مباحث کلاسی

طراحی الگوریتم

مدرس : مهدی مجد

بهار ۹۹

بسم الله الرحمن الرحیم

مقایسه الگوریتم ها :

دو ملاک مورد بررسی قرار می گیرد.

۱- زمان اجرا
۲- حافظه مصرفی

برای بررسی زمان اجرا می توان برنامه ها را بر روی یک سیستم مشترک اجرا نمود سپس زمان اجرای آنها را محاسبه و مقایسه نمود. اما معمولا زمان اجرای الگوریتم ها را قبل پیاده سازی و تبدیل به برنامه کامپیوتری می‌خواهیم بررسی نماییم. لذا از روش های دیگری باید استفاده کنیم.

روش دیگر بررسی دستورالعمل‌های موجود در الگوریتم و تجمیع زمان اجرای آنها می‌باشد. این روش از روش قبل دقت کمتری دارد ولی بدلیل هزینه بالا مناسب نمی باشد. برای ساده تر کردن کار محاسبه زمان اجرا همه دستورالعمل ها را به شکل واحدهای کاری مشابه در نظر گرفته تعداد واحد های کاری را شمارش می‌کنیم. این محاسبه دقت کمتری نسبت روش قبل دارد ولی همین دقت برای بررسی زمان اجرای الگوریتم‌ها مناسب می-باشد. در روش با توجه به دستورالعمل‌های موجود در الگوریتم تعداد واحد های کاری را شمارش می‌کنیم.

مثال : تعداد واحدهای کاری الگوریتم‌های ر=زیر را شمارش کنید.

```
int a , b , s ;          0
cin >> a >> b ;          2
s = a + b ;              1
cout << s;               1
```

مجموع تعداد واحهای کاری : 4

```
int   i , a , s=0 ;              1
for(i=1 ; i  <= 40 ; i ++)       41
{    cin>> a  ;                  40
```

```
    s = s + a ; }       40
cout << s;              1
```

مجموع تعداد واحهای کاری : 123

```
int   i,j , a , s=0 ;               1
for (i=1 ; i  <= 40 ; i ++)     41
        for (j=5 ; j  <= 20 ; j ++)    40* (20-5+1 +1)= 680
{    cin>> a  ;          40 * ( 20 -5 +1) = 640
    s = s + a ; }       40 * ( 20 -5 +1) = 640
cout << s;              1
```

مجموع تعداد واحهای کاری : 2003

تمرین:

```
int   i , a , s=0 ;
for(i=10 ; i  <= 40 ; i ++)
{    cin>> a  ;
    s = s + a ; }
for(i=10 ; i  <= 40 ; i ++)
   cout << i  ;
cout << s;
```

در بررسی زمان اجرای الگوریتم‌ها به دنبال کشف شرایطی هستیم که احتمال وقوع بحران در آنها وجود دارد. سیستم‌های پردازشی امروزه که قدرت پردازش بالایی دارند؛ در حالتی که تعداد واحدهای کاری ثابت بوده به

تعداد و نوع ورودی ها وابسته نمی‌باشد مشکلی پیدا نکرده در مدت زمان قابل قبولی الگوریتم را اجرا خواهند کرد. در حالتی که زمان اجرای الگوریتم به تعداد ورودی‌ها وابسته می‌باشد نیز با تعداد ورودی کم نیز بحرانی ایجاد نخواهد شد. احتمال وقوع بحران در شرایطی ایجاد می‌شود که تعداد ورودی‌ها به بی‌نهایت میل کند. مقدار بی‌نهایت به مسئله مورد نظر وابسته می‌باشد که در ادامه مورد بررسی قرار خواهد گرفت.

در مثال زیر تعداد واحدهای کاری به تعداد ورودی‌ها وابسته می‌باشد.

تابعی بنویسید که یک ارایه را دریافت کرده مجموع مقادیر موجود در آن را محاسبه نماید. سپس حاصل جمع را به مقدار برگشتی برگرداند. این الگوریتم را از نظر تعداد واحد کاری بررسی کنید.

```
int  f ( int *a , int n)              1
{
int s = 0;                            1
 for ( i= 0 ; i < n-1 ; i ++)         n-1-0+1+1 = n+1
        s += a[i];                    n-1-0+1 = n
return s ;                            0
}
```

تعداد واحد کاری: 2n+3

طبقه بندی زمان اجرای الگوریتم های مختلف :

۱- **زمان اجرای ثابت :** الگوریتم هایی که زمان اجرای آنها به تعداد ورودی ها وابسته نمی باشد. –تعداد واحدهای کاری به صورت یک مقدار ثابت $f(n) = c$

۲- **زمان لگاریتمی :** الگوریتم هایی که زمان اجرای آنها به صورت لگاریتمی به تعداد ورودی ها وابسته است. $f(n) = \log (n)$

۳- **زمان اجرای خطی** : زمان اجرا به صورت خطی به تعداد ورودی ها وابسته می باشد. به عبارت دیگر هر مقدار تعداد ورودی ها بیشتر شود به همان اندازه هم زمان اجرا افزایش خواهد یافت. $f(n) = n$

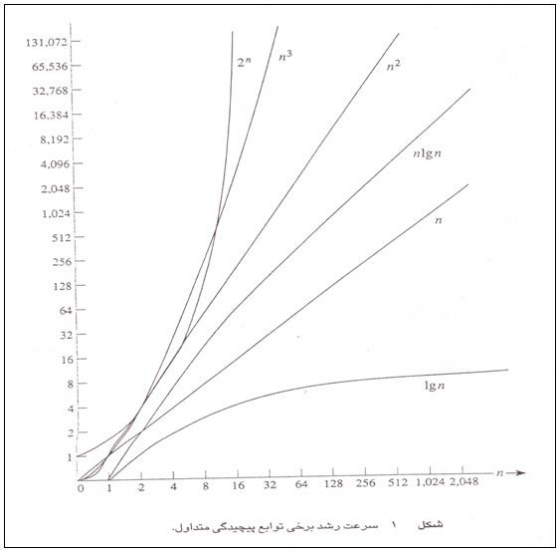
۴- **زمان اجرای n log n** : زمان اجرای الگوریتم به صورت $n \log n$ به تعداد ورودی ها وابسته است. $f(n) = n \log (n)$

۵- **زمان اجرای توانی** : زمان اجرای الگوریتم به صورت n^C به تعداد ورودی ها وابسته است. که در اینجا C یک مقدار ثابت همانند ۱،۲،۳،۴.... می باشد. $f(n) = n^c$

۶- **زمان اجرای نمایی** : زمان اجرای الگوریتم به صورت C^n به تعداد ورودی ها وابسته است. که در اینجا C یک مقدار ثابت همانند ۱،۲،۳،۴.... می باشد. $f(n) = C^n$

نکته : زمان اجرای لگاریتمی - هنگامی مرتبه اجرای قطعه کدی به صورت لگاریتمی خواهد شد که در آن متغییر کنترل کننده یک حلقه تکرار به صورت **ضرب** یا **تقسیم** تغییر کند.

مقایسه رشد زمان اجراهای مختلف با افزایش تعداد ورودی ها :



شکل ۱ سرعت رشد برخی توابع پیچیدگی متداول.

نماد های مجانبی :

ابزاری جهت مقایسه دو زمان اجرای مختلف با هم

۳ نماد را بررسی می کنیم و مورد استفاده قرار می دهیم.

نماد اول : نماد **O(n)** – که به صورت نماد بیگ او خوانده می شود.

$f(n) \in \boldsymbol{O}(g(n))$

تعریف ساده big - o : بیان اینک زمان اجرای یک تابع از یک مقداری کوچتر یا مساوی است.

یا به عبارتی تعیین یک سقف

تعریف ریاضی big – o :

$$\exists\, n_0, c > 0 \mid \forall\, n \geq n_0 \rightarrow f(n) \leq c\, g(n)$$

مثال : درستی عبارت زیر را اثبات کنید:

$$2n - 4 \in O(n^2)$$

$f(n) = 2n - 4$

$g(n) = n^2$

$n_0 = 1$, $c = 1$

$n >= 1$

$2n - 4 <= n^2$

$0 <= n^2 - (2n - 4)$ $\qquad n^2 - 2n + 4 = 0$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$a = 1$, $b = -2$, $c = 4$

$$x1, x2 = \frac{-(-2) \pm \sqrt{(-2)^2 - 4(1)(4)}}{2(1)}$$

به دلیل آنکه زیر رادیکال منفی می باشد معادله ریشه نخواهد داشت. به دلیل آنکه $a > 0$ می باشد همه نمودار بالای محور خواهد بود و عبارت همیشه مثبت خواهد شد. لذا نامعادله بدست آمده صحیح می‌باشد پس اثبات درستی عبارت مجانبی کامل می‌باشد.

مثال : درستی عبارت زیر را اثبات کنید:

$$9n + 4 \in O(n)$$

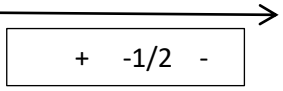
$$\exists\, n_0, c > 0 \mid \forall\, n \geq n_0 \rightarrow f(n) \leq c\, g(n)$$

$f(n) = 9n + 4$, $g(n) = n$

$n_0 = 1$, $c = 1$ - $n \geq 1 \rightarrow 9n + 4 \leq n$, $8n + 4 \leq 0$

تعیین علامت معادله $8n + 4 = 0$

$8n = -4$, $n = -\frac{1}{2}$



پس عبارت بدست آمده برای n های بزرگتر از ۱ منفی بوده نامعادله صحیح نمی‌باشد. لذا مقدار c را تغییر خواهیم داد تا ضریب تابع g افزایش یافته نامعادله اصلاح شود.

$c = 10 \rightarrow n >= 1 - 9n + 4 \leq 10n \rightarrow 0 \leq n - 4$

در اینجا عبارت n-4 برای n>1 باید بزرگتر از صفر باشد. بدلیل آنکه ریشه معادله ۴ می باشد. عبارت برای n>4 مثبت خواهد شد. پس مجبور هستیم n_0 را نیز تغییر دهیم تا مقادیر کوچکتر از ۴ نیز از نامعدله حذف شوند. لذا مقدار جدید $n_0=5$ رانظر میگیریم تا اثبات درستی عبارت مجانبی کامل شود.

نماد دوم : نماد **Ω(n)** – که به صورت نماد امگا خوانده می شود.

$$f(n) \in \boldsymbol{\Omega}(g(n))$$

تعریف ساده Ω : بیان اینک زمان اجرای یک تابع بزرگتر یا مساوی مقداری معین می‌باشد.

یا به عبارتی یک کف (مقدار حداقل) برای زمان اجرای الگوریتم تعیین می‌شود.

تعریف ریاضی Ω :

$$\exists\, n_0, c > 0 \mid \forall\, n \geq\ n_0\ \rightarrow g\ (n)\ \leq c\ f\ (n)$$

مثال : درستی یا نادرستی عبارت زیر را با بیان دلیل مشخص کنید.

$$2n + 4\ \in \Omega(n^2)$$

پاسخ :

نادرست - بدلیل آنکه عبارت سمت راست از مرتبه توانی می باشد نمی‌تواند یک کف برای عبارت سمت چپ که یک عبارت خطی است باشد.

مثال : درستی عبارت زیر را اثبات کنید:

$$n^2 + 4\ \in \Omega(5n)$$

پاسخ :

$f\ (\ n\ ) \in \boldsymbol{\Omega}\ (\ g\ (\ n\ )\ )$

$\exists\, n_0, c > 0 \mid \forall\, n \geq\ n_0\ \rightarrow g\ (n)\ \leq c\ f\ (n)$

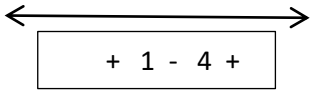
$f(n) = n^2 + 4$, $g(n) = 5n$

$n_0 = 1$, $c = 1$ - $n \geq 1 \rightarrow 5n \leq n^2 + 4$, $0 \leq n^2 - 5n + 4$

تعیین علامت معادله $n^2 - 5n + 4 = 0$

$$x1, x2 = \frac{-(-5) \pm \sqrt{(-5)^2 - 4(1)(4)}}{2(1)} = \frac{5 \pm 3}{2} = 1\ , 4$$

x1= 1 , x2 = 4



مقدار n_0 را تغییر می دهیم تا n های کوچکتر ۴ حذف شوند. حال عبارت برای $n_0 = 5$ صحیح بوده اثبات کامل خواهد شد.

نماد سوم : نماد **θ(n)** – که به صورت نماد تتا خوانده می شود.

$f\ (\ n\ ) \in \boldsymbol{\theta}\ (\ g\ (\ n\ )\ )$

تعریف ساده θ : بیان اینکه زمان اجرای یک تابع هم ارز مقداری معین می‌باشد.

یا به عبارت دیگر یک مقدار هم ارز برای زمان اجرای الگوریتم تعیین می‌شود.

سوال : آیا امکان دارد که دو عبارت زیر به صورت هم زمان صحیح باشند؟

- $f(n) \in O\ (\ g(n)\ )$
- $f(n) \in \Omega\ (\ g(n)\ )$

پاسخ : تنها در صورتی دو عبارت فوق به صورت هم زمان صحیح می باشند که دو تابع زمان اجرای f(n) و g(n) هم ارز هم باشند. که تعریف هم‌ارزی نیز بر همین اساس تعریف می شود.

$$\left.\begin{array}{l} f(n) \in O\ (\ g(n)\ ) \\ f(n) \in \Omega\ (\ g(n)\ ) \end{array}\right\} \Leftrightarrow\ f(n)\ \in\ \theta\ (\ g(n))$$

اگر و فقط اگر یک تابع زمان اجرا هم سقف و هم کف یک تابع زمان اجرای دیگر باشد این دو تابع هم‌ارز خواهند بود. که بر همین اساس تعریف ریاضی هم‌ارزی نیز با ترکیب تعریف ریاضی دو نماد O و Ω بیان خواهد شد.

تعریف ریاضی θ :

$$\exists\ n_0, c1, c2 > 0\ |\ \forall\ n \geq\ n_0\ \rightarrow c1\ f\ (n) \leq g\ (n)\ \leq c2\ f\ (n)$$

مثال : با استفاده از تعریف ریاضی هم ارزی درستی عبارت زیر را اثبات کنید.

$$n^2\ +\ 4n\ +\ 2\ \in\ \theta\ (\ n^2)$$

اثبات ریاضی در دو بخش باید انجام شود.

$$n_0 = 1\ ,\ c1 = 1 \quad \forall\ n \geq\ 1 \quad \rightarrow \quad n^2\ +\ 4n\ +\ 2\ \leq\ (\ n^2)$$

$$\rightarrow \quad 4n\ +\ 2\ \leq\ 0$$

اما عبارت فوق به ازای n>1 صحیح نمی باشد. به منظور بدست اوردن یک عبارت درست مقدار زیر C_1 را اصلاح خواهیم کرد.

$$n_0 = 1\ ,\ C_1 = 1/2\ |\ \forall\ n \geq\ 1 \quad \rightarrow \quad 2(n^2\ +\ 4n\ +\ 2) \leq\ (\ n^2\ )\ \rightarrow$$

الگوریتم های بازگشتی :

الگوریتمی بازگشتی می باشد که درآن یک تابع خودش را فراخوانی کند. جهت آشنایی با توابع بازگشتی تعدادی از این توابع رو بررسی خواهیم کرد.

تذکر : برای نوشتن یک تابع بازگشتی ابتدا باید یک ساختار بازگشتی در مسئله مورد نظر کشف نمود. سپس ساختار کشف شده را در تابع مورد نظر پیاده سازی نمود.

مثال : تابع بازگشتی بنویسید که دو مقدار صحیح را دریافت کرده حاصلضرب آنها را به صورت بازگشتی محاسبه کرده بازگشت دهد.

ساختار بازگشتی: تبدیل ضرب به چند عمل جمع تکراری

$$a \times b = (a + a + a + \cdots + a) = a + a \times (b - 1)$$

$$f(a, b) = a + f(a, b - 1) \qquad b > 1$$

برای خاتمه فراخوانی بازگشتی باید انتهای فراخوانی بازگشتی را با یک شرط مشخص نمود.

$$f(a, b) = a \qquad b = 1$$

حال با استفاده از ساختار بدست آمده تابع را می نویسیم.

```
int f ( int a , int b)
{ if ( b > 1 )
        return  ( a + f ( a , b – 1 ) ) ;
else
        return  a ;
}
```

مثال : تابع بازگشتی بنویسید که مقداری را دریافت کرده فاکتوریل آن را برگرداند.

ساختار باگشتی :

$$n! = 1 \times 2 \times 3 \times \ldots \times n - 1 \times n = n \times (n - 1)!$$

$$f(n) = n \times f(n - 1) \quad , \quad n > 1$$

$$f(n) = 1 \quad , \quad n = 1$$

تابع بازگشتی فاکتوریل :

```
int f ( int n )
{ if ( b > 1 )
        return  ( n + f (n – 1 ) ) ;
else
        return  n ;
}
```

تمرین : تابع بازگشتی بنویسید که دو مقدار صحیح را دریافت کرده a^b را به صورت بازگشتی محاسبه کرده بازگشت دهد.

ارزیابی زمان اجرای توابع بازگشتی:

به منظور ارزیابی زمان اجرای این تابع ها به روش های گذشته نمی‌توان عمل نمود. برای ارزیابی زمان اجرای تابع های بازگشتی ابتدا باید تابع زمان اجرا را به صورت بازگشتی بدست آوریم سپس با استفاده از روش‌هایی که در ادامه به بررسی آنها خواهیم پرداخت حل نماییم. حل رابطه بازگشتی به معنی تبدیل رابطه بازگشتی به یک رابطه غیر بازگشتی می باشد. (تذکر : برای محاسبه تابع بازگشتی با یک ورودی مشخص باید آن را تا رسیدن به شرط مرزی به صورت بازگشتی محاسبه نمود که هزینه زمانی بالایی را نیاز خواهد داشت اما محاسبه تابع غیربازگشتی به راحتی انجام می پذیرد.)

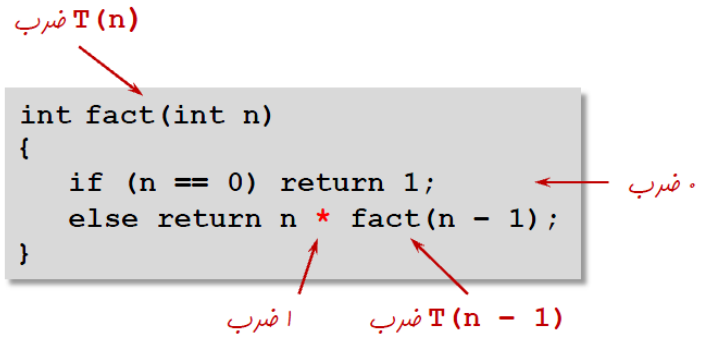
مراحل تحلیل یک الگوریتم بازگشتی

۱- تعین عمل اصلی

۲- محاسبه ی تعداد دفعات اجرای عمل اصلی به صورت یک تابع بازگشتی از اندازه ی ورودی

۳- حل رابطه ی بازگشتی به دست آمده

مثال :



نتیجه :

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1) + 1 & n \geq 1 \end{cases}$$

مثال :مرتب سازی ادغامی

```
void MergeSort ( int  *a, int  *aux, int  lo , int  hi )
{       if ( hi <= lo ) return ;
        int  mid = lo + ( hi - lo ) / 2 ;
        MergSort ( a , aux , lo , mid );                T ( n / 2 )
        MergSort ( a , aux , mid +1  , hi );            T ( n / 2 )
        merge ( a , aux , a , mid , hi);                 n - 1
}
```

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + n - 1 & n > 1 \end{cases}$$

حل روابط بازگشتی : تبدیل رابطه بازگشتی به رابطه رابطه غیربازگشتی

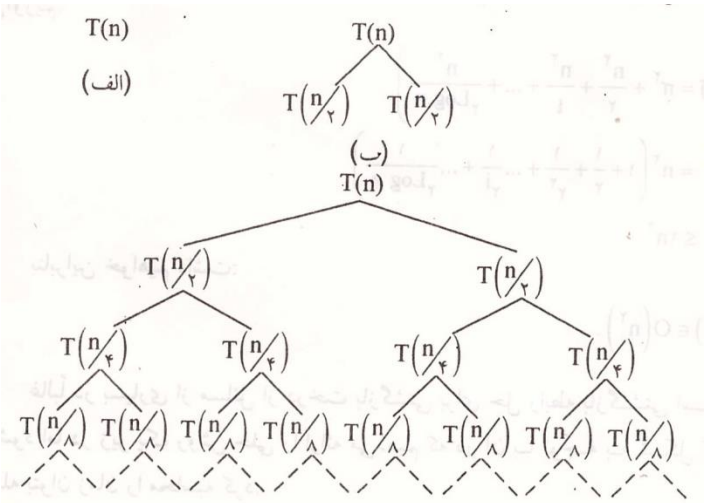
۱- درخت بازگشت : یکی از روش‌های خوب برای حل یا حدس روابط بازگشتی از طریق تکرار، استفاده از درخت بازگشت می‌باشد.

در این روش، درختی برای رابطه بازگشتی تشکیل می‌شود که در آن ریشه درخت، مقدار اولیه عبارت غیربازگشتی را دارا می باشد. در سطح بعدی درخت به تعداد جملات بازگشتی، گره ایجاد می‌شود که مقدار هر گره با توجه به درجه تقسیم، بر حسب مقدار n محاسبه می‌شود. سطح‌های درخت تا زمانی که n به مقدار ثابت نرسیده، تشکیل می‌شوند. برای محاسبه مقدار رابطه بازگشتی، نخست مجموع هر سطح درخت محاسبه می‌شود سپس مجموع کلی عبارت هر سطح را محاسبه می‌کنیم. با محاسبه این مقدار، جواب رابطه بازگشتی حاصل می‌شود.

توضیح این روش را با رابطه بازگشتی:

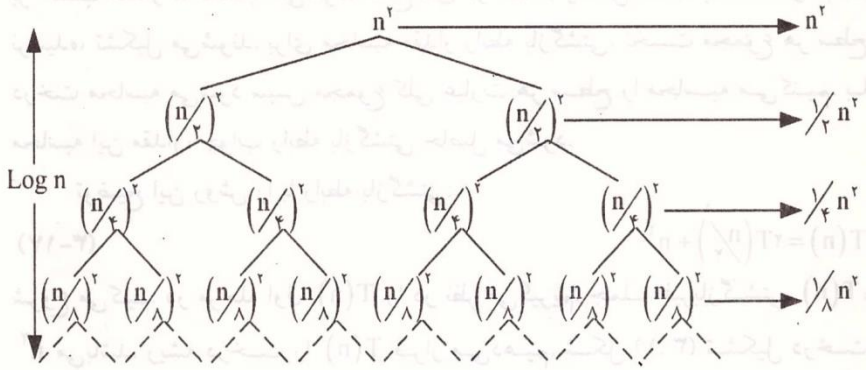
$$T(n) = ۲T\left(n/۲\right) + n^۲$$

شروع می‌کنیم. در مرحله اول $T(n)$ را در نظر می‌گیریم. جمله غیربازگشتی $T(n)$، $n^۲$ می‌باشد. ریشه درخت را $T(n)$ قرار می‌دهیم. شکل (۱. ۳) تشکیل درخت به‌صورت مرحله به مرحله را نمایش می‌دهد. در قسمت (الف) شکل، $T(n)$ نمایش داده شده است. در قسمت دوم با توجه به اینکه دو بازگشت $T\left(n/۲\right)$ در $T(n)$ وجود دارد یک درخت دودوئی ساخته می‌شود. با تکرار ساخت سطح‌های درخت شکل (ج) ساخته می‌شود.



مراحل تشکیل درخت بازگشتی برای $T(n)$

حال برای محاسبه مقدار رابطه بازگشتی با استفاده از درخت بازگشتی، مقدار جمله غیربازگشت در هر گره را محاسبه می‌کنیم. درخت شکل (۲-۳) حاصل این محاسبه می‌باشد.



درخت بازگشت رابطه بازگشتی $T(n)$

برای محاسبه مقدار رابطه بازگشتی، مجموع تمام سطح های درخت را بدست می‌آوریم:

$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots + \frac{n^2}{2^{\text{Log } n}}$$

$$= n^2\left(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots \frac{1}{2^i} + \ldots \frac{1}{2^{\text{Log } n}}\right)$$

$$\leq 2n^2$$

بنابراین خواهیم داشت:

$$T(n) \in O(n^2).$$

۲- جایگذاری مکرر :

این روش نیازمند یک حدس برای جواب نیست، بلکه با استفاده از جای‌گذاریهای متوالی می‌تواند، جواب مناسب را تولید کند. در این روش با توجه به خاصیت رابطه بازگشتی به ازای nهای مختلف (که در نهایت به یک مقدار ثابت می‌رسد) و جای‌گذاری آنها در هم جواب مسئله حاصل می‌شود.

مثال: رابطه بازگشتی زیر را حل کنید.

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1) + 1 & n \geq 1 \end{cases}$$

```
T(n) = T(n - 1) + 1
     = T(n - 2) + 1 + 1     = T(n - 2) + 2
     = T(n - 3) + 1 + 2     = T(n - 3) + 3
     = T(n - 4) + 1 + 3     = T(n - 4) + 4
                            ...
                            = T(n - i) + i
```

$$T(n-i) = T(0) \Rightarrow n - i = 0$$
$$\Rightarrow \underline{i = n}$$

$$T(n) = T(n-n) + n$$
$$= T(0) + n$$
$$= 0 + n$$
$$= n$$

مثال: رابطه بازگشتی زیر را حل کنید.

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + 1 & n > 1, n = 2^k \end{cases}$$

$$T(n) = T(n/2^1) + 1 = T(n/2^1) + 1$$
$$= T(n/2^2) + 1 + 1 = T(n/2^2) + 2$$
$$= T(n/2^3) + 1 + 1 + 1 = T(n/2^3) + 3$$
$$= T(n/2^4) + 1 + 1 + 1 + 1 = T(n/2^4) + 4$$
$$\dots$$
$$= T(n/2^i) + i$$

$$T(n/2^i) = T(1) \Rightarrow n/2^i = 1$$
$$\Rightarrow \underline{i = \lg n}$$

$$T(n) = T(1) + \lg n$$
$$= 1 + \lg n$$

تمرین : رابطه بازگشتی زیر را حل کنید.

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + n - 1 & n > 1, n = 2^k \end{cases}$$

۳- حدس و استقراء :

- حدس جواب با استفاده از چند جمله ی اولیه
- اثبات حدس به وسیله استقرا

استقرا شامل دو مرحله می باشد. ۱- اثبات پایه ۲- اثبات گام

برای پایه معمولا از شرایط مرزی مسئله استفاده می شود و برای اثبات گام رابطه بازگشتی با مقدار K را به عنوان فرض در نظر گرفته، رابطه بازگشتی با مقدار K+1 را به عنوان حکم اثبات می کنیم.

مثال : رابطه بازگشتی زیر را حل کنید.

$$T(n) = T(n-1) + 5 \qquad T(1) = 5$$

حدس : ابتدا با بدست آوردن رابطه بازگشتی برای مقداری مشخص سعی می کنیم که یک رابطه غیر بازگشتی را حدس بزنیم.

$$T(4) = T(3) + 5$$
$$T(3) = T(2) + 5$$
$$T(2) = T(1) + 5$$
$$T(1) = 5$$

T(4) = 20

T(3) = 15

T(2) = 10

T (n) = 5 n

حالل برای تشکیل استقرا ابتدا برقرار بودن پایه ثابت می کنیم. معمولا شرایط مرزی را به عنوان پایه استفاده می کنیم.

T(1) = 5 (1) = 5

پس از اثبات پایه فرض و حکم را تشکیل می دهیم و آن را اثبات می کنیم.

فرض : T (k) = 5 k

حکم : T (k +1) = 5 (k + 1)

اثبات حکم : T (k + 1) = T (k+1 – 1) + 5 = T (k) + 5

استفاده از فرض $= 5\,k + 5 \quad = 5\,(k+1)$

مثال: رابطه بازگشتی زیر را حل کنید.

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1) + 1 & n \geq 1 \end{cases}$$

حدس

```
T(1) = T(0) + 1 = 0 + 1 = 1
T(2) = T(1) + 1 = 1 + 1 = 2
T(3) = T(2) + 1 = 2 + 1 = 3
T(4) = T(3) + 1 = 3 + 1 = 4
...
T(n) = n
```

فرض $T(k) = T(k-1) + 1$

حکم $T(k+1) = T(k) + 1$

استقرا

پایه به ازای $n = 0$ برقرار است.

```
T(0) = 0
```

گام استقرا.

```
T(n + 1) = T(n) + 1 = n + 1
```

مثال: رابطه بازگشتی زیر را حل کنید.

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + 1 & n > 1 \end{cases}$$

حدس

```
T(2) = 2T(1) + 1 =  2 + 1 =  3
T(3) = 2T(2) + 1 =  6 + 1 =  7
T(4) = 2T(3) + 1 = 14 + 1 = 15
T(5) = 2T(4) + 1 = 30 + 1 = 31
...
T(n) = 2^n - 1
```

استقرا

پایه به ازای $n = 1$ برقرار است.

$$T(1) = 2^1 - 1 = 1$$

گام استقرا.

$$\begin{aligned} T(n+1) &= 2T(n) + 1 \\ &= 2(2^n - 1) + 1 \\ &= 2^{n+1} - 2 + 1 \\ &= 2^{n+1} - 1 \end{aligned}$$

۴- حل رابطه های بازگشتی همگن :

تفاوت میان رابطه‌های همگن و غیرهمگن :

رابطه بازگشتی زیر را در نظر بگیرید.

$$t(n) = C_1t(n-1) + C_2t(n-2) + C_3\, t(n-3) + \ldots + Cmt(n-m) + f(n)$$

اگر بخش غیر بازگشتی (f(n)) در رابطه بازگشتی وجود نداشته باشد (f(n)=0) عبارت را همگن می نامیم.
ولی اگر f(n) هر چیزی غیر صفر باشد عبارت غیر همگن نامیده می شود.

برای حل عبارت های بازگشتی همگن ابتدا تمام عبارت را به یک سمت انتقال می دهیم.

$$t(n) - C_1t(n-1) - C_2t(n-2) - C_3\, t(n-3) - \ldots - Cmt(n-m) = 0$$

سپس به جای عبارت های بازگشتی توانی از r را قرار می دهیم.

$r^{(n)} - C_1 r^{(n-1)} - C_2^{r(n-2)} - C_3 r^{(n-3)} - \dots - Cmr^{(n-m)} = 0$

در ادامه از کوچکترین توان r فاکتور خواهیم گرفت.

$r^{(n-m)} (r^{(m)} - C_1 r^{(m-1)} - C_2^{r(m-2)} - C_3 r^{(m-3)} - \dots - Cm) = 0$

قسمت داخل پرانتز را برابر صفر قرار داده ریشه های معادله را به دست می آوریم. این معادله معادله مشخصه رابطه بازگشتی نامیده می شود.

$r^{(m)} - C_1 r^{(m-1)} - C_2^{r(m-2)} - C_3 r^{(m-3)} - \dots - Cm = 0$

اگر ریشه های معادله را به صورت $r_1, r_2, .., r_m$ در نظر بگیریم. رابطه غیر بازگشتی زیر جواب مساله خواهد بود.

$t(n) = d_1 (r_1^n) + d_2 (r_2^n) + \dots + d_1 (r_1^n)$

که در اینجا d ها نیز مقادیر ثابت می باشند. حال با جایگزاری شرایط مرزی مقادیر ثابت را بدست آورده جواب نهایی را شکل خواهیم داد.

مثال : رابطه بازگشتی همگن زیر را حل کنید.

$t(n) = 3t(n-1) + 4t(n-2) \qquad t(0) = 0 , \ t(1) = 1$

برای حل روش فوق را اعمال می کنیم.

$t(n) - 3t(n-1) - 4t(n-2) = 0$

$r^{(n)} - 3r^{(n-1)} - 4r^{(n-2)} = 0$

$r^{(n-2)} (r^{(2)} - 3r^{(1)} - 4) = 0$

$r^{(2)} - 3r^{(1)} - 4 = 0 \qquad r = \frac{-(-3) \pm \sqrt{(-3)^2 - 4(1)(-4)}}{2(1)} = \frac{3 \pm \sqrt{9+16}}{2} = \frac{3 \pm 5}{2} = -1 , 4$

$t(n) = d_1 (-1)^n + d_2 (4)^n$

$t(0) = d_1 (-1)^0 + d_2 (4)^0 = \boldsymbol{d_1 + d_2 = 0}$

$$t(1) = d_1 (-1)^1 + d_2 (4)^1 = -\boldsymbol{d_1} + \boldsymbol{4d_2} = \boldsymbol{1}$$

جمع دو عبارت : $5d_2 = 1 \quad \rightarrow \quad d_2 = {}^1\!/_5$

عبارت اول : $d_1 + {}^1\!/_5 = 0 \quad \rightarrow \quad d_1 = -{}^1\!/_5$

جواب نهایی : $t(n) = -{}^1\!/_5 (-1)^n + {}^1\!/_5 (4)^n$

تمرین : رابطه بازگشتی همگن زیر را حل کنید.

t(n) = - 2 t(n-1) - 3t(n-2) t(0) = 0 , t(1) = 1

۵- روش قضیه اصلی :

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^c) \qquad (a > 0, b > 1) \quad \Rightarrow \quad T(n) \in \begin{cases} \Theta\left(n^{\log_b a}\right) & \log_b a > c \\ \Theta(n^c \lg n) & \log_b a = c \\ \Theta(n^c) & \log_b a < c \end{cases}$$

$$T(n) = 4T(n/2) + \Theta(n^1) \overset{(1)}{\Rightarrow} T(n) \in \Theta(n^2) \qquad (a = 4, b = 2, c = 1) \Rightarrow \log_b a > c$$

$$T(n) = 4T(n/2) + \Theta(n^2) \overset{(2)}{\Rightarrow} T(n) \in \Theta(n^2 \lg n) \qquad (a = 4, b = 2, c = 2) \Rightarrow \log_b a = c$$

$$T(n) = 4T(n/2) + \Theta(n^3) \overset{(3)}{\Rightarrow} T(n) \in \Theta(n^3) \qquad (a = 4, b = 2, c = 3) \Rightarrow \log_b a < c$$

روش تقسیم و غلبه

یکی از روش‌های پرکاربرد و محبوب برای طراحی الگوریتم‌ها روش Divide and Conquer است که در زبان فارسی به صورت تقسیم و حل یا تقسیم و غلبه ترجمه شده است.

در این روش، داده‌ها به دو یا چند دسته تقسیم شده و حل می‌شوند. سپس با ترکیب مناسب نتایج به دست آمده از این زیرمسئله‌ها، مسئله‌ی اصلی حل می‌شود. در صورتی که زیرمسئله خود به اندازه‌ی کافی بزرگ باشد،

می‌توان از همین روش برای حل آن استفاده کرد. تقسیمات متوالی زیرمسئله‌ها تا جایی ادامه پیدا می‌کند که به اندازه‌ی کافی کوچک شده باشند و بتوان آنها را با روش‌های دیگر به راحتی حل نمود.

برای آشنایی بیشتر، چند الگوریتم که با روش حل و تقسیم پیاده‌سازی شده‌اند معرفی می‌شوند.

مرتب‌سازی سریع (Quick Sort)

در روش مرتب‌سازی سریع داده‌های ورودی را بر حسب یکی از عناصر به نام محور به دو قسمت نه لزوما هم‌اندازه تقسیم کرده و هر کدام را جداگانه مرتب می‌کنیم.

مرتب‌سازی ادغامی(Merge Sort)

در این روش نیز همانند روش مرتب‌سازی سریع داده‌ها به دو قسمت تقسیم می‌شوند .اما روش تقسیم کردن داده‌ها متفاوت است .طوری که قسمت‌های به دست آمده از تقسیم تقریبا هم‌اندازه هستند .پس از مرتب کردن زیر آرایه‌ها، با ادغام آنها آرایه‌ی اصلی به صورت مرتب‌شده حاصل می‌شود.

ضرب استراسن

روش ضرب استراسن برای بهینه کردن عمل ضرب ماتریس‌ها توسط شخصی به نام استراسن معرفی شده است . در این روش هر کدام از ماتریس‌ها به چهار زیرماتریس تقسیم شده و عملیات ضرب با استفاده از آنها و رابطه‌هایی که استراسن عنوان کرده انجام می‌شود .با استفاده از این روش مرتبه‌ی اجرایی ضرب ماتریس از $O(n3)$ به $O(n2.8)$ کاهش پیدا می‌کند که در ماتریس‌هایی با ابعاد بزرگ منجر به افزایش سرعت چشمگیری می‌شود.

ضرب چندجمله‌ای‌ها و ضرب اعداد بسیار بزرگ

با استفاده از روش تقسیم و حل می‌توان روشی بهینه‌تر از ضرب عادی چندجمله‌ای‌ها برای آنها تعریف کرد .در این روش چندجمله‌ای‌ها به دو قسمت تقسیم شده و با استفاده از یک سری روابط، ضرب و جمع شده و نتیجه‌ی نهایی را می‌دهند .از همین روش با اندکی تغییر برای ضرب اعداد بسیار بزرگ هم می‌توان استفاده کرد که با اعمال آن، مرتبه‌ی ضرب از $O(n2)$ به $O(n1.58)$ کاهش پیدا می‌کند.

مسئله‌ی کاشی‌کاری (فرش کردن صفحه‌ی شطرنجی با موزاییک‌های Lشکل)

مسئله‌ی کاشی‌کاری از مسائل جالب طراحی الگوریتم است. در این مسئله از شما خواسته می‌شود سطح یک قطعه زمین را که به صورت صفحه‌ی شطرنج شبکه‌بندی شده است با استفاده از کاشی‌هایی با شکل Lبپوشانید . به طوری که خانه‌ی خاصی از این شبکه خالی بماند. می‌توان فرض کرد خانه‌ی خالی برای باغچه یا حوضچه‌ی کوچکی کنار گذاشته شده است.

مسئله‌ی تنظیم جدول مسابقات (تورنمنت)

شما را مسئول تهیه‌ی جدول مسابقات تیم‌های فوتبال محله کرده‌اند !این بازی‌ها به صورت دوره‌ای بوده و هر تیمی باید با تمام تیم‌های دیگر بازی کند. در ضمن هر تیم در روز تنها می‌تواند یک بازی انجام دهد .برای تهیه‌ی جدول مسابقات به صورت بهینه -که مسابقات در کوتاهترین زمان ممکن برگزار شود -الگوریتمی ابداع شده است که از روش تقسیم و حل استفاده می‌کند.

جستجوی دودویی (Binary Search)

برای یافتن عنصری در یک آرایه‌ی نامرتب چاره‌ای نداریم جز این که از روش جستجوی خطی استفاده کنیم .در این روش از ابتدای آرایه شروع کرده و تک‌تک عناصر را بررسی می‌کنیم، تا زمانی که به عنصر دلخواه برسیم . بنابراین اگر آرایه‌ی مورد نظر n عنصر داشته باشد، مرتبه‌ی اجرایی روش جستجوی خطی $O(n)$ خواهد بود .اما در مورد آرایه‌ی مرتب روش دیگری نیز وجود دارد :روش جستجوی دودویی.

در جستجوی دودویی عنصر وسط آرایه را بررسی می‌کنیم .در صورتی که این عنصر همان عنصر مورد نظر باشد، جستجو تمام می‌شود .اگر نه، باید قسمت‌های راست و چپ عنصر وسط را که تقریبا به یک اندازه هستند مورد جستجو قرار دهیم .اما آیا لازم است هر دو سمت جستجو شود؟ چون آرایه مرتب شده است، عناصر کوچکتر از عنصر مرکز همگی در سمت چپ و عناصر بزرگتر در سمت راست آن قرار دارند)آرایه را مرتب شده‌ی صعودی در نظر گرفته‌ام .(با توجه به این مسئله که عنصر مورد نظر ما کوچکتر از عنصر مرکزی آرایه است یا بزرگتر، تنها یکی از دو زیر آرایه را بررسی می‌کنیم .در نتیجه در هر مرحله بازه‌ی مورد جستجو نصف می‌شود.

برای یافتن مرتبه‌ی اجرایی الگوریتم به این صورت عمل می‌کنیم : فرض کنید $T(n)$ حداکثر تعداد مقایسه‌های لازم برای یافتن عنصر مورد نظرمان در میان nداده باشد .در روش جستجوی خطی $T(n)=n$ بود .در روش

جستجوی دودویی ابتدا عنصر وسط را با عنصر مورد نظرمان مقایسه می‌کنیم .اگر عنصر یافت نشود، در مرحله‌ی بعد تنها یکی از دو زیر آرایه بررسی می‌شود که نصف کل عناصر را دارد .پس می‌توان نوشت:

$$T(n)=T(n2)+1$$

با حل این رابطه‌ی بازگشتی به عبارت زیر می‌رسیم:

$$T(n)=[log2n]+1$$

که در آن منظور از []علامت جزء صحیح است .مرتبه‌ی اجرایی این الگوریتم $O(logn)$ است که در مقایسه با $O(n)$ در جستجوی خطی بسیار کارآمد است .به عنوان نمونه، برای یافتن عنصری در یک آرایه به طول یک میلیون، در بدترین حالت با استفاده از جستجوی خطی یک میلیون مقایسه و در جستجوی دودویی تنها بیست مقایسه نیاز است.

نکته : ممکن است این سوال مطرح شود که هزینه‌ی مرتب کردن یک آرایه به مراتب بیشتر از هزینه‌ی جستجوی خطی است. چه لزومی به انجام این کار است؟ لیست قبول‌شدگان کنکور را در نظر بگیرید .این لیست پس از آماده شدن تنها یک بار بر اساس شماره‌ی داوطلبی مرتب شده و در سایت سازمان سنجش قرار می‌گیرد . پس از آن، طی چند روز میلیون‌ها مراجعه و جستجو در لیست برای مشاهده‌ی نتیجه صورت می‌گیرد .در این حالت مطمئنا هزینه‌ی چند میلیون جستجویی که انجام می‌شود، بسیار مهم‌تر از هزینه‌ی یک بار مرتب‌سازی آن است .بیشتر کاربردهای این روش جستجو هم در همین نوع داده‌ها است.

پیاده‌سازی الگوریتم‌های تقسیم و حل

در اکثر مواقع می‌توان الگوریتم‌های طراحی شده توسط روش تقسیم و حل را به صورت بازگشتی پیاده‌سازی کرد .چرا که هر کدام از زیرمسئله‌ها یک مسئله از همان نوع با پارامترهای متفاوت هستند.

در مورد جستجوی دودویی، پیاده‌سازی بازگشتی روش به زبان برنامه‌نویسی C++ به این ترتیب است :

```
1    int BinarySearch_1(int arr[], int left, int right, int x){
2     if(left > right){
```

```
3       return -1;
4     }
5     int m = (left + right) / 2, result;
6     if(arr[m] == x)
7     {
8       result = m;
9     }
10    else if(arr[m] > x){
11      result = BinarySearch_1(arr, left, m - 1, x);
12    }
13    else{
14      result = BinarySearch_1(arr, m + 1, right , x);
15    }
16    return result;
17  }
```

تابع فوق چهار پارامتر arr (آرایه‌ای که جستجو در آن انجام می‌شود) ، leftو right (مرزهای محدوده‌ای که جستجو باید در آن انجام شود) و x (عنصری که باید پیدا کنیم) را دریافت کرده و اندیس محلی که عنصر مورد نظر یافت شده بر می‌گرداند به عنوان مثال اگر به دنبال عدد 78در آرایه‌ی arr به طول هزار هستیم، باید تابع را به صورت زیر فراخوانی کنیم:

```
i = BinarySearch_1(arr, 0, 999, 78);
```

مقدار **i** پس از بازگشت از تابع برابر اندیس محل عنصر 78 در آرایه خواهد بود .اگر این عنصر یافت نشود مقدار ۱- بازگشت داده می‌شود.

روش جستجوی دودویی پیاده‌سازی غیربازگشتی ساده‌تری هم دارد:

```
1    int BinarySearch_2(int arr[], int n, int x)
2    {
3     int left = 0, right = n - 1, m;
4     while(left <= right){
5      m = (left + right) / 2;
6      if (arr[m] == x){
7       return m;
8      }
9      if(arr[m] > x){
10      right = m - 1;
11     }
12     else{
13      left = m + 1;
14     }
```

```
15    }
16    return -1;
17  }
```

که در آن nتعداد عناصر آرایه است .مسلما سرعت اجرای این روش بیشتر از روش بازگشتی آن است .اما همیشه اینگونه نیست که پیاده‌سازی غیربازگشتی یک تابع سریعتر از پیاده‌سازی بازگشتی آن باشد.

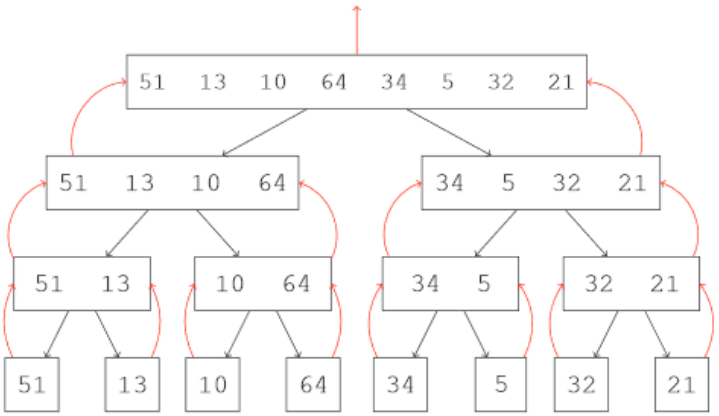
نکته : در معرفی روش تقسیم و حل عنوان کردم که اگر زیرمسئله به اندازه‌ی کافی کوچک باشد از این روش برای حل آن استفاده نمی‌کنیم .تعیین این اندازه -که به آن مقدار آستانه گویند -بر اساس روش طراحی الگوریتم و همینطور روش‌های دیگر موجود صورت می‌گیرد که بحث مفصلی را می‌طلبد.

نکته : زمانی که مسئله را به چند زیرمسئله تقسیم می‌کنیم، اگر تقسیم طوری باشد که هر زیرمسئله خودش نزدیک به nورودی داشته باشد، الگوریتم کارا نخواهد بود .نمونه‌ی چنین مسائلی محاسبه‌ی بازگشتی جمله‌ی nام دنباله‌ی فیبوناتچی است.

مرتب‌سازی ادغامی

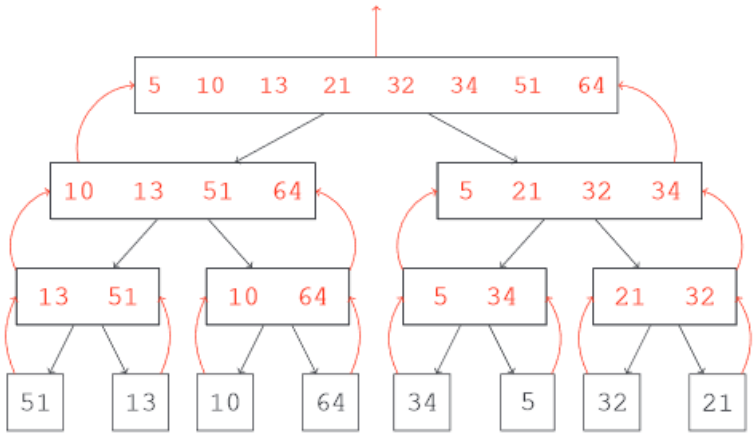
مرتب‌سازی ادغامی نوعی از الگوریتم‌های مرتب‌سازی (sort) است. طرز کار این الگوریتم تقریباً به صورت زیر است:

- توالی n عدد را به دو نیمه تقسیم می‌کنیم.
- به طور بازگشتی دو نیمه را مرتب می‌کنیم.
- دو نیمه مرتب شده را دریک توالی مرتب منفرد ادغام می‌کنیم.



در تصویر فوق ما ۸ عدد را به اعداد منفرد تقسیم کردیم. زمانی که این کار صورت گرفت، فرایند مرتب‌سازی را آغاز می‌کنیم.

عدد ۵۱ را با ۱۳ مقایسه می‌کنیم و عدد کوچک‌تر را در سمت چپ و عدد بزرگ‌تر را در سمت راست قرار می‌دهیم. این کار را برای اعداد ۱۰ و ۶۴، ۳۴ و 5، 32و ۲۱ نیز تکرار می‌کنیم.



ما این کار را برای سطوح بعد نیز انجام می‌دهیم تا به یک لیست کاملاً مرتب ادغام شده در سطح فوقانی دست یابیم. در سطح دوم از بالا یعنی زمانی که 2نیمه هر کدام با ۴ عدد داریم، ۲ اشاره‌گر ایجاد می‌کنیم. اشاره‌گر شماره 1به عدد ۱۰ و اشاره‌گر شماره ۲ به عدد ۵ اشاره می‌کند.

در ادامه عدد ۱۰ و ۵ مقایسه می‌شود و چون ۵ کوچک‌تر از ۱۰ است، ۵ به عنوان کوچک‌ترین عنصر در ابتدای لیست قرار می‌گیرد.

سپس عدد ۱۰ و ۲۱ مقایسه می‌شوند و چون ۱۰ از ۲۱ کوچک‌تر است و از آنجا که می دانیم ۱۰ کوچک‌ترین آیتم در لیست اول است (این اطلاع از فرایند مرتب‌سازی و ادغام ما در مراحل قبلی ناشی می‌شود) از این رو ۱۰ در ابتدای لیست و پس از ۵ قرار می‌گیرد. این کار برای همه اعداد موجود انجام می‌یابد تا این که به یک لیست کاملاً مرتب منفرد دست پیدا کنیم.

لازم به ذکر است که ما ۲ لیست از اعداد نامرتب را با هم مقایسه نمی‌کنیم؛ بلکه ما ۲ لیست از اعداد مرتب را با هم مقایسه می‌کنیم. ما می دانیم که ۱۰ در لیست سمت چپ کوچک‌ترین عدد است. ما می‌دانیم که ۱۳ دومین عدد کوچک مرتب شده در لیست سمت چپ است.

```
Algorithm Merge_Sort(list)
if n > 1
   copy A[1...n/2] to B[1...n/2]
   copy A[(n/2 + 1)...n] to C[1...n/2]
   merge_sort(b)
   merge_sort(c)
   merge(b, c, a)
end
```

معیار ما در این کد n است. اگر n بزرگ‌تر از ۱ باشد در این صورت باید مسئله را باز هم تجزیه کرده و حل کنیم. اگر n برابر با ۱ باشد، اگر ۱ آیتم در آرایه باشد در این صورت آرایه هم اینک مرتب است.

	p=4						q=4			
B:	10	13	51	64		C:	5	21	32	34

	i	*j*	*k*	*A*[]
before loop	1	1	1	empty
end of 1st iteration	1	2	2	5
end of 2nd iteration	2	2	3	5 10
end of 3rd	3	2	4	5 10 13
end of 4th	3	3	5	5 10 13 21
end of 5th	3	4	6	5 10 13 21 32
end of 6th	3	5	7	5 10 13 21 32 34
after if-else				5 10 13 21 32 34 51 64

به جدول اثر فوق توجه کنید ما در این جدول ۲ لیست داریم که هر یک شامل ۴ آیتم هستند. این لیست‌ها مرتب‌سازی شده‌اند. ما می‌خواهیم آن‌ها را ادغام کنیم i .یک اشاره‌گر در لیست اول یعنی B است j .یک اشاره‌گر در لیست دوم یعنی C است K .همان-k امین عنصر در لیست مرتب‌سازی شده جدید است.

دقت کنید که چگونه عناصر را مانند مثال‌های قبلی این نوشته با هم مقایسه می‌کنیم. هر گره به زمانی برابر با O(p) نیاز دارد که p تعداد عناصر موجود در لیست است.

هر سطح به زمانی برابر با O(n) نیاز دارد، زیرا تعداد کل اعداد صحیحی در هر سطح برابر با n است. در کل O(log n) سطح وجود دارد و از این رو زمان کلی برای اجرای این الگوریتم برابر با O(n log n) است

اعداد فیبوناچی

اعداد فیبوناچی را می‌توان در طبیعت مشاهده کرد. این اعداد از ۱ آغاز می‌شوند و عدد بعدی برابر با عدد کنونی + عدد قبلی است. در این مورد عدد دوم به صورت ۲ = ۱ + ۱ است. سپس عدد سوم به صورت ۳ = ۱ + ۲ و همین طور ۵ = ۲ 3 +است و این روند تا به آخر ادامه می‌یابد.

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

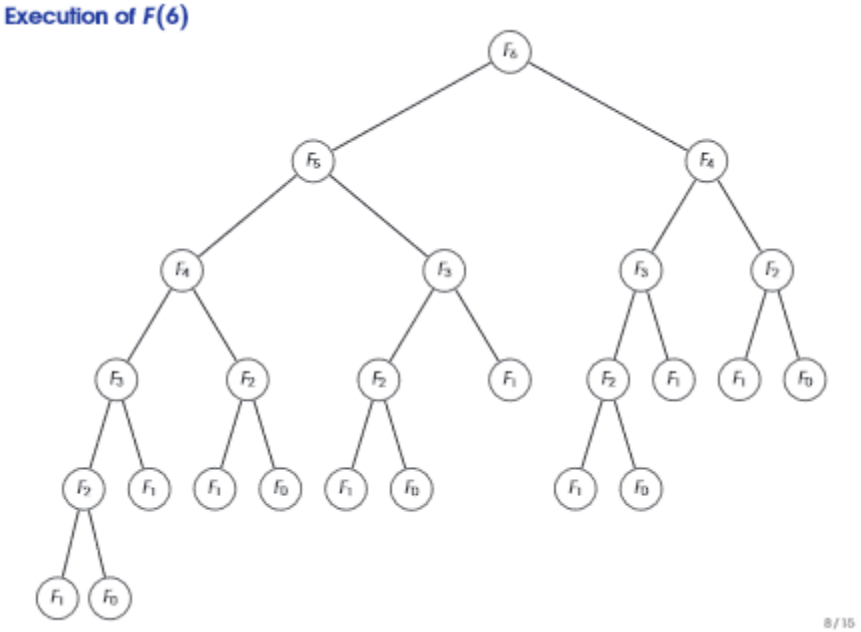
این تعریف رسمی اعداد فیبوناچی است. اگر n=0 یا n=1 باشد، خروجی ۱ خواهد بود و در غیر این صورت مجموع اعداد قبلی با عدد کنونی جمع می‌شود.

n	0	1	2	3	4	5	6	7	8	9	10
$F(n)$	1	1	2	3	5	8	13	21	34	55	89

جدول محاسبه F(n)

الگوریتم محاسبه اعداد فیبوناچی به صورت زیر است:

```
Algorithm F(n)
if ((n == 0) or (n == 1)) then
  return 1
else
  return F(n-1)+F(n-2)
```



در مثال فوق N برابر با ۶ است. از آنجا که N بزرگ‌تر از ۰ یا ۱ است، آن بخش را نادیده می‌گیریم. سپس F(5) +F(4)را محاسبه می‌کنیم F(5) .به صورت جمع F(4) + F(2) محاسبه می‌شود.

در نهایت وقتی به حالت‌های پایه برسیم که F برابر با ۰ یا ۱ باشد، اعداد 1 را خواهیم داشت. در هر مرحله عدد قبلی را به مرحله بالاتر بازگشت می‌دهیم. بنابراین محاسبه زیر را خواهیم داشت:

$F(1) + F(0) = 2$

و سپس

$F2 + F1 = 2 + 1 = 3$

روش حریصانه

روش حریصانه (Greedy)یکی از روش‌های مشهور و پرکاربرد طراحی الگوریتم‌ها است که با ساختاری ساده در حل بسیاری از مسائل استفاده می‌شود .این روش اغلب در حل مسائل بهینه‌سازی مورد استفاده است. در حالت کلی این روش سرعت و مرتبه‌ی اجرایی بهتری نسبت به روش‌های مشابه خود دارد؛ اما متناسب با مسئله ممکن است به یک جواب بهینه‌ی سراسری ختم نشود.

در روش حریصانه رسیدن به هدف در هر گام مستقل از گام قبلی و بعدی است. یعنی در هر مرحله برای رسیدن به هدف نهایی، مستقل از این که در مراحل قبلی چه انتخاب‌هایی صورت گرفته و انتخاب فعلی ممکن است چه انتخاب‌هایی در پی داشته باشد، انتخابی که در ظاهر بهترین انتخاب ممکن است صورت می‌پذیرد. به همین دلیل است که به این روش، روش حریصانه گفته می‌شود. زمانی که یک دزد عجول و حریص وارد خانه‌ای می‌شود، در مسیر حرکت خود هر وسیله و کالای با ارزشی را داخل کیسه می‌اندازد. وی در این حالت چندان توجهی نمی‌کند که چه اشیائی را قبلا برداشته و ممکن است در آینده چه اشیاء گرانبهاتری به دست آورد. او در هر گام تنها از بین اشیاء دم دست خود با ارزش‌ترین آن را انتخاب کرده و به وسایل قبلی اضافه می‌کند.

این روش کاربردهای عمومی دیگری نیز دارد. زمانی که در مقابل خرید از یک فروشگاه یک اسکناس تحویل فروشنده داده می‌شود، وی با یک حساب سرانگشتی سعی می‌کند با حداقل اسکناس‌ها و سکه‌های ممکن باقیمانده‌ی پول را تولید کرده و به خریدار تحویل دهد .این حساب سرانگشتی ممکن است روش حریصانه باشد.

ساختار روش حریصانه

کلیت روش حریصانه در هر مرحله، انتخاب یک عنصر از عناصر موجود است. این عنصر قسمتی از جواب مسئله است که به مجموعه عناصر نهایی اضافه می‌شود. در طی این مسیر گام‌های زیر اتفاق می‌افتد:

۱- **روال انتخاب حریصانه**: در این گام یک عنصر برای اضافه شدن به مجموعه جواب انتخاب می‌شود. معیار یا روال انتخاب عنصر برای اضافه شدن، ارزش آن عنصر است .بستگی به نوع مسئله هر عنصر ارزشی دارد که با ارزشترین آنها انتخاب می‌شود.

۲- **امکان‌سنجی و افزودن** :پس از انتخاب یک عنصر به صورت حریصانه، باید بررسی شود که آیا امکان اضافه کردن آن به مجموعه جواب‌های قبلی وجود دارد یا نه .گاهی اضافه شدن عنصر یکی از شرایط اولیه‌ی مسئله را نقض می‌کند که باید به آن توجه نمود .اگر اضافه کردن این عنصر هیچ شرطی را نقض نکند، عنصر اضافه خواهد شد؛ وگرنه کنار گذاشته شده و بر اساس گام اول عنصر دیگری برای اضافه شدن انتخاب می‌شود. اگر گزینه‌ی دیگری برای انتخاب وجود نداشته باشد، اجرای الگوریتم به اتمام می‌رسد.

۳- **بررسی اتمام الگوریتم** :در هر مرحله پس از اتمام گام 2و اضافه شدن یک عنصر جدید به مجموعه جواب، باید بررسی کنیم که آیا به یک جواب مطلوب رسیده‌ایم یا نه؟ اگر نرسیده باشیم به گام اول رفته و چرخه را در مراحل بعدی ادامه می‌دهیم.

به زبان ساده، در روش حریصانه طی هر مرحله یک عنصر به روش حریصانه به مجموعه جواب اضافه شده، شرط محدودیت‌ها بررسی شده و در صورت نبود مشکل، عنصر و عناصر بعدی به همین ترتیب به مجموعه جواب اضافه می‌شوند .در طی این گام‌ها اگر به یک شرط نهایی خاص برسیم، یا امکان انتخاب عنصر دیگری برای اضافه کردن به مجموعه جواب وجود نداشته باشد، الگوریتم پایان یافته و مجموعه جواب به دست آمده به عنوان جواب بهینه ارائه می‌شود .توجه داشته باشید که ممکن است بر اساس نوع مسئله، ترتیب اضافه شدن عناصر به مجموعه جواب اهمیت داشته باشد.

```
set_greedy(C){
  S = ∅;
  while(!solution(s) && C != ∅){
    X = select(C);
    C = C - {x};
    if (feasible(S,{x})){
      S = S + {x};
```

```
    }
  }
  if(solution(S)){
    return S;
  }
  else
    return Ø;
}
```

مسئله‌ی خرد کردن پول

مسئله‌ی خرد کردن پول از جمله مسائل کلاسیک طراحی الگوریتم‌ها است که مثال مناسبی برای شیوه‌ی عملکرد روش حریصانه است .هدف از این مسئله، تهیه کردن میزان مشخصی پول با حداقل استفاده از اسکناس‌ها و سکه‌های موجود است.

اگر از فروشگاهی ۱۴هزار تومان خرید کرده و یک اسکناس۵۰هزار تومانی تحویل فروشنده دهید، فروشنده باید مبلغ ۳۶ هزار تومان به شما بازگرداند .برای این کار روش‌های مختلفی وجود دارد .وی می‌تواند ۳۶ اسکناس هزار تومانی یا ۱۸ اسکناس دو هزار تومانی به شما بازگرداند؛ یا حتی ۷۲ اسکناس ۵۰۰ تومانی !به همین ترتیب ترکیبات مختلفی از اسکناس‌ها وجود دارد که مبلغ ۳۶ هزار تومان را تولید می‌کنند .در نهایت معمولا سه اسکناس ده هزار تومانی، یک اسکناس پنج هزار تومانی و یک اسکناس هزار تومانی چیزی است که شما از فروشنده دریافت می‌کنید .البته ممکن است فروشنده به دلیل نداشتن اسکناس هزار تومانی از دو اسکناس ۵۰۰تومانی استفاده کند. ولی در حال حاضر فرض بر این است که از همه‌ی اسکناس‌ها به اندازه‌ی کافی موجود است.

فروشنده چگونه پرداخت این اسکناس‌ها را محاسبه می‌کند؟ پر ارزش‌ترین اسکناس‌های موجود پنجاه هزار تومان ارزش دارند که برای مبلغ ۳۶ هزار تومان مناسب نیستند. پس آن اسکناس‌ها کنار گذاشته می‌شوند. در مرحله‌ی بعد اسکناس ده هزار تومانی بررسی می‌شود. چون ده هزار کوچکتر از ۳۶ هزار است، یک اسکناس ده

هزار تومانی انتخاب شده و برای پرداخت به مشتری کنار گذاشته می‌شود .در ادامه مبلغ ۲۶ هزار تومان باقیمانده است که باز هم می‌توان از اسکناس ده هزار تومانی استفاده کرد .پس فروشنده دو بار متوالی دیگر دو اسکناس ده هزار تومانی کنار می‌گذارد. به این ترتیب شش هزار تومان باقی می‌ماند که از ده هزار تومان کمتر است. پس انتخاب مجدد اسکناس ده هزار تومانی راه به جایی نمی‌برد و باید سراغ مبالغ کوچکتر رفت. اسکناس پنج هزار تومانی گزینه‌ی بعدی است و امکان انتخاب دارد. پس یک اسکناس از این نوع نیز انتخاب شده و تنها هزار تومان باقی می‌ماند. در این حالت اسکناس پنج هزار تومانی بزرگتر از مبلغ مورد نیاز است. پس انتخاب مجدد آن کنار گذاشته می‌شود. اسکناس بعدی دو هزار تومانی است که آن هم بزرگتر از هزار تومان است. از این اسکناس نیز صرف نظر کرده و نوبت به اسکناس بعدی با ارزش هزار تومان می‌رسد. در نهایت با یک اسکناس هزار تومانی مبلغ باقیمانده صفر شده و کل مبلغ مشتری آماده‌ی پرداخت است.

پیاده‌سازی مسئله‌ی خرد کردن پول به روش حریصانه

تابع زیر یک پیاده‌سازی از الگوریتم شرح داده شده به زبان برنامه‌نویسی C++ است:

```
1  int ChangeCoin(int Coins[], int Amount, int Result[]){
2      int i = 0, Count = 0, j = 0;
3      while(Coins[i] != -1 && Amount > 0){
4          if(Amount >= Coins[i]){
5              Amount -= Coins[i];
6              Result[j++] = Coins[i];
7              Count++;
8          }
9          else
```

```
10          i++;
11      }
12      if(Amount != 0)
13          return -1;
14      return Count;
15  }
```

در این تابع، Coins ارزش سکه‌ها و اسکناس‌های موجود و Amount مقدار مورد نیاز برای تولید را مشخص می‌کنند .پاسخ نهایی در آرایه Result قرار گرفته و تعداد آنها با متغیر Count به محل فراخوانی بازگشت داده می‌شود .فرض شده است انتهای لیست سکه‌ها و اسکناس‌ها با عدد ۱- مشخص شده و به ترتیب نزولی مرتب هستند .یعنی عنصر اول بزرگترین اسکناس موجود است.

اگر ارزش سکه‌ها به گونه‌ای باشد که نتوان مقدار خاصی را تولید کرد، عدد ۱- به عنوان تعداد سکه‌ها و اسکناس‌های یافت شده بازگردانده می‌شود. این حالت زمانی اتفاق می‌افتد که سکه یا اسکناس با ارزش یک واحد موجود نباشد. در غیر اینصورت هر مبلغی را حداقل با استفاده از همین سکه‌ها و اسکناس‌های یک واحدی می‌توان تولید کرد. مقدار بازگشتی تابع زمانی صفر می‌شود که مقدار Amount از همان ابتدا صفر بوده باشد.

اگرچه این روش عملکردی ساده دارد، اما همواره به جواب بهینه ختم نمی‌شود. مثلا اگر سکه‌هایی با ارزش ۱، ۷ و ۱۰ واحد پول موجود بوده و هدف تولید مبلغی با ارزش ۱۴ واحد باشد، بر اساس این روش ابتدا یک سکه با ارزش ۱۰ واحد انتخاب می‌شود .برای ۴ واحد باقیمانده چاره‌ای نیست جز اینکه چهار سکه‌ی یک واحدی انتخاب شود. پس در کل ۵ سکه برای تولید ۱۴ واحد انتخاب شد. این در حالی است که می‌شد تنها با انتخاب دو سکه‌ی ۷ واحدی همان مبلغ را تولید کرد.

برای این مسئله الگوریتم دیگری با استفاده از روش برنامه‌نویسی پویا ارائه شده است که همواره به یک جواب بهینه - در صورت وجود - می‌رسد.

کاربردهای روش حریصانه

از جمله کاربردهای مشهور روش حریصانه مسائلی هستند که در ادامه معرفی می‌شوند.

۱- **مسئله‌ی کوله‌پشتی کسری** : در این مسئله هدف پر کردن یک کوله‌پشتی از وسایل پر ارزشی است که وزن‌های مختلفی دارند .این کوله‌پشتی باید به نحوی پر شود که وزن بار آن از حد مجاز بیشتر نشده و ارزش وسایل داخل آن بیشینه باشد .در مسئله‌ی کوله‌پشتی کسری بر خلاف کوله‌پشتی صفر و یک این امکان وجود دارد که بتوان کسری از یک وسیله -مثل پارچه -را جدا کرده و به وسایل داخل کوله‌پشتی اضافه کرد.

۲- **تولید درخت پوشای کمینه** : روش‌های پریم و کروسکال دو روش مشهور تولید درخت پوشای کمینه از یک گراف وزن‌دار هستند که از روش حریصانه بهره می‌برند .منظور از درخت پوشای کمینه، درخت پوشایی از گراف است که مجموع وزن یال‌های آن کمتر یا مساوی مجموع وزن یال‌های سایر درخت‌های پوشای آن گراف است.

۳- **محاسبه‌ی کوتاهترین مسیرهای تک‌مبدأ** : زمانی که قصد داریم کوتاهترین مسیر از یک مبدأ مشخص به تمامی رئوس دیگر گراف را محاسبه کنیم، الگوریتمی مانند دیکسترا، با استفاده از روش حریصانه به کمک ما می‌آید.

۴-**کدگذاری و فشرده‌سازی اطلاعات**: کد هافمن یکی از روش‌های فشرده‌سازی اطلاعات است با کدگذاری مجدد کاراکترهای موجود در اطلاعات بر اساس میزان استفاده‌ی آنها، سعی در کم کردن حجم فایل می‌کند .بر اساس این روش، کاراکتری با استفاده‌ی بالا با کد کوتاهتر و کاراکتری با استفاده‌ی کم با کد طولانی‌تر جایگزین می‌شود.

لازم به ذکر است که موارد اشاره شده به عنوان کاربردهای روش حریصانه بر خلاف مثال خرد کردن پول به طور قطع یک جواب بهینه تولید می‌کنند .این جواب اگرچه ممکن است منحصر به فرد نباشد، اما همواره بهینه بوده و مرتبه‌ی اجرایی آن معمولا کمتر یا مساوی روش‌هایی مانند تقسیم و غلبه یا برنامه‌نویسی پویا است.

حل مساله کوله پشتی

هدف، قرار دادن این اشیا در کوله‌پشتی با ظرفیت W به صورتی است که مقدار ارزش بیشینه حاصل شود. به بیان دیگر، دو آرایه صحیح val[0..n-1] و wt[0..n-1] وجود دارند که به ترتیب نشانگر مقادیر و وزن‌های

تخصیص داده شده به n عنصر هستند. همچنین، یک عدد صحیح W نیز داده شده است که ظرفیت کوله پشتی را نشان می‌دهد. هدف، پیدا کردن زیرمجموعه‌ای با مقدار بیشینه val[] است که در آن، مجموع وزن‌ها کوچک‌تر یا مساوی W باشد. در حالت از مسئله امکان خورد کردن اشیا وجود ندارد و باید یک شی را به طور کامل انتخاب کرد و یا اصلا انتخاب نکرد. این گونه از مساله کوله پشتی را، «مساله کوله پشتی ۱-۰» می‌گویند. اما در حالت دیگر امکان انتخاب کسری از یک شی به منظور پر کردن ظرفیت باقیمانده وجود دارد که این حالت «مساله کوله پشتی کسری» نام دارد.

حل مسئله کوله پشتی کسری از طریق روش حریصانه امکان پذیر می باشد. در روش ابتدا اشیاء به ترتیب نزولی بر حسب ارزش به واحد وزن مرتب می‌کند. سپس از شی شماره ۱ (ارزشمندترین شی) شروع می‌کند. بیشترین تعداد ممکن از آن را در کوله‌پشتی قرار می‌دهد، تا زمانی که دیگر جای خالی‌ای برای آن نوع باقی نماند. آنگاه سراغ شی بعدی می‌رود. (در اینجا فرض شده که محدودیتی برای تعداد اشیا نداریم) اما اگر تعداد مجاز از هر شی محدود باشد، ممکن است خروجی این الگوریتم از پاسخ بهینه بسیار دور باشد.

الگوریتم کوله‌پشتی با استفاده از روش حریصانه

w : ظرفیت کوله پشتی - n : تعداد اشیاء

```
void Greedy Knapsack(w,n)
{
sort (p,w)
for(i=0;i<n;i++)
        x[i]=۰
```

X : آرایه ای که نشان دهنده انتخاب اشیاء می باشد. مقدار اولیه صفر به معنی فرض اولیه عدم انتخاب می باشد.

```
u=w;
```

w : ظرفیت اولیه کوله پشتی u: ظرفیت باقیمانده کوله پشتی

```
for(i=0;i<n;i++)
        if (W[i]>u);
```

انتخاب اشیاء به ترتیب ارزش (ارزش به وزن) – هنگامی که شئی در ظرفیت باقیمانده جا نشود روند انتخاب قطع خواهد شد.

```
                break ;
        else
```

انتخاب شئ i ام

```
                { x[i]=1;
                  u = u - w[i]
```

```
}
if(i<n)
   x[i]=u/wi ;
}
```

در صورتی که شئ انتخاب نشده ای باقیمانده باشد کسری(u/w_i) از آن انتخاب خواهد شد.

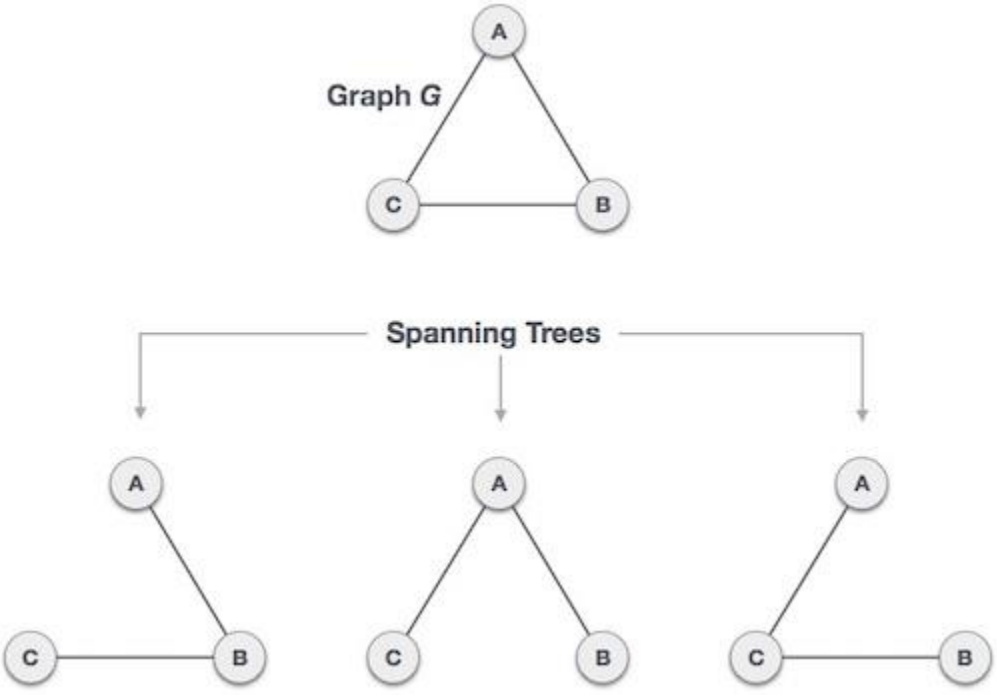
تحلیل پیچیدگی زمانی الگوریتم کوله پشتی کسری

با توجه به الگوریتم مشخص می شودمرتبه زمانی تحت تاثیر مرتبه زمانی مرتب سازی اشیاء به صورت نزولی می‌باشد. بنا بر این زمان الگوریتم $\theta(nlogn)$ خواهد بود.

تولید درخت پوشای کمینه :

درخت پوشا زیرمجموعه‌ای از گراف G است که همه رئوس آن با کمترین مقدار یال‌های ممکن پوشش یافته است. از این رو یک درخت پوشا دور ندارد و هیچ رأس ناهمبندی در آن دیده نمی‌شود.

بر اساس تعریف فوق می‌توانیم نتیجه بگیریم که هر گراف کاملاً همبند و غیر جهت‌دار G دست‌کم یک درخت پوشا دارد. یک گراف ناهمبند؛ هیچ درخت پوشایی ندارد، چون امکان پوشش همه رئوس آن میسر نیست.



درخت‌های پوشای فوق را در یک گراف کامل می‌توان یافت. یک گراف کامل غیر جهت‌دار می‌تواند بیشینه n^{n-2} درخت پوشا داشته باشد که n تعداد گره‌های آن است. در نمونه فوق n برابر با ۳ است و از این رو $3^{3-2} = 3$ درخت پوشا می‌توان در آن یافت.

مشخصات کلی درخت پوشا

اینک که دانستیم یک گراف می‌تواند بیش از یک درخت پوشا داشته باشد. در ادامه چند مورد از مشخصات درخت پوشای همبند با درخت G را بررسی می‌کنیم:

- یک گراف همبند G می‌تواند بیش از یک درخت پوشا داشته باشد.
- همه درخت‌های پوشای گراف G تعداد یکسانی از یال‌ها و رئوس را دارند.
- درخت پوشا هیچ دوری ندارد.
- با حذف یک یال از درخت پوشا، به گراف غیر همبند تبدیل می‌شود، یعنی درخت پوشا دارای کمینه اتصال‌های ممکن است.
- افزودن یک یال به درخت پوشا موجب ایجاد یک مدار یا طوقه می‌شود، یعنی درخت پوشا در حالت بیشینه غیر دوری (maximally acyclic) است.

مشخصات ریاضیاتی درخت پوشا

- درخت پوشا n-1 یال دارد که n تعداد گره‌ها (رأس‌ها) ی آن است.
- با حذف e-n+1 یال از یک گراف کامل می‌توانیم یک درخت پوشا بسازیم.
- یک گراف کامل می‌تواند در بیشینه حالت خود n^{n-2} عدد درخت پوشا داشته باشد.

از این رو می‌توانیم نتیجه بگیریم که درخت‌های پوشا زیرمجموعه‌ی از گراف همبند G هستند و گراف‌های ناهمبند، درخت پوشا ندارند.

کاربرد درخت پوشا

درخت پوشا اساساً برای یافتن کوتاه‌ترین مسیر بین همه گره‌ها در یک گراف مورد استفاده قرار می‌گیرد. کاربردهای رایج درخت‌های پوشا به صورت زیر هستند:

- برنامه‌ریزی شبکه تأسیسات شهری
- پروتکل مسیریابی شبکه رایانه‌ای
- تحلیل خوشه

با مثال کوچکی کاربردهای درخت پوشا را بررسی می‌کنیم. شبکه یک شهر را به صورت یک گراف بزرگ تصور کنید. اینک می‌خواهیم خطوط تلفن را به روشی توزیع کنیم که با کمترین مقدار سیم بتوان همه گره‌های شهر را به شبکه وصل کرد .این همان جایی است که درخت پوشا وارد عمل می‌شود.

درخت پوشای کمینه (MST)

در یک گراف وزن‌دار، درخت پوشای کمینه، آن درخت پوشایی است که کمترین وزن را نسبت به دیگر درخت‌های پوشای همان گراف داشته باشد. در موقعیت‌های دنیای واقعی این وزن می‌تواند بر اساس مسافت، ازدحام، بار ترافیکی، یا هر مقدار دلخواهی که به یال‌ها اختصاص می‌یابد اندازه‌گیری شود.

الگوریتم درخت پوشای کمینه

دو مورد از مهم‌ترین الگوریتم‌های درخت پوشای کمینه به صورت زیر هستند:

- الگوریتم کروسکال
- الگوریتم پریم

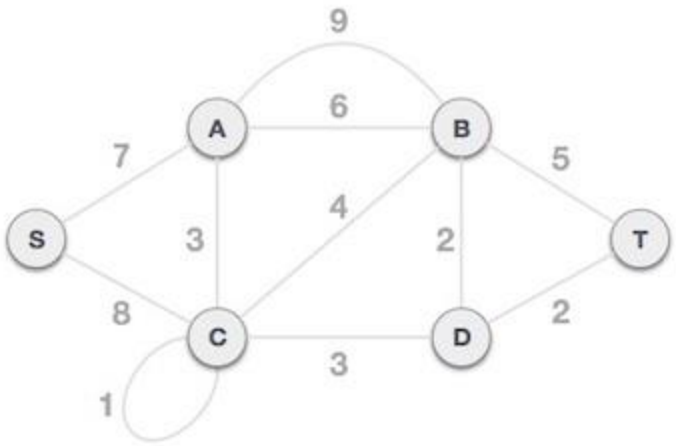
هر دو این الگوریتم‌ها در دسته الگوریتم‌های حریصانه قرار می‌گیرند که در ادامه هر یک را بررسی می‌کنیم:

الگوریتم درخت پوشای کمینه کروسکال (kruskal)

الگوریتم کروسکال برای یافتن درخت پوشای با کمترین هزینه از رویکرد حریصانه بهره می‌گیرد. این الگوریتم با گراف به صورت یک جنگل برخورد می‌کند که در آن هر گره یک درخت منفرد محسوب می‌شود. یک درخت

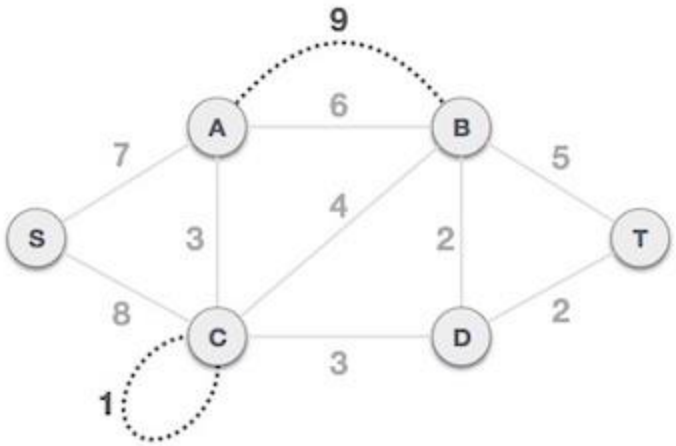
زمانی به درخت دیگر وصل می‌شود اگر و فقط اگر در میان همه گزینه‌های موجود، کمترین هزینه را داشته باشد و مشخصات درخت پوشای کمینه (MST) را نیز نقض نکند.

برای درک الگوریتم کروسکال، مثال زیر را در نظر بگیرید:

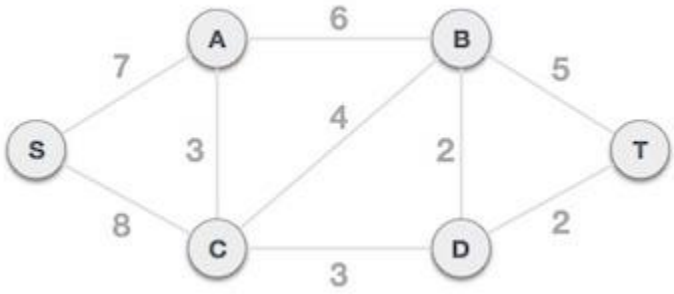


گام ۱ – حذف همه یال‌های طوقه و موازی

همه یال‌های طوقه و یال‌های موازی را از گراف فوق حذف می‌کنیم.



در مورد یال‌های موازی، آن یالی را نگه می‌داریم که کمترین هزینه را دارد و بقیه را حذف می‌کنیم.



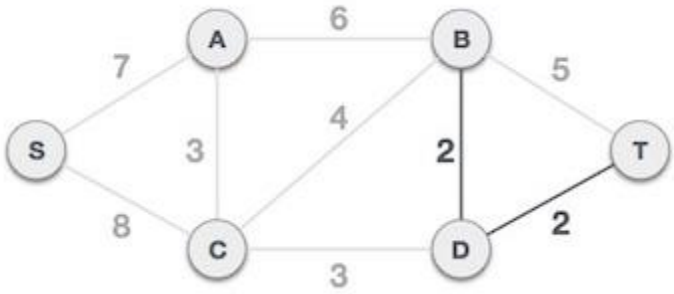
گام ۲ – چیدمان همه یال‌ها به ترتیب افزایش وزن

در این مرحله مجموعه‌ای از یال‌ها و وزن‌هایشان ایجاد می‌کنیم و آن‌ها را بر اساس ترتیب افزایش وزن (هزینه) می‌چینیم.

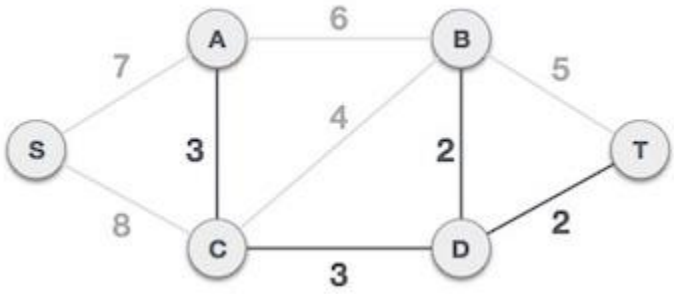
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

گام ۳ – یالی که کمترین وزن را دارد اضافه می‌کنیم

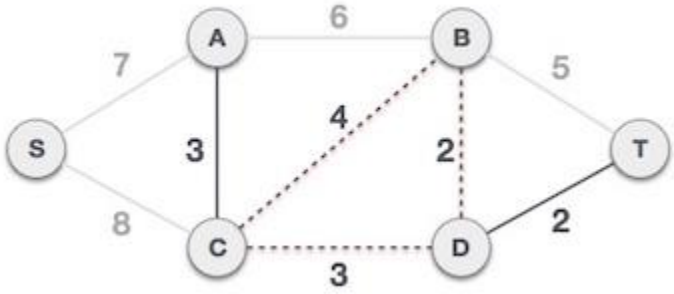
اینک برای افزودن یال‌ها به گراف، کار خود را از یالی آغاز می‌کنیم که کمترین وزن را دارد. در تمام طول این فرایند بررسی می‌کنیم که دور ایجاد نشود وهمچنین درخت پوشا ایجاد نشده باشد. در موردی که با افزودن یک یال مشخصات درخت (بدون دور بودن) نقض شود، نمی‌بایست این یال را وارد گراف کنیم.



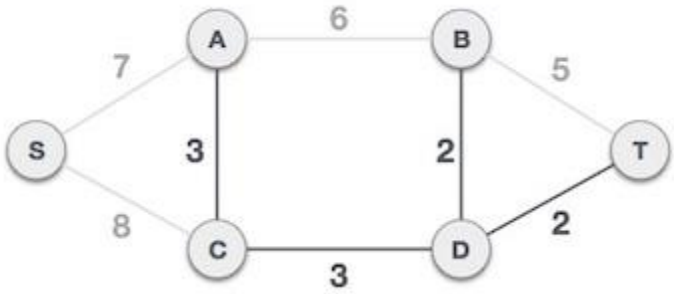
کمترین هزینه ۲ است و یال‌های مربوط به آن B,D و D,T هستند. آن‌ها را به گراف اضافه می‌کنیم. با افزودن این دو یال، مشخصات درخت پوشا نقض نمی‌شود، بنابراین کار خود را ادامه داده و یال‌های بعدی را انتخاب می‌کنیم. در مرحله بعد هزینه ۳ است و یال‌های مربوطه A,C و C,D هستند. آن‌ها را نیز اضافه می‌کنیم.



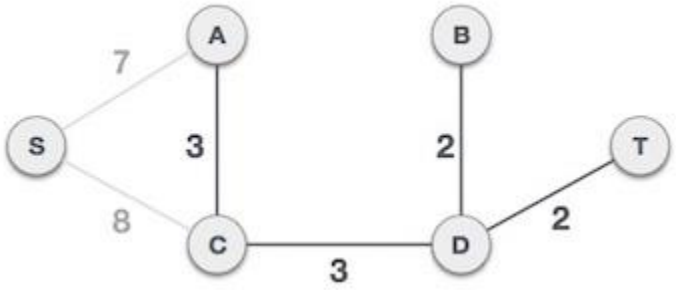
هزینه بعدی در جدول، ۴ است و می‌بینیم که با افزودن یال مربوط به این وزن، دور در گراف ایجاد می‌شود.



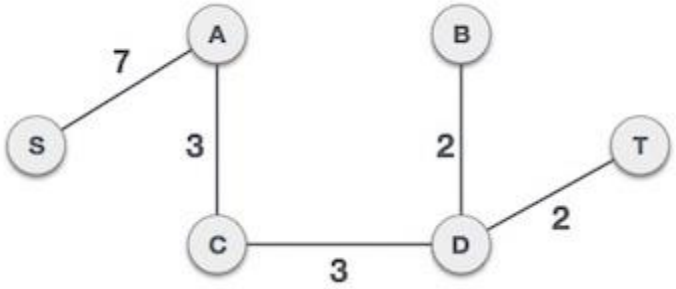
این یال را نادیده می‌گیریم، چون قرار است در این فرایند همه یال‌هایی که باعث ایجاد دور در گراف می‌شوند را نادیده بگیریم.



اینک مشاهده می‌کنیم که یال‌های مربوط به وزن‌های ۵ و ۶ نیز دور ایجاد می‌کنند. پس آن‌ها را نیز نادیده گرفته و به کار خود ادامه می‌دهیم:



در این زمان تنها یک گره مانده است که باید اضافه شود. بین دو یال با کمترین هزینه ۷ و ۸ می‌بایست یالی که وزن ۷ دارد را اضافه کنیم.



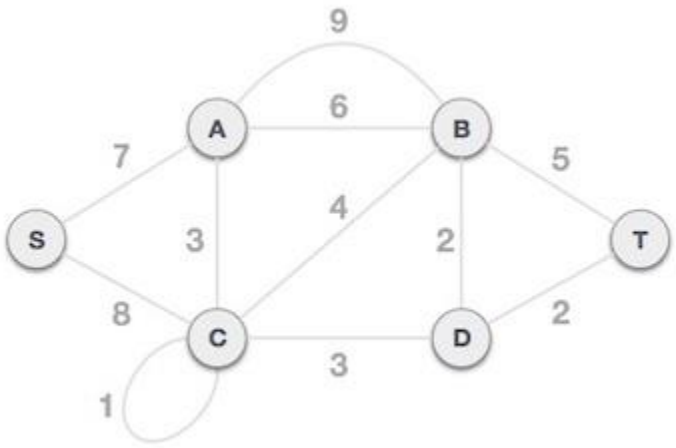
با افزودن یال S,A همه گره‌ها در گراف شامل شده‌اند و اینک درخت پوشای با کمترین هزینه را داریم.

الگوریتم درخت پوشای پریم (Prim)

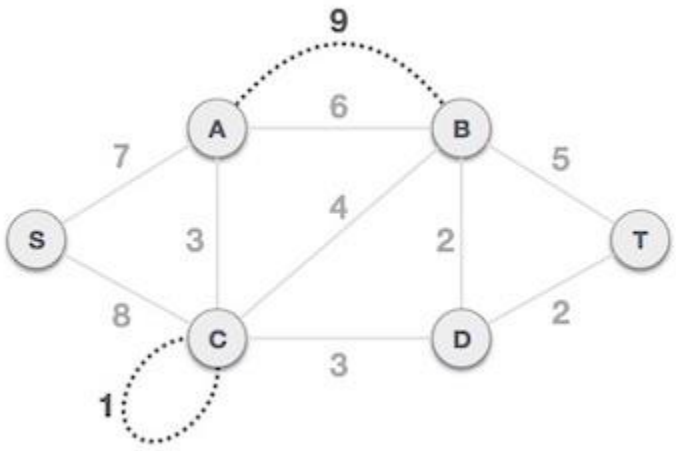
الگوریتم پریم برای یافتن درخت پوشای با کمترین هزینه (همانند الگوریتم کروسکال که در بخش قبل بررسی کردیم) از رویکرد حریصانه بهره می‌گیرد. الگوریتم پریم شباهت‌هایی با الگوریتم‌های «کوتاه‌ترین مسیر، اول» (shortest path first) دارد.

الگوریتم پریم در تضاد با الگوریتم کروسکال است، چون با گره‌ها به عنوان یک درخت منفرد برخورد می‌کند و به افزودن گره‌ها به یک درخت پوشا از گراف مفروض ادامه می‌دهد.

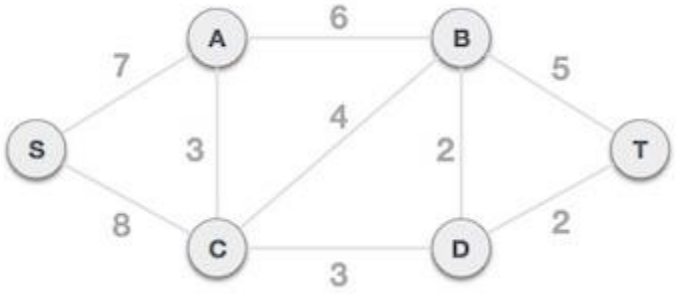
برای درک این تضاد با الگوریتم کروسکال و برای این که بتوانیم الگوریتم پریم را به خوبی درک کنیم از همان مثال بخش قبلی استفاده می‌کنیم.



گام ۱ – حذف همه یال‌های طوقه و موازی



همه یال‌های طوقه و همچنین یال‌های موازی را از گراف مفروض حذف می‌کنیم .در مورد یال‌های موازی، یال‌هایی را حفظ می‌کنیم که کمترین هزینه را دارند و بقیه را حذف می‌کنیم.

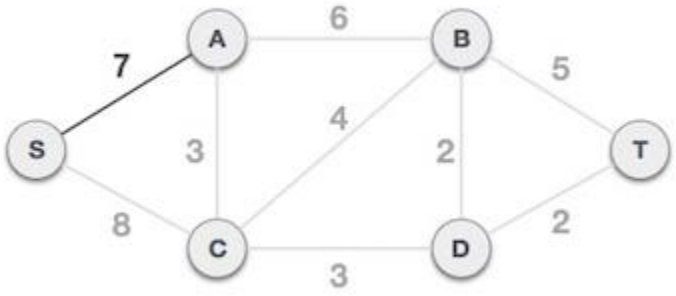


گام ۲ – یک گره دلخواه را به عنوان ریشه انتخاب کنید

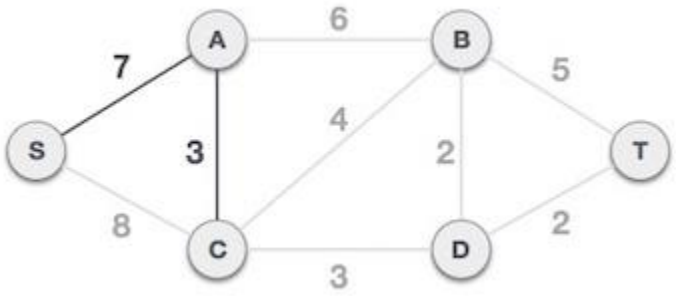
در این مورد ما گره S را به عنوان گره ریشه درخت پوشای پریم در نظر می‌گیریم. این گره به طور دلخواه انتخاب شده است و هر گره دیگری به جای آن می‌توان انتخاب کرد. ممکن است تعجب کنید که چطور ممکن است هر گرهی به عنوان گره ریشه انتخاب شود، پاسخ این است که در درخت پوشا، همه گره‌ها در یک گراف گنجانده می‌شوند و از آن جا که گراف، همبند است، در این صورت می‌بایست دست‌کم یک یال برای هر گره باشد که آن را به بقیه درخت متصل سازد.

گام ۳ – بررسی یال‌های خروجی و انتخاب یالی که کمترین هزینه را دارد .

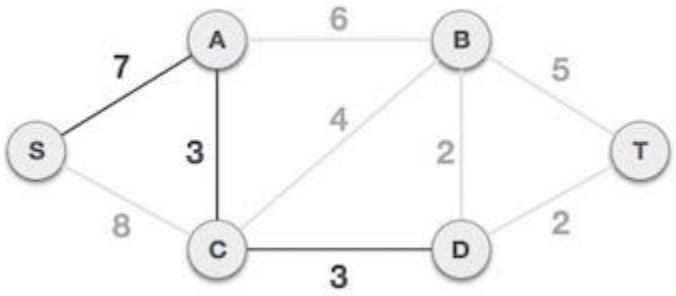
پس از انتخاب گره S به عنوان ریشه می‌بینیم که یال S,A و S,C دو یالی هستند که به ترتیب وزن‌های ۷ و ۸ دارند. یال S,A انتخاب می‌شود چون کوچک‌تر از یال دیگر است.



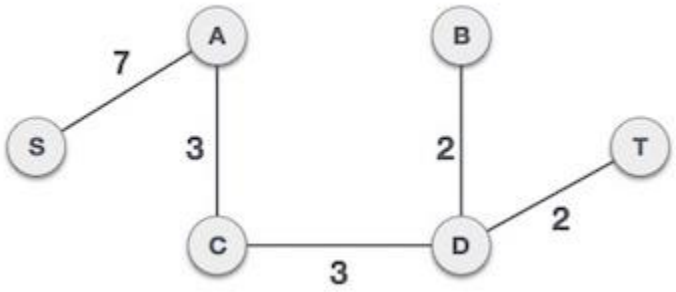
اینک با درخت S-7-Aبه صورت یک گره رفتار می‌شود و همه یال‌هایی که از آن خارج می‌شوند را بررسی می‌کنیم. یالی را انتخاب می‌کنیم که کمترین هزینه را دارد و آن را در درخت می‌گنجانیم.



پس از این مرحله، درخت S-7-A-3-Cتشکیل می‌یابد. اینک بار دیگر با آن به صورت یک گره برخورد می‌کنیم و همه یال‌ها را مجدداً بررسی می‌کنیم. با این حال، مجدداً تنها یالی که کمترین هزینه را دارد انتخاب می‌کنیم. در این مورد، C-3-Dیک یال جدید است که هزینه آن کمتر از یال‌های دیگر است.



پس از افزودن گره D به درخت پوشا اینک دو یال داریم که از آن خارج می‌شوند و هزینه یکسانی دارند، یعنی D-2-Tو .D-2-B بنابراین می‌توانیم هر یک از آن‌ها را که می‌خواهیم به درخت اضافه کنیم .اما در مرحله بعد یال ۲ کمترین هزینه را دارد. از این رو یک درخت پوشا را با گنجاندن هر دو یال نشان می‌دهیم.



بدین ترتیب درمی‌یابیم که خروجی درخت پوشای گراف با استفاده از هر دو الگوریتم یکسان خواهد بود.

نکته : در هر دو روش می توان با بررسی تعداد یال‌های انتخاب شده پوشا بودن درخت را بررسی نمود. یک درخت با n راس هنگامی پوشا است و بدون دور که فقط و فقط n-1 یال داشته باشد.

شبه کد کروسکال

۱. تمام یال‌ها را به ترتیب صعودی وزن مرتب کن
۲. برای هر راس v یک مجموعه بساز.
۳. یال (u,v) را انتخاب کن.
۴. اگر مجموعه‌ی u و v یکی نیستند. کارهای زیر را انجام بده.
    ۱. مجموعه‌ی آنها را ادغام کن.

۲. یال (u,v) را به عنوان یالی از درخت پوشای کمینه بردار.

۵. اگر هنوز n-1 یال انتخاب نشده به شماره ۳ برو.

۶. پایان

پیچیدگی الگوریتم

مرتب کردن یال ها و بررسی یال‌ها از $O(m+mlogn)$ است که برابر $O(mlogn)$ است و هربار اتصال دو مولفه از $O(n)$است که چون اتصال $n-1$ بار انجام می‌شود پیچیدگی الگوریتم $O(mlogn+n^2)$ می‌شود.

شبه کد پریم

۱. راس‌های گراف را به دو مجوعه V' (شامل همه راس‌های گراف به جز راس دلخواه v) و V (شامل v) افراز کن.

۲. تا زمانی که V شامل تمام راس‌ها نشده کارهای زیر را انجام بده

a. بین تمام یال‌هایی که بین مجوعه V و V' کم‌وزن‌ترین را انتخاب کن(مثلا یال (u,v) که u در V

b. راس v را از V' حذف و به V اضافه کن.

c. یال (u,v) را به عنوان یالی از درخت پوشای کمینه انتخاب کن.

۳. پایان

پیچیدگی الگوریتم

یک روش خوب برای بهینه کردن الگوریتم نگه داشتن کم‌ترین فاصله‌ی هر راس تا راس‌های انتخابیست. وقتی یک راس به مجموعه‌ی ما اضافه شد. فاصله‌ی بقیه راس‌ها را به روز می‌کنیم. در این صورت هر بار برای پیدا کردن راس نزدیک‌تر و به روزرسانی $O(n)$ عملیات انجام می‌دهیم و چون n بار این کار انجام می‌دهیم، پیچیدگی کل الگوریتم $O(n^2)$ می‌شود .اگر فاصله‌هر راس تا نزدیک‌ترین راس از بین راس‌هایی که در مجموعه

قرار گرفته‌اند را در داه‌ساختاری مناسب ذخیره کنیم می‌توانیم پیچیدگی الگوریتم را بهتر کنیم. اگر این داده‌ساختار هرم کمینه باشد، چون حذف و اضافه از $O(logn)$ است و به ازای یال حداکثر یک بار عملیات اضافه کردن رخ می‌دهد و چون قطعا تعداد عملیات حذف کم‌تر از تعداد عملیات اضافه کردن است پیچیدگی الگوریتم به $O(mlogn+n)$ تغییر می‌کند که برای حالاتی که تعداد یال‌ها از $\theta(n^2logn)$ کم‌تر باشد پیچیدگی کل کاهش پیدا می‌کند. حال اگر از هرم فیبوناچی استفاده کنیم پیچیدگی به $O(m+n)$کاهش پیدا می‌کند.

مسئله کد گذاری هافمن(Huffman Coding)

»کد هافمن (Huffman Code) «نوع خاصی از «کدهای پیشوندی (Prefix Codes) «بهینه است که اغلب برای فشرده‌سازی بی‌اتلاف اطلاعات مورد استفاده قرار می‌گیرد. فرایند پیدا کردن یا استفاده از این کد به وسیله کدگذاری هافمن (Huffman coding)، با بهره‌گیری از الگوریتمی انجام می‌شود که توسط «دیوید آ هافمن (David A. Huffman) «توسعه داده شده است.

کدهای پیشوندی نوعی از کدها (توالی بیت‌ها) هستند که در آن‌ها کد اختصاص داده شده به یک کاراکتر پیشوند کد تخصیص داده شده به هیچ کاراکتر دیگری نیست. این، روشی است که کدگذاری هافمن با استفاده از آن اطمینان حاصل می‌کند که هیچ ابهامی هنگام رمزگشایی توالی بیت‌های (جریان بیت) تولید شده وجود نخواهد داشت. در ادامه، برای درک بهتر موضوع، مثالی ارائه شده است .فرض می‌شود که چهار کاراکتر c ، b، a و d موجود هستند و کدهای طول متغیر متناظر با آن‌ها به ترتیب ۰۰، ۰۱، ۰ و ۱ است. این کدگذاری موجب ابهام می‌شود زیرا کد تخصیص یافته بهc ، پیشوند کدهای تخصیص یافته به a و b است .اگر جریان رشته فشرده شده ۰۰۰۱ است، خروجی که از حالت فشرده خارج شود امکان دارد cccd یا ccb یا acd یا ab باشد. دو بخش اصلی مهم در کدگذاری هافمن وجود دارد:

۱. ساخت درخت هافمن از کاراکترهای ورودی
۲. پیمایش درخت هافمن و تخصیص کد به کاراکترها

مراحل ساخت درخت هافمن

در اینجا، ورودی آرایه‌ای از کاراکترهای یکتا با تکرار وقوع هر یک و خروجی یک «درخت هافمن» (درخت هافمن) است:

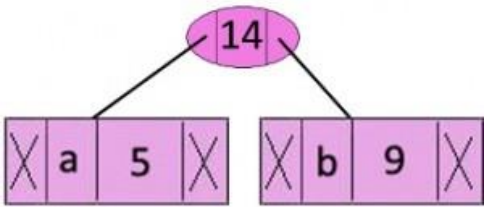
۱. یک گره برگ برای هر کاراکتر یکتا بساز و همچنین، «هرم کمینه (Min Heap) «از همه گره‌های برگ را بساز (هرم کمینه به عنوان صف اولویت استفاده می‌شود. مقدار فیلد تکرار برای مقایسه دو گره در هرم کمینه مورد استفاده قرار می‌گیرد. به طور اولیه، کاراکتری با کمترین تکرار در ریشه است.(
۲. دو گره با حداقل تکرار از هرم کمینه را استخراج کن.
۳. یک گره داخلی با فرکانسی برابر با مجموع تکرارهای دو گره را بساز .اولین گره استخراج شده را به عنوان فرزند سمت چپ و دیگر گره استخراج شده را به عنوان گره سمت راست قرار بده. این گره را به هرم کمینه اضافه کن.
۴. گام‌های ۲ و ۳ را تا هنگامی که هرم تنها حاوی یک گره باشد تکرار کن. گره باقی‌مانده، گره ریشه و درخت کامل است.

در ادامه، برای درک بهتر موضوع، یک مثال بیان شده است.

کاراکتر	تعداد
a	5
b	9
c	12
d	13
e	16
f	45

گام ۱ :یک هرم کمینه بساز که شامل ۶ گره است و هر گره، نشانگر ریشه درخت با یک گره یکتا است.

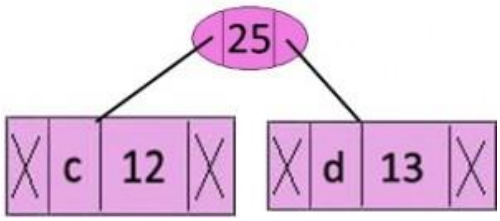
گام ۲ :دو گره با کمترین تکرار را از درخت کمینه استخراج کن. گره داخلی جدید با تکرار ۱۴ = ۹ + ۵ را اضافه کن.



اکنون، هرم کمینه حاوی ۵ گره است که ۴ گره، هر یک با یک عنصر مجرد، ریشه‌های درخت‌ها هستند و یک گره هرم نیز ریشه درخت با ۳ عنصر است.

کاراکتر	تعداد
c	12
d	13
Internal Node	14
e	16
f	45

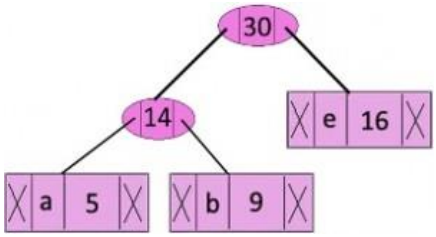
گام ۳ :دو گره کمینه را از هرم استخراج کن. یک گره داخلی جدید با تکرار ۲۵ = ۱۲ + ۱۳ را اضافه کن.



اکنون، هرم کمینه حاوی ۴ گره است که دو گره هر یک با تنها یک عنصر ریشه‌های درخت‌ها هستند و دو گره هرم با بیش از یک گره، ریشه درخت هستند.

کاراکتر	تعداد
Internal Node	14
e	16
Internal Node	25
f	45

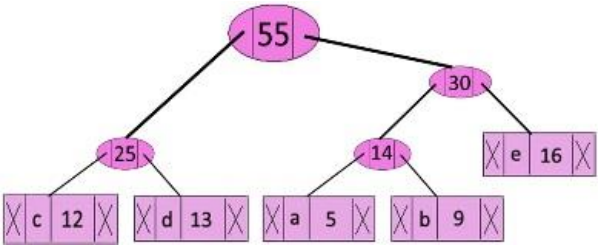
گام ۴ :دو گره با کمترین تکرار را از هرم استخراج کن. یک گره داخلی جدید با تکرار ۳۰ = ۱۶ + ۱۴ اضافه کن.



اکنون، هرم اصلی حاوی ۳ گره است.

کاراکتر	تعداد
Internal Node	25
Internal Node	30
f	45

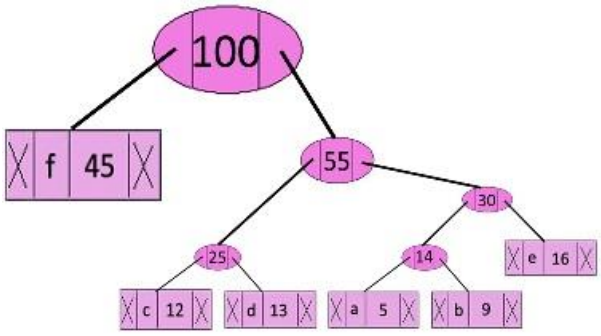
گام ۵ :دو گره با تکرار کمتر را استخراج کن. یک گره داخلی با تکرار ۵۵ = ۳۰ + ۲۵ را اضافه کن.



اکنون، هرم اصلی حاوی دو گره است.

کاراکتر	تعداد
f	45
Internal Node	55

گام ۶ :دو گره با کمترین تکرار را استخراج کن. یک گره داخلی جدید با تکرار ۱۰۰ = ۵۵ + ۴۵ را اضافه کن.



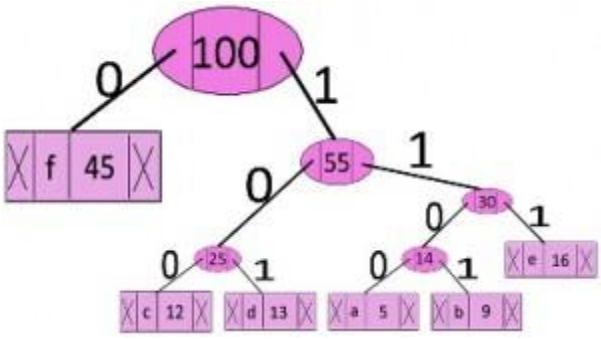
اکنون، هرم کمینه تنها حاوی یک گره است.

کاراکتر	تعداد
Internal Node	100

به دلیل آنکه هرم تنها حاوی یک گره است، الگوریتم در این مرحله متوقف می‌شود.

چاپ کدها از درخت هافمن

پیمایش درخت ساخته شده، از ریشه آغاز می‌شود. برای این کار، باید از یک آرایه کمکی استفاده شود. در این راستا، هنگامی که به فرزند سمت چپ حرکت می‌شود، ۰ باید در آرایه نوشته شود و در حالیکه به سمت فرزند سمت راست حرکت می‌شود، ۱ را باید در آرایه نوشت. آرایه را هنگامی که یک گره برگ مشاهده شد، چاپ کن.



کدها به صورت زیر هستند:

character	code-word
f	0
c	100

d 101

a 1100

b 1101

e 111

روش برنامه‌نویسی پویا (Dynamic Programming)

یکی از روش‌های پرکاربرد و مشهور طراحی الگوریتم روش برنامه‌نویسی پویا یا برنامه‌ریزی پویا، برنامه‌سازی پویا (Dynamic Programming) – است .این روش همچون روش تقسیم و حل (Divide and Conquer) بر پایه‌ی تقسیم مسئله بر زیرمسئله‌ها کار می‌کند .اما تفاوت‌های چشم‌گیری با آن دارد.

زمانی که یک مسئله به دو یا چند زیرمسئله تقسیم می‌شود، دو حالت ممکن است پیش بیاید:

۱- داده‌های زیرمسئله‌ها هیچ اشتراکی با هم نداشته و کاملا مستقل از هم هستند .نمونه‌ی چنین مواردی مرتب‌سازی آرایه‌ها با روش ادغام یا روش سریع است که داده‌ها به دو قسمت تقسیم شده و به صورت مجزا مرتب می‌شوند .در این حالت داده‌های یکی از بخش‌ها هیچ ارتباطی با داده‌های بخش دیگر نداشته و در نتیجه حاصل از آن بخش اثری ندارند .معمولا روش تقسیم و حل برای چنین مسائلی کارآیی خوبی دارد.
۲- داده‌های زیرمسئله وابسته به هم بوده و یا با هم اشتراک دارند .در این حالت به اصطلاح زیرمسئله‌ها هم‌پوشانی دارند .نمونه‌ی بارز چنین مسائلی محاسبه‌ی جمله‌ی nام دنباله‌ی اعداد فیبوناچی است.

دنباله‌ی اعداد فیبوناچی (فیبوناتچی)

دنباله‌ی اعداد فیبوناچی (Fibonacci) یکی از دنباله‌های عددی مشهور ریاضیات با تعریف بازگشتی زیر است:

$$F(n)=F(n-1)+F(n-2) \quad n>2 \quad , \quad F(1)=F(2)=1$$

محاسبه‌ی جمله‌ی nام دنباله به محاسبه‌ی دو جمله‌ی قبلی آن نیاز دارد. پس می‌توان گفت محاسبه‌ی $F(n-1)$ و $F(n-2)$ دو زیرمسئله برای مسئله‌ی اصلی هستند. اما در عین حال این دو زیرمسئله از هم مستقل نیستند .برای محاسبه‌ی $F(n-1)$ بر اساس رابطه‌ی بالا باید داشته باشیم:

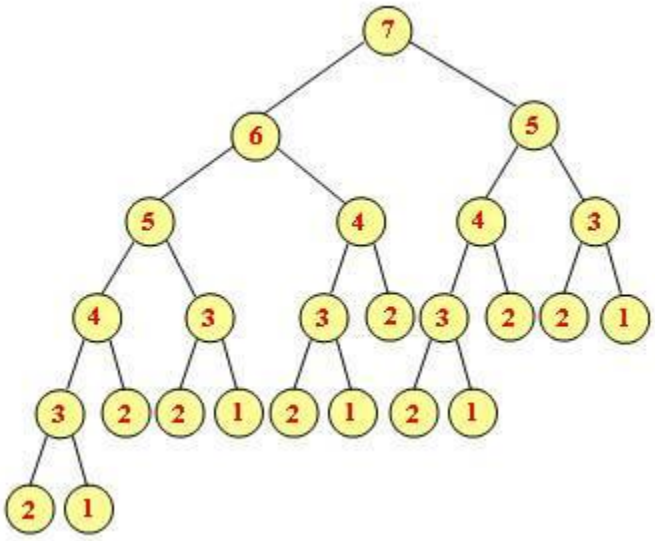
$$F(n-1)=F(n-2)+F(n-3)$$

که نشان می‌دهد خود $F(n-1)$ وابسته به $F(n-2)$ است.

اگر این مسئله را به روش تقسیم و حل - که ساده‌ترین روش است -حل کنیم:

```
1  int fibo(int n){
2    if(n > 2)
3      return fibo(n - 1) + fibo(n - 2);
4    return 1;
5  }
```

تابع fibo مقدار n را دریافت کرده و به صورت بازگشتی و بر اساس رابطه‌ی ذکر شده، جمله‌ی nام دنباله‌ی فیبوناچی را محاسبه می‌کند .حال درخت فراخوانی‌های بازگشتی تابع را به ازای n = 7 رسم می‌کنیم.



هر گره درخت، فراخوانی تابع را با مقدار داخل آن نشان می‌دهد .برای محاسبه‌ی جمله‌ی هفتم دنباله‌ی فیبوناچی تابع fiboبه صورت fibo(7) فراخوانی می‌شود که آن هم fibo(6) وfibo(5) را فراخوانی می‌کند و الی آخر .همانطور که مشاهده می‌کنید، برای محاسبه‌ی این جمله،fibo(7) یک بار، fibo(6) یک بار، fibo(5) دو بار، fibo(4) سه بار، fibo(3) پنج بار، fibo(2) هشت بار، fibo(1) پنج بار و روی هم رفته تابع fibo بیست و پنج بار فراخوانی می‌شود.

ما خود چگونه جملات دنباله‌ی فیبوناچی را محاسبه می‌کنیم؟ ابتدا جمله‌ی اول و دوم را جمع زده و جمله‌ی سوم را محاسبه می‌کنیم. سپس با استفاده از جمله‌ی به دست آمده و جمله‌ی دوم، جمله‌ی چهارم را محاسبه می‌کنیم و همینطور ادامه می‌دهیم:

1 1 2(= 1 + 1)

1 1 2 3(= 2 + 1)

1 1 2 3 5(= 3 + 2)

1 1 2 3 5 8(= 5 + 3)

1 1 2 3 5 8 13(= 8 + 5)

و به این ترتیب جمله‌ی هفتم دنباله تنها با پنج محاسبه‌ی ساده به دست می‌آید .در حالت کلی با استفاده از این روش تنها به n - 2عمل جمع نیاز است که نشان از الگوریتمی با مرتبه‌ی خطی دارد .در حالی که می‌توان ثابت کرد در حالت اول تعداد کل فراخوانی‌های بازگشتی تابع از مرتبه‌ی نمایی است .دلیل اختلاف این دو عدد در این است که در حالت دوم، هر جمله‌ی دنباله فقط و فقط یک بار محاسبه می‌شود .این همان روش برنامه‌نویسی پویا است.

در برنامه‌نویسی پویا مسئله به صورت جزء به کل حل می‌شود .یعنی ابتدا زیرمسائل خرد حل شده و نتیجه‌ی آنها در مکانی ذخیره می‌شود .سپس به سمت زیرمسائل کلی‌تر رفته و با استفاده از داده‌های از پیش محاسبه شده، آنها نیز حل می‌شوند .در مورد دنباله‌ی فیبوناچی می‌توان نوشت:

```
1  int fibo(int n){
2      int f[MAX], i;
3      f[1] = f[2] = 1;
4      for(i = 3 ; i <= n ; i++)
```

```
5       f[i] = f[i - 1] + f[i - 2];
6     return f[n];
7   }
```

در این روش ما جملات دنباله‌ها را پس از محاسبه در یک آرایه ذخیره می‌کنیم .برای این کار به جای حرکت از کل به جزء (یعنی از n به 1 که در روش تقسیم و حل استفاده می‌شود(، از جزء به سمت کل حرکت می‌کنیم.هر جمله‌ی دنباله تنها به دو جمله‌ی قبل خود نیاز دارد که با حرکت جزء به کل قبلا محاسبه شده‌اند و نیاز به محاسبه‌ی مجدد آنها نیست .البته این کد را می‌توان ساده‌تر کرد:

```
1   int fibo(int n){
2     int i, f1, f2, f3;
3     f1 = f2 = 1;
4     for(i = 3 ; i <= n ; i++){
5       f3 = f1 + f2;
6       f1 = f2;
7       f2 = f3;
8     }
9     return f3;
10  }
```

تحلیل این تابع ساده را به خود شما وا می‌گذارم.

نکته‌ی مهم این است که اگر زیرمسئله‌ها هم‌پوشانی نداشته باشند روش برنامه‌نویسی پویا هیچ کمکی به ما نخواهد کرد .چرا که خاصیت اصلی این روش ذخیره داده‌هایی است که ممکن است به کرات به آنها مراجعه شود .حال اگر هیچ اشتراکی در کار نباشد، طبیعتا از هر داده تنها یک بار استفاده خواهد شد.

برنامه‌نویسی پویا برای طراحی الگوریتم‌های محاسبه‌ی حالت‌های بهینه‌ی مسائل نیز کاربرد زیادی دارد .به عنوان مثال در یافتن کوتاهترین مسیر بین دو نقطه، محاسبه‌ی بهینه‌ترین حالت ضرب زنجیری ماتریس‌ها، درخت جستجوی بهینه، مسئله‌ی فروشنده‌ی دوره‌گرد، محاسبه‌ی ضرب چندجمله‌ای‌ها، مسئله‌ی کوله‌پشتی صفر و یک و چندین مسئله‌ی دیگر، از برنامه‌نویسی پویا استفاده می‌شود .شرط اساسی امکان استفاده از این روش برای محاسبه‌ی حالت بهینه به اصل بهینگی مشهور است.

اصل بهینگی :اصل بهینگی یعنی حل مسئله به صورت بهینه، حاوی حل بهینه‌ی تمامی زیرمسائل آن نیز باشد.

به عبارت دیگر، مسئله باید به گونه‌ای باشد که با یافتن حل بهینه‌ی آن، حل بهینه‌ی همه زیرمسئله‌ها نیز به دست آمده باشد .به عنوان مثال، در یافتن کوتاهترین مسیر بین دو شهر، مسیر بین مبدأ و هر گرهی که در مسیر بهینه وجود دارد، بهینه‌ترین مسیر بین آن دو نیز هست.