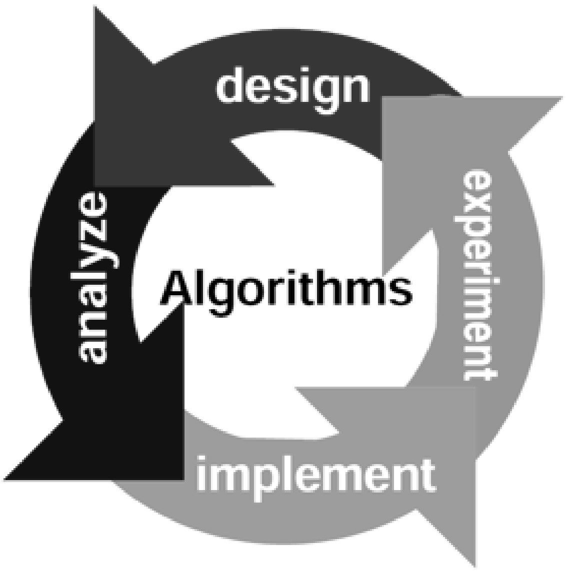
به‌نام خداوند جان و خرد   کزین برتر اندیشه برنگذرد

طراحی و تحلیل الگوریتم‌ها



جواد شاهپریان

گروه مهندسی کامپیوتر

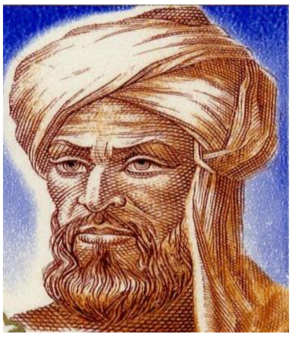
دانشکده فنی دانشگاه آزاد اسلامی واحد تهران جنوب

ویراست سوم

مهرماه ۱۳۹۱

با تشکر از دانشجویان درس «طراحی الگوریتم‌ها» دانشکده فنی دانشگاه آزاد اسلامی واحد تهران جنوب که در امر آماده‌سازی و تدوین این جزوه کمال همکاری را داشته‌اند.

به امید موفقیت روز افزون این عزیزان



تاریخچه

واژه الگوریتم از نام ریاضی‌دان ایرانی قرن نهم میلادی، محمد ابن موسی خوارزمی گرفته شده است. کتاب معروف الجبر و المقابله خوارزمی حاوی دستورالعمل‌های مختلف برای حل مسائل محاسباتی است. کتاب محاسبه با عددهای هندی او با عنوان "Algoritmi de numero Indorum" به لاتین ترجمه شد که معادل "Algoritmi on the numbers of the Indians" است. Algoritmi در حقیقت برگردان شده‌ی لاتین نام نویسنده است که بعد از آن کلمه Algorithm به وجود آمد. نام خوارزمی هم در ترجمه به جای الخوارزمی به صورت الگوریتمی تصنیف گردید و الفاظ الگوریسم و نظایر آنها در زبان‌های اروپایی که به معنای فن محاسبه ارقام یا علامت‌های دیگر است، از آن مشتق شده است. در حال حاضر این کلمه شامل تمام روش‌های معین حل مسئله یا انجام یک کار می‌شود.

تعریف الگوریتم

الگوریتم مجموعه‌ای متناهی از دستورالعمل‌هاست که به‌صورت دقیق و بدون ابهام بیان شده‌اند. اگر این دستورالعمل‌ها به ترتیب خاصی اجرا شوند، می‌توانند مسئله‌ای را حل کنند. به‌عبارت دیگر، الگوریتم روشی گام به گام برای حل مسئله است.

تحلیل الگوریتم

هر الگوریتم ممکن است عمل مورد نظر را با دستورات مختلف در مدت زمان و یا کار کمتر یا بیشتری نسبت به الگوریتم دیگر انجام دهد. به همین دلیل انتخاب الگوریتم مناسب و کارا اهمیت زیادی در موفق بودن و کارایی برنامه دارد.

تحلیل الگوریتم‌ها رشته‌ای است که به بررسی کارایی الگوریتم‌ها می‌پردازد. موضوع تحلیل الگوریتم‌ها در مورد تعیین میزان منابعی است که برای اجرای هر الگوریتم لازم است. در این منابع معمولا زمان و حافظه هم در نظر گرفته می‌شوند. کارایی یا پیچیدگی هر الگوریتم را با تابعی نشان می‌دهند که تعداد مراحل لازم برای اجرای الگوریتم را برحسب طول داده ورودی نشان می‌دهد. غالبا مشاهده می‌شود که یک مسئله را با استفاده از چندین تکنیک مختلف می‌توان حل نمود ولی فقط یکی از آنها به الگوریتمی منجر می‌شود که از بقیه سریعتر است.

سرفصل مطالب درس طراحی الگوریتم‌ها
ارائه شده از سوی شورای عالی برنامه ریزی وزارت علوم تحقیقات و فن‌آوری

یادآوری مطالب مهم در درس ساختمان‌داده و تکمیل نکات ارائه شده در خصوص: استقراء ریاضی و روش‌های بازگشتی، پیچیدگی الگوریتم‌ها و آنالیز آنها، نمادهای **O**، Θ، Ω، **o**. روش‌های حل مسئله: در هر روشی تعدادی مسئله مهم انتخاب و الگوریتم‌های هر یک گفته شده و اثبات و آنالیز گردد. روش تقسیم و حل (مسائل: ماکزیمم و مینیمم یک آرایه، ضرب دو عدد **n** بیتی، روش **Strassen** در ضرب ماتریس‌ها، تورنمنت بازی‌ها، مرتب‌کردن براساس **Quicksort**). روش برنامه‌سازی پویا (مسائل: ضرب ماتریس‌ها، کوله پشتی، مثلث‌بندی بهینه یک چندضلعی، طولانی‌ترین زیرترتیب مشترک، حروفچینی یک پاراگراف). روش حریصانه (مسائل: مسائل زمانبندی، خردکردن پول، کد هافمن). روش‌های مبتنی بر جستجوی کامل و تکنیک‌های محدودکردن فضای جستجو، استفاده از درخت بازی و **α-β Pruning** (بازی‌های **Puzzle, tic-tac-toe**). روش‌های مکاشفه‌ای برای حل مسائل مشکل (مسئله فروشنده دوره گرد)، الگوریتم‌های گراف شامل: روش‌های جستجوی گراف (عمقی، سطحی)، گراف‌های بدون جهت (الگوریتم‌های **Dijkstra**، درخت پوشای مینیمال، اجزاء همبند، کاملاً همبند و مسائل دیگر). گراف‌های جهت‌دار (الگوریتم‌های **Floyd** ، مرتب‌کردن **Topological** اجزاء دو همبند و...)، شبکه‌های ماکزیمم جریان و مسائل مربوطه.

مراجع :

1. R. E. Neapolitan and K. Naimipour, Foundations of Algorithms Using C++ Pseudo Code, Third Edition, Jones and Bartlett Publishers, 2004.

2. Cormen, Leiserson, Rivest and Stein, Introduction to Algorithms, Second Edition, MTT Press, 2001.

3. E. Horowitz and S. Sahni, Fundamntals of Computer Algorithms, Computer Science Press 1978.

4. Aho, Hopcroft, Ullman, Data Structures & Algorithms, Addison-Wesley, 1985.

5. Udi Manber, Introduction to Algorithms: A Creative Approach, Addison-Wesley, 1987.

مراجع تدریس:

1. Cormen, Leiserson, Rivest and Stein, Introduction to Algorithms, Third Edition, MIT Press, 2009.

2. R. E. Neapolitan and K. Naimipour, Foundations of Algorithms Using C++ Pseudo Code, Third Edition, Jones and Bartlett Publishers, 2008.

فهرست

فصل اول: تحلیل الگوریتم‌ها

فصل دوم: روش‌های طراحی الگوریتم‌ها

فصل سوم: روش تقسیم و حل

فصل چهارم: برنامه نویسی پویا

فصل پنجم: الگوریتم حریصانه

فصل ششم: پیمایش گراف‌ها

فصل اول: تحلیل الگوریتم ها

هدف از تحلیل یک الگوریتم ارائه یک تخمین از زمان اجرای آن الگوریتم است. به عبارت دیگر، منظور از تحلیل الگوریتم، بررسی تغییر رفتار یک الگوریتم در مقابل تغییر اندازه‌ی ورودی است.

تحلیل برنامه‌ها از یک جنبه به دو صورت مورد بررسی قرار می‌گیرد: $\left\{ \begin{array}{l} \text{زمانی} \\ \text{حافظه} \end{array} \right.$

مرتبه زمانی (Order)

پیچیدگی زمانی: تعیین می‌کند که برای هر مقدار از اندازه‌ی ورودی چندبار عمل مبنایی اجرا می‌شود.

پیچیدگی حافظه: میزان حافظه‌ای که برنامه در حین اجرا به آن نیاز دارد را تعیین می‌کند.

عمل مبنایی (عمل اصلی): عملی است که بیش‌ترین بار محاسباتی روی آن اعمال می‌شود. به عنوان مثال در جستجوی خطی عمل مبنایی مقایسه و در ضرب ماتریس‌ها عمل مبنایی ضرب و جمع است.

۱-۱ عوامل موثر بر سرعت اجرای الگوریتم‌ها

۱. نوع برنامه نویسی
۲. نوع کامپایلر
۳. سخت افزار
۴. اندازه ورودی
۵. ترکیب ورودی

۱-۱-۱ ترکیب ورودی

۱. بهترین حالت (Best Case): $T_B(n)$
۲. حالت متوسط (Average Case): $T_A(n)$
۳. بدترین حالت (Worst Case): $T_W(n)$

در اینجا T همان «زمان اجرای الگوریتم» و n همان «اندازه‌ی ورودی» است.

مثال: در الگوریتم جستجوی دودویی درون یک آرایه مرتب (فرض می‌کنیم صعودی است) به دنبال یک عنصر خاص (کلید) می‌گردیم. در ابتدا کلید را با عنصر وسط آرایه مقایسه می‌کنیم، سه حالت زیر به‌وجود می‌آید:

۱) کلید درون آرایه موجود است و در وسط آرایه قرار دارد: در این صورت فقط یک مقایسه صورت می‌گیرد و این بهترین حالت است. $T_B(n) = O(1)$

۲) کلید درون آرایه موجود است و در جایی غیر از وسط آرایه قرار دارد: در این صورت کلید با عنصر وسط مقایسه می‌شود اگر بزرگ‌تر از آن بود در زیر آرایه سمت راست و اگر کوچک‌تر از آن بود در زیر آرایه سمت چپ به جستجو می‌پردازد و این حالت متوسط است. $T_A(n) = O(\log n)$

۳) کلید درون آرایه موجود نیست: در این صورت الگوریتم تا پایان در حال مقایسه کردن است و این بدترین حالت است که مرتبه زمانی آن با حالت متوسط یکسان است. $T_W(n) = O(\log n)$

تحلیل برنامه‌ها از جنبه دیگر نیز به دو صورت انجام می‌پذیرد:
- تحلیل جزئی
- تحلیل کلی

۲-۱ تعیین مرتبه زمانی برنامه‌ها با تحلیل جزئی

۱-۲-۱ تحلیل زمانی حلقه for

```
for (i=a; i<=b; i++)     →  (b-a+1)+1   تعداد دفعات اجرای شرط حلقه برابر است با
   {
    ⋮      →  (b-a+1)   تعداد دفعات اجرای بدنه حلقه برابر است با
   }
```

۲-۲-۱ تحلیل زمانی الگوریتم مرتب‌سازی درجی (Insertion Sort)

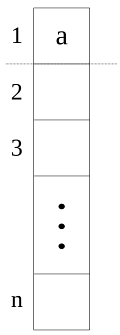
فرض:

۱. اندیس‌ها از ۱ شروع می‌شوند.

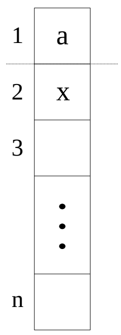
۲. اعداد به صورت صعودی مرتب می‌شوند.

۳. اگر زمان اجرای هر خط کد با C نمایش داده شود، در تعداد اجرای آن ضرب می‌شود و زمان اجرای هرخط حساب می‌شود.

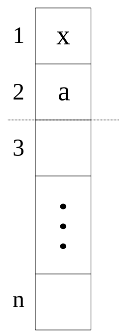
مثال: می‌خواهیم عناصر ...a,x,y را که درون یک آرایه قرار دارد مرتب کنیم. می‌دانیم که ... >a > y > x است. نحوه مرتب سازی درجی این عناصر با رسم شکل در زیر نمایش داده شده است.



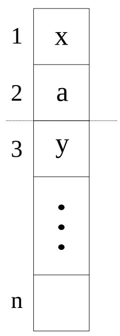
اولین عنصر آرایه را در نظر می گیریم.
این عنصر به تنهایی مرتب است.



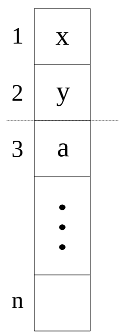
دومین عنصر را وارد آرایه می کنیم و
با عنصر قبلی مقایسه می کنیم.



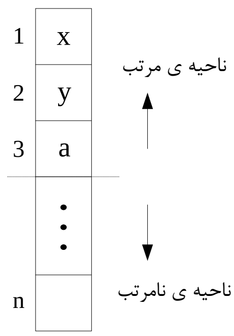
x با a مقایسه می شود و چون کوچک تر
از آن است جایشان عوض می شود.



سومین عنصر را وارد آرایه می کنیم و با
هر دو عنصر قبلی مقایسه می کنیم.



y با a مقایسه می شود و چون کوچک تر
از آن است جایشان عوض می شود.



y با x مقایسه می شود و چون بزرگ تر از
آن است جایشان عوض نمی شود.

برای سایر عناصر آرایه هم به همین ترتیب عمل می‌کنیم. سایر عناصر آرایه را نیز به ترتیب اولیه وارد کرده و آن را با تمامی عناصر مرتب قبلی مقایسه کرده در صورت نیاز جابه‌جا می‌کنیم. این عمل را تا جایی ادامه می‌دهیم که همه‌ی عناصر مرتب شده و در جای اصلی خود قرار بگیرند.

```
void  insertionsort  (int N, elementType A[ ])
{
   int  i,k;
   elementType x;
   for (k=2; k<=N; k++)   →  C1 * N
        {
        x= A[k];   →  C2 * (N − 1)
        i=k-1;   →  C3 * (N − 1)
        while(x<A[i]  &&  i>0)   →  C4 * k
                {
                A[i+1]=A[i];   →  C5 * (k − 1)      → کل حلقه (N − 1) بار اجرا می‌شود
                i=i-1;    →  C6 * (k − 1)
                }
        A[i+1]=x;   →  C7 * (N − 1)
        }
 }
```

❖ مدت زمان اجرای دستورات while متغییر بوده و به مقدار i بستگی دارد، بنابراین با تغییر i حالت‌های متفاوتی بوجود می‌آید. ابتدا زمان اجرای این الگوریتم را برای بدترین حالت بررسی می‌کنیم.

- **تحلیل در بدترین حالت:** بدترین حالت زمانی است که مقادیر درون آرایه به صورت نزولی مرتب باشند. در این صورت هم مقایسه انجام می‌شود و هم جابجایی صورت می‌گیرد و تمام عناصر در هر مرحله جابه‌جا می‌شوند.

برای محاسبه زمان اجرای بدترین حالت باید بتوانیم بین مدت زمان اجرای دستورات حلقه while و دستورات حلقه for رابطه‌ای بیابیم، یعنی باید k را برحسب N بدست آوریم.

$\sum_{k=2}^{N} k = 2 + 3 + \cdots + N = \frac{N(N+1)}{2} - 1 \quad \rightarrow$ زمان اجرای شرط while

$\sum_{k=2}^{N}(k-1) = \sum_{k=2}^{N} k - \sum_{k=2}^{N} 1 = \left[\frac{N(N+1)}{2} - 1\right] - [N-1] = \frac{N(N-1)}{2} \quad \rightarrow$ زمان اجرای بدنه while

در این حالت مرتبه‌ی زمانی N^2 است. $T_W(N) = AN^2 + BN + C = O(N^2)$

- **تحلیل در بهترین حالت:** بهترین حالت زمانی است که مقادیر درون آرایه به‌صورت صعودی مرتب باشند. در این صورت فقط مقایسه انجام می‌شود و جابه‌جایی صورت نمی‌گیرد، بنابراین وارد بدنه‌ی حلقه while نمی‌شود و فقط شرط دستور while را (N-1) بار اجرا می‌کند. $T_B(N) = A'N + B' = O(N)$

- **تحلیل درحالت متوسط:** حالت متوسط زمانی است که مقادیر درون آرایه نه صعودی و نه نزولی باشند، به عبارت دیگر حالت مشخصی نداشته باشند. این حالت نزدیک به بدترین حالت است، زیرا حتی اگر میانگین نیز بگیریم مرتبه‌ی زمانی $\frac{1}{2}AN^2$ می‌شود که عدد $\frac{1}{2}$ مقدار ثابتی است و تاثیری در مرتبه ندارد. $T_A(n) = O(n^2)$

❖ در تعیین مرتبه زمانی الگوریتم‌ها جمله با بالاترین درجه که نماینده یک چندجمله‌ای است مهم‌ترین عنصر است.

تحلیل جزئی برنامه‌ها نیاز به صرف وقت زیادی دارد و هم‌چنین دشوار است، علاوه بر آن باید به خوبی بدانیم که الگوریتم چگونه کار می‌کند، یعنی باید بتوانیم عملکرد آن را به صورت جزئی بیان کنیم. به دلیل این‌که فقط جمله با بالاترین درجه برای ما مهم است، بنابراین ما با انجام تحلیل جزئی کاری اضافی انجام می‌دهیم. با توجه به معایب تحلیل جزئی از آن به ندرت استفاده می‌کنیم و به سراغ تحلیل کلی برنامه‌ها می‌رویم.

برای انجام تحلیل کلی که فقط جمله با بالاترین درجه را به‌دست می‌آورد نیاز به شناخت نمادهای O, θ, Ω است.

۱-۳ تحلیل از دیدگاه ریاضی

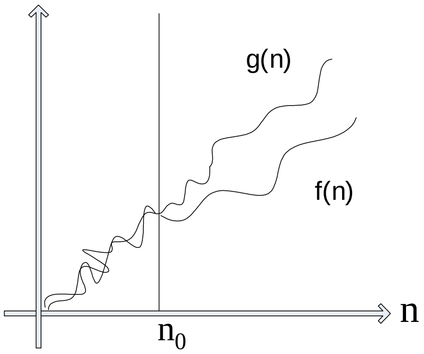
۱-۳-۱ نمادها

۱. **نماد O**

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

$$\{\exists c, n_0 > 0,\ \forall n \ge n_0 : 0 \le f(n) \le cg(n)\}$$

این تعریف به این معناست که $g(n)$ حد بالای تابع $f(n)$ است.



$$f(n) = O(g(n)) \implies \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \ , \quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$

برای به‌دست آوردن تقریب درست‌تر و نزدیک‌تر به واقعیت کوچکترین کران بالا را در نظر می‌گیریم، یعنی:

$$f(n) = O(n^2) = O(n^3) = O(n^4) = \cdots = O(n^n) \quad \Longrightarrow \quad f(n) = O(n^2)$$

همواره درست است

$$\begin{cases} f(n) = O(n^2) & \Longrightarrow \quad f(n) = O(n^3) \\ f(n) = O(n^2) & \Longleftarrow \quad f(n) = O(n^3) \end{cases}$$

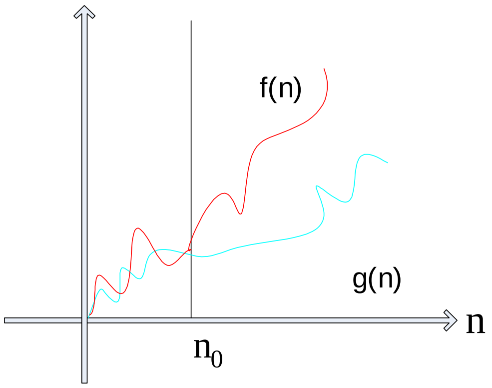
لزوماً درست نیست

۲. نماد Ω

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

$$\{\exists c, n_0 > 0,\ \forall n \geq n_0:\ 0 \leq cg(n) \leq f(n)\}$$

این تعریف به این معناست که $g(n)$ حد پائین تابع $f(n)$ است.



$$f(n) = \Omega(g(n)) \quad \Longrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \quad , \quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

لزوماً درست نیست

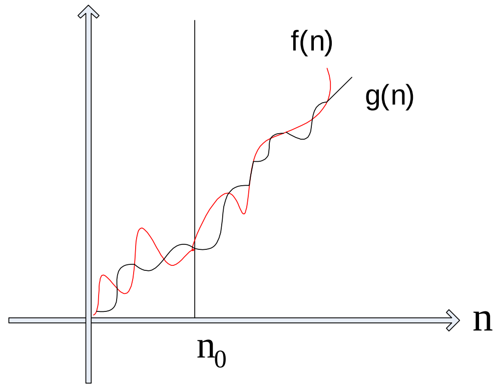
$$\begin{cases} f(n) = \Omega(n^2) & \Longrightarrow \quad f(n) = \Omega(n^3) \\ f(n) = \Omega(n^2) & \Longleftarrow \quad f(n) = \Omega(n^3) \end{cases}$$

همواره درست است

۳. نماد $\boldsymbol{\Theta}$

$$\boldsymbol{f(n) = \theta\big(g(n)\big)}$$

$$\{\exists c_1, c_2, n_0 > 0,\ \forall n \geq n_0:\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$



$$f(n) = \theta\big(g(n)\big) \quad \Longrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{g(n)}{f(n)} = 1$$

$$f(n) = \theta(n^2) \ \underset{\text{غلط}}{\overset{\text{غلط}}{\rightleftarrows}} \ f(n) = \theta(n^3)$$

۲-۳-۱ رشد مرتبه زمانی(Order)

$$\underbrace{O(1)}_{O(\text{عدد ثابت})} < O(\log_2 n) < O(n) < O(n \log n) < O(n^2) < \cdots < O(2^n) < O(n!) < O(n^n)$$

در تعیین رشد مرتبه زمانی رشد تابع مهم است و نه شکل تابع

n	n^2	n^3
1	1	1
2	4	8
3	9	27
4	16	64
⋮	⋮	⋮

با زیاد شدن مقدار n رشد تابع n^3 سریعتر از تابع n^2 می شود.

مثال: مرتبه زمانی توابع زیر را تعیین کنید.

1) $f(n) = \sqrt{n} \quad \rightarrow \quad f(n) = O\left(n^{\frac{1}{2}}\right)$

2) $f(n) = \log n \quad \rightarrow \quad f(n) = O(\log n)$

3) $f(n) = \log^2 n \quad \rightarrow \quad f(n) = O((\log n)^2)$

4) $f(n) = \log n^2 \quad \rightarrow \quad f(n) = O(2\log n) = O(\log n)$

5) $f(n) = \log n^k \quad \rightarrow \quad f(n) = O(k\log n) = O(\log n)$

۲-۳-۱ یک جمع‌بندی کلی

$$\boldsymbol{a = f(n)}\,,\boldsymbol{b = g(n)} \qquad \begin{cases} f(n) = O\big(g(n)\big) \rightarrow a \le b \\ f(n) = \theta\big(g(n)\big) \rightarrow a = b \\ f(n) = \Omega\big(g(n)\big) \rightarrow a \ge b \\ f(n) = o\big(g(n)\big) \rightarrow a < b \\ f(n) = \omega\big(g(n)\big) \rightarrow a > b \end{cases}$$

۱-۴ تعیین مرتبه زمانی بدون تحلیل جزئی

برنامه‌ها به دو صورت هستند:
- بازگشتی (Recursive)
- غیر بازگشتی (Non Recursive)

۱-۴-۱ تحلیل برنامه‌های غیربازگشتی

در برنامه‌های غیر بازگشتی برای تحلیل زمانی برنامه، مرتبه‌ی زمانی هر بلوک را مشخص می‌کنیم. ماکسیمم مرتبه زمانی بلوک‌ها، مرتبه‌ی زمانی کل برنامه است.

$$f(n) = \begin{cases} [B_1] = O(g_1(n)) \\ [B_2] = O(g_2(n)) \\ [B_3] = O(g_3(n)) \\ \vdots \\ [B_n] = O(g_n(n)) \end{cases}$$

$f(n) = O\left(Max(g_1(n), g_2(n), g_3(n), \cdots, g_n(n))\right) \rightarrow$ مرتبه زمانی کل برنامه

۱-۴-۱-۱ تحلیل بلوک مبتنی بر حلقه

for (i=1; i<=n; i++) → n مرتبه

{ … }

مرتبه‌ی زمانی $O(n)$ است.

اگر n عدد ثابتی باشد مرتبه‌ی زمانی $O(1)$ است

مثال: مرتبه زمانی (order) حلقه‌های تودرتو.

for (i=1; i<=n; i++) → n مرتبه

for (j=1; j<=n; j++) → n مرتبه

{…}

مرتبه زمانی $O(n^2)$ می‌شود.

$$\underbrace{n + n + n + \cdots + n}_{\text{تا } n} = n \times n = n^2$$

اگر در حلقه‌ی درونی بجای n مقدار m قرار دهیم، مرتبه زمانی $O(n.m)$ می‌شود.

اگر در حلقه‌ی درونی بجای n مقدار i قرار دهیم، مرتبه زمانی طبق فرمول زیر $O(n^2)$ می‌شود.

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

مثال:

```
for (i=1; i<=n; i++)   →  n مرتبه
  {
     j = n;
     while (j>1)
          j = j/3;
  }
```

$$\rightarrow \quad \frac{n}{3^k}=1 \quad \Rightarrow \quad n=3^k \quad \Rightarrow \quad k=\log_3 n$$

مرتبه‌ی زمانی کل $O(n\log_3 n)$ است.

۲-۱-۴-۱ تحلیل مبتنی بر بلوک شرطی

مرتبه زمانی بدنه if و else را جداگانه بدست می‌آوریم، چون بسته به شرط هر بار فقط یکی از آنها اجرا می‌شوند، مرتبه زمانی کل بلوک شرطی ماکسیمم مقدار آنها می‌شود.

```
if (…..)
   {
   ⋮        O(g1(n))
   }
else
   {
   ⋮        O(g2(n))
   }
```

$$f(n)=O\left(Max(g_1(n),g_2(n))\right) \rightarrow$$ مرتبه زمانی کل شرط

۲-۴-۱ تحلیل برنامه‌های بازگشتی

برنامه‌های بازگشتی ماهیت استقرایی دارند. برای تحلیل این‌گونه برنامه‌ها به شیوه استقرایی عمل می‌کنیم، بدین معنی که در ابتدا الگوریتم بازگشتی را برای آن طراحی کرده و رابطه بازگشتی را می‌نویسیم، سپس حالت پایه الگوریتم را مشخص می‌کنیم و در نهایت رابطه بازگشتی را حل می‌کنیم.

مسئله بازگشتی: مسئله‌ای است که برای حل آن نیاز به حل همان مسئله در ابعاد کوچک‌تر است.

۱-۲-۴-۱ تابع فاکتوریل

تعریف فاکتوریل
$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

تعریف بازگشتی تابع فاکتوریل
$$n! = n * (n-1)!$$

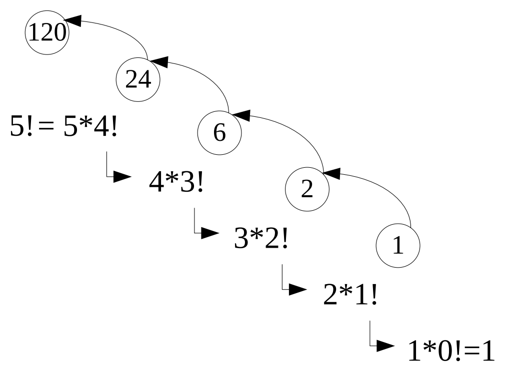
$\Lsh \quad (n-1) * (n-2)!$

$\Lsh \quad \cdots$

$\Lsh \quad 1! = 1$

$\Lsh \ 1 * 0! = 1 \quad \rightarrow$ شرط پایه

مثال: نمایش محاسبه مقدار !5 به‌صورت بازگشتی.



الگوریتم و رابطه بازگشتی تابع فاکتوریل

```
int  fact (int n)
  {
    if (n==0)      return 1;
    return     n* fact (n-1);
  }
```

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

$\Lsh$ عملیات ضرب

حل رابطه بازگشتی تابع فاکتوریل

1: $T(n) = T(n-1) + 1$

2: $T(n-1) = T(n-2) + 1 \quad \Rightarrow \quad T(n) = T(n-2) + 1 + 1$

3: $T(n-2) = T(n-3) + 1 \quad \Rightarrow \quad T(n) = T(n-3) + 1 + 1 + 1$

⋮

k: $T(n-k+1) = T(n-k) + 1 \quad \Rightarrow \quad T(n) = T(n-k) + 1 + \underbrace{1 + \cdots + 1}_{\text{k مرتبه}}$

$T(1) = 1 \quad \Rightarrow \quad n-k = 1 \quad \Rightarrow \quad k = n-1 \quad \Rightarrow \quad T(n) = T(n-n+1) + 1 + \underbrace{1 + \cdots + 1}_{\text{(n-1) مرتبه}}$

$\Longrightarrow \quad T(n) = T(1) + n - 1 \quad \Longrightarrow \quad T(n) = n \quad \Longrightarrow \quad \boxed{T(n) = O(n)}$

۲-۲-۴-۱ سری فیبوناچی

هدف به‌دست آوردن مقدار جمله nام فیبوناچی است.

الگوریتم و رابطه بازگشتی سری فیبوناچی

$n = 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad \ldots$

$\quad\;\; 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad \ldots$

```
int  fibo (int n)
  {
    if (n==1 || n==2)      return 1;
     return     fibo(n-1)+fibo(n-2);
   }
```

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

عملیات جمع

مثال: مرتبه زمانی تکه برنامه‌های زیر را بدون تحلیل جزئی بدست آورید.

```
int  f1 (int n)
  {
    if (n==1)      return 1;
    return     f1(n-1) + f1(n-1);
  }
```

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

1: $\quad T(n) = 2T(n-1)+1$

2: $\quad T(n-1) = 2T(n-2)+1 \quad \Rightarrow \quad T(n) = 2\times 2\,T(n-2)+2+1$

3: $\quad T(n-2) = 2T(n-3)+1 \quad \Rightarrow \quad T(n) = 2\times 2\times 2\,T(n-3)+4+2+1$

$\vdots$

k: $\quad T(n-k+1) = 2T(n-k)+1 \;\Rightarrow\; T(n) = \underbrace{2\times 2\times\cdots\times 2}_{k\ \text{مرتبه}}\,T(n-k)+2^{k-1}+\cdots+2^0$

$$T(n) = 2^k\,T(n-k)+\sum_{i=0}^{k-1} 2^i$$

$$T(1)=1 \quad\Rightarrow\quad n-k=1 \quad\Rightarrow\quad k=n-1 \quad\Rightarrow\quad T(n) = 2^{(n-1)}\,T(n-n+1)+\sum_{i=0}^{n-2} 2^i$$

$$T(n) = 2^{(n-1)}\underbrace{T(1)}_{1}+\sum_{i=0}^{n-2} 2^i = \sum_{i=0}^{n-1} 2^i = \underbrace{2^{n-1}+2^{n-2}+\cdots+2^1+2^0}_{n\ \text{جمله}} = 2^n-1 = O(2^n)$$

$$\boxed{T(n) = O(2^n)}$$

یاد آوری: با فرمول تصاعد هندسی زیر می‌توان راحت‌تر به جواب رسید.

تصاعد هندسی : $\dfrac{t_1(1-q^n)}{1-q} \quad\Rightarrow\quad \displaystyle\sum_{i=0}^{n-1} 2^i = \frac{1\times(1-2^n)}{1-2} = 2^n-1$

در این‌جا t_1 جمله اول سری و q قدر نسبت است.

```
int  f2 (int n)
  {
    if (n==1)      return 1;
    return        2* f2(n-1);
  }
```

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

1: $T(n) = T(n-1) + 1$

2: $T(n-1) = T(n-2) + 1 \quad \Rightarrow \quad T(n) = T(n-2) + 1 + 1$

3: $T(n-2) = T(n-3) + 1 \quad \Rightarrow \quad T(n) = T(n-3) + 1 + 1 + 1$

⋮

k: $T(n-k+1) = T(n-k) + 1 \quad \Rightarrow \quad T(n) = T(n-k) + \underbrace{1 + 1 + \cdots + 1}_{k \text{ مرتبه}}$

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

$T(1) = 1 \quad \Rightarrow \quad n - k = 1 \quad \Rightarrow \quad k = n - 1 \quad \Rightarrow \quad T(n) = T(n - n + 1) + n - 1$

$\Rightarrow \quad T(n) = \underbrace{T(1)}_{1} + n - 1 \quad \Rightarrow \quad T(n) = n \quad \Rightarrow \quad \boxed{T(n) = O(n)}$

با توجه به نتیجه مسائل قبل و نتایج بسیاری از مسائل مشابه، فرمول‌های زیر بدست آمده‌اند که برای بدست آوردن مرتبه زمانی مسائلی با این فرم، می‌توان از آنها استفاده کرد.

$$\begin{cases} T(n) = K\,T(n-1) + C = O(K^n) & \forall\ K \geq 2 \\ T(n) = K\,T(n-B) + C = O\left(K^{\frac{n}{B}}\right) & \forall\ K \geq 2, B \geq 1 \end{cases}$$

حال با استفاده از فرمول‌های بالا می‌توانیم مرتبه زمانی سری فیبوناچی را که در صفحه قبل گفته شد به صورت تقریبی بدست آوریم.

$T(n) = T(n-1) + T(n-2) + 1$

داریم: $T(n-2) \leq T(n-1)$

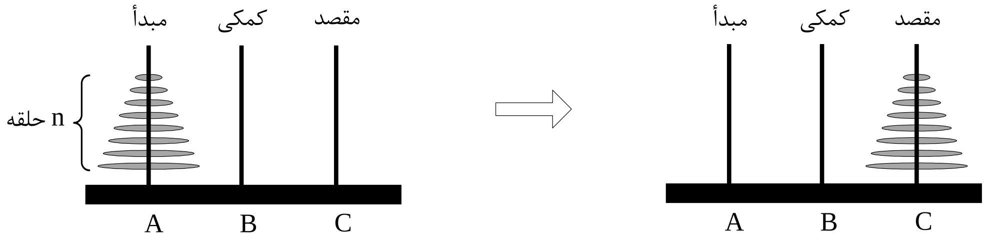
$T(n) \leq T(n-1) + T(n-1) + 1 \leq 2T(n-1) + 1 \quad \Rightarrow \quad T(n) = O(2^n)$

✓ **تمرین: مقدار دقیق سری فیبوناچی را به‌دست آورید.**

۵-۱ برج‌های هانوی

۱-۵-۱ برج هانوی۱

در مسئله برج هانوی۱ هدف ارائه الگوریتمی است که به وسیله آن بتوان تعداد n دیسک را با کمک گرفتن از میله B و با همان ترتیب اولیه، از میله A به میله C انتقال داد.



شرط‌های اساسی در روش برج‌های هانوی

۱) برای جابجایی دیسک‌ها فقط از میله‌ها می‌توان استفاده کرد.
۲) در هر مرحله فقط ۱ دیسک را می‌توان جابجا کرد.
۳) در مراحل میانی نباید هیچ دیسک بزرگ‌تری روی دیسک کوچک‌تر از خودش قرار بگیرد.

مسئله برج‌های هانوی را به‌صورت استقرایی (بازگشتی) تعریف می‌کنیم:

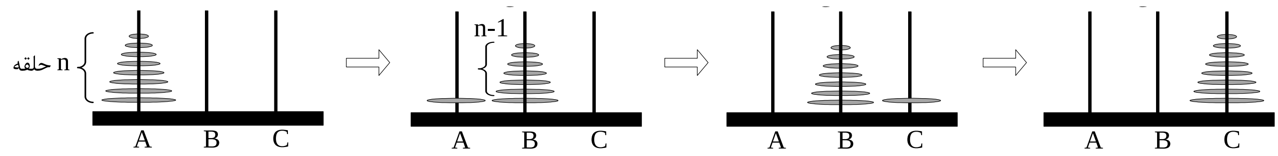
شرط پایه: فقط یک دیسک داریم که روی میله مبدأ قرار گرفته است (n=1) که n تعداد دیسک‌ها است. یک دیسک را می‌توانیم مستقیم از مبدأ به مقصد (A→C) منتقل کنیم.

فرض: مسئله برای n-1 حل شده است، یعنی می‌دانیم n-1 دیسک را چگونه منتقل کنیم.

حکم: انتقال n دیسک به مقصد.

حل مسئله برج‌های هانوی به روش غیر بازگشتی بسیار دشوار است. این مسئله به روش بازگشتی به صورت زیر حل می‌شود:

- **مرحله اول:** (n-1) دیسک به روش بازگشتی از میله مبدا به میله کمکی منتقل می‌شود (طبق فرض).
- **مرحله دوم:** به شرط پایه رسیدیم که باید ۱ دیسک باقی مانده در میله مبدا به میله مقصد منتقل شود.
- **مرحله سوم:** (n-1) دیسک از روی میله کمکی به میله مقصد منتقل می‌شود (طبق فرض).



الگوریتم و رابطه بازگشتی مسئله برج هانوی[1]

مرتبه زمانی این الگوریتم (تعداد عملیات) برابر تعداد جابه‌جایی دیسک‌ها می‌باشد.

```
تعداد دیسک‌ها   مبدأ   کمکی   مقصد
Hanoi (n, A, B, C)
 {
   if (n==1)     move (A→C);            1
   else  {
          Hanoi (n-1, A, C, B);         T(n-1)
          Move (A→C);                   1
          Hanoi (n-1, B, A, C);         T(n-1)
         }
 }
```

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

مرتبه زمانی الگوریتم برج‌های هانوی $O(2^n)$ است.

آرگومان دوم همواره مبدأ، سوم همواره کمکی و چهارم همواره مقصد است. نام کاراکتر مهم نیست بلکه مکان آن مهم است. مثلاً هر کاراکتری که در آرگومان دوم باشد حکم مبدأ را دارد.

با فرض تعداد ۶۴ دیسک (n=۶۴) و این‌که فرض کنیم جابه‌جایی و انتقال هر دیسک فقط ۱ ثانیه طول می‌کشد (حالت بسیار ایده‌آل)، انتقال ۶۴ دیسک به میله مقصد چند سال طول می‌کشد؟

تقریباً <u>۶۰۰ میلیارد</u> سال

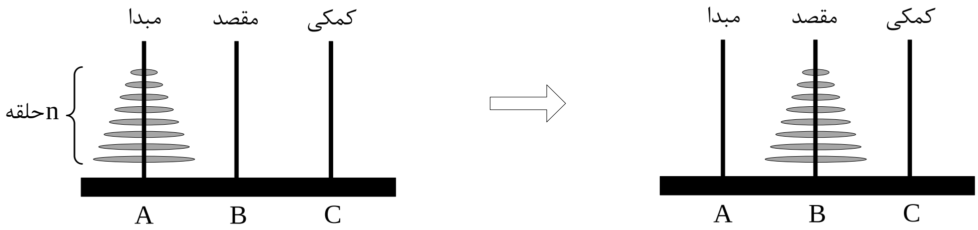
تعداد دقیق جابه‌جایی‌ها: $2^{64}=18446744073709551615$

۲-۵-۱ برج هانوی۲

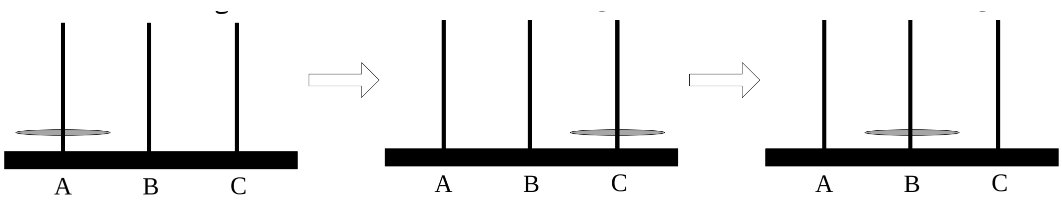
در این مسئله جدید از برج‌های هانوی، ۲ شرط قرار داده شده است:

۱. انتقال مستقیم نداریم.
۲. میله‌ی مقصد میله‌ی B خواهد بود.

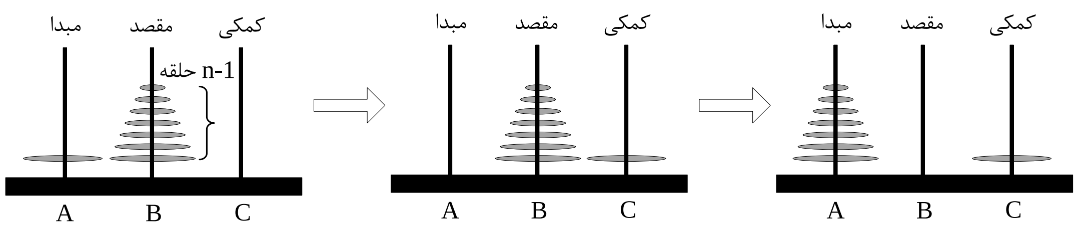
صورت مسئله: می‌خواهیم n حلقه را به‌صورت بازگشتی و با کمک گرفتن از میله C از مبدا به مقصد انتقال دهیم.

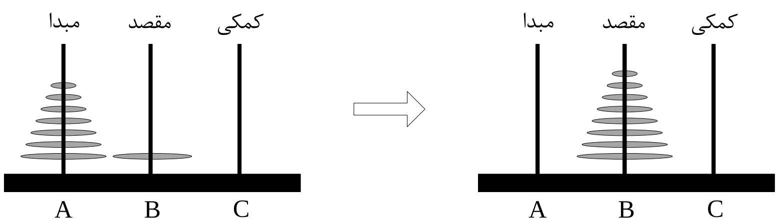


شرط پایه: اگر یک حلقه در مبدا باقی ماند آن را ابتدا به میله کمکی و سپس به میله مقصد انتقال می‌دهیم.



راه حل:





برنامه‌ی برج هانوی۲

```
Hanoi2 (n,A,B,C)
 {
   if (n==1)
         {  move (A→C)                          1
            move (C→B)  }                       1
   else
         {  Hanoi2 (n-1,A,B,C)                  T(n-1)
            move (A→C)                           1
            Hanoi2 (n-1,B,A,C)                  T(n-1)
            move (C→B)                           1
            Hanoi2 (n-1,A,B,C)  }               T(n-1)
 }
```

رابطه‌ی بازگشتی مسئله برج هانوی۲ به‌صورت زیر است:

$$T(n) = \begin{cases} 2 & n = 1 \\ 3T(n-1) + 2 & n > 1 \end{cases}$$

1: $T(n) = 3T(n-1) + 2$

2: $T(n) = 3[3T(n-2) + 2] + 2 = 3^2T(n-2) + (3^1 \times 2) + 2$

3: $T(n) = 3^2[3T(n-3) + 2] + (3^1 \times 2) + 2 = 3^3T(n-3) + (3^2 \times 2) + (3^1 \times 2) + (3^0 \times 2)$

$\vdots$

k: $T(n) = 3^kT(n-k) + (3^{k-1} \times 2) + \cdots + (3^1 \times 2) + (3^0 \times 2)$

$T(n) = 3^kT(n-k) + 2[3^{k-1} + 3^{k-2} + \cdots + 3^1 + 3^0]$

$T(1) = 2 \quad \rightarrow \quad n - k = 1 \quad \rightarrow \quad k = n - 1$

$T(n) = 3^{n-1}T(1) + 2[3^{n-2} + \cdots + 3^1 + 3^0] = 2 \times [3^{n-1} + 3^{n-2} + \cdots + 3^1 + 3^0]$

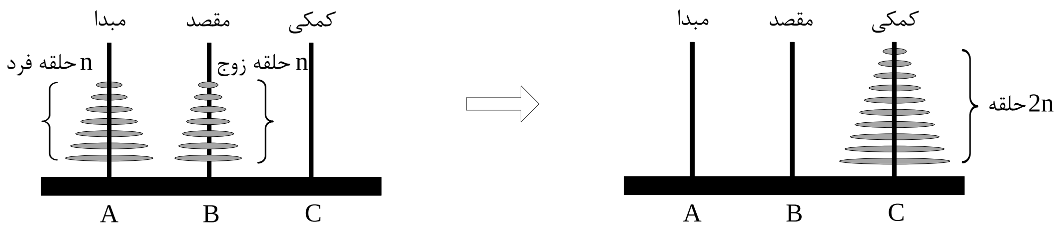
$$T(n) = 2 \times \sum_{i=0}^{n-1} 3^i = 2 \times \frac{1 - 3^n}{1 - 3} = 3^n - 1 \qquad \Longrightarrow \qquad \boxed{T(n) = 3^n - 1 \quad \rightarrow \quad O(3^n)}$$

۳-۵-۱ برج هانوی توسعه یافته

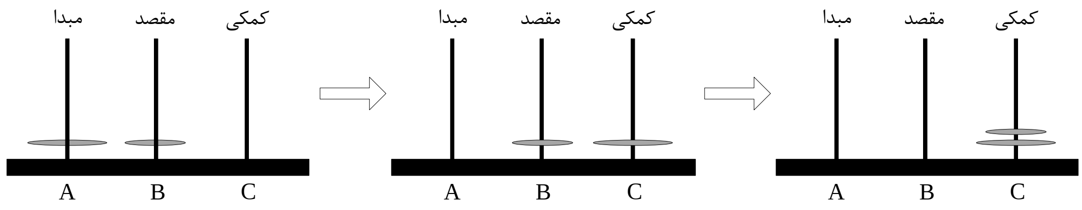
در این مسئله قوانین برج هانوی۱ صادق است با این تفاوت که n حلقه فرد در یک میله و n حلقه زوج در میله دیگر داریم که اندازه حلقه‌ها به صورت زیر تغییر می‌کند:

اندازه : $1 > 2 > 3 > 4 > \cdots > 2n - 1 > 2n$

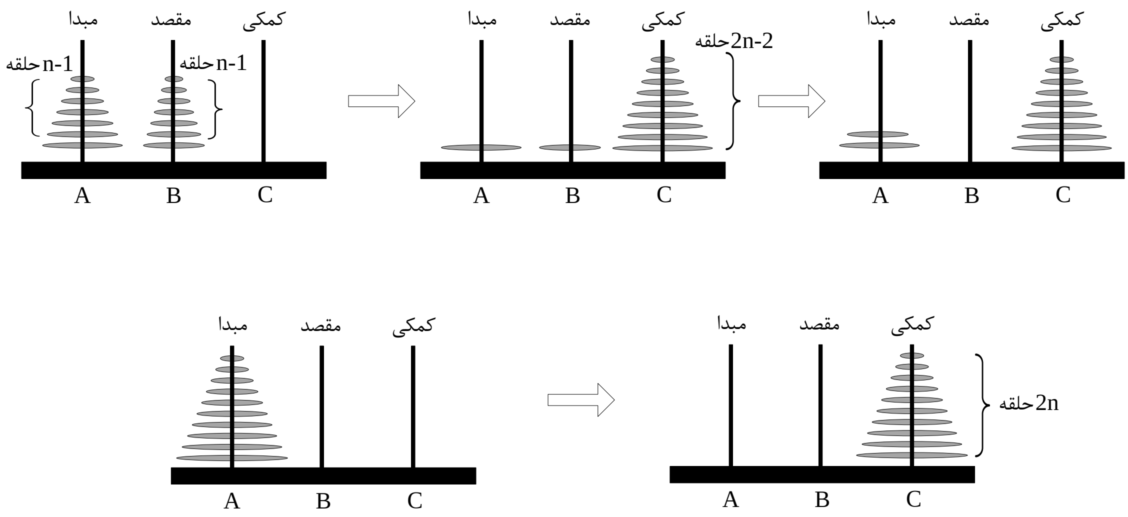
صورت مسئله: n حلقه فرد و n حلقه زوج را از مبدا به میله‌ی C انتقال می‌دهیم.



شرط پایه: اگر یک حلقه زوج و یک حلقه فرد در مبدا باقی بماند ابتدا حلقه فرد و سپس حلقه زوج را به مقصد انتقال می‌دهیم.



راه حل:



برنامه‌ی برج هانوی توسعه یافته

```
Ehanoi (n,A,B,C)
 {
  if (n==2)
        {  move (A→C)                        1
            move (B→C)  }                    1
  else
        {  Ehanoi (n-1,A,B,C)               T(n-1)
            move (B→A)                        1
            hanoi (2n-2,C,B,A)              2^(2n-2)-1
            hanoi (2n,A,B,C)  }             2^(2n)-1
 }
```

رابطه‌ی بازگشتی مسئله برج هانوی توسعه یافته

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

$$\boxed{T(n) = T(n-1) + \frac{4^n}{4} + 4^n \quad \Longrightarrow \quad O(4^n)}$$

- ✓ **سئوال: مرتبه اجرایی برج هانوی توسعه یافته را به‌دست آورید.**

فصل دوم: روش‌های طراحی الگوریتم‌ها

۲-۱ استقراء ریاضی

انواع استقراء
- ۱. استقراء ساده
- ۲. استقراء قوی

هر دو استقراء ساده و قوی از سه قسمت شرط پایه، فرض و حکم تشکیل می‌شوند. تفاوت استقراء ساده و قوی در فرض آنها است.

۱) **شرط پایه:** مسئله به ازاء کوچک‌ترین مقدار ممکن برای n اثبات می‌شود.

۲) **فرض**
- **استقراء ساده:** فرض می‌کنیم مسئله به ازاء (n-1) برقرار است.
- **استقراء قوی:** فرض می‌کنیم مسئله به ازاء $n' < n$ برقرار است.

۳) **حکم:** سعی می‌کنیم با استفاده از فرض استقراء و فرضیات بدیهی موجود حکم را ثابت کنیم.

۲-۲ اثبات سازنده (Constructive)

روشی است که علاوه بر اثبات مسئله راه حل آن را هم ارائه می‌کند.

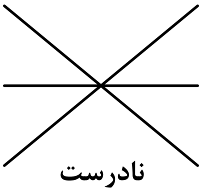
❖ مسئله

الف) تعداد نواحی حاصل از رسم n خط در صفحه را بدست آورید؟

ب) این نواحی با چند رنگ قابل رنگ آمیزی هستند، به‌صورتی که هیچ دو ناحیه مجاوری هم‌رنگ نباشند و کم‌ترین تعداد رنگ به کار رفته شود؟

شرط: هیچ دو خطی موازی نیستند و هیچ سه خطی از یک نقطه عبور نمی‌کنند.

نادرست

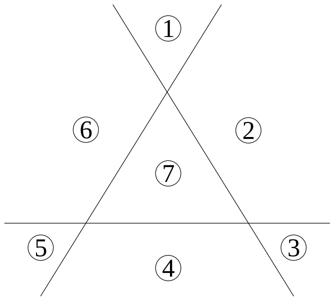


نادرست

راه حل قسمت الف: فرض می‌کنیم تعداد نواحی حاصل از رسم n خط در صفحه S(n) باشد.

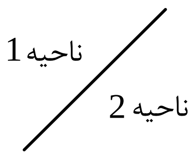
S(n) = تعداد نواحی حاصل از رسم n خط در صفحه

مثال: تعداد نواحی حاصل از رسم سه خط درصفحه در شکل زیر نشان داده شده است.



شرط پایه: به ازاء n=1 مسئله را اثبات می‌کنیم.

$$n = 1$$
$$S(1) = 2$$



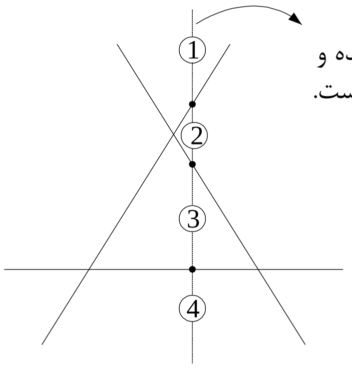
فرض: فرض می‌کنیم مسئله به ازاء S(n-1) حل شده است.

S(n-1) = x

حکم: مسئله را به ازاء S(n) اثبات می‌کنیم.

S(n) = S(n-1) + y = x + y

چون هیچ دو خطی موازی هم نیستند، بنابراین خط جدیدی که اضافه می‌شود همه‌ی خطوط را قطع خواهد کرد، یعنی خط n ام در (n-1) نقطه قطع می‌شود و در نتیجه n ناحیه‌ی جدید اضافه خواهد شد.



خط چهارم در سه نقطه قطع شده و
چهارم ناحیه جدید اضافه کرده اااست.

خط n ام نیز در(n-1) نقطه قطع می شود
و n ناحیه جدید اضافه می کند.

y = n

بنابراین داریم:

$$S(n)=S(n-1)+y \quad \Longrightarrow \quad \boxed{S(n)=S(n-1)+n}$$

$$1:\quad S(n)=S(n-1)+n$$

$$2:\quad S(n)=[S(n-2)+(n-1)]+n$$

$$3:\quad S(n)=\big[[S(n-3)+(n-2)]+(n-1)\big]+n$$

$$\vdots$$

$$k:\quad S(n)=S(n-k)+(n-k+1)+\cdots+(n-2)+(n-1)+n$$

$$S(1)=2 \quad \Longrightarrow \quad n-k=1 \quad \Longrightarrow \quad k=n-1$$

$$S(n)=S\big(n-(n-1)\big)+(n-(n-1)+1)+\cdots+(n-2)+(n-1)+n$$

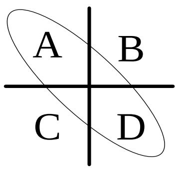
$$S(n)=\underbrace{S(1)}_{2}+2+3+\cdots+(n-2)+(n-1)+n$$

$$S(n)=\underbrace{2}_{1+1}+2+3+\cdots+(n-2)+(n-1)+n$$

$$S(n)=1+1+2+3+\cdots+(n-2)+(n-1)+n$$

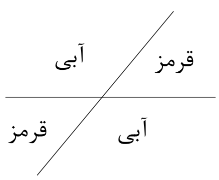
$$\boxed{S(n)=1+\frac{n(n+1)}{2}=\frac{1}{2}(n^2+n+2)}$$

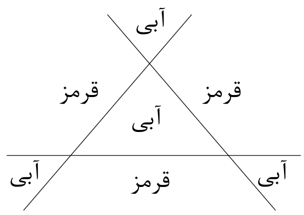
راه حل قسمت ب: دو ناحیه مجاور نواحی هستند که ضلع مشترک داشته باشند.



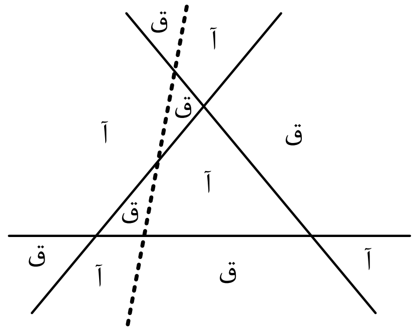
دو ناحیه A و B دارای ضلع مشترک هستند و مجاور محسوب می شوند.

دو ناحیه A و D مجاور نیستند.





وقتی خط n ام را رسم کردیم، رنگ‌های تمامی نواحی یک سمت خط را تا انتها معکوس می‌کنیم و رنگ‌های سمت دیگر خط را بدون تغییر می‌گذاریم. به این ترتیب با دو رنگ می‌توانیم کل نواحی را رنگ کنیم.



رنگ های سمت راست بدون تغییر باقی می مانند. ← → رنگ های سمت چپ خط جدید را تا انتها معکوس می کنیم.

بدترین حالت رنگ آمیزی زمانی است که خط جدید ایجاد شده وسط شکل قرار گیرد، در این حالت باید رنگ نصف نواحی تغییر کند. تعداد تعویض رنگ‌ها به ازاء رسم هر خط جدید به صورت زیر بدست می‌آید:

$$S'(n) = \frac{1}{2}\,S(n) = \frac{1}{2}\left[1 + \frac{n(n+1)}{2}\right] \quad \Longrightarrow \quad O(n^2)$$

✓ **سئوال: حداکثر تعداد نواحی حاصل از رسم n زاویه در صفحه را به‌دست آورید؟**

۳-۲ کد خاکستری (Gray Code)

هدف این روش تخصیص کد به n شیء متمایز است به‌صورتی که کدهای تخصیص داده شده به هر یک از اشیاء متوالی تنها در یک بیت اختلاف داشته باشند.

انواع کد خاکستری

۱. **باز:** هیچ رابطه‌ای بین کد شیء اول و آخر وجود ندارد. (در بیش از یک بیت اختلاف دارند)
۲. **بسته:** کد شیء اول و آخر تنها در یک بیت اختلاف دارند. (اشیاء به‌صورت دوار دیده می‌شوند)

نکته: با تعداد شیء فرد نمی‌توان Gray Code بسته نوشت. در این صورت خودمان یک عنصر به عناصر فرد اضافه می‌کنیم تا زوج شوند، مسئله را برای زوج حل کرده و کدگذاری می‌کنیم، در نهایت عنصر اضافه شده را حذف می‌کنیم.

❖ مسئله

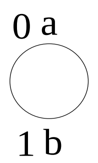
برای n شیء متمایز کدهایی اختصاص دهید که اشیاء پشت سرهم تنها در یک بیت اختلاف داشته باشند.

راه حل: باید الگوریتمی ارائه کنیم که علاوه بر حل مسئله، راه حل و پیچیدگی زمانی آن را هم بدست آورد.

فرض می‌کنیم که می‌خواهیم مسئله را برای تعداد اشیاء زوج حل کنیم (n=2k).

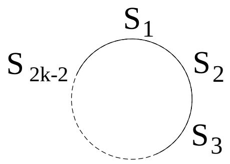
شرط پایه: مسئله را به ازاء k=1 اثبات می‌کنیم.

$k=1 \longrightarrow n=2$



فرض: فرض می‌کنیم برای تعداد (k-1)، یعنی (n=2k-2) مسئله حل شده است.

$k-1 \longrightarrow n=2k-2$

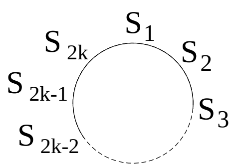


حکم: ثابت می‌کنیم برای تعداد k ، یعنی (n=2k) نیز مسئله برقرار است.

$k \longrightarrow n=2k$

کد S_1 = کد S_{2k}

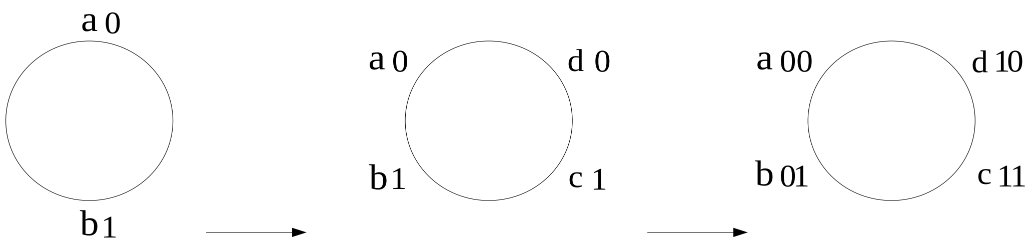
کد S_{2k-2} = کد S_{2k-1}



در هر مرحله دو عنصر به تعداد عناصر اضافه می‌کنیم و عمل کدگذاری را انجام می‌دهیم. برای اینکه دو عنصر اضافه شده با عناصر قبلی و بعدی خود در یک بیت اختلاف داشته باشند، کد عنصر آخر را برابر کد عنصر اول قرار می‌دهیم ($S_1=S_{2k}$) و همچنین کد عنصر یکی مانده به آخر را برابر کد عنصر قبلی‌اش قرار می‌دهیم ($S_{2k-1}=S_{2k-2}$). در هر مرحله به همه کدها یک یا صفر اضافه می‌کنیم. اگر به سمت راست کدها یک اضافه کردیم به سمت راست دو عنصر جدید صفر اضافه می‌کنیم و بالعکس. همین‌طور اگر به سمت چپ کدها یک اضافه کردیم به سمت چپ دو عنصر جدید نیز صفر اضافه می‌کنیم و بالعکس.

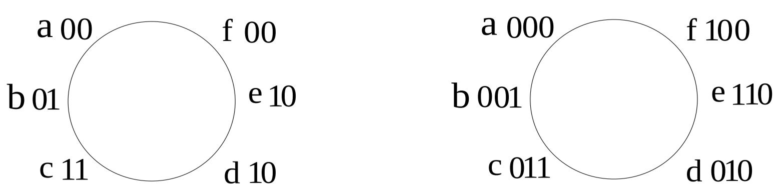
- **در حالت کلی برای n شیء به $\frac{n}{2}$ بیت برای کدگذاری نیاز است.**

مثال: برای شش حرف a,b,c,d,e,f کدهایی اختصاص دهید که فقط در یک بیت اختلاف داشته باشند.



دو عنصر اضافه می کنیم
کد a را به d و کد b را به c می دهیم.

به سمت چپ a و b صفر
و به سمت چپ c و d یک اضافه می کنیم.



دو عنصر اضافه می کنیم
کد a را به f و کد d را به e می دهیم

به سمت چپ عناصر قبلی صفر
و به سمت چپ e و f یک اضافه می کنیم.

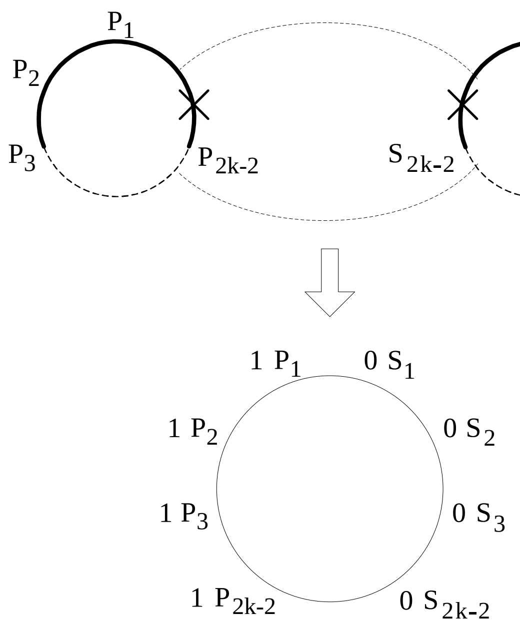
۱-۳-۲ حل مسئله در حالت بهینه

❖ مسئله

کد گذاری را طوری انجام دهید که کدگذاری n شیء با $\log_2 n$ بیت انجام شود.

راه حل: باید الگوریتمی ارائه کنیم که در آن برای کدگذاری هر بار تعداد عناصر نصف شوند. در این حالت فرض و حکم مانند حالت کلی است.

در هر مرحله تعداد عناصر را نصف می‌کنیم تا به شرط پایه که تعداد دو عنصر است برسیم. کدگذاری را برای دو عنصر انجام می‌دهیم، در مرحله بعد تعداد عناصر دو برابر می‌شوند، کدهای متناظر را با یکدیگر برابر قرار می‌دهیم. سپس به سمت راست یا چپ نیمی از کدها یک و به نیمی دیگر صفر اضافه می‌کنیم. این مراحل را تا پایان ادامه می‌دهیم.



P_1 کد = S_1 کد

P_2 کد = S_2 کد

⋮

P_{2k-2} کد = S_{2k-2} کد

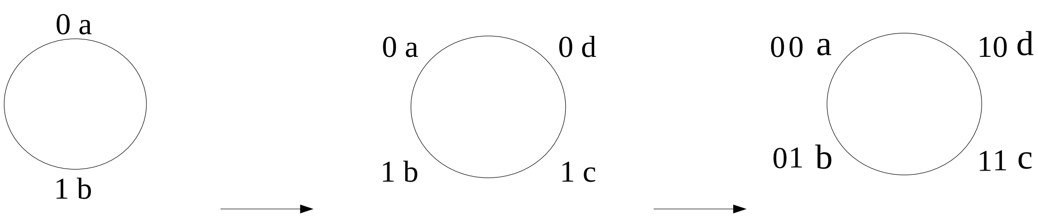
- **در حالت بهینه برای n شیء به $\lceil \log n \rceil$ بیت برای کدگذاری نیاز است.**

مثال: برای هشت حرف a,b,c,d,e,f,g,h کدهایی اختصاص دهید که فقط در یک بیت اختلاف داشته باشند.

تعداد عناصر را نصف می کنیم تا به دو عنصر برسیم

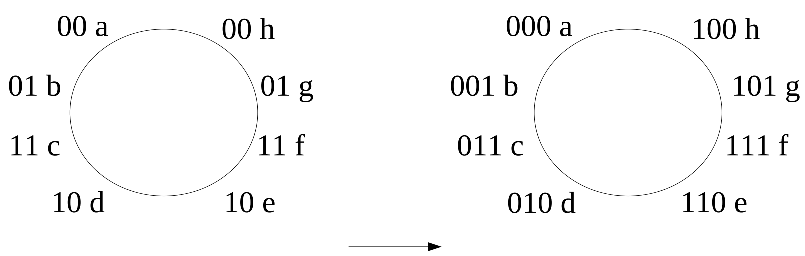
a b c d | e f g h

a b | c d



کدگذاری را برای دو عنصر انجام می دهیم
تعداد عناصر را دو برابر می کنیم و
نظیر به نظیر کدها را برابر قرار می دهیم

به سمت چپ کدهای قبلی صفر و به سمت
چپ کدهای جدید یک اضافه می کنیم



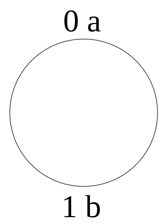
تعداد عناصر را دو برابر می کنیم، کدها را نظیر به نظیر برابر قرار می دهیم. به
سمت چپ کدهای قبلی صفر و به سمت چپ کدهای جدید یک اضافه می کنیم.

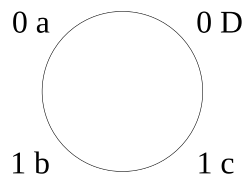
مثال: برای شش حرف a,b,c,d,e,f کدهایی اختصاص دهید که در یک بیت اختلاف داشته باشند(حالت بهینه).

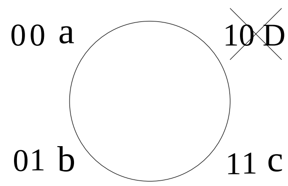
a b c | d e f

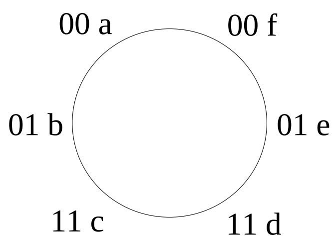
a b | c D

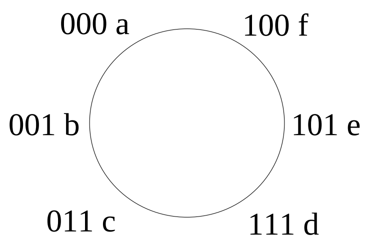
تعداد عناصر را نصف می کنیم تا به دو عنصر برسیم. چون تعداد عناصر بعدی زوج نیست خودمان یک عنصر اضافه می کنیم تا تعداد زوج شود. کدگذاری را انجام می دهیم و در نهایت کد اضافه شده را حذف می کنیم.











۲-۴ ارزیابی چند جمله‌ای‌ها

۲-۴-۱ روش معمولی

چند جمله‌ای زیر را در نظر بگیرید:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0 \qquad a_n \neq 0$$

برای ارزیابی این چند جمله‌ای مقدار آن را به ازای $x = x_0$ بدست می‌آوریم:

$$P_n(x_0) = a_n {x_0}^n + a_{n-1} {x_0}^{n-1} + \cdots + a_1 {x_0}^1 + a_0$$

برای ارزیابی یک چند جمله‌ای از درجه n تعداد ضرب‌ها و جمع‌های مورد نیاز به صورت زیر بدست می‌آیند:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$$

$$x^n = \underbrace{x \times x \times x \times \cdots \times x}_{\text{تا } n} \quad \Longrightarrow \quad \text{ضرب } n-1 \quad \Longrightarrow \quad a_n \times x^n \quad \Longrightarrow \quad \text{ضرب } n$$

$$x^{n-1} = \underbrace{x \times x \times x \times \cdots \times x}_{\text{تا } n-1} \quad \Longrightarrow \quad \text{ضرب } n-2 \quad \Longrightarrow \quad a_{n-1} \times x^{n-1} \quad \Longrightarrow \quad \text{ضرب } n-1$$

تعداد ضرب $= \boldsymbol{(n) + (n-1) + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2}}$

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$$

n

تعداد جمع = n

رابطه‌ی بازگشتی برای ارزیابی یک چندجمله‌ای از درجه n به روش معمولی به صورت زیر است:

$$P_n(x) = P_{n-1}(x) + a_n x^n$$

$$\begin{cases} \text{تعداد جمع} \quad T(n) = T(n-1) + 1 = n & \Longrightarrow \quad O(n) \\ \text{تعداد ضرب} \quad T(n) = T(n-1) + n = \dfrac{n(n+1)}{2} & \Longrightarrow \quad O(n^2) \end{cases}$$

۲-۴-۲ روش هرنر (Horner's Method)

می‌خواهیم با استفاده از روش هرنر تعداد ضرب‌ها را از n^2 به n کاهش دهیم.

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0 \qquad a_n \neq 0$$

در هر جمله از یک x فاکتور می‌گیریم:

$$P_n(x) = a_0 + x\left(a_1 + x\left(a_2 + x\left(a_3 + \cdots + x(a_{n-1} + x a_n)\right)\right)\cdots\right)$$

$$P'_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + \cdots + a_1$$

$$P_n(x) = x.P'_{n-1}(x) + a_0$$

رابطه‌ی بازگشتی برای ارزیابی یک چند جمله‌ای از درجه n به روش هرنر به صورت زیر است:

$$P_n(x) = x.P'_{n-1}(x) + a_0$$

$$\begin{cases} \text{تعداد جمع} \quad T(n) = T(n-1) + 1 = n \quad \Longrightarrow \quad O(n) \\ \text{تعداد ضرب} \quad T(n) = T(n-1) + 1 = n \quad \Longrightarrow \quad O(n) \end{cases}$$

- **با استفاده از روش هرنر تعدادضرب‌ها را از n^2 به n کاهش دادیم.**

	تعداد جمع	تعداد ضرب
روش معمولی	n	$\frac{n(n+1)}{2}$
روش هرنر	n	n

فصل سوم: روش تقسیم و حل (Divide & Conquer)

۳-۱ الگوریتم تقسیم و حل

در روش تقسیم و حل یک مسئله به زیر مسائل کوچک‌تر تقسیم می‌شود، هرکدام از زیر مسائل نیز به همین شیوه به زیر مسائل کوچک‌تر دیگر تقسیم می‌شوند و این روند تا جایی ادامه پیدا می‌کند که مسئله ساده شده و دیگر قابل تقسیم نباشد. می‌توان گفت روش تقسیم و حل یک روش بالا به پایین (Top-down) است. سپس زیر مسائل به صورت بازگشتی حل می‌شوند و نتایج حاصل از حل زیر مسائل بگونه‌ای ادغام می‌شوند که پاسخ مسئله‌ی اصلی بدست آید.

روش تقسیم و حل از سه مرحله زیر تشکیل شده است:

۱. **تقسیم:** در مرحله تقسیم به دنبال یافتن زیر مسائلی در دامنه مسئله‌ی اصلی هستیم بطوری که شرایط آنها بر مسئله‌ی اولیه منطبق باشد .
۲. **حل:** در مرحله حل زیر مسائل به صورت بازگشتی حل می‌شوند.
۳. **ادغام:** در مرحله ادغام پاسخ‌های بدست آمده از حل زیر مسائل بگونه‌ای ادغام می‌شوند که پاسخ کلی بدست آید.

هزینه ادغام + هزینه تقسیم = هزینه کل

نکات

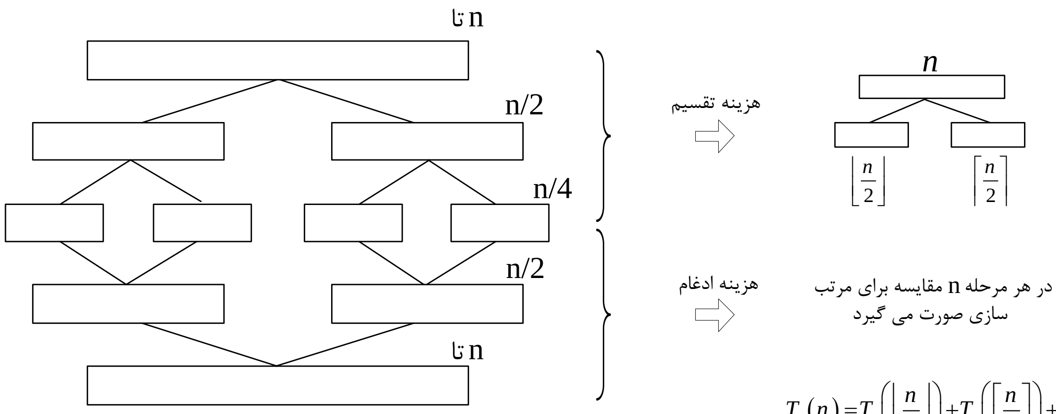
- در مرحله ادغام از کنار هم گذاشتن پاسخ زیر مسائل لزوماً جواب کلی بدست نمی‌آید.
- زیر مسائل از جنس مسئله‌ی اصلی هستند.
- معمولاً هزینه‌ی مرحله‌ی تقسیم عکس هزینه‌ی مرحله‌ی ادغام است.
- در مرحله تقسیم باید سعی کنیم مسئله را به‌گونه‌ای تقسیم کنیم که زیر مسائل بدست آمده کسری از مسئله‌ی اصلی باشند و نه تفریق، در این صورت تعداد مراحل کمتری برای محاسبه نیاز است.

$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \cdots$ مرحله $\log n$

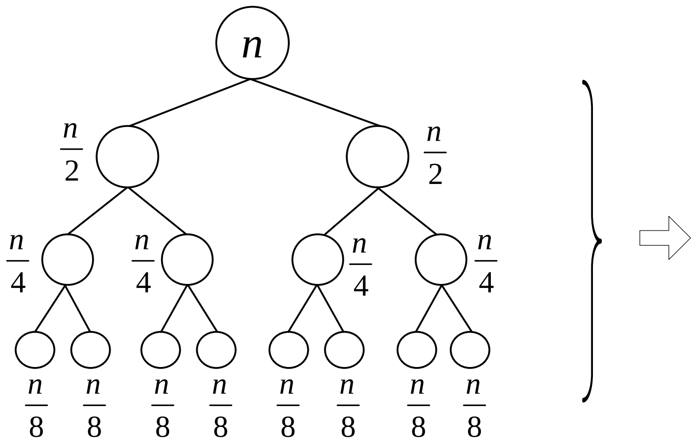
$n \rightarrow n-1 \rightarrow n-2 \rightarrow \cdots$ مرحله n

مثال: در شکل زیر می‌خواهیم n تایی نامرتب را به روش تقسیم و حل مرتب کنیم.

ابتدا n عنصر را به دو قسمت تقسیم می‌کنیم، سپس هر یک از دو قسمت به‌دست آمده را نیز جداگانه به دو قسمت تقسیم می‌کنیم و این عمل را از بالا به پایین تا جایی ادامه می‌دهیم که به تک عنصرها برسیم. سپس تک عنصرها را دو به دو مقایسه می‌کنیم و آن‌ها را مرتب کرده ادغام می‌کنیم و این عمل مقایسه، مرتب سازی و ادغام را از پایین به بالا به صورت عکس انجام می‌دهیم، تا جایی که n عنصر اولیه به‌صورت مرتب شده به‌دست آیند.



$$T(n)=\underbrace{T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)+T\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{تقسیم}}+\underset{\text{ادغام}}{n}$$



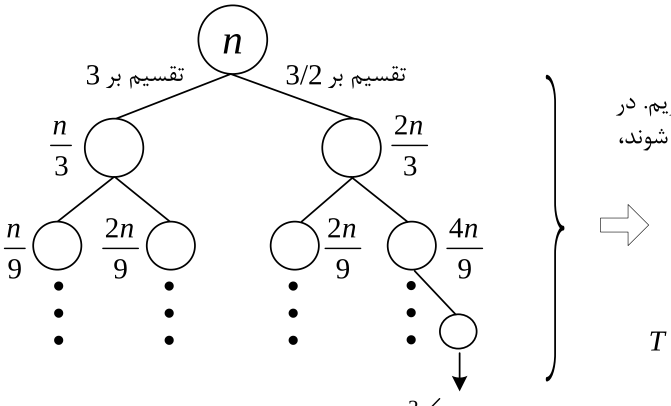
عناصر هر بار بر ۲ تقسیم می شوند و هزینه این کار از مرتبه $\log n$ است.

در هر مرحله عناصر با یکدیگر مقایسه شده و مرتب می شوند و هزینه این کار از مرتبه n است.

$$T(n)=2T\left(\frac{n}{2}\right)+n \Rightarrow O(n\log n)$$

✓ **تمرین: مقدار دقیق مرتبه زمانی را در مرتب سازی به روش تقسیم و حل (مثال بالا) به‌دست آورید.**

مثال: در شکل زیر می‌خواهیم n تایی نامرتب را به روش تقسیم و حل مرتب کنیم، با این تفاوت که عناصر در هر مرحله بر 3 و 3/2 تقسیم می‌شوند.



این شاخه هر بار تقسیم بر $3/2$ می شود، بنابراین دیرتر به 1 می رسد و طولانی ترین شاخه است.

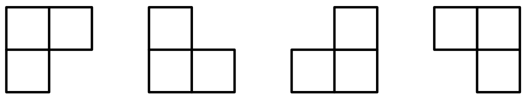
برای بررسی پیچیدگی زمانی بدترین حالت را در نظر می گیریم. در اینجا بدترین حالت زمانی است که عناصر بر $3/2$ تقسیم می شوند، بنابراین هزینه این کار از مرتبه $\log_{3/2} n$ است.

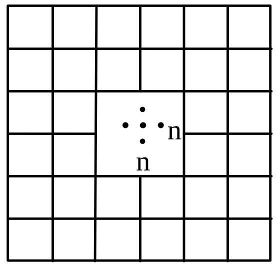
در هر مرحله عناصر با یکدیگر مقایسه شده و مرتب می شوند و هزینه این کار از مرتبه n است.

$$T(n)=T\left(\frac{n}{3}\right)+T\left(\frac{2n}{3}\right)+n \Rightarrow O(n\log_{3/2} n)$$

۳-۱-۱ مسئله موزاییک کردن سطح مربع (فرش کردن صفحه شطرنجی)

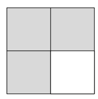
می‌خواهیم سطح مربعی به ضلع n را با موزاییک‌هایی با الگوهای زیر موزاییک کنیم.





راه حل: فرض می‌کنیم n توانی از 2 است ($n=2^k$). با استفاده از قضیه تقسیم و حل سعی می‌کنیم مسئله را به مسائل کوچک‌تر بشکنیم. مسائل کوچک را حل می‌کنیم و پاسخ‌ها را ادغام می‌کنیم تا جایی که مسئله اصلی حل شود.

شرط پایه: به ازاء $k=1$ مسئله را حل می‌کنیم.

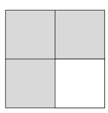


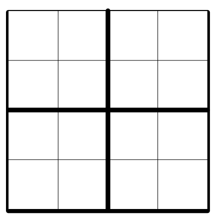
$$k = 1 \rightarrow n = 2^1 = 2$$

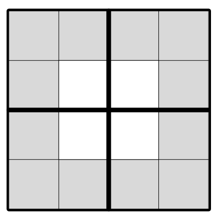
با جایگذاری هریک از الگوها در شکل پایه می‌بینیم که همواره یک خانه خالی باقی می‌ماند.

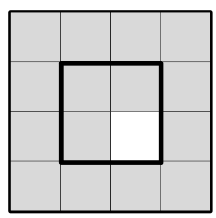
مساحت مربع $= n \times n = 2^k \times 2^k = 2^{2k} \quad \Longrightarrow \quad 2^{2k} \bmod 3 = 1$

حال با استفاده از شکل پایه به‌دست آمده ابعاد را بیش‌تر می‌کنیم و به این ترتیب کل سطح موزاییک می‌شود. برای موزاییک شدن کامل گاهی مجبوریم شکل پایه را دوران دهیم تا قسمت خالی در گوشه قرار گیرد.


شکل پایه


سطحی که می خواهیم موزاییک کنیم.


چهار عدد از شکل پایه را در هر قسمت قرار داده و برای موزاییک شدن کامل آن ها را به این صورت دوران داده ایم.


با دوران مناسب به اندازه یک شکل پایه جای خالی ایجاد شده که آن را نیز پر می کنیم.

۳-۲ قضیه‌ی اصلی (Master Method)

قضیه‌ی اصلی روشی است که مرتبه زمانی روابط بازگشتی که از فرم زیر تبعیت می‌کنند را بدست می‌آورد.

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \qquad a \geq 1\,, b > 1$$

حالت اول (Case1): اگر $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$ آنگاه به ازای $\varepsilon > 0$ داریم:

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

حالت دوم (Case2): اگر $f(n) = \theta\left(n^{\log_b a}\right)$ آنگاه داریم:

$$T(n) = \theta\left(n^{\log_b a} \times \log_2 n\right)$$

حالت سوم (Case3): اگر $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ آنگاه به ازای $\varepsilon > 0$ داریم:

$$T(n) = \theta\big(f(n)\big)$$

مثال: رابطه‌های بازگشتی زیر را با استفاده از قضیه‌ی اصلی حل کنید.

$$\mathbf{1)}\quad \boldsymbol{T(n) = T\left(\frac{n}{2}\right) + n}$$

$$a = 1\,, b = 2\,, f(n) = n$$

$$\left.\begin{array}{l}\log_b a = \log_2 1 = 0 \\ \\ f(n) = n\end{array}\right\} \quad f(n) = \Omega\left(n^{\log_2 1+\varepsilon=1}\right) \implies n^1 = n^{0+\varepsilon} \implies T(n) = \theta(n) \quad \text{حالت سوم}$$

$$\mathbf{2)}\quad \boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + n}$$

$$a = 2\,, b = 2\,, f(n) = n$$

$$\left.\begin{array}{l}\log_b a = \log_2 2 = 1 \\ \\ f(n) = n\end{array}\right\} \quad f(n) = \theta\left(n^{\log_2 2}\right) \implies n^1 = n^{1+\varepsilon} \implies T(n) = \theta(n \times \log_2 n) \quad \text{حالت دوم}$$

$$\mathbf{3)}\ \boldsymbol{T(n) = 4\,T\left(\frac{n}{2}\right) + n}$$

$a = 4\,, b = 2\,, f(n) = n$

$$\left.\begin{array}{l} \log_b a = \log_2 4 = 2 \\ \\ f(n) = n \end{array}\right\} \quad f(n) = O\left(n^{\log_2 4-\varepsilon=1}\right) \quad \Longrightarrow \quad n^1 = n^{2-\varepsilon} \quad \Longrightarrow \quad T(n) = \theta(n^2)$$ حالت اول

$$\mathbf{4)}\ \boldsymbol{T(n) = T\left(\frac{2n}{3}\right) + 1}$$

$a = 1\,, b = \frac{3}{2}\,, f(n) = 1$

$$\left.\begin{array}{l} \log_b a = \log_{3/2} 1 = 0 \\ \\ f(n) = 1 \end{array}\right\} \quad T(n) = \theta(\mathrm{n}^0 \times \log n) = \theta(\log n)$$ حالت دوم

$$\mathbf{5)}\ \boldsymbol{T(n) = 3\,T\left(\frac{n}{4}\right) + n\log n}$$

$a = 3\,, b = 4\,, f(n) = n\log n$

$$\left.\begin{array}{l} \log_b a = \log_4 3 < 1 \\ \\ f(n) = n\log n \end{array}\right\} \quad T(n) = \theta(n\log n)$$ حالت سوم

✓ **تمرین: مسائل زیر را با استفاده از قضیه‌ی اصلی حل کنید.**

1) $T(n) = 2\,T\left(\frac{n}{2}\right) + \log(n!)$
2) $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{6}\right) + \theta\left(n^{\sqrt{\log n}}\right)$
3) $T(n) = 2\,T\left(\frac{n}{4}\right) + 3\sqrt{n} + \log^2 n$
4) $T(n) = 2\,T\left(\frac{n}{8}\right) + \sqrt[3]{n}$

۳-۳ جستجوی دودویی (Binary Search)

آرایه‌ای مرتب شده (فرض می‌کنیم صعودی مرتب شده است) از اعداد داریم که درون آن به دنبال عنصری مانند x می‌گردیم. با استفاده از الگوریتم جستجوی دودویی می‌توانیم با هزینه‌ی کمی مقدار مورد نظر خود را بیابیم.

ابتدا عنصر x که به دنبال آن هستیم با عنصر وسط آرایه مقایسه می‌شود، اگر x دقیقاً با عنصر وسط برابر بود الگوریتم پایان می‌یابد و مقدار x برابر همان مقدار عنصر وسط می‌شود. اگر x با عنصر وسط برابر نبود آرایه به دو زیر آرایه تقسیم می‌شود که زیر آرایه سمت راست، عناصر بزرگ‌تر از x و زیر آرایه سمت چپ عناصر کوچک‌تر از x هستند. اگر x بزرگ‌تر از عنصر وسط بود با عناصر زیر آرایه سمت راست و اگر کوچک‌تر بود با عناصر زیر آرایه سمت چپ مقایسه می‌شود. این الگوریتم تا جایی ادامه پیدا می‌کند که عنصر مورد نظر پیدا شود و یا عناصر آرایه تمام شوند.

به عنوان مثال در آرایه‌ی زیر به دنبال عدد x می‌گردیم.

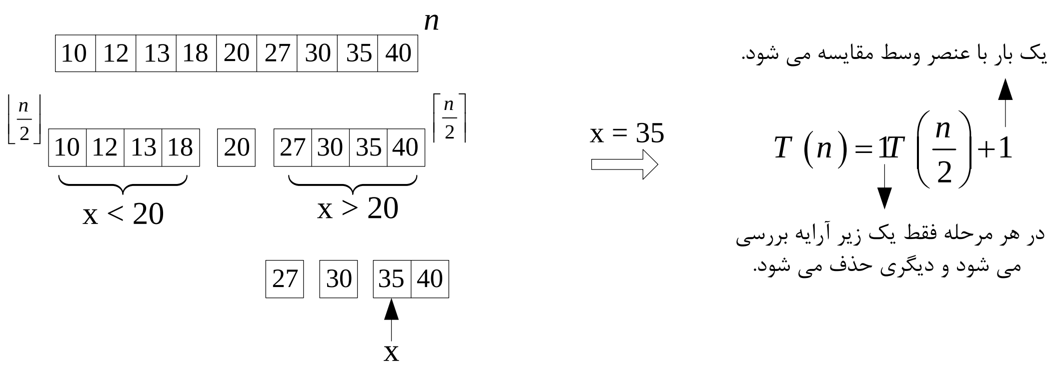
10	12	13	18	20	27	30	35	40

۳-۳-۱ بهترین حالت

بهترین حالت زمانی است که x دقیقاً در وسط آرایه قرار داشته باشد در این صورت فقط یک مقایسه صورت می‌گیرد و الگوریتم پایان می‌یابد.

۳-۳-۲ حالت متوسط

حالت متوسط زمانی است که x درون آرایه و جایی غیر از وسط باشد. در این صورت x با عنصر وسط مقایسه می‌شود، اگر بزرگ‌تر از آن باشد به زیر آرایه سمت راست و در غیر این صورت به زیر آرایه سمت چپ می‌رود و دوباره همین الگور تکرار می‌شود.



$$T(n) = 1\,T\left(\frac{n}{2}\right) + 1 \quad \Longrightarrow \quad \left.\begin{matrix} n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \\ f(n) = 1 \end{matrix}\right\} \quad \Longrightarrow \quad \theta(\log n)$$

۳-۳-۳ بدترین حالت

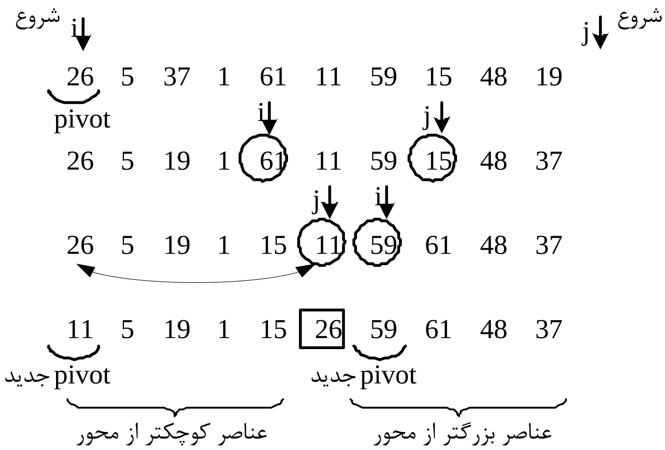
بدترین حالت زمانی است که x درون آرایه نباشد، مرتبه زمانی این حالت نیز مانند حالت متوسط است.

$$T(n) = 1\,T\left(\frac{n}{2}\right) + 1 \quad \Rightarrow \quad \left.\begin{matrix} n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \\ f(n) = 1 \end{matrix}\right\} \quad \Rightarrow \quad \theta(\log n)$$

- ✓ **تمرین: مقدار دقیق مرتبه زمانی الگوریتم جستجوی دودویی را به‌دست آورید.**

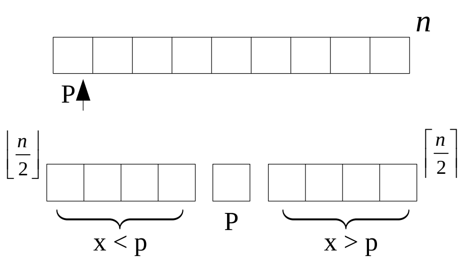
۳-۴ مرتب سازی سریع (Quick Sort)

در این نوع مرتب سازی یک عنصر به عنوان محور (pivot) انتخاب می‌کنیم (معمولاً اولین عنصر) و دو اندیس i و j در نظر می‌گیریم. i از سمت محور شروع به حرکت می‌کند تا به عنصری بزرگ‌تر از محور برسد و j از سمت دیگر شروع به حرکت می‌کند تا به عنصری کوچک‌تر از محور برسد، سپس دو عنصر با یکدیگر جابه‌جا می‌شوند. این عمل تا جایی انجام می‌شود که i و j از یکدیگر عبور نکرده باشند. پس از رد شدن i از j جای j با pivot عوض می‌شود. در این صورت آرایه اولیه به دو زیر آرایه تقسیم می‌شود که عناصر زیر آرایه سمت راست محور، بزرگ‌تر از آن و زیر آرایه سمت چپ کوچک‌تر از آن هستند. این عمل به صورت بازگشتی برای زیر آرایه‌های ایجاد شده اجرا می‌شود، تا جایی که کل آرایه مرتب شود.



۳-۴-۱ بهترین حالت

بهترین حالت زمانی است که عنصر محور دقیقاً وسط قرار گیرد و دو آرایه یکسان ایجاد کند.



در هر مرحله هر دو زیر آرایه بررسی می شوند.

$$T(n) = 2T\left(\frac{n}{2}\right) + \underbrace{\theta(n)}_{(n-1)}$$

هزینه لازم برای افراز

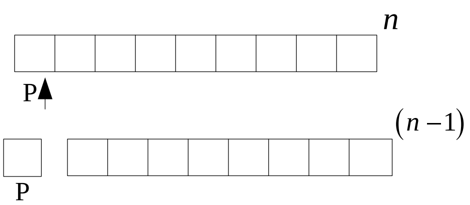
$$T(n) = 2\,T\left(\frac{n}{2}\right) + (n-1) \quad \Rightarrow \quad \left.\begin{matrix} n^{\log_b a} = n^{\log_2 2} = n^1 = n \\ f(n) = (n-1) \end{matrix}\right\} \quad \Rightarrow \quad \theta(n \log n)$$

۳-۴-۲ حالت متوسط

حالت متوسط حالتی است که آرایه شکل مشخصی نداشته باشد. مرتبه زمانی در این حالت مانند مرتبه زمانی بهترین حالت است.

۳-۴-۳ بدترین حالت

بدترین حالت زمانی است که آرایه مرتب باشد (نزولی یا صعودی).



چون آرایه مرتب است تمام عناصر بزرگ تر
یا کوچک تر از عنصر محور می شوند.

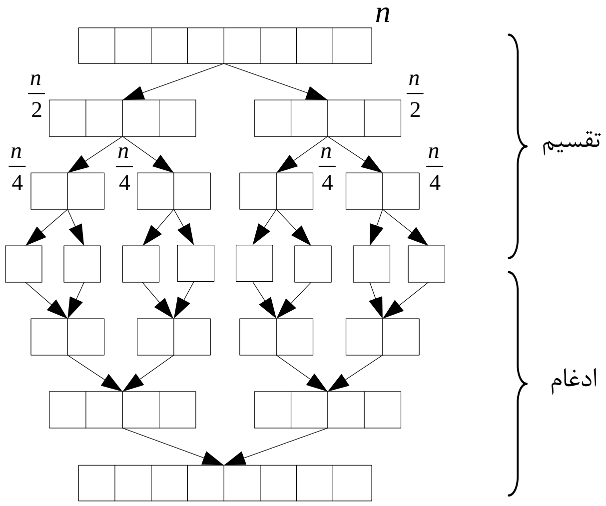
$$T(n) = T(n-1) + \theta(n)$$

$$T(n) = T(n-1) + (n-1) = \frac{n(n-1)}{2} \quad \Rightarrow \quad \theta(n^2)$$

✓ **تمرین: مقدار دقیق مرتبه زمانی الگوریتم مرتب سازی سریع را به‌دست آورید.**

۳-۵ مرتب سازی ادغامی (Merge Sort)

در این نوع مرتب سازی، آرایه عناصر ورودی در هر مرحله به دو زیر آرایه تقسیم می‌شود و این کار تا جایی ادامه پیدا می‌کند که به آرایه‌های تک عنصری برسیم (آرایه تک عنصری مرتب است)، از این پس آرایه‌ها را با مقایسه کردن عناصر آن‌ها ادغام می‌کنیم تا مرتب شوند. اگر تعداد عناصر آرایه ورودی فرد باشد هنگام تقسیم آرایه به دو زیر آرایه یکی از آرایه‌ها یک عنصر از آرایه دیگر بیشتر خواهد داشت.



در هر مرحله هر دو زیر آرایه بررسی می شوند.

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$$

هزینه ادغام

بهترین حالت، حالت متوسط و بدترین حالت هر سه یکسان هستند زیرا در هر صورت باید مقایسه صورت بگیرد.

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \theta(n) \quad \Longrightarrow \quad \left.\begin{array}{l} n^{\log_b a} = n^{\log_2 2} = n^1 = n \\ f(n) = \theta(n) \end{array}\right\} \quad \Longrightarrow \quad \theta(n \log n)$$

✓ **تمرین: مقدار دقیق مرتبه زمانی الگوریتم مرتب سازی ادغامی را به‌دست آورید.**

۳-۶ ضرب چند جمله‌ای ها

ضرب دو چند جمله‌ای زیر را در نظر بگیرید:

$$\left(x^3 - 2x^2 + 4x - 3\right)\left(x^2 + 4x + 5\right) = x^5 + 2x^4 + x^3 + 3x^2 + 8x - 15$$

3	2	1	0
1	-2	4	-3

درجه 3

جمله 4

2	1	0
1	4	5

درجه 2

جمله 3

5	4	3	2	1	0
1	2	1	3	8	-15

درجه 3+2=5

جمله 3+2+1=6

می‌خواهیم دو چند جمله‌ای P و Q زیر را در یک‌دیگر ضرب کنیم:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0 \qquad a_n \neq 0$$

$$Q_n(x) = b_n x^n + b_{n-1} x^{n-1} + \cdots + b_1 x^1 + b_0 \qquad b_n \neq 0$$

تک تک جملات P در تک تک جملات Q ضرب می‌شوند. برنامه زیر حاصل‌ضرب دو چندجمله‌ای را به‌دست می‌آورد.

```
for (i=0; i<=n; i++)
    for (j=0; j<=n; j++)
        C[i+j] = C[i+j] + A[i]*B[j];
```

$$\underbrace{P_n(x)}_{\text{جمله } (n+1)} \times \underbrace{Q_n(x)}_{\text{جمله } (n+1)} = \underbrace{C_{2n}(x)}_{\text{جمله } (2n+1)} = \sum_{i=0}^{2n} C_i x^i$$

نحوه ذخیره سازی جملات در آرایه به این صورت است که فقط ضرایب در خانه های مربوط به خود در آرایه ذخیره می شوند.

P:

0	1	2		n
a_0	a_1	a_2	$\cdots$	a_n

Q:

0	1	2		n
b_0	b_1	b_2	$\cdots$	b_n

C:

0	1	2		2n
c_0	c_1	c_2	$\cdots$	c_{2n}

$$\text{تعداد ضرب} = (n+1)\times(n+1) = (n+1)^2 = n^2+2n+1 \quad \Longrightarrow \quad O(n^2)$$

برای محاسبه حاصل‌ضرب این دو چندجمله‌ای مرتبه زمانی ضرب $O(n^2)$ است. می‌خواهیم با استفاده از روش تقسیم و حل مرتبه زمانی را کاهش دهیم. برای این کار هر یک از جملات P و Q را به‌صورت زیر از وسط نصف می‌کنیم:

$$P_n(x) = \left(a_n x^n + a_{n-1}x^{n-1} + \cdots + a_{\frac{n}{2}} x^{\frac{n}{2}}\right) + \underbrace{\cdots + a_1 x^1 + a_0}_{B}$$

$$Q_n(x) = \left(b_n x^n + b_{n-1}x^{n-1} + \cdots + b_{\frac{n}{2}} x^{\frac{n}{2}}\right) + \underbrace{\cdots + b_1 x^1 + b_0}_{D}$$

B و D نیز چندجمله‌ای از درجه $\frac{n}{2}$ هستند.

$$P_n(x) = x^{\frac{n}{2}} \underbrace{\left(a_n x^{\frac{n}{2}} + \cdots + a_{\frac{n}{2}} x^0\right)}_{A} + B$$

$$Q_n(x) = x^{\frac{n}{2}} \underbrace{\left(b_n x^{\frac{n}{2}} + \cdots + b_{\frac{n}{2}} x^0\right)}_{C} + D$$

C و A نیز چندجمله‌ای از درجه $\frac{n}{2}$ هستند.

P و Q به‌صورت زیر حاصل می‌شوند:

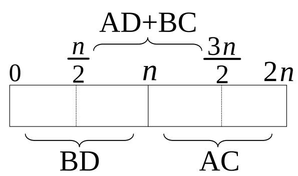
$$P_n(x) = x^{\frac{n}{2}} A_{\frac{n}{2}}(x) + B_{\frac{n}{2}}(x)$$

$$Q_n(x) = x^{\frac{n}{2}} C_{\frac{n}{2}}(x) + D_{\frac{n}{2}}(x)$$

$$P_n(x) \times Q_n(x) = x^n \underbrace{AC}_{*} + x^{\frac{n}{2}} \left(\underbrace{AD}_{*} + \underbrace{BC}_{*}\right) + \underbrace{BD}_{*}$$

نحوه ذخیره سازی جملات در آرایه پس از اعمال روش تقسیم و حل به صورت زیر است.

آرایه حاصل ضرب :



در حاصل‌ضرب ایجاد شده از P و Q که در خط بالا نشان داده شده ۴ تا ضرب ایجاد شده است، بنابراین:

$$T(n) = 4\,T\left(\frac{n}{2}\right) + O(n) \qquad \Longrightarrow \qquad O(n^2)$$

با توجه به محاسبات انجام شده معلوم می‌شود که مرتبه زمانی کاهش نیافته است. حال برای کاهش مرتبه زمانی از اتحاد زیر استفاده می‌کنیم:

$$AD + BC = (A - B)(D - C) + AC + BD$$

اتحاد به‌دست آمده را در فرمول حاصل‌ضرب جایگذاری می‌کنیم:

$$P_n(x) \times Q_n(x) = x^n \underbrace{AC}_{*} + x^{\frac{n}{2}} \left[\underbrace{(A - B)(D - C)}_{*} + \underbrace{AC + BD}_{\text{تکراری}}\right] + \underbrace{BD}_{*}$$

حال تعداد ضرب‌ها به ۳ ضرب کاهش یافت و داریم:

$$T(n) = 3\,T\left(\frac{n}{2}\right) + \underbrace{O(n)}_{?} \qquad \Longrightarrow \qquad n^{\log_2 3} = n^{1.6} \qquad \Longrightarrow \qquad \theta(n^{1.6})$$

- **مرتبه زمانی از $O(n^2)$ به $\theta(n^{1.6})$ کاهش یافت.**

- ✓ **تمرین: تعداد دقیق جمع (مقدار دقیق $O(n)$) را در رابطه بازگشتی صفحه قبل به‌دست آورید.**
- ✓ **تمرین: الگوریتمی طراحی کنید که با استفاده از روش تقسیم و حل دو عدد بزرگ را در هم ضرب کند.**
- ✓ **مرتبه زمانی این الگوریتم چند است؟**

۳-۷ ضرب ماتریس‌ها به روش استراسن

ضرب دو ماتریس زیر را در نظر بگیرید:

$$\begin{bmatrix} a_{11} & \cdots & a_{1K} \\ \vdots & \ddots & \vdots \\ a_{R1} & \cdots & a_{RK} \end{bmatrix}_{R\times K} \times \begin{bmatrix} b_{11} & \cdots & b_{1P} \\ \vdots & \ddots & \vdots \\ b_{K1} & \cdots & b_{KP} \end{bmatrix}_{K\times P} = \begin{bmatrix} c_{11} & \cdots & c_{1P} \\ \vdots & \ddots & \vdots \\ c_{R1} & \cdots & c_{RP} \end{bmatrix}_{R\times P}$$

تعداد کل ضرب‌ها $= R \times K \times P$

حاصل‌ضرب دو ماتریس 2×2 یک ماتریس 2×2 است که به‌صورت زیر نشان داده می‌شود:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

مولفه‌های حاصل عبارتند از:

$$C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad i = 1,2 \quad j = 1,2 \quad n = 2$$

الگوریتم ضرب عادی ماتریس‌ها به‌صورت زیر است:

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      {
         C[i][j]=0;
         for (k=0; k<n; k++)
         C[i][j]=C[i][j]+A[i][k]*B[k][j];
      }
```

رابطه‌ی بازگشتی و مرتبه زمانی ضرب عادی ماتریس‌ها به‌صورت زیر است:

$$T(n) = \underbrace{8}_{\text{ضرب}} T\left(\frac{n}{2}\right) + \underbrace{4\left(\frac{n}{2}\right)^2}_{\text{جمع}} \quad \Rightarrow \quad \log_2 8 = 3 \quad \Rightarrow \quad \theta(n^3)$$

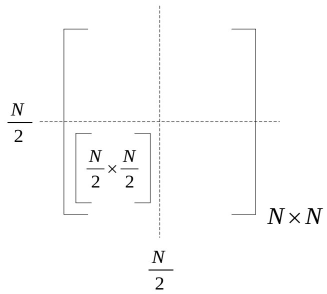
در ضرب ماتریس‌ها (ماتریس مربعی $n \times n$) به روش عادی داریم:

تعداد ضرب‌ها $= n^3$

تعداد جمع‌ها $= n^3 - n^2$

حال می‌خواهیم با روش استراسن مرتبه زمانی ضرب ماتریس‌ها را کاهش دهیم.

ماتریس‌های مورد نظر در روش استراسن باید مربعی باشند، بنابراین اگر ماتریسی مربعی نبود با افزودن سطر و ستون صفر آن را مربعی می‌کنیم. سپس آنقدر ماتریس را بر 2 تقسیم می‌کنیم تا ماتریس 2×2 به‌دست آید. در واقع با این کار مسئله را با استفاده از روش تقسیم و حل به مسائل کوچک‌تر تقسیم کرده‌ایم.



با استفاده از فرمول‌های زیر مقدار M ها را به‌دست می‌آوریم. این فرمول‌ها برای ماتریس‌های با درایه‌های ماتریسی هستند.

$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$

$M_2 = (A_{21} + A_{22})B_{11}$

$M_3 = A_{11}(B_{11} - B_{22})$

$M_4 = A_{22}(B_{21} - B_{11})$

$M_5 = (A_{11} + A_{22})B_{22}$

$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$

$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

سپس مقادیرحاصل را در فرمول‌های زیرکه برای ماتریس‌های با درایه‌های تک‌عنصری هستند قرار می‌دهیم و mها را به‌دست می‌آوریم.

$m_1 = (a_{11} + a_{22})(b_{11} + b_{22})$

$m_2 = (a_{21} + a_{22})b_{11}$

$m_3 = a_{11}(b_{11} - b_{22})$

$m_4 = a_{22}(b_{21} - b_{11})$

$m_5 = (a_{11} + a_{22})b_{22}$

$m_6 = (a_{21} - a_{11})(b_{11} + b_{12})$

$m_7 = (a_{12} - a_{22})(b_{21} + b_{22})$

پس از به‌دست آوردن مقادیر mها، آن‌ها را در فرمول زیر جایگذاری کرده و مقدار C را به‌دست می‌آوریم.

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

به عنوان مثال ضرب ماتریس‌های زیر را در نظر می‌گیریم:

$$\left[\begin{array}{cc} \overset{A_{11}}{\begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix}} & \overset{A_{12}}{\begin{bmatrix} 1 & 5 \\ 7 & 2 \end{bmatrix}} \\ \underset{A_{21}}{\begin{bmatrix} 1 & 3 \\ 4 & 1 \end{bmatrix}} & \underset{A_{22}}{\begin{bmatrix} 7 & 9 \\ 5 & 4 \end{bmatrix}} \end{array}\right]_{4\times 4} \times \left[\begin{array}{cc} \overset{B_{11}}{\begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}} & \overset{B_{12}}{\begin{bmatrix} 4 & 6 \\ 1 & 0 \end{bmatrix}} \\ \underset{B_{21}}{\begin{bmatrix} 0 & 1 \\ 4 & 2 \end{bmatrix}} & \underset{B_{22}}{\begin{bmatrix} 3 & 5 \\ 7 & 1 \end{bmatrix}} \end{array}\right]_{4\times 4}$$

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_1 = \left(\overset{A_{11}}{\begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix}} + \overset{A_{22}}{\begin{bmatrix} 7 & 9 \\ 5 & 4 \end{bmatrix}}\right) \times \left(\overset{B_{11}}{\begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}} + \overset{B_{22}}{\begin{bmatrix} 3 & 5 \\ 7 & 1 \end{bmatrix}}\right) = \begin{bmatrix} 10 & 11 \\ 6 & 4 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 9 & 4 \end{bmatrix}$$

$$\begin{bmatrix} \overset{}{10}_{\leftarrow a_{11}} & 11 \\ 6 & \underset{\downarrow a_{22}}{4} \end{bmatrix} \times \begin{bmatrix} \overset{b_{11} \uparrow}{4} & 5 \\ 9 & 4_{\rightarrow b_{22}} \end{bmatrix}$$

$$m_1 = (a_{11} + a_{22})(b_{11} + b_{22}) = (10 + 4)(4 + 4) = 14 \times 8 = 112$$

به همین ترتیب سایر m ها را نیز به‌دست می‌آوریم و در نهایت در ماتریس C جایگذاری می‌کنیم.

رابطه‌ی بازگشتی و مرتبه زمانی ضرب ماتریس‌ها به روش استراسن به‌صورت زیر است:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 7\,T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 & n > 1 \end{cases} \qquad \log 7 = 2.81 \quad \Longrightarrow \quad \theta(n^{2.81})$$

جدول مقایسه دو الگوریتم ضرب ماتریس‌های $\boldsymbol{n \times n}$

	تعداد ضرب	تعداد جمع‌ها و تفریق‌ها
الگوریتم استاندارد ضرب	n^3	$n^3 - n^2$
الگوریتم استراسن	$n^{2.81}$	$6n^{2.81} - 6n^2$

- ✓ **تمرین عملی: ضرب ماتریس‌ها به روش معمولی و استراسن را به‌طور کامل برای ماتریس $\boldsymbol{n \times n}$ پیاده سازی کنید و زمان‌های واقعی آن‌ها را با یکدیگر مقایسه کنید.**

فصل چهارم: برنامه نویسی پویا (Dynamic Programming)

۴-۱ برنامه نویسی پویا

مسئله‌هایی‌که حل آن‌ها به روش تقسیم‌و حل موجب حل تکراری زیر مسئله‌ها می‌شود به روش برنامه‌نویسی پویا (DP) حل می‌شوند. این‌گونه مسئله‌ها را بهتر است به‌جای حل از بالا به پائین و به صورت بازگشتی، از پائین به بالا حل کنیم. زیر مسئله‌ها فقط یک بار حل می‌شوند و حاصل آن‌ها در جدولی ذخیره می‌شود تا اگر برای حل زیر مسئله‌های بزرگتر مکرراً به نتیجه حل یک زیر مسئله کوچکتر نیاز باشد آن را بتوان بدون پرداخت هزینه‌ای از جدول بدست آورد.

۴-۱-۱ ویژگی‌های مسائلی که به روش DP حل می‌شوند

۱) مسئله معمولاً بهینه سازی است. (کمینه سازی یا بیشینه سازی)
۲) برای حل مسئله باید زیر مسئله‌ها هم به صورت بهینه حل شوند.
۳) حل مسئله به روش استقرایی یا تقسیم و حل موجب حل تکراری یک زیر مسئله دلخواه خواهد شد.

۴-۲ الگوریتم ترکیب

به دو روش ضریب دوجمله ای را حل می کنیم و مرتبه زمانی آنها را محاسبه می‌کنیم:

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!} \quad , \quad 0 \le m \le n \quad \Rightarrow \quad \text{پیچیدگی زمانی} = O(n!)$$

۴-۲-۱ روش تقسیم و حل

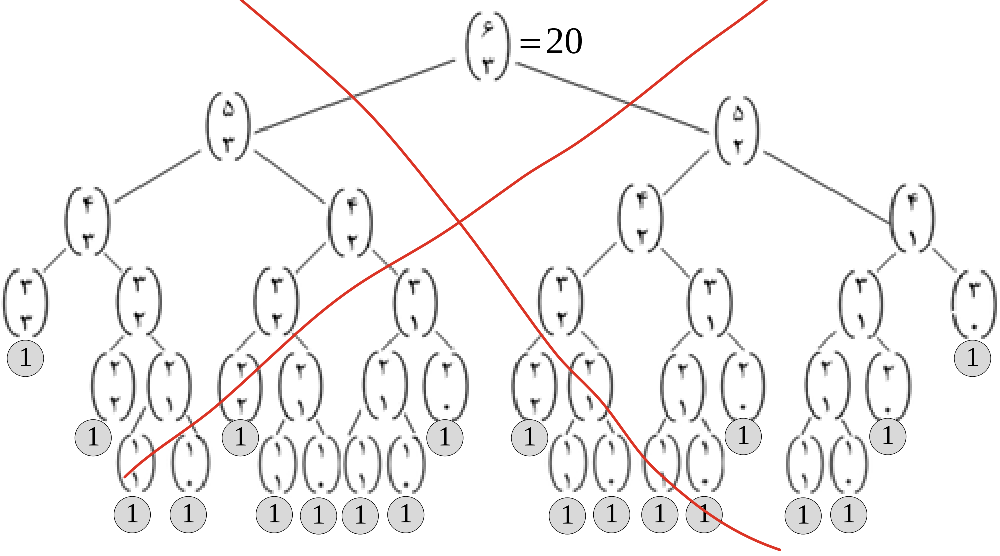
$$\binom{n}{m} = \begin{cases} 1 & if\ m = n\ or\ m = 0 \\ \binom{n-1}{m-1} + \binom{n-1}{m} & 0 < m < n \end{cases}$$

الگوریتم ترکیب به روش تقسیم و حل

```
int  CombinationDC  (int n,int m)
 {
   if (n==m || m==0)
         return 1;
   return  CombinationDC (n-1,m-1) + CombinationDC (n-1,m);
 }
```

مثال: برای ترکیب $\binom{6}{3}$ نمودار درختی را رسم می‌کنیم.

$$\binom{6}{3} = \frac{6!}{3! \times 3!} = 20$$



مشاهده می‌کنیم که در این روش عناصر تکراری زیادی وجود دارند و ما آنها را در هر مرحله مکرراً محاسبه می‌کنیم.

رابطه بازگشتی این روش به صورت زیر است:

$$T(n,m) = \begin{cases} 1 & m = n \quad or \quad m = 0 \\ T(n-1,m-1) + T(n-1,m) + 1 & 0 < m < n \end{cases}$$

در این روش عمل مبنایی جمع است، بنابراین پیچیدگی زمانی برای تعداد عملیات جمع به صورت زیر بدست می‌آید:

تعداد عملیات جمع = $\binom{n}{m} - 1$

۴-۲-۲ روش پویا

در روش پویا ویژگی بازگشتی ایجاد می‌کنیم. جدولی به ابعاد $(m+1)\times(n+1)$ ایجاد می‌کنیم که در آن مقدار هر خانه از مجموع دو خانه بالاتر جدول محاسبه می‌شود. چون در این روش نتایج محاسبات میانی ذخیره می‌شوند دیگر مقادیر تکراری را محاسبه نمی‌کنیم و این باعث کاهش مرتبه زمانی می‌شود.

مثال: با استفاده از روش پویا مقدار $\binom{10}{6}$ را محاسبه کنید و مرتبه زمانی آن را بدست آورید؟

۱-۲-۲-۴ الگوریتم ترکیب با استفاده از روش اول پویا

```
int  CDP1 (int n, int m)
 {
   int  i, j, c[n+1][m+1];
   for (i=0; i<=n; i++)
          c[i][0]=1;        →   ستون اول ۱ می شود
   for (j=1; j<=m; j++)
           c[j][j]=1;        →   قطر ۱ می شود
   for (i=2; i<=n; i++)
       for (j=1; j<=min(i-1,m); j++)
              c[i][j]=c[i-1][j]+c[i-1][j-1];
   return  c[n][m];
   }
```

→ j

i \ j	0	1	2	3	4	5	m 6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1
7	1	7	21	35	35	21	7
8	1	8	28	56	70	56	28
9	1	9	36	84	126	126	84
n 10	1	10	45	120	210	252	210

↓ i

$(n-m)$

n=10
m=6

$$\underbrace{1+2+\cdots+m-1}_{\frac{m(m-1)}{2}}+\underbrace{m+m+\cdots+m}_{(n-m)}$$

$$\frac{m(m-1)}{2}+m(n-m)=\frac{m(2n-m-1)}{2}$$

۴-۲-۲-۲ الگوریتم ترکیب با استفاده از روش دوم پویا (پویا یک بعدی)

```
int  CDP2 (int n, int m)
 {
   int  i, j, a[m+1];
   a[0]=1;
   for (i=1; i<=n; i++)
      {   if (i<=m)      a[i]=1;
          for (j=min(i-1,m) ; j>=1; j--)
                 a[j]=a[j]+a[j-1];  }
   return  a[m];
   }
```

الگوریتم پویا یک بعدی از لحاظ مرتبه زمانی مانند الگوریتم پویا دو بعدی است ولی حافظه‌ی کم‌تری اشغال می‌کند.

۴-۲-۲-۳ الگوریتم ترکیب با استفاده از روش سوم پویا (یک بعدی و بهینه)

```
int  CDP3 (int n, int m)
 {
   int  i, j, a[m+1];
   a[0]=1;
   for (i=1; i<=n; i++)
      {   if (i<=m)      a[i]=1;
          for (j=min(i-1,m);  j>=Max(1,m-n+i);  j--)
                 a[j]=a[j]+a[j-1];  }
   return  a[m];
   }
```

جدول مقایسه مرتبه زمانی و مرتبه حافظه برای روش‌های اول، دوم و سوم پویا:

	مرتبه زمانی	مرتبه حافظه
DP1	$\frac{m}{2}(2n-m-1)$	$(m+1)(n+1)$
DP2	$\frac{m}{2}(2n-m-1)$	$(m+1)$
DP3	$m(n-m)$	$(m+1)$

۴-۳ ضرب زنجیره‌ای ماتریس‌ها (نحوه‌ی پرانتز بندی ضرب ماتریس‌ها)

دو ماتریس A و B با ابعاد زیر داریم:

$$\mathrm{A}_{\mathrm{p}\times\mathrm{q}} \times \mathrm{B}_{\mathrm{q}\times\mathrm{r}} = H_{\mathrm{p}\times\mathrm{r}}$$

تعداد ضرب‌های لازم برای ضرب ماتریس A در ماتریس B در حالت کلی برابر است با:

تعداد ضرب = p×q×r

حال فرض کنید می‌خواهیم سه ماتریس A ، B و C را در هم ضرب کنیم. این کار را می‌توان به چند صورت انجام داد:

$$\mathrm{A}_{\mathrm{p}\times\mathrm{q}} \;,\; \mathrm{B}_{\mathrm{q}\times\mathrm{r}} \;,\; \mathrm{C}_{\mathrm{r}\times\mathrm{w}}$$

$$(A\times B)\times C \quad \rightarrow \quad \text{تعداد ضرب} = pqr + prw$$

$$A\times (B\times C) \quad \rightarrow \quad \text{تعداد ضرب} = pqw + qrw$$

با تغییر جای پرانتز تعداد ضرب‌های متفاوتی ایجاد می‌شوند. حالتی که تعداد ضرب‌های آن کمتر باشد بهترین حالت است.

$$pqr + prw \underset{\text{مقایسه}}{\boxed{?}} pqw + qrw$$

$$\frac{pqr + prw}{pqrw} \boxed{?} \frac{pqw + qrw}{pqrw}$$

$$\frac{1}{w} + \frac{1}{q} \boxed{?} \frac{1}{r} + \frac{1}{p}$$

هر سمتی که کم‌تر بود تعداد ضرب کم‌تری می‌خواهد.

مثال: تعداد ضرب‌های لازم برای ضرب ماتریس‌های داده شده زیر چند است؟

$$A_{2\times3} \times B_{3\times5} \times C_{5\times1} = ?$$

$$(A\times B)\times C \quad \rightarrow \quad \text{تعداد ضرب} = 2\times3\times5 + 2\times5\times1 = 30 + 10 = 40$$

$$A\times (B\times C) \quad \rightarrow \quad \text{تعداد ضرب} = 2\times3\times1 + 3\times5\times1 = 6 + 15 = 21$$

حال می‌خواهیم این مسئله را برای n ماتریس تعمیم دهیم. یعنی پرانتز گذاری را برای n ماتریس بگونه‌ای انجام دهیم که کم‌ترین تعداد ضرب حاصل شود.

n ماتریس $M_1 \ldots M_n$ داده شده‌اند. می‌خواهیم این ماتریس‌ها را به صورت زیر در هم ضرب کنیم:

$$\underset{[d_0 \times d_n]}{M_{1n}} = \underset{[d_0 \times d_1]}{M_1} \times \underset{[d_1 \times d_2]}{M_2} \times \cdots \times \underset{[d_{n-1} \times d_n]}{M_n}$$

$M_i \rightarrow$ شماره ماتریس

M_i ابعاد ماتریس $= [d_{i-1} \times d_i]$

ترتیب انجام اعمال ضرب را طوری تعیین کنید تا تعداد کل ضرب‌های اعداد حقیقی کمینه شود.

راه حل: برای حل بازگشتی این مسئله، ابتدا زیر مسئله را تعریف می‌کنیم:

$$\underset{[d_{i-1} \times d_j]}{M_{ij}} = \underset{[d_{i-1} \times d_i]}{M_i} \times \underset{[d_i \times d_{i+1}]}{M_{i+1}} \times \cdots \times \underset{[d_{j-1} \times d_j]}{M_j}$$

M_{ij} یعنی حاصل‌ضرب ماتریس‌های M_i تا M_j

نکات:

- i و j ابعاد نیستند بلکه شماره یا اندیس ماتریس هستند.
- C[i,j] : حداقل تعداد ضرب‌های مورد نیاز برای ضرب ماتریس‌های M_i تا M_j

فرض کنید اگر ماتریس‌های $M_1 \ldots M_n$ را از محل k ام پرانتز بندی کنیم تعداد ضرب‌ها می‌نیمم شود. در این‌صورت $1 \le k < n$ است.

$$M_{1n} = \underbrace{(M_1 \times M_2 \times \cdots \times M_k)}_{M_{1k}} \times \underbrace{(M_{k+1} \times \cdots \times M_n)}_{M_{k+1,n}}$$

$$\underbrace{[d_0 \times d_k] \qquad [d_k \times d_n]}_{\text{تعداد ضرب} = d_0 d_k d_n}$$

تعداد کمینه ضرب اعداد حقیقی به صورت زیر می‌شود:

$$C[1,n] = \begin{cases} 0 & n = 1 \\ \min\limits_{1 \le k < n} \{C[1,k] + C[k+1,n] + d_0 d_k d_n\} & n > 1 \end{cases}$$

حال فرض کنید اگر ماتریس‌های $M_i \ldots M_j$ را از محل k ام پرانتز بندی کنیم تعداد ضرب‌ها می‌نیمم شود. در این‌صورت $i \le k < j$ یا $i \le k \le j-1$ است.

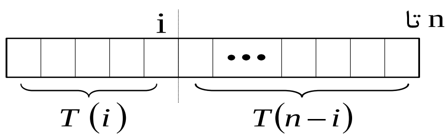
$$M_{ij} = (\underbrace{M_i \times M_{i+1} \times \cdots \times M_k}_{M_{ik}}) \times (\underbrace{M_{k+1} \times \cdots \times M_j}_{M_{k+1,j}})$$

$$\underbrace{[d_{i-1} \times d_k] \qquad [d_k \times d_j]}_{\text{تعداد ضرب} = d_{i-1} d_k d_j}$$

$$C[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \{C[i,k] + C[k+1,j] + d_{i-1} d_k d_j\} & i < j \end{cases}$$

۱-۳-۴ تعداد حالات پرانتز بندی برای ضرب n ماتریس

فرض می‌کنیم T(n) تعداد حالات پرانتز بندی برای ضرب n ماتریس است. n ماتریس را به صورت آرایه زیر نمایش می‌دهیم، آرایه را از محل ماتریس i ام به دو زیر آرایه تقسیم می‌کنیم. تعداد حالات پرانتز بندی برای زیر آرایه اول به صورت T(i) و برای زیر آرایه دوم به صورت T(n-i) می‌شود. بنابراین تعداد حالات پرانتز بندی برای کل ماتریس از ضرب تعداد حالات ماتریس اول در تعداد حالات ماتریس دوم بدست می‌آید.



i می‌تواند از ۱ تا n تغییر کند، بنابراین به ازای i های مختلف نیز حالت‌های متفاوتی پدید می‌آید. در کل تعداد حالات پرانتز بندی برای ضرب n ماتریس از فرمول زیر محاسبه می‌شود:

$$T(n) = \begin{cases} 1 & n = 1 \\ \sum\limits_{i=1}^{n-1} T(i) \times T(n-i) & n > 1 \end{cases}$$

ثابت شده است که تعداد حالات پرانتز بندی برای ضرب n ماتریس از فرمول کاتالان به ازای (n-1) نیز بدست می‌آید:

$$\text{فرمول کاتالان} \; C(n) = \frac{1}{n+1}\binom{2n}{n} \qquad \Longrightarrow \qquad \text{تعداد حالات پرانتز بندی} \; T(n) = C(n-1) = \frac{1}{n}\binom{2n-2}{n-1}$$

مثال: تعداد حالات مختلف پرانتزبندی برای ضرب ۴ ماتریس را محاسبه کنید.

$$T(4) = C(3) = \frac{1}{4}\binom{6}{3} = \frac{20}{4} = 5$$

مثال: ماتریس‌های زیر را چگونه پرانتز بندی کنیم تا تعداد ضرب‌های حاصل از آنها کم‌ترین مقدار شود؟

$$\boldsymbol{A_1} \times \boldsymbol{A_2} \times \boldsymbol{A_3} \times \boldsymbol{A_4}$$

$$[\mathbf{2} \times \mathbf{5}] \quad [\mathbf{5} \times \mathbf{3}] \quad [\mathbf{3} \times \mathbf{1}] \quad [\mathbf{1} \times \mathbf{6}]$$

یک جدول 4×4 رسم می‌کنیم و مقادیر داخل آن را به‌دست آورده جایگذاری می‌کنیم. برای ضرب‌های دوتایی نیازی نیست از فرمول استفاده کنیم، زیرا فقط یک مقدار برای k داریم. طبق فرمول عناصر قطر اصلی همگی صفر می‌شوند. عناصر زیر قطر اصلی مقداری ندارند زیرا i< j است. مسائل را به صورت قطری حل می‌کنیم زیرا در مراحل بالاتر به نتایج زیر مسائل نیاز داریم.

$$C[1,1] = C[2,2] = C[3,3] = C[4,4] = 0$$

$$C[1,2] = 2 \times 5 \times 3 = 30$$

$$C[2,3] = 5 \times 3 \times 1 = 15$$

$$C[3,4] = 3 \times 1 \times 6 = 18$$

$$C[1,3] = \min \begin{cases} k = 1 & C[1,1] + C[2,3] + d_0 d_1 d_3 = 0 + 15 + (2 \times 5 \times 1) = 25 \\ k = 2 & C[1,2] + C[3,3] + d_0 d_2 d_3 = 30 + 0 + (2 \times 3 \times 1) = 36 \end{cases}$$

$$1 \le k < 3$$

$(A_1) \times (A_2 \times A_3) \quad \rightarrow \quad$ به‌ازای $k = 1$ می‌نیمم شده

$$C[2,4] = \min \begin{cases} k = 2 & C[2,2] + C[3,4] + d_1 d_2 d_4 = 0 + 18 + (5 \times 3 \times 6) = 108 \\ k = 3 & C[2,3] + C[4,4] + d_1 d_3 d_4 = 15 + 0 + (5 \times 1 \times 6) = 45 \end{cases}$$

$$2 \le k < 4$$

$(A_2 \times A_3) \times (A_4) \quad \rightarrow \quad$ به‌ازای $k = 3$ می‌نیمم شده

$$C[1,4] = \min_{1 \le k < 4} \begin{cases} k=1 & C[1,1]+C[2,4]+d_0d_1d_4 = 0+45+(2\times5\times6)=105 \\ k=2 & C[1,2]+C[3,4]+d_0d_2d_4 = 30+18+(2\times3\times6)=84 \\ k=3 & C[1,3]+C[4,4]+d_0d_3d_4 = 25+0+(2\times1\times6)=37 \end{cases}$$

توضیح در مورد نحوه پرانتز گذاری

۱) $C[1,4]$ به ازای $k=3$ کمینه شده است بنابراین از محل $k=3$ تقسیم می‌شود.

$(A_1 \times A_2 \times A_3)\times(A_4)$

۲) A_4 قابل تقسیم نیست، $C[1,3]$ به ازای $k=1$ کمینه شده است بنابراین از محل $k=1$ تقسیم می‌شود.

$(A_1)\times(A_2 \times A_3)$

$$\left.\begin{array}{lll} k=3 & \rightarrow & (A_1\times A_2\times A_3)\times A_4 \\ \text{از مراحل قبل داشتیم} & \rightarrow & A_1\times(A_2\times A_3) \end{array}\right\} \Longrightarrow (A_1\times(A_2\times A_3))\times A_4$$

	1	2	3	4
1	0	30	K=1 25	K=3 37
2		0	15	K=3 45
3			0	18
4				0

$$(A_1\times(A_2\times A_3))\times A_4$$

- **مرتبه زمانی ضرب ماتریس‌ها $\boldsymbol{\theta(n^3)}$ است.**

۴-۴ الگوریتم فلوید (Floyd)

هدف الگوریتم فلوید یافتن کوتاهترین مسیر بین هر دو رأس دلخواه یک گراف است (تمام رئوس). این الگوریتم علاوه بر خود مسیر اندازه آن را هم به‌دست می‌آورد.

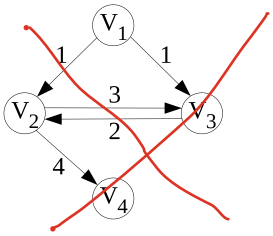
انواع کوتاهترین مسیرها

۱) از یک مبدا به یک مقصد
۲) از یک مبدا به همه رأس‌ها
۳) از همه رأس‌ها به همه رأس‌ها
۴) از همه رأس‌ها به یک رأس

تعاریف:

۱) گراف مورد بحث در این الگوریتم یک گراف جهت‌دار و وزن‌دار (بدون وزن منفی) است.
۲) **مسیر:** مجموعه‌ای از رئوس است به‌طوری که از یک رأس به رأس دیگر یک یال وجود دارد.
۳) **مسیر ساده:** مسیری است که رأس تکراری ندارد، یعنی از یک رأس دو بار عبور نمی‌کنیم پس دور و طوقه بوجود نمی‌آیند.
۴) **طول مسیر:** مجموع وزن یال‌های موجود در مسیر است.
۵) کوتاهترین مسیر یک مسیر ساده است (دور ندارد).

مسئله کوتاهترین مسیر یک مسئله بهینه سازی است، زیرا زیرمسئله‌های آن (زیر مسیرها) بهینه هستند، یعنی از کنار هم قرار گرفتن کوتاهترین مسیرها، کوتاهترین مسیر به‌دست می‌آید. ولی مسئله طولانی‌ترین مسیر یک مسئله بهینه سازی نیست، زیرا از کنار هم قرار گرفتن مسیرهای طولانی لزوماً طولانی‌ترین مسیر به‌دست نمی‌آید. این مسئله جز مسائل NPC محسوب می‌شود. شکل زیر مثال نقضی است که این قضیه را نشان می‌دهد:

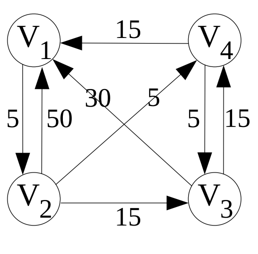


در این گراف طولانی‌ترین مسیر ساده از V_1 به V_4 به‌صورت زیر است:

$\langle V_1, V_3, V_2, V_4 \rangle \quad \rightarrow$ طولانی‌ترین مسیر 7 است

در حالی که ما می‌دانیم طولانی‌ترین مسیر از V_1 به V_3 مسیر $\langle V_1, V_2, V_3 \rangle$ است. در واقع در مسیر بالا از V_1 به V_3 طولانی‌ترین مسیر را در نظر نگرفتیم.

در مثال شکل زیر داریم:

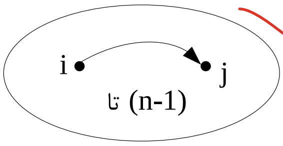


$\langle V_1, V_2, V_3 \rangle \quad \rightarrow$ یک مسیر ساده است

$\langle V_2, \underline{V_3}, V_4, \underline{V_3}, V_1 \rangle \quad \rightarrow$ یک مسیر ساده نیست زیرا رأس تکراری داریم

$$\left.\begin{array}{l} \langle V_2 \rightarrow V_3 \rangle : \langle V_2, V_3 \rangle = 15 \\ \langle V_2 \rightarrow V_3 \rangle : \langle V_2, V_4, V_3 \rangle = 10 \end{array}\right\} \rightarrow D[2,3] = 10 \rightarrow$$ کوتاهترین مسیر 10 است

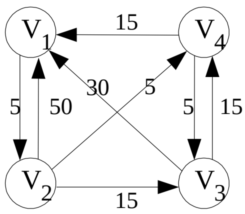
در گراف زیر اگر بخواهیم از رأس i به رأس j برویم، از رأس i به رأس بعدی (n-2) انتخاب وجود دارد و برای رأس بعدی نیز (n-3) انتخاب وجود دارد، بنابراین داریم:



گراف کامل با n راس

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

می‌خواهیم با بهینه سازی کاری کنیم که مرتبه زمانی این الگوریتم را از $O(n!)$ به $\theta(n^3)$ کاهش دهیم.



$$W=\begin{cases}0 & i=j\\ \infty & \text{ندارد یال } j \text{ به } i\\ \text{وزن یال} & \text{دارد یال } j \text{ به } i\end{cases}$$

ماتریس مجاورت یا ماتریس وزن گراف

$$W[i,j]=\begin{bmatrix}0 & 5 & \infty & \infty\\ 50 & 0 & 15 & 5\\ 30 & \infty & 0 & 15\\ 15 & \infty & 5 & 0\end{bmatrix}$$

$D[i,j]$ نهایی = کوتاهترین مسیر از V_i به V_j

$D^{(k)}[i,j]$ = یعنی کوتاهترین مسیر از V_i به V_j فقط با استفاده از رئوس میانی $V_1 \cdots V_k$

$D^{(1)}[i,j]$ = یعنی کوتاهترین مسیر از V_i به V_j فقط با استفاده از رأس میانی V_1 $\quad (V_i \rightarrow 1 \rightarrow V_j)$

$$D^{(0)}=W \longrightarrow D^{(1)} \longrightarrow D^{(2)} \longrightarrow \cdots \longrightarrow D^{(n)}$$

هیچ راس میانی در آن دخالت نداشته و مستقیم بوده اند.	راس V_1 راس میانی بوده است.	رئوس V_1 و V_2 رئوس میانی هستند.	شامل اندازه کوتاهترین مسیرها است.

$$D^{(1)}[i,j]=min\{D^{(0)}[i,j],D^{(0)}[i,1]+D^{(0)}[1,j]\}$$

کوتاهترین مسیر از i به j با استفاده از رأس V_1

$$D^{(2)}[i,j]=min\{D^{(1)}[i,j],D^{(1)}[i,2]+D^{(1)}[2,j]\}$$

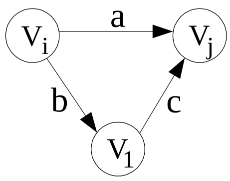
کوتاهترین مسیر از i به j با استفاده از رئوس V_1 و V_2

⋮
⋮

$$D^{(k)}[i,j] = min\left\{\underbrace{D^{(k-1)}[i,j]}, \underbrace{D^{(k-1)}[i,k] + D^{(k-1)}[k,j]}\right\}$$

کوتاهترین مسیر از i به j با استفاده از رئوس V_1 و V_2 … V_k که در $1 \le k \le n$ است.

فرض می‌کنیم می‌خواهیم از V_i به V_j برویم. می‌توانیم مستقیم برویم و یا از رأس کمکی V_1 استفاده کنیم. در نهایت بین مسیرها کوتاهترین را انتخاب می‌کنیم.

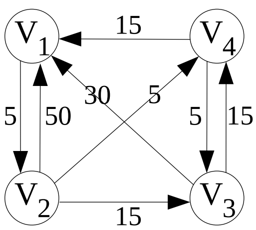


$$a \;\substack{>\\=\\<}\; b + c$$

قطعه برنامه الگوریتم فلوید $\boldsymbol{\theta(n^3)}$

```
for  (k=1; k<=n; k++)
     for (i=1; i<=n; i++)
          for (j=1; j<=n; j++)
                 D[ i ][ j ] = min( D[ i ][ j ] , D[ i ][ k ] + D[ k ][ j ] )
```

مثال: با استفاده از الگوریتم فلوید کوتاه‌ترین مسیرها را برای گراف زیر به‌دست آورید.



حل: ابتدا ماتریس مجاورت گراف را به‌دست می‌آوریم.

$$D^{(0)} = W = \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{bmatrix}$$

حال با استفاده از مقادیر ماتریس $D^{(0)}$ و جایگذاری در فرمول ماتریس $D^{(1)}$ را به‌دست می‌آوریم.

$$D^{(1)}[2,3] = min\{D^{(0)}[2,3], D^{(0)}[2,1] + D^{(0)}[1,3]\} = min\{15, \infty\} = 15$$

$$D^{(1)}[2,4] = min\{D^{(0)}[2,4], D^{(0)}[2,1] + D^{(0)}[1,4]\} = min\{5, \infty\} = 5$$

$$D^{(1)}[3,2] = min\{D^{(0)}[3,2], D^{(0)}[3,1] + D^{(0)}[1,2]\} = min\{\infty, 35\} = 35$$

$$D^{(1)}[3,4] = min\{D^{(0)}[3,4], D^{(0)}[3,1] + D^{(0)}[1,4]\} = min\{15, \infty\} = 15$$

$$D^{(1)}[4,2] = min\{D^{(0)}[4,2], D^{(0)}[4,1] + D^{(0)}[1,2]\} = min\{\infty, 20\} = 20$$

$$D^{(1)}[4,3] = min\{D^{(0)}[4,3], D^{(0)}[4,1] + D^{(0)}[1,3]\} = min\{5, \infty\} = 5$$

$$D^{(1)} = \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix}$$

هنگام محاسبه‌ی $D^{(i)}$ مقادیر سطر i و ستون i برای مرحله بعد تغییر نمی‌کنند، قطر نیز همواره صفر است، بنابراین آن‌ها را محاسبه نمی‌کنیم.

به همین ترتیب با استفاده از مقادیر ماتریس‌های قبلی و جایگذاری در فرمول‌ها ماتریس‌های جدید را به‌دست می‌آوریم.

$$D^{(2)} = \begin{bmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \quad D^{(3)} = \begin{bmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \quad D^{(4)} = \begin{bmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix}$$

برای این‌که بتوانیم کوتاهترین مسیر را به‌دست آوریم از ماتریس P (ماتریس مسیر) استفاده می‌کنیم. در مرحله اول که می‌خواهیم ماتریس مجاورت را به‌دست آوریم تمامی مقادیر ماتریس مسیر را صفر قرار می‌دهیم. سپس وقتی ماتریس $D^{(1)}$ را محاسبه کردیم تغییرات ایجاد شده در آن را در ماتریس P نیز اعمال می‌کنیم، یعنی در هر مرحله آخرین مقداری را که باعث می‌نیمم شدن D شده در ماتریس P قرار می‌دهیم و به این ترتیب هر بار که ماتریس‌های جدید را به‌دست می‌آوریم در صورت تغییر، مقادیر ماتریس P را نیز تغییر می‌دهیم.

$$P[i,j] = \begin{cases} 0 & \text{اگر هیچ راس واسطه‌ای وجود نداشته باشد} \\ \text{بزرگ‌ترین اندیس از یک راس واسطه روی کوتاهترین مسیر از } V_i \text{ به } V_j & \text{اگر حداقل یک راس واسطه وجود داشته باشد} \end{cases}$$

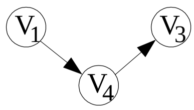
ماتریس مسیر نهایی، کوتاهترین مسیر و طول آن به‌صورت زیر به‌دست می‌آیند:

هیچ راس میانی در آن دخالت نداشته است.

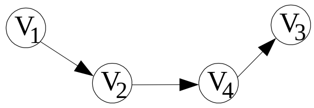
برای این که از ۱ به ۳ برویم آخرین بار از راس ۴ عبور کرده ایم. و راس ۴ باعث می نیمم شدن D شده است.

$$P = \begin{bmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$P[1,3] = 4$



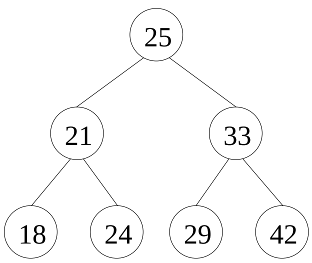
$P[1,4] = 2$



کوتاهترین مسیر $V_1 \to V_2 \to V_4 \to V_3$ است و طول آن 15 است.

۴-۵ درخت جستجوی دودویی بهینه OBST

درخت جستجوی دودویی (BST): یک درخت دودویی است که در آن مقدار هر گره از مقدار هر گره در زیردرخت سمت چپ آن بزرگ‌تر و از مقدار هر گره در زیردرخت سمت راست آن کوچک‌تر است.



هر گره دارای یک کلید منحصر به‌فرد (یکتا) است که با محتوای گره متفاوت است. در این مسئله در جستجوی کلید هستیم.

تعدادی کلید داریم که مرتب شده هستند.

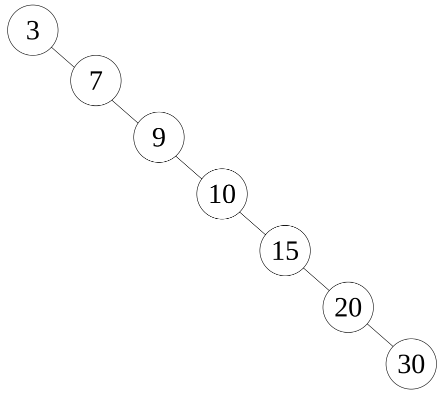
key1 < key2 < key3 < key4 < … < keyn

هر کلید یک احتمال دارد که احتمال keyi برابر با Pi است.

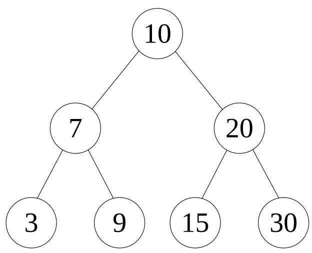
می‌خواهیم با استفاده از این کلیدها یک درخت جستجوی دودویی بهینه بسازیم.

درخت جستجوی دودویی بهینه (Optimal Binary Search Tree) OBST: یک درخت جستجوی دودویی است که در آن کلیدها به‌گونه‌ای قرار گرفته‌اند که میانگین جستجو برای یافتن همه‌ی کلیدها کمینه است.

- اگر احتمال رخدادها یکسان باشند، درخت حاصل متوازن می‌شود.
- اگر احتمال رخدادها متفاوت باشند کلیدهای با احتمال بیش‌تر به ریشه نزدیک‌تر هستند، یعنی با تعداد مقایسه کم‌تر به آن‌ها دسترسی پیدا می‌کنیم.
-



28مقایسه



17مقایسه

۱+ عمق = مقایسه

اگر درخت متوازن باشد تعداد مقایسه ها می نیمم می شود، به شرطی که گره ها متفاوت نباشند و احتمال رخداد همه یکسان باشد.

P_i =احتمال رخداد کلید i ام

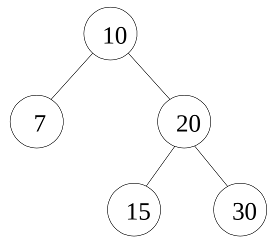
C_i = تعداد مقایسه‌های لازم برای یافتن کلید i ام

d_i = عمق کلید i ام

میانگین زمان جستجو در BST $= \sum_{i=1}^{n} C_i P_i = \sum_{i=1}^{n} (d_i + 1) P_i$

فرق بین درخت جستجوی دودویی با درخت جستجوی دودویی بهینه و زمان جستجو در آن‌ها در زیر نشان داده شده است.

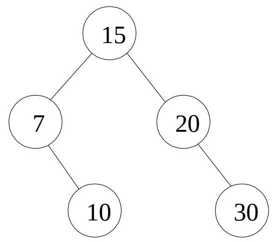
درخت جستجوی دودویی



	key1	key2	key3	key4	key5
value مقدار	7	10	15	20	30
P_i احتمال	0.2	0.1	0.5	0.1	0.1
$C_i = d_i + 1$	2	1	3	2	3
$C_i\, P_i$	0.4	0.1	1.5	0.2	0.3

$$\sum C_i P_i = \sum (d_i + 1) P_i = 2.5$$

درخت جستجوی دودویی بهینه



	key1	key2	key3	key4	key5
value مقدار	7	10	15	20	30
P_i احتمال	0.2	0.1	0.5	0.1	0.1
$C_i = d_i + 1$	2	3	1	2	3
$C_i\, P_i$	0.4	0.3	0.5	0.2	0.3

$$\sum C_i P_i = \sum (d_i + 1) P_i = 1.7$$

برای یافتن درخت جستجوی دودویی بهینه دو روش داریم:

۱. تمام حالت‌های ممکن درخت‌ها را رسم کرده و حالت بهینه را به‌دست آوریم. در این روش پیچیدگی زمانی از مرتبه $O(n!)$ می‌شود، زیرا برای یافتن تمام حالت‌های ممکن از فرمول عدد کاتالان استفاده می‌کنیم.

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{1}{n+1} \times \frac{2n!}{(2n-n)!\,n!} \quad \Longrightarrow \quad O(n!)$$

۲. با استفاده از الگوریتم OBST درخت را رسم می‌کنیم.

مثال: برای کلیدهای داده شده زیر درخت جستجوی دودویی بهینه را رسم کنید.

$\boldsymbol{A \rightarrow key1 \rightarrow P_1 = 0.7}$

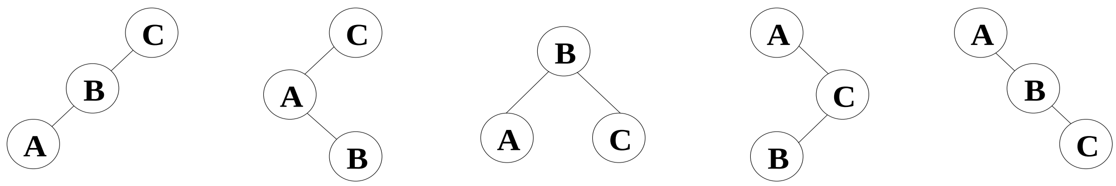
$\boldsymbol{B \rightarrow key2 \rightarrow P_2 = 0.2}$

$\boldsymbol{C \rightarrow key3 \rightarrow P_3 = 0.1}$

در این مثال برای رسم OBST از روش اول استفاده می‌کنیم. ابتدا با استفاده از عدد کاتالان تعداد حالات ممکن برای رسم درخت را به‌دست می‌آوریم:

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{1}{3+1}\binom{6}{3} = 5$$

5 درخت جستجوی دودویی بهینه به صورت زیر می‌توان رسم کرد:



1 2 3 4 5

درخت 1 $\rightarrow 3 \times 0.7 + 2 \times 0.2 + 1 \times 0.1 = 2.6$

(1 مقایسه برای C که احتمال آن 0.1 است) + (2 مقایسه برای B که احتمال آن 0.2 است) + (3 مقایسه برای A که احتمال آن 0.7 است)

درخت 2 $\rightarrow 2 \times 0.7 + 3 \times 0.2 + 1 \times 0.1 = 2.1$

درخت 3 $\rightarrow 2 \times 0.7 + 1 \times 0.2 + 2 \times 0.1 = 1.8$

درخت 4 $\rightarrow 1 \times 0.7 + 3 \times 0.2 + 2 \times 0.1 = 1.5$

درخت 5 $\rightarrow 1 \times 0.7 + 2 \times 0.2 + 3 \times 0.1 = 1.4$

از نتایج به‌دست آمده در بالا می‌توان نتیجه گرفت درخت 5 که کمترین مقدار را دارد یک درخت جستجوی دودویی بهینه است.

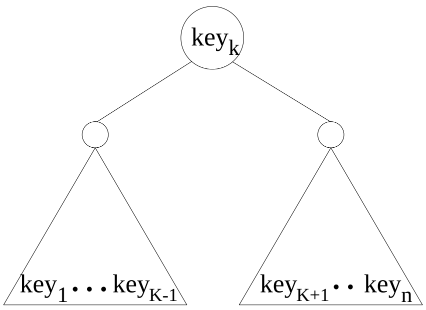
۴-۵-۱ اصل بهینگی

$$BST \text{ میانگین زمان جستجو در } = \sum_{i=1}^{n} C_i P_i = \sum_{i=1}^{n} (d_i + 1) P_i$$

با استفاده از الگوریتم OBST می‌خواهیم کاری کنیم که میانگین زمان جستجو بهینه (می‌نیمم) شود.

n تا کلید داریم، نمی‌دانیم کدام کلید را در ریشه قرار دهیم تا درخت حاصل بهینه شده و میانگین زمان جستجو کمینه شود.

فرض می‌کنیم اگر key_k را به‌عنوان ریشه قرار دهیم درخت حاصل OBST می‌شود، در این صورت رابطه‌ی زیر حاصل می‌شود:



$$A[1,n] = \overbrace{A[1,k-1]}^{\text{میانگین زمان جستجو در زیردرخت چپ}} + \underset{\text{زمان اضافی برای مقایسه در ریشه}}{P_1 + P_2 + \cdots + P_{k-1}} + \overbrace{P_k}^{\text{زمان جستجو برای ریشه}} + \underbrace{A[k+1,n]}_{\text{میانگین زمان جستجو در زیردرخت راست}} + \overbrace{P_{k+1} + \cdots + P_n}^{\text{زمان اضافی برای مقایسه در ریشه}}$$

$$A[1,n] = A[1,k-1] + A[k+1,n] + \sum_{m=1}^{n} P_m$$

حال به ازای k های متفاوت $(1 \le k \le n)$ مقادیر را به‌دست آورده و حالت کمینه را انتخاب می‌کنیم:

$$A[1,n] = \min_{1 \le k \le n}\{A[1,k-1] + A[k+1,n]\} + \sum_{m=1}^{n} P_m$$

حداقل زمان جستجو در یک BST با کلیدهای $key_i < key_{i+1} < \cdots < key_j$ که در آن $1 \le i \le j \le n$ به‌صورت زیر بدست می‌آید:

$$A[i,j] = \begin{cases} \min\limits_{i \le k \le j}\{A[i,k-1] + A[k+1,j]\} + \sum_{m=i}^{j} P_m & i < j \\ P_i & i = j \end{cases}$$

۴-۵-۲ جدول $(n+1)\times(n+1)$

برای رسم OBST دو جدول $(n+1)\times(n+1)$ با نام‌های A و R به‌صورت زیر رسم می‌کنیم:

مثال: برای کلیدهای داده شده زیر درخت جستجوی دودویی بهینه را رسم کنید.

$\boldsymbol{A \rightarrow key1 \rightarrow P_1 = 0.7}$

$\boldsymbol{B \rightarrow key2 \rightarrow P_2 = 0.2}$

$\boldsymbol{C \rightarrow key3 \rightarrow P_3 = 0.1}$

0 . . . . n

1 . . . n+1

	0	1	2	3
1	0	0.7	1.1	1.4
2		0	0.2	0.4
3			0	0.1
4				0

بهترین زمان برای جستجو

قطر اصلی صفر می شود

ماتریس ها قطری پر می شوند

در قطر اول احتمال ها را می نویسیم

جدول A

0 . . . . n

1 . . . n+1

	0	1	2	3
1	0	1	1	1
2		0	2	2
3			0	3
4				0

از بین 3کلید باید کلید1در ریشه قرار داشته باشد تا درخت بهینه رسم شود

از 3 تا 3 یک ریشه می باشد که خود 3 می شود

جدول R

$$A[1,2] = min\begin{Bmatrix} k=1 \rightarrow & A[1,0]+A[2,2] \\ k=2 \rightarrow & A[1,1]+A[3,2] \end{Bmatrix} + P_1 + P_2 = min\begin{Bmatrix} 0+0.2 \\ 0.7+0 \end{Bmatrix} + 0.7 + 0.2 = 1.1$$

$$A[2,3] = min\begin{Bmatrix} k=2 \rightarrow & A[2,1]+A[3,3] \\ k=3 \rightarrow & A[2,2]+A[4,3] \end{Bmatrix} + P_2 + P_3 = min\begin{Bmatrix} 0+0.1 \\ 0.2+0 \end{Bmatrix} + 0.2 + 0.1 = 0.4$$

$$A[1,3] = min\begin{Bmatrix} k = 1 \rightarrow & A[1,0] + A[2,3] \\ k = 2 \rightarrow & A[1,1] + A[3,3] \\ k = 3 \rightarrow & A[1,2] + A[4,3] \end{Bmatrix} + P_1 + P_2 + P_3 = min\begin{Bmatrix} 0 + 0.4 \\ 0.7 + 0.1 \\ 1.1 + 0 \end{Bmatrix} + 0.7 + 0.2 +$$
$$= 1.4$$

حال با استفاده از جدول R درخت OBST را رسم می‌کنیم.

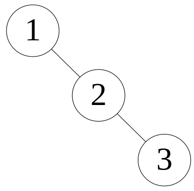
$R[1,3] = 1 \rightarrow$ کلید 1 باید در ریشه قرار گیرد

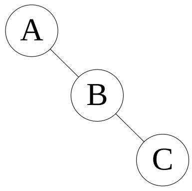
$R[2,3] = 2 \rightarrow$ 2 فرزند 1 است و چون بزرگ‌تر از 1 است در سمت راست 1 قرار می‌گیرد

در نهایت کلید 3 باقی می‌ماند که چون بزرگ‌تر از 2 است در سمت راست آن قرار می‌گیرد.

R[1,3]=1

R[2,3]=2

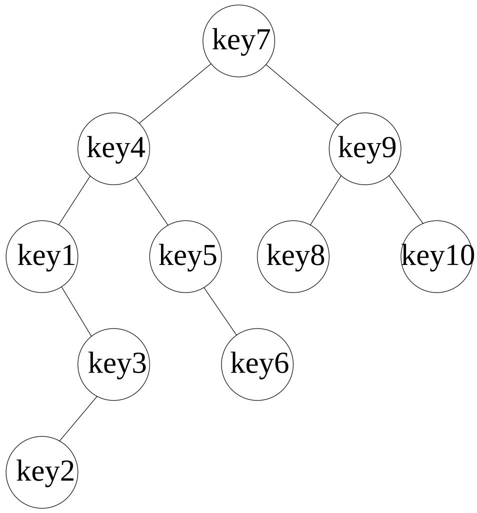




مثال: فرض می‌کنیم 10 کلید $k_1 \dots k_n$ داریم و هم‌چنین مقادیر زیر را از جدول R به‌دست آورده‌ایم،

$\boldsymbol{R[1,10] = 7} \quad \boldsymbol{R[1,6] = 4} \quad \boldsymbol{R[8,10] = 9} \quad \boldsymbol{R[1,3] = 1} \quad \boldsymbol{R[2,3] = 3} \quad \boldsymbol{R[5,6] = 5}$

درخت حاصل به شکل زیر در می‌آید:



۴-۶ فروشنده دوره‌گرد TSP (Travelling Salesman Problem)

کوتاهترین مسیر را با شروع از یک شهر و بازگشت به شهر اولیه به‌دست می‌آورد به‌طوری که از همه شهرها یک‌بار عبور کنیم.

گراف موجود یک گراف جهت‌دار و وزن‌دار (بدون وزن منفی) است.

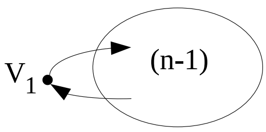
تور یا مدار هامیلتونی: مسیری از یک رأس به خودش است که از هر رأس دقیقاً یک‌بار عبور می‌کند.

تور بهینه: در یک گراف جهت‌دار و وزن‌دار یک تور با حداقل وزن است، یعنی تور با طول حداقل است.

رأس شروع تاثیری بر طول تور بهینه ندارد بنابراین v_1 را به عنوان رأس شروع انتخاب می‌کنیم.

برای یافتن تور بهینه در یک گراف به دو روش می‌توانیم عمل کنیم:

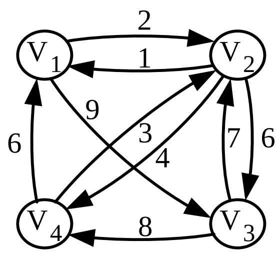
۱) **روش غیر هوشمندانه:** تمام تورهای ممکن را به‌دست آورده و طول آن‌ها را محاسبه می‌کنیم و از میان آن‌ها مقدار کمینه را انتخاب می‌کنیم. در این صورت مرتبه زمانی یافتن تور بهینه در یک گراف کامل با n رأس به‌صورت زیر به‌دست می‌آید:



گراف با n راس

$$(n-1)\times(n-2)\times(n-3)\times\cdots\times 2\times 1=(n-1)! \longrightarrow O(n!)$$

مثال: در گراف شکل زیر طول تور بهینه و مسیر آن را به‌دست آورید.



حل به روش غیر هوشمندانه:

نکته: کل تورهای ممکن در گراف کامل با n رأس از رابطه زیر به‌دست می‌آید:

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

کل تورهای ممکن در گراف را به‌دست می‌آوریم و از بین آن‌ها مقدار کمینه را انتخاب می‌کنیم.

$\{V_1, V_2, V_3, V_4, V_1\} = 22$

$\{V_1, V_3, V_2, V_4, V_1\} = 26$

$\{V_1, V_3, V_4, V_2, V_1\} = 21 \quad \rightarrow$ تور بهینه

۲) **الگوریتم فروشنده دوره‌گرد:** با استفاده از این الگوریتم مرتبه زمانی یافتن تور بهینه را کاهش می‌دهیم.

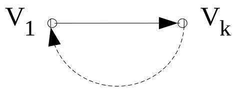
$\{\cdots\cdots\cdots\}$ = مجموعه رئوس (ترتیب در آن مهم نیست)

$[\cdots\cdots\cdots]$ = مسیر (ترتیب در آن مهم است).

V = مجموعه شامل تمام رئوس گراف است.

$(A \subset V)$ = زیرمجموعه‌ای از رئوس گراف است.

اصل بهینگی: اگر v_k (راس k ام) نخستین راس پس از v_1 روی هر تور بهینه باشد زیر مسیر آن تور از v_k به v_1 باید کوتاهترین مسیر از v_k به v_1 باشد که از هر کدام از رئوس دیگر دقیقاً یک‌بار عبور کند.



V-{V1, Vk}

$D[V_i][A]$ = طول کوتاهترین مسیر از V_i به V_1 که از هر رأس در مجموعه A دقیقاً یک‌بار عبور می‌کند.

$V - \{V_1, V_j\}$ = شامل همه رئوس به‌جز V_1 و V_j است.

مثال: فرض کنید مجموعه‌های $\boldsymbol{V = \{v_1, v_2, v_3, v_4\}}$, $\boldsymbol{A = \{v_3, v_4\}}$ را داریم، $\boldsymbol{D[V_2][A]}$ را به‌دست آورید.

$D[v_2][A] = min\{length[v_2, v_3, v_4, v_1], length[v_2, v_4, v_3, v_1]\} = min(20, \infty) = 20$

مثال: فرض کنید مجموعه‌های $\boldsymbol{V = \{v_1, v_2, v_3, v_4\}}$, $\boldsymbol{A = \{v_3\}}$ را داریم، $\boldsymbol{D[V_2][A]}$ را به‌دست آورید.

$D[v_2][A] = length[v_2, v_3, v_1] = \infty$

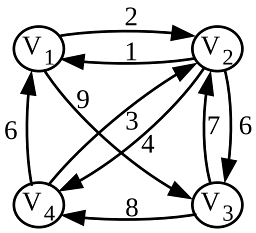
طول تور بهینه $= min\big(W[1][j] + D[V_j][V - \{V_1, V_j\}]\big) \quad \rightarrow \quad 2 \le j \le n$

$$D[V_i][A] = \min_{v_j \epsilon A}\big(W[i][j] + D[V_j][A - \{V_j\}]\big) \quad \rightarrow \quad v_i \notin A \,, i \neq 1 \,, A \neq \phi$$

از i به 1 می‌رویم با استفاده از رئوس مجموعه A

$D[V_i][\emptyset] = W[i][1] \quad \rightarrow$ از i به 1 می‌رویم بدون استفاده از هیچ رأسی (مسیر مستقیم)

مثال: در گراف شکل زیر طول و مسیر تور بهینه را با استفاده از الگوریتم فروشنده دوره‌گرد به‌دست آورید.



حل به روش TSP:

طول تور بهینه $= \min\limits_{2\le j\le n}\{W[1][j]+D[V_j][V-\{V_1,V_j\}]\}$

$$D[V_i][A]=\min_{\forall j\in A}\left(W[i][j]+D[V_j][A-\{V_j\}]\right)$$

$$D = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccccccc} \emptyset & \{V_2\} & \{V_3\} & \{V_4\} & \{V_2,V_3\} & \{V_2,V_4\} & \{V_3,V_4\} & \{V_2,V_3,V_4\} \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & \infty & 17 & 12 & 23 & 21 \\ 1 & 1 & \infty & 10 & 14 & 8 & 20 & 18 \\ \infty & 8 & \infty & 14 & 8 & 12 & 14 & 12 \\ 6 & 4 & \infty & 6 & \infty & 4 & \infty & 23 \end{bmatrix} \end{array}$$

$D[V_1][\emptyset]=W[1][1]=0$

$D[V_2][\emptyset]=W[2][1]=1$

$D[V_3][\emptyset]=W[3][1]=\infty$

$D[V_4][\emptyset]=W[4][1]=6$

$D[V_1][V_2]=W[1][2]+D[V_2][\emptyset]=2+1=3$

$D[V_2][V_2]=W[2][2]+D[V_2][\emptyset]=0+1=1$

$D[V_3][V_2]=W[3][2]+D[V_2][\emptyset]=7+1=8$

$D[V_4][V_2]=W[4][2]+D[V_2][\emptyset]=3+1=4$

$D[V_1][V_3]=W[1][3]+D[V_3][\emptyset]=9+\infty=\infty$

$D[V_2][V_3]=W[2][3]+D[V_3][\emptyset]=6+\infty=\infty$

$D[V_3][V_3]=W[3][3]+D[V_3][\emptyset]=0+\infty=\infty$

$D[V_4][V_3]=W[4][3]+D[V_3][\emptyset]=\infty+\infty=\infty$

$D[V_1][V_4] = W[1][4] + D[V_4][\emptyset] = \infty + 6 = \infty$

$D[V_2][V_4] = W[2][4] + D[V_4][\emptyset] = 4 + 6 = 10$

$D[V_3][V_4] = W[3][4] + D[V_4][\emptyset] = 8 + 6 = 14$

$D[V_4][V_4] = W[4][4] + D[V_4][\emptyset] = 0 + 6 = 6$

$$D[V_1][V_2V_3] = min\begin{Bmatrix} V_2 \rightarrow W[1][2] + D[V_2][V_3] = 2 + \infty = \infty \\ V_3 \rightarrow W[1][3] + D[V_3][V_2] = 9 + 8 = 17 \end{Bmatrix} = 17$$

$$D[V_2][V_2V_3] = min\begin{Bmatrix} V_2 \rightarrow W[2][2] + D[V_2][V_3] = 0 + \infty = \infty \\ V_3 \rightarrow W[2][3] + D[V_3][V_2] = 6 + 8 = 14 \end{Bmatrix} = 14$$

$$D[V_3][V_2V_3] = min\begin{Bmatrix} V_2 \rightarrow W[3][2] + D[V_2][V_3] = 7 + \infty = \infty \\ V_3 \rightarrow W[3][3] + D[V_3][V_2] = 0 + 8 = 8 \end{Bmatrix} = 8$$

$$D[V_4][V_2V_3] = min\begin{Bmatrix} V_2 \rightarrow W[4][2] + D[V_2][V_3] = 3 + \infty = \infty \\ V_3 \rightarrow W[4][3] + D[V_3][V_2] = \infty + 8 = \infty \end{Bmatrix} = \infty$$

$$D[V_1][V_2V_4] = min\begin{Bmatrix} V_2 \rightarrow W[1][2] + D[V_2][V_4] = 2 + 10 = 12 \\ V_4 \rightarrow W[1][4] + D[V_4][V_2] = \infty + 4 = \infty \end{Bmatrix} = 12$$

$$D[V_2][V_2V_4] = min\begin{Bmatrix} V_2 \rightarrow W[2][2] + D[V_2][V_4] = 0 + 10 = 10 \\ V_4 \rightarrow W[2][4] + D[V_4][V_2] = 4 + 4 = 8 \end{Bmatrix} = 8$$

$$D[V_3][V_2V_4] = min\begin{Bmatrix} V_2 \rightarrow W[3][2] + D[V_2][V_4] = 7 + 10 = 17 \\ V_4 \rightarrow W[3][4] + D[V_4][V_2] = 8 + 4 = 12 \end{Bmatrix} = 12$$

$$D[V_4][V_2V_4] = min\begin{Bmatrix} V_2 \rightarrow W[4][2] + D[V_2][V_4] = 3 + 10 = 13 \\ V_4 \rightarrow W[4][4] + D[V_4][V_2] = 0 + 4 = 4 \end{Bmatrix} = 4$$

$$D[V_1][V_3V_4] = min\begin{Bmatrix} V_3 \rightarrow W[1][3] + D[V_3][V_4] = 9 + 14 = 23 \\ V_4 \rightarrow W[1][4] + D[V_4][V_3] = \infty + \infty = \infty \end{Bmatrix} = 23$$

$$D[V_2][V_3V_4] = min\begin{Bmatrix} V_3 \rightarrow W[2][3] + D[V_3][V_4] = 6 + 14 = 20 \\ V_4 \rightarrow W[2][4] + D[V_4][V_3] = 4 + \infty = \infty \end{Bmatrix} = 20$$

$$D[V_3][V_3V_4] = min\begin{Bmatrix} V_3 \rightarrow W[3][3] + D[V_3][V_4] = 0 + 14 = 14 \\ V_4 \rightarrow W[3][4] + D[V_4][V_3] = 8 + \infty = \infty \end{Bmatrix} = 14$$

$$D[V_4][V_3V_4] = min\begin{Bmatrix} V_3 \rightarrow W[4][3] + D[V_3][V_4] = \infty + 14 = \infty \\ V_4 \rightarrow W[4][4] + D[V_4][V_3] = 0 + \infty = \infty \end{Bmatrix} = \infty$$

$$D[V_1][V_2V_3V_4] = min \begin{Bmatrix} V_2 \to W[1][2] + D[V_2][V_3V_4] = 2 + 20 = 22 \\ V_3 \to W[1][3] + D[V_3][V_2V_4] = 9 + 12 = 21 \\ V_4 \to W[1][4] + D[V_4][V_2V_3] = \infty + \infty = \infty \end{Bmatrix} = 21$$

$$D[V_2][V_2V_3V_4] = min \begin{Bmatrix} V_2 \to W[2][2] + D[V_2][V_3V_4] = 0 + 20 = 22 \\ V_3 \to W[2][3] + D[V_3][V_2V_4] = 6 + 12 = 18 \\ V_4 \to W[2][4] + D[V_4][V_2V_3] = 4 + \infty = \infty \end{Bmatrix} = 18$$

$$D[V_3][V_2V_3V_4] = min \begin{Bmatrix} V_2 \to W[3][2] + D[V_2][V_3V_4] = 7 + 20 = 27 \\ V_3 \to W[3][3] + D[V_3][V_2V_4] = 0 + 12 = 12 \\ V_4 \to W[3][4] + D[V_4][V_2V_3] = 8 + \infty = \infty \end{Bmatrix} = 12$$

$$D[V_4][V_2V_3V_4] = min \begin{Bmatrix} V_2 \to W[4][2] + D[V_2][V_3V_4] = 3 + 20 = 23 \\ V_3 \to W[4][3] + D[V_3][V_2V_4] = \infty + 12 = \infty \\ V_4 \to W[4][4] + D[V_4][V_2V_3] = 0 + \infty = \infty \end{Bmatrix} = 23$$

حال با استفاده از معلومات به‌دست آمده از محاسبات بالا ماتریس مسیر P را به‌دست می‌آوریم:

$$P = \begin{array}{c|cccccccc|} & \varnothing & \{V_2\} & \{V_3\} & \{V_4\} & \{V_2,V_3\} & \{V_2,V_4\} & \{V_3,V_4\} & \{V_2,V_3,V_4\} \\ \hline V_1 & - & V_2 & V_3 & V_4 & V_3 & V_2 & V_3 & V_3 \\ V_2 & - & V_2 & V_3 & V_4 & V_3 & V_4 & V_3 & V_3 \\ V_3 & - & V_2 & V_3 & V_4 & V_3 & V_4 & V_3 & V_3 \\ V_4 & - & V_2 & V_3 & V_4 & V_3 & V_4 & V_3 & V_2 \end{array}$$

با استفاده از ماتریس مسیر به‌دست آمده مسیر و طول تور بهینه را به‌صورت زیر محاسبه می‌کنیم:

$P[V_1][V_2V_3V_4] = V_3 \quad \Rightarrow \quad V_1 \longrightarrow V_3$

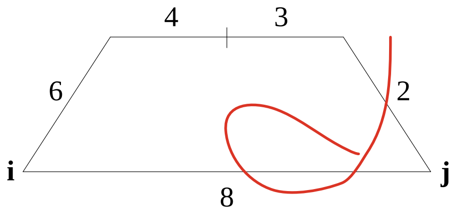
$P[V_3][V_2V_4] = V_4 \quad \Rightarrow \quad V_1 \longrightarrow V_3 \longrightarrow V_4$

$P[V_4][V_2] = V_2 \quad \Rightarrow \quad V_1 \longrightarrow V_3 \longrightarrow V_4 \longrightarrow V_2$

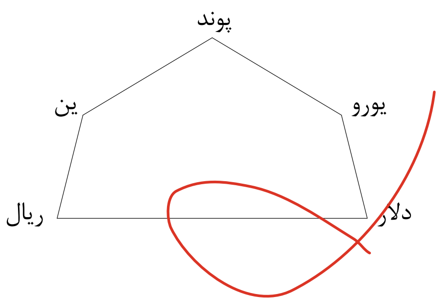
مسیر و طول تور بهینه $V_1 \underset{9}{\to} V_3 \underset{8}{\to} V_4 \underset{3}{\to} V_2 \underset{1}{\to} V_1 \quad \Rightarrow \quad 21$

❖ مرتبه زمانی الگوریتم فروشنده دوره‌گرد $O(n^2.2^n)$ است. مرتبه زمانی به‌دست آمده نمایی است بنابراین این مسئله جز مسائل NPC است.

✓ **سئوال: گراف زیر نشان دهنده نقشه یک شهر با محل تقاطع‌ها و خیابان‌ها است. وزن هر یال نشان دهنده‌ی میزان ترافیک در آن خیابان است. برای رفتن از i به j چه مسیری را انتخاب کنیم که سریع‌تر به مقصد برسیم؟**

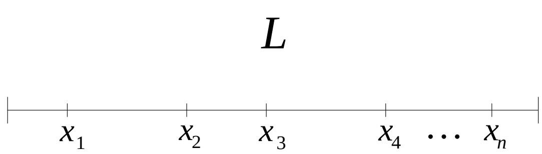


✓**سئوال: مقداری پول به صورت ریال داریم که می‌خواهیم آن را به دلار تبدیل کنیم. با توجه به شکل زیر کدام تبدیل ارز را انتخاب کنیم که سود آن بیش‌تر باشد؟**



✓ **سئوال: برای الگوریتم چوب بری زیر راه حل پویا ارائه دهید و آن را تحلیل نمایید.**

یک قطعه چوب به طول L داده شده است. می‌خواهیم این چوب را از مختصات‌های $x_1 < x_2 < \cdots < x_n$ نسبت به ابتدای چوب از سمت چپ ببریم. برای این کار باید n بار یک قطعه چوب که در ابتدا همان قطعه چوب اصلی است را برداریم و در ماشین قرار دهیم و از نقطه‌ای ببریم و آن را به دو قطعه کوچک‌تر تقسیم کنیم. می‌دانیم که هزینه‌ی برش قطعه چوبی به طول k برابر k است. به چه ترتیب چوب اصلی را از نقاط داده شده ببریم تا مجموع هزینه برش‌ها کیمنه شود؟



فصل پنجم: الگوریتم حریصانه (Greedy Algorithm)

۵-۱ الگوریتم حریصانه

الگوریتم حریصانه الگوریتمی است که برای تصمیم‌گیری و انتخاب در هر لحظه فقط شرایط خاص آن لحظه را در نظر می‌گیرد و به انتخاب‌های گذشته و نحوه عمل آینده کاری ندارد و در هر مرحله سعی می‌کند بهترین عملی را که می‌تواند انجام دهد.

- الگوریتم‌های حریصانه لزوماً بهینه نیستند ولی الگوریتم‌های ساده‌ای هستند و معمولاً مرتبه زمانی آن‌ها پایین است (مرتبه‌ی آن‌ها غالباً چند جمله‌ای است).
- الگوریتم‌های حریصانه سریع هستند ولی دقت لازم را در به‌دست آوردن جواب ندارند.
- برخی الگوریتم‌های حریصانه جواب بهینه تولید می‌کنند، یعنی هم سریع هستند و هم پاسخ آن‌ها بهینه است.

مسائلی که با الگوریتم‌های حریصانه قابل حل هستند دارای خصوصیات زیر هستند:

۱. مسائل بهینه سازی هستند.
۲. برای حل بهینه مسائل باید زیر مسائل آن‌ها نیز بهینه حل شوند.
۳. انتخاب حریصانه در این گونه مسائل بهترین انتخاب است و عوض نمی‌شود.

مسائلی که به روش پویا حل می‌شدند نیز ویژگی‌های ۱ و ۲ را دارند.

مسائلی که به روش حریصانه حل می‌شوند مراحل زیر را طی می‌کنند:

۱. روال انتخاب: یعنی عناصر بعدی بر اساس یک معیار خاص که به نظر بهینه می‌آید انتخاب می‌شوند.
۲. تحقیق عملی بودن: امکان پذیر بودن را بررسی می‌کند.
۳. تحقیق حل

۵-۱-۱ الگوریتم حریصانه خرد کردن پول

در این الگوریتم هدف خرد کردن اسکناس درشت با استفاده از اسکناس‌های ریز است بطوری‌که کم‌ترین اسکناس ریز استفاده شود. فرض می‌کنیم تعداد اسکناس‌های ریز بی‌نهایت است (به تعداد کافی داریم). بهینه بودن یا نبودن این الگوریتم بستگی به نحوه انتخاب واحدهای پولی دارد.

مثال: می‌خواهیم ۱۵ تومان پول را خرد کنیم. واحدهای پولی که در اختیار داریم ۱۲، ۵ و ۱ تومانی هستند. به چه صورت می‌توانیم آن را خرد کنیم؟

روش حریصانه: مقدار ۱۵ تومان به‌صورت ۱،۱،۱،۱۲ تومانی خرد می‌شود.

روش غیرحریصانه و بهینه: مقدار ۱۵ تومان به‌صورت ۵،۵،۵ تومانی خرد می‌شود.

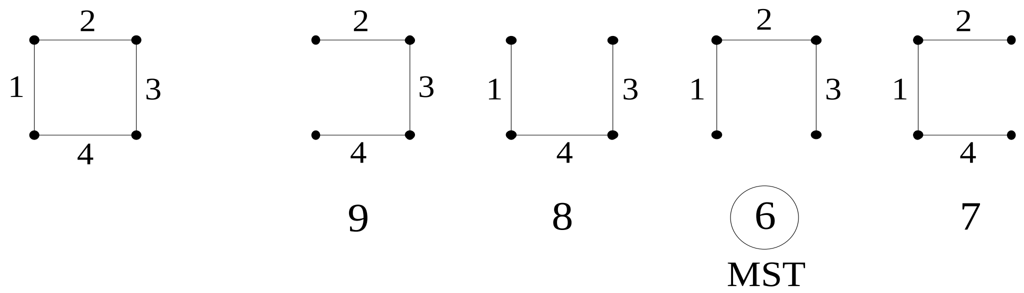
همان‌گونه که دیده می‌شود در روش حریصانه از ۴ اسکناس و در روش غیرحریصانه از ۳ اسکناس استفاده شده است، بنابراین معلوم می‌شود که روش حریصانه لزوماً بهینه نیست.

۵-۲ درخت پوشای کمینه(مینیمم) (Minimum Spanning Tree) MST

درخت: یک گراف همبند بدون دور را درخت می‌گوییم.

درخت پوشا: اگر یال‌های یک گراف همبند را آنقدر حذف کنیم که در نهایت یک گراف همبند بدون دور داشته باشیم، به آن درخت، درخت پوشا می‌گوییم. به عبارت دیگر یک درخت پوشا شامل همه‌ی رئوس و بعضی از یال‌ها است و منحصر بفرد نیست.

درخت پوشای مینیمم: اگر درخت پوشایی داشته باشیم که مجموع وزن یال‌های آن کمترین مقدار باشد به آن درخت پوشای مینیمم می‌گوییم. درخت پوشای مینیمم از نظر وزنی یکتا است ولی از لحاظ شکلی ممکن است یکتا نباشد.



با استفاده از الگوریتم‌های پریم (Prim) و کروسکال (Kruskal) درخت پوشای مینیمم را به‌دست می‌آوریم.

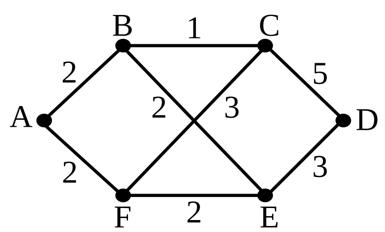
۵-۲-۱ الگوریتم پریم (Prim)

از این الگوریتم برای به‌دست آوردن درخت پوشای مینیمم استفاده می‌کنیم.

مراحل الگوریتم:

۱. یک رأس دلخواه به عنوان رأس شروع انتخاب می‌کنیم (مثلاً v_1 را به عنوان رأس شروع انتخاب می‌کنیم. $V'=\{v_1\}$)
۲. کم‌وزن‌ترین یال متصل به عناصر مجموعه‌ی V' را انتخاب می‌کنیم (فرض کنید v_k رأسی باشد که انتخاب می‌شود، یعنی v_k رأس دیگر یال انتخابی است)
۳. رأس جدید را به مجموعه‌ی قبلی اضافه می‌کنیم $V' = V' \cup \{v_k\}$ $(V' = \{v_1, v_k\})$
۴. اگر $V'=V$ باشد (کل رئوس انتخاب شده باشند) الگوریتم پایان می‌یابد، در غیر این‌صورت مرحله ۲ تکرار می‌شود.

مثال: درخت پوشای مینیمم (MST) را برای گراف زیر با استفاده از الگوریتم پریم به‌دست آورید.

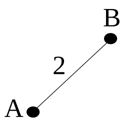


مجموعه رئوس $V=\{A , B , C , D , E , F\}$ است.

$V'=\{A\}$

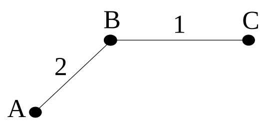
راس A را به عنوان راس شروع انتخاب می کنیم و
کم وزن ترین یال متصل به آن را رسم می کنیم.

$V'=\{A , B\}$

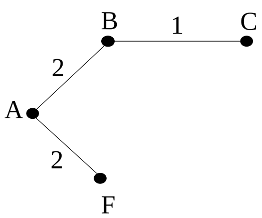


به همین ترتیب کم وزن ترین یال های متصل به
مجموعه رئوس جدید را به ترتیب انتخاب می کنیم.

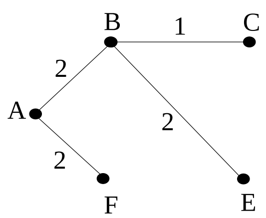
$V'=\{A , B , C\}$



$V'=\{A , B , C , F\}$

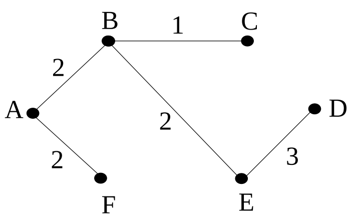


$V'=\{A , B , C , F , E\}$



پایان الگوریتم
درخت پوشای کمینه با وزن 10 حاصل شد.

$V'=\{A , B , C , F , E , D\}=V$



چند نکته:

- در الگوریتم پریم رأس شروع دلخواه انتخاب می‌شود.
- جواب الگوریتم پریم بهینه است.
- این الگوریتم به‌صورت پیوسته عمل می‌کند، یعنی در مراحل میانی گراف همبند باقی می‌ماند.
- این الگوریتم بر روی رئوس عمل می‌کند.
- اندازه وزن MST یکتاست ولی ممکن است شکل‌های مختلفی داشته باشد.

۵-۲-۱-۱ مرتبه زمانی الگوریتم پریم

در گراف با n رأس پس از انتخاب رأس شروع (n-1) رأس باقی می‌ماند، پس از انتخاب رأس دوم (n-2) رأس باقی می‌ماند و این ادامه می‌یابد تا جایی که همه رئوس انتخاب شوند، بنابراین داریم:

$$n + (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i = \theta(n^2) = \theta(V^2)$$

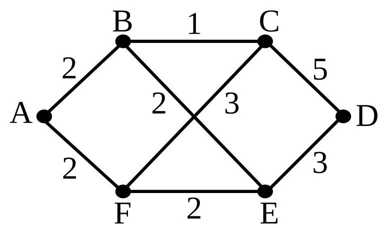
۵-۲-۲ الگوریتم کروسکال (Kruskal) یا راشال

از این الگوریتم برای به‌دست آوردن درخت پوشای مینیمم استفاده می‌کنیم.

مراحل الگوریتم:

۱. یال‌ها را بر اساس وزن آن‌ها به ترتیب صعودی مرتب می‌کنیم.
۲. یال‌های مرتب شده را از وزن کم به ترتیب به یک گراف تهی اضافه می‌کنیم. قبل از اضافه کردن هر یال باید این نکته را بررسی کنیم که آیا اضافه کردن یال باعث ایجاد دور می‌شود یا نه (نباید دور ایجاد شود).

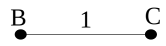
مثال: درخت پوشای مینیمم (MST) را برای گراف زیر با استفاده از الگوریتم کروسکال به‌دست آورید.



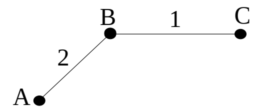
ابتدا یال‌ها را بر اساس وزن آن‌ها به‌صورت صعودی مرتب می‌کنیم:

$$e(B,C) < e(A,B) < e(B,E) < e(A,F) < e(F,E) < e(C,F) < e(E,D) < e(C,D)$$

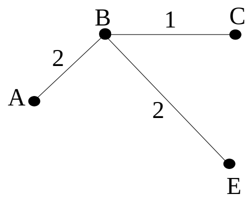
یال اول



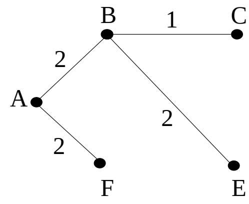
یال دوم



یال سوم



یال چهارم

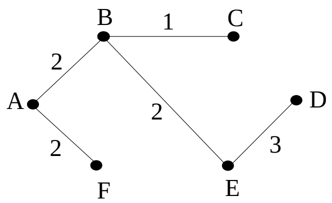


در مرحله پنجم یال e(F,E) باعث ایجاد دور می شود پس آن را انتخاب نمی کنیم و به سراغ یال بعدی می رویم.

در مرحله ششم یال e(C,F) باعث ایجاد دور می شود پس آن را انتخاب نمی کنیم و به سراغ یال بعدی می رویم.

درخت پوشای کمینه با وزن 10 حاصل شد.

یال هفتم



در مرحله هشتم نیز یال e(C,D) باعث ایجاد دور می شود پس آن را انتخاب نمی کنیم.
در این مرحله الگوریتم به پایان می رسد.

چند نکته:

- جواب الگوریتم کروسکال بهینه است.
- از نظر وزنی پاسخ این الگوریتم مانند الگوریتم پریم است.
- این الگوریتم به‌صورت گسسته عمل می‌کند. یعنی در مراحل میانی ممکن است گراف ناهمبند باشد.
- این الگوریتم بر روی یال‌ها عمل می‌کند.

۱-۲-۲-۵ مرتبه زمانی الگوریتم کروسکال

این الگوریتم در مرحله اول عمل مرتب سازی یال‌ها را انجام می‌دهد. اگر تعداد یال‌ها E باشد مرتبه زمانی این کار $\theta(E\log E)$ می‌شود.

$$\overbrace{(V-1)}^{\text{درخت}} \leq E \leq \overbrace{\left(\frac{V(V-1)}{2}\right)}^{\text{گراف کامل}}$$

$$\theta(V\log V) \qquad\qquad \theta(V^2\log V)$$

اگر گراف به سمت درخت برود، تعداد یال‌ها کم‌تر می‌شوند، در این حالت الگوریتم کروسکال مناسب‌تر است.

اگر گراف به سمت گراف کامل برود، تعداد یال‌ها بیش‌تر می‌شوند، در این حالت الگوریتم پریم $\left(\theta(v^2)\right)$ مناسب‌تر است.

۳-۲-۵ مقایسه الگوریتم‌های پریم و کروسکال

الگوریتم پریم	الگوریتم کروسکال
حریصانه	حریصانه
MST	MST
جواب بهینه	جواب بهینه
وزن‌ها برابر	وزن‌ها برابر
بر روی رئوس عمل می‌کند	بر روی یال‌ها عمل می‌کند
پیوسته است	گسسته است
$\theta(V^2)$	$\theta(V^2 \log V)$ یا $\theta(V \log V)$

۳-۵ الگوریتم دایکسترا (Dijkstra)

این الگوریتم برای پیدا کردن کوتاهترین مسیر از یک رأس به سایر رئوس به‌کار می‌رود. در این الگوریتم گراف به‌کار برده شده وزن دار (بدون وزن منفی) و جهت دار است.

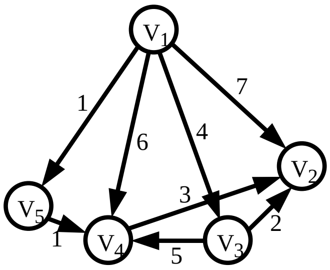
ایده اصلی این الگوریتم انتخاب گره S_i و محاسبه‌ی کوتاهترین فاصله در ترتیب $S_0, S_1, S_2, \ldots, S_{n-1}$ است.

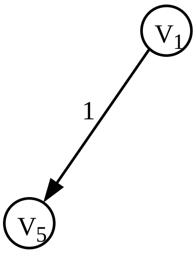
$$d(S,S_0) \le d(S,S_1) \le d(S,S_2) \le \cdots \le d(S,S_{n-1})$$

روش اول:

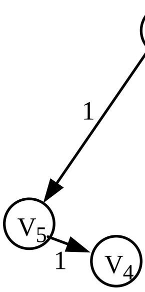
۱. یالی را انتخاب می‌کنیم که شامل گره مبدا باشد. با انتخاب این انتخاب مجموعه‌ای با دو گره ایجاد می‌شود.
۲. کوتاهترین مسیر از این مجموعه به بقیه‌ی گره‌ها را می‌یابیم و از بین آن‌ها کوتاهترین را انتخاب می‌کنیم. با انتخاب چنین یالی یک گره دیگر به مجموعه اضافه می‌شود.
۳. مرحله دوم را تا افزودن تمام گره‌ها به مجموعه تکرار می‌کنیم.

مثال: الگوریتم دیکسترا را با شروع از رأس v_1 روی گراف زیر اعمال کنید.



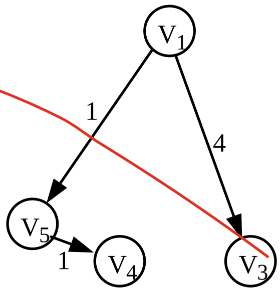


کوچک ترین یال شامل راس مبدا $e(v_1\ v_5)$ است



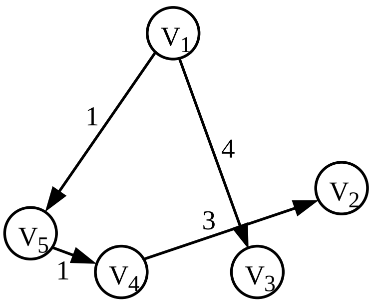
کوتاهترین مسیر از V_1 به V_4

$[V_1\ V_5\ V_4]=2$



کوتاهترین مسیر از V_1 به V_3

$[V_1\ \ V_3]=4$



کوتاهترین مسیر از V_1 به V_2

$[V_1\ V_5\ V_4\ V_2]=5$

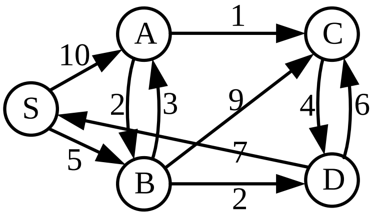
تمامی گره ها به مجموعه اضافه شده است.

پایان الگوریتم

روش دوم: با استفاده از یک جدول پیاده سازی می‌شود.

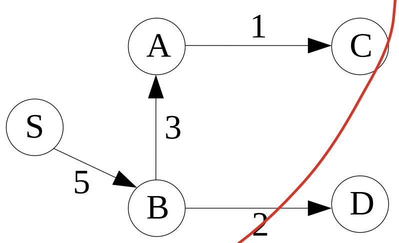
۱. گره‌ای را به عنوان گره مبدا انتخاب می‌کنیم و فاصله آن تا بقیه گره‌ها را به‌دست می‌آوریم و در جدول قرار می‌دهیم.
۲. با در نظر گرفتن جدول، گره‌ای که کم‌ترین فاصله تا گره مبدا را دارد اضافه نموده و فاصله گره مبدا تا بقیه‌ی گره‌ها را با استفاده از گره جدید عوض می‌کنیم.
۳. مرحله دوم را تا افزودن تمام گره‌ها به مجموعه تکرار می‌کنیم.

مثال: الگوریتم دیکسترا را با شروع از رأس S روی گراف زیر اعمال کنید.



حل:

	S	A	B	C	D
{S}		10 (S , A)	5 (S , B)*	∞	∞
{S , B}		8 (S , B , A)		14 (S , B , C)	7 (S , B , D)*
{S , B , D}		8 (S , B , A)*		13 (S , B , D , C)	
{S , B , D , A}				9 (S , B , A , C)	
{S , B , D , A , C}					



۴-۵ کدگذاری هافمن

فرض کنید حروف a, b, c ,d به صورت زیر کدگذاری شده باشند و بخواهیم عبارت حاصل از رشته زیر را بدست آوریم. در این‌صورت طبق آن‌چه شکل نشان می‌دهد، دچار ابهام می‌شویم. کد پیشوندی باعث ابهام می‌شود زیرا یک کد، پیشوند کد دیگر است. روش کدگذاری هافمن این ابهام را برطرف می‌کند.

a=10
b=11
c=1110
d=101

$\overbrace{\underbrace{11}_{b}\underbrace{10}_{a}}^{c}10111011...$

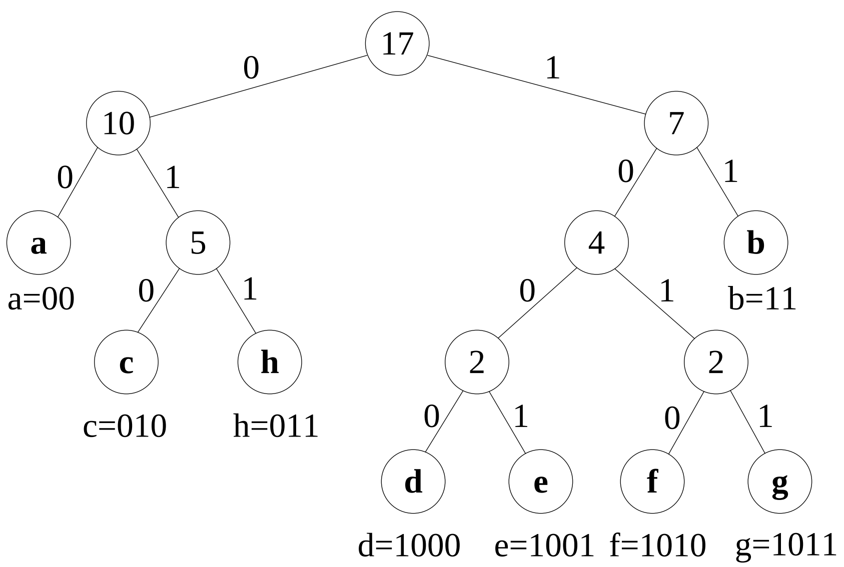
روش کدگذاری هافمن علاوه بر برطرف کردن ابهام کد پیشوندی باعث فشرده سازی متن نیز می‌شود. در روش فشرده سازی متن، ابتدا متن آنالیز شده و تعداد تکرار هر کاراکتر محاسبه می‌شود، سپس به کاراکترهایی که بیشترین تعداد تکرار را دارند کوچکترین کدها تعلق می‌گیرد تا فضای کمتری از حافظه اشغال شود.

مثال: درصد فشرده سازی عبارت زیر در روش کدگذاری هافمن چند است؟

a b a a b c c d c a b e f g h a h

ابتدا جدولی مانند جدول زیر تشکیل می‌دهیم و تعداد تکرار عناصر را مقابل هر کاراکتر می‌نویسیم. سپس از کمترین عددها شروع می‌کنیم و دو به دو آن‌ها را مطابق زیر جمع می‌کنیم و یک درخت می‌سازیم. یک قانون تعریف می‌کنیم و آن را تا آخر ادامه می‌دهیم. در این مثال قانون این است که فرزندان راست با شماره 1 و فرزندان چپ با شماره 0 نام‌گذاری شوند.

کاراکتر	a	b	c	d	e	f	g	h
تعداد تکرار هر کاراکتر	5	3	3	1	1	1	1	2



2*5+2*3+3*3+4*1+4*1+4*1+4*1+3*2=47 bit → در روش کدگذاری هافمن → 48 bit

عدد حاصل از روش کدگذاری هافمن را به اولین عدد بخش‌پذیر بر 8 تبدیل می‌کنیم تا بتوانیم آن را ذخیره کنیم.

17*8=136 bit → بدون روش کدگذاری هافمن

هر کاراکتر کد اسکی برای ذخیره شدن به 8 بیت نیاز دارد.

درصد فشرده‌سازی → $\frac{48}{136} * 100 = 35.29\%$

۵-۵ مسائل زمان‌بندی

۵-۵-۱ حالت ساده

یک سرویس دهنده و n تا کار (job) داریم، همه‌ی کارها در زمان صفر موجود هستند و زمان مورد نیاز برای هر کار هم داده شده است.

$Waiting\ Time = W_i$ → زمان انتظار

$Service\ Time = S_i$ → زمان سرویس

$Response\ Time\ (RT) = W_i + S_i$ → زمان پاسخ

کارها را به چه ترتیب زمان‌بندی کنیم که زمان پاسخ کمینه شود؟

الگوریتم حریصانه: کارها را از زمان سرویس کم به زمان سرویس زیاد مرتب می‌کنیم (صعودی) و به همین ترتیب سرویس می‌دهیم، در این صورت زمان پاسخ حداقل می‌شود.

مثال: برای جدول زیر زمان سرویس بهینه را محاسبه کنید.

i	S_i
1	7
2	1
3	4
4	3

حل:

۱) کارها را به ترتیب افزایش زمان سرویس مرتب کرده و زمان سرویس را محاسبه می‌کنیم.

$$\underbrace{(0+1)}_{j_2} + \underbrace{(1+3)}_{j_4} + \underbrace{(4+4)}_{j_3} + \underbrace{(8+7)}_{j_1} = 28 \quad \rightarrow \quad \min(RT)$$ پاسخ بهینه

۲) زمان سرویس را به ترتیب کارهای موجود در جدول محاسبه می‌کنیم.

$$\underbrace{(0+7)}_{j_1} + \underbrace{(7+1)}_{j_2} + \underbrace{(8+4)}_{j_3} + \underbrace{(12+3)}_{j_4} = 42$$

۳) کارها را به ترتیب کاهش زمان سرویس مرتب کرده و زمان سرویس را محاسبه می‌کنیم.

$$\underbrace{(0+7)}_{j_1} + \underbrace{(7+4)}_{j_3} + \underbrace{(11+3)}_{j_4} + \underbrace{(14+1)}_{j_2} = 47$$

مرتبه زمانی این کار $\theta(n \log n)$ است، زیرا فقط مرتب سازی صورت گرفته است.

۲-۵-۵ زمان‌بندی کارها با جریمه‌ی تاخیر

n تا کار داریم که زمان اجرای هر کدام ۱ واحد زمانی است. d_i (dead line) مهلت انجام کار i ام است و اگر کار بعد از d_i انجام شود جریمه‌ای معادل w_i به آن تعلق می‌گیرد. میزان تاخیر تاثیری در کل جریمه ندارد.

کارها را به چه ترتیب زمان‌بندی کنیم که مجموع جریمه‌ها کمینه شود؟

الگوریتم حریصانه: کارها را از زمان جریمه زیاد به زمان جریمه کم (نزولی) مرتب می‌کنیم و به همین ترتیب سرویس می‌دهیم، در این صورت جریمه حداقل می‌شود.

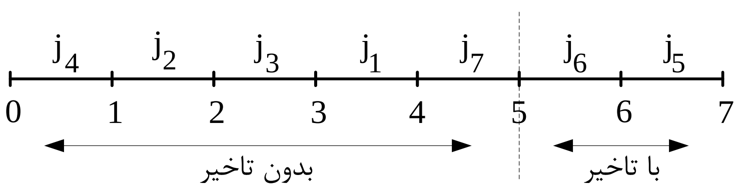
فرض می‌کنیم کار j_i را انتخاب کرده‌ایم، سپس از بازه $[d_i-1 , d_i]$ شروع می‌کنیم و بازه‌های سمت چپ را از راست به چپ بررسی می‌کنیم و j_i را در سمت راست‌ترین بازه‌ی خالی قرار می‌دهیم. اگر بازه‌ی خالی قبل از d_i پیدا نکنیم این کار با تاخیر انجام می‌شود و بعد از تعیین کارهای بدون تاخیر زمان‌بندی می‌شود.

مثال: برای جدول زیر جریمه کمینه را محاسبه کنید.

کار	1	2	3	4	5	6	7
مهلت d_i	4	2	4	3	1	4	6
جریمه W_i	70	60	50	40	30	20	10

حل:

کارها را بر اساس جریمه مرتب کرده‌ایم.



جریمه = 30 + 20 = 50

❖ مرتبه زمانی این کار $O(n^2)$ است، زیرا داریم:

$$\underbrace{\theta(n\log n)}_{\text{مرتب سازی}} + \underbrace{\theta(n^2)}_{\text{جستجو}} = O(n^2)$$

۵-۶ انتخاب فعالیت (Activity Selection)

n تا کار داریم که همگی می‌خواهند از یک منبع غیر قابل اشتراک استفاده کنند، یعنی در هر لحظه فقط یک کار می‌تواند انجام بگیرد. هر فعالیت زمان شروع Si و زمان پایان Fi دارد.

فعالیت‌ها را به چه ترتیبی انجام دهیم که بیش‌ترین تعداد فعالیت انجام شود؟

راه غیر هوشمندانه: تمام جایگشت‌ها را محاسبه کرده و تعداد بیشینه را به‌دست آوریم. در این صورت مرتبه زمانی $O(n!)$ می‌شود.

الگوریتم حریصانه: فعالیت‌ها را بر اساس زمان پایان و به‌صورت صعودی مرتب کرده و پس از انتخاب فعالیت آن‌هایی را که در تضاد هستند حذف می‌کنیم.

مثال: برای جدول زیر بیش‌ترین تعداد فعالیت را محاسبه کنید.

F_i	S_i	J_i
4	1	1
5	3	2
6	0	3
7	5	4
8	3	5
9	5	6
10	6	7
11	8	8
12	8	9
13	2	10
14	12	11

حل:

F_i	S_i	J_i	
4	1	1	انتخاب
5	~~3~~	2	
6	~~0~~	3	
7	5	4	انتخاب
8	~~3~~	5	
9	~~5~~	6	
10	~~6~~	7	
11	8	8	انتخاب
12	~~8~~	9	
13	~~2~~	10	
14	12	11	انتخاب

مرتب شده اند

مرتبه زمانی این کار $\theta(n\log n)$ است که فقط صرف مرتب سازی می‌شود.

۵-۷ مسئله کوله پشتی

nتا کالا با ارزش‌های متفاوتی داریم. یک کوله پشتی به اندازه W را می‌خواهیم با این بارها پر کنیم.

۵-۷-۱ کوله پشتی صفر و یک

در این الگوریتم کالاها قابل تقسیم نیستند، بنابراین یا انتخاب می‌شوند و یا نمی‌شوند. هم‌چنین کوله پشتی ظرفیت محدودی دارد یعنی وزن خاص و حجم خاصی را می‌تواند تحمل کند.

مجموعه کالاها $S=\{s_1,s_2,s_3,\cdots,s_n\}$

اعداد صحیح
$$\begin{cases} w_i & \text{وزن یا حجم کالای } i \text{ ام} \\ P_i & \text{ارزش کالای } i \text{ ام} \\ W & \text{حداکثر گنجایش کوله پشتی} \end{cases}$$

هدف این است که زیر مجموعه‌ای از S را انتخاب کنیم که کوچک‌تر یا برابر W باشد و دارای بیش‌ترین ارزش باشد.

راه حل غیرهوشمندانه: تمام حالت‌ها را بررسی کنیم (تمام زیر مجموعه‌های S) و با مقایسه کردن آن‌ها بهترین حالت را انتخاب کنیم. که در این صورت مرتبه زمانی $O(2^n)$ می‌شود، زیرا 2^n زیرمجموعه داریم.

راه حل حریصانه: کالاها را بر اساس ارزش مرتب کنیم و سپس به ترتیب کالاهای با بیش‌ترین ارزش را انتخاب کنیم که این روش لزوماً بهینه نیست.

مثال: سه کالا با شماره‌های ۱ تا ۳ داریم که مقادیر وزن و ارزش آن‌ها در جدول زیر داده شده است. می‌خواهیم کوله پشتی را با این کالاها پر کنیم ولی حداکثر گنجایش کوله پشتی ما W=30 است. کالاها را چگونه انتخاب کنیم که دارای بیش‌ترین ارزش بوده و از حداکثر گنجایش کوله پشتی استفاده کند؟

S_i	1	2	3
w_i	25	10	10
P_i	10	9	9

	وزن	ارزش
روش حریصانه ←	25	10
حالت بهینه ←	10+10	18

راه حل دیگر: کالاها را بر اساس نسبت ارزش به وزن $\frac{P_i}{w_i}$ مرتب کنیم و سپس به ترتیب کالاهای با بیش‌ترین مقدار را انتخاب کنیم که این روش نیز لزوماً بهینه نیست.

مثال: برای جدول زیر نیز مانند مثال بالا کالاها را به‌گونه‌ای انتخاب کنید که دارای بیش‌ترین ارزش بوده و از حداکثر گنجایش کوله پشتی استفاده کند؟ فرض: W=30

S_i	1	2	3
w_i	5	10	20
P_i	50	60	140
P_i/w_i	10	6	7

		نسبت ارزش به وزن	ارزش
روش حریصانه	→	10+7	50+140=190
حالت بهینه	→	7+6	140+60=200

مسئله کوله پشتی صفر و یک جز مسائل NPC است.

۵-۷-۲ کوله پشتی کسری

در این الگوریتم کالاها قابل تقسیم هستند، بنابراین الگوریتم حریصانه و بهینه است.

مثال: برای جدول مثال بالا کالاها را به‌گونه‌ای انتخاب کنید که دارای بیش‌ترین ارزش بوده و از حداکثر گنجایش کوله پشتی استفاده کند. فرض: W=30

S_i	1	2	3
w_i	5	10	20
P_i	50	60	140
P_i/w_i	10	6	7

		نسبت ارزش به وزن	ارزش
روش حریصانه و بهینه	→	$S_1 + S_3 + \frac{1}{2} S_2$	$50 + 140 + \frac{1}{2} \times 60 = 220$

مرتبه زمانی اجرای این الگوریتم $\theta(n \log n)$ است، زیرا کالاها را بر اساس $\frac{P_i}{w_i}$ مرتب کردیم.

یک راهبرد حریصانه واضح این است که کالاهای با بیش‌ترین ارزش زودتر از همه برداشته شوند، یعنی آن‌ها را به ترتیب افزایش ارزش مرتب کرده و انتخاب کنیم.

مسائل

۱) n تا بازه داریم می‌خواهیم مجموعه بازه‌های دو به دو ناهمپوشان با بیشترین طول را بیابیم.

$$\text{بازه} = n: [L_i, U_i]$$
$$i = 1, \cdots, n$$
$$\text{طول بازه} = U_i - L_i$$

۲) n شیء داده شده‌اند که حجم شیء i برابر V_i است. V_iها اعداد حقیقی بین صفر و یک هستند. می‌خواهیم این اشیا را در صندوق‌هایی که حجم هر کدام آن ها ۱ است بسته‌بندی کنیم. به طوری که تعداد صندوق‌ها حداقل شود.

۳) یک گراف کامل با ۱۰ رأس داریم که به ترتیب ۱ تا ۱۰ شماره‌گذاری شده است. اگر وزن هر یال از تابع زیر بدست آید، مجموع وزن یا ل های MST چقدر است؟

$$W_{i.j} = W_{j,i} = \begin{cases} i+j & i+j \geq 5 \\ i^2 + j^2 & i+j < 5 \end{cases}$$

۴) حداقل تعداد ضرب :

$$A_{13\times5} \times B_{5\times89} \times C_{89\times3} \times D_{3\times34}$$

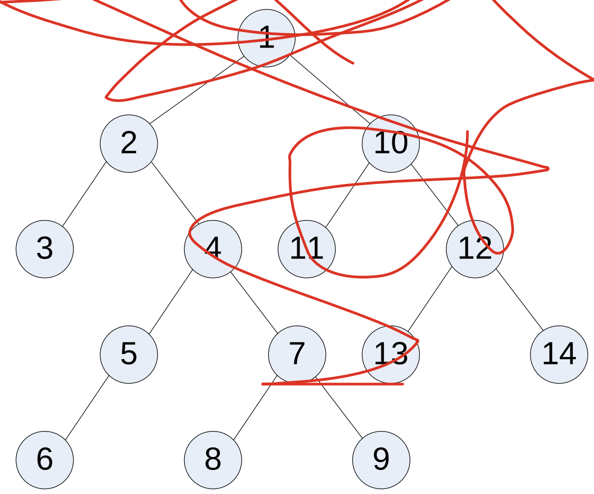
فصل ششم: پیمایش گراف‌ها

۱-۶ پیمایش عمقی DFS (Depth First Search)

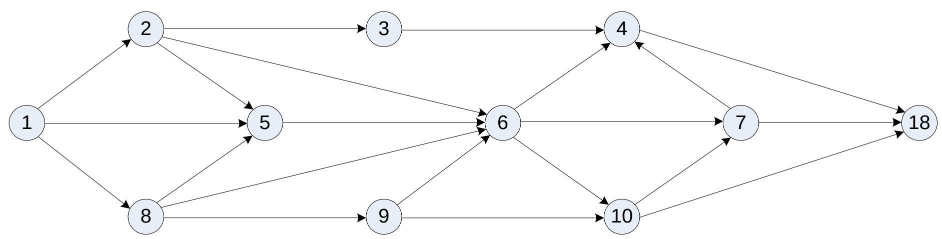
گراف : ۱. جهت‌دار

۲. بدون جهت

اولویت گره (ترتیب ملاقات : بر حسب شماره، حروف الفبا، ...) در DFS گره آغازین پیمایش باید مشخص باشد.



پیمایش را از رأس آغازین شروع می‌کنیم و آن را داخل Stack قرار می‌دهیم. رأس ملاقات شده از پشته خارج می‌شود و همسایگان ملاقات شده‌اش به صورت معکوس وارد پشته می‌شود. اگر رأسی ملاقات شده بود وارد پشته نمی‌شود و تا زمانی این را ادامه می‌دهیم که پشته خالی شود و الگوریتم پایان پذیرد.



اولویت با اعداد است.

DFS: 1 , 2 , 3 , 4 , 11 , 5 , 6 , 7 , 10 , 8 , 9

پشته :

۱	8	5	2	6	5	3	4	11	10	7	4	9

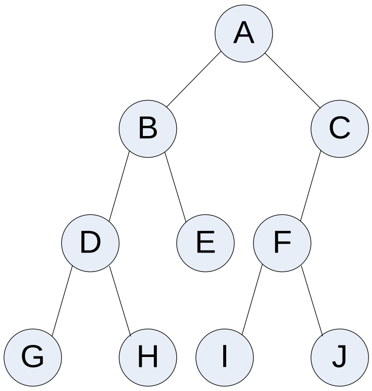
$$E \to n^2 \to O(n^2)$$

$$\to n \to O(n)$$

$$\boxed{O(V+E)}$$

۶-۲ پیمایش سطحی BFS (Breadth First Search)

گراف : ۱. جهت‌دار
۲. بدون جهت



BFS : A, B, C, D, E, F, G, H, I, J

۶-۲-۱ پیمایش BFS

هر گره به این صورت که گره آغازین مشخص باشد، هر گره درون صف قرار می‌گیرد و پس از ملاقاتش، همسایگانش در انتهای صف به ترتیب درج می‌شود تا زمانی که صف خالی شود.

صف مثال قبل :

۱	2	5	8	3	6	9	4	7	10	11	

Front **Rear**

یکی از کاربردهای پیمایش، تعیین همبند بودن گراف است.

در صورت همبند بودن در هر کدام از پیمایش‌ها باید تمامی گره‌ها پیمایش شوند، در غیر این صورت گراف نیمبند است. (در گراف غیر جهت‌دار، معمولاً گراف همبند است)

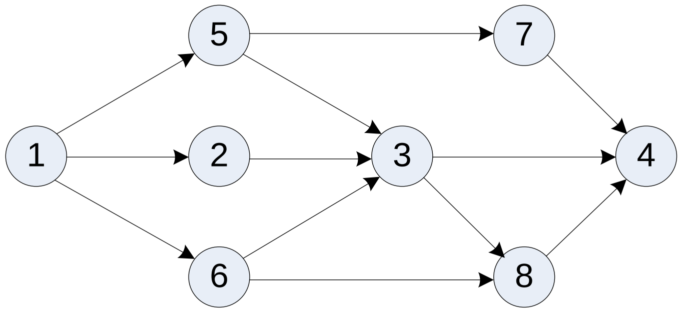
در گراف جهت‌دار دسترس‌پذیر بودن تمامی گره‌ها مدنظر است.

در یک گراف بدون جهت، همبند بودن یا نبودن با پیمایش BFS، DFS معلوم می‌شود. یعنی اگر تمام گره‌ها در دسترس باشند،گراف همبند است.

۶-۲-۲ مرتب سازی توپولوژیک ، ترتیب توپولوژیک (Topological sort)

یکی از کاربردهای پیمایش گراف است، که عبارت است از مرتب‌سازی خطی تمام رئوس گراف به طوری که اگر G (گراف) شامل یال (U,V) باشد آنگاه U قبل از V ظاهر شود.

این نوع مرتب‌سازی فقط برای یک گراف جهت‌دار بدون دور تعریف شده است.



زمان پایان = زمانی که دیگر به این گره بر نمی‌گردیم.

اولویت با رعایت ضوابط می‌تواند عوض شود :

1 → 6 → 5 → 7 → 2 → 3 → 8 → 4

۱. DFS را بر روی گراف اعمال می‌کنیم تا زمان پایان هر رأس انجام می شود. پس از محاسبه زمان پایان هر رأس آن را در ابتدای یک لیست درج می‌کنیم.
۲. لیست حاصل ترتیب توپولوژیک گراف است.

DFS: 1 , 2 , 3 , 4 , 8 , 5 , 7 , 6

در بخش های قبل روش تقسیم و حل و پویا و صریحانه که عمدتاً برای به دست آوردن راه حل‌های سریع و چندجمله‌ای برای مسائل استفاده می‌شوند مورد بررسی قرار گرفت ولی پیدا کردن چنین الگوریتم‌هایی کار ساده‌ای نیست و تمام مسائل را با این روش‌ها نمی‌توان حل کرد.

اگر مسئله‌ای به روش‌های فوق حل نشود ممکن است تنها راه حل آن جستجوی فضای حالت باشد. یعنی برای پیداکردن جواب، کلیه حالات را بررسی می‌کنیم. که به روش‌های زیر انجام می‌شود:

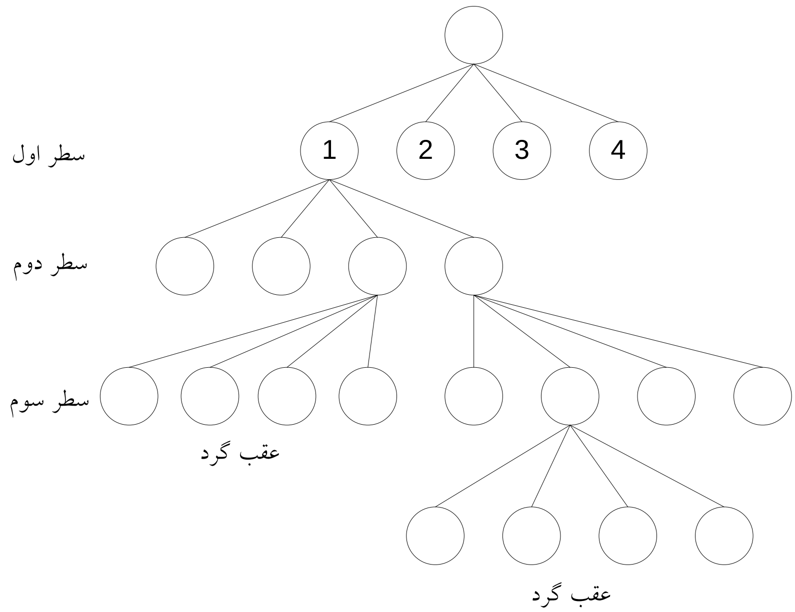
۶-۲-۲-۱ روش عقب‌گرد یا پس‌گرد (Back Tracking)

در این قسمت کلیه فضای حالت قابل قبول را با نظم خاصی ایجاد و جستجو می‌کنیم به این صورت که در هر مرحله ممکن است با چند انتخاب روبرو شویم، که یکی از انتخاب‌ها را با فرض درست بودن انتخاب می‌کنیم اگر به جواب نرسیدیم با عقب‌گرد، انتخاب خود را عوض می‌کنیم و این کار را تا تمام شدن انتخاب‌ها انجام می‌دهیم. این روش می‌تواند با یافتن اولین جواب متوقف شود و یا اینکه جستجو برای یافتن کلیه جواب‌ها ادامه یابد.

الگوریتم‌های عقب گرد معمولاً دارای هزینه‌های نمایی هستند.

مثال: چهار وزیر در صفحه n×n هستند که حرکت‌های آنها فقط به صورت افقی یا عمودی یا ضربدری است. می‌خواهیم این وزیرها را طوری بچینیم که همدیگر را تحدید نکنند؟

	۱	۲	۳	۴
۱		x	O	
۲	O			x
۳	x			O
۴		O	X	



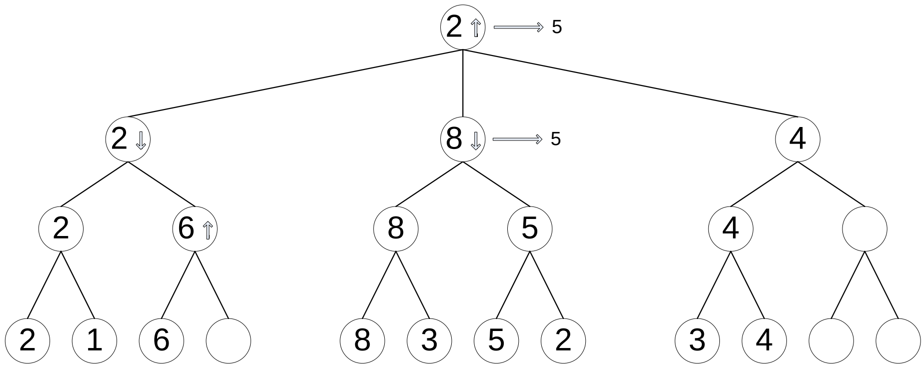
۲-۲-۲-۶ درخت‌بازی
یکی از روش‌های جستجوی فضای مسئله می‌باشد که جهت تعیین استراتژی برد و انجام بهترین بازی ممکن در هر مرحله می‌باشد. هر گره درخت یک وضعیت از درخت است و وضعیت‌هایی که با حرکت بعد قابل تولید هستند به عنوان فرزندان گره تعیین می‌شوند.

برگ‌ها، خاتمه بازی هستند و هر مسیر تا رسیدن به یک برگ بیانگر یک بازی است که یک یال این بازی توسط بازیکن اول و یال بعد توسط بازیکن دوم انتخاب می‌شود و به هر گره عددی نسبت داده می‌شود که بستگی به ظرفیت بازی دارد.

۳-۲-۲-۶ محدود کردن فضای جستجو
در مسائل فوق سعی بر این بود که تمامی فضای حالت مورد بررسی قرار بگیرد اما با استفاده از روش‌های زیر می‌توان فضای جستجو را کاهش داد.

۱) هرس کردن ۲) انشعاب یا محدوده (Branch & Bound)

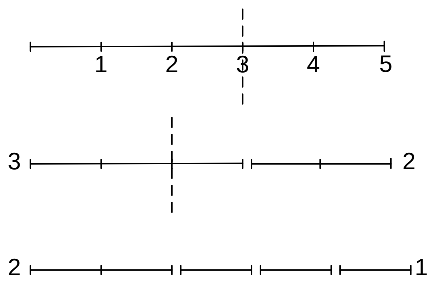
روش هرس کردن:



گاهی اوقات جستجوی برخی از شاخه‌ها بی‌تأثیر است و می‌توان از جستجوی آن صرف نظر کرد. در شکل مورد نظر امتیاز ۶ باید افزایش پیدا کند. در صورتی که در گره پدر آن امتیاز ۲ باید کاهش پیدا کند. با ادامه شاخه ۶ مقدار ما از ۶ کمتر نخواهد شد پس چون در سطح قبل تصمیم داریم امتیاز ۲ را کاهش دهیم پس ادامه دادن شاخه فرزند سمت راست ۶ بی‌تأثیر است و هرس می شود.

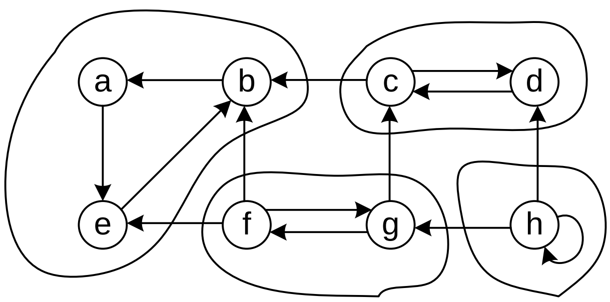
مثال: یک چوب به طول L، می‌خواهیم از نقاط $X_1,X_2,...,X_n$ ببریم. که $X_1<X_2<...<X_n$ ، هزینه برش قطعه‌ای به طول k ، k می‌باشد. چوب را به چه صورتی برش بزنیم که هزینه برش کمترین باشد؟

به روش DF حل می‌شود.

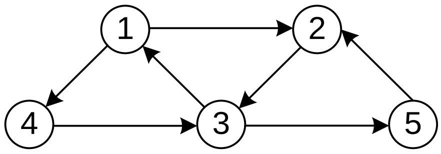


۳-۲-۶ اجزاء قویاً همبند scc

$$uRv \Leftrightarrow v \to u, u \to v$$



۴ جزء قویا همبند دارد.



کل این شکل یک جزء قویا همبند دارد.

* بزرگترین را انتخاب می‌کنیم تا جایی که بتوانیم کل را در نظر می‌گیریم.