقسیم عدد بزرگ بر 10^m
- باقیمانده عدد بزرگ بر 10^m

برای ادغام نتایج به دست آمده به دو عملیات جمع و ضرب نیاز داریم که آن‌ها نیز در زمان خطی قابل پیـاده‌سـازی هستند.

- جمع دو عدد بزرگ
- ضرب عدد بزرگ در 10^m

بر اساس توضیحات داده شده، اگر n را توانی از ۲ فرض کرده و سایر اعمال (که همگی از مرتبه‌ی خطی هستند) را به صورت cn نشان دهیم، شکل بازگشتی تابع ارائه شده به صورت زیر است؛ بـر اسـاس قضـیه مسـتر، پیچیـدگی زمانی الگوریتم برابر با $O(n^{\log 4}) = O(n^2)$ خواهد بود. برای سایر مقادیر n نیز به همین پیچیدگی زمـانی مـی رسیم. شبه‌کد زیر را برای این الگوریتم ارائه کرده‌ایم. به دلیل بزرگی حجم کد، از آوردن پیاده‌سازی واقعی صـرفنظر شده است.

$$T(n) = 4T(\frac{n}{2}) + cn$$

الگوریتم ضرب اعداد صحیح بزرگ

۱- **صورت مسأله:** دریافت دو عدد صحیح بزرگ و محاسبه‌ی حاصلضرب آن‌ها

۲- **ورودی:** دو عدد صحیح بزرگ x و y

۳- **خروجی:** حاصلضرب $x \times y$

```
bignumber mul(bignumber x, bignumber y) {
    int n = max(number of digits of x, number of digits of y)
    if (x == 0 or y == 0)
        return 0
    else if (n is smaller than a certain threshold)
        return x * y
    else {
        int m = floor(n / 2)
        bignumber x1 = div(x, m), x2 = rem(x, m)
        bignumber y1 = div(y, m), y2 = rem(y, m)
        return mul10(mul(x1, y1), 2 * m) + mul10(mul(x1, y2) + mul(x2,
y1), m) + mul(x2, y2);
    }
}
```

در شبه‌کد بالا، عملگر * برای ضرب معمولی دو عدد کوچک به کار رفته است. برای ضرب اعداد بزرگ از تابع mul استفاده می‌شود. سه تابع mul10، div و rem برای محاسبه ضرب، تقسیم و باقیمانده اعداد صحیح بزرگ در/بر 10^m به کار گرفته شده‌اند. برای مثال، فراخوانی div(x, 5) حاصل تقسیم $n \div 10^5$ را محاسبه می‌نماید. عملگر + نیز برای جمع دو عدد صحیح بزرگ به کار می‌رود.

در الگوریتم ارائه شده، چهار ضرب و تعدادی عملیات از مرتبه‌ی خطی انجام می‌شود. هر چهار ضرب بر روی اعدادی با طول $\frac{n}{2}$ صورت می‌گیرد. آقای کاراتسوبا[۵۰] الگوریتمی ارائه کرده که با اعمال تغییراتی، تعداد ضرب‌ها را یکی کاهش داده و به سایر عملیات‌ها افزوده است. فرمول الگوریتم قبل را به شکلی جدید بازنویسی کرده‌ایم:

$$x \times y = z_2 \times 10^{2m} + z_1 \times 10^m + z_0$$

$$z_2 = x_1 y_1$$

$$z_1 = x_1 y_2 + x_2 y_1$$

$$z_0 = x_2 y_2$$

[۵۰] Karatsuba

برای کاهش تعداد ضرب‌ها، z_1 را به روشی دیگر محاسبه می‌کنیم. با اضافه شدن چهار عمل جمع و تفریق، یک عمل ضرب کاهش پیدا کرده است. با این ایده، تنها خط آخر از شبه‌کد ارائه شده، تغییر خواهد کرد که در ادامه نشان داده شده است. عملگر – دو عدد بزرگ را از هم کم می‌کند.

$$z_1 = (x_1 + x_2)(y_1 + y_2) - z_2 - z_0$$

```
bignumber mul(bignumber x, bignumber y) {
    …
    else {
        …
        bignumber z2 = mul(x1, y1)
        bignumber z0 = mul(x2, y2)
        bignumber z1 = mul(x2 + x1, y2 + y1) - z2 - z1

        return mul10(z2, 2 * m) + mul10(z1, m) + z0;
    }
}
```

برای تحلیل پیچیدگی زمانی این الگوریتم، اگر n را توانی از دو فرض کنیم، رابطه‌ی بازگشتی زیر را خواهیم داشت. بر اساس قضیه مستر، پیچیدگی زمانی برابر است با $O(n^{\log 3}) \approx O(n^{1.585})$. به ازای سایر مقادیر n نیز، مرتبه‌ی زمانی مشابهی خواهیم داشت.

$$T(n) = 3T(\frac{n}{2}) + cn$$

٤-١٤- تطبیق الگو

الگوریتم‌های تطبیق الگو یا جستجوی رشته، وجود یک رشته (الگو) را در رشته‌ی دیگری بررسی کرده و نتیجه را اعلام می‌کنند. در این بخش دو روش تطبیق الگو را بررسی خواهیم کرد. یکی از روش‌ها به شکل معمولی به یافتن یک تطبیق در رشته می‌پردازد و از پیچیدگی زمانی مناسبی برخوردار نیست و دیگری روشی بهینه را برای این منظور به کار می‌برد.

الگوریتم‌های تطبیق الگو

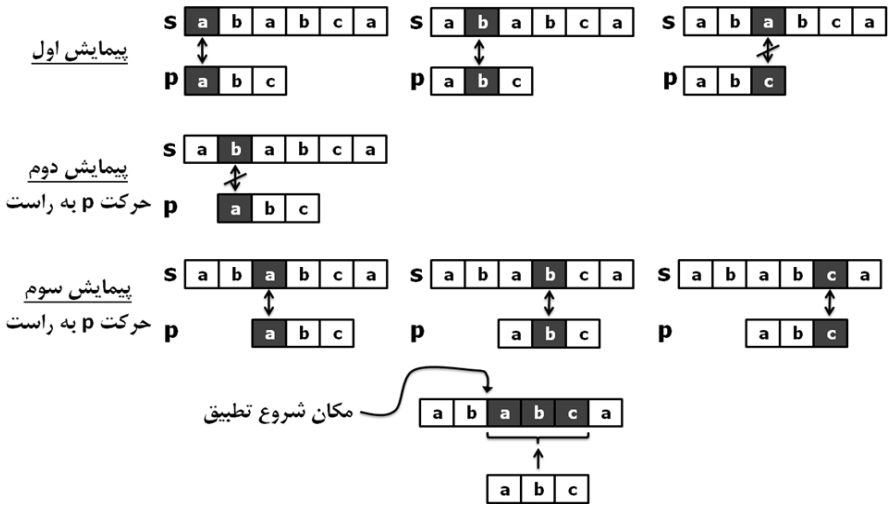
۱- **صورت مسأله:** بررسی وجود رشته‌ی p در رشته‌ی s.

۲- **ورودی:** دو رشته‌ی s و p با طول‌های n و m.

۳- **خروجی:** در صورت وجود رشته‌ی p در رشته‌ی s، مکان آغاز تطبیق در رشته‌ی s برگردانده می‌شود و در غیر این صورت، عدم وجود رشته اطلاع داده می‌شود.

٤-١٤-١- روش معمولی برای تطبیق الگو

ساده‌ترین روش برای یافتن یک تطبیق از رشته‌ی p ، آزمایش همه‌ی حالت‌های ممکن است؛ برای این منظور، نخستین نویسه از رشته‌ی p را با نخستین نویسه از رشته‌ی s مقایسه می‌کنیم؛ اگر باهم برابر بودند، نویسه‌ی دوم از هر دو رشته را با یکدیگر مقایسه می‌کنیم؛ اگر همه‌ی نویسه‌های p با نویسه‌های مقایسه شده از s برابر بود، جستجو به پایان رسیده و تطبیق مورد نظر یافت شده است؛ در غیر این صورت رشته‌ی p را به اندازه‌ی یک نویسه به جلو حرکت داده و مقایسه را از سر می‌گیریم. شکل ۴۴,۴ مراحل کار این روش را نمایش می‌دهد.



شکل ٤٤,٤. مراحل انجام روش معمولی برای حل یک مسأله تطبیق الگو

کد زیر این روش را پیاده‌سازی می‌کند:

```
int ordinal_pattern_matching(string s, string p) {
    int n = s.length();
    int m = p.length();
    for (int i = 0; i<n - m + 1; i++) {
        bool match = true;
        for (int j = 0; j < m; j++)
            if (s[i + j] != p[j]) {
                match = false;
                break;
            }
        if (match) // found a match
            return i;
    }
    return -1;        // no match
}
```

پیچیدگی زمانی این الگوریتم برابر با $O((n-m+1)m) = O(nm)$ است. این پیچیدگی حاصل از دو حلقه‌ی تودرتو است که به ترتیب $n-m+1$ و m مرتبه تکرار می‌شوند (در بدترین حالت). این روش برای رشته‌هایی با طول زیاد، بسیار کند عمل می‌کند. در چنین شرایطی الگوریتم معرفی شده در بخش بعد، راه‌گشا خواهد بود.

٤-١٤-٢- الگوریتم کی-ام-پی برای تطبیق الگو

هر الگوریتمی برای یافتن یک تطبیق الگو، باید دست کم همه‌ی نویسه‌های دو رشته‌ی s و p را یکبار بررسی کند؛ بنابراین پیچیدگی بهینه برای این الگوریتم $O(n+m)$ خواهد بود. با استفاده از الگوریتم کی-ام-پی[۵۱] می‌توانیم به این پیچیدگی اجرایی دست پیدا کنیم.

الگوریتم کی-ام-پی، هر نویسه‌ی رشته‌ی s را فقط یک بار مقایسه می‌کند و با جلوگیری از تکرار در مقایسه، پیچیدگی اجرایی را بهبود می‌بخشد. برای نمونه، روش پیشین نویسه‌های دوم و سوم (b و a) از رشته‌ی s را دو بار مقایسه می‌کند. حال یک پرسش مطرح می‌شود: الگوریتم کی-ام-پی چگونه تکرار در مقایسه را از میان می‌برد؟

این روش، برای جلوگیری از تکرار، پیش از آغاز جستجو، یک جدول پیشوندی از رشته‌ی p تشکیل داده و از آن در جستجو بهره می‌گیرد. بنابراین، روش کی-ام-پی از دو بخش زیر تشکیل می‌شود که در ادامه توضیح داده خواهند شد.

۱- ساخت جدول پیشوندی از رشته‌ی p

۲- جستجو برای یافتن یک تطبیق الگو با به‌کارگیری جدول پیشوندی

٤-١٤-٢-١- ساخت جدول پیشوندی

رشته‌ی p را به صورت $b_0b_2...b_m$ درنظر بگیرید. رشته‌ی $b_0b_2...b_k$ با شرط $0 \le k < m$، پیشوند رشته‌ی p و رشته‌ی $b_kb_{k+1}...b_m$ با شرط $0 < k \le m$، پسوند رشته‌ی p نامیده می‌شوند؛ برای توضیح چگونگی ساخت جدول پیشوندی به این دو مفهوم نیاز داریم. رشته‌ی $abcdabcb$ را در نظر بگیرید. جدول پیشوندی برای هر نویسه از رشته‌ی p به صورت نشان داده شده در جدول ۲۰,۴ پر می‌شود.

جدول ٤,۲۰. جدول پیشوندی رشته‌ی $abcdabcb$

مکان هر نویسه	۰	۱	۲	۳	۴	۵	۶	۷
مقدار جدول پیشوندی	-۱	۰	۰	۰	۰	۱	۲	۰

برای نمونه مقدار نویسه‌ی نمایه‌ی ششم ($p[5]='b'$) بر پایه‌ی قاعده‌ی زیر بدست می‌آید:

رشته‌ی $b_0b_2...b_5$ را رشته‌ای مستقل در نظر بگیرید ($abcdab$). طول بزرگترین پیشوندی از این رشته که پسوند آن نیز باشد، برابر ۲ است؛ بنابراین مقداری که در خانه‌ی مربوطه از جدول پیشوندی قرار می‌گیرد، اندیس

---

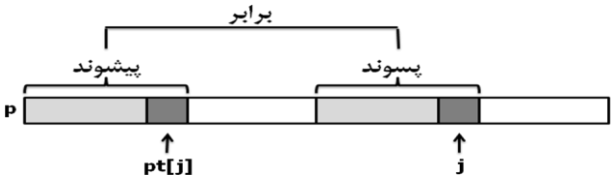
[۵۱] Knuth-Morris-Pratt

آخرین نویسه از پیشوند است (مقدار ۱: یک واحد کمتر از طول بزرگترین پیشوند). مقدار نویسه‌ی اول (نمایه‌ی صفر ام)، ۱- در نظر گرفته می‌شود.
جدول پیشوندی رشته‌ی نمونه‌ی *bcdfgfgbcbcd* به صورت جدول ۲۱,۴ است.

جدول ۲۱,٤. جدول پیشوندی رشته‌ی *bcdfgfgbcbcd*

مکان هر نویسه	۰	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱
مقدار جدول پیشوندی	-۱	۰	۰	۰	۰	۰	۰	۰	۱	۰	۱	۲

شکل ۴۵,۴ نمایی از چگونگی به دست آوردن مقدار جدول پیشوندی برای نویسه‌ی موجود در نمایه‌ی j _ام از رشته‌ی p را نشان می‌دهد. pt در این شکل به جدول پیشوندی رشته‌ی p اشاره دارد.



شکل ٤٥,٤. نحوه به دست آوردن مقدار جدول پیشوندی برای نویسه‌ی موجود در نمایه‌ی j _ام از رشته‌ی p

کد زیر، جدول پیشوندی pt را از رشته‌ی ورودی p می‌سازد. جدول پیشوندی به صورت عمومی تعریف شده و طول آن برای سادگی برابر با ۱۰۰ قرار داده شده است.

```
#include<string>
using namespace std;
#define MAX_N 100
int pt[MAX_N];
void make_prefix_table(string p) {
    int m = p.length();
    int last_match = 0;
    pt[0] = -1;
    for (int i = 0; i<m - 1; i++) {
        while (last_match>0 && p[i + 1] != p[last_match])
            last_match = pt[last_match];
        if (p[i + 1] == p[last_match]) {
            pt[i + 1] = last_match;
            last_match++;
        }
        else
            pt[i + 1] = 0;
    }
}
```

حلقه‌ی موجود در این تابع، $m-1$ مرتبه تکرار می‌شود؛ بنابراین پیچیدگی زمانی آن $O(m)$ است. پرسش دیگری که ممکن است به وجود بیاید، کاربرد جدول پیشوندی است! روش کی-ام-پی برای حرکت دادن رشته‌ی p در طول رشته‌ی s، از مقادیر موجود در جدول پیشوندی بهره می‌برد. نمونه‌ای از این کاربرد در بخش بعد آورده شده است.

٤-١٤-٢-٢- تطبیق الگو با استفاده از جدول پیشوندی

برای روشن‌تر شدن کاربرد جدول پیشوندی، نمونه‌ای از روش کار الگوریتم کی-ام-پی را در ادامه می‌آوریم. رشته‌ی s و p را به صورت زیر درنظر بگیرید:

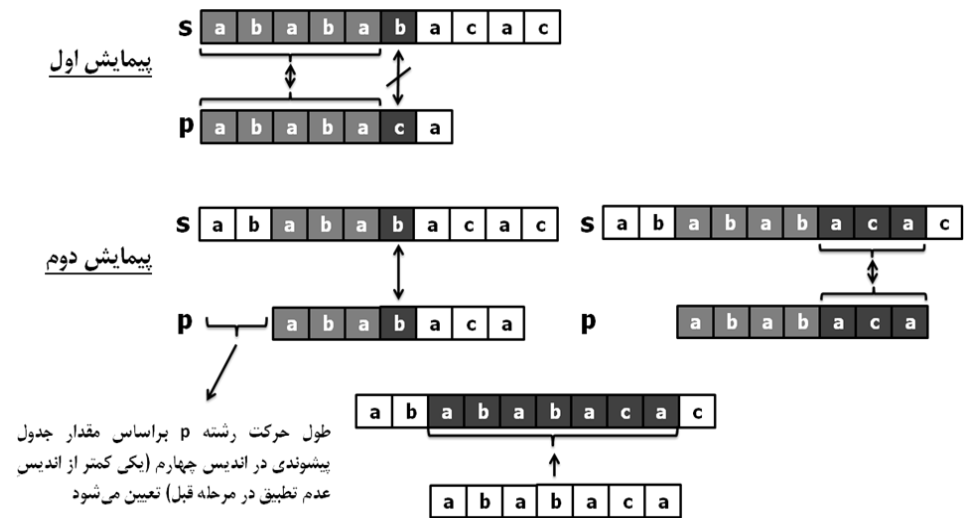
$$s : ababababcac$$

$$p : ababaca$$

در ابتدا، جدول پیشوندی رشته‌ی p را به صورت جدول ۲۲,۴ تشکیل می‌دهیم. بقیه مراحل در شکل ۴۶,۴ نشان داده شده است.

جدول ٢٢,٤. جدول پیشوندی رشته $ababaca$

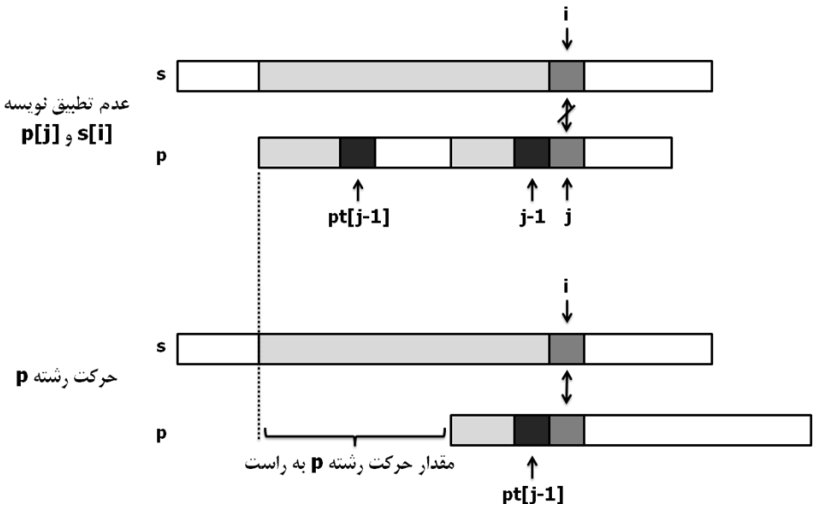
مکان هر نویسه	۰	۱	۲	۳	۴	۵	۶
مقدار جدول پیشوندی	-۱	۰	۰	۱	۲	۱	۰



شکل ٤٦,٤. مراحل جستجوی رشته s **درون رشته** p **با استفاده از جدول پیشوندی**

در پیمایش اول، نویسه‌ی b از رشته‌ی s با نویسه‌ی c از رشته‌ی p برابر نیست؛ در نتیجه رشته‌ی p باید به سمت راست حرکت داده شود. این عدم تطبیق در نمایه‌ی پنجم از رشته‌ی p رخ داده است. بنابراین می‌دانیم پنج نویسه‌ی پیشین از رشته‌ی p با رشته‌ی s مطابقت داشته‌اند. از طرفی مقدار $pt[4]=2$ به این معنی است که سه (نمایه‌ها از صفر آغاز می‌شوند؛ پس $3 \rightarrow 0,1,2 \rightarrow 2$) نویسه‌ی آخر از رشته‌ی p برابر با سه نویسه‌ی اول

از این رشته هستند (پیشوندی به طول سه، برابر با پسوندی به همین طول). در نتیجه می‌توانیم رشته‌ی p را به اندازه‌ی 2 نویسه به سمت راست حرکت داده و مقایسه را ادامه دهیم. این حرکت تفاوت روش کی–ام–پی و روش ساده را روشن می‌سازد. شکل ۴۷,۴ نمایی از عملکرد روش کی–ام–پی را هنگام حرکت دادن رشته‌ی p به سمت راست نشان می‌دهد.



شکل ٤٧,٤. عملکرد روش کی–ام–پی را هنگام حرکت دادن رشته‌ی p به سمت راست

کد زیر الگوریتم کی–ام–پی را با استفاده از جدول پیشوندی پیاده‌سازی می‌کند:

```
int KMP_pattern_matching(string s, string p) {
    int n = s.length();
    int m = p.length();
    make_prefix_table(p);
    int last_match = 0;
    for (int i = 0; i<n; i++) {
        while (last_match>0 && s[i] != p[last_match])
            last_match = pt[last_match - 1] + 1;  // move p to right
        if (s[i] == p[last_match])
            last_match++;
        if (last_match == m)
            return i - m + 1;
    }
    return -1; // no match
}
```

الگوریتم ارائه شده پس از یافتن نخستین تطبیق الگو در دو رشته، خارج می‌شود. به راحتی می‌توان این الگوریتم را برای یافتن همه‌ی تطبیق‌ها گسترش داد. برای تحلیل پیچیدگی زمانی این الگوریتم، دو بخش اصلی آن را بررسی می‌کنیم:

۱- پیچیدگی زمانی ساخت جدول پیشوندی همان‌طور که در بخش گذشته گفته شد، برابر با $O(m)$ است.

۲- حلقه‌ی for نیز حداکثر n مرتبه تکرار می‌شود؛ در نتیجه پیچیدگی اجرایی آن $O(n)$ است.

پیچیدگی زمانی کل برابر با مجموع دو پیچیدگی بالا و از مرتبه‌ی $O(n+m) \in O(n)$ خواهد بود؛ زیرا $m \leq n$ است. روش‌های زیر نیز از روش‌های تطبیق الگو هستند که فقط به پیچیدگی زمانی آن‌ها اشاره می‌کنیم.

۱- الگوریتم بویِر-مور[۵۲]: از روش‌های بسیار مناسب و در عمل از کاربردی‌ترین روش‌ها است؛ برای رشته‌های با طول زیاد بسیار سریع عمل می‌کند. پیچیدگی زمانی این روش در بهترین حالت از درجه‌ی زیر-خطی است ($O(\frac{n}{m})$)؛ یعنی قادر به یافتن الگو با انجام کمتر از n مقایسه است.

۲- الگوریتم رابین-کارپ[۵۳]: به علت پیچیدگی زمانی $O(nm)$ در بدترین حالت، در عمل کاربرد چندانی ندارد.

جدول ۲۳,۴ پیچیدگی چهار روش اشاره شده را دسته‌بندی می‌کند.

جدول ٢٣,٤. مقایسه پیچیدگی زمانی روش‌های متفاوت برای حل مسأله تطابق الگو

روش جستجوی رشته	پیچیدگی زمانی در حالت میانگین	پیچیدگی زمانی در بدترین حالت
روش معمولی	$O(nm)$	$O(nm)$
روش کی-ام-پی	$O(n)$	$O(n)$
روش بویِر-مور	$\approx O(\frac{n}{m})$	$O(n)$
روش رابین-کارپ	$O(n)$	$O(nm)$

٤-١٥- غربال اِراتُستِنِس

غربال اِراتُستِنِس[۵۴] روشی بسیار ساده، زیبا و دیرینه برای یافتن اعداد اول است که توسط اِراتُستِنِس، ریاضیدان یونانی ابداع شده است. این روش برای اعداد کوچکتر از 10^7 بسیار سریع عمل می‌کند و پیاده‌سازی آن نیز آسان و سریع است.

---

[۵۲] Boyer–Moore

[۵۳] Rabin-Karp

[۵۴] Sieve of Eratosthenes

برای یافتن اعداد اول موجود در بازه‌ی $[2..n]$، ابتدا آن‌ها را در یک لیست قرار داده و مراحل زیر را بر روی آن پیاده می‌کنیم.

۱- عدد نخست در لیست را برگزیده و به عنوان عدد اول علامت می‌زنیم. سپس همه ضرایب آن را از لیست خط می‌زنیم.

۲- عدد خط نخورده‌ی بعدی را برگزیده و به عنوان عدد اول علامت می‌زنیم و در ادامه مانند مرحله پیشین عمل می‌کنیم.

۳- تا زمانی که همه اعداد خط نخورده‌اند، مرحله ۲ را تکرار می‌کنیم.

نکته‌های زیر از عوامل سرعت دهنده به این روش هستند:

۱- مراحل بالا تا زمانی که به نخستین عدد خط نخورده بزرگتر از $\sqrt{n}$ برسیم، ادامه پیدا می‌کند.

۲- برای حذف ضرایب عدد x، عملیات حذف را از ضریب x^2 آغاز می‌کنیم.

۳- در پایان کار، همه اعداد خط نخورده‌ی بزرگتر از $\sqrt{n}$ را به عنوان عدد اول علامت می‌زنیم.

برای روشن شدن توضیحات بالا، غربال اراتستن را مرحله به مرحله، برای یافتن اعداد اول میان ۲ تا ۲۹ به کار می‌گیریم.

۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵
۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹

در مرحله‌ی نخست، عدد ۲ را به عنوان عدد اول علامت زده و ضرایب آن را با آغاز از $2^2 = 4$ حذف می‌کنیم.

۲ ۳ ~~۴~~ ۵ ~~۶~~ ۷ ~~۸~~ ۹ ~~۱۰~~ ۱۱ ~~۱۲~~ ۱۳ ~~۱۴~~ ۱۵
~~۱۶~~ ۱۷ ~~۱۸~~ ۱۹ ~~۲۰~~ ۲۱ ~~۲۲~~ ۲۳ ~~۲۴~~ ۲۵ ~~۲۶~~ ۲۷ ~~۲۸~~ ۲۹

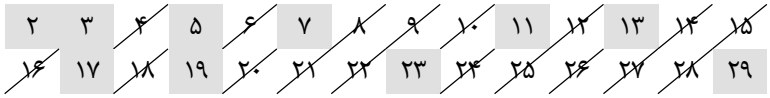
در مرحله دوم، عدد خط نخورده‌ی بعدی یعنی ۳ را به عنوان عدد اول علامت زده و ضرایب آن را با آغاز از $3^2 = 9$ حذف می‌کنیم.

۲ ۳ ~~۴~~ ۵ ~~۶~~ ۷ ~~۸~~ ~~۹~~ ~~۱۰~~ ۱۱ ~~۱۲~~ ۱۳ ~~۱۴~~ ~~۱۵~~
~~۱۶~~ ۱۷ ~~۱۸~~ ۱۹ ~~۲۰~~ ~~۲۱~~ ~~۲۲~~ ۲۳ ~~۲۴~~ ۲۵ ~~۲۶~~ ~~۲۷~~ ~~۲۸~~ ۲۹

در مرحله‌ی سوم، عدد ۵ را به عنوان عدد اول علامت زده و ضرایب آن را با آغاز از $5^2 = 25$ حذف می‌کنیم.

۲ ۳ ~~۴~~ ۵ ~~۶~~ ۷ ~~۸~~ ~~۹~~ ~~۱۰~~ ۱۱ ~~۱۲~~ ۱۳ ~~۱۴~~ ~~۱۵~~
~~۱۶~~ ۱۷ ~~۱۸~~ ۱۹ ~~۲۰~~ ~~۲۱~~ ~~۲۲~~ ۲۳ ~~۲۴~~ ~~۲۵~~ ~~۲۶~~ ~~۲۷~~ ~~۲۸~~ ۲۹

عدد خط نخورده‌ی بعدی، عدد ۷ است که بزرگتر از $\sqrt{29} = 5$ است؛ بنابراین همه‌ی اعداد خط نخورده‌ی باقیمانده را به عنوان عدد اول علامت می‌زنیم.



ضعف این روش در فضای اشغالی توسط آن است. زیرا این روش برای یافتن اعداد اول میان $[2..n]$ به فضایی به طول n احتیاج دارد. کد زیر این روش را پیاده‌سازی می‌کند. با اجرای کد زیر، اعداد اول کوچکتر از n در خروجی چاپ می‌شوند.

```
#include<iostream>
#include<cmath>
using namespace std;
const int n = 100;
int primes[n] = {0}; // 0:uncheck, 1:prime, -1:removed
void Eratosthenes() {
    int limit = (int)sqrt((double)n);
    int coeff_limit = 0;
    int curr_prime = 2;
    while (curr_prime <= limit) {
        primes[curr_prime] = 1;
        coeff_limit = n / curr_prime;
        // remove coefficient of current prime
        for (int i = curr_prime; i <= coeff_limit; i++)
            primes[i * curr_prime] = -1;
        // find next prime
        for (int i = curr_prime + 1; i <= n; i++)
            if (primes[i] == 0) {
                curr_prime = i;
                break;
            }
    }
    // mark remained numbers as prime
    for (int i = curr_prime; i <= n; i++)
        if (primes[i] == 0)
            primes[i] = 1;
}
int main() {
    Eratosthenes();
    // print primes in output
    for (int i = 2; i <= n; i++)
        if (primes[i] == 1)
            cout << i << endl;
    return 0;
}
```

خروجی حاصل از اجرای این کد، در شکل ۴۸,۴ نمایش داده شده است.

~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

شکل ٤٨,٤. نتیجه اجرای روش غربال اِراتُستِنِس برای یافتن اعداد اول در بازه‌ی ۱ تا ۱۰۰

تمرین‌ها

۱- برنامه‌ی زیر عنصر x را در آرایه‌ی S جستجو می‌کند و فراخوانی اولیه آن به شکل location(1,n) است. پیچیدگی زمانی و فضایی آن را محاسبه کنید.

```
index location(index low, index high) {
    index mid;
    if (low > high)
        return 0;
    else {
        mid = (low + high) / 2;
        if (x == S[mid])
            return  mid;
        else if (x < S[mid])
            return location(low, mid - 1);
        else
            return location(mid + 1, high);
    }
}
```

۲- دو برنامه‌ی زیر برای محاسبه‌ی ضریب دوجمله‌ای نوشته شده‌اند. پیچیدگی زمانی آن‌ها را با هم مقایسه کنید.

برنامه‌ی اول:

```
int bin(int n, int k) {
    if (k == 0 || n == k)
        return 1;
```

```
    else
        return bin(n - 1, k - 1) + bin(n - 1, k);
}
```

برنامه دوم:

```
int bin2(int n, int k) {
    index i, j;
    int B[0..n][0..k]
    for (i = 0; i ≤ n; i++)
        if (j == 0 || j == i)
            B[i][j] = 1;
        else
            B[i][j] = B[i - 1][j - 1] + B[i - 1][j];
    return B[n][k];
}
```

۳- برنامه‌ای بنویسید که گراف شکل ۴۹,۴ را به صورت اول‌سطح پیمایش کند. گراف به صورت زیر در ورودی داده می‌شود.

a. در خط نخست تعداد گره‌های گراف n، آورده می‌شود. شماره‌ی گره‌ها از ۱ تا n است.

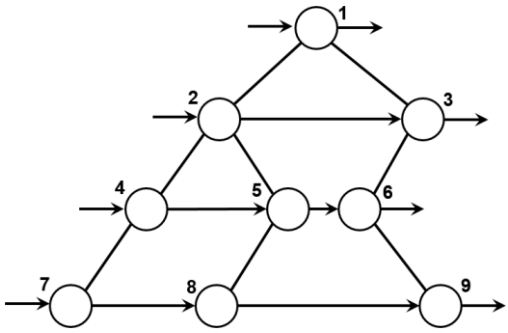
b. در خط نخست، گره‌ی آغازین آورده می‌شود (آغاز پیمایش) و سپس تعداد گره‌های متصل به آن و خود آن گره‌ها داده می‌شود.

c. در $n-1$ خط بعد، شماره‌ی یک گره، تعداد گره‌های متصل به آن و سپس خود آن گره‌ها آورده می‌شود.

d. دقت کنید که ترتیب گره‌ها لزوما صعودی نیست.

e. ترتیب ملاقات گره‌ها را در خروجی چاپ کنید. در هر سطح، گره‌ها به ترتیب صعودی ملاقات می‌شوند.

Input	Output
9 1 2 2 3 2 3 1 4 5 3 2 1 6 4 2 2 7 5 2 2 8 6 2 3 9 7 1 4 8 1 5 9 1 6	1 2 3 4 5 6 7 8 9



شکل ٤٩,٤. نمونه‌ای از یک گراف

۴- برنامه‌ای بنویسید که نقشه‌ی ایران را به صورت یک گراف دریافت کرده و شهرهای موجود در آن را با چهار رنگ متفاوت رنگ‌آمیزی کند؛ به شکلی که هیچ دو شهر مجاوری دارای رنگ‌های یکسان نباشند (شکل ۵۰,۴).

شکل ٥٠,٤. نقشه ایران

۵- دو روش مرتب‌سازی را با پیچیدگی خطی‌لگاریتمی پیاده‌سازی کرده و سرعت آن‌ها را برای یک مجموعه اعداد به طول ۵۰۰۰ با هم مقایسه کنید.

۶- برنامه‌ای بنویسید که یک پرونده متنی کوچک را دریافت کرده، محتوای آن را به صورت بهینه رمز کرده و در خروجی چاپ کند.

۷- برنامه‌ای بنویسید که دو رشته از ورودی دریافت کرده و بررسی کند که رشته‌ی دوم چند مرتبه در رشته‌ی اول تکرار شده است. سرعت عمل برنامه را برای دو رشته‌ی بلند نیز بسنجید.

۸- برنامه‌ای بنویسید که یک پرونده متنی کوچک را دریافت کرده و فراوانی همه‌ی کلمات موجود در آن را محاسبه کند. سپس کلماتی را که بیش از ۱ مرتبه تکرار شده‌اند، حذف کرده و حاصل را در فایل جدیدی بنویسد.

۹- برنامه‌ای بنویسید که عملگرهای جمع، تفریق، ضرب و تقسیم را بر روی اعداد صحیح بزرگ انجام دهد. برای ضرب و تقسیم از الگوریتم‌های بهتر از درجه‌ی دوم استفاده کنید.

مراجع

Arefin, A.S., 2006. *Art of Programming Contest*, Reviewed by Dr. M. Lutfar Rahman and Forworded by Professor Miguel Revilla, University de Valladolid, Spain, Gyankosh Prokashoni, Dhaka, Bangladesh, First Ed., ISBN: 984-32-3382-4.

Cormen, T.H. et al., 2009. *Introduction to algorithms* 3rd ed., MIT Press and McGraw-Hill.

Skiena, S.S. & Revilla, M.A., 2006. *Programming challenges: The programming contest training manual*, Springer Science & Business Media.

فصل ۵- مسأله‌های ریاضی

یکی از دسته‌های مهم و در عین حال جذاب از مسائلی که در مسابقات برنامه‌نویسی مطرح می‌شوند، مسأله‌های ریاضی هستند. این دسته مسائل معمولا ظاهر ساده‌ای دارند ولی پیدا کردن جواب مناسب برای آنها کار دشواری است. نکته مثبت در مورد این مسائل این است که اگر از زاویه‌ی درست به آنها نگاه کنید، حل آنها می‌تواند بسیار ساده باشد. به همین دلیل آشنایی با مفاهیم ریاضی و توانایی مدلسازی ریاضی از مسائل، یکی از پیش‌نیازهای اصلی برای شرکت در مسابقات برنامه‌نویسی محسوب می‌شود.

یکی از سخت‌ترین انواع مسأله‌های ریاضی، مسأله‌های هندسی هستند. برخی از مسأله‌های هندسی به دلیل پیچیدگی الگوریتم یا پیچیدگی ساختمان داده‌ای که برای پیاده‌سازی آن نیاز است، بسیار دشوار می‌نمایاند و برای حل آنها افزون بر داشتن دانش پایه‌ای از هندسه تحلیلی و درک ویژگی‌های هندسی مسأله، تسلط کافی بر ساختمان داده‌ها و الگوریتم‌ها و نیز آشنایی با محاسبات عددی هم نیاز است. اگر ویژگی‌های هندسی مسأله به خوبی درک نشود آن‌گاه هیچ الگوریتم یا ساختمان داده‌ای نمی‌تواند در حل مسأله‌های این حوزه کمک کند.

حوزه‌ی دیگری از مباحث ریاضی که می‌تواند در موفقیت در مسابقات بسیار موثر باشد، ترکیبیات و نظریه‌ی اعداد است. به همین دلیل، در این فصل به معرفی پیشنیازها و تعاریف اصلی در این زمینه نیز پرداخته می‌شود.

به دلیل گستردگی موضوع در این فصل از کتاب به اثبات‌های ریاضی پرداخته نمی‌شود و کوشش می‌شود با ساده‌ترین زبان، به چند نمونه از الگوریتم‌های مشهور در این حوزه پرداخته شود. برای مطالعه بیشتر و به ویژه بررسی اثبات‌های ریاضی می‌توان به مراجع معرفی شده در پایان فصل مراجعه نمود.

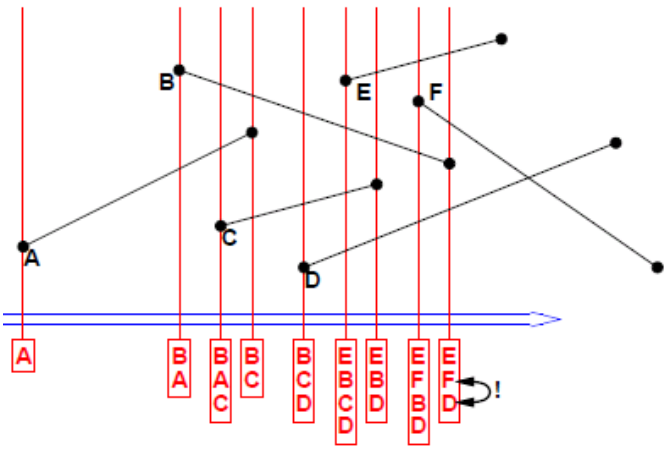
۵-۱- تقاطع پاره‌خط‌ها

یکی از مسائل هندسی پرکاربرد، مسأله تقاطع پاره‌خط‌ها است. در این مسأله تقاطع میان چندین پاره‌خط به دست می‌آید. به سادگی می‌توان با به دست آوردن تقاطع میان جفت جفت پاره‌خط‌ها تقاطع میان آنها را به دست آورد ولی مرتبه این راه حل n^2 است. برای مسائل بزرگی که تقاطع‌ها باید میان تعداد زیادی پاره‌خط به دست آید و تعداد این تقاطع‌ها بسیار کم است، راه‌حل‌های با مرتبه کمتری نیز وجود دارد. در این دست راه‌حل‌ها که الگوریتم‌های حساس به خروجی نامیده می‌شوند، مرتبه الگوریتم به تعداد پاسخ‌ها وابسته است. در این مسأله، تعداد تقاطع‌ها مشخص‌کننده مرتبه الگوریتم است. در اینجا به جای اینکه تقاطع میان همه‌ی پاره‌خط‌ها حساب شود، فقط تقاطع میان پاره‌خط هایی که نزدیک به هم هستند به دست می‌آید.

مسأله یافتن تقاطع پارهخطها

۱- **صورت مسأله:** یافتن نقطههای تقاطع میان تعدادی پارهخط داده شده در صفحه
۲- **ورودی:** مجموعهای از پارهخطها در صفحه به نام S
۳- **خروجی:** مجموعه نقطههای برخورد میان پارهخطهای داده شده همراه با پارهخطهایی که هر کدام از نقطههای تقاطع را به وجود آوردهاند.

برای این که بتوان پارهخطهای نزدیک به هم را یافت، در آغاز همهی نقطههای آغاز و پایان پارهخطها را بر حسب مقدار x (و اگر مساوی بود بر حسب y) مرتب میکنیم؛ سپس از سمت چپترین نقطه آغاز کرده و یک خط پیمایشی (یا جارویی) را جلو میبریم. زمانی که به نخستین نقطه از یک پارهخط دیگر رسیدیم، آن را به مجموعه پارهخطهای نزدیک به هم میافزاییم و به پیمایش ادامه میدهیم؛ تا به نقطهی آغاز پارهخط بعدی برسیم و آن را بیفزاییم و این روال را ادامه میدهیم. اگر به نقطهی پایانی هرکدام از پارهخطها برسیم، مجموعه به دست آمده تا این لحظه را به یک صف از مجموعهها میافزاییم و بررسی میکنیم آیا میان پارهخطهای درون مجموعههای کنونی درون صف برخوردی وجود دارد یا خیر. سپس آن پارهخط را از مجموعه پارهخطهای نزدیک هم حذف کرده و یک مجموعه جدید به دست میآوریم. به این ترتیب ادامه میدهیم تا همهی پارهخطها پیمایش شده و به نقطهی پایانی (سمت راستترین نقطه) برسیم. به این ترتیب پارهخطهای نزدیک به هم و تقاطعها به دست میآیند. این روش به این دلیل درست کار میکند که پارهخطهای از هم دور (از نظر مختصات x) با هم تقاطعی نخواهند داشت و نیازی نیست وجود تقاطع میان این پارهخطهای از هم دور بررسی شود. شکل ۱,۵ چگونگی انجام این کار را تا اندازهای نشان داده و مشخص میکند چگونه مجموعه تغییر مییابد. به هر کدام از نقطههایی که هنوز خط پیمایش به آنها نرسیده است و دربردارندهی نقطههای آغاز و پایان پارهخطها و نقطههای برخورد یافت شده تا این لحظه هستند، رخداد گفته میشود؛ و مجموعه مرتب شده از چپ به راست این رخدادها، صف رخداد نامیده میشود.

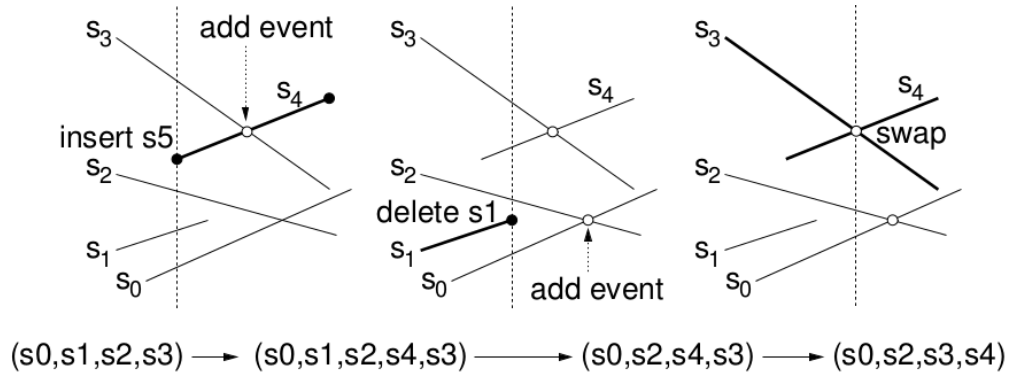


شکل ۱,۵. چگونگی تقریبی یافتن نقطههای برخورد

البته در این روش هنوز تعداد بررسی برخوردها زیاد است، زیرا الگوریتم تقریباً همان مرتبه n^2 را دارد. برای این که بتوان این تعداد را کاهش داد و وابسته به تعداد برخوردها کرد، باید بررسی‌های کمتری انجام شود. برای کاهش بررسی‌ها، باید فقط دو پاره‌خط کناری پاره‌خطی که بناست حذف شود، بررسی گردند؛ و این بررسی نیز فقط از نقطه‌ی پایانی پاره‌خطی که بناست حذف شود انجام شود و برخوردهای پیش از آن یافته شده باشند. همچنین در نقطه‌های برخورد یافت شده باید تغییری در ترتیب پاره‌خط‌ها ایجاد شود تا نشان‌دهنده‌ی اثر این برخورد باشد.
برای نگهداری وضعیت این مجموعه‌ی پاره‌خط‌ها نسبت به خط پیمایشی، باید یک ساختمان داده دیگر شبیه به صف نیز نگهداری شود که وضعیت پیمایش نامیده می‌شود. چگونگی پیاده‌سازی دقیق‌تر این ساختمان داده به شکل الگوریتم، در ادامه آورده شده است.

۱- همه‌ی نقطه‌های آغاز و پایان پاره‌خط‌ها به ترتیب درونِ صف رخدادهای Q از چپ به راست گذاشته شود.
۲- وضعیت پیمایش T به صورت خالی ساخته شود.
۳- تا هنگامی که Q خالی نشده است مرحله‌های زیر انجام شود:
۳-۱- نقطه p_i (رخداد i اُم) از آغاز صف Q برداشته شود.
۳-۲- این نقطه (رخداد) می‌تواند سه وضعیت زیر را داشته باشد.
۳-۲-۱- نقطه‌ی آغاز یک پاره‌خط باشد.
۳_۲_۱_۱_ پاره‌خطی که این نقطه، آغاز آن است به T افزوده شود.
۳_۲_۱_۲_ بسته به مختصات y نقطه کنونی و مختصات y دو پاره‌خط بالا و پایین این پاره خط نسبت به خط پیمایش، برخورد میان این پاره‌خط و دو پاره‌خط بالایی و پایینی بررسی شود.
۳_۲_۱_۳_ نقطه یا نقطه‌های برخورد یافت شده به T افزوده شود.
۳-۲-۲- نقطه‌ی پایان یک پاره‌خط باشد.
۳_۲_۲_۱_ این پاره‌خط از T برداشته شود.
۳_۲_۲_۲_ بررسی برخوردِ میان دو پاره‌خط پیش و پس این پاره‌خط در T
۳_۲_۲_۳_ نقطه یا نقطه‌های برخورد یافت شده به T افزوده شود.
۳-۲-۳- نقطه‌ی برخورد باشد.
۳_۲_۳_۱_ ترتیب دو پاره‌خطی که در این نقطه برخورد داشته‌اند در T جابجا شود.
۳_۲_۳_۲_ بررسی برخورد هر کدام از این دو پاره‌خط جابجا شده در T با پاره‌خط‌های جدید کناری خود در T (هر کدام فقط با یک پاره‌خط دیگر بررسی را انجام می‌دهند)
۳_۲_۳_۳_ نقطه یا نقطه‌های برخورد یافت شده به T افزوده شود.

شکل ۲,۵ چگونگی اجرای این الگوریتم را بر روی تعدادی پاره‌خط نشان می‌دهد. همچنین تغییر T به همراه پیمایش خط پیمایش و وضعیت‌های پیش آمده در شکل نشان داده شده است.



شکل ۲,۵. چگونگی اجرای الگوریتم یافتن برخوردها

خط پیمایشی می‌تواند یک خط افقی به جای خط عمودی باشد و به جای این که پیمایش را از چپ به راست انجام دهد، از بالا به پایین عمل پیمایش را انجام دهد. در آغاز کار نیز نقطه‌ها بر پایه‌ی مختصات y مرتب شوند. روند کار به طور کامل همانند خط عمودی است و نقطه‌های آغاز و پایان در نظر گرفته می‌شود. برخی پیاده‌سازی‌های این الگوریتم بر پایه‌ی خط افقی است و پیمایش از بالا به پایین انجام می‌شود.

برای سادگی الگوریتم نوشته شده، برخی از حالت‌ها در نظر گرفته نشدند. باید بررسی شود که برای حالت‌هایی که چند پاره‌خط با هم در یک نقطه برخورد کرده‌اند، عمل درستی انجام شده و این برخوردها گزارش شوند؛ به ویژه زمانی که این برخوردها در نقطه‌ی پایانی یک پاره‌خط باشند؛ همچنین شاید نقطه‌ی پایان یک پاره‌خط، نقطه‌ی آغاز پاره‌خط دیگری نیز باشد. حالتی که پاره‌خط عمودی در مجموعه پاره‌خط‌ها باشد نیز باید بررسی شود.

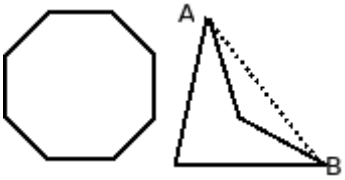
۵-۲- محاسبه زاویه‌های چندضلعی

چندضلعی ها دنباله‌ی بسته‌ای از پاره‌خط‌های به هم متصل هستند. منظور از دنباله بسته این است که راسهای اول و آخر این دنباله یکی هستند. در اینجا منظور از پاره‌خط‌های متصل این است که انتهای هر پاره‌خط با ابتدای پاره‌خط بعدی در دنباله مشترک است.

چندضلعی‌ها ساختارهای پایه‌ای مهمی برای نشان دادن شکل‌های مختلف در یک صفحه هستند. در این حالت به جای این که صراحتا از لیستی از پاره‌خط‌ها برای نشان دادن یک شکل استفاده کنیم، می‌توانیم به سادگی n راس واقع بر محیط چندضلعی را در یک لیست ذخیره کنیم. بنابراین هر دو راس متوالی در این لیست، نشانگر یکی از پاره‌خط‌های تشکیل‌دهنده چندضلعی هستند. بنابراین می‌توان ساختار ساده‌ای به این صورت برای ذخیره‌سازی یک چندضلعی تعریف کرد:

```
struct polygon {
    int n;              /* number of points in polygon */
    point p[MAXPOLY];   /* array of points in polygon */
};
```

به یک چندضلعی، چندضلعی محدب گفته می‌شود اگر با اتصال هر دو نقطه‌ی درون آن به یکدیگر، پاره‌خطی که این دو را به هم پیوند می‌زند از شکل بیرون نزده و کامل درون شکل باشد. برای نمونه در شکل ۳,۵ دو چند ضلعی نشان داده شده که در آن چند ضلعی سمت چپ محدب است. با این حال چند ضلعی سمت راست یک چندضلعی محدب نیست و چندضلعی مقعر نامیده می‌شود. برای نمونه اگر میان دونقطه‌ی A و B در این چند ضلعی پاره‌خطی کشیده شود آن‌گاه پاره‌خط از شکل بیرون می‌زند. با کمی دقت به وضوح می‌توان دریافت که اندازه زوایای درونی یک چندضلعی محدب نمی تواند از ۱۸۰ درجه یا π رادیان تجاوز کند.



شکل ۳,۵. نمونه‌ای از یک چندضلعی محدب (سمت چپ) و یک چندضلعی مقعر (سمت راست)

یکی از مسائل جالب در مورد چندضلعی‌های محدب محاسبه زاویه ایجاد شده توسط سه نقطه متوالی است. با استفاده از قانون پادساعتگرد بودن نیازی به دانستن مقدار واقعی زاویه‌ها در اکثر الگوریتم‌های هندسی حذف می‌شود. این رویه که با $ccw(a,b,c)$ نشان داده می‌شود، بررسی می‌کند که آیا نقطه‌ی c در سمت راست پاره‌خط فرضی ترسیم شده بین نقاط a و b واقع شده است یا خیر. اگر این شرط برقرار باشد، اندازه زاویه‌ای که با پیمایش پادساعتگرد از نقطه a تا c به دست می‌آید، کمتر از ۱۸۰ درجه خواهد بود. در غیر این صورت، یا نقطه c در سمت چپ خط فرضی ترسیم شده بین نقاط a و b واقع شده است یا این که سه نقطه دقیقا در یک راستا قرار دارند. برای بررسی کردن این سه حالت، با استفاده از مقدار علامت‌دار مساحت مثلث متشکل از سه نقطه a، b و c به سادگی امکان‌پذیر است. با استفاده از روابط هندسی می‌توان نشان داد که مساحت مثلثی از رابطه زیر قابل محاسبه است:

$$\frac{1}{2}.A(T)=\frac{1}{2}.\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix}=\left(a_xb_y-a_yb_x+a_yc_x-a_xc_y+b_xc_y-c_xb_y\right)/2$$

کد بعدی پیاده‌سازی این رابطه ریاضی را نشان می‌دهد:

```
double signed_triangle_area(point a, point b, point c) {
    return ((a[X] * b[Y] - a[Y] * b[X] + a[Y] * c[X]
        - a[X] * c[Y] + b[X] * c[Y] - c[X] * b[Y]) / 2.0);
}
```

با استفاده از تابع بالا به سادگی می‌توان سه حالت مربوط به ارتباط نقاط a، b و c را بررسی کرد. سه تابع نشان داده شده در کد بعدی برای بررسی این سه حالت پیاده‌سازی شده‌اند:

```
#define EPSILON 0.000001
bool ccw(point a, point b, point c) {
    return (signed_triangle_area(a, b, c) > EPSILON);
}
bool cw(point a, point b, point c) {
    return (signed_triangle_area(a, b, c) < EPSILON);
}
bool collinear(point a, point b, point c) {
    double signed_triangle_area();
    return (fabs(signed_triangle_area(a, b, c)) <= EPSILON);
}
```

در کد بالا برای جلوگیری از خطای محاسباتی مربوط به اعداد اعشاری از یک مقدار مثبت بسیار کوچک به جای صفر استفاده شده است.

۵-۳- مساحت چندضلعی محدب

یکی از مسألەهای جالب در مورد چندضلعیهای محدب، نحوەی محاسبه مساحت آنهاست. در این بخش به معرفی یک الگوریتم ساده و در عین حال جالب برای محاسبه مساحت چندضلعیهای محدب پرداخته میشود.

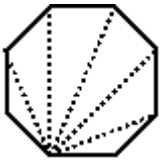
مسأله مساحت چندضلعی محدب

۱- **صورت مسأله:** محاسبه مساحت چندضلعی محدب برای تعدادی نقطه ($x_0, x_1, ..., x_{n-2}, x_{n-1}$)

۲- **ورودی:** مجموعهای از نقطهها روی صفحه که به ترتیب ساعتگرد مرتب شدهاند و این نقطهها تشکیل یک چندضلعی محدب میدهند و هیچ سه نقطهای بر یک خط مستقیم قرار ندارند.

۳- **خروجی:** مساحت چندضلعی

اگر از نقطه x_0 از چند ضلعی به همهی نقطههای دیگر چند ضلعی به جز دو نقطه کنار آن x_0 و x_{n-1} پارەخطهایی کشیده شود (شکل ۴,۵) آنگاه مساحت چندضلعی به کمک جمع کردن مساحت مثلثهای درون آن به دست میآید. به این روش، مثلثسازی گفته میشود چون مساله محاسبه مساحت چندضلعی را از طریق محاسبه مساحت مثلثهای کوچکتر تشکیلدهنده آن حل میکند.



شکل ٤,٥. تقسیمبندی چند ضلعی به تعدادی مثلث

روند ساخت مثلث‌ها به این صورت است که نخستین مثلث از سه نقطه‌ی (x_0, x_1, x_2) ساخته می‌شود. مثلث دوم از نقطه‌های (x_0,x_2,x_3) ساخته می‌شود و مثلث سوم از نقطه‌های (x_0, x_3, x_4) ساخته می‌شود و همین روند ادامه می‌یابد و مثلث پایانی از نقطه‌های (x_0, x_{n-3}, x_{n-2}) ساخته می‌شود.

برای محاسبه مساحت هر مثلث نیاز است تا طول ضلع‌ها به دست آید. طول پاره‌خط میان دو نقطه p و q به کمک فاصله اقلیدسی به این صورت به دست می‌آید:

$$|pq| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

اگر اندازه سه ضلع مثلث a، b و c باشد مساحت مثلث بر پایه نصف محیط مثلث $p = \frac{a+b+c}{2}$ به کمک رابطه هرون[۵۵] به این صورت محاسبه می‌شود:

$$Area = \sqrt{p \times (p-a) \times (p-b) \times (p-c)}$$

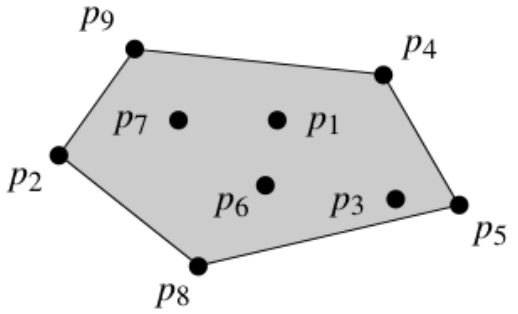
شبه کد این الگوریتم در ادامه آمده است. تابع‌های sqrt و pow2 در این شبه کد به ترتیب برای محاسبه‌ی جذر و توان دوم یک عدد اعشاری استفاده شده‌اند.

```
float ComputeArea(Point points[1..n]) {
    float area = 0, p, a, b, c;
    for i = 2 to n - 1 {
        a = Distance(points[1], points[i]);
        b = Distance(points[1], points[i + 1]);
        c = Distance(points[i], points[i + 1]);
        p = (a + b + c) / 2;
        area += sqrt(p * (p - a) * (p - b) * (p - c));
    }
    return area;
}
float Distance(Point p, Point q) {
    return sqrt(pow2(p.x - q.x) + pow(p.y - q.y));
}
```

۵-٤- پوشش محدب

پوشش محدب به نام‌های غشا، پوش، پوسته یا رویه‌ی محدب نیز شناخته می‌شود. مجموعه‌ای از چند نقطه در صفحه را در نظر بگیرید. پوشش محدب این مجموعه چندضلعی محدبی با حداقل تعداد رئوس است که شامل همه‌ی نقطه‌های درون این مجموعه باشد. در شکل ۵,۵ مجموعه‌ای از نقاط به همراه پوشش محدب آنها نشان داده شده است. با توجه به شکل ۵,۵، ورودی این مسأله مجموعه نقطه‌های p_1 تا p_9 است و خروجی الگوریتم باید نقطه‌های P_4, P_5, P_8, P_2, P_9 باشد.

[۵۵] Heron's formula



شکل ۵,۵. مجموعه‌ای از نقاط به همراه پوشش محدب آنها

مسأله پوشش محدب

۴- **صورت مسأله:** یافتن پوشش محدب ((CH(P)) برای تعدادی نقطه در مجموعه P

۵- **ورودی:** مجموعه‌ای از نقطه‌ها (P) روی صفحه

۶- **خروجی:** مجموعه‌ی مرتب L از گره‌ها (نقطه‌ها) که با گذر از آن‌ها (کشیدن پاره‌خط‌ها) در جهت عقربه های ساعت، پوشش محدب CH(P) به دست می‌آید.

در ادامه دو روش برای حل مسأله پوشش محدب ارائه می‌شود.

۵-٤-۱- نخستین کوشش برای حل

پاره‌خط‌هایی از نقطه‌ها در پوشش گذاشته می‌شوند که همه‌ی نقاط دیگر شکل نسبت به آن نقطه‌ها در یک طرف دیگر قرار گیرند و به این ترتیب مشخص می‌شود چه پاره‌خط‌هایی در پوشش قرار دارند. اگر جهت حرکت عقربه‌های ساعت را در نظر بگیریم، در این صورت همه نقطه‌های دیگر نسبت به پاره‌خط‌های بیرونی در سمت راست آن پاره خط‌ها خواهند بود. البته اگر جهت عکس حرکت عقربه‌های ساعت در نظر گرفته شود، آنگاه همه‌ی نقطه‌ها در سمت چپ پاره‌خط‌های روی پوشش خواهند بود. به این ترتیب، نخستین راه حل به این صورت است که باید همه‌ی پاره خط‌های ممکن در مجموعه نقطه‌ها را بررسی کرده و ببینیم نقطه‌های دیگر، نسبت به هر پاره‌خط چه وضعی دارند؛ پاره‌خط‌هایی روی پوشش خواهند بود که همه‌ی نقطه‌های دیگر در سمت راست آن‌ها باشند. مراحل این الگوریتم در ادامه آمده است.

۱- ساخت مجموعه‌ی تهی یال‌های E $E \leftarrow \phi$

۲- برای هر زوج مرتب از نقطه‌های مجزا (p,q) در P ($\forall (p,q) \in P \times P, p \neq q$):

۲-۱- $valid \leftarrow true$

۲-۲- برای همه‌ی نقطه‌های r در P به جز دو نقطه p و q ($\forall r \in P, r \neq p, r \neq q$)

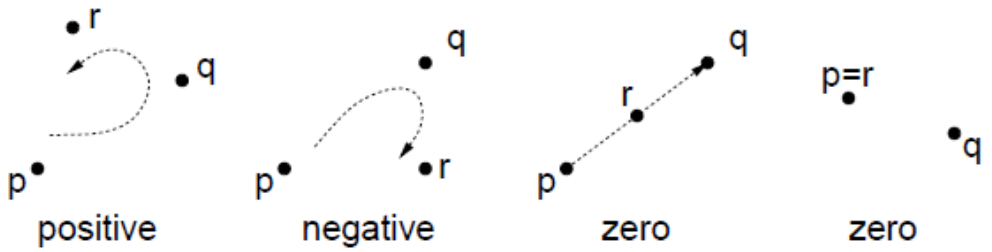
۲-۲-۱- اگر r در سمت چپ خط جهت‌دار از p به q قرار داشت.

۲-۲-۱-۱- $valid \leftarrow false$

۲-۳- اگر مقدارvalid برابر true بود آن‌گاه یال جهت‌دار $\overrightarrow{pq}$ به مجموعه یال‌ها E افزوده شود.

۳- از روی مجموعه یال‌های E مجموعه مرتب از گره‌ها در L ساخته شده و به عنوان جواب برگردانده می‌شود.

یکی از بخش‌های به ظاهر دشوار در پیاده‌سازی الگوریتم بالا، در ۲-۲-۱ است. در اینجا از بحث بردارها در ریاضی کمک گرفته می‌شود که برای سه نقطه (p,q,r) در صفحه، جهت در شکل ۶,۵ نشان داده شده است. دقت شود که ترتیب این سه نقطه (جهت بردارها) در اینجا مهم است.



شکل ٦,٥. جهت‌های سه نقطه در صفحه نسبت به هم

اگر چرخش در جهت عکس حرکت عقربه‌های ساعت باشد، نتیجه مثبت است (r در سمت چپ $\overrightarrow{pq}$ است) و اگر چرخش در جهت حرکت عقربه‌های ساعت باشد، نتیجه منفی است. اگر سه نقطه روی یک خط باشند یا دو نقطه روی هم باشند نتیجه صفر است. برای به دست آوردن نتیجه، باید این دترمینان محاسبه شود:

$$orient(p,q,r) = \det\begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}$$

در فضای یک بعدی، جهت برای دو عدد p و q به صورت تفاضل $orient(p,q) = q - p$ تعریف می‌شود که همان ترتیب عددها است. اگر $p < q$ باشد نتیجه مثبت است و اگر $p > q$ باشد، نتیجه منفی است و اگر برابر باشند، نتیجه صفر است. به این ترتیب سه عملگر $=,>,<$ در فضای یک بعدی تعریف شد. در فضای دو بعدی نیز دترمینان داده شده در بالا همین نقش را بازی می‌کند و به نوعی ترتیب تعریف می‌کند.

در فضای سه بعدی نیز برای چهار نقطه همین تعریف انجام می‌شود و دترمینان مشابهی (البته برای ماتریس ۴ در ۴) به کار برده می‌شود. در فضای سه بعدی نتیجه‌ی مثبت به معنای جهت چرخش انگشتان دست راست و نتیجه منفی به معنای جهت چرخش انگشتان دست چپ (در فیزیک الکتریسیته همانند این کار برای جهت میدان انجام می‌شود) و نتیجه صفر به معنای این است که چهار نقطه در یک صفحه هستند. در حالت کلی در فضای d بعدی نیز برای d+1 نقطه همین تعریف انجام می‌شود.

نکته بعدی در این الگوریتم در خط ۳ دیده می‌شود و این که چگونه از روی مجموعه یال‌های E، مجموعه مرتب از گره‌ها در L ساخته شود. یکی از یال‌های درون مجموعه یال‌ها را به دلخواه انتخاب می‌کنیم و مبدأ آن را (یال‌ها جهت‌دار هستند) به عنوان نخستین نقطه درون مجموعه گره‌ها می‌گذاریم؛ سپس مقصد را به مجموعه گره‌ها می

افزاییم و پس از آن دنبال یالی می‌گردیم که نقطه مبدأ آن، نقطه مقصد یالی باشد که هم اکنون دو نقطه آن را درون مجموعه گره‌ها گذاشته‌ایم. برای یال یافت شده نیز نقطه مقصد را در مجموعه گره‌ها می‌گذاریم و دنبال یال دیگری می‌گردیم که نقطه مبدأ آن، نقطه مقصد آخرین یالی باشد که نقاط آن را در مجموعه گره‌ها گذاشته‌ایم. این کار را ادامه می‌دهیم تا تنها یک یال در مجموعه یال‌ها باقی بماند و نیازی به افزون این یال پایانی نیست زیرا مبدأ آن پیش از این در یال پیشین گذاشته شده و مقصد آن نخستین گره‌ای است که در مجموعه‌ی گره‌ها گذاشته‌ایم و بدین ترتیب مجموعه گره‌ها آماده می‌شود.

پیچیدگی زمانی: در دستور ۲ باید همه زوج نقطه‌ها بررسی شوند و بنابراین (n-1) n حالت برای n نقطه خواهد داشت و دستور ۲-۲ درون دستور ۲ به ازای n-2 نقطه اجرا می‌شود؛ بنابراین مرتبه دستور ۲ برابر با n^3 است. یک پیاده‌سازی ساده از دستور ۳، مرتبه n^2 خواهد داشت. بنابراین الگوریتم از مرتبه n^3 خواهد بود.

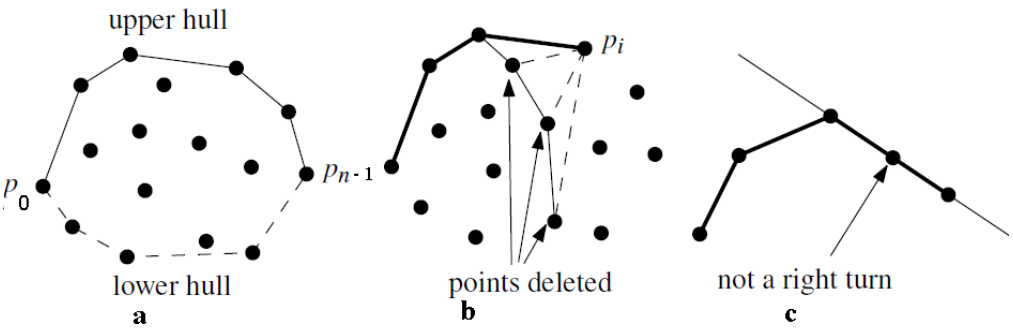
اگر در نقطه‌های ورودی مسأله، نقاطی باشد که سه نقطه (یا بیشتر) روی یک خط باشند، آنگاه الگوریتم درست کار نمی‌کند. در حالتی که سه نقطه بر روی یک خط باشند، باید فقط دو نقطه‌ی دو سر یال در نظر گرفته شوند و نقطه میانی (یا نقطه‌های میانی) در نظر گرفته نشوند. در این حالت دترمینان داده شده صفر خواهد بود و که باعث کوچکتر شدن بُعد مسأله از دیدگاه ریاضی می‌شود. بنابراین در بررسی شرط دستور ۲-۲-۱ (اگر نقطه r در سمت چپ دو نقطه دیگر باشد) این شرط نیز افزوده شود که اگر نقطه روی ادامه خط بود (دترمینان برابر صفر بود و نقطه آن در میان پاره خط pq قرار نداشت، بلکه در ادامه آن بود)، آنگاه باز مقدار valid برابر false گذاشته شود.

پایداری: اگر مختصات نقطه‌ها، عددهای اعشاری باشد این امکان وجود دارد که در نتیجه گرد شدن عددهای اعشاری در محاسباتی که انجام شده است (یا می‌شود)، مختصات نقطه‌ها از عدد اصلی خود متفاوت باشند. اگر سه نقطه q،p و r بسیار به هم نزدیک باشند و نقطه‌های دیگر با فاصله نسبت به آنها در سمت راستشان باشند، آنگاه این امکان هست که الگوریتم به اشتباه r را در سمت راست p و q در نظر بگیرد و p را در سمت راست r و q در نظر بگیرید و همچنین q را در سمت راست r و q در نظر بگیرد؛ در این حالت الگوریتم هر سه یال نامبرده را می پذیرد در حالی که نادرست است و روشن است که در هندسه چنین چیزی شدنی نیست ولی به دلیل خطای گرد گردن در محاسبات این وضعیت پیش آمده است؛ به این ترتیب الگوریتم در مرحله پایانی که در حال تبدیل یال‌ها به گره‌ها است، به گره‌ای می‌رسد که از آن دو یال بیرون آمده است. وضعیت بدتر درست عکس این حالت پیش می‌آید که هر کدام از نقطه‌ها سمت چپ دو نقطه دیگر در نظر گرفته می‌شوند و الگوریتم هر سه یال را حذف می‌کند و به این ترتیب الگوریتم در مرحله پایانی که در حال تبدیل یال‌ها به گره‌ها است به گره‌ای می‌رسد که از آن هیچ یالی بیرون نیامده است و در الگوریتم دچار شکست شده و کار مناسبی نمی‌تواند انجام دهد. بنابراین گرچه الگوریتم درست است ولی در حالت‌های ویژه‌ای در مختصات حقیقی نقطه‌ها پایداری ندارد؛ درست بودن الگوریتم با این فرض است که محاسبات عددی انجام شده بدون هیچ مشکلی باشند و محاسبات کاملاً دقیق و بدون گرد کردن باشند.

۵-٤-۲- الگوریتم کارآ و پایدار

در الگوریتم پیشین هیچ گونه بهینه‌سازی انجام نشده بود و راه حل هندسی، مستقیم به الگوریتم تبدیل شده بود. در این بخش یک الگوریتم افزایشی توضیح داده می‌شود که نقطه‌های ورودی مسأله، یکی یکی به الگوریتم داده شده

و نتیجه پس از افزوده شدن هر نقطه تغییر داده می‌شود. نخست، نقطه‌ها بر پایه مختصات x خود از چپ بـه راسـت مرتب می‌شوند و در صورت برابر بودن x برای دو یا چند نقطه از آنها، بر پایه مختصات y مرتب می‌شوند.
با توجه به این که نقطه‌ها از چپ به راست مرتب شده‌اند نخستین نقطه (p_0)، سمت چپ‌ترین نقطه در میان نقطه‌ها هست و بنابراین حتما در مجموعه نقطه‌های پوشش محدب خواهد بود. بنابراین اگر این نقطه آغاز یافتن دیگر نقطه های روی پوشش محدب باشد و حرکت به سمت راست انجام شود (در جهت عقربه‌های ساعت)، آنگاه دیگر نقطه‌ها روی پوشش محدب در بخش بالایی را می‌توان یافت. در صورتی نقطه‌ای به این نیمـه بـالایی از پوشـش محـدب افزوده می‌شود که جهت حرکت به سمت راست باشد (دترمینال سه نقطه متوالی که نقطه‌ی تازه افزوده شده، نقطـه سوم است) و اگر جهت راست نبود آنگاه باید آن قدر به عقب برگردیم و نقطه‌های درون فهرست مرتب (بیشتر شبیه پشته با آن رفتار می‌کنیم) بالایی را از آن برمی‌داریم که نقطه جدید با دو نقطه روی لیست مرتب جهت راست داشته باشند یا این که فقط یک نقطه در مجموعه باقی بماند که این نقطه جدید نیز به عنوان نقطه دوم به لیسـت افـزوده می‌شود. شکل ۷,۵ چگونگی انجام این کار را نشان می‌دهد.



شکل ۷,۵. چگونگی اجرای الگوریتم کارآی یافتن پوشش محدب

به همین روش نقطه پایانی (p_{n-1}) در فهرست نقطه‌های ورودی سمت راست‌ترین نقطه در مجموعه نقطه‌های مرتب شده ورودی است. بنابراین درون نقطه‌های نهایی تشکیل‌دهنده پوشش محدب خواهد بود. اگر از این نقطه به سمت چپ حرکت انجام شود (عکس جهت عقربه‌های ساعت)، نقطه‌های روی بخش پایینی پوشـش محـدب یافـت مـی شوند. با ترکیب کردن مناسب بخش بالایی و پایینی یافت شده پوشش محدب به دست می‌آید. الگوریتم ایـن روش در ادامه آورده شده است.

۱- مرتب‌سازی نقطه‌ها بر حسب مختصات x آنها و در صورت مساوی بودن x، مرتب‌سازی بر حسب y
۲- قرار دادن نقطه‌هایp_0 و p_1 به ترتیب در مجموعه مرتب L_{upper}
۳- برای i=2 **تا** i=n-1 این مراحل را انجام می‌دهیم:
۳-۱- نقطه p_i در پایان L_{upper} گذاشته شود.
۳-۲- تا هنگامی که بیشتر از دو نقطه در L_{upper} هست **و** سه نقطه پایانی در L_{upper} جهت سـمت راسـت ندارند:
۳-۲-۱- نقطه میانی از این سه نقطه پایانی از درون L_{upper} برداشته شود.

۴- نقطههای p_{n-1} و p_{n-2} به ترتیب در L_{lower} گذاشته شود.
۵- برای i=n-3 تا 0 مرحلههای زیر انجام شود:
۵-۱- نقطه p_i در پایان L_{lower} گذاشته شود.
۵-۲- تا هنگامی که بیشتر از دو نقطه در L_{lower} هست **و** سه نقطه پایانی در L_{lower} جهت سمت راست ندارند:
۵-۲-۱- نقطه میانی از این سه نقطه پایانی از درون L_{lower} برداشته شود.
۶- نقطههای p_0 و p_{n-1} از L_{lower} برداشته شود زیرا در L_{upper} هستند.
۷- نخست، نقطههای L_{upper} و سپس نقطههای L_{lower} به ترتیب در فهرست پایانی L گذاشته شوند.

پیچیدگی زمانی: مرتبسازی میتواند در O(n log(n)) انجام شود. حلقه ۳ از مرتبه O(n) است. تعداد اجرای دستور حذف در حلقه ۳-۲ محدود به n است و هر نقطه فقط یک بار برداشته میشود، پس مرتبه یافتن پوشش بالایی از O(n) است. برای پوشش پایینی نیز همین وضعیت وجود دارد و در مجموع با توجه به مرتبسازی، مرتبه الگوریتم O(n log(n)) است.
پایداری: اگر مشکل گرد شدن در محاسبات عددهای حقیقی در مختصات نقطهها پیش آید، آنگاه نقطه یا نقطه هایی که نباید در مجموعه باشد درون آن گذاشته میشود یا برعکس، نقطه یا نقطههایی که باید باشد از درون آن برداشته میشود؛ با این همه، الگوریتم همواره مجموعهای از نقطهها را در پاسخ میآورد که پوشش محدب را اغلب ایجاد میکنند؛ الگوریتم در هیچ کدام از مرحلهها، بدون دلیل متوقف نمیشود و به کار خود تا پایان ادامه میدهد. مشکل دیگر هنگامی رخ میدهد که در شناسایی جهت راست به مشکل بخورد؛ این مشکل زمانی پیش میآید که دقت محاسبات کمتر یا نزدیک به کوچکترین بخشهای اعشار مختصات نقطههای ورودی باشد. برای پرهیز از چنین وضعیتی نیز میتوان نقطههایی که مختصات بسیار نزدیکی نسبت به دقت محاسبات دارند، یکی در نظر گرفت و فقط یکی از آنها را در ورودی گذاشت.
این الگوریتم می تواند برای به دست آوردن پوشش محدب بر روی چندین پارهخط نیز به کار رود و کافی است که نقطههای پایانی هر پارهخط به عنوان ورودی الگوریتم داده شود تا پوشش محدب بر روی پارهخطها به دست آید.

۵-۵- ترکیبیات

ترکیبیات دستهای از روشهای ریاضی برای نحوه شمارش است. در مسابقات برنامهنویسی معمولا سوالات زیادی در مورد نحوه شمارش اشیا مطرح میشود. به همین دلیل آشنایی با این مبحث یکی از پیشنیازهای مهم برای شرکت در مسابقات برنامهنویسی است.
به خاطر داشته باشید که معمولا حل مسائل ترکیبیاتی وابستگی شدیدی به توانایی ایدهپردازی و خلاقیت فرد دارد. اگر بتوانید به یک مسأله ترکیبیاتی از زاویه درست نگاه کنید، حل آن به فرآیند بسیار سادهای تبدیل میشود. در طرف مقابل، اگر سعی کنید مسأله را بدون تحلیل کامل و تنها از طریق پیادهسازی یک روش پیچیده حل کنید، علاوه بر شکست خوردن در حل مسأله، به احتمال زیاد زمان زیادی در مسابقه را از دست خواهید داد. به همین دلیل، توانایی حل مسائل ترکیبیاتی عامل بسیار تاثیرگذاری در موفقیت در مسابقات برنامهنویسی محسوب میشود.

در این بخش به معرفی تکنیک‌های پرکاربرد در شاخه ترکیبیات پرداخته می‌شود.

۵-۵-۱- تکنیک‌های شمارشی اصلی

در این قسمت به معرفی تکنیک‌های ساده‌ی ترکیبیات برای شمارش اشاره می‌کنیم. در صورتی که با قوانین اصلی ترکیبیات برای شمارش آشنا هستید، می‌توانید از مطالعه این قسمت صرف نظر کنید.

قانون ضرب: قانون ضرب بیان می‌کند که اگر مجموعه A شامل $|A|$ حالت متفاوت و مجموعه B شامل $|B|$ حالت متفاوت باشد، ترکیب یکی از حالت‌های مجموعه A با یکی از حالت‌های مجموعه B دارای $|A|\times|B|$ حالت متفاوت خواهد بود. به عنوان مثال، کسی که دارای ۳ پیراهن و ۴ شلوار باشد، به ۳×۴=۱۲ حالت می‌تواند لباس بپوشد.

قانون جمع: قانون جمع بیان می‌کند که اگر مجموعه A شامل $|A|$ حالت متفاوت و مجموعه B شامل $|B|$ حالت متفاوت باشد، رخ دادن یکی از حالت‌های یکی از این دو مجموعه دارای $|A|+|B|$ حالت متفاوت خواهد بود. به عنوان مثال، کسی که دارای ۳ پیراهن و ۴ شلوار باشد، به ۳+۴=۷ حالت می‌تواند یکی از لباس‌های خود را برای شستشو به خشکشویی ببرد.

رابطه‌ی شمول و عدم شمول: قانون جمع در واقع حالت خاصی از رابطه‌ی عمومی‌تری است که در آن ممکن است دو مجموعه A و B با هم اشتراک داشته باشند. در این صورت با استفاده از رابطه‌ی شمول و عدم شمول تعداد حالت‌های رخ دادن یکی از حالت‌های یکی از این دو مجموعه به این صورت محاسبه می‌شود:

$$|A\,\mathrm{Y}\,B|=|A|+|B|-|A\,\mathrm{I}\;B|$$

به عنوان مثال، فرض کنید مجموعه A نشان‌دهنده رنگ‌های پیراهن‌های یک شخص و مجموعه B نشان‌دهنده رنگ‌های شلوارهای او باشد. واضح است که از طریق رابطه‌ی شمول و عدم شمول به سادگی می‌توان تعداد رنگ‌های متمایز لباس‌های این شخص را محاسبه کرد. دلیل تفاضل عبارت $|A\,\mathrm{I}\;B|$ در رابطه بالا این است که جمع اندازه‌ی دو مجموعه A و B باعث می‌شود که اشتراک این دو مجموعه دو بار شمرده شود. به همین دلیل باید این مقدار را یک بار از مقدار محاسبه شده کم کرد تا مطمئن شویم که هرکدام از اعضای این دو مجموعه فقط یک بار شمرده شده‌اند. رابطه‌ی شمول و عدم شمول را به سادگی می‌توان به تعداد بیشتری از مجموعه‌ها بسط داد. این رابطه برای سه مجموعه به این صورت محاسبه می‌شود:

$$|A\,\mathrm{Y}\,B\,\mathrm{Y}\,C|=|A|+|B|+|C|-|A\,\mathrm{I}\;B|-|A\,\mathrm{I}\;C|-|B\,\mathrm{I}\;C|+|A\,\mathrm{I}\;B\,\mathrm{I}\;C|$$

یکی از تکنیک‌های پرکاربرد برای شمارش اعضای یک مجموعه، برقرار کردن **تناظر یک به یک** بین اعضای آن مجموعه و یک مجموعه‌ی (احتمالا شناخته شده‌ی) دیگر است. به این ترتیب با شمارش تعداد اعضای مجموعه‌ی دوم، تعداد اعضای مجموعه‌ی مورد نظر نیز به دست می‌آید.

به عنوان مثال، با شمردن تعداد شلوارهای درون یک کلاس، می‌توان تعداد دانش‌آموزان درون آن کلاس را به دست آورد. این ادعا درست است زیرا هر دانش‌آموز در کلاس دقیقا یک شلوار پوشیده است و هر شلوار توسط یک دانش‌آموز پوشیده شده است (پس تناظر یک به یک بین این دو مجموعه برقرار است).

در ادامه به توضیح چند تکنیک ساده‌ی دیگر برای شمارش پرداخته می‌شود:

جایگشت: جایگشت به معنی تعداد حالت‌های قرارگیری n شیء در کنار یکدیگر است، به طوری که هر شیء دقیقا یک بار ظاهر شود. این تعداد برای n شیء برابر $n!=\prod_{i=1}^{n} i$ است. به عنوان مثال، $3!=6$ جایگشت سه شیء را می‌توان به صورت 123، 132، 213، 231، 312 و 321 در نظر گرفت. تعداد جایگشت‌های ۱۰ شیء، برابر $10!=3,628,800$ است که عدد بسیار بزرگی است و مدیریت این تعداد حالت برای رایانه مستلزم زمان بسیار زیادی است. چنین وضعیتی در بسیاری از مسائل ترکیبیاتی می‌تواند دچار مشکل کند.

زیرمجموعه: انتخاب تعدادی شیء از میان n شیء موجود، زیرمجموعه‌ای از مجموعه‌ی آن n شیء ایجاد می کند. هر مجموعه n عضوی دارای 2^n زیرمجموعه متمایز است. به عنوان مثال، یک مجموعه 3 دارای $2^3=8$ زیرمجموعه است که عبارتند از: 1، 2، 3، 12، 13، 23، 123 و مجموعه‌ی تهی. یک مجموعه‌ی 20 عضوی دارای $2^{20}=1,048,576$ زیرمجموعه‌ی متمایز است.

رشته‌ها: یک رشته دنباله‌ای از نویسه‌هایی است که با تکرار می‌توانند ظاهر شوند. با داشتن m نویسه، می‌توان m^n رشته به طول n ایجاد کرد. به عنوان مثال، $2^3=8$ رشته به طول 3 که با دو نویسه قابل ایجاد شدن هستند، عبارتند از: 000، 001، 010، 011، 100، 101، 110 و 111.

۵-۵-۲- رابطه‌های بازگشتی

با استفاده از رابطه‌های بازگشتی می‌توان به سادگی بسیاری از ساختارهای بازگشتی را شمارش کرد. ساختارهای بازگشتی شامل درخت‌ها، لیست‌ها، رابطه‌های خوش تعریف و الگوریتم‌های تقسیم و غلبه می‌شوند.

یک رابطه بازگشتی رابطه‌ای است که توسط خودش تعریف شده است. بسیاری از ساختارها را می‌توان به سادگی توسط رابطه‌های بازگشتی تعریف کرد. هر چندجمله‌ای را می‌توان به صورت بازگشتی نشان داد:

$$a_n = a_{n-1} + 1, a_1 = 1 \rightarrow a_n = n$$

نکته جالب این که حتی بسیاری از توابعی که به سادگی توسط روش‌های عادی قابل نمایش نیستند، با رابطه‌های بازگشتی تعریف می‌شوند، مثلا:

$$a_n = na_{n-1}, a_1 = 1 \rightarrow a_n = n!$$

معمولا برای بسیاری از مسائل ترکیبیاتی می‌توان رابطه‌های بازگشتی ارائه کرد. لازم به ذکر است که حل روابط بازگشتی با تکنیک‌های ریاضی کار بسیار راحتی نیست؛ با این حال همانطور که در ادامه خواهیم دید، پیاده‌سازی رابطه‌های بازگشتی در برنامه‌ها کار بسیار ساده‌ای است.

۵-۵-۳- ضریب‌های دوجمله‌ای

یکی از دسته‌های بسیار مهم از مسائل شمارشی، ضریب‌های دوجمله‌ای هستند. منظور از $\binom{n}{k}$ تعداد حالت‌های انتخاب k شیء از بین n است که به صورت ترکیب k از n خوانده می‌شود. برای آشنایی با مفهوم ترکیب، به مثال‌های بعدی توجه کنید.

کمیته: به چند طریق می‌توان کمیته‌ای k نفره از بین n نفر تشکیل داد؟ طبق تعریف، جواب برابر $\binom{n}{k}$ است.

مسیرهای موجود در یک شبکه: به چند طریق می‌توان از گوشه‌ی بالایی سمت چپ یک شبکه $n\times m$ به گوشه‌ی پایینی سمت راست رفت، به طوری با هر قدم یک واحد به سمت راست یا پایین حرکت کنیم؟ واضح است که در چنین شبکه‌ای هر مسیر شامل $n+m$ قدم خواهد بود (n قدم به سمت پایین و m قدم به سمت راست). هر مسیر با مجموعه‌ی متفاوتی از حرکات به سمت پایین، نشان‌دهنده یک مسیر متمایز است. بنابراین در کل $\binom{n+m}{n}$ مسیر متمایز وجود دارد.

ضریب‌های دوجمله‌ای $(a+b)^n$: به این عبارت توجه کنید:

$$(a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3$$

با کمی دقت متوجه خواهید شد که ضریب عبارت a^kb^{n-k} در دوجمله‌ای $(a+b)^n$ برابر $\binom{n}{k}$ است.

مثلث خیام-پاسکال: مثلث خیام-پاسکال یک آرایه‌ی مثلثی از ضریب‌های دوجمله‌ای است. مقدار هر عدد درون این مثلث از جمع دو عدد بالایی آن تشکیل شده و سطر اول دارای عدد 1 است. در ادامه شش سطر اول این مثلث نشان داده شده است:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
```

با کمی دقت متوجه می‌شوید که اعداد درون این مثلث در واقع همان ضرایب دوجمله‌ای را نشان می‌دهند. یکی از نکات جالب دیگر در مورد این مثلث این است که روابط جالبی بین اعداد درون مثلث را نشان می‌دهد. یکی از این نکات این است که جمع اعداد سطر $(n+1)$ام این مثلث برابر 2^n است.

چطور می‌توان مقدار ضریب‌های دوجمله‌ای را حساب کرد؟ می‌دانیم که $\binom{n}{k} = \frac{n!}{k!(n-k)!}$؛ بنابراین می‌توان مقدار $\binom{n}{k}$ را با محاسبه‌ی مقدار فاکتوریل‌ها به دست آورد. با این حال این روش یک ایراد بسیار بزرگ دارد: مقدار هرکدام از فاکتوریل‌ها ممکن است از مقدار مجاز اعداد صحیح بیرون بزند (حتی اگر مقدار نهایی $\binom{n}{k}$ در محدوده‌ی مجاز اعداد صحیح قرار بگیرد).

یک روش پایدارتر برای محاسبه‌ی مقدار $\binom{n}{k}$ استفاده از رابطه‌ی بازگشتی زیر است که در مثلث خیام–پاسکال نهفته است:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

هر رابطه‌ی بازگشتی به حالت پایه نیاز دارد. برای مشخص کردن حالت پایه باید ببینیم که کدام یک از ضریب‌های دوجمله‌ای را بدون محاسبه می‌توانیم تعیین کنیم. عبارت اول مجموع بالا نهایتا ما را به عبارت $\binom{n-k}{0}$ می‌رساند و واضح است که تعداد حالت‌های انتخاب صفر شیء از یک مجموعه برابر یک است (مجموعه‌ی تهی). عبارت دوم مجموع بالا ما را به عبارت $\binom{k}{k}$ می‌رساند. مشخص است که تنها به یک طریق می‌توان k شیء را از یک مجموعه به اندازه‌ی k انتخاب کرد (خود مجموعه). به این ترتیب تعریف رابطه‌ی بازگشتی محاسبه ضریب‌های دوجمله‌ای کامل شده است. در ادامه یک نمونه از پیاده‌سازی این رابطه‌ی بازگشتی نشان داده شده است:

```
#define MAXN 100 /* largest n or m */
long binomial_coefficient(int n, int m) {
    int i, j; /* counters */
    long bc[MAXN][MAXN]; /* table of binomial coefficients */
    for (i = 0; i <= n; i++)
        bc[i][0] = 1;
    for (j = 0; j <= n; j++)
        bc[j][j] = 1;
    for (i = 1; i <= n; i++)
        for (j = 1; j < i; j++)
            bc[i][j] = bc[i - 1][j - 1] + bc[i - 1][j];
    return bc[n][m];
}
```

نمونه کد نشان داده شده از روش برنامه‌نویسی پویا برای پیاده‌سازی رابطه‌ی بازگشتی ضریب‌های دوجمله‌ای استفاده کرده است.

۵-۶- نظریه‌ی اعداد

نظریه‌ی اعداد یکی از شاخه‌های جذاب ریاضیات است. اثبات اقلیدس که تعداد اعداد اول بینهایت است، همانند دو هزار سال پیش همچنان واضح و جالب است. اعداد صحیح مفاهیم بسیار مهمی در ریاضیات هستند و به همین دلیل کشف خصوصیات جدید در مورد آنها در واقع معادل با کشف حقیقتی جدید در مورد دنیای طبیعی است. از این رو مطالعه‌ی اعداد صحیح کار بسیار جالبی است.

مدت زیادی است که از رایانه‌ها برای تحقیقات نظریه‌ی اعداد استفاده می‌شود. انجام محاسبات نظریه‌ی اعداد روی اعداد صحیح بزرگ نیاز به کارایی بالایی دارد. خوشبختانه در این زمینه الگوریتم‌های هوشمندانه‌ای برای کمک به ریاضیدانان وجود دارد.

مسائل نظریه‌ی اعداد به علت سادگی در تعریف، در مسابقات برنامه‌نویسی مکررا استفاده می‌شوند. با این که این شاخه محدوده‌ی وسیعی از مباحث را شامل می‌شود، برای شرکت در مسابقات برنامه‌نویسی داشتن پیش زمینه‌ی کوچکی از اطلاعات مانند اعداد اول و بزرگترین مقسوم علیه مشترک (ب.م.م) کافی است. در این فصل به بررسی چند نمونه از روش‌های رایانه‌ای برای حل مسائل نظریه‌ی اعداد می‌پردازیم.

۵-۶-۱- اعداد اول

یک عدد اول عدد صحیح $p > 1$ است که فقط بر 1 و خودش بخش‌پذیر است. به عبارت دیگر، اگر p یک عدد اول باشد، عبارت $p = a \times b$ به ازای $a \le b$ به این معناست که $a = 1$ و $b = p$. کوچکترین ده عدد اول عبارتند از: 2، 3، 5، 7، 11، 13، 17، 19، 23 و 29. به هر عددی که اول نباشد، مرکب گفته می‌شود.

اهمیت اصلی اعداد اول به این دلیل است که هر عدد صحیح را می‌توان تنها به یک طریق به صورت حاصل ضرب مجموعه‌ای از اعداد اول نوشت. به عنوان مثال، $105 = 3 \times 5 \times 7$ و $32 = 2 \times 2 \times 2 \times 2 \times 2$. به اعداد اولی که حاصل ضرب آنها عدد n را تولید می‌کند، عامل‌های اول n گفته می‌شود.

۵-۶-۱-۱- یافتن اعداد اول

ساده‌ترین راه برای بررسی اول بودن یک عدد مانند n از طریق تقسیم‌های متوالی است. برای این کار باید از کوچکترین مقسوم علیه احتمالی شروع کرده و بخش‌پذیری عدد مورد نظر بر تمامی مقسوم علیه‌های احتمالی را بررسی کنیم. از آنجایی که 2 تنها عدد اول زوج است، پس از اطمینان حاصل کردن از زوج نبودن عدد مورد نظر، فقط کافیست اعداد فرد را به عنوان مقسوم علیه‌های احتمالی در نظر بگیریم. علاوه بر این، اگر n بر هیچکدام از مقسوم علیه‌های احتمالی کوچکتر از $\sqrt{n}$ بخش‌پذیر نبود، یعنی n یک عدد اول است (چرا؟).

یافتن تمامی عامل‌های اول یک عدد شامل یافتن تعداد تکرارهای هرکدام از عامل‌ها نیز می‌شود. پیاده‌سازی این فرآیند در ادامه نشان داده شده است:

```
#include<iostream>
void prime_factorization(long x) {
```

```
    long i; /* counter */
    long c; /* remaining product to factor */
    c = x;
    while ((c % 2) == 0) {
        printf("%ld\n", 2);
        c = c / 2;
    }
    i = 3;
    while (i <= (sqrt(c) + 1)) {
        if ((c % i) == 0) {
            printf("%ld\n", i);
            c = c / i;
        }
        else
            i = i + 2;
    }
    if (c > 1)
        printf("%ld\n", c);
}
```

یکی از راه‌های رسیدن به عملکرد بالاتر تابع prime_factorization، این است که فراخوانی تابع sqrt(c) را تنها زمانی انجام دهیم که مقدار عددی c تغییر می‌کند. روش‌های متفاوت دیگری نیز برای یافتن عامل‌های اول یک عدد وجود دارد؛ با این حال تابع معرفی شده در این بخش عملکرد خوبی دارد و برای استفاده در مسابقات برنامه‌نویسی مناسب است. به همین دلیل در این قسمت به معرفی سایر روش‌ها نمی‌پردازیم.

۵-٦-۱-۲- شمارش اعداد اول

یک سوال مهم در مورد اعداد اول این است که آیا تعداد آنها محدود است؟ مشخص است که هرچه محدوده‌ی اعداد بزرگتر را بررسی کنیم، باید تعداد اعداد اول کاهش یابد. ولی آیا می‌توان ادعا کرد که از یک عدد مشخصی به بعد، عدد اول دیگری وجود ندارد؟ جواب این سوال منفی است. اقلیدس از طریق برهان خلف این سوال را جواب داده است. با این که دانستن پاسخ این سوال برای شرکت در مسابقات برنامه‌نویسی نیاز نیست، ولی به دلیل جالب بودن آن، در این قسمت اثبات پاسخ این سوال را شرح می‌دهیم.

فرض کنید که تعداد متنهاهی از اعداد اول وجود دارد. این اعداد را با $\{p_1,...,p_n\}$ نشان می‌دهیم. حال عدد دیگری را به صورت $m=1+\prod_{i=1}^{n} p_i$ تعریف می‌کنیم. از آنجایی که m به وضوح از تمامی اعداد اول قبلی بزرگتر است، پس باید یک عدد مرکب باشد. بنابراین m باید بر یکی از اعداد اول مجموعه $\{p_1,...,p_n\}$ بخش‌پذیر باشد. ولی این یک تناقض است، زیرا واضح است که با تقسیم m بر هرکدام از اعداد مجموعه‌ی $\{p_1,...,p_n\}$، باقیمانده برابر 1 خواهد بود. به این ترتیب $\{p_1,...,p_n\}$ نمی‌تواند مجموعه‌ی تمامی اعداد اول باشد، زیرا m نیز یک عدد اول است. به این ترتیب ثابت کردیم که مجموعه اعداد اول متناهی نیست.

۵-۶-۲- بخش‌پذیری

تمرکز اصلی نظریه‌ی اعداد بر بخش‌پذیری اعداد صحیح است. می‌گوییم که عدد b عدد a را عاد می‌کند (و با $b \mid a$ نشان می‌دهیم)، اگر به ازای یک عدد صحیح k تساوی $a = bk$ برقرار باشد. در این حالت همچنین می‌توان گفت که b یک عامل برای a بوده یا a مضربی از b است. از این تعریف می‌توان به سادگی نتیجه گرفت که 1 کوچکترین مقسوم علیه هر عدد صحیح غیر صفر است.

چطور می‌توان تمامی مقسوم علیه‌های یک عدد صحیح را پیدا کرد؟ از قضیه‌ی اعداد اول به خاطر داریم که هر عدد صحیح n را می‌توان به صورت منحصر به فرد بر اساس حاصل ضرب عامل‌های اول آن نشان داد. بنابراین هرکدام از مقسوم علیه‌های n نیز نتیجه‌ی ضرب زیرمجموعه‌ای از عامل‌های اول آن هستند. مجموعه‌ی تمامی مقسوم علیه‌های یک عدد صحیح را می‌توان به سادگی از طریق یک الگوریتم عقب‌گرد محاسبه کرد.

۵-۶-۲-۱- بزرگترین مقسوم علیه مشترک (ب.م.م.)

از آنجایی که 1 هر عدد صحیحی را عاد می‌کند، کوچکترین مقسوم علیه مشترک هر جفت عدد صحیح a و b برابر 1 است. نکته مهمتر در مورد مقسوم علیه، بزرگترین مقسوم علیه مشترک دو عدد، یا ب.م.م.، یا $\gcd$ است. کسری مانند $\frac{a}{b}$ را در نظر بگیرید (مثلا $\frac{24}{36}$). برای به دست آوردن ساده‌ترین شکل این کسر، باید صورت و مخرج کسر را به ب.م.م. آنها تقسیم کرد (در این مثال ب.م.م. صورت و مخرج 12 است). اگر ب.م.م. دو عدد برابر 1 باشد، می‌گوییم که دو عدد نسبت به هم اول هستند.

اقلیدس برای یافتن ب.م.م. دو عدد صحیح روشی ارائه داد که شاید قدیمی‌ترین و جالب‌ترین الگوریتم طول تاریخ باشد. البته روش‌های ساده‌تری نیز برای پیدا کردن مقسوم علیه دو عدد وجود دارد. یکی از این روش‌ها بررسی بخش‌پذیری عدد دوم بر تمامی مقسوم علیه‌های عدد اول است. روش جالب‌تر می‌تواند به دست آوردن عامل‌های اول هر دو عدد، و سپس محاسبه‌ی حاصل ضرب عامل‌های اول مشترک آنها باشد. ولی هر دوی این روش‌ها نیاز به انجام محاسبات ریاضی زیادی دارند. به همین دلیل، الگوریتم اقلیدس برای حل این مسأله گزینه‌ی مناسب‌تری است.

الگوریتم اقلیدس روی دو مشاهده پایه‌گذاری شده است:

مشاهده‌ی اول: اگر $a \mid b$، آنگاه $\gcd(a,b) = b$. درستی این ادعا کاملا واضح است. زیرا $b \mid a$ به این معنی است که به ازای عدد صحیح k داریم: $a = bk$، و بنابراین $\gcd(bk,b) = b$.

مشاهده‌ی دوم: اگر $a = bt + r$ (t و r عدد صحیح هستند)، آنگاه $\gcd(a,b) = \gcd(b,r)$. چرا؟ طبق تعریف، $\gcd(a,b) = \gcd(bt+r,b)$. هر مقسوم علیه مشترک a و b حتما باید بر r نیز بخش‌پذیر باشد؛ زیرا bt بر هرکدام از مقسوم علیه‌های b بخش‌پذیر است.

الگوریتم اقلیدس بازگشتی است. به این صورت که در هر مرحله عدد بزرگتر را با باقیمانده‌ی حاصل از تقسیم آن بر عدد کوچکتر جایگزین می‌کند. به این ترتیب در هر مرحله یکی از پارامترها تقریبا نصف می‌شود و الگوریتم پیچیدگی زمانی لگاریتمی خواهد داشت. به عنوان مثال، دو عدد $a = 34398$ و $b = 2132$ را در نظر بگیرید. مراحل محاسبه‌ی ب.م.م. این دو عدد بر اساس الگوریتم اقلیدس در ادامه نشان داده شده است (منظور از $a\%b$ باقیمانده‌ی حاصل از تقسیم a بر b است):

$$\gcd(34398,2132)=\gcd(34398\%2132,2132)=\gcd(2132,286)$$
$$\gcd(2132,286)=\gcd(2132\%286,286)=\gcd(286,130)$$
$$\gcd(286,130)=\gcd(286\%130,130)=\gcd(130,26)$$
$$\gcd(130,26)=\gcd(130\%26,26)=\gcd(26,0)$$

بنابراین، $\gcd(34398,2132)=26$.

الگوریتم اقلیدس کاربردهای دیگری به جز بـه دسـت آوردن ب.م.م. دو عـدد نیـز دارد. از ایـن الگـوریتم همچنـین می‌توان برای پیدا کردن دو عدد صحیح x و y در این رابطه استفاده کرد:

$$a.x+b.y=\gcd(a,b)$$

می‌دانیم که $\gcd(a,b)=\gcd(b,a')$ به طوری که $a'=a-b\lfloor a/b\rfloor$. فرض کنید دو مقدار x' و y' را می‌دانیم به طوری که به صورت بازگشتی داشته باشیم:

$$b.x'+a'.y'=\gcd(a,b)$$

با جایگزینی مقدار a' در رابطه‌ی بالا خواهیم داشت:

$$b.x'+(a-b\lfloor a/b\rfloor).y'=\gcd(a,b)$$

به این ترتیب با مرتب کردن عبارت‌های رابطه‌ی بالا، مقدار نهایی x و y به دست می‌آید. برای اتمام طراحی این الگوریتم، نیاز به یک حالت پایه داریم. حالت پایه‌ی این الگوریتم به این صورت خواهد بود:

$$a.1+0.0=\gcd(a,0)$$

با اجرای این الگوریتم بر روی مثال قبل، به عبارت $34398\times 15+2132\times -242=26$ می‌رسیم. یک پیاده‌سازی از این الگوریتم در ادامه نشان داده شده است. تابع floor در این کد برای به دست آوردن جزء صحیح پایین یک عدد اعشاری استفاده شده است.

```
/* Find the gcd(p,q) and x,y such that p*x + q*y = gcd(p,q) */
long gcd(long p, long q, long *x, long *y)
{
    long x1, y1; /* previous coefficients */
    long g; /* value of gcd(p,q) */
    if (q > p)
        return gcd(q, p, y, x);
    if (q == 0) {
        *x = 1;
        *y = 0;
        return(p);
    }
    g = gcd(q, p%q, &x1, &y1);
    *x = y1;
    *y = x1 - floor(p / q) * y1;
    return g;
}
```

۵-۶-۲-۲- کوچکترین مضرب مشترک (ک.م.م.)

یک تابع مفید دیگر برای دو عدد صحیح، تابع کوچکترین مضرب مشترک، یا ک.م.م.، یا lcm است. ک.م.م. دو عدد صحیح برابر است با کوچکترین عدد صحیحی که بر هر دو عدد بخش‌پذیر است. به عنوان مثال، ک.م.م. دو عدد 24 و 36 برابر 72 است.

کوچکترین مضرب مشترک برای به دست آوردن همزمانی دو رویداد متناوب کاربرد دارد. به عنوان مثال، اولین زمان بعد از سال ۲۰۰۰ که انتخابات ریاست جمهوری (که هر ۴ سال یک بار برگزار می‌شود) با سرشماری همگانی (که هر ۱۰ سال یک بار انجام می‌شود) همزمان رخ می‌دهند، چه سالی است؟ این اتفاق هر ۲۰ سال یک بار رخ می‌دهد، زیرا $lcm(4,10)=20$.

این نکته کاملا واضح است که $lcm(x,y)\geq \max(x,y)$. به طور مشابه، از آنجایی که $x.y$ هم مضرب x و هم مضرب y است، می‌توان نتیجه گرفت که: $lcm(x,y)\leq x.y$. در واقع مقدار ک.م.م. دو عدد تنها زمانی از حاصلضرب آنها کوچکتر می‌شود که مجموعه‌ی عامل‌های اول آنها با یکدیگر اشتراک داشته باشند.

با توجه به گزاره‌های گفته شده و با استفاده از الگوریتم اقلیدس برای محاسبه‌ی ب.م.م.، می‌توان به روشی ساده برای محاسبه‌ی ک.م.م. دو عدد رسید. به این صورت که $lcm(x,y)=\frac{xy}{\gcd(x,y)}$.

۵-۷- محاسبات پیمانه‌ای

در برخی محاسبات ریاضی، نیازی به دانستن جواب نهایی یک رابطه نیست؛ بلکه به دست آوردن باقیمانده می‌تواند حلال مسأله باشد. به عنوان مثال، فرض کنید روز تولد شما در سال جاری در روز چهارشنبه باشد. تولد شما در سال آینده در چه روزی از هفته خواهد بود؟ شما می‌توانید جواب این سوال را با دانستن باقیمانده‌ی تقسیم تعداد روزهای بین دو روز تولد متوالی خود (که برابر ۳۶۵ یا ۳۶۶ روز است) بر عدد ۷ به دست آورید. به این ترتیب، اگر تعداد روزها برابر ۳۶۵ باشد، روز تولد شما در سال آینده پنجشنبه خواهد بود (زیرا باقیمانده تقسیم ۳۶۵ بر عدد ۷ برابر ۱ است).

کلید چنین روش‌هایی، محاسبات پیمانه‌ای است. همانطور که در مثال قبل نشان داده شد، این محاسبات می‌توانند ما را از سر و کار داشتن با اعداد بزرگ نجات داده و مستقیما به سمت پاسخ نهایی مسأله هدایت کنند.

در چنین محاسباتی به مقسوم علیه پیمانه نیز گفته می‌شود. برای سهولت در انجام این محاسبات لازم است که با عملگرهای ساده‌ی جمع، تفاضل، ضرب، تقسیم و تاثیر آنها بر پیمانه‌ها آشنا باشید.

- **جمع**: حاصل $(x+y)\%n$ چیست؟ این عبارت را می‌توان به این صورت ساده کرد تا دیگر نیازی به جمع بستن اعداد بزرگ نداشته باشیم:

$$(x\%n+y\%n)\%n$$

اگر فردی 12,345 تومان از مادرش و 9,467 تومان از پدرش بگیرد، چقدر پول خرد (کمتر از 1000 تومان) خواهد داشت؟

$$((12,345\%1000)+(9,467\%1000))\%1000=(345+467)\%1000=812$$

- **تفاضل**: تفاضل در واقع همان جمع با یک عدد منفی است. فرد مثال قبل، با خرج کردن 937 تومان، چه مقدار پول خرد خواهد داشت؟

$$((812\%1000)-(937\%1000))\%1000=-125\%1000=875$$

دقت کنید که در مثال بالا چگونه یک مقدار منفی را با اضافه کردن مضربی از پیمانه (n) تبدیل به یک عدد مثبت کردیم. در کل بهتر است که همیشه باقیمانده را بین 0 و $n-1$ نگه داریم تا مطمئن شویم که همیشه با کوچکترین باقیمانده‌ی ممکن کار می‌کنیم.

- **ضرب**: از آنجایی که ضرب در واقع چند عمل متوالی جمع است، می‌توان نوشت:

$$xy\%n=(x\%n)(y\%n)\%n$$

اگر فردی 172 ساعت کار کند و در هر ساعت 3300 تومان درآمد داشته باشد، در نهایت چه مقدار پول خرد خواهد داشت؟

$$((172\%1000)(3300\%1000))\%1000=51600\%1000=600$$

به این ترتیب، از آنجایی که عملگر توان نیز برابر چندین ضرب متوالی است، می‌توان نوشت:

$$x^y\%n=(x\%n)^y\%n$$

از آنجایی که توان یکی از عملگرهایی است که به سادگی می‌تواند تولید اعداد بسیار بزرگ تولید کند، محاسبات پیمانه‌ای در ساده‌سازی عباراتی که دارای توان هستند، بسیار موثرند.

- **تقسیم**: عملگر تقسیم نسبت به سه عملگر قبلی ساختار پیچیده‌تری دارد و در بخش بعد مورد بحث قرار خواهد گرفت.

از محاسبات پیمانه‌ای می‌توان در حل مسائل بسیار متنوعی استفاده کرد. در ادامه به چند نوع از این مسائل اشاره می شود.

پیدا کردن آخرین رقم: آخرین رقم عدد 2^{100} چیست؟ با کمی تفکر به سادگی می‌توان متوجه شد که امکان محاسبه مستقیم این مقدار توسط یک رایانه امکان‌پذیر نیست. ولی با استفاده از محاسبات پیمانه‌ای، می‌توان این سوال را با یک حساب سر انگشتی به سادگی حل کرد. چیزی که به دنبال آن هستیم، باقیمانده‌ی 2^{100} در پیمانه‌ی 10 است. با به توان رساندن‌های متوالی و ساده‌سازی باقیمانده، می‌توان این مقدار را به سادگی به دست آورد.

$$2^3\%10=8$$
$$2^6\%10=8\times8\%10=4$$
$$2^{12}\%10=4\times4\%10=6$$
$$2^{24}\%10=6\times6\%10=6$$
$$2^{48}\%10=6\times6\%10=6$$
$$2^{96}\%10=6\times6\%10=6$$
$$2^{100}\%10=2^{96}\times2^3\times2^1\%10=6\times8\times2\%10=6$$

الگوریتم رمزنگاری آر–اس–اِی: یکی از کاربردهای جالب محاسبات پیمانه‌ای در کار با اعداد بزرگ، در رمزنگاری کلید عمومی است که به آن الگوریتم رمزنگاری آر–اس–اِی گفته می‌شود. در این الگوریتم، پیغام ما با کد

کردن آن در قالب عدد m و به توان k رساندن آن انجام می‌شود که k همان کلید عمومی یا کلید رمزنگاری است. در نهایت باقیمانده‌ی تقسیم نتیجه بر n حساب می‌شود. از آنجایی که m، k و n همگی اعداد صحیح بزرگ هستند، محاسبه $m^k \% n$ با استفاده از روش‌های توضیح داده شده کار بسیار ساده‌ای است.

محاسبات مربوط به تقویم: همانطور که در مثال روز تولد در ابتدای این بخش نشان داده شد، محاسبات پیمانه ای برای به دست آوردن مشخصات یک روز خاص، و در مسائل دیگری که در مورد تعداد ساعت‌ها و مفاهیم مشابه هستند نیز کاربرد دارند.

تمرین‌ها

۱- با استفاده از رویه ccw که در قسمت ۵-۲ معرفی شد، برنامه‌ای بنویسید که توسط آن بتوان بررسی کرد که آیا مجموعه‌ای از نقاط در یک صفحه دو بعدی که به ترتیب ساعتگرد مرتب شده‌اند، تشکیل یک چندضلعی محدب را می‌دهند یا خیر. در خط اول ورودی تعداد راس‌های چندضلعی مشخص می‌شود. در ادامه در هر خط مختصات هرکدام از راس‌ها (به ترتیب ساعتگرد) مشخص می‌شود. در انتها در صورتی که چندضلعی مشخص شده محدب باشد، عبارت YES و در غیر این صورت عبارت NO در خروجی چاپ می‌شود.

Input	Output
4 0 0 0 1 1 1 1 0	YES

۲- روش دیگری برای محاسبه مساحت یک چندضلعی محدب ارائه دهید و سپس تابع ComputeArea را بر مبنای این روش پیاده‌سازی کنید (راهنمایی: از محاسبه مساحت زیر یک پاره‌خط استفاده کنید).

۳- همانطور که در بخش ۵-۴-۱ گفته شد، خطای گرد کردن اعداد اعشاری می‌تواند موجب بروز دو نوع اشتباه در محاسبه پوشش محدب مجموعه نقاط ورودی شود. این دو نوع اشتباه را با رسم شکل نشان دهید.

۴- برای به دست آوردن پوشش محدب بر روی تعدادی نقطه راه‌حل‌های دیگری نیز ارائه شده است. اگر نقطه p1، سمت راست‌ترین نقطه (بر پایه محور x) باشد آن گاه این نقطه بر روی پوشش محدب قرار دارد. اکنون یک خط عمودی در نظر بگیرید که از این نقطه می‌گذرد. اگر این خط در جهت عقربه‌های ساعت بچرخد، آن‌گاه به نقطه‌ی دیگری در مجموعه نقطه‌ها برخورد می‌کند که این نقطه نیز بر روی پوشش محدب قرار دارد. دوباره اگر یک خط عمودی بر روی این نقطه جدید گذرانده شود و در جهت عقربه‌های ساعت چرخانده شود، نقطه بعدی روی پوشش محدب به دست می‌آید؛ اگر این کار ادامه یابد آخرین نقطه، که همان نقطه‌ی نخست (سمت راست ترین نقطه) است، به دست می‌آید و الگوریتم به پایان می‌رسد.

۴-۱- شبه‌کدی برای پیاده‌سازی این الگوریتم بنویسید.

۴-۲- در حالتی که سه نقطه در یک راستا قرار داشته باشند، چه وضعیتی برای الگوریتم پیش می‌آید؟

۴-۳- پایداری این الگوریتم درباره مشکل گرد شدن عددهای اعشاری در محاسبات را بررسی کنید.

۵- یکی دیگر از روش‌های حل پوشش محدب، به کارگیری تقسیم و غلبه است. اگر دو چند ضلعی محدب وجود داشته باشد، آن گاه می‌توان الگوریتمی نوشت که چند ضلعی محدبی از اجتماع این دو چند ضلعی به دست آورد. الگوریتمی برای به دست آوردن اجتماع دو چند ضلعی محدب ارائه کنید که از مرتبه n باشد. با کمک روش تقسیم و غلبه و اجتماع چند ضلعی‌های محدب، الگوریتمی بر پایه تقسیم و غلبه بنویسید که پوشش محدب برای n نقطه را با مرتبه n log(n) به دست آورد.

۶- تعداد رشته‌های دودویی به طول n با تعداد زیرمجموعه‌های یک مجموعه n عضوی برابر است. این تعداد را به دست آورده و دلیل برقراری این تساوی را شرح دهید.

۷- همانطور که در قسمت ۵-۵-۳ گفته شد، کد نشان داده شده برای پیاده‌سازی رابطه‌ی بازگشتی ضریب‌های دوجمله‌ای، از روش برنامه‌نویسی پویا استفاده می‌کند. تابع binomial_coefficient را بدون استفاده از روش برنامه‌نویسی پویا مجددا پیاده‌سازی کنید.

۸- اثبات ارائه شده در قسمت ۵-۶-۱-۲ را برای اثبات نامتناهی بودن مجموعه‌ی اعداد اول در نظر بگیرید. فرض کنید که نخستین n عدد اول را در یکدیگر ضرب کرده و مقدار حاصل را با یک جمع کنیم (همانند رویه انجام شده در اثبات). آیا عدد به دست آمده همواره مرکب است؟ در صورت مثبت بودن جواب، اثبات آن را ارائه دهید یا در غیر این صورت مثال نقضی برای آن پیدا کنید.

۹- یک الگوریتم عقب‌گرد برای محاسبه‌ی مجموعه‌ی تمامی مقسوم علیه‌های یک عدد صحیح طراحی و پیاده‌سازی کنید.

مراجع

Berg, M. de et al., 2008. *Computational geometry - algorithms and applications* 3rd ed., Springer-Verlag.

Skiena, S.S. & Revilla, M.A., 2006. *Programming challenges: The programming contest training manual*, Springer Science & Business Media.

فصل ٦- حل مسأله

معمولا بسیاری از سوالات مطرح شده در مسابقات برنامه‌نویسی از طریق تبدیل به مسائل معروف الگـوریتمی قابـل حل هستند. به همین دلیـل، آشـنایی بـا ایـن مسـائل و راه حل‌هـای آنهـا یکـی از مهارت‌هـای اصـلی مـورد نیـاز شرکت‌کنندگان در مسابقات برنامه‌نویسی است. در این فصل به معرفی پرکاربردترین مسائل الگوریتمی می‌پـردازیم. کدهای ارائه شده به صورت کامـل و بـه زبـان سـی++ خواهنـد بـود؛ در پیاده‌سـازی مسـأله‌ها، سـاختمان‌داده‌ها و کتابخانه‌های معرفی شده در فصل‌های پیشین را به کار خواهیم برد. برای هر مسأله، ساختار زیر را به کار می‌بریم:

۱- بدنه: بخش آغازین هر مسأله که صورت مسأله و سایر جزئیات مربوط به آن در این بخش ارائه می‌شود.
۲- ورودی: توضیح قالب ورودی در این بخش قرار می‌گیرد.
۳- خروجی: خروجی مورد نیاز برای مسأله و شکل آن در این بخش توضیح داده می‌شود.
۴- راه‌حل: راه‌حل مسأله به زبان سی++ در این بخش ارائه شده و توضیحات لازم درباره‌ی آن داده می‌شود.

برای کار با ورودی و خروجی، از پرونده‌ها در همه‌ی مسأله‌ها استفاده مـی‌کنـیم. ورودی همـه‌ی مسـائل از پرونـده "input.txt" خوانده شده و خروجی در پرونده "output.txt" نوشته می‌شود. زمانی که آدرس به صورت نسبی و تنها با نام پرونده ارائه می‌گردد، پرونده‌ی مربوطه باید در کنار کد برنامه قرار گیرد.
هر کدام از بخش‌های این فصل، به یکی از مسائل تخصیص داده شده است. از اشاره به نـوع الگـوریتم مـورد نیـاز برای حل مسأله (پویا، حریصانه و ..) در صورت مسأله‌ها خودداری می‌کنیم تا خواننـدگان بتواننـد بـا ذهنـی بـاز بـه جستجوی راه‌حل‌ها بپردازند. برای بعضی از پرسش‌ها، محدوده‌ی متغیرهای مسأله نیز بیان می‌شود تا راه‌حـل ارائـه شده در محدوده‌ی زمانی قابل قبولی نسبت به اندازه‌ی ورودی قرار داشته باشد.

بخش زیر که مربوط به سرآیندها و پرونده‌های ورودی و خروجی است، در همه‌ی کدهای ارائه شده ثابت اسـت و از تکرار مجدد آن در هر مسأله خودداری می‌کنیم. تنها سرآیندهای مختص به هر مسأله، در کد مربوط بـه راه‌حـل آن آورده می‌شود.

```
#include <fstream>
using namespace std;
```

```
ifstream fin("input.txt");
ofstream fout("output.txt");
```

٦-١- هشت‌وزیر

مسأله‌ی هشت‌وزیر از مسائل شناخته شده است و اغلب برنامه‌نویس‌ها دست کم نام آن را شنیده‌اند. در این مسأله باید هشت وزیر را در صفحه شطرنج به شکلی قرار داد که یکدیگر را تهدید نکنند. از آنجا که در فصل قبل درباره‌ی این مسأله توضیحات لازم داده شد، از توضیح بیشتر درباره‌ی آن می‌پرهیزیم. در این بخش، برنامه‌ای می‌نویسیم که همه‌ی راه‌حل‌های مسأله هشت‌وزیر را به دست آورد.

ورودی

برای این مسأله نیاز به ورودی وجود ندارد و تعداد وزیرها، هشت عدد در نظر گرفته می‌شود.

خروجی

هر راه‌حل از هشت عدد تشکیل شده که هر عدد مکان یک وزیر در یک سطر از صفحه‌ی شطرنج است. نخستین عدد، مکان نخستین وزیر در سطر اول و به همین ترتیب مکان سایر وزیرها مشخص می‌شود. هر راه‌حل در یک سطر از پرونده‌ی خروجی نوشته می‌شود.

راه‌حل

کد ارائه شده کاملا مشابه الگوریتم ارائه شده در فصل پیش است؛ با این تفاوت که تابع print_answer برای نوشتن همه‌ی راه‌حل‌ها در خروجی به کار گرفته شده است. برای توضیحات بیشتر درباره‌ی کد، به فصل قبل مراجعه کنید.

```
#define QUEENS 8
bool col[QUEENS];
bool diagonal1[QUEENS];
bool diagonal2[QUEENS];
int answer[QUEENS];
void print_answer() {
    for (int i = 1; i < QUEENS; i++)
        fout << answer[i] << " ";
    fout << answer[QUEENS] << endl;
}
void queen8(int r) {
    if (r > QUEENS) {
        print_answer();
        return;
    }
    for (int c = 1; c <= QUEENS; c++) {
        if (col[c] || diagonal1[c - r + 1] || diagonal2[c + r])
```

```
            continue;
        answer[r] = c;
        col[c] = diagonal1[c - r + QUEENS] = diagonal2[c + r] = true;
        queen8(r + 1);
        col[c] = diagonal1[c - r + QUEENS] = diagonal2[c + r] = false;
    }
}
int main() {
    queen8(1);
    fout.close();
    return 0;
}
```

٦-٢- سودوکو

مسألهی سودوکو[۵۶] در دستهی مسائل سرگرمی قرار میگیرد. در این بازی یک جدول ۹X۹ وجود دارد که باید با توجه به قوانین زیر، از اعداد ۱ تا ۹ پر شود:

۱- در هیچ سطری نباید اعداد تکراری وجود داشته باشد.

۲- در هیچ ستونی نباید اعداد تکراری وجود داشته باشد.

۳- در هیچیک از نه مربع ۳X۳ نباید اعداد تکراری وجود داشته باشد.

شکل ۱,۶ یک جدول سودوکو را نمایش میدهد که با رعایت قوانین گفته شده، از اعداد ۱ تا ۹ پر شده است. در این بخش میخواهیم برنامهای مینویسیم که جدول سودوکو را حل کند.

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

شکل ۱,٦. نمونهای از یک جدول سودوکو

[۵۶] Sudoku

ورودی

این مسأله مانند هشت‌وزیر به ورودی نیاز ندارد. البته می‌توان با کمی تغییر، جدولی نیمه‌پر را برای ورودی تعیین کرد که این شکل از جدول سودوکو در مجلات دیده می‌شود. در این بخش هر دو شکل مسأله را حل خواهیم کرد. برای دریافت جدول نیمه‌پر از ورودی، در مکان‌های خالی، صفر و در سایر مکان‌ها عدد مورد نظر را قرار می‌دهیم. ورودی زیر، نمونه‌ای از یک جدول نیمه‌پر است که می‌تواند ورودی مسأله سودوکو باشد:

Input
1 0 3 0 5 0 7 0 0 4 0 6 7 8 0 1 2 3 7 0 0 0 0 0 0 5 6 0 0 4 3 0 5 8 0 7 0 6 5 8 9 7 2 1 4 0 0 7 0 1 4 3 0 5 5 3 1 6 4 2 9 7 8 6 4 2 0 0 0 0 0 1 0 0 0 0 0 1 6 4 2

خروجی

جدول حل شده، متشکل از ۸۱ عدد (نه عدد در هر سطر) خواهد بود که در پرونده‌ی خروجی نوشته می‌شود.

راه‌حل

یکی از راه‌حل‌های ساده برای حل جدول سودوکو، استفاده از روش عقب‌گرد است. برای هر یک از خانه‌های جدول، اعداد ۱ تا ۹ را بررسی کرده و هرکدام که قرار گرفتن آن‌ها در خانه‌ی مورد نظر، امکان‌پذیر باشد را امتحان می‌کنیم. اگر به نقطه‌ای برسیم که نتوانیم حل جدول را ادامه دهیم، یعنی هیچ عددی برای خانه‌ی کنونی گزینش‌پذیر نباشد، به عقب بازگشته و اعداد قرار گرفته در خانه‌های پیشین را تغییر می‌دهیم. پیاده‌سازی این راه حل در ادامه نشان داده شده است.

```
#define MAX_D 9
#define SQUARE_L 3
int table[MAX_D][MAX_D];
void print_solution() {
    for (int i = 0; i<MAX_D; i++) {
        for (int j = 0; j<MAX_D - 1; j++)
            fout << table[i][j] << " ";
        fout << table[i][MAX_D - 1] << endl;
    }
}
bool possible(int r, int c, int val) {
```

```
    for (int i = 0; i<MAX_D; i++)  // check current row
        if (i != c && table[r][i] == val)
            return false;
    for (int i = 0; i<MAX_D; i++)  // check current col
        if (i != r && table[i][c] == val)
            return false;
    // check current square
    int square_r = (r / SQUARE_L) * SQUARE_L;
    int square_c = (c / SQUARE_L) * SQUARE_L;
    for (int i = square_r; i<square_r + SQUARE_L; i++)
        for (int j = square_c; j<square_c + SQUARE_L; j++)
            if (i != r && j != c && table[i][j] == val)
                return false;
}
void sudoku(int r, int c) {
    if (r == MAX_D && c == 0) {
        print_solution();
        fout.close();
        exit(0);  // exist when first solution found
    }
    for (int num = 1; num <= MAX_D; num++) {
        if (possible(r, c, num)) {
            table[r][c] = num;
            if (c == MAX_D - 1)
                sudoku(r + 1, 0);// goto next row
            else
                sudoku(r, c + 1);// goto next col
            table[r][c] = 0;
        }
    }
}
int main() {
    sudoku(0, 0);
    return 0;
}
```

کد بالا پس از یافتن نخستین راه‌حل مسأله، خارج می‌شود. با برداشتن کد exit(0)، همه‌ی راه‌حل‌ها یافت شده و در خروجی نوشته می‌شوند؛ البته از آنجا که روش پیاده شده، روش عقب‌گرد ساده است، یافتن همه‌ی راه‌حل‌ها کمی زمان‌بر خواهد بود.

تابع possible سطر، ستون و مربع کنونی را برای قرار گرفتن عدد val در خانه‌ی کنونی ([r,c]) بررسی کرده و اگر این امر امکان‌پذیر باشد، مقدار true را برمی‌گرداند. برای به دست آوردن مربع مربوط به خانه‌ی کنونی، نمایه‌ی آن خانه را بر ۳ (طول و عرض مربع) تقسیم کرده تا شماره مربع را بدست بیاوریم؛ سپس این شماره را در عدد ۳ ضرب می‌کنیم تا نمایه‌ی خانه‌ی آغاز مربع بدست آید. تابع print_solution راه‌حل یافت شده را به شکلی که در بخش خروجی توضیح داده شد، در پرونده خروجی می‌نویسد.

حال به حل یک جدول نیمه‌پر می‌پردازیم. فرض می‌کنیم جدول نیمه‌پر ورودی صحیح باشد، زیرا در غیر این صورت، راه‌حلی یافت نخواهد شد. کد پیشین را برای این حالت جدید تغییر می‌دهیم. در آغاز باید جدول نیمه‌پر را خوانده و به این صورت در آرایه دوبعدی table قرار دهیم:

```
void fill_table() {
    for (int i = 0; i < MAX_D; i++)
        for (int j = 0; j < MAX_D; j++)
            fin >> table[i][j];
}
```

این تابع را پیش از فراخوانی تابع sudoku فراخوانی می‌کنیم. تابع sudoku باید از خانه‌هایی که پیش از این پر شده‌اند، گذشته و به پر کردن خانه‌های خالی بپردازد:

```
void sudoku(int r, int c) {
    if (r == MAX_D && c == 0) {
        print_solution();
        fout.close();
        exit(0);  // exist when first solution found
    }
    if (table[r][c] != 0) {
        if (c == MAX_D - 1)
            sudoku(r + 1, 0);// goto next row
        else
            sudoku(r, c + 1);// goto next col
    }
    else {
        for (int num = 1; num <= MAX_D; num++) {
            if (possible(r, c, num)) {
                table[r][c] = num;
                if (c == MAX_D - 1)
                    sudoku(r + 1, 0);// goto next row
                else
                    sudoku(r, c + 1);// goto next col
                table[r][c] = 0;
            }
        }
    }
}
```

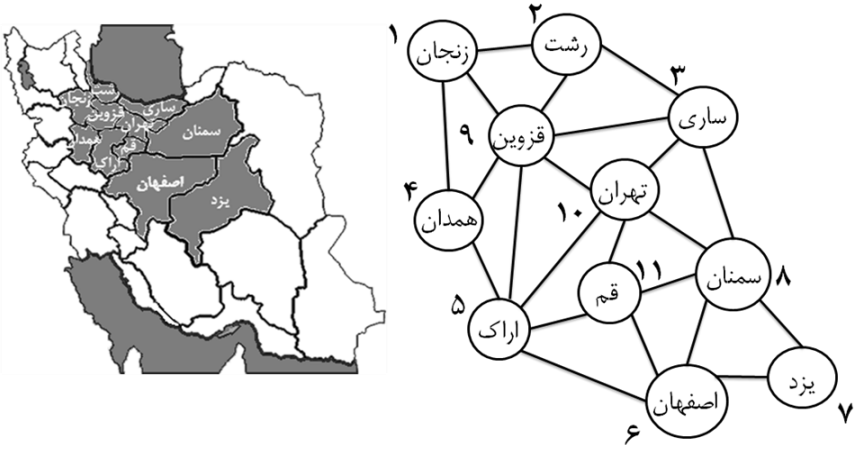
به‌کارگیری وارسی پیش‌رو برای حل جدول سودوکو، سرعت اجرای آن را تا حد زیادی افزایش خواهد داد. یک نمونه از روش پیش‌رو برای هر خانه دامنه‌ای به طول ۹ در نظر می‌گیرد که در آغاز اعداد ۱ تا ۹ در آن قرار دارد. با قرار دادن هر عدد در هر خانه از جدول، دامنه‌ی سایر خانه‌های مرتبط (سطر، ستون و مربع) اصلاح شده و عدد مربوطه از آن‌ها حذف می‌شود. با این تغییر، نیاز به فراخوانی تابع possible در هر مرحله نخواهد بود و از طرفی حالت‌های بن‌بست نیز کاهش خواهد یافت؛ در نتیجه تعداد عقب‌گردها کاهش یافته و سرعت اجرای کد افزایش می‌یابد.

٦-۳- رنگ‌آمیزی گراف

شهرهای درون یک نقشه را با چهار رنگ متمایز $\{c_1, c_2, c_3, c_4\}$ به گونه‌ای رنگ‌آمیزی کنید که هیچ دو شهر مجاوری دارای رنگ‌های یکسان نباشند.

ورودی

آرایه‌ای دوبعدی که نشان‌دهنده‌ی مجاورت شهرها است. این آرایه در واقع ماتریس مجاورت گراف مسطح معادل با نقشه است. برای سادگی بیشتر، به هر شهر یک شماره نسبت می‌دهیم. شکل ۲٫۶ بخش مورد نظر از نقشه‌ی ایران (شهرهای مورد نظر تیره‌تر شده‌اند) و گراف معادل با آن را نشان می‌دهد.



شکل ۲٫٦. نقشه‌ی ایران و گراف متناظر با برخی از شهرها

ورودی متناظر با گراف نشان داده شده در شکل ۲٫۶، این ماتریس خواهد بود:

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	0	1	0	0	0	0	1	0	0
2	1	0	1	0	0	0	0	0	1	0	0
3	0	1	0	0	0	0	0	1	1	1	0
4	1	0	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	1	0	0	1	1	1
6	0	0	0	0	1	0	1	1	0	0	1
7	0	0	0	0	0	1	0	1	0	0	0
8	0	0	1	0	0	1	1	0	0	1	1
9	1	1	1	1	1	0	0	0	0	1	0
10	0	0	1	0	1	0	0	1	1	0	1
11	0	0	0	0	1	1	0	1	0	1	0

خروجی

شماره هر شهر و رنگ نسبت داده شده به آن در پرونده خروجی نوشته می‌شود. هر شهر و رنگ مربوط به آن در یک خط قرار می‌گیرند.

راه‌حل

برای حل این پرسش از جستجوی عقب‌گرد بهره می‌بریم. نخستین راه‌حل یافت شده را در پرونده خروجی نوشته و کار را متوقف می‌کنیم. برای پیاده‌سازی راه‌حل عقب‌گرد، به تابعی برای بررسی رنگ کنونی نیاز داریم. وظیفه این تابع، بررسی مناسب بودن رنگ کنونی برای گره‌ی کنونی است. اگر رنگ مشابه‌ای در گره‌های مجاور وجود داشته باشد، این تابع مقدار false را بازخواهد گرداند. رنگ‌آمیزی را از گره‌ی شماره ۱ آغاز کرده و به ترتیب شماره‌ی گره‌ها، آن‌ها را رنگ می‌کنیم. رنگ‌های برگزیده شده را در آرایه curr_color قرار می‌دهیم تا تابع valid بتواند به آن‌ها دسترسی داشته باشد و از طرفی بتوان آن‌ها را در پرونده خروجی نوشت. آرایه‌ی colors شامل ۴ رنگ مورد نظر بوده و آرایه‌ی دوبعدی graph در واقع ماتریس مجاورت گراف است که یال‌های میان گره‌ها را مشخص می‌کند. تابع coloring_graph در هر حالتی، یک ترتیب درست از رنگ‌ها را بدست خواهد آورد زیرا به سادگی ثابت می‌شود که با به‌کارگیری ۴ رنگ (و بیشتر) می‌توان هر گراف مسطحی را رنگ‌آمیزی کرد.

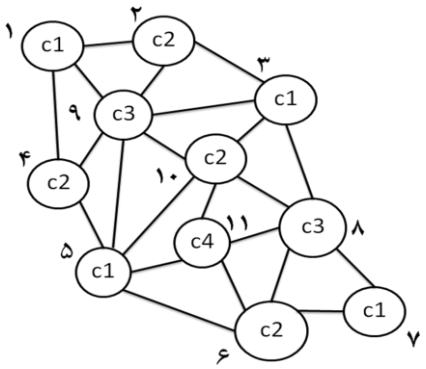
```
#include <string>
#define MAX_N 12
#define MAX_COLOR 4
int graph[MAX_N][MAX_N];
string curr_color[MAX_N];
string colors[MAX_COLOR] = {"c1", "c2", "c3", "c4"};
void print_solution() {
    for (int i = 1; i < MAX_N; i++)
        fout << i << " " << curr_color[i] << endl;
}
bool valid(int node, int c) {
    for (int i = 1; i < MAX_N; i++) {
        if (graph[node][i] == 1 && curr_color[i] == colors[c])
            return false;
    }
    return true;
}
void coloring_graph(int node) {
    if (node == MAX_N) {
        print_solution();
        fout.close();
        exit(0);
    }
    for (int c = 0; c < MAX_COLOR; c++) {
        if (valid(node, c)) {
```

```
            curr_color[node] = colors[c];
            coloring_graph(node + 1);
        }
    }
}
int main() {
    for (int i = 1; i < MAX_N; i++)
        for (int j = 1; j < MAX_N; j++)
            fin >> graph[i][j];
    fin.close();
    coloring_graph(1);
    return 0;
}
```

اگر گره‌ها را بر اساس بیشترین محدودیت (تعداد یال‌های متصل به هر گره) مرتب کرده و سپس آن‌ها رنگ کنیم، سرعت اجرای الگوریتم افزایش پیدا خواهد کرد؛ زیرا چنین گره‌هایی عامل اصلی عقب‌گرد در برنامه هستند و اگر آن‌ها را زودتر رنگ کنیم، تعداد عقب‌گردها کاهش یافته و در نتیجه سرعت اجرای برنامه افزایش خواهد یافت.

با تغییر ثابت‌های تعریف شده در کد، به راحتی می‌توانید هر گراف مسطحی را با هر تعداد گره و رنگ، رنگ‌آمیزی کنید. خروجی کد ارائه شده برای ورودی مسأله به این صورت خواهد بود (شکل ۳,۶):

```
Output
1 c1
2 c2
3 c1
4 c2
5 c1
6 c2
7 c1
8 c3
9 c3
10 c2
11 c4
```



شکل ۳,٦. خروجی الگوریتم رنگ‌آمیزی گراف برای گراف شکل ۲,٤

٦-٤- مرتب‌سازی با کمترین تعداد جابجایی

دنباله‌ای متشکل از سه عدد ۱، ۲ و ۳ در ورودی داده شده است. طول این دنباله می‌تواند در بازه‌ی ۱ تا ۱۰۰۰ باشد. برای مرتب‌سازی این دنباله، فقط می‌توانیم از عمل جابجایی استفاده کنیم؛ جابجایی به معنی عوض کردن مکان دو عنصر در دنباله است. کمترین تعداد جابجایی‌های مورد نیاز، برای مرتب کردن این دنباله (به صورت غیرنزولی) را بدست آورید.

ورودی

دنباله‌ای به طول N از اعداد ۱، ۲ و ۳. در نخستین خط از پرونده ورودی، مقدار N و در خط‌های بعدی، اعداد موجود در دنباله (به تعداد N) قید می‌شوند. مثالی از ورودی به این صورت است:

Input
5 1 3 2 1 3

خروجی

کمترین تعداد جابجایی‌های مورد نیاز برای مرتب کردن غیرنزولی دنباله‌ی ورودی. برای نمونه، خروجی متناظر با ورودی داده شده به صورت زیر است. با جابجایی عنصر دوم (عدد ۳) و عنصر چهارم (عدد ۱)، دنباله‌ی ورودی مرتب خواهد شد.

Output
1

راه‌حل

ابتدا ساده‌ترین راه‌حل برای این مسأله را توضیح داده و سپس راه‌حلی بهتر ارائه خواهیم کرد. ابتدا دنباله‌ی ورودی را مرتب می‌کنیم؛ با این کار، مکان نهایی هر عنصر مشخص می‌شود. بهترین جابجایی، دو عنصر را به مکان نهایی خود منتقل می‌کند؛ بنابراین ابتدا به جستجوی زوج عنصرهایی می‌پردازیم که جابجایی‌شان، هر دوی آن‌ها را به مکان نهایی خود منتقل می‌کند. پس از پایان این جابجایی‌ها، بقیه عنصرها را به مکان نهایی خود منتقل می‌کنیم.

```
#define MAX_N 1000+1
int n, swaps_counter;
int sequence[MAX_N];
int sorted[MAX_N];
int compare(const void* a, const void* b) {
    return *(int*)a - *(int*)b;
}
int sort_by_exchange() {
    swaps_counter = 0;
    // find swaps that correct two elements
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (sequence[i] != sorted[i] && sequence[j] != sorted[j]
                && sequence[i] == sorted[j] && sequence[j] == sorted[i]) {
                swap(sequence[i], sequence[j]);
                swaps_counter++;
            }
    int bad_pos = 0; // sort other elements
    for (int i = 0; i < n; i++)
        if (sequence[i] != sorted[i])
            bad_pos++;
    swaps_counter += (bad_pos / 3) * 2;
    return swaps_counter;
}
int main() {
    fin >> n;
    for (int i = 0; i<n; i++) {
        fin >> sequence[i];
        sorted[i] = sequence[i];
    }
    qsort(sorted, n, sizeof(int), compare);
    fout << sort_by_exchange() << endl;
    fin.close();
    fout.close();
    return 0;
}
```

تابع sort_by_exchange از دو بخش تشکیل شده است:

۱- یافتن جابجایی‌هایی که دو عنصر را به مکان نهایی خود منتقل می‌کند.

۲- جابجایی سایر عنصرها به مکان نهایی‌شان.

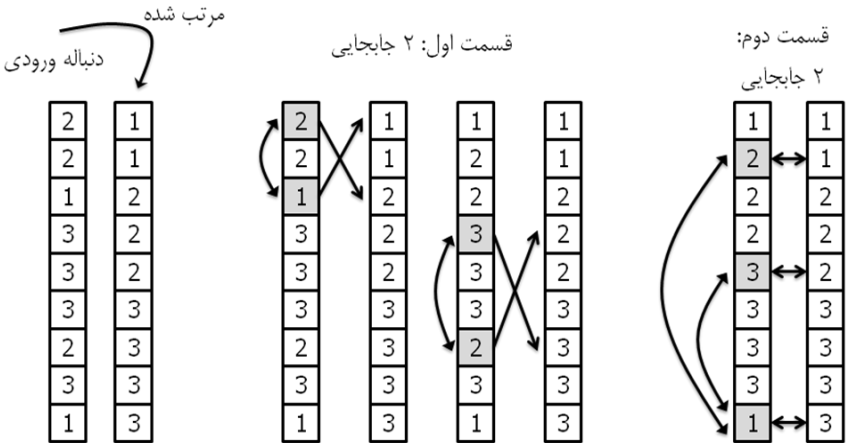
بخش اول نیاز به توضیح اضافی ندارد. پس از پایان کار بخش نخست، تعدادی از عنصرها هنوز در مکان نامناسبی قرار دارند (شاید این تعداد صفر باشد). در چنین حالتی دست کم باید سه عنصر وجود داشته باشد، زیرا اگر فقط دو عنصر در مکان نامناسب قرار داشته باشند، حتما توسط بخش اول به مکان نهایی خود منتقل شده‌اند. برای جابجایی سه عنصر به مکان اصلی خود، به دو جابجایی نیاز داریم (چرا؟). بنابراین هر سه عنصر با دو جابجایی به مکان نهایی خود منتقل می‌شوند و بر این اساس، برای به دست آوردن تعداد جابجایی‌های مورد نیاز برای عناصر باقیمانده، تعداد

آن‌ها را بر ۳ تقسیم کرده (به دست آوردن تعداد سه‌تایی‌ها) و نتیجه را در ۲ ضرب می‌کنیم (دو جابجایی برای هر سه عنصر). مجموع تعداد بدست آمده از بخش اول و دوم، پاسخ مسأله است.

```
swaps_counter += (bad_pos / 3) * 2;
```

در ادامه یک نمونه ورودی و خروجی متناظر با آن، آورده شده است. شکل ۴,۶ روند کار برنامه برای حل این نمونه را نشان می‌دهد.

Input	Output
9 2 2 1 3 3 3 2 3 1	4



شکل ٤,٦. مرتب‌سازی مجموعه‌ای از اعداد صحیح با کمترین تعداد جابجایی

در راه‌حل پیشنهادی، از روش سریع برای مرتب‌سازی و از دو حلقه برای یافتن جابجایی‌ها استفاده شده است. پیچیدگی فضایی این روش $O(n^2)$ است. با استفاده از محدودیت موجود در مقدار اعداد دنباله (۱، ۲ و ۳)، می‌توان روش دیگری ارائه کرد که از پیچیدگی زمانی $O(n)$ برخوردار باشد. برای مرتب‌سازی دنباله‌ی ورودی، می‌توان از شمارش تعداد ۱ها، ۲ها و ۳ها استفاده کرد؛ زیرا در آغاز دنباله‌ی مرتب، تعدادی ۱، در میانه‌ی آن، تعدادی ۲ و در پایان آن، تعدادی ۳ خواهد آمد. علاوه‌براین نیازی به استفاده از دنباله‌ی دیگری برای نگهداری دنباله‌ی مرتب شده

نیست و فقط نگهداری تعداد ۱ها، ۲ها و ۳ها کافی است. برای یافتن تعداد جابجایی‌ها نیز، به این صورت عمل می‌کنیم:

۱- ۲هایی که در بخش آغازین دنباله (مکان ۱ها) قرار دارند را با ۱هایی که در بخش میانی (مکان ۲ها) دنباله قرار دارند، جابجا می‌کنیم.

۲- ۱هایی که در بخش پایانی دنباله قرار دارند را با ۳هایی که در بخش آغازین دنباله قرار دارند، جابجا می‌کنیم.

۳- ۲هایی که در بخش پایانی دنباله (مکان ۳ها) قرار دارند را با ۳هایی که در بخش میانی دنباله قرار دارند، جابجا می‌کنیم.

۴- سایر عناصر را مانند بخش دوم کد پیشین، به مکان نهایی خود منتقل می‌کنیم.

پیاده‌سازی این روش که هم از نظر زمانی و هم از نظر فضایی بهتر از روش پیشین است، به این صورت است:

```
#define MAX_N 1000 + 1
int n, swaps_counter;
int sequence[MAX_N];
int c[4] = {0};  // 1,2,3 count
             // pj[i]: number of element i in part j
int p1[4] = {0}, p2[4] = {0}, p3[4] = {0};
int sort_by_exchange() {
    int i;
    for (i = 0; i < c[1]; i++)
        p1[sequence[i]]++;
    for (; i < c[1] + c[2]; i++)
        p2[sequence[i]]++;
    for (; i < c[1] + c[2] + c[3]; i++)
        p3[sequence[i]]++;
    int diff = 0;
    // 1. swap 2 with 1 in part 2
    diff = min(p1[2], p2[1]);
    p1[1] += diff; p1[2] -= diff;
    p2[1] -= diff; p2[2] += diff;
    swaps_counter += diff;
    // 2. swap 3 with 1 in part 3
    diff = min(p1[3], p3[1]);
    p1[1] += diff; p1[3] -= diff;
    p3[1] -= diff; p3[3] += diff;
    swaps_counter += diff;
    // 3. swap 2 with 3 in part 2
    diff = min(p2[3], p3[2]);
    p2[2] += diff; p2[3] -= diff;
    p3[2] -= diff; p3[3] += diff;
    swaps_counter += diff;
    // 4. swap other element
    swaps_counter += 2 * (p1[2] + p1[3]);
    return swaps_counter;
```

```
}
int main() {
    fin >> n;
    for (int i = 0; i < n; i++) {
        fin >> sequence[i];
        c[sequence[i]]++;
    }
    fout << sort_by_exchange() << endl;
    fin.close();
    fout.close();
    return 0;
}
```

٦-٥- اتصال شهرها با کمترین هزینه

فرض کنید N شهر در یک استان وجود دارند و میان آن‌ها هیچ جاده‌ای وجود ندارد. حال مسئولین تصمیم گرفته‌اند که میان این شهرها، جاده‌هایی ایجاد کنند. از طرفی برای کاهش هزینه، مسئولین قصد دارند جاده‌کشی را به صورتی انجام دهند که کوتاه‌ترین جاده‌های ممکن ایجاد شود؛ زیرا هرچه طول جاده‌ها بیشتر باشد، هزینه‌ی ساخت آن‌ها نیز بیشتر خواهد شد. تعداد شهرها (N) متغیر بوده و در ورودی مسأله قید می‌شود ($3 \le N \le 100$). مسافت میان شهرها را دریافت کرده و کمترین مجموع طول جاده‌هایی را که می‌توان ایجاد کرد، بدست آورید.

ورودی

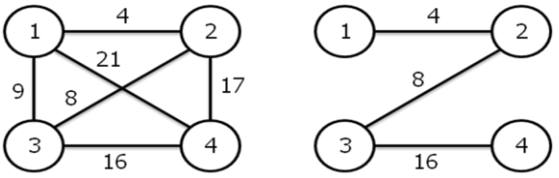
تعداد شهرهای مورد نظر که باید میان آن‌ها جاده کشید (N) و ماتریسی $N \times N$ که مسافت میان هر شهر تا سایر شهرها را مشخص می‌کند؛ برای نمونه، مقدار موجود در سطر i_ام و ستون j _ام مشخص‌کننده‌ی مسافت میان شهر i و j است. مسافت رفت و برگشت میان دو شهر برابر است، بنابراین ماتریس ورودی متقارن خواهد بود. مسافت میان دو شهر از مقدار ۱۰۰۰۰۰ تجاوز نخواهد کرد. ورودی زیر یک نمونه ورودی برای این مسأله است:

Input
4 0 4 9 21 4 0 8 17 9 8 0 16 21 17 16 0

خروجی

کمترین مجموع طول جاده‌هایی که می‌تواند همه‌ی شهرها را به هم متصل کند، در پرونده‌ی خروجی قرار خواهد گرفت. برای نمونه، خروجی مربوط به ورودی بالا به صورت زیر است. شکل ۵،۶، گراف متناظر با ورودی و خروجی ارائه شده را مشخص می‌کند.

Output
28



شکل ٥,٦. یک گراف نمونه و درخت پوشای کمینه‌ی متناظر با آن

راه‌حل

این مسأله از نمونه مسأله‌های درخت پوشای کمینه است. روش کروسکال را برای حل این مسأله به کار می‌بریم. برای استفاده از این روش، به مجموعه‌ی یال‌ها نیاز داریم؛ از این‌رو ساختمان‌داده‌ی زیر را برای نگهداری یک یال تعریف کرده و از ماتریس ورودی، مجموعه‌ی یال‌ها را بدست می‌آوریم.

```
struct edge {
    int x;
    int y;
    int cost;
};
```

الگوریتم کروسکال را برای این مسأله، به این صورت پیاده‌سازی می‌کنیم:

```
#include <algorithm>
#define MAX_N 100 + 2
#define MAX_E (100 * 99)/2 + 10// max edges count: n(n-1)/2
edge edges[MAX_E];
int n;
int compare(const void* a, const void* b) {
    return (((edge*)a)->cost - ((edge*)b)->cost);
}
int components[MAX_N] = {0};
int depth[MAX_N] = {0};
int kruskal() {
    int min_cost = 0;
    for (int i = 0; i < n; i++) {
        components[i] = i;   // initialize components
        depth[i] = 0;     // depth of all tree is 0 (in start)
    }
    int selected_edge_count = 0;
```

```
    int next_e_index = 0;
    int s, d;
    while (selected_edge_count < n - 1) {
        s = edges[next_e_index].x;
        d = edges[next_e_index].y;
        // check two vertics in same group or not
        while (components[s] != s)
            s = components[s];
        while (components[d] != d)
            d = components[d];
        if (s != d) {
            if (depth[s]<depth[d])
                components[s] = d;
            else if (depth[s] > depth[d])
                components[d] = s;
            else {
                components[s] = d;
                depth[d]++;
            }
            min_cost += edges[next_e_index].cost;
            selected_edge_count++;
        }
        next_e_index++;
    }
    return min_cost;
}
```

پیاده‌سازی بالا دقیقا معادل با کد ارائه شده برای این الگوریتم در فصل پیش است. در این پیاده‌سازی، از مجموعه‌های از هم جدا استفاده کرده‌ایم. متغیرهای components و depth برای نگهداری مجموعه‌ها و عمق هریک از آن‌ها تعریف شده‌اند. در آغاز، هر شهر یک مجموعه (یک درخت) با یک عضو و عمق صفر در نظر گرفته می‌شود و به مرور با اتصال شهرها به یکدیگر، تعداد عضوهای هر مجموعه و عمق آن افزایش می‌یابد.

خروجی تابع kruskal از نوع int است زیرا بیشترین طول جاده‌ها از محدوده یک متغیر ۴ بایتی تجاوز نمی‌کند. تعداد یال‌های درخت پوشای کمینه، $N-1$ یال است (حداکثر ۹۹ یال) و اگر همه‌ی آن‌ها، مقداری (طول جاده) برابر با بیشینه‌ی طول جاده‌ها (۱۰۰۰۰۰) داشته باشند، خواهیم داشت:

$$(n-1)*100,000 = 99*100,000 = 9,900,000$$

این مقدار در محدوده‌ی نوع int قرار می‌گیرد، زیرا:

$$-2,147,483,648 \le INT \le 2,147,483,647$$

آرایه‌ی edges هنگام دریافت ورودی پر شده و تابع kruskal از آن استفاده خواهد کرد. تابع main به صورت زیر خواهد بود:

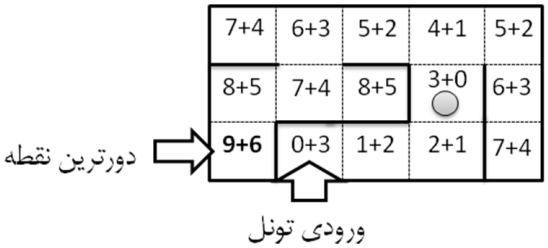
```
int main() {
    int e_counter = 0, road_cost;
    fin >> n;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            fin >> road_cost;
            if (j <= i)
                continue;
            edges[e_counter].x = i;
            edges[e_counter].y = j;
            edges[e_counter].cost = road_cost;
            e_counter++;
        }
    // sort edges by length of road (asc)
    qsort(edges, e_counter, sizeof(edge), compare);
    fout << kruskal() << endl;
    fin.close();
    fout.close();
    return 0;
}
```

شرط if یال‌های تکراری را نادیده گرفته و در مجموعه‌ی یال‌ها قرار نمی‌دهد، زیرا ماتریس ورودی متقارن است. نیازی به نگهداری ماتریس ورودی نبوده و نگهداری یال‌ها برای الگوریتم کروسکال کافی است. پیش از فراخوانی تابع kruskal، مجموعه‌ی یال‌ها را به صورت غیرنزولی مرتب می‌کنیم؛ زیرا لازمه‌ی به‌کارگیری الگوریتم کروسکال، مرتب بودن یال‌ها است.

٦-٦- بزرگترین کوتاه‌ترین مسیر

در یک تونل راه‌های متفاوتی وجود دارد. شخصی در یک مکان از تونل قرار داشته و قصد دارد دورترین نقطه نسبت به مکان کنونی خود و ورودی تونل را پیدا کند. دورترین مکان نسبت به مکان کنونی و ورودی تونل، مکانی است که مجموع کوتاه‌ترین فاصله‌ی آن تا مکان کنونی و کوتاه‌ترین فاصله‌ی آن تا ورودی تونل بیشینه باشد.

برای نمونه شکل ۶٫۶ یک تونل نمونه را نمایش داده و دورترین نقطه نسبت به مکان کنونی و ورودی تونل را مشخص کرده است. مسافت هر خانه تا خانه‌ی مجاور، یک واحد در نظر گرفته می‌شود. خطهای پررنگ در شکل، نشان‌دهنده‌ی دیوارها هستند.



شکل ٦,٦. نمونه‌ای از یک تونل، خط‌های پررنگ نشان‌دهنده‌ی دیوارها هستند.

ورودی

در ابتدای ورودی، تعداد سطر و ستون‌های تونل و سپس مکان کنونی شخص و ورودی تونل آورده می‌شود. برای نمایش ساده‌ی تونل، وجود دیوار در چهار همسایگی (چپ، بالا، راست و پایین) هر خانه، با عدد ۰ و عدم وجود دیوار با ۱ نشان داده می‌شود. ترتیب خانه‌ها از نخستین سلول (سمت چپ) در نخستین سطر به سمت آخرین سلول (سمت راست) در آخرین سطر است (شکل ۷,۶).

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

شکل ٧,٦. ترتیب شماره‌گذاری خانه‌های یک تونل

ورودی نشان داده شده در شکل ۷,۶ به صورت زیر است:

```
Input
3 5
2 4 3 2
0 0 1 0
1 0 1 1
1 0 1 0
1 0 1 1
1 0 0 1
0 0 1 1
1 1 1 0
1 0 0 0
0 1 0 1
0 1 0 1
0 1 0 0
0 0 1 0
1 0 1 0
1 1 0 0
0 1 0 0
```

[3,5] مشخص کننده ۳ سطر و ۵ ستون و $3*5=15$ خانه در تونل است؛ بنابراین در ورودی، ۱۵ چهارتایی خواهیم داشت. [2,4] مکان کنونی شخص و [3,2] ورودی تونل را مشخص می‌کند. چهارتایی بعدی، دیوارهای خانه‌ی اول، چهارتایی دوم، دیوارهای خانه‌ی دوم و به همین صورت هر چهارتایی، مشخص کننده‌ی چهار دیوار هر خانه است. برای نمونه، نخستین چهارتایی [0,0,1,0] در ورودی بالا را در نظر بگیرید: ۰ نخست به معنای وجود دیوار در سمت چپ خانه‌ی اول، ۰ دوم به معنی وجود دیوار در بالای آن، ۱ نشان‌دهنده‌ی عدم وجود دیوار در سمت راست آن و آخرین ۰، وجود دیوار در پایین خانه‌ی ۱ را مشخص می‌سازد.

خروجی

دورترین خانه از مکان کنونی و ورودی تونل را در پرونده خروجی بنویسید. برای نمونه، خروجی مربوط به ورودی بالا به صورت زیر خواهد بود:

Output
15

راه‌حل

این مسأله را با استفاده از پیمایش اول‌عمق حل می‌کنیم. از مکان کنونی، پیمایش را آغاز کرده و مسافت همه‌ی مکان‌ها نسبت به این نقطه را بدست می‌آوریم. سپس از ورودی تونل، پیمایش را آغاز کرده و مسافت همه‌ی مکان‌ها نسبت به این نقطه را به دست آورده و با مقدار مسافت پیشین جمع می‌کنیم. در پایان بیشترین مجموع را به دست آورده و به خروجی می‌بریم.

برای به دست آوردن کوتاه‌ترین مسافت‌ها نسبت به مکان کنونی، پیمایش اول‌عمق را به صورت زیر به کار می‌بریم.

```
struct point {
    int x;
    int y;
};
struct adj {  // left, top, right, bottom
    int ltrb[4];
};
#define MAX_N 50 + 1
adj links[MAX_N][MAX_N];
int r, c;
point curr, entrance;
int distances1[MAX_N][MAX_N], distances2[MAX_N][MAX_N];
int adj_pos[4][2] = {{0, -1}, {-1, 0}, {0, 1}, {1, 0}};
void dfs_search_source(int x, int y, int dist) {
    distances1[x][y] = dist;
    for (int i = 0; i<4; i++) {
```

```
        int adj_x = x + adj_pos[i][0];
        int adj_y = y + adj_pos[i][1];
        if (links[x][y].ltrb[i] == 1 && (distances1[adj_x][adj_y] == -1
||
            distances1[adj_x][adj_y]>dist + 1))
            dfs_search_source(adj_x, adj_y, dist + 1);
    }
}
```

آرایه‌ی links برای نگهداری تونل به کار برده می‌شود؛ هر خانه از این آرایه دارای چهار بخش است که وضعیت دیوارهای موجود در چهار سوی آن را مشخص می‌کند. کوتاه‌ترین مسافت‌های بدست آمده نسبت به مکان کنونی و ورودی تونل در دو آرایه‌ی distances1 و distances2 قرار داده می‌شود. برای بررسی چهار همسایه‌ی خانه‌ی $[x, y]$، آرایه‌ی ajd_pos را به کار می‌بریم؛ با استفاده از این آرایه، بررسی دیوارها به سادگی و توسط یک حلقه صورت خواهد گرفت.

برای محاسبه‌ی مسافت‌ها نسبت به ورودی، تابعی مشابه با تابع dfs_search_source به کار می‌بریم؛ با این تفاوت که، مقادیر محاسبه شده در آرایه‌ی distances2 قرار داده می‌شوند.

```
void dfs_search_entrance(int x, int y, int dist) {
    distances2[x][y] = dist;
    for (int i = 0; i < 4; i++) {
        int adj_x = x + adj_pos[i][0];
        int adj_y = y + adj_pos[i][1];
        if (links[x][y].ltrb[i] == 1 && (distances2[adj_x][adj_y] == -1
            || distances2[adj_x][adj_y] > dist + 1))
            dfs_search_entrance(adj_x, adj_y, dist + 1);
    }
}
```

عملکرد این تابع به صورت زیر است:

۱- فراخوانی تابع با نقطه مبدا و مسافت صفر صورت می‌گیرد.

۲- چهار همسایگی نقطه‌ی ورودی بررسی شده و اگر دیواری میان این خانه و همسایه‌ی مورد نظر وجود نداشته باشد، تابع برای آن همسایگی به صورت بازگشتی فراخوانی می‌شود. برای جلوگیری از بازگشت‌های بی‌مورد و تکراری، شرط زیر نیز بررسی شده است. این شرط، فقط در صورتی تابع را برای همسایه‌ی $[adj\_x, adj\_y]$ فراخوانی می‌کند که مسافت بدست آمده برای آن همسایه از مقدار مسافت کنونی آن (که در فراخوانی‌های پیشین تابع محاسبه شده) کمتر باشد.

```
distances2[adj_x][adj_y] > dist + 1
```

پس از محاسبه‌ی مسافت همه‌ی خانه‌های تونل نسبت به خانه‌ی کنونی و ورودی تونل، کافی است مکانی را که دارای بیشترین مجموع مسافت است، جستجو کنیم:

```
int find_min_of_max() {
    int max_of_mins = 0;
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            if (distances1[i][j] + distances2[i][j] > max_of_mins)
                max_of_mins = distances1[i][j] + distances2[i][j];
    return max_of_mins;
}
```

نحوه‌ی دریافت ورودی و مقادیر آرایه‌ی links در ادامه نشان داده شده است. دو تابع جستجوی اول‌عمق dfs_search_source و dfs_search_entrance، برای نقطه‌ی کنونی و ورودی تونل فراخوانی می‌شوند. برای فرستادن این دو نقطه برای دو تابع مذکور از مختصات آن‌ها یک واحد کم می‌کنیم؛ زیرا نمایه‌ی آرایه‌ها از صفر آغاز می‌شود.

```
int main() {
    int temp;
    fin >> r >> c;
    fin >> curr.x >> curr.y >> entrance.x >> entrance.y;
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++) {
            fin >> links[i][j].ltrb[0] >> links[i][j].ltrb[1]
                >> links[i][j].ltrb[2] >> links[i][j].ltrb[3];
            distances1[i][j] = distances2[i][j] = -1;
        }
    dfs_search_source(curr.x - 1, curr.y - 1, 0);
    dfs_search_entrance(entrance.x - 1, entrance.y - 1, 0);
    fout << find_min_of_max() << endl;
    fout.close();
    fin.close();
    return 0;
}
```

٦-٧- برگزیدن بهترین شهر برای زندگی

در یک استان چند شهر وجود دارند که هر یک فاصله‌ای مشخص نسبت به دیگر شهرها دارند. شخصی دارای شغلی است که روزانه رفت‌وآمد زیادی به دیگر شهرها دارد؛ از این رو قصد دارد شهری را که مجموع مسافت آن نسبت به دیگر شهرها کمینه است، برای زندگی برگزیند؛ زیرا در این صورت هزینه‌ی رفت و آمدش نیز کمینه می‌شود. در این مسأله باید چنین شهری را بیابیم.

ورودی

خط اول ورودی، تعداد شهرها (N) را مشخص می‌کند. شهرها با شماره‌ی 1 تا N مشخص می‌شوند. تعداد شهرها حداقل ۵ و حداکثر ۱۰۰ خواهد بود. در خط‌های بعد، ماتریسی $N \times N$ شامل مسافت هر شهر نسبت به دیگر شهرها آورده می‌شود. این ماتریس نسبت به قطر اصلی متقارن است و مقادیر آن از ۱۰۰۰۰ تجاوز نمی‌کند. ورودی زیر، یک نمونه‌ی قابل قبول برای این مسأله است:

Input
5 0 5 10 15 20 25 5 0 30 35 40 45 10 30 0 50 55 15 35 50 0 60 25 45 55 60 0

خروجی

شماره‌ی شهری که مجموع مسافت آن نسبت به دیگر شهرها کمینه است، باید در خروجی نوشته شود. خروجی مربوط به ورودی بالا برابر خواهد بود با:

Output
1

راه‌حل

این مسأله نیاز به یافتن کوتاه‌ترین مسیرها دارد. از طرفی باید همه‌ی کوتاه‌ترین مسیرها از هر شهر به دیگر شهرها را محاسبه کنیم؛ بنابراین الگوریتم فلوید–وارشال راه‌گشا خواهد بود. برای حل این مسأله، ابتدا با استفاده از الگوریتم فلوید–وارشال، همه‌ی کوتاه‌ترین مسیرها را می‌یابیم. سپس مجموع همه‌ی کوتاه‌ترین مسیرها از هر شهر به دیگر شهرها را محاسبه کرده و کمترین آن‌ها را برمی‌گزینیم.

```
#define MAX_N 100 + 1
#define MAX_DIST 10000 + 1
int n;
int distances[MAX_N][MAX_N], min_path[MAX_N][MAX_N];
void floyd_warshall() {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            min_path[i][j] = distances[i][j];
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
```

```
            for (int j = 1; j <= n; j++)
                min_path[i][j] = min(distances[i][j]
                    , distances[i][k] + distances[k][j]);
}
int find_min_sum() {
    int min_sum = MAX_N * MAX_DIST, min_city = -1;
    int curr_sum;
    for (int i = 1; i <= n; i++) {
        curr_sum = 0;
        for (int j = 1; j <= n; j++)
            curr_sum += min_path[i][j];
        if (curr_sum<min_sum) {
            min_sum = curr_sum;
            min_city = i;
        }
    }
    return min_city;
}
int main() {
    fin >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            fin >> distances[i][j];
    floyd_warshall();
    fout << find_min_sum() << endl;
    fin.close();
    fout.close();
    return 0;
}
```

عملکرد تابع floyd-warshall روشن بوده و قبلا با آن آشنا شده‌اید. تابع find_min_sum، مجموع همه‌ی مسافت‌ها از هر شهر به دیگر شهرها را محاسبه کرده و کمترین آن‌ها را برمی‌گزینند. آنگاه، شماره‌ی شهری که دارای کمترین مجموع مسافت‌ها است را برمی‌گرداند.

٦-٨- سیستم پولی

یک سیستم پولی از مجموعه‌ای از سکه‌ها تشکیل می‌شود که هر سکه دارای ارزشی مشخص و یکتاست. برای نمونه مجموعه‌ی $\{۱، ۲، ۳\}$ یک سیستم پولی است که از سکه‌های ۱، ۲ و ۳ ریالی تشکیل شده است. حال می‌خواهیم بدانیم برای به دست آوردن ارزشی مشخص (مثلا ۱۰ ریال) در یک سیستم پولی، چند روش وجود دارد. برای نمونه برای به دست آوردن مقدار ۱۰ ریال با استفاده از سیستم پولی $\{۱، ۲، ۵\}$، ده روش وجود دارد که عبارتند از:

```
10*1
5*2
2*5
1*5 + 2*2 + 1*1
```

```
1*5 + 1*2 + 3*1
1*5 + 5*1
4*2 + 2*1
3*2 + 4*1
2*2 + 6*1
1*2 + 8*1
```

ورودی

عدد نخست در پرونده ورودی، تعداد سکه‌های سیستم پولی را مشخص می‌کند ($1 \le V \le 25$). عـدد دوم، مقـدار پولی است که باید با به‌کارگیری سیستم پولی به آن دست یابیم ($1 \le N \le 10,000$). در خط بعدی پرونـده، V عدد خواهیم داشت که مشخص کننده‌ی مقدار سکه‌های سیستم پولی هستند. برای نمونه، ورودی مربوط به نمونه‌ی بالا در ادامه آمده است:

Input
3 10 1 2 5

خروجی

تعداد روش‌هایی که می‌توان با به‌کارگیری سیستم پولی، مقدار مشخص شده را به دسـت آورد، در پرونـده خروجـی نوشته می‌شود. این مقدار حداکثر در محدوده‌ی یک متغیر ۸ بایتی علامت‌دار خواهد بود. برای نمونه، خروجی مربوط به ورودی بالا به صورت است:

Output
10

راه‌حل

این مسأله از دسته مسأله‌های پویا است. بنابراین دارای یک الگوی بازگشتی برای به دست آوردن مراحـل جدیـد از مرحله (مرحله‌های) پیشین است که در ادامه به آن می‌پردازیم.

آرایه‌ی دوبعدی $count$ را برای نگهداری تعداد روش‌های بدست آمـده در نظـر می‌گیـریم. $count[n][v]$ بـه معنی تعداد روش‌های ممکن برای ساخت مجموع n با استفاده از سـکه‌های $1..v$ اسـت. فـرض مـی‌کنیم مقـدار $count[i][j-1]$ را محاسبه کرده و حال قصد محاسبه‌ی $count[i][j]$ را داریم. سـکه‌ی i یـا در سـاختن مجموع مورد نظر شرکت می‌کند یا نمی‌کند.

۱- اگر سکه‌ی i_ام در ساخت مجموع شرکت نکند، تعداد حالت‌های قبل را خواهیم داشت: $count[i][j]$

۲- اگر سکه‌ی i_ام در ساخت مجموع شرکت کند، ابتدا مقدار آن را از مجموع کم کرده و سپس مقدار باقیمانده را با استفاده از سکه‌های $1..j$ (دوباره می‌توان سکه‌ی کنونی را به کار برد) می‌سازیم:

$$count[i - coins[j]][j]$$

از مجموع دو مقدار بالا، خواهیم داشت:

$$count[i][j] = count[i][j-1] + count[i - coins[j]][j]$$

$coins[j]$ مقدار سکه‌ی j_ام را مشخص می‌کند. بر اساس توضیحات بالا، مقدار $count[0][v] = 1$ خواهد بود، زیرا برای ساختن مجموع صفر با هر تعداد سکه‌ای، تنها یک راه‌حل وجود دارد. کد ارائه شده به سادگی این راه‌حل پویا را پیاده‌سازی کرده و نیاز به توضیح بیشتری ندارد.

```
#define max_sum 10000 + 1
#define max_coins 25 + 1
int v, n;
int coins[26];
long long count[max_sum][max_coins];
int main() {
    fin >> v >> n;
    for (int i = 1; i <= v; i++)
        fin >> coins[i];
    for (int i = 0; i <= v; i++)
        count[0][i] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= v; j++) {
            count[i][j] = count[i][j - 1];
            if (i - coins[j] >= 0)
                count[i][j] += count[i - coins[j]][j];
        }
    fout << count[n][v] << endl;
    fin.close();
    fout.close();
    return 0;
}
```

٦-۹- بالاترین امتیاز

چند تن از اساتید دانشگاه، می‌خواهند چند پرسش را برای یک مسابقه‌ی برنامه‌نویسی انتخاب کنند. تعداد مشخصی پرسش در مجموعه‌ی پرسش‌ها وجود دارد که هر یک به زمان مشخصی برای حل شدن نیاز دارند. حل هرکدام از این پرسش‌ها، امتیاز معینی برای حل‌کننده در پی خواهد داشت. برای راحتی، هر دسته از پرسش‌ها را در گروه جداگانه‌ای قرار داده‌اند؛ به شکلی که پرسش‌های موجود در یک گروه، نیاز به زمان یکسانی برای حل شدن داشته و امتیاز یکسانی پس از حل شدن در پی خواهند داشت. تعداد گروه‌ها از ۱۰۰۰۰ تجاوز نمی‌کند ($1 \le N \le 10{,}000$). از آنجا که زمان مسابقه محدود است ($1 \le M \le 10{,}000$)؛ اساتید مورد اشاره، قصد

دارند پرسش‌ها را طوری برگزینند که مجموع امتیاز آن‌ها بیشینه باشد. از هر گروه می‌توان به تعداد دلخواه پرسش انتخاب کرد (حتی صفر). بیشترین مجموع امتیاز ممکن را که می‌توان از برگزیدن پرسش‌ها برای مسابقه بدست آورد، محاسبه کنید.

ورودی

در خط نخست ورودی، زمان مسابقه (M) و تعداد گروه‌ها (N) آورده می‌شود. در N خط بعدی از پرونده، دو مشخصه مورد نیاز برای هر گروه قرار می‌گیرد؛ یعنی امتیاز حاصل از حل پرسش‌های آن گروه و زمان مورد نیاز برای حل سوالاتش. ورودی زیر یک نمونه‌ی صحیح برای این مسأله است. در این نمونه، زمان مسابقه ۳۰۰ دقیقه و تعداد گروه پرسش‌ها ۴ است. پرسش‌های گروه اول در مدت زمان ۶۰ دقیقه قابل حل بوده و حل آن‌ها، ۱۰۰ امتیاز را در پی خواهد داشت.

Input
300 4 100 60 250 120 120 100 35 20

خروجی

بیشترین امتیاز ممکن که می‌توان از برگزیدن پرسش برای مسابقه به دست آورد، در خروجی نوشته می‌شود. توجه داشته باشید که برگزیدن پرسش‌ها باید به صورتی انجام شود که مجموع زمان مورد نیاز برای حل آن‌ها از زمان کل مسابقه تجاوز نکند. خروجی زیر مربوط به ورودی بالا است:

Output
605

برگزیدن دو پرسش از گروه ۲ ($2 * 250 = 500$) و برگزیدن سه پرسش از گروه ۴ ($3 * 35 = 105$) مجموع امتیاز ۶۰۵ را بدست می‌دهد که بیشینه‌ی امتیازات ممکن است.

راه‌حل

این پرسش در دسته‌ی مسائل کوله‌پشتی قرار می‌گیرد. در این دسته از مسأله‌ها، باید کوله‌پشتی را به شکلی پر کنیم که ارزش اشیاء موجود در آن بیشینه شده و از ظرفیت محدود کوله‌پشتی نیز تجاوز نکند. در این پرسش با محدودیت زمان مسابقه مواجه هستیم و باید در این محدوده‌ی زمانی، پرسش‌هایی را برگزینیم که مجموع امتیازات حاصل از حل آن‌ها بیشینه شود.

برای مسائل کوله‌پشتی صفر و یک، راه‌حل پویا راهکاری مناسب است. آرایه‌ای به طول مدت زمان مسابقه در نظر می‌گیریم. برای نمونه، محتوای خانه maxpoint[m] بیشترین مجموع امتیازی است که می‌توان برای مسابقه‌ای به مدت m دقیقه به دست آورد. مقداردهی این آرایه را از خانه‌ی اول آغاز کرده و به ترتیب برای به دست آوردن مقدار هر خانه، از خانه‌های پیشین استفاده می‌کنیم. فرض کنید time[i] و point[i] زمان و امتیاز گروه i_ام را مشخص می‌کنند. در این صورت الگوریتم پویای زیر راه‌حل مسأله خواهد بود:

می‌خواهیم maxpoint[m] را با فرض داشتن maxpoint[m-1] محاسبه کنیم. یا امکان اضافه کردن پرسشی جدید وجود دارد و یا ندارد. اگر نتوان هیچ پرسش جدیدی اضافه کرد، مقدار maxpoint[m] همان maxpoint[m-1] خواهد بود و غیر این صورت، پرسش جدید را اضافه کرده و امتیاز آن را با امتیاز بیشینه در زمان باقیمانده (زمان این پرسش از زمان کل مسابقه، کسر می‌شود) جمع می‌کنیم:

$$max\_point[m] = MAX(\ max\_point[m-1],\ max\_point[\ m - time[i]\ ] + point[i]\ )$$

کد زیر با استفاده از قاعده‌ی توضیح داده شده، راه‌حل کامل را ارائه می‌دهد:

```
#define MAX_N 10000 + 5
struct pgroup {
    int spoint;
    int stime;
} problemsg[MAX_N];
int max_point[MAX_N];
int max_minute, categories_num;
int find_max_point() {
    for (int m = 1; m <= max_minute; m++) {
        max_point[m] = max_point[m - 1];
        for (int c = 1; c <= categories_num; c++) {
            if (problemsg[c].stime <= m)
                max_point[m] = max(max_point[m],
                    max_point[m - problemsg[c].stime] +
problemsg[c].spoint);
        }
    }
    return max_point[max_minute];
}
int main() {
    fin >> max_minute >> categories_num;
    for (int i = 1; i <= categories_num; i++)
        fin >> problemsg[i].spoint >> problemsg[i].stime;
    fout << find_max_point() << endl;
    fin.close();
    fout.close();
    return 0;
}
```

٦-١٠- رمزنگاری بهینه

جمعی از مهندسین رایانه برای ارسال رشته‌ها به یکدیگر، روشی ساده و در عین حال بهینه برگزیده‌اند. در این روش، رشته‌ی مورد نظر را با استفاده از الگوریتم هافمن فشرده کرده، به دنباله‌ای از صفر و یک‌ها تبدیل و سپس برای یکدیگر ارسال می‌کنند. در طرف مقصد، رشته‌ی رمز شده رمزگشایی شده و سپس خوانده می‌شود. طول رشته‌ی ورودی حداکثر برابر با ۵۰۰ نویسه خواهد بود. رشته‌ی مورد نظر را از پرونده ورودی دریافت کرده و معادل رمز شده‌ی آن را ایجاد کنید. نویسه‌های رشته‌ی ورودی از حروف کوچک انگلیسی تشکیل شده‌اند.

ورودی

رشته‌ای حداکثر به طول ۵۰۰ نویسه در پرونده ورودی قرار می‌گیرد. ورودی زیر یک نمونه را نشان می‌دهد:

Input
thisisatest

خروجی

رشته‌ی رمز شده با به‌کارگیری الگوریتم هافمن، در پرونده خروجی نوشته می‌شود.

Output
010011111011110000011101001

کد یافت شده برای هر نویسه به صورت زیر است:

```
t 01    h 001    a 000
s 10    i 111    e 110
```

همان‌طور که می‌بینید، کد به دست آمده برای نویسه‌های t و s که فراوانی‌شان بیشتر از سایر نویسه‌ها است، کوتاه‌تر از سایرین است. این ویژگی موجب کوتاه‌تر شدن کد تولید شده توسط الگوریتم هافمن نسبت به روش‌های غیربهینه است.

راه‌حل

ایجاد کد هافمن با استفاده از روش توضیح داده شده در فصل ۴ انجام می‌شود. تنها بخش جدید راه‌حلی که در ادامه ارائه خواهد شد، یافتن کد مربوط به هر نویسه و نوشتن آن در پرونده خروجی است. ساختار node، کلاس

mycomparison، تابع huffman_coding و صف اولویت‌دار pq مطابق با بخش مربوط به الگوریتم هافمن هستند؛ به همین دلیل از تکرار آن‌ها خودداری می‌شود.

```
#include <queue>
#include <string>
#define MAX_C 26
struct node {
    char symbol;
    int weight;
    node* left;
    node* right;
};
string strinput;
int charcount[MAX_C];
string charcode[MAX_C];
void get_code(node* cnode, string strcode) {
    if (cnode->symbol == 0) {
        strcode.push_back('0');
        get_maked_code(cnode->left, strcode);
        strcode = strcode.substr(0, strcode.length() - 1);
        strcode.push_back('1');
        get_maked_code(cnode->right, strcode);
        strcode = strcode.substr(0, strcode.length() - 1);
    }
    else
        charcode[cnode->symbol - 'a'] = strcode;
}
void write_code() {
    for (int i = 0; i<strinput.length(); i++)
        fout << charcode[strinput[i] - 'a'];
}
int main() {
    fin >> strinput;
    for (int i = 0; i<strinput.length(); i++)
        charcount[strinput[i] - 'a'] ++;
    node* chars;
    for (int i = 0; i < MAX_C; i++) {
        if (charcount[i] == 0)
            continue;
        chars = new node();
        chars->symbol = 'a' + i;
        chars->weight = charcount[i];
        chars->left = chars->right = NULL;
        pq.push(chars);
    }
    get_code(huffman_coding(), "");
    write_code();
    fin.close();
    fout.close();
```

```
    return 0;
}
```

پس از دریافت ورودی، فراوانی هر نویسه را به دست می‌آوریم. سپس نویسه مربوطه را به همراه فراوانی آن، در صف اولویت‌دار قرار می‌دهیم. آنگاه، تابع get_code به صورت بازگشتی، کد مربوط به همه‌ی نویسه‌های موجود در رشته را به دست آورده و در آرایه‌ی charcode قرار می‌دهد. در پایان، تابع write_code شکل کد شده‌ی رشته‌ی ورودی را در پرونده‌ی خروجی می‌نویسد.

تمرین‌ها

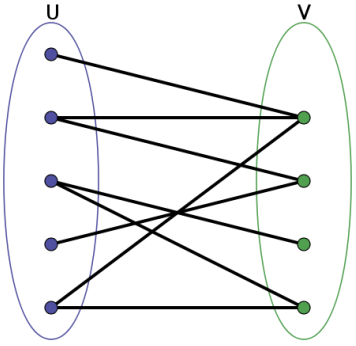
۱- نسخه‌های دیگری از مسأله‌ی هشت وزیر را در نظر بگیرید که در هرکدام از آنها به جای وزیر، از یکی از مهره‌های اسب، فیل، شاه یا قلعه استفاده شود. راه‌حل‌های لازم برای حل هرکدام از این مسائل را پیاده‌سازی کنید. در هر حالت حداکثر چند مهره را می‌توان در صفحه‌ی شطرنج قرار داد به طوری که هیچکدام از آنها دیگری را تهدید نکند؟

۲- حالت کلی‌تری از مسأله‌ی هشت وزیر، مسأله‌ی n-وزیر است که همانطور که از نام آن پیداست، در مورد n وزیر در یک صفحه‌ی شطرنج $n \times n$ است. در این مسأله همانند حالت هشت وزیر، باید وزیرها را به گونه‌ای در صفحه قرار داد که هیچکدام از آنها دیگری را تهدید نکند. راه حل ارائه شده در قسمت ۶-۱ را برای حل مسأله‌ی n-وزیر تعمیم داده و پیاده‌سازی کنید.

۳- روش وارسی پیش رو برای حل مسأله‌ی سودوکو را که در بخش ۶-۲ شرح داده شد، پیاده‌سازی کنید.

۴- راه حل ارائه شده برای حل مسأله‌ی رنگ‌آمیزی گراف در بخش ۶-۳ را پیاده‌سازی کنید.

۵- یک گراف دو بخشی گرافی است که بتوان رئوس آن را به دو مجموعه‌ی مجزای U و V تقسیم‌بندی کرد، به گونه‌ای که دو سر هیچ یالی درون یکی از این مجموعه‌ها قرار نداشته باشد. به عبارت دیگر هر کدام از یال‌های چنین گرافی، یکی از رئوس مجموعه‌ی U را به یکی از رئوس مجموعه‌ی V وصل می‌کند. نمونه‌ای از یک گراف دو بخشی در شکل ۸,۶ نشان داده شده است. با تحلیل راه حل مسأله‌ی رنگ‌آمیزی گراف، روشی برای تعیین دو بخشی بودن یا نبودن یک گراف دلخواه ارائه کنید.



شکل ۸,۶. نمونه‌ای از یک گراف دو بخشی

۶- مسأله‌ی مرتب‌سازی با کمترین تعداد جابجایی را در صورتی حل کنید که دنباله‌ی اعداد ورودی متشکل از اعداد ۱، ۲، ۳ و ۴ باشد.

۷- حالت دیگری از مسأله‌ی اتصال شهرها را در نظر بگیرید که در آن می خواهیم با بیشترین هزینه‌ی ممکن شهرها به یکدیگر متصل شوند. روشی برای حل این مسأله طراحی و پیاده‌سازی کنید.

۸- راه حل جدیدی برای حل مسأله‌ی بزرگترین کوتاهترین مسیر بر اساس جستجوی اول سطح طراحی و پیاده‌سازی کنید.

۹- در مسأله‌ی برگزیدن بهترین شهر برای زندگی، فرض کنید که درصد تعداد دفعاتی که شخص روزانه به هرکدام از شهرها مسافرت می‌کند نیز در ورودی مشخص شود. با ایجاد تغییرات لازم در راه حل ارائه شده در قسمت ۷-۶، بهترین شهر برای زندگی را در این حالت به دست آورید.

مراجع

Arefin, A.S., 2006. *Art of Programming Contest*, Reviewed by Dr. M. Lutfar Rahman and Forworded by Professor Miguel Revilla, University de Valladolid, Spain, Gyankosh Prokashoni, Dhaka, Bangladesh, First Ed., ISBN: 984-32-3382-4.

Bondy, John Adrian, and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*. Vol. 290. London: Macmillan, 1976.

Cormen, T.H. et al., 2009. *Introduction to algorithms* 3rd ed., MIT Press and McGraw-Hill.

Jordan, B., Brett, S. (2009). "*A survey of known results and research areas for n-queens*". Discrete Mathematics 309 (1). pp. 1–31.

D.A. Huffman, "*A Method for the Construction of Minimum-Redundancy Codes*", Proceedings of the I.R.E., September 1952, pp 1098–1102.

Jean-Paul, D., "*The Science Behind Sudoku*", Scientific American magazine, June 2006.

Skiena, S.S. & Revilla, M.A., 2006. *Programming challenges: The programming contest training manual*, Springer Science & Business Media.

فصل ۷- مهارت در مسابقات

این فصل که فصل آخر این کتاب نیز است، مختص برنامه‌نویس‌هایی است که قصد شرکت در مسابقات برنامه نویسی (مانند اِی-سی-ام) را دارند. در آغاز این فصل، نکته‌هایی کلی درباره‌ی مسابقات، روش عملکرد و برنامه‌ریزی برای مسابقه و نکته‌هایی درباره‌ی دسته‌بندی پرسش‌ها آورده می‌شود. سپس دسته‌بندی موضوعی پرسش‌ها و توضیح کوتاهی درباره‌ی هر یک از آن‌ها آورده خواهد شد. چند سایت فعال در زمینه مسابقات برنامه نویسی نیز معرفی می‌شود. آنگاه به معرفی اجمالی نرم‌افزار پی-سی-تو[۵۷] که برای داوری مسابقات جهانی اِی-سی-ام استفاده می‌شود، پرداخته شده است. در پایان، مجموعه سوالات چند مسابقه‌ی برنامه‌نویسی اِی-سی-ام آورده شده و سپس راه‌حل آن‌ها ارائه گردیده است. سوالات عمدتا متعلق به مسابقات داخلی برگزار شده در ایران هستند.

۷-۱- نقشه‌ای کلی برای مسابقه

بهترین روش برای طی کردن یک «مسیر موفق» در رسیدن به «موفقیت در مسابقه»، به‌کارگیری یک «نقشه‌ی از پیش تعیین شده» است. این نقشه عملکرد شما را در هنگام رسیدن به مشکل یا موفقیت تعیین می‌کند؛ بنابراین شما به عملی که در مسابقه، هنگام برخورد به مشکل یا پیدا کردن راه‌حل درست انجام خواهید داد، کاملا مسلط خواهید بود.

در آغاز همه‌ی پرسش‌ها را سریع بخوانید و نکاتی را از همه‌ی آن‌ها استخراج کنید: الگوریتم، پیچیدگی، اعداد، ساختمان‌های داده‌ای، جزئیات زیرکانه پرسش و غیره. یک استراتژی مناسب می‌تواند به این صورت باشد:

- الگوریتم‌های ممکن را بررسی کرده و ساده‌ترین آن‌ها را برگزینید.
- پیچیدگی فضایی و زمانی الگوریتم را محاسبه و مواردی که الگوریتم جوابگوی آن‌ها است و یا به مشکل برمی‌خورد را پیش‌بینی کنید.
- کوشش کنید الگوریتم را بشکنید (اشکال آن را پیدا کنید).
- پرسش‌ها را به ترتیب زیر مرتب کنید:
  - تکراری بودن
  - سادگی

---

[۵۷] PC²

- ناآشنا بودن
- سخت بودن

پس از انجام مراحل مذکور، یک مسأله را برگزیده و حل آن را به این ترتیب آغاز کنید:

- الگوریتم انتخابی را نهایی کنید.
- داده‌های آزمونی برای حالت‌های گول‌زننده و ظریف برگزینید.
- ساختمان‌های داده‌ای مورد نیاز را انتخاب کنید.
- کد دریافت ورودی را نوشته و آن را آزمایش کنید (ورودی دریافتی را با کمی کد اضافه بسنجید).
- کد مربوط به خروجی داده‌ها را نوشته و آزمایش کنید.
- برنامه را به صورت گام به گام آزمایش کنید (با قرار دادن داده‌های خروجی آزمایشی در هر بخش).
- بررسی کنید که آیا برنامه درست کار می‌کند؟ (داده‌ی آزمایشی خود سوال را به کار ببرید).
- کوشش کنید اشکال برنامه‌ای که نوشته‌اید را بیابید (داده‌های آزمایشی مورد نظرتان را به کار ببرید).
- برنامه را بخش به بخش بهینه کنید (در حدی که نیاز است: بسته به محدودیت زمانی پرسش و ...). برای آزمایش کارکرد صحیح برنامه در «زمان مورد نظر پرسش»، بزرگ‌ترین داده‌های آزمایشی را امتحان کنید تا نحوه عملکرد برنامه‌ی شما در بدترین حالت ممکن مشخص شود.

۷-۲- استراتژی مدیریت زمان

یک نقشه برای مشکلات گوناگونی که در حین مسابقه به آن برمی‌خورید، تهیه کنید (این مشکلات قابل پیش‌بینی هستند). مشکلاتی که در حین مسابقه به آن برخواهید خورد و باید برای مقابله با آن، عکس‌العملی از خود نشان دهید، تصور کنید. پرسش اصلی این است که «چه زمانی باید زمان بیشتری را برای رفع اشکال یک برنامه صرف کرد و چه زمانی باید از آن گذشت و به سراغ پرسش بعد رفت؟».

موضوعات زیر را درنظر بگیرید:

- چه مدتی را تا به حال برای مسأله‌ی فعلی سپری کرده‌اید؟
- تصور می‌کنید برنامه‌تان چه نوع اشکالی دارد؟
- آیا الگوریتم‌تان اشتباه است؟
- آیا ساختمان داده‌ای برنامه‌تان باید تغییر کند؟
- آیا هیچ اطلاعی درباره‌ی بخش اشتباه برنامه دارید؟ (که شما را در پیدا کردن آن راهنمایی کند)
- صرف زمان کوتاهی برای رفع اشکال برنامه (کمتر از ۲۰ دقیقه) بهتر از هر کار دیگری است. زیرا احتمالا شما نمی‌توانید برنامه دیگری را در کمتر از ۴۵ دقیقه بنویسید.
- چه زمانی باید به سراغ برنامه‌ای رفت که یک مرتبه در حل آن با شکست مواجه شده‌اید؟
- چه زمانی باید زمان بیشتری را برای بهینه‌سازی یک برنامه صرف کرد؟ و چه زمانی باید از آن گذشت؟
- از حالا آغاز کنید، گذشته را فراموش کنید و به آینده متمرکز شوید. چگونه می‌توانید از ساعت‌هایی که در اختیار دارید، بهتر استفاده کرده و امتیازات بیشتری بدست بیاوردید؟

پیش از ثبت یک پاسخ به نکته‌های زیر توجه کنید:

- مدت زمانی که از مسابقه باقی مانده است.
- خروجی‌های آزمایشی را (که برای اشکال‌زدایی اضافه کرده‌اید) حذف کنید.

۷-۳- نکته‌ها و فوت و فن‌های مهم

- در مواقعی که استفاده از جستجوی جامع (آزمایش همه‌ی حالت‌های ممکن برای رسیدن به پاسخ) امکان پذیر است، آن را به کار ببرید.
- ساده کد نوشتن یک مهارت است!
- به محدودیت‌های پرسش توجه کنید (محدودیت‌ها زیرکانه تعیین می‌شوند).
- اگر استفاده از حافظه‌ی بزرگ، کار را ساده می‌کند، آن را به کار ببرید (اگر می‌توانید از هدر رفتن آن جلوگیری کنید، این کار را بکنید)!
- کدهای اضافی که برای آزمایش برنامه اضافه کرده‌اید حذف نکنید، بلکه آن‌ها را به توضیح تبدیل کنید.
- فقط تا جایی که مورد نیاز سوال است، برنامه را بهینه‌سازی کرده و از صرف زمان بیشتر برای آن خودداری کنید.
- ممکن است پس از حل پرسش، راه حل ساده‌تری نیز به ذهنتان خطور کند و نسخه‌های جدیدتری از برنامه را بنویسید؛ نسخه‌های قدیمی را حذف نکنید.
- نکاتی برای کدنویسی:
  - فضاهای خالی میان کدها، خوانایی را بالا می‌برد.
  - از نام‌های معنادار برای متغیرها استفاده کنید.
  - از نام‌های تکراری استفاده نکنید.
  - پالایش گام به گام. هر بخش از کد را جداگانه آزمایش کنید.
  - پیش از بخش‌های مهم کد، توضیح بنویسید.
- از اشاره‌گرها تا جایی که می‌توانید دوری کنید.
- از گرفتن حافظه‌های پویا دوری کنید. همه فضاهای مورد نیاز را به صورت ایستا تعریف کنید.
- از به‌کارگیری اعداد اعشاری پرهیز کنید. ولی اگر مجبورید، از مقایسه تساوی آن‌ها خودداری کنید و یا روشی برای مقایسه‌ی درست آن‌ها (تساوی با ضریب خطا) در نظر بگیرید.
- نکته‌هایی درباره‌ی نوشتن توضیحات:
  - توضیح باید کوتاه باشد، نه نثری مسجع!
  - توضیحات باید موارد کلی را توضیح دهند، نه موارد جزئی (مثلا نوشتن توضیح برای کد i++ اشتباه است و هیچ کمکی نخواهد کرد).
  - کدها را زیرکانه تشریح کنید.
  - هر بخشی که برای فهمش نیاز به تفکر است، نیاز به توضیح دارد.
  - درباره‌ی آرایه‌ها و نمایه‌هایش، توضیحات می‌توانند کمک خوبی باشند.

- نکته‌هایی از هر مسابقه تهیه کرده و کارایی خود را درباره‌ی پرسش‌های مختلف آن بسنجید: موفقیت، اشتباه و مواردی که برای انجام بهتر کارها می‌تواند کمک کند. این نکته‌ها کمک بسیار خوبی برای ارتقا قدرت برنامه‌نویسی شما هستند.

۷-٤- قاعده‌هایی مبتنی بر تجربه

- اگر قصد دارید مدت زمان اجرای برنامه را به ازای مجموعه‌ای از داده‌ها محاسبه کنید، نخستین نکته این خواهد بود: رایانه‌های امروزی می‌توانند تعداد ۱۰۰ میلیون عمل را در هر ثانیه انجام دهند (دست کم). بنابراین در برنامه‌ای که محدودیت ۵ ثانیه‌ای برای اجرای آن تعیین شده است، می‌توانید تا ۵۰۰ میلیون عمل را در نظر داشته باشید. اغلب سوالات در مسابقات، محدودیت ۱ ثانیه‌ای برای تولید خروجی (پس از دریافت ورودی) دارند.
- بیشترین به‌کارگیری حافظه: ۱۸ مگابایت
- 2^{10} تقریبا برابر است با 10^3.
- اگر برنامه‌ی شما دارای k حلقه‌ی تودرتو است که هر کدام n بار تکرار می‌شوند، پیچیدگی زمانی آن $O(n^k)$ خواهد بود.
- اگر برنامه‌ی شما بازگشتی است و b مرتبه خود را فراخوانی می‌کند و این عمل را l بار تکرار می‌شود، پیچیدگی آن $O(b^l)$ خواهد بود.
- بهترین پیچیدگی زمانی برای مرتب کردن یک آرایه n تایی، $O(n \log n)$ است.
- مراقب محدوده‌ی اعداد باشید!

چند نمونه:

یک حلقه با n بار تکرار دارای پیچیدگی زمانی $O(n)$ است:

```
sum = 0
for i = 1 to n
     sum = sum + i
```

دو حلقه تودرتو دارای پیچیدگی زمانی $O(n^2)$ خواهد بود:

```
// fill array a with N elements
for i = 1 to n - 1
   for j = i + 1 to n
      if (a[i] > a[j])
         swap(a[i], a[j])
```

۷-۵- الگوهایی از راه‌حل‌های مختلف

فیلتر کردن در مقابل ایجاد کردن

برنامه‌هایی که تعداد زیادی از پاسخ‌ها را ایجاد کرده تا پاسخ درست را بیابند، روش فیلترکردن را به کار می‌برند (مانند مسأله‌ی هشت‌وزیر). در برابر این روش، به روشی که به طور مستقیم و بدون به‌کارگیری هیچ فیلتری پاسخ های درست را می‌سازد، روش ایجاد کردن می‌گوییم. روش فیلترکردن به راحتی (و سریع) پیاده‌سازی می‌شود ولی به کندی اجرا می‌شود. اگر روش فیلتر کردن در محدوده‌ی زمانی مشخص شده پاسخگوی سوال است؛ آن را به کار ببرید و در غیر این صورت باید مستقیما پاسخ‌های درست را ایجاد کنید. به بیانی دیگر باید یک ایجادکننده بنویسید.

از پیش محاسبه کردن

گاهی ایجاد یک جدول از داده‌ها یا ساختارهایی از آن‌ها، امکان رسیدن به نتیجه را بسیار سریع‌تر می‌کند. در این روش، فضا را فدای زمان می‌کنیم. این داده‌های از پیش محاسبه شده می‌توانند پیش از اجرای برنامه ایجاد شده و در برنامه به کار روند؛ و یا در آغاز برنامه محاسبه شده و به کار گرفته شوند. برای نمونه در برنامه‌ای که قرار است حروف بزرگ را به حروف کوچک تبدیل کند، استفاده از یک جدول، کار را بسیار سریع و بی‌نیاز از به‌کارگیری هر شرطی می‌کند؛ و یا در برنامه‌ای که با اعداد اول سر و کار دارد، ایجاد فهرست بزرگی از اعداد اول و به‌کارگیری آن‌ها در برنامه، در اکثر اوقات عملی و پرکاربرد است.

تجزیه

در مسابقات برنامه‌نویسی، به صورت کلی کمتر از ۲۰ الگوریتم اصلی وجود دارد. بنابراین طرح پرسش‌هایی که حل آن‌ها نیازمند به استفاده از ترکیبی از دو یا چند الگوریتم است، طبیعی به نظر می‌رسد. بخش دشوار کار، شناسایی الگوریتم‌های مورد نیاز پرسش و به‌کارگیری ترکیبی آن‌ها است. برای نمونه، گاهی شما می‌توانید یک الگوریتم را با یک حلقه و یا یک الگوریتم دیگر ترکیب کنید. توجه داشته باشید که گاهی نیاز است یک الگوریتم را دوبار، در دو بخش مستقل از داده‌هایتان به کار ببرید تا زمان اجرای برنامه را کاهش دهید.

تقارن

بسیاری از از برنامه‌ها متقارن هستند، یعنی چند بخش کاملا مشابه دارند. برنامه‌های متقارن می‌توانند دارای دو، چهار، هشت یا تعداد بیشتری از بخش‌های متقارن باشند. بنابراین کوشش کنید تقارن‌ها را بیابید تا زمان اجرای برنامه را کاهش دهید. برای نمونه، برای پیاده‌سازی برنامه‌ای متقارنی که دارای چهار بخش متقارن است، کافی است یکی از بخش‌های (یک چهارم) آن را پیاده‌سازی کرده و به نتیجه‌ی دلخواه دست پیدا کنید.

روش بازگشتی در مقابل روش رو به جلو

بسیاری از پرسش‌ها در مسابقات، از روش بازگشتی ساده‌تر و بهتر حل می‌شوند تا با استفاده از روش رو به جلو (معمولی). البته در هنگام به‌کارگیری روش‌های بازگشتی و روش‌هایی که با الگویی ویژه در داده‌ها به جستجو می پردازند، باید خیلی مراقب بود. بنابراین برای یافتن راه‌حل پرسش‌ها، این روش را نیز مدنظر داشته باشید.

ساده‌سازی

بعضی از پرسش‌ها می‌توانند به پرسش‌های دیگری کاهش یابند که با حل آن‌ها، می‌توان به پاسخ پرسش اصلی رسید (کل به جزء). البته باید دید میان این دو (پرسش جدید و پرسش اصلی)، پیاده‌سازی کدام یک ساده‌تر است! با استفاده از این روش، گاهی می‌توان با ایجاد کمی تغییر در برنامه‌ای کوچک‌تر به پاسخ برنامه‌ی اصلی رسید (جزء به کل).

۷-٦- دسته‌بندی مسأله‌ها

بر اساس بررسی‌های انجام شده پیرامون مسأله‌های مسابقات برنامه‌نویسی، چند موضوع کلی وجود دارند که همیشه مورد توجه طراحان قرار گرفته‌اند. تقسیم‌بندی زیر اغلب این موضوعات را پوشش می‌دهد. موضوعاتی که ممکن است کمی ناآشناتر باشند، به صورت جزئی توضیح داده شده‌اند.

۱. برنامه‌نویسی پوی

۲. الگوریتم‌های حریصان

۳. جستجوی جامع

نخستین راه حلی که برای حل پرسش‌ها باید مدنظر داشته باشید، جستجوی جامع است. اگر زمان و فضای تعیین شده برای مسأله اجازه‌ی استفاده از این روش را می‌داد، حتما آن را به کار ببرید، زیرا به سادگی می‌توان آن را پیاده‌سازی و اشکال‌زدایی نمود.

٤. پرکردن جریانی

یافتن تعداد گراف‌های همبند در یک گراف غیرهمبند به پرکردن جریانی معروف است. در این‌گونه مسائل، گراف غیرهمبند ورودی پیمایش شده و در میان گراف‌های همبند موجود در آن به دنبال پاسخ مسأله می‌گردیم. با استفاده از دو روش اول‌عمق و اول‌سطح می‌توان مانند یک جریان، گره‌های گراف را پیمایش کرد. به فرآیند جستجوی گره‌ها و گذشتن از گره‌های دیده شده، «پرکردن» گفته می‌شود؛ پیش‌روی از هر گره به گره‌ی دیگر نیز جریان نامیده می‌شود.

٥. کوتاهترین مسی

٦. جستجوی بازگشت

۷. درخت پوشای کمینه

۸. مسائل کوله پشتی

۹. هندسه‌ی محاسبات

این گونه مسأله‌ها می‌توانند از سخت‌ترین نوع پرسش‌ها باشند، زیرا گاهی به درک هندسی و محاسباتی بالایی نیاز دارند.

۱۰. جریان شبکه‌ای

یافتن بیشترین جریان ممکن از مبدا به مقصد در شبکه (گراف)، جریان شبکه‌ای نامیده می‌شود.

۱۱. مسیر اویلری[۵۸]

۱۲. پوشش محدب دوبعد

دریافت مجموعه‌ای از نقاط از ورودی و یافتن یک چندضلعی محدب (با کمترین تعداد ضلع) که نقاط داده شده را پوشش دهد؛ به شکلی که همه نقاط، درون آن و یا برروی ضلع‌ها واقع شوند. روشن است که اضلاع چندضلعی باید از اتصال نقاط تشکیل شوند.

۱۳. اعداد بزرگ

برخی از مسائل مربوط به انجام محاسبات ریاضی بر روی اعداد با اندازه‌های بسیار بزرگ می‌شوند؛ اعدادی که ذخیره‌سازی آنها در نوع‌های داده‌ای معمولی امکان‌پذیر نیست. کار با اعداد بزرگ به چند بخش تقسیم می‌شود:

- مقایسه اعداد بزرگ
- اعمال ریاضی بر روی اعداد بزرگ: جمع، تفریق، ضرب، تقسیم، توان و غیره.

۱۴. جستجوی اکتشافی

این جستجو توسط یک تابع اکتشافی به پیشروی در درخت جستجو می‌پردازد. تابع اکتشافی میزان نزدیک بودن حالت‌ها به پاسخ را تعیین می‌کند. توضیحاتی که در بخش جستجوی عقب‌گرد داده شد، می‌تواند در این زمینه کارا باشد.

۱۵. الگوریتم‌های تقریبی

در بعضی از مسائل، یافتن پاسخ دقیق ممکن نیست. در این حالت‌ها هدف از جستجو، یافتن تقریبی از پاسخ است.

۱۶. برنامه‌های خاص

این دسته از مسأله‌ها در هیچ یک از گروه‌های عمومی دسته‌بندی نمی‌شوند. هر برنامه‌ی خاص، متفاوت از بقیه است. هریک از این مسائل، دارای روش حل مربوط به خود هستند. ممکن است به یک ساختمان‌داده‌ی ویژه یا حلقه و شرطی ویژه نیاز داشته باشند. این گونه مسأله‌ها را باید به دقت خواند و نکته‌های موجود در متن پرسش را به صورت کامل استخراج کرد. امکان نیاز به بهینه‌سازی نیز برای این مسأله‌ها وجود دارند؛ بنابراین به سطحی از تحلیل نیاز است.

اغلب مسأله‌ها، از موضوعات ذکر شده و یا ترکیبی از آن‌ها طرح می‌شوند. با تسلط بر حدود چهل درصد از مباحث یاد شده، می‌توان موفقیت همیشگی در مسابقات را تضمین کرد. البته تسلط امری نسبی است و هر چه بیشتر شود، موثرتر است.

---

[۵۸] Eulerian path

برای تبدیل شدن به یک برنامه‌نویس ماهر، باید با یک زبان برنامه‌نویسی، ساختمان‌داده‌های مورد نیاز و الگوریتم‌های معمول آشنا شده و بر آن‌ها تسلط پیدا کنید؛ سه عاملی که در این کتاب به تفصیل به آنها پرداخته شد؛ در ادامه‌ی این مسیر، با تمرین و استمرار در حل مسأله‌های گوناگون، به موفقیت حتمی دست خواهید یافت.

۷-۷- مسابقات معروف و سایت‌های مرتبط

برای علاقمندان به برنامه‌نویسی شرکت در مسابقات مرتبط تمرینی ایده‌آل برای تقویت مهارت برنامه‌نویسی محسوب می‌شود. تعداد زیادی از وب‌سایت‌ها در این رابطه فعالیت می‌کنند و کاربران بسیاری نیز دارند. در این بخش، تعدادی از آن‌ها و مسابقات برگزار شده توسط آن‌ها را معرفی خواهیم کرد. موارد معرفی شده تنها بخش کوچکی از مجموعه‌ی بزرگ این سایت‌ها هستند.

۷-۷-۱- مسابقات اِی-سی-ام

مسابقه‌ی برنامه‌نویسی دانشجویی اِی-سی-ام از بزرگترین مسابقات سالیانه‌ی برنامه‌نویسی است که از سال ۱۹۷۷ برگزار می‌شود و تأکید آن بر تقویت مهارت‌های برنامه‌نویسی، کار گروهی و توان حل مسأله است. این رقابت علمی بین‌المللی که در نوع خود معتبرترین مسابقه محسوب می‌شود، دانسته‌های علمی، مهارت‌های برنامه‌نویسی و توان کار گروهی دانشجویان دانشگاه‌های جهان را در شش قاره به رقابت می‌گذارد.

اِی-سی-ام حروف اختصاری انجمن ماشین‌های محاسبه‌ای[۵۹] است که اولین انجمن علوم رایانه در دنیا بوده و در سال ۱۹۴۷ تاسیس شده است. مسابقات برنامه‌نویسی که توسط این انجمن به صورت سالیانه برگزار می‌شود، به نام مسابقات اِی-سی-ام یا اِی-سی-ام-آی-سی-پی-سی[۶۰] شناخته می‌شود (آی-سی-پی-سی مخفف مسابقات برنامه‌نویسی سالیانه‌ی بین‌المللی[۶۱] است). نماد اصلی این مسابقات در شکل ۱,۷ نشان داده شده است. سه بخش موجود در این شکل، به معنای تفکر، خلاقیت و حل مسأله هستند.

شکل ۱,۷. نماد اصلی مسابقات برنامه‌نویسی اِی-سی-ام

[۵۹] Association for Computing Machinery

[۶۰] ICPC

[۶۱] International Collegiate Programming Contest

اطلاعات تکمیلی در مورد این مسابقه از وب سایت اصلی به آدرس https://icpc.baylor.edu/ آن قابل دسترسی است.

لازم به ذکر است که این مسابقات هرساله در یکی از دانشگاه‌های منتخب ایران نیز برگزار می‌شود و به همین دلیل ایران نیز دارای یک سایت اختصاصی مربوط به این مسابقات است. اخبار مربوط به برگزاری این مسابقات در ایران از وب‌سایت اختصاصی مذکور به آدرس http://acm.blog.ir/ قابل دسترسی است.

۷-۷-۲- ACM-ICPC Live Archive

این سایت شامل آرشیو سوالات همه مسابقات منطقه‌ای و جهانی اِی-سی-ام است. می‌توانید مسابقه‌ی برگزار شده در سال و کشور دلخواه خود را انتخاب کرده و سوالات مربوط به آن را مشاهده کنید. این سایت که سوالات مربوط به مسابقات برگزار شده از سال ۱۹۸۸ تا زمان حال را در خود جای داده است، در آدرس https://icpcarchive.ecs.baylor.edu/ موجود است.

۷-۷-۳- Google Code Jam

مسابقات برنامه‌نویسی دیگری نیز وجود دارند که طرفداران قابل توجهی به خود جلب کرده‌اند. از جمله‌ی این مسابقات می‌توان به مسابقه‌ی برگزار شده توسط شرکت گوگل اشاره کرد. این مسابقه یازده سال است که برگزار می‌شود و دارای قوانین نسبتا متفاوتی در مقایسه با مسابقات اِی-سی-ام است. این مسابقه دارای چهار دوره‌ی برخط و یک مرحله‌ی نهایی حضوری است. ثبت‌نام در این مسابقه برای همه آزاد است. برای کسب اطلاعات بیشتر در مورد این مسابقه می‌توانید به آدرس https://code.google.com/codejam/ مراجعه کنید.

۷-۷-۴- CodeForces

این سایت از فعال‌ترین‌ها در زمینه‌ی برگزاری مسابقات برنامه‌نویسی است. قوانین مسابقات این سایت بسیار جالب بوده و در دو دسته یک و دو برگزار می‌شود. کاربران با شرکت در مسابقات امتیاز کسب می‌کنند و با رسیدن به امتیاز مشخصی، از دسته‌ی دو به دسته‌ی یک ارتقا می‌یابند. این سایت، به صورت منظم به برگزاری مسابقات مختلف می‌پردازد که اغلب مدت زمان آن‌ها از دو ساعت تجاوز نمی‌کند. از این رو، امکان شرکت در مسابقات برای اغلب دانشجویان فراهم است. اشخاصی که در زمان مقرر موفق به شرکت در مسابقه نمی‌شوند، می‌توانند به صورت مجازی پس از زمان مسابقه در آن شرکت کرده و نتیجه کسب شده توسط خود را با سایرین مقایسه کنند.

در حین مسابقه، هر شرکت‌کننده پس از اتمام حل یک سوال می‌تواند به هک کردن پاسخ‌های ارسال شده توسط سایر شرکت‌کنندگان بپردازد. هک اصطلاحی است که این سایت به یافتن اشکال در راه‌حل سایرین اختصاص داده است. این سایت از آدرس http://codeforces.com/ قابل دسترسی است.

۷-۷-۵- ShareCode

این سایت توسط چند تن از دانشجویان دانشگاه تهران پیاده‌سازی شده و قابلیت برگزاری مسابقات آنلاین را دارد. این سایت ابزار مناسبی برای حل سوالات اِی-سی-ام به چندین زبان برنامه‌نویسی مختلف است. از جمله‌ی این

زبان‌ها می‌توان به سی، سی++، جاوا[۶۲]، پاسکال[۶۳] و پایتون[۶۴] اشاره کرد. این سایت از آدرس http://sharecode.ir/ در دسترس است.

۷-۷-۶- Timus Online Judge

این سایت از ابزارهای آنلاین داوری باسابقه به حساب می‌آید. در این سایت، می‌توانید علاوه بر حل سوالات موجود در آرشیو آن، مسابقه‌ی برخط ایجاد کنید؛ با انتخاب چند سوال از آرشیو سوالات، مدت مسابقه و زمان برگزاری آن به سادگی یک مسابقه‌ی برخط ایجاد می‌شود. بیش از ده زبان برنامه‌نویسی گوناگون توسط این سایت پشتیبانی می‌شوند. علاوه‌بر زبان‌های معمول، مواردی چون گو[۶۵]، سی‌شارپ[۶۶]، وی-بی-دات‌نت[۶۷]، روبی[۶۸]، پایتون و هاسکِل[۶۹] نیز در این فهرست دیده می‌شود. این سایت در آدرس http://acm.timus.ru/ قابل دسترسی است.

۷-۷-۷- TJU Online Judge

وجود بیش از چهار هزار مسأله در آرشیو سوالات این سایت، از نقاط قوت آن محسوب می‌شود. این سایت نیز قابلیت ایجاد مسابقه‌ی برخط را دارد. این سایت تنها از سه زبان برنامه‌نویسی سی، سی++ و جاوا پشتیبانی می‌کند و از آدرس http://acm.tju.edu.cn/toj/ قابل دسترسی است.

۷-۸- مسابقات برنامه‌نویسی بیان

شرکت بیان از جمله شرکت‌های ایرانی است که در چند سال اخیر در زمینه‌ی برگزاری مسابقات برنامه‌نویسی فعال بوده است. قوانین مسابقات این شرکت کمی با قوانین مسابقات اِی-سی-ام متفاوت است. مسابقات میزبانی شده توسط این شرکت در سطح بین‌المللی برگزار می‌شود. ثبت نام در این مسابقات کاملا رایگان است. اطلاعات بیشتر در مورد این مسابقه در وب سایت اصلی آن در آدرس http://contest.bayan.ir/fa/ موجود است.

۷-۷-۹- UVa Online Judge

این ابزار علاوه بر داوری برخط و آرشیوی غنی از سوالات، دارای مجموعه‌ای از سوالات موضوعی است. با استفاده از این مجموعه سوالات، می‌توانید در هر موضوع دلخواه به سادگی به سوالات گوناگون دسترسی داشت باشید. این وب سایت در آدرس http://uva.onlinejudge.org/ قابل دسترسی است.

۷-۷-۱۰- Virtual Judge

---

[۶۲] Java
[۶۳] Pascal
[۶۴] Python
[۶۵] Go
[۶۶] C#
[۶۷] VB.NET
[۶۸] Ruby
[۶۹] Haskell

این سایت در واقع یک ابزار داوری برخط نیست، بلکه قابلیت داوری سوالات متعلق به داورهای برخط دیگر را دارد. با استفاده از این سایت می‌توانید مسابقه‌ای شامل سوالات متعلق به وب‌سایت‌های زیر ایجاد کنید:

POJ ZOJ UVALive SGU URAL HUST SPOJ HDU HYSBZ UVA CodeForces Z-Trening Aizu LightOJ UESTC NBUT FZU CSU SCU

اگر نیاز به استفاده از سوالات مختلف از آرشیوهای گوناگون دارید، این سایت ابزاری منحصر به فرد برای شما خواهد بود. این سایت در آدرس http://acm.hust.edu.cn/vjudge/toIndex.action/ قابل دسترسی است.

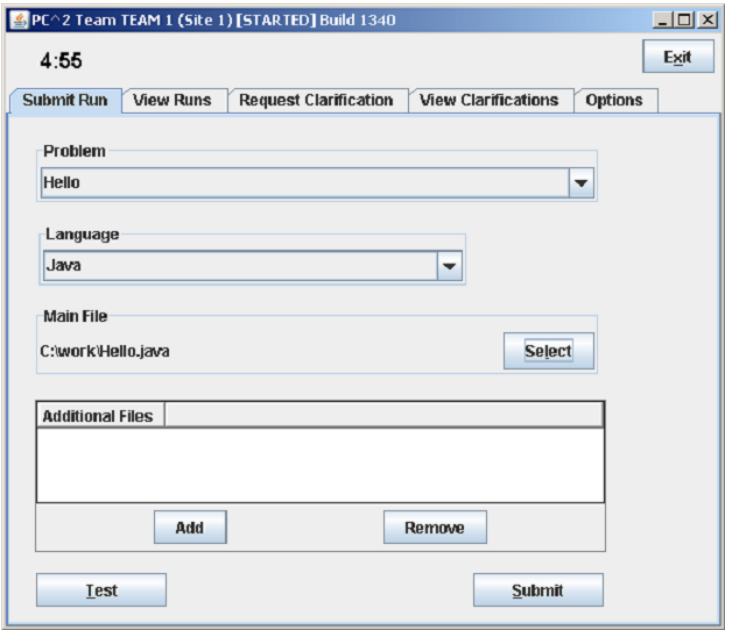
۷-۸- نرم‌افزار پی-سی-تو

نرم‌افزار پی-سی-تو سیستمی برای اجرای مسابقات برنامه‌نویسی است که به زبان جاوا پیاده‌سازی و تا به حال نه نسخه از آن ارائه شده است. این نرم‌افزار در مسابقات منطقه‌ای و جهانی اِی-سی-ام برای برگزاری مسابقات مورد استفاده قرار می‌گیرد. در ادامه توضیحات مختصری در مورد نحوه‌ی استفاده از آن داده شده است. برای آشنایی بیشتر با این ابزار قدرتمند می‌توانید به وب‌سایت آن به آدرس http://www.ecs.csus.edu/pc2/ مراجعه کنید.

شرکت‌کنندگان در ابتدای کار باید با نام کاربری و رمز عبوری که در اختیارشان قرار داده شده، وارد نرم‌افزار شوند (شکل ۲,۷).

شکل ۲,۷. صفحه‌ی اول نرم‌افزار پی-سی-تو

بعد از ورود موفقیت‌آمیز به نرم‌افزار، با صفحه‌ی اصلی نرم‌افزار مواجه خواهید شد (شکل ۳,۷). در سمت چپ-بالای پنجره زمان باقیمانده از مسابقه را مشاهده می‌کنید. این پنجره از ۵ سربرگ تشکیل شده است که در ادامه به تفکیک توضیح داده شده‌اند.

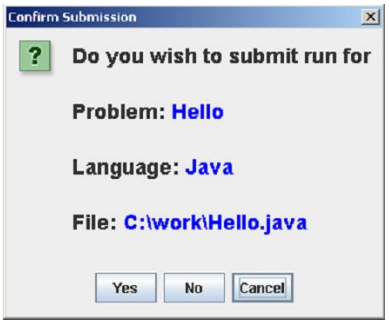


شکل ۳,۷. صفحه‌ی اصلی نرم‌افزار پی-سی-تو

سربرگ Sumbit Run

پس از حل یک سوال باید آن را از طریق سربرگ Submit Run برای داوری ارسال کنید. از فهرست بازشوی Problem، مسأله‌ی مورد نظر و از فهرست بازشوی Language، زبان برنامه‌نویسی مورد نظر را انتخاب کنید. فایل شامل کد برنامه را توسط دکمه Select انتخاب کرده و پس از حصول اطمینان از انتخاب درست همه‌ی موارد اشاره شده، بر روی دکمه‌ی Submit کلیک کنید. دقت کنید که تنها فایل کد برنامه را انتخاب کنید؛ نیازی به انتخاب فایل اجرایی و یا سایر فایل‌های مرتبط با برنامه نیست (شکل ۳,۷).

پس از کلیک بر روی دکمه‌ی Submit، پنجره‌ی نشان داده شده در شکل ۴,۷ ظاهر خواهد شد تا برای بار آخر به شما حق انتخاب برای ارسال پاسخ را بدهد.



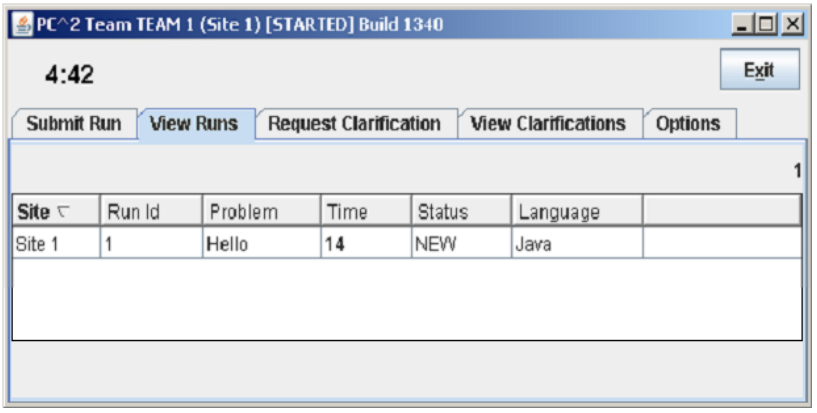
شکل ٤,۷. پنجره‌ی بررسی نهایی قبل از ارسال فایل کد برنامه

پس از ارسال شدن فایل و دریافت آن توسط داوران، پیامی مبنی بر دریافت فایل داده خواهد شد (شکل ۵,۷).

شکل ۵,۷. پیام مربوط به دریافت فایل کد برنامه توسط داوران

سربرگ View Runs

وضعیت پاسخ‌های ارسال شده برای داوری در سربرگ View Runs قابل مشاهده هستند. در ابتدا که یک پاسخ ارسال می‌شود، وضعیت آن در حالت NEW نمایش داده می‌شود (شکل ۶,۷).

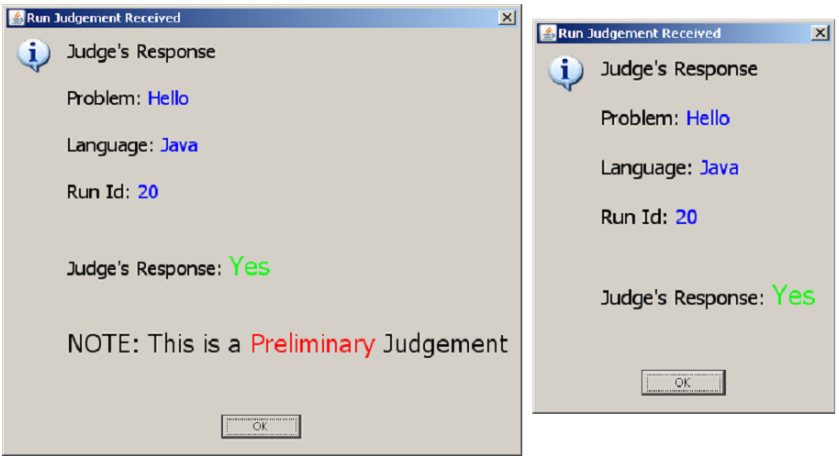


شکل ٦,۷. مشاهده‌ی وضعیت ارسال‌های قبلی در سربرگ View Runs

پس از اتمام داوری، نتیجه‌ی آن به یکی از دو شکل زیر به اطلاع شرکت‌کننده خواهد رسید.

- اعلام نتیجه‌ی اولیه: نتیجه‌ی اولیه در واقع «بهترین تخمین» از نتیجه‌ای است که داوران تصمیم خواهند گرفت. این نتایج با نیت کمک به تیم‌ها ارسال می‌شوند و در آینده‌ای نزدیک، نتیجه‌ی نهایی به اطلاع تیم خواهد رسید. نتیجه‌ی نهایی ممکن است متفاوت باشد!
- اعلام نتیجه‌ی نهایی: مدتی پس از ارسال سوال، بسته به ترافیک موجود نتیجه‌ی بررسی سوال به شکل یک پنجره‌ی جدید به اطلاع تیم خواهد رسید. این نتیجه در فهرست موجود در سربرگ View Runs نیز قابل مشاهده خواهد بود.

دو تصویر موجود در شکل ۷,۷ نمایی از نتیجه‌ی اولیه و نهایی را نمایش داده‌اند.



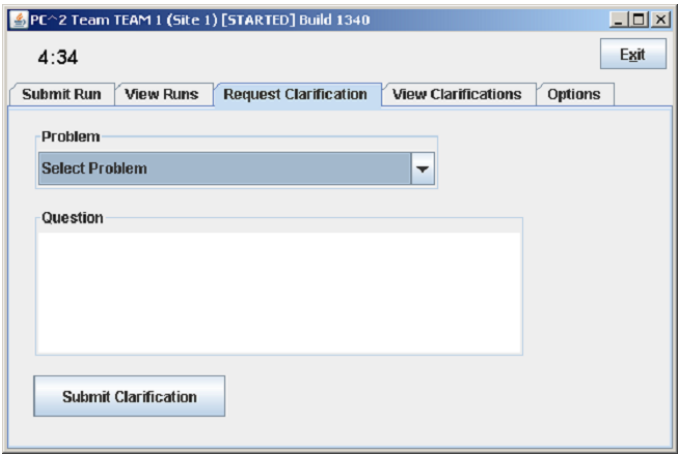
شکل ۷,۷. پنجره‌ی مربوط به نتیجه‌ی اولیه و نتیجه‌ی نهایی در نرم‌افزار پی-سی-تو

پاسخ‌های معمول که ممکن است توسط داوران اعلام شوند، در ادامه به اختصار توضیح داده شده‌اند:

- Yes (Accept): این پیام به معنی پذیرش پاسخ ارسال شده می‌باشد.
- Wrong Answer: پاسخ تولید شده توسط برنامه‌ی ارسالی با پاسخ درست مسأله، تفاوت دارد. در این مواقع، باید به نکات ریز سوال دقت کنید؛ مانند بازه‌ی متغیرها، طول آرایه‌های مورد استفاده، چاپ کردن کاراکتر خط جدید در انتهای خطوط و غیره.
- Compilation Error: برنامه‌ی ارسال شده، بر روی رایانه‌ی داور به درستی ترجمه نشده است. احتمالا برنامه‌ی نوشته شده دارای خطای نحوی است و یا از کتابخانه/دستور غیرمجازی استفاده می‌کند.
- Runtime Error: برنامه در زمان اجرا دچار خطا شده است. از علت‌های محتمل برای این مسأله می‌توان به خارج شدن از حد آرایه و تقسیم بر صفر اشاره کرد.
- Time Limit Exceed: مدت زمان اجرای برنامه از حد مجاز فراتر رفته است. احتمال دارد برنامه در یک حلقه‌ی بی‌نهایت به دام افتاده یا الگوریتم استفاده شده در آن غیربهینه باشد.
- Output Limit Exceed: خروجی تولید شده توسط برنامه بزرگ‌تر از حد مجاز است.

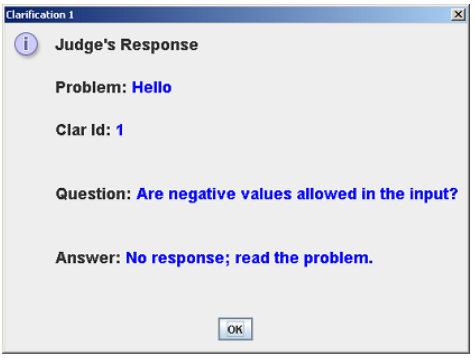
سربرگ Request Clarification

در صورتی که از نظرتان، ابهامی در رابطه با یکی از سوالات وجود دارد، می‌توانید با استفاده از سربرگ Request Clarification سوال خود را از داوران بپرسید. برای این منظور، از فهرست بازشوی Problem، مسأله‌ی مورد نظر را انتخاب کرده و سوال خود را در جعبه‌ی متن Question وارد کنید. با کلیک بر روی دکمه‌ی Submit Clarification، سوال شما برای داوران ارسال خواهد شد (شکل ۸,۷).



شکل ۸,۷. استفاده از سربرگ Request Clarification برای سوال پرسیدن از داوران

در صورتی که داوران تشخیص دهند سوال شما مناسب نبوده و یا نیاز به پاسخ ندارد، معمولا پاسخی با محتوای «No Response; read the problem» ارسال خواهند کرد (شکل ۹,۷).



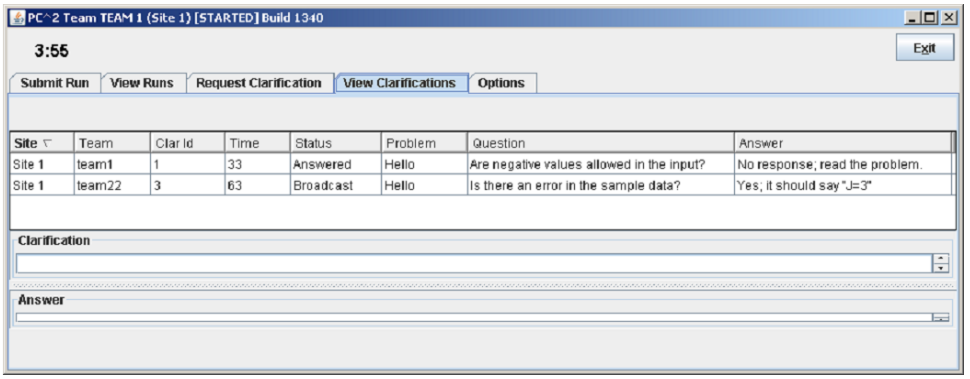
شکل ۹,۷. نمونه‌ای از پاسخ داوران به سوال پرسیده شده

سربرگ View Clarification

در سربرگ View Clarification می‌توانید پاسخ‌های ارسال شده توسط داوران را مشاهده کنید. این پاسخ‌ها در دو دسته‌ی زیر قرار می‌گیرند:

۱. پاسخ‌هایی که مستقیما به سوال شما داده شده‌اند.
۲. پاسخ‌هایی که به صورت عمومی و برای همه‌ی شرکت کنندگان ارسال شده‌اند.

در شکل ۱۰,۷، سطر اول از فهرست پاسخی از نوع اول و سطر دوم از فهرست، پاسخی از نوع دوم را نمایش می دهد.



شکل ۱۰،۷. نمونه ای از مشاهده‌ی دو پیام ارسال شده از سوی داوران در سربرگ View Clarification

خروج از نرم‌افزار

با کلیک بر روی دکمه‌ی Exit در سمت راست–بالای پنجره، از نرم‌افزار خارج خواهید شد. دقت کنید که خروج از نرم‌افزار تاثیری بر وضعیت سوالات ارسال شده و یا پاسخ‌های ارسالی از سوی داوران نخواهد داشت. با وارد شدن مجدد به نرم‌افزار، می‌توان نتیجه‌ی پاسخ‌های ارسال شده توسط داوران را در بخش مربوطه مشاهده کرد. هر چند پنجره‌ی فوری نمایش نتیجه‌ی داوری تنها در حالتی که نرم‌افزار در حال اجرا باشد، ظاهر خواهد شد.

نکته

در بعضی سیستم‌عامل‌ها (مانند ویندوز)، یک پنجره خط دستور نیز همزمان با اجرای نرم‌افزار ظاهر خواهد شد. این پنجره را می‌توان کمینه کرد ولی نباید آن را بست؛ زیرا این عمل موجب بسته شدن نرم‌افزار خواهد شد.

۷–۹– نمونه سوالات مسابقات

در این بخش، مجموعه‌ای از سوالات مسابقات اِی–سی–ام دانشگاه کاشان و دیگر دانشگاه‌های کشور گردآوری شده‌اند. این مسابقات در سطح ملی برگزار شده‌اند. راه‌حل این سوالات در بخش بعدی ارائه گشته است. علاقمندان می‌توانند به حل این سوالات اقدام نمایند و تجربیات خود را در زمینه‌ی حل مسأله تقویت کنند. به دلیل انگلیسی بودن صورت سوالات در اغلب مسابقات (حتی در مسابقات داخلی)، صورت اصلی اکثر سوالات این بخش انگلیسی است. در این بخش این گونه مسائل ابتدا به زبان انگلیسی و سپس به زبان فارسی مطرح می‌شوند. با این حال، به منظور آشنایی بیشتر با حال و هوای مسابقات برنامه‌نویسی، به اکیدا توصیه می‌شود که از ترجمه‌ی سوالات انگلیسی استفاده نکنید. برای مجموعه‌ی کاملی از سوالات و راه‌حل‌های آن‌ها در مسابقات مختلف ملی و بین‌المللی می‌توانید به لوح فشرده‌ی کتاب مراجعه کنید.

۷–۹–۱– مسابقات ملی سال ۹۱ – دانشگاه کاشان – ده سوال

Problem A: Anagrams

Anagrams is word game that involves rearranging letter tiles to form words. Each piece has a lower or upper Latin letter on it.

Little Hamed has received an Anagrams puzzle for his birthday. But he does not know the rules of the game. So he simply arranges tiles beside each other to form new words and sentences. Little Hamed arranged the words to form a sentence. He was so happy because he managed to use all the pieces of letters he got. But after a more careful look, he thought the sentence is not that beautiful. He decided to rearrange it to make a more beautiful sentence. This time he does not want to necessarily use all the pieces. The new puzzle Little Hamed wants to form, lack some letters and some letters are extra.

There is a toy shop in the vicinity that sells or exchanges letter tiles. Little Hamed can exchange a tile for a tile with the same letter but different case for C Tomans. He can exchange a tile with a tile of any letter and any case for S Tomans. He can also buy a new tile for B Tomans. The prices are reasonable relative to each, i.e. $C < S < B$.

Now, given the first puzzle Little Hamed has arranged, help him obtain all the tiles that are required to form his new puzzle with minimum possible cost.

Input (Standard Input)

First line of input contains a single integer $t(t \le 100)$, the number of tests that follow. The first line of each test contains three space-separated integers $C, S, B(1 \le C < S < B \le 10^6)$ the cost of changing case of a tile, exchange a tile with a different tile and buying a new tile, respectively.
Next two lines describe the puzzle that Little Hamed has arranged and the new puzzle he wants to form. Each puzzle contains only lower and upper Latin letters and space character. Length of each puzzle does not exceed 100. There will be no leading or trailing spaces in description of a puzzle and two words in a puzzle are separated with a single space. Note that space is not puzzle tile and is only used to separate different words of the puzzle.

Output (Standard Output)

For each test you should output a single line, the minimum cost to prepare all the tiles required two form the second puzzle.

Sample Input and Output

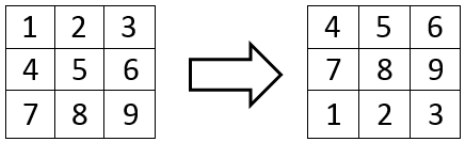
Sample Input	Sample Output
3 1 100 10000 write SaMpLe is not sImPlE 1 100 10000 Happy PMP Zart PMP	405 300 60300

1 100 10000 Last war of PMP Reincarnation of PMP	

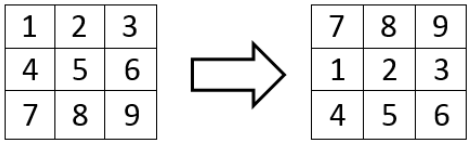
Problem B: Beautiful Mind

Jamshid Kāshānī was a great scientist in Persia. He was also a great mathematician. The king of the time that was not satisfied with courtesy of Jamshid toward himself, put Jamshid in jail and commanded his death.
Before they execute him, as sign of good faith, he offered Jamshid a riddle and told him if he can solve the riddle faster than him, he's free to go. The riddle is:
"Given a $n \times n$ matrix, you should simulate a series of operations on it. The operations are:

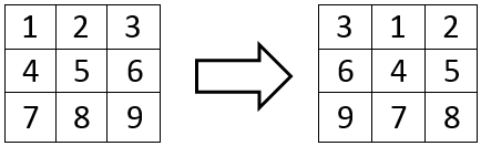
- **U**pward: shift the rows one place up. First row will be placed the last row.



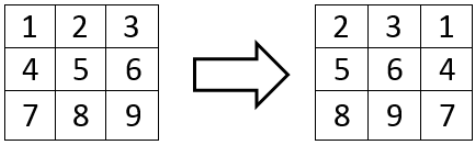
- **D**ownward: shift the rows one place down. Last row will be placed before first row.



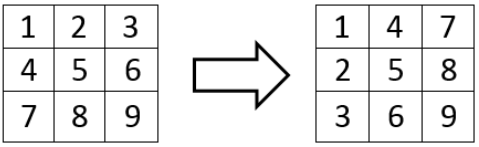
- **F**orward: shift the columns one place to right. Last column will be placed before first column.



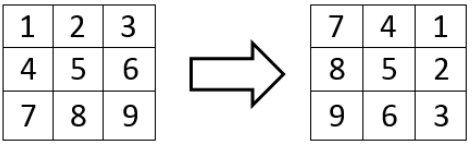
- **B**ackward: shift the columns one place to left. First column will be placed after last column.



- **T**ranspose: transpose the matrix. First row will be placed on first column, second row on second column and so on.



- **R**otate: rotate the matrix 90 degrees clockwise.



You should simulate all the operations on input matrix one by one and find the resulting matrix."

Jamshid solved the riddle way quicker than the king and was released.

Your task is to solve the above riddle.

Input (Standard Input)

First line of input contains a single integer $t(t \le 20)$. The first line of each test contains a single integer $n(1 \le n \le 50)$. Next n lines each contain n positive integers. j^{th} number of i^{th} line is the number at row i and column j of matrix. These numbers are not greater than 100. After that there will be a line with the sequence of operations performed on matrix in order from left to right. It is a string of length at most 100. Each operation is shown by a character, "U" for upwards, "D" for downwards, "F" for forwards, "B" for backwards, "T" for transpose and "R" for rotate.

Output (Standard Output)

For each test you should print a $n \times n$, the resulting matrix after performing all operations on it. Output matrix should be formatted as input matrix (i.e. you should print n lines each containing n integers) Numbers should be separated by a single space.
You should print an empty line after each test.

Sample Input and Output

Sample Input	Sample Output
2 4 4 8 12 16 3 7 11 15 2 6 10 14 1 5 9 13 R 3	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 37 28 19 64 55 46 91 82 73

19 28 37 46 55 64 73 82 91 UFBDTR	

Problem C: Cards

You have got a set of cards. On each card an integer between 1 to m is written. The number of cards that have number i written on their side is equal to n_i.

You shuffle the cards and distribute them among m bins so that i^{th} bin receives n_i cards. In each bin cards are stacked on top of each other. All possible distributions of cards are equally likely to occur.

We call a distribution of cards nice with respect to i^{th} bin, if it has the following property. Starting with i^{th} bin, in each step you pick (and remove) the topmost card from it. The number written on the card specifies the bin from which you should pick the next card. You keep picking cards in this way until you cannot pick any more cards. The distribution is nice if you can remove all the cards following these instructions. In other words, after the last move there should not be any cards remained.

He is curious to find the probability that a random distribution of cards is nice with respect to first bin, and also with respect to a random bin. Put differently, can we remove all the cards if we pick the first card from the first bin? What if we choose the starting bin uniformly at random?

Input (Standard Input)

Input contains at most 150 tests. In the first line of each test there will be an integer $m(2 \le m \le 1000)$. Parameter m is the number of bins that the cards are distributed among and also it is the greatest number written on a card.

Next line will contain m integers $n_i\,(1 \le n_i \le 10^9)$. Parameter n_i is the number of cards that i^{th} bin receives and also it is the number of cards that number i is written on them.
Input terminates with $m = 0$.

Output (Standard Output)

For each test you should output the probabilities that a random distribution of cards is nice with respect to first bin followed by the answer to the same question with respect to a random bin. These two numbers should be separated with a single space and should be formatted as irreducible fraction p/q such that greatest common divisor of p and q is equal to 1.

Sample Input and Output

Sample Input	Sample Output
2 1 1 3 10920 2184 6552 0	1/2 1/2 1/3 5/9

Problem D: Dictionary Attack

You are performing a dictionary attack to hack an unfortunate user's account. You have a list of n potential words that the user is most likely to use. This list is prefix free, i.e. no word in the list is a prefix of another. You guess that the password is concatenation of one or more words of the list without any delimiters.

The average person usually uses passwords that have the following criteria:

- The number of words used in password is limited as the users can memorize only a few number words. Depending on the number of words in your dictionary, you assume the number of words used in a password is at most m. If a word is used multiple times, all occurrences of that word are taken into account.
- Users are reluctant to use long passwords as they are error prone to type. So the length of a password is at most L.

You are setting up a program to run over all passwords one by one in lexicographical order. But even for computers it takes days to check all possible combinations. Especially if you setup a time interval between two consecutive queries to prevent system administrators become suspicious.

The program you have written gives an index k in the input, and starts its work from the k^{th} lexicographically smallest word. In each run, the program iterates over possible words in search space until it finds the password. If it fails to identify the credentials for 12 hours, it exits. The only thing it reports is the index of last checked word.

You noted that the program is inefficient when the input argument gets large. You want to optimize it. Your task is to find the k^{th} smallest possible password with a quick algorithm.

Input (Standard Input)

First line of input contains a single integer $t\,(t \le 100)$, the number of tests that follow. The first line of each test contains four space-separated integers $n, m, L, k\,(1 \le n \le 1000, 1 \le m \le 10, 1 \le L \le 1000, 1 \le k \le 10^9)$, the number of words in the dictionary, maximum number of words used in a password, maximum length of a

password and the index of the password in the sorted list of all possible combinations that you are looking for.

Each of the next n lines contains a string of length at most 300. Words consist of only lowercase Latin letters. It is guaranteed that no word in the list is a prefix of another.

Output (Standard Output)

For each test you should output a single line, the k^{th} lexicographically smallest possible password. If the total number of possible password is less than k, you should print "`empty`" instead.

Sample Input and Output

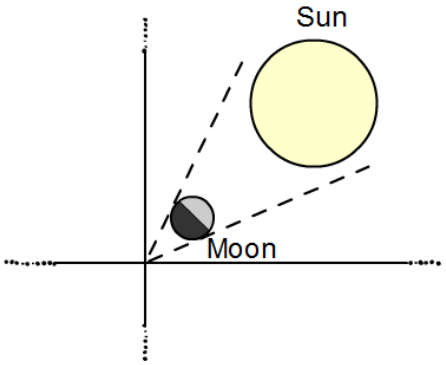
Sample Input	Sample Output
6 2 2 11 1 simple sample 2 2 11 2 simple sample 2 2 12 2 simple sample 2 2 11 3 simple sample 2 2 12 7 simple sample 4 4 10 17 zart jamshid pmp happy	sample simple samplesample empty empty zartpmp

Problem E: Eclipse

When the Moon passes between the Sun and the Earth, it may fully or partially block the Sun. This phenomenon is called solar eclipse.

Qiyās-ud-din Jamshid Kāshānī was one of the best Persian astronomers and mathematicians who lived in 8^{th} century. He has written treatise on astronomical observational instruments

and invented a mechanical planetary computer which could solve a number of planetary problems.



An example of total eclipse

Jamshid is observing the sky to see the next happening of this beautiful and rare event. He has calculated exact location and radius of the Moon and the Sun. He wants to determine the type of eclipse that is happening.

For simplicity, Jamshid assumes that he is standing in origin of a plane. He considers an object or part of it visible, if any point of that object except for the points on the boundary can be directly seen from the origin.

Input (Standard Input)

First line of input contains a single integer $t\,(t \le 10000)$, the number tests that follow. Each test consists of two lines, describing the Moon and the Sun respectively. Astronomical objects are defined as three integers $x, y, r\,(-10^4 \le x, y \le 10^4, 1 \le r \le 1000)$ specifying their position and radius. It is guaranteed that radius of the Sun is greater than or equal to radius of the Moon.

You are standing at the origin watching the sky. It is guaranteed that the Sun and the Moon are non-intersecting and do not touch each other. Also none of them encloses the origin neither on its interior nor its boundary.

Output (Standard Output)

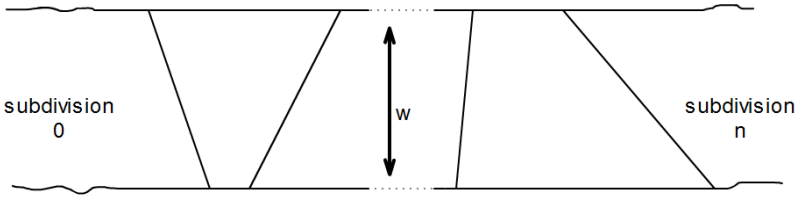
For each test you should output a single line. If the disk of the Sun is fully obscured by the Moon, output "`Total eclipse`". Otherwise if the blocking is partial, that is you can directly see a point strictly inside the circle of the Sun, output "`Annular eclipse`". In the case when the Moon is not blocking the Sun at all and all points strictly inside the circle of the Sun are visible to you, output "`None`".

Sample Input and Output

Sample Input	Sample Output
3 1 1 1 10 10 2 1 2 1 5 5 2 1 10 1 10 2 2	Total eclipse Annular eclipse None

Problem F: Farmers of Kasian

The land of Kasian is in the form of an infinite strip. It has a width of W . Farmers of Kasian have divided their land into subdivisions with n line segments that fully extend across the strip and are non-intersecting. Subdivisions are numbered from 0 through n (inclusive) from left to right.



An example of subdivisions of Kasian

The governor of Kasian is collecting taxes from the farmers. He has sent an agent to calculate and collect the taxes. As there are quite a lot of subdivisions, the agent has decided to visit only m random locations and collect the tax from the owners of those subdivisions.

Unfortunately because of the complications in the territory of farmers, the agent cannot accurately decide on which subdivision he is standing. So he asked you as an experienced - programmer to do this job for him.

Input (Standard Input)

First line of input contains a single integer $t\,(t \le 10)$, the number tests that follow. On the first line of each test there are two integers $n, W\,(1 \le n \le 5 \times 10^4, 1 \le W \le 10^9)$, the number of subdivisions and the parameter W as defined in the problem statement. From now on we assume the land is the unbounded area in between lines $y = 0$ and $y = W$.

Next n lines will contain two integers x_1 and $x_2\,(-10^9 \le x_1, x_2 \le 10^9)$. It means that there is line segment from $(x_1, 0)$ to (x_2, W). It is guaranteed that these line segments are disjoint.

Next line consists of a single integer $m(1 \le m \le 5 \times 10^4)$, the number of locations the agent checks. After that there are m lines follow, each containing two integers $x_i, y_i (-10^9 \le x_i \le 10^9, 0 \le y_i \le W)$ the i^{th} location that the agent checks. No query point lie on any line segment.

Output (Standard Output)

For each test you should output m lines. i^{th} line for each test should be the number of subdivision for the i^{th} query point. The answer for each query should be a number from 0 to n.

There should be exactly one empty line after each test.

Sample Input and Output

Sample Input	Sample Output
2 1 2 10 0 2 5 0 5 2 4 10 8 8 3 1 4 5 15 12 5 2 1 9 10 3 1 6 0 14 5	0 1 0 3 1 2 4

Problem G: Goodfellas

Ta'zieh is a special kind of traditional Persian performing arts. It dates before the Islamic era and the tragedy of Siawush is one of the best examples.

Ta'zieh is usually performed on a round stage. Long ago when Ta'zieh was more popular, people of different families was gathering around the stage to watch the show. Each family had some male members and some female members. People could stand around the stage in any order although they should comply with these restrictions:

- A man and a woman from different families cannot stand beside each other.

- People can stand in several circular rows. But in each row they should stand tight, so everybody is considered to be adjacent to the persons (if any) who are standing next and previous to him.

You are studying the behavior of ancient people to learn more about their social attitude, traditions etc. Given description of some families in a Ta'zieh show, you have to find the minimum number of rows they could form while all restrictions are satisfied.

Input (Standard Input)

Input contains several test cases. Number of tests is at most 100. The first line of each test contains a single integer $n(1 \le n \le 100)$, the number of families.

Next line will contain n integer's m_i, the number of men in the i^{th} family. Description of women in each family w_i will follow in the same format $(0 \le w_i, m_i \le 9)$. It is guaranteed that for any i between 1 and n, $w_i + m_i > 0$. Input terminates with $n = 0$.

Output (Standard Output)

For each test you should output the minimum number of circular rows that are required, so that each people can watch the show without violating the rules.

Sample Input and Output

Sample Input	Sample Output
2 1 1 0 1 3 2 1 3 0 9 2 0	2 1

Problem H: Human Experience

Given n pairs of integers (a_i, b_i), you should calculate the following equation modulo $10^9 + 7$:

$$LCM\,(a_1^{b_1}, a_2^{b_2}, ..., a_n^{b_n})$$

where LCM is the least common multiple of a set of numbers.

Input (Standard Input)

The first line of input contains a single integer $t(t \le 50)$, the number of tests that follow. The first line of each test contain an integer $n(1 \le n \le 10^5)$, the number of pairs of

integers. The i^{th} of next n lines contains two integers $a_i, b_i (1 \le a_i \le 10^6, 0 \le b_i \le 10^9)$, the base and the power of i^{th} pair as described in the statement.

Output (Standard Output)

For each test you should output a single line, the answer of equation modulo $10^9 + 7$.

Sample Input and Output

Sample Input	Sample Output
2 3 2 4 15 2 6 3 1 2 1000	10800 688423210

Problem I: Interesting Knapsack

Knapsack is well-known optimization problem. In 0-1 version of problem, you are given a list of n items each with a size and a value, and a bag of size B, you choose a subset of items that fit in the bag and their total value is maximized. This problem is proved to be NP-hard, but it has pseudo-polynomial time dynamic programming algorithm.

In this problem you should solve the exact same problem with one additional constraint: either your bag is relatively small or the maximum value of items is limited.

Input (Standard Input)

Input contains at most 50 tests. The first line of each test contains $n, B (1 \le n \le 100, 1 \le B \le 10^8)$, the number of items, and the size of the knapsack. The second line contains n integers with the i^{th} integer being $v_i (1 \le v_i \le 10^7)$, the value of i^{th} item. Third line describes the sizes of items $s_i (1 \le s_i \le 10^7)$ in the same format.

It is guaranteed that in each test either $B \le 50000$ or $\max_{1 \le i \le n} v_i \le 500$.

Input terminates with a line "0 0".

Output (Standard Output)

For each test you should output a single line, the maximum total value of items in the knapsack.

Sample Input and Output

Sample Input	Sample Output
2 100000 1 500 1 100000 3 10 3 5 7 4 6 7 5 100 1 2 3 4 5 5 4 3 2 1 0 0	500 8 15

Problem J: Joining Vertices

You are given a weighted undirected graph with n vertices and m edges. In one move you can join any two different vertices into a single vertex and the cost of this operation is the length of the shortest path between the two vertices. Please note that after joining some vertices, multiple edges or self-loops might appear in the graph. Joining vertices is continued with the induced graph until only one vertex remains.

Your task is to join all vertices into one, such that total cost of operations is minimized.

Input (Standard Input)

First line of input contains a single integer $t\,(t \le 40)$, the number tests that follow. First line of each test contains integers $n, m\,(1 \le n \le 1000, m \le 200000)$, the number of vertices and edges in the graph, respectively. Next m lines contain three integers $u_i, v_i, w_i\,(1 \le u_i, v_i \le n, 1 \le w_i \le 10^6)$,which means there is an undirected edge between u_i, v_i with the weight of w_i. The graph does not contain multiple edges or self-loops.

Output (Standard Output)

For each test you should output the minimum total cost of operations for contracting all vertices into one, on a single line. If it is not possible, print "`impossible`" instead.

Sample Input and Output

Sample Input	Sample Output
2 4 5 1 2 2 3 1 1 4 3 3 2 4 3 3 2 5 4 2 1 2 10 3 4 20	6 impossible

مسأله‌ی ۱: آناگِرَم

آناگِرَم نام یک بازی با کلمات است که شامل چیدن حروف در کنار یکدیگر برای درست کردن کلمات می‌شود. روی هر قطعه یک حرف بزرگ یا کوچک انگلیسی نوشته شده است.

به حامد کوچک یک پازل آناگِرَم به عنوان هدیه‌ی روز تولد داده شده است. ولی اون قوانین بازی را نمی‌داند. او فقط قطعات را کنار یکدیگر قرار می‌دهد تا توسط آنها کلمات و جملات جدیدی ایجاد کند. به این ترتیب او با کمک این قطعات، یک جمله درست کرد. او خیلی خوشحال شده بود زیرا توانسته بود از تمامی قطعاتی که در اختیار داشت در این جمله استفاده کند. ولی بعد از بررسی دوباره، متوجه شد که جمله‌ی درست شده خیلی هم جالب نیست! به همین دلیل تصمیم گرفت مجددا جای قطعات را با یکدیگر عوض کند تا جمله‌ی جالب‌تری درست کند. او این بار نمی‌خواهد که لزوما از تمامی قطعات استفاده کند. پازل جدیدی که حامد کوچک می‌خواهد درست کند، چند حرف کم و زیاد دارد.

در همان نزدیکی یک اسباب‌بازی فروشی وجود دارد که قطعات حروف را فروخته یا تعویض می‌کند. حامد کوچک می‌تواند قطعه‌ی یک حرف را با قطعه‌ی دیگری از همان حرف ولی با اندازه‌ی متفاوت به ازای C تومان تعویض کند. همچنین می‌تواند یک قطعه را با قطعه‌ای مربوط به هر حرف دیگری و با هر اندازه‌ای به ازای S تومان تعویض کند. علاوه بر این، می‌تواند یک قطعه‌ی جدید را به قیمت B تومان بخرد. قیمت‌ها نیز نسبت به یکدیگر منطقی هستند، یعنی $C < S < B$.

حالا، با فرض در اختیار داشتن پازل اولیه، به حامد کمک کنید که تمامی قطعات مورد نیاز برای درست کردن پازل نهایی را با حداقل هزینه به دست بیاورد.

ورودی (ورودی استاندارد)

در خط اول یک عدد صحیح $t(t \le 100)$ t می‌آید که بیانگر تعداد آزمون‌هاست. خط اول هر آزمون شامل سه عدد صحیح C، S و B $(1 \le C < S < B \le 10^6)$ است که توسط فاصله از همدیگر جدا شده‌اند و به ترتیب نشان‌دهنده‌ی هزینه‌ی تعویض اندازه‌ی یک قطعه، تعویض یک قطعه با یک قطعه‌ی متفاوت و خرید یک قطعه‌ی جدید هستند.

در دو خط بعد پازلی که حامد کوچک درست کرده است و پازلی که می‌خواهد درست کند، معرفی می‌شود. هر پازل تنها از حروف انگلیسی کوچک و بزرگ و نویسه‌ی فاصله تشکیل شده است. طول هر پازل از 100 تجاوز نمی‌کند. قبل و بعد از پازل‌ها نویسه‌ی فاصله نمی‌آید و دو کلمه‌ی متوالی در یک پازل توسط یک نویسه‌ی فاصله از هم جدا می‌شوند. دقت کنید که نویسه‌ی فاصله جزئی از قطعات پازل نیست و تنها برای جداسازی کلمات درون پازل استفاده می‌شود.

خروجی (خروجی استاندارد)

برای هر آزمون شما باید یک خط در خروجی چاپ کنید که حاوی حداقل هزینه‌ی لازم برای به دست آوردن تمامی قطعات مورد نیاز در پازل نهایی است.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 1 100 10000 write SaMpLe is not sImPlE 1 100 10000 Happy PMP Zart PMP 1 100 10000 Last war of PMP Reincarnation of PMP	405 300 60300

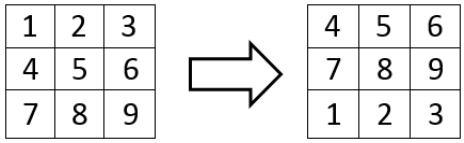
مسأله‌ی ۲: ذهن زیبا

جمشید کاشانی یکی از دانشمندان بزرگ پارسی بود. او یک ریاضیدان قدرتمند نیز بود. پادشاه وقت که از عدم اطاعت جمشید از خودش ناراحت بود، دستور حبس و سپس کشتن جمشید را داد.

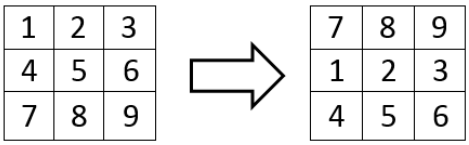
قبل از اجرای دستور اعدام، به عنوان آخرین امید، پادشاه به جمشید یک معما گفت و اعلام کرد که اگر او بتواند معما را زودتر از پادشاه حل کند، آزاد می‌شود. معما به این صورت است:

با فرض در اختیار داشتن یک ماتریس $n \times n$، شما باید چندین عملیات ریاضی را شبیه‌سازی کنید. این عملیات عبارتند از:

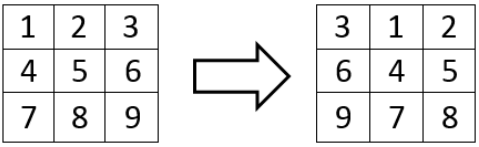
- بالا (U): هر سطر را به یک سطر بالاتر منتقل کنید. سطر اول به سطر آخر منتقل خواهد شد.



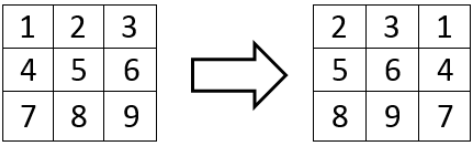
- پایین (D): هر سطر را به یک سطر پایین‌تر منتقل کنید. سطر آخر به سطر اول منتقل خواهد شد.



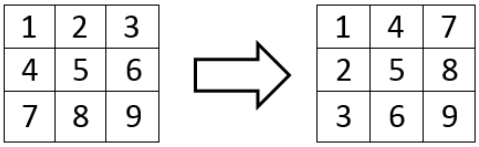
- جلو (F): هر ستون را به یک ستون راست‌تر منتقل کنید. ستون آخر به ستون اول منتقل خواهد شد.



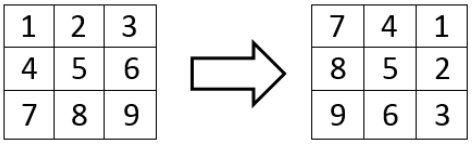
- عقب (B): هر ستون را به یک ستون چپ‌تر منتقل کنید. ستون اول به ستون آخر منتقل خواهد شد.



- ترانهاده (T): ماتریس را ترانهاده کنید. سطر اول به ستون اول خواهد رفت، سطر دوم به ستون دوم و به همین ترتیب الی آخر.



- چرخش (R): ماتریس را ۹۰ درجه در جهت عقربه‌های ساعت بچرخانید.



شما باید تمامی این عملیات ها به نوبت روی ماتریس ورودی اعمال کرده و ماتریس نهایی را به دست آورید.
جمشید این معما را زودتر از پادشاه حل کرد و آزاد شد.
وظیفه‌ی شما حل این معما است.

ورودی (ورودی استاندارد)

خط اول ورودی شامل یک عدد صحیح t $(t \le 20)$ است که تعداد آزمون‌ها را مشخص می‌کند. خط اول هر آزمون شامل یک عدد صحیح n $(1 \le n \le 50)$ است. در n خط بعدی، n عدد صحیح مثبت خواهد آمد. عدد j ام خط i ام، عدد سطر i ام و ستون j ام ماتریس است. این اعداد از 100 بزرگتر نیستند. بعد از آن یک خط می‌آید که دنباله‌ای از عملیاتی را که باید روی ماتریس انجام شوند، از چپ به راست معرفی می‌کند. این دنباله به صورت یک رشته‌ی حداکثر به طول 100 خواهد بود. هر عملیات به صورت یک نویسه نشان داده می‌شود؛ "U" به معنی بالا، "D" به معنی پایین، "F" به معنی جلو، "B" به معنی عقب، "T" به معنی ترانهاده، و "R" به معنی چرخش خواهد بود.

خروجی (خروجی استاندارد)

برای هر آزمون شما باید ماتریس $n \times n$ نتیجه را بعد از اعمال تمامی عملیات خواسته شده چاپ کنید. قالب ماتریس خروجی مانند ماتریس ورودی است (شما باید n خط که هرکدام شامل n عدد صحیح است، چاپ کنید). اعداد باید توسط یک نویسه‌ی فاصله از هم جدا شده باشند.
بعد از هر آزمون باید یک خط خالی نیز چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 4 4 8 12 16 3 7 11 15 2 6 10 14 1 5 9 13 R 3 19 28 37 46 55 64 73 82 91 UFBDTR	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 37 28 19 64 55 46 91 82 73

مسأله‌ی ۳: کارت‌ها

شما مجموعه‌ای از کارت ها در اختیار دارید. روی هر کارت یک عدد صحیح بین 1 تا m نوشته شده است. تعداد کارت‌هایی که عدد i روی آنها نوشته شده است، برابر n_i است.

شما کارت ها را بُر زده و آنها را در m سطل قرار می‌دهید به طوری که سطل iام n_i کارت داشته باشد. در هر سطل کارت‌ها به صورت پشته روی یکدیگر قرار می‌گیرند. تمامی توزیع‌های ممکن کارت‌ها با احتمال برابر ممکن است رخ دهند.

ما به توزیع خاصی از کارت‌ها با توجه به سطل iام جالب می‌گوییم، اگر این خاصیت را داشته باشد:

با شروع از سطل iام، در هر گام کارت بالایی درون سطل را برداشته (و حذف می‌کنید). عدد نوشته شده روی کارت مشخص می‌کند که کارت بعدی را باید از کدام سطل بردارید. به همین ترتیب به برداشتن کارت‌ها ادامه می‌دهید تا وقتی که کارت دیگری را نتوانید حذف کنید. به یک توزیع از کارت‌ها جالب می‌گوییم اگر بتوان با این روش تمامی کارت‌ها را حذف کرد. به عبارت دیگر، بعد از آخرین حرکت نباید هیچ کارت دیگری باقی مانده باشد.

می‌خواهیم بدانیم که احتمال این که یک توزیع تصادفی از کارت‌ها با توجه به سطل اول و همچنین با توجه به یک سطل تصادفی جالب باشد، چقدر است. به عبارت دیگر، آیا می‌توان همه‌ی کارت‌ها را با شروع از سطل اول حذف کرد؟ اگر سطل شروع را به طول تصادفی و یکنواخت از میان تمامی سطل‌ها انتخاب کنیم چطور؟

ورودی (ورودی استاندارد)

ورودی شامل حداکثر 150 آزمون خواهد بود. در خط اول هر آزمون یک عدد صحیح m $(2 \le m \le 1000)$ خواهد آمد. پارامتر m تعداد سطل هایی است که کارت‌ها میان آنها توزیع شده‌اند و همچنین بزرگترین عدد نوشته شده روی کارت‌ها است. در خط بعد m عدد صحیح n_i $(1 \le n_i \le 10^9)$ می‌آید. پارامتر n_i تعداد کارت‌هایی است که درون سطل iام قرار می‌گیرد و همچنین تعداد کارت‌هایی است که عدد i روی آنها نوشته شده است. ورودی با عدد $m = 0$ خاتمه می‌یابد.

خروجی (خروجی استاندارد)

برای هر آزمون باید احتمال جالب بودن یک توزیع تصادفی کارت‌ها را با توجه به سطل اول و همچنین نسبت به یک سطل تصادفی چاپ کنید. این دو عدد باید توسط یک نویسه‌ی فاصله از هم جدا شده و در قالب یک کسر ساده نشدنی p/q باشند به طوری که بزرگترین مقسوم علیه مشترک p و q برابر 1 باشد.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 1 1 3 10920 2184 6552 0	1/2 1/2 1/3 5/9

مسأله‌ی ٤: حمله‌ی واژه‌نامه‌ای

شما در حال انجام یک حمله‌ی واژه‌نامه‌ای برای هک کردن حساب کاربری یک کاربر بدشانس هستید. شما لیستی از n کلمه که کاربر مورد نظر به احتمال زیاد از آنها استفاده کرده است، در اختیار دارید. این لیست غیرپیشوندی است؛ یعنی هیچ کلمه‌ای در این لیست، پیشوندی از کلمه‌ای دیگر نیست. شما حدس می‌زنید که رمز عبور ترکیبی از یک یا چند کلمه در لیست بدون هیچ نویسه‌ی میانی باشد.

یک شخص معمولا از رمزهایی استفاده می‌کند که این ویژگی‌ها را داشته باشند:

- تعداد کلمات استفاده شده در رمز عبور محدود است زیرا کاربران تنها می‌توانند تعداد محدودی از کلمات را حفظ کنند. با توجه به تعداد کلمات در واژه‌نامه، شما حدس می‌زنید که تعداد کلمات استفاده شده در رمز عبور حداکثر m باشد. اگر یک کلمه چندین بار استفاده شود، تمامی تکرارهای آن کلمه حساب می‌شوند.
- کاربران میلی به استفاده از رمزهای طولانی ندارند؛ زیرا هنگام تایپ کردن ممکن است دچار خطا شوند. به همین ترتیب حداکثر طول رمز عبور برابر L است.

شما می‌خواهید برنامه‌ای بنویسید که تمامی رمزهای عبور احتمالی را یکی یکی و به ترتیب حروف الفبا بررسی کند. ولی بررسی کردن تمامی ترکیب‌های ممکن حتی برای رایانه‌ها هم ممکن است چندین روز طول بکشد. مخصوصا اگر شما بخواهید بین بررسی‌های متوالی زمانی مکث کنید تا مدیران امنیتی سیستم‌ها مشکوک نشوند.

برنامه‌ای که شما نوشته‌اید اندیس k در ورودی را داده و با k کوچکترین کلمه بر حسب ترتیب الفبا شروع می‌کند. در هر بار اجرا، برنامه کلمات احتمالی در فضای جستجو را پیمایش می‌کند تا زمانی که رمز عبور را پیدا کند. اگر برنامه نتواند با گذشت ۱۲ ساعت رمز را پیدا کند، متوقف شده و تنها چیزی که بر می‌گرداند اندیس آخرین کلمه‌ی بررسی شده است.

شما متوجه شدید که این برنامه زمانی که اندازه ورودی بزرگ باشد، کارا نخواهد بود و می‌خواهید آن را بهینه کنید. وظیفه‌ی شما پیدا کردن k کوچکترین کلمه‌ی احتمالی با یک الگوریتم سریع است.

ورودی (ورودی استاندارد)

در خط اول یک عدد صحیح t $(t \le 100)$ می‌آید که تعداد آزمون‌های بعدی را مشخص می‌کند. خط نخست هر آزمون شامل چهار عدد صحیح n، m، L و k $(1 \le n \le 1000, 1 \le m \le 10, 1 \le L \le 1000, 1 \le k \le 10^9)$ است که توسط نویسه‌ی فاصله از یکدیگر جدا شده‌اند و به ترتیب بیانگر این مقادیر هستند: تعداد کلمات درون واژه‌نامه، حداکثر کلمات استفاده شده در یک رمز عبور، حداکثر طول رمز عبور و اندیس رمز عبور در لیست مرتب شده‌ی تمامی ترکیب‌ها که شما به دنبال آن هستید.

در n خط بعد، رشته‌ای به طول حداکثر 300 می‌آید. کلمات تنها شامل حروف انگلیسی کوچک هستند. تضمین می‌شود که هیچ کلمه‌ای در این لیست، پیشوندی از کلمه‌ای دیگر در لیست نیست.

خروجی (خروجی استاندارد)

برای هر آزمون شما باید خروجی را در یک خط چاپ کنید، که همان k کوچکترین رمز عبور احتمالی بر حسب حروف الفبا است. اگر حداکثر تعداد رمزهای احتمالی کمتر از k است، باید عبارت "empty" را چاپ کنید.

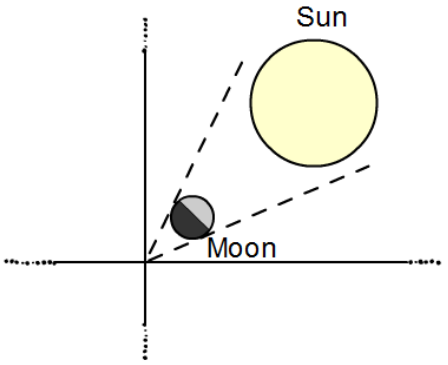
نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
6 2 2 11 1 simple sample 2 2 11 2 simple sample 2 2 12 2 simple sample 2 2 11 3 simple sample 2 2 12 7 simple sample 4 4 10 17 zart jamshid pmp happy	sample simple samplesample empty empty zartpmp

مسأله‌ی ۵: خورشیدگرفتگی

زمانی که ماه میان خورشید و زمین قرار می‌گیرد، ممکن است جلوی بخشی از خورشید یا تمام آن قرار بگیرد. به این پدیده خورشیدگرفتگی گفته می‌شود.

غیاث‌الدین جمشید کاشانی یکی از اولین ستاره‌شناسان و ریاضیدانان پارسی بود که در قرن هشتم زندگی می‌کرد. او مقاله‌ای در مورد ابزارهای مشاهدات پدیده‌های ستاره‌شناسی نوشته است و همچنین یک رایانه‌ی مکانیکی نجومی اختراع کرده است که می‌تواند تعدادی از مسائل نجومی را حل کند.



مثالی از یک خورشیدگرفتی کامل

جمشید در حال مشاهده کردن آسمان است تا متوجه شود که این پدیده زیبا و نادر چه زمانی اتفاق می‌افتد. او محل و شعاع دقیق ماه و خورشید را محاسبه کرده است و می‌خواهد نوع خورشیدگرفتگی بعدی را تعیین کند.
برای ساده‌سازی، جمشید فرض می‌کند که در مبدا مختصات قرار دارد. او فرض می‌کند که یک شی یا بخشی از آن قابل رویت است، اگر هر نقطه‌ای از آن شی به جز نقاط روی مرزهای آن به طور مستقیم از مبدا مختصات قابل رویت باشد.

ورودی (ورودی استاندارد)

خط اول ورودی حاوی یک عدد صحیح t $(t \le 10000)$ است که تعداد آزمون‌ها را مشخص می‌کند. هر آزمون از دو خط تشکیل شده است که به ترتیب وضعیت ماه و خورشید را شرح می‌دهند. اشیای نجومی با سه عدد صحیح x، y و r $(-10^4 \le x, y \le 10^4, 1 \le r \le 1000)$ مشخص می‌شوند که به ترتیب مختصات و شعاع آنها را نشان می‌دهند. تضمین می‌شود که شعاع خورشید بزرگتر یا مساوی شعاع ماه باشد.
شما در مبدا مختصات ایستاده و در حال رصد کردن خورشید هستید. تضمین می‌شود که خورشید و ماه با هم اشتراکی نداشته و حتی مرزهای آنها در تماس با یکدیگر نیست. همچنین مبدا مختصات در مرز یا درون هیچکدام از آنها قرار ندارد.

خروجی (خروجی استاندارد)

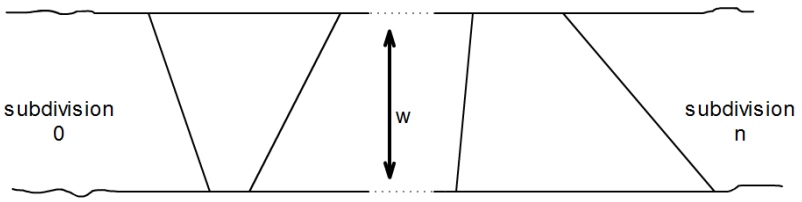
برای هر آزمون شما باید خروجی را در یک خط چاپ کنید. اگر دیسک خورشید به طور کامل توسط ماه پوشش داده می‌شود، خروجی "Total eclipse" است. در غیر این صورت اگر خورشیدگرفتگی جزئی است، به این معنی که حداقل یکی از نقاطِ کاملا درونِ خورشید، قابل رویت است، خروجی "Annular eclipse" خواهد بود. در صورتی که ماه در جلوی خورشید قرار ندارد و تمامی نقاط درون خورشید مستقیما توسط شما قابل رویت هستند، خروجی "None" است.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 1 1 1 10 10 2 1 2 1 5 5 2 1 10 1 10 2 2	Total eclipse Annular eclipse None

مسأله‌ی ٦: کشاورزان کَسیان

دهکده‌ی کَسیان به فرم یک باریکه‌ی بی‌پایان است. عرض این باریکه W است. کشاورزان کَسیان زمین‌های خود را توسط n پاره‌خط به زیربخش‌هایی تقسیم کرده‌اند. این پاره‌خط‌ها عرض باریکه را به طور کامل می‌پوشانند و با یکدیگر اشتراکی ندارند. زیربخش‌ها از چپ به راست از 0 تا n شماره‌گذاری شده‌اند.



نمونه‌ای از زیربخش‌های کَسیان

یکی از فرمانداران کَسیان می‌خواهد از کشاورزان مالیات بگیرد. او ماموری برای محاسبه و جمع‌آوری مالیات‌ها فرستاده است. از آنجایی که تعداد زیربخش‌ها بسیار زیاد است، مامور تصمیم گرفته است که تنها به m محل تصادفی رفته و از صاحبان آن زیربخش‌ها مالیات جمع‌آوری کند.

متاسفانه به علت ابهامات مربوط به محل‌های هرکدام از کشاورزان، مامور نمی‌تواند دقیقا تعیین کند که الان در کدام زیربخش ایستاده است. بنابراین از شما به عنوان یک برنامه‌نویس با تجربه خواسته است که این کار را برای او انجام دهید.

ورودی (ورودی استاندارد)

خط اول ورودی شامل یک عدد صحیح t $(t \le 10)$ می‌شود که تعداد آزمون‌ها را مشخص می‌کند. در خط اول هر آزمون، دو عدد صحیح n و W $(1 \le n \le 5\times10^4, 1 \le W \le 10^9)$ می‌آید که به ترتیب تعداد زیربخش‌ها و پارامتر W را که در صورت مسأله شرح داده شد، مشخص می‌کنند. از این پس فرض می‌کنیم که زمین یک محدوده‌ی بدون مرز بین خطوط $y=0$ و $y=W$ است.

در n خط بعد، دو عدد صحیح x_1 و x_2 $(-10^9 \le x_1, x_2 \le 10^9)$ می‌آید. این اعداد به این معنی هستند که یک پاره‌خط از نقطه‌ی $(x_1,0)$ تا (x_2,W) وجود دارد. تضمین می‌شود که این پاره‌خط‌ها اشتراکی با یکدیگر ندارند.

در خط بعد یک عدد صحیح m $(1 \le m \le 5\times10^4)$ می‌آید که تعداد مکان‌هایی است که مامور می‌خواهد بررسی کند. بعد از آن m خط دیگر می‌آید که در هرکدام از آنها دو عدد صحیح x_i و y_i $(-10^9 \le x_i \le 10^9, 0 \le y_i \le W)$ برای مشخص کردن مکان iام مورد نظر مامور هستند. نقاط مورد بررسی روی هیچ‌کدام از پاره‌خط‌ها قرار نمی‌گیرند.

خروجی (خروجی استاندارد)

برای هر آزمون شما باید m خط چاپ کنید. خط iام هر آزمون شماره‌ی زیربخش نقطه iام است. پاسخ مربوط به هرکدام از نقاط باید عددی بین 0 تا n باشد.

پس از هر آزمون باید یک خط خالی چاپ شود.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 1 2 10 0 2 5 0 5 2 4 10 8 8 3 1 4 5 15 12 5 2 1 9 10 3 1 6 0 14 5	0 1 0 3 1 2 4

مسأله‌ی ۷: رفقای خوب

تعزیه نوع خاصی از هنرهای نمایشی پارسی است و به دوره‌ی پیش از اسلام بر می‌گردد. تراژدی سیاوش یکی از بهترین نمونه‌های این هنر نمایشی است.

تعزیه معمولا روی یک صحنه‌ی نمایش دایره‌ای اجرا می‌شود. در زمان‌های دور که تعزیه محبوب‌تر بود، افراد خانواده‌های مختلف دور یکدیگر جمع می‌شدند تا نظاره‌گر نمایش باشند. هر خانواده چند عضو مرد و چند عضو زن داشت. مردم می‌توانستند دور صحنه‌ی نمایش با هر ترتیبی قرار بگیرند. البته محدودیت‌هایی به این ترتیب وجود داشت:

- یک مرد و یک زن از خانواده‌های مختلف نمی‌توانند کنار یکدیگر بایستند.
- مردم می‌توانند در چندین ردیف حلقوی قرار بگیرند. ولی در هر ردیف باید کاملا نزدیک یکدیگر بایستند. به این ترتیب هرکسی مجاور افرادی که (در صورت وجود) قبل یا بعد از او ایستاده‌اند، محسوب می‌شود.

شما در حال مطالعه‌ی رفتار مردم قدیمی هستید تا بیشتر در مورد ویژگی‌ها و سنت‌های اجتماعی آنها بیاموزید. با توجه به توضیحات مربوط به چند خانواده در یک نمایش تعزیه، شما باید حداقل تعداد سطرهایی را که آنها با رعایت محدودیت‌های ذکر شده می‌توانند تشکیل دهند، مشخص کنید.

ورودی (ورودی استاندارد)

ورودی شامل چند آزمون می‌شود. تعداد آزمون‌ها حداکثر 100 است. در خط اول هر آزمون یک عدد صحیح n $(1 \le n \le 100)$ می‌آید که همان تعداد خانواده‌ها است.

در خط بعد n عدد صحیح m_i می‌آید که تعداد مردها در خانواده‌ی iام را مشخص می‌کند. اطلاعات مربوط به زن‌های هر خانواده w_i نیز همانند قالب قبلی در خط بعد می‌آید $(0 \le w_i, m_i \le 9)$. تضمین می‌شود که برای هر i بین 1 و n، $w_i + m_i > 0$. ورودی با $n = 0$ خاتمه می‌یابد.

خروجی (خروجی استاندارد)

برای هر آزمون شما باید حداقل تعداد سطرهای حلقوی مورد نیاز را چاپ کنید تا مردم بتوانند بدون نقض قوانین به مشاهده‌ی نمایش بپردازند.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 1 1 0 1 3 2 1 3 0 9 2 0	2 1

مسأله‌ی ۸: تجربه‌ی انسان

شما باید با گرفتن n جفت عدد صحیح (a_i, b_i)، مقدار باقیمانده‌ی حاصل از تقسیم معادله‌ی زیر را در پیمانه‌ی $10^9 + 7$ محاسبه کنید:

$$LCM\,(a_1^{b_1}, a_2^{b_2}, ..., a_n^{b_n})$$

منظور از LCM کوچکترین مضرب مشترک مجموعه‌ای از اعداد است.

ورودی (ورودی استاندارد)

خط اول ورودی حاوی یک عدد صحیح t $(t \le 50)$ است که تعداد آزمون‌ها را مشخص می‌کند. در خط اول هر آزمون، یک عدد صحیح n $(1 \le n \le 10^5)$ می آید که تعداد جفت‌های اعداد صحیح است. خط iام از n خط بعدی حاوی دو عدد صحیح a_i و b_i $(1 \le a_i \le 10^6, 0 \le b_i \le 10^9)$ است که نشان‌دهنده‌ی پایه و توان جفت iام است (همانگونه که در صورت مسأله نشان داده شد).

خروجی (خروجی استاندارد)

برای هر آزمون شما باید یک خط چاپ کنید که همان جواب باقیمانده‌ی مقدار معادله در پیمانه‌ی $10^9 + 7$ است.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 3 2 4 15 2 6 3 1 2 1000	10800 688423210

مسأله‌ی ۹: کوله‌پشتی جالب

کوله‌پشتی یک مسأله‌ی معروف بهینه‌سازی است. در نسخه‌ی صفر و یک این مسأله، به شما لیستی از n عنصر به همراه اندازه و ارزش آنها و همچنین یک کوله به اندازه‌ی B داده می‌شود. شما باید زیرمجموعه‌ای از عناصر را که در کوله جا می‌گیرند، انتخاب کنید به صورتی که ارزش نهایی آنها بیشینه شود. کوله‌پشتی یک مسأله‌ی NP-hard است؛ با این حال یک الگوریتم پویای شبه چندجمله‌ای دارد.

در این مسأله شما باید دقیقا همین مسأله را با یک محدودیت اضافی حل کنید: یا کوله‌ی شما خیلی کوچک است یا حداکثر ارزش عناصر محدود است.

ورودی (ورودی استاندارد)

ورودی حداکثر دارای ۵۰ آزمون خواهد بود. خط اول هر ورودی دارای دو عدد صحیح n و B $(1 \le n \le 100, 1 \le B \le 10^8)$ است که نشان‌دهنده‌ی تعداد عناصر و اندازه‌ی کوله‌پشتی هستند. در خط دوم n عدد صحیح می‌آید که عدد iام آن یعنی v_i $(1 \le v_i \le 10^7)$ ارزش عنصر iام را مشخص

می‌کند. خط سوم اندازه‌های هرکدام از عناصر یعنی s_i $(1 \le s_i \le 10^7)$ را مشابه قالب خط قبل مشخص می‌کند.

تضمین می‌شود که در هر آزمون $B \le 50000$ یا $\max_{1 \le i \le n} v_i \le 500$.

ورودی با خط "0 0" خاتمه می‌یابد.

خروجی (خروجی استاندارد)

برای هر آزمون شما باید یک خط چاپ کنید که حداکثر ارزش عناصر درون کوله‌پشتی را مشخص می‌کند.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 100000 1 500 1 100000 3 10 3 5 7 4 6 7 5 100 1 2 3 4 5 5 4 3 2 1 0 0	500 8 15

مسأله‌ی ۱۰: اتصال رئوس

به شما یک گراف بدون جهت وزن‌دار با n راس و m یال داده می‌شود. شما در یک حرکت می‌توانید هرکدام از دو راس متفاوت را به یک راس تبدیل کنید که هزینه‌ی این عملیات برابر طول کوتاه‌ترین مسیر بین دو راس است. توجه کنید که بعد از اتصال چند راس، ممکن است یال‌های چندگانه یا چرخه در گراف به وجود بیاید. اتصال راس‌ها تا زمانی که گراف القایی تنها یک راس داشته باشد ادامه می‌یابد.

وظیفه‌ی شما این است که تمامی راس‌ها را به هم متصل کنید تا یک راس باقی بماند، به طوری که هزینه‌ی نهایی این عملیات کمینه شود.

ورودی (ورودی استاندارد)

در خط اول ورودی عدد صحیح t $(t \le 40)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند. خط اول هر آزمون شامل دو عدد صحیح n و m $(1 \le n \le 1000, m \le 200000)$ است که به ترتیب تعداد راس‌ها و یال‌های گراف را مشخص می‌کند. در m خط بعد سه عدد صحیح u_i، v_i و w_i $(1 \le u_i, v_i \le n, 1 \le w_i \le 10^6)$ می‌آید که نشان‌دهنده‌ی وجود یک یال با وزن w_i بین راس‌های u_i و v_i است. گراف حاوی یال‌های چندگانه یا چرخه نخواهد بود.

خروجی (خروجی استاندارد)

به ازای هر آزمون شما باید حداقل هزینه‌ی مورد نیاز برای منقبض کردن تمامی رئوس تا رسیدن به یک راس را در یک خط چاپ کنید. اگر چنین چیزی ممکن نیست، کلمه‌ی "impossible" را چاپ کنید.

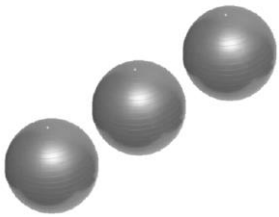
نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 4 5 1 2 2 3 1 1 4 3 3 2 4 3 3 2 5 4 2 1 2 10 3 4 20	6 impossible

۷-۹-۲- مسابقات ملی سال ۹۲ – دانشگاه کاشان – دوازده سوال

Problem A: Arrangement of RGB Balls

Sohrab & Sepehr have n colorful balls, R of them are red, B of them are blue and G of them are green.



They want to arrange these balls in a single line. The only condition should be met is, any three consecutive balls, should not have two or more balls of the same color. It is your job to help them find out how many different ways these balls can be arranged, if all the same colored balls are indistinguishable.

Two colorful arrangements A and B are different, if there is a position i such that $color(A_i) \neq color(B_i), (1 \leq i \leq n)$.

Input (Standard Input)

The first line of input contains an integer $T\,(T \leq 100)$ indicating the number of test cases.

Each of next T lines have three integer numbers, $R, G, B (1 \le R+G+B \le 10^6)$, indicating number of red, green and blue balls, respectively.

Output (Standard Output)

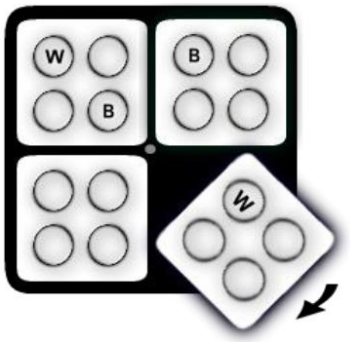
For each test case print the number of different possible colorful lines that can be made.

Sample Input and Output

Sample Input	Sample Output
4 5 5 5 3 3 2 0 0 1 3 4 5	6 2 1 0

Problem B: Bentago

Pentago is a two-player abstract strategy game invented by Tomas Flodén which played on a 6×6 board. "Bentago" is similar to Pentago, but is played on a board 4×4 divided into four 2×2 sub-boards. Taking turns, the two players place a marble of their color (either black or white) onto an unoccupied space on the board and then rotate one of the sub-boards by 90 degrees either clockwise or anticlockwise. The white player starts the game. A player wins the game by having three of their marbles in a vertical, horizontal or diagonal alignment after the sub-board rotation in his move. If all 16 spaces on the board are occupied without a row of three being formed then the game is draw. Furthermore, if after some turn, both players have three marbles in a vertical, horizontal or diagonal alignment, the game is draw too.



Sohrab & Sepehr are playing Bentago and Soodabeh is watching their game. By having the current game state, she wants to know the result of the game. Note that Sohrab plays as white and Sepehr plays as black and both of them play optimally well.

Input (Standard Input)

The first line of input contains an integer $T\,(T \le 50)$, indicating the number of test cases.

Each test case consists of five lines. Each of the first four lines contains four characters; these lines describe the position on the board. The fifth line of each test case is empty.

The position on the board is specified using characters "W" (white marble), "B" (black marble), and "." (free cell). You may assume that all positions in input are reachable in a real game in accordance with the described rules and all of given positions need at least one turn to terminate.

Output (Standard Output)

For each test case if white player wins the game, output "Sohrab". If black player wins, output "Sepehr", In the case of a draw, output "Draw". (quotes for clarification).

Sample Input and Output

Sample Input	Sample Output
3 BWWB WBWB .B.. .W.. WBWB BWWB .W.. .... BBWB W..W WW.. B...	Sohrab Sepehr Draw

Problem C: Combination Lock

Sohrab & Sepehr found a combination lock. The lock has rotating dials in which all of them have n adjacent sides. Integer numbers from 1 to n are written on dials respectively. Rotating one dial clockwise increases the number on dial. Numbers 1 and n are adjacent that is if the dial number is n and you increase it, the dial will set to number 1. Sohrab & Sepehr have invented a game with this lock and they are about to play it.

At first, Sohrab takes the lock and rotates dials in order to set them to any arbitrary combination. Sepehr has n turns to change the dial numbers. In i^{th} turn he can select one of dials that he didn't select before and rotates it exactly i times clockwise.

Sepehr is the winner of game if after his n turns, the combination lock has all of numbers between 1 to n, (i.e. formed a permutation of numbers) otherwise Sohrab is the winner of the game.

You are given the combination lock which Sepehr has received. Determine who will be the winner of the game in case you know Sepehr plays optimally well.

Input (Standard Input)

The first line of input contains an integer $T\ (T \le 100)$, indicating the number of test cases.

Each test contains an integer $n\ (n \le 13)$ indicating number of dials on the lock, followed by n space separated integers, $A_i\ (1 \le A_i \le n)$ describing Sohrab's chosen combination.

Output (Standard Output)

For each test case output "Sohrab" or "Sepehr", corresponds to the winning side.

Sample Input and Output

Sample Input	Sample Output
2 2 1 2 3 2 1 3	Sohrab Sepehr

Problem D: Dreamer Land of Kashan

Sohrab & Sepehr are responsible for Kashan's taxi organization which has n taxi stations and m taxis. There exists exactly one road between any two taxi stations. Each of taxis has chosen their favorite road to drive in, that has not been chosen by any other taxi.

Sohrab & Sepehr soon realized that some stations are not accessible from each other. Now they want to solve this issue. After a while of consulting, these options are presented:

- Forcing some taxis to change their road by paying them. Each taxi requests some money for changing his favorite road to any other road.
- Employing some new taxis to drive in a specified road has determined by Sohrab & Sepehr. The cost of employing a new taxi c is and they are able to employ as many new taxis as they want.

Sohrab & Sepehr's primary concern is to find minimum total cost needed to solve this big problem. Also, if possible, they prefer to solve this problem with employing new taxis as a few

as possible. As now, you are informed of their goal; your mission is to help them solve three problems. Firstly, how many new taxis must be employed? Secondly, how many old taxis must be forced to change their road? Ultimately, what is the minimum total cost?

Input (Standard Input)

The first line of input contains an integer $T\,(T \le 35)$ indicating the number of test cases.

Each test case starts with one line contains three integer numbers n, m and $c\,(2 \le n \le 10^4, 0 \le m \le 10^4, 1 \le c \le 10^5)$, the number of stations in the city, the number of taxis that have already employed and the cost of employing new taxis respectively.

Then the next m lines contains three integers a_i, b_i and $r_i\,(1 \le a_i, b_i \le n, a_i \ne b_i, 1 \le r_i \le 10^5)$, indicating that the i^{th} taxi has selected the road between stations a_i and b_i and he wants r_i unit of money to change his road. Stations are numbered in the range of 1 to n. It is guaranteed that no two taxis select the same road.

Output (Standard Output)

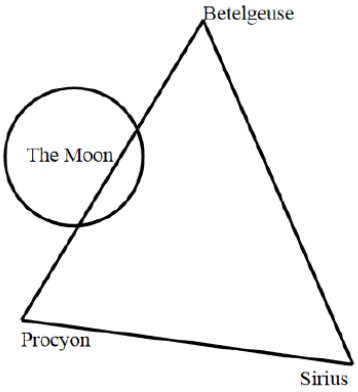
For each test case output three space separated numbers, representing the number of new taxis, the number of old taxis which should be forced to change their road and the total cost.

Sample Input and Output

Sample Input	Sample Output
3 3 1 100 1 2 100 4 3 10 1 2 10 2 3 20 1 3 30 5 3 15 3 5 10 5 2 20 2 3 30	1 0 100 0 1 10 1 1 25

Problem E: Era of Winter Triangle

Sohrab & Sepehr have decided to pursue their studies in astronomy. They have recently been interested in the Winter Triangle. The Winter Triangle is made of Sirius, Procyon and Betelgeuse which are three of the ten brightest objects observable from the Earth. They are planning to observe the Winter Triangle in the desserts around Kashan. They are wondering if the Moon has got any points in common with the Winter Triangle.



We model sky as a 2D plain. Stars are just like points in that plain. We have their coordinates, and also we model Moon as a circle.

Your job is to check if there is any intersection between the Winter Triangle and the Moon or not. (At least in one point)

Input (Standard Input)

The first line of the input contains an integer $T\,(T \le 1000)$, indicating the number of test cases.

Each of the first three lines of test cases represents a pair of integers describing triangle's vertices.

The next line contains three integers X, Y, R. The point (X, Y) is the center of the Moon, and R is radius of the Moon.

All coordinate numbers in the input are less than or equal in absolute value and $(1 \le R \le 1000)$. Also it's guaranteed that no three vertices of the given triangles lie on a straight line.

Output (Standard Output)

For each test case print a single line containing "YES" if the Winter Triangle and the Moon have at least one common point; and "NO" otherwise.

Sample Input and Output

Sample Input	Sample Output
3 1 1 3 1 2 3 2 0 2 1 1 3 1 2 3 2 0 1 1 1 3 1 2 3 0 0 1	YES YES NO

Problem F: Find The Problem!

Sohrab & Sepehr are problem-setters of 5^{th} Kashan programming contest. After they prepared their problems, the sponsor of event gives them two new raw problems. He asks them to add the two new problems to the problem set. Now you should solve one of those problems. Sohrab & Sepehr doesn't have any time to prepare the problem, so they explain the original one given by sponsor and you have to discover what it actually asks, then solve it!

Machine	Part	Code
R1	P1	C1
R1	P2	C2
R1	P3	C3
R1	P4	C4
R2	P1	C5
R2	P5	C6
R2	P6	C7
R2	P7	C8
R3	P5	C9

Fig1. Mysterious table

	P1	P2	P3	P4	P5	P6	P7
R1	C1	C2	C3	C4			
R2	C5				C6	C7	C8
R3					C9		

Fig2. Strange table

In this problem you are given a mysterious table and you should change it to a strange one! See the example to find out how to do it.

Input (Standard Input)

The first line of input contains an integer $T\,(T \le 50)$ indicating the number of test cases.

First line of each test case contains an integer number $N\,(1 \le N \le 50)$ which is the number of data rows in the mysterious table. In each of next N lines there are three space-separated words $Machine_i, Part_i, Code_i$ representing saved data fields in the i^{th} row of mysterious table. All data in the table only consists of uppercase English letters and digits. Length of all words are less than or equal to 3 characters. It's guaranteed that in the given mysterious table all pairs $(Machine_i, Part_i)$ are distinct.

Output (Standard Output)

For each test case, output a strange table just like the sample input and output. There are two secret points you must consider:

- All data fields must contain exactly 3 characters and must be left justified. If the length of any field data is less than 3, you should add trailing space characters to it.
- Row and column names should appear in ascending lexicographical order.

Sample Input and Output

Sample Input	Sample Output
2 9 R1 P1 C1 R1 P2 C2 R1 P3 C3 R1 P4 C4 R2 P1 C5 R2 P5 C6 R2 P6 C7 R2 P7 C8 R3 P5 C9 1 R1 P1 CCC	+---+---+---+---+---+---+---+---+ | |P1 |P2 |P3 |P4 |P5 |P6 |P7 | +---+---+---+---+---+---+---+---+ |R1 |C1 |C2 |C3 |C4 | | | | +---+---+---+---+---+---+---+---+ |R2 |C5 | | | |C6 |C7 |C8 | +---+---+---+---+---+---+---+---+ |R3 | | | | |C9 | | | +---+---+---+---+---+---+---+---+ +---+---+ | |P1 | +---+---+ |R1 |CCC| +---+---+

Hint

The lexicographical order of strings is the order we are all used to, the "dictionary" order. Such comparison is used in all modern programming languages to compare strings. Formally, a string p of length n is lexicographically less than string q of length m, if one of the two statements is correct:

$n < m$, and p is the beginning (prefix) of string q (for example, "aba" is less than string "abaa"),

$p_1 = q_1, p_2 = q_2, \ldots, p_{k-1} = q_{k-1}, p_k < q_k$ for some $k\,(1 \le k \le \min(n, m))$, here characters in strings are numbered starting from.

Problem G: General Sohrab & General Sepehr!

Sohrab & Sepehr are generals of the Kashan army. During daily PT formation, soldiers stand in a line, numbered from 1 to n (left to right). In addition, each of them is either faced to right or left direction.

Sohrab & Sepehr give training commands to soldiers. Sohrab only likes to give "Sit Down" or "Stand Up" commands, and Sepehr only likes to give "Turn Around" command. Note that each commands to a specific soldier. So after some training commands, a number of soldiers are standing while some of them are sitting. Also some of them are in directions other than their initial direction.

Two soldiers are called "Strong Soldier Pair" (SSP), if they meet all conditions below:

- Both soldiers are standing.
- Left soldier faced to right and right soldier faced to left.
- There is no standing soldier between them.

At some points, Sohrab & Sepehr want to know how many "Strong Soldier Pair" exist, in which both soldiers are between L and R (inclusive).

Your program should work with 3 types of queries:

1. Sohrab commands to soldier number X. (Toggle between sitting or standing position)
2. Sepehr commands to soldier number X. (Toggle between facing to right or facing to left)
3. How many "Strong Soldier Pair" lies between L and R? (inclusive).

Input (Standard Input)

The first line of input contains an integer $T\,(T \le 15)$ indicating the number of test cases.

First line of each test case contains two integer numbers n and $q\,(2 \le n \le 10^5, 1 \le q \le 10^5)$ indicating number of soldiers and number of queries respectively.

The next line contains a string S of length n that each character is either '<' or '>', indicating direction of soldiers. ('<' for left and '>' for right)

The i^{th} of the following q lines first contains an integer $T_i\,(1 \le T_i \le 3)$ indicating type of i^{th} query. If the i^{th} query is of type 1 or 2, then next follow one integer $X_i\,(1 \le X_i \le n)$. If the i^{th} operation is of type 3, then next follow two integers $L_i, R_i\,(1 \le L_i \le n-1, L_i+1 \le R_i \le n)$. The numbers on the lines are separated by single spaces.

Output (Standard Output)

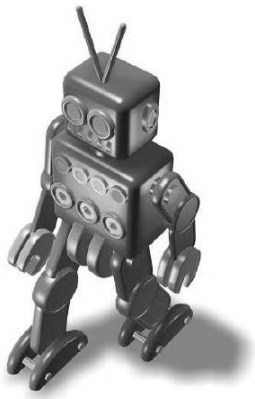
For each query of type 3, print a single line, the number of "Strong Soldier Pairs" that lies in the given segment. Print the answers to the queries in the order in which the queries go in the input.

Sample Input and Output

Sample Input	Sample Output
2 6 10 ><><>< 3 1 6 1 2 3 1 6 2 3 3 1 6 3 2 6 3 2 5 1 2 3 1 6 3 3 4 4 4 >><< 3 1 4 1 2 1 3 3 1 4	3 2 2 1 0 2 0 1 1

Problem H:

Sohrab and Sepehr bought a new robot. The robot has a special program to eat fruits. To feed the robot, its owner should put some fruits in a row on the ground. Fruits are numbered from left to right starting from 1. Additionally the owner should choose and insert a special digital counter inside the back of robot's head. The digital counter only can count from 0 to $M-1$ and in case of overflow, it starts from 0. (e.g. if $M=8$ and digital counter has number 6, it will has 1 after counting 3 more times). In addition, the owner should set four integers L, R, X, Y before run the feeding program.



Note that for technical reasons both of L and R should be a multiple of M. In robot's instruction manual there is an algorithm that shows how feeding program works. In the algorithm some of fruits called "ILLEGAL". A fruit is "ILLEGAL" if at least one of these conditions is true:

a. It is one of first L fruits.
b. It is one of last R fruits.
c. The number that special digital counter shows, when robot stands beside that fruit, is less than X.
d. The number that special digital counter shows, when robot stands beside that fruit, is greater than Y.

According to instruction manual of robot, the feeding algorithm works in this way:

1. Get the integer numbers L, R, X, Y from your owner.
2. Wait until your owner mounts a special counter on your head.
3. Set special counter to 0.
4. Stand beside of fruit number 1.
5. If the fruit that is in your side is not "ILLEGAL", then eat it.
6. If recently you ate an unripe fruit, turn off yourself immediately!
7. See the next step. If after the next step there is a fruit in your side go to line 8, otherwise go to line 11.
8. Go 1 step forward.
9. Increase special digital counter one unit.
10. Go to line 5.
11. Increase special digital counter one unit.

12. If special digital counter shows 0 go to line 13, otherwise turn off yourself immediately.
13. Mission completed successfully.

Sohrab & Sepehr read the instruction manual. However, they didn't understand anything of it! Unfortunately, their robot is hungry!

They bring all fruits that exist at home and put them on the ground. Some of fruits are unripe and others are ripe.

Before run robot's feeding program they should choose a digital counter with size M , and set four integer numbers L, R, X, Y for the robot. They would like know between any feasible values for M, L, R, X, Y what is the maximum number of fruits that their robot can eat without turning off itself. They just know you as an expert that can help them out. You are given the number of fruits and the indexes of unripe fruits. Write a program that find maximum possible fruits, the robot can eat according to robot's feeding algorithm.

Input (Standard Input)

First line of input contains an integer number $T\,(T \le 50)$ indicating number of test cases.

Each test case has two lines. First line of each test case contains two integer numbers $n\,(n \le 20000)$ and $m\,(0 \le m \le n)$ indicating number of fruits and number of unripe fruits. In the second line there are m spaces separated integers $A_i\,(1 \le i \le m, 1 \le A_i \le n)$ that indicates indexes of unripe fruits. Numbers in A is sorted in ascending order.

Output (Standard Output)

For each test case output the maximum number of fruits the robot can eat with correct configuration.

Sample Input and Output

Sample Input	Sample Output
3 5 1 3 6 2 1 4 12 3 1 8 9	2 4 6

Problem I: ICPC Team Selection

Sohrab & Sepehr are coaches of ACM/ICPC at the University of Kashan. This year, director of Tehran site allows them to send only one team to participate in the regional contest. This is an unfavorable situation, since there are too many good contestants at the university and only three of them will have a chance to be in the team.

Sohrab & Sepehr use some interesting policy to select team members. At the first they declare that every volunteer should have an account in TupCoder site. TupCoder is a site in which holds regular programming contests and each member have a rating considering his/her earned results. So every member of TupCoder has a unique overall ranking. (For members have equal ratings, there is some rules to break the tie). It is obvious that if a student has better rank, he is probably be a better contestant for ICPC.

In addition Sohrab & Sepehr should not forget newbies because participating in regional contest is an excellent motivation for them to practice for contests ahead. Thus, Sohrab & Sepehr consider registration date of members on TupCoder too.

According to K&P's policy, every possible team should have three posts like a football team! A newbie, an experienced, and a wild member. Below is definition of every post:

1. Newbie member: This member should have worst TupCoder rank, and joined to TupCoder after other teammates.
2. Experienced member: he/she stands between newbie and wild in ranking and had joined TupCoder before other teammates.
3. Wild member: he/she should have best TupCoder rank in the team, but joined between newbie and experienced member.

Three students can form a team, if their team has a member for all of three above posts. All possible teams that can be formed is equiprobable.

You are given number of volunteers at the university of kashan, registration date and the ranking on TupCoder. For every student output probability that he/she may be selected as a kashan's team member.

Input (Standard Input)

First line of input contains an integer number $T\,(T \le 30)$ indicating the number of test cases.

First line of each test case contains an integer number $N\,(3 \le N \le 5000)$ indicating number of volunteer students. $(i+1)^{th}$ line of each test contains D_i, R_i, indicating registration date and ranking of student i. D_i is in $YYYY/MM/DD$ format and is a correct date between $1990/01/01$ to $2012/12/31$ and $1 \le R_i \le 100000$.

Output (Standard Output)

For each test case if Sohrab & Sepehr can't select any team with such rules print "Rules Should Be Changed!" in a line (quotes for clarification). Otherwise, print chances of every student to be chosen in Kashan's team, in order of that appeared in input. All numbers should be rounded and fixed to 5 digits after decimal. Print exactly one space between them.

Sample Input and Output

Sample Input	Sample Output
3 3 2010/03/07 1500 2010/02/25 2500 2012/08/02 3500 5 2007/08/05 4 2007/07/11 5 2005/11/01 3 2006/05/31 2 2004/10/10 1 3 2006/05/31 123 2009/12/14 234 2008/07/19 345	1.00000 1.00000 1.00000 0.50000 0.50000 1.00000 1.00000 0.00000 Rules Should Be Changed!

Problem J: Jokey Problem!

Sohrab & Sepehr are mayors of Kashan. They are calling agents of all NGO's in the Dreamers land to a very important meeting. In the meeting room there is a round table that has M seats that are numbered from 0 to $M-1$ in clockwise order. (The table is circular so seats $M-1$ and 0 are adjacent as well).

One day before meeting all groups should claim their desired seats for their agents. Each group selects one segment of adjacent seats. For this they choose two seat numbers F_i and L_i, it means that they want all seats between F_i and L_i in clockwise order inclusive.

Sohrab & Sepehr know how many seats we have and which seats is selected by each group. They want to know what is the maximum possible group can take part in the meeting.

Input (Standard Input)

The first line of test cases has an integer number $T\,(T \le 25)$ that indicates number of test cases.

In the first line of each test case there is two integers N and M, the number of groups and number of seats respectively. $(1 \le N \le 10^5, 3 \le M \le 10^8)$

In the each of next N lines there is two numbers F_i and L_i that indicates first and last seats that group i selected (From F_i to L_i in clockwise order and inclusive). $(1 \le F_i, L_i \le M - 1)$

Output (Standard Output)

For each test case print a single integer number indicates maximum possible number of groups they can invite to meeting.

Sample Input and Output

Sample Input	Sample Output
2 5 10 8 0 9 1 2 4 3 6 5 7 6 10 0 1 1 2 2 3 3 5 5 8 8 0	3 3

Problem K: Killer Challenge

Sohrab has an integer sequence A of length n, he has to split the sequence into consecutive subsequences. Let R be the result of multiplication of sum of all subsequences. Furthermore, he has an integer number P which is $P = p_1 \times p_2 \times ... \times p_n$. (p_i Is the different prime numbers for each i between 1 to n)

He would like to maximize the value of $GCD(P,R)$. In addition, because Sohrab likes addition more than multiplication, he wants to split original array to minimum number of subsequences. You should help him to achieve this goal.

Input (Standard Input)

The first line of input contains integer number $T\,(T \le 100)$ the number of test cases.

Each test case starts with a line contains two integer numbers, $n(1 \le n \le 100)$ which represents the number of elements of Sohrab's original sequence, and the number $P\,(2 \le P \le 10^6)$. The next line contains n space separated integer numbers representing the sequence $(1 \le A_i \le 10^6, 1 \le i \le n)$.

Output (Standard Output)

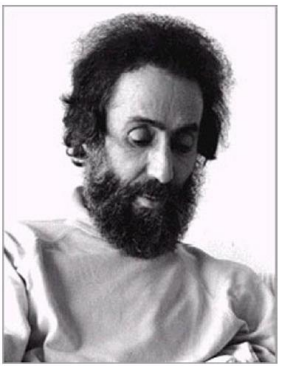
For each test case print two integer numbers separated by a single space, the first must be the maximum of $GCD(P,R)$ can be obtained by splitting the sequence and the next is the minimum number of cuts needed to obtain that GCD number.

Sample Input and Output

Sample Input	Sample Output
3 2 2 1 2 3 10 9 5 3 4 6 7 1 4 5	2 1 5 2 6 1

Problem L: Lexicographically Minimal Poem

Sohrab Sepehri (Persian:سهراب سپهری) (October 7, 1928 - April 21,1980) was a notable modern Persian poet and painter.



He was born in Kashan in Isfahan province. He is considered to be one of the five most famous modern Persian (Iranian) poets who have practised "New Poetry" (a kind of poetry that often has neither meter nor rhyme). Other practitioners of this form were Nima Youshij, Ahmad Shamlou, Mehdi Akhavan-Sales, and Forough Farrokhzad.

His poetry is full of humanity and concern for human values. He loved nature and refers to it frequently.

After all he was through, Sohrab eventually has become a great poet. Sepehr is now a professional cryptographer! They would like to have a little fun and play an interesting game. The game is based on substitution cipher method.

Substitution cipher is defined by a substitution key, assigning each letter of alphabet to another letter (the assignment is a one-to-one mapping between lower-case English letters). The key is a 26-letter string mapping the i^{th} English letter to the key_i.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧
x	m	u	r	p	w	e	l	i	f	s	o	b	d	h	q	k	c	a	j	t	z	n	g	v	y

Fig1. An example of a substitution key.

For example, after encryption of the poem with the key in the figure above, all occurrences of character 'a' will be replaced with 'x' and the same goes for all other characters. The result of this process is called "cipher-text".

Sohrab selects one of his best poems, and encrypts it using substitution cipher with a key. He repeats this process with the same key for X times (the result of each stage of encryption is given as input to the next stage). Then he gives the result of the final stage of encryption to Sepehr; to make it harder for Sepehr to decrypt the ciphertext, Sohrab doesn't give him the number X (number of times he encrypted the poem). Sohrab gives Sepehr just the encryption key and the cipher-text.

Because Sepehr likes Sohrab's poem, he wants to decrypt his encrypted poems. Sepehr knows that he can decrypt the encrypted poems because he knows that the encryption method is so weak. Not knowing the X, he might find many different texts (candidate poem). There's a

hint which Sohrab gives Sepehr: "among all the candidate poems, one which is lexicographically minimal is my poem".

As Sepehr knows you as a good programmer, he wants you to find Sohrab's poem.

The lexicographic order of strings is the familiar to us "dictionary" order. Formally, the string p of lengths n is lexicographically smaller than string q of the same length, if $p_1 = q_1, p_2 = q_2, ..., p_{k-1} < q_{k-1}, p_k < q_k$ for some $i (1 \le i \le n)$. Here characters in the strings are numbered from 1. The characters of the strings are compared in the alphabetic order.

Input (Standard Input)

In the first line of input there is an integer number $T (T \le 25)$ indicating number of test-cases.

Each test-case contains three lines. In the first line of each test-case there is an integer number $(n \le 10^6)$.

Second line contains string S. String S has length n and only consist of lowercase English letters and representing encrypted poem.

In the third line, the key of encryption is given as a string consists of 26 lowercase English letters. key has each of English letters axactly one.

Output (Standard Output)

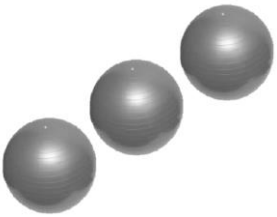
For each test case print the poem of Sohrab in a single line.

Sample Input and Output

Sample Input	Sample Output
3 7 pumiuau xmurpwelifsobdhqkcajtzngvy 29 yzqjxynzysyrmttbjaymyrgyisvno zcdefghijkamnopqrstulwxybv 25 bezmnftheuecthepegucagtz cbafedihglkjonmrqputsxwvzy	acmicpc ahlekashanamroozegarambadnist beyondtheseathereisacity

مسأله‌ی ۱: چینش توپ‌های قرمز، سبز و آبی

سهراب و سپهر n توپ رنگی دارند که R تای آنها قرمز، B تای آنها آبی و G تای آنها سبز هستند.



آنها می‌خواهند این توپ‌ها را در یک خط کنار هم بچینند. تنها قانونی که باید در اینجا رعایت شود، این است که در بین هر سه توپ متوالی، نباید تعداد دو یا بیشتر از توپ‌های هم رنگ وجود داشته باشد. وظیفه‌ی شما این است که به آنها کمک کنید که متوجه شوند چند راه مختلف برای چینش توپ‌ها به این صورت وجود دارد (توپ‌های همرنگ کاملا یکسان و مشابه فرض می‌شوند).

دو چینش رنگی A و B را متفاوت فرض می‌کنیم، اگر مکانی مانند i وجود داشته باشد که $(1 \le i \le n)\ \ color(A_i) \ne color(B_i)$.

ورودی (ورودی استاندارد)

در خط اول ورودی عدد صحیح T $(T \le 100)$ می‌آید که نشان‌دهنده‌ی تعداد آزمون‌ها است. هرکدام از T خط بعدی حاوی سه عدد صحیح R، G و B $(1 \le R + G + B \le 10^6)$ است که به ترتیب تعداد توپ‌های قرمز، سبز و آبی را مشخص می‌کنند.

خروجی (خروجی استاندارد)

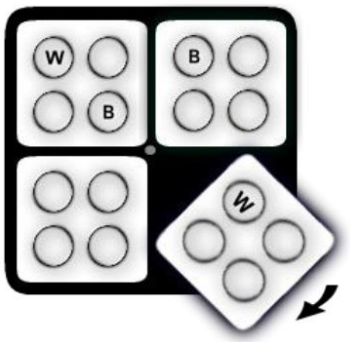
برای هر آزمون تعداد راه‌های مختلف برای چیدن توپ‌های رنگی در یک خط را چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
4 5 5 5 3 3 2 0 0 1 3 4 5	6 2 1 0

مسأله‌ی ۲: بِنتاگو

پِنتاگو یک بازی استراتژی انتزاعی دو نفره است که توسط توماس فِلودِن[۷۰] اختراع شده و روی یک تخته‌ی 6×6 بازی می‌شود. بِنتاگو نیز مشابه پِنتاگو است، ولی از یک تخته‌ی 4×4 که به چهار زیرتخته‌ی 2×2 تقسیم می‌شود، استفاده می‌کند. دو بازیکن به نوبت مهره‌ای از رنگ مربوط به خود (که مشکی یا سفید است) را در یکی از فضاهای اشغال نشده‌ی تخته قرار می‌دهند و سپس یکی از زیرتخته‌ها را 90 درجه در جهت ساعتگرد یا پادساعتگرد می‌چرخانند. بازیکن سفید بازی را آغاز می‌کند. بازیکنی که پس از چرخاندن زیرتخته در نوبت خود تعداد سه یا بیشتری از مهره‌های خود را در یک خط افقی، عمودی یا قطری قرار دهد، برنده می‌شود. اگر تمامی 16 فضای روی تخته اشغال شوند، بدون این که ردیفی از سه مهره‌ی همرنگ تشکیل شود، نتیجه بازی مساوی خواهد بود. علاوه بر این، اگر در یک نوبت هر دو بازیکن دارای سه مهره همرنگ به صورت افقی، عمودی یا قطری شوند، بازی مساوی اعلام می‌شود.



سهراب و سپهر در حال انجام بازی بِنتاگو هستند و سودابه نظاره‌گر بازی است. با مشاهده‌ی حالت فعلی بازی، سودابه می‌خواهد نتیجه‌ی بازی را بداند. دقت کنید که سهراب بازیکن سفید و سپهر بازیکن مشکی است و هر دوی آنها به صورت بهینه بازی می‌کنند.

ورودی (ورودی استاندارد)

در خط اول ورودی عدد صحیح T $(T \le 50)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند. هر آزمون در پنج خط معرفی می‌شود. هرکدام از چهار خط نخست حاوی چهار نویسه خواهد بود؛ این خطوط نشان‌دهنده‌ی موقعیت‌های روی تخته هستند. خط پنجم هر آزمون نیز خالی است.

موقعیت روی تخته با یکی از نویسه‌های "W" (مهره‌ی سفید)، "B" (مهره‌ی مشکی)، و "." (موقعیت خالی) نشان داده می‌شود. فرض کنید که تمامی موقعیت‌های ورودی در یک بازی واقعی با توجه به قوانین شرح داده شده امکان‌پذیر هستند و هرکدام از آزمون‌ها حداقل به یک نوبت برای به اتمام رسیدن نیاز دارند.

---

[۷۰] Tomas Flodén

خروجی (خروجی استاندارد)

برای هر آزمون، اگر بازیکن سفید برنده‌ی بازی می‌شود، عبارت "Sohrab" و اگر بازیکن مشکلی برنده می‌شود، عبارت "Sepehr" را چاپ کنید. در صورت تساوی بازی، خروجی "Draw" خواهد بود.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 BWWB WBWB .B.. .W.. WBWB BWWB .W.. .... BBWB W..W WW.. B...	Sohrab Sepehr Draw

مسأله‌ی ۳: قفل رمزدار

سهراب و سپهر یک قفل رمزدار پیدا کردند. قفل دارای شماره‌های چرخانی است که هرکدام از آنها n وجه مجاور دارند. اعداد صحیح 1 تا n به ترتیب روی هرکدام از شماره‌ها نوشته شده است. چرخاندن هر شماره در جهت عقربه‌های ساعت عدد آن شماره را افزایش می‌دهد. اعداد 1 و n نیز با هم مجاور هستند؛ بنابراین اگر شماره روی n باشد و آن را افزایش دهیم، شماره به عدد 1 تغییر خواهد کرد. سهراب و سپهر یک بازی با این قفل اختراع کرده‌اند و می‌خواهند آن را انجام دهند.

نخست، سهراب قفل را برداشته و شماره‌ها را می‌چرخاند تا آنها را روی یک عدد دلخواه تنظیم کند. سپهر n نوبت برای تغییر شماره‌ها دارد. در نوبت iام او می تواند یکی از شماره‌هایی که تاکنون انتخاب نکرده است را انتخاب کرده و آن را دقیقا i بار در جهت عقربه‌های ساعت بچرخاند.

سپهر برنده‌ی بازی می‌شود اگر بعد از n نوبتی که دارد، قفل رمزدار تمامی اعداد بین 1 تا n را نشان دهند (یعنی جایگشتی از این اعداد حاصل شود)؛ در غیر این صورت سهراب برنده‌ی بازی اعلام می‌شود.

به شما قفل رمزداری که سپهر در اختیار دارد، داده می‌شود. با فرض این که او بهینه بازی می‌کند، تعیین کنید که چه کسی برنده خواهد شد.

ورودی (ورودی استاندارد)

در خط اول ورودی عدد صحیح T $(T \le 100)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند. هر آزمون شامل یک عدد صحیح n $(n \le 13)$ است که تعداد شماره‌های روی قفل را تعیین می‌کند و پس از آن n عدد A_i $(1 \le A_i \le n)$ که با نویسه‌ی فاصله از هم جدا شده‌اند، می‌آید. این اعداد، ترکیب انتخابی سهراب را نشان می‌دهند.

خروجی (خروجی استاندارد)

برای هر آزمون، برنده‌ی بازی را با یکی از دو کلمه‌ی "Sohrab" یا "Sepehr" اعلام کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 2 1 2 3 2 1 3	Sohrab Sepehr

مسأله‌ی ٤: کاشان، سرزمین رویاها

سهراب و سپهر مسئول تاکسیرانی کاشان هستند که دارای n ایستگاه تاکسی و m تاکسی است. بین هر دو ایستگاه تاکسی دقیقا یک راه وجود دارد. هرکدام از تاکسی‌ها یکی راه‌های مورد علاقه‌ی خودشان را که توسط تاکسی دیگری اشغال نشده است، برای کار کردن انتخاب کرده‌اند.

سهراب و سپهر به سرعت دریافتند که برخی از ایستگاه‌ها به یکدیگر قابل دسترسی نیستند. حالا می‌خواهند این مشکل را حل کنند. بعد از مقداری مشاوره، این گزینه‌ها پیشنهاد شده است:

- مجبور کردن بعضی از تاکسی‌ها به تعویض راه از طریق پول دادن به آنها. هر تاکسی برای تعویض راه فعلی خود مقدار خاصی پول می‌گیرد.
- اضافه کردن چند تاکسی جدید که در راهی که سهراب و سپهر تعیین کرده‌اند، کار کنند. هزینه اضافه کردن یک تاکسی جدید برابر c است و آنها می‌توانند هر تعداد تاکسی که صلاح دیدند، به شهر اضافه کنند.

نگرانی اصلی سهراب و سپهر این است که حداقل هزینه‌ی ممکن برای حل این مشکل را پیدا کنند. همچنین در صورت امکان آنها ترجیح می‌دهند که این کار را با اضافه کردن حداقل تعداد ممکن تاکسی انجام دهند. حالا که شما از هدف آنها مطلع شده‌اید، وظیفه‌ی شما کمک کردن به آنها در حل سه مسأله است. اول، چند عدد تاکسی جدید باید اضافه شود؟ دوم، چند عدد از تاکسی‌های قدیمی باید مجبور به تعویض راه‌های خود شوند؟ و آخر، حداقل هزینه‌ی ممکن چقدر است؟

ورودی (ورودی استاندارد)

در خط اول ورودی عدد صحیح T $(T \le 35)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند. هر آزمون با یک خط حاوی سه عدد صحیح n، m و c $(2 \le n \le 10^4, 0 \le m \le 10^4, 1 \le c \le 10^5)$ آغاز می‌شود که به ترتیب نشان‌دهنده‌ی تعداد ایستگاه‌های درون شهر، تعداد تاکسی‌های فعلی و هزینه‌ی اضافه کردن تاکسی‌های جدید هستند.

سپس در m خط بعدی سه عدد صحیح a_i، b_i و r_i $(1 \le a_i, b_i \le n, a_i \ne b_i, 1 \le r_i \le 10^5)$ می‌آیند که مشخص می‌کند که تاکسی iام راه بین ایستگاه‌های a_i و b_i را انتخاب کرده است و برای تعویض راه r_i واحد پول می‌گیرد. ایستگاه‌ها از 1 تا n شماره‌گذاری شده‌اند و تضمین می‌شود که هیچ دو تاکسی در یک راه کار نمی‌کنند.

خروجی (خروجی استاندارد)

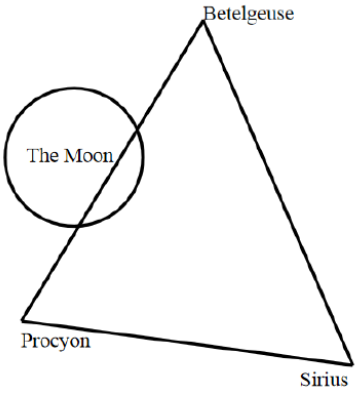
برای هر آزمون سه عدد صحیح که با نویسه‌ی فاصله از هم جدا شده‌اند، چاپ کنید. این سه عدد به ترتیب نشان‌دهنده‌ی تعداد تاکسی‌های جدید، تعداد تاکسی‌های قدیمی که باید راه خود را عوض کنند و هزینه‌ی نهایی خواهند بود.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 3 1 100 1 2 100 4 3 10 1 2 10 2 3 20 1 3 30 5 3 15 3 5 10 5 2 20 2 3 30	1 0 100 0 1 10 1 1 25

مسأله‌ی ۵: دوره‌ی مثلث زمستانی

سهراب و سپهر تصمیم گرفتند که مطالعات خود را در رشته‌ی نجوم ادامه دهند. آنها اخیرا به مثلث زمستانی علاقمند شده‌اند. مثلث زمستانی از ستاره‌های شباهنگ، شعرای شامی و اِبط الجوزا تشکیل شده است که سه تا از ده پرنورترین اشیایی قابل رویت از زمین هستند. آنها می‌خواهند مثلث زمستانی را در صحراهای اطراف کاشان رصد کنند و می‌خواهند متوجه شوند که آیا ماه به نقطه‌ی اشتراکی با مثلث زمستانی رسیده است یا خیر.



ما فضا را به صورت یک صفحه‌ی دوبعدی مدل می‌کنیم. ستاره‌ها نقطه‌هایی در این صفحه هستند که مختصات آنها را در اختیار داریم. همچنین ماه را به صورت یک دایره مدل می‌کنیم.

وظیفه‌ی شما این است که بررسی کنید که آیا اشتراکی بین ستاره‌ی زمستانی و ماه وجود دارد یا خیر (حداقل یک نقطه).

ورودی (ورودی استاندارد)

در خط اول ورودی عدد صحیح T $(T \le 1000)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند. در هر آزمون، سه خط نخست حاوی یک جفت از اعداد صحیح هستند که مختصات رئوس مثلث را مشخص می‌کنند.
در خط بعد سه عدد صحیح X ، Y و R می‌آید. نقطه‌ی (X,Y) مرکز و R شعاع ماه را مشخص می‌کند.
تمامی مقادیر مختصات در ورودی کمتر یا مساوی یک مقدار مطلق $(1 \le R \le 1000)$ هستند. همچنین تضمین می‌شود که هیچ سه راسی از مثلث‌های داده شده روی یک خط مستقیم قرار نمی‌گیرند.

خروجی (خروجی استاندارد)

برای هر آزمون یک خط چاپ کنید. اگر حداقل یک نقطه‌ی اشتراک بین ماه و مثلث زمستانی وجود دارد، کلمه‌ی "YES" و در غیر این صورت کلمه‌ی "NO" را چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 1 1 3 1 2 3 2 0 2 1 1 3 1 2 3 2 0 1 1 1 3 1 2 3 0 0 1	YES YES NO

مسأله‌ی ٦: بیابید مسأله را!

سهراب و سپهر طراحان سوال پنجمین مسابقه‌ی برنامه‌نویسی کاشان هستند. پس از آماده‌سازی مسائل، حامی مالی این رویداد به آنها دو مسأله‌ی جدید می‌دهد و از آنها می‌خواهد که آنها را به مجموعه‌ی مسائل اضافه کنند. حال شما باید یکی از آن مسائل را حل کنید. سهراب و سپهر اصلا وقت آماده‌سازی این مسائل را ندارند. به همین دلیل مسأله‌ی اصلی مطرح شده توسط حامی مالی را توضیح می‌دهند و شما باید کشف کنید که این مسأله به دنبال چه چیزی است، و سپس آن را حل کنید!

Machine	Part	Code
R1	P1	C1
R1	P2	C2
R1	P3	C3
R1	P4	C4
R2	P1	C5
R2	P5	C6
R2	P6	C7
R2	P7	C8
R3	P5	C9

Fig1. Mysterious table

	P1	P2	P3	P4	P5	P6	P7
R1	C1	C2	C3	C4			
R2	C5				C6	C7	C8
R3					C9		

Fig2. Strange table

در این مسأله به شما یک جدول مرموز داده شده و شما باید آنها را به یک جدول عجیب تبدیل کنید! به مثال داده شده توجه کنید تا متوجه شوید که چگونه باید این کار را انجام داد.

ورودی (ورودی استاندارد)

در خط اول ورودی یک عدد صحیح T $(T \leq 50)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند.

خط اول هر آزمون حاوی یک عدد صحیح N $(1 \leq N \leq 50)$ است که تعداد سطرهای جدول مرموز را مشخص می‌کند. در N خط بعدی سه عدد $Machine_i$، $Part_i$ و $Code_i$ می‌آید که اعداد ذخیره شده در سطر iام جدول مرموز را مشخص می‌کنند. هر داده در جدول فقط شامل حروف انگلیسی بزرگ و ارقام است. طول همه‌ی کلمات کمتر یا مساوی ۳ نویسه است. تضمین می‌شود که در جدول مرموز داده شده، تمامی جفت‌های $(Machine_i, Part_i)$ متمایز هستند.

خروجی (خروجی استاندارد)

برای هر آزمون، یک جدول عجیب مشابه مثال داده در نمونه‌ی ورودی و خروجی چاپ کنید. دو نکته سری وجود دارد که باید به آنها توجه کنید:

- تمامی فیلدهای داده‌ای باید دقیقا از 3 نویسه تشکیل شده و از چپ چیده شده باشند. اگر طول هرکدام از داده‌های فیلدها کمتر از 3 است، شما باید نویسه‌های فاصله‌ی مورد نیاز را به آن بیفزایید.
- نام سطرها و ستون‌ها باید به ترتیب صعودی حروف الفبا ظاهر شود.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 9 R1 P1 C1 R1 P2 C2 R1 P3 C3 R1 P4 C4 R2 P1 C5 R2 P5 C6 R2 P6 C7 R2 P7 C8 R3 P5 C9 1 R1 P1 CCC	+---+---+---+---+---+---+---+---+ | |P1 |P2 |P3 |P4 |P5 |P6 |P7 | +---+---+---+---+---+---+---+---+ |R1 |C1 |C2 |C3 |C4 | | | | +---+---+---+---+---+---+---+---+ |R2 |C5 | | | |C6 |C7 |C8 | +---+---+---+---+---+---+---+---+ |R3 | | | | |C9 | | | +---+---+---+---+---+---+---+---+ +---+---+ | |P1 | +---+---+ |R1 |CCC| +---+---+

راهنمایی

ترتیب الفبایی رشته‌ها همان ترتیبی است که در واژه‌نامه از آن استفاده می‌کنیم. این مقایسه در تمامی زبان‌های برنامه‌نویسی مدرن برای مقایسه‌ی رشته ها استفاده می‌شود. به طور رسمی، رشته‌ی p به طول n طبق ترتیب حروف الفبا از رشته‌ی q با طول m کوچکتر است، اگر یکی از این جملات صحیح باشد:

- $n < m$ و p پیشوند رشته‌ی q باشد (به عنوان مثال، "aba" از رشته‌ی "abaa" کوچکتر است).
- $p_1 = q_1 ، p_2 = q_2 ، ...., p_k < q_k, p_{k-1} = q_{k-1}$ به طوری که $1 \le k \le \min(n, m)$. در اینجا شماره‌گزاری نویسه‌های دو از نویسه‌ی آغازین شروع می‌شود.

مسأله‌ی ۷: ژنرال سهراب و ژنرال سپهر!

سهراب و سپهر ژنرال‌های ارتش کاشان هستند. در تمرین‌های روزانه، سربازان در یک خط ایستاده و از چپ به راست از 1 تا n شماره‌گزاری می‌شوند. علاوه بر این، هرکدام از آنها یا رو به راست یا رو به چپ ایستاده است.

سهراب و سپهر به سربازان دستورات تمرینی می‌دهند. سهراب فقط به دستور «بنشین» یا «بایست» علاقه دارد. سپهر هم فقط به دستور «بچرخ» علاقه دارد. توجه کنید که هرکدام از این دستورات به یک سرباز خاص داده می‌شود. به این ترتیب پس از اجرای چند دستور تمرینی، ممکن است تعدادی از سربازان ایستاده تعدادی دیگر نشسته باشند. همچنین برخی از آنها ممکن است رو به جهتی متفاوت از جهت اولیه‌ی خود باشند.

به دو سرباز «جفت سرباز قوی» یا SSP گفته می‌شود، اگر این سه شرط را رعایت کنند:

- هر دو سرباز ایستاده باشند.
- سرباز چپ رو به راست و سرباز راست رو به چپ باشد.
- سرباز دیگری بین آنها ایستاده نباشد.

گاهی اوقات، سهراب و سپهر می‌خواهند بدانند که چند «جفت سرباز قوی» وجود دارد به طوری که هر دو سرباز بین L و R قرار داشته باشند.

شما باید برنامه‌ای بنویسید که به سه نوع درخواست رسیدگی کند:

۱. سهراب به سرباز شماره‌ی X فرمان می‌دهد (که در نتیجه بین حالت نشسته و ایستاده تغییر حالت می‌دهد).
۲. سپهر به سرباز شماره‌ی X فرمان می‌دهد (که در نتیجه بین حالت رو به چپ و رو به راست تغییر حالت می‌دهد).
۳. اعلام این که چند «جفت سرباز قوی» بین L و R (شامل L و R) قرار دارد.

ورودی (ورودی استاندارد)

در خط اول ورودی عدد صحیح T $(T \le 15)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند.

در خط اول هر آزمون دو عدد صحیح n و q $(2 \le n \le 10^5, 1 \le q \le 10^5)$ می‌آید که به ترتیب مشخص‌کننده‌ی تعداد سربازان و تعداد درخواست‌ها هستند.

در خط بعد رشته‌ی S به طول n می‌آید که در آن هر نویسه یا '>' یا '<' است که نشان‌دهنده‌ی جهت‌گیری اول هرکدام از سربازان است ('>' برای چپ و '<' برای راست).

خط iام از q خط بعدی حاوی یک عدد صحیح T_i $(1 \le T_i \le 3)$ است که نوع درخواست iام را تعیین می‌کند. اگر درخواست iام از نوع 1 یا 2 باشد، پس از آن یک عدد صحیح X_i $(1 \le X_i \le n)$ می‌آید. اگر درخواست iام از نوع 3 باشد، پس از آن دو عدد صحیح L_i و R_i $(1 \le L_i \le n-1, L_i+1 \le R_i \le n)$ می‌آید. اعداد در خطوط توسط یک نویسه‌ی فاصله از هم جدا می‌شوند.

خروجی (خروجی استاندارد)

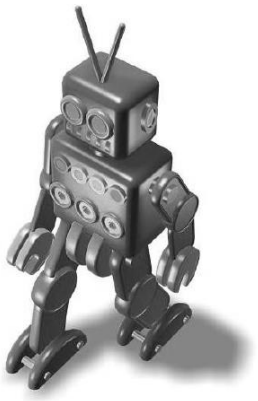
برای هر درخواست از نوع 3، یک خط چاپ کنید که در آن تعداد «جفت سربازهای قوی» که در بازه‌ی مشخص شده هستند، نشان داده شده است. به درخواست‌ها به ترتیبی که در ورودی آمده‌اند، پاسخ دهید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 6 10 ><><>< 3 1 6 1 2 3 1 6 2 3 3 1 6 3 2 6 3 2 5 1 2 3 1 6 3 3 4 4 4 >><< 3 1 4 1 2 1 3 3 1 4	3 2 2 1 0 2 0 1 1

مسأله‌ی ۸

سهراب و سپهر یک ربات جدید خریده‌اند که برنامه‌ی ویژه‌ای برای میوه خوردن دارد. برای میوه دادن به ربات، صاحبش باید چند میوه را در یک ردیف روی زمین قرار دهد. میوه‌ها از چپ به راست از 1 شماره‌گذاری می‌شوند. علاوه بر این صاحب ربات باید یک شمارنده‌ی دیجیتالی خاص را انتخاب کرده و در محفظه ای پشت سر ربات وارد کند. این شمارنده فقط اعداد 0 تا $M-1$ را می‌پذیرد و در صورت سرریز، مجددا از 0 آغاز می‌کند (مثلا اگر $M=8$ و شمارنده عدد 6 را نشان دهد، پس از 3 بار افزایش شمارنده به عدد 1 خواهد رسید). علاوه بر این، صاحب ربات باید چهار عدد صحیح L، R، X و Y را قبل از اجرای برنامه‌ی میوه دادن مشخص کند.



دقت کنید که به دلایل مسائل فنی L و R باید مضربی از M باشند. در دستورالعمل ربات الگوریتمی هست که نشان می‌دهد برنامه‌ی میوه دادن چگونه کار می‌کند. در این الگوریتم برخی از میوه‌ها «ممنوعه» فرض می‌شوند. یک میوه «ممنوعه» است اگر حداقل یکی از شروط زیر را داشته باشد:

۱. یکی از L میوه‌ی اول باشد.
۲. یکی از R میوه‌ی آخر باشد.
۳. عدد شمارنده‌ی دیجیتال وقتی ربات کنار میوه است، کمتر از X باشد.
۴. عدد شمارنده‌ی دیجیتال وقتی ربات کنار میوه است، بیشتر از Y باشد.

با توجه به دستورالعمل ربات، الگوریتم میوه خوردن به این ترتیب کار می‌کند:

۱. اعداد L، R، X و Y را از صاحبت بگیر.
۲. منتظر باش تا صاحبت یک شمارنده‌ی خاص را روی سرت وارد کند.
۳. مقدار شمارنده‌ی خاص را 0 کن.
۴. کنار میوه‌ی شماره‌ی 1 بایست.
۵. اگر میوه‌ای که کنارش هستی، «ممنوعه» نیست، آن را بخور.
۶. اگر یک میوه‌ی نارس خورده‌ای، سریعا خود را خاموش کن!
۷. قدم بعدی را ببین. اگر بعد از قدم بعدی میوه‌ای کنارت هست، به خط ۸ برو. وگرنه به خط ۱۱ برو.
۸. یک گام به جلو برو.
۹. مقدار شمارنده‌ی خاص را یک واحد زیاد کن.
۱۰. به خط ۵ برو.
۱۱. مقدار شمارنده‌ی خاص را یک واحد زیاد کن.
۱۲. اگر شمارنده‌ی خاص مقدار 0 را نشان می‌دهد، به خط ۱۳ برو، در غیر این صورت فورا خودت را خاموش کن.
۱۳. ماموریت با موفقیت انجام شد.

سهراب و سپهر دستورالعمل را خواندند. با این حال، هیچ چیز از آن متوجه نشدند! متاسفانه، ربات آنها گرسنه است! آنها تمام میوه‌های درون خانه را می‌آورند و آنها را روی زمین قرار می‌دهند. بعضی از میوه‌ها نارس و بقیه رسیده هستند.

قبل از اجرای برنامه‌ی میوه خرودن ربات، آنها باید یک شمارنده‌ی دیجیتالی به اندازه‌ی M انتخاب کنند و چهار عدد L، R، X و Y را برای ربات تعیین کنند. آنها می‌خواهند بدانند بین مقادیر ممکن برای M، L، R، X و Y حداکثر تعداد میوه‌ای که ربات بدون خاموش شدن می‌تواند بخورد، چقدر است. تنها شخص ماهری که آنها می‌شناسند که می‌تواند به آنها کمک کند، شما هستید. به شما تعداد میوه‌ها و اندیس میوه‌های نارس داده شده است. برنامه‌ای بنویسید که حداکثر تعداد میوه‌هایی را که ربات می‌تواند با توجه به الگوریتم میوه خوردن بخورد، پیدا کند.

ورودی (ورودی استاندارد)

در خط اول ورودی عدد صحیح T $(T \le 50)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند.

هر آزمون شامل دو خط است. در خط اول هر آزمون دو عدد صحیح n $(n \le 20000)$ و m $(0 \le m \le n)$ می‌آید که مشخص‌کننده‌ی تعداد میوه‌ها و تعداد میوه‌های نارس هستند. در خط دوم m عدد صحیح A_i $(1 \le i \le m, 1 \le A_i \le n)$ می‌آید که توسط نویسه‌ی فاصله از هم جدا شده‌اند. این اعداد اندیس میوه‌های نارس هستند. اعداد درون A به ترتیب صعودی هستند.

خروجی (خروجی استاندارد)

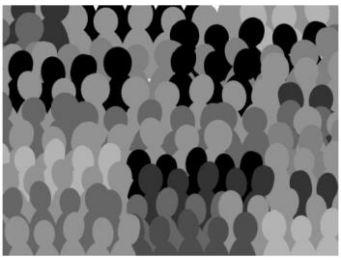
برای هر آزمون حداکثر تعداد میوه‌هایی را که ربات می‌تواند با تنظیمات درست بخورد، چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 5 1 3 6 2 1 4 12 3 1 8 9	2 4 6

مسأله‌ی ۹: انتخاب تیم‌های آی-سی-پی-سی

سهراب و سپهر داوران مسابقه‌ی اِی-سی-ام-آی-سی-پی-سی در دانشگاه کاشان هستند. امسال، مدیر سایت تهران به آنها اجازه‌ی اعزام تنها یک تیم به مسابقات منطقه‌ای داده است. این یک موقعیت پیچیده است؛ زیرا شرکت‌کنندگان بسیار خوبی در دانشگاه هستند و تنها سه تا از آنها شانس بودن در تیم نهایی را خواهند داشت.



سهراب و سپهر سیاست جالبی برای انتخاب اعضای تیم نهایی پیش گرفته‌اند. در ابتدا آنها اعلام می‌کنند که هر داوطلب باید یک حساب کاربری در سایت تاپکُدِر داشته باشد. تاپکُدِر سایتی است که مسابقات برنامه‌نویسی را به صورت منظم برقرار می‌کند. پس هر عضو تاپکُدِر یک رده‌بندی منحصر به فرد دارد (برای اعضایی که نمرات برابری دارند، قوانینی برای رده‌بندی وجود دارد). مشخص است که اگر یک دانش‌آموز رده‌بندی بهتری داشته باشد، احتمالا مسابقه‌دهنده‌ی بهتری برای آی–سی–پی–سی است.

علاوه بر این، سهراب و سپهر نباید تازه‌کاران را از قلم بیندازند؛ زیرا آنها برای شرکت در مسابقات منطقه‌ای انگیزه‌ی بسیار زیادی دارند و به همین دلیل زیاد تمرین می‌کنند.

بر اساس سیاست سهراب و سپهر، هر تیم احتمالی باید دارای سه عضو متفاوت مانند یک تیم فوتبال باشد! یک تازه‌کار، یک حرفه‌ای و یک با انرژی. در اینجا توضیح هرکدام از این سه نقش آمده است:

۱. عضو تازه‌کار: این عضو بدترین رده‌بندی را در تاپکُدِر دارد و بعد از دو عضو دیگر در تاپکُدِر ثبت نام کرده است.
۲. عضو حرفه‌ای: رده‌بندی او بین عضو تازه‌کار و عضو با انرژی است و قبل از دو عضو دیگر در تاپکُدِر ثبت نام کرده است.
۳. عضو با انرژی: او بهترین رده‌بندی را در تاپکُدِر دارد و ما بین عضو تازه‌کار و حرفه‌ای در تاپکُدِر ثبت نام کرده است.

سه دانش آموز می‌توانند با هم یک تیم تشکیل دهند، اگر تیم آنها هرکدام از سه عضو بالا را داشته باشد. تمامی تیم‌های ممکن احتمال برابری برای تشکیل شدن دارند.

به شما تعداد داوطلبان در دانشگاه کاشان، تاریخ ثبت نام و رده‌بندی تاپکُدِر آنها داده شده است. برای هر دانش‌آموز احتمال این را که او به عنوان یکی از اعضای تیم نهایی دانشگاه کاشان انتخاب شود، محاسبه کنید.

ورودی (ورودی استاندارد)

در خط اول عدد صحیح T $(T \le 30)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند.

در خط اول هر آزمون عدد صحیح N $(3 \le N \le 5000)$ می‌آید که تعداد دانش‌آموزان داوطلب را نشان می‌دهد. در خط $(i+1)$ام هر آزمون دو عدد D_i و R_i می‌آید که تاریخ ثبت نام و رده‌بندی دانش‌آموز iام را مشخص می‌کند. D_i به صورت $YYYY/MM/DD$ و یک تاریخ معتبر بین $1990/01/01$ تا $2012/12/31$ است و $1 \le R_i \le 100000$.

خروجی (خروجی استاندارد)

برای هر آزمون اگر سهراب و سپهر نمی‌توانند هیچ تیمی را با توجه به قوانین تشکیل دهند، عبارت "Rules Should Be Changed!" را در یک خط چاپ کنید. در غیر این صورت، احتمال این که هر کدام از دانش‌آموزان در تیم کاشان انتخاب شوند را به ترتیب ظاهر شدن در ورودی چاپ کنید. تمامی اعداد باید تا پنج رقم اعشار گرد شوند. بین هر دو عدد یک نویسه‌ی فاصله چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 3 2010/03/07 1500 2010/02/25 2500 2012/08/02 3500 5 2007/08/05 4 2007/07/11 5 2005/11/01 3 2006/05/31 2 2004/10/10 1 3 2006/05/31 123 2009/12/14 234 2008/07/19 345	1.00000 1.00000 1.00000 0.50000 0.50000 1.00000 1.00000 0.00000 Rules Should Be Changed!

مسأله‌ی ۱۰: مسأله‌ی شوخی!

سهراب و سپهر شهرداران کاشان هستند. آنها ماموران تمامی سازمان‌های مردم نهاد در سرزمین رویاها را به یک جلسه‌ی خیلی مهم دعوت می‌کنند. در اتاق جلسه یک میز گرد وجود دارد که M صندلی با شماره‌های 0 تا $M-1$ به ترتیب ساعتگرد در آن وجود دارد (میز گرد است پس صندلی‌های $M-1$ و 0 مجاور هستند).

یک روز پیش از جلسه تمامی گروه‌ها باید صندلی ماموران خود را درخواست کنند. هر گروه یک قسمت از صندلی‌های مجاور را انتخاب می‌کند. برای این کار آنها دو شماره صندلی F_i و L_i را انتخاب می‌کنند؛ به این معنی که آنها تمامی صندلی‌های بین F_i و L_i (از جمله F_i و L_i) را به ترتیب ساعتگرد می‌خواهند.
سهراب و سپهر می‌دانند که چند عدد صندلی وجود دارد و کدام صندلی‌ها توسط کدام گروه انتخاب شده است. آنها می‌خواهند بدانند که حداکثر تعداد گروه‌هایی که می‌توانند در جلسه شرکت کنند، چند است.

ورودی (ورودی استاندارد)

در خط اول عدد صحیح T $(T \le 25)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند.
در خط اول هر آزمون دو عدد صحیح N و M می‌آید که به ترتیب تعداد گروه‌ها و تعداد صندلی‌ها را مشخص می‌کند $(1 \le N \le 10^5, 3 \le M \le 10^8)$.
در N خط بعد دو عدد صحیح F_i و L_i می‌آید که به ترتیب اولین و آخرین صندلی انتخاب شده توسط گروه iام را مشخص می‌کنند (که همان صندلی‌های از شماره‌ی F_i تا L_i به ترتیب ساعتگرد هستند). ضمنا $(1 \le F_i, L_i \le M-1)$.

خروجی (خروجی استاندارد)

برای هر آزمون یک عدد صحیح چاپ کنید که همان حداکثر تعداد گروه‌هایی است که می‌توان به جلسه دعوت نمود.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 5 10 8 0 9 1 2 4 3 6 5 7 6 10 0 1 1 2 2 3 3 5 5 8 8 0	3 3

مسأله‌ی ۱۱: چالش قاتل

سهراب دنباله‌ی A به طول n از اعداد صحیح در اختیار دارد و می‌خواهد که آن را به چندین زیردنباله‌ی متوالی تقسیم کند. فرض کنید R حاصل ضرب مجموع تمامی زیردنباله‌ها باشد. علاوه بر این، او یک عدد صحیح P در اختیار دارد که برابر است با $P = p_1 \times p_2 \times ... \times p_n$ (p_i ها اعداد اول متفاوتی هستند به طوری که i بین 1 تا n است).

او می‌خواهد مقدار $GCD(P,R)$ را بیشینه کند. علاوه بر این سهراب به جمع بیشتر از ضرب علاقمند است، می‌خواهد که آرایه‌ی اصلی را به حداقل تعداد زیردنباله‌ها تقسیم کند. شما باید به او در رسیدن به هدفش کمک کنید.

ورودی (ورودی استاندارد)

در خط اول عدد صحیح T ($T \leq 100$) می‌آید که تعداد آزمون‌ها را مشخص می‌کند.

هر آزمون شامل عدد صحیح n ($1 \leq n \leq 100$) که تعداد اعضای دنباله‌ی اصلی سهراب است، و عدد P ($2 \leq P \leq 10^6$) است. خط بعدی حاوی n عدد صحیح است که با نویسه‌ی فاصله از هم جدا شده‌اند ($1 \leq A_i \leq 10^6, 1 \leq i \leq n$).

خروجی (خروجی استاندارد)

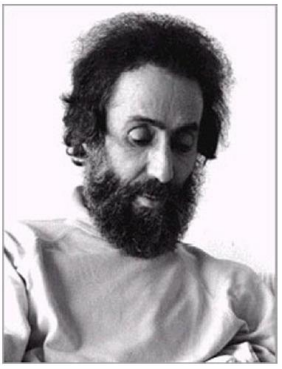
برای هر آزمون دو عدد صحیح که با یک نویسه‌ی فاصله از هم جدا شده‌اند، چاپ کنید. عدد اول حداکثر $GCD(P,R)$ است که با تقسیم کردن دنباله ممکن است به دست بیاید و عدد دوم حداقل تعداد برش‌های لازم برای رسیدن به آن مقدار GCD است.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 2 2 1 2 3 10 9 5 3 4 6 7 1 4 5	2 1 5 2 6 1

مسأله‌ی ۱۲: کوچکترین شعر الفبایی

سهراب سپهری (۱۵ مهر ۱۳۰۷ – ۱ اردیبهشت ۱۳۵۹) یک شاعر و نقاش مدرن مشهور است.



او متولد کاشان در استان اصفهان بود. او یکی از پنج مشهورترین شاعران ایرانی در نوشتن شعر نو است (گونه‌ای از شعر که اغلب قافیه و ردیف ندارد). از شاعران هم‌سبک او می‌توان به نیما یوشیج، احمد شاملو، مهدی اخوان ثالث و فروغ فرخزاد اشاره کرد.

شعر او سرشار از انسانیت و اهمیت به ارزشهای انسانی است. او عاشق طبیعت بود در اشعار خود به دفعات به آن اشاره می‌کند.

سهراب بعد از تمام ماجراهایی که در زندگی داشت، تبدیل به شاعری بزرگ شد. او حالا یک رمزنگار حرفه‌ای نیز شده است! او می‌خواهد یک بازی جذاب انجام دهد تا کمی سرگرم شود. اساس این بازی روش رمزهای جایگزین است.

رمز جایگزین با یک کلید جایگزینی تعریف می‌شود؛ به این ترتیب که به هرکدام از حروف الفبا یک حرف دیگر نسبت می‌دهد (این نسبت دادن به صورت یک نگاشت یک به یک بین حروف انگلیسی است).

کلید یک رشته‌ی ۲۶ حرفی است که حرف iام را به key_i نگاشت می‌کند.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧	↧
x	m	u	r	p	w	e	l	i	f	s	o	b	d	h	q	k	c	a	j	t	z	n	g	v	y

مثالی از یک کلید جایگزینی

به عنوان مثال، پس از رمزنگاری یک شعر با استفاده از کلید نشان داده شده در شکل، تمامی نویسه‌های 'a' با نویسه‌ی 'x' جایگزین می‌شود و به همین ترتیب تمامی نویسه‌های دیگر نیز جایگزین می‌شوند. به نتیجه‌ی این فرآیند «متن رمز شده» گفته می‌شود.

سهراب یکی از بهترین اشعار خود را انتخاب کرده و آن را با استفاده از رمز جایگزینی و یک کلید رمزنگاری می‌کند. او این فرآیند را با یک کلید مشترک X بار تکرار می‌کند (نتیجه‌ی هر مرحله از رمزنگاری به عنوان ورودی مرحله‌ی بعد استفاده می‌شود). او سپس نتیجه‌ی مرحله آخر رمزنگاری را به سپهر می‌دهد. برای این که کار رمزگشایی «متن رمز شده» برای سپهر دشوارتر شود، سهراب به او عدد X (تعداد دفعات رمزنگاری شعر) را نمی دهد. سهراب تنها کلید رمزنگاری و «متن رمز شده» را به سپهر می‌دهد.

چون سپهر به اشعار سهراب علاقمند است، می‌خواهد شعرهای رمزنگاری شده را رمزگشایی کند. سپهر می‌داند که می‌تواند اشعار رمزنگاری شده را رمزگشایی کند؛ چون می‌داند که روش رمزگزاری استفاده شده بسیار ضعیف است. به علت ندانستن مقدار X، او ممکن است متن‌های زیادی پیدا کند (شعرهای کاندیدا). سهراب یک راهنمایی هم به سهراب می‌کند: «میان همه‌ی اشعار کاندیدا، شعر من شعری است که از نظر الفبایی از همه کوچکتر است».

چون سپهر شما را به عنوان یک برنامه‌نویس خوب می‌شناسد، از شما می‌خواهد که شعر سهراب را بیابید.

ترتیب الفبایی رشته‌ها همان ترتیب معروف «واژه‌نامه‌ای» است. رشته‌ی p به طول n از نظر الفبایی از رشته‌ی q با همان طول کوچکتر است، اگر به ازای یک i $(1 \le i \le n)$، داشته باشیم:

$$p_1 = q_1, p_2 = q_2, \ldots, p_{k-1} < q_{k-1}, p_k < q_k$$

در رابطه‌ی بالا نویسه‌های رشته‌ها با شروع از 1 شماره‌گذاری شده‌اند. نویسه‌های رشته‌ها با ترتیب الفبایی با هم مقایسه می‌شوند.

ورودی (ورودی استاندارد)

در خط اول عدد صحیح T $(T \le 25)$ می‌آید که تعداد آزمون‌ها را مشخص می‌کند.

هر آزمون شامل سه خط می‌شود. در خط اول هر آزمون یک عدد صحیح می‌آید $(n \le 10^6)$.

خط دوم شامل رشته‌ی S است. رشته‌ی S طول n است و فقط شامل حروف انگلیسی کوچک بوده و شعر رمزنگاری شده را نشان می‌دهد.

در خط سوم، key که کلید رمزنگاری است در قالب ۲۶ حرف انگلیسی کوچک می‌آید. در رشته‌ی key هرکدام از حروف دقیقا یک بار ظاهر می‌شود.

خروجی (خروجی استاندارد)

برای هر آزمون شعر سهراب را در یک خط چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 7 pumiuau xmurpwelifsobdhqkcajtzngvy 29 yzqjxynzysyrmttbjaymyrgyisvno zcdefghijkamnopqrstulwxybv 25 bezmnftheuecthepegucagtz cbafedihglkjonmrqputsxwvzy	acmicpc ahlekashanamroozegarambadnist beyondtheseathereisacity

۷-۹-۳- رقابت برنامه‌نویسی دانشجویی، مرحله اینترنتی سال ۸۷ – دانشگاه آزاد اسلامی مشهد – هفت سوال

سوال ۱: هواشناسی

در یک آمارگیری دمای هوای برخی شهرهای ایران را در روزهای مختلف هفته داریم. می‌خواهیم گرم‌ترین شهر بین سردترین شهرهای تک‌تک روزهای هفته را مشخص کنیم.

ورودی (از پرونده‌ی weather.in)

تعداد مورد تست در خط اول قرار دارد ($0 < t \le 20$). در ادامه به تعداد این موارد، برای هر یک از هفت روز هفته، ابتدا نام روز (سه حرف اول نام روز هفته) و سپس در خط‌های بعدی نام شهر (*city*) و دمای آن (T) در آن روز خواهد آمد که ($-100 < T < 100$) و *city* حداکثر ۲۰ کاراکتر خواهد بود. نام روز هفته یکی از کلمات *Mon*، *Tue*، *Wed*، *Thu*، *Fri*، *Sat* و *Sun* خواهد بود. هر مجموعه داده‌های ورودی با یک خط که در آن کلمه‌ی *End* نوشته شده پایان می‌یابد. می‌توانید فرض کنید که هیچ نام شهری یکی از هشت کلمه (*Sun...Mon* و *End*) نیست.

خروجی (خروجی استاندارد)

برای هر مورد تست، در خروجی نام شهری که بین سردترین شهرهای روزهای هفته بیشترین دما را دارد، چاپ کرده و سپس روزی که شهر در آن سردترین هوا را داشته و بین سردترین شهرهای روزهای دیگر، بیشترین دما را دارد را با یک فاصله در خروجی چاپ نمایید. در صورتی که بیش از یک شهر این شرایط را دارد، شهری را که از نظر ترتیب حروف الفبا اول است را در خروجی بنویسید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
1 Mon Tehran 12 Ardabil -2 Ilam 21 Kerman 24 Tue Tabriz -5 Mashhad 5 Kerman 24 Wed Tehran 9 Mashhad 7 Khalkhal -9 Kerman 25 Thu Tehran 12 Mashhad 14 Fri Shiraz 18 Isfahan 15 Tehran 10 Sat Mashhad 12 Kerman 16 Shiraz 17 Sun Tehran 16 Tabriz 3 End	Mashhad Sat

سوال ۲: پیام کوتاه

برای نوشتن یک متن وسط دستگاه تلفن همراه، حداقل چند بار فشردن کلید نیاز است؟ فرض کنیم کلیدهای تلفن همراه در هنگام نوشتن متن به ترتیب زیر باشند:

.,?!1	abc2	def3
ghi4	jkl5	mno6
pqrs7	tuv8	wxyz9
	[Space]0[NewLine]	[Change Case]

به این معنی که با فشردن یک بار کلید ۵ حرف j ، دوبار حرف k، سه بار حرف l و چهار بار رقم ۵ نوشته خواهد شد. با فشردن یک بار کلید ۰ فاصله، دو بار رقم ۰ نوشته می‌شود و سه بار فشردن کلید باعث رفتن به ابتدای خط بعدی خواهد شد. اگر تعداد بیشتری این کلیدها فشرده شوند، این ترتیب تکرار خواهد شد. با فشردن کلید [Change Case] از آن به بعد بزرگ یا کوچک بودن تغییر می‌کند. یعنی اگر در حال نوشتن با حروف کوچک هستیم، با فشردن این کلید، از آن به بعد، حروف بزرگ تایپ می‌شوند و بالعکس. در ابتدای کار نوشتن با حروف کوچک آغاز می‌شود.

ورودی (از پرونده‌ی shortmessage.in)

تعداد موارد تست در ورودی داده نمی‌شود ($0 < t \le 20$). در ورودی به تعداد این موارد ابتدا عبارت Case #X: نوشته می‌شود که X شماره‌ی آن مورد است و با ۱ شروع می‌شود. متن پیام کوتاه نوشته شده و تا جایی که مورد تست بعدی شروع شود و یا موارد تست تمام شود، ادامه دارد. هر پیام کوتاه فقط از حروف بزرگ و کوچک، اعداد، فاصله، خط جدید و نشانه‌های !?,. تشکیل شده است و حداکثر از ۱۰ خط مجزا تشکیل شده که در هر خط ماکزیمم ۸۰ کاراکتر وجود دارد.

خروجی (خروجی استاندارد)

برای هر مورد تست تعداد حداقل کلیدهای فشرده شده‌ی مورد نیاز در یک خط چاپ شود.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
Case #1: This a single line Test! Case #2: thIs Is A MulTy LiNe TeSt 0123 !	59 99

سوال ۳: تست

در یک آزمون تستی، برای هر جواب درست A نمره و برای هر جواب نادرست B نمره در نظر گرفته شده می‌شود. مثلا اگر A و B به ترتیب ۳ و ۱- باشند و تعداد کل سوال‌ها ۴ باشد، ممکن است کسی به دو سوال جواب درست و به یک سوال جواب نادرست بدهد و یک سوال را بدون جواب بگذارد. نمره کل او ۵ خواهد شد. اما در یک آزمون ۴ سوالی، هیچ حالتی نیست که نمره یک نفر ده شود.

بدیهی است که با K سوال، نمره کل بین $B \times K$ و $A \times K$ قرار دارد. اما مثال بالا نشان می‌دهد که همه اعداد صحیح بین این دو نمی‌توانند نمره کل باشند. برنامه‌ای بنویسید که تعداد حالت‌های ممکن برای نمره در یک آزمون

K سوالی با A و B مشخص را محاسبه کند. مثلا بـرای $K=2$، $A=3$ و $B=-1$، نمـره کـل ۶ حالـت ممکن دارد که عبارتند از:

-2 -1 0 2 3 6

ورودی (از پرونده‌ی tests.in)

در فایل ورودی ابتدا تعداد نمونه‌های ورودی قرار دارد ($0<t\le 20$) و در t سطر بعدی مقادیر مختلـف K و A و B قرار دارد که $K\le 100$ و $0<A\le 10$ و $-10\le B\le 0$.

خروجی (خروجی استاندارد)

خروجی باید شامل t سطر باشد که هر سطر تعداد حالت‌های ممکن برای نمره کل در آزمون تستی متناظر را نشان می‌دهد.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 2 3 -1 3 3 -1	6 9

سوال ٤: خون‌گرمی

در سایت‌هایی مانند اورکات، یک شبکه n نفری از افراد وجود دارد که هر کدام با تعدادی از افـراد دیگـر در شـبکه دوست هستند. فرض کنید که هر نفر با یک شماره ۱ تا n شناسایی می‌شود و یک ماتریس $n\times n$ رابطه دوستی بین افراد را نشان می‌دهد. این ماتریس یک ماتریس متقارن است که قطر اصلی آن صفر است و هر عنصر a_{ij} که فرد شماره i با فرد شماره j دوست باشد مقدار ۱ و در غیر این صورت مقدار ۰ دارد. هر فردی را کـه حـداقل یـک دوست دارد و ضمنا تعداد دوستانش بیشتر از یا مساوی با تعداد دوستان هر یک از دوستانش باشد، خونگرم می‌نامیم. از شما خواسته می‌شود که برنامه‌ای برای تعیین افراد خونگرم بنویسید.

ورودی (از پرونده‌ی gregariousness.in)

در فایل ورودی ابتدا تعداد نمونه‌های ورودی t قرار دارد $(1\le t\le 20)$. به دنبال آن به ازای هر یـک از t نمونـه ورودی ابتدا تعداد افراد شبکه یعنی n قرار دارد $(1\le n\le 100)$ و سپس یک ماتریس $n\times n$ که رابطه دوستی بین افراد را نشان می‌دهد. این ماتریس دقیقا n سطر دارد که هرکدام از n کاراکتر ۰ یا ۱ تشکیل شده است.

خروجی (خروجی استاندارد)

خروجی باید شامل t سطر باشد که در هر سطر شماره همه افراد خونگرم در شبکه ورودی نظیر آن به ترتیب صعودی مشخص شده باشد. بین هر دو مقدار متوالی در یک سطر یک فضای خالی باید قرار داشته باشد. فرض می‌شود که در هر نمونه ورودی حداقل یک فرد خونگرم وجود دارد.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 3 0 1 1 1 0 0 1 0 0 2 0 1 1 0	1 1 2

سوال ۵: جدول صفر و یک

جدولی $N \times N$ داریم که هر کدام از خانه‌های آن با یکی از دو مقدار صفر یا یک پر شده است. در هر مرحله می‌توانیم یک ستون یا یک سطر را انتخاب کنیم و تمام اعضای آن ستون یا سطر را معکوس کنیم. یعنی صفرها را به یک و یک‌ها را به صفر تبدیل کنیم. می‌خواهیم در کمترین تعداد حرکت، تمام اعضای جدول را به صفر تبدیل کنیم. برنامه‌ای بنویسید که با گرفتن جدول اولیه، تعداد کمترین حرکت لازم را اعلام کند، یا اعلام کند این کار ممکن نیست.

ورودی (از پرونده‌ی zeroone.in)

تعداد موارد تست در خط اول قرار دارد $(0 < t \leq 60)$. سطر اول هر مجموعه داده شامل N که $N \leq 200$ است. هر کدام از N سطر بعدی شامل اعضای سطر N _ام جدول است. اعداد هر سطر با کاراکتر فاصله از همدیگر جدا شده‌اند.

خروجی (خروجی استاندارد)

برای هر مجموعه داده، در سطری مجزا، در صورت امکان تعداد کمترین تعداد حرکت لازم را چاپ کنید و در غیر این صورت کلمه Impossible را چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 4 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 5 1 1 1 1 0 1 1 0 1 0 1 1 0 1 0 1 1 1 1 0 1 1 1 0 1	4 Impossible

سوال ٦: دنباله‌ها

دنباله اعداد $a_1...a_n$ را متناوب می‌نامیم، اگر به ازای nهای زوج داشته باشیم $a_i > a_{i-1}$ و به ازای i_های فرد (به جز یک) داشته باشیم $a_i < a_{i-1}$.

دنباله‌ی $b_1..b_m$ را زیر دنباله‌ی دنباله‌ی $a_1...a_n$ می‌نامیم، اگر آن را بتوان با حذف صفر یا چند عضو (نه لزوما مجاور) دنباله‌ی a بدست بیاوریم.

ورودی (از پرونده‌ی series1.in)

تعداد موارد تست در خط اول قرار دارد $(0 < t \le 60)$. سطر اول هر مجموعه داده شامل n که $(1 \le n \le 500)$ طول دنباله اعداد است. سطر دوم شامل n عدد $(-10^9 \le a_i \le 10^9)$ است.

خروجی (خروجی استاندارد)

برای هر مجموعه داده، طول طولانی‌ترین زیر دنباله متناوب آن را چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 10 1 2 1 2 1 2 1 2 1 2 10 3 4 -3 4 5 -5 5 6 -5 55	10 8

سوال ۷: جعبه‌های مهره و حسن کچل ۲

حسن کچل و دوست خوبش سنک حچل، بازی زیر را با هم انجام می‌دهند. در ابتدای بازی تعدادی جعبه که هر کدام شامل تعدادی مهره هستند وجود دارند. بازی را حسن کچل شروع می‌کند و به نوبت حرکات را انجام می‌دهند. در هر نوبت، جعبه‌ای شامل m مهره از روی میز برداشته می‌شود و به تعدادی جعبه جدید با مهره‌های نامساوی تقسیم می‌شود و دوباره روی میز گذاشته می‌شود. مثلا، اگر جعبه ۷ مهره داشته باشد، می‌توانیم آن را به صورت (۶،۱)، (۵،۲)، (۴،۳) و (۴،۲،۱) تقسیم کنیم. کسی که نتواند هیچ حرکتی را انجام دهد، به عنوان بازنده بازی اعلام می‌شود.

برنامه‌ای بنویسید که اعلام کند که اگر هر دو بازیکن به صورت بهینه بازی کنند، آیا حسن کچل می‌تواند برنده بازی شود یا نه.

ورودی (از پرونده‌ی game2.in)

تعداد موارد تست در خط اول قرار دارد $(0 < t \le 60)$. سطر اول هر مجموعه شامل تعداد جعبه‌ها در آغاز بازی است. سطر بعدی شامل n عدد است که عدد i_ام نشانگر تعداد مهره‌ها در جعبه i_ام است $(1 \le n \le 50, 0 \le a_i \le 300)$.

خروجی (خروجی استاندارد)

برای هر مجموعه داده، در سطری مجزا، در صورتی که حسن کچل بتواند بازی را ببرد عبارت YES و در غیر این صورت عبارت NO را چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
4 2 1 2 3 3 1 4 4 3 3 3 3 1 0	NO YES NO NO

۷-۹-٤- مسابقات هفتگی سایت CodeForces- پنج سوال

Problem A: K-Periodic Array

This task will exclusively concentrate only on the arrays where all elements equal 1 and/or 2.

Array a is k -period if its length is divisible by k and there is such array b of length k, that a is represented by array b written exactly $\frac{n}{k}$ times consecutively. In other words, array a is k -periodic, if it has period of length k.

For example, any array is n-periodic, where n is the array length. Array [2, 1, 2, 1, 2, 1] is at the same time 2-periodic and 6-periodic and array [1, 2, 1, 1, 2, 1, 1, 2, 1] is at the same time 3-periodic and 9-periodic.

For the given array a, consisting only of numbers one and two, find the minimum number of elements to change to make the array k -periodic. If the array already is k -periodic, then the required value equals 0.

Input (Standard input)

The first line of the input contains a pair of integers $n, k\,(1 \le k \le n \le 100)$, where n is the length of the array and the value n is divisible by k. The second line contains the sequence of elements of the given array $a_1, a_2, ..., a_n\,(1 \le a_i \le 2)$, a_i is the i^{th} element of the array.

Output (Standard output)

Print the minimum number of array elements we need to change to make the array k -periodic. If the array already is k -periodic, then print 0.

Sample test(s)

Sample Input	Sample Output
6 2 2 1 2 2 2 1	1

Sample Input	Sample Output
8 4 1 1 2 1 1 1 2 1	0

Sample Input	Sample Output
9 3 2 1 1 1 2 1 1 1 2	3

Note

In the first sample it is enough to change the fourth element from 2 to 1, then the array changes to [2, 1, 2, 1, 2, 1].

In the second sample, the given array already is 4-periodic.

In the third sample it is enough to replace each occurrence of number two by number one. In this case the array will look as [1, 1, 1, 1, 1, 1, 1, 1, 1] — this array is simultaneously 1-, 3- and 9-periodic.

Problem B: Fox Dividing Cheese

Two little greedy bears have found two pieces of cheese in the forest of weight a and b grams, correspondingly. The bears are so greedy that they are ready to fight for the larger piece. That's where the fox comes in and starts the dialog: "Little bears, wait a little, I want to make your pieces equal" "Come off it fox, how are you going to do that?", the curious bears asked. "It's easy", said the fox. "If the mass of a certain piece is divisible by two, then I can eat exactly a half of the piece. If the mass of a certain piece is divisible by three, then I can eat exactly two-thirds, and if the mass is divisible by five, then I can eat four-fifths. I'll eat a little here and there and make the pieces equal".

The little bears realize that the fox's proposal contains a catch. But at the same time they realize that they can not make the two pieces equal themselves. So they agreed to her proposal, but on one condition: the fox should make the pieces equal as quickly as possible. Find the minimum number of operations the fox needs to make pieces equal.

Input (Standard input)

The first line contains two space-separated integers a and $b (1 \le a, b \le 10^9)$.

Output (Standard output)

If the fox is lying to the little bears and it is impossible to make the pieces equal, print -1. Otherwise, print the required minimum number of operations. If the pieces of the cheese are initially equal, the required number is 0.

Sample test(s)

Sample Input	Sample Output
15 20	3

Sample Input	Sample Output
14 18	-1

Sample Input	Sample Output
6 6	0

Problem C: Hamburgers

Polycarpus loves hamburgers very much. He especially adores the hamburgers he makes with his own hands. Polycarpus thinks that there are only three decent ingredients to make hamburgers from: a bread, sausage and cheese. He writes down the recipe of his favorite "Le Hamburger de Polycarpus" as a string of letters 'B' (bread), 'S' (sausage) and 'C' (cheese). The ingredients in the recipe go from bottom to top, for example, recipe "BSCBS" represents the hamburger where the ingredients go from bottom to top as bread, sausage, cheese, bread and sausage again.

Polycarpus has n_b pieces of bread, n_s pieces of sausage and n_c pieces of cheese in the kitchen. Besides, the shop nearby has all three ingredients, the prices are p_b rubles for a piece of bread, p_s for a piece of sausage and p_c for a piece of cheese.

Polycarpus has r rubles and he is ready to shop on them. What maximum number of hamburgers can he cook? You can assume that Polycarpus cannot break or slice any of the pieces of bread, sausage or cheese. Besides, the shop has an unlimited number of pieces of each ingredient.

Input (Standard input)

The first line of the input contains a non-empty string that describes the recipe of "Le Hamburger de Polycarpus". The length of the string doesn't exceed 100, the string contains only letters 'B' (uppercase English B), 'S' (uppercase English S) and 'C' (uppercase English C).

The second line contains three integers $n_b, n_s, n_c \, (1 \le n_b n_s, n_c \le 100)$ — the number of the pieces of bread, sausage and cheese on Polycarpus' kitchen. The third line contains three integers $p_b, p_s, p_c \, (1 \le p_b p_s, p_c \le 100)$ — the price of one piece of bread, sausage and cheese in the shop. Finally, the fourth line contains integer $r \, (1 \le r \le 10^{12})$ — the number of rubles Polycarpus has.

Please, do not write the %lld specifier to read or write 64-bit integers in C++. It is preferred to use the cin, cout streams or the %I64dspecifier.

Output (Standard output)

Print the maximum number of hamburgers Polycarpus can make. If he can't make any hamburger, print 0.

Sample test(s)

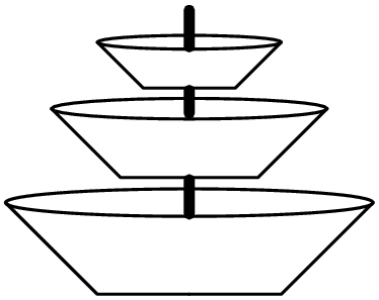
Sample Input	Sample Output
BBBSSC 6 4 1 1 2 3 4	2

Sample Input	Sample Output
BBC 1 10 1 1 10 1 21	7

Sample Input	Sample Output
BSC 1 1 1 1 1 3 1000000000000	200000000001

Problem D: Vessels

There is a system of n vessels arranged one above the other as shown in the figure below. Assume that the vessels are numbered from 1 to n, in the order from the highest to the lowest, the volume of the i^{th} vessel is a_i liters.



Initially, all the vessels are empty. In some vessels water is poured. All the water that overflows from the i^{th} vessel goes to the $(i+1)^{th}$ one. The liquid that overflows from the n^{th} vessel spills on the floor.

Your task is to simulate pouring water into the vessels. To do this, you will need to handle two types of queries:

- Add x_i liters of water to the p_i^{th} vessel;
- Print the number of liters of water in the k_i^{th} vessel.

When you reply to the second request you can assume that all the water poured up to this point, has already overflown between the vessels.

Input (Standard input)

The first line contains integer n — the number of vessels $(1 \le n \le 2.10^5)$. The second line contains n integers $a_1, a_2, ..., a_n$ — the vessels' capacities $(1 \le a_i \le 10^9)$. The vessels' capacities do not necessarily increase from the top vessels to the bottom ones (see the second sample). The third line contains integer m — the number of queries $(1 \le m \le 2.10^5)$. Each of the next m lines contains the description of one query. The query of the first type is

represented as "$1\ p_i\ x_i$", the query of the second type is represented as "$2\ k_i$" $(1 \le p_i \le n, 1 \le x_i \le 10^9, 1 \le k_i \le n)$.

Output (Standard output)

For each query, print on a single line the number of liters of water in the corresponding vessel.

Sample test(s)

Sample Input	Sample Output
2 5 10 6 1 1 4 2 1 1 2 5 1 1 4 2 1 2 2	4 5 8

Sample Input	Sample Output
3 5 10 8 6 1 1 12 2 2 1 1 6 1 3 2 2 2 2 3	7 10 5

Problem E: Subway Innovation

Berland is going through tough times — the dirt price has dropped and that is a blow to the country's economy. Everybody knows that Berland is the top world dirt exporter!

The President of Berland was forced to leave only K of the currently existing n subway stations.

The subway stations are located on a straight line one after another, the trains consecutively visit the stations as they move. You can assume that the stations are on the Ox axis, the i^{th} station is at point with coordinate x_i. In such case the distance between stations i and j is calculated by a simple formula $|x_i - x_j|$.

Currently, the Ministry of Transport is choosing which stations to close and which ones to leave. Obviously, the residents of the capital won't be too enthusiastic about the innovation, so it was decided to show the best side to the people. The Ministry of Transport wants to choose such k stations that minimize the average commute time in the subway!

Assuming that the train speed is constant (it is a fixed value), the average commute time in the subway is calculated as the sum of pairwise distances between stations, divided by the number of pairs (that is $\frac{n.(n-1)}{2}$) and divided by the speed of the train.

Help the Minister of Transport to solve this difficult problem. Write a program that, given the location of the stations selects such k stations that the average commute time in the subway is minimized.

Input (Standard input)

The first line of the input contains integer $n(3 \le n \le 3.10^5)$ — the number of the stations before the innovation. The second line contains the coordinates of the stations $x_1, x_2, ..., x_n(-10^8 \le x_i \le 10^8)$. The third line contains integer $k(2 \le k \le n-1)$ — the number of stations after the innovation.

The station coordinates are distinct and not necessarily sorted.

Output (Standard output)

Print a sequence of k distinct integers $t_1, t_2, ..., t_k(1 \le t_j \le n)$ — the numbers of the stations that should be left after the innovation in arbitrary order. Assume that the stations are numbered 1 through n in the order they are given in the input. The number of stations you print must have the minimum possible average commute time among all possible ways to choose k stations. If there are multiple such ways, you are allowed to print any of them.

Sample test(s)

Sample Input	Sample Output
3 1 100 101 2	2 3

Note

In the sample testcase the optimal answer is to destroy the first station (with $x = 1$). The average commute time will be equal to 1 in this way.

مسأله‌ی ۱: آرایه‌ی k-متناوب

این مسأله منحصرا روی آرایه‌هایی تمرکز می‌کند که عناصر درون آنها برابر 1 یـا 2 اسـت. آرایـه‌ی a k-متنـاوب است اگر طول آن بخش‌پذیر بر k بوده و آرایه‌ی b به طول k نیز وجود داشته باشد، به طوری کـه a بـا کنـار

هم قرار دادن b به تعداد دقیقا $\frac{n}{k}$ بار به وجود بیاید. به عبارت دیگر، a k–متناوب است، اگر دوره‌ی تناوب آن برابر k باشد.

به عنوان مثال، اگر طول یک آرایه را با n نشان دهیم، هر آرایه‌ای n–متناوب است. آرایه‌ی [2, 1, 2, 1, 2, 1] همزمان 2–متناوب و 6–متناوب است و آرایه‌ی [1, 2, 1, 1, 2, 1, 1, 2, 1] به طور همزمان 3–متناوب و 9–متناوب است.

برای یک آرایه‌ی دلخواه a، که تنها شامل اعداد 1 و 2 است، حداقل تعداد عناصری را که با تغییر دادن آنها آرایه k–متناوب می‌شود، پیدا کنید. اگر آرایه بدون هیچ تغییری k–متناوب است، جواب مسأله 0 است.

ورودی (ورودی استاندارد)

خط اول ورودی حاوی دو عدد صحیح n و k $(1 \le k \le n \le 100)$ است، که n طول آرایه بوده و بر k بخش‌پذیر است. خط دوم شامل n عدد $a_1, a_2, ..., a_n$ $(1 \le a_i \le 2)$ است که همان دنباله‌ی عناصر آرایه‌ی مورد نظر است (a_i عنصر iام آرایه است).

خروجی (خروجی استاندارد)

حداقل تعداد از عناصری که با تغییر آنها آرایه k–متناوب می‌شود را چاپ کنید. اگر آرایه بدون هیچ تغییری k–متناوب است، 0 را چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
6 2 2 1 2 2 2 1	1

Sample Input	Sample Output
8 4 1 1 2 1 1 1 2 1	0

Sample Input	Sample Output
9 3 2 1 1 1 2 1 1 1 2	3

نکته

در مثال اول کافی است که عنصر چهارم را از 2 به 1 تغییر دهیم، بنابراین آرایه تبدیل به [2, 1, 2, 1, 2, 1] می شود.

در مثال دوم، آرایه‌ی داده شده 4–متناوب است.

در مثال سوم، کافی است تمامی اعداد 2 را با عدد 1 جایگزین کنیم. در این حال آرایه تبدیل به [1, 1, 1, 1, 1, 1, 1, 1] می‌شود (که به طور همزمان 1–، 3– و 9–متناوب است).

مسأله‌ی ۲: تقسیم کردن پنیر توسط روباه

دو خرس کوچک حریص در جنگل دو تکه پنیر با وزن‌های a و b گرم پیدا کرده‌اند. خرس‌ها آنقدر حریص هستند که حاضرند برای به دست آوردن تکه‌ی سنگین‌تر مبارزه کنند. اینجاست که روباه وارد ماجرا می‌شود و به آنها می‌گوید: «خرس‌های کوچولو، یه لحظه صبر کنین! من می‌خوام وزن تکه‌ها رو یکی کنم». خرس‌های کنجکاو پرسیدند: «بیخیال روباه! چطوری می‌خوای این کارو بکنی؟». روباه گفت: «خیلی آسونه، اگر وزن یکی از تیکه‌ها بر ۲ بخش‌پذیر باشه، من می‌تونم دقیقا نصف اون رو بخورم. اگر وزن یکی از تیکه‌ها بر ۳ بخش‌پذیر باشه، من می‌تونم دو سومش رو بخورم. اگر هم وزن بر ۵ بخش‌پذیر باشه، من می‌تونم چهار پنجمش رو بخورم. همینجوری یکمی از این تیکه و یکمی از اون تیکه می‌خورم تا وزنشون یکی بشه».

خرس‌های کوچک متوجه شدند که پیشنهاد روباه هم چندان بد نیست. در عین حال متوجه شدند که آنها نمی‌توانند خودشان وزن تکه‌ها را برابر کنند. پس با تصمیم او موافقت کردند، ولی با یک شرط: روباه باید در سریعترین زمان ممکن وزن‌ها را برابر کند. حداقل تعداد عملیاتی که روباه برای یکسان‌سازی وزن‌ها نیاز دارد را پیدا کنید.

ورودی (ورودی استاندارد)

خط اول ورودی شامل دو عدد صحیح a و b $(1 \le a, b \le 10^9)$ است که با نویسه‌ی فاصله از هم جدا شده‌اند.

خروجی (خروجی استاندارد)

اگر روباه به خرس‌ها دروغ گفته است و امکان یکسان‌سازی وزن تکه‌ها وجود ندارد، 1- را چاپ کنید. در غیر این صورت، حداقل تعداد عملیات‌های مورد نیاز را چاپ کنید. اگر تکه‌های پنیر از همان ابتدا وزن یکسانی دارند، جواب مسأله 0 است.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
15 20	3

Sample Input	Sample Output
14 18	-1

Sample Input	Sample Output
6 6	0

مسأله‌ی ۳: همبرگر

پولیکارپوس عاشق همبرگر است. او به ویژه همبرگرهایی که دستپخت خودش هستند را دوست دارد. او فکر می‌کند که تنها سه ماده‌ی غذایی برای درست کردن همبرگر کافی است: یک نان، سوسیس و پنیر. او دستور پخت «همبرگر پولیکارپوس» که مورد علاقه‌اش است را به صورت رشته‌ای از حروف 'B' (نان)، 'S' (سوسیس) و 'C' (پنیر) می‌نویسد. مواد درون دستور پخت از بالا به پایین نوشته می‌شوند. به عنوان مثال، دستور "BSCBS" به معنی همبرگری است که مواد تشکیل‌دهنده‌ی آن از بالا به پایین نان، سوسیس، پنیر، نان و سوسیس هستند.

پولیکارپوس n_b تکه نان، n_s تکه سوسیس، و n_c تکه پنیر در آشپزخانه دارد. علاوه بر این، مغازه‌ای در آن نزدیکی است که هر سه مواد غذایی را دارد. قیمت یک تکه نان p_b روبل، قیمت یک تکه سوسیس p_s روبل و قیمت یک تکه پنیر p_c است.

پولیکارپوس r روبل دارد و می‌تواند با آنها خرید کند. او حداکثر چند همبرگر می‌تواند بپزد؟ فرض کنید که پولیکارپوس نمی‌تواند تکه‌های نان، سوسیس و پنیر را ببرد. علاوه بر این، مغازه بی‌نهایت عدد از تکه‌های هرکدام از مواد غذایی را دارد.

ورودی (ورودی استاندارد)

خط اول ورودی حاول یک رشته‌ی غیرتهی است که دستورالعمل «همبرگر پولیکارپوس» را توضیح می‌دهد. طول این رشته از 100 تجاوز نمی‌کند. رشته تنها شامل حروف 'B' (حرف انگلیسی B بزرگ)، 'S' (حرف انگلیسی S بزرگ) و 'C' (حرف انگلیسی C بزرگ) است.

خط دوم شامل سه عدد صحیح n_b، n_s و n_c $(1 \le n_b n_s, n_c \le 100)$ است که به ترتیب تعداد تکه‌های نان، سوسیس و پنیر در آشپزخانه‌ی پولیکارپوس است. در خط سوم سه عدد صحیح p_b، p_s و p_c $(1 \le p_b p_s, p_c \le 100)$ می‌آید که به ترتیب قیمت یک تکه‌ی نان، سوسیس و پنیر در مغازه است. در خط چهارم هم عدد صحیح r $(1 \le r \le 10^{12})$ می‌آید که مقدار پول‌های پولیکارپوس بر حسب روبل است.

لطفا از %lld برای خواندن یا نوشتن اعداد صحیح ۶۴ بیتی در سی‌پلاس‌پلاس استفاده نکنید. بهتر است که از جریان‌های cin، cout یا %I64d استفاده کنید.

خروجی (خروجی استاندارد)

حداکثر تعداد همبرگرهایی که پولیکارپوس می‌تواند بپزد را چاپ کنید. اگر او نمی‌تواند هیچ همبرگری بپزد، 0 را چاپ کنید.

نمونه‌ی ورودی و خروجی

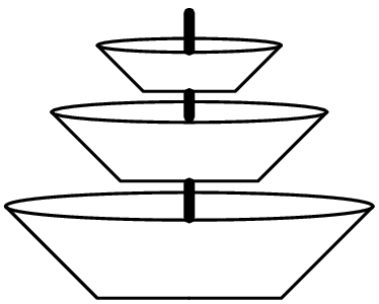
Sample Input	Sample Output
BBBSSC 6 4 1 1 2 3 4	2

Sample Input	Sample Output
BBC 1 10 1 1 10 1 21	7

Sample Input	Sample Output
BSC 1 1 1 1 1 3 1000000000000	200000000001

مسأله‌ی ٤: ظروف

سامانه‌ای از n ظرف وجود دارد که مانند شکل نشان داده روی یکدیگر قرار گرفته‌اند. فرض کنید که ظروف از بالاترین تا پایین‌ترینشان از 1 تا n شماره‌گذاری شده‌اند و حجم ظرف iام برابر a_i لیتر است.



در ابتدا تمامی ظروف خالی هستند. در بعضی از ظرف‌ها آب ریخته می‌شود. تمامی آبی که از ظرف iام سرریز می کند، درون ظرف $(i+1)$ام ریخته می‌شود. آبی که از ظرف nام سرریز می‌کند، روی زمین می‌ریزد.
وظیفه‌ی شما این است که ریختن آب در ظروف را شبیه‌سازی کنید. برای این کار، باید دو نوع درخواست را مدیریت کنید:

- x_i لیتر آب درون ظرف p_i بریزید.
- حجم آب درون ظرف k_i را چاپ کنید.

زمانی که شما درخواست دوم را پردازش می‌کنید، می‌توانید فرض کنید که تمامی آب ریخته شده تاکنون، سرریزهای لازم را درون ظروف انجام داده است.

ورودی (ورودی استاندارد)

در خط اول عدد صحیح n می‌آید که تعداد ظروف را مشخص می‌کند $(1 \le n \le 2.10^5)$. در خط دوم n عدد صحیح a_1، a_2،...، a_n می‌آید که حجم ظروف را مشخص می‌کنند $(1 \le a_i \le 10^9)$. حجم ظروف لزوما از بالا به پایین افزایش نمی‌یابد (به نمونه‌ی ورودی دوم توجه کنید). خط سوم شامل عدد صحیح m است که همان تعداد درخواست‌ها است $(1 \le m \le 2.10^5)$. هرکدام از m خط بعدی شرح یکی از درخواست‌ها را ارائه می‌دهد. درخواست نوع اول به صورت "$1\ p_i\ x_i$" و درخواست نوع دوم به صورت "$2\ k_i$" نشان داده می‌شود $(1 \le p_i \le n, 1 \le x_i \le 10^9, 1 \le k_i \le n)$.

خروجی (خروجی استاندارد)

برای هر درخواست، یک خط چاپ کنید که حاوی مقدار آب درون ظرف مورد نظر بر حسب لیتر است.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
2 5 10 6 1 1 4 2 1 1 2 5 1 1 4 2 1 2 2	4 5 8

Sample Input	Sample Output
3 5 10 8 6 1 1 12 2 2 1 1 6 1 3 2 2 2 2 3	7 10 5

مسأله‌ی ۵: ابتکار مترو

بِرلَند روزهای سختی را پشت سر می‌گذارد. قیمت خاک به شدت کاهش یافته است و این می‌تواند اقتصاد کشور را نابود کند. هرکسی می‌داند که بِرلَند بزرگترین صادرکننده‌ی خاک است.

رییس‌جمهور بِرلَند مجبور شده بود که تنها K ایستگاه مترو از n ایستگاه فعلی را باز بگذارد.

ایستگاه‌های مترو روی یک خط صاف یکی پس از دیگری قرار گرفته‌اند. قطارها در حین حرکت به ترتیب از ایستگاه‌ها عبور می‌کنند. فرض کنید که ایستگاه‌ها روی محور x قرار گرفته‌اند و ایستگاه iام در مختصات x_i قرار دارد. به این ترتیب فاصله‌ی بین دو ایستگاه i و j با رابطه‌ی ساده‌ی $|x_i - x_j|$ قابل محاسبه است.

در حال حاضر وزارت راه می‌خواهد تصمیم بگیرد که کدام ایستگاه را ببندد و کدام را باز بگذارد. واضح است که ساکنین پایتخت چندان از این ابتکار خوشحال نیستند؛ به همین دلیل قرار شد که حین انجام این کار صلاح مردم در نظر گرفته شود. وزارت راه می‌خواهد k ایستگاه را به گونه‌ای انتخاب کند که حداقل زمان رفت و آمد در مترو را کمینه کند.

با فرض این که سرعت یک قطار ثابت است (مقدار آن ثابت می‌ماند)، زمان رفت و آمد متوسط در مترو با جمع فاصله‌ی بین تمامی جفت ایستگاه‌ها و تقسیم آن به تعداد جفت‌ها (که برابر است با $\frac{n.(n-1)}{2}$) و سپس تقسیم آن بر سرعت قطار محاسبه می‌شود.

به وزارت راه کمک کنید که این مسأله‌ی دشوار را حل کند. برنامه‌ای بنویسید که با گرفتن مکان ایستگاه‌ها، k ایستگاه را به گونه‌ای انتخاب کند که زمان رفت و آمد متوسط مترو حداقل شود.

ورودی (ورودی استاندارد)

در خط اول ورودی عدد صحیح n $(3 \le n \le 3.10^5)$ می‌آید که تعداد ایستگاه‌ها قبل از این ابتکار است. در خط دوم، مختصات ایستگاه‌ها یعنی x_1، x_2،...، x_n $(-10^8 \le x_i \le 10^8)$ می‌آید. در خط سوم عدد صحیح k $(2 \le k \le n-1)$ می‌آید که همان تعداد ایستگاه‌ها بعد از این ابتکار است.

مختصات هر ایستگاه منحصر به فرد است ولی ایستگاه‌ها لزوما به ترتیب در ورودی ظاهر نمی‌شوند.

خروجی (خروجی استاندارد)

دنباله‌ای از k عدد صحیح متمایز t_1، t_2،...، t_n $(1 \le t_j \le n)$ به ترتیب دلخواه چاپ کنید که شماره‌ی ایستگاه‌هایی هستند که باید بعد از ابتکار باید باقی بمانند. فرض کنید که ایستگاه‌ها از 1 تا n به ترتیب ظاهر شدن در ورودی شماره‌گذاری می‌شوند. شماره‌ی ایستگاه‌هایی که شما چاپ می‌کنید باید حداقل زمان رفت و آمد متوسط ممکن بین تمامی راه‌های موجود برای انتخاب k ایستگاه را داشته باشند. اگر چندین راه برای این کار وجود دارد، شما مجاز هستید که هرکدام از آنها را که می‌خواهید، چاپ کنید.

نمونه‌ی ورودی و خروجی

Sample Input	Sample Output
3 1 100 101 2	2 3

نکته

در نمونه‌ی نشان داده شده، جواب بهینه این است که ایستگاه نخست (با $x=1$) را نابود کنیم. در این حالت زمان رفت و آمد متوسط برابر 1 خواهد شد.

۷-۱۰- راه‌حل‌های سوالات مسابقات

در این بخش، راه‌حل سوالات بخش قبل ارائه گشته است. به دلیل حجیم بودن سرآیندهای بعضی سوالات، در ابتدا مجموعه‌ی کامل سرآیندهای استفاده شده (در همه راه‌حل‌ها) آورده شده و در ادامه از آوردن مجدد سرآیندها خودداری شده است.

مجموعه‌ی سرآیندها

```
#include <iostream>
#include <algorithm>
#include <memory.h>
#include <cstdlib>
#include <sstream>
#include <iomanip>
#include <cstdio>
#include <string>
#include <vector>
#include <queue>
#include <ctime>
#include <cmath>
#include <deque>
#include <stack>
#include <list>
#include <map>
#include <set>
using namespace std;
```

۷-۱۰-۱- مسابقات ملی سال ۹۱ – دانشگاه کاشان

راه‌حل سوال A

```
#define int64 long long
#define Max 1000000
int a, b, c;
int cnt1[110], cnt2[110];
string str;
```

```
int getId(char x) {
    if (x >= 'a' && x <= 'z')
        return x - 'a';
    return x - 'A' + 26;
}
int main() {
    int T;
    for (cin >> T; T--; ) {
        cin >> a >> b >> c;
        memset(cnt1, 0, sizeof cnt1);
        memset(cnt2, 0, sizeof cnt2);
        getline(cin, str);
        getline(cin, str);
        for (int i = 0; i < str.length(); i++)
            cnt1[getId(str[i])]++;
        getline(cin, str);
        for (int i = 0; i < str.length(); i++)
            cnt2[getId(str[i])]++;
        int res = 0;
        for (int i = 0; i < 52; i++) {
            int t = min(cnt1[i], cnt2[i]);
            cnt1[i] -= t;
            cnt2[i] -= t;
        }
        for (int i = 0; i < 26; i++) {
            cnt1[i] += cnt1[i + 26];
            cnt2[i] += cnt2[i + 26];
            int t = min(cnt1[i], cnt2[i]);
            cnt1[i] -= t;
            cnt2[i] -= t;
            res += t * a;
        }
        int tot1 = 0, tot2 = 0;
        for (int i = 0; i < 26; i++) {
            tot1 += cnt1[i];
            tot2 += cnt2[i];
        }
        int t = min(tot1, tot2);
        tot1 -= t;
        tot2 -= t;
        res += t * b + tot2 * c;
        cout << res << endl;
    }
    return 0;
}
```

راه‌حل سوال B

```
int n;
int mat[60][60];
string str;
```

```
void act(char ch) {
    if (ch == 'U') {
        for (int j = 1; j <= n; j++)
            mat[n + 1][j] = mat[1][j];
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                mat[i][j] = mat[i + 1][j];
    }
    if (ch == 'D') {
        for (int j = 1; j <= n; j++)
            mat[0][j] = mat[n][j];
        for (int i = n; i >= 1; i--)
            for (int j = 1; j <= n; j++)
                mat[i][j] = mat[i - 1][j];
    }
    if (ch == 'B') {
        for (int i = 1; i <= n; i++)
            mat[i][n + 1] = mat[i][1];
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                mat[i][j] = mat[i][j + 1];
    }
    if (ch == 'F') {
        for (int i = 1; i <= n; i++)
            mat[i][0] = mat[i][n];
        for (int i = 1; i <= n; i++)
            for (int j = n; j >= 1; j--)
                mat[i][j] = mat[i][j - 1];
    }
    if (ch == 'T' || ch == 'R') {
        for (int i = 1; i <= n; i++)
            for (int j = i + 1; j <= n; j++)
                swap(mat[i][j], mat[j][i]);
    }
    if (ch == 'R') {
        for (int i = 1; i <= n; i++)
            for (int j = 1; j < n - j + 1; j++)
                swap(mat[i][j], mat[i][n - j + 1]);
    }
}
int main() {
    int T;
    for (cin >> T; T--; ) {
        cin >> n;
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                cin >> mat[i][j];
        cin >> str;
        for (int i = 0; i < str.length(); i++)
            act(str[i]);
        for (int i = 1; i <= n; i++) {
```

```
            for (int j = 1; j <= n; j++) {
                if (j > 1) cout << ' ';
                cout << mat[i][j];
            }
            cout << endl;
        }
        cout << endl;
    }
    return 0;
}
```

راه‌حل سوال C

```
#define int64 long long
int n;
int64 a0, S;
int64 gcd(int64 a, int64 b) {
    return (b == 0 ? a : gcd(b, a % b));
}
int main() {
    while (cin >> n, n) {
        S = 0;
        int64 t;
        for (int i = 0; i < n; i++) {
            cin >> t;
            S += t;
            if (i == 0)
                a0 = t;
        }
        int64 d = gcd(a0, S);
        cout << "1/" << n << ' ' << a0 / d << '/' << S / d << endl;
    }
    return 0;
}
```

راه‌حل سوال D

```
#define int64 long long
#define Inf 2000000000LL
int n, m, l;
int len[1010];
char ch[1010];
int64 k;
int64 dp[1010][11];
string str[1010];
int64 mem(int l, int rem) {
    if (l < 0 || rem < 0)
        return 0;
    if (dp[l][rem] != -1)
        return dp[l][rem];
```

```
    int64 &ref = dp[l][rem];
    ref = 1;
    for (int i = 0; i < n; i++)
        ref += mem(l - len[i], rem - 1);
    ref = min(ref, Inf);
    return ref;
}
int main() {
    int T;
    for (scanf("%d", &T) > 0; T--; ) {
        scanf("%d%d%d%d", &n, &m, &l, &k);
        for (int i = 0; i < n; i++) {
            scanf("%s", ch);
            str[i] = ch;
        }
        sort(str, str + n);
        for (int i = 0; i < n; i++)
            len[i] = str[i].length();
        memset(dp, -1, sizeof dp);
        int64 tot = mem(l, m) - 1;
        if (tot < k) {
            cout << "empty" << endl;
            continue;
        }
        string res = "";
        while (k) {
            for (int i = 0; i < n; i++) {
                int64 t = mem(l - res.size() - len[i], m - 1);
                if (t < k)
                    k -= t;
                else {
                    res += str[i];
                    m--, k--;
                    break;
                }
            }
        }
        cout << res << endl;
    }
    return 0;
}
```

راه‌حل سوال E

```
#define pdd pair< double, double >
#define EPS 1e-7
#define PI (2*acos(0.))
bool les(double a, double b) {
    return a + EPS < b;
}
bool lesQ(double a, double b) {
```

```
    return a - EPS < b;
}
bool greQ(double a, double b) {
    return a + EPS > b;
}
double dist(double x1, double y1, double x2, double y2) {
    double dx = x1 - x2;
    double dy = y1 - y2;
    return sqrt(dx * dx + dy * dy);
}
pdd getRange(double x, double y, double r) {
    double center = atan2(y, x);
    double len = dist(0, 0, x, y);
    double delta = asin(r / len);
    return pdd(center - delta, center + delta);
}
int intersection(double l1, double r1, double l2, double r2) {
    if (greQ(max(l1, l2), min(r1, r2)))   return 0;
    if (lesQ(l1, l2) && greQ(r1, r2))  return 2;
    return 1;
}
int main() {
    int t;
    for (cin >> t; t--; ) {
        int x[2], y[2], r[2];
        pdd range[2];
        for (int i = 0; i < 2; i++) {
            cin >> x[i] >> y[i] >> r[i];
            range[i] = getRange(x[i], y[i], r[i]);
        }
        int type = 0;
        for (int i = -2; i <= 2; i++) {
            int tmp = intersection(range[0].first + i * 2 * PI,
range[0].second + i * 2 * PI, range[1].first, range[1].second);
            if (tmp != 0) {
                type = tmp;
                break;
            }
        }
        if (les(dist(0, 0, x[0], y[0]), dist(0, 0, x[1], y[1])) && type
!= 0) {
            if (type == 1)    cout << "Annular eclipse" << endl;
            else          cout << "Total eclipse" << endl;
        }
        else {
            cout << "None" << endl;
        }
    }
    return 0;
}
```

راه‌حل سوال F

```
int leftTurn(P p1, P p2, P p3) {
    long long dx1 = p2.first - p1.first, dy1 = p2.second - p1.second;
    long long dx2 = p3.first - p2.first, dy2 = p3.second - p2.second;
    long long cr = (dx1 * dy2 - dx2 * dy1);
    if (cr > 0)
        cr = 1;
    else if (cr < 0)
        cr = -1;
    return cr;
}
P seg[MAXN];
int n, T, Y, q;
int main() {
    for (cin >> T; T--; ) {
        cin >> n >> Y;
        for (int i = 0; i < n; i++)
            cin >> seg[i].first >> seg[i].second;
        sort(seg, seg + n);
        cin >> q;
        for (int i = 0; i < q; i++) {
            int x, y;
            cin >> x >> y;
            int l = 0, r = n - 1, res = -1;
            while (l <= r) {
                int mid = (l + r) / 2;
                int cr = leftTurn(P(seg[mid].first, 0), P(seg[mid].second,
Y), P(x, y));
                if (cr == 1)
                    r = mid - 1;
                else
                    res = mid, l = mid + 1;
            }
            if (res != -1 && leftTurn(P(seg[res].first, 0),
P(seg[res].second, Y), P(x, y)) == 0)
                cout << "On the " << res + 1 << "segment" << endl;
            else cout << res + 1 << endl;
        }
        cout << endl;
    }
    return 0;
}
```

راه‌حل سوال G

```
int n;
int sm, sw, pr;
int tm, tw;
```

```
int m[110], w[110];
int main() {
    while (cin >> n, n) {
        sm = sw = pr = 0;
        tm = tw = -1;
        int a, b;
        for (int i = 0; i < n; i++)
            cin >> m[i];
        for (int i = 0; i < n; i++)
            cin >> w[i];
        for (int i = 0; i < n; i++) {
            sm += m[i], sw += w[i];
            pr += min(m[i], w[i]);
            if (min(m[i], w[i]) > 0)
                tm = m[i], tw = w[i];
        }
        if (pr > 1 || sm == 0 || sw == 0)
            cout << 1 << endl;
        else if (tm == -1)
            cout << 2 << endl;
        else if ((tm == sm || tm > 1) && (tw == sw || tw > 1))
            cout << 1 << endl;
        else cout << 2 << endl;
    }
    return 0;
}
```

راه‌حل سوال H

```
#define MAXN 1000008
#define P pair< long long, long long >
bool pr[MAXN];
vector< int > fact[MAXN];
P arr[MAXN];
int T, n;
long long power[MAXN];
long long mod = 1000000007;
long long modP(long long a, long long b, long long p) {
    if (b == 0)
        return 1;
    long long t = modP(a, b / 2, p);
    t = (t * t) % p;
    return (t * ((b % 2 == 0) ? 1 : a)) % p;
}
int main() {
    pr[0] = pr[1] = true;
    for (int i = 2; i < MAXN; i++)
        if (!pr[i]) {
            fact[i].push_back(i);
            for (int j = 2 * i; j < MAXN; j += i)
                pr[j] = true, fact[j].push_back(i);
```

```
        }
    for (cin >> T; T--; ) {
        long long res = 1;
        memset(power, 0, sizeof power);
        cin >> n;
        for (int i = 0; i < n; i++) {
            cin >> arr[i].first >> arr[i].second;
            for (int j = 0; j < fact[arr[i].first].size(); j++) {
                int f = fact[arr[i].first][j], cnt = 0;
                long long temp = arr[i].first;
                while (temp % f == 0)
                    temp /= f, cnt++;
                power[f] = max(power[f], cnt * arr[i].second);
            }
        }
        for (int i = 0; i < MAXN; i++)
            if (!pr[i]) {
                res *= modP(i, power[i], mod);
                res %= mod;
            }
        cout << res << endl;
    }
    return 0;
}
```

راه‌حل سوال I

```
#define Inf 2000000000
int n, sz;
int v[110], s[110];
int dp[110][51000];
int mem1(int idx, int val) {
    if (val == 0)
        return 0;
    if (idx < 0 || val < 0)
        return Inf;
    if (dp[idx][val] != -1)
        return dp[idx][val];
    return dp[idx][val] = min(mem1(idx - 1, val), mem1(idx - 1, val -
v[idx]) + s[idx]);
}
int mem2(int idx, int rem) {
    if (rem < 0)
        return -Inf;
    if (rem == 0 || idx == n)
        return 0;
    if (dp[idx][rem] != -1)
        return dp[idx][rem];
    return dp[idx][rem] = max(mem2(idx + 1, rem), mem2(idx + 1, rem -
s[idx]) + v[idx]);
}
```

```
int main() {
    int T = 0;
    while (cin >> n >> sz, n) {
        int mxv = 0;
        int sums = 0, sumv = 0;
        for (int i = 0; i < n; i++) {
            cin >> v[i];
            mxv = max(v[i], mxv);
            sumv += v[i];
        }
        for (int i = 0; i < n; i++) {
            cin >> s[i];
            sums += s[i];
        }
        if (sums <= sz) {
            cout << sumv << endl;
            continue;
        }
        memset(dp, -1, sizeof dp);
        int res = 0;
        if (mxv <= 500) {
            for (int i = 1; i <= mxv * n; i++)
                if (mem1(n - 1, i) <= sz)
                    res = i;
        }
        else res = mem2(0, sz);
        cout << res << endl;
    }
    return 0;
}
```

راه‌حل سوال J

```
#define MAXV 1010
#define MAXE 100010
#define Inf 1000000000
#define P pair<int, int>
int n, m;
int mn[MAXV];
bool mark[MAXV];
vector <P> adj[MAXV];
int main() {
    int T;
    for (scanf("%d", &T); T--; ) {
        scanf("%d%d", &n, &m);
        for (int i = 0; i < n; i++)
            mn[i] = Inf, mark[i] = false;
        for (int i = 0; i < n; i++)
            if (!adj[i].empty())
                adj[i].clear();
        int a, b, c;
```

```
        for (int i = 0; i < m; i++) {
            scanf("%d%d%d", &a, &b, &c);
            a--, b--;
            adj[a].push_back(P(b, c));
            adj[b].push_back(P(a, c));
        }
        int res = 0;
        mn[0] = 0;
        for (int i = 0; i < n; i++) {
            int idx = -1, tmp = Inf;
            for (int j = 0; j < n; j++)
                if (!mark[j] && tmp > mn[j])
                    idx = j, tmp = mn[j];
            if (idx == -1) {
                res = -1;
                break;
            }
            res += mn[idx];
            mark[idx] = true;
            for (int j = 0; j < adj[idx].size(); j++) {
                int k = adj[idx][j].first;
                if (!mark[k])
                    mn[k] = min(mn[k], adj[idx][j].second);
            }
        }
        if (res == -1)
            cout << "impossible" << endl;
        else cout << res << endl;
    }
    return 0;
}
```

۷-۱۰-۲- مسابقات ملی سال ۹۲ – دانشگاه کاشان

راه‌حل سوال A

```
int main() {
    int t, c[3], mn;
    cin >> t;
    while (t--) {
        cin >> c[0] >> c[1] >> c[2];
        sort(c, c + 3);
        if (c[0] + c[1] + c[2] == 1)
            cout << 1 << endl;
        else if (c[0] == c[1] && c[1] == c[2])
            cout << 6 << endl;
        else if (c[0] == c[1] && c[2] == c[1] + 1)
            cout << 2 << endl;
        else if (c[1] == c[0] + 1 && c[1] == c[2])
            cout << 2 << endl;
        else
```

```
            cout << 0 << endl;
    }
    return 0;
}
```

راه‌حل سوال B

```
#define bound res != 1
using namespace std;
struct node {
    char board[4][4];
    void print() {
        for (int i = 0;i<4;i++, cerr << endl)
            for (int j = 0;j<4;j++)
                cerr << board[i][j];
        cerr << endl;
    }
    node() {
        for (int i = 0;i < 4;i++)
            for (int j = 0;j < 4;j++)
                board[i][j] = '.';
    }
    void rotate(int b, int dir) {
        int x = (b / 2) * 2;
        int y = (b % 2) * 2;
        if (dir == 1) {
            char t = board[x][y];
            board[x][y] = board[x + 1][y];
            board[x + 1][y] = board[x + 1][y + 1];
            board[x + 1][y + 1] = board[x][y + 1];
            board[x][y + 1] = t;
        }
        else {
            char t = board[x][y];
            board[x][y] = board[x][y + 1];
            board[x][y + 1] = board[x + 1][y + 1];
            board[x + 1][y + 1] = board[x + 1][y];
            board[x + 1][y] = t;
        }
    }
    int get_val() {
        int res = 0;
        for (int i = 0;i < 4;i++)
            for (int j = 0;j < 4;j++)   {
                res *= 3;
                res += (board[i][j] == '.' ? 0 : (board[i][j] == 'B' ? 1 :
2));
            }
        return res;
    }
    bool is_winner(char c) {
```

```
        for (int i = 0; i < 4;i++) {
            if (board[i][1] == c && board[i][2] == c && (board[i][0] == c
|| board[i][3] == c))
                return true;
            if (board[1][i] == c && board[2][i] == c && (board[0][i] == c
|| board[3][i] == c))
                return true;
        }
        if (board[1][1] == c && board[2][2] == c && (board[0][0] == c ||
board[3][3] == c))
            return true;
        if (board[1][0] == c && board[2][1] == c && board[3][2] == c)
            return true;
        if (board[0][1] == c && board[1][2] == c && board[2][3] == c)
            return true;
        if (board[1][2] == c && board[2][1] == c && (board[3][0] == c ||
board[0][3] == c))
            return true;
        if (board[1][3] == c && board[2][2] == c && board[3][1] == c)
            return true;
        if (board[0][2] == c && board[1][1] == c && board[2][0] == c)
            return true;
        return false;
    }
    bool is_full() {
        for (int i = 0; i < 4; i++)
            for (int j = 0; j < 4; j++)
                if (board[i][j] == '.')
                    return false;
        return true;
    }
};
short mem[43046721][2]; // 3^16
int play(node state, bool turn) {// O = 0 and X = 1
    int v = state.get_val();
    if (mem[v][turn] != -1)
        return mem[v][turn];
    char c = (turn == 0 ? 'W' : 'B');
    char nc = (turn == 1 ? 'W' : 'B');
    bool I_have_3 = state.is_winner(c);
    bool He_has_3 = state.is_winner(nc);
    if (I_have_3 && He_has_3)
        return mem[v][turn] = 2;
    if (He_has_3)
        return mem[v][turn] = 0;
    if (I_have_3)
        return mem[v][turn] = 1;
    if (state.is_full())
        return mem[v][turn] = 2;
    int res = 0, t;
    for (int i = 0; i < 4 && bound; i++)
```

```
        for (int j = 0; j < 4 && bound; j++)
            if (state.board[i][j] == '.') {
                state.board[i][j] = c;
                for (int p = 0; p < 4 && bound; p++) {
                    for (int d = -1; d < 2 && bound; d += 2) {
                        state.rotate(p, d);
                        t = play(state, 1 - turn);
                        state.rotate(p, -d);
                        if (t == 0) {
                            res = 1; // I can win
                        }
                        else if (t == 1) {
                            // do nothing
                        }
                        else {
                            if (res == 0)
                                res = 2;
                        }
                    }
                }
                state.board[i][j] = '.';
            }
    return mem[v][turn] = res;
}
int main() {
    memset(mem, -1, sizeof mem);
    node start;
    int t;
    cin >> t;
    string s[4];
    while (t--)  {
        for (int i = 0;i<4;i++)
            cin >> s[i];
        int X = 0, O = 0;
        for (int i = 0; i < 4 ; i++)
            for (int j = 0; j < 4 ; j++) {
                start.board[i][j] = s[i][j];
                X += s[i][j] == 'B';
                O += s[i][j] == 'W';
            }
        assert(!start.is_winner('B'));
        assert(!start.is_winner('W'));
        assert(!start.is_full());
        int ans, ret = play(start, O > X);
        if (ret == 2)
            ans = 2;
        else
            ans = O > X ? 1 - ret : ret;
        if (ans == 2)
            cout << "Draw" << endl;
        else if (ans == 1)
```

```
            cout << "Sohrab" << endl;
        else
            cout << "Sepehr" << endl;
    }
    return 0;
}
```

راه‌حل سوال C

```
int main() {
    int t, n, s, arr[15];
    cin >> t;
    while (t--) {
        cin >> n;
        s = 0;
        for (int i = 0; i < n ; i++)   {
            cin >> arr[i];
            s += arr[i];
        }
        if (s % n != 0)
            cout << "Sohrab" << endl;
        else
            cout << "Sepehr" << endl;
    }
    return 0;
}
```

راه‌حل سوال D

```
struct Edge {
    int u;
    int v;
    int cost;
} Elist[100001];
bool operator<(Edge a, Edge b) {
    return a.cost > b.cost;
}
struct UnionFind {
    vector<int> data;
    UnionFind(int size) : data(size, -1) {}
    bool unionSet(int x, int y) {
        x = root(x); y = root(y);
        if (x != y) {
            if (data[y] < data[x]) swap(x, y);
            data[x] += data[y]; data[y] = x;
        }
        return x != y;
    }
    bool findSet(int x, int y) {
        return root(x) == root(y);
    }
```

```
    int root(int x) {
        return data[x] < 0 ? x : data[x] = root(data[x]);
    }
    int size(int x) {
        return -data[root(x)];
    }
};
set<int> comp;
bool used[100001];
int main() {
    int t, n, m, c, a, b, d;
    cin >> t;
    assert(t <= 35);
    while (t--)   {
        cin >> n >> m >> c;
        assert(n >= 2 && n <= 10000 && m >= 0 && m <= 10000 && c >= 1 &&
c <= 100000);
        for (int i = 0; i < m ; i++)   {
            cin >> a >> b >> d;
            assert(a >= 1 && a <= n && b >= 1 && b <= n && d <= 100000 &&
a != b);
            Elist[i].u = a - 1;
            Elist[i].v = b - 1;
            Elist[i].cost = d;
        }
        UnionFind S(n);
        memset(used, 0, sizeof used);
        comp.clear();
        sort(Elist, Elist + m);
        for (int i = 0; i < m; i++) {
            if (S.findSet(Elist[i].u, Elist[i].v))
                continue;
            S.unionSet(Elist[i].u, Elist[i].v);
            used[i] = 1;
        }
        for (int i = 0; i < n; i++)
            comp.insert(S.root(i));
        int need = comp.size() - 1;
        int total = 0;
        int newtaxi = 0;
        int oldtaxi = 0;
        for (int i = m - 1; i >= 0 && need > 0; i--) {
            if (used[i])
                continue;
            if (Elist[i].cost > c) {
                newtaxi++;
                total += c;
                i++;
                need--;
            }
            else {
```

```
                oldtaxi++;
                total += Elist[i].cost;
                need--;
            }
        }
        while (need) {
            need--;
            newtaxi++;
            total += c;
        }
        cout << newtaxi << " " << oldtaxi << " " << total << endl;
    }
    return 0;
}
```

راه‌حل سوال E

```
#define int long long
int p[3][2], x, y, r;
int cross(int x1, int y1, int x2, int y2) {
    return x1 * y2 - y1 * x2;
}
int dot(int x1, int y1, int x2, int y2) {
    return x1 * x2 + y1 * y2;
}
bool is_center_in_triangle() {
    bool has_neg = false;
    bool has_pos = false;
    int cr;
    for (int i = 0; i < 3; i++) {
        cr = cross(x - p[i][0], y - p[i][1], p[(i + 1) % 3][0] - p[i][0],
p[(i + 1) % 3][1] - p[i][1]);
        has_neg = has_neg || cr < 0;
        has_pos = has_pos || cr > 0;
    }
    return has_neg xor has_pos;
}
int p2(int x) {
    return x * x;
}
pair<int, int> point_to_segment2(int x1, int y1, int x2, int y2, int x3,
int y3) {
    pair<int, int> res;
    if (dot(x2 - x1, y2 - y1, x3 - x2, y3 - y2) > 0) {
        res.first = (x3 - x2)*(x3 - x2) + (y3 - y2)*(y3 - y2);
        res.second = 1;
    }
    else if (dot(x1 - x2, y1 - y2, x3 - x1, y3 - y1) > 0) {
        res.first = (x3 - x1)*(x3 - x1) + (y3 - y1)*(y3 - y1);
        res.second = 1;
    }
```

```
    else {
        res.first = p2((x2 - x1)*(y3 - y1) - (y2 - y1)*(x3 - x1));
        res.second = (x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1);
    }
    return res;
}
main() {
    int t;
    cin >> t;
    assert(t <= 1000 && t > 0);
    while (t--)   {
        for (int i = 0; i < 3; i++) {
            cin >> p[i][0] >> p[i][1];
            assert(p[i][0] <= 1000 && p[i][1] <= 1000 && p[i][0] >= -1000
&& p[i][1] >= -1000);
        }
        assert(cross(p[2][0] - p[0][0], p[2][1] - p[0][1], p[1][0] -
p[0][0], p[1][1] - p[0][1]) != 0);
        cin >> x >> y >> r;
        assert(x <= 1000 && y <= 1000 && x >= -1000 && y >= -1000);
        assert(r >= 1 && r <= 1000);
        if (is_center_in_triangle())
            cout << "YES" << endl;
        else {
            pair<int, int> res;
            bool has_intersect = false;
            for (int i = 0; i < 3; i++) {
                res = point_to_segment2(p[i][0], p[i][1], p[(i + 1) %
3][0], p[(i + 1) % 3][1], x, y);
                has_intersect = has_intersect || (res.first <= res.second
* r * r);
            }
            if (has_intersect)
                cout << "YES" << endl;
            else
                cout << "NO" << endl;
        }
    }
    return 0;
}
```

راه‌حل سوال F

```
map<pair<string, string>, string> M;
set<string> rows;
set<string> cols;
string just3(string a) {
    while (a.size() < 3)
        a += ' ';
    return a;
}
```

```
int main() {
    int t;
    cin >> t;
    while (t--)   {
        M.clear();
        rows.clear();
        cols.clear();
        int n;
        string m, p, c;
        cin >> n;
        for (int i = 0; i < n; i++) {
            cin >> m >> p >> c;
            M[make_pair(m, p)] = c;
            rows.insert(m);
            cols.insert(p);
        }
        string line_separator = "+";
        for (int i = 0; i < cols.size() + 1; i++)
            line_separator += "---+";
        cout << line_separator << endl;
        cout << "|   |";
        for (set<string>::iterator it = cols.begin();it !=
cols.end();it++)
            cout << just3(*it) << "|";
        cout << endl;
        cout << line_separator << endl;
        for (set<string>::iterator rowit = rows.begin();rowit !=
rows.end();rowit++) {
            cout << "|" << just3(*rowit) << "|";
            for (set<string>::iterator colit = cols.begin();colit !=
cols.end();colit++) {
                if (M.find(make_pair(*rowit, *colit)) != M.end())
                    cout << just3(M[make_pair(*rowit, *colit)]) << "|";
                else
                    cout << "   |";
            }
            cout << endl;
            cout << line_separator << endl;
        }
    }
    return 0;
}
```

راه‌حل سوال G

```
#define N 100010
#define lchild node<<1
#define rchild (node<<1) + 1
int L[N << 2], R[N << 2], leftmost[N << 2], rightmost[N << 2], sum[N <<
2];
string s;
```

```
void go_up(int node) {
    sum[node] = sum[lchild] + sum[rchild];
    if (leftmost[rchild] != -1 && rightmost[lchild] != -1) {
        if (s[leftmost[rchild]] == '<' && s[rightmost[lchild]] == '>')
            sum[node]++;
    }
    if (leftmost[lchild] != -1)
        leftmost[node] = leftmost[lchild];
    else if (leftmost[rchild] != -1)
        leftmost[node] = leftmost[rchild];
    else
        leftmost[node] = -1;
    if (rightmost[rchild] != -1)
        rightmost[node] = rightmost[rchild];
    else if (rightmost[lchild] != -1)
        rightmost[node] = rightmost[lchild];
    else
        rightmost[node] = -1;
}
void build_tree(int node, int l, int r) {
    L[node] = l;
    R[node] = r;
    leftmost[node] = l;
    rightmost[node] = r;
    if (l == r)   {
        sum[node] = 0;
        return;
    }
    int m = (l + r) / 2;
    build_tree(lchild, l, m);
    build_tree(rchild, m + 1, r);
    go_up(node);
    return;
}
void stand_sit(int node, int x) {
    if (x < L[node] || x > R[node])
        return;
    if (L[node] == R[node]) {
        int cur = leftmost[node];
        leftmost[node] = rightmost[node] = (cur == -1 ? L[node] : -1);
    }
    else {
        stand_sit(lchild, x);
        stand_sit(rchild, x);
        go_up(node);
    }
    return;
}
void turn(int node, int x) {
    if (x < L[node] || x > R[node])
        return;
```

```
    if (L[node] == R[node]) {
        char ch = s[L[node]];
        s[L[node]] = ch == '<' ? '>' : '<';
    }
    else {
        turn(lchild, x);
        turn(rchild, x);
        go_up(node);
    }
    return;
}
int query(int node, int left, int right) {
    if (right < L[node] || R[node] < left)
        return 0;
    if (left <= L[node] && R[node] <= right)
        return sum[node];
    int res = query(lchild, left, right) + query(rchild, left, right);
    if (leftmost[rchild] <= right && rightmost[lchild] <= right &&
        left <= rightmost[lchild] && left <= leftmost[rchild] &&
        s[leftmost[rchild]] == '<' && s[rightmost[lchild]] == '>')
        res++;
    return res;
}
int main() {
    int t, n, m;
    cin >> t;
    while (t--)   {
        cin >> n >> m;
        assert(n <= 100000 && n >= 1);
        assert(m <= 100000 && m >= 1);
        cin >> s;
        assert(s.size() == n);
        build_tree(1, 0, n - 1);
        int q, x, l, r;
        for (int i = 0;i<m;i++) {
            cin >> q;
            assert(q <= 3 && q >= 1);
            if (q == 1) {
                cin >> x;
                assert(x <= n && x >= 1);
                x--;
                stand_sit(1, x);
            }
            else if (q == 2) {
                cin >> x;
                assert(x <= n && x >= 1);
                x--;
                turn(1, x);
            }
            else {
                cin >> l >> r;
```

```
            assert(l <= n && r <= n && r >= 1 && l >= 1);
            assert(l <= r);
            l--, r--;
            cout << query(1, l, r) << endl;
        }
    }
}
return 0;
}
```

راه‌حل سوال H

```
#define int long long
#define GOD 0x7fffffffffffffll
#define INF 0x7fffffff
vector<vector<int> > col;
void print(vector<vector<int> > v) {
    for (int i = 0;i<v.size();i++, cout << endl)
        for (int j = 0;j<v[0].size();j++)
            cout << v[i][j] << " ";
}
vector<vector<int> > transpose(vector<vector<int> > v) {
    int n = v.size();
    int m = v[0].size();
    vector<int> temp;
    vector<vector<int> > res;
    for (int j = 0; j < m; j++) {
        temp.clear();
        for (int i = 0; i < n; i++)
            temp.push_back(v[i][j]);
        res.push_back(temp);
    }
    return res; // temp
}
int L;
int arr[20001];
int max_sum(int n) {
    int m = L / n;
    vector < vector < int > > v(n, vector < int >(m));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            v[i][j] = arr[m * i + j];
    if (n > m) {
        v = transpose(v);
        swap(m, n);
    }
    int max_elem = -GOD;
    for (int i = 0;i<n;i++)
        for (int j = 0;j<m;j++)
            max_elem = max(max_elem, v[i][j]);
    if (max_elem < 0)
```

```
        return max_elem;
    col.resize(n + 1);
    for (int i = 0;i<n + 1;i++)
        col[i].resize(m);
    for (int j = 0;j<m;j++)
        col[0][j] = 0;
    for (int i = 0;i<n;i++)
        for (int j = 0;j<m;j++)
            col[i + 1][j] = col[i][j] + v[i][j];
    int res = 0;
    int mx, s;
    for (int upper = 0; upper < n; upper++)
        for (int lower = upper; lower < n; lower++) {
            mx = 0;
            s = 0;
            for (int j = 0;j<m;j++) {
                s = max(0ll, s + col[lower + 1][j] - col[upper][j]);
                mx = max(mx, s);
            }
            res = max(res, mx);
        }
    return res;
}
main() {
    int t, m, a;
    cin >> t;
    while (t--)   {
        cin >> L >> m;
        for (int i = 0;i<L;i++)
            arr[i] = 1;
        for (int i = 0;i<m;i++) {
            cin >> a;
            arr[a - 1] = -INF;
        }
        int ans = 0;
        for (int i = 1;i*i <= L;i++) {
            if (L % i != 0)
                continue;
            ans = max(ans, max_sum(i));
            if (i * i != L)
                ans = max(ans, max_sum(L / i));
        }
        cout << ans << endl;
    }
    return 0;
}
```

راه‌حل سوال I

```
#define int long long
int arr[5001], R[5001], kheng[5001], exp[5001], wild[5001], ans[5001];
```

```
int gcnt[5001][5001];
int lcnt[5001][5001];
string D[5001];
pair<int, int> L[5001];
int Q(char c) {
    return c - '0';
}
int get_idx(string s) {
    int year = Q(s[0]) * 1000 + Q(s[1]) * 100 + Q(s[2]) * 10 + Q(s[3]);
    int month = Q(s[5]) * 10 + Q(s[6]);
    int day = Q(s[8]) * 10 + Q(s[9]);
    return 300000 - ((year - 1900) * 150 * 150 + month * 150 + day);
}
main() {
    int n, t, u;
    cin >> t;
    int y = 0;
    while (t--)   {
        y++;
        cin >> n;
        for (int i = 0;i<n;i++) {
            cin >> D[i] >> R[i];
            L[i] = make_pair(-R[i], i);
        }
        sort(L, L + n);
        for (int i = 0;i<n;i++)
            arr[i] = get_idx(D[L[i].second]);
        int denum = 0;
        for (int i = 0;i<n;i++) {
            gcnt[i][0] = (arr[0] > arr[i] ? 1 : 0);
            lcnt[i][0] = (arr[0] < arr[i] ? 1 : 0);
            for (int j = 1;j<n;j++) {
                gcnt[i][j] = gcnt[i][j - 1] + (arr[j] > arr[i] ? 1 : 0);
                lcnt[i][j] = lcnt[i][j - 1] + (arr[j] < arr[i] ? 1 : 0);
            }
        }
        memset(kheng, 0, sizeof kheng);
        memset(exp, 0, sizeof exp);
        memset(wild, 0, sizeof wild);
        for (int i = 0;i<n;i++)
            for (int j = i + 2;j<n;j++)
                if (arr[j] > arr[i]) {
                    u = gcnt[j][j] - gcnt[j][i];
                    denum += u;
                    kheng[i] += u;
                    wild[j] += u;
                }
        for (int i = 1;i<n - 1;i++)
            for (int j = i + 1;j<n;j++)
                if (arr[j] < arr[i])
                    exp[i] += lcnt[j][i];
```

```
        if (denum == 0)
            cout << "Rules Should Be Changed!" << endl;
        else {
            for (int i = 0;i<n;i++)
                ans[L[i].second] = wild[i] + exp[i] + kheng[i];
            for (int i = 0;i<n;i++) {
                cout << setiosflags(ios::fixed | ios::showpoint) <<
setprecision(5) << double(ans[i]) / denum;
                if (i + 1 != n)
                    cout << " ";
            }
            cout << endl;
        }
    }
}
```

راه‌حل سوال J

```
bool un(int a, int b, int c, int d) {
    int mn = min(b, d);
    int mx = max(a, c);
    return mn >= mx;
}
struct segment {
    int l;
    int r;
    int idx;
    segment() {}
    segment(int _l, int _r, int _idx) {
        l = _l;
        r = _r;
        idx = _idx;
    }
    void print() {
        cout << l << " to " << r << " idx: " << idx << endl;
    }
};
bool operator<(segment a, segment b) {
    if (a.r < b.r)
        return true;
    if (a.r > b.r)
        return false;
    if (a.l < b.l)
        return true;
    if (a.l > b.l)
        return false;
    return a.idx < b.idx;
}
vector<segment> v;
queue<segment> q;
int main() {
```

```
    int n, m, l, r;
    int t;
    cin >> t;
    while (t--)   {
        cin >> n >> m;
        while (q.size())
            q.pop();
        v.clear();
        for (int i = 0;i<n;i++) {
            cin >> l >> r;
            if (r < l)
                for (int j = 0;j < 3;j++)
                    v.push_back(segment(l + j * m, r + j * m + m, i));
            else
                for (int j = 0;j < 3;j++)
                    v.push_back(segment(l + j * m, r + j * m, i));
        }
        int cur = -1;
        sort(v.begin(), v.end());
        segment top;
        for (int i = 0;i<v.size();i++) {
            if (v[i].l <= cur)
                continue;
            if (q.size() == 0) {
                cur = v[i].r;
                v[i].l += m;
                v[i].r += m;
                q.push(v[i]);
            }
            else {
                top = q.front();
                if (top.idx == v[i].idx)
                    break;
                else {
                    if (un(top.l, top.r, v[i].l, v[i].r)) {
                        q.pop();
                        cur = v[i].r;
                        v[i].l += m;
                        v[i].r += m;
                        q.push(v[i]);
                    }
                    else {
                        cur = v[i].r;
                        v[i].l += m;
                        v[i].r += m;
                        q.push(v[i]);
                    }

                }
            }
        }
```

```
        cout << q.size() << endl;
    }
    return 0;
}
```

راه‌حل سوال K

```
int A[101], S[101], n, P, p;
int masks[101][101];
vector<int> factors;
pair<int, int> dp[2][101][1 << 7];
int res_pre[1 << 7];
int main() {
    int t;
    cin >> t;
    while (t--) {
        cin >> n >> P;
        memset(masks, 0, sizeof masks);
        factors.clear();
        p = P;
        for (int i = 2;i*i <= P && p > 1;i++)
            while (p % i == 0) {
                factors.push_back(i);
                p /= i;
            }
        if (p > 1)
            factors.push_back(p);
        sort(factors.begin(), factors.end());
        for (int i = 0;i<factors.size() - 1;i++)
            assert(factors[i] != factors[i + 1]);
        for (int i = 0;i<n;i++)
            cin >> A[i];
        S[0] = 0;
        for (int i = 0;i<n;i++)
            S[i + 1] = S[i] + A[i];
        for (int i = 0;i<n;i++)
            for (int j = i;j<n;j++)
                for (int k = 0;k<factors.size();k++)
                    if ((S[j + 1] - S[i]) % factors[k] == 0)
                        masks[i][j] |= (1 << k);
        for (int mask = 0;mask<(1 << factors.size());mask++) {
            res_pre[mask] = 1;
            for (int j = 0;j<factors.size();j++)
                if (mask & (1 << j))
                    res_pre[mask] *= factors[j];
        }
        pair<int, int> r1, r2;
        for (int r = n - 1; r >= 0; r--)
            for (int first = 0;first <= r; first++)
                for (int mask = 0;mask <= (1 << factors.size()) -
1;mask++)
```

```
                if (r == n - 1)
                    dp[r & 1][first][mask] = make_pair(res_pre[mask |
masks[first][r]], 1);
                else {
                    r1 = dp[~r & 1][first][mask];
                    r2 = dp[~r & 1][r + 1][mask | masks[first][r]];
                    if (r1.first > r2.first)
                        dp[r & 1][first][mask] = r1;
                    else if (r2.first > r1.first)
                        dp[r & 1][first][mask] = make_pair(r2.first,
r2.second + 1);
                    else
                        dp[r & 1][first][mask] = make_pair(r1.first,
min(r1.second, r2.second + 1));
                }
        cout << dp[0][0][0].first << " " << dp[0][0][0].second - 1 <<
endl;
    }
    return 0;
}
```

راه‌حل سوال L

```
string s, key, res;
vector<int> cycles[27];
vector<int> idx[27];
int color[27] = { 0 };
int place_in_ring[27];
int number_of_cycles;
int first_occurence_of_ring[27];
int main() {
    int t, n;
    cin >> t;
    while (t--) {
        cin >> n >> s >> key;
        res = "";
        for (int i = 0;i<27;i++) {
            cycles[i].clear();
            idx[i].clear();
        }
        memset(color, 0, sizeof color);
        memset(place_in_ring, 0, sizeof place_in_ring);
        number_of_cycles = 1;
        memset(first_occurence_of_ring, -1, sizeof
first_occurence_of_ring);
        for (int i = 0;i<key.size();i++) {
            if (color[i])
                continue;
            int cur = i;
            int p = 0;
            while (color[cur] == 0) {
```

```
                color[cur] = number_of_cycles;
                cycles[number_of_cycles].push_back(cur);
                place_in_ring[cur] = p;
                cur = key[cur] - 'a';
                p++;
            }
            number_of_cycles++;
        }
        vector<int> good_positions;
        string ans = "{";
        for (int i = 0;i<s.size();i++)
            if (first_occurence_of_ring[color[s[i] - 'a']] == -1) {
                first_occurence_of_ring[color[s[i] - 'a']] = i;
                good_positions.push_back(i);
                ans += s[i];
            }
        int max_of_answer = 1;
        for (int i = 1;i<number_of_cycles;i++)
            max_of_answer *= cycles[i].size();
        int gp, col, mn = max_of_answer + 2;
        for (int i = 0;i <= max_of_answer;i++) {
            string temp;
            for (int j = 0;j<good_positions.size();j++) {
                gp = good_positions[j];
                col = color[s[gp] - 'a'];
                temp += char(cycles[col][(place_in_ring[s[gp] - 'a'] + i)
% cycles[col].size()] + 'a');
            }
            if (temp < ans) {
                ans = temp;
                mn = i;
            }
        }
        for (int i = 0;i<s.size();i++) {
            col = color[s[i] - 'a'];
            res += char(cycles[col][(place_in_ring[s[i] - 'a'] + mn) %
cycles[col].size()] + 'a');
        }
        cout << res << endl;
    }
    return 0;
}
```

۷-۱۰-۳- رقابت برنامه‌نویسی دانشجویی، مرحله اینترنتی سال ۸۷ – دانشگاه آزاد اسلامی مشهد

سوال A

```
ifstream fin("weather.in");
#define cin fin
int t;
```

```
map<string, int> ma;
int main() {
    cin >> t;
    ma["Sat"] = 0;
    ma["Fri"] = 1;
    ma["Thu"] = 2;
    ma["Wed"] = 3;
    ma["Tue"] = 4;
    ma["Mon"] = 5;
    ma["Sun"] = 6;
    string cur;
    string bestcity;
    int besttemp;
    string ruz;
    string tempruz;
    string bestruz;
    int besttempruz;
    for (int ii = 0; ii<t; ii++) {
        besttemp = -200000;
        cin >> cur;
        for (; cur != "End";) {
            tempruz = cur;
            int besttempruz = 2000000;
            while (1) {
                cin >> cur;
                if (ma.find(cur) != ma.end() || cur == "End")
                    break;
                int temp;
                cin >> temp;
                if (temp < besttempruz || (temp == besttempruz && cur <
bestruz)) {
                    besttempruz = temp;
                    bestruz = cur;
                }
            }
            if (besttempruz > besttemp || (besttemp == besttempruz &&
bestcity > bestruz)) {
                besttemp = besttempruz;
                bestcity = bestruz;
                ruz = tempruz;
            }
        }
        cout << bestcity << ' ' << ruz << endl;
    }
    return 0;
}
```

سوال B

```
ifstream fin("shortmessage.in");
#define cin fin
```

```
map<char, int> ma;
string str;
int cur;
int tedad;
bool harf(char ch) {
    return (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z');
}
bool bozorg(char & ch) {
    if (ch >= 'A' && ch <= 'Z') {
        ch = ch - 'A' + 'a';
        return true;
    }
    return false;
}
int main() {
    cur = 0;
    tedad = 0;
    for (char ch = 'a'; ch <= 'r'; ch++)
        ma[ch] = (ch - 'a') % 3 + 1;
    for (char ch = 't'; ch <= 'y'; ch++)
        ma[ch] = (ch - 't') % 3 + 1;
    ma['s'] = ma['z'] = 4;
    ma['.'] = 1;
    ma[','] = 2;
    ma['?'] = 3;
    ma['!'] = 4;
    for (char ch = '1'; ch <= '9'; ch++)
        ma[ch] = 4;
    ma['1'] = ma['7'] = ma['9'] = 5;
    ma[' '] = 1;
    ma['0'] = 2;
    bool first = true;
    while (getline(cin, str)) {
        if (str.find('#') < str.length()) {
            if (!first) {
                cur = 0;
                cout << tedad << endl;
                tedad = 0;
            }
            else first = false;
            getline(cin, str);
        }
        for (int i = 0; i<str.length(); i++)
            if (!harf(str[i]))
                tedad += ma[str[i]];
            else if (bozorg(str[i])) {
                if (cur == 0)
                    tedad++;
                cur = 1;
                tedad += ma[str[i]];
            }
```

```
            else {
                if (cur == 1)
                    tedad++;
                cur = 0;
                tedad += ma[str[i]];
            }
            tedad += 3;
    }
    cout << tedad << endl;
    return 0;
}
```

سوال C

```
ifstream fin("tests.in");
#define cin fin
int t, k, a, b;
bool mark[101][4000];
int main() {
    cin >> t;
    for (int ii = 0; ii<t; ii++) {
        cin >> k >> a >> b;
        memset(mark, 0, sizeof mark);
        mark[0][2000] = true;
        for (int i = 1; i <= k; i++)
            for (int j = 0; j<4000; j++)
                if (mark[i - 1][j])
                    mark[i][j] = mark[i][j + a] = mark[i][j + b] = true;
        int cnt = 0;
        for (int i = 0; i<4000; i++)
            if (mark[k][i])
                cnt++;
        cout << cnt << endl;
    }
    return 0;
}
```

سوال D

```
ifstream fin("gregariousness.in");
#define cin fin
int t, n;
int graph[1000][1000];
int num[1000];
bool iskhun[1000];
int main() {
    cin >> t;
    for (int ii = 0; ii<t; ii++) {
        cin >> n;
        memset(num, 0, sizeof num);
```

```
        memset(iskhun, 0, sizeof iskhun);
        for (int i = 0; i<n; i++)
            for (int j = 0; j<n; j++)
                cin >> graph[i][j];
        for (int i = 0; i<n; i++)
            for (int j = 0; j<n; j++)
                if (graph[i][j] == 1)
                    num[i]++;
        for (int i = 0; i<n; i++)
            if (num[i]>0) {
                iskhun[i] = true;
                for (int j = 0; j<n; j++)
                    if (num[i] < num[j] && graph[i][j] == 1)
                        iskhun[i] = false;
            }

        bool first = true;
        for (int i = 0; i<n; i++)
            if (iskhun[i])
                if (first) {
                    cout << i + 1;
                    first = false;
                }
                else
                    cout << ' ' << i + 1;
        cout << endl;

    }
    return 0;
}
```

سوال E

```
ifstream fin("zeroone.in");
#define cin fin
int mat[200][200];
int tmat[200][200];
int t, n;
int swapr(int i) {
    for (int j = 0; j<n; j++)
        tmat[i][j] = 1 - tmat[i][j];
    return 0;
}
int swapc(int i) {
    for (int j = 0; j<n; j++)
        tmat[j][i] = 1 - tmat[j][i];
    return 0;
}
int solve() {
    int res = 0;
    for (int i = 0; i<n; i++)
```

```
        if (tmat[0][i] == 1) {
            swapc(i);
            res++;
        }
    for (int i = 1; i<n; i++) {
        int exist = false;
        int all = true;
        for (int j = 0; j<n; j++)
            if (tmat[i][j] == 1)
                exist = true;
            else
                all = false;
        if (exist && !all)
            return 4000000;
        if (exist)
            res++;
    }
    return res;
}
int main() {
    cin >> t;
    for (int ii = 0; ii<t; ii++) {
        cin >> n;
        for (int i = 0; i<n; i++)
            for (int j = 0; j<n; j++)
                cin >> mat[i][j];
        memcpy(tmat, mat, sizeof mat);
        swapr(0);
        int res = 4000000;
        int temp = solve();
        if (temp + 1 < res)
            res = temp + 1;
        memcpy(tmat, mat, sizeof mat);
        temp = solve();
        if (temp < res)
            res = temp;
        if (res >= 4000000)
            cout << "Impossible" << endl;
        else
            cout << res << endl;
    }
    return 0;
}
```

سوال F

```
ifstream fin("series1.in");
#define cin fin
int zoj[600], fard[600];
int a[1000];
int Fard(int t);
```

```
int Zoj(int t) {
    if (zoj[t] != -1)
        return zoj[t];
    if (t == 0)
        return -2;
    int & res = zoj[t];
    res = -2;
    for (int i = 0; i<t; i++)
        if (Fard(i) != -2 && a[i] < a[t] && Fard(i) + 1 > res)
            res = Fard(i) + 1;
    return res;
}
int Fard(int t) {
    if (t == 0)
        return 1;
    if (fard[t] != -1)
        return fard[t];
    int & res = fard[t];
    res = 1;
    for (int i = 0; i<t; i++)
        if (Zoj(i) != -2 && a[i] > a[t] && Zoj(i) + 1 > res)
            res = Zoj(i) + 1;
    return res;
}
int t, n;
int main() {
    cin >> t;
    for (int i = 0; i<t; i++) {
        memset(zoj, -1, sizeof zoj);
        memset(fard, -1, sizeof fard);
        cin >> n;
        for (int i = 0; i<n; i++)
            cin >> a[i];
        int res = 0;
        for (int i = 0; i<n; i++) {
            if (Zoj(i) > res)
                res = Zoj(i);
            if (Fard(i)> res)
                res = Fard(i);
        }
        cout << res << endl;
    }
    return 0;
}
```

سوال G

```
ifstream fin("game2.in");
#define cin fin
vector<int> mat[301][301];
bool isset[301][301];
```

```
int grund[301];
bool matset[301][301][301];
bool gset[301][301];
int Grund(int a);
void solve(int a, int b) {
    if (isset[a][b])
        return;
    if (b == 0) {
        mat[a][b].push_back(0);
        return;
    }
    isset[a][b] = true;
    for (int i = a + 1; i <= b; i++) {
        solve(i, b - i);
        for (int j = 0; j<mat[i][b - i].size(); j++)
            matset[a][b][Grund(i) ^ mat[i][b - i][j]] = true;
    }
    for (int i = 0; i<300; i++)
        if (matset[a][b][i])
            mat[a][b].push_back(i);
}
int Grund(int a) {
    if (grund[a] != -1)
        return grund[a];
    int& res = grund[a];
    for (int i = 1; i<a; i++) {
        solve(i, a - i);
        for (int j = 0; j<mat[i][a - i].size(); j++)
            gset[a][mat[i][a - i][j] ^ Grund(i)] = true;
    }
    res = 0;
    for (int i = 0; i < 10000000; i++)
        if (!gset[a][i])
            return res = i;
}
int t, n;
int main() {
    memset(isset, 0, sizeof isset);
    memset(grund, -1, sizeof grund);
    memset(matset, 0, sizeof matset);
    memset(gset, 0, sizeof gset);
    cin >> t;
    grund[0] = 0;
    grund[1] = 0;
    grund[2] = 0;
    //cout<<Grund(4)<<' '<<Grund(5)<<' '<<Grund(6)<<' '<<Grund(7)<<'
'<<Grund(300)<<endl;
    for (int ii = 0; ii<t; ii++) {
        cin >> n;
        int res = 0;
        for (int i = 0; i<n; i++) {
```

```
            int temp;
            cin >> temp;
            res ^= Grund(temp);
        }
        if (res == 0)
            cout << "NO" << endl;
        else
            cout << "YES" << endl;
    }
    return 0;
}
```

۷-۱۰-۴- مسابقات هفتگی سایت CodeForces – پنج سوال

سوال A

```
#define MAXK 101
int ones[MAXK];
int twos[MAXK];
int n, k;
int main() {
    int num;
    cin >> n >> k;
    for (int i = 0; i<n; i++) {
        cin >> num;
        if (num == 1)
            ones[i%k]++;
        else
            twos[i%k]++;
    }
    int minCount = 0;
    for (int i = 0; i<k; i++)
        minCount += min(ones[i], twos[i]);
    cout << minCount << endl;
    return 0;
}
```

سوال B

```
#define MAXK 101
int gcd(int a, int b) {
    while ((a%b) != 0) {
        int temp = a;
        a = b;
        b = temp%b;
    }
    return b;
}
int mazareb235(int num) {
    int count = 0;
```

```
    while (num>1) {
        if ((num % 2) == 0)  {
            count++;
            num /= 2;
        }
        else if ((num % 3) == 0) {
            count++;
            num /= 3;
        }
        else if ((num % 5) == 0) {
            count++;
            num /= 5;
        }
        else
            return -1;
    }
    return count;
}
int main() {
    int a, b;
    cin >> a >> b;
    if (a<b)
        swap(a, b);
    int g = gcd(a, b);
    int anum = a / g;
    int bnum = b / g;
    int aCount = mazareb235(anum);
    int bCount = mazareb235(bnum);
    if (aCount == -1 || bCount == -1)
        cout << -1 << endl;
    else
        cout << aCount + bCount << endl;
    return 0;
}
```

سوال C

```
int needed[3], ns[3], ps[3];
long long r = 0;
int main() {
    string sand;
    cin >> sand;
    for (int i = 0; i<sand.size(); i++) {
        switch (sand[i]) {
            case 'B':
                needed[0]++;
                break;
            case 'S':
                needed[1]++;
                break;
            case 'C':
```

```
                needed[2]++;
                break;
            }
        }
        for (int i = 0; i<3; i++)
            cin >> ns[i];
        long long sandPrice = 0, supplies = 0;
        for (int i = 0; i<3; i++) {
            cin >> ps[i];
            sandPrice += needed[i] * ps[i];
            supplies += needed[i] == 0 ? 0 : ns[i];
        }
        cin >> r;
        long long count = 0;
        while (supplies>0) {
            bool newAdded = true;
            for (int i = 0; i<3; i++) {
                if (ns[i] >= needed[i]) {
                    ns[i] -= needed[i];
                    supplies -= needed[i];
                }
                else if (r >= (needed[i] - ns[i])*ps[i]) {
                    supplies -= ns[i];
                    r -= (needed[i] - ns[i])*ps[i];
                    ns[i] = 0;
                }
                else {
                    newAdded = false;
                    break;
                }
            }
            if (newAdded) count++;
            else break;
        }
        count += r / sandPrice;
        cout << count << endl;
        return 0;
}
```

سوال D

```
#define MAXV 200001
int vessels[MAXV], current[MAXV], n = 0;
set<int> notFulls;
int main() {
    scanf("%d", &n);
    for (int i = 0; i<n; i++) {
        scanf("%d", &vessels[i]);
        notFulls.insert(i);
    }
    int m, type, k, p, x, rem;
```

```
    scanf("%d", &m);
    for (int i = 0; i<m; i++) {
        scanf("%d", &type);
        switch (type) {
            case 2:
                scanf("%d", &k);
                printf("%d\n", current[k - 1]);
                break;
            case 1:
                scanf("%d %d", &p, &x);
                p--;
                current[p] += x;
                while (p<n && current[p]>vessels[p]) {
                    rem = current[p] - vessels[p];
                    current[p] -= rem;
                    notFulls.erase(p);
                    set<int>::iterator nextnum = notFulls.upper_bound(p);
                    if (nextnum != notFulls.end()) {
                        p = *nextnum;
                        current[p] += rem;
                    }
                }
                break;
        }
    }
    return 0;
}
```

سوال E

```
const int N = 3 * 100000;
int n;
vector<pair<int, int> > v;
long long sum[N + 1];
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) {
        int p;
        scanf("%d", &p);
        v.push_back(make_pair(p, i));
    }
    sort(v.begin(), v.end());
    sum[0] = 0;
    for (int i = 0; i < n; ++i)
        sum[i + 1] = sum[i] + v[i].first;
    int k;
    scanf("%d", &k);
    long long ans = 1ll << 61;
    int cntx = 0;
    long long cur = 0;
    for (int i = 1; i < k; ++i)
```

```
        cur += v[i].first * i - (sum[i] - sum[0]);
    ans = cur;
    for (int i = 0; i + k < n; ++i) {
        cur -= (sum[i + k] - sum[i + 1]) - (long long)v[i].first * (k -
1);
        cur += (long long)v[i + k].first * (k - 1) - (sum[i + k] - sum[i
+ 1]);
        if (cur < ans) {
            cntx = i + 1;
            ans = cur;
        }
    }
    for (int i = 0; i < k; ++i) {
        printf("%d%c", v[i + cntx].second + 1, " \n"[i == k - 1]);
    }
    return 0;
}
```

مراجع

Arefin, A.S., 2006. *Art of Programming Contest*, Reviewed by Dr. M. Lutfar Rahman and Forworded by Professor Miguel Revilla, University de Valladolid, Spain, Gyankosh Prokashoni, Dhaka, Bangladesh, First Ed., ISBN: 984-32-3382-4.

Skiena, S.S. & Revilla, M.A., 2006. *Programming challenges: The programming contest training manual*, Springer Science & Business Media.

واژه‌نامه‌ی فارسی به انگلیسی

ا

اتصال	Linking
آدرس‌دهی باز	Open addressing
آدرس‌دهی بسته	Closed addressing
ارجاع	Reference
از پیش محاسبه کردن	Pre-computing
استثنا	Exception
اشاره‌گر به تابع	Function pointer
اصلاح‌کننده	Modifier
الگو	Template
الگوریتم	Algorithm
الگوریتم تقریبی	Approximation algorithm
الگوریتم حریصانه	Greedy algorithm
انتزاعی	Abstract
ایجاد کردن	Generating
ایجاد کننده	Generator

ب

بازگشت	Recursion
بالا هرمی	Upheap
بدون قالب	Unformatted
برخورد	Collision
برخط	Online
بردار	Vector
بررسی پیش رو	Forward checking
برنامه	Program
برنامه‌نویسی	Programming
برنامه‌نویسی پویا	Dynamic programming
بزرگترین مقسوم علیه مشترک (ب.م.م.)	Greatest common divisor (gcd)
بلادرنگ	Realtime
بهینه	Optimal

پ

پارامتر	Parameter
پاره‌خط	Line segment
پایداری	Stability
پایین هرمی	Down heap
پرچم‌های مستقل	Independent flags
پرونده اجرایی	Executable file
پر کردن جریانی	Flood fill
پسوند	Suffix
پسوندی	Postorder
پشته	Stack
پنجره خط دستور	Command-line window
پوشش محدب	Convex hull
پیچیدگی زمانی	Time complexity
پیچیدگی فضایی	Space complexity
پیشوند	Prefix
پیشوندی	Preorder
پیمانه	Modulus
پیمایشگر	Iterator

ت

تابع	Function, Method
تابع اکتشافی	Heuristic function
تابع درهم‌سازی	Hash function
تجزیه	Decomposition
ترکیبیات	Combinatorics
تطبیق الگو	Pattern matching
تقارن	Symmetry
تقسیم و غلبه	Divide and conquer
تغییر اندازه‌ی پویا	Dynamic resizing
تناظر یک به یک	Bijection
تناوب	Period
توضیح	Comment

ج

جایگشت	Permutation
جداکننده	Delimiter
جدول پیشوندی	Prefix table
جدول درهم‌سازی	Hash table
جدول درهم‌سازی کامل	Perfect hash table
جریان	Stream
جریان خروجی خطای استاندارد	Standard error output stream
جریان شبکه‌ای	Network flow
جستجوی اکتشافی	Heuristic search
جستجوی اول سطح	Breadth-first search
جستجوی اول عمق	Depth-first search
جستجوی اول عمق عمیق‌شونده تکراری	Iterative deeping depth-first search
جستجوی بیشینه کمینه	Minimax search
جستجوی جامع	Brute-force search
جستجوی رشته	String search
جستجوی عقبگرد	Backtracking search

چ

چرخش یگانه	Single rotation
چرخش دوگانه	Double rotation
چرخه	Loop
چندضلعی	Polygon

ح

حذف معکوس	Reverse-delete
حساس به خروجی	Output sensitive

خ

خصیصه	Field
خط پیمایشی	Sweep line
خطای زمان اتصال	Link-time error
خطای زمان اجرا	Run-time error
خطای زمان ترجمه	Compile-time error
خطای منطقی	Logical error

د

درخت	Tree

درخت پوشای کمینه	Minimum spanning tree
درخت جستجوی دودویی	Binary search tree
درخت کامل	Complete tree
درهم‌سازی باز	Open hashing
درهم‌سازی ترکیبی	Coalesced hashing
درهم‌سازی دوباره	Double hashing
درهم‌سازی رابین‌هود	Robin Hood hasing
درهم‌سازی فاخته	Cuckoo hashing
دنباله فیبوناچی	Fibonacci sequence
دودویی	Binary
ر	
رابطه‌ی بازگشتی	Recurrent relation
رابطه‌ی شمول و عدم شمول	Inclusion-exclusion formula
رخداد	Event
رشته	String
رفع برخورد	Collision resolution
رمزنگاری	Encryption
ز	
زبان برنامه‌نویسی	Programming language
زمان‌بندی	Scheduling
زنجیرسازی	Chaining
س	
ساده‌سازی	Simplification
ساختار	Structure
سرآیند	Header
سرریز	Overflow
سلام دنیا!	Hello world!
ص	
صف	Queue
صف اولویت	Priority queue
ع	
عامل اول	Prime factor
عامل بارگیری	Load factor

عامل توازن	Balance factor
عدد اول	Prime number
عدد بزرگ	Big number
عدد مرکب	Composite number
عدم اصابت در حافظه‌ی نهان	Cache miss
عملگر استخراج‌کننده	Extractor Operator
ف	
فضای حالت	State space
فضای خالی	White space
فیلتر کردن	Filtering
ق	
قالب	Format
قالب اعشاری	Float format
قانون پادساعتگرد بودن	Counterclockwise predicate
قانون جمع	Sum rule
قانون ضرب	Product rule
قویاً همبند	Strongly connected
ک	
کاوش خطی	Linear probing
کاوش درجه دوم	Quadratic probing
کتابخانه	Library
کتابخانه الگوی استاندارد	Standard template library
کد	Code
کدنویسی	Coding
کد اسکی	ASCII code
کد پیشوندی	Prefix code
کد شی	Object code
کلاس	Class
کوتاه‌ترین مسیر	Shortest path
کوچکترین مضرب مشترک (ک.م.م.)	Least common multiple (lcm)
گ	
گراف	Graph
گراف بی‌جهت	Undirected graph

گراف دو بخشی	Bipartite graph
گراف همبند	Connected graph
گراف وزن‌دار	Weighted graph
ل	
لیست	List
م	
ماتریس مجاورت	Adjacency matrix
مبنای عددی	Numerical base
مترجم	Compiler
متصل‌کننده	Linker
متن	Text
متناوب	Periodic
مثلث‌سازی	Triangulation
مجازی	Virtual
مجموعه	Set
مجموعه جدا	Disjoint set
مجموعه چندگانه	Multiset
محاسبات پیمانه‌ای	Modulus arithmetic
محدب	Convex
محیط برنامه‌نویسی	Programming environment
محیط توسعه برنامه	Program development environment
محیط توسعه مجتمع	Integrated development environment
مدار	Circuit
مرتبه	Order
مرتب‌سازی ادغامی	Merge sort
مرتب‌سازی انتخابی	Selection sort
مرتب‌سازی برخط	Online sorting
مرتب‌سازی بینگو	Bingo sort
مرتب‌سازی پیوندی	Strand sort
مرتب‌سازی ترکیبی	Cocktail sort
مرتب‌سازی تند	Fast sort
مرتب‌سازی توپولوژیکی	Topological sort
مرتب‌سازی حبابی	Bubble sort
مرتب‌سازی درجا	In-place sort

مرتب‌سازی درجی	Insertion sort
مرتب‌سازی درخت دودویی	Binary tree sort
مرتب‌سازی درون‌گرا	Introspective sort / IntroSort
مرتب‌سازی سریع	Quicksort
مرتب‌سازی کتابخانه‌ای	Library sort
مرتب‌سازی هرمی	Heap sort
مسأله‌ی ارضای محدودیت	Constraint satisfaction problem
مسأله‌ی خاص	Ad hoc problem
مسأله‌ی کوله‌پشتی	Knapsack problem
مسطح	Planar
مسیر	Path
مضرب	Multiple
مقدار	Value
مقسوم علیه	Divisor
مقعر	Concave
موقعیت	Adjustment
میانگیر	Buffer
میانگیر پرونده	File buffer
میانوندی	Inorder
میانگیر رشته	String buffer

ن

نامتوازن	Unbalanced
نظریه تضاد روز تولد	Birthday paradox theory
نظریه گراف	Graph theory
نگاشت	Map
نویسه	Character

و

وضعیت پیمایش	Sweep status

ه

هرم	Heap
هرم بیشینه	Max heap
هرم‌سازی	Heapify
هرم کمینه	Min heap
هندسه‌ی محاسباتی	Computational geometry

واژه‌نامه‌ی انگلیسی به فارسی

A

Abstract	انتزاعی
Adjacency matrix	ماتریس مجاورت
Adjustment	موقعیت
Ad hoc problem	مسأله‌ی خاص
Algorithm	الگوریتم
Approximation algorithm	الگوریتم تقریبی
ASCII code	کد اسکی

B

Bijection	تناظر یک به یک
Backtracking search	جستجوی عقبگرد
Balance factor	عامل توازن
Big number	عدد بزرگ
Binary	دودویی
Binary search tree	درخت جستجوی دودویی
Binary tree sort	مرتب‌سازی درخت دودویی
Bingo sort	مرتب‌سازی بینگو
Bipartite graph	گراف دو بخشی
Birthday paradox theory	نظریه تضاد روز تولد
Breadth-first search	جستجوی اول سطح
Brute-force search	جستجوی جامع
Bubble sort	مرتب‌سازی حبابی
Buffer	میانگیر

C

Cache miss	عدم اصابت در حافظه‌ی نهان
Chaining	زنجیرسازی
Character	نویسه
Circuit	مدار
Class	کلاس
Closed addressing	آدرس‌دهی بسته
Coalesced hashing	درهم‌سازی ترکیبی
Cocktail sort	مرتب‌سازی ترکیبی

Code	کد
Coding	کدنویسی
Collision	برخورد
Collision resolution	رفع برخورد
Combinatorics	ترکیبیات
Command-line window	پنجره خط دستور
Comment	توضیح
Compiler	مترجم
Compile-time error	خطای زمان ترجمه
Complete tree	درخت کامل
Composite number	عدد مرکب
Computational geometry	هندسه‌ی محاسباتی
Concave	مقعر
Connected graph	گراف همبند
Constraint satisfaction problem	مسأله‌ی ارضای محدودیت
Convex	محدب
Convex hull	پوشش محدب
Counterclockwise predicate	قانون پادساعتگرد بودن
Cuckoo hashing	درهم‌سازی فاخته

D

Decomposition	تجزیه
Delimiter	جداکننده
Depth-first search	جستجوی اول عمق
Disjoint set	مجموعه جدا
Divide and conquer	تقسیم و غلبه
Divisor	مقسوم علیه
Double hashing	درهم‌سازی دوباره
Double rotation	چرخش دوگانه
Down heap	پایین هرمی
Dynamic programming	برنامه‌نویسی پویا
Dynamic resizing	تغییر اندازه‌ی پویا

E

Encryption	رمزنگاری
Event	رخداد

Exception	استثنا
Executable file	پرونده اجرایی
Extractor Operator	عملگر استخراج‌کننده
F	
Fast sort	مرتب‌سازی تند
Fibonacci sequence	دنباله فیبوناچی
Field	خصیصه
File buffer	میانگیر پرونده
Filtering	فیلتر کردن
Float format	قالب اعشاری
Flood fill	پر کردن جریانی
Format	قالب
Forward checking	بررسی پیش رو
Function	تابع
Function pointer	اشاره‌گر به تابع
G	
Generating	ایجاد کردن
Generator	ایجادکننده
Graph	گراف
Graph theory	نظریه گراف
Greatest common divisor (gcd)	بزرگترین مقسوم علیه مشترک (ب.م.م.)
Greedy algorithm	الگوریتم حریصانه
H	
Hash table	جدول درهم‌سازی
Hash function	تابع درهم‌سازی
Header	سرآیند
Heap	هرم
Heapify	هرم‌سازی
Heap sort	مرتب‌سازی هرمی
Heuristic function	تابع اکتشافی
Heuristic search	جستجوی اکتشافی
Hello world!	سلام دنیا!
I	
Inclusion-exclusion formula	رابطه شمول و عدم شمول

Independent flags	پرچم‌های مستقل
Inorder	میانوندی
Insertion sort	مرتب‌سازی درجی
Introspective sort / IntroSort	مرتب‌سازی درون‌گرا
In-place sort	مرتب‌سازی درجا
Integrated development environment	محیط توسعه مجتمع
Iterative deeping depth-first search	جستجوی اول عمق عمیق‌شونده تکراری
Iterator	پیمایشگر
K	
Knapsack problem	مسأله‌ی کوله‌پشتی
L	
Least common multiple (lcm)	کوچکترین مضرب مشترک (ک.م.م.)
Library	کتابخانه
Library sort	مرتب‌سازی کتابخانه‌ای
Linear probing	کاوش خطی
Line segment	پاره‌خط
Linker	متصل‌کننده
Linking	اتصال
Link-time error	خطای زمان اتصال
List	لیست
Load factor	عامل بارگیری
Logical error	خطای منطقی
Loop	چرخه
M	
Map	نگاشت
Max heap	هرم بیشینه
Merge sort	مرتب‌سازی ادغامی
Method	تابع
Min heap	هرم کمینه
Minimax search	جستجوی بیشینه کمینه
Minimum spanning tree	درخت پوشای کمینه
Modifier	اصلاح‌کننده
Modulus	پیمانه
Modulus arithmetic	محاسبات پیمانه‌ای

Multiple	مضرب
Multiset	مجموعه چندگانه
N	
Network flow	جریان شبکه‌ای
Numerical base	مبنای عددی
O	
Object code	کد شی
Online	برخط
Online sorting	مرتب‌سازی برخط
Open addressing	آدرس‌دهی باز
Open hashing	درهم‌سازی باز
Optimal	بهینه
Order	مرتبه
Output sensitive	حساس به خروجی
Overflow	سرریز
P	
Path	مسیر
Parameter	پارامتر
Pattern matching	تطبیق الگو
Period	تناوب
Periodic	متناوب
Perfect hash table	جدول درهم‌سازی کامل
Permutation	جایگشت
Planar	مسطح
Polygon	چندضلعی
Postorder	پسوندی
Prefix	پیشوند
Prefix code	کد پیشوندی
Prefix table	جدول پیشوندی
Preorder	پیشوندی
Pre-computing	از پیش محاسبه کردن
Priority queue	صف اولویت
Prime factor	عامل اول
Prime number	عدد اول

Product rule	قانون ضرب
Program	برنامه
Programming	برنامه‌نویسی
Programming environment	محیط برنامه‌نویسی
Programming language	زبان برنامه‌نویسی
Q	
Quadratic probing	کاوش درجه دوم
Quicksort	مرتب‌سازی سریع
Queue	صف
R	
Realtime	بلادرنگ
Recursion	بازگشت
Recurrent relation	رابطه‌ی بازگشتی
Reference	ارجاع
Reverse-delete	حذف معکوس
Robin Hood hasing	درهم‌سازی رابین‌هود
Run-time error	خطای زمان اجرا
S	
Scheduling	زمان‌بندی
Selection sort	مرتب‌سازی انتخابی
Set	مجموعه
Shortest path	کوتاه‌ترین مسیر
Simplification	ساده‌سازی
Single rotation	چرخش یگانه
Space complexity	پیچیدگی فضایی
Stability	پایداری
Stack	پشته
Standard error output stream	جریان خروجی خطای استاندارد
Standard template library	کتابخانه الگوی استاندارد
State space	فضای حالت
Strand sort	مرتب‌سازی پیوندی
Stream	جریان
String	رشته
String buffer	میانگیر رشته

String search	جستجوی رشته
Strongly connected	قویاً همبند
Structure	ساختار
Suffix	پسوند
Sum rule	قانون جمع
Sweep line	خط پیمایشی
Sweep status	وضعیت پیمایش
Symmetry	تقارن
T	
Template	الگو
Text	متن
Time complexity	پیچیدگی زمانی
Topological sort	مرتب‌سازی توپولوژیکی
Tree	درخت
Triangulation	مثلث‌سازی
U	
Unbalanced	نامتوازن
Undirected graph	گراف بی‌جهت
Unformatted	بدون قالب
Upheap	بالا هرمی
V	
Value	مقدار
Vector	بردار
Virtual	مجازی
W	
Weighted graph	گراف وزن‌دار
White space	فضای خالی