کتابچه نمونه سوالات چیست:

سایت ما **افتخار** دارد برای اولین بار در ایران توانسته است کتابچه نمونه سوالات تمام دروس پیام نور که هر یک حاوی تمامی آزمون های برگزار شده پیام نور (تمامی نیمسالهای موجود حتی الامکان با جواب) را در یک فایل به نام کتابچه جمع آوری کند و هر ترم نیز آن را آپدیت نماید.

مراحل ساخت یک کتابچه نمونه سوال

(برای آشنایی با زحمت بسیار زیاد تولید آن در هر ترم) :

دسته بندی فایلها – سرچ بر اساس کد درس – چسباندن سوال و جواب – پیدا کردن یک درس در نیمسالهای مختلف و چسباندن به کتابچه همان درس – چسباندن نیمسالهای مختلف یک درس به یکدیگر – وارد کردن اطلاعات تک تک نیمسالها در سایت – آپلود کتابچه و خیلی موارد دیگر..

همچنین با توجه به تغییرات کدهای درسی دانشگاه استثنائات زیادی در ساخت کتابچه بوجود می آید که کار ساخت کتابچه را بسیار پیچیده می کند .

WWW.PNUEB.COM

فصل اول : روش‌های تحلیل الگوریتم

یک مجموعه متناهی از دستورات که برای حل یک مسئله خاص ارائه شده باشد و دارای سه ویژگی زیر باشد را یک الگوریتم برای آن مسئله می گویند:

۱- مرحله به مرحله (پله به پله) باشد.

۲- هر مرحله ای خالی از ابهام باشد.

۳- در بین دستورات، دستوری برای اختتام (شرط توقف) وجود داشته باشد.

الگوریتمها توسط زبان های برنامه نویسی روی کامپیوترها پیاده سازی می شوند. در بین برنامه های مختلفی که به یک منظور مشترک نوشته می شوند، برنامه هایی مطلوبترند که اولاً سرعت اجرای بیشتری داشته باشند، ثانیاً حافظه کمتری نیاز داشته باشند. بحث مدیریت حافظه به این درس مربوط نمی‌شود و ما توجه خود را تنها روی سرعت معطوف خواهیم کرد. در سرعت اجرای یک برنامه، عوامل بسیاری از جمله: سرعت سخت افزار، نوع کامپایلر، مهارت برنامه نویس و پیچیدگی زمانی الگوریتم آن دخالت دارند. شرایطی مانند نوع سخت افزار، نوع کامپایلر، و مهارت های برنامه‌نویسی را می توان برای تمام برنامه های ارائه شده به گونه ای محیا کرد که یکسان باشد ، و تنها عامل منطقی که در سرعت اجرای برنامه اثر می گذارد، همانا پیچیدگی زمانی الگوریتم آن است. برای محاسبه پیچیدگی زمانی الگوریتم باید موارد زیر را شمرده و تعداد هر کدام را در هزینه زمانی آن ضرب کرده و در انتها همگی را با هم جمع کنیم:

۱- اعمال جایگزینی

۲- اعمال محاسباتی

۳- تکرار حلقه ها

۴- توابع بازگشتی

نکته: در شمارش حلقه هایی که شامل *for* می باشند، حدود بالا و پایین *for* را از هم کم کرده و بر طول جهش تقسیم کرده و حاصل را با عدد یک جمع می کنیم.

فصل دوم : روش‌های تحلیل الگوریتم‌های بازگشتی

الگوریتمی را بازگشتی می گویند که برای محاسبهٔ مقدار تابع، نیاز به فراخوانی خودش داشته باشد. مزیت الگوریتمهای بازگشتی عبارتند از سادگی پیاده سازی، سادگی در فهم الگوریتم، و غیره.

ایراد بزرگ الگوریتمها بازگشتی این است که در اغلب مواقع حافظه و زمان اجرای زیادی دارند.

الگوریتمهای بازگشتی شامل دو مرحلهٔ مهم هستند:

۱- عمل فراخوانی

۲- بازگشت از یک فراخوانی

در مرحلهٔ اول، که همان مرحلهٔ فراخوانی است، اعمال زیر انجام می شوند:

الف- کلیهٔ متغیرهای محلی و مقادیر آنها در پشته قرار می گیرند.

ب- آدرس بازگشت به پشته منتقل می شود.

ج- عمل انتقال پارامترها صورت می گیرد.

د- بعد از انجام مراحل بالا، کنترل برنامه به ابتدای پردازه جدید اشاره می کند.

در مرحلهٔ دوم، که همان مرحلهٔ بازگشت است، اعمال زیر انجام می شوند.

الف- متغیرهای محلی از ابتدای پشته حذف شده و در خود متغیرها قرار می گیرند.

ب- آدرس بازگشت از بالای پشته به دست می آید.

ج- آخرین اطلاعات از پشته حذف می شود.

د- کنترل برنامه از آدرس بازگشت قسمت (ب) ادامه می یابد.

برای فهم بیشتر این مراحل، مطالعهٔ مثالهای ارائه شده در صفحات ۴۰ تا ۴۹ کتاب را توصیه می کنیم.

فصل سوم : حل روابط بازگشتی

برای محاسبهٔ زمان اجرای الگوریتمهای بازگشتی، اغلب به رابطه های بازگشتی بر می خوریم. ما در این فصل به چند روش مقدماتی حل روابط بازگشتی اشاره خواهیم کرد.

روش اول: روش حدس و استقراء:

در این روش ابتدا جواب را حدس می زنیم، سپس با استقراء ثابت می کنیم. باید دقت کرد که اگر حدس اولیهٔ ما قابل اثبات نبود، باید حدس دیگری بزنیم و اثباتش کنیم.

مثال: رابطه بازگشتی زیر را توسط روش حدس و استقراء حل کنید:

$$T(n)=\begin{cases}۳ & \text{اگر} \quad n=۱\\ ۲\,T(\frac{n}{۲})+۵n & \text{اگر} \quad n>۱\end{cases}$$

با توجه به اینکه هر بار مسئله را به دو زیر مسئله به اندازه نصف می شکنیم، بنابراین بعد از $\log_۲^n$ مرتبه، مسئله ای به اندازهٔ یک خواهیم داشت. چون زیر مسئله ها را با زمان $۵n$ با هم ترکیب می کنیم، حدس می زنیم کل زمان مسئله برابر $O\left(n\log_۲^n\right)$ باشد. برای اثبات حدس خود باید ثابت کنیم یک $C>۰$ هست که

$$T(n)\le Cn\log_۲^n$$

برای n=۱ سمت چپ برابر۳ و سمت راست برابر صفر است که تناقض می کند. پس حدی خود را تصحیح می کنیم و حدس جدید خود را به صورت

$$T(n)\le C\,n\log_۲^n+b$$

قرار می دهیم.

برای اثبات می بینیم برای $n=۱$ باید b را بزرگتر از ۳ انتخاب کرد. حال فرض کنیم برای همهٔ kهای کوچکتر از n حکم برقرار باشد و حکم را برای $k=n\ge ۲$ ثابت می کنیم.

$$T(k)=۲T\left(\frac{k}{۲}\right)+۵k\le ۲\left(C\frac{k}{۲}\log_۲^{k/۲}+b\right)+۵k\le Ck\log_۲^k-Ck+۲b+۵k$$

می توان C را طوری انتخاب کرد که C > ۵+b، و در نتیجه با توجه به رابطهٔ بالا، رابطه‌ی زیر بدست می‌آید :

$$T(k) \le Ck \log_a^k + b$$

و حکم ثابت می شود.

روش دوم: روش تکدار با جایگذاری:

در این روش نیازی به حدس جواب نیست و می توان با جایگذاریهای متوالی، جواب مناسب را به دست آورد.

مثال: رابطهٔ بازگشتی زیر را توسط روش تکرار یا جایگذاری حل کنید:

$$T(n) = \begin{cases} ۳ & n = ۱ \\ ۲\,T(\frac{n}{۲}) + ۵n & n > ۱ \end{cases}$$

حل:

$$T(n) = ۲T\left(\frac{n}{۲}\right) + ۵n = ۲\left(۲T(\frac{n}{۴}) + ۵\frac{n}{۲}\right) + ۵n$$

$$= ۴T\left(\frac{n}{۴}\right) + ۵n\left(۱ + \frac{۱}{۲}\right) = \ldots = ۲^k T\left(\frac{n}{۲^k}\right) + ۵n\left(۱ + \frac{۱}{۲} + \ldots + \frac{۱}{۲^k}\right)$$

چون نهایتاً رابطهٔ بالا به T(۱) می رسد، پس

$$\frac{n}{۲^k} = ۱ \Rightarrow n = ۲^k \Rightarrow k = \log_۲^n$$

پس با توجه با آخرین رابطهٔ $T(n)$، و با توجه اینکه $۱ + \frac{۱}{۲} + \ldots + \frac{۱}{۲^k} < k$ داریم:

$$T(n) = nT(۱) + ۵n\left(۱ + \frac{۱}{۲} + \frac{۱}{۴} + \ldots + \frac{۱}{۲^k}\right) < ۳n + ۵nk$$

$$= ۳n + ۵n \log_۲^n \Rightarrow T(n) \in O\left(n \log_۲^n\right).$$

روش سوم: استفاده از قضیهٔ اصلی:

قضیه اصلی: فرض کنیم $a \ge ۱$, $b \ge ۱$ دو عدد ثابت باشند و $f(n)$ تابعی بر حسب n باشد. همچنین فرض کنیم رابطهٔ بازگشتی $T(n)$ به صورت زیر تعریف شده باشد.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

آنگاه:

الف) اگر به ازای یک $\varepsilon > ۰$ داشته باشیم $f(n) \in O\left(n \log_b^a - \varepsilon\right)$ آنگاه

$T(n) \in \theta\left(n \log_b^a\right).$

ب) اگر $f(n) \in \theta\left(n \log_b^a\right)$ آنگاه

$T(n) \in \theta\left(n \log_b^a . \log_۲^n\right)$

ج) اگر یک $\varepsilon > ۰$ ، و یک $C>۰$ داشته باشیم $af\left(\frac{n}{b}\right) \le Cf(n)$، آنگاه داریم:

$T(n) \in \theta\left(f(n)\right)$

مثال:

$$T(n) = \begin{cases} ۳ & \text{اگر} \quad n = ۱ \\ ۲\,T(\frac{n}{۲}) + ۵n & \text{اگر} \quad n > ۱ \end{cases}$$

حل: در اینجا a=b=۲ پس $\log_b^a = ۱$ چون

$$f(n) = ۵n \in \theta(n) = \theta\left(n^{\log_b^a}\right)$$

پس طبق قسمت ب) داریم:

$$T(n) \in \theta\left(n^{\log_b^a} . \log_۲^n\right) = \theta\left(n \log_۲^n\right).$$

روش چهارم:

این روش برای حل معادلات خطی مرتبهٔ دوم، که ضرایب ثابت دارند و همگن هستند، یعنی به صورت $T(n) = aT(n-۱) + bT(n-۲)$ هستند، به کار می رود. معادلهٔ درجهٔ دوم $x^۲ - ax - b = ۰$ را حل می کنیم.

اگر جوابهای $x_۱, x_۲$ متمایز بودند آنگاه $T(n) = c_۱x_۱^n + c_۲x_۱^n$.

اگر $x_۱$ جواب مضاعف آن بود، آنگاه $T(n) = c_۱x_۱^n + c_۲n\,x_۱^n$

توجه: اگر رابطهٔ اصلی ناهمگن بود، باید با ضرب کردن در مقادیر مناسب، و تغییر اندیس، رابطه را به یک رابطه همگن تبدیل کرد.

مثال: $T(n)=۲T(n-۱)+۳^n$

حل: یکبار n ها را به n+۱ تبدیل می کنیم، و یکبار هم دو طرف رابطه را در۳ ضرب می کنیم، داریم:

$$\begin{cases} ۳T(n)=۶T(n-۱)+۳^{n+۱} \\ T(n+۱)=۲T(n)+۳^{n+۱} \end{cases}$$

با کم کردن دو رابطه از هم داریم:

$$T(n+۱)-۳T(n)=۲T(n)-۶T(n-۱) \Rightarrow$$
$$T(n+۱)=۵T(n)-۶T(n-۱) \Rightarrow$$
$$x^۲-۵x+۶=۰ \Rightarrow x=۱,\ x=۵ \Rightarrow$$
$$T(x)=c_۱۱^n+c_۲۵^n \Rightarrow T(n)\in O(۵^n)$$

فصل چهارم: روش تقسیم و حل(*Divide and Conquer*)

الگوریتم تقسیم و حل، از یک ساختار ۳ مرحله ای پیروی می کند.

۱) تقسیم: مسئله به چند زیر مسئلهٔ کوچکتر تقسیم می شود.

۲) حل: اگر حجم زیر مسئله ها به اندازهٔ کافی کوچک بود آنها را مستقیماً حل میکند، و الا زیر مسئله‌ها را از طریق بازگشتی حل می کند.

۳) ترکیب: با ادغام جوابهای زیر مسئله ها، مسئلهٔ اصلی را حل می کند.

در حالت کلی، شکل عمومی یک الگوریتم تقسیم و حل را می توان چنین نوشت:

```
Result   Dandc(p
{
  if (Smal(p))
     return S(P);
   else{
    Divide P into Smaller instances P1,....,PK for k≥1;
    /* Apply Dandc to each of these subproblems */
    return combine (Dandc(P1),..., Dandc(Pk));
          }
    }
```

پس به طور خلاصه، الگوریتم تقسیم و حل، یک الگوریتم از بالا به پایین است و در جاهایی کارآیی دارد که بتوان مسئله را به تعدادی زیر مسئله تقسیم کرد که اولاً همهٔ زیر مسئله ها از ماهیت خود مسئله‌ی اصلی باشند، ثانیاً حجم هر زیر مسئله، تقسیم شده ای از حجم مسئلهٔ اصلی باشد، ثالثاً تعداد زیر مسئله ها به اندازهٔ غیر منطقی زیاد نشود.

در تحلیل الگوریتم های تقسیم و حل همواره به رابطه های بازگشتی از نوع

$$T(n) = aT\left(\frac{n}{b}\right) + F(n)$$

برخورد می کنیم که حل کردن این نوع روابط بازگشتی را در فصل ۳ یادآور شدیم.

از معروفترین مسائلی که برای آنها الگوریتم تقسیم و حل طراحی می شوند، می توان به مسائل جستجوی دودویی، مرتب سازی ادغامی، مرتب سازی سریع، ضرب ماتریسها از روش استراسن، ضرب اعداد بزرگی و غیره اشاره کرد که مشروح هر یک از آنها را می توان در کتاب درسی مطالعه کرد.

فصل پنجم : روش حریصانه(*Greedy*)

در نگرش حریصانه، ما در هر مرحله بهترین انتخاب را انجام می دهیم. قابل توجه است که طبق معیارهای موجود در آن مرحله انتخاب را انجام می دهیم و اصلاً به عوارض بعدی این انتخاب فکر نمی‌کنیم. به عبارت دیگر، روش حریصانه، از آینده نگری دور است و به اصطلاح معروف «دم را غنیمت شمردن» است.

خصوصیات کلی یک الگوریتم حریصانه عبارتند از:

۱- نتیجه نهایی یک الگوریتم حریصانه، مجموعه ای (اغلب مرتب) از زیر داده های اصلی مسأله است.

۲- مرحله به مرحله، یک مؤلفه از مجموعه جواب انتخاب می شود.

۳- ایده آل است که تابع هدف، با توجه به مجموعه جواب، بهینه شود.

۴- در هر مرحله، طبق یک روال انتخاب (*select*) یک مؤلفه از جواب انتخاب می شود، که البته جواب انتخاب شده، قطعی و غیر قابل تصحیح می باشد.

اجزای تشکیل دهنده یک الگوریتم حریصانه عبارتند از:

۱- مجموعه ای از انتخاب های ممکن برای مؤلفه های جواب به نام C و مجموعه مؤلفه های انتخاب شده تا به حال به نام S

۲- روالی به نام *select*، برای انتخاب مؤلفه های بعدی جواب از مجموعه انتخاب های ممکن.

۳- روالی به نام *feasible*، که عناصر انتخاب شده توسط روال *select* را جهت قرار گرفتن در مجموعه جواب یا رد آن بررسی می کند.

۴- روالی به نام *solution*، برای بررسی اینکه مشخص کند در نهایت، جواب حاصل شده است یا خیر.

۵- یک تابع هدف، که هدف مسئله بهینه کردن این تابع است.

شکل کلی یک الگوریتم حریصانه را می توان چنین عنوان کرد:

```
Set      Greedy (C)
{
     /* C  Contains  of  inputs. */
S = ∅
while   (!Solution(S) and  C!= ∅)
  {    X = Select (C)
       C = C − {X}
       if (feasible(S, X)); Then  S = SU{X}
     }
     if (Solution (S)) then return(S)
     else return(∅)
   }
```

از مسائل مقدماتی معروفی که توسط روش حریصانه حل می شوند می توان به مسئله خرد کردن پول، مسئله کوله پشتی، مسئله زمانبندی و الگوریتم های پرایم، دیکسترا، کروسکال، و هافمن اشاره کرد، که هر یک از آنها در کتاب به تفصیل توضیح داده شده است.

فصل ششم : برنامه نویسی پویـا

برنامه نویسی پویا، نام روشی است که مانند روش تقسیم و حل مسئله را بـه نمونـه هـای کـوچکتر تقسیم می کند ولی برخلاف روش تقسیم و حل که نمونه های کوچکتر مشابه را چنـدین بـار حـل می کند، در این روش نمونه های کوچکتر پس از حل شدن ذخیره مـی شـوند تـا در موقع نیـاز از آنها استفاده شود. در واقع برنامه نویسی پویا یک روش از پایین به بالا است.

هرگاه بخواهیم از روش برنامه نویـسی پویـا اسـتفاده کنـیم، بایـد آرایـه ای تعریـف کنـیم کـه اولاً جملات اولیهٔ آن را بتوان از روی داده هـای مـسئله مـشخص کـرد، ثانیـاً هـدف مـسئله را بتـوان بـه صورت جمله ای از آن بیان کرد، ثالثاً با یافتن یک رابطهٔ بازگشتی بتوان از داده ها به هدف رسید.

چون هر الگوریتمی که به روش برنامه نویسی پویا طرح می شود بر یک رابطـه‌ی، بازگـشتی سـوار است، پس قطعاً به جواب بهینه می رسد و برخلاف روش حریصانه لزومی ندارد که در انتها با بیـان قضیه ای بهینه بودن جواب را ثابت کنیم. به دلیل استوار بودن این روش بر روشهای ریاضی، اکثـراً آنرا سختترین و فنی ترین روش می دانند.

برنامه نویسی پویا را می توان برای حل کلیهٔ مسائل بهینه سازیی به کار گرفـت کـه در شـرط زیـر، موسوم به اصل بهینگی صدق کنند:

اصل بهینگی:

می گوئیم شرط بهینگی در یک مسئله صدق می کند اگر یک حل بهینه برای نمونـه ای از مـسئله، شامل حل بهینه برای همهٔ زیر نمونه های آن نیز باشد.

به طور تجربی می توان گفت که **اکثر** مسائل کمینه سازی (*minimum*) در اصل بهینگی صـدق می‌کنند و **اکثر** مسائل شبیه سازی (*maximum*) در آن اصل صدق نمـی کننـد. دقـت شـود کـه گفتیم **اکثراً** چون مثلاً مسئلهٔ کوله پشتی ۱ – ۰ از نوع بیشیه است ولی در شـرط فـوق نیـز صـدق می‌کند.

از معروفترین مسائل مقدماتی که با روش برنامه نویسی پویا حـل مـی شـوند و در کتـاب نیـز مـورد بررسی قرار گرفته اند می توان به مسائل ضرب زنجیر ماتریسها، درخت جستجوی دودویـی بهینـه، مسئله فروشنده دوره گرد، مسئله کوله پشتی ۱ – ۰، و الگوریتم فلوید اشاره کرد.

فصل هفتم : تکنیک عقبگرد

روش بازگشت به عقب یکی از روشهای قدرتمند و پر کاربرد در طراحی الگوریتم است. اغلب مسائلی که به روش بازگشت به عقب حل می‌شوند از نوعی هستند که از قوانین، مفاهیم، نمایش، و پیمایش درختها سود می برند. از آنجایی که اکثر مسائلی که توسط درخت ها حل می شوند، از نوع تصمیم گیری هستند، پس اکثر مسائلی که توسط روش بازگشت به عقب حل می شوند، از نوع تصمیم‌گیری هستند. در این نوع مسائل اصولاً باید از یک مجموعهٔ وسیع که همهٔ گزینه های انتخابی را در بردارد، یک مجموع کوچکتر که همان جواب می باشد را انتخاب کرد. اکثر الگوریتمهای بازگشت به عقب از مرتبهٔ نمایی می باشند. دقت شود که با اعمال این روش، پیچیدگی الگوریتم پایین نمی آید ولی تعداد واقعی محاسبات کم می شود. در اینجا برخی از ویژگیهای مسائلی که به روش بازگشت به عقب حل می شوند، و همچنین ویژگیهای خود روش بازگشت به عقب، را متذکر می‌شویم.

۱) اکثر مسائلی که به روش بازگشت به عقب حل می شوند، ذاتاً سخت می باشند و از مرتبهٔ نمایی هستند.

۲) روش بازگشت به عقب از قوانین و تکنیکهای درختها استفاده می کند.

۳) مسائلی که به روش بازگشت به عقب حل می شوند، اکثراً از نوع مسائل تصمیم گیری می‌باشند.

۴) این روش، تمام جوابهای مسئله را پیدا می کند.

روش بازگشت به عقب، اصلاح شدهٔ جستجوی عمقی یک درخت است. باید دقت کرد که همانطور که از اسم این روش پیدا است، هر کجا به بن بست خوردیم بر میگردیم و آخرین انتخاب خود را تصحیح می کنیم(برخلاف روش حریصانه که در آن امکان تصحیح کردن انتخابهای قبلی نیست).

از معروفترین مسائلی که در سطح مقدماتی با این روش حل می شوند ، می توان به مسائل : n- وزیر ، حاصل جمع مجموعه ها، رنگ آمیزی گرافها، مدارهای همیلتونی، و کوله پشتی صفر و یک، اشاره کرد که در کتاب به تفصیل بررسی شده اند.

در انتها یاد آور می شویم که شکل کلی یک الگوریتم بازگشت به عقب به صورت زیر است، در اینجا، منظور از یک گره امید بخش، یعنی گرهی که امید است در فرزندان آن بتوان مسیری به سوی جواب مسئله پیدا کرد.

```
Void   backtrack   (node V)
   {
      node u ;
      if (promising (V))
         if(there is a solution at.V)
            write the solution ;
         else
            for (each child u of V)
                  back track (u) ;
      }
```

فصل هشتم : انشعاب و تحدید(*Branch and Bound*)

روش انشعاب و تحدید یکی دیگر از روشهای پیمایش و جستجوی درختها و گرافهاست. فضای حالت مسئله ای که می خواهد با این روش حل شود باید قابل نمایش توسط یک گراف باشد.

بر خلاف روش بازگشت به عقب که الگوی جستجوی درخت جستجوی عمقی بود، در روش انشعاب و تحدید روش جستجوی درخت، جستجوی عرضی(پهنی) میباشد. در این روش الگوریتمی به نام هرس کردن ارائه می‌دهیم که جستجو را بهینه میکند. اما این روش هرس کردن، فقط تعداد جستجو را کم می کند، ولی از نظر پیچیدگی الگوریتمهای نمایی ایجاد می کند.

بر خلاف جستجوی عمقی، هیچ الگوریتم بازگشتی ساده ای برای جستجوی عرضی وجود ندارد، ولی می توان آن را توسط یک صف، یا یک صف اولویت دار، پیاده سازی کرد. تکنیک کلی این کار در صفحات ۲۹۴ و ۲۹۵ بیان شده است.