دانشگاه اراک

دانشکده فنی و مهندسی

عنوان

جزوه درس طراحی الگوریتم

استاد راهنما

دکتر سیّد حمید حاج سیّد جوادی

آذر ماه ۱۳۸۴

بسم اللّه الرّحمن الرّحیم

فهرست مطالب

فصل ۳ یادآوری برخی از ساختمان داده ها

فصل ۵ پویش گراف ها

فصل ٦ نگاهی مختصر بر ارائه دو سمینار

مقدمه

جزوه‌ای که هم اکنون در اختیار شما قرار گرفته است ، جزوه‌ی درس طراحی الگوریتم می باشد که توسط جمعی از دانشجویان رشته مهندسی کامپیوتر دانشکده فنی و مهندسی دانشگاه اراک در آذر ماه سال ۱۳۸۴ توسط نرم افزار فارسی‌تک تهیه وتدوین گردیده است و در اختیار علاقه مندان گذاشته شده است .

دراین قسمت جا دارد از تمامی عزیزانی که در تهیه‌ی این جزوه همکاری نموده‌اند ، دانشجویان رشته مهندسی کامپیوتر ورودی سال ۱۳۸۲ دانشکده فنی و مهندسی دانشگاه اراک ، به خصوص از زحمات بی دریغ خانم ها فاطمه رمضانی و زهرا احمدی به خاطر ویرایش آن کمال تشکر را داشته باشم .

حمید حاج سیّد جوادی
دانشکده فنی و مهندسی
دانشگاه اراک
آذر ماه ۱۳۸۴

فصل ۱

معرفی نمادهای مجانبی

۱. نمادهای مجانبی

ابزارهائی که توسط آنها می توان زمان اجراو یا حافظه گرفته شده دو یا چند الگوریتم را با هم مقایسه نماییم .

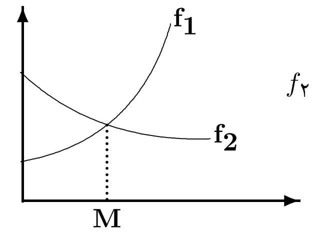
۱.۱ نماد big-O

اگرR^{+} $f: N \longrightarrow$ باشد آنگاه:

$$O(f(n)) = \{g{:}N \rightarrow R^{+} \,|\exists c \in R^{+} \,, \overset{\infty}{\forall}\, n \in N \;\; g(n) \leq cf(n)\}$$

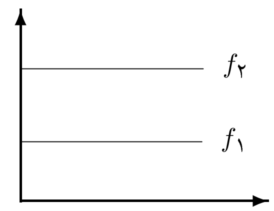
$$f(n) = \begin{cases} n^{۳} & \text{n<1000} \\ ۲n^{۲} & \text{n}\geq\text{100} \end{cases}$$

$$O(n^{۲}) = \{n^{۲}, n\lg n, f(n), ...\}$$



$f_{۲} \in O(f_{۱})$

در شکل قبل حتی اگر $f_{۲}$ صد برابر شود باز هم از $O(f_{۱})$ است .



$f_۱ \in O(f_۲)$

$f_۲ \in O(f_۱)$

در شکل بالا اگر $f_۱$ در یک ضریب ضرب شود داریم $f_۲ \in O(f_۱)$ ،پس چیزی که مهم است ضریب c نیست ، مهم درجه‌است .
big-O همان مفهوم کران بالا را دارد مثلاً برای حافظه مصرفی, big-O کران بالای مصرف حافظه است. پس الگوریتمی مفید است که کران بالایی زمان اجرای آن پایین باشد.

۲.۱ نماد $big-\Omega$

اگر $f : N \longrightarrow R^+$ آنگاه:

$$\Omega(f(n)) = \{g : N \rightarrow R^+ \,|\exists d > ۰ \quad \overset{\infty}{\forall} n \in N \quad f(n) \leq dg(n)\}$$

در اینجا کوچکتر مساوی یا کوچکتر تفاوتی ندارند زیرا ضرایب را می توانیم تغییر دهیم:

$f(n) < dg(n) \Longrightarrow f(n) \leq dg(n)$

$f(n) \leq dg(n) \Longrightarrow f(n) < d'g(n)$

همچنین داریم :

$$f_۲(n) \in \Omega(f_۱(n)) \Longleftrightarrow f_۱(n) \in O(f_۲(n))$$

۳.۱ نماد θ

اگر $f : N \longrightarrow R^+$ آنگاه‌داریم : $\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

مثال :

$O(n^۲) = \{n^۲, n^۲ + ۴, nlogn, ...\}$

$\Omega(n^۲) = \{۲n^۲, ۲n^۲ + \sqrt{n}, n^۳ + n^۲, n^۴, n^۲logn, ...\}$

$\theta(n^۲) = \{n^۲, n^۲ + n, ۲n^۲, ...\}$

نکته :
برای تشخیص بدی الگوریتم باید از Ω استفاده نمود،در واقع باید کران پایین بالایی برای آن به دست آوریم .

۴.۱ نماد $small-o$

اگر $f:N\longrightarrow R^{+}$ آنگاه:

$$o(f(n))=\{g:N\rightarrow R^{+}\,|\forall\, c>0\quad \overset{\infty}{\forall} n\in N\quad g(n)\leq cf(n)\}$$

در اینجا کوچکتر مساوی یا کوچکتر تفاوتی ندارندتنها اهمیت در آن است که به ازای هر ضریب c صدق کند.

مثال :

کدام یک از عبارت های فوق درست است :$2n\in o(n)$ یا$n\in o(2n)$ ؟

$$2n\in o(n)\Longleftrightarrow \forall c>0\quad \overset{\infty}{\forall} n\in N\quad 2n\leq cn\overset{c=1}{\Longleftrightarrow} F\Longrightarrow 2n\notin o(n)$$

$$n\in o(2n)\Longleftrightarrow \forall c>0\quad \overset{\infty}{\forall} n\in N\quad n\leq c2n\overset{c=\frac{1}{2}}{\Longleftrightarrow} F\Longrightarrow n\notin o(2n)$$

۵.۱ نماد $small-\omega$

اگر $f:N\longrightarrow R^{+}$ آنگاه:

$$\omega(f(n))=\{g:N\rightarrow R^{+}\,|\forall\, c>0\quad \overset{\infty}{\forall} n\in N\quad g(n)\geq cf(n)\}$$

مثال: نشان دهید :

$$\omega(f(n))\bigcap o(f(n))=\emptyset$$

حل: اگر $g(n)\in\omega(f(n))\bigcap o(f(n))$ آنگاه برای $d_0>0$ یک M_1چنان موجود است که :

$$\forall n\geq M_1\quad g(n)\geq d_0 f(n)\quad (g(n)\in\omega(f(n)))$$

حال برای $c_0=\frac{d_0}{2}$ یک M_2 چنان موجود است که:

$$\forall n\geq M_2\quad g(n)\leq c_0 f(n)\quad (g(n)\in o(f(n)))\Longrightarrow g(n)\leq\frac{d_0}{2}f(n)\Longrightarrow$$

$$\forall n\geq Max\{M_1,M_2\}\quad \frac{d_0}{2}f(n)\geq d_0 f(n)\Longrightarrow d_0\geq 2d_0\Longrightarrow d_0\leq 0$$

پس به تناقض رسیدیم ،بنابر این حکم ثابت می شود.

۶.۱ قضیه ماکزیمم گیری

$$1)f(n)+g(n)\in O(MAX\{f(n),g(n)\})$$
$$2)f(n)+g(n)\in\Theta(MAX\{f(n),g(n)\})$$

اثبات ۱ :

$$f(n)\leq MAX\{f(n),g(n)\}\quad,\quad g(n)\leq MAX\{f(n),g(n)\}\Longrightarrow$$
$$f(n)+g(n)\leq 2MAX\{f(n),g(n)\}\Longrightarrow f(n)+g(n)\in O(MAX\{f(n),g(n)\})$$

اثبات ۲ :

$$MAX\{f(n),g(n)\} \leq f(n)+g(n) \Longrightarrow f(n)+g(n) \in \Omega(MAX\{f(n),g(n)\})$$
$$f(n)+g(n) \in O(MAX\{f(n),g(n)\}), f(n)+g(n) \in \Omega(MAX\{f(n),g(n)\})$$
$$\Longrightarrow f(n)+g(n) \in \Theta(MAX\{f(n),g(n)\})$$

تمرین:

نشان دهید $\log n! \in \theta(n\log n)$.

حل:

راه اول :

اثبات با استفاده از فرمول استرلینگ $(n! = (\frac{n}{e})^n\sqrt{۲\pi n})$

$$n! = (\frac{n}{e})^n\sqrt{۲\pi n} \Longrightarrow \log n! = log(\frac{n}{e})^n + \log\sqrt{۲\pi n}$$
$$\Longrightarrow \log n! = n\log\frac{n}{e} + \log\sqrt{۲\pi n}$$
$$\Longrightarrow \log n! = n\log n - \log e^n + \log\sqrt{۲\pi n}$$
$$\Longrightarrow \log n! = n\log n + \log\frac{\sqrt{۲\pi n}}{e^n}$$
$$\Longrightarrow \log n! \in \Theta(n\log n)$$

راه دوم :

در اینجا باید اثبات نماییم $\log n! \in O(n\log n)$ و $\log n! \in \Omega(n \log n)$.

$$logn! = logn + log(n-۱) + \ldots + log۱ \leq \underbrace{logn + logn + \ldots + logn}_{n} \leq nlogn$$
$$\Longrightarrow logn! \in O(nlogn)$$
$$\log n! = \log n + \log(n-۱) + \ldots + \log ۱ \geq \frac{n}{۲}\log\frac{n}{۲} \Longrightarrow$$
$$\log n! \geq \frac{n}{۲}\log n - \frac{n}{۲}\log ۲ \geq \frac{۱}{۴}n\log n \Longrightarrow logn! \in \Omega(nlogn)$$
$$\Longrightarrow \log n! \in \Theta(n\log n)$$

راه سوم :

به تساوی زیر دقت فرمایید :

$$(n!)^۲ = (۱\times۲\times\ldots\times(n-۱)n)^۲ \Rightarrow$$
$$(n!)^۲ = (۱\times۲\times\ldots\times(n-۱)n)(n(n-۱)\times\ldots\times۱) = \prod_{x=۱}^{n} x(n-x+۱)$$
$$\begin{cases} y = -x^۲ + (n+۱)x \\ x = \frac{n+۱}{۲} \Longrightarrow y_{max} = \frac{(n+۱)^۲}{۴} \\ x = ۱ \Rightarrow y = n, x = n \Rightarrow y = n \Longrightarrow y_{min} = n \end{cases}$$

$$\prod_{x=1}^{n} n \leq (n!)^2 \leq \prod_{x=1}^{n} \frac{(n+1)^2}{4} \Longrightarrow n^n \leq (n!)^2 \leq (\frac{n+1}{2})^{2n} \Longrightarrow$$

$$nlogn \leq 2logn! \leq 2nlog\frac{n+1}{2} \leq 2nlogn \Longrightarrow \frac{n}{2}logn \leq logn! \leq nlogn$$

$$\Longrightarrow \log n! \in \theta(n \log n)$$

۱.۷ اثبات چند قضیه

قضیه :

برای $o(f(n))$ داریم :

$o(f(n)) \subseteq O(f(n)) \backslash \Omega(f(n))$
$o(f(n)) \subseteq O(f(n)) \backslash \Theta(f(n))$

برهان قسمت اول:

فرض می نماییم $g(n) \in o(f(n))$ نشان می دهیم $g(n) \in O(f(n))$ ولی $g(n) \notin \Omega(f(n))$. ابتدا نشان می دهیم $g(n) \in O(f(n))$ است .

فرض می کنیم که $c = 5$ است آنگاه چون $g(n) \in o(f(n))$ است در این صورت برای یک M_1، $n \geq M_1$ داریم $g(n) \leq 5f(n)$ پس $g(n) \in O(f(n))$ است حال نشان می دهیم $g(n) \notin \Omega(f(n))$ ، با فرض اینکه $g(n) \in \Omega(f(n))$ باشد آنگاه برای یک $d = d_0$ داریم‌یک M_1 که برای هر $n \geq M_1$، $d_0 f(n) \leq g(n)$ و از طرفی $g(n) \in o(f(n))$ پس برای $c = \frac{d_0}{2}$ یک M_2 چنان وجود دارد که $n \geq M_2$ داریم $g(n) \leq \frac{d_0}{2} f(n)$ پس برای هر $n \geq MAX\{M_1, M_2\}$ داریم $d_0 \leq 0$ که تناقض است .

برهان قسمت دوم:$(A \backslash B = A \backslash (A \cap B))$

$o(f(n)) \subseteq O(f(n)) \backslash \Omega(f(n)) = O(f(n)) \backslash (O(f(n))) \cap \Omega(f(n))$
$\Longrightarrow o(f(n)) \subseteq O(f(n)) \backslash \theta(f(n))$

مثال : نشان دهید قضیه فوق در حالت کلی نمی تواندبه تساوی تبدیل شود.

حل:

با استفاده از مثال نقض این موضوع را نشان می دهیم ،فرض کنید تابع زیر را داشته باشیم :

$$g(n)=\begin{cases} 12 & \text{های فرد } n \\ n & \text{های زوج } n \end{cases}$$

داریم :

$$M = ۱۲ \quad , n \geq ۱۳ \Rightarrow g(n) \leq n \Longrightarrow g(n) \in O(n)$$

حال فرض می کنیم $g(n) \in \Omega(n)$ باشد ، پس داریم :

$$g(n) \in \Omega(n) \Leftrightarrow \exists c > ۰ \quad \overset{\infty}{\forall} n \quad g(n) \geq cn \Leftrightarrow \exists c > ۰ \quad \overset{\infty}{\forall} n \quad \frac{g(n)}{c} \geq n \Leftrightarrow F$$

(اعداد فرد از بالا کران دار نیستند) $\Longrightarrow g(n) \notin \Omega(n)$

حال ببینیم رابطه $g(n) \in o(n)$ برقرار است یا نه ،فرض کنید این رابطه برقرار باشد پس داریم :

$$g(n) \in o(n) \Longleftrightarrow \forall c > ۰ \quad \overset{\infty}{\forall} n \quad g(n) \leq cn, c = \frac{۱}{۲} \Longrightarrow g(n) \leq \frac{n}{۲} \Longleftrightarrow F$$

$$\Longrightarrow g(n) \notin o(n)$$

پس مشاهده می شود قضیه فوق در حالت کلی نمی تواندبه مساوی تبدیل شود.

تمرین : نشان دهیدO وΩ انعکاسی وتعدی هستند در حالی که تقارنی نمی باشند .

حل :

اثبات خاصیت انعکاسی :

$$f(n) \in O(f(n)) \Longleftrightarrow \exists c > ۰ \quad \overset{\infty}{\forall} \quad n \in N \quad f(n) \leq ۱ \times f(n)$$

اثبات خاصیت تعدی :

فرض کنید $g(n) \in O(f(n))$ و $f(n) \in O(h(n))$ باشد از آنجایی که $g(n) \in O(f(n))$ بنابراین:

برای یک $c_۱ \geq ۰$ و یک $M_۱ > ۰$ و هر $n \geq M_۱$ داریم: $g(n) \leq c_۱ f(n)$

وبه همین ترتیب برای $f(n) \in O(h(n))$ برای یک $c_۲ \geq ۰$ و یک $M_۲ > ۰$ و هر $n \geq M_۲$ داریم: $f(n) \leq c_۲ h(n)$

حال برای $M = MAX\{M_۱, M_۲\}$ و هر $n \geq M$ داریم :

$$\begin{cases} g(n) \leq c_۱ f(n) \\ f(n) \leq c_۲ h(n) \end{cases} \Longrightarrow g(n) \leq c_۱ f(n) \leq c_۱ c_۲ h(n) \Longrightarrow g(n) \leq ch(n)$$

$$\Longrightarrow g(n) \in O(h(n))$$

این دو خاصیت برای Ω نیز به همین ترتیب اثبات می شوند.

مثال نقض برای عدم داشتن خاصیت تقارنی :

$n \in O(n^2)$ در حالی که $n^2 \notin O(n)$ ، فرض می نماییم $n^2 \in O(n)$ در نتیجه باید یک $c > 0$ و یک M_1 و $n \geq M_1$ داریم:

$n^2 \leq cn \Rightarrow n \leq c \Rightarrow$ (اعداد طبیعی از بالا کراندارند.)

تمرین : نشان دهید θ روی مجموعه تمام توابع f یک رابطه هم ارزی است .

حل :

به خاطر داشتن خاصیت انعکاسی O, Ω داریم :

$$f(n) \in O(f(n)) \quad , f(n) \in \Omega(f(n)) \Rightarrow f(n) \in \theta(f(n))$$

پس θ دارای خاصیت انعکاسی می باشد.

برای بررسی خاصیت تقارنی نیز فرض می کنیم $g(n) \in \theta(f(n))$ پس داریم :

$$g(n) \in \theta(f(n)) \Longrightarrow g(n) \in O(f(n)), g(n) \in \Omega(f(n)) \Longrightarrow f(n) \in \Omega(g(n)),$$
$$f(n) \in O(g(n)) \Longrightarrow f(n) \in \Omega(g(n)) \cap Og(n)) \Longrightarrow f(n) \in \theta(g(n))$$

که نشان می دهد θ دارای خاصیت تقارنی نیزمی باشد.

همچنین به خاطر داشتن خاصیت تعدی O, Ω داریم :

$$f(n) \in \theta(g(n)) \quad , g(n) \in \theta(h(n)) \Longrightarrow$$
$$f(n) \in O(g(n)) \quad , g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$$
$$f(n) \in \Omega(g(n)) \quad , g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$$
$$\Longrightarrow f(n) \in \theta(h(n))$$

همان طور که مشاهده شد θ دارای هر سه خاصیت انعکاسی ،تقارنی و تعدی است پس روی مجموعه تمام توابع f یک رابطه هم ارزی است .

تمرین : اگر داشته باشیم :

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}, \qquad f, g : N \longrightarrow R^+$$

اثبات نمایید :

۱) چنانچه $L = 0$ آنگاه $f(n) \in o(g(n))$.

۲) چنانچه $L = \infty$ آنگاه $g(n) \in o(f(n))$.

۳) چنانچه $0 < L < \infty$ آنگاه $g(n) \in \theta(f(n))$.

نکته :

نماد تساوی در مورد θ درست بکار می رود زیرا یک کلاس هم ارزی است ولی در مورد O, o, Ω, ω درست نمی باشد.

مثال :

ادعاهای زیر را اثبات و یا رد کنید:

- $f(n) \in O(g(n)) \Longrightarrow 2^{f(n)} \in O(2^{g(n)})$

حل : بامثال نقض این ادعا را رد می نماییم

$$n \log_2^n \in O(\log_2^{n!}) \Longrightarrow 2^{n \log_2^n} \in O(2^{\log_2^{n!}}) \Longrightarrow n^n \in O(n!)$$

که تناقض است و یا مثال نقض دیگر:

$$2 \log_2^n \in O(\log_2^n) \Longrightarrow 2^{\log_2^{n^2}} \in O(2^{\log_2^n}) \Longrightarrow n^2 \in O(n)$$

- $f(n) \in O(g(n)) \Longrightarrow (f(n))^k \in O(g(n)^k)$

حل:

(در تحلیل الگوریتم ها f و g را بزرگتر از یک در نظر می گیریم بنابراین می توانیم به توان برسانیم‌ولی اگر از نظر زمانی بزرگتر از یک بگیریم درست نیست .)

$$f(n) \leq cg(n) \Longrightarrow (f(n))^k \leq c^k(g(n)^k) \Longrightarrow (f(n))^k \in O(g(n)^k)$$

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

حل:

$$f(n) = \frac{1}{n} \Longrightarrow f^2(n) = \frac{1}{n^2} \Longrightarrow f(n) \notin O(f^2(n))$$

فصل ۲

الگوریتم های بازگشتی

۲. معادلات والگوریتم های بازگشتی

۱.۲ معادلات بازگشتی

منظور از یک معادله‌ی بازگشتی معادله‌ای است که هر جمله آن وابسته به یک یا چند جمله‌ی ما قبل آن است و به علاوه برای یک یا چند مقدار اولیه از قبل تعریف شده است .

در این قسمت به حل معادلات بازگشتی یک متغیره با ضرایب حقیقی می‌پردازیم که در شکل عمومی زیر صدق می کنند:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = f(n) \ a_i \in R, f : N \longrightarrow R^{\geq 0}, k \in N$$

ضرایب $t_0, \cdots, t_{k-1}$ از قبل مشخص هستند،k ثابت است و فقط یک متغیر n داریم. به معادله بالا معادله بازگشتی با ضرایب ثابت گویند .

برای حل این معادلات معمولا از معادله مشخصه استفاده می کنیم. در ابتدا فرض می کنیم معادله بازگشتی از نوع همگن باشد یعنی $f(n) = 0$، بنا براین حل معادله بازگشتی با ضرایب ثابت زیر را داریم:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

برای حل این معادله معادله مشخصه را به صورت زیر می نویسیم.هر جمله t_i را به x^i تبدیل می کنیم. بنابراین:

$$a_0x^n+a_1x^{n-1}+\cdots+a_kx^{n-k}=0 \Rightarrow x^{n-k}\underbrace{(a_0x^k+a_1x^{k-1}+\cdots+a_k)}_{\substack{\Downarrow \\ \text{معادله مشخصه (مفسر)}}}=0$$

سپس ریشه های معادله‌ی مشخصه را مشخص می کنیم .

۱.۱.۲ بحث در مورد ریشه ها

۱)معادله دارای k ریشه حقیقی دو به دو متمایز $r_1,\ldots,r_k$ باشددر این صورت جواب معادله به فرم زیر است:

$$t_n=c_1(r_1)^n+c_2(r_2)^n+\ldots+c_k(r_k)^n$$

۲)معادله دارای ریشه‌های حقیقی ودر عین حال برخی تکراری باشد، برای مثال ریشه r_p تکراری از بستایی m باشد (یعنی عبارت دارای فاکتور$(x-r_p)^m$ است ولی فاقد فاکتور $(x-r_p)^{m+1}$است) در این صورت جواب معادله به فرم زیر است :

$$t_n=c_1(r_1)^n+c_2(r_2)^n+\ldots+cp_0(r_p)^n+cp_1n(r_p)^n+$$
$$\ldots+cp_{m-1}n^{m-1}(r_p)^n+\ldots+c_t(r_t)^n$$

۳)گر معادله دارای ریشه های موهومی و غیر حقیقی‌باشد، بااستفاده‌از قضیه دموآور دقیقاً شبیه آن چه که در مورد ریشه های حقیقی معادله مشخصه گفته شده می توان جواب را محاسبه کرد.

مثال :

معادلات بازگشتی زیر راحل کنید:

$$g(n)=\begin{cases} n & n\leq 1 \\ 5g(n-1)-6g(n-2) & else \end{cases}$$

حل :

$$g(n)-5g(n-1)+6g(n-2)=0 \Longrightarrow x^2-5x+6=0$$
$$\Rightarrow x_1=3, x_2=2 \Rightarrow g(n)=c_1 3^n+c_2 2^n$$
$$g(0)=0, g(1)=1 \Rightarrow c_1=1 \quad , c_2=-1 \Longrightarrow g(n)=3^n-2^n$$

$$t_n = \begin{cases} 2t_{n-1} - t_{n-2} \\ t_0 = 1 \\ t_1 = 3 \end{cases}$$

حل :

$$t_n - 2t_{n-1} + t_{n-2} = 0 \Longrightarrow x^2 - 2x + 1 = 0 \Rightarrow x_1 = x_2 = 1$$
$$\Rightarrow t_n = c_1 + c_2 n \quad t_0 = 1 \quad , t_1 = 3 \Rightarrow c_1 = 1, c_2 = 2 \Longrightarrow t_n = 1 + 2n$$

$$t_n = \begin{cases} 2t_{n-1} + 1 \\ t_0 = 0 \\ t_1 = 1 \end{cases}$$

حل :

$$t_n = 2t_{n-1} + 1 \quad , t_{n-1} = 2t_{n-2} + 1 \Rightarrow t_n - t_{n-1} = 2t_{n-1} - 2t_{n-2}$$
$$\Rightarrow t_n - 3t_{n-1} + 2t_{n-2} = 0 \Rightarrow x^2 - 3x + 2 = 0$$
$$\Rightarrow x_1 = 1, x_2 = 2 \quad , t_0 = 0, t_1 = 1 \Longrightarrow t_n = 2^n - 1$$

حل معادلات بازگشتی غیرهمگن باضرایب ثابت:

معادلات بازگشتی غیرهمگن باضرایب ثابت که در شکل عمومی زیر صدق می کنندرا در نظر بگیرید :

$$a_0 t_n + a_1 t_{n-1} + a_2 t_{n-2} + \ldots + a_k t_{n-k} =$$
$$b_1^n p_1(n) + b_2^n p_2(n) + \ldots + b_m^n p_m(n)$$

($p_i(n)$ چند جمله‌ای از درجه d_i هستند و a_i, b_j هااعداد حقیقی و ثابت هستند.)

برای این دسته از معادلات معادله مشخصه را به صورت زیر می نویسیم :

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \cdots (x - b_m)^{d_m+1} = 0$$

سپس ریشه های معادله را محاسبه می کنیم. پس از محاسبه ریشه های معادله با روشی دقیقاً مشابه آنچه در مورد معادلات همگن گفته شد می توان جواب عمومی معادله را محاسبه کرد.

برای مثال معادله‌ی زیررا داریم:

$$t_n - 7t_{n-1} + 12t_{n-2} = 2^n + n + 3^n(n+1)$$

معادله‌ی مشخصه را بدست می آوریم :

$$(x^2 - 7x + 12)(x-2)^{0+1}(x-1)^{1+1}(x-3)^{1+1} = 0$$
$$\Rightarrow (x-3)(x-4)(x-2)(x-1)^2(x-3)^2 = 0$$

$\Rightarrow r_1 = 1, r_2 = 1, r_3 = 2, r_4 = 3, r_5 = 3, r_6 = 3, r_7 = 4$

$\Rightarrow t_n = c_1(1)^n + c_2 n(1)^n + c_3(2)^n + c_4(3)^n + c_5 n(3)^n + c_6 n^2(3)^n + c_7(4)^n$

مثال :

معادله بازگشتی زیر را حل کنید:

$$t(n) = \begin{cases} 1 & n = 1 \\ 2t(\frac{n}{2}) + n & o.w \end{cases}$$

حل:

این مثال را با استفاده از تغییر متغیر زیر حل می کنیم .

$n = 2^k \Rightarrow t(n) = t(2^k)$

$t(2^k) = 2t(\frac{2^k}{2}) + 2^k = 2t(2^{k-1}) + 2^k$

$g(k) = t(n)$

$\Rightarrow g(k) = 2g(k-1) + 2^k \Rightarrow (x-2)(x-2) = 0 \Rightarrow r_1 = 2, r_2 = 2$

$\Longrightarrow g(k) = c_1 2^k + c_2 k 2^k \qquad 2^k = n \Longrightarrow \log_2 n = k$

$t(n) = c_1 n + c_2 n \log_2^n$

$t(1) = c_1 = 1 \qquad t(2) = 2t(1) + 2 = 4 = 2c_1 + 2c_2 \Rightarrow c_2 = 1$

$\Longrightarrow t(n) = n + n\log_2^n$

مثال :

معادله بازگشتی زیر را حل کنید:

$$T(n) = \begin{cases} 1 & n = 1 \\ \frac{3}{2} & n = 2 \\ \frac{3}{2}T(\frac{n}{2}) - \frac{1}{2}T(\frac{n}{4}) - \frac{1}{n} & o.w \end{cases}$$

حل:

باز با استفاده از تغییر متغیر داریم :

$n = 2^k \Longrightarrow T(2^k) = \frac{3}{2}T(2^{k-1}) - \frac{1}{2}T(2^{k-2}) - \frac{1}{2^k} \Longrightarrow$

$g(k) = \frac{3}{2}g(k-1) - \frac{1}{2}g(k-2) - (\frac{1}{2})^k \Longrightarrow (x^2 - \frac{3}{2}x - \frac{1}{2})(x - \frac{1}{2}) = 0 \Rightarrow$

$r_1 = r_2 = \frac{1}{2}, r_3 = 1 \Longrightarrow g(k) = c_1 + c_2(\frac{1}{2})^k + c_3 k(\frac{1}{2})^k \Longrightarrow$

$$T(n) = c_1 + c_2\frac{1}{n} + c_3\frac{\lg n}{n} \quad , \begin{cases} c_1 + c_2 = 1 \\ c_1 + \frac{c_2}{2} + \frac{c_3}{2} = \frac{3}{2} \\ c_1 + \frac{c_2}{4} + \frac{c_3}{2} = \frac{3}{2} \end{cases}$$

$$\Longrightarrow T(n) = 1 + \frac{\lg n}{n}$$

مثال : معادله زیر را حل کنید.

$$\begin{cases} t(n) + nt(n-1) = 2n! \\ t(0) = 1 \end{cases}$$

حل:

$$t(n) + nt(n-1) = 2n! \Longrightarrow \frac{t(n)}{n!} + \frac{t(n-1)}{(n-1)!} = 2$$

$$\frac{t(n)}{n!} = g(n)$$

$$g(n) + g(n-1) = 2 \Longrightarrow (x-1)(x+1) = 0$$
$$\Longrightarrow r_1 = -1, r_2 = 1 \Longrightarrow g(n) = c_1(-1)^n + c_2(1)^n$$
$$t(n) = \frac{-1}{2}n!(-1)^n + \frac{3}{2}n!$$

مثال :

معادله زیر را حل کنید.

$$T(n) = \begin{cases} a & n = 0 \\ b & n = 1 \\ \frac{1+T(n-1)}{T(n-2)} & o.w \end{cases}$$

حل :

$$T(0) = a \quad T(1) = b \quad T(2) = \frac{1+b}{a} \quad T(3) = \frac{1+a+b}{ab} \quad T(4) = \frac{a+1}{b}$$
$$T(5) = a \quad T(6) = b \quad T(7) = \frac{1+b}{a} \Longrightarrow T(n) = T(n \mod 5) \quad n > 4$$

مثال :

معادله زیر را حل کنید.

$$t(n) = \begin{cases} \frac{1}{4-t(n-1)} & n > 1 \\ 1 & n = 1 \end{cases}$$

حل:

$$t(2) = \frac{1}{3}, t(3) = \frac{3}{11}, t(4) = \frac{11}{41} \Longrightarrow t_n = \frac{p_n}{q_n}$$

$$\frac{p_{n+1}}{q_{n+1}} = t(n+1) = \frac{q_n}{4q_n - p_n} \Longrightarrow q_{n+1} = 4q_n - p_n$$

$$p_{n+1} = q_n \Longrightarrow p_n = q_{n-1}$$
$$q_{n+1} = 4q_n - q_{n-1} \Longrightarrow (x^2 - 4x + 1) = 0 \Longrightarrow x = 2 \pm \sqrt{3}$$
$$\Longrightarrow q_n = c_1(2+\sqrt{3})^n + c_2(2-\sqrt{3})^n \quad q_2 = 3, \quad q_3 = 11$$

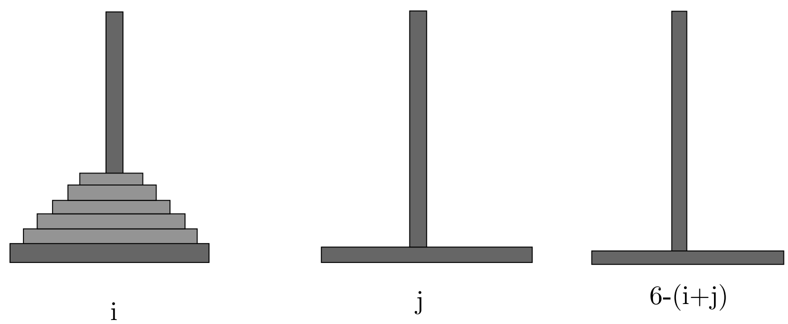
۲.۲ الگوریتم های بازگشتی

در برخی زبان های برنامه نویسی این امکان وجود دارد که ساختار استقرایی را پیاده سازی کنیم، منظور از ساختار استقرایی آن است که یک راه حل مقدماتی برای یک یا چند وضعیت اولیه از مسأله از قبل مشخص باشدو به علاوه یک رابطه استقرایی داشته باشیم که بتواند جمله اخیر را به‌یک یا چند جمله‌ی ماقبل آن مرتبط نماید یعنی آنکه اگر راه حل مسأله تا قبل از مرحله‌ی اخیر دانسته شود بتوانیم این راه حل را توسعه دهیم وبه یک راه حل در این مرحله دست یابیم.

مسأله برج های هانوی:

فرض کنید سه میله به شماره های ۱و۲و۳ شماره گذاری شده اند، روی یکی از میله ها n دیسک قرار دارد، این n دیسک روی میلهi ام طوری قرار گرفته اند که هرگز دیسک بزرگتری روی دیسک کوچکتر قرار نگرفته است. هدف آن است که این n دیسک را به میلهj ام منتقل کرد که این کار با استفاده از میله کمکی باقیمانده صورت می گیرد و هرگز دیسک بزرگتر روی دیسک کوچکتر قرارنمی گیرد و هر بار تنها یک دیسک جابجا میشود.

$i \longrightarrow j \qquad\qquad i+j+x = ۱+۲+۳ \longrightarrow x = ٦-(i+j)$




```
void  hanoi(int n,int i,int j)
    {
      if(n > 0) {
              hanoi(n-1,i,6-(i+j)) ;
                  cout << i << "→" << j ;
              hanoi(n-1,6-(i+j),j);
               }
    }
```

$T(۰) = ۰$

$T(n) = T(n-۱) + ۱ + T(n-۱) \Longrightarrow$

$T(n) = ۲\ T(n-۱) + ۱ \Rightarrow T(n) = ۲^n - ۱$

تمرین:

الگوریتمی بنویسید که به وسیله آن در مسأله برج هانوی هیچ دیسکی را نتوان مستقیماً از میله i ام به میله j ام یا بالعکس منتقل کرد (رابطه بازگشتی که تعداد حرکت های لازم برای انتقال n دیسک به میله j ام را می دهد محاسبه کنید.)

حل :

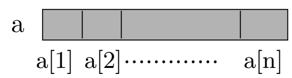
```
Void Hanoi (int i , int j)
{
    if (n > ۰)
      {
         Hanoi(n-1, i , j);
         cout≪ i ≪ " → " ≪ ٦ − (i + j);
         Hanoi(n-1, j , i);
         cout≪ ٦ − (i + j) ≪ " → " ≪ j;
         Hanoi(n-1, i , j);
      }
}
```

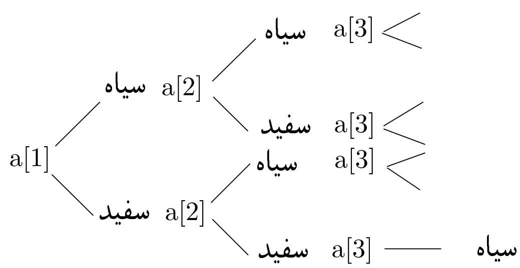
$$\begin{cases} T(۰) = ۰ \\ T(۱) = ۲ \\ T(n) = ۳\ T(n-۱) + ۲ \end{cases} \quad \Rightarrow T(n) = ۳^n - ۱$$

مثال :

فرض کنید نواری به طول n داریم که می خواهیم برخی از خانه های این نوار را طوری رنگ آمیزی کنیم که هیچ سه خانه سفیدی مجاور هم نباشند. دراین صورت الگوریتم بازگشتی که تمام حالت های ممکن برای این مقصود را ایجاد می کند را به دست آورید. همچنین رابطه بازگشتی که تعداد حالات ممکن را برای این مقصود به ما می دهدرا نیز بدست آورید.(فرض آن است که n خانه سفید رنگ اند و با رنگ سیاه می خواهیم رنگ آمیزی کنیم.)

حل:






```
Void Coloring (int n){
    if (n == ۱){
        printf('○');        printf('●');}
    else if (n == ۲) {
        printf('○○');     printf('○●');     printf('●○');     printf('●●'); }
    else if (n == ۳) {
        printf('○●●');     printf('●○●');     printf('●●○');     printf('○○●');
        printf('○ ● ○');     printf('● ○ ○');     printf('● ● ●'); }
    else {
        ●.Coloring(n-1);
        ○●.Coloring(n-2);
        ○ ○ ●.Coloring(n-3);
        }
                }
```

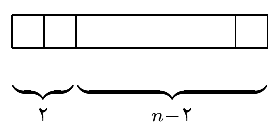
$$a(n)=\begin{cases} ۲ & n=۱\\ ۴ & n=۲\\ ۷ & n=۳\\ ۱\times a(n-۱)+۱\times a(n-۲)+۱\times a(n-۳) & else\end{cases}$$

مثال :

آرایه ای به طول n در اختیار داریم.می خواهیم با حداقل تعداد مقایسه‌هاماکزیمم و می نیمم آرایه را بدست آوریم . رابطه بازگشتی را ارائه دهید که حداقل تعداد مقایسه‌ها را محاسبه کند.

حل :

اگر n زوج باشدمطابق شکل زیر می توانیم دو خانه آرایه را جدا نموده پس می توان گفت که یک مقایسه برای بدست آوردن ماکزیمم و می نیمم دوخانه اول نیازداریم و باید ماکزیمم n-2 عدد را باماکزیمم دو عدد اول و می نیمم n-2 عدد را با می نیمم دو عدد اول مقایسه کنیم .پس جمعاً سه مقایسه نیاز است[۱] .



پس داریم :

$$T(n)=\begin{cases} ۱ & n=۲\\ T(n-۲)+۳ & n=۲k\end{cases} \Longrightarrow T(n)=\frac{۳}{۲}n-۲$$

و اگر n فرد باشد یک خانه‌ی آرایه را جدا نموده پس n-1 خانه‌ی باقی‌مانده عددی زوج است و مطابق رابطه‌ی قبل تعداد مقایسه ها را برای این n-1 خانه‌به دست می آوریم ،سپس دو مقایسه نیز برای پیدا نمودن ماکزیمم ومی نیمم نهایی بین خانه‌ی اول و ماکزیمم ومی نیمم n-1 خانه‌نیاز داریم ،پس جمعاً $۲+(۲-(۱-n)\frac{۳}{۲})$ مقایسه برای n فردنیاز است .پس در کل داریم :

۱ (اگر n زوج باشدمی توانیم آرایه را نصف کنیم ، آنگاه $۲T(\frac{n}{۲})$ مقایسه داریم و دو مقایسه برای ماکزیمم و می نیمم هر گروه . پس در کل $T(n)=۲T(\frac{n}{۲})+۲$ حالت داریم .که همان $T(n)=\frac{۳}{۲}n-۲$ را به ما می دهد.)

$$T(n) = \begin{cases} 0 & n = 1 \\ \frac{3}{2}(n-1) & n = 2k+1 \\ \frac{3}{2}n - 2 & n = 2k \end{cases}$$

درخت AVL :

درخت جستجوی دودوئی که اختلاف ارتفاع زیر درخت چپ و راست هر گره ۱ باشد . ارتفاع این درختاز log n است . پس برای مقایسه باO(log n) می توان گره ها را ویزیت کرد .

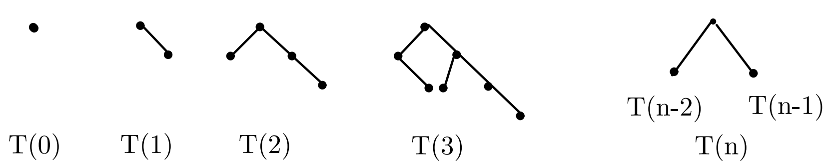
درخت جستجوی دودوئی : درخت دودوئی که هر گره آن دارای برچسب است بطوریکه فرزند چپ کوچکتر مساوی فرزند راست باشد .

مثال :

رابطه بازگشتی که حداقل تعداد گره های یک درخت AVL به ارتفاع ۴ را به ما بدهد کدام است ؟

حل :

ارتفاع زیر درخت سمت راست درخت AVL با ارتفاع n ، n-1 و ارتفاع زیر درخت سمت چپ n-2 است .



$$T(n) = \begin{cases} 1 & n = 0 \\ 2 & n = 1 \\ 4 & n = 2 \\ T(n-1) + T(n-2) + 1 & o.w \end{cases}$$

بین دنباله فیبوناچی و دنباله $T(n) = T(n-2) + T(n-1) + 1$ رابطه ای بصورت زیر برقرار است:

$$
\begin{array}{ccccccccccc}
1 , & 1 , & 2 , & 3 , & 5 , & 8 , & 13 , & 21 , & 34 , & 55 \\
 & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 & & 1 , & 2 , & 4 , & 7 , & 12 , & 20 , & 33 , & 54
\end{array}
$$

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

$$f(n) = f(n-1) + f(n-2) \quad f(0) = 1 \quad , f(1) = 1$$

معادله فیبوناچی‌با شرایط زیر را حل کنید .

$$f(n) = f(n-1) + f(n-2) \qquad f(0) = 0, f(1) = 1$$

$$x^2 - x - 1 = 0 \quad \Rightarrow \quad r_1 = \tfrac{1+\sqrt{5}}{2} \quad , r_2 = \tfrac{1-\sqrt{5}}{2}$$

$$f(n) = c_1(\tfrac{1+\sqrt{5}}{2})^n + c_2(\tfrac{1-\sqrt{5}}{2})^n$$

$$\Rightarrow f(n) = \tfrac{1}{\sqrt{5}}(\tfrac{1+\sqrt{5}}{2})^n - \tfrac{1}{\sqrt{5}}(\tfrac{1-\sqrt{5}}{2})^n$$

حداکثر مقایسه برای ویزیت گره‌ی خاص در درخت AVL :

$$n \geq T(h) = f(h+2) - 1 \Longrightarrow n \geq \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{h+2} - 1 \Longrightarrow$$

$$n + 1 \geq \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{h+2} \Longrightarrow \log_{\varphi}^{(n+1)} \geq \log_{\varphi}^{\frac{1}{\sqrt{5}}} + (h+2) \Longrightarrow$$

$$h + 2 \leq \log_{\varphi}^{(n+1)} - \log_{\varphi}^{\frac{1}{\sqrt{5}}} \Longrightarrow h \in O(\log_{\varphi}^{\sqrt{5}(n+1)}) \Longrightarrow h \in O(\log_2^n)$$

یا

$$h + 2 \leq \log_{\varphi}^{(n+1)} - \log_{\varphi}^{\frac{1}{\sqrt{5}}} = \log_2^{(n+1)} \times \log_{\varphi}^2 + \log_{\varphi}^{\sqrt{5}} = 1.44 \log_2^{(n+1)} + 1.67$$

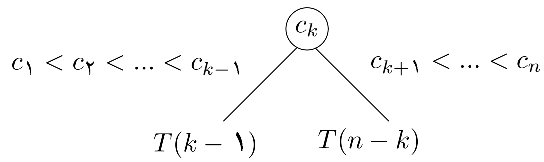
$$\Longrightarrow h \leq 1.44 \log_2^{(n+1)} - 0.33 \Longrightarrow h \in O(\log_2^n)$$

بنابراین حداکثر مقایسه برای ویزیت یک گره به اندازه ارتفاع درخت یعنی $\log_2^n$ است .

مثال :

رابطه‌ای که تعداد درختهای جستجوی دودوئی با n کلید متمایز را به ما بدهد محاسبه کنید .

(n کلید دو به دو متمایز $c_1 < c_2 < \cdots < c_n$)



$$T(1) = 1 \quad T(2) = 2$$

$$T(n) = \sum_{k=1}^{n} T(k-1) \times T(n-k) = \frac{1}{n+1}\binom{2n}{n} \quad \text{(اعداد کاتالان)}$$

مثال :

می خواهیم حاصلضرب n ماتریس $A_1, A_2, \cdots, A_n$ را محاسبه کنیم . به چند طریق می توان این ماتریس ها را جهت ضرب پرانتزگذاری کرد ؟

حل :

$$\begin{array}{ll}
T(0) = 0 & \\
T(1) = 1 & (A) \\
T(2) = 1 & (AB) \\
T(3) = 2 & A(BC), (AB)C \\
T(4) = 5 & A(BCD), (AB)(CD), (ABC)D
\end{array}$$

$$\Rightarrow T(4) = T(1).T(3) + T(2)T(2) + T(3).T(1)$$

$$\vdots$$

$$\underbrace{(A_1 \cdots A_m)}_{m}\underbrace{(A_{m+1} \cdots A_n)}_{n-m} \Rightarrow$$

$$T(n) = \sum_{m=1}^{n-1} T(m)T(n-m) = \frac{1}{n}\binom{2n-2}{n-1}$$

تمرین:

دو رابطه ی دو مثال قبل را با استفاده از تابع مولد اثبات نمایید.

مثال:

کدی در مبنای ۱۰ به طول n زمانی معتبر است که تعداد صفر های ظاهر شده در آن زوج باشد یک رابطه بازگشتی بنویسید که تعداد کل کد های معتبر به طول n را محاسبه نماید.

حل:

$a(n) =$ تعداد تمام کدهای معتبر به طول n

$a(n-1) =$ تعداد تمام کدهای معتبر به طول n-1

(در حالتی که خانه اول صفر نباشد.)

$$\overbrace{\boxed{1,2,...,9}}^{\text{۹ حالت}}\overbrace{\boxed{\qquad\qquad}\boxed{\;\;}}^{a(n-1)}$$

$b(n-1) =$ تعداد تمام رشته هایی به طول n-1 که تعداد صفر هایش فرد است.

(در حالتی که خانه اول صفر باشد.)

$$\boxed{0}\overbrace{\boxed{\qquad\qquad}\boxed{\;\;}}^{b(n-1)}$$

$$\Longrightarrow a(n) = 9a(n-1) + b(n-1)$$

$$\begin{array}{cccc} 1 & 2 & & n-1 \\ \boxed{10} & \boxed{10} & \boxed{\ldots\ldots\ldots} & \boxed{10} \end{array}$$

تعداد تمام رشته ها به طول (n-1)

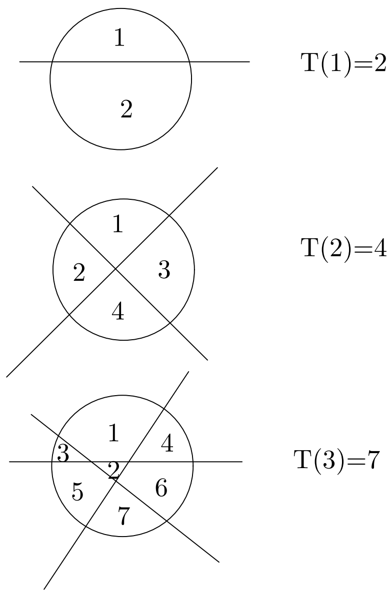
$= a(n\text{-}1)+b(n\text{-}1)=10^{n-1}$

$$a(n-1) + b(n-1) = 10^{n-1} \Rightarrow b(n-1) = 10^{n-1} - a(n-1)$$
$$\Longrightarrow a(n) = 9a(n-1) + 10^{n-1} - a(n-1) = 8a(n-1) + 10^{n-1}$$

مثال :

رابطه بازگشتی بنو یسید که ماکزیمم نواحی را که می توان با n خط بدست آورد نشان دهد.

حل :



$$T(1)=2, T(2)=4, T(3)=7, T(4)=11, \ldots$$

با $n-1$ خط $T(n-1)$ ناحیه می توان ایجاد کرد .پس برای اضا فه کردن خط nام باید $n-1$ خط قبل را قطع کرد که n ناحیه جدید اضافه می شود پس در نتیجه $T(n)=T(n-1)+n$ ناحیه خواهیم داشت.

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

مثال :

فرض کنید عدد طبیعی $n \geq 2$ را دریافت کرده‌ایم به چند طریق می توان این عدد طبیعی را به مجموع اعداد طبیعی بزرگتر مساوی ۲ افراز کرد؟

حل :

$$\begin{array}{ll} T(2)=1: & 2 \\ T(3)=1: & 3 \\ T(4)=2: & 2+2,4 \\ T(5)=3: & 2+3,3+2,5 \\ T(6)=5: & 2+2+2,2+4,3+3,4+2,6 \end{array}$$

برای k امین عدد طبیعی باید به افرازهای عدد $k-2$ ام عبارت « ۲+ » را اضافه کنیم و به جمع وندهای اول افرازهای $k-1$ امین عدد ، یک واحد اضافه کنیم :

$$T(5): \quad \{2+(3)\},\{\underbrace{(2+1)+2}_{3+2}\},\{\underbrace{(4+1)}_{5}\}$$

$$T(6): \quad \{2+(2+2)\},\{2+(4)\},\{\underbrace{(2+1)+3}_{3+3}\},\{\underbrace{(3+1)+2}_{4+2}\},\{\underbrace{(5+1)}_{6}\}$$

در نتیجه داریم:

$$\mathrm{T(n)}=\begin{cases} 1 & n=2 \\ 1 & n=3 \\ T(n-1)+T(n-2) & o.w \end{cases}$$

مثال :

رشته ای به طول n شامل ارقام ۰،۱،۲ زمانی معتبر است که دو صفر متوالی یا دو یک متوالی نداشته باشد، رابطه بازگشتی بنویسید که تعداد رشته های معتبر به طول n را نشان دهد.

جواب :

α_n : تمام کدهای معتبر به طول n.

a_n : تمام کدهای معتبر به طول n که با یک شروع شوند.

b_n : تمام کدهای معتبر به طول n که با صفر شروع شوند.

c_n : تمام کدهای معتبر به طول n که با دو شروع شوند.

$$\alpha_n = a_n + b_n + c_n$$

$$a_n = b_{n-1} + c_{n-1} \quad ,b_n = a_{n-1} + c_{n-1} \quad ,c_n = a_{n-1} + b_{n-1} + c_{n-1}$$

$$\Rightarrow c_n = \alpha_{n-1} \Rightarrow c_{n-1} = \alpha_{n-2}$$

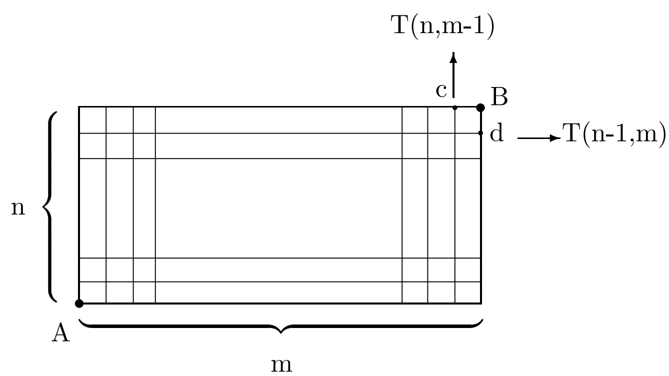
$$\Longrightarrow \alpha_n = a_n + b_n + c_n = 2a_{n-1} + 2b_{n-1} + 3c_{n-1} =$$

$$۲(a_{n-۱}+b_{n-۱}+c_{n-۱})+c_{n-۱}=۲\alpha_{n-۱}+c_{n-۱}$$
$$\Longrightarrow \alpha_n = ۲\alpha_{n-۱}+\alpha_{n-۲}$$

مثال :

یک صفحه $n \times m$ داریم. موشی در این صفحه است که فقط می تواند به طرف بالا و یا به طرف راست حرکت کند. رابطه بازگشتی را بنویسید که تمام مسیرهای ممکن برای حرکت موش از مبدأ A به مقصد B را ارا ئه دهد.

حل :



در شکل بالا اگر تعداد سطرهای جدول n و تعداد ستو نهای جدول m باشد موش برا ی رسیدن به نقطه B یا باید ازنقطه c بگذرد که در این حالت $T(n,m-۱)$ مسیروجود دارد و یا باید از نقطه d عبور کند که در این حالت $T(n-۱,m)$ مسیر وجود دارد. در نتیجه داریم :

$$T(m,n)=۱\times T(n,m-۱)+T(n-۱,m)\times ۱ \quad ,T(n,\circ)=T(\circ,m)=\circ$$
$$T(m,n)=\binom{m+n}{n,m}$$

تمرین:

در یک مثلث متساوی الاضلاع ، اضلاع را به n قسمت مساوی تقسیم می نماییم و به موازات ضلع ها آنها را به هم وصل می نماییم ،رابطه‌ای بازگشتی به دست آورید که تمام متوازی الاضلاع های ساخته شده رابه دست دهد.

تمرین :

صفحه ای n*n در اختیار داریم. ماری در این صفحه وجود دارد که تنها به طرف پایین یا راست حرکت می کند.این مار از بالای صفحه شروع به حرکت نموده و وقتی

که به قطر می رسد به طرف پایین حرکت می کند. تعداد حالتهایی که می تواند به نقطه‌ی مقابل برسدرا بدست آورید؟

تمرین :

یک n ضلعی محدب در اختیار داریم. به چند حالت می توان آن رابه مثلث های مختلف افراز کرد؟

تمرین:

زیر مجموعه های یک مجموعه n عضوی را به دو روش باینری و بازگشتی بیابید.

راهنمایی : در روش باینری، به ازای هر عضو یک بیت در نظر می گیریم اگر عضو زیر مجموعه باشد آن را ۱ و اگر عضو زیر مجموعه نباشد آن را ۰ در نظر می گیریم.

1	2	3	4	
0	0	0	0	{}
0	0	0	1	{4}
0	0	1	0	{3}

در روش بازگشتی نیز زیر مجموعه های n-1 عضوی را دوبار می نویسیم و در ردیف دوم n امین عضو را اضافه می کنیم.

تمرین:

برنامه ای بنویسید که عدد n را گرفته و مربع جادویی آن که در آن اعداد ۱ تا $n^{۲}$ چیده شده اند را به ما بدهد.

تمرین:

بازی tic tac toe را پیاده سازی نمایید.

۳.۲ قضیه اساسی (Master Theorem)

اگر $T(n) = a\,T(\frac{n}{b}) + f(n)$ که در آن $a \geq ۱\ ,\ b > ۱\ ,\ f : N \to R^{+}$ آنگاه نتایج زیر برقرار است :

۱) اگر $f(n) \in O(n^{log\,{}^{a}_{b} - \epsilon})$ برای یک $\epsilon > ۰$ برقرار باشد آنگاه

نتیجه می شود که $T(n) \in \Theta(n^{log\,^a_b})$.

۲) اگر $f(n) \in \Theta(n^{log\,^a_b}(log\ n)^k)$ باشد آنگاه $T(n) \in \Theta(n^{log\,^a_b}(log\ n)^{k+1})$.

۳) اگر $f(n) \in \Omega(n^{log\,^a_b+\epsilon})$ و برای یک $0 < \delta < 1\ ,\ \epsilon > 0$ از یک جایی به بعد داشته باشیم $a\ f(\frac{n}{b}) < \delta\ f(n)$ آنگاه $T(n) \in \Theta(f(n))$.

(البته اگر به جای $\frac{n}{b}$ ، $\lceil \frac{n}{b} \rceil$ یا $\lfloor \frac{n}{b} \rfloor$ نیز قرار گیرد باز نتیجه‌ی بالا برقرار است .)

اثبات:

اثبات شامل دو قسمت است.
اولین قسمت قضیه راتحت یک فرض ساده که تابع T(n) روی n های برابر توان های صحیح از $b > 1$ یعنی $n = 1, b, b^2, ..$ آنالیز می کندودومین قسمت نشان میدهد که چطور این آنالیز می تواند به همه اعداد مثبت طبیعی توسعه پیدا کند و صرفا تکنیکهای ریاضی برای بررسی کف وسقف مسئله به کار می رود.
با این حال ما باید مراقب باشیم وقتی برای اثبات قضیه ای فرض را ساده می نماییم این ساده سازی نباید ما را به نتایج غلط برساند .به عنوان مثال تابعی مانند T(n) را در نظر بگیرید که وقتی n توان هایی از ۲ است برابر n است ،در این صورت هیچ ضمانتی نیست که T(n)∈O(n) ، زیرا می توان تابع T(n) را به این صورت تعریف نمود:

$$\mathrm{T(n)}= \begin{cases} n & \mathrm{n=1,2,3,..} \\ n^2 & \mathrm{otherwise} \end{cases}$$

در حالی که $\mathrm{T(n)}\in\mathrm{O}(n^2)$.

اثبات برای توان های صحیح :

اولین قسمتاثبات قضیه اساسی آنالیزرابطه بازگشتی زیراست.

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

تحت این فرض که n توان صحیحی از $b > ۱$ است که الزامی به صحیح بودن b نیست. این آنالیز در سه لم بیان می گردد. اولین لم مسأله‌ی حل رابطه ی بازگشتی مذکور را به مسأله‌ی تساوی یک عبارت شامل یک سیگما کوچک می کند . دومین لم محدوده‌ی سیگما را تعیین می نماید و سومین لم از دو لم قبلی برای اثبات قضیه‌ی اساسی با فرض اینکه n توان های صحیحی از b است ، استفاده می نماید.

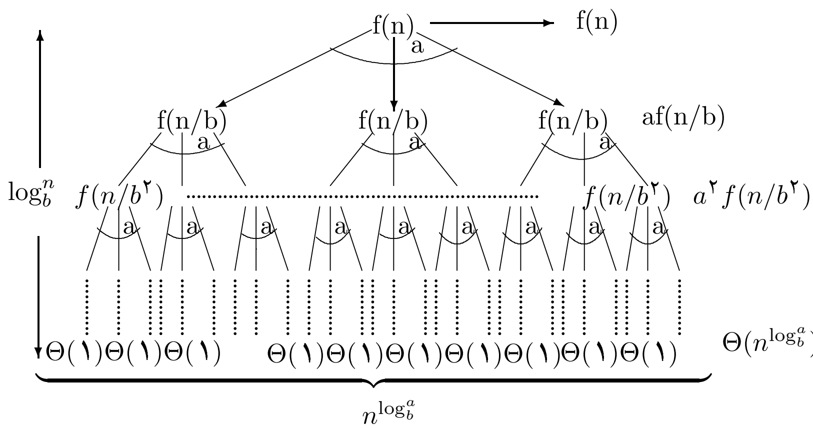
لم ۱ :
فرض کنید $a \geq ۱$ و $b > ۱$ دو عدد ثابت باشند و $f(n)$ تابعی نامنفی از توان‌های صحیحی از b باشد. تابع T(n) را روی توان های صحیح b به صورت زیر تعریف می کنیم .

$$\mathrm{T(n)}= \begin{cases} \theta(1) & \mathrm{n=1} \\ aT(\frac{n}{b}) + f(n) & n = b^i \end{cases}$$

که i عدد صحیح مثبتی است .
حال ثابت می کنیم :

$$T(n) = \theta(n^{\log_b^a}) + \sum_{j=۰}^{\log_b^{n-۱}} a^j f(n/b^j) \qquad (۱)$$

اثبات :
ما از درخت بازگشتی استفاده می کنیم که ریشه‌ی آن دارای ارزش $f(n)$ است و دارای a فرزند می باشد، که هر کدام دارای ارزش $f(n/b)$ هستند. هرکدام از این فرزند ها دارای a فرزند با ارزش $f(n/b^۲)$ است ،پس ما $a^۲$ گره با فاصله ۲ از ریشه داریم . به طور کلی تعداد a^j گره با فاصله j از ریشه وجود دارد که هر کدام دارای ارزش $f(\frac{n^j}{b})$ می باشد. ارزش هر برگ این درخت برابر $T(۱) = \Theta(۱)$ است و هر برگ در عمق $\log_b^n$ قرار دارد. پس درختی با ارتفاع $\log_b^n$ و تعداد $n^{\log_b^a}$ برگ خواهیم داشت .



ما می توانیم رابطه (1) را با جمع کردن ارزش هر سطح از درخت و ارزش برگ های آن محاسبه کنیم .همانگونه که در شکل نشان داده شده است ، ارزش هر سطح مثل j که دارای گره های داخلی‌ست به صورت $a^j f(n/b^j)$ می باشدوبنابراین مجموع همه گره های داخلی سطح ها به صورت زیر است :

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

وارزش تمام برگ ها نیز برابر $\Theta(n^{\log_b^a})$ می باشد .به این ترتیب رابطه (1) به دست می آید.

در این قسمت باید به نکته ظریفی اشاره نمودوآن اینکه بر حسب درخت بازگشتی سه مورد نام برده شده در قضیه‌ی اساسی می تواند به ترتیب مطابق با سه مورد زیر باشد:

۱) ارزش کل درخت به ارزش برگ هاوابسته باشد ، یعنی ریشه و گره های داخلی تاثیر چندانی نداشته باشند.

۲) ارزش کل درخت به‌صورت تقریباً مساوی در سطوح درخت پخش شده باشد .

۳) ارزش کل درخت به ارزش ریشه بستگی داشته باشدو برگ ها و گره های داخلی تاثیر چندانی نداشته باشند .

لم ۲ :

فرض کنید $a \geq 1$ و $b > 1$ دو عدد ثابت باشند و $f(n)$ تابعی نامنفی از توان‌های صحیحی از b باشد. و تابع g(n) را به صورت زیرداشته باشیم :

$$g(n) = \sum_{j=0}^{\log_b^{n-1}} a^j f(n/b^j) \qquad (2)$$

آنگاه خواهیم داشت :

۱)اگر $f(n) = O(n^{\log_b^a - \epsilon})$ ، برای مقادیر ثابت مثبت $\epsilon > 0$ ، داریم $g(n) = O(n^{\log_b^a})$.

۲)اگر $f(n) = \Theta(n^{\log_b^a})$ آنگاه $g(n) = \Theta(n^{\log_b^a}.\log n)$.

۳)اگر برای بعضی از مقادیر ثابت $c < 1$ و تمام مقادیر ثابت $n \geq b$ داشته باشیم $af(n/b) \leq cf(n)$ آنگاه $g(n) = \Theta(f(n))$.

اثبات:

- برای اثبات مورداول داریم $f(n) = O(n^{\log_b^a - \epsilon})$ پس در نتیجه داریم $f(\frac{n}{b^j}) = O((n/b^j)^{\log_b^a - \epsilon})$ که با جایگذاری دررابطه (۲) رابطه زیر بدست می آید :

$$g(n) = O\left(\sum_{j=0}^{\log_b^{n-1}} a^j \left(\frac{n}{b^j}\right)^{\log_b^a - \epsilon}\right) \qquad (3)$$

همچنین بااستفاده از سری های هندسی داریم :

$$\begin{aligned}
\sum_{j=0}^{\log_b^{n-1}} a^j \left(\frac{n}{b^j}\right)^{\log_b^a - \epsilon} &= n^{\log_b^a - \epsilon} \sum_{j=0}^{\log_b^{n-1}} \left(\frac{ab^\epsilon}{b^{\log_b^a}}\right)^j \\
&= n^{\log_b^a - \epsilon} \sum_{j=0}^{\log_b^{n-1}} (b^\epsilon)^j \\
&= n^{\log_b^a - \epsilon} \left(\frac{b^{\epsilon \log_b^n} - 1}{b^\epsilon - 1}\right) \\
&= n^{\log_b^a - \epsilon} \left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right).
\end{aligned}$$

به طوری که b وϵ ثابت هستند .می توان عبارت آخر را به صورت $n^{\log_b^a - \epsilon} O(n^\epsilon) = O(n^{\log_b^a})$ نوشت.

با قرار دادن این عبارت در رابطه (۳) رابطه زیر بدست می آید:

$$g(n) = O(n^{\log_b^a})$$

بدین ترتیب مورد اول اثبات می شود.

- برای مورد دوم با رابطه‌ی $f(n) = \Theta(n^{\log_b^a})$ داریم

$$f(n/b^j) = \Theta((n/b^j)^{\log_b^a})$$

و با جایگذاری دررابطه (۲) رابطه (۴) بدست می آید:

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b^{n-1}} a^j \left(\frac{n}{b^j}\right)^{\log_b^a}\right) \qquad (۴)$$

باز با ساده سازی روابط داریم :

$$\sum_{j=0}^{\log_b^{n-1}} a^j \left(\frac{n}{b^j}\right)^{\log_b^a} = n^{\log_b^a} \sum_{j=0}^{\log_b^{n-1}} \left(\frac{a}{b^{\log_b^a}}\right)^j$$

$$= n^{\log_b^a} \sum_{j=0}^{\log_b^{n-1}} 1$$

$$= n^{\log_b^a} \log_b^n$$

که با جانشین کردن این عبارت در رابطه (۴) داریم :

$$g(n) = \Theta(n^{\log_b^a} \log_b^n) \Longrightarrow$$

$$g(n) = \Theta(n^{\log_b^a} \log n)$$

که حالت دوم نیز ثابت شد.

- در مورد سوم با استفاده از رابطه (۲) می توانیم نتیجه بگیریم که برای توان های صحیح b ،داریم $g(n) = \Omega(f(n))$. با این فرض که $af(n/b) \leq cf(n)$ برای بعضی مقادیر ثابت $c < 1$ و تمام مقادیر $n \geq b$ داریم $f(n/b) \leq (c/a)f(n)$.

$$f(n/b) \leq (c/a)f(n) \Rightarrow f(n/b^j) \leq (c/a)^j f(n) \Rightarrow a^j f(n/b^j) \leq (c^j f(n))$$

حال با کمی دستکاری در رابطه (۲) داریم :

$$
\begin{aligned}
g(n) &= \sum_{j=۰}^{\log_b^{n-۱}} a^j f(n/b^j) \\
&\leq \sum_{j=۰}^{\log_b^{n-۱}} c^j f(n) \\
&\leq f(n) \sum_{j=۰}^{\infty} c^j \\
&= f(n) \left(\frac{۱}{۱-C} \right) \\
&= O(f(n)).
\end{aligned}
$$

که در آن c ثابت است.پس برای توان های صحیح b ، داریم $g(n) = \Omega(f(n)), g(n) = O(f(n))$ بنابراین $g(n) = \Theta(f(n))$. بدین ترتیب مورد سوم لم ۲ نیز اثبات شد.

لم ۳

فرض کنید $a \geq ۱$ و $b > ۱$ دو عدد ثابت باشند و $f(n)$ تابعی نامنفی از توان‌های صحیحی ازb باشد. تابع T(n) را روی توان های صحیح b به صورت زیر تعریف می کنیم .

$$
\mathrm{T(n)}= \begin{cases} \theta(1) & \mathrm{n=1} \\ aT(\frac{n}{b}) + f(n) & n = b^i \end{cases}
$$

که i عدد صحیح مثبتی است .در این صورت خواهیم داشت :

۱) اگر $f(n) = O(n^{\log_b^a - \epsilon})$ برای بعضی مقادیر ثابت $\epsilon > ۰$، آنگاه داریم $T(n) = \Theta(n^{\log_b^a})$.

۲) اگر $f(n) = \Theta(n^{\log_b^a})$ آنگاه $T(n) = \Theta(n^{\log_b^a} \log n)$.

۳) اگر $f(n) = \Omega(n^{\log_b^a + \epsilon})$ ،برای بعضی مقادیر ثابت $\epsilon > ۰$ واگر برای بعضی مقادیر ثابت $c < ۱$ و مقادیر بزرگ n داشته باشیم $af(n/b) \leq cf(n)$ آنگاه داریم $T(n) = \Theta(f(n))$.

اثبات:
ما از موارد لم ۲ برای ارزیابی رابطه (۱) از لم ۱ استفاده می کنیم:
برای مورد اول داریم :

$$T(n) = \Theta(n^{\log_b^a}) + O(n^{\log_b^a}) = \Theta(n^{\log_b^a}),$$

و برای مورد ۲ :

$$T(n) = \Theta(n^{\log_b^a}) + \Theta(n^{\log_b^a} \log n) = \Theta(n^{\log_b^a} \log n),$$

و برای مورد ۳ :

$$T(n) = \Theta(n^{\log_b^a}) + \Theta(f(n)) = \Theta(f(n)),$$

زیرا:

$$f(n) = \Omega(n^{\log_b^{a+\epsilon}})$$

مثال : معادله زیر را حل کنید.

$$\mathrm{T(n)}=\begin{cases} ۱ & \mathrm{n=1} \\ ۳\ T(\frac{n}{۲}) + \sqrt{n} & \mathrm{else} \end{cases}$$

حل :(بر اساس قسمت اول قضیه)

$a = ۳\ ,\ b = ۲$

$$f(n) = \sqrt{n} = n^{\frac{۱}{۲}} \ \in\ O(n^{log\ \frac{۳}{۲} - \epsilon})$$
$$\Rightarrow\ T(n)\ \in\ \Theta(n^{log\ \frac{۳}{۲}})$$

مثال : معادله زیر را حل کنید.

$$\mathrm{T(n)}=\begin{cases} ۰ & \mathrm{n=1} \\ ۲T(\frac{n}{۲}) + n - ۱ & \mathrm{else} \end{cases}$$

حل :(بر اساس قسمت دوم قضیه)

$$T(n) = ۲\ T(\frac{n}{۲}) + n - ۱$$
$$f(n) = n - ۱\ \in\ \Theta(n) \qquad a = ۲\ ,\ b = ۲$$
$$\Theta(n) = \Theta(n^{log\ \frac{۲}{۲}}(log\ n)^{۰})\ \Rightarrow\ T(n)\ \in\ \Theta(n(log\ n)^{۰+۱})$$
$$\Rightarrow\ T(n)\ \in\ \Theta(n\ log\ n)$$

مثال : معادله زیر را حل کنید.

$$T(n) = \begin{cases} ۱ & \mathrm{n=1} \\ ۳\ T(\frac{n}{۲}) + n^{۲} & \mathrm{else} \end{cases}$$

حل :(بر اساس قسمت سوم قضیه)

$$f(n) = n^{۲}\ \in\ \Omega(n^{log\ \frac{۳}{۲}+\epsilon}) \qquad a = ۳\ ,\ b = ۲$$
$$a\ f(\frac{n}{b}) = ۳\ (\frac{n}{۲})^{۲}\ \leq\ \delta\ n^{۲}\ \Rightarrow \delta = \frac{۴}{۵}$$
$$\Rightarrow\ T(n)\ \in\ \Theta(n^{۲})$$

مثال : معادله زیر را حل کنید.

$$T(n) = \begin{cases} \mathrm{c} & \mathrm{n=1} \\ ۲\ T(\frac{n}{۲}) + \log n! & \mathrm{else} \end{cases}$$

حل :

$$f(n) = log\ n!\ \in\ \Theta(n\ logn) \qquad a = b = ۲$$
$$\Theta(n\ logn) = \Theta(n^{log\ \frac{۲}{۲}}(logn))$$
$$\Rightarrow\ T(n)\ \in\ \Theta(n\ (logn)^{۱+۱})\ \Rightarrow\ T(n)\ \in\ \Theta(n\ (logn)^{۲})$$

۴.۲ آنالیز الگوریتم ها

اصل پایایی

اولین اصل ، اصل پایایی می باشد.اگر یک الگوریتم یا یک قطعه برنامه را به دو شکل مختلف پیاده سازی نموده که یکی زمان t_1 و دیگری زمان t_2 را برای اجرا نیاز داشته باشند آنگاه $t_1 \in \theta(t_2)$.بدین معنی که پیاده سازی الگوریتم تأثیری در مرتبه‌ی بزرگی الگوریتم ندارد.

اصل ترتیب گذاری

اگر p_2, p_1 دو تکه برنامه باشند که یکی زمان t_1 و دیگری زمان t_2 را برای اجرا نیاز داشته باشد، علاوه بر آن این دو تکه برنامه هیچ تاثیر جانبی بر هم نداشته باشند، در این حالت زمان لازم برای اجرای متوالی p_1, p_2 برابر $t_1 + t_2$ است .

$$t_1 + t_2 \in \theta(max\{t_1, t_2\})$$

اصل دستورات اتمیک

هر دستور اتمیک دستوری است که در سطح سخت افزار به اجزای کوچکتر تقسیم نمی شود، زمان اجرای این دستورات از $\theta(1)$ است. و همچنین دستورات مقدماتی از ترکیب تعداد متناهی دستورات اتمیک بوجود می آیند که‌زمانی به اندازه $\theta(1)$ نیاز دارند.

آنالیز زمانی :

آنالیز حلقه while:

$i \leftarrow 1$

while $(i <= m)$

  $p(i)$

  $i \leftarrow i + 1$

فرض کنید L زمان لازم برای اجرای الگوریتم باشد، می خواهیم یک کران بالا برای الگوریتم بیابیم. به این ترتیب عمل می کنیم :[1]

$L \leq O(1) +$	برای $i \leftarrow 1$
$O((m + 1) \times 1) +$	برای مقایسه
$O(mt) +$	برای انجام p(i) ها
$O(m) +$	برای اجرای $i \leftarrow i + 1$

[1] فرض کنید یک کران بالایی برای اجراهای متمایز $p(i)$، t باشد.یعنی $p(i) \in O(t)$.(البته از یک جایی به بعد)

برای عمل goto به while $\quad O(m)$

$$\Rightarrow L < O(۲ + ۳m + mt) \Rightarrow L \in O(max\{۱, m, mt\})$$

باید توجه نمود که‌هر سه جز این ماکزیمم گیری مهم هستندو نمی توان هیچ کدام را حذف کرد.

اگر پراکندگی زمانی اجرای $p(i)$ ها زیاد باشد در آن صورت برای هر i یک کران بالایی $p(i)$ رامحاسبه کرده، اگر این کران بالایی را t_i بنامیم در آن صورت زمان اجرا به صورت زیر است:

$$L \in O\left(max\{۱, m, \sum_{i=۱}^{m} t_i\}\right)$$

آنالیز حلقه for:

هر دستور for در سطح سخت افزار مانند while پیاده‌سازی می شود، بنابراین زمان اجرای حلقه for مثل زمان اجرای حلقه while است.

for $i \leftarrow ۱$ to $n - ۱$ do

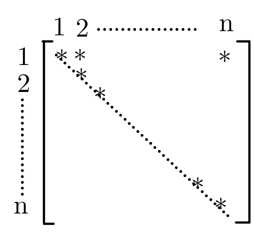
for $j \leftarrow i + ۱$ to n do

write('*');

$$t_i = \sum_{j=i+۱}^{n} (۱) = n - (i + ۱) + ۱ = n - i$$

$$L \in \sum_{i=۱}^{n-۱} t_i = \sum_{i=۱}^{n-۱} (n - i) = n(n - ۱) - \frac{(n - ۱)n}{۲} = \frac{(n - ۱)n}{۲} \in O(n^۲)$$

این موضوع مثل این است که تعداد ستاره های بالای قطر اصلی در شکل ، که هر کدام از O(1) هستند را بشماریم .که برابر $\frac{n^۲-n}{۲}$ هستند.



$\frac{n^۲-n}{۲} \in O(n^۲)$

۵.۲ الگوریتم های مرتب سازی

۱.۵.۲ الگوریتم مرتب سازی انتخابی *Selection Sort*

procedure Selection Sort(T [1···n])

fori ← 1 to n-1 do

$$O(1)\left\{\begin{array}{l}\textbf{minx} \leftarrow \textbf{T[i]}\\ \textbf{minj} \leftarrow \textbf{i}\end{array}\right\}\Omega(1)$$

for j← i+1 to n do

$$O(1)\left\{\begin{array}{l}\textbf{if(T[j]} < \textbf{minx)then}\\ \quad\textbf{minx} \leftarrow \textbf{T[j]}\\ \quad\textbf{minj} \leftarrow \textbf{j}\end{array}\right\}\Omega(1)$$

$$O(1)\left\{\begin{array}{l}\textbf{T[minj]} \leftarrow \textbf{T[i]}\\ \textbf{T[i]} \leftarrow \textbf{minx}\end{array}\right\}\Omega(1)$$

محاسبه کران الگوریتم :

$$\left\{\begin{array}{l} L \leq \sum_{i=1}^{n-1}(O(1)+\sum_{j=i+1}^{n}O(1)) \in O(n^2)\\ L > \sum_{i=1}^{n-1}(\Omega(1)+\sum_{j=i+1}^{n}\Omega(1)) \in \Omega(n^2)\end{array}\right. \Rightarrow L \in \Theta(n^2)$$

دراین نوع پیاده‌سازی زمان‌اجرا ارتباطی با ورودی ها ندارد.

۲.۵.۲ الگوریتم مرتب سازی حبابی *Bubble Sort*

```
Procedere Bubble Sort(T[1..n])
    for i ← 1 to n-1 do
        for j ← i+1 to n do
            if(T[j] < T[i]) then
                swap(T[i], T[j])
```

تعدادجابجایی ها درمرتب سازی انتخابی همیشه $n-1$ است در حالی که درمرتب سازی حبابی در بهترین حالت 0 و در بدترین حالت $\frac{n(n-1)}{2}$ است. بنابراین کران بالای این الگوریتم $O(n^2)$ و کران پایین آن $\Omega(1)$ است .

۳.۵.۲ الگوریتم مرتب سازی درجی *Insertion Sort*

```
Procedere Insertion Sort(T[1..n])
    for i← 2 to n do
        x← T [i]
        j← i-1
        while (j > ۰ && T[j] > x)
            T[j+1] ← T[j]
            j − −
        T[j+1] ← x
```

در این الگوریتم اگر شرط while همیشه false باشد کران پایین الگوریتم از $\Omega(n)$ و کران بالای آن به صورت $\sum_{i=۲}^{n}(O(۱)+O(i)) \in O(n^۲)$ می باشد .پس به صورتی است که به نوع ورودی بستگی دارد .بدین منظور این الگوریتم را در بهترین حالت بدترین حالت و حالت متوسط بررسی می نماییم .

آنالیز بهترین حالت (Best Case Analysis)

بهترین حالت زمانی رخ می دهد که آرایه مرتب باشد، در این حالت با ورود هر عضو جدید باید آن عضو در همان موقعیت درج شود. یعنی حلقه while نباید اجرا شود در این حالت زمان مقایسه شرط while از O(1) است، در نتیجه زمان اجرای الگوریتم از O(n) خواهد بود زیرا n-1 بار اجرا می شود. از طرفی یک کران پایین برای این الگوریتم $\Omega(n)$ بوده است، در نتیجه زمان اجرا از $\theta(n)$ می باشد.

آنالیز بدترین حالت (Worst Case Analysis)

اگر داده ها به صورت نزولی مرتب شده باشند در این صورت با ورود هرعضو جدید (مثل عضو iام) i مقایسه نیاز است. در نتیجه به اندازه $\Omega(i)$ زمان برای دستور while نیاز داریم.پس زمان لازم برای اجرای این الگوریتم $\Omega(n^۲)$ خواهد بود و از طرفی سقف بالایی برای اجرای الگوریتم $O(n^۲)$ است در نتیجه زمان اجرای الگوریتم از $\theta(n^۲)$ خواهد بود.

آنالیز حالت متوسط (Average Case Analysis)

موقعیت نهایی	i	i-1	i-2	.....	1
تعداد مقایسه ها	1	2	3	.....	i

امید ریاضی $E(x) = \sum_{x \in X} x \times f(x)$

$f(x) = \frac{۱}{i}$:احتمالی که x در یک خانه می افتد.

c_i : متوسط زمان لازم برای درج عنصر i ام در موقعیت مناسب درزیرآرایه T[1..i]

$$c_i = \sum_{k=۱}^{i} k \times \frac{۱}{i} = \frac{۱}{i} \sum_{k=۱}^{i} k = \frac{۱}{i} \times \frac{i(i+۱)}{۲} = \frac{i+۱}{۲}$$

در نتیجه متوسط زمان لازم برای اجرای الگوریتم مرتب سازی برابر است با:

$$\sum_{i=۲}^{n} c_i = \sum_{i=۲}^{n} \frac{i+۱}{۲} = \frac{۱}{۲}(\sum_{i=۲}^{n} i + \sum_{i=۲}^{n} ۱) = \frac{۱}{۲}(\frac{n(n+۱)}{۲} - ۱ + n - ۱) = \theta(n^۲)$$

هنگامی که تعداد داده ها کم است رفتار insertion sort از بقیه sort ها بهتر است ولی وقتی تعداد داده ها زیاد باشد چون بهترین حالت خیلی دیر اتفاق می افتدو نمی تواند مناسب باشد.

$n!$ کل حالت هاست که بهترین حالت $\frac{۱}{n!}$است .

۴.۵.۲ مرتب سازی لانه کبوتری *Pigeon hole Sort*

در این مرتب سازی ابتدا آرایه ای به اندازه‌ی بزرگترین عنصر آرایه‌ی نامرتب یعنی u در نظر گرفته می شود، که این آرایه در واقع فراوانی هر عنصر در آرایه‌ی نامرتب را در خود نگاه می دارد.سپس از روی این آرایه آرایه‌ی اولیه را به صورت مرتب باز نویسی می کند .پس این مرتب سازی مبتنی بر فراوانی هاست وجمع فراوانی ها یعنی همان مجموع عناصر u برابر تعداد کل داده هاست .

```
Procedure pigeon hole sort(T[1..n])
    let  m = max{T[i]}   T[i] ∈ Z^{>0},  1≤ i ≤ n
    array u[1..m]
    for i← 1  to  m  do              // θ(m)
        u[i] ← 0
    for i ← 1  to  n  do             // θ(n)
        k← T[i]
        u[k]++
    k← 1
    for i← 1  to  m  do
        while u[i] ≠ 0  do
            T[k]← i
            u[i] − −
            k++
```

آنالیز الگوریتم :

while در حلقه for به اندازه $\sum_{i=1}^{n}(u[i]+1)$ بار و حلقه‌ی for به اندازه m+1 بار اجرا می گردد، بنابراین برای این قسمت از الگوریتم داریم :

$$\sum_{i=1}^{m}(u[i]+1)=\sum_{i=1}^{m}u[i]+\sum_{i=1}^{m}1=n+m$$

بنابراین زمان اجرای این الگوریتم از$\theta(n+m)$ می باشد.

به مثال زیر دقت نمایید:

آرایه نامرتب

T	2	1	3	2	1	2	5	3	4	2	5

m=5

	1	2	3	4	5
u	0	0	0	0	0

	1	2	3	4	5
u	2	4	2	1	2

آرایه مرتب شده

T	1	1	2	2	2	2	3	3	4	5	5

دو تا ۱ — چهار تا ۲ — دو تا ۳ — یکی ۴ — دو تا ۵

۵.۵.۲ جستجوی دودویی *Binary Search*

```
Binary Search (A, temp){
    left = 0;
    right = lenght(A)-1;
    while (left< right) do
        middle= (left+right) / 2;
        if (temp > A[middle])
            left= middle + 1;
        else
            right= middle-1;
    return left;}
```

این الگوریتم temp را در آرایه‌ی A جستجو می نماید .در صورت پیداشدن مکان آن و در صورت پیدا نشدن جایی که temp قرار است باشد را بر می گرداند.

محاسبه زمان اجرای Binary Search:

$$i \leftarrow left, \quad j \leftarrow right, \qquad i = ۰, j = n - ۱$$

d طول آرایه‌ای است که قرار است temp در آن یافت شود.

$$d = j - i + ۱ = n - ۱ - ۰ + ۱ = n \qquad k = middle = \frac{i+j}{۲}$$

فرض کنید k, j, i مقادیر $middle, right, left$ قبل از دستور if باشند، پس از گذر از دستور if دو حالت ممکن است رخ دهد و مقادیر $\hat{d}, \hat{j}, \hat{i}$ را به ترتیب برای مقادیر right, left و طول زیر آرایه‌ی جدید در نظر می گیریم.

- $temp \geq A[middle] \;\; \Rightarrow \;\; \hat{i} = k+۱, \hat{j} = j, \hat{d} = \hat{j} - \hat{i} + ۱ = j - k = j - (\frac{i+j}{۲}) = \frac{j-i}{۲} < \frac{j-i+۱}{۲} = \frac{d}{۲} \Rightarrow \hat{d} < \frac{d}{۲}$
- $else \Rightarrow \hat{j} = k - ۱, \hat{i} = i, \hat{d} = \hat{j} - \hat{i} + ۱ = k - ۱ - i + ۱ = \frac{i+j}{۲} - i = \frac{j-i}{۲} < \frac{j-i+۱}{۲} = \frac{d}{۲} \Rightarrow \hat{d} < \frac{d}{۲}$

پس در هر بار طول آرایه نصف می گردد.

حال برای شرط خروج داریم :

$i \geq j \Rightarrow j - i \leq 0 \Rightarrow j - i + 1 \leq 1 \Rightarrow d \leq 1$

$d_0 = n \;\; d_1 < \frac{d_0}{2} = \frac{n}{2} \;\; d_2 < \frac{d_1}{2} = \frac{n}{4} \cdots$

$d_k \leq 1 \Rightarrow \frac{n}{2^k} \leq 1 \Rightarrow n \leq 2^k \Rightarrow \log_2^n \leq k$

$\Rightarrow k = \lceil \log_2^n \rceil \Rightarrow O\left(\log_2^n\right)$ کران بالایی

۶.۵.۲ مرتب سازی دودویی درجی *Binary Insertion Sort*

```
Binary Insertion Sort(A[1...n]){
   for (i = 1; i < n; i++){
      temp= A[i];
      left=1;
      right=; i
      Binary Search algorithm   // for array A , temp=A[i], left=1 , right=i
      for (j = i; j > left; j--)
        swap(A[j-1],A[j]);
                }
                   }
```

	worst case	best case	average case
insertion sort	O(n^2)	O(n^2)	O(n^2)
Binary insertion sort	O(n^2)	O(nlog n)	O(n^2)

آنالیز بهترین حالت :

- $\sum_{i=1}^{n-1} \lfloor \log_2^{i+1} \rfloor = \sum_{i=2}^{n} \lfloor \log_2^{i} \rfloor \simeq (n+1)\lfloor \log_2^{n+1} \rfloor + 2^{\lfloor \log(n+1) \rfloor + 1} + 2$

  $\in O(nlogn)$ *or*

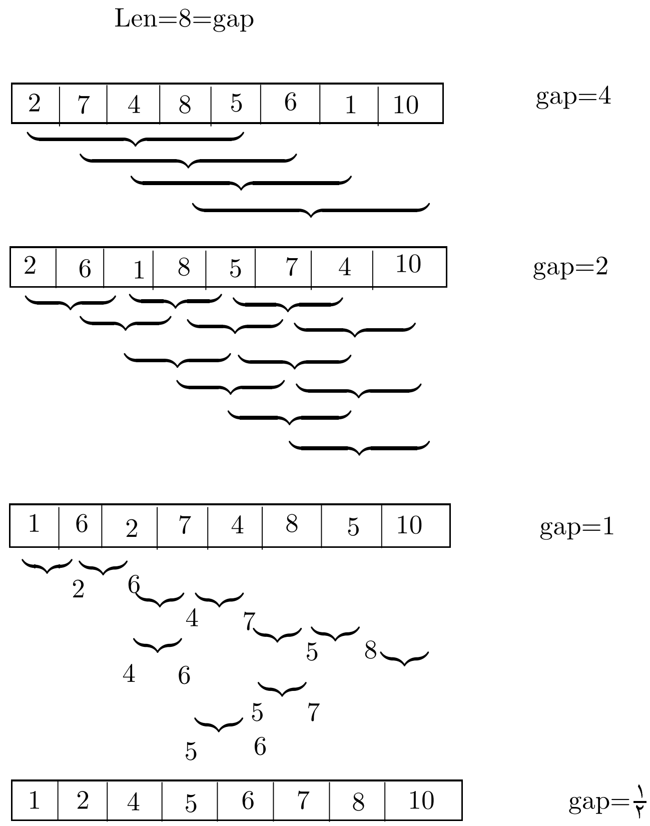
- $\sum_{i=2}^{n} \lfloor \log_2^{i} \rfloor \in O\left(\sum_{i=2}^{n} \log_2^{i}\right) \in O\left(\int_2^n \log_2^x \, dx\right) \in O(nlogn)$

۷.۵.۲ الگوریتم *Shell Sort*

```
void Shell-Sort(int gap,int A[1...n]){
    while((gap/=2)≥1){
      for(i=0;i < length(A);i++){
        int j=i;
        while((j ≥ gap) && (A[j-gap] > A[j])){
          swap(A[j - gap],A[j])
          j -=gap;
        }
      }
    }
}
```

در این الگوریتم ابتدا gap ی به اندازه طول آرایه در نظر می گیریم ، سپس این gap در هر بار عبور نصف می گردد و عناصر با این فاصله با هم مقایسه می گردند.

به مثالی در این مورد دقت فرمایید:
در این مثال ابتدا از سر آرایه عناصر با فاصله ۴ با هم مقایسه می شوند و در صورت لزوم با هم جابجا می گردند، پس جای ۷ و ٦ با هم و ۴ و۱ با هم عوض می شوند. سپس طول فاصله نصف شده و باید دوباره عناصر آرایه از اول با فاصله‌ی ۲ با هم مقایسه شوند ،در این قسمت ابتدا ۲ و۱ با هم جابجا می شوند، سپس ٦ و ۸ با هم مقایسه شده و جابجایی صورت نمی پذیرد و به همین ترتیب ۲ با ۵ و الی آخر تا اینکه فاصله از یک کمتر می شود و از الگوریتم خارج می گردیم که در این حالت آرایه به صورت مرتب شده در آمده است .



بهترین زمان هنگامی است که while دوم اجرا نشود .

	best case	worst case	average case
shell-sort	$n \log n$	$n^{۱.۵}$	$n^{۱.۲۵}$

اعداد بدست آمده به gap بستگی دارند.این الگوریتم را با لیست پیوندی نمی توان پیاده سازی نمود .

۸.۵.۲ الگوریتم *Bucket Sort1*

```
unsighned const m=max
  void Bucket-Sort1(int A[ ],int n)
    int buckets[m];
    for (int i=0;i<m;i++)
      buckets[i]=0;
     for(i=0;i< n;i++)
      ++buckets[A[i]];
    for(int j=0,i=0;j< m;j++)
     for(int k=buckets[j];k>0;- - k)
```
$$// \sum_{j=1}^{m-1} buckets[j] = n$$
```
      A[i++]=j;
```

این الگوریتم همان Pigeon hole sort است و زمان اجرای آن نیز از O(n+m) می باشد.

2	3	4	5	1	2	1

$\Downarrow$

0	2	2	1	1	1

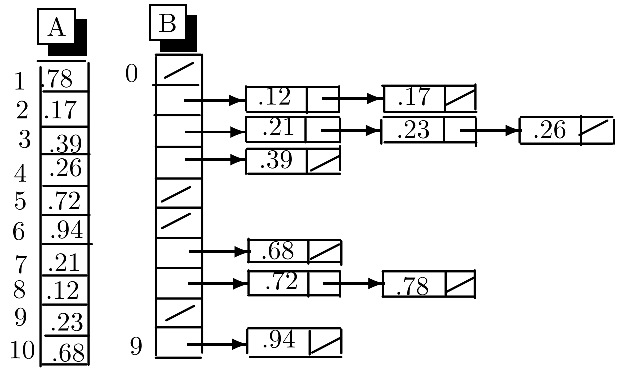
$\Downarrow$

1	1	2	2	3	4	5

۹.۵.۲ الگوریتم *Bucket Sort2*

این الگوریتم نوعی دیگر از Bucket-Sort است که برای مرتب سازی داده های تصادفی که در بازه $[0, 1)$ واقع شده‌اند،مناسب است .

```
Bucket-Sort2(A)
  n ← [A]
  for i ← 1 to n do
     insert A[i] into list B[⌊nA[i]⌋]
  for i ← 0 to n-1 do
     sort list B[i] with insertion sort
  concatenate the list B[0],B[1],...,B[n-1] together in order
```



۱۰.۵.۲ الگوریتم *Bin Sort*

```
for(int i=0;i<m;i++)          //O(m)
     bin [i]= -1;
for(int i=0;i < n;i++)
     bin[a[i]]=a[i];          //O(n)
j=0;
for(int i=0;i < m;i++){
     if(bin[i]!= -1)          //O(n+m)
        a[j]=b[i];
      j++;}
```

زمان اجرای این الگوریتم از $O(m+n)$ است .باید توجه نمود که در این الگوریتم عناصر تکراری نمی توانند باشند ،اگر بخواهیم عناصر تکراری هم داشته باشیم باید برای هر خانه‌ی آرایه یک لیست پیوندی در نظر بگیریم .

a:

3	7	2	8	1

bin:

-1	1	2	3	-1	-1	-1	7	8

a:

1	2	3	7	8

۱۱.۵.۲ الگوریتم *Counting Sort*

```
counting-sort
 for i←0 to k do          k is maximum of elements
     c[i]←0
 for j← 1 to n do
     c[a[j]]←c[a[j]]+1;
 for i← 2 to k do
     c[i] ← c[i]+c[i-1];
 for j ← n downto 1 do
     b[c[a[j]]] ← a[j];
     c[a[j]] ← c[a[j]] -1 ;
```

زمان اجرای این الگوریتم از $\theta(n+k)$ می باشد.
نکته : هر الگوریتم مرتب سازی مبتنی برمقایسه به حد اقل $\log n!$ تعداد مقایسه نیاز دارد .در بدترین حالت حداقل $\lceil \log n! \rceil$ مقایسه نیاز دارد .

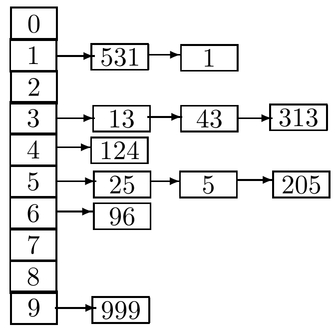
$\lceil \log n! \rceil \in \Omega(n \log n)$

پس مستقل از نوع ورودی در بدترین حالت از$\theta(n \log n)$ می باشد، اما این الگوریتم می تواند در زمانی کمتر از$\theta(n \log n)$ نیز صورت بپذیرد.

۱۲.۵.۲ الگوریتم *Radix sort*

در این الگوریتم بین عناصر مقایسه ای صورت نمی پذیرد.این الگوریتم از نوع stable sort است. دراین روش ابتدا عناصر را به ترتیب یکان مرتب می نماییم .سپس عناصر را به ترتیب وارد آرایه اصلی می نماییم . حال عناصر را به ترتیب دهگان مرتب می نماییم و وارد لیست می نماییم و وارد آرایه می نماییم وبه همین ترتیب ادامه می دهیم تا آرایه مرتب گردد.زمان اجرای این الگوریتم از $O(n\alpha(n))$ می باشد که $\alpha(n)$ حداکثر تعداد ارقام ظاهر شده بین n عدد است .
به مثال زیر دقت کنید :

25 13 43 124 313 513 5 1 999 96 205



حال عناصر را به ترتیب زیر وارد آرایه اصلی می نماییم .

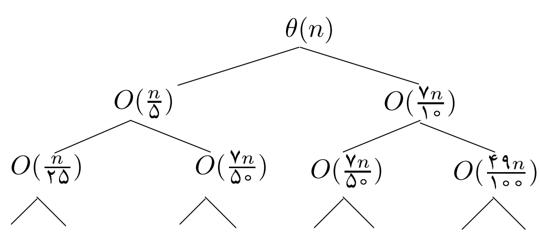
531,1,13,43,313,24,25,5,205,96,999

و به ادامه‌ی مرتب سازی می پردازیم .

۶.۲ عمل Trace کردن (حل کردن) معادلات بازگشتی

- $\begin{cases} T(n) = T(\frac{n}{۵}) + T(\frac{۷n}{۱۰}) + \theta(n) \\ T(۱) = C \end{cases}$

برای این منظور یک درخت بازگشت می سازیم که ریشه آن $\theta(n)$ می باشد.



درخت بالا درخت متقارن پایین رونده نیست یعنی همه زیر درخت ها به صورت یکسان پایین نمی آیند. در این مواقع اگر درخت متقارن نباشد باید O را حساب کنیم .

$$\left(\frac{۷}{۱۰}\right)^i n \leq ۱ \Longrightarrow n \leq \left(\frac{۱۰}{۷}\right)^i \Longrightarrow \log^n_{\frac{۱۰}{۷}} \leq i \Longrightarrow i = \lceil \log^n_{\frac{۱۰}{۷}} \rceil \Longrightarrow$$

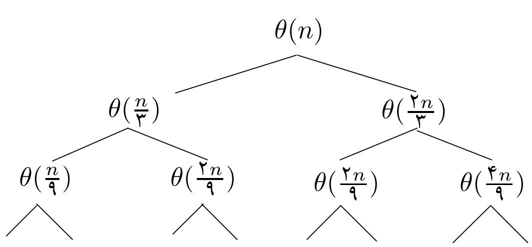
$$T(n) \leq \sum_{i=۰}^{\lceil \log^n_{\frac{۱۰}{۷}} \rceil} \left(\frac{۹}{۱۰}\right)^i n = \frac{۱ - \left(\frac{۹}{۱۰}\right)^{\log^n_{\frac{۱۰}{۷}} + ۱}}{۱ - \frac{۹}{۱۰}}$$

$$۵^{\frac{۱}{۳}} > \frac{۱۰}{۷} \Longrightarrow \log^n_{۵^{\frac{۱}{۳}}} > \log^n_{\frac{۱۰}{۷}} \Longrightarrow ۳\log^n_۵ > \log^n_{\frac{۱۰}{۷}} \Longrightarrow$$

$$T(n) \leq \sum_{i=۰}^{\lceil \log^n_{\frac{۱۰}{۷}} \rceil} \left(\frac{۹}{۱۰}\right)^i n \leq \sum_{i=۰}^{۳\log^n_۵} \left(\frac{۹}{۱۰}\right)^i n$$

معادله بازگشتی زیر را حل کنید.

- $T(n) = T(\frac{n}{۳}) + T(\frac{۲n}{۳}) + \theta(n)$



$$\left(\frac{۲}{۳}\right)^i n \leq ۱ \Longrightarrow n \leq \left(\frac{۳}{۲}\right)^i \Longrightarrow \log^n_{\frac{۳}{۲}} \leq ۱ \Longrightarrow i = \left\lceil \log^n_{\frac{۳}{۲}} \right\rceil$$

$$\Longrightarrow T(n) \leq \sum_{i=۰}^{\left\lceil \log^n_{\frac{۳}{۲}} \right\rceil} n \leq O(n \log^n_{\frac{۳}{۲}})$$

۷.۲ گذری بر اعداد کاتالان Catalan Number

محاسبه تعداد درختان دودویی با n گره[۱]:

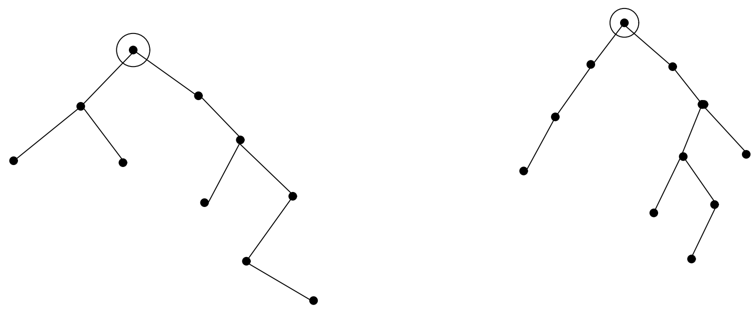
یک درخت گرافی است بی‌سو، که همبند است وطوقه یا دوری ندارد.در اینجا درختان دودویی ریشه دار را بررسی می کنیم .

در شکل ۱ دودرخت دودویی می بینیم که درآنها راسی که دورآن دایره کشیده ایم معرف ریشه است.این درختها را به دلیل اینکه از هر راس حداکثردویال (به نام شاخه) به سمت پایین رسم شده‌اند،دودویی می نامند.

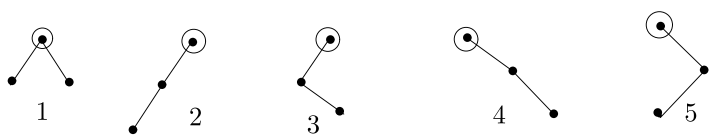
بخصوص این درختان دودویی ریشه دار،مرتب اندبدین معناکه شاخه چپی که از راس به پایین می آیدمتفاوت از شاخه راستی‌که ازهمان راس به پایین می آید در نظر گرفته می شود.در حالت وجود سه راس،پنج درخت دودویی مرتب ریشه‌دار ممکن درشکل دو نشان داده شده (اگرترتیب مهم نباشد،چهار درخت دودویی آخردارای یک ساختارند.)

هدف ماشمارش b_n،تعداد درختان دودویی مرتب ریشه دار مربوط به n راس، $n \geq ۰$ است .با فرض اینکه مقادیر b_i به ازای $۰ \leq i \leq n$ در دست باشد برای به دست آوردن $b_{n+۱}$،راسی را به عنوان ریشه انتخاب و توجه می کنیم که مثل شکل سه زیر ساختارهایی که از چپ وراست ریشه پایین می آیند درختهای(دودویی مرتب ریشه دار) کوچکتری هستند که تعداد راسهای آنهاn است.این درختهای کوچکتر را زیر درختهای درخت مفروض می نامند.از جمله این زیردرختهای ممکن،زیر درخت تهی است که برای آن داریم $b_۰ = ۱$

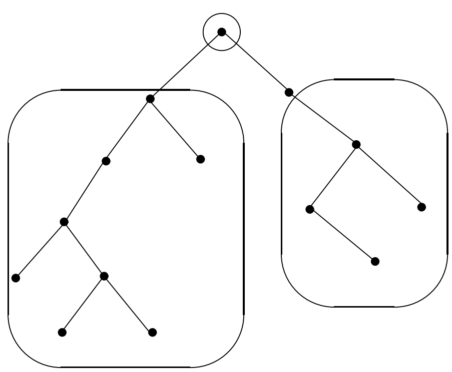
[۱] اثبات برگرفته از کتاب ریاضیات گسسته گریمالدی



شکل ۱



شکل ۲



شکل ۳

اینک بررسی می کنیم که چگونه n راس را می توان در این دو زیر درخت طبقه بندی کرد.

(1) 0 راس در سمت چپ قراردارد،n راس در سمت راست.این امر نتیجه می دهدکه

همه زیرساختارهایی که در b_{n+1} شمرده می شوندبرابر b_0b_n اند.

(2) 1 راس در سمت چپ قرار دارد، $n-1$ راس درسمت راست،که b_1b_{n-1} درخت دوتایی مرتب ریشه دار،با $n+1$ راس به دست می دهد.

$\vdots$

(i+1) i راس در سمت چپ، $n-i$ راس در سمت راست قرار دارند که در این حالت b_{n+1} برابر b_ib_{n-i} است.

$\vdots$

(n+1) n راس در سمت چپ است و هیچ راسی درسمت راست نیست،در این صورت b_{n+1} برابر b_nb_0 از درختهاست .

بنابراین،به ازای همه مقادیر $n \geq 0$

$$b_{n+1} = b_0b_n + b_1b_{n-1} + b_2b_{n-2} + \cdots + b_{n-1}b_1 + b_nb_0$$

و

$$\sum_{n=0}^{\infty} b_{n+1}x^{n+1} = \sum_{n=0}^{\infty}(b_0b_n + b_1b_{n-1} + \cdots + b_{n-1}b_1 + b_nb_0)x^{n+1} \qquad (1)$$

می دانیم که اگر $f(x) = \sum_{i=0}^{\infty} a_ix^i$ تابع مولّد برای دنباله $a_0, a_1, a_2, \cdots$ باشد ، آنگاه $[f(x)]^2$ ، پیچش دنباله $a_0, a_1, \cdots$ با خودش ، یعنی

$$a_0a_0, a_0a_1 + a_1a_0, a_0a_2 + a_1a_1 + a_2a_0, \cdots$$

$$, a_0a_n + a_1a_{n-1} + a_2a_{n-2} + \cdots + a_{n-1}a_1 + a_na_0$$

را تولید می کند. اینک فرض کنید $f(x) = \sum_{n=0}^{\infty} b_nx^n$ تابع مولّد $b_0, b_1, b_2, \cdots$ است. معادله (۱) را به صورت

$$(f(x) - b_0) = x\sum_{n=0}^{\infty}(b_0b_n + b_1b_{n-1} + \cdots + b_nb_0)x^n = x[f(x)]^2$$

می نویسیم .

این معادله ما را به معادله درجه دوم زیر می رساند :

$$x[f(x)]^{2} - f(x) + 1 = 0$$

پس

$$f(x) = \frac{[1 \pm \sqrt{1-4x}]}{(2x)}$$

امّا

$$\sqrt{1-4x} = (1-4x)^{1/2} = \binom{1/2}{0} + \binom{1/2}{1}(-4x) + \binom{1/2}{2}(-4x)^{2} + \cdots$$

که در آن ضریب x^n, $n \geq 1$ برابر است با :

$$\begin{aligned}
\binom{1/2}{n}(-4)^n &= \frac{(1/2)((1/2)-1)((1/2)-2)\cdots((1/2)-n+1)}{n!}(-4)^n \\
&= (-1)^{n-1}\frac{(1/2)(1/2)(3/2)\cdots((2n-3)/2)}{n!}(-4)^n \\
&= \frac{(-1)2^n(1)(3)\cdots(2n-3)}{n!} \\
&= \frac{(-1)2^n(n!)(1)(3)\cdots(2n-3)(2n-1)}{(n!)(n!)(2n-1)} \\
&= \frac{(-1)(2)(4)\cdots(2n)(1)(3)\cdots(2n-1)}{(2n-1)(n!)(n!)} = \frac{(-1)}{(2n-1)}\binom{2n}{n}
\end{aligned}$$

در $f(x)$ رادیکال منفی را انتخاب می کنیم ؛ درغیراین صورت ، مقادیری منفی برای b_n ها پیدا می کنیم ؛ پس

$$f(x) = \frac{1}{2x}\left[1 - \left[1 - \sum_{n=1}^{\infty}\frac{1}{(2n-1)}\binom{2n}{n}x^n\right]\right]$$

و b_n، ضریب x^n در $f(x)$ ، نصف ضریب x^{n+1} در

$$\sum_{n=1}^{\infty}\frac{1}{(2n-1)}\binom{2n}{n}x^n$$

است، به قسمی که

$$b_n = \frac{1}{2}\left[\frac{1}{2(n+1)-1}\right]\binom{2(n+1)}{(n+1)} = \frac{(2n)!}{(n+1)!(n!)} = \frac{1}{(n+1)}\binom{2n}{n}$$

اعداد b_n را که منسوب به ریاضیدان بلژیکی اورژن کاتالان (۱۸۱۴ – ۱۸۹۴) اند ، اعداد کاتالان می نامند. کاتالان آنها را در تعیین تعداد راههای داخل پرانتزگذاشتن عبارت $x_1x_2x_3\cdots x_n$ به کار برده است . هفت عدد اول کاتالان $b_0 = 1, b_1 = 1, b_2 = 2, b_3 = 5, b_4 = 14, b_5 = 42$ و $b_6 = 132$ هستند.

فصل ۳

یادآوری برخی از ساختمان داده ها

۳. برخی از ساختمان داده ها

از آنجایی که هر گونه اطلاعاتی بایستی در قسمتی از حافظه ذخیره گردد و اینکه لزوماً داده ها از یک نوع نیستند ،می توانند در حافظه به اشکال مختلفی ذخیره شوند و همچنین به اشکال مختلفی واکشی شوند. و به علاوه هر داده‌ای با توجه به میزان بزرگی آن ممکن است طول متفاوتی از حافظه را به خود اختصاص دهد . بنابراین مجبور هستیم با توجه به ساختار داده‌ای که داریم انواع مختلف را معرفی نماییم . به طورکلی انواع داده‌ای را می توانیم به دو شکل نوع داده‌ای ساده و مرکب دسته بندی نماییم .

نوع داده ای ساده یا اتمی انواع داده هایی هستند که زبان های بر نامه نویسی آنان را تعریف نموده و در اختیار کاربر قرار می دهد، مانند:int، bool، char و غیره.

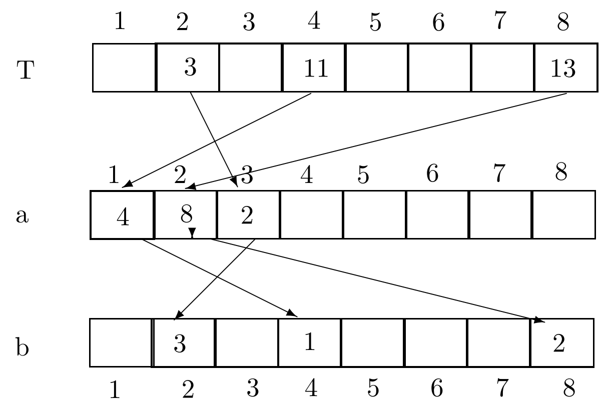
نوع داده‌ای مرکب توسط برنامه نویس ساخته می شود و خود کاربرآنان را به برنامه معرفی می کند.مانند : union ، struct ، و غیره .

آرایه : ساده ترین نوع ساختمان داده است که از تعدادی از حافظه های منطقاً مجاور هم استفاده می کند که از نظر نوع و طول یکسان هستند:

```
x : array ['a'..'z'] of integer
int x[26];
```

۱.۳ آرایه اسپارس (Sparse array)

فرض کنیم آرایه ای داریم که قرار است مقدار گذاری شود ولی نحوه‌ی مقدار گذاری به شکل اسپارس است ، یعنی لزوماً تمام خانه های آرایه مقدار گذاری نمی شوند و فقط برخی از خانه‌های خاص مقدارگذاری می شوند. برای این منظور مقدار گذاری کردن آرایه با مقدار صفر منطقی نیست ، چون طول آرایه بزرگ است و قرار نیست تمام خانه های آرایه در آینده واکشی شوند . برای این منظور ناچاریم از دو آرایه کمکی هم طول با آرایه‌ی اصلی استفاده نماییم .برای درک بهتر ارتباط این آرایه ها به مثال زیر دقت کنید:



ابتدا مقدار متغیر ctr را صفر می کنیم .در هر بار مقدار گذاری آرایه‌ی اصلی یعنی T ، یک واحد به ctr اضافه می کنیم و a[ctr] را برابر اندیس خانه ای از T قرار می دهیم که مقدارگذاری شده است و همچنین [اندیس]b را برابر ctr قرار می دهیم . به این ترتیب مقدارگذاری انجام می پذیرد.

ctr=0 →ctr=ctr+1=1 T[4]=11 → a[ctr]=a[1]=4 b[4]=1

ctr=1 →ctr=ctr+1=2 T[8]=13 → a[ctr]=a[2]=8 b[8]=2

ctr=2 →ctr=ctr+1=3 T[2]=3 → a[ctr]=a[3]=2 b[2]=3

حال اگر بخواهیم بدانیم خانه ای را ما اندیس گذاری کرده ایم یا نه باید دو شرط زیر برقرار باشند:

$$1 \leq b[i] \leq ctr \quad , \quad a[b[i]] = i$$

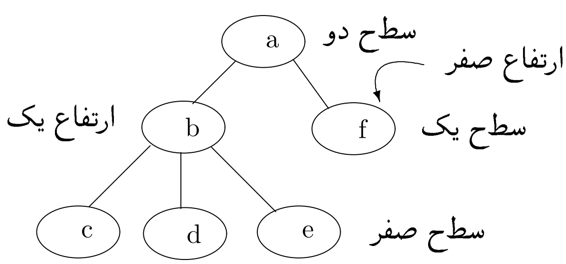
اگر چیزی آدرس دهی نکرده باشیم واضح است که ctr صفر است .
به عنوان مثال می خواهیم بدانیم در آرایه های قبل خانه‌ی 8 ام T را ما مقدارگذاری کرده ایم یا نه .

ممکن است این خانه را ما مقدار دهی کرده باشیم $\Rightarrow 3 \le b[8] \le 1$, $b[8]=2$

این خانه را ما مقدار دهی کرده ایم $\Rightarrow a[b[8]]=a[2]=8$

۲.۳ درخت دودویی

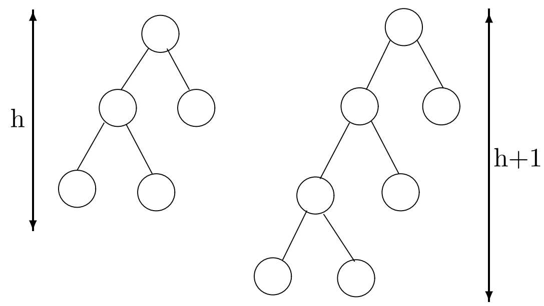
درخت دودویی درخت ریشه داری است که هر گره‌آن حداکثر دو فرزند دارد.
درخت پر: یک درخت باینری است که تمام برگهای آن در سطح صفر اتفاق می افتد.
ارتفاع یک گره از درخت : طول بزرگترین مسیر از آن گره تا برگ است . ارتفاع یک در خت ، ارتفاع ریشه است.
عمق یک گره(Depth) : عبارتست از فاصله آن گره تا ریشه (تعداد یالهای سپری شده تا آن گره) .
سطح یک گره(Level): برای هر گره سطح آن گره برابر است با تفاضل ارتفاع آن درخت از عمق آن گره.
مثال :



گره	ارتفاع	عمق	سطح
a	2	0	2
b	1	1	1
c	0	2	0
d	0	2	0
e	0	2	0
f	0	1	1

مثال: در یک درخت دودویی با n گره تمام گره های داخلی آن دقیقاًدارای دو فرزندمی باشند.E(T) مجموع عمق برگ های درخت و I(T) مجموع عمق گره های

داخلی این درخت می باشد. اگر T(n)= E(T)−I(T) باشد آنگاه رابطه بازگشتی آن کدام است ؟



T(n) T(n+2)

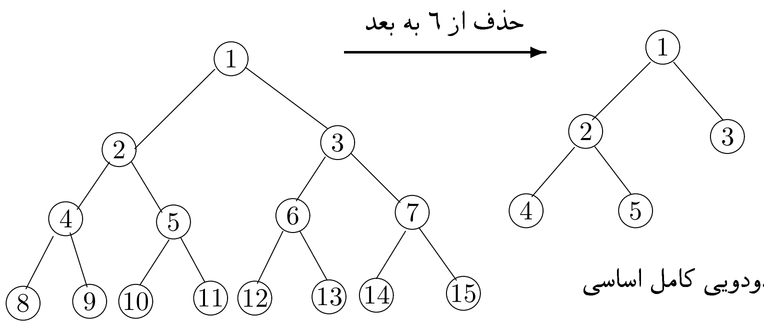
$$E(T) = E(T') - h + h + ۱ + h + ۱$$

$$I(T) = I(T') + h$$

$$\Rightarrow E(T) - I(T) = E(T') - I(T') + ۲ \Rightarrow T(n + ۲) = T(n) + ۲$$

درخت دودویی کامل : درخت دودویی است که هر گره آن هیچ فرزند یا دقیقاً دو فرزند دارد.
درخت دودویی کامل اساسی : درخت دودویی را درخت کامل اساسی می نامیم که تمام گره های داخلی آن دقیقاً دارای دو فرزندباشندمگر گره‌هایی از سطح یک (سطح یکی به آخر). اگر گره‌ای دراین سطح یک فرزندداشته باشد آن فرزند بایستی فرزندچپ باشد وبه علاوه تمام گره های هم سطح آن که در سمت چپ آن آمده‌اند دقیقاً باید دو فرزند داشته باشند و تمام گره های هم سطح آن که در سمت راست قرار گرفته‌اند بایستی فرزند نداشته باشند، همچنین اگر گره ای فرزند نداشته باشد تمام هم سطحی های راست آن باید بدون فرزند باشند. دراین درخت تمام برگ ها در سطح صفر یا یک رخ می دهند پس درخت همیشه متوازن است .
تعریف دیگری از درخت دودویی کامل اساسی :
اگریک درخت دودویی پر را به طور فرضی از ریشه‌به طرف برگ ها در نظر بگیریم و در هرسطح از چپ به راست شماره های صعودی را به هر گره نسبت دهیم ، اگر از

شماره ای به بعد گره ها را حذف کنیم ، درختی که به دست می آید را کامل اساسی گویند.



درخت دودویی کامل اساسی

در برخی کتب منظور از درخت دودویی کامل، درخت دودویی کامل اساسی است .

۳.۳ درخت max heap

درخت دودویی کامل اساسی است که هر گره آن با عددی برچسب خورده که برچسب پدر از برچسب فرزندانش کوچکتر نیست .(برچسب منحصر به فرد نیست ولی کلید منحصر به فرد است .) بهترین ساختمان داده برای پیاده سازی heap آرایه است .به طوری که فرزندان عنصر i ام در خانه های 2i , 2i+1 هستند. همیشه‌عنصر ماکزیمم سر آرایه است ، پس پیدا کردن ماکزیمم از$\theta(\text{۱})$ است . پیدا کردن عنصر مینیمم نیز، چون تنها باید برگ ها را چک کرد.

$$\text{۱}\cdots\lfloor\tfrac{n}{\text{۲}}\rfloor \quad \underbrace{\lfloor\frac{n}{\text{۲}}\rfloor+\text{۱}\cdots n}_{\lceil\frac{n}{\text{۲}}\rceil}$$

(تعداد مقایسه ها $\lceil\frac{n}{\text{۲}}\rceil$ است) .

دو عملیات مهم در heap، sift-up(perculate) وsift-down هستند.عملیات sift-up زمانی لازم است و صورت می گیرد که برچسب یک گره از عددی کمتر به عددی بیشتر تغییر یابدوعملیات sift-down زمانی که عددی بیشتر جایگزین عددی کمتر گردد.

procedure alter − heap (T[1..n],i,v)
{ T[1..n] is a heap ,the value of T[i] is set to v and the heap property is re-established we suppose the $1 \le i \le n$ }
x ← T[i]
T[i] ← v
if v < x then sift-down (T,i)
else percolate (T,i)
....
procedure sift − down (T[1..n],i)
{ this procedure sifts node i down so as to re-establish the heap property in T[1..n] we suppose that T would be a heap if T[i] were sufficiently large we also suppose that $1 \le i \le n$ }
k ← i
repeat
j ← k {find the larger of node j }
if (2j≤ n and T[2j]> T[k])
k ← 2j
if (2j < n and T[2j+1]> T[k])
k ← 2j+1
exchange T[j] and T[k]
{if $j == k$ then the node has arrived at its final position }
until $j = k$ $//O(\log_2^n)$

فرض کنید تعداد گره های heap , n باشد وارتفاع heap, h باشد آنگاه :

$$2^0 + 2^1 + ... + 2^{h-1} < n \le 2^0 + 2^1 + ... + 2^h \Rightarrow \frac{2^h - 1}{2-1} < n \le \frac{2^{h+1}-1}{2-1}$$

- $n \le 2^{h+1}-1 \Longrightarrow n+1 \le 2^{h+1} \Longrightarrow \lg(n+1) \le h+1$
$\Rightarrow h \ge \lg(n+1) - 1 \Longrightarrow h = \lceil \lg(n+1) - 1 \rceil = \lfloor \lg n \rfloor$

- $n > 2^h - 1 \Rightarrow n+1 > 2^h \Rightarrow h < \lg n + 1 \Rightarrow h - 1 < \lg(n+1) - 1 \le h$

پس زمان الگوریتم از $O(\log_2^n)$ است.

procedure percolate(T[1..n],i)
{we suppose that T would be a heap if T[i] were sufficiently small,
we also suppose that 1≤i≤ n the parametr n is not used here}
k ← i
repeat
j←k
if ((j>1) && (T[$\frac{j}{۲}$] < T[k]))
k← $\frac{j}{۲}$
exchange T[j] and T[k]
until j=k; $//O(\log n)$
....
function find − max(T[1..n])
{ returns the largest element of the heap T[1..n]}
return T[1]; $//\theta(1)$
....
procedure delete − max(T[1..n])
Delete the root}
T[1]←T[n]
sift-Down(T[1..n-1],1) $//O(\log n)$

حذف ،حذف منطقی است. آخرین عنصر به جای اولین عنصر قرار می گیرد و عمل sift-Down انجام می شود.

procedure insert − Node(T[1..n],v)
T[n+1] ← v
percolate(T[1..n+1],n+1) $\{O(\lg n)\}$

ساخت درخت heap :

Procedure slow − MakeHeap(T[1..n])
{this procedure makes the array T[1..n] into a heap}
for i ← 2 to n do
percolate(T[1..n],i) $//\ O(n \log n)$

$Procedure\ \ MakeHeap(\text{T[1..n]})$

For $i \leftarrow \lfloor \frac{n}{2} \rfloor$ DownTo 1 Do

SiftDown(T, i) $\qquad //O(n)$

قضیه : الگوریتم بالااز یک آرایه دلخواه به طول n با مرتبه O(n) می تواند Heap بسازد.
برهان :
حلقه Repeat در الگوریتم SiftDown برای یک گره در سطح r به وضوح حداکثر r+1 چرخش دارد (r بار پایین می آید و یک بار هم با خودش.) حال اگر ارتفاع این درخت Heap را $K = \lfloor \log_2^n \rfloor$ را در نظر بگیریم تعداد کل حرکت های لازم از فرمول زیر محاسبه میشود:

t: تعداد چرخش های حلقه Repeat در SiftDown در حلقه For.

$$t \leq 2 \times 2^{k-1} + 3 \times 2^{k-2} + \ldots + (k+1) \times 2^{0} \Rightarrow$$
$$t \leq -2^{k} + 2^{k} + 2 \times 2^{k-1} + 3 \times 2^{k-2} + \ldots + (k+1) \times 2^{0} \Rightarrow$$
$$t < -2^{k} + 2^{k+1}(2^{-1} + 2 \times 2^{-2} + 3 \times 2^{-3} + \ldots) \Rightarrow$$
$$t < -2^{k} + 2^{k+1} \sum_{n=1}^{\infty} n(\tfrac{1}{2})^{n} < -2^{k} + 2^{k+1} \times 2 \Rightarrow t < 2^{k}(2^{2} - 1) \Rightarrow$$
$$\Rightarrow t < 3 \times 2^{k} \Rightarrow t < 3n \Rightarrow t \in O(n) \qquad K = \lfloor \log_2^n \rfloor \Rightarrow 2^{k} \simeq n$$

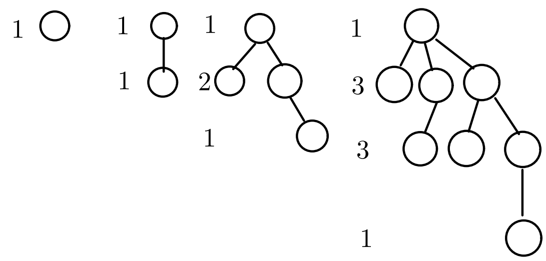
$$\frac{1}{1-x} = 1 + x + x^{2} + \ldots \quad \Rightarrow \frac{1}{(1-x)^{2}} = 1 + 2x + 3x^{2} + \ldots$$
$$\Rightarrow \frac{x}{(1-x)^{2}} = x + 2x^{2} + 3x^{3} + \ldots \qquad \Rightarrow \frac{x}{(1-x)^{2}} = \sum_{n=1}^{\infty} nx^{n}$$
$$\Rightarrow \sum_{n=1}^{\infty} n(\tfrac{1}{2})^{n} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^{2}} = 2$$

۴.۳ Binomial Heap (هیپ دوجمله‌ای)

یک Binomial Heap یک جنگل از درخت های دوجمله‌ای است.
(توجه : برای ساختن یک heap به ادغام دو heap نیاز داریم .که یکی n_1 گره ودیگری n_2 گره دارد.زمان ادغام این دو heap ($O(n_1 + n_2)$ می باشد.)

۱.۴.۳ Binomial Tree(درخت دوجمله ای)

این نوع درخت به صورت بازگشتی تعریف می شود.در این نوع به ریشه B_{n-1} خود B_{n-1} رابه عنوان فرزند راست اضافه می کنند.



B_0 B_1 B_2 B_3

$$B_n = \begin{cases} \bigcirc & n=0 \\ \text{به } B_{n-1} \text{ خود } B_{n-1} \text{ را اضافه می نماییم} & \end{cases}$$

این درخت،درخت دوجمله‌ای نامیده می شود؛ زیرا ضرایب بسط $(x+y)^n$ است.

- تعداد گره های B_n برابر است با 2^n .
- تعدادبرگ های B_n برابر است با 2^{n-1} .
- تعدادگره های داخلی B_n برابر 2^{n-1} است .
- تعداد گره های B_n در سطح k ام برابر است با $\binom{n}{k}$. (سطح یا عمق فرقی ندارند زیرا درخت متقارن است)
- درجه ریشه B_n نیزn است ، درجه این گره از همه‌گره‌های دیگر بیشتر است .

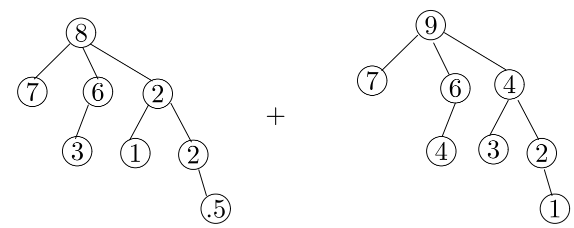
۲.۴.۳ Max Binomial Tree

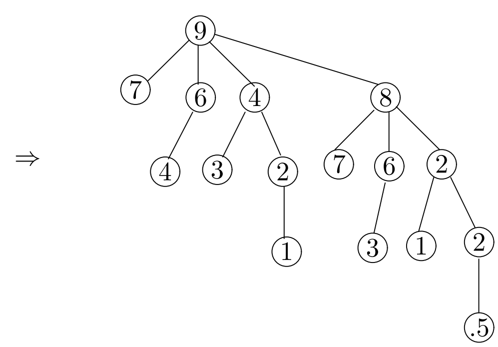
اگر هر گره یک درخت دوجمله‌ای با عددی برچسب خورده باشد،که عدد گرهٔ پدر کوچکتر از فرزندانش نباشدآن را یک Max Binamial Tree گویند.

۳.۴.۳ The Merge Of Max Binomial Trees

دو Max Binomial Tree B_n و Max Binomial Tree B'_n، B_{n+1} را می دهند .نحوه merge به شکلی است که‌بین هر دو درخت ، درختی که عدد برچسب ریشه آن کوچکتر است فرزند راست درخت دیگر خواهد شد.
عمل merge از$\theta(1)$ است . فقط مقایسه وجود دارد و عنصر min فرزند راست است.

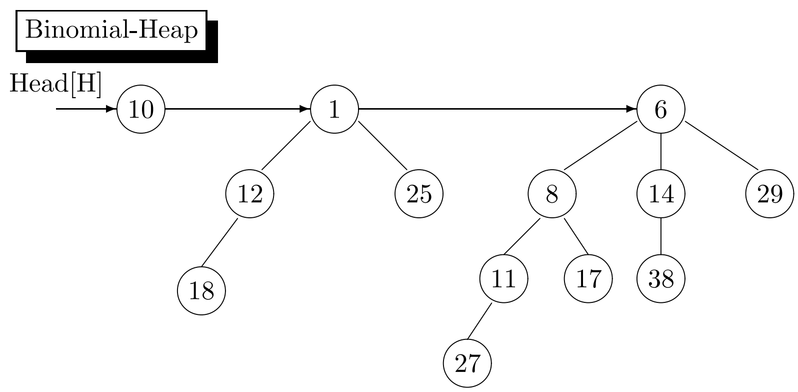
به مثال زیر دقت فرمایید:





۴.۴.۳ Binomial Heap

یک Binomial Heap، H مجموعه ای از Binomial Tree هاست .



در شکل بالا یک Min Binomial Heap را مشاهده می نمایید که خواص زیر را دارد:

۱. هر Binomial Tree خواص یک درخت Heap مینیمم را داراست، یعنی کلید هر Node بزرگتر مساوی کلید پدرش است.[۱]

۲. برای هر عدد صحیح نامنفی k یک Binomial Tree در H وجود دارد که درجه‌ی آن k است. این خاصیت بدین معناست که یک Binomial Heap با n، Node حداکثر از $\lfloor \lg n \rfloor + 1$، Binomial Tree تشکیل شده.

۵.۴.۳ عملیات بر روی Min Binomial Heap

- Make-Heap() : یک درخت Heap می سازد که خالی است و هیچ عضوی ندارد.
- Insert(H, x) : Node، x را به درخت H اضافه می کند.
- Minimum(H) : اشاره گری به Node ی از درخت H برمی گرداند که کمترین کلید را داشته باشد.
- Extract-Min(H) : Nodeی از Heap که کمترین مقدار را داشته باشد را Delete می کند و خود Node را برمی گرداند.

۱ توجه داشته باشید Heap هارا هم می توان به صورت Heap مینیمم تعریف نمود و هم به صورت Heap ماکزیمم در هرصورت عملیات بر روی آن ها تفاوت چندانی ندارد.

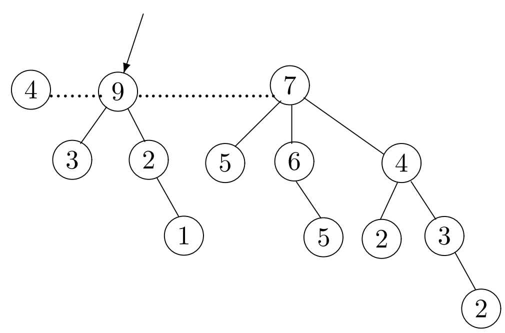
- $\text{Union}(H_۱, H_۲)$: درخت Heapی را برمی گرداند که شامل همه Node های درخت های Heap، $H_۱$ و $H_۲$ را از بین میبرد.

- Decrease-Key(H, x, k) : به Node، x از درخت H مقدار جدید k را می دهد که این مقدار جدید کمتر از مقدار قبلی آن است. $(x > k)$

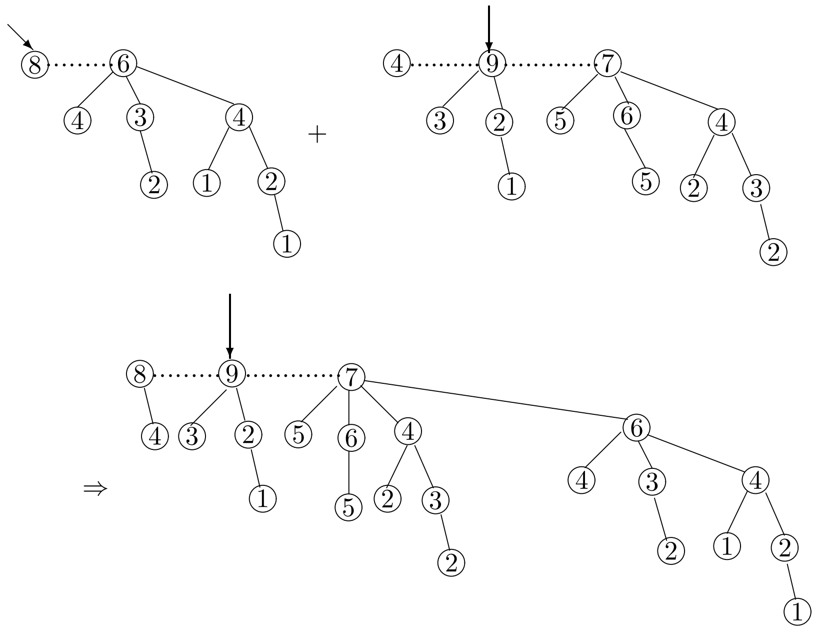
- Delete(H, x) : Node، x را از درخت H حذف می کند.

۶.۴.۳ Max Binomial Heap

یک Max Binomial Heap جنگلی از درختان دو جمله ای ماکزیمم است؛ که ریشه های تمام این درخت ها از طریق اشاره گر هایی به هم متصلند و به علاوه یک فلش آزاد به ریشه ای از درخت وجود دارد که عدد برچسب آن بین سایر ریشه های جنگل ماکزیمم باشد.مانند مثال زیر:



در زیر یک نمونه از عملیات merge دو Max Binomial Heap را مشاهده می نمایید. زمان این الگوریتم نیازا$\theta(۱)$ است،زیرا تعداد ریشه ها کم است.



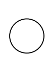
تمرین :توسط Binomial Heap هادو دیکشنری قدیم و جدید را ترکیب کنید.

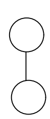
۵.۳ FIBONACCI HEAP

۱.۵.۳ Fibonacci Tree

تعریف:این نوع درخت به صورت بازگشتی تعریف می شود:

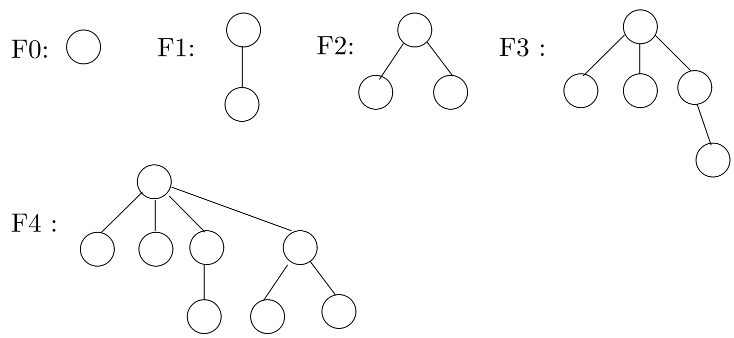
Fn =

 n=0

 n=1

به ریشه $F_{n-۱}, F_{n-۲}$ را به عنوان فرزند سمت راست اضافه می کنیم else

در این نوع درخت به ریشه F_{n-1}، F_{n-2} را به عنوان فرزند راست اضافه کنید.



۲.۵.۳ Max Fibonacci Tree

درخت فیبوناچی است که هر گره آن دارای برچسب است و برچسب پدر ازبرچسب فرزند هاکوچکتر نمی‌باشد.

اگر سری فیبوناچی را به این ترتیب شماره‌گذاری کنیم :

$f_0 = 0\ ,\ f_1 = 1\ ,\ f_2 = 1\ ,\ f_3 = 2, \ldots$

آنگاه تعداد گره‌های F_n برابر است با f_{n+2} و تعداد برگ های F_n برابر است با f_{n+1} . تعداد گره‌های داخلی F_n برابر است با f_n ($n \geq 0$) .

۳.۵.۳ Fibonacci Heap

یک Fibonacci Heap جنگلی از Fibonacci Tree هاست که ریشه های این درختان از طریق اشاره گری به هم متصلند .

۴.۵.۳ Max Fibonacci Heap

جنگلی از درختان Max Fibonacci Tree است که مطابق Max Binomial Heap است ولی عمل Merge کردن بین F_i و F_{i+1} صورت میگیرد و حاصل F_{i+2} است و در اینجا در Merge کردن آنها باید F_i را به عنوان فرزند راست F_{i+1} اضافه کرد. پس

از اضافه کردن دو حالت ممکن است اتفاق افتد، یا عدد ریشه F_{i+1} بزرگتر یا مساوی عدد ریشه F_i است که کار تمام شده است و یا در غیر اینصورت باید یک عمل SiftDown از ریشه درخت F_{i+1} به سمت F_i داشته باشیم . پس زمان به F_i بستگی دارد و حداکثر برابر لگاریتم ارتفاع درخت است.

افزایش برگ های درخت فیبوناچی نسبت به اندیس خیلی سریع است زیرا رشد تابع فیبوناچی زیاد است ولی ارتفاع به خوبی و آرامی رشد می کند و برابر است با $Height(F_n) = \lceil \frac{n}{2} \rceil$ در نتیجه عمل SiftDown بسیار هزینه ندارد و در حد $\lg^*$ است.
تمرین : برنامه ای بنویسید که درختان های فیبوناچی را بدهد.

۶.۳ درختان ۳—۲

تعریف [۲] :
یک درخت ۳—۲ یک درخت جستجو می باشد که دارای ویژگی های زیر می باشد.

- هر گره داخلی یا 2-node یا 3-node است.یک 2-node دارای یک عنصر و یک 3-node دارای دو عنصر می باشد.

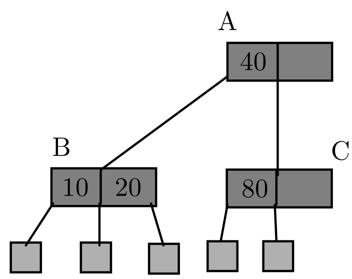
- فرض کنید LeftChild و MiddleChil نشان دهنده فرزندان یک 2-node باشند. همچنین فرض کنید dataL عنصر این گره و dataL.key کلید آن باشند. تمام عناصر در زیر درخت ۳—۲ با ریشه LeftChild دارای کلید کمتر از dataL.key هستند، در حالی که تمامی عناصر در زیر درخت ۳—۲ با ریشه MiddleChild دارای کلید بزرگتر از dataL.key می باشد.

- فرض کنید RightChild،MiddleChild، LeftChild نشان دهنده یک 3-node هستند. فرض کنید dataL,dataR دو عنصر این گره باشند.
آنگاه $\text{dataL.key} < \text{dataR.key}$ بوده و همچنین همه کلید ها در زیر درخت ۳—۲ با ریشه LeftChild کمتر ازdataL.key تمام کلید های در زیر درخت ۳—۲ با ریشه MiddleChild کمتر ازdataR.key و بزرگتر از dataL.key و تمام کلید ها در زیر درخت ۳—۲با ریشه RightChild بزرگتر از dataR.key هستند.

- تمام گره های خارجی در یک سطح قرار دارند.

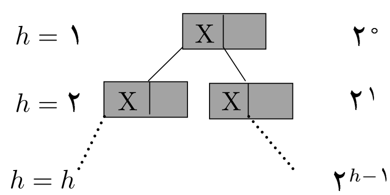
---
[۲] برگرفته از کتاب FUNDAMENTALS OF DATA STRUCTURES IN C++, Ellis Horowitz,Sartaj Sahni,Dinesh Mehta

مثالی از درخت ۳—۲ در شکل ۱ ارائه شده است.



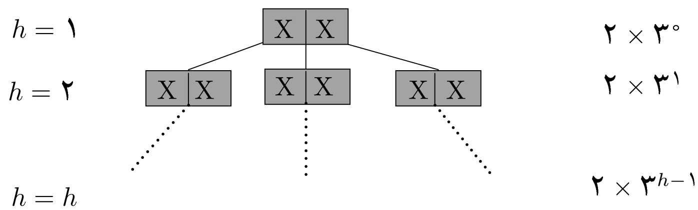
شکل ۱ مثالی از درخت ۳—۲

گره های خارجی به طور واقعی و فیزیکی در کامپیوتر نمایش داده نمی شوند. در عوض عضو داده فرزند متناظر هر گره خارجی برابر با صفر قرار می گیرد.
تعداد این عناصر در یک درخت ۳—۲ با ارتفاع h و n عنصر بین $۲^h - ۱$ و $۳^h - ۱$ است . برای مشاهده این مطلب ، توجه کنید که کرانه اول زمانی که هر گره داخلی در یک 2-node قرار می گیرد ، اعمال می گردد در حالی که کرانه دوم زمانی که هر گره داخلی در یک 3-node واقع می گردد ، اعمال خواهد شد.این دو وضعیت ، دو کرانه فوق را ارائه می کنند.یک درخت ۳—۲ دارای تعدادی 2-node و 3-node خواهد بود.



2-node

$$n = ۲^۰ + ۲^۱ + \cdots + ۲^{h-۱} \Rightarrow n = \frac{۲^h - ۱}{۲ - ۱} = ۲^h - ۱ \Rightarrow h = \left\lceil \log_۲^{n+۱} \right\rceil$$



3-node

$n = ۲ \times ۳^{۰} + ۲ \times ۳^{۱} + \cdots + ۲ \times ۳^{h-۱} \Rightarrow n = \frac{۲\times(۳^{h}-۱)}{۳-۱} = ۳^{h} - ۱ \Rightarrow h = \left\lceil \log_{۳}^{n+۱} \right\rceil$

بنابراین :

$\left\lceil \log_{۳}^{n+۱} \right\rceil < h < \left\lceil \log_{۲}^{n+۱} \right\rceil$

نمایش یک درخت ۳−۲ با استفاده از کلاس ها :

```
templete<class KeyType >class Two3;//forward declaration
templete< class KeyType >
class Two3Node{
friend class Two3 < KeyType >;
private :
    Element< KeyType > dataL,dataR;
    Two3Node *LeftChild, *MiddleChild, *RightChild;};
templete< class KeyType >
class Two3{
public:
    Two3(KeyType max,Two3Node< KeyType > *init=0)
      :MAXKEY(max), root(init){};//constractor
     Boolean Insert(const Element< KeyType > &);
```

```
    Boolean Delete (const Element< KeyType > &);
    Tow3Node< KeyType > *Search(const Element < KeyType >&);
private:
    Tow3Node< KeyType > *root;
    KeyType MAXKEY;
};
```

در اینجا فرض می کنیم که هیچ عنصر معتبری دارای کلید $MAXKEY$ نباشد و قرارداد می‌کنیم که یک $2-node$ دارای $dataR.key = MAXKEY$ است . تنها عنصر این گره در dataL ذخیره می‌شود و LeftChild و MiddleChild به دو فرزند آن اشاره می کنند. به عضو داده RightChild می توان هر مقدار دلخواه نسبت داد.

۱.۶.۳ جستجوی یک درخت ۳−۲

به سادگی می توان الگوریتم درختان جستجوی دودوئی را برای به دست آوردن تابع جستجوی Tow3::Search که گره حاوی عنصر با کلیدx را جستجو می کند تعمیم داد. تابع جستجو از تابعی به نام compare استفاده می کند که یک کلید x را با کلیدهای واقع در یک گره مشخص مانند p مقایسه می‌کند.این تابع به ترتیب مقادیر 1،2،3 یا4 را بسته به این که آیا x کمتر از کلید اول ، بین کلیدهای اول و دوم، بزرگتر از کلید دوم یا برابر با یکی از کلیدهای درp باشد ، برگشت می دهد.تعداد تکرار حلقه های for محدود به ارتفاع درخت۳−۲ می باشد . بنابراین اگر درخت دارای n گره باشد، آنگاه پیچیدگی تابع Tow3::Search برابر $O(\log n)$ خواهد بود.

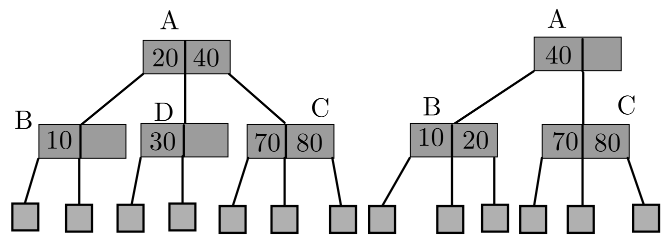
```
templete< class KeyType >
Tow3Node< KeyType > *Tow3< KeyType >::
Search(const Element< KeyType >& x)
//If the element x is not in the tree,then return 0,Otherwise
//return a pointer to the node that contains this element.
    {
        for(Tow3Node< KeyType > *p=root;p;)
```

```
switch(p→ compare(x)){
  case 1 :p=p→ LeftChild;break;
  case 2 :p=p→ MiddleChild;break;
  case 3 :p=p→RightChild;break;
  case 4 :return p ;// x is one of the keys in p
     }
  }
```

۳.۶.۲ درج به داخل یک درخت ۳—۲

درج به داخل یک درخت ۳—۲ نسبتاً ساده می باشد. برای مثال درج عنصر ۷۰ به داخل درخت ۳—۲ شکل ۱ را در نظر بگیرید.در ابتدا جستجوی لازم برای یافتن این کلید را انجام می‌دهیم. اگر کلید قبلاً در درخت باشد آنگاه بخاطر اینکه تمام کلیدها در درخت ۳—۲ منحصربه فرد هستند عمل درج با شکست روبرو شده و انجام نمی گیرد.در این مثال چون عنصر ۷۰ در درخت ۳—۲ وجود ندارد لذا آن را به داخل درخت درج می کنیم .برای این کار لازم است در خلال جستجوی عنصر ۷۰ ، بدانیم که با کدام گره روبرو می شویم . توجه داشته باشید هرگاه کلیدی را که در درخت ۳—۲ وجود ندارد مورد جستجو قرار دهیم ، جستجو با یک گره برگ منحصر به فرد روبرو خواهد شد. گره برگی که در طی جستجوی عنصر ۷۰ با آن مواجه خواهیم بود گره C با کلید ۸۰ است .از آنجا که این گره تنها دارای یک عنصر است ،عنصر جدید را می توان در این نقطه درج کرد. درخت حاصل درشکل ۲ قسمت (الف) نشان داده شده است .

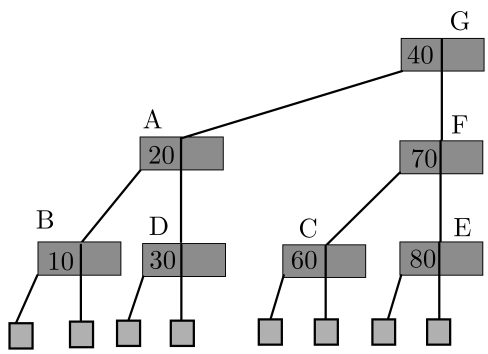
حال فرض کنید می خواهیم عنصر x با کلید ۳۰ در درخت درج کنیم . این بار جستجو با گره برگ B روبرو میشویم. از آنجا که B یک 3-node بوده لذا لازم است گره جدیدی به نام D را ایجاد کنیم . D شامل عنصری است که از بین دو عنصر موجود در B وx بزرگترین کلید را داراست. عنصر با کوچکترین کلید در B قرار خواهد داشت و عنصر با کلید متوسط همراه با اشاره گری بهD در گره پدر A از B درج خواهد شد.درخت حاصل در شکل ۲ قسمت(ب) ارائه شده است .



(ب)درج ۳۰ (الف)درج ۷۰

شکل ۲—درج به درون درخت ۳—۲شکل ۱

به عنوان آخرین مثال ، جایگذاری عنصرx با کلید ۶۰ را به داخل درخت ۳—۲ شکل ۲ قسمت (ب) در نظر بگیرید. گره برگ در طی جستجوی ۶۰ در گره C ملاقات می گردد.چون C یک 3-node است یک گره جدید E ایجاد می گردد. این گره حاوی عنصری با بزرگترین کلید است (۸۰). گرهC نیز حاوی عنصری با کوچکترین کلید خواهد بو(۶۰).عنصر با کلید متوسط (۷۰) همراه با اشاره گری به گره جدید E ، به داخل گره پدر A از C جایگذاری می گردد. مجدداً از آنجا که A یک 3-node است گره جدید F حاوی عنصری با بزرگترین کلید در بین ۷۰ ،۴۰، ۲۰ ایجاد می شود. مانند قبل ، A حاوی عنصری با کوچکترین کلید است . B و D به ترتیب به عنوان فرزندان چپ و وسط A باقی می ماند، به ترتیب C و E نیز به عنوان فرزندان F به داخل این گره پدر جایگذاری می شوند . چون A پدری ندارد ، یک گره جدید G برای درخت۳—۲ ایجاد می شود. این گره حاوی عنصری با کلید ۴۰ همراه با اشاره گر فرزند چپ به A و اشاره گر فرزند میانی به F خواهد بود، درخت ۳—۲ جدید در شکل ۳ ارائه شده است.



شکل۳—درج ۶۰به داخل درخت ۳—۲شکل ۲ قسمت (ب)

هر زمان که خواسته باشیم عنصری را به داخل یک 3-node مانند p اضافه کنیم، گره جدیدی مانند q را ایجاد خواهیم کرد . که به این کار تقسیم گره ها می گویند . می گوییم p به p ، q و عنصرمیانی تقسیم شده است . تابع درج به صورت زیر می باشد.

```
templete< class KeyType >
  Boolean Tow3< KeyType >::Insert(const Element< KeyType >& y)
  //Insert the element y into the 2-3 tree only if it does not already
  // contain an element with the same key.
  {
    Tow3Node< KeyType > *p;
    Element < KeyType > x=y;
    if(x.key>=MAXKEY) return FALSE; //invalid key
    if(!root){NewRoot(x,0); return TRUE;} //empty 2-3 tree
    if(!(p=FindNode(x))){
     insertionError();
     return FALSE;}//key already in 2-3 tree
    for(Tow3Node< KeyType > *a=0; ; )
      if(p→dataR.key==MAXKEY){// p is a 2-node
        p→PutIn(x,a);
        return TRUE; }
      else{// p is a 3-node
        Tow3Node< KeyType > *olda=a;
        a=new (Tow3Node< KeyType >);
        x=Split(p,x,olda,a);
        if(root==p){ //root has been split
          NewRoot(x,a);
          return TRUE; }
        else p=p→ parent();
      }//end of p is a 3-node and for loop
  }//end of Insert
```

این تابع از چندین تابع استفاده می کند. وظیفه ای که هر کدام از این توابع بر عهده دارند ، به صورت زیر می باشد:
(از پارامتر الگوی $< KeyType >$ برای سادگی صرف نظر شده است)

- void Tow3::NewRoot(const Element& x,Tow3Node *a) :

این تابع زمانی فراخوانی می شود که ریشه درخت ۳—۲ باید تغییر کند . تا زمانی که Middle Child، a را می سازد عنصر منفرد x به درون ریشه جدید (new root) درج می شود. ریشه قدیمی نیز تبدیل به LeftChild (فرزند چپ) ریشه جدید خواهد شد.

- Tow3Node* Tow3::Find(const Element & x):

این تابع نسخه اصلاح شده Tow3::Search (برنامه ۱) است. تابع فوق یک درخت ۳—۲ غیر تهی را برای حضور عنصری با کلید x.key مورد جستجو قرار می دهد. اگر این کلید وجود داشته باشد آنگاه مقدار صفر برگشت داده می شود. در غیر این صورت اشاره گری به گره برگ ملاقات شده در این جستجو را بر می گرداند. تابع Tow3::Insert از متغیر p برای برای ذخیره اشاره گری که توسط تابع FindNode برگشت داده می شود ، استفاده می کند . FindNode همچنین ساختمان داده ای را ایجاد می کند که ما را به بازگشت از p تا root (ریشه) قادر می سازد . این ساختمان داده ای می تواند لیستی از گره های موجود در مسیر root تا p باشد . به چنین ساختمان داده ای نیاز است زیرا پس از تقسیم یک گره ، دسترس به والد گره تقسیم شده ضروری است .

- void InsertionError():

وقتی بخواهیم عنصری را با کلیدش مساوی با عنصری از درخت است در آن درج کنیم خطایی رخ خواهد داد . این تابع خطا را اعلام خواهد کرد .

- void Tow3Node::PutIn(const Element& x,Tow3Node *a) :

از این تابع برای درج عنصر x به داخل گره(this) که واقعاً دارای یک عنصر است ، استفاده می کنیم . در زیر درخت a را دقیقاً در سمت راست x قرار می دهیم و در نتیجه اگر x برابر dataL گردد آنگاه a مساوی Middle Child شده مقادیر قبلی dataL و MiddleChild به ترتیب برابر dataR و RightChild می شوند . اگر x تبدیل به dataR گردد آنگاه a تبدیل به RightChild خواهد شد.

- Eelement& Tow3::Split(Tow3Node* p,Element& x,Tow3Node *olda,*a):

این تابع بر روی یک Tow3Node مانند (p) که ابتدا شامل دو عنصر به شرح زیر

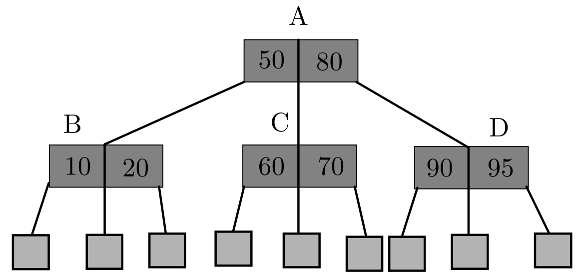
می باشد عمل می کند.گره خالی ایجاد شده که a به آن اشاره می کند حاوی عنصری با بزرگترین کلید از میان دو عنصر اولیه موجود در p و x می باشد. عنصر دارای کوچکترین کلید ، تنها عنصر باقی مانده در p می باشد. سه اشاره گر فرزندان اصلی p و اشاره گر olda چهار عضو داده فرزندی که باید در p و a تعریف شوند را اشغال می کنند . این تابع عنصر با کلید میانه را بر می گرداند .

در تابع Insert ، x نشان دهنده عنصری است که می خواهد در p درج شود و a نیز نشان دهنده گره ای است که به تازگی در آخرین تکرار حلقه for ایجاد شده است . برای تجزیه و تحلیل پیچیدگی مشاهده می کنیم که زمان کل صرف شده بستگی به عمق درخت ۳—۲ دارد . بنابراین جایگذاری به داخل یک درخت ۳—۲ با n عنصر در زمانی برابر با $O(\log n)$ صورت می گیرد.

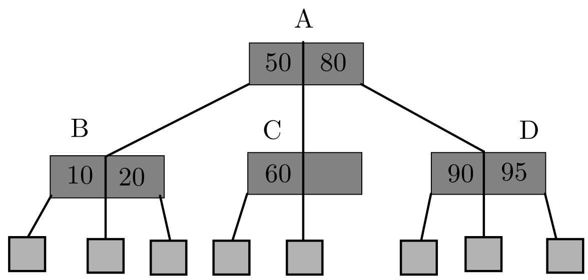
۳.۶.۳ حذف از یک درخت ۳—۲

حذف از یک درخت ۳—۲ به لحاظ مفهومی مشکل تر از جایگذاری نیست . اگر عنصری را حذف کنیم که گره برگ نباشد آنگاه با تبدیل گره حذف شده به یک عنصر مناسب که در گره برگ است و حذف برگ عمل حذف را انجام می دهیم . برای مثال اگر بخواهیم عنصر با کلید ۵۰ که در ریشه شکل ۴ قسمت (الف) قرار دارد را حذف کنیم آنگاه این عنصر ممکن است با عنصری حاوی کلید ۲۰ یا عنصری با کلید ۶۰ تعویض شود و هر دو اینها در گره برگ قرار دارند . در حالت کلی می توانیم از عنصر با بزرگترین کلید در زیر درخت واقع در سمت چپ و یا عنصر باکوچکترین کلید واقع در زیر درخت سمت راست عنصری که باید حذف شود، استفاده کنیم.

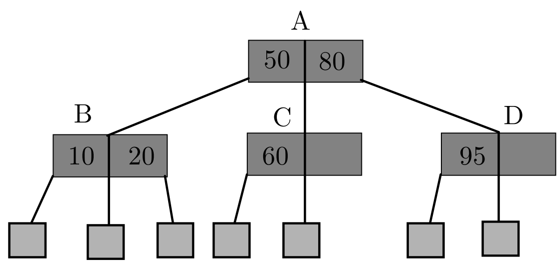
در نتیجه تنها حذف از یک گره برگ را در نظر می گیریم . این بحث را بر روی درخت شکل ۴ قسمت (الف) ادامه می دهیم . برای حذف عنصر با کلید ۷۰ فقط کافی است که در گره C ، MAXKEY را در dataR.key قرار می دهیم . نتیجه کار در شکل ۴ (ب) آورده شده است . برای خذف با کلید ۹۰ از درخت ۳—۲ شکل ۴ قسمت (ب) می بایست dataR را با dataL منتقل می کنیم و در گره D نیز dataR.key را با MAXKEY قرار می دهیم (dataR.key=MAXKEY) .نتیجه این کار در درخت ۳—۲ شکل ۴ قسمت(پ) ارئه شده است .



(الف)درخت ۳—۲اولیه



(ب) حذف گره ۷۰

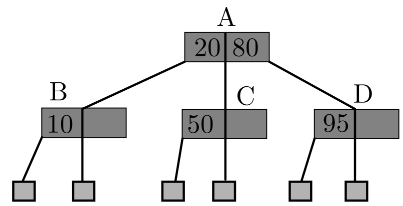


(پ)حذف گره ۹۰

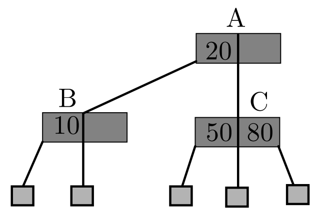
شکل ۴ ، قسمت های الف و ب و پ

حال حذف عنصر با کلید ۶۰ را در نظر بگیرید .این امر موجب می شود که گره C تهی گردد . از آنجا که همزاد چپ گره C یعنی B یک 3-node است ، از این رو می توانیم عنصر با کلید ۲۰ را به مکان dataL گره پدر، A ، انتقال داده و کلید ۵۰ را از پدر به گره C انتقال دهیم بعد از برقراری dataR.key=MAXKEY در B درخت ۳—۲ به صورت آنچه که در شکل ۴ قسمت (ت) ارائه شده، تبدیل می گردد . این

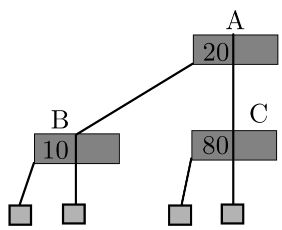
فرآیند انتقال و جابجایی داده ها چرخش (rotation) نامیده می شود . هنگامی که عنصربا کلید ۹۵ حذف می گردد، گره D تهی می شود . مانند هنگامی که حذف ٦۰ انجام شد rotation ممکن نمی باشد چرا که همزاد سمت چپ یعنی C یک 2-node است . این بار ۸۰ را به داخل همزاد چپ C منتقل کرده و گره D را حذف می کنیم . به این فرآیند ترکیب (combine) گفته می شود.در ترکیب یک گره حذف می شود در صورتی که در چرخش هیچگونه گره‌ای حذف نمی گردد. حذف عنصر ۹۵ منتهی به درخت ۳−۲ شکل ۴ قسمت (ث) می گردد. حذف عنصر با کلید ۵۰ از این درخت نیز موجب ارائه درخت ۳−۲ شکل ۴ قسمت (ج) خواهد شد. حال حذف عنصر با کلید ۱۰ را از این درخت در نظر بگیرید که موجب می شود تا گره B تهی گردد. اکنون بررسی می کنیم که آیا همزاد سمت راست B یعنی C یک 2-node است یا یک 3-node. اگر C یک 3-node باشد می توانیم چرخشی مشابه با آنچه که برای حذف ٦۰ صورت گرفت ،انجام دهیم و اگر یک 2-node باشد آنگاه یک عمل ترکیب صورت می گیرد. در اینجا چون C یک 2-node است ، مشابه وضعیت حذف ۹۵ عمل می کنیم. این بار عناصر با کلیدهای ۲۰ و ۸۰ به B منتقل شده و گره C حذف می گردد.هرچند این مسأله موجب می شد که پدر گره A دارای هیچگونه عنصری نباشد. اگر پدر ریشه نبود می توانستیم مانند آنچه که برای C انجام دادیم ، همزاد چپ و راست آن را مورد بررسی و تست قرار دهیم (حذف عنصر ٦۰)و در نتیجه D (حذف عنصر۹۵) تهی می گردید. چون A ریشه است ، به سادگی حذف شده و B ریشه جدید خواهد شد (شکل ۴ قسمت (چ)).



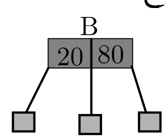
(ت)حذف ۶۰



(ث)حذف ۹۵



(ج)حذف ۵۰



(چ)حذف ۱۰

شکل ۴

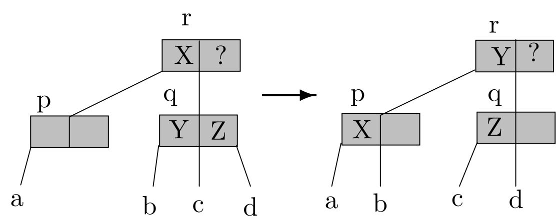
مراحل حذف ازگره برگ یک درخت ۳—۲

مرحله ۱: گره p را در صورت لزوم برای منعکس کردن وضعیتش پس از حذف عنصر مورد نظر تغییر دهید .

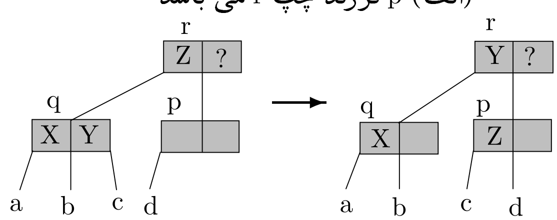
مرحله ۲:

```
for(;p has zero element && p!=root;p=r){
        left r be the parent of p and let q be the left or right sibling of p ;
        if(q is a 3-node) perform a rotation
        else perform a combine;}
```

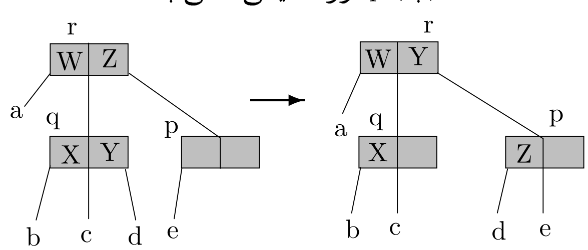
مرحله ۳: اگر p عنصری ندارد،آنگاه p باید ریشه باشد . فرزند سمت چپ p تبدیل به ریشه شده و گره p حذف می شود.
سه وضعیت برای یک چرخش بسته به این که p فرزند چپ ، راست و یا میانی پدر خود یعنی r باشد یاخیر به وجود می آید . اگر p فرزند سمت چپ r باشد، آنگاهq را به عنوان فرزند سمت چپ p فرض کنید . توجه داشته باشید که بدون در نظر گرفتن این که r یک 2-node یا3-node است ، q به درستی تعریف شده است . سه چرخش به صورت نموداری در شکل ۵ ارائه شده است . علامت ؟ نشان می دهد که عضو داده مربوطه زیاد اهمیتی ندارد .c،b،a و d نشان دهنده فرزندان گره ها هستند(یعنی ریشه های زیر درختان).



(الف) p فرزند چپ r می باشد



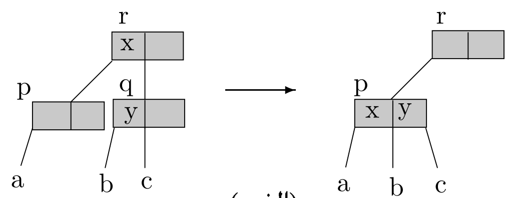
(ب) p فرزند میانی r می باشد



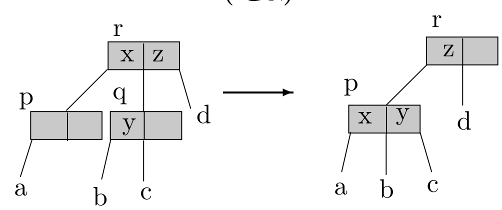
(پ) p فرزند راست r می باشد

شکل ۵—سه وضعیت مختلف برای چرخش در درخت ۳—۲

دو وضعیت عملیات combine وقتی که p فرزند سمت چپ r است در شکل ٦ نشان داده شده است .



(الف)



(ب)

شکل ٦

شبه کد مرحله‌ی اول از مراحل حذف یک گره برگ در درخت ۳—۲ به صورت زیر می باشد:

```
template < class KeyType >
Two3< KeyType >::DeleteKey(Two3Node< KeyType > *p,
const Element< KeyType >& x)
//Key x.Key is to be deleted from the leaf node p.
{
    if( x.Key==p→dataL.Key) // first element
      if(p→dataR.Key!=MAXKEY )
      { // p is a 3-node
        p →dataL = p → dataR;
        p → dataR.Key =MAXKEY;}
      else p →dataL.Key=MAXKEY; // p is a 2-node
    else p → dataR.Key =MAXKEY;} // delete second element
```

همچنین کد عملیات های Rotation وCombine وقتی p فرزند چپ r است به صورت زیر می باشد:

```
//Rotation when p is the left child of r and q is the middle child of r.
p → dataL= r → dataL;
p → MiddleChild = q → LeftChild;
r → dataL = q → dataL;
q → dataL = q → dataR;
q → LeftChild = q → MiddleChild;
q → MiddleChild = q → RightChild;
q → dataR.Key = MAXKEY;
..............................................................
//Combine when p is the left child of r and q is the right sibling of p.
p → dataL= r → dataL;
p → dataR= q → dataL;
p → MiddleChild = q → LeftChild;
q → RightChild = q → MiddleChild;
if(r → dataR.Key = MAXKEY)// r was a 2-node
      r → dataL.Key = MAXKEY;
else {
      r → dataL= r → dataR;
      r → dataR.Key = MAXKEY ;
      r → MiddleChild = r → RightChild;
      }
```

۴.۶.۳ تجزیه و تحلیل عملکرد حذف از یک درخت ۳—۲

مشخص است که یک عملکرد ترکیب یا چرخش به تنهایی در زمانی برابر با O(1) انجام می گیرد . اگر یک چرخش انجام شود، عمل حذف کامل می گردد .اگر یک ترکیب صورت گیرد، p در درخت ۳—۲ به یک سطح بالاتر منتقل می شود . بنابراین تعداد ترکیباتی که در خلال یک حذف می توانند صورت گیرند ازارتفاع درخت ۳—۲ تجاوز نخواهد کرد و در نتیجه عمل حذف از یک درخت ۳—۲ با n عنصر به زمانی برابربا $O(\log n)$ نیاز دارد.

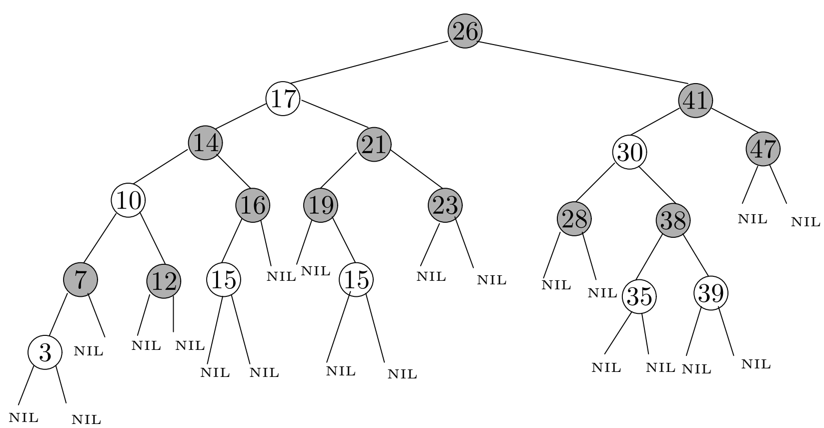
۷.۳ درخت قرمز – سیاه Red-Black

یک درخت قرمز–سیاه به صورت زیر می باشد:[۳]

۱.۷.۳ خواص درخت قرمز – سیاه

- هر گره یا قرمز و یا سیاه است .
- هر برگ nil سیاه است .
- دو فرزند یک گره قرمز ، سیاه هستند. (در یک گره قرمز،فرزند قرمز نمی تواند باشد)
- هر مسیر ساده از یک گره به برگ فرزند (نه لزوماً فرزند مستقیم) شامل تعداد یکسانی گره سیاه است .
- ریشه درخت سیاه است .(این شرط از شروط اساسی نمی باشد)

مانند مثال زیر:



[۳] مطالب این قسمت برگرفته از جزوه دکتر محمد قدسی می باشد.

۲.۷.۳ تعاریف و قضایای ابتدایی

Black-Height(x): بنابر تعریف Black-Height(x) یا bh(X) برابر است با تعداد گره های سیاه از x تا یک برگ فرزند[۴].

uncle(x) :

```
if parent[x]=right[parent[parent[x]]] then
        uncle[x]:=left[parent[parent[x]]]
else uncle[x]:=right[parent[parent[x]]]
```

قضیه ۱ : حداکثر ارتفاع یک درخت RB که دارای n گره داخلی است ، برابر $2\log_2^{(n+1)}$ است .

اثبات :برای اثبات قضیه فوق ابتدا نشان می دهیم که یک زیر درخت به ریشه دلخواه x حداقل دارای $2^{bh(x)}-1$ گره داخلی است . برای اثبات از استقرا بر روی ارتفاع گره داخلی x در درخت استفاده می شود.

پایه استقراء :

اگر ارتفاع x صفر باشد ، x حتماً برگ و در نتیجه nil است .بنابراین زیر درخت به ریشه x حداقل $2^{bh(x)}-1$ یعنی $2^0-1=0$ گره داخلی دارد.

گام استقرا ئی :

گره داخلی x را در نظر بگیرید . bh(x) عددی مثبت می باشد و x دارای دو فرزند می باشد .هر کدام از فرزندان بر حسب آنکه قرمز باشند یا سیاه ،Black - Height آنها برابر bh(x) یا bh(x)-1 خواهد بود . به علت آنکه هر فرزند x ارتفاعی کمتر از خود x دارد می توانیم با استفاده از فرض استقرا نتیجه بگیریم که هر زیر درخت به ریشه یک فرزند x حداقل $2^{bh(x)-1}-1$ گره داخلی دارد .بنابراین زیر درخت به ریشهٔ x حداقل $2^{bh(x)-1}-1+2^{bh(x)-1}-1+1=2^{bh(x)}-1$ گره داخلی خواهد داشت .

از طرف دیگر می دانیم که در درخت به ارتفاع h حداقل نیمی از گره ها (بدون در نظر داشتن ریشه) بر روی هر مسیر ساده از ریشه به برگ ، سیاه هستند . زیرا در غیر این صورت خاصیت ۳ از خواص درخت نقض خواهد شد .در نتیجه Black-Height ریشه حداقل $\frac{h}{2}$ خواهد بود.

$$n \geq 2^{\frac{h}{2}}-1 \Longrightarrow 2^{\frac{h}{2}} \leq n+1 \Longrightarrow h \leq 2\log_2^{(n+1)}$$

نتیجه : ارتفاع درخت RB، $O(\log n)$ است .

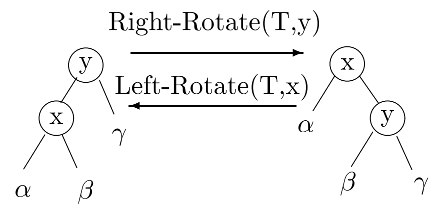
[۴] رنگ خود x بی تأثیر است و آنرا نمی شماریم .

۳.۷.۳ دوران

برای بازیابی خواص درخت RB بعد از عملیات درج و حذف در بعضی شرایط مجبور خواهیم شد که رنگ بعضی از گره ها را عوض کنیم یا تغییراتی در ساختار اشاره گر ها اعمال نماییم . به همین منظور دو عمل دوران (راستگرد و چپگرد) تعریف می نماییم . با توجه به شکل زیر واضح است که این دو عمل هیچ اشکالی در خواص یک درخت دودویی ایجاد نخواهد کرد .

نکته : هنگامی که یک دوران چپگرد انجام می شود فرض می نماییم که فرزند راست گره مورد نظر nil نباشد .

واضح است که دوران از $O(1)$ باشد زیرا تنها اشاره گر ها در اثر یک دوران تغییر می نمایند و سایر اجزا بدون تغییر می ماند .



Left-Rotate(T,x)

```
y ← right[x]              // SET y
right[x] ← left[y]        // Turn y's left subtree into x's right subtree
if left[y] ≠ NIL then
      p[left[y]] ← x
p[y] ← p[x]               // Link x's parent to y
if p[x] =Nil then
      root[T] ← y
      else if x= left[p[x]]then
            left[p[x]] ← y
            else right[p[x]] ←y
left[y] ← x               // Put x on y's left
p[x]← y
```

۴.۷.۳ درج

بر اساس درج در درخت دودوئی x را درج می نماییم برای این کار x را قرمز می نماییم ، واضح است که bh[5] برای تمام گره ها ثابت می ماند . اگر پدر x سیاه باشد درج به پایان می رسد اما اگر پدر x قرمز باشد به سراغ y=uncle[x] می رویم . در قسمت بعدی ایده اصلی اینست که در هر مرحله با حفظ خاصیت ۴ مربوط به درخت RB اشکال موجود که مربوط به عدم برقراری خاصیت ۳ می باشد را برای ارتفاع فعلی درخت حل نماییم و مشکل را به ارتفاعات بالاتر ارسال نماییم و این کار را برای سطوح بالاتر آنقدر ادامه دهیم تا به ریشه برسیم یا شرط سیاه بودن پدر x برقرار شود و به انتها برسیم . معلوم می باشد که درخت تنها زمانی که پدر x قرمز است به تصحیح نیاز دارد.

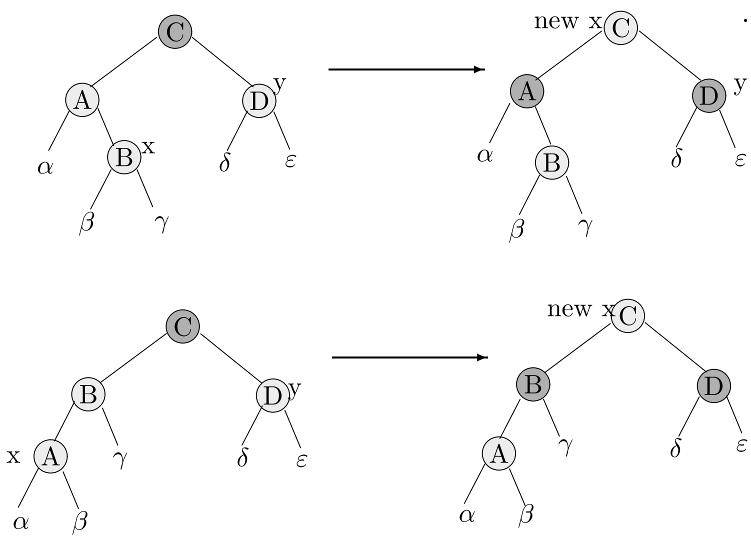
بر حسب آنکه Parent[x] فرزند چپ یا راست Parent[Parent[x]] است شش حالت متفاوت رخ می دهد که سه حالت دیگر با سه حالت زیر متقارن است[6]:

حالت اول : y قرمز است .

حالت دوم : y سیاه و x فرزند راست پدرش است .

حالت سوم : y سیاه و x فرزند چپ پدرش است .

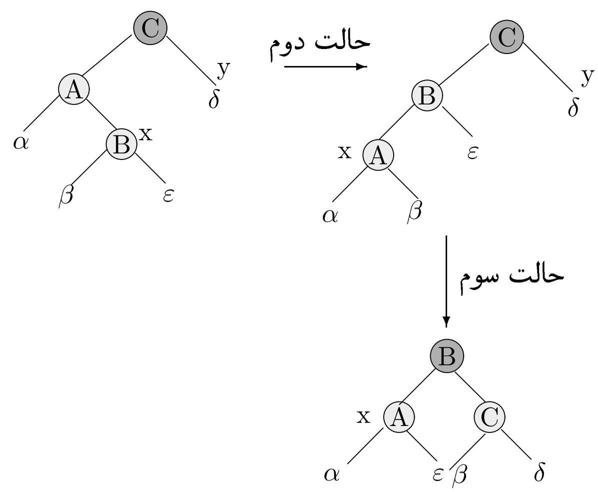
نکته : در پایان اجرای RB-insert برای حفظ خاصیت ۵ ریشه درخت سیاه می شود .



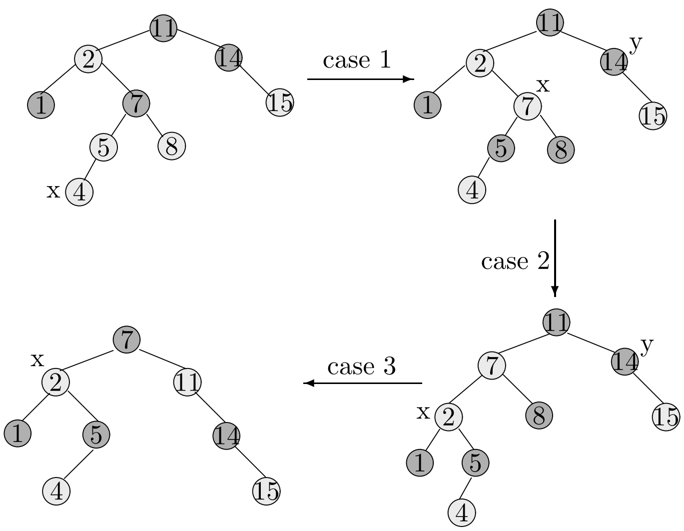
حالت اول

[5] Black-Height

[6] این حالات بر اساس رنگ y=uncle[x] تعیین می شود




```
RB-Insert(T,x)
    Tree-Insert(T,x)
    color[x]← RED
    while x≠root[T] and color[p[x]]=RED do
        if p[x] =left[p[p[x]]] then
            y ← right[p[p[x]]]
            if color[y] =RED then
                color[p[x]] ← BLACK          // Case 1
                color[y] ← BLACK             // Case 1
                color[p[p[x]]] ← RED         // Case 1
                x ← p[p[x]]                  // Case 1
             else if x = right[p[x]] then
                x ← p[x]                     // Case 2
                LEFT-Rotate(T,x)             // Case 2
            color[p[x]] ← BLACK              // Case 3
            color[p[x]] ← RED                // Case 3
            RIGHT-Rotate(T,p[p[x]])          // Case 3
        else(same as then clause with "right" and "left" exchanged)
    color[root[t]]←BLACK
```



درج یک گره

۵.۷.۳ حذف

برای ساده سازی شرایط مرزی در پیاده سازی RB-Delete یک گره حفاظتی به نام nil[T] برای درخت T تعریف می نماییم . این گره حاوی همان مولفه هایی است که در گره های معمولی درخت وجود دارد . مولفه رنگ این شیء سیاه می باشد و سایر مولفه ها می توانند مقادیر دلخواه بگیرند .

با استفاده ازین شی ء محافظتی می توانیم با یک فرزند nil مانند یک گره معمولی درخت که پدرش x می باشد رفتار نماییم (استفاده مولفه Parent مربوط به این شی ء محافظتی در فرا خوانی RB-Delete-Fixup و اولین اجرای حلقه while می باشد .)

بنابراین باید در درخت RB تمام اشاره گرهای nil را به nil[T] اشاره دهیم (در این مرحله ، مولفه Parent متناظر با این شی ء هنوز مقدار خاص و قابل استفاده‌ای در خود ندارد.)

برای حذف یک عنصر ابتدا به روشی مانند حذف از درخت دودوئی جستجو ، عنصر مورد نظر را حذف و سپس درخت را تصحیح می نماییم .باید دقت نماییم که شبه دستورات مربوط به حذف از درخت RB با همتای خود در درخت جستجوی دودوئی دقیقاً یکسان نیست چون در درخت RB اشاره گرهای nil به nil[T] اشاره می

نماید و همچنین مولفه Parent[x] در خط ۷ بدون هیچ شرطی مقدار دهی می شود . زیرا در صورتی هم که x یک برگ nil باشد، nil[T].Parent مقدار خواهد گرفت . علاوه بر دو تغییر ذکر شده ، در صورتی که رنگ y سیاه باشد ، درخت در خطوط ۱۶ و ۱۷ با فراخوانی RB-Delete-Fixup تصحیح می شود تا خاصیت ۴ در درخت RB بازیابی شود .

نکته : اگر RB-Delete با گره z بر روی درخت T فراخوانی شود یا خود آن گره حذف می شود یا عنصر بعدی (Successor) آن حذف می شود . در حالت دوم دستور Key[z]:=Key[Succ(z)] نیز اجرا خواهد شد .

قرارداد : گره حذف شده را y می نامیم .

اگر y سیاه باشد ، حذف از آن موجب می شود مسیری که در گذشته از y عبور می کرده ، یک گره سیاه کمتر از سایر مسیر ها داشته باشد و این باعث از بین رفتن خاصیت ۴ در درخت می شود . ما می توانیم این مشکل را با فرض اینکه بتوانیم گره x (که در حقیقت فرزند چپ یا راست گره y قبل از حذف شدن y بوده است) را یک بار اضافه تر سیاه کنیم ، حل نماییم . یعنی عبور از گره x به معنای عبور از دو گره سیاه باشد . بنابراین زمانی که گره y را حذف می کنیم رنگ آن را به فرزندش یعنی x می فرستیم . تنها مشکل این است که امکان دارد x دوبار سیاه شده باشد . (خودش از قبل سیاه باشد) و این باعث از بین رفتن خاصیت ۱ در درخت می شود .

هدف ما این است که یک رنگ سیاه اضافی را به طرف ارتفاعات بالاتر در درخت بفرستیم تا به یکی از حالت های زیر برسیم :

الف) x به یک گره قرمز اشاره نماید که در این حالت ما آن را سیاه می نماییم .

ب) x به ریشه اشاره کند که در این حالت ، سیاه اضافی را می توان دور انداخت .

پ) با دوران های مناسب و رنگ آمیزی مجدد بتوان خاصیت ۱ را احیا نمود .

اگر در یک مرحله موفق نشویم که به یکی از سه وضعیت بالا برسیم ممکن است با ۴ حالت متفاوت رو به رو شویم (در حقیقت بر حسب اینکه x فرزند چپ یا راست باشد ۸ وضعیت دو به دو متقارن به وجود می آید .) قرارداد ها :

- در هر مرحله x گره ای است که دو بر چسب سیاه دارد .
- w=sibling(x) خواهد بود .
- گره های تیره رنگ سیاه ، گره های خاکستری قرمز و گره های خاکستری روشن نمایا نگر گره با رنگ نامعلوم است مانند c',c .

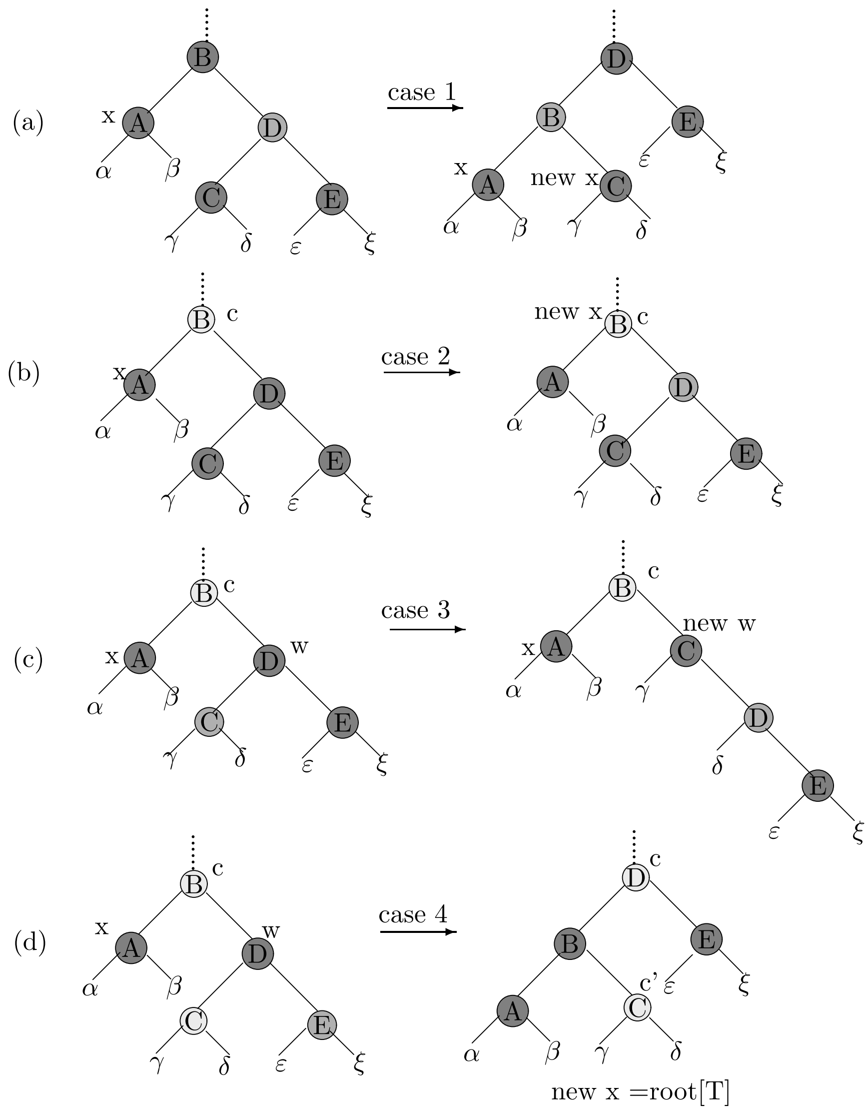
حالت هایی که ممکن است به وجود آید به صورت زیر است :

۱)w قرمز است (بنابراین فرزندان سیاه دارد)

۲)w و دو فرزندش سیاه است .

۳)w سیاه و فرزند چپ آن قرمز و فرزند راست آن سیاه است .

۴)w سیاه و فرزند راست آن قرمز است .



۱) حالت اول : حالت ۱ با تعویض رنگ گره های D و B با یکدیگر و انجام دوران چپ گرد به یکی از حالت های ۲ و یا ۳ یا ۴ تبدیل می شود .

۲) حالت دوم : رنگ سیاه اضافی که با اشاره گر x نمایش داده شده است با قرمز کردن D و اشاره دادن x به B به یک ارتفاع بالاتر منتقل می شود . اگر از طریق حالت اول وارد حالت ۲ شده باشیم چون c قرمز است حلقه به پایان خواهد رسید .

۳) حالت سوم : حالت ۳ با تعویض رنگ گره های C و D و انجام یک دوران راستگرد به حالت ۴ تبدیل می شود .

۴) در حالت ۴ رنگ سیاه اضافی که با x نمایش داده شده است را می توان با عوض کردن چند رنگ و یک دوران چپ گرد بدون آنکه به خواص درخت لطمه زد حذف نمود و حلقه در این مر حله به پایان می رسد .

زمان این الگوریتم هم از $O(\lg n)$ است .

```
RB-Delete(T,z)
if left[z]=nil[T] or right[z]=nil[T] then
     y ← z
     else y ← Tree-Successor(z)
if left[y] ≠ nil[T] then
     x ← left[y]
     else x ← right[y]
p[x] ← p[y]
if p[y] = nil[t] then
     root[T] ← x
     else if y = left [p[y]] then
          left[p[y]] ← x
          else right[p[y]]← x
if y ≠ z then
     key[z]← key[y]
          // if y has other fields , copy them , too
if color[y] = BLACK then
```

```
      RB-Delete-Fixup(T,x)
return y
```

RB-Delete-Fixup(T,x)

```
while x ≠ root[T] and color[x] =BLACK do
  if( x = left[p[x]]) then
      w ← right[p[x]]
      if( color[w]=RED) then
         color[w] ← BLACK                                   //case1
         color[p[x]] ← RED                                  //case1
         LEFT-Rotate(T,p[x])                                //case1
         w ← right[p[x]]                                    //case1
      if(color[left[w]]=BLACk and color[right[w]]=BLACK) then//case1
         color[w]=RED                                       //case2
         x ← p[x]                                           //case2
     else if color[right[w]]=BLACK then
         color[left[w]] = BLACK                             //case3
         color[w] ← RED                                     //case3
         RIGHT-Rotate(T,w)                                  //case3
         w←right[p[x]]                                      //case3
       color[w]←color[p[x]]                                 //case4
       color[p[x]]←BLACK                                    //case4
       color[right[w]] ← BLACK                              //case4
       LEFT-Rotate(T,p[x])                                  //case4
       x←root[T]                                            //case4
   else(same as then clause with "right" and "left" exchanged)
color[x]← BLACK
```

۸.۳ مجموعه های مجزا (Disjoin sets)

دو مجموعه AوB را هرگاه اشتراک آن دو تهی باشد مجزا می نامند.ساده‌ترین نوع داده‌ای برای پیاده سازی مجموعه n عضوی که باشماره های ۱و۲و... وn برچسب خورده‌انداستفاده‌ازآرایه‌است.

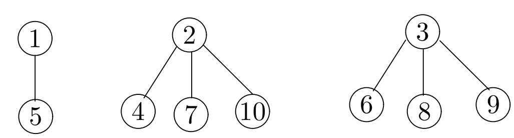
بایک آرایه به سادگی می توان عناصرمجموعه ۱و۲و...وn را که به دسته های مجزا افراز شده‌اندنمایش داد. برای این منظور هر زیر مجموعه را با نام عنصر می نیمم آن مشخص نموده و علاوه بر آن عنصر پدر را برابر با همین عنصر می نیمم قرار می دهیم .در ابتدا نیز آرایه set رابا مقادیراندیس های آن مقداردهی می نماییم .

set={1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 }

set

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

1={1 , 5 } 2={2 , 4 , 7 , 10 } 3={3 , 6 , 8 , 9 }



سپس برای هر عنصر عدد گره پدرش را در آرایه set قرار می دهیم .

set

1	2	3	2	1	3	2	3	3	2
1	2	3	4	5	6	7	8	9	10

```
function find1(x)
   { find the lable of the set containing x}
   return set[x]
```

تابع find1 برچسب مجموعه‌ای که x در آن قرار دارد رابر می گرداند .این الگوریتم از $\theta(1)$ است .

تابع merge1 دو مجموعه‌ای که شامل a وb هستند را با هم ادغام می نماید.زمان این تابع از $\theta(n)$ است .

```
function merge1(a,b)
  {merge the sets labled a and b }
        i←min(a,b)
        j←max(a,b)
    fork←1 to n do
      if set[k] = j then set[k]← i
```

حال دو مجموعه را به صورت دیگری ترکیب می نماییم و توابع find2 و merge2 رابه صورت زیر تعریف می کنیم :

```
function find2(x)
    r ←x
    while(set[r]≠ r) do
          r←set[r]
    return r
```

```
function merge2(a,b)
    if(a<b) then set[b]← a
    else set[a]← b
```

تابع find2از O(n) است زیرادر بدترین حالت ارتفاع حداکثرخطی‌است و تابع merge2 از $\theta(۱)$ است که نسبت به الگوریتم قبلی بهینه تر است .
در دو درخت قبلی کلید ریشه کلیدی است که برچسب آن کوچکتر است در الگوریتم دوم find دارای زمان بیشتری است پس باید روی ارتفاع کار کرده‌واز بزرگ شدن آن جلوگیری کرد.

برای این منظور برای هر گره از درخت ارتفاعی در نظر می گیریم و اولویت را ارتفاع قرار می‌دهیم نه کوچک بودن کلید .

در این صورت توابع merge3 و find3 را به صورت زیر تعریف می نماییم :

```
function merge3(a,b)
    if(height[a] = height[b])
```

```
            set[b]← a
            height[a] ++
            r← a
    else
    if(height[a]>height[b]) then
            set[b]← a
             r ← a
    else
             set[a] ← b
             r ← b
    return r

function find3(x)
         r ← x
        while(set[r]≠r) do
             r← set[r]
        i←x
         while i≠ r do
             j ← set[i]
             set[i] ← r
             i ← j
        return r
```

زمان این الگوریتم‌از $\log n$ نیز کمتراست.

فصل ۴

معرفی روش های مختلف الگوریتم نویسی

۴. انواع روش های برنامه نویسی

از انواع تکنیک های برنامه نویسی می توان به موارد زیر اشاره نمود:

- تکنیک حریصانه
- تکنیک تقسیم و حل
- تکنیک برنامه‌نویسی پویا
- تکنیک بازگشت به عقب
- تکنیک شاخه و حد
- تکنیک تورنمنت بازی ها
- الگوریتم های احتمالی
- الگوریتم های موازی

۱.۴ الگوریتم های حریصانه (Greedy Algorithms)

این الگوریتم ها اولین دسته ودر واقع ساده ترین روش های الگوریتم نویسی هستند. در این روش برای حل همواره سعی می شود راحت ترین و در عین حال پر ارزش ترین راه حل در هر گام انتخاب شود.تکنیک حریصانه در واقع یک روش برای حل مسائل بهینه سازی است. اما نکته حائز اهمیت در این روش آنست که در این روش هرگز دغدغه این موضوع وجود ندارد که انتخاب پر ارزش ترین عنصر ممکن است به جای آنکه به‌یک حل منجر شود به بیراهه و یا به بن بست منتهی شود.

ساختار یک الگوریتم حریصانه مطابق Control Abstraction زیراست:

```
Fuction Greedy(C):set //solution set
{ C is a set of condidates}
S ← ∅ { S ⊆ C is a set of solution ,if it is possible }
{ Greedy loop}
While (C ≠ ∅ & not Solutions(S))
    x ← Select(C)
    C ← C \ { x }
     if feasible ( S ∪ { x }) then
       S ← S ∪ { x }
if (solution(S)) then return S
else    " there are no solutions."
```

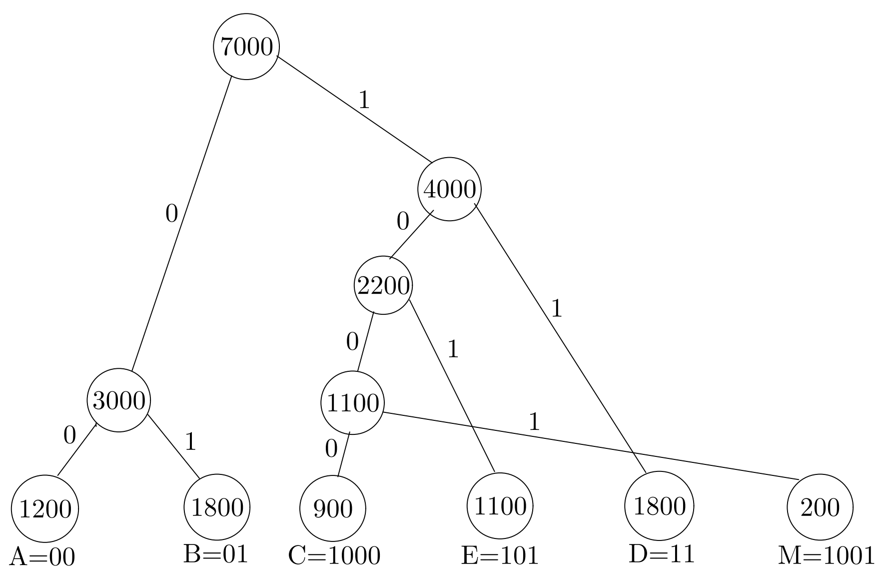
S درابتداتهی است در صورت وجود جواب به مجموعه اضافه می شود هر بار که تابع Select یک عنصر از مجموعه C را انتخاب می کند قاعدتاً از مجموعه C حذف می شود. وظیفه تابع feasible اینست که آیا اضافه کردن عنصر از مجموعه Cبه مجموعه S ممکن است یا خیر.بسته به نوع مسأله توابع Select و feasible , solution قابل تغییر هستند .

۱.۱.۴ الگوریتم فشرده سازی هافمن

یک روش فشرده سازی فایل ها روش هافمن است . دراین روش برنامه ورودی نویسه به نویسه دریافت می شود سپس فرکانس هر یک از نویسه های به کار رفته در متن محاسبه می گرددو آن را در جدولی قرار می دهیم با استفاده از فراوانی نویسه ها یک درخت دودوئی به شرح زیر بنا می کنیم:

به ازای هر نویسه یک برگ از درخت را می سازیم ،سپس دو گره ای که دارای فرکانس کمتری هستند را با هم ترکیب کرده و برای آن دو گره یک گره پدر می سازیم که فرکانس این گره پدر برابر مجموع فرکانس های دو گره سازنده آن است. بین دو گره اخیر و گره هایی که تا بحال به بازی گرفته نشده اند این روند را ادامه می دهیم تا ریشه‌درخت به دست آید.فرکانس ریشه باید با مجموع فرکانس های برگ ها برابر باشد، حال که درخت ساخته شد برای هر یال درخت یک برچسب صفر یا یک در نظر می گیریم .هریال سمت راست را برچسب یک و هر یال سمت چپ را بر چسب صفر می زنیم. بدین ترتیب در طی مسیری که از سمت ریشه به سمت برگ ها حرکت می کنیم ،یک ریشه باینری متناظر با برگ بدست خواهد آمد که آن رشته کد متناظر باآن برگ است .

برای مثال فرض کنید متنی شامل ۷۰۰۰حرف الفبایی در اختیار داریم از این تعداد ۱۸۰۰ حرف B ،۱۲۰۰ حرفA ،۹۰۰ حرفC، ۱۸۰۰ حرفD ،۱۱۰۰ حرفE و۲۰۰ حرفM داریم .می خواهیم حداقل تعداد بیت های لازم برای فشرده سازی‌این متن را به دست آوریم .



عنصری که فرکانس بالاتری دارد ، کد کوچکتری دارد.بدین ترتیب می توان با مقدار کمتری از حافظه ، کل اطلاعات را نمایش داد. در این مثال هر کاراکتر در حالت عادی ۸بیت از حافظه را اشغال می نماید،در حالی که پس از فشرده سازی اگر به جای هر کاراکتر کد دودویی متناظرش را قرار دهیم هر کاراکتر حداکثر۴ بیت را اشغال می نماید.کل بیت های مصرف شده در این مثال به صورت زیر به دست می آید:

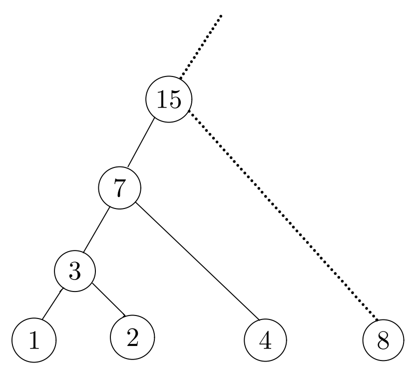
$2 \times 1200 + 2 \times 1800 + 4 \times 900 + 3 \times 1100 + 2 \times 1800 + 4 \times 200 = 17300$

که بسیار کمتر از حالت عادی آن یعنی $7000 \times 8 = 56000$ می باشد.

به عنوان مثال می خواهیم رشته DBAD را فشرده نماییم ،برای این کاربه جای هرکاراکتر معادل آن را قرارداده و کد ۱۱۰۱۰۰۱۱ به دست می آید که معادل یک کاراکتر مثلاً X است ویا به عکس کد۰۱۰۰۱۰۰۱۱۰۱ را داریم می خواهیم بدانیم این کد معادل چه رشته‌ای می باشد، برای این کار از سمت چپ به راست کد باینری را خوانده و همزمان از ریشه درخت آن را دنبال کرده تا به کاراکتر مورد نظر برسیم .بدین ترتیب رشته‌ی BAME به‌دست می آید.

باید توجه داشت که هر کدام از ین کد ها به صورت منحصر به فرد به دست می آیند و هر کد کوچکتر پیشوندی از کد بزرگتر نمی باشد.

فرض کنید که n کاراکتر را می خواهیم به روش هافمن کد کنیم در این حالت در بد ترین زمان طول حداکثر نویسه $n-1$ خواهد بود . بزرگترین حالت زمانی رخ می دهد که برای عنصر i ام فراوانی آن بزرگتر از مجموع فراوانی از عنصر ۱ تا $i-1$ ام است .



۲.۱.۴ الگوریتم های درخت پوشای مینیمال MST

فرض کنید G=<N,A> یک گراف همبندغیر جهت دار ساده باشدکه مجموعه رئوس آن N ومجموعه یالهای آن Aباشد.منظورازیک درخت‌پوشاازاین گراف درختی است که شامل همه رئوس گراف بوده ولی در عین حال شامل برخی از یالهای این گراف است بطوریکه ساختار بدست آمده خود درخت باشد(همبندوفاقد دور).

قضیه :

T=<V,E> درخت است اگروتنهااگر هر یک از گزاره های زیر به تنهایی برقرار باشد:

۱. T فاقد دور باشد، p=q+1 (p تعداد رئوس ، q تعداد یال ها).

۲. T همبند وp=q+1 .

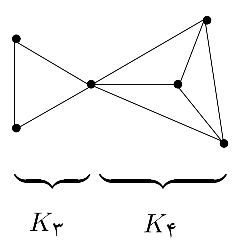
۳. بین هردو راس متمایز دقیقایک مسیر وجود داشته باشد.

۴. همبند و فاقد دور باشد.

با p نقطه در صفحه $p^{p-۲}$ درخت نشانه دار(درختی که هر یال آن بر چسب دارد)می توان رسم کرد.

مثال :

گراف زیر چند درخت پوشا دارد؟(راهنمائی: درخت کامل Kp دارای $p^{p-۲}$ درخت پوشا است.)



حل:

$$۳^{۳-۲} \times ۴^{۴-۲} = ۴۸$$

حال اگر گراف G با شرایط مذکور گراف وزن دار باشدکه هر یک از یالهای آن با عددی نامنفی بر چسب خورده باشد می توان یک درخت پوشای می نیمم از این گراف استخراج کردمنظور از یک درخت پوشای مینیمم (minimum spaning tree)

درختی است که‌شامل همه رئوس گراف وبرخی از یالهای آن است بطوریکه مجموع وزن یالهای آن در بین سایر درخت های پوشا کمینه است .

توجه :هر گراف همبند وزن دار غیر جهت دار ساده حتماً دارای یک درخت پوشای مینیمم است که لزوما می تواند منحصر به فرد نباشد یعنی اینکه میتواند بیش از این درخت پوشا بدست آید.
به طور کلی چند الگوریتم برای یافتن درخت پوشای مینیمم وجود دارد :
Prim , Kruskal, Boruvka ,Sollin

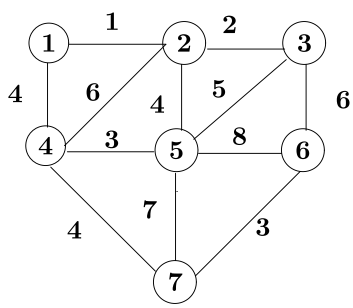
الگوریتم kruskal:

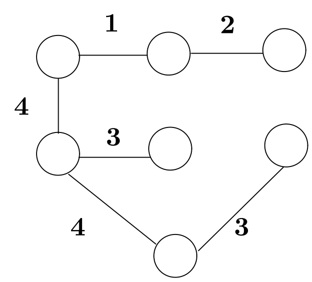
این الگوریتم به شیوه زیر یک درخت پوشای مینیمال را می سازد:
ابتدا یالهای گراف را به ترتیب صعودی مرتب میکند سپس n مجموعه به تعداد عناصر مجموعه رئوس گراف می سازد(فرض بر این است که عناصر مجموعه از 1تا n برچسب خورده‌اند)حال حلقه greedy آن قدرتکرار میشود تا n-1 یال انتخاب شوند .یالی انتخاب می شود که کمترین وزن ممکن را داشته باشد سپس تابع find برای دو سراین یال فراخوانی میشودیعنی اگر e=uv باشد find (u) ,find(v) محاسبه می شوند.اگر حاصل این دو برابر نباشد یال مذکور به مجموعه یالهای گراف افزوده میشود هم چنین دو مولفه ای که از find(v),find(u) به دست آمده اند با هم ترکیب می شونداین روند آنقدر ادامه می یابد تا T شامل n-1 یال شود.

```
function kruskal(G=< N,A > : graph,length :A → R+): set of edge
        sort A by increasing length
         initialize n sets , each set containing an element of N
        T← ∅
        {greedy loop}
        repeat
            e←{u,v}
            A← A \ {e}
            ucomp←find (u)
            vcomp←find (v)
```

```
        if(vcomp ≠ ucomp) then
          T←T ∪{e}
          merge(ucomp,vcomp)
      until T is containing n-1 elements
  return T
```

به عنوان مثال گراف زیر را در نظر بگیرید:





step	T	connected components
initialization	-	{1}{2}{3}{4}{5}{6}{7}
1	{1,2}	{1,2}{3}{4}{5}{6}{7}
2	{2,3}	{1,2,3}{4}{5}{6}{7}
3	{4,5}	{1,2,3}{4,5}{6}{7}
4	{6,7}	{1,2,3}{4,5}{6,7}
5	{1,4}	{1,2,3,4,5}{6,7}
6	{2,5}	reject
7	{4,7}	{1,2,3,4,5,6,7}

در این جا زمان merge,find به‌اندازه $\log n$ است ، پس باn بار فراخوانی داریم $n \log n$.
$a = |A|$ تعدادیالهاست و n تعداد رئوس .از طرفی داریم :

$(n-۱) \leq a \leq \frac{n(n-۱)}{۲}$

$\Rightarrow \log(n-۱) \leq \log a \leq \log n + \log(n-۱) - \log ۲$

$\Rightarrow \log n \leq \log a \leq ۲ \log n$

پس $\log a$ هم درجه $\log n$است پس زمان sort از$O(a \log a)$ خواهد بود.

الگوریتم Prim :

این الگوریتم به شیوه زیر درخت پوشا را می سازد:
ابتدا مجموعه ای به نام B را که شامل یکی از رئوس گراف G است در نظرمی‌گیرد سپس بین تمام یال های گراف که یک سر آنها در B است و سر دیگر آن در B نیست ، یالی را می یابد که کمترین بر چسب را داشته باشد،سپس رأس دیگر را به B اضافه می کند.این روند را آنقدر ادامه میدهد تا B برابر مجموعه N یعنی رئوس گراف شود.
اگر این الگوریتم را با ساختمان داده آرایه های دو بعدی (ماتریسی) پیاده سازی نماییم زمان لازم در حد $O(n^2)$ است . اما اگر با یک binary heap پیاده سازی کنیم زمان اجرا$O((a+n)logn)$ است و اگر با fibonacci heap پیاده سازی نماییم زمان اجرا $O(a+nlogn)$ است .پس تغییرپیاده سازی و ساختمان داده ای زمان اجرا را تغییر می دهد.

```
function Prim(G=< N, A >:graph,length:A ⟶ R⁺):set of edges
    B ⟵ {an arbitary element of N}
    T ⟵ ∅
    while B ≠ N do
        e ⟵ { u,v} such that e is minimum and u∈ B,v∈ N\B
        T ⟵ {e} ∪ T
        B ⟵ B ∪ {v}
```

پیاده سازی از طریق ماتریس مجاورت :

```
function Prim(L[1..n,1..n]):set of edges
    1. {initialization : only node 1 is in B}
    2.T ⟵ φ {will contain the edges of the minimum spanning tree}
    3. for i=2 to n do
    4.       nearest[i] ⟵ 1
    5.       mindist[i] ⟵ L[i,1]
    6. {greedy loop}
    7. repeat n-1 times
    8.       min ← ∞
    9.       for j ←2 to n do
    10.          if 0≤ mindist[j] < min Then
```

```
11.            min ← mindist[j]
12.            k ← j
13.      T=T ∪ {nearest[k],k}
14.      mindist[k] ← -1{add k to B}
15.      for j ← 2 to n do
16.        if L[j,k] < mindist[j] Then
17.            mindist[j] ←L[j,k]
18.            nearest[j] ←k Return T
```

پیاده سازی الگوریتم PRIM با heap :

```
HEAP-PRIM(G)
1. A ⟵ φ
2. for each x ∈ V do key[x] ⟵ ∞
3. H ⟵ Build-heap(key)
4. select arbitrary vertex v
5. S ⟵ {v}
6. Decrease-key(H,v,0)
7. Extract-min(H)
8. for each x adjacent to v do
     decrease-key(H,x,w(v,x)), π [x] ⟵ v
9. for i=1 to |V| - 1 do
10.      begin
11.        Extract-min(H),let v be the vertex
12.        A ⟵ A∪ {(v,π[v])}
13.        S ⟵ S ∪ {v}
14.        for each x ∈ V-S adjacent to v do
15.          if w(x,v) < key[x] Then
16.            Decrease-key(H,x,w(x,v)),π[x] ⟵v
17.      end
18. return A
```

آنالیز الگوریتم :
در این الگوریتم heap به دو صورت می تواند پیاده سازی شود.که دو حالت آن و زمان لازم برای آن ها در زیر شرح داده شده است :

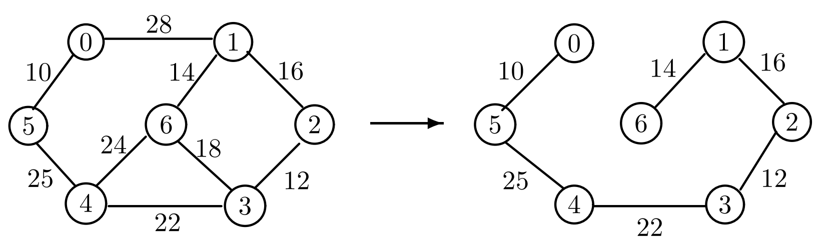
- استفاده از binary-heap:
در این حالت تابع Build-heap که برای مایک heap با مقادیر اولیه key[x] می سازد ، زمانی برابر $O(|V|)$ می گیرد. تابع Extract-Min کوچکترین مقدار heap را بر می گرداندو آن را از heap خارج می سازد،که در این حالت به دست آوردن کوچکترین مقدار heap از$\Theta(1)$ است ،زیرا این مقدار در ریشه قرار دارد،سپس باید این مقدار را برداشته ،آخرین عنصر heap را جایگزین آن نماییم و دوباره به یک heap دست پیدا کنیم ،که این عملیات از $O(log|V|)$ می باشد.
تابع Decrease-Key مقدار یک عنصراز heap را با مقدار کوچکتری جایگزین می نماید و سپس دوباره ساختار heap را بازسازی می نماید ، که در این حالت این عملیات نیز از $O(log|V|)$ می باشد.
به این ترتیب زمان کل الگوریتم از $O(|E|log|V| + |V|log|V|) = O((|E| + |V|)log|V|)$ خواهد شد.

- استفاده از fibonacci-heap(amortized):
در این حالت تابع Build-heap همان زمان $O(|V|)$ را می گیرد، تابع Extract-Min نیز همان زمان $O(log|V|)$ را می گیرد ،اما تابع Decrease-Key زمان $O(1)$ را می گیرد.
به این صورت زمان کل الگوریتم از $O(|E| + |V|log|V|)$ خواهد شد.

اگر گراف مورد نظر یک گراف اسپارس باشد ، یعنی $|E| = \Theta(|V|)$ ، در آن صورت استفاده ازfibonacci-heap کمک چندانی به ما نخواهد کرد زیرا هر دو هیپ زمانی برابر $O(|V|log|V|)$ را خواهند گرفت .
اگر گراف مورد نظر یک گراف متراکم (dense graph) باشد یعنی $|E| = \Theta(|V|^2)$ ، زمان استفاده از binary-heap برابر$O(|V|^2log|V|)$ و زمان استفاده از fibonacci-heap برابر$O(|V|^2)$ خواهد بود.
بنابراین استفاده از fibonacci-heap زمانی بهتر است که $|E|$ بزرگتر از $|V|$ باشد.
و این در حالی است که پیاده سازی این الگوریتم به وسیله ماتریس مجاورت زمانی از $O(|V|^2)$ می گیرد.

الگوریتم Sollin :

این الگوریتم را با مثال زیر شرح می دهیم ، ابتدا تمام رأس ها را بدون یال قرار می چینیم سپس از راس صفر شروع به مقایسه می نماییم آن که از نظر بر چسب کوچکتر است یک یال را تشکیل می دهد در این صورت سه درخت $\underbrace{0,5}_{A}$، $\underbrace{1,6}_{B}$ و $\underbrace{2,3,4}_{C}$ تشکیل می شود . مسیر هایی که از درخت B به C متصل می شود دارای ارزش های ۱۶و۱۸و۲۴ است و مسیر هایی که درخت A به B و C متصل می شود دارای ارزش ۲۸و۲۵ است پس کوتاهترین مسیر ۱٦ است که درخت B را به C متصل می کند و بین ۲۵و۲۸ نیز کوتاهترین مسیر ۲۵ است.




```
algorithm Sollin(G)
    begin
    F_o=(v,k);
    i=0;
    while there is more than one tree F_j do
    for each tree T_j in forest F_j do
        choose the minimum weighted edge (u,v) joining some
        vertix u in Tj to a vertix v in some other tree Tk in forest Fj
         from the otherforest F_{j+i} by joining all T_j and T_k of F_j with tree
         converesponding select edge
         i++;
    end;
```

الگوریتم Boruvka :

```
Boruvka( G = (V, E) , w ) {
    initialize each vertex to be its own component ;
    A = {} ; //A holds edges of the MST
    do {
      for ( each component C ) {
        find the lightest edge (u,v) with u in C and v not in C ;
        add { u, v } to A ( unless it is already there ) ;}
      apply DFS to graph H = (V, A) , to compute the new components
    } while (there are 2 or more components) ;
    return A ;} // return final MST edges
BORUVKA(V,E):
F = ( V , ∅ )
while F has more than one component
    choose leader using DFS
    FIND-SAFE-EDGES(V , E)
    for each leader v̄ add safe ( v̄ ) to F
FIND-SAFE-EDGES(V, E):
for each leader v̄
     safe(v̄)← ∞
for each edge (u,v)∈ E
     ū ← leader(u)
     v̄ ← leader(v)
     if ū ≠ v̄
        if (w(u,v) < w( safe( ū ))
         safe ( ū) ← (u, v)
        if (w (u,v) < w(safe(v̄ ))
         safe ( v̄) ← (u, v)
```

۳.۱.۴ الگوریتم کوله پشتی Knapsack :

فرض کنید که کوله پشتی داریم که وزن W را می تواند تحمل کند.می خواهیم این کوله را با شیء های $۱, ۲, ..., n$ پرکنیم. فرض کنید هر یک از شیء ها مانند شیء i ، دارای وزن w_i و ارزش v_i باشد. هدف آنست که برای پر کردن کوله پشتی به بیشترین ارزش ممکن دست یابیم برای این منظور می توان از هر شی یک واحد یا صفر واحد و یا کسری از واحد انتخاب کرد. در نتیجه:

$\sum x_i w_i \leq W \quad , \quad x_i \in [۰, ۱]$

انتخاب هابا ارزش نسبت مستقیم دارندو ما می خواهیم به $max\ [\sum x_i v_i]$ برسیم .
ابتدا نسبت $\frac{v_i}{w_i}$ را برای تمام اشیاء بدست می آوریم .سپس آنها را به تربیت نزولی مرتب می کنیم بنابراین از سر لیست یکی یکی اشیاء را انتخاب می کنیم و در کوله قرار می دهیم .تا جائی که نتوانیم شی ای را به طور کامل داخل کوله بیندازیم یعنی با انتخاب آن شی وزن اشیائی که در داخل کوله پشتی ریخته ایم یعنی weight بیشتر از W شود.
در این صورت برای این شی یعنی شی k ام به اندازه $x[k] = \frac{W-weight}{W[k]}$ انتخاب می شود. با انتخاب این شی وزن قطعات داخل کولهبه W خواهد رسید. و ارزش آنها نیز max خواهد شد. زمان این الگوریتم نیز از$O(n \log n)$ است زیرا فقطمرتب سازی داریم .

```
function Knapsack(w[1..n],v[1..n],W):array[1..n]
 for i ←1 to n do
     x[i] ←0
 weight← 0
 {gready loop}
 while (weight<W) do
    i ← the best remaining object
     if(weight+w[i]≤ W)then
       x[i]←1
       weight←weight+w[i]
     else
       x[i]← (W−weight)/w[i]
      weight ← W
 return x
```

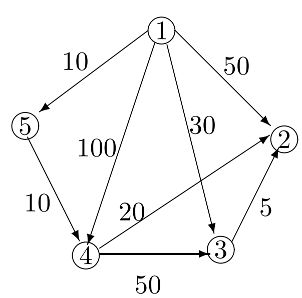
۴.۱.۴ الگوریتم DIJKSTRA :

این الگوریتم الگوریتم مسیر یابی است. فرض کنید گرافی جهت دار در اختیار داریم که هر یال آن با عددی نامنفی برچسب خورده است. می خواهیم طول کوتاهترین مسیر از رأسی به نام رأس منبع را تا سایر رئوس به دست آوریم . برای این منظور ماتریس وزن گراف رابه عنوان ورودی الگوریتم در نظر می گیریم.

```
function Dijkstra(L[1..n,1..n]):array D[2..n]
C←—{2,..,n}   {s=N\C exists only implicity}
for i ←— 2 to n
    D[i]←— L[1,i]
repeat n-2 times
    v←— some element of C minimizing D[v]
    C←— C\{v}
    for each w∈ C do
      D[w]←— min(D[w], D[v]+L[v,w])
return D
```

$$L_{ij} = \begin{cases} \text{weight of edge} & \text{(i,j)} \in \text{E} \\ 0 & \text{i=j} \\ \infty & \text{else} \end{cases}$$

مثال :



step	v	C	D
initilization	-	2,3,4,5	[50,30,100,10]
1	5	2,3,4	[50,30,20,10]
2	4	2,3	[40,30,20,10]
3	3	2	[35,30,20,10]

این الگوریتم تنها هزینه عبور از هر یال را می دهد. پس برای اینکه مسیر را داشته باشیم باید آرایه ای دیگر داشته باشیم تا گره های عبوری را در آن بریزیم پس خواهیم داشت :

```
function Dijkstra(L[1..n,1..n]):array D[2..n] , array P[2..n]
C⟵{2,..,n}{s=N\C exists only implicity}
for i ⟵ 2 to n
     D[i]⟵ L[1,i]
     P[i]⟵0
repeat n-2 times
    v⟵ some element of C minimizing D[v]
    C⟵ C\{v}
     for each w∈ C do
      if ( D[w]>D[v]+L[v,w] ) then
         D[w]⟵ D[v]+L[v,w]
         P[w]⟵ v
return D , P
```

تحلیل زمانی :

زمان ضریبی از تعداد مراحل اجراست که برابر است با:

$$(n-۲)+(n-۳)+\ldots+۲+۱=\frac{(n-۱)(n-۲)}{۲}\in\theta(n^۲)$$

پیدا کردن مسیر:

مسیر را از روی آرایه P به دست می آوریم. برای مثال قبل داریم :

	2	3	4	5
P	3	0	5	0

درایه هایی که صفر هستند بدین معنی هستند که از رأس ۱ به آن رأس مستقیم می رویم واین مسیر خود کوتاه ترین مسیر است .اما درایه های غیر صفر،مثلاً رأس ۴ ،چون درایه‌ی ۴ حاوی ۵ است پس برای رفتن به ۴ باید ابتدا به ۵ رویم و اما درایه‌ی ۵ خود حاوی صفر است پس برای رفتن به ۴ابتدا از ۱ به ۵ و سپس به ۴ می رویم .

زمان اجرای این الگوریتم باپیاده سازی Fibonacci-heap از O(a+nlogn) وبا Binary-heapاز O((a+n)logn) است.

۵.۱.۴ الگوریتم های زمان بندی (timetable or scheduling)

یکی از مباحث بسیار مهم در مباحثی چون سیستم عامل و یا بهینه سازی زمان عبارت است از زمان بندی .زمان بندی به مفهوم تقسیم زمان یک server بین چند کار است به طوری که در عین آنکه تمام مشتریان سرویس می گیرند بتوانیم به هدف مورد نظر نیز دست یابیم .اگر هدف minimum کردن زمان مشتریان در صف است به یک شیوه و اگر هدف بدست آوردن بیشترین سود یا امتیاز است به شیوه دیگر عمل کنیم .

از دید بحث الگوریتمی مسائل زمان بندی یک دسته از مسائل سخت می باشند که با توجه به شرایط آن جوابی مناسب با نوع مسئله بیابیم.

زمان بندی به دو دسته ساده (simple) و مهلت دار (with deadline) تقسیم می شود.

زمان بندی ساده

فرض نمایید یک server داریم که می خواهد به تعداد مشخصی مشتری برای مثال n مشتری سرویس دهد. زمان سرویس هر مشتری از قبل مشخص می باشد .فرض بر این است که مشتری iام زمان سرویسی به اندازه $t_i(۱ \leq i \leq n)$ را نیاز دارد و در هر زمان server می تواند به یک مشتری سرویس دهد .می خواهیم ببینیم به چه ترتیبی می توان به این مشتریان سرویس دهیم تا زمان سپری شده در سیستم از نقطه نظر مشتریان برای دریافت سرویس کمینه گردد . یعنی اینکه از نگاه مشتریان متوسط زمان تلف شده ی آن ها می نیمم باشد.

زمان انتظار مشتری iام: $T_i = t_۱ + t_۲ + ... + t_{i-۱} + t_i$

متوسط زمان انتظار برای دریافت سرویس: $\bar{T} = \sum \frac{T_i}{n}$

برای کمینه شدن $\bar{T}$ لازم و کافی است $\sum T$ کمینه شود .

برای درک بهتر الگوریتمی که می خواهیم بدست آوریم به مثال زیر می پردازیم :
فرض نمایید این server به ۳ مشتری سرویس می دهد .زمان برای مشتری اول ۵ ، دوم ۱۰ و سوم ۳ است که این ۳ مشتری به ٦ حالت می توانند سرویس بگیرند.

$$
\begin{array}{ll}
۱,۲,۳ & \Longrightarrow T = ۵ + (۵ + ۱۰) + (۵ + ۱۰ + ۳) \\
۱,۳,۲ & \Longrightarrow T = ۵ + (۵ + ۳) + (۵ + ۳ + ۱۰) \\
۲,۱,۳ & \Longrightarrow T = ۱۰ + (۱۰ + ۵) + (۱۰ + ۵ + ۳) \\
۲,۳,۱ & \Longrightarrow T = ۱۰ + (۱۰ + ۳) + (۱۰ + ۳ + ۵) \\
۳,۱,۲ & \Longrightarrow T = ۳ + (۳ + ۵) + (۳ + ۵ + ۱۰)\surd \\
۳,۲,۱ & \Longrightarrow T = ۳ + (۳ + ۱۰) + (۳ + ۱۰ + ۵)
\end{array}
$$

در این سیستم برای سرویس مشتریان را بر حسب زمان سرویس به ترتیب صعودی مرتب می نماییم و به ترتیب از ابتدای لیست به آنها سرویس می دهیم و این روند را آنقدر ادامه می دهیم تا تمام کارها سرویسشان را بگیرند. چون تنها زمان مرتب سازی را داریم از $O(n \log n)$ است .

زمان بندی مهلت دار(Scheduling with deedline)

فرض نمایید n کار اجرایی داریم که هر یک به یک واحد زمان برای اجرا نیاز دارند و در هر زمان(T=1,2,...) فقط یک کار می تواند انجام شود . کار i ام سودی به اندازه $g_i \geq ۰$ می رساند اگر وتنها اگر این کار را در زمان کمتر مساوی d_i که مهلت آن باشد به اتمام رسیده باشد. می خواهیم بدانیم به چه نحوی باید به این مشتریان سرویس داد تا به بیشترین سود دسترسی یابیم .

مثال :

فرض کنید که چهار مشتری داریم که به ترتیب میزان سود وdeedline آن ها به شرح زیر است :

i	1	2	3	4
g_i	50	10	15	30
d_i	2	1	2	1

sequence	profit	sequence	profit
1	50	2,1	60
2	10	2,3	25
3	15	3,1	65
4	30	4,1	80⟶ optimum
1,3	65	4,3	45

دنباله شدنی یا fisible

دنباله ای را fisible گوئیم اگر بتوانیم به ترتیب آنها را اجرا کرد . بنابراین هر مجموعه fisible دارای یک دنباله fisible است. برای مثال مجموعه2,1 ، fisible است اما دنباله 1,2 ، fisible نمی باشد .

فرض نمایید j مجموعه ای از k شغل باشد و بدون آن که از کلیت مطلب کم شود فرض کنید که کارها با $۱,۲,...,k$ برچسب خورده اند و به صورت $d_۱ \leq d_۲ \leq ... \leq d_k$ باشند .در این صورت j ، fisible است اگر وتنها اگر $۱,...,k$ ، fisible باشد .

برای رسیدن به بیشترین سود ابتدا شغل ها را براساس سودی که می رسانند به ترتیب نزولی مرتب می کنیم و سپس با به تعویق انداختن deedline آن ها به بیشترین سود دست می یابیم .

```
Function sequence(d[0..n]):k , array[1..k]
array j[0..n]
{The schedule is constructed step by step in the array j ,the variable k
say how many jobs are already in the schedule}
d[0]←j[0]←0 {sentinels}
k← j[1]←1{job 1 is always choosen}
{gready loop}
for i ← 2 to n do {decreasing order of g}
    r ← k
    while d[j[r]]>MAX(d[i],r) do r ← r-1
    if d[i]> r then
      for m ← k step -1 to r+1 do     j[m+1]←j[m]
     j[r+1]←i
      k ← k+1
return k,j[1..k]
```

برای مثال داریم :

i	1	2	3	4	5	6
g_i	20	15	10	7	5	3
d_i	3	1	1	3	1	3

3

Initialization:

1						

↑

1 3

Try 2:

2	1					

↑

Try 3: unchanged

1 3 3

Try 4:

2	1	4			

↑

Try 5: unchanged

Try 6: unchanged

$\Longrightarrow$ optimal sequence: 2 , 1 , 4 value=42

زمان اجرای این الگوریتم از$O(n^{۲})$ است .

الگوریتم دوم زمان بندی با استفاده از Disjoin Set :

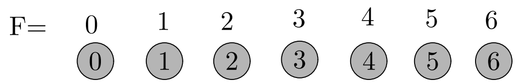
این الگوریتم بر اساس ساختمان داده مجموعه مجزا مسأله زمان بندی را انجام می دهد ،همان طور که قبلاً مشاهده شد تابع find ریشه‌ی درخت را می یابد و تابع merge دو درخت با ریشه های مشخص رابا هم ترکیب می کند و آن که کلیدش کمتر است فرزند ریشه‌ی دیگر خواهد شد.

```
Function sequence2(d[1..n]):k,array[1..k]
  array j ,F[0..n]
  {initialization}
  for i ← 0 to n do
     j[i] ← 0
     F[i]← i
     initialize set [i]
  {gready loop}
  for i ← 1 to n do {decreasing order of g}
     k ← find(min(n,d[i]))
     m ← F[k]
     if m ≠ 0 then
       j[m] ←i   , l← find(m-1)
     F[k]←F[l]
  merge(k,l)
  k ← 0
  for i ←1 to n do
     if j[i]>0 then   k←k+1  , j[k]←j[i]
  return k,j[1..k]
```

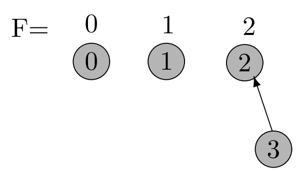
اگر پدر عنصری خودش شد مشخص می شود که آن ریشه است .

برای مثال قبل داریم :

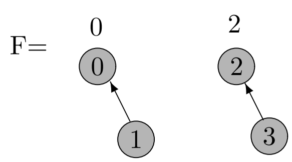
Initilize : L=min(6, max(d,1))= 3



Try 1: d_1=3 , assign task 1 to position 3

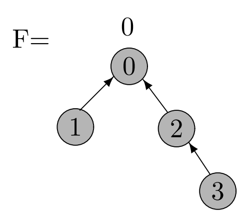


Try 2: d_2=1 assign task 2 to position 1



Try 3: d_3=1 no free position available since the F value is 0

Try 4: d_4=3 assign task 4 to position 2



Try 5: d_5=1 no free position available

Try 6: d_6=3 no free position available

$\Longrightarrow$ optimal sequence: 2 , 1 , 4 value=42

زمان اجرای این الگوریتم از $O(nlogn)$ است .

۲.۴ تقسیم وحل (devide and conquer)

D & C تکنیک برنامه نویسی است که در آن مسأله اصلی با استفاده از یک ساختار بازگشتی به زیر مسأله‌هایی تقسیم می شودکه دقیقاً شبیه مسأله اصلی هستند با این تفاوت که ازنظر اندازه کوچکتر یا ساده تر از مسأله اصلی هستند. با حل زیر مسأله های به اندازه کافی کوچک و ساده شده وترکیب آنها با هم می توان به یک جواب در صورت امکان برای مسأله اصلی دست یافت.
ساختار کلی یک الگوریتم D& C مطابق زیر است:

```
function DC(x)
if x is sufficiently small or simple then return adhoc(x);
decompose x into smaller instances x_1, x_2, ..., x_l;
for i←1 to l do y_i ←DC(x_i);
recombine the y_i's to obtain a solution y for x;
return y;
```

۱.۲.۴ ضرب اعداد بزرگ

اگر بخواهیم دو عدد را که عدد بزرگتر n رقمی است در هم ضرب نماییم به طور معمول از $O(n^۲)$ می باشد. حال اگر به صورت زیر عمل کنیم ،چه اتفاقی می افتد؟

$$۹۸۱ \times ۱۲۳۴ = (\underbrace{۹}_{w} \times ۱۰^۲ + \underbrace{۸۱}_{x}) \times (\underbrace{۱۲}_{y} \times ۱۰^۲ + \underbrace{۳۴}_{z}) =$$
$$(w \times ۱۰^۲ + x)(y \times ۱۰^۲ + z) = wy \times ۱۰^۴ + (wz + xy) \times ۱۰^۲ + xz$$

باز هم چهار عدد $\frac{n}{۲}$ باید با هم جمع شوند .پس داریم :

$$T(n) \leq ۴T(\tfrac{n}{۲}) + \theta(n) \Rightarrow T(n) \in O(n^۲)$$

همانطور که ملاحظه می فرمایید این الگوریتم نیزاز $O(n^۲)$ است که زمان را بهتر نمی کند درنتیجه D&C همیشه بهتر نمی باشد.

$$(wz + xy) = \underbrace{(w + x)(y + z)}_{r} - \underbrace{wy}_{p} - \underbrace{xz}_{q} = r - p - q$$
$$\Rightarrow wy \times ۱۰^۴ + (wz + xy) \times ۱۰^۲ + xz = p \times ۱۰^۴ + (r - p - q) \times ۱۰^۲ + q$$

$$\Rightarrow T(n) \le ۳T(\tfrac{n}{۲}) + \theta(n) \Longrightarrow T(n) \in O(n^{\log_۲^۳})$$

برای جمع دو عدد $\frac{n}{۲}$ رقمی $\frac{n}{۲}$ یا$\frac{n}{۲}+1$ جمع داریم پس مضربی از $\frac{n}{۲}$ است در نتیجه از $\theta(n)$ است.

الگوریتم :

```
large-integer prod(large-integer u , large-integer v)
    {
    large-integer x,y,w,z,r,p,q
    int m , n
    n=Maximum(number of digits in u , number of digits in v )
    if(u==0 || v==0)
         return 0
    else if(n≤ threshold )
        return u×v obtained in usual way
    else
      {
        m=⌊n/۲⌋
        x=u divide 10^m ;
        y= u rem 10^m
        w=v divide 10^m ;
        z= v rem 10^m
         r=prod(x+y,w+z)
         p=prod(x,w)
         q=prod(y,z)
        return p×10^(۲m) + (r-p-q)×10^m+q
      }
    }
```

۲.۲.۴ الگوریتم merge sort

procedure merge-sort (T[1..n])

if n is sufficiently small then insertion sort(T[1..n])

else{

$//arrayu[۱ \dots \lfloor \frac{n}{۲} \rfloor + ۱], array \quad v[۱ \dots \lceil \frac{n}{۲} \rceil + ۱]$

$u[۱ \dots \lfloor \frac{n}{۲} \rfloor] \leftarrow T[۱ \dots \lfloor \frac{n}{۲} \rfloor]$

$v[۱ \dots \lceil \frac{n}{۲} \rceil] \leftarrow T[۱ + \lfloor \frac{n}{۲} \rfloor \dots n]$

$merge - sort(u[۱ \dots \lfloor \frac{n}{۲} \rfloor])$

$merge - sort(v[۱ \dots \lceil \frac{n}{۲} \rceil])$

merge(u,v,T)}

در حالتی که کوچک باشد احتمال مرتب بودن زیاد است پس ازinsertion sort استفاده میشود.

procedure merge(u[1..m+1],v[1..n+1],T[1..m+n]){

i,j← 1

u[m+1]=v[n+1] ← ∞

for k← 1 to m+n Do

if u[i]<v[j]

T[k]←u[i]

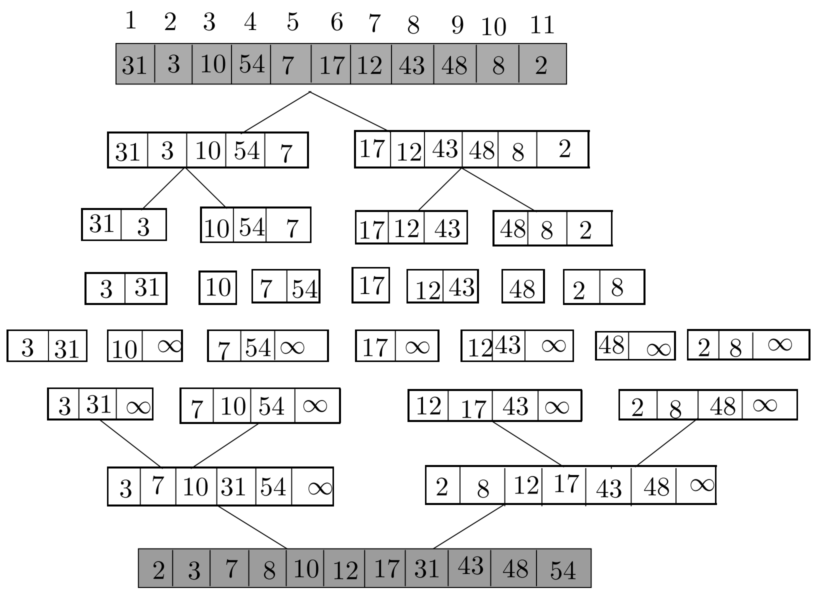
i++

else

T[k]←v[j]

j++}

مثال :
آرایه‌ی زیر را با این روش مرتب می نماییم :



آنالیز الگوریتم :

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \theta(n)$$

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

$$\Longrightarrow T(n) \in O(nlogn)$$

این الگوریتم stable نمی باشد اما اگر عبارت $if\ u[i] < v[j]$ را به‌عبارت $if\ u[i] \leq v[j]$ تبدیل کنیم stable خواهد بود. همچنین این الگوریتم inplace نمی باشد زیرا از حافظه‌ی کمکی استفاده می نماید.

۳.۲.۴ الگوریتم Quick Sort

```
Procedure pivot(T[1...n] , var L)
  {permutes the elements in array T[i...j] and return a value L such that
   at the end,i ≤L ≤ j, T[k] < p for all i ≤k < L, T[L]=p and
   T[k] > p for all L<k≤j , where p is the initial value of T[i] }
   p ←T[i]
   k←i , L←j+1
   repeat k←k+1 until T[k] > p or k ≥ j
   repeat L←L-1 until T[L] ≤ p
   while(k < L) do
       swap(T[L],T[k])
       repeat k ←k+1 until T[k] > p
       repeat L ←L-1 until T[L] ≤ p
   swap(T[i],T[L])

procedure quicksort (T[i...j])
   {sorts subarray T[i...j] into nondecreasing order}
   if (j - i) is sufficiently small then sort with insertion-sort(T)
   else
       pivot(T[i ... j],L)
       quicksort(T[i ... L -1])
       quicksort(T[L+1 ... j])
```

مثال :

i										j
5	1	17	4	48	7	3	9	8	5	25
1	2	3	4	5	6	7	8	9	10	11

p=T[i]=5

$k \leftarrow i$

$L \leftarrow j+1$

5	1	17	4	48	7	3	9	8	5	25
		↑							↑	
		k							L	

5	1	5	4	48	7	3	9	8	17	25
				↑		↑				
				k		L				

5	1	5	4	3	7	48	9	8	17	25
				↑	↑					
				L	k					

k از L جلوترافتادپس جای عنصر L ام با p عوض می شود.

3	1	5	4	5	7	48	9	8	17	25
				⇑						

محاسبه زمان اجرای الگوریتم:

- آنا لیزدربدترین حا لت :

ازآنجاکه در pivot باید از دو طرف یکی اول و دیگری آخر آرایه حرکت کرد پس همیشه از θ(n) است .

$$\begin{cases} \mathrm{T(n)=T(n\text{-}1)}+\theta(\mathrm{n}) \\ \mathrm{T(n)=c} \qquad \mathrm{n}\leq n_{\circ} \end{cases} \Longrightarrow \mathrm{T(n)}=\theta(n^{۲})$$

بدترین حالت زمانی اتفاق می افتد که pivot در هر بار یا درجای خود و یا در خانه متقارن یعنی (n-i+1) که i محل فعلی آنست ، قرار گیرد یعنی آنکه یا داده ها صعودی اند و یا نزولی مرتب شده با شند.

- آنا لیز الگوریتم در بهترین حا لت :

دراین الگوریتم بهترین حالت هنگامی اتفاق می افتد که pivot در نقاط میانی باشد ، در این صورت زمان اجرای الگوریتم برابر است با:

$$\mathrm{T(n)}\simeq ۲T(\tfrac{n}{۲})+\theta(n) \Rightarrow T(n)=\theta(n\log n)$$

مثال :
در گونه ای جدیدی از الگوریتم quick sort ، $2\sqrt{n}+1$ عنصر اول آرایه را با یک الگوریتم مقدماتی مانند insertion sort مرتب می نماییم ، سپس عنصر میانه را به عنوان pivot درنظر می گیریم ، در بد ترین حا لت زمان اجرای این الگوریتم از چه رابطه بازگشتی به دست می آ ید؟

حل:

$$\text{عنصر میانه} = \frac{۲\sqrt{n}+۱+۱}{۲} = \sqrt{n}+۱$$

پس بدترین زما ن آنست که این عنصر در جای خود باقی بماند پس دو دسته عنصر داریم ، یک دسته $\sqrt{n}$ عدد و دسته دوم از $\sqrt{n}+2$ به بعد.

زمان لازم برای مرتب سازی insertion هم $O(n) \in O((۲\sqrt{n}+۱)^۲)$ است
.

$$T(n) \leq T(\sqrt{n}) + \overbrace{T(n-\sqrt{n})}^{T(n-(\sqrt{n}+2))} + \overbrace{\theta(n)+O(n)}^{\theta(n)}$$

- آنا لیز الگوریتم در حالت متوسط :

$$\overbrace{۱\ldots k}^{k-۱}\overbrace{(k+۱)\ldots n}^{n-k}$$

$$T(n) = T(k-۱) + T(n-(k+۱)+۱) + \theta(n) =$$

$$\frac{۱}{n}\sum_{k=۱}^{n}(T(k-۱)+T(n-k)) + \theta(n) = \frac{۱}{n}\sum_{k=۱}^{n}T(k-۱) + \frac{۱}{n}\sum_{k=۱}^{n}T(n-k) + \theta(n) = \frac{۱}{n}\sum_{k=۰}^{n-۱}T(k) + \frac{۱}{n}\sum_{k=۰}^{n-۱}T(k') + \theta(n) = \frac{۲}{n}\sum_{k=۰}^{n-۱}T(k) + \theta(n) = \frac{۲}{n}(T(۰)+T(۱)) + \frac{۲}{n}\sum_{k=۲}^{n-۱}T(k) + \theta(n) = \frac{۲}{n}\times a + \frac{۲}{n}\sum_{k=۲}^{n-۱}T(k) + \theta(n)$$

حال با استقرای ریاضی نشان می دهیم که زمان در حالت متوسط $\theta(n\lg n)$ است. واضح است که اگر n=۲ باشد زمان الگوریتم ۱ است. از طرفی $2=\theta(2\lg 2)=\theta(n\lg n)$ است ،پس گام اول برقرار است .حال فرض کنید برای هر $k<n$ زمان لازم $\theta(k\lg k)$ است ،آنگاه

$$T(n)=\frac{2}{n}\times a+\frac{2}{n}\sum_{k=2}^{n-1}T(k)+\theta(n)=\frac{2}{n}a+\frac{2}{n}\sum_{k=2}^{n-1}\theta(klogk)+\theta(n)$$

$$\to T(n)=\frac{2}{n}\times a+\theta((\frac{2}{n})\sum_{k=2}^{n-1}(klogk))+\theta(n)$$

$$=\frac{2}{n}\times a+\theta(\frac{2}{n}\int_{2}^{n-1}xlogxdx)+\theta(n)$$

$$\to T(n)=\theta(nlogn)$$

تمرین:

الگوریتم quick sort را طوری تغییر دهید که k امین کوچکترین عنصر را به دست آوریم .

تمرین:

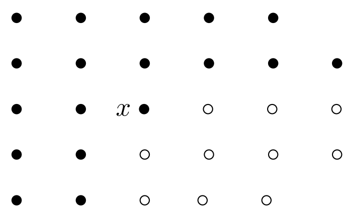
پیدا کردن میانه ها را با استفاده از یک تغییر در quick sort به دست آورید.

حال می خواهیم k امین کوچکترین عدد را به شیوه زیر بدست آوریم:

ابتدا n عنصررا به $(\lfloor n/5\rfloor)$ دسته‌های ۵ تایی تقسیم می کنیم . حداکثر یک دسته ممکن است تعدادی کمتر از ۵ عنصر داشته باشد بنابراین $(\lceil n/5\rceil)$ گروه داریم که حداکثر یکی از دسته ها دارای n mod 5 عنصر است.

سپس عنصر میانه‌ی هر یک از دسته های ۵ عضوی را با استفاده از الگوریتم insertion sort محاسبه می نماییم . واضح است اگر تعداد عناصر یک دسته زوج باشد عنصر میانه را عنصر کوچکتر فرض می کنیم . با استفاده از یک تابع بازگشتی بنام select میانه میانه های عناصر را بدست می آوریم، با استفاده از یک نسخه توسعه یافته pivot را حول میانه میانه ها یعنی x تقسیم می کنیم . بنابراین فرض کنید که i تعداد عناصر بخش پایین نقطه تقسیم باشد پس $n-i$ تعداد عناصر در بخش بالای نقطه تقسیم باشد . اگر k کوچکتر مساوی i باشد از select به صورت بازگشتی برای

یافتن k امین عنصر کمینه در بخش پایینی استفاده می کنیم ودر غیر این صورت $k-i$ امین عنصر کمینه را در بخش بزرگتر پیدا می نماییم.



حداقل تعداد عناصر بزرگتر از x: $۳(\frac{۱}{۲}\lceil\frac{n}{۵}\rceil - ۲) \ge \frac{۳n}{۱۰} - ۶$

حداکثر تعداد عناصر کوچکتر از x: $n - (\frac{۳n}{۱۰} - ۶) = \frac{۷n}{۱۰} + ۶$

برای بررسی زمان الگوریتم باید ذکر شود که برای مرتب سازی دسته‌های ۵تایی از insertion sort استفاده می شود که از $O(۱)$ است و چون $\lceil n/۵ \rceil$ دسته داریم پس از $O(n)$ است واز طرفی دیگر باید میانه میانه‌هابدست آید پس $T(\lceil n/۵ \rceil)$ است. در بدترین حالت ممکن است که k امین عنصر در یکی از دسته‌های $\frac{۷n}{۱۰} + ۶$ یا $\frac{۳n}{۱۰} - ۶$ رخ دهد زمان دسته بزرگتر را محاسبه می کنیم در کل داریم :

$$T(n) = T(\lceil n/۵ \rceil) + T(۷n/۱۰ + ۶) + O(n)$$
$$T(n) \in O(n)$$

در الگوریتمی ارائه شده به زبان دیگر داریم $۲(n/۵) - ۱$ حداکثر تعداد عناصری است که می توانند در دو طرف میانه میانه ها باشند.

حداکثر تعداد عناصری که می توانند در یک طرف میانه میانه ها باشند:

$$\frac{۱}{۲}((n-۱) - ۲(\frac{n}{۵} - ۱)) + ۲(\frac{n}{۵} - ۱) = \frac{۷n}{۱۰} - \frac{۳}{۲}$$

$$\Longrightarrow T(n) \le T(\lceil \frac{n}{۵} \rceil) + T(\frac{۷n}{۱۰} - \frac{۳}{۲}) + O(n)$$

۴.۲.۴ الگوریتم استراسن (الگوریتم ضرب ماتریس ها)

فرض کنید دو ماتریس در اختیار داریم از مرتبهٔ توانی از ۲ ، می خواهیم با روشی سریع تر از ضرب معمولی‌حاصل ضرب ماتریس ها را محاسبه کنیم. استراسن نشان داد که می توان ضرب ماتریس ها را که در حالت عادی به زمانی به اندازهٔ $\theta(n^3)$ نیاز دارد ، به زمانی معادل با $\theta(n^{\log_2^7})$ کاهش داد.

برای این منظور استراسن با بلوکه کردن ماتریس ها توانست ضرب دو ماتریس 2×2 را که در حالت عادی به ۸ ضرب نیاز دارد به ۷ ضرب کاهش دهد.
مانند مثال زیر:

$$C = \left(\begin{array}{cc|cc} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ \hline c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{array}\right) = \begin{pmatrix} c'_{11} & c'_{12} \\ c'_{21} & c'_{22} \end{pmatrix}$$

حال می خواهیم دو ماتریس $n \times n$ ، A وB را در هم ضرب نماییم :

$$A = \begin{pmatrix} a_{11}^{\nearrow \frac{n}{2}\times\frac{n}{2}} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}_{n\times n} \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}_{n\times n}, C = A \times B$$

$$m_1 = (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11})$$
$$m_2 = a_{11}b_{11}$$
$$m_3 = a_{12}b_{21}$$
$$m_4 = (a_{11} - a_{21})(b_{22} - b_{12})$$
$$m_5 = (a_{21} + a_{22})(b_{12} - b_{11})$$
$$m_6 = (a_{12} - a_{21} + a_{11} - a_{22})b_{22}$$
$$m_7 = a_{22}(b_{11} + b_{22} - b_{12} - b_{21})$$

ماتریس C به صورت زیر به دست می آید:

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

بنابراین ۷ عمل ضرب و۱۸ عمل جمع برای ماتریس های $\frac{n}{2} \times \frac{n}{2}$ نیاز داریم .
پس خواهیم داشت :

$$T(n) = 7T(\tfrac{n}{2}) + 18\left(\tfrac{n}{2}\right)^2$$
$$18\left(\tfrac{n}{2}\right)^2 \in O(n^{log_2^7 - \epsilon}) \Longrightarrow T(n) \in O(n^{log_2^7})$$

چون ممکن است بعضی از درایه ها صفر شوند به جای θ از O استفاده نموديم .

تمرین:

می خواهیم به روش استراسن ضرب دو ماتریس از مرتبه ۲۴ را انجام دهیم.حداقل تعداد ضرب چقدر است؟

$$T(24) = 7T(12) = 7^2T(6) = 7^3T(3) = 7^3 \times 3^3$$

مثال :

می خواهیم الگوریتمی را که کمترین تعداد ضرب های لازم برای محاسبهٔ a^n رابه ما می دهد به دست آوریم .

$$a^n = \begin{cases} (a^{\frac{n}{2}})^2 & \text{زوج } n \\ a \times a^{n-1} & \text{فرد } n \\ a & n = 1 \end{cases}$$

حل : T(n) را برابر تعداد ضرب ها می گیریم . بنابراین داریم :

$$T(n) = \begin{cases} T(\frac{n}{2}) + 1 & \text{زوج } n \\ T(n-1) + 1 & \text{فرد } n \\ 0 & n = 1 \end{cases}$$

$$\Rightarrow T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + 1 & \text{زوج } n \\ T(n-1) + 1 = T(\frac{n-1}{2}) + 2 = T(\lfloor \frac{n}{2} \rfloor) + 2 & \text{فرد } n \\ 0 & n = 1 \end{cases}$$

$$\Rightarrow T(n) = T(\lfloor \frac{n}{2} \rfloor) + \theta(1) \quad , \theta(1) = \theta(n^{\log_2 1}(\log n)^0)$$

$$\Rightarrow T(n) = \theta(\log n)$$

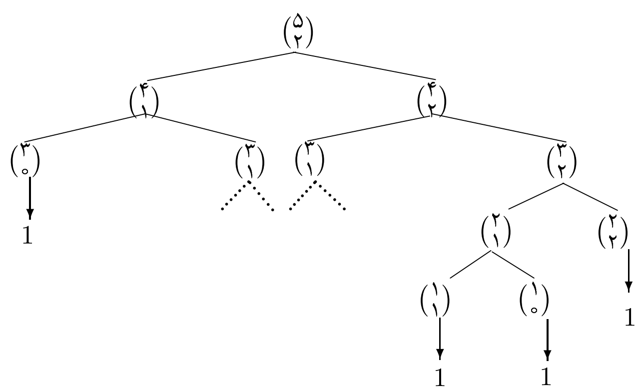
۳.۴ برنامه نویسی پویا Dynamic Programming

برنامه نویسی پویا یک روش نوشتن الگوریتم هاست که در آن مسأله اصلی را با استفاده از یک فرمول بازگشتی حل می نماییم یعنی در ابتدا برای مسأله یک ضابطه بازگشتی می یابیم. سپس با استفاده از یک حافظه سعی بر آن داریم که در ابتدا مسأله را برای مقادیر اولیه حل کنیم سپس با ترکیب حل های مقدماتی تر حل مسأله را در حالت بالاتر دنبال می نماییم . این روند را تا آنجا ادامه می دهیم که به پاسخی برای مسأله اصلی دست یابیم . روش DP از آنجا ابداع گردیده است که بتواند راه حل های تکراری را که ممکن است در روش D&C به وجود آید را حذف نماید.

در این جا راه حل راه حلی Bottom - Up است .

۱.۳.۴ محاسبه $\binom{n}{k}$:

$$\binom{n}{k} = \begin{cases} 1 & k=0 \quad or \quad k=n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & else \end{cases}$$



تعداد جمع هایکی از $\binom{n}{k}$ کمتر است یعنی حداقل زمان لازم برای این الگوریتم $\Omega(\binom{n}{k})$ است.

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

فرض: $g(n,k)=T(n,k)+1$

$$\text{g(n,k)}=\begin{cases} 1 & k=0 \quad or \quad k=n \\ g(n-1,k)+g(n-1,k-1) & else \end{cases}$$

$$g(n,k)=T(n-1,k)+T(n-1,k-1)+1+1=(T(n-1,k)+1)+$$
$$(T(n-1,k-1)+1)=g(n-1,k)+g(n-1,k-1)\Rightarrow g(n,k)=\binom{n}{k}$$
$$\Rightarrow T(n,k)=\binom{n}{k}-1$$

به طورکلی داریم :

$$\text{T(n,k)}=\begin{cases} 1-l & k=0 \quad or \quad k=n \\ T(n-1,k)+T(n-1,k-1)+l & else \end{cases}$$

برای $0\geq 1-l$ جواب معادله بازگشتی $T(n,k)=\binom{n}{k}-l$ است .

برای محاسبه این ترکیب با استفاده از برنامه نویسی دینامیک به طریق زیر عمل می کنیم:

برداری به نام c در نظر می گیریم و می دانیم:

c[n,k]=c[n-1,k-1]+c[n-1,k]

پس کافی است جدول زیر را تشکیل دهیم .

c	0		1		2	. . . .	k-1		k
0	1								
1	1	→	1						
			↓						
2	1	→	2	→	1				
			↓		↓				
3	1	→	3	→	3				
.	.								
.	.								
.	.								
.	.								
.	.								
n-1	1						c[n-1,k-1]	→	c[n-1,k]
									↓
n	1								c[n,k]

پس زمان محاسبه از $\theta(nk)$ خواهد بود ، زیرا n سطر و k ستون داریم .

۲.۳.۴ مسأله خرد کردن پول ها

می خواهیم N واحد پول را توسط سکه های $d_۱, d_۲, \ldots d_n$ واحدی خرد کنیم. فرض بر این است که از هر سکه به اندازه کافی موجود است . می خواهیم بدانیم به چه ترتیبی می توانیم این N واحد پول را خرد کنیم به طوری که از کمترین تعداد سکه ها استفاده کنیم.

برای حل این مسأله تعریف می کنیم c[i,j] را حداقل تعداد سکه لازم برای خرد کردن j واحد پول توسط سکه های $d_۱$, ... , d_i واحدی .

$$c[i,j]=\begin{cases} 0 & j=0 \\ \infty & i=1 , j < d_۱ \\ 1+c[1 , j-d_۱] & i=1 , j \geq d_۱ \\ c[i-1 , j] & i > 1 , j < d_i \\ \min\{ c[i-1 , j],1+c[i , j - d_i]\} & i >1 , j \geq d_i \end{cases}$$

به عنوان مثال ۸ واحد پول به صورت زیر خرد می گردد:

amount	0	1	2	3	4	5	6	7	8
$d_۱$=1	0	1	2	3	4	5	6	7	8
$d_۲$=4	0	1	2	3	1	2	3	4	2
$d_۳$=6	0	1	2	3	1	2	1	2	2

الگوریتم:

```
Function coins(N,n)
 {array d[1..n] specifies the coin,in example there are 1,4,6 units}
 array d[1..n]
 array C[0..n,0..N]
 for i ←1 to n do
     C[i,0]←0
 for i ←1 to n do
     for j ←1 to N do
         if(i=1 and j<d[1]) then C[i,j]← ∞
         else if(i=1 and j≥ d[1]) then C[i,j]← 1+C[1,j-d[1]]
         else if(i>1 and j<d[i])then C[i,j]← C[i-1,j]
         else C[i,j]← min{C[i-1,j],1+C[i,j-d[i]]}
```

زمان از $\theta((N+۱)n)$ است، زیرا n سطروN+1 ستون داریم .

۳.۳.۴ مسأله کوله پشتی $\{0,1\}$

فرض کنید کوله پشتی داریم که وزن W را می تواند تحمل کند . می خواهیم آن را با اشیای n . . . 1,2,. پر کنیم. وزن شی i ام w_i وارزش آن v_i می باشد.برای پر کردن آن هر بار می توان آن شی را انتخاب کرد یا نکرد(حالت کسری نداریم). بنابراین باید $\sum x_i w_i \leq W$ و $\max(\sum x_i v_i)$ را داشته باشیم که در آن $x_i \in \{0,1\}$.

برای حل‌تعریف می کنیم v[i,j] حداکثر ارزشی که یک کوله پشتی با وزن قابل تحمل j می تواند با اشیای i . . . 1,2,. داشته باشد. بنابراین داریم :

$$V[i,j]=\begin{cases} 0 & j=0, \forall\ i \\ -\infty \text{ or } 0 & i=1\ ,\ 0<j<w_1 \\ V_1 & i=1\ ,\ j \geq w_1 \\ V[i-1,j] & i>1\ ,0<j<w_i \\ \max\{\ V[i-1,j], V_i+V[i-1,j-w_i]\} & i>1\ ,\ j\geq w_i \end{cases}$$

به مثال زیر دقت نمایید:

weight	unit	0	1	2	3	4	5	6	7	8	9	10	11
$w_1=1$	$v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2=2$	$v_2=6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3=5$	$v_3=6$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4=6$	$v_4=22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5=7$	$v_5=28$	0	1	6	7	7	18	22	28	29	34	35	40

همان طور که مشاهده می شود زمان از $\theta((W+1)n)$ است .

۴.۳.۴ الگوریتم Floyd :

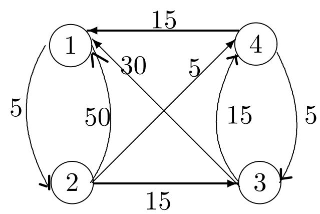
این الگوریتم هزینه کوتاه ترین مسیر بین هر دو راس متمایز از یک گراف جهت دار وزن دار را که وزن هر یال آن نامنفی است محاسبه می کند.

```
Function Floyd (L[1..n,1..n]):array D[1..n,1..n]
   D←L
   for k ← 1 to n do
      for i ← 1 to n do
         for j ← 1 to n do
            D[i,j] ← min(D[i,j], D[i,k] + D[k,j])
   return D
```

زمان این الگوریتم از $\theta(n^{۳})$ است .برای گرفتن مسیر می توان از آرایه کمکی P استفاده نمود:

```
Function Floyd (L[1..n,1..n]):array D[1..n,1..n],array P[1..n,1..n]
    D←L
    P← ∅
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                if D[i,j] > D[i,k]+D[k,j]
                    D[i,j] ← D[i,k]+D[k,j]
                    P[i,j] ← k
    return D , P
```

به مثال زیر دقت نمایید:



$$D_{۰} = L = \begin{pmatrix} ۰ & ۵ & \infty & \infty \\ ۵۰ & ۰ & ۱۵ & ۵ \\ ۳۰ & \infty & ۰ & ۱۵ \\ ۱۵ & \infty & ۵ & ۰ \end{pmatrix} \qquad D_{۱} = \begin{pmatrix} ۰ & ۵ & \infty & \infty \\ ۵۰ & ۰ & ۱۵ & ۵ \\ ۳۰ & ۳۵ & ۰ & ۱۵ \\ ۱۵ & ۲۰ & ۵ & ۰ \end{pmatrix}$$

$$D_{۲} = \begin{pmatrix} ۰ & ۵ & ۲۰ & ۱۰ \\ ۵۰ & ۰ & ۱۵ & ۵ \\ ۳۰ & ۳۵ & ۰ & ۱۵ \\ ۱۵ & ۲۰ & ۵ & ۰ \end{pmatrix} \qquad D_{۳} = \begin{pmatrix} ۰ & ۵ & ۲۰ & ۱۰ \\ ۴۵ & ۰ & ۱۵ & ۵ \\ ۳۰ & ۳۵ & ۰ & ۱۵ \\ ۱۵ & ۲۰ & ۵ & ۰ \end{pmatrix}$$

$$D_{۴} = \begin{pmatrix} ۰ & ۵ & ۱۵ & ۱۰ \\ ۲۰ & ۰ & ۱۰ & ۵ \\ ۳۰ & ۳۵ & ۰ & ۱۵ \\ ۱۵ & ۲۰ & ۵ & ۰ \end{pmatrix} \qquad P = \begin{pmatrix} ۰ & ۰ & ۴ & ۲ \\ ۴ & ۰ & ۴ & ۰ \\ ۰ & ۱ & ۰ & ۰ \\ ۰ & ۱ & ۰ & ۰ \end{pmatrix}$$

تمرین : اگر وزن یال های گراف منفی باشد چه مشکلی می تواند ایجاد کند؟

تمرین : تمرین بالا را در مورد الگوریتم Dijkstra بررسی نمایید ؟

۵.۳.۴ ضرب زنجیره‌ای ماتریس ها (chained matrix multiplication)

n ماتریس $A_1, A_2, ..., A_n$ مفروض هستند.هر ماتریس A_iاز مرتبه $d_{i-1} \times d_i$ می باشد. می خواهیم حاصل A=$A_1 A_2 .....A_n$ رامحاسبه نماییم به طوریکه این ضرب در کمترین زمان ممکن اتفاق افتد. بدیهی است برای این کار ماتریس ها باید به گونه ای پرانتزگذاری شوند که تعداد ضربهای آنها کمینه شود.

برای این منظور ماتریس M=$(m_{ij})_{n*n}$ را چنان می سازیم‌که m_{ij} حداقل تعداد ضرب لازم برای محاسبه ضرب $A_i A_{i+1} .....A_j$ باشد.

$m_{ii} = 0$ تعداد ضرب لازم‌در محاسبه A_i

$m_{ii+1} = d_{i-1} d_i d_{i+1}$ تعداد ضرب لازم‌در محاسبه $A_i A_{i+1}$

می دانیم در ضرب ماتریس ها روابط زیر را داریم :

$A = (a_{ij})_{p \times q}$ $B = (b_{ij})_{q \times r}$ $C = (c_{ij})_{p \times r} = AB$

$\Rightarrow c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$

```
for i=1 to p do
    for j=1 to r do                    ∈ θ(pqr)
        for k=1 to q do
            c_ij = c_ij + a_ik b_kj
```

بنابراین داریم :

$(A_i A_{i+1} ...A_k)(A_{k+1} ...A_j)$

$$m_{ij} = \begin{cases} 0 & \text{i=j} \\ min_{i \le k \le j-1}\{m_{ik} + m_{k+1,j} + d_{i-1} d_k d_j\} & \text{j>i , i=1,2,.....n-1} \end{cases}$$

با تغییر متغیر زیر داریم :

j=i+s $\Rightarrow$

$$m_{i,i+s} = \begin{cases} 0 & \text{s=0 , i=1,...,n} \\ min_{i \le k \le i+s-1}\{m_{ik} + m_{k+1,i+s} + d_{i-1} d_k d_{i+s}\} & \text{i=1,2,.....n-s ,} \\ & 1 \le \text{ s } \le \text{n-1} \end{cases}$$

الگوریتم :

```
int minmult(int n,const int d[ ],index p[ ][ ]){
    index i,j,k,diagonal;
    int M[1..n][1..n]
    for (i=1;i ≤ n;i++)
      M[i][[i]=0;
    for (diagonal=1;diagonal ≤ n-1;diagonal++)
      for (i=1;i ≤ n-diagonal;i++){
        j ← i+diagonal
        m[i][j] = min_{i≤k≤j−۱}{m[i][k] + m[k + ۱][j] + d[i − ۱]d[k]d[j]}
        p[i][j]=a value of k that gave the minimum
      }
return M[1][n];}
```

زمان اجرای الگوریتم به‌صورت زیر است :

$$\sum_{s=۱}^{n-۱} s(n-s) = \Theta(n^{۳})$$

مثال :در ماتریس های زیر حداقل تعداد ضرب ها را بدست آورید.

$A_۱ = A_{۱۳\times ۵} \qquad A_۲ = B_{۵\times ۸۹} \qquad A_۳ = C_{۸۹\times ۳} \qquad A_۴ = D_{۳\times ۳۴}$

$$\text{s=0} \Longrightarrow \begin{cases} m_{۱۱} = ۰ \\ m_{۲۲} = ۰ \\ m_{۳۳} = ۰ \\ m_{۴۴} = ۰ \end{cases}$$

$$\text{s=1} \Longrightarrow \begin{cases} m_{۱۲} = d_۰ d_۱ d_۲ = ۱۳ \times ۵ \times ۸۹ = ۵۷۸۵ \\ m_{۲۳} = d_۱ d_۲ d_۳ = ۵ \times ۸۹ \times ۳ = ۱۳۳۵ \\ m_{۳۴} = d_۲ d_۳ d_۴ = ۸۹ \times ۳ \times ۳۴ = ۹۰۷۸ \end{cases}$$

$$\text{s=2} \Longrightarrow \begin{cases} m_{۱۳} = \{m_{۱۱} + m_{۲۳} + ۱۳ \times ۵ \times ۳, m_{۱۲} + m_{۳۳} + ۱۳ \times ۸۹ \times ۳\} \\ = min\{۱۵۳۰, ۹۲۵۶\} = ۱۵۳۰ \\ m_{۲۴} = ۱۸۴۵ \end{cases}$$

$$\text{s=3} \Longrightarrow \begin{cases} m_{14} = min\{m_{11} + m_{24} + d_0 d_1 d_4, m_{12} + m_{34} + d_0 d_2 d_4, m_{13} + \\ m_{44} + d_0 d_3 d_4\} = 2856 \end{cases}$$

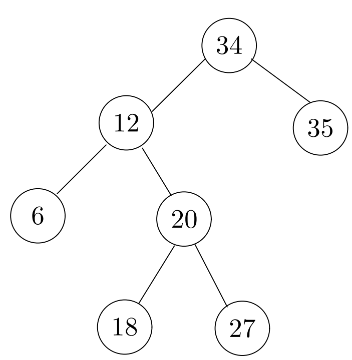
۶.۳.۴ درخت جستجوی دودویی بهینه

فرض کنید که n کلید در اختیار داریم که می خواهیم این n کلید را روی یک درخت جستجوی دودویی قرار دهیم. هر یک از این کلید ها دارای احتمال معینی برای دسترسی می باشند. کلید ها به ترتیب صعودی مرتب شده اند. (کلید منحصر به فرد است)فرض بر این است که احتمال دسترسی به کلید i ام p_i می باشد. واضح است که $\sum_{i=1}^{n} p_i = 1$. می خواهیم بررسی نماییم که به چه ترتیبی این n کلید را به یک درخت جستجوی دودویی تبدیل نماییم تا متوسط هزینه دسترسی به گره ها می نیمم باشد.

برای مثال فرض کنید که جدول شماره کلید ها و احتمال دسترسی به آنها مطابق زیر باشد:

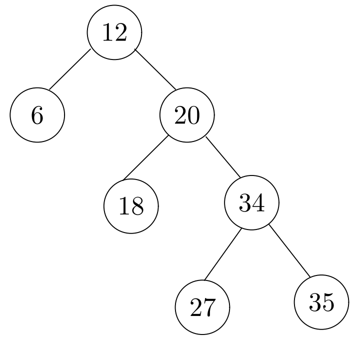
node	6	12	18	20	27	34	35
probability	0.2	0.25	0.05	0.1	0.05	0.3	0.05

یکی ازحالاتی که می توان یک درخت جستجوی دودویی داشت شکل زیر است .می خواهیم متوسط هزینه دسترسی به مجموع گره ها را بیابیم .پس متغیر تصادفی تعداد مقایسه ها برای دسترسی می باشد.



$$1\times 0.3+2\times 0.25+2\times 0.05+3\times 0.2+3\times 0.1+4\times 0.05+4\times 0.05 = 2.2$$

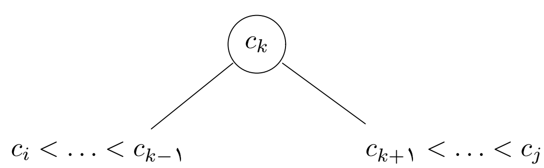
متوسط هزینه دسترسی به تمام گره هابرابر $\sum_{i=۱}^{n} p_i(depth(c_i)+۱)$ می باشد ، حال اگر درخت به صورت زیر باشد مقدار دیگری به دست خواهد آمد.



$$۱\times ۰.۲۵+۲\times ۰.۲+۲\times ۰.۱+۳\times ۰.۰۵+۳\times ۰.۳+۴\times ۰.۰۵+۴\times ۰.۰۵ = ۲.۳$$

راه دینامیک برای حل مسئله :

فرض کنید c_{ij} حداقل هزینه برای دسترسی به گره‌های $c_i, c_{i+۱}, \dots, c_j$ باشند. بنابراین $c_{۱n}$ جواب مسأله است . یک گره داریم که در ریشه است.



$1) i = j \Rightarrow c_{ii} = p_i$

$2) j > i \quad \Rightarrow c_i < \dots < c_{k-۱} < c_k < c_{k+۱} < \dots < c_j$

هزینه برای ریشه‌های سمت چپ c_k برابر $c_{i,k-۱}$ است و هزینه برای سمت راست $c_{k+۱,j}$ است. پس وقتی که ریشه c_k را داریم یک احتمال از تمام فرزندان c_k به مجموع اضافه می شود. برای خود ریشه نیز p_k است .

$$c_{i,k-۱}+c_{k+۱,j}+p_k+p_i+\dots+p_{k-۱}+p_{k+۱}+\dots+p_j = c_{ik-۱}+c_{k+۱j}+\sum_{t=i}^{j} p_t$$

$$\Rightarrow c_{ij} = min_{i\leq k\leq j}\{(c_{ik-۱}+c_{k+۱j})+\sum_{t=i}^{j}p_t\}$$

تعریف می نماییم $j=s+i$ پس داریم :

$$c_{i,i+s} = \begin{cases} p_i & s=۰, i=۱,...,n \\ min_{i\leq k\leq i+s}\{c_{i,k-۱}+c_{k+۱,i+s}\}+\sum_{t=i}^{i+s}p_t & ۱\leq s\leq n-۱ \\ & ,۱\leq i\leq n-s \end{cases}$$

کد این برنامه شبیه ضرب زنجیره ای ماتریس ها است.

و برای زمان اجرا نیز داریم: $\sum_{s=۱}^{n-۱}(n-s)(s+۱)=\theta(n^۳)$.

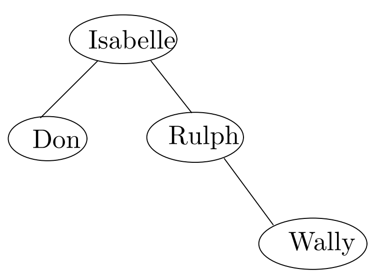
مثال :

فرض کنید ۴ کلید زیر را در اختیار داریم:

Don	Isabelle	Rulph	Wally
key[1]	key[2]	key[3]	key[4]
$p_۱=\frac{۳}{۸}$	$p_۲=\frac{۳}{۸}$	$p_۳=\frac{۱}{۸}$	$p_۴=\frac{۱}{۸}$

C	1	2	3	4
1	$\frac{۳}{۸}$	$\frac{۹}{۸}$	$\frac{۱۱}{۸}$	$\frac{۷}{۴}$
2		$\frac{۳}{۸}$	$\frac{۵}{۸}$	1
3			$\frac{۱}{۸}$	$\frac{۳}{۸}$
4				$\frac{۱}{۸}$

P	1	2	3	4
1	1	1	2	2
2		2	2	2
3			3	3
4				4



P ماتریس مسیر است .

۷.۳.۴ بزرگترین زیررشته مشترک

فرض کنید $Y = y_1y_2\dots y_n,\ X = x_1x_2\dots x_n$ دو رشته باشند. می خواهیم بزرگترین زیر رشته مشترک بین این دو را بیابیم.منظور از یک زیر رشته از یک رشته، رشته ای است که از حذف هیچ یا یک یا چند حرف از آن رشته بدست آمده باشد.
حل دینامیک:

تعریف می نماییم $X_i = x_1x_2\dots x_i$,$Y_j = y_1y_2\dots y_j$.

LCS=Longest Common Subsequence

$$LCS[X_i, Y_j] = \begin{cases} 0 & i = 0 \quad or \quad j = 0 \\ 1 + LCS[X_{i-1}, Y_{j-1}] & x_i = y_j, i \neq 0, j \neq 0 \\ Max\{LCS[X_{i-1}, Y_j], LCS[X_i, Y_{j-1}]\} & x_i \neq y_j, i \neq 0, j \neq 0 \end{cases}$$

```
LCS-length(X,Y)
m← length(X)
n← length(Y)
for i← 1   to   m   do
        c[i,0]← 0
for j←0   to   n   do
        c[0,j]←0
for i←1   to   m   do
    for j←1   to   n   do
        if (x_i = y_j) then
               c[i,j] ⟵ c[i-1,j-1]+ 1
               b[i,j] ⟵ " ↖ "
          else if (c[i-1,j] ≥ c[i,j-1]) then
               c[i,j] ⟵ c[i-1,j]
               b[i,j] ⟵ " ↑ "
  else
               c[i,j] ⟵ c[i,j-1]
               b[i,j] ⟵ " ← "
 return c and b                    //θ(nm)
```

```
Print-LCS(b,X,i,j)
if i=0 or j=0 then
    return
if b[i,j]="↖" then
    Print-LCS(b,X,i-1,j-1)
    Print x_i
if b[i,j]="↑ " then
    Print-LCS(b,X,i-1,j)
else Print-LCS(b,X,i,j-1)            //O(n + m)
```

باید توجه نمود که این جواب منحصر به فرد نمی باشد.برای به دست آوردن زیر رشته مشترک باید جهت فلش ها را دنبال نماییم.

مثال :

X=ABCBDAB Y=BDCABA LCS=BCBA

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

۸.۳.۴ مسأله ی مسابقات جهانی

دو تیم A و B آنقدر مسابقه می دهند تا یکی از دو تیم به n پیروزی دست یابد در آن صورت این تیم می تواند به مرحله‌ی بعد مسابقات راه یابد. بدیهی است که این دو تیم باید حداقل n و حداکثر $2n-1$ بازی داشته باشند (حالت تساوی وجود ندارد).
اگر احتمال برد تیم A در هر بازی p و احتمال شکست A در نتیجه برد B برابر q باشد و فرض کنیم که هر بازی به شکل مستقل از بازی دیگر صورت می گیرد چنانچه $p(i,j)$ احتمال برد A باشد مشروط بر اینکه تیم A به i پیروزی و تیم B به j پیروزی نیاز داشته باشد.

- رابطه ی احتمالی $p(i,j)$ به شکل بازگشتی زیر است :

$$\begin{cases} p(i,j) = p*p(i-1,j) + q*p(i,j-1)\ , & i \geq 1, j \geq 1 \\ p(0,j) = 1 & \forall j \geq 1 \\ p(i,0) = 0 & \forall i \geq 1 \end{cases}$$

- حل به روش Devide and conquer :

```
function p(i,j){
    if i=0 then return 1
    else if j=0 then return 0
    else return p*p(i-1,j)+q*p(i,j-1)
};
```

حال می خواهیم تعداد جمع های لازم برای محاسبه ی $p(n,n)$ را بیابیم و رابطه ی بازگشتی زیر را برای تعداد جمع ها داریم :

$$\begin{cases} g(i,j) = g(i-1,j) + g(i,j-1) + 1 \\ g(i,0) = g(0,j) = 0 \end{cases}$$

با تغییر متغیّر $h(i+j,j) = g(i,j)$ رابطه ی بالا به صورت زیر در می آید:

$$\begin{cases} h(i+j,j) = h(i+j-1,j) + h(i+j-1,j-1) + 1 \\ h(i+0,0) = h(0+j,j) = 0 \end{cases}$$

با استفاده از فرمول پاسکال تعداد جمع های به کار رفته $\binom{i+j}{j} - 1$ خواهد بود.
ما می دانیم که تعداد ضرب ها در رابطه ی بازگشتی مورد بررسی دو برابر تعداد

جمع هاست . پس برای محاسبه ی $p(n,n)$ به تعداد $2\binom{2n}{n}-2$ ضرب انجام می شود. و درنتیجه کد بالا دارای زمانی در حد $\Omega\binom{2n}{n}$ است ،که بزرگتر از $\frac{4^n}{2n+1}$ می باشد .پس استفاده از این روش کارآمد نیست .

- حل به روش Dynamic programming :

```
function series(n,p){
  array p[0..n,0..n]
  q = 1 - p
  { fill from topleft to diagonal }
  for s = 1 to n do
    p[0,s] = 1 , p[s,0] = 0
    for k = 1 to s - 1 do
      p[k,s - k] = p * p[k - 1,s - k] + q * p[k,s - k - 1]
  { fill from below diagonal to bottomright }
  for s = 1 to n do
    for k = 0 to n - s do
      p[s + k,n - k] = p * p[s + k - 1,n - k] + q * p[s + k,n - k - 1]
  return p[ n , n ]};
```

به این ترتیب پیچیدگی زمانی از $\theta(n^2)$ شد که بسیار بهینه تر از روش قبل است .

$$\begin{array}{c|c}
 & \begin{array}{ccccccc}0 & 1 & 2 & 3 & 4 & \cdots & n\end{array} \\
\begin{array}{c}0\\1\\2\\3\\\vdots\\n\end{array} & \left\{\begin{array}{ccccccc}
0 & 1 & 1 & 1 & & \cdots & \\
0 & p & p+pq & & & & \\
0 & p^2 & p^3+pq+pq^2 & & & & \\
0 & & & & & & \\
\vdots & & & & & & \\
0 & & & & & &
\end{array}\right\}
\end{array}$$

۹.۳.۴ مسأله فروشنده دوره گرد

فرض کنید یک فروشنده دوره گرد می خواهد به n شهر برود و برگردد به طوری که کمترین هزینه مسیر را داشته باشد یعنی از هر به شهر دیگر یک مسیر با هزینه مشخص وجود دارد.

حل: در اینجا گراف جهت دار با n راس را توسط آرایه دو بعدی W نشان می دهیم در آنصورت W[i][j] نشان دهنده طول یال بین راس i و j است. minlenght نشان دهنده تور بهینه است ، P آرایه دو بعدی است که سطرهای آن از ۱ تا n است و ستون آن توسط زیر مجموعه های $V - v_۱$ اندیس گذاری شده است . P[i][A] اندیس نخستین راس پس از v_i روی کوتاه ترین مسیر از v_i به $v_۱$ است که از همه رئوس A دقیقاً یکبار می گذرد .

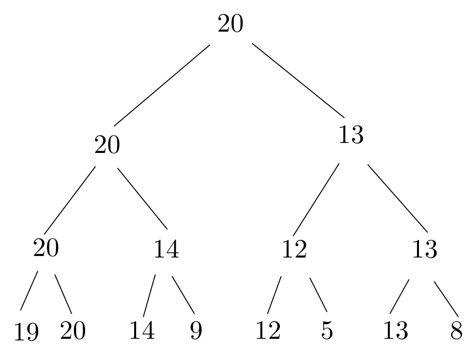
```
void travel(int n,const number W,index P,number & minlenght)
  {
  index i,j,k;
  number D[1..n][subset of V-{v۱}];
  for (i=2 ;i ≤ n;i++)
     D[i][∅]=W[i][1];
  for(k=1;k≤n-2;k++)
     for(all subset A ⊆ V -{v۱}contaning k vertices)
        for (i such that i≠1 and v_i is not in A){
           D[i][A]=min_{j:v_j∈A}(W[i][j]+D[j][A-{v_j});
           P[i][A]=value of j that gave the minimum ;
           }
  D[1][V-{v۱}]=min_{۲≤j≤n}(W[1][j]+D[j][V-{v۱, v_j}])
  P[1][V-{v۱}]=value of j that gave the minimum;
  minleght=D[1][V-{v۱}];
  }
```

۴.۴ تورنمنت بازی ها

می خواهیم با استفاده از یک الگوریتم قطعی دومین بزرگترین عنصر را بین n عنصر بیابیم روش تورنمنت از روش جام های حذفی الگو برداری شده است .در این روش برای بدست آوردن قهرمان تیم هابه شیوه زیر عمل می شود:

تیم های موجود به دسته های ۲ تایی تقسیم شده سپس هر دو تیم با هم مسابقه می دهند قهرمان این تیم ها می تواند به دور بعد راه یابد سپس این روند با بر گزاری تورنمنت بعدی ادامه می یابد تا آنکه قهرمان قهرمان ها مشخص شود.

مثال:



در اینجا اگر تعداد عناصر مضربی از توان ۲ نباشد ارتفاع $\lceil \lg n \rceil$ است.تعداد عناصر $-\infty$ برابر $2^h - n$ است.

$$n = ۲^k : \sum_{i=۱}^{\lg n} \frac{n}{۲^i} = \frac{\frac{n}{۲}(۱ - (\frac{۱}{۲})^{\lg n})}{۱ - \frac{۱}{۲}} = n(۱ - \frac{۱}{n}) = n - ۱$$

این روش برای بدست آوردن ماکزیمم عناصری که تعدادشان توانی از ۲ نباشد نیزبه همان n-1 مقایسه نیاز دارد،که فرقی با روش معمولی ندارد اما برای بدست آوردن دومین بزرگترین عنصر مناسب است .

برای بدست آوردن دومین بزرگترین عنصر باید ماکسیمم را در بین بازنده های ریشه (یعنی ۲۰)پیدا نماییم . پس به اندازه ارتفاع درخت ،عنصر بازنده به ریشه داریم. پس تعداد مقایسه ها $\lceil \lg n \rceil$ -1 است . پس در کل n-1+$\lceil \lg n \rceil$-1=n+$\lceil \lg n \rceil$-2 مقایسه داریم .

تمرین : نشان دهید این تعداد مقایسه حداقل تعداد مقایسه برای بدست آوردن دومین بزرگترین عنصر است .

۵.۴ B &T (Back Tracking)

بازگشت به عقب گرد تکنیک برنامه نویسی است که در آن برای حل یک مسئله از یک گراف جهت دار (درخت جهت دار)استفاده می گردد .یعنی هر نقطه از فضای مسئله را متناظر با یک راس و یا یک یال از یک گراف یا درخت جهت دار در نظر می گیریم . آنچه که در مورد حل مسائل B&T مهم است آنست که بایستی به نکته زیر توجه شود . الف : فاکتور شاخه ب : تابع Promissing

فاکتور شاخه یعنی حداکثر تعداد گره هایی که می توان از یک رأس به عنوان فرزند دسترسی داشت و تابع promissing تابعی است که بررسی می کند که آیا انتخاب گره اخیر شدنی (fisible) است یا خیر . منظور از امکان انتخاب گره اخیر یعنی اینکه با انتخاب این گره تناقض یا تداخلی‌به وجود خواهد آمد یا خیر . مزیت این روش دادن تمام حالت هاست .

۱.۵.۴ مساله n وزیر

فرض کنید که می خواهیم در یک صفحه شطرنج n×n، n وزیر را قرار دهیم به طوری که هیچ دو وزیری یکدیگر را تهدید ننمایند. در اینجا باید چک شود که تداخل وزیرiام وkام اتفاق نیفتد . وزیر iدر خانه [i,col[i]] است . وزیر kام در خانه [k,col[k]] است. پس باید برای $k < i$ دو حالت زیر رخ دهد:

col[i]$\neq col[k], |col[i] - col[k]| \neq$ i-k

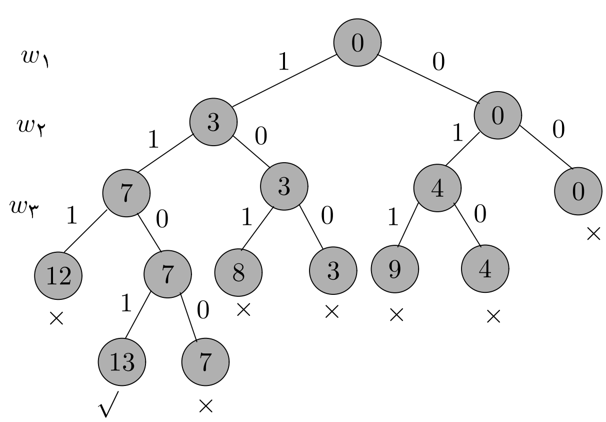
```
bool promssing (index i) {
index k;
bool switch;
k=1;
switch=true;
while(k < i && switch){
    if (col[i]==col[k] || abs(col[i]-col[k])==i-k)
      switch=false;
    k++;}//{end of while}
return switch;}
```

فاکتور شاخه: درهر شاخهn ستون داریم پس فاکتور شاخه nاست.

```
void queens (index i){
index j;
if (promissing (i))
    if (i==n)
        cout<< col[1] through col[n];
    else
        for(j=1;j <= n;j++){
            col[i+1]=j;
            queens(i+1);}
}
```

مثال : فرض کنید که یک کوله پشتی داریم که می تواند وزن W راتحمل نماید. می خواهیم این کوله را با اشیا $۱, ۲, ..., n$ پر کنیم.هر شی دارای وزن w_i است.در اینجا یک یا هیچ شی را میتوانیم انتخاب نماییم.نحوه پر کردن باید به گونه ای باشد که دقیقاً وزن اشیایی که انتخاب می شوند برابر w باشد. می خواهیم تمام حالتهای ممکن را بیابیم.

فرض کنید که وزن کوله پشتی 13 و وزن اشیا $w_۱ = ۳,\ w_۲ = ۴, w_۳ = ۵$ و $w_۴ = ٦$ باشد. در این درخت اگر برچسب یال یک باشد آن شی اضافه می شود اگر صفر باشد اضافه نمیشود .این درخت اول عمق است. فاکتور شاخه 2 است زیرا در هرگره دوفرزند داریم.



$weight_k = \sum_{i=۱}^{k-۱} x_i w_i \ , \quad total_k = \sum_{i=k}^{n} w_i$

```
void sum-of-subset(index i,int weight,int total){
        if (promissing(i)){
            if(weight==W)
                cout<<x[1] through x[n];
            else{
                x[i+1]⟵ 1
                sum-of-subset(i+1,weight+w[i+1],total-w[i+1]);
                x[i+1]⟵ 0
                sum-of-subset(i+1,weight,total-w[i+1]);}
                                    }
            }
bool promissing(index i){
  return
  ((weight + total ≥ W)&&((weight == W)||(weight + w[i + ۱] ≤ W))
}
```

۲.۵.۴ مسأله یافتن دور همیلتونی

```
void hamiltonian(index i)
    index j;
    if(promissing(i))
      if(i==n-1)
       cout ≪ vindex[0] through vindex[n-1];
    else
      for(j = 2 ;j ≤n ; j++){
        vindex[i+1] = j ;
        hamiltonian(i+1);}
    }
```

```
bool promissing(index i){
 index j;
 bool switch;
 if (i==n-1 && !w[vindex[n-1]][vindex[0]])
   switch=false;
 else if (i > 0 && !w[vindex[i-1]][vindex[i]])
   switch=false;
 else {
   switch=true;
   j=1;
   while(j <i && switch){
     if(vindex[i]==vindex[j])
       switch=false;
     j=j+1;}
       }
 return switch;}
```

۳.۵.۴ مسأله m-coloring

می‌خواهیم رئوس یک گراف که شامل n رأس می‌باشد را با m رنگ چنان رنگ آمیزی کنیم که هیچ دو رأس مجاوری همرنگ نباشند.

در این مسأله فاکتور شاخه m است .

```
void m-coloring (index i){
    int color;
    if (promissing(i))
      if (i == n)
        cout << vcolor[1] through vcolor[n]
      else
        for (color = 1; color <= m; color++) {
            volor[i+1] = color;
            m-coloring(i+1);}
  }
```

```
bool promissing (index i) {
    index j;
    bool switch = true;
    j = 1;
    while (j < i && switch) {
      if (W[i][j] && vcolor[i] == vcolor[j])
        switch = false;
        j++; }
    return switch;
}
```

۶.۴ تکنیک Branch and Bound (B&B)

یک تکنیک پیاده‌سازی الگوریتم هاست که شباهت فراوانی به روش Back Tracking دارد. معمولاً در پیاده‌سازی روش B&B یک Bound (مقدار اولیه) برای هر گره در نظر گرفته می‌شود، در نتیجه بایستی برای انتخاب هر مسیر و انتخاب هر گره توجه کنیم که اگر هزینهٔ رسیدن از این گره به گره مقصد بیشتر از باندی باشد که تا حالا در نظر گرفته‌ایم، آن مسیر را در نظر نگیریم. در واقع آن مسیر هرس می‌شود. غالباً برای پیاده‌سازی این روش از ساختار BFS استفاده می‌گردد.

فصل ۵

پویش گراف ها

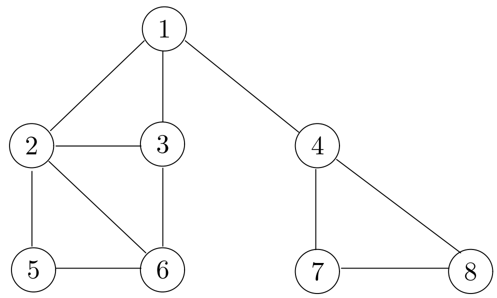
۵. پویش گراف ها Exploring graphs

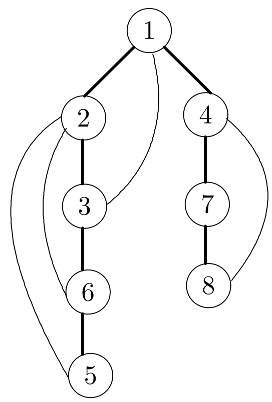
۱.۵ الگوریتم DFS (Depth First Search)

```
 procedure DF-Search(G=< N,A > )
for each v ∈ N do mark[v] ← not-visited
for each v ∈ N do
    if mark[v] ≠ visited then
        DFS(v)


 procedure DFS(v)
{Node v has not previously been visited}
mark[v] ← visited
for each node w adjacent to v do
    if mark[w] ≠ visited then
        DFS(w)
```

به عنوان مثال داریم :





1. DFS(1) $\Longrightarrow$ mark[1]=visited , mark[2] $\neq$ visited
2. DFS(2)
3. DFS(3)
4. DFS(6)
5. DFS(5)$\rightarrow$ اینجا همه نودهای مجاور visit شده‌اند.
6. DFS(4)
7. DFS(7)
8. DFS(8)

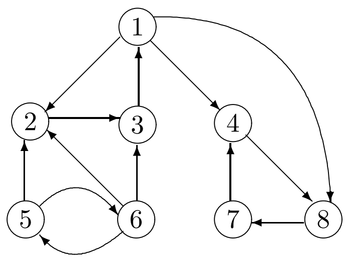
مثال :

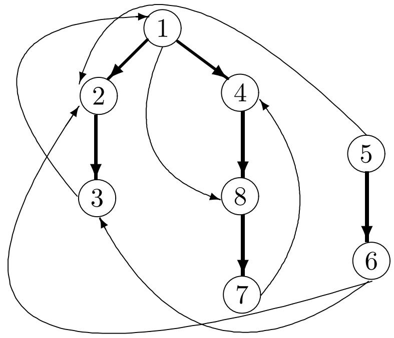
نشان دهید پیاده سازی از الگوریتم *DF-Search* وجود دارد که در زمان $\theta(|N|+|A|)$ پیاده سازی می شود. که N تعداد رئوس و A تعداد یالهاست .

حل:

در پاسخ باید خاطر نشان نمود که پیاده سازی الگوریتم بالا دقیقا به همان زمان ذکر شده نیاز دارد و می توان از پیاده سازی با لیست استفاده نمود.

همانطور که در مثال زیر مشهود است در یک گراف جهت دار ممکن است که با یک رأس نتوان تمام رئوس را پیمایش کرد.چنانچه گراف غیر همبند باشد واضح است که به اندازه تعداد مؤلفه های گراف بایستی روال *DF-Search* فراخوانی شود.





1. DFS(1)
2. DFS(2)
3. DFS(3)
4. DFS(4)
5. DFS(8)
6. DFS(7)
7. DFS(5)
8. DFS(6)

۱.۱.۵ پیاده سازی با پشته

```
Procedure DFS2(v)
p ←empty-stack
Mark[v]←visited
push v on to p
while p is not empty do
    while there exites a node w adjacent to top(p)
    such that mark[w]≠visited do
        mark[w]← visited
          push w on to p{w is the new top(p)}
   pop(p)
```

۲.۵ الگوریتم BFS (Breath First Search)

```
procedure    BF-Search(G)
  for each v ∈ N do Mark[v]← not-visited
   for each v ∈ N do
     if Mark[v] ≠ visited then
       BFS(v)

procedure BFS(v)
   Q← empty-queue
   mark[v] ← visited
   enqueue   v   into   Q
   While    Q is not empty   do
         u ← first(Q)
          dequeue   u   from   Q
          for   each node w adjacent to u   do
            if Mark[w] ≠ visited then
               Mark[w] ⟵ visited
               enqueue   w into   Q
```

زمان الگوریتم کمتر از $O(|N| + |A|)$ است زیرا تمام یال ها نیز به visit کردن ندارند.

به عنوان نمونه برای مثال اول داریم :

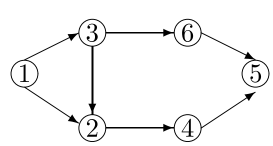
	node visited	Q
۱.	۱	~~۱~~, ۲, ۳, ۴
۲.	۲	~~۲~~, ۳, ۴, ۵, ٦
۳.	۳	~~۳~~, ۴, ۵, ٦
۴.	۴	~~۴~~, ۵, ٦, ۷, ۸
۵.	۵	~~۵~~, ٦, ۷, ۸
٦.	٦	~~٦~~, ۷, ۸
۷.	۷	~~۷~~, ۸
۸.	۸	~~۸~~

۳.۵ مرتب سازی توپولوژی Topological Sort

فرض کنید گرافی داریم که جهت دار و فاقد دور است (از رأس جلوتر به عقب تر یالی نداریم) در این صورت یک روش برای مرتب سازی رئوس این گراف آن است که به شیوه زیر عمل کنیم :

ابتدا رأسی را می یابیم که درجه ورودی آن صفر است. (پیمایش اول عمق است اما با شرط اضافه) آن رأس را انتخاب نموده سپس آن رأس و تمام یال های متصل به آن را حذف می کنیم . مجدداً رویه سابق را ادامه میدهیم . این روند را آنقدر ادامه میدهیم تا تمام رئوس گراف پیموده شوند . بدیهی است که این روش برگرفته از روش DFS و دقیقاً مشابه آن است ، اما با شرایط محدود کننده بیشتر .درنتیجه زمان اجرای آن در حد $\theta(|N| + |A|)$ است زیرا زمان شرایط محدود کننده از $\theta(۱)$ است.

باید توجه داشت که جواب منحصر به فرد نمی باشد و چند خروجی مختلف از این الگوریتم می توانیم داشته باشیم . به عنوان مثال داریم :



برای این مثال یکی از خروجی ها می تواند به‌این صورت باشد: 1,3,2,4,6,5

۴.۵ الگوریتم Bellman Ford

یکی از الگوریتم هایی که می تواند برای مسیریابی استفاده شود الگوریتمBellman Ford است . این الگوریتم برای یک گراف جهت دار که برچسب یالهای آن میتواند منفی باشد جهت یافتن کوتاهترین مسیر از گرهی به نام گره منبع تا سایر رئوس بکار می رود . البته این الگوریتم زمانی می تواند درست کار کند که دوری با وزن منفی نداشته باشد.

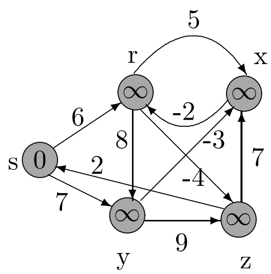
V[G] مجموعه رئوس گراف و G=$< V, E >$ و s راس منبع می باشند .

```
INITILIZE − SINGLE − SOURCE(G, s)
    for each vertex   v ∈ V[G]   do
        d[v] ← ∞
        π[v] ← NIL
    d[s] ← 0
```

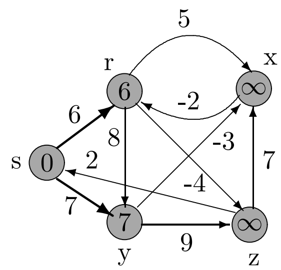
```
RELAX(u, v, w)
    if   d[v] >d[u]+w[u,v]   then
        d[v] ← d[u] + w[u, v]
        π[v] ← u
```

```
BELLMAN − FORD(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    for   i← 1   to (|V[G]|-1)   do
        for   each   edge   (u, v) ∈ E(G)   do
            RELAX(u,v,w)
    for   each   edge   (u, v) ∈ E(G)
        if d[v] > d[u]+w(u,v)   then
            return FALSE
    return TRUE        //θ(|V||E|)
```

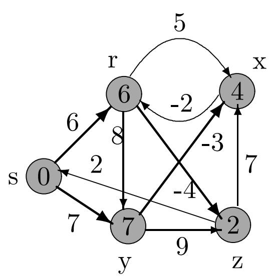
به عنوان مثال داریم :



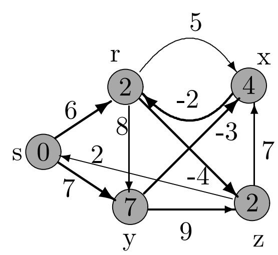
(a)



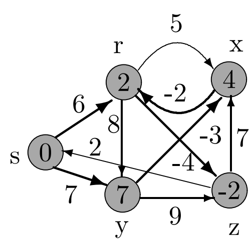
(b)



(c)



(d)



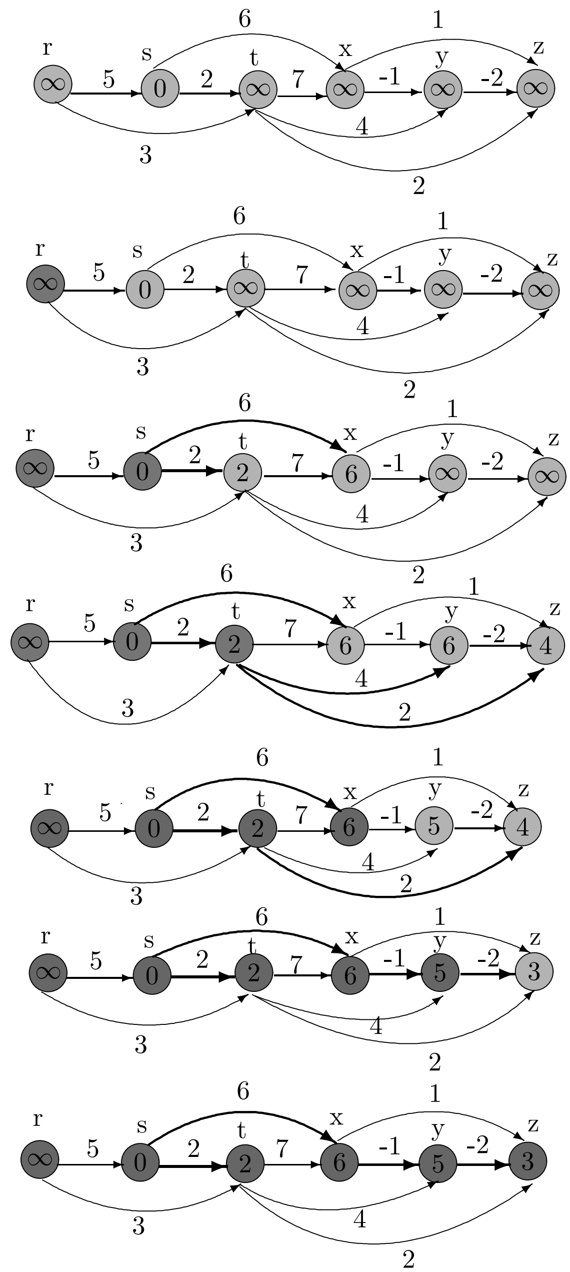
(e)

۵.۵ الگوریتم DAG

این الگوریتم کوتاه ترین مسیر از یک گره مشخص تا تک تک رئوس در یک گراف جهت دار فاقد دور را به ما می دهد.

```
    DAG-Shortest-Path(G,w,s)
1.  Topologically sort the vertics of G
2.  initialize-single-source(G,s)
3.  for each  vertex  u,taken in Topologically  sorted order do
        for  each  vertex v∈Adj[u]  do
                RELAX(u,v,w)
```

پیاده سازی این الگوریتم با لیست مجاورت است و زمان به علت مرتب سازی توپولوژیکال برابر$\theta(|V|+|E|)$ است. به مثال زیر دقت کنید:



فصل ٦

نگاهی مختصر بر ارائه دو سمینار

١.٦ LOOP INVARIANT

LOOP INVARIANT چیست ؟

این که ما بدانیم درهر حلقه چه اتفاقی می افتد کار بسیار مشکلی است .
حلقه های بی پایان یا حلقه هایی که بدون رسیدن به هدف مورد نظر پایان می پذیرند، یک مشکل رایج در برنامه نویسی کامپیوتر هستند.در این زمینه این Loop Invariant است که به ما کمک می کند.
در واقع ما از Loop Invariant ها استفاده می نماییم تا بدانیم آیا یک الگوریتم درست کار می کند یا خیر.
Loop Invariant یک یا گاهی چند عبارت فرما لیته است که رابطه بین متغیر ها در برنامه‌ی مارا بیان می کند که قبل از اجرای حلقه، هر زمان داخل حلقه و همچنین در انتهای حلقه درست باقی می ماند.

در این قسمت سعی بر آن است که مفاهیم اولیه LOOP INVARINT و طرز استفاده از آن به صورت خیلی مختصر توضیح داده شود و در انتها نیز چند مثال مختلف از استفاده از آن آورده شده است .

در این قسمت یک الگوی کلی استفاده از Loop Invariant را مشاهده

می نمایید:

```
    ...
// the Loop Invariant must be true here
while ( TEST CONDITION ) {
        //top of the loop
        ...
        //bottom of the loop
        //the Loop Invariant must be true here
}
// Termination + Loop Invariant ⇒ Goal
...
```

وقتی که مقادیر متغیر ها در حلقه تغییر کنند درستی Loop Invariant تغییری نمی کند. برای مشخص نمودن Loop Invariant باید به این نکته توجه نمود که عبارات بسیاری وجود دارند که قبل و بعد از هر تکرار حلقه ثابت و درست باقی می مانند اما عبارتی Loop Invarint است که با کار حلقه مرتبط است و از آن مهم تر این که با پایان یافتن حلقه درستی Loop Invariant ما را به نتیجه دلخواه و مورد انتظار از آن حلقه می رساند. در این قسمت چند اصطلاح را مشخص می نماییم :

Pre-condition:آن چیزی است که باید قبل از اجرای حلقه درست باشد.

Post-condition: آن چیزی است که بعد از اجرای کامل حلقه درست باقی می ماند و به ازای شرط خروج از حلقه در Loop Invariant به دست می آید که همان نتیجه مورد انتظار از حلقه است.

Loop Variant: شرطی است که اجرای حلقه را کنترل می کند و در واقع خروج از حلقه‌یا ادامه‌ی حلقه را کنترل می نماید.

برای چک کردن کار یک حلقه باید نکات زیر را مورد توجه قرار دهیم :

۱. *مطمئن شویم* Pre-condition *قبل از اینکه حلقه آغاز شود درست است .*

۲. *نشان دهیم* Loop Invariant *برای* Pre-condition *درست است.*

۳. *نشان دهیم اجرای* Loop Variant *اثر جانبی بر درستی* Loop Invariant *ندارد.*

۴. *نشان دهیم* Loop Invariant *بعد از هر اجرای حلقه درست است.*

۵ـ *نشان دهیم‌به مجرد پایان حلقه* Loop Invariant *دلالت بر درستی* Post-condition *دارد.*

٦ـ *نشان دهیم هر تکرار حلقه* Loop Variant *راافزایش یا کاهش می دهد.*

برای نشان دادن درستی مورد چهارم باید نشان دهیم‌اگر loop Invariant قبل از یک تکرار حلقه درست است قبل از تکرار بعدی نیز درست می ماند.

در این قسمت باید به نکته جالبی توجه نماییم و آن این است که اجرای مراحل ۴ـ۱ مشابه استقرای ریاضی است .در واقع با نشان دادن مورد ۲ پایه استقرا را بنا نهاده ایم وبا نشان دادن مورد ۴ گام های استقرا را پیموده ایم با این تفاوت که مورد ۵ باعث می شود که ما یک نوع استقرای محدود داشته باشیم .در واقع با پایان یافتن حلقه استقرا متوقف می گردد.
حال با ذکر چند مثال انجام مراحل فوق را نشان می دهیم :

۱)الگوریتمی که مجموع اعداد صحیح از 1 تا n را محاسبه می نماید:

```
int sum=0;
int k=0;
while(k < n){
    k++;
    sum+=k;
        }
```

....

$precondition : sum = 0, k = 0$

$$postcondition : sum = \sum_{i=1}^{n} i$$

$$loop\ \ invariant : sum_k = \sum_{i=1}^{k} i$$

INITIALIZATION:

همان طور که مشاهده می گردد loop invariant برای precondition درست است :

$$k = 0 \Rightarrow sum = \sum_{i=1}^{0} i = 0 = sum$$

MAINTENANCE:

حال نشان می دهیم‌اگر loop invariant برای مرحله j ام از تکرار حلقه درست باشد برای مرحله j+1 ام از تکرار حلقه نیز درست است :

$$sum_j = \sum_{j=1}^{k_{j+1}} j \quad , \quad k_{j+1} = k_j + 1$$

$$sum_{j+1} = sum_j + k_{j+1} = (\sum_{j=1}^{k_j} j) + k_{j+1} = (\sum_{j=1}^{k_j} j) + k_j + 1 = \sum_{j=1}^{k_{j+1}} j$$

TERMINATION:

حال شرط خروج از حلقه را چک می نماییم که به ازای آن loop invariant عبارت postcondition را به ما می دهد.شرط خروج از حلقه $k = n$ می باشدکه به ازای آن داریم :

$$sum = \sum_{i=1}^{n} i \equiv postcondition$$

۲)الگوریتمی که فاکتوریل عددی صحیح وبزرگتر از صفر را محاسبه می نمایید:

```
int factorial(n){
    i =1;
    fact =1;
    while(i ! = n){
      i++;
      fact=fact×i; }
     return fact;
                 }
```

....

$precondition : n \geq 1$

$loop\ \ invariant : fact = i\,!$

$postcondition : fact = n\,!$

....

INITIALIZATION:

$i = ۱ \Rightarrow fact = ۱! = ۱ \Rightarrow fact = i\,!$

MAINTENANCE:

$fact' = j'! \ \ , \ \ j = j' + ۱ \ \ , \ \ fact = fact' \times j$

$\Rightarrow fact = j'! \times j = j' \times (j' + ۱) = (j' + ۱)! = j! \Rightarrow fact = j!$

TERMINATION:

$i = n \ \ , \ \ fact = i! \Rightarrow fact = n! \equiv postcondition$

نکته : این الگوریتم ،الگوریتمی است که در آن اهمیت شرط precondition را به خوبی نشان می دهد،زیرا اگر شرط$n \geq ۱$ را در نظر نگرفته وn را برابر صفر بگیریم حلقه بی نهایت بار تکرار خواهد شدو ما را به نتیجه دلخواه نخواهد رساند.

۳)الگوریتمی که بزرگترین مقسوم علیه مشترک بین دو عدد صحیح بزرگتر از صفر را بر می گرداند:

```
int gcd(int m ,int n){
    int mprime = m;
    int nprime = n;
    while(mprime ! =nprime){
      if(mprime > nprime)
        mprime - = nprime;
      else
        nprime - = mprime;}
    return mprime;
                }
```

....

$precondition : m, n \in Z^{+}$

$loop\ \ invariant : gcd[m,n] = gcd[mprime, nprime]$

$postcondition : gcd[m,n] = mprime$

....

INITIALIZATION:

$mprime = m \quad nprime = n \Rightarrow gcd[m,n] = gcd[mprime, nprime]$

MAINTENANCE:

$gcd[m,n] = gcd[mprime_i, nprime_i]$

- $if(mprime_i > nprime_i):$
  $mprime_{i+1} = mprime_i - nprime_i, \quad nprime_{i+1} = nprime_i$
  $\Rightarrow gcd[mprime_{i+1}, nprime_{i+1}] = gcd[mprime_i - nprime_i, nprime_i] =$
  $gcd[mprime_i, nprime_i] = gcd[m,n]$

- $else:$
  $nprime_{i+1} = nprime_i - mprime_i, \quad mprime_{i+1} = mprime_i$
  $\Rightarrow gcd[mprime_{i+1}, nprime_{i+1}] = gcd[mprime_i, nprime_i - mprime_i] =$
  $gcd[mprime_i, nprime_i] = gcd[m,n]$

TERMINATION:

$mprime = nprime \Rightarrow gcd[m,n] = gcd[mprime, nprime] =$
$gcd[mprime, mprime] = mprime \equiv postcondition$

۴)الگوریتمی که k جمله‌ی اول بسط تیلور e^n را محاسبه می نماید:

1. double TaylorExp(double n ,int k){

```
double result =1;
int count =0;
int denom=1;
while (count<k){
  count ++;
  denom×=count;
  result+=pow(n,count)/denom;
            }
return result;
                    }
```

....

$precondition : n \in Z \quad k \in N, k > 0$

$loop\ \ invariant : result = 1 + n + \frac{n^2}{2!} + \cdots + \frac{n^{count}}{count!} \quad , denom = count!$

$postcondition : 1 + n + \frac{n^2}{2!} + \cdots + \frac{n^K}{K!} = result$

....

INITIALIZATION:

$count = 0, \quad denom = 1, \quad result = 1 \Rightarrow \frac{n^0}{0!} = 1 = result$

MAINTENANCE:

$result_i = 1 + n + \frac{n^2}{2!} + \cdots + \frac{n^{count_i}}{count_i!} \quad , denom_i = count_i!$

$\Rightarrow denom_{i+1} = denom_i \times count_{i+1} = (count_i!) \times count_{i+1} = (count_{i+1})!,$

$result_{i+1} = result_i + \frac{n^{count_{i+1}}}{(count_{i+1})!} = 1 + n + \frac{n^2}{2!} + \cdots + \frac{n^{count_i}}{count_i!} + \frac{n^{count_{i+1}}}{(count_{i+1})!}$

TERMINATION:

$count = k \Rightarrow result = 1 + n + \frac{n^2}{2!} + \cdots + \frac{n^K}{K!} \equiv postcondition$

۵)با استفاده از loop invariant نشان دهیدالگوریتم زیر جمع دو عدد طبیعی را انجام می دهد:

```
function add(y,z)
comment   return y + z, where y,z ∈ N
1.   x := 0 ; c := 0 ; d := 1 ;
2.   while(y > 0) ∨ (z > 0) ∨ (c > 0)do
3.     a := y mod 2;
       b :=z mod 2;
4.     if a ⊕ b ⊕ c   then x : = x+d ;
5.     c := (a ∧ b) ∨ (b ∧ c) ∨ (a ∧ c);
6.     d :=2 d;   y := ⌊ y /2 ⌋;
       z:=⌊ z /2 ⌋;
7.   return(x)
```

....

loop invariant : $(y_j + z_j + c_j)d_j + x_j = y_0 + z_0$

اگر y و z اولیه را y_0 و z_0 در نظر بگیریم قبل از شروع حلقه داریم :

$x_0 = 0, c_0 = 0, d_0 = 1 \Rightarrow$

$(y_j + z_j + c_j)d_j + x_j = (y_0 + z_0 + 0) \times 1 + 0 = y_0 + z_0$

حال فرض می نماییم loop invariant برای مرحله‌ی j ام برقرار است یعنی :

$(y_j + z_j + c_j)d_j + x_j = y_0 + z_0$

همچنین طبق روال حلقه داریم :

$a_{j+1} = y_j \bmod 2 \quad , b_{j+1} = z_j \bmod 2$

$y_{j+1} = \lfloor y_j / 2 \rfloor \quad , z_{j+1} = \lfloor z_j /2 \rfloor \quad , d_{j+1} = 2 \times d_j$

همچنین همان طور که در خط پنجم مشاهده می شود c_{j+1} که در این مرحله‌از a_{j+1} و b_{j+1} و c_j به دست می آید تنها وقتی یک است که دو تا از a_{j+1} و b_{j+1} و c_j یک باشند،پس می توان c_{j+1} را به صورت زیر نوشت :

$c_{j+1} = \lfloor (a_{j+1} + b_{j+1} + c_j) \ / \ 2 \ \rfloor$

به همین صورت هم مطابق خط چهارم وقتی x_j با d_j جمع شده و x_{j+1} را به وجود می آورند که عبارت $a_{j+1} \oplus b_{j+1} \oplus c_j$ یک باشد پس می توان x_{j+1} را به صورت زیر تعریف نمود:

$x_{j+1} = x_j + d_j((a_{j+1} + b_{j+1} + c_j) \bmod 2)$

برای‌ادامه اثبات نیاز به یک رابطه مهم دیگر که آن را رابطه (*) می نامیم وبه صورت زیر است نیز داریم :

$$2\lfloor \ n \ /2 \rfloor + (\ n \bmod 2) = n \quad \forall n \in \mathbf{N} \qquad (*)$$

حال طرف اول تساوی loop invariant را نوشته وداریم :

$$(y_{j+1} + z_{j+1} + c_{j+1})d_{j+1} + x_{j+1} =$$

$$(\lfloor y_j/2 \rfloor + \lfloor z_j/2 \rfloor + \lfloor (y_j \bmod 2 + z_j \bmod 2 + c_j)/2 \rfloor) \times 2d_j + x_j +$$

$$d_j \, ((y_j \bmod 2 + z_j \bmod 2 + c_j) \ \bmod 2) = (\lfloor y_j/2 \rfloor + \lfloor z_j/2 \rfloor) \times 2d_j$$

$$+x_j + (\lfloor (y_j \bmod 2 + z_j \bmod 2 + c_j)/2 \rfloor) \times 2d_j +$$

$$d_j \, ((y_j \bmod 2 + z_j \bmod 2 + c_j) \ \bmod 2) \underline{\underline{(*)}} (\lfloor y_j/2 \rfloor + \lfloor z_j/2 \rfloor) \times 2d_j$$

$$+x_j + d_j \, (y_j \bmod 2 + z_j \bmod 2 + c_j) = \lfloor y_j/2 \rfloor \times 2d_j +$$

$$d_j \, (y_j \bmod 2) + \lfloor z_j/2 \rfloor \times 2d_j + d_j \, (z_j \bmod 2) + c_j \times d_j + x_j \underline{\underline{(*)}}$$

$$(y_j + z_j + c_j) \, d_j + x_j = y_0 + z_0$$

حال زمانی را در نظر می گیریم که حلقه خاتمه پیدا می کند،اگر بعد از k تکرار حلقه خاتمه یابد طبق loop invarint داریم :

$(y_k + z_k + c_k)d_k + x_k = y_0 + z_0$

همچنین چون هر بار y وz در هر تکرار به مقادیر $\lfloor y/2 \rfloor$ و$\lfloor z/2 \rfloor$ کاهش می یابند زمان خروج از حلقه یعنی بعد از k امین تکرار داریم :

$y_k = z_k = c_k = 0 \Rightarrow x_k = y_0 + z_0$

پس مشاهده می شود که وقتی الگوریتم به پایان می رسد مقدار x ای که بر گردانده می شود برابر مجموع y و z اولیه‌ی ماست .

۶)با استفاده از loop invariant نشان دهیدالگوریتم زیر ضرب دو عدد طبیعی را انجام می دهد:

function *multiply(y,z)*

comment *return y z , where y , z* $\in$ **N**

1. $x := 0$
2. **while** $(z > 0)$**do**
3. $x := x + y \times (z \bmod 2);$
4. $y := 2\, y;\quad z := \lfloor z/2 \rfloor;$
5. **return**(x)

....

loop invariant : $x_j + y_j \times z_j = y_\circ \times z_\circ$

مطابق مثال قبل مقادیر اولیه y و z را $y_\circ$ و $z_\circ$ در نظرمی گیریم ، درابتدا $x_\circ = \circ$ بوده و داریم :

$x_\circ = \circ \Rightarrow x_j + y_j \times z_j = x_\circ + y_\circ \times z_\circ = y_\circ \times z_\circ$

حال فرض می نماییم loop invariant برای مرحله‌ی j ام برقرار است یعنی :

$x_j + y_j \times z_j = y_\circ \times z_\circ$

همچنین طبق روال حلقه داریم :

$x_{j+۱} = x_j + y_j(z_j \bmod 2) \quad y_{j+۱} = 2\ y_j \quad z_{j+۱} = \lfloor z_j/2 \rfloor$

پس داریم :

$x_{j+۱} + y_{j+۱} \times z_{j+۱} = x_j + y_j(z_j \bmod 2) + 2y_j(\lfloor z_j/2 \rfloor) =$
$x_j + y_j((z_j \bmod 2) + 2\lfloor z_j/2^{\rfloor)}) \overset{(*)}{=} x_j + y_j \times z_j = y_\circ \times z_\circ$

ودر انتها که حلقه پایان می یابد مقدار z برابر صفر خواهد شد که اگر این اتفاق در k امین تکرار حلقه رخ دهد داریم :

$$z_k = 0, x_k + y_k \times z_k = y_0 z_0 \Rightarrow x_k + y_k \times z_k = x_k + y_k \times 0 = x_k = y_0 \times z_0$$

پس مشاهده شد که وقتی الگوریتم به پایان می رسد مقدار x ای که برگردانده می شودبرابر ضرب مقادیر اولیه‌ی y و z خواهد بود و این همان نتیجه مطلوب و مورد انتظار ماست .

۷)با استفاده از loop invariant نشان دهیدالگوریتم زیر خارج قسمت و باقی مانده تقسیم دو عدد طبیعی را بر می گرداند:

function *divide(y,z)*

comment *return q, r* $\in \mathbf{N}$ *such that* $y = qz+r$

and $r < z$, *where* y , $z \in \mathbf{N}$

```
r :=y; q :=0; w :=z;
while w ≤y do w :=2 w;
while w > z do
  q :=2 q; w := ⌊ w /2 ⌋ ;
  if w ≤ r then
    r :=r - w; q :=q+1;
return (q , r)
```

....

loop invariant : $q_j w_j + r_j = y_0, r_j < w_j$

مقدار اولیه‌ی y را y_0 در در نظر می گیریم ،قبل از ورود به حلقه $q = 0$ و $r = y_0$ است پس داریم :

$$r = y_0 \quad , q = 0 \Rightarrow q_j w_j + r_j = 0 + y_0 = y_0$$

حال فرض می کنیم loop invariant برای مرحله‌ی j ام برقرار باشد،یعنی :

$q_jw_j + r_j = y_\circ, r_j < w_j$

حال برای تعیین وضعیت متغیر ها در تکرار بعدی لازم است قدری روی دو حلقه الگوریتم تامل نماییم،قبل از شروع حلقه اول w را برابر z در نظر گرفتیم ،در حلقه اول تا موقعی که شرط $w \leq y$ برقرار است w را دو برابر می نماید ،پس در انتهای این حلقه wای به دست می آید که‌از y بزرگتر است وهمچنین عددی زوج می باشد،حال وارد حلقه دوم می شویم ،در این حلقه تا موقعی که $z < w$ است w نصف شده و کف آن محاسبه می گردد،واضح است که همواره $\lfloor w/۲ \rfloor$ عددی زوج است و این به این خاطر است که در ابتدای امر ما$z = w$ در نظر گرفتیم ،اگر z زوج باشد w نیز زوج بوده و با هر بار دو برابر شدن نیز زوج باقی می ماند وبا هر بار نصف شدن نیز می توانیم علامت کف را نادیده بگیریم ،حال اگر مقدارz فرد باشد بعد از انتهای حلقه اول w ی زوجی در اختیار داریم ، اما در حلقه دوم نیز این w به دست آمده هر بار نصف شده وکف آن محاسبه می گردد و اگر بخواهد مقدار فردی داشته باشد برابر با خود z خواهد بود و این در حا لی است که شرط برقراری حلقه $z < w$ است ،پس در هر بار تکرار در این حلقه مقدار $\lfloor w/۲ \rfloor$ برابر با$w/۲$ می باشد.

حال در تکرار $j + ۱$ ام داریم : $\quad q_{j+۱} = ۲q_j \quad , w_{j+۱} = \lfloor w_j/۲ \rfloor$

حال باید دو حالت را بررسی نماییم :

(a) اگر شرط خط پنجم برقرار نباشد داریم :

$$w_{j+۱} > r_j \Rightarrow \begin{cases} ۱: \quad r_{j+۱} = r_j \quad , q_{j+۱}w_{j+۱} + r_{j+۱} = \\ \quad ۲q_j\lfloor w_j/۲ \rfloor + r_j = ۲q_j(w_j/۲) + r_j = q_jw_j + r_j = y_\circ \\ ۲: \quad w_{j+۱} > r_j \Rightarrow r_j < w_{j+۱} \quad , r_{j+۱} = r_j \\ \quad \Rightarrow r_{j+۱} < w_{j+۱} \end{cases}$$

(b) اگر شرط خط پنجم برقرار باشد داریم :

$$w_{j+1} < r_j \Rightarrow \begin{cases} 1: & r_{j+1} = r_j - w_{j+1} = r_j - \lfloor w_j/2 \rfloor, q_{j+1} = \\ & q_{j+1} + 1 = 2q_j + 1 \Rightarrow q_{j+1}w_{j+1} + r_{j+1} = \\ & (2q_j + 1)\lfloor w_j/2 \rfloor + r_j - \lfloor w_j/2 \rfloor = \\ & q_j w_j + \lfloor w_j/2 \rfloor + r_j - \lfloor w_j/2 \rfloor = y_0 \\ & \\ 2: & w_{j+1} < r_j, r_{j+1} = r_j - w_{j+1} \quad (1) \end{cases}$$

حال باداشتن (۱) باید اثبات نماییم $r_{j+1} < w_{j+1}$ ،برای این کار از برهان خلف استفاده نموده و فرض می نماییم $r_{j+1} \geq w_{j+1}$ پس داریم :

$r_j - w_{j+1} \geq w_{j+1} \Rightarrow r_j \geq 2w_{j+1} \Rightarrow r_j \geq 2\lfloor w_j/2 \rfloor \Rightarrow r_j \geq w_j$

که این نتیجه با فرض ما که همان برقرار بودن loop invariant برای مرحله‌ی j ام بود تناقض داشته و داریم: $r_{j+1} < w_{j+1}$ ، به این ترتیب قسمت دوم loop invariant نیز برای مرحله‌ی j+1 ام نیز برقرار است .

در انتها نیز وقتی از حلقه‌ی دوم خارج شده والگوریتم بعد از k تکرار به پایان می رسد داریم $w_k = z_0$.

پس طبق loop invariant داریم : $r_k < z_0 \quad , q_k z_0 + r_k = y_0$ پس الگوریتم خارج قسمت وباقی مانده‌ی تقسیم دو عدد صحیح را برای ما برگرداند.

۸)با استفاده از loop invariant نشان دهیدالگوریتم زیر عددی حقیقی را به توان عددی طبیعی می رساند:

function *power(y,z)*

comment *return* y^z, *where* $y \in \mathbf{R}, z \in \mathbf{N}$

```
x :=1;
while z > 0 do
  if z is odd then x := x.y;
  z :=⌊z /2 ⌋;
  y :=y^2;
 return (x)
```

loop invariant : $x_j {y_j}^{z_j} = {y_0}^{z_0}$

مقادیر اولیه y و z را y_0 و z_0 در نظرمی گیریم ، در ابتدا $x_0 = 1$ بوده وطبق loop invariant داریم :

$$x_j {y_j}^{z_j} = {y_0}^{z_0}$$

حال بر اساس loop invariant برای مرحله j ام و طبق روال حلقه برای مرحله j+1 ام داریم :

$$x_j {y_j}^{z_j} = {y_0}^{z_0},$$

$$\begin{cases} \text{اگر z فرد باشد} \Rightarrow x_{j+1} = x_j y_j, \quad z_{j+1} = \lfloor z_j/2 \rfloor, \quad y_{j+1} = {y_j}^2 \\ \text{اگر z زوج باشد} \Rightarrow x_{j+1} = x_j, \quad z_{j+1} = \lfloor z_j/2 \rfloor, \quad y_{j+1} = {y_j}^2 \end{cases}$$

پس داریم :

$$\begin{cases} \text{اگر z فرد باشد} : \quad x_{j+1} {y_{j+1}}^{z_{j+1}} = x_j y_j \times (y_j)^{2 \times \lfloor z_j/2 \rfloor} = \\ \quad x_j y_j \times (y_j)^{2 \times (z_j - 1)/2} = x_j {y_j}^{z_j} = {y_0}^{z_0} \\ \\ \text{اگر z زوج باشد} : \quad x_{j+1} {y_{j+1}}^{z_{j+1}} = x_j \times (y_j)^{2 \times \lfloor z_j/2 \rfloor} = \\ \quad x_j \times (y_j)^{2 \times (z_j)/2} = x_j {y_j}^{z_j} = {y_0}^{z_0} \end{cases}$$

در انتهای کار نیز حلقه موقعی به پایان می رسد که در یک مرحله مانند مرحله k ام داشته باشیم z=0 پس داریم :

$$x_k \times {y_k}^{z_k} = {y_0}^{z_0}, z_k = 0 \Rightarrow x_k = {y_0}^{z_0}$$

بدین ترتیب x ای که در انتهای الگوریتم به ما تحویل داده می شود y اولیه به توان z اولیه است ،پس این الگوریتم نیز نتیجه‌ی مطلوب را به ما داد.

۹)با استفاده از loop invariant نشان دهیدالگوریتم زیر مقادیر موجود در آرایه $A[۱\cdots n]$ رابا یکدیگر جمع می نماید:

```
function sum(A)
  comment   return Σ_{i=۱}^{n} A[i]
    s :=0;
    for i := 1 ton do
      s :=s+A[i]
     return (s)
```

....

$$\textbf{loop invariant} : s_j = \sum_{i=1}^{j} A[i]$$

قبل از ورود به حلقه مجموع عناصر آرایه برابر صفر است و ما نیز طبق loop invariant داریم :

$$j = ۰ \Rightarrow s = s_۰ = \sum_{i=۱}^{۰} A[i] = ۰$$

حال فرض می کنیم loop invarint برای مرحله‌ی j ام برقرار باشد ،برای مرحله‌ی j+1 ام داریم :

$$s_{j+۱} = s_j + A[j+۱] = (\sum_{i=۱}^{j} A[i]) + A[j+۱] = \sum_{i=۱}^{j+۱} A[i]$$

در هنگام خروج از حلقه نیز داریم :

$$j = n \Rightarrow s = s_n = \sum_{i=۱}^{n} A[i]$$

که همان نتیجه‌ی مورد انتظار ما از الگوریتم است .

۱۰) با استفاده از loop invariant نشان دهید الگوریتم زیر مقدار چند جمله‌ای $a_n x^n + a_{n-1}x^{n-1} + \cdots a_1 x + a_0$ را با استفاده از روش Horner محاسبه می کند که در آن ضرایب در آرایه A[0..n] ذخیره شده‌اند:

$A[i] = a_i \quad for \quad all \quad 0 \le i \le n$

function *Horner(A , n)*

comment *return* $\sum_{i=0}^{n} A[i].x^i$

1. *v := 0*
2. **for** *i := n* **downto** *0* **do**
3. *v := A[i]+v.x*
4. **return** *(v)*

....

$$\textbf{loop invariant}: v_j = \sum_{i=j}^{n} A[i]x^{i-j}$$

باید توجه نمود که این الگوریتم از حلقه‌ی for کاهشی استفاده می نماید ،لذا ما loop invariant را به صورت بالا در نظر گرفته و قبل از ورود به حلقه داریم j=n+1 پس طبق loop invariant داریم :

$$j = n+1 \Rightarrow v = \sum_{i=n+1}^{n} A[i]x^{i-(n+1)} = 0$$

حال اگر loop invariant برای مرحله‌ی j ام برقرار باشد ،در مرحله‌ی بعدی داریم :

$$v_j = A[j] + v_{j+1}.x \Rightarrow v_{j+1} = \frac{v_j - A[j]}{x} = \frac{\sum_{i=j}^{n} A[i]x^{i-j} - A[j]}{x} = \frac{\sum_{i=j+1}^{n} A[i]x^{i-j}}{x} =$$

$$\sum_{i=j+1}^{n} A[i]x^{i-j-1} = \sum_{i=j+1}^{n} A[i]x^{i-(j+1)}$$

در هنگام خروج از حلقه j = 0 شده و داریم : $j = 0 \Rightarrow v = \sum_{i=0}^{n} A[i]x^i$ که همان نتیجه‌ی مورد انتظار ماست .

۲.٦ آنالیز استهلاکی (Amortized Analysis)

یکی از روشهای آنالیز یک مجموعه از عملیات ، آنالیز استهلاکی است . در آنالیز استهلاکی زمان اجرای یک مجموعه از عملیات ساختمان داده ای روی تمام عملیات اجرا شده سرشکن می شود . از این روش برای نشان دادن این نکته استفاده می کنیم که هزینه متوسط هر عمل کوچک است حتی اگر یک عمل درون یک مجموعه هزینه زیادی داشته باشد . اگر چه ما در مورد میانگین و متوسط صحبت می کنیم اما آنالیز استهلاکی با آنالیز در حالت میانگین تفاوت دارد و روش های آماری را شامل نمی شود . یک آنالیز استهلاکی زمان اجرای متوسط هر عمل در بدترین حالت را ضمانت می کند .

انوع آنالیز استهلاکی

سه روش از پر کاربردترین روش های آنالیز استهلاکی عبارتند از:

- آنالیز تجمعی (Aggregate Analysis)
- روش حسابی (Acconting Method)
- روش پتانسیل (Potential Method)

۱.۲.٦ آنالیز تجمعی (Aggregate Analysis)

در آنالیز تجمعی نشان می دهیم به ازای تمام n ها،یک مجموعه از n عملیات در بدترین حالت ،مجموعاً $T(n)$ را می گیرد . بنابر این در بدترین حالت ، هزینه متوسط یا هزینه استهلاکی هر عملیات $\frac{T(n)}{n}$ است . توجه کنید که این هزینه متوسط برای هر عملیات صادق است ، حتی وقتی که چندین نوع از عملیات در مجموعه موجود هستند . اما در دو روش بعدی ممکن است هزینه های متوسط متفاوتی را به عملیات های مختلف نسبت دهیم . این روش اگر چه ساده است اما دقت دو روش بعدی را ندارد . در عمل روش های حسابی و پتانسیل یک هزینه استهلاکی مخصوص به هر عمل اختصاص می دهند .

مثال ۱ : در اولین مثال از آنالیز تجمعی ، ساختمان داده پشته [۱] را آنالیز می کنیم .
دو عمل اصلی پشته که از $O(۱)$ هستند عبارتست از: $\text{push}(S,x)$: عنصر x را وارد پشته S میکند.
$\text{pop}(S)$: بالاترین عنصر پشته S را برداشته ، آن را برمی گرداند.
از آنجایی که هر کدام از این عملیات ها در زمان $O(۱)$ اجرا می شوند فرض می کنیم هزینه هر کدام ۱ باشد . بنابراین هزینه نهایی یک مجموعه از n عملیات push و pop ، n است .
حال ما یک عمل $\text{multipop}(S,k)$ را به پشته اضافه می کنیم که k عنصر را از بالای پشته S برمیدارد . و یااگر پشته S ، کمتر از k عنصر داشته باشد ، آن را خالی می کند .
در شبه کد زیر تابع Stack-Empty مقدار TRUE می گیرد اگر هیچ عنصری در پشته موجود نباشد ، در غیر این صورت مقدار FALSE را برمی گرداند .

```
   Multipop(S, k)
1. while   not   Stack-Empty(S)   and   k ≠ ۰
2.         do pop(S)
3.         k ← k − ۱
```

در اینجا یک مجموعه از n عملیات push و pop و multipop را روی یک پشته که در ابتدا خالیست آنالیزمی کنیم ، دراین مجموعه در بدترین حالت اگر سایز پشته حداکثر n باشد هزینه عمل multipop از $O(n)$ است . در بدترین حالت زمان اجرای هر عملیات در پشته از $O(n)$ است بنابراین هزینه کل $O(n^۲)$ می شود .
اگر چه این تحلیل درست است اما نتیجه به دست آمده با توجه به هزینه در بدترین حالت محکم نیست . بااستفاده از روش آنالیز تجمعی میتوانیم با توجه به مجموعه شامل n عملیات یک کران بالایی بهتر بدست بیاوریم . در حقیقت ، اگر چه عمل multipop به تنهایی هزینه زیادی می برد اما هر مجموعه از n عملیات push و pop و multipop روی یک پشته در ابتدای خالی حداکثر ، هزینه $O(n)$ را دارد .
چون هر عنصر به ازای هر بار ورود به پشته فقط می تواند یکبار از پشته برداشته شود . بنابراین ، تعداد فراخوانی تابع pop (با در نظر گرفتن multipop) ، روی یک پشته غیر تهی به تعداد فراخوانی تابع push است که حداکثر برابر n است . به

[۱] stack

ازای هر n ، هر مجموعه از n push ، pop و multipop زمان کل $O(n)$ را می گیرد . هزینه میانگین یا سرشکن هر عمل $O(۱) = \frac{O(n)}{n}$ است . در آنالیز تجمعی ، هزینه استهلاکی برای هر عمل در واقع همان هزینه متوسط ان عمل است . بنابراین در این مثال هزینه استهلاکی هر ۳ عمل پشته $O(۱)$ است .
دوباره تأکید می شود که اگر چه ما هزینه متوسط را محاسبه کرده ایم اما از روش های آماری استفاده نکرده ایم .

مثال ۲ : مثال دیگر از این روش مسأله شمارنده دودویی k-bit افزایشی[۲] است . یک آرایه $A[۰..k-۱]$ به طول length[A]=k به عنوان شمارنده است . عدد دودویی x که در شمارنده ذخیره می شود دارای کمترین ارزش در $A[۰]$ و بیشترین ارزش در $A[k-۱]$ است. عمل افزایش توسط تابع زیر صورت می گیرد :

```
   INCREMENT(A)
i ← ۰
while    i < length[A]   and   A[i] = ۱
       do    A[i] ← ۰
             i ← i + ۱
if    i < length[A]
       then    A[i] ← ۱
```

اجرای INCREMENT به تنهایی در بدترین حالت (زمانی که همه بیت ها ۱ باشند)زمان $\Theta(k)$ را می گیرد بنابرین یک رشته از n عمل INCREMENT در بدترین حالت روی یک شمارنده که در ابتدا صفر است ، زمان $O(nk)$ را می گیرد .
با توجه به این که در هر فراخوانی همه بیت ها تغییر نمی کنند می توانیم کران دقیقتری ارائه دهیم به طوری که در بدترین حالت اجرای یک رشته از n عملیات INCREMENT از $O(n)$ شود . $A[۰]$ در هر بار فراخوانی تابع تغییر می کند ، $A[۱]$ در n بار اجرای تابع هر $\lfloor \frac{n}{۲} \rfloor$ بار تغییر می کند و $A[۲]$ هر $\lfloor \frac{n}{۴} \rfloor$ بار تغییر می کند .
به طور کلی ، برای $i = ۰, ۱, ۲, \cdots, \lfloor \log_۲^n \rfloor$ ، بیت $A[i]$ ، هر $\lfloor \frac{n}{۲^i} \rfloor$ بار تغییر می کند .
برای $i > \lfloor \log_۲^n \rfloor$ ، بیت $A[i]$ هرگز تغییر نمی کند .

[۲] Incrementing a Binary Counter

تعداد کل تغییر ها در مجموعه برابر است با:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

بنابراین هزینه متوسط هر عملیات $\frac{O(2n)}{n} = O(1)$ می شود .

۶.۲.۲ روش حسابی (Accounting Method)

در روش حسابی از آنالیز استهلاکی ، به عملیات های مختلف هزینه های متفاوتی اختصاص داده می شود که این شارژ گاهی ممکن است از هزینه واقعی عمل کمتر یا بیشتر باشد. هزینه ای که به عنوان شارژ روی یک عمل ذخیره می شود را هزینه استهلاک گوییم . هنگامی که هزینه استهلاکی بیشتر از هزینه واقعی عمل باشد ، اختلاف به عنوان اعتبار آن عمل به خصوص در ساختمان داده در نظر گرفته می شود . اعتبار می تواند بعداً در پرداخت های بعدی برای عملیات هایی که هزینه استهلاکشان کمتر از هزینه واقعی است استفاده شود . این روش با روش تجمعی بسیار متفاوت است .

ابتدا باید هزینه استهلاکی هر عملیات بدقت مشخص شود . اگر ما می خواهیم از روش استهلاکی برای اثبات اینکه هزینه متوسط در بدترین حالت هر عملیات کوچک است ، استفاده کنیم باید هزینه استهلاکی کلی یک کران بالایی برای هزینه واقعی کل باشد .

اگر هزینه واقعی عمل i ام را با C_i و هزینه استهلاکی عمل i ام را با $\hat{C}_i$ مشخص کنیم ، داریم :

$$\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i$$

اعتبار نهایی ذخیره شده در ساختمان داده اختلاف بین هزینه واقعی و استهلاکی است .

$$\sum_{i=1}^{n} \hat{C}_i - \sum_{i=1}^{n} C_i$$

این مقدار باید همیشه غیر منفی باشد . اگر اعتبار نهایی منفی شد آنگاه هزینه استهلاکی نهایی زیر هزینه واقعی کل قرار می گیرد و در نتیجه هزینه استهلاکی کل یک کران بالا برای هزینه واقعی نخواهد بود . بنابراین باید مراقب باشیم اعتبار نهایی

در ساختمان داده منفی نشود .

مثال ۱: مثال پشته را در نظر بگیرید یادآوری می کنیم که هزینه واقعی عملیات به صورت زیر است :

Push : 1
Pop : 2
Multipop : 3

فرض کنید مقادیر استهلاکی برای هر عمل بصورت زیر باشد .

Push : 2
Pop : 0
Multipop : 0

توجه کنید اگر چه هزینه واقعی multipop متغیر است اما هزینه استهلاکی تابع صفر است وقتی یک عنصر را وارد پشته می کنیم ۱ واحد (به اندازه هزینه واقعی) از هزینه استهلاکی برای انجام این کار می پردازیم و یک واحد باقی مانده در شئ ذخیره می شود .

این اعتبار ذخیره شده روی شئ در واقع هزینه برداشتن شئ از پشته است . وقتی تابع pop فراخوانی می شود هزینه برداشتن عنصر از این اعتبار ذخیره شده تامین می شود . بنابر این ما همیشه اعتبار کافی برای انجام عمل pop یا multipop روی شئ را داریم . تا زمانی که عنصری در پشته است اعتبار هیچ گاه منفی نمی شود .

برای هر مجموعه از n عمل push و pop و multipop هزینه استهلاک $O(n)$ است که کران بالا برای هزینه واقعی نیز هست .

مثال ۲: مثال شمارنده دودویی افزایشی را بررسی می کنیم . قبلاً دیدیم که زمان اجرای این تابع متناسب با تعداد بیت هایی است که تغییر می کنند . هزینه استهلاکی تغییر یک از صفر به یک را ۲ در نظر می گیریم . هنگامی که یک بیت ۱ می شود یک واحد پرداخت می شود و واحد دیگر اعتبار برای زمانی که بیت را به صفر تغییر دهیم ذخیره می شود . در هر زمانی از اجرا ، هر ۱ در شمارنده یک واحد اعتبار ذخیره دارد بنابراین برای صفر کردن آن ، اعتبار لازم موجود است و نیازی به اعتبار جدید نیست و چون تعداد یک ها در شمارنده هیچ گاه منفی نیست پس اعتبار نهایی نیز هیچ گاه منفی نمی شود . بنابراین هزینه سرشکن $O(n)$ است که کران بالا برای هزینه واقعی است .

۳.۲.۶ روش پتانسیل (Potential Method)

سومین روش آنالیز سرشکنی ، روش پتانسیل است که بر خلاف روش های دیگر که روی یک شئ مشخص در یک ساختمان داده کار می کردند ، روش پتانسیل روی ساختمان داده ها کامل کار می کند . فرق آن با روش حسابی اینست که مقدار اضافی ذخیره شده یا همان سود در اینجا به عنوان انرژی پتانسیل تعریف می شود .
اگر تعداد اجراهای ساختمان داده از ۱ تا n باشد و D_0 ساختمان داده اولیه باشد برای هر $i = 1, 2, \cdots, n$ تعریف می کنیم :

$C_i =$ زمان واقعی انجام عملیات i ام

$D_i =$ ساختمان داده بعد از انجام عملیات i ام

$\Phi =$(Potential Function) تابع پتانسیل که هر ساختمان داده D_i را به اعداد حقیقی می برد

$\Phi : D_i \to RealNumber$

$\hat{C}'_i = C_i + \Phi(D_i) - \Phi(D_{i-1})$ هزینه سرشکنی در مرحله i ام

هزینه استهلاکی کل برای n عملیات:

$$\sum_{i=1}^{n} \hat{C}_i = \sum_{i=1}^{n} [C_i + \Phi(D_i) - \Phi(D_{i-1})] = \sum_{i=1}^{n} C_i + \sum_{i=1}^{n} \Phi(D_i) - \Phi(D_{i-1})$$

$$= \sum_{i=1}^{n} C_i + \Phi(D_n) - \Phi(D_0)$$

اگرفرض کنیم $\Phi(D_n) > \Phi(D_0)$ بنابراین هزینه استهلاکی کل کران بالایی برای هزینه واقعی کل است . چون ما نمی دانیم چند عملیات ممکن است انجام شود . بنابراین اگر فرض کنیم برای هر i :

$$\left.\begin{array}{l} \Phi(D_i) \geq \Phi(D_0) \\ \Phi(D_0) = 0 \end{array}\right\} \Rightarrow \Phi(D_i) \geq 0$$

یعنی تابع پتانسیل یک تابع غیر منفی است . بنابراین تغییرات پتانسیل برابر است با:

$\Phi(D_i) - \Phi(0) > 0 \qquad \text{for all i}$

هزینه استهلاکی بدست آمده در اینجا به انتخاب تابع پتانسیل وابسته است . به ازای تابع پتانسیل‌های مختلف هزینه های استهلاکی مختلف خواهیم داشت . بنابراین مهمترین کار ما در روش پتانسیل انتخاب بهترین تابع پتانسیل است .

مثال ۱ : مثال پشته را بررسی می کنیم . تعریف می کنیم :

$\Phi(D_i) =$ تعداد عناصر داخل پشته

$D_\circ =$ پشته خالی

$\Phi(D_\circ) = \circ$

چون عناصر داخل یک آرایه هیچ گاه منفی نمی شود پس

$\Phi(D_i) \geq \circ = \Phi(D_\circ)$

فرض می کنیم در مرحله $i - ۱$ ، پشته s عنصر داشته باشد

$\Phi(D_{i-۱}) = s$

اگر push درمرحله i ام اجرا شود:

$\Phi(D_i) - \Phi(D_{i-۱}) = (s+۱) - s = ۱$

$\hat{C}_i = C_i + \Phi(D_i) - \phi(D_{i-۱}) = ۱ + ۱ = ۲ \rightarrow O(۱)$

اگر تابع pop در مرحله i ام انجام شود:

$\Phi(D_i) - \Phi(D_{i-۱}) = (s-۱) - s = -۱$

$\hat{C}_i = C_i + \Phi(D_i) - \phi(D_{i-۱}) = ۱ - ۱ = \circ \rightarrow O(۱)$

اگر تابع multipop اجرا شود:

$k' = min(s,k)\ \Phi(D_i) - \Phi(D_{i-۱}) = (s-k') - s = -k'$

$\hat{C}_i = C_i + \Phi(D_i) - \phi(D_{i-۱}) = k' - k' = \circ \rightarrow O(۱)$

بنابراین هزینه استهلاکی کل از $O(n)$ است .

مثال ۲: در شمارنده دودویی داریم :

$\Phi(D_i) = b_i$ تعداد یک ها در مرحله i ام

واضح است که

$\Phi(D_i) \geq \circ$

$t_i =$ تعداد یک هایی که به صفر تبدیل می شوند در مرحله i ام

$C_i = t_i + ۱$

$$\left.\begin{array}{l} if \quad b_i = ۰ \Rightarrow b_{i-۱} = k = t_i \\ if \quad b_i > ۰ \Rightarrow b_i = b_{i-۱} - t_i + ۱ \end{array}\right\} \Rightarrow b_i \leq b_{i-۱} - t_i + ۱$$

$$\Phi(D_i) - \Phi(D_{i-۱}) \leq b_{i-۱} - t_i + ۱ - b_{i-۱} = -t_i + ۱$$

$$C'_i = C_i + \Phi(D_i) - \Phi(D_{i-۱}) \leq t_i + ۱ - t_i + ۱ = ۲ \Rightarrow C'_i \leq ۲$$

$$\sum_{i=۱}^{n} C'_i \leq ۲n \rightarrow O(n)$$