الگوریتم دکتر باقری

نیمسال اول سال 1399

علی فرجی 9634024

سرفصل ها

۱- مقدمه

۲- حل روابط بازگشتی

a- روش جایگذاری اصلی

b- روش درخت بازگشتی

۳- تحلیل زمان مرتب سازی پشته ای و مرتب سازی سریع

۴- الگوریتم های مرتب سازی زمان خطی

a- Counting Sort

b- Radix sort

c- Bucket Sort

۵- پیدا کردن میانه و مسأله ی selection

۶- تحلیل سرشکن یا amortized analysis

۷- روش های طراحی الگوریتم

a- روش تقسیم و غلبه

b- برنامه نویسی پویا

c- روش عقبگرد

d- روش branc & bound

۸ - مرور الگوریتم های گراف

a- DFS

b- BFS

c- کوتاهترین مسیر

I - Floyd

II - Dijkstra

d- MST

I - Kruskal

II - Prim

۹- الگوریتم های جریان بیشینه (به علت اینکه خیلی از مسائل به این مسئله مدل می شوند خیلی کاربرد دارد)

۱۰- پیچیدگی محاسبات

a- پیدا کردن مینیمم

b- مرتب کردن

c- مسائلی فروشنده دوره گرد (یک مسائلی باز است)

تمرین و پروژه

کوئیز

میان ترم

پایان ترم

مقدمه

الگوریتم: مجموعه‌ای از دستورات کامپیوتر است که به ترتیب مشخص اجرا می‌شود تا یک مساله حل شود و حتماً خاتمه یابد.

* حل مساله توسط کامپیوتر نیازمند الگوریتم است.

* آیا الگوریتم ما صحیح است؟ باید Correcness Proof انجام دهیم.
* تحلیل زمانی الگوریتم
* تحلیل فضایی (حافظه‌ی مورد نیاز) الگوریتم

جایگاه الگوریتم:

طراحی الگوریتم ← پیاده سازی الگوریتم ← اجرای روی کامپیوتر

سخت افزار و نرم افزار هر دو مهمند

مثلاً مرتب سازی $n = 10^6$ عدد:

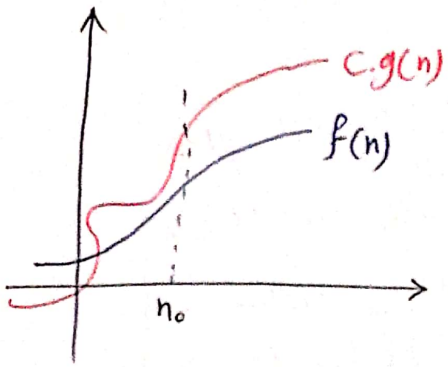
	الگوریتم A روی کامپیوتر A	الگوریتم B روی کامپیوتر B
سرعت:	10^7 Ins/sec	10^9 Ins/sec
زمان:	$n\log_2 n$	n^2
زمان حل:	$\frac{10^6 \times \log 10^6}{10^7} \simeq$	$\frac{10^6 \times 10^6}{10^9} \simeq$

* کامپیوتری ۱۰۰ برابر کندتر از کامپیوتر دیگر است ۱۰۰۰ برابر بهتر عمل می‌کند. (الگوریتم وقتی مساله بزرگ می‌شود خیلی اثرگذار است)
* ما در این درس روی تحلیل زمانی تمرکز می‌کنیم با اینکه تحلیل حافظه نیز بسیار مهم است.
* در تحلیل زمان الگوریتم جنس ما زمان نیست بلکه تعداد دستورات است.

⊛ رشد توابع ($o, O, \Theta, \Omega, \omega$)

بیگ اُ :

$$f(n) \in O(g(n)) \Longleftrightarrow \exists c>0, n_0>0 \mid \forall n \geq n_0 : f(n) \leq c.g(n)$$



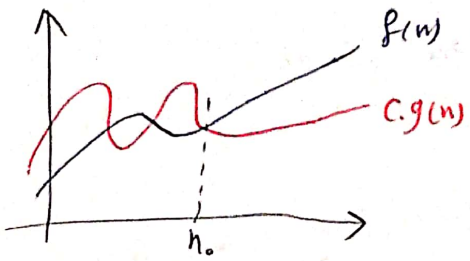
* یعنی رشد g حد بالای رشد f است.

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \begin{cases} \nearrow c \neq 0 & \text{ثابت} \\ \searrow \infty \end{cases}$$

$$f(n) = 3n+10 \in O(n^2)$$
$$\in O(n)$$
$$\notin O(\sqrt{n})$$

بیگ اُمگا Ω :

$$f(n) \in \Omega(g(n)) \Longleftrightarrow \exists c>0, n_0>0 \mid \forall n>n_0 : f(n) \geq c.g(n)$$



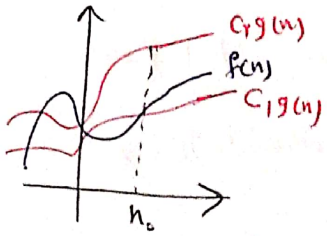
* یعنی رشد g حد پایین رشد f است.

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \begin{cases} \nearrow c \neq 0 & \text{ثابت} \\ \searrow 0 \end{cases}$$

$$f(n) = 3n+10 \in \Omega(\sqrt{n})$$
$$\in \Omega(n)$$
$$\notin \Omega(n^2)$$

تتا Θ :

$$f(n) \in \Theta(g(n)) \Leftrightarrow \exists C_1, C_2 > 0 \text{ و } n_0 > 0 \mid \forall n \geq n_0 : C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$$



* یعنی رشد تابع f و g با هم برابر است.

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = c \neq 0$$

$$f(n) = 10 + 3n \in \Theta(n)$$
$$\in \Theta(2n + 10000)$$
$$\in \Theta(n+1)$$
$$\notin \Theta(\sqrt{n})$$
$$\notin \Theta(n^2)$$

لیتل او o :

$$f(n) \in o(g(n)) \Leftrightarrow \forall c > 0 \ \exists n_0 > 0 \mid \forall n > n_0 : f(n) \leq c \cdot g(n)$$

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty$$

* یعنی رشد تابع g اکیداً از f بیشتر است.

$$f(n) = 3n + 10 \in O(n^2)$$
$$\notin O(n)$$
$$\notin o(\sqrt{n})$$

لیتل اومگا ω :

$$f(n) \in \omega(g(n)) \Leftrightarrow \forall c > 0, \exists n_0 > 0 \mid \forall n > n_0 : f(n) \geq c \cdot g(n)$$

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

* یعنی و یک حد پایین اکید برای رشد f است.

← اومگا برای گفتن سختی مساله است و عملاً مشخص کردن آن اهمیتی ندارد.

← تنها چیزی است که می‌توانیم برای یک الگوریتم مشخص کنیم.

مفهوم اندازه‌ی مساله:

* مثلاً برای مساله‌ی مرتب سازی n عدد

اندازه مساله = n

زمان الگوریتم = $n\log n$

* مثلاً برای ضرب دو عدد n بیتی:

اندازه مساله می‌شود تعداد بیت ها یا همان 2n.

نکته‌ی مهم: سایز مساله همیشه n است، سوال این است که در هر مساله سایز مساله یا همان n چیست؟

مثال: ما n شعر داریم و می‌خواهیم فاصله را حساب کنیم مثلا یک ماتریس nxn برای نگه‌داری اطلاعات نیاز داریم. اندازه مساله = $N = n\times n$

مثال: اگر یک مساله داشتیم که در آن یک عدد طبیعی ورودی باشد که بزرگی عدد n است پردازش می‌شود:

سایز مساله ← $N=\lg(n)$ تعداد بیت ها

* حال اگر زمان الگوریتم $T(n)=n$ باشد پس زمان مساله برحسب اندازه مساله می‌شود:

$T(N)=2^N$

* تابع زمان الگوریتم تابعی از اندازه‌ی مساله است و اندازه‌ی مساله برابر است با تعداد بیت لازم برای کدگذاری ورودی مساله در یک رشته‌ی دودویی.

⊛ حل روابط بازگشتی:

* در اکثر مسایل خود مساله برحسب خودش ولی با اندازه‌ی کوچکتر تعریف می‌شود.
* این گونه مسایل به ما یک رابطه‌ی بازگشتی می‌دهد که برای پیدا کردن زمان الگوریتم باید حل روابط بازگشتی را بلد باشیم که به حالت بسته بنویسیم.

مثال: برای مرتب سازی ادغامی داریم: $T(n) = 2T(\frac{n}{2}) + n$

← روش ها:

۱- درخت بازگشت یا recursion tree

۲- روش قضیه‌ی اصلی master theorem

۳- حدس با جایگذاری و اثبات و استقراء

* روش حدس این گونه است که با جایگذاری رابطه در خودش و مشاهده‌ی روند حدس می‌زنیم و بررسی می‌کنیم.
* ما در اینجا برخلاف درس ریاضیات گسسته به دنبال تابع دقیق نیستیم بلکه فقط می‌خواهیم رشد آن را تعیین کنیم پس از آن ساده تر است. ☺

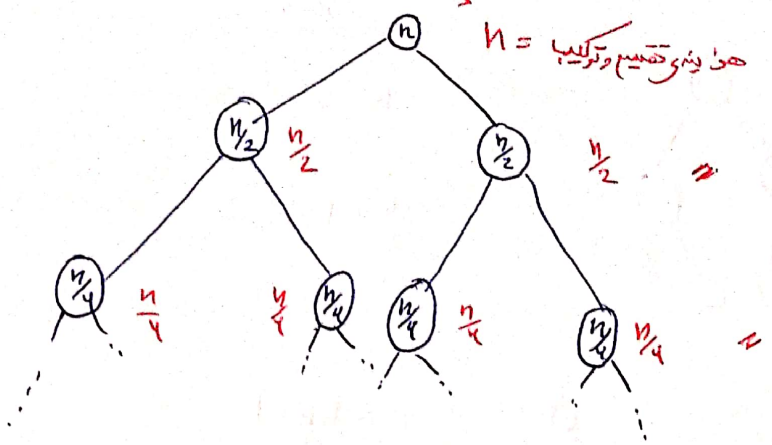
← روش درخت بازگشت:

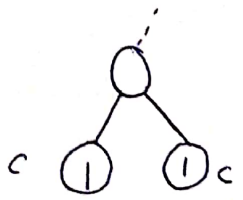
مثال:

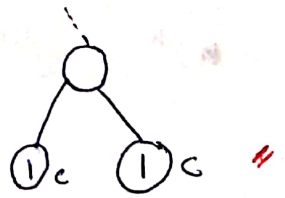
دوتا زیرمساله حل می‌کنیم.

$$T(n) = 2T(\tfrac{n}{2}) + n \qquad T(1) = c$$

n = هزینه‌ی تقسیم و ترکیب







هزینه‌ی کل می‌شود هزینه‌ی ترکیب و تقسیم به علاوه‌ی هزینه‌های c در برگ‌ها :

$T(n) =$ جمع هزینه‌ی گره‌های داخلی + جمع هزینه‌ی برگ‌ها

هزینه سطح $\sum$ سطح‌ها

تعداد برگ‌ها × هزینه برگ

$$\text{هزینه سطح‌ها} = n + \underbrace{\left(\frac{n}{2}+\frac{n}{2}\right)}_{n} + \underbrace{\left(\frac{n}{4}+\frac{n}{4}+\frac{n}{4}+\frac{n}{4}\right)}_{n} + \cdots$$

→ هزینه نصف و تعداد برابر می‌شود پس هزینه سطح‌ها ثابت می‌ماند

* هزینه‌ی هر سطح n می‌شود !!

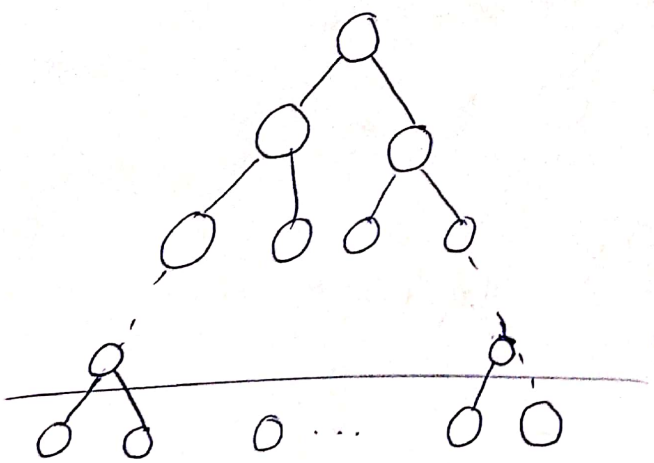
حال تعداد سطح‌ها را محاسبه می‌کنیم :

$$n \to \frac{n}{2} \to \frac{n}{4} \to \cdots \to 1$$

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

k → شماره سطح برگ‌ها

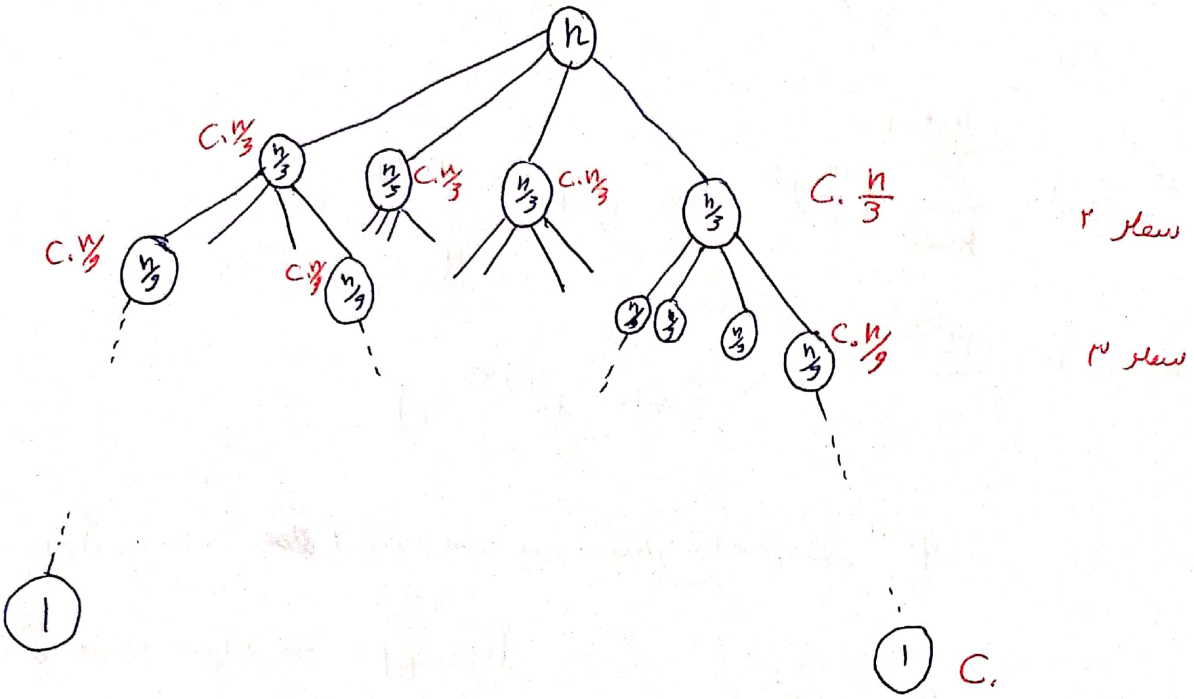
اندازه مساله هر گره	سطح
$\frac{n}{2^0}$	$k=0$
$\frac{n}{2}$	$k=1$
$\frac{n}{2^2}$	$k=3$
⋮	⋮
	$k=\lg n-1$
$\frac{n}{2^k}=1$	$k=\lg n$



$$\text{تعداد برگ‌ها} = 2^k = 2^{\lg n} = n^{\log_2 2} = n$$

$$\text{هزینه کل} = n.\log_2(n) + C.n \in O(n.\lg(n))$$

مثال: $T(n) = 4T(n/3) + O(n)$ و $T(1) = c$


n
C.n/3
n/3
C.n/3
C.n/3
C.

هزینه‌ی برگ‌ها = تعداد برگ × هزینه‌ی یک برگ

$$\text{هزینه‌ی غیر برگ‌ها} = \sum_{i=1}^{\log_3 n} c.\, n\left(\frac{4}{3}\right)^{i-1}$$

$$= c.n \sum_{k=0}^{\log_3 n - 1} \left(\frac{4}{3}\right)^{k} = c.n \frac{\left(\frac{4}{3}\right)^{\log_3 n} - 1}{\frac{4}{3} - 1}$$

$$= c.n \frac{n^{\log_3 4/3} - 1}{1/3} = 3cn\left(n^{\log_3 4/3} - 1\right)$$

تعداد گره ها در هر مرحله 4 برابر می‌شود پس در سطر i ام می‌شود 4^{i-1}

← من قبل تعداد سطر آخر = $\log_3 n + 1$

$$\Longrightarrow \text{تعداد برگ‌ها} = 4^{\log_3 n + 1 - 1} = 4^{\log_3 n} = n^{\log_3 4}$$

$$\Longrightarrow \text{هزینه برگ‌ها} = c.n^{\log_3 4}$$

$$\xRightarrow{\text{کل}} T(n) = 3cn\left(n^{\log_3 4/3} - 1\right) + cn^{\log_3 4}$$

$$\rightsquigarrow n.n^{\log_3 4/3}$$

$$\rightsquigarrow n.n^{\log_3 4 - \cancel{\log_3 3}^{\,1}}$$

$$\rightsquigarrow n.n^{\log_3 4 - 1}$$

$$\rightsquigarrow n^{\log_3 4}$$

$$\rightsquigarrow n^{\log_3 4}$$

$$\longrightarrow T(n) \in \Theta\left(n^{\log_3 4}\right)$$

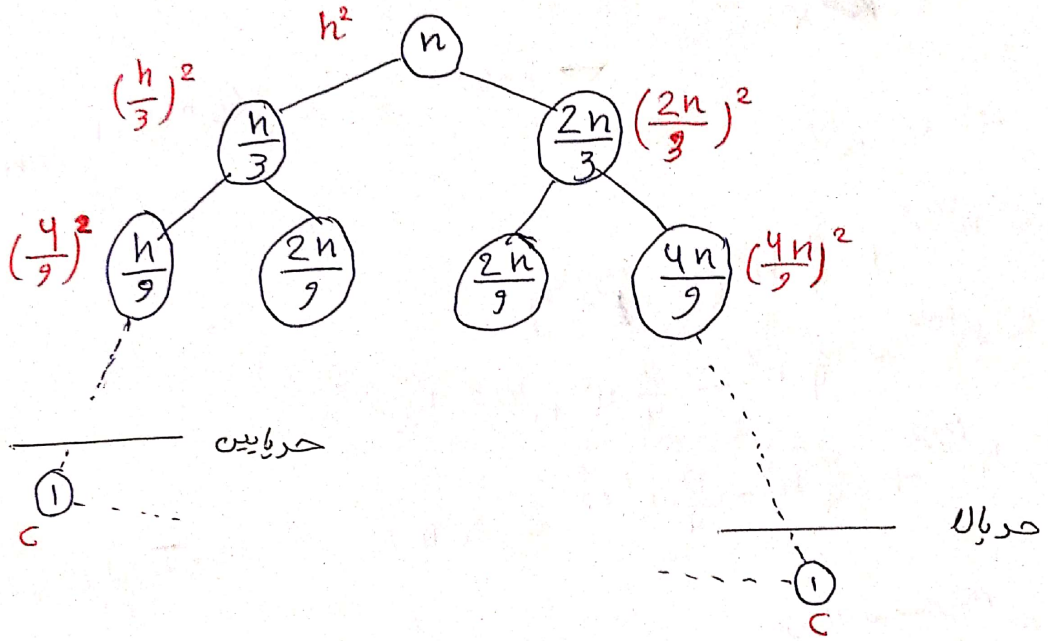
$$\in O\left(n^{\log_3 4}\right)$$

یادآوری: تصاعد هندسی

$a, aq, aq^2, \dots$

$$\text{جمع جملات} = \frac{a(q^n-1)}{q-1}$$

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n^2 \quad , \quad T(1) = c$$



* درخت دیگر بالانس نیست و سمت راست دیر به 1 می‌رسد.

* چون درخت بالانس نیست می‌توانیم تا لایه‌ی قبل برگ سمت چپ‌ترین را حساب کنیم و فرض کنیم در این سطح درخت تمام می‌شود ولی خب این مقدار کمتر از مقدار واقعی می‌شود که می‌گوییم حد پایین هزینه یا می‌توانیم فرض کنیم تا آخر سمت راست ترین می‌رود و بعد تمام می‌شود که این هم بیشتر از مقدار واقعی است چون درخت بالانس نیست.

ما در واقع حد بالا پایین دو حساب می‌کنیم و می‌گوییم رشد چیزی مابین این دو حد است.

$T(n$

⊗ حد پایین:

سایز مسأله در برگ $= \frac{n}{3^{i-1}} = 1 \leftarrow$ i برگ $= \log_3 n + 1$

هزینه‌ی سطح i $= \left(\frac{5}{9}\right)^{i-1} n^2$

هزینه گره‌های غیربرگ $= \sum_{i=1}^{\log_3 n} \left(\frac{5}{9}\right)^{i-1} \cdot n^2 = n^2 \sum_{k=0}^{\log_3 n - 1} \left(\frac{5}{9}\right)^k = n^2 \cdot \frac{1-(5/9)^{\log_3 n}}{1-5/9}$

$$= \frac{9}{4} n^2 \left(1 - n^{\log_3 5/9}\right)$$

هزینه‌ی برگ‌ها $= C \cdot$ تعداد برگ $\times$ $\leftarrow$ تعداد برگ $= 2^{i-1} = 2^{(\log_3 n + 1) - 1} = n^{\log_3 2}$

$$= C \cdot n^{\log_3 2}$$

حد پایین

$$T(n) = C \cdot n^{\log_3 2} + \frac{9}{4} n^2 \left(1 - n^{\log_3 5/9}\right)$$

$$= C n^{\log_3 2} + \frac{9}{4} n^2 - \frac{9}{4} n^2 n^{\log_3 5 - \cancel{\log_3 9}^{\,2}}$$

$$= C n^{\log_3 2} + \frac{9}{4} n^2 - \frac{9}{4} n^{\log_3 5} \in \Theta(n^2) \Rightarrow T(n) \in \Omega(n^2)$$

⊗ حد بالا:

$\left(\frac{2}{3}\right)^{i-1} n =$ سایز مسأله در سطح i ام سمت راست‌ترین شاخه

سایز مسأله در برگ $= 1 \rightarrow \left(\frac{2}{3}\right)^{i-1} n = 1$

$\rightarrow i = \log_{\frac{3}{2}} n + 1$

یک واحد کمتر $\Rightarrow i = \log_{\frac{3}{2}} n$

هزینه‌ی گره‌های غیر برگ $= \sum_{i=1}^{\log_{3/2} n} \left(\frac{5}{9}\right)^{i-1} n^2$

تصاعد هندسی

$$= n^2 \frac{1-(5/9)^{\log_{3/2} n}}{1-5/9} = \frac{9}{4} n^2 \left(1 - \left(\frac{5}{9}\right)^{\log_{3/2} n}\right)$$

$$\ldots = \frac{9}{4} n^2 - \frac{9}{4} n^{\log_{\frac{3}{2}} 5/4} \in \Theta(n^2)$$

هزینه‌ی برگ = تعداد برگ‌ها × هزینه‌ی یک برگ

تعداد گره‌ها در سطح i $= 2^{i-1}$

تعداد برگ‌ها $= 2^{(\log_{\frac{3}{2}} n + 1) - 1} = n^{\log_{\frac{3}{2}} 2}$

$\longrightarrow$ هزینه برگ‌ها $= c.n^{\log_{\frac{3}{2}} 2}$

حد بالا $T(n) = c.n^{\log_{\frac{3}{2}} 2} + \frac{9}{4}(n^2 - n^{\log_{\frac{3}{2}} 5/4}) \in \Theta(n^2) \Rightarrow T(n) \in O(n^2)$

$T(n) \in \Omega(n^2)$ ← حد پایین

$T(n) \in O(n^2)$ ← حد بالا

$T(n) \in \Theta(n^2)$

← روش قضیه‌ی اصلی:

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

$a \geq 1 \qquad b > 1$

* حالت اوّل: اگر $\varepsilon > 0$ وجود داشته باشد که $f(n) \in O(n^{\log_b a - \varepsilon})$ آن‌گاه:

$T(n) \in \Theta(n^{\log_b a})$

* حالت دوّم: اگر $f(n) \in \Theta(n^{\log_b a})$ آن‌گاه:

$T(n) \in \Theta(n^{\log_b a} . \lg(n))$

* حالت سوّم: اگر $\varepsilon > 0$ وجود داشته باشد که $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ و c وجود داشته باشد که $0 < c < 1$ و $af(\frac{n}{b}) \leq c.f(n)$ آن‌گاه:

$T(n) \in \Theta(f(n))$

مثال ۱: $a=2, b=2$ و $f(n)=n$

$$T(n)=2T(n/2)+n$$

$$n^{\log_b a} = n^{\log_2 2} = n \quad ? \quad f(n)=n \rightarrow n \in \Theta(f(n))$$

حالت دوم قضیه اصلی ←

$$T(n) \in \Theta(n.\lg(n))$$

مثال ۲: $T(n)=4T(\frac{n}{2})+n$

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

$$n^{\log_b a} = n^{\log_2 4} = n^2 \quad ? \quad f(n)=n$$

حالت یک را چک می‌کنیم:

$$f(n)=n \in O(n^{2-\varepsilon})$$

اگر $2-\varepsilon \geq 1$ ← $0<\varepsilon\leq 1$ ← پس وجود دارد و

حالت اول قضیه است ← $T(n)=O(n^2)$

مثال ۳:

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

$a=2$
$b=2$
$f(n)=n^2$

$$n^{\log_2 2} = n$$

بررسی حالت سوم:

$$f(n) \in \Omega(n^{\log_2 2+\varepsilon})$$

$$1+\varepsilon \leq 2 \rightarrow \exists \varepsilon: 0<\varepsilon\leq 1$$

چک کردن شرط دوم:

$$af(\frac{n}{b}) \leq c.f(n) \longrightarrow 2(\frac{n}{2})^2 \leq c.n^2 \rightarrow \frac{1}{2} \leq c < 2$$

برقراره پس حالت سوم است و $T(n) \in \Theta(n^2)$.

مثال ۴ :

$$T(n) = 2T(n/2) + n\log_2 n$$

* حالت

$a=2$
$b=2$
$f(n)=n.\lg n$

$$n^{\log_b a} = n \quad , \quad f(n) = n.\lg(n)$$

$$\exists\, \varepsilon>0 : f(n) = n.\log_2 n \in \Omega(n^{1+\varepsilon})$$

وجود ندارد $\varepsilon>0$ به طوری که $\log_2 n \in \Omega(n^{\varepsilon})$ ، رشد توابع چند جمله‌ای از توابع لگاریتمی

بیشتر است (مثلاً $n^{\frac{1}{2}}$ و ...) پس وجود ندارد کلاً.

متغیر

این مثال با قضیه‌ی اصلی حل نمی‌شود.

قضیه‌ی اصلی:

عدد صحیح

برای حالت دوم: اگر $f(n) \in \Theta(n^{\log_b a}.\lg^k(n))$ و $k \geq 0$ آن‌گاه:

$$T(n) \in \Theta(n^{\log_b a}.(\log_2 n)^{k+1})$$

* حال می‌توان مثال بالا را حل کرد:

$$T(n) \in \Theta(n.\log_2^2 k)$$

خلاصه شروط حالات

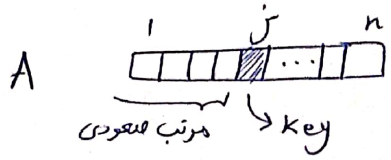
$f(n) \in O(n^{\log_b a - \varepsilon})$ ← اول

$f(n) \in \Omega(n^{\log_b a + \varepsilon})$ ← سوم

t_j

⊛ تحلیل الگوریتم مرتب‌سازی درجی :

روش افزایشی است.




```
Insertion_Sort(A, n)

c1 1. for (j  2 to n)
2. c2(n-1)   | i ← j-1
c3(n-1) 3.   | Key ← A[j]
4.           | while i>0 & key < A[i]
5.           |     | A[i+1] ← A[i]
6.           |     | i ← i-1
c7(n-1) 7.   | A[i+1] ← key
```

هزینه‌ی خط 4 : $c_4 \sum_{j=2}^{n} (t_j + 1)$

هزینه‌ی خط 5 : $c_5 \sum_{j=2}^{n} t_j$

هزینه‌ی خط 6 : $c_6 \sum_{j=2}^{n} t_j$ → تعداد تکرار بدنه‌ی حلقه‌ی while در گام j ام

$0 \leq t_j \leq j-1$

* بهترین حالت یعنی $t_j = 0$ وقتی رخ می‌دهد که آرایه مرتب صعودی باشد از قبل
* بدترین حالت وقتی رخ می‌دهد که آرایه نزولی مرتب باشد از قبل.

بهترین حالت $\Rightarrow t_j = 0 \rightarrow T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$

$$= an + b \in \Theta(n) \rightarrow T(n) \in \Omega(n)$$

بدترین حالت $\Rightarrow t_j = j-1 \rightarrow T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j-1)$

$$+ c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1)$$

$$= a'n^2 + b'n + c' \in \Theta(n^2) \rightarrow T(n) \in O(n^2)$$

$T(n) = \Theta(n^2)$ ⟵ $t_j = \frac{j-1}{2}$ * حالت متوسط یعنی

محاسبه‌ی حالت متوسط الگوریتم:

متغیر تصادفی

$X \in \{X_i\} \qquad 1 \leq i \leq n$

میانگین یا امید ریاضی

$$\bar{X} = E[X] = \sum_{i=1}^{n} P\{X = X_i\} . X_i$$

متغیر تصادفی شاخص Indicator random variable

A یک رخ داد

$$I\{A\} = \begin{cases} 0 & \text{if A does not occurs} \\ 1 & \text{if A occurs} \end{cases}$$

متغیر تصادفی شاخص

$X_H = I\{Y = H\}$

$Y \in \{H, T\}$

$$E[X_H] = 1 . Pr\{X_H = 1\} + 0 . Pr\{X_H = 0\}$$
$$= 1 . Pr\{Y = H\} + 0 . Pr\{Y = T\} = Pr\{H\}$$

امید ریاضی متغیر تصادفی شاخص

$$E[I\{A\}] = Pr\{A\}$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\sum_{j=2}^{n}(t_j+1)\right) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} t_j$$
$$+ c_7 (n-1)$$

$E[T(n)]$ هدف: میانگین زمان اجرا

متغیر تصادفی = t_j $\qquad 0 \leq t_j \leq j-1$

می‌دانیم که:

$$E[aX+b.Y] = a.E[X] + b.E[Y]$$

پس:

$$E[T(n)] = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} (E[t_j]+1) + c_5 \sum_{j=2}^{n} (E[t_j]) + c_6 \sum_{j=2}^{n} E[t_j] + c_7(n-1)$$

$$E[t_j] = ?$$

یک راه محاسبه:

$$E[t_j] = \sum_{k=0}^{j-1} k.Pr\{t_j = k\}$$

راه حل دوم:

$$X_j = \begin{cases} 0 & key < A[i] \\ 1 & o.w. \end{cases} \qquad 1 \le i \le j-1$$

$$t_j = \sum_{i=1}^{j-1} x_i$$ تکرار به خروجی while

$$E[t_j] = E\left[\sum_{i=1}^{j-1} x_i\right] = \sum_{i=1}^{j-1} E[x_i]$$

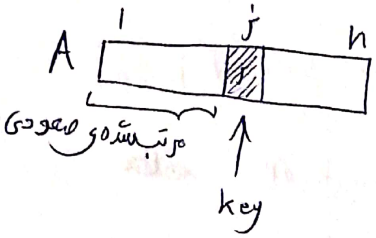
$$\underset{1 \le i}{E[X_i]} = Pr\{key < A[i]\} \qquad k = A[i] \quad 1 \le i \le j-1$$

* اگر فرض کنیم که آرایه‌ی ورودی به صورت تصادفی با توزیع یکنواخت به ما داده شده باشد آنگاه:

$$Pr\{key < A[i]\} = 1/2 \qquad E[t_j] = \sum_{j=1}^{j-1} \frac{1}{2} = \frac{j-1}{2}$$

$$E[T(n)] = a.n^2 + bn + c \in \Theta(n^2)$$

اثبات درستی الگوریتم مرتب سازی درجی (Correctness Proof)



ثابت حلقه Loop Invariant

* گزاره‌ای که در طول اجرای حلقه درست است.

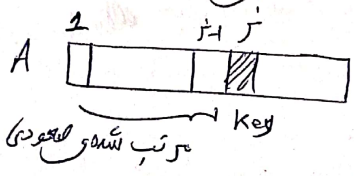
در ابتدای گام j ام حلقه‌ی الگوریتم آرایه‌ی ۱ تا j-1 شامل همان عناصر اولیه ۱ تا j-1 است اما به صورت مرتب شده‌ی صعودی.

اثبات ثابت حلقه با استقراء:

* حالت پایه: ابتدای گام حلقه j=2:

در ابتدای گام j=2 آرایه‌ی ۱ تا ۱ شامل همان عناصر اولیه‌ی آرایه‌ی ۱ تا ۱ است اما به صورت مرتب شده که به طور واضح درست است.

* حالت گذاره: فرض کنید که ثابت حلقه برای گام j ام درست است و برای گام j+1 اثبات کنیم.

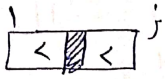


فرض استقراء

در یک گام از حلقه‌ی for الگوریتم:

① ترتیب عناصر خانه‌ی 1 تا j-1 عوض نمی‌شود بلکه فقط شیفت به راست پیدا می‌کند.

② کلید در جایی درج می‌شود که از همه‌ی عناصر راست خود کوچکتر است و از همه‌ی عناصر آرایه‌ی قبل خود بزرگتر یا مساوی است.



① و ② ← در انتهای گام j (یا در ابتدای گام j+1 ام) عناصر آرایه‌ی ۱ تا j شامل همان عناصر اولیه‌ی ۱ تا j آرایه است اما به صورت صعودی مرتب شده‌اند.

← چون مداکثر یک شیفت به راست داریم!

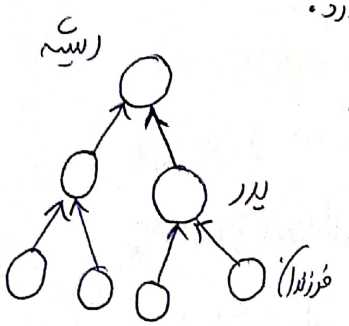
بنابراین گزاره‌ی ثابت حلقه درست است (با استقراء)

* در حالت پایانی الگوریتم که حلقه‌ی for تمام می‌شود j=n+1 است.

در ابتدای گام j=n+1 ام حلقه‌ی for الگوریتم، آرایه‌ی ۱ تا n=(j-1) شامل همان عناصر ۱ تا n است اما به صورت صعودی مرتب شده که این گزاره درستی الگوریتم را نتیجه می‌دهد.

⊛ تحلیل مرتب سازی پشته‌ای (Heap-Sort):

heap یک ساختمان داده‌ی درختی است و دو نوع min و max دارد.



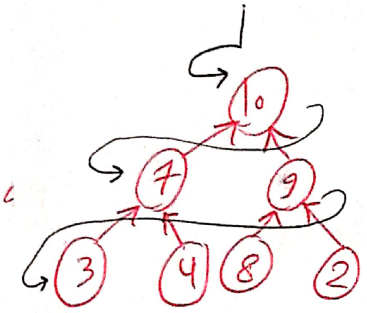
min-heap ، مقدار گره فرزندان بیشتر مساوی گره پدر است.

max-heap ، مقدار گره فرزندان کمتر مساوی گره پدر است.

← تمام سطوح درخت پر باشد ← تمام فرزندان وجود داشته باشند

← درخت perfect یا کاملاً پر می‌گوییم

✳ هیپ یک ساختمان داده درختی دودویی است که درخت کامل است.



Max-Heap

پیاده سازی heap با آرایه:

عناصر آن را از بالا به پایین و از چپ به راست در داخل آرایه به ترتیب می‌چینیم:

	۱	۲	۳	۴	۵	۶	۷
A	10	7	9	3	4	8	2

$$\begin{cases} \text{left_Child}(i) = 2i \\ \text{right_child}(i) = 2i+1 \\ \text{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor \end{cases}$$

فرزند سمت چپ i

فرزند سمت راست i

والد i

اگر اندیس خروجی بزرگتر از heap-size شود یعنی فرزند وجود ندارد.

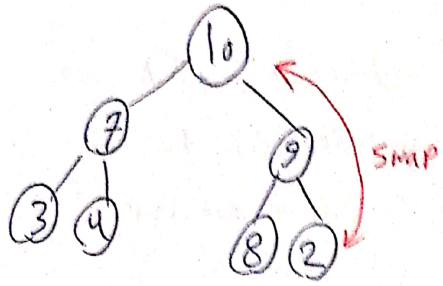
heap-size = 7.

اندیس گره برگ = i
گره برگ فرزند ندارد

$\rightarrow 2i > \text{heap size} \rightarrow i > \frac{\text{heap-size}}{2}$

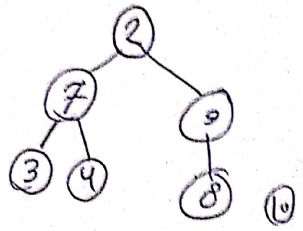
← از نصف به بعد آرایه همگی برگ هستند.

مرتب سازی ریشه ای:



heap-size --

⇒



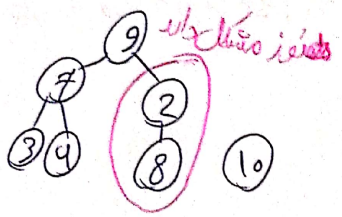
⇒ ...

به یک الگوریتم کلی نیاز داریم که یک درخت را به عنوان ورودی بگیرد که زیر درخت سمت راست و چپ ریشه‌ی آن max_heap باشد و مقدار گره ریشه لزوماً از دو فرزند خود بزرگتر یا مساوی نباشد

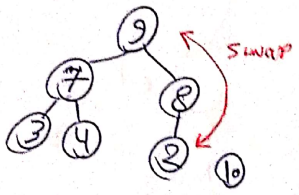
⇐ در این صورت بین ریشه و فرزندان آن max می‌گیریم و هرکدام بزرگتر بود جای ریشه قرار می‌دهیم

و همین کار را برای زیردرختی که ریشه آن عوض شده است تکرار می‌کنیم:

... ⇒

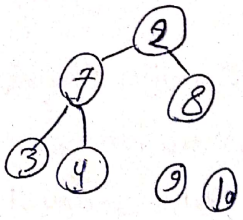


تکرار روند بالا برای این درخت ⇒



heapsize --

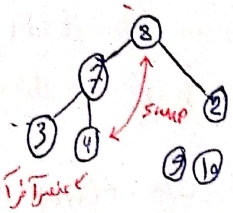
⇒



A [2|7|8|3|4|9|10]

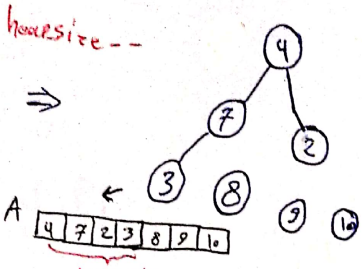
heapsize

heapify ⇒



عنصر آخر آرایه‌ی heap

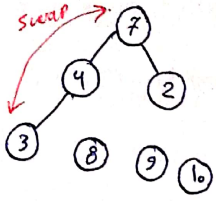
heapsize -- ⇒



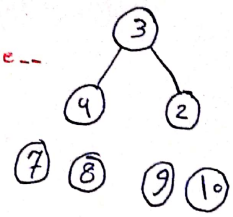
A [4|7|2|3|8|9|10]

heap-size

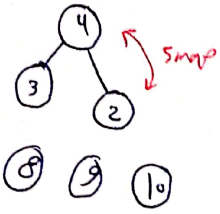
⇒



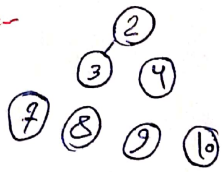
heap size -- ⇒



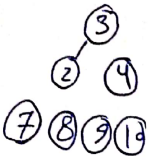
heapify ⇒



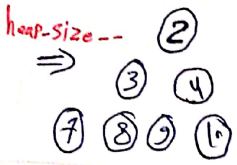
heapsize-- ⇒



heapify ⇒



heap-size -- ⇒



heapify ⇒

الگوریتم تمام می‌شود.

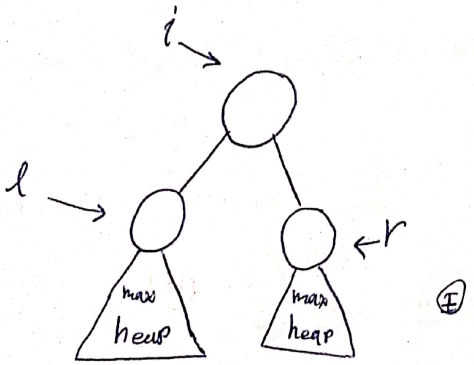
چون فرزندی ندارد

Max-Heapify (A, i)

از کدام گره می‌خواهی هیپیفای کنی که ابتدا 1 است.

ورودی A آرایه‌ای است که شامل درخت دودویی کامل که زیردرخت چپ و راست آن max-heap است.

```
l ← 2i , r ← 2i+1
if l ≤ heapsize and A[l] > A[i] then
    largest ← l
else largest ← i
if r ≤ heapsize and A[r] > A[largest] then
    largest ← r
if i ≠ largest then
    swap(A[i], A[largest])
    Max-Heapify (A, largest)
```



شرط بازگشت در جایی است که چپ و راست یا وجود نداشته باشند یا درخت خودش چپ باشد.

* تحلیل زمانی:

در بدترین حالت تا ارتفاع درخت طی می‌شود و در هر سطح از ارتفاع هزینه‌ی پیدا کردن عنصر max بین ریشه و زیردرخت چپ و راست و عوض کردن جای ریشه با عنصر max مقدار ثابت $O(1)$ است.

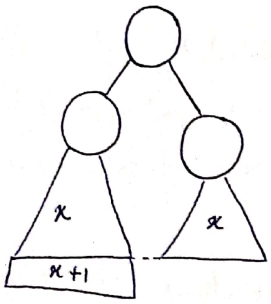
ارتفاع درخت حداقل $\log_2 n$ است.

* درخت heap یک درخت کامل است که ارتفاع آن $\Theta(\lg n)$ است حتماً.

پس هزینه‌ی max-heapify می‌شود $O(1) \times O(\log_2 n) \Leftarrow O(\log_2 n)$

* تحلیل به روشی دیگر:

نوشتن زمان به صورت بازگشتی (با توجه به شکل که کشیدیم)



می‌تواند درخت ما به صورت مقابل باشد:

x = تعداد عناصر زیردرخت راست.

$$3x+2=n \rightarrow x=\frac{n-2}{3}$$

$$x \simeq \frac{n}{3}$$

بنابراین زیردرخت چپ $\frac{2}{3}n$ عنصر دارد.

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

محاسبه‌ی مرتبه‌ی زمان الگوریتم:

$a=1 \qquad b=\frac{3}{2} \qquad f(n)\in O(1)$

$$n^{\log_b a}=n^{\log_{\frac{3}{2}} 1}=n^0 \quad ? \quad O(1)=f(n) \longrightarrow n^0\in\Theta(f(n))$$

پس حالت دوم قضیه اصلی است:

مرتبه $T(n)\in\Theta(n^{\log_b a}\cdot\log_2 n)=\Theta(\log_2 n)$

مرتبه عضو $\Theta(\lg n)$ است پس خود الگوریتم: $T(n)\in O(\lg(n))$

زمان Max_heapify $\in O(\lg(n))$

نکته‌ی مهم: بدترین حالتی که ما فرض کرده‌ایم امکان‌پذیر نیست چون اگر در مرحله‌ی اول زیردرخت چپ مانند شکل بالا باشد پس درخت اصلاً کامل نمی‌شود که ☺ ما واقعاً حالت بدی را در نظر گرفتیم که اکثراً پیش نمی‌آید.

بهترین حالت هم این است که خود heap باشد و در $\Theta(1)$ اجرا شود الگوریتم پس:

زمان میانگین $\in O(\lg(n))$

$\in \Omega(1)$

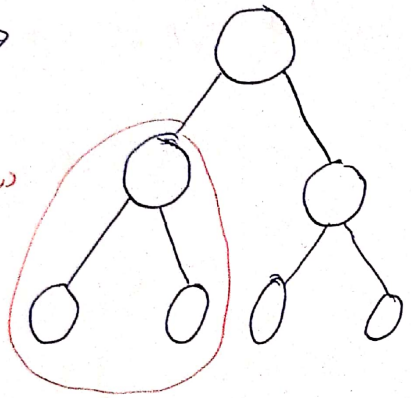
توجه: حالت $n/2$ را در نظر نمی‌گیریم چون اینکه درخت ما Perfect است یک حالت غیردقیق است، نه بدترین حالت است و نه بهترین حالت است.

← ساختن Max-heap از روی یک آرایه دلخواه:

هر آرایه‌ای یک پارامایش از یک هیپ است.

هر برگ به تنهایی یک max heap است (چون فرزند ندارد و نمی‌تواند شرط هیپ بودن را نقض کند)

آرایه ⇒



در این زیردرخت، فرزندان یک max heap است.

* پس با Max-Heapify می‌توان آن را به یک max heap تبدیل کرد.

⇒ بنابراین با شروع از یک سطح بالاتر از برگ‌ها اگر تابع max-heapify را برای هر سطح صدا بزنیم و سطح به سطح بالا برویم تا ریشه وقتی برای ریشه max-heapify صدا زده می‌شود کل درخت یک max-heap خواهد بود.

```
Build_Max_Heap (A)
1.    heap_size = size (A)
2.    for i ← ⌊heap_size/2⌋ down to 1 do
3.        Max-Heapify (A, i)
```

برگ فرزند ندارد $\Rightarrow 2i > \text{heap size}$

$$i > \frac{\text{heapsize}}{2}$$

← تحلیل زمان:

Max_Heapify متعلق است به $O(\log_2 n)$ و $\frac{n}{2}$ با صدا زده می‌شود پس:

زمان ساخت Heap $\in O(\frac{n}{2} \cdot \lg n) = O(n \cdot \lg n)$

سوال: آیا ساخت هیپ واقعاً زمانش در بدترین حالت این است؟ یعنی این را touch می‌کند اصلاً؟
یا اینکه یک حد بالای کوچکتری هم وجود دارد برایش؟

جواب: می‌توان نشان داد که زمان ساخت Max_Heap متعلق به $O(n)$ است.

* اگر دقیقتر بررسی کنیم:

← تعداد خیلی زیادی گره داریم که ارتفاع خیلی کمتر از $\lg n$ دارند و تعداد کمی گره داریم که ارتفاع نزدیک به $\lg n$ دارند ⇐ در نتیجه نشاید یک حد بالای کمتر از $n \log_2 n$ پیدا کنیم.

heapify در

← یعنی اینکه وقتی در پایین هستیم و داریم هیپیفای می‌کنیم همشون در درجه نیستند ولی ما در نظر گرفتیم که هستند.

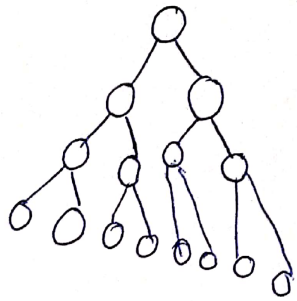
← در پایین تعداد زیادی گره داریم که هزینه heapify کمی دارند.
← در بالا تعداد کمی گره داریم که هزینه‌ی heapify زیادی دارند.

$h = \lg n$

$\vdots$

$h = 1$

$h = 0$



هزینه‌ی هیپیفای در ارتفاع h: $O(h)$

تعداد گره‌ها در ارتفاع h: $\frac{n}{2^{h+1}}$

$0 \leq h \leq \log_2 n$

$$\text{هزینه} = \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} \times O(h) = O\left(n \cdot \sum_{h=0}^{\lg n} \frac{h}{2^h}\right) =$$

یادآوری:

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2} \qquad |x| < 1$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 \qquad k = h, \quad x = 1/2$$

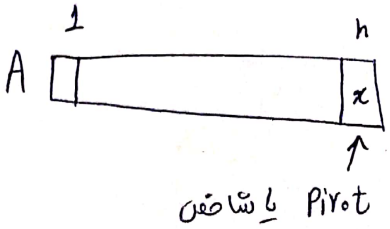
$$\Rightarrow \text{هزینه} = O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(2n)$$

$$= O(n)$$

```
Heap_Sort(A)
c1  1.  h ← length[A]
O(n)  2.  build_Max_Heap(A)
c3n  3.  for i ← n down to 2 do
c4(n-1)  4.      swap(A[1], A[i])
c5(n-1)  5.      heap_size --
O(n.logn)  6.    max_heapify(A, 1)   → روی کل درخت
```

$$\text{تحلیل زمانی} = c_1 + O(n) + c_3(n) + c_4(n-1) + c_5(n-1) + O(n.\lg n) = O(n.\lg n)$$

مرتب سازی سریع (Quick sort) :



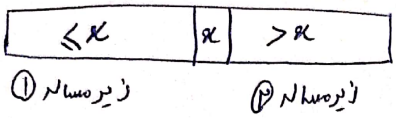
مبتنی بر روش Divide & Conquer .

روش تقسیم و غلبه:

① تقسیم مساله به مسایل کوچکتر (زیرمسایل)

② حل بازگشتی مسایل کوچکتر

③ ترکیب جواب زیرمسایل و بدست آوردن جواب مساله اصلی

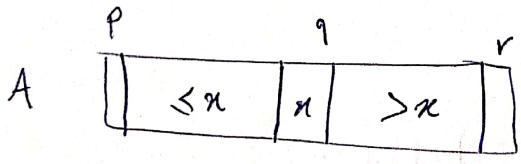


گام ① :

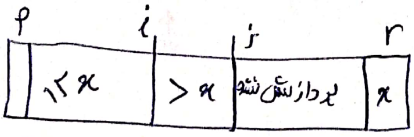
* Partition کردن : اونهایی که کوچکتر از شاخص هستند بیار سمت چپ و بقیه را ببر سمت راست و این x بین این دو باشد.

گام ② : حل بازگشتی زیرمسالهی ① و ②

گام ③ : برای ترکیب جواب زیر مساله ① و ② نیازی به کار اضافی نیست.



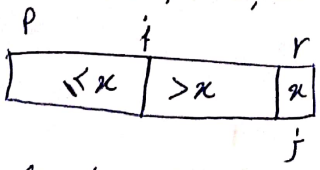
پس از اتمام Partition :



در حین Partition کردن :

← تکلیف: باید مشخص بشه و ...

پس از اجرای حلقه : (صفحه‌ی بعد در ابتدای خط 7)

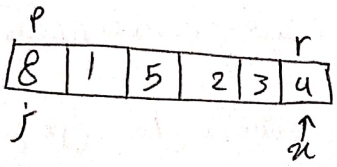


حالا باید x رو با یه جوری i+1 جابجا کنیم که x بیاد وسط.

← (خط 7 .)

```
Partition(A, p, r)
    x ← A[r]                          c1
    i ← p-1                           c2
    for j ← p to r-1 do               c3 n
        if A[j] ≤ x then              c4 (n-1)
            i ← i+1                   c5 (n-1)
            swap(A[i], A[j])          c6 (n-1)
    swap(A[i+1], A[r])                c7
    return i+1                        c8
```

مثلاً آرایه:

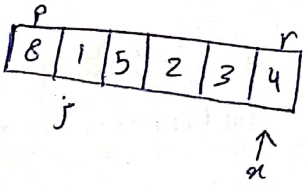


$x = 4$

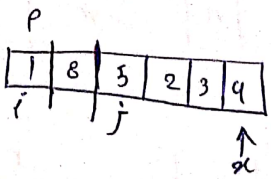
$i = p-1$

$j = p$

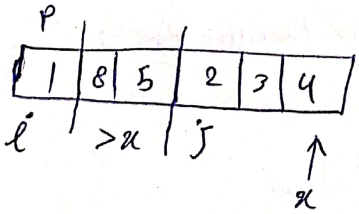
$j = p+1$



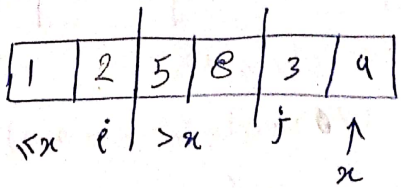
$\Rightarrow$



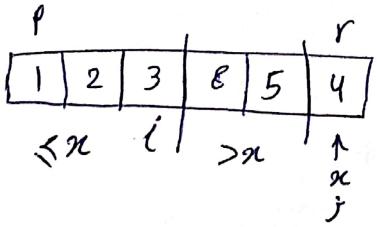
$\Rightarrow$



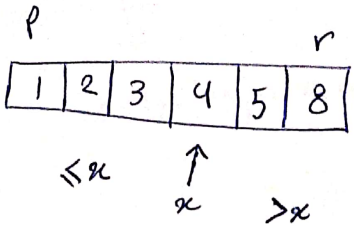
$\Rightarrow$



$\Rightarrow$



دستور آخر $\Rightarrow$



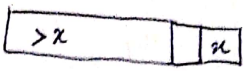
مرتب است و همواره

این جواب پارتیشن است ولی نکته اینجاست که اتفاقاً این جا این اتفاق نمی افتد.

تحلیل زمانی پارتیشن:

بهترین حالت: شرط if برقرار نباشد، کی رخ می‌دهد؟ همان اول کار همه از شاخص بزرگتر باشند

وقتی که آرایه مرتب شده‌ی نزولی باشد.



در همین حالت بهترین:

فقط هزینه‌ی دستور 5 و 6 حذف می‌شود و هزینه دستور 4 به جای خود باقی است.

بهترین $T(n) = c_1 + c_2 + c_3 n + c_4 (n-1) + c_7 + c_8 = \Theta(n)$

$$\Rightarrow T(n) \in \Omega(n)$$

در بدترین حالت:

همه عناصر کوچکتر از شاخص (مرتب شده‌ی صعودی)

بدترین $T(n) = c_1 + c_2 + c_3 n + c_4 (n-1) + c_6 (n-1) + c_5 (n-1) + c_7 + c_8 = \Theta(n)$

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

$$\left.\begin{array}{l} T(n) \in \Omega(n) \\ T(n) \in O(n) \end{array}\right\} \longrightarrow T(n) \in \Theta(n)$$

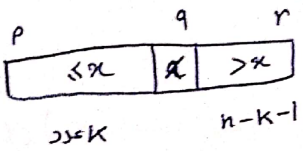
شرط ادامه

```
Quick-Sort(A, p, r)
    if p<r then                          c1
    q ← Partition(A, p, r)               Θ(n)   گام تقسیم
    Quick-Sort(A, p, q-1)
    Quick-Sort(A, q+1, r)                        گام حل بازگشتی
```

گام ۱ 2.
گام ۲ 3.
4.
گام ۳ 5.

پوچ

گام ترکیب



$T(k)$ خط 3

$T(n-k-1)$ خط 4

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

زمان اجرای Quick Sort

← باید روی k بحث کرد!

تحلیل زمان الگوریتم Quick Sort :

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

بحث روی مقادیر k : $0 \leq k \leq n-1$

* اگر $k=0$ آن‌گاه :

$$T(n) = T(0) + T(n-1) + \Theta(n)$$
$$= T(n-1) + \Theta(n) = \Theta(n^2)$$

$k=0$ چه موقع رخ می‌دهد؟ آرایه‌ی ورودی ترتیب نزولی داشته باشد.

* اگر $k=n-1$ آن‌گاه :

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n) = \Theta(n^2)$$

$k=n-1$ چه موقع رخ می‌دهد؟ آرایه‌ی ورودی ترتیب صعودی داشته باشد.

$0 \leq k \leq n-1$

$\downarrow$ $T(n) = \Theta(n^2)$

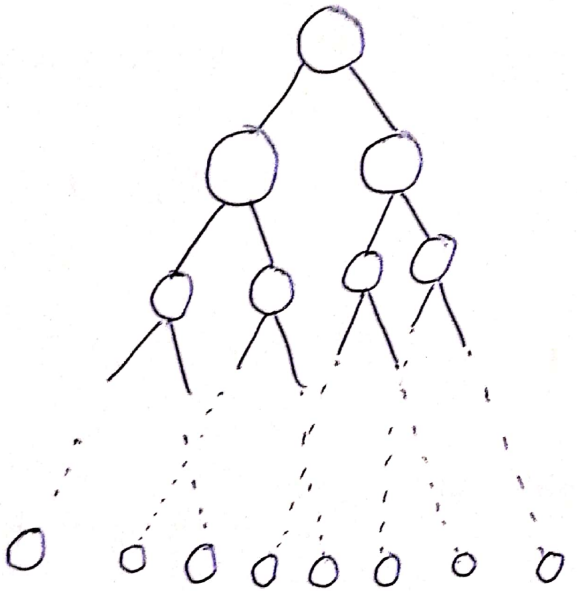
$\searrow$ $T(n) \in \Theta(n^2)$

؟

* اگر $k = n/2$

$$T(n) = T(\tfrac{n}{2}) + T(\tfrac{n}{2} - 1) + \Theta(n)$$
$$\approx 2T(\tfrac{n}{2}) + \Theta(n)$$
$$= \Theta(n \lg n)$$

$k = n/2$ وقتی که تقسیم ما بالا شده و کمترین مقدار رو داره تابع :



نکته‌ای ما بحث نمی‌پردازیم ولی مهمه :

از کجا بدونیم مثلا مینیمم داره وسط و ... اینطوریه که از رابطه‌ی اصلی بر حسب K مشتق می‌گیریم و مشتق دوّم و با تعیین علامت میشه max , min و رشد تابع رو بررسی کرد برای مقادیر K.

بهترین حالت $k = n/2$ ← $T(n) \in \Theta(n \lg n)$

بدترین حالت $k = 0 = n-1$ ← $T(n) \in \Theta(n^2)$

} Quick Sort

حالت متوسط با محاسبه‌ی امید ریاضی: $T(n) \in \Theta(n \log n)$

اثبات درستی الگوریتم Quick sort:

← حالت پایه $p = r$ که خب برای یک آرایه درست و صعودی مرتب شده است.

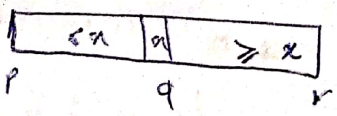
← فرض می‌کنیم برای کمتر از n عنصر این الگوریتم درست کار می‌کند. (اگر پارتیشن درست کار کند)

~~← برای n+1 عنصر هم حتما درست است چون به صورت بازگشتی دوبار صدا زده می‌شود برای هر بخش که هر کدوم کمتر از n عنصر دارند.~~

~~* می‌ماند بخش Partition که باید درستی آن را اثبات کنیم.~~

← برای $n > 1$:

فرض استقرا: برای هر آرایه با کمتر از n عنصر، Quick-sort آن را به صورت صحیح مرتب صعودی می‌کند.



بنا به فرض استقرا، نیمه‌ی اول و دوم آرایه صعودی مرتب شده است (کمتر از n است هر کدام)

برای هر $p \leq i, j \leq r$

نشان می‌دهیم اگر $i < j \Rightarrow A[i] \leq A[j]$

اگر $i < q$ و $p \leq i \Rightarrow A[i] \leq A[j]$ بنا به فرض استقرا

(هر دو نیمه‌ی اول)

اگر $r \geq j$ و $q < i$ ⇐ $A[i] \leq A[j]$ بنا به فرض استقراء

(هر دو در نیمه دوم)

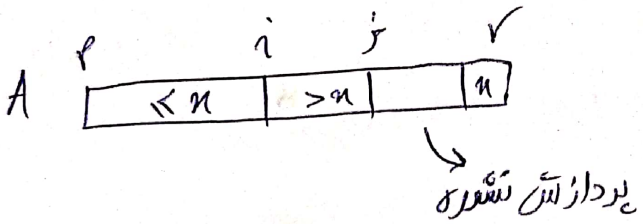
اگر $p \leq i < q$ و $q \leq j \leq r$ آنگاه $A[i] \leq x < A[j]$ (چون پارتیشن درست کار می‌کند)

بنابراین ثابت می‌شود که الگوریتم درست کار می‌کند.

* ما همه جا فرض کردیم که Partition درست کار می‌کند پس باید درستی این بخش را ثابت کنیم:

← اثبات صحت Partition :

نکته: در الگوریتم‌های بازگشتی استقرا خیلی آسان است ولی برای غیر بازگشتی‌ها می‌توانیم روی متغیر loop استقراء بزنیم.



loop invariant

ثابت حلقه: در ابتدای گام j حلقه for الگوریتم partition خانه $A[p..i]$ شامل عناصر $\leq x$ و خانه $A[i+1,\dots,j-1]$ شامل عناصر $> x$ و خانه‌ی $A[j..r-1]$ شامل عناصر پردازش نشده است و $A[r]=x$

← پایه استقراء: ابتدای حلقه $j = p$ $\quad i = p-1$ $\quad x = A[r]$

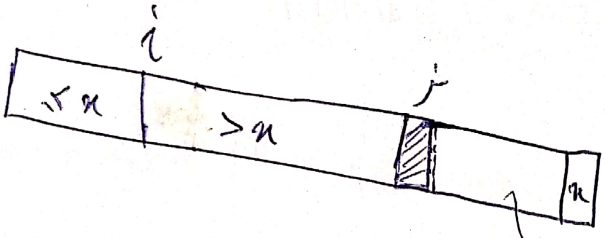
$A[p \dots i] = A[p \dots p-1] =$ تهی $\quad$ عناصر $\leq x$

$A[i+1, \dots, j-1] = A[p \dots p-1] =$ تهی $\quad$ عناصر $> x$

$A[j \dots r-1] = A[p \dots r-1]$ $\quad$ عناصر پردازش نشده

پس پایه‌ی استقراء برقرار است.

← حالت گذار: اگر ثابت حلقه برای j درست است باید ثابت کنیم برای $j+1$ هم درست است



پردازش نشده

فرض می‌کنیم در ابتدای گام jام حلقه‌ی for ثابت حلقه برقرار باشد. در گام jام

در حالت رخ می‌دهد: $A[j] \leq x$ یا $A[j] > x$

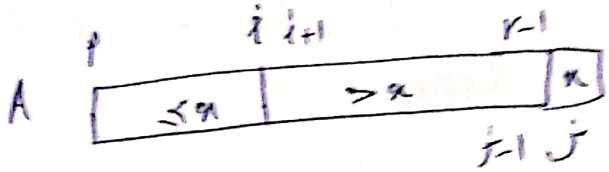
وقتی هر دو حالت را چک می‌کنیم می‌بینیم که در هر دو حالت باز ثابت حلقه برقرار است.

بنابراین ثابت حلقه با استقراء ثابت می‌شود.

نتیجه: ثابت حلقه برای $j = r$ هم برقرار است (در انتهای حلقه‌ی for)

نتیجه: در ابتدای گام $j = r$ آرایه $A[p \dots i]$ شامل عناصر $\leq x$ و آرایه‌ی $A[i+1, \dots, r-1]$ شامل عناصر $> x$ و $A[r] = x$ است.

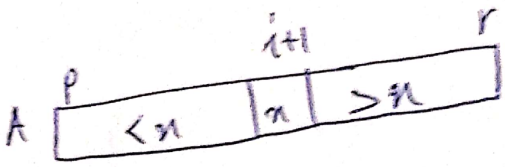
در $r=r$ داریم :



۲ دستور آخر الگوریتم :

```
swap (A[i+1], A[r])
return i+1
```

i+1 برگردانده می‌شود که اندیس شاخص x است.



بنابراین الگوریتم Partition درست کار می‌کند.

تحلیل حالت متوسط الگوریتم QuickSort

```
Randomized_partition (A, p, r)
    i ← RANDOM (p, r)
    swap (A[i], A[r])
    return partition (A, p, r)

Randomized_quicksort (A, p, r)
    if p<r then
        q ← Randomized_partition (A, p, r)
        Randomized_quicksort (A, p, q-1)
        Randomized_quicksort (A, q+1, r)
```

کوچه این همون Quick Sort است فقط اومدیم آمار و احتمال رو با Random کردن پارتیشن دخیل کردیم.

می‌تونستیم این رو فرض کنیم روی کد اصلی ولی نوشتیم که قشنگ واضح باشه.

* بنابراین شاخص می‌تواند هر یک از اعداد ورودی آرایه با احتمال مساوی باشد.

* هزینه‌ی quicksort متناسب با هزینه‌ی انجام شده در Partition.

* هزینه‌ی انجام شده در Partition برابر است با تعداد مقایسه های انجام شده بین اعداد با شاخص

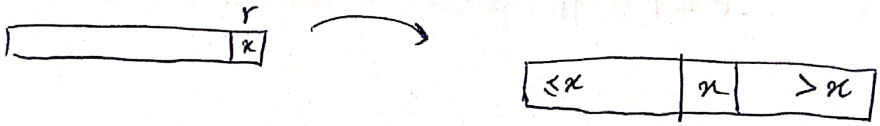
⊗ نام گذاری اعداد ورودی آرایه:

$$z_1, z_2, \dots, z_n$$

z_i : i امین کوچکترین عدد در آرایه

$$Z_{ij} = \{ z_i, z_{i+1}, \dots, z_j \}$$
$(i \le j)$

نکته: هر زوج عدد حداکثر یکبار مقایسه می‌شوند، یکبار با شاخص مقایسه می‌شود و دیگر با هم مقایسه نمی‌شود.
در هر بار هم هر عددی با شاخص اون بخش مقایسه می‌شود.



متغیر تصادفی شاخص

$$X_{ij} = I\{ z_i \text{ is Compared to } z_j \}$$

تعداد کل مقایسه ها (هزینه کل) $= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} = X$

میانگین هزینه کل $= E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right]$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \text{Probab}\{ z_j \text{ با } z_i \text{ مقایسه شود} \}$$

در $j=2$ داریم:

(۳۸)

اگر شاخص k که برای مجموعه $z_{i,j}$ انتخاب می‌شود z_i یا z_j باشد آن‌گاه z_i با z_j مقایسه می‌شود و گرنه به دو یا تیکشن متفاوت می‌روند و مقایسه نمی‌شوند.

$$Prob\{z_i \text{ با } z_j \text{ مقایسه شود}\} = \frac{2}{j-i+1}$$

انتخاب ۲ عنصر مطلوب

تعداد کل عناصر بین i و j

$$\rightarrow E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \qquad k = j-i$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \underbrace{\sum_{k=1}^{n} \frac{2}{k}}_{O(\lg n)} = O(n.\lg n)$$

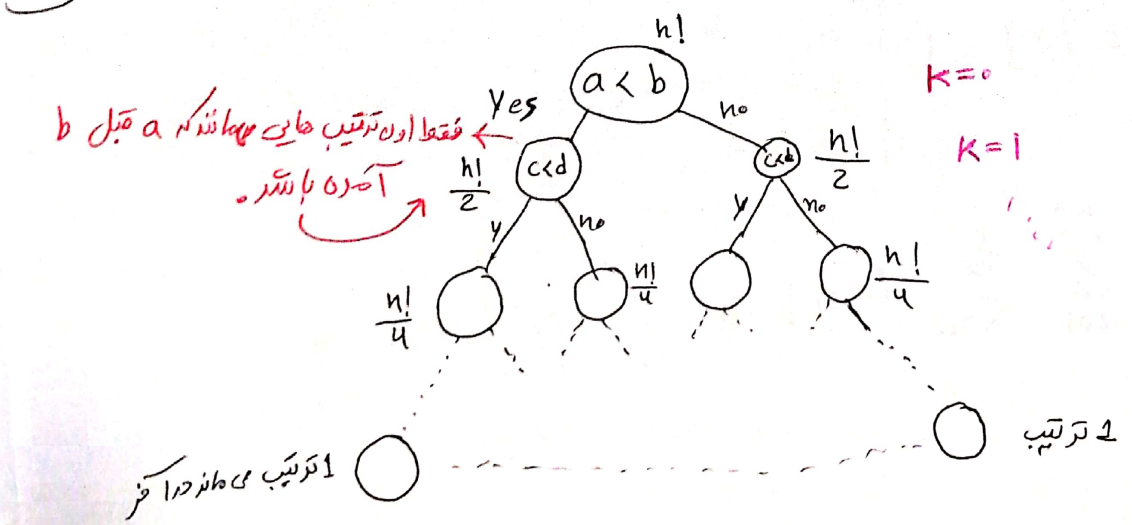
بنابراین حد بالای هزینه‌ی quick sort در حالت متوسط $n.\lg n$ است.

حالت متوسط $T(n) \in O(n.\lg n)$

مرتب‌سازی زمان خطی (linear time sorting)

می‌شود ثابت کرد که زمان اجرای الگوریتم‌های مرتب‌سازی که براساس مقایسه کار می‌کنند در بدترین حالت $\Omega(n.\lg n)$ است.

کلیات اثبات: n عدد $n!$ ترتیب مختلف دارد که فقط یک ترتیب از $n!$ ترتیب مرتب شده‌ی مطلوب است.



پس عمق درخت تا $\lg(n!)$ می‌رود.

$$\frac{n!}{2^k} = 1 \rightarrow k = \log_2(n!) \in \Theta(n \lg n)$$

$$k = 0 \cdots \log_2 n! \longrightarrow \log_2 n! + 1 = \text{تعداد سطوح}$$

* ۱ یک درخت باینری که $n!$ برگ دارد حداقل ارتفاع درخت $\lg n!$ $(\Theta(n \lg n))$ است.

* ۲ تعداد مقایسه‌ها در بدترین حالت مساوی ارتفاع درخت است.

از ① و ② ← حداقل تعداد مقایسه‌ها در بدترین حالت $\Theta(n \log n)$ است.

$$T(n) \in \Omega(n \lg n)$$

زمان الگوریتم مرتب‌سازی مبتنی بر مقایسه در بدترین حالت

پس بنابراین هیچ موقع الگوریتم مقایسه‌ای نمی‌تواند پیدا کرد که رشد زمان در بدترین حالت از $n \lg n$ کمتر شود.

* الگوریتم‌هایی وجود دارد که بر مبنای مقایسه کار نمی‌کنند:

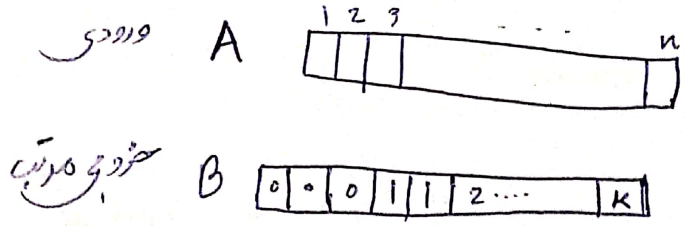
① Counting sort

② Radix sort

③ Bucket sort

الگوریتم Counting Sort :

فرض: n عدد صحیح در بازه 0 تا K داریم:



شمارنده‌ی C

	0	1	2 ...	K
C	0	0	...	0

C[i] قرار است تعداد iهای آرایه‌ی A را بشمارد.

مثال:

A

0	1	3	1	2	3

0 ≤ اعداد ≤ 3

می‌آییم جمع جمعی می‌زنیم بعد از ساخت C

	0	1	2	3
C	0	0	0	0
C	1	1	0	1
C	1	1	0	1
C	1	2	1	2

جمع تجمعی ⟹

C	1	3	4	6

محل آخرین ۲ است

محل آخرین خانه‌ی ۳ است

هر کدوم رو در خانه متناظرش می‌گذاره و کم می‌کنه از شمارنده تا به خونه‌ی قبلی رو مشخص کنه...

```
Counting_Sort (A, n, k)
for i ← 0 to k do                    (k+2) c1
    C[i] = 0                         (k+1) c2
for i ← 1 to n do                    (n+1) c3
    C[A[i]] ++                       n c4
for i ← 1 to k do                    (k+1) c5
    C[i] = C[i] + C[i-1]             k c6
for i ← n downto 1 do                (n+1) c7
    B[C[A[i]]] ← A[i]                n c8
    C[A[i]] --                       n c9
```

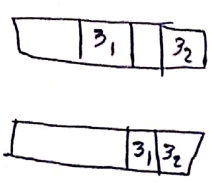
زمان اجرای الگوریتم Counting Sort

$$T(n) = a.n + b.k + c$$
$$= \Theta(n+k)$$

خیلی بستگی به k داده مثلاً برای نمره دانشجوها این همیشه ثابته ولی ممکنه به خود n هم وابسته باشد.

$$\text{if } k \in O(n) \Longrightarrow T(n) = \Theta(n)$$

نکته: در الگوریتم مرتب سازی stable ترتیب اعداد مساوی در ورودی با ترتیب آنها در خروجی یکی است:



* چون ممکن است این ها کلید رکورد های اطلاعاتی باشد و مهم باشد ترتیب. این کار با همان آرایه ی تجمعی انجام می شود وگرنه می توانستیم بعد از شمردن شروع به پر کردن آرایه خروجی بکنیم.

پس: پس Counting sort یک الگوریتم stable است.

الگوریتم Raddix Sort :

n عدد صحیح d رقمی در مبنای K داریم!

در حالت متداول از رقم‌های با ارزش شروع به مقایسه و مرتب‌سازی می‌کنیم ولی در Raddix sort از سمت راست یا رقم کم ارزش شروع می‌کنیم.

	یکان	دهگان	صدگان
329	211	107	107
457	457	509	211
317	317	211	317
211	107	317	329
509	329	329	457
107	509	457	509

← نکته اینجاست که در این الگوریتم باید توجه کنیم که stable است و ترتیب برابرها رو در هر مرحله عوض نمی‌کنه

← اگه از با ارزش شروع بشه به یه سری گروه تقسیم می‌شود که در هر گروه باید الگوریتم دوباره تکرار بشه.
در اصل با این روش حواسمون هست که stable بشه و از کم ارزش میاد بالاتر.

← سوال مهم امتحان: چرا باید به جای روش متداول از Raddix استفاده کنیم؟ زمان و هزینه چه فرقی می‌کند؟

Raddix_sort (A, d)

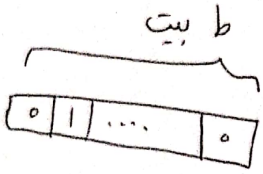
1. for $i \leftarrow 1$ to d do
2. Use a stable sort to sort A on digit i

$T(n) \in \Theta(d(n+k))$

d بار اجرای Counting sort

← اگر $d \in O(1)$ تعداد رقم ها و $K \in O(n)$ مبنا

آن گاه $T(n) \in \Theta(n)$ زمان radix sort



بعد می شود با مبنا و رقم بازی کرد مثلاً

* مبنای 2

* هر ۲ بیت یک رقم پس $b/2$ رقم می شود مبنای ۴

* هر ۳ بیت یک رقم پس $b/3$ رقم می شود مبنای ۸

$$\Longrightarrow \Theta \in \left(\frac{b}{r}(n+2^r)\right)$$

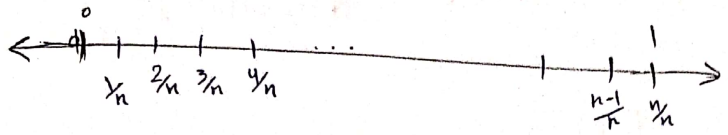
زمان الگوریتم radixsort برای b بیت در مبنای r

می توان مقدار بهینه r را مثلاً پیدا کرد.

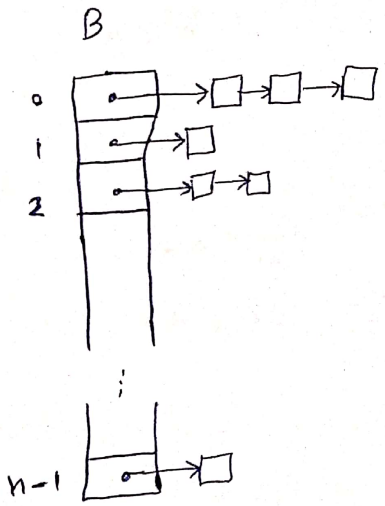
الگوریتم Bucket Sort

ورودی: n عدد حقیقی در بازه‌ی ۰ تا ۱ که به صورت یکنواخت در این بازه توزیع شده است.

نکته: به نوعی تعمیم quick sort می‌تواند باشد.



آرایه باکت



$$0 \leq B[0] < \frac{1}{n}$$

$$\frac{1}{n} \leq B[1] < \frac{2}{n}$$

$$\frac{2}{n} \leq B[2] < \frac{3}{n}$$

$$\vdots$$

$$\frac{i}{n} \leq B[i] < \frac{i+1}{n}$$

$$\vdots$$

$$\frac{n-1}{n} \leq B[n-1] \leq 1$$

هر Bucket یک لینک لیست است.

```
Bucket_Sort (A)

1. O(1)  n ← length[A]

2.  O(n) { for i ← 1 to n do
3.         {     insert A[i] into bucket B[ ⌊n × A[i]⌋ ]
                                              اندیس باکتی که باید برود

4.  O(n) { for i ← 0 to n-1 do
5.         {     Sort B[i] with insertion sort

6.  O(n)  Concatenate the bucket B[0] ... B[n-1]
```

هزینه‌ی خط ۵ و ۶ $\sum_{i=0}^{n-1} O(n_i^2)$ n_i = تعداد اعضای باکت i

اگر توزیع اعداد در باکت‌ها یکنواخت باشد میانگین n_i یک عدد ثابت است

$$\sum_{i=0}^{n-1} O(n_i) = O(n)$$

در این حالت $T(n) \in \Theta(n)$ میانگین زمان الگوریتم

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

می‌توان ثابت کرد که $E[n_i^2] = 2 - \frac{1}{n}$

$$X_{i,j} = I\{A[j] \text{ falls in bucket } i\}$$

$$n_i = \sum_{j=1}^{n} X_{i,j} \qquad 0 \leq i \leq n-1$$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{i,j}\right)^2\right]$$

$$= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij} X_{ik}\right]$$

$$= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\leq j\leq n}\sum_{\substack{1\leq k\leq n \\ k\neq j}} X_{ij} X_{ij}\right]$$

$$= \sum_{j=1} E[X_{ij}^2] + \sum_{1\leq j\leq}\sum_{\substack{1\leq k\leq n \\ k\neq j}} E[X_{ij}\cdot X_{ik}]$$

احتمال اینکه در باکت i باشد

$$E[X_{ij}^2] = 1\cdot\frac{1}{n} + 0\cdot\left(1-\frac{1}{n}\right) = \frac{1}{n}$$

$$E[X_{ij}\cdot X_{ik}] = E[X_{ij}]\cdot E[X_{ik}]$$

چون حالت $k=j$ رو جدا کردیم پس مستقل هستند

$$= \frac{1}{n}\cdot\frac{1}{n} = \frac{1}{n^2}$$

$$E[n_i] = \sum_{j=1}^{n}\frac{1}{n} + \sum\sum_{k\neq j}\frac{1}{n^2}$$

$$= 1 + \frac{n-1}{n} = 2 - \frac{1}{n} \xrightarrow{\text{بنابراین}} E[T(n)] = O(n)$$

نکته‌ای که پرسیدن در مورد Radix Sort :

* اگر براساس رقم‌های پرارزش مرتب کنیم اول چی می‌شود؟

231
125
432
212
230
505
411

رقم سمت چپ ⟹

125
231
212
230
432
411
505

⇒

125
212
231
230
911
432
505

125
212
230
231
411
432
505

← اگر از سمت چپ شروع کنیم مشکلی که پیش می‌آید باید اونهایی که صدشان یکسان دارند خودشان به گروه‌هایی تبدیل می‌شود که باید به صورت Recursive همین الگوریتم برای اون دسته اجرا بشه و...

← به صورت Recursive می‌شود.

← کدش رو باید بنویسیم

↳ تمرین: کدش را بنویسید و تحلیل زمانی کنید و بیارید؟

پیدا کردن میانه (Median)

یعنی یک عنصر که نیمی از اعداد از آن کوچکتر و نیمی از اعداد از آن بزرگتر هستند.

حالت کلی تر:

پیدا کردن k امین کوچکترین عدد یعنی عنصری که k-1 عدد از آن کوچکتر باشد.

به این مساله، مساله‌ی Selection گفته می‌شود.

راه حل‌ها:

① Sort کنیم و k امین عدد را برداریم.

راه حل‌های دیگر:

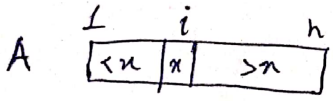
② k بار heapify ← $O(k.\lg n + n)$ (n: ساخت heap)

③ k بار min ← $O(k.n)$

④ mean میانگین ← جواب با میانه خیلی فرق دارد.

جهت کلاسی

⊛ ⑤ از Partition استفاده کنیم:



اگر $i=k$ باشد آن گاه جواب همین x است.

اگر $k<i$ باشد آن گاه دنبال جواب k امین عدد در نیمه‌ی اول آرایه می‌گردیم.

اگر $k>i$ باشد آن گاه دنبال $(k-i)$ امین عدد در نیمه‌ی دوم آرایه می‌گردیم.

```
select(A, p, r, k)
if p = r then                          c1
    return A[p]                        c2
q ← partition(A, p, r)                 Θ(n)
i ← p - q + 1
if i = k then                          c
    return A[q]
else if k < i then
    return select(A, p, q-1, k)        T(m)
else return select(A, q+1, r, k-i)     m<n
```

۵

انتخاب

تحلیل زمان k امین عدد با پارتیشن :

$$T(n) = T(m) + \Theta(n)$$

پس زمان $T(n)$ به m (اندازه هر قسمت) بستگی دارد.

اگر $m = n/2$

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

$a = 1$

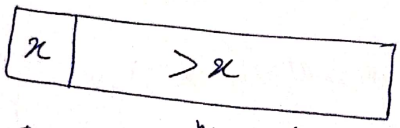
$b = 2$

$f(n) = \Theta(n)$

$n^{\log_b a} = n^{\log_2 1} = n^0 = \Theta(1) \overset{?}{<} f(n)$

یعنی اگر پارتیشن بالانس باشد آنگاه زمان الگوریتم در بدترین حالت $T(n) = \Theta(n)$ است.

اگر $m = n-1$



$m = n-1$

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

$$= \Theta(n^2)$$

در بدترین حالت مطلق

بدترین راه حل اول است که $O(n \cdot \log n)$ است

یعنی اگر پارتیشن بالانس نباشد زمان ما درجه ۲ می‌شود که بدتر از راه حل اول است

اما اگر بتوان به صورت بالانس پارتیشن کرد $T(n) \in \Theta(n)$ می‌شود

سوال: چه شاخصی انتخاب کنیم که پارتیشن بالانس باشد؟

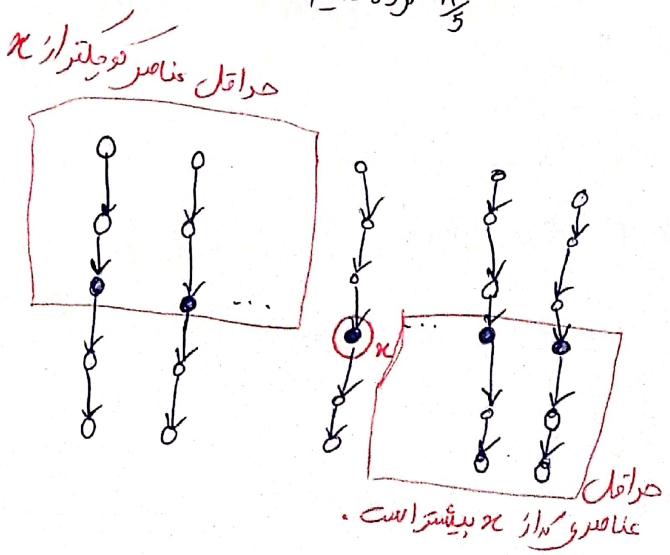
میانه !!!

۵۰

راه حل:

تقسیم اعداد به گروه های 5 تایی.

$\frac{n}{5}$ گروه داریم.



* میانه های گروه ها را برمیداریم و مجدد همین کار را برای میانه ها تکرار می کنیم، آخر به یک گروه 5 تایی می رسیم که میانه آن را برمیداریم، این میانه را میانه میانه ها می نامیم و به عنوان شاخص x استفاده می کنیم.

* حداقل $\frac{3n}{10}$ عدد از x کوچکتر است پس حداکثر $\frac{7n}{10}$ عدد از x بزرگتر است.

* حداقل $\frac{3n}{10}$ عدد از x بزرگتر است پس حداکثر $\frac{7n}{10}$ عدد از x کوچکتر است.

$$\frac{3n}{10} \leq \text{تعداد اعداد هر قسمت پارتیشن} \leq \frac{7n}{10}$$

چون می خواهیم مسالمی میانه را با میانه حل کنیم پس می آییم یه زیرمسالمی کوچکتر تعریف می کنیم که بتوانیم روی آن بصورت بازگشتی حرا بزنیم، مسالمی کوچک ما همان گروه های ۵ تایی است یعنی پیدا کردن میانه یک گروه ۵ تایی

الگوریتم selection :

۱- الگوریتم را به گروه های ۵ تایی تقسیم می کنیم. $O(n)$

۲- میانه ی هر گروه را با مرتب سازی (مثلاً درجی) پیدا می کنیم. $\frac{n}{5} \times O(1) = O(n)$

۳- الگوریتم selection را به صورت بازگشتی روی این $\frac{n}{5}$ میانه فراخوانی می کنیم تا میانه ی این میانه ها یعنی x را پیدا کنیم. $T(n/5)$

مرتب سازی ۵ عدد

۴- آرایه ی ورودی را با استفاده از شاخص x پارتیشن می کنیم. $O(n)$

فرض کنید که اندیس شاخص در آرایه ی پارتیشن شده i باشد.

۵- * اگر $i=k$ آن گاه x را برمی گردانیم و گرنه: $T(\frac{7n}{10})$ بدترین

* اگر $k<i$ آن گاه الگوریتم selection را بازگشتی روی نیمه ی اول برای پیدا کردن k امین عدد فراخوانی می کنیم

* اگر $k>i$ آن گاه الگوریتم selection را به صورت بازگشتی برای روی نیمه ی دوم برای پیدا کردن $(k-i)$ امین عدد فراخوانی می کنیم

$$T(n) = \overbrace{T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right)}^{\text{به خود } n \text{ نمی رسد } \frac{9n}{10}} + O(n)$$

⋮

باید بتوانیم با روش درخت بازگشت این را حل کنیم.

$$= O(n)$$

نکته ی مهم: با استفاده از الگوریتم selection می توان quick sort را بهبود داد که حالت بد $O(n^2)$ رخ ندهد و همواره $O(n \cdot \log n)$ باشد.

تحلیل سرشکن (Amortized Analysis)

تحلیل بدترین حالت و همواره بدترین حالت ها را در نظر می‌گیرید.

مثلاً n عمل انجام می‌شود، بعضی از اعمال $O(1)$ و بعضی $O(n)$ زمان می‌برد.

$O(n^2) = n \times O(n)$ بنابراین هزینه کل این n عمل $O(n^2)$ است. این حرف درستی است ولی دقیق نیست.

شاید با تحلیل دقیق‌تر بتوان حد بالای بهتری بدست آورد.

مثلاً با بررسی مساله ممکن است متوجه شویم که فقط تعداد ثابت از عملیات $O(n)$ است و بقیه عملیات $O(1)$ است. پس هزینه‌ی کل می‌شود:

$$C \times O(n) + (n-c) \times O(1) = O(n)$$

$$\text{هزینه سرشکن} = \frac{\text{هزینه کل}}{\text{تعداد عملیات}} = O(1)$$

تحلیل سرشکن: با تحلیل دقیق‌تر بتوانیم حد بالای بهتری برای هزینه الگوریتم‌ها بدست بیاوریم.

روش‌های تحلیل سرشکن:

1. روش تحلیل aggregate:

هزینه کل عملیات به صورت دقیق محاسبه می‌شود سپس این هزینه بر تعداد عملیات تقسیم می‌شود تا هزینه‌ی سرشکن بدست آید.

۲) روش accounting (حسابداری)

برای هر نوع عمل یک هزینه سرشکن تعریف می‌شود.
هنگام انجام یک عمل هزینه سرشکن آن پرداخت می‌شود اگر این هزینه‌ی سرشکن بیشتر از هزینه واقعی عمل باشد حساب ما طلبکار می‌شود و اگر هزینه‌ی سرشکن از هزینه واقعی کمتر باشد بابت این عمل بدهکار می‌شویم.
این طلب‌ها و بدهی‌ها در یک ساختمان داده ثبت می‌شود.

۳) روش تابع پتانسیل (Potential method)

مثل روش accounting است فقط بجای اینکه تک تک طلب‌ها یا بدهی‌ها ثبت شود سرجمع ما و طلب‌ها توسط یک تابع بیان می‌شود

* روش aggregate :

مثال stack :

عملیات
$\theta(1)$ push(s,x)
$\theta(1)$ pop(s)
$\theta(\min\{s,k\}) \in O(n)$ multipop(s,k)

بدترین حالت

n عمل انجام می‌دهیم.

```
multi_pop (s,k)
while not stack.empty(s) and k ≠ 0 do
    pop(s)
    k ← k-1
```

تحلیل بدترین حالت : $n \times O(n) = O(n^2)$

تحلیل aggregate عملیات stack :

با دقت در ماهیت push و pop می‌توان فهمید که هر چه که pop می‌شود یک زمانی push شده است.
پس کل pop هایی که در عمل pop یا multi pop انجام می‌شود، بیشتر از تعداد کل push ها نیست، اما تعداد push ها حداکثر n است پس:

هزینه‌ی کل pop ها در pop یا multipop $\in O(n)$

هزینه push ها $\in O(n)$

هزینه‌ی کل پوشته‌ها + هزینه‌ی کل Pop ها = هزینه‌ی کل n عمل $\ni O(n)$

هزینه‌ی سرشکن هر یک از عملیات multipop یا push یا Pop $= \frac{O(n)}{n} = O(1)$

شمارنده	هزینه
0000	۴
0001	۲۱
0010	۲
0011	۳۱
0100	۳
0101	۱
0110	۲
0111	۱
1000	۴
1001	۱
1010	۲
⋮	⋮

(*) شمارنده‌ی باینری:

* یک عمل Increament داریم!
* هزینه‌ی واقعی تغییر هر بیت $\Theta(1)$

هزینه‌ی Increament: هزینه: 1

کمترین { 00.....0 , 00.....1 }

بیشترین { 011.....1 , 100.....0 } هزینه: تعداد بیت شمارنده

تحلیل بدترین حالت:

n عمل Increament

هر عمل Increament در زمان حداکثر برابر تعداد بیت های شمارنده

تعداد بیت ها $\in O(\lg n)$

پس هزینه‌ی کل:

$O(\text{تعداد بیت} \times n)$

$O(\lg n \cdot n)$

تحلیل aggregate :

الگوی تغییر بیت ها در شمارنده :

بیت اول در هر بار عمل Increament تغییر می کند. پس هزینه ی کل این بیت n است.

بیت دوم در هر دو بار عمل " تغییر می کند. پس هزینه ی کل این بیت $\frac{n}{2}$ است

$\frac{n}{4}$

$\frac{n}{8}$

⋮

هزینه ی بیت $\log n$ ام می شود 1 .

پس هزینه ی کل :

$$\text{هزینه کل} = \sum_{i=0}^{\lg n} \frac{n}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} = n \frac{1}{1-\frac{1}{2}} = 2n$$

$$\text{هزینه سرشکن هر عمل increament} = \frac{2n}{n} = 2 \in O(1)$$

* روش accounting :

هر عمل که انجام می شود به آن یک هزینه ی سرشکن پرداخت می شود. اگر این هزینه سرشکن از هزینه ی واقعی عمل بیشتر باشد به اندازه ی این مقدار credit+ در نظر گرفته می شود و اگر کمتر باشد به اندازه ی credit- در نظر گرفته می شود.

$\hat{c}_i$ هزینه سرشکن عمل i ام

c_i هزینه ی واقعی عمل i ام

اگر ← $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

مهم در این روش.

در این صورت تضمین می شود که جمع هزینه های سرشکن حداقل برابر جمع هزینه های واقعی باشد.

مثال Stack :

	هزینه واقعی	هزینه سرشکن
① push	1	2
② pop	1	0
③ multipop	min(k, s)	0

تعداد عناصری که باید pop شود.

اندازه Stack

چون هر چیزی که pop می‌شود باید قبلاً push شده باشد، می‌آییم هزینه‌ی pop را زمانی که push می‌کنیم پرداخت می‌کنیم و هزینه‌ی pop و multipop را صفر می‌گیریم چون اعتبارش رو داریم و حساب شده قبلاً موقع push.

چرا 2 ← (هزینه push یک است و یک هزینه اضافی برای pop می‌دهیم)

$$\sum \hat{c}_i - c_i = \text{تعداد عناصر روی Stack} \geq 0$$

پس شرط ما برقرار است.

* برای n عمل از این سه عمل داریم:

$$2 \times n + 0 \times n + 0 \times n = 2n$$

پایان مثال

مثال: Increamining a binary counter

هزینه‌ی تبدیل بیت 1 به صفر را در زمانی که این بیت از صفر به یک تبدیل می‌کنیم پس هزینه سرشکن تبدیل بیت 0 به 1 را ۲ در نظر می‌گیریم که یک واحد برای تبدیل صرف می‌شود یک واحد به عنوان Credit ذخیره می‌شود و موقع تبدیل این بیت 1 به صفر هزینه می‌شود.

در هر عمل Increamant چند بیت 1 ممکن است صفر شود ولی فقط یک بیت صفر به یک تبدیل می‌شود پس هزینه‌ی عمل Increament نیز ۲ در نظر گرفته می‌شود

پس هزینه‌ی کل عملیات $= 2n$

$$\sum \hat{c}_i - \sum c_i = \text{تعداد بیت‌های } 1 \geq 0 \rightarrow$$

پس شرط برقرار است.

پس کل هزینه‌های ما از $2n$ بیشتر نیست.

* روش Potential :

مزیت این روش با روش قبل این است که در روش accounting اعتبار جداجدا کنار هر عنصر در ساختمان داده ذخیره می‌شود ولی در این روش یک تابع پتانسیل تعریف می‌شود که کل Credit را نشان می‌دهد.

ϕ | باید مثبت باشد

D_0 : حالت اولیه‌ی ساختمان داده

D_i : حالت ساختمان داده بعد از انجام عمل i ام

c_i : هزینه‌ی واقعی عمل i ام

ϕ : تابع پتانسیل

هزینه‌ی سرشکن عمل i ام

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

حذف تلسکوپی می‌شوند.

در نتیجه

⊗ مثال Stack:

ϕ را به صورت زیر تعریف می کنیم:

$\phi =$ تعداد عناصر داخل Stack ≥ 0

با این تابع پتانسیل، هزینه‌ی سرشکن هر عمل را حساب می کنیم و نهایتاً جمع هزینه های سرشکن حد بالای جمع هزینه های واقعی خواهد بود:

هزینه سرشکن push:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + 1 = 2$$

هزینه سرشکن pop:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (-1) = 0$$

هزینه سرشکن multi-pop:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = \min(k, s) + (-\min(k, s)) = 0$$

تعداد روی استک عوض می شود در دو حالت { تعدادی که به استک اضافه می شود / (منفی تعدادی که برداشته شده)

جمع هزینه های سرشکن $\leq 2 \times n + 0 \times n + 0 \times n = 2n$

* در کتاب مثال های پیشرفته تری موجود است مثل بزرگ کردن آرایه که مطالعه کنید.

البته نیازی نیست و برای بحث ما همین حد کافیه. ← Dynamic table

⊗ طراحی الگوریتم:

۱- روش تقسیم و غلبه (Divide & conquer)

۲- روش برنامه نویسی پویا (Dynamic programming)

۳- روش حریصانه (Greedy)

۴- روش جستجوی فضای حالات: روش عقبگرد (back tracking)

۴-۱- روش شاخه و حد (branch & bound)

* روش تقسیم و غلبه:

۳ گام دارد:

① گام تقسیم: تقسیم مساله به زیرمسایل کوچکتر

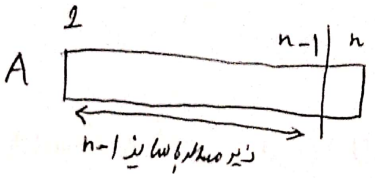
② گام بازگشتی: حل بازگشتی مسایل

③ گام ترکیب: ترکیب جواب زیر مسایل و بدست آوردن جواب مساله اصلی

مثلاً: مرتب سازی

- الگوریتم Merge sort
  - آرایه ۲ قسمت می‌شود $\Theta(1)$
  - ۲ فراخوانی نیمه اول و دوم
  - $\Theta(n)$ Merge
- الگوریتم Quick sort
  - $\Theta(n)$ Partition
  - دو فراخوانی نیمه اول و دوم
  - خالی

* الگوریتم مرتب سازی درج را نیز می‌توان یک الگوریتم مبتنی بر روش تقسیم و غلبه در نظر گرفت:



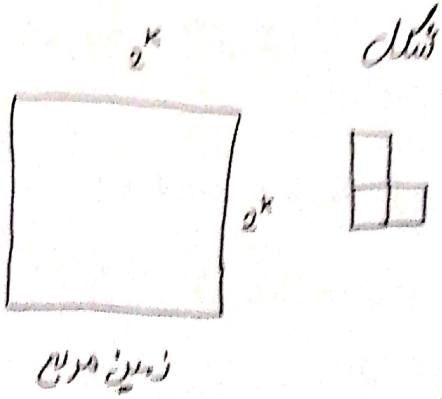
گام ۱: تقسیم آرایه به دو قسمت ۱ تا n-1 و n

گام ۲: حل بازگشتی قسمت اول

گام ۳: گام آخر در حلقه‌ی for (یکبار اجرای حلقه‌ی while

در گام آخر حلقه for در شبه‌کد نیز بازگشتی)

مساله‌ی موزائیک کردن یک زمین مربع شکل با موزائیک L شکل



مساحت زمین $= 2^k \times 2^k$

مساحت موزائیک $= 3$

می‌دانیم که $2^k \times 2^k$ می‌شود 4^k حال پس داریم: $4^k \overset{3}{\equiv} 1$

پس در بهترین حالت یک خانه‌ی خواهد ماند

پس در بهترین حالت یک خانه‌ی خواهد ماند

← هدف: جوری موزائیک می‌کنیم که فقط یک خانه‌ی 1×1 خالی بماند.

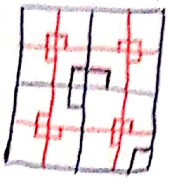
گام 1: تقسیم زمین مربع به 4 مربع مساوی

گام 2: موزائیک کردن، بازگشتی هر یک از این 4 قسمت

مساحت هر زیرمساله $= 2^{k-1} \times 2^{k-1} = 4^{k-1}$

$4^{k-1} \overset{3}{\equiv} 1 \Rightarrow$ یعنی در هر زیرمساله باز هم یک زمین 1×1 خالی می‌ماند.

پس برای حل این مشکل می‌آییم خودمان می‌کنیم زیرمساله به صورت مقابل مثلاً حل شود.



حال وسط را به کمک یک L پر می‌کنیم و ....

گام 3: قرار دادن موزائیک آخر در 3 خانه خالی وسط که کنار هم اند.

* تحلیل زمانی: $T(n) = 4T(\frac{n}{4}) + \underbrace{O(1)}_{\text{گام ۲}} + \underbrace{O(1)}_{\text{گام ۳}}$ مساحت $= n$

$T(n) \in O(n)$ ← حالت اول و پایه‌ی اصلی

مسالهی مسابقات ورزشی:

n تیم فوتبال می‌خواهند با هم مسابقه بدهند (هر یک با هر یک)

هر تیم در یک روز فقط می‌تواند با یک تیم حداکثر مسابقه بدهد.

هدف: جدول زمانبندی مسابقات در کمترین تعداد روز؟!

تیم \ روز	1	2	3 ..	m
1				
2				
3				
⋮				
n				

تعداد یال‌های گراف کامل برای جهت:

$$\text{تعداد کل مسابقات} = \frac{n(n-1)}{2}$$

حداکثر تعداد مسابقات در یک روز $\leq \left\lfloor \frac{n}{2} \right\rfloor$

حداقل تعداد روزهایی که باید مسابقه برگزار شود $= \frac{\frac{n(n-1)}{2}}{n/2} = (n-1)$ است.

(تقسیم)

* اگر n زوج باشد ← n-1 روز کافی است.
* اگر n فرد باشد ← n روز کافی است. (یه تیم فرضی می‌گیریم و نشون می‌دیم مثل زوج)

روز →

تیم ↓	1 2 3 4 ... k	k+1
1 2 3 ⋮ $\frac{n}{2}$	جدول زمانبندی مسابقات گروه A	$\frac{n}{2}+1$ ... n
$\frac{n}{2}+1$ ⋮ n	جدول مسابقات گروه B	1 2 3 ... $\frac{n}{2}$ 2 3 4 ... $\frac{n}{2}$ 1 3 4 5 .. $\frac{n}{2}$ 1 2 ⋮

گروه A

گروه B

زیرمسالهی

☒ حل مسالهی زمان‌بندی:

روش تقسیم و غلبه:

مربع لاتین

تمام ترکیب

(هر سطر آن جواب داده شده‌ی چپ به سطر دیگر است.)

مثلاً برای $n=8$ تیم:

← ابتدا $n=4$ را حل می‌کنیم

← ابتدا باید برای $n=2$ حل شود.

← ۲ تیم را مستقیم حل می‌کنیم و جدول زمان‌بندی را برمی‌گردانیم:

تیم \ روز	1	2
1		2
2		1

تعداد روز $= n-1 = 1$

حال برای $n=4$ کامل می‌کنیم:

۳ روز نیاز است

تیم \ روز	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

مربع لاتین

حال برای $n=8$ را حل می‌کنیم:

7 روز می‌خواهیم:

تیم \ روز	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	6	5	8
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	3	4	1
7	8	5	6	3	4	1	2
8	7	6	5	4	1	2	3

+4 ↓ ↑ +4

برای نصف بعدی می‌توانیم مستقلاً حل کنیم و یا اینکه مثلاً $n=4$ برای دومی فقط شماره‌ها ثابت و $+\frac{n}{2}$ کنیم ← یعنی ۴+ کنیم اینجا

(لیونه‌های مربع‌های لاتین را مشخص کردم)

بالایی هم مربع لاتین برای $n=4$ فقط شیفت به راست هر سطر شد!!

سوال: وقتی فرد $= n$ باشد چه؟

یک تیم را اضافه می‌کنیم که بشود $n+1$ و حل می‌کنیم سپس همه‌ی بازی‌های این تیم را حذف می‌کنیم در آخر.

مثلاً برای $n=3$:

$1+n=4 \xleftarrow{+1} n=3$

تیم \ روز	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

حذف 4 $\Longrightarrow$

1	2	3	استراحت
2	1	استراحت	3
3	استراحت	1	2

مثلاً برای $n=5$: $n+1=6 \leftarrow$

$\frac{6}{2}=3 \leftarrow 3+1=4$

روز \	1	2	3	4	5	
1	2	3	4	6	5	
2	1	5	3	4	6	
3	6	1	2	5	4	
4	5	6	1	2	3	1
5	4	2	6	3	1	2
6	3	4	5	1	2	3

* 5 تیم 5 روز می‌خواهد.

مربع لاتین

تبدیل اربع کردن مربع

تمام تعدی روزهای استراحت

دو تیم اربع کردیم و این بازی‌ها تکراری اند.

در نهایت پس از تشکیل مربع لاتین این ستون را پاک می‌کنیم.

$n=5 \Longrightarrow$

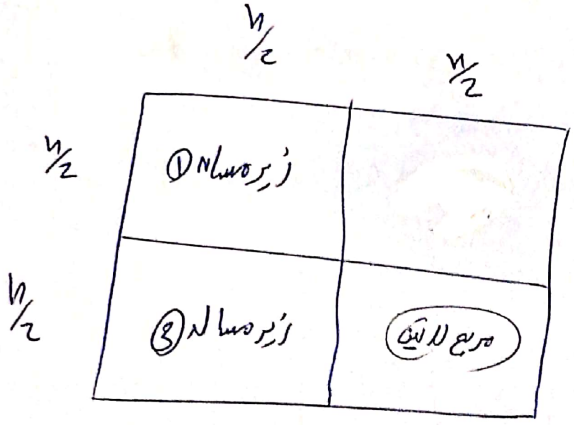
	1	2	3	4	5
1	2	3	4	–	5
2	1	5	3	4	–
3	–	1	2	5	4
4	5	–	1	2	3
5	4	2	–	3	1

روز 6

نکته: مربع بالا یک لاتین نیست.

(سطر سوم)

تحلیل زمان مسالمی زمانبندی مسابقات ورزشی :



یک ماتریس $n \times n$ باید برای زمان بندی پر شود.

سایز مساله $= N = n \times n$ اندازه ی ماتریس

$$T(N) = 2T\left(\frac{N}{4}\right) + O\left(\frac{N}{2}\right)$$

سایز زیرمساله ها $= \frac{n}{2} \times \frac{n}{2} = \frac{n \times n}{4} = \frac{N}{4}$

هزینه ی تقسیم و ترکیب

$$= O(1) + O\left(\frac{n}{2} \times \frac{n}{2} + \frac{n}{2} * \frac{n}{2}\right) = O\left(\frac{n^2}{2}\right) = O\left(\frac{N}{2}\right)$$

تقسیم

مربع لاتین بالایی ، مربع لاتین پایین

$T(N) \in O(N)$ ، یعنی $O(N)$ خطی براساس سایز مساله

* حالت سوم قضیه ی اصلی

نکته : اگر فقط یکی از زیرمساله ها را حل کنیم و اون یکی را از روی قبلی دربیاوریم :

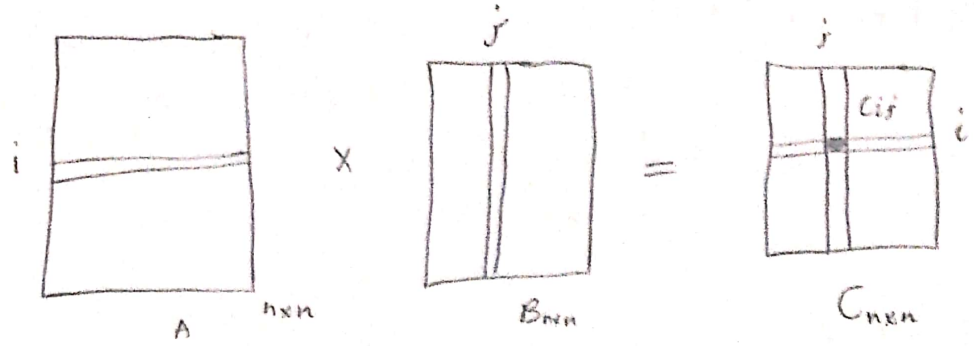
بازگشتی

$$T(N) = T\left(\frac{N}{4}\right) + O\left(\underbrace{\frac{n}{2} \times \frac{n}{2}}_{\text{زیرمساله ①}} + \underbrace{\frac{n}{2} \times \frac{n}{2}}_{\text{لاتین}} + \underbrace{\frac{n}{2} \times \frac{n}{2}}_{\text{بالایی}}\right)$$

$$= T\left(\frac{N}{4}\right) + O(N)$$

$$\xrightarrow{\text{قضیه ی اصلی}} T(N) \in O(N)$$

مسأله‌ی ضرب دو ماتریس مربعی:

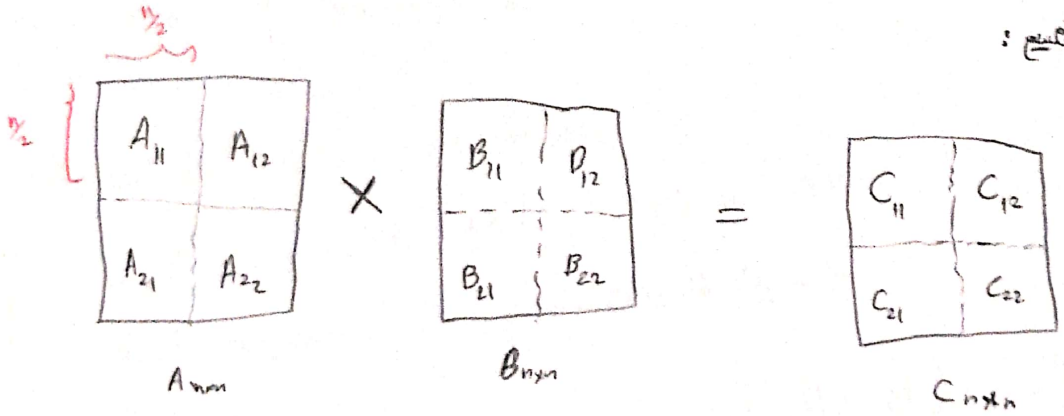


برای محاسبه‌ی C_{ij} : n ضرب و n-1 جمع داریم.

پس هزینه‌ی کل محاسبات $\Theta(n^3)$

در واقع همین روش متداول در ضرب ماتریس‌ها خودش یک روش تقسیم و غلبه است.
محاسبه‌ی هر C_{ij} یک زیرمسأله است که ضرب یک ماتریس سطری در یک ماتریس ستونی است و n^2 زیرمسأله دارد.

به روش دیگر تقسیم:



گام اول:

گام دوم: ضرب ماتریس‌های $\frac{n}{2} \times \frac{n}{2}$ در هم ← بازگشتی است.

گام سوم: چگونگی ترکیب زیرمسائل:

$$C_{11} = \underbrace{A_{11} \times B_{11}}_{\text{دارد } \frac{n^2}{4}} + \underbrace{A_{11} \times B_{21}}_{\text{دارد } \frac{n^2}{4}}$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

حل 8 زیرمسأله و ترکیب (جمع) جواب آنها پاسخ مسأله را به ما می‌دهد.

تحلیل زمانی

$$T(n) = \underbrace{8}_{\text{تعداد زیرمسئله}}\, T(\underbrace{n/2}_{\text{هزینه تقسیم}}) + \underbrace{O(1)}_{\text{هزینه ترکیب}} + O(n^2)$$

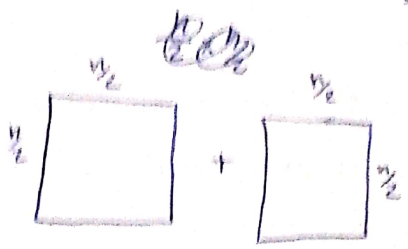
هزینه تقسیم ثابت است
و خط‌ها تازه وسط رو پیدا کنیم.

$4 \times \left(\frac{n^2}{4}\right) = n^2$

تعداد پیمایش سطرها و ستون‌ها

حالت اول
قضیه اصلی $\Rightarrow T(n) = \Theta(n^3)$ $\leftarrow \log_2 8 = 3$

* توضیح جمع‌ها:



دقیقاً $\frac{n^2}{4}$ جمع انجام می‌شود.

⊗ روش استراسن در ضرب ماتریس‌ها:

اومد گفت اون ۲ از کجا میاد؟ اون $\log_2 7$ پس گلوگاه مساله است نه اون n^2 در مساله.

اینجا، کل هشتی تعداد زیرمساله از ۸ زیرمساله به ۷ زیرمساله.

$Q_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$

$Q_2 = (A_{21} + A_{22}) \times B_{11}$

$Q_3 = A_{11} \times (B_{12} - B_{22})$

$Q_4 = A_{22} \times (-B_{11} + B_{21})$

$Q_5 = (A_{11} + A_{12}) \times B_{22}$

$Q_6 = (-A_{11} + A_{21}) \times (B_{11} + B_{12})$

$Q_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

$\Rightarrow$

$C_{11} = Q_1 + Q_4 - Q_5 + Q_7$

$C_{12} = Q_3 + Q_5$

$C_{21} = Q_2 + Q_4$

$C_{22} = Q_1 + Q_3 - Q_2 + Q_6$

اندازه S_i ها $\frac{n}{2} \times \frac{n}{2}$ است.

هزینه‌ی ترکیب و تقسیم :

$$O(1) + (10 + 8)\frac{n^2}{4} = 4.5n^2 + \overset{\text{تقسیم}}{O(1)}$$

تحلیل زمان

تحلیل زمان $\Rightarrow$ $T(n) = 7T(\frac{n}{2}) + O(n^2)$

$$= \Theta(n^{\log_2 7}) \simeq \Theta(n^{2.8})$$

این تابع برحسب بعد ماتریس بیان شده است.

ولی در واقع باید برحسب سایز مساله (حجم input) بیان شود

ورودی ما دو ماتریس $n \times n$ است :

سایز مساله $= N = n \times n + n \times n = 2n^2$

$$= \Theta(n^2)$$

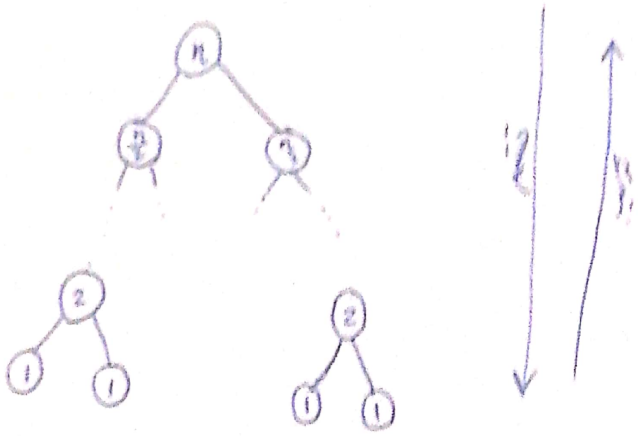
برای قبلی

$\longrightarrow T(N) = 8T(\frac{N}{4}) + O(N)$

حالت اول قضیه اصلی $\longrightarrow T(N) = \Theta(N^{\log_4 8})$

* روش برنامه‌نویسی پویا

بهبودی بر روش D&C است.



در روش D&C

این روش به صورت DFS پیاده‌سازی می‌شود

* تفاوت‌های DP و D&C :

① روش DP از پایین به بالا و سطح به سطح حل می‌کند. (فقط حرکت رو به بالا یا همان گام ترکیب رو داریم فقط)

② در DP اگر زیرمساله قبلاً حل شده باشد دوباره آن را حل نمی‌کند ولی D&C میره حل می‌کند.

③ در روش DP از خاصیت optimal substructure استفاده می‌شود.

مسائلی کوتاه‌ترین مسیر این ویژگی را دارد که هر جواب بهینه اگر به چند قسمت تقسیم شود هر قسمت خودش بهینه است.

* طولانی‌ترین مسیر این ویژگی را ندارد.

شباهت : مانند DC از حل زیرمسایل برای حل کامل مساله استفاده می‌کند.

مثال: پرانتزگذاری ضرب ماتریس‌ها

$$\underbrace{(M_1 \times M_2 \times \cdots M_k)}_{k \text{ تا اولی}} \times \underbrace{(M_{k+1} \times \cdots \times M_n)}_{n-k \text{ تا دومی}}$$

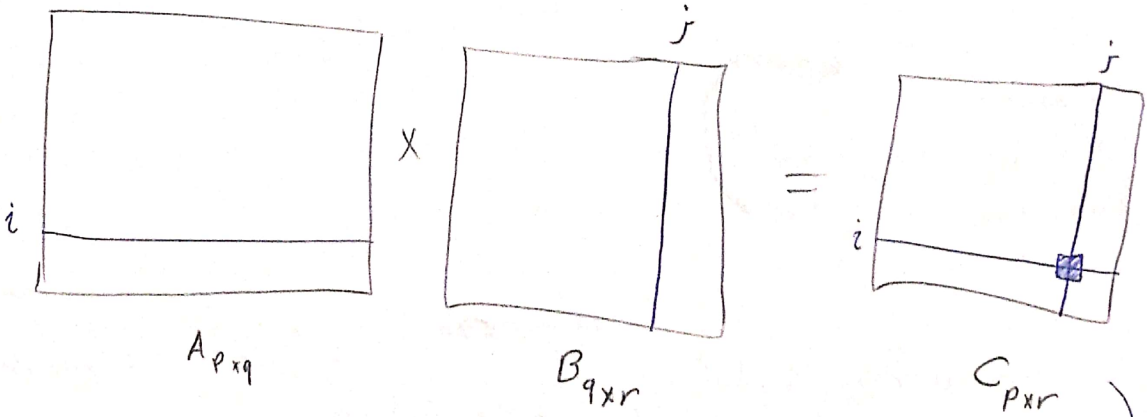
* چند نوع پرانتزگذاری داریم؟

$$P(n) = \sum_{k=1}^{n-1} P(k) \times P(n-k)$$

$$= \frac{1}{n+1} \binom{2n}{n}$$

مسأله: پیدا کردن بهترین پرانتزگذاری که تعداد ضرب هایی که انجام می‌شود کمینه شود.

← نحوه‌ی پرانتزگذاری روی تعداد ضرب های انجام شده تاثیر می‌گذارد.



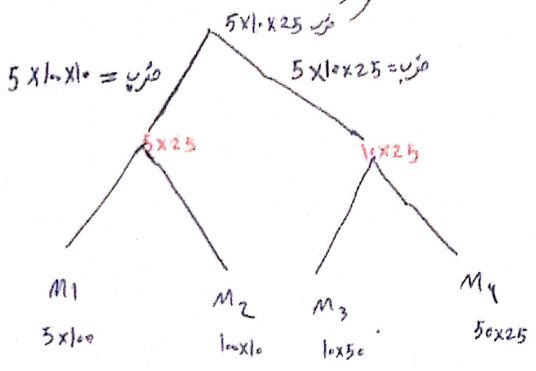
← برای هر خانه باید q تا ضرب انجام بشود.

تعداد کل ضرب ها = $p \times q \times r$

$M_1 \times M_2 \times M_3 \times M_4$

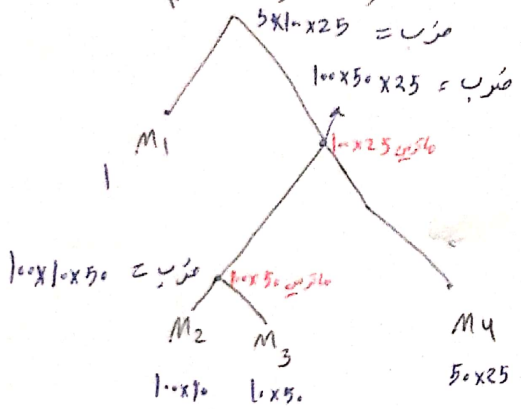
$5\times10 \quad 10\times10 \quad 10\times50 \quad 50\times25$

$((M_1 \times M_2) \times (M_3 \times M_4))$



کل ضرب ها $= 5000 + 12500 + 1250 = 18750$

$M_1 \times ((M_2 \times M_3) \times M_4)$



کل ضرب ها $= 187500$

پس این پرانتزگذاری ۱۰ برابر ضرب بیشتر می‌خواهد.

که ۱۰ برابر کمتر از ضرب بالایی است و مشخصا نحوه‌ی پرانتزگذاری آن فرق می‌کند.

← هدف: پیدا کردن پرانتزگذاری که تعداد ضرب کل را کمینه کند.

راه‌ها

- بررسی همه‌ی پرانتزگذاری‌ها ← زمان نمایی می‌شود (عدد کاتالان)
- روش ابتکاری
- استفاده از برنامه‌نویسی پویا.

* حل با DP :

چون مبنای این روش همان D&C است پس باید مساله را بشکنیم و رابطه‌ی مساله‌ی اصلی و زیرمسایلی که جلوتر را پیدا کنیم.

زیرمساله ① — آخرین ضرب — زیرمساله ②

$$(M_1 \times M_2 \times \cdots \times M_k) \times (M_{k+1} \cdots M_n)$$

* فرض می‌کنیم پرانتزگذاری بهینه را داریم و در این پرانتزگذاری، آخرین ضرب بین M_k و M_{k+1} است.

* می‌شود هر طرف را یک زیرمساله در نظر گرفت.

آیا optimal substructure برقرار است؟

OS است چون اگر زیرمساله‌ی یک optimal نباشد چون پرانتزگذاری با برهان خلف می‌توان ثابت کرد که optimal باشد و در مجموع هم چون تعداد ضرب‌ها = تعداد ضرب زیرمساله ① + تعداد ضرب زیرمساله ② + تعداد ضرب آخر

کم‌تر می‌شود که طبق پس بهینه نبوده است اگر زیرمساله بهینه نباشد.

تعداد ضرب مرحله‌ی آخر فقط به محل ضرب وابسته است نه به زیرمساله‌ها. (به k وابسته است)

تعریف دقیق‌تر رابطه‌ی مساله با زیرمسائل:

زیرمساله در حالت کلی

$$P_{ij} = ((\underbrace{M_i \times \cdots \times M_k}_{d_{i-1} \times d_k}) \times (\underbrace{M_{k+1} \times \cdots \times M_j}_{d_k \times d_j}))$$

ابعاد $d_{i-1} \times d_i$ ، $d_{k-1} \times d_k$ ، $d_k \times d_{k+1}$ ، $d_{j-1} \times d_j$

تعداد ضرب‌های آخر $= d_{i-1} \times d_k \times d_j$

C_{ij} = تعداد ضرب‌ها در مساله‌ی P_{ij} $= C_{ik} + C_{k+1,j} + d_{i-1} \times d_k \, d_j$

با فرض اینکه ضرب آخر بین M_{k+1} و M_k است.

چون نمی‌دونیم آخری کجاست می‌آییم Min می‌گیریم از همه جاهاش رو پیدا کنیم:

$$C_{i,j} = \min_{i \le k < j} \left\{ C_{i,k} + C_{k+1,j} + d_{i-1} \times d_k \times d_j \right\}$$

فرمول نهایی تعداد ضرب ها در مسالمی $P_{i,j}$

زیر مساله ها و تکرار دوباره حل نمی‌شوند:

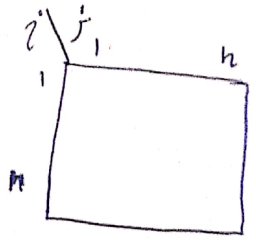
برای نگه‌داری $C_{i,j}$ ها چه حافظه‌ای مناسب است؟

$$C_{i,j} = \min_{i \le k < j} \{ \dots \}$$

$$\text{مساله اصلی} = C_{1,n} = \min_{1 \le k < n} \{ \dots \}$$

$1 \le i \le n$

$1 \le j \le n$



$C_{n \times n}$

کوچکترین زیرمساله؟

$P_{i,j}$: M_{i} $d_{i-1} \times d_i$

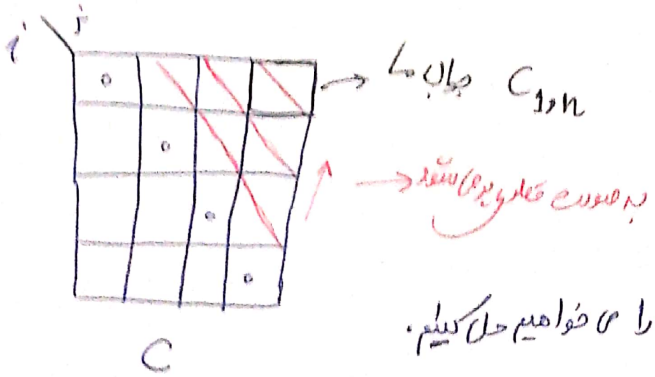
$C_{i,j} = \emptyset$

مثلا برای ۴ ماتریس:

$M_1 \times M_2 \times M_3 \times M_4$

$5\times100 \quad 100\times10 \quad 10\times50 \quad 50\times25$

$C_{i,i} = 0$



نکته: زیرمساله‌ها باید به ترتیبی حل شوند که وقتی یک مساله را می‌خواهیم حل کنیم، قبلاً تمام زیرمسایل آن حل شده باشد.

$$C_{12} = \min_{1\le k<2} \{ C_{1,k} + C_{k+1,2} + d_0 d_k d_2 \}$$ فقط می‌تواند $k=1$ باشد

$$C_{1,2} = C_{1,1} + C_{2,2} + d_0 d_1 d_2$$

$$= 0 + 0 + 5\times100\times10 = 5000$$

$$C_{2,3} = \min_{2\le k<3} \{ C_{2,k} + C_{k+1,3} + d_1 d_k d_3 \}$$ فقط $k=2$

$$= C_{2,2} + C_{3,3} + d_1 d_2 d_3 = 0 + 0 + 100\times10\times50 = 50{,}000$$

$$C_{3,4} = \min_{3\le k<4} \{ C_{3,k} + C_{k+1,4} + d_2 d_k d_4 \}$$ فقط $k=3$

$$= 0 + 0 + 10\times50\times25 = 12500$$

زیرمساله‌ها باید ۲ حل شوند.

(۷۴)

$$C_{1,3} = \min_{1 \le k < 3} \{ C_{1,k} + C_{k+1,3} + d_0 d_k d_3 \}$$

$$= \min \begin{cases} k=1 \rightarrow C_{1,1} + C_{2,3} + d_0 d_1 d_3 = 0 + 50000 + 5 \times 100 \times 50 \\ \qquad = 75000 \\ k=2 \rightarrow C_{1,2} + C_{3,3} + d_0 d_2 d_3 = 5000 + 0 + 5 \times 10 \times 50 \\ \qquad = 7500 \end{cases}$$

$$C_{2,4} = \min_{2 \le k < 4} \{ C_{2,k} + C_{k+1,4} + d_1 d_k d_4 \}$$

$$= \min \begin{cases} k=2 \rightarrow C_{2,2} + C_{3,4} + d_1 d_2 d_4 = 0 + 12500 + 100 \times 10 \times 25 = 37500 \\ k=3 \rightarrow C_{2,3} + C_{4,4} + d_1 d_3 d_4 = 50000 + 0 + 100 \times 50 \times 25 = 175000 \end{cases}$$

زیرمسأله‌های ۲ تایی حل شد!

$$C_{1,4} = \min_{1 \le k < 4} \{ C_{1,k} + C_{k+1,4} + d_0 d_k d_4 \}$$

$$= \min \begin{cases} k=1 \rightarrow C_{1,1} + C_{2,4} + d_0 d_1 d_4 = 0 + 37500 + 5 \times 100 \times 25 \\ \qquad = 50000 \\ k=2 \rightarrow C_{1,2} + C_{3,4} + d_0 d_2 d_4 = 5000 + 12500 + 5 \times 10 \times 25 \\ \qquad = 18750 \\ k=3 \rightarrow C_{1,3} + C_{4,4} + d_0 d_3 d_4 = 7500 + 0 + 5 \times 50 \times 25 \\ \qquad = 13750 \end{cases}$$

از روی محاسبات قبلی پس از هر مرحله جدول زیر پر می‌شود:

$i \backslash j$	1	2	3	4
1	0	5000 k=1	7500 k=2	12750 k=3
2		0	5000 k=2	3750 k=2
3			0	1250 k=3
4				0

C

تعداد کل ضرب‌ها

جواب نهایی از روی k=3 می‌فهمیم که

$((M_1 \times M_2 \times M_3) \times M_4)$

چگونه پرانتزگذاری شود

ضرب آخر

از روی ماتریس مقابل می‌فهمیم

$(M_1 \times M_2) \times M_3$

ابتدا دو تا را ضرب

تحلیل زمانی:

* یک روش بدست آوردن بهترین حالت پرانتزگذاری شمارش تعداد حالات است که زمانش صرف تحلیل در درجه‌ی عدد کاتالان است ←

$$\Omega\left(\frac{1}{n+1}\binom{2n}{n}\right)$$

* زمان برای روش برنامه‌نویسی پویا:

$O(n^2)$ زیرمساله حل می‌شود.

$$C_{i,j} = \min_{i \le k < j} \{ C_{i,k} + C_{k+1,j} + d_{i-1} d_k d_j \}$$

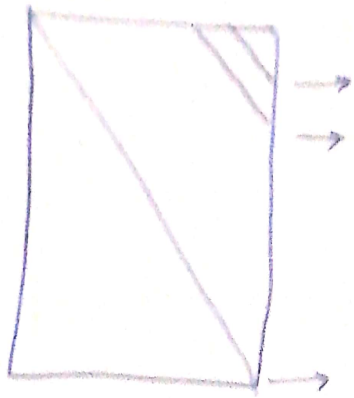
زمان محاسبه $\in O(n)$

k حداکثر n-1 حالت دارد.

در هر حالت محاسبه عبارت زمان $O(1)$ می‌بود که در کل می‌شود $O(n)$

زمان حل مساله = زمان هر زیرمساله × تعداد زیرمساله $= O(n) \times O(n^2) = O(n^3)$

سوال: سر شکل حساب کنیم چی می‌شود؟



یک مساحتی $n-1$

دو مساحتی $n-2$

$\vdots$

n مساحتی 1

→ در نهایت در ثابت فرق می‌کند نه در درجه.

$$(n-1)\times 1+(n-2)\times 2+\cdots+(n-(n-1))\times(n-1)$$

$$=\sum_{k=1}^{n-1}(n-k)k=n\sum_{k=1}^{n-1}k-\sum_{k=1}^{n-1}k^2$$

$$=\cdots\quad O(n^3)$$

(۷۷) (LCS) (Longest common subsequence) ← مساله پیدا کردن طولانی ترین زیردنباله مشترک

$\vec{a} = 1, 3, 5, 2, 1, 4, 3$

$\vec{b} = 3, 5, 1, 2, 1, 3$

n حرف

m حرف

چند تا از زیردنباله های a →

$$\begin{cases} 1,3,1,3 \\ 5,2,1,3 \\ 1,5,3 \end{cases}$$

زیردنباله مشترک b و a $= 3, 5, 2, 1, 3$

$$\sum_{k=1}^{m} \binom{m}{k} \times O(n) = O(2^m \cdot n)$$

فرض $m < n$

راه حل: پیدا کردن زیررشته های یکی در دیگری:

که زمان ما نمایی می شود.

تعداد زیردنباله های $m-k$ تایی رشته به طول m

↓

(k تا حذف می کنیم بقیش میشه رشته برو مانده)

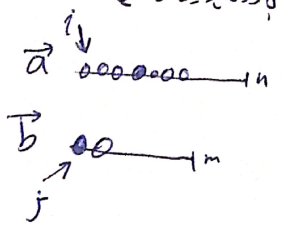
البته

$$\boxed{\binom{m}{k} = \binom{m}{m-k}}$$

زمان پیدا کردن رشته $m-k$ تایی در رشته n تایی

↓

با دو تا پوینتر و چک کردن انجام می شود.



راه دیگر (استفاده از برنامه‌ریزی پویا):

برای حل کردن این مساله با D.P. فرض می‌کنیم مساله حل شده و جواب را داریم:

طولانی‌ترین زیر دنباله مشترک $\vec{b}$ و $\vec{a}$

$\vec{C}: c_1, c_2, \dots c_{k-1}, c_k$

$\vec{a}: a_1, a_2, \dots a_n$

$\vec{b}: b_1, b_2, \dots b_m$

* اگر $a_n = b_m$ آن‌گاه $c_k = a_n = b_m$ است حتماً (برهان خلف می‌شود اثبات کرد)
* اگر $a_n \neq b_m$ آن‌گاه ۳ حالت داریم (حتماً یکی از این ۳ حالت است).

$c_k = a_n \neq b_m$ ① یا

$c_k = b_m \neq a_n$ ② یا

$c_k \neq a_n$
$c_k \neq b_m$ ③ یا

مساله‌ی $P_{ij} = \begin{cases} \vec{a}: a_1, a_2 \dots a_i \\ \vec{b}: b_1, b_2, b_3, \dots b_j \end{cases}$

می‌تواند رخ بدهد چون از درخت دیگر بزرگتر نیست.

طول طولانی‌ترین زیردنباله مشترک

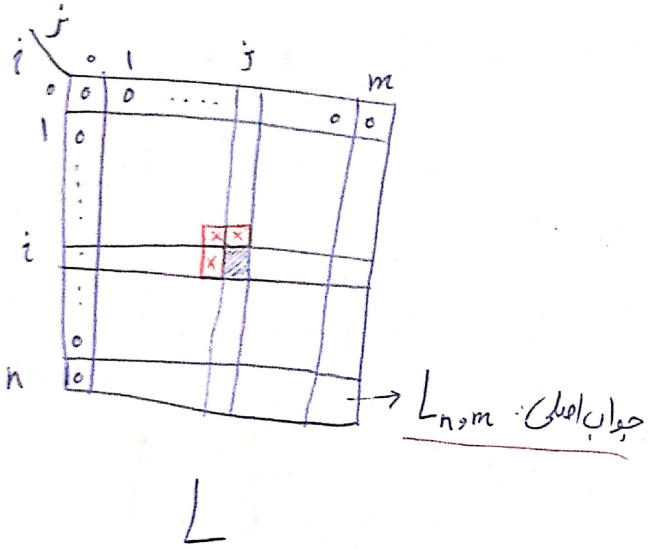
$$L_{ij} = \begin{cases} L_{i-1,j-1} + 1 & \text{اگر } b_j = a_i \\ \max\{L_{i-1,j}, L_{i,j-1}, \cancel{L_{i-1,j-1}}\} & \text{وگرنه} \end{cases}$$

ما فقط یک دلیل جواب ها.

مسالمی $0 \le i \le n$

$0 \le j \le m$

* آرایه دو بعدی $(n+1)\times(m+1)$



* کوچکترین زیرمسالمه

$L_{i,j}=0$ اگر $i=0$ یا $j=0$

L

جدول باید طوری پر شود که وقتی می خواهیم حل کنیم که زیر مسالمه ها پیش حل شده باشند.

روی شکل با توجه به رابطه برای حل مربع ها دستور خورده

باید جواب مربع های مشخص زده شده را داشته باشیم. (از روی رابطه ی $L_{i,j}$)

پس خانه های ماتریس را سطری (یا) سطری و از چپ به راست و بالا به پایین حل می کنیم.

ما می آییم به صورت از چپ به راست و از بالا به پایین به صورت سطری حل می کنیم.

دو تا for نیاز داره که

با یک مثال جلو می رویم:

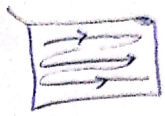
مثال:

$\vec{a}$: a b a a c d b a

$\vec{b}$: a a b c d

فلش نشان می دهد که جواب از کدام خانه ی قبلی آمده است.

	b		a	a	b	c	d
a		0	1	2	3	4	5
a	0	0	0	0	0	0	0
a	1	0	0+1=1 (a=a)	1 (a=a)	1 (b≠a)	1 (c≠a)	1 (d=a)
b	2	0	1	1 (max در سطر)	2	2	2
a	3	0	1	2	2	2	2
a	4	0	1	2	2	2	2
c	5	0	1	2	2	3	3
d	6	0	1	2	2	3	4
b	7	0	1	2	3	3	4
a	8	0	1	2	3	3	4



به این صورت خانه ها را پر می کنیم

جواب نهایی $L_{n,m} \rightarrow L_{8,5}$

طولانی ترین دنباله ی مشترک طولش ۴ است.

با فلش ها هم خود زیر دنباله را در می آوریم:

وقتی ↖ می آییم یعنی برابر بوده می توییم از هر دو کم کنیم و در مشترک بنویسیم!

وقتی در بقیه موارد یعنی ↑ یا ← یعنی مساوی نیستند و یکی کم می کنیم از دنباله و می رویم جلو.

LCS:

a, b, c, d — برای شاخه بالایی

a, a, c, d — برای شاخه پایینی

اونجایی که دو مسیر داریم:

مرتبی نمی کند در کل طول ۴ می شود.

تحلیل زمانی:

زمان کل = تعداد زیرمساله × زمان حل یک زیرمساله

تعداد زیرمسایل: $(n+1)(m+1) = O(m.n)$

زمان حل یک زیرمساله = زمان حل تروتو L = $O(1)$ → قبلی‌ها را داریم و فقط باید مقایسه و یه عملیات محاسبه می‌شود.

زمان کل $= O(m.n) \cdot O(1) = O(m.n)$

← درجه‌ی ۲ برحسب سایز مساله که $O(n+m)$ است.
سایز

* مسأله‌ی مثلث‌بندی بهینه‌ی چند ضلعی محدب (Convex polygon triangulation):

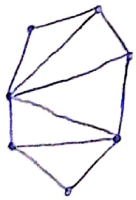
Convex polygon
محدب

Polygon

مثلث‌بندی: رسم وترهای غیر متقاطع که مثلث‌بندی کنند دیگه!!

← مثلث‌بندی که مجموع طول وترهای آن کمینه باشد.

مثلاً



(* همیشه فرض می‌کنیم که ما راه‌حل بهینه‌ی دو ضلعی را از زاویه میدیم ولی می‌کنیم در DP)

می‌توانیم به دو قسمت کنیم و هر یک را جداگانه حل کنیم که
به صورت بهینه حل شود ← خاصیت O.S. را دارد.

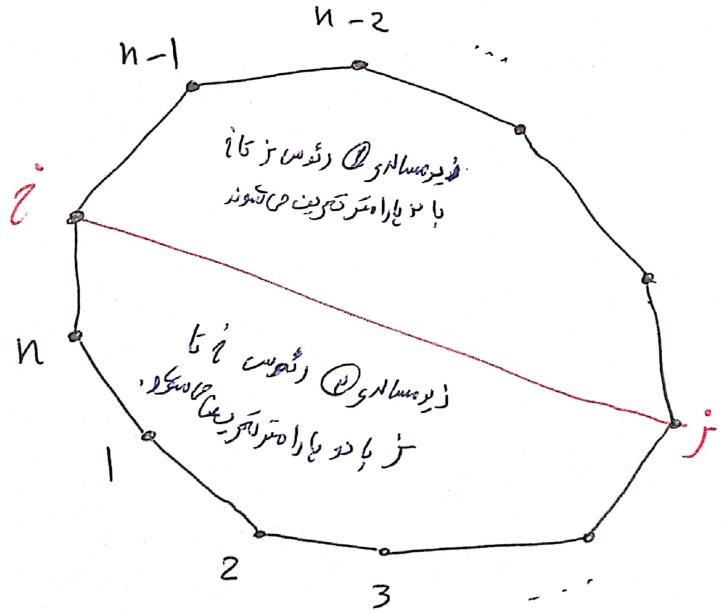
* یک انتخاب طبیعی برای تقسیم مساله به زیرمسایل این است که یک وتر را در نظر بگیریم و این وتر مساله را به ۲ زیرمساله تقسیم می کند!

سوال) آیا این نحوه‌ی تقسیم مناسب است؟

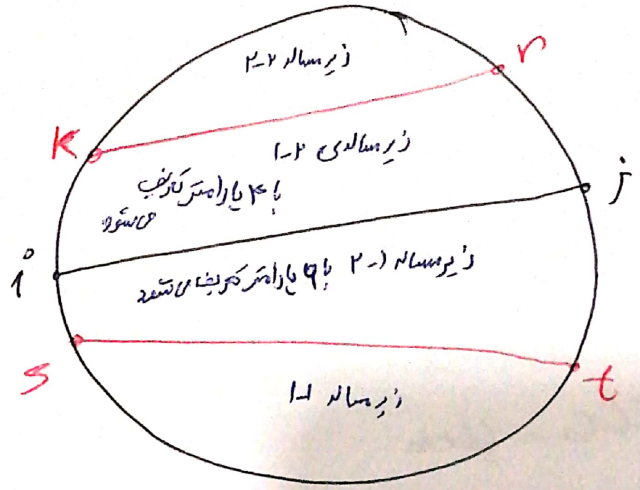
یعنی ← آیا این نحوه‌ی تقسیم به راحتی برای زیرمسایل هم قابل تعمیم است؟

آیا زمان الگوریتم حاصل از این تقسیم مناسب است؟

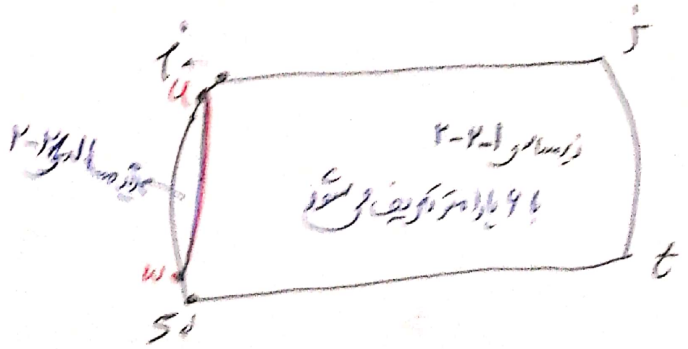
تقسیم مسأله:



تقسیم هر زیرمساله:



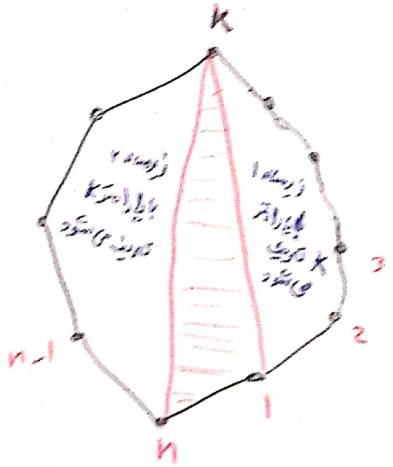
تقسیم بیشتر زیرمسایل:



تقسیم زیرمساله ۱-۲

* اگر هر زیرمساله با k پارامتر تعریف شود آن‌گاه تعداد زیرمسایل برابر است با $\binom{n}{k} \in O(n^k)$

* بنابراین این نحوه‌ی تقسیم مناسب نیست چون به راحتی قابل تقسیم نیست و تعداد زیرمسایل زیاد می‌شود.



روش تقسیم را عوض می‌کنیم:

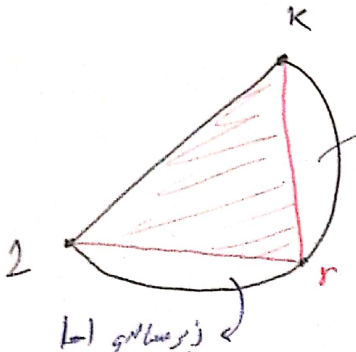
یک راس می‌گیریم مثل k و به او n وصل می‌کنیم.

یک راه ماشوره

آخری

اولی

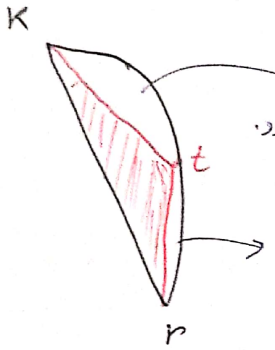
تعمیم و تقسیم بیشتر زیرمساله ها :



زیرمسالهی ۱-۲
با ۲ پارامتر تعریف می شود.
پارامترهای k و r .

زیرمسالهی ۱-۱
با یک پارامتر r تعریف می شود.

تقسیم بیشتر زیرمسالهی ۱-۲ :



زیرمسالهی ۱-۲-۲
با دو پارامتر t و k تعریف می شود.

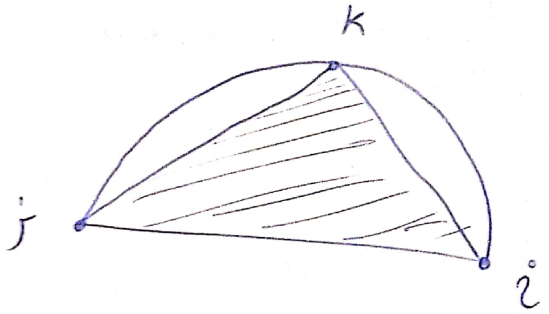
زیرمسالهی ۱-۲-۱ با
دو پارامتر r و t تعریف
می شود.

نتیجه : پس هر زیرمساله با حداکثر ۲ پارامتر تعریف می شود پس تعداد زیرمسایل برابر است با :

$$\binom{n}{2} \in O(n^2)$$

نکته‌ی دیگر: هر زیرمساله‌ای که با ۲ پارامتر تعریف شود می‌توان نشان داد که زیرمساله‌های آن هم حداکثر با ۲ پارامتر تعریف می‌شوند.

مهم: می‌توان ثابت کرد با تغییر k هر مثلث بندی را تولید کرد.



زیرمساله : P_{ij}

طول وترها : L_{ij}

$$L_{ij} = \min_{i+1 \le k \le j-1} \{ L_{ik} + L_{kj} + \lambda(i,k) + \lambda(k,j) \}$$

* زمان حل هر زیرمساله: در بدترین حالت $i=1$ و $j=n$ است که باید بین $n-2$ حالت min بگیریم پس $O(n)$ می‌شود.

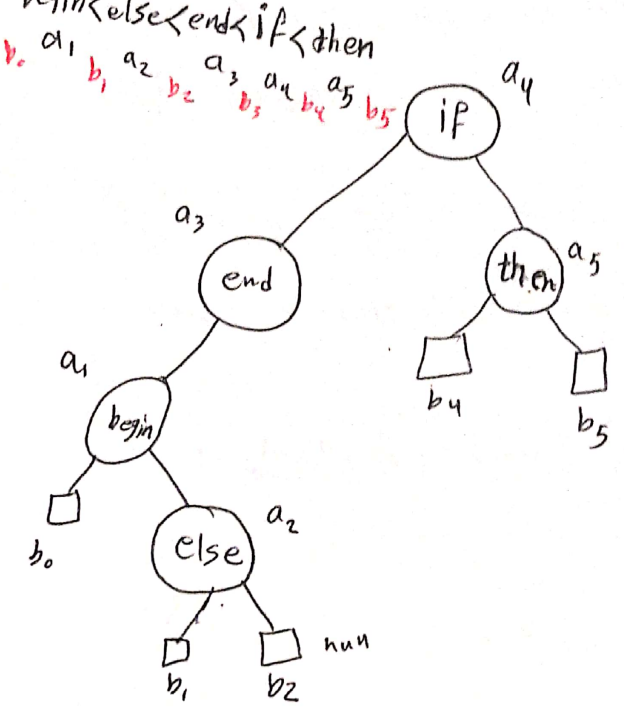
$O(n^3)$ زمان کل ← $O(n^2)$ تعداد زیرمسایل
$O(n)$ زمان حل هر زیرمساله } *

* مساله‌ی پیدا کردن درخت جست‌وجوی بهینه دودویی: (Optimal Binary Search Tree) (OBST):

فرض کنید n کلمه داریم و می‌خواهیم با این کلمات یک BST بسازیم (دیکشنری)

$$a_1 < a_2 < a_3 \dots < a_n < a_n \rightarrow$$ مرتب کلمات

begin < else < end < if < then



مثل
Ali < Reza
Ali < AliReza

جست‌وجوی موفق مثل جست‌وجوی کلمه‌ی else

جست‌وجوی ناموفق مثل جست‌وجوی کلمه‌ی ali

هدف: با داشتن تعداد کلمه و تکرار آنها در متن می‌خواهیم BST بسازیم که در کل برای یافتن این کلمات در BST کمترین تعداد مقایسه انجام شود.

$P_{i,j}$: $a_{i+1} < a_{i+2} < \dots < a_j$

$a_{i+1} \to p_{i+1}$ ، $a_{i+2} \to p_{i+2}$ ، $a_j \to p_j$

تعداد تکرار کلمه معتبر

بزرگتر از a_{i+1} است ولی کوچکتر از a_{i+2} می‌باشد

b_i ، b_{i+1} ، b_{i+2} ، b_{j-1} ، b_j

کلمات نامعتبر

$b_i \to q_i$ ، $b_{i+1} \to q_{i+1}$ ، $b_{i+2} \to q_{i+2}$ ، $b_{j-1} \to q_{j-1}$ ، $b_j \to q_j$

تعداد کلمات نامعتبر

تکرار

خروجی: OBST درخت BST بهینه:

که اگر کلمات معتبر و نامعتبر موجود در متن را در BST جستجو کنم کمترین تعداد مقایسه در کل انجام می‌شود.

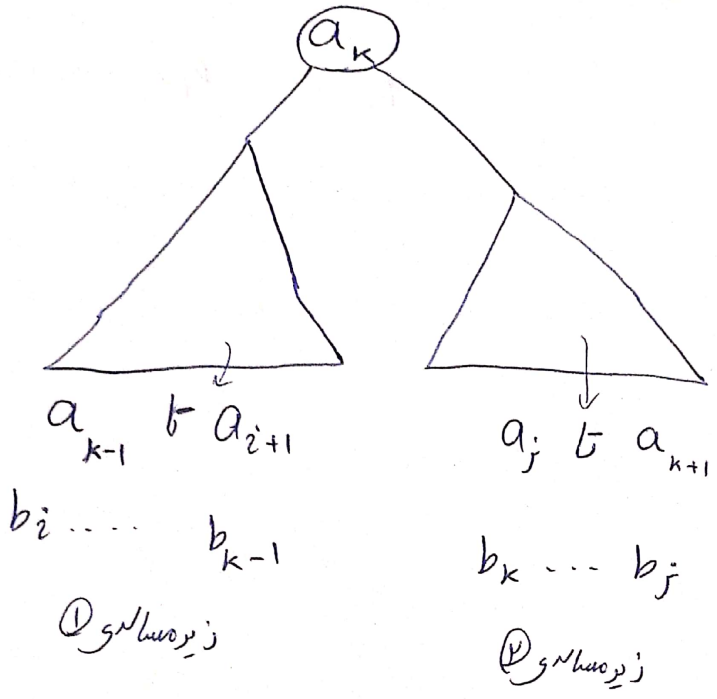
یک راه حل: ساخت تمام درخت‌های BST که عدد کاتالان تا است و محاسبه‌ی تعداد مقایسه‌ها و پیدا کردن بهینه ← زمان نمایی

راه حل دوم: استفاده از برنامه‌ریزی پویا

تمرین: (امتحانی)

روش دوم: برنامه ریزی پویا:

فرض کنید که درخت OBST پیدا شده است:



نحوه شکست به زیرمساله ها:

شکستن از ریشه

چون برای هر کلمه در هر صورت با ریشه مقایسه خواهد شد.

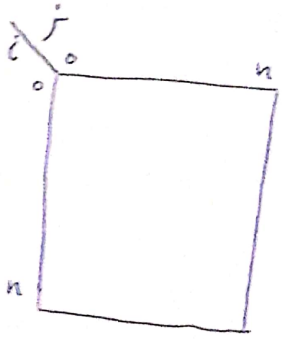
تعداد مقایسه ها در مساله = تعداد مقایسه ها در زیرمساله ① + تعداد مقایسه ها در زیرمساله ② + تعداد کل کلمات معتبر و نامعتبر

$\Rightarrow$ تعداد کل کلمات $C_{i,j} = \underbrace{\left(q_i + \sum_{r=i+1}^{j} (p_r + q_r)\right)}_{\text{تعداد کل کلمات } W_{i,j}} + \underbrace{C_{i,k-1}}_{\text{زیرمساله ①}} + \underbrace{C_{k,j}}_{\text{زیرمساله ②}}$

$$C_{i,j} = \min_{i<k\leq j} \left\{ C_{i,k-1} + C_{k,j} \right\} + W_{i,j}$$

$$W_{i,j} = W_{i-1,j-1} + (q_i + p_i)$$

حافظه‌ی نگه‌داری جواب‌ها:



C یا w

نکته: a_i ها کلمات معتبر زبان هستند ولی b_i ها مجموعه از کلمات هستند مثلاً b_i همه کلماتی که کوچکتر از a_{i+1} اند و بزرگتر از a_i هستند.

مثلاً تعداد کلمه‌ای بالا باشد بهتره توی درخت بالا باشه که کمترین مقایسه نیاز باشه برای پیدا کردنش و اون کلماتی که کمتر هستند در متن چون کم‌اند اشکالی ندارد در درخت پایین باشند (بچینه)

$$P_{i,i+1} = a_{i+1}$$

$b_i \to q_i$

$P_{i+1} \to b_{i+1} \to q_{i+1}$

کوچکترین زیرمسأله یک درخت با یک کلمه‌ی معتبر است.

$$\hookrightarrow C_{i,i+1} = q_i + P_{i+1} + q_{i+1}$$

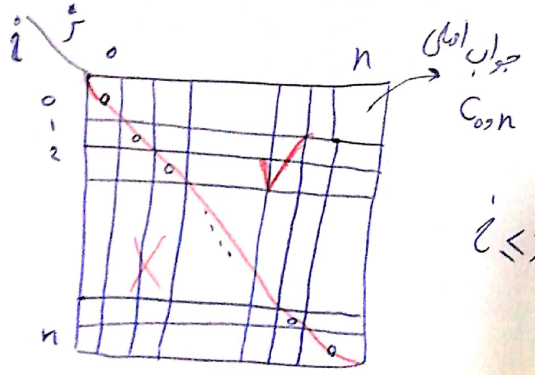
یا

فرقی نمی‌کند کدام را کوچکترین بگیریم.

$P_{i,i}$ درخت تهی

$C_{i,i} = \emptyset$

کوچکترین زیرمسأله (*)



$j \geq i$

ما حافظه را به صورت بالا مثلثی کار می‌کنیم

باید به صورت قطری پر شود تا در بعدی استفاده شود.

تمرین: (امتحانی)

مثال:

i	0	1	2	3	4	5
p_i		0.15	0.10	.05	.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.1

از اول کمتر

از دوم کمتر

از اول بیشتر

۵ کلمه معتبر

کلاً 60 درصد کلمات نامعتبرند درصد کلمات معتبر

کلاً 40 درصد کلمات نامعتبرند درصد کلمات نامعتبر

OBST را بدست آورید.

باید ماتریس رو بکشیم و حل کنیم.

به عنوان تمرین حل شود.

تحلیل زمانی:

هزینه‌ی کل = تعداد زیرمسایل × زمان حل یک زیرمساله

$= O(n^2) \times O(n)$

تعداد زیرمسایل: تعداد ماتریس

$= O(n^3)$

min بین h مقدار

نکته‌ی قابل توجه:

بهتره کوچکترین زیرمساله را همان درخت

بودجه بگیریم چون اگر درخت یک کلمه معتبر بگیریم ممکن است در مساله‌ای به درخت بودجه بخوریم و این می‌شود استثنا ولی اگر از اول بودجه بگیریم درخت یک کلمه معتبر را یکی بریم پایین می‌شود بودجه و استثنا نخواهیم داشت روی کوچکترین زیرمساله. (در کل فرق نمی‌کند)

نکته: (بهبود حد بالای زمان اجرا)

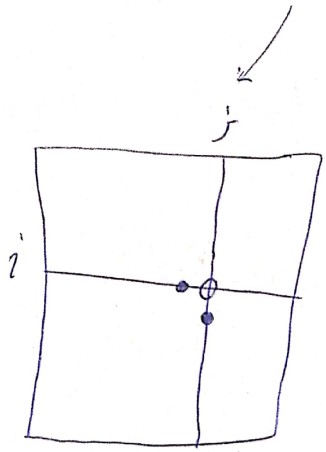
$r_{\text{ژوج}} =$ اندیس ریشه درخت مساله $P_{\text{ژوج}}$
(یعنی k که a_k ریشه است)

می‌توان نشان داد:

$$r_{i,\text{ژوج}-1} \leq r_{i,\text{ژوج}} \leq r_{i+1,\text{ژوج}}$$

پس نیاز نیست کل دو بررسی کنیم
برای پیدا کردن k می‌دونیم که مابین
اندیس ریشه قبلی و بعدی است.

$$\underbrace{a_{i+1} \leq a_{i+2} \leq a_{i+3} \leq \cdots \leq a_{\text{ژوج}-1}}_{r_{i,\text{ژوج}-1}} \leq \overbrace{a_{\text{ژوج}}}^{r_{i+1,\text{ژوج}}}$$



مهم: با توجه به نکته‌ی گفته شده می‌توان اثبات کرد
زمان اجرای الگوریتم $O(n^2)$ است.
(به عنوان تمرین این کار را انجام دهیم)

* روش حریصانه: (Greedy)

این روش بهبود یافته‌ی برنامه نویسی پویا است.

در روش برنامه نویسی پویا چند option وجود دارد که جلو از قبل نمی دانیم که کدام option ها را به جواب بهینه می رساند همه را در نظر می گیریم و بین آنها min یا max می گیریم.
ولی در روش حریصانه با یک استدلال فقط یک option در نظر گرفته می شود.

* مسأله‌ی ضرب ماتریس ها (پرانتزگذاری)

$$M_1 \times M_2 \times M_3 \times M_4$$

در DP آخرین ضرب می تواند بین M_1 تا M_4 باشد و نمی دانیم کجاست پس همه را در نظر می گیریم،

ولی در روش حریصانه برای

$$\underset{5\times10}{M_1} \times \underset{10\times1}{M_2} \times \underset{1\times20}{M_3} \times \underset{20\times30}{M_4}$$

استدلال: بهتر است آن دو ماتریس که بُعد بین آنها بیشترین است اول ضرب شود.

$$((M_1 \times M_2) \times (M_3 \times M_4))$$

* روش حریصانه از بین چند option در هر مرحله فقط یک option را انتخاب می کند، که به این انتخاب، انتخاب حریصانه گفته می شود (Greedy Choice).

تذکر: این انتخاب لزوماً منجر به انتخاب بهینه نمی شود.

اگر بهینه بود می گوییم الگوریتم حریصانه ما بهینه است ولی ممکن است در مسایلی بهینه نباشد.

* روش حریصانه بهبودی برای برای روش D.P است.

* در روش حریصانه فقط یک انتخاب را در نظر می‌گیریم (انتخاب حریصانه) برخلاف D.P که همه‌ی انتخاب‌ها را در نظر می‌گیرد.

در روش حریصانه از همان ابتدا روی انتخاب حریصانه تمرکز می‌کنیم:

مثال: فرض کنید یک اسکناس ۲۸ تومانی داریم و سکه‌های ۱ و ۲ و ۵ و ۱۰ تومانی داریم.

می‌خواهیم اسکناس را به گونه‌ای خرد کنیم که کمترین تعداد سکه را داشته باشیم.

انتخاب حریصانه: ابتدا انتخاب سکه با ارزش بیشتر:

$$28 = \underline{\underline{2}} \times 10 + \underline{\underline{1}} \times 5 + \underline{\underline{1}} \times 2 + \underline{\underline{1}} \times 1$$

5 سکه انتخاب می‌شود.

آیا این انتخاب بهینه است؟ بله بهینه است و کمترین می‌شود.

مثال دیگر: اسکناس ۱۵ تومانی و سکه‌ها ۱ و 5 و 11 تومانی

$$15 = 1 \times 11 + 4 \times 1$$

تعداد سکه‌ها = 5 تا

آیا این انتخاب حریصانه بهینه است؟ خیر چون با 2 سکه 5 و 2 سکه‌ی 1 می‌شود با ۳ سکه همین اسکناس را خرد کرد.

نکته: پس الگوریتم‌های حریصانه لزوماً جواب بهینه را نمی‌دهند و اشکالی هم ندارد ولی تلاش می‌کنیم که جواب بهینه را بدهد.

⊗ مسأله‌ی زمان‌بندی به کمک حریصانه:

⊗ n کار داریم که زمان انجام هر کدام $t_1 \dots t_n$ است. ترتیب زمان اجرا کارها چگونه باشد که متوسط زمان پاسخ کمینه باشد؟

مثلاً برای سه کار:

ترتیب اجرا کارها		زمان پاسخ
T_1		t_1
T_2		t_1+t_2
T_3		$t_1+t_2+t_3$

میانگین زمان پاسخ $= \dfrac{3t_1+2t_2+t_3}{3}$

اگر بخواهیم این میانگین زمان پاسخ کمینه شود، کار T_1 و T_2 و T_3 بایستی چه زمان اجراهایی داشته باشند؟ یعنی اجرای T_1 چه ویژگی باید داشته باشد؟

چون زمان اجرای کار اول در میانگین بیشترین ضریب را دارد پس این کار اول باید کمترین زمان اجرا را داشته باشد، بنابراین کارها باید به ترتیب صعودی زمان اجرای خود انجام شوند.

انتخاب حریصانه: کارها به ترتیب صعودی زمان اجرا انجام شوند.

آیا بهینه است؟ بله بهینه است و اثبات می‌شود.

(مثل صف بدون دندانه توی فوتبالیها :))

* زمان الگوریتم: اول sort می‌کنیم $O(n\log n)$

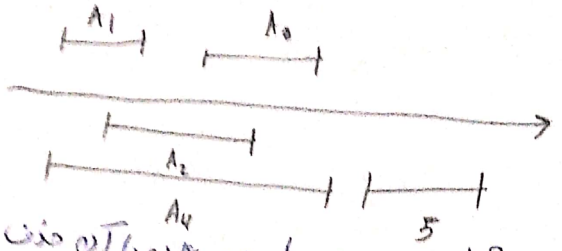
بعد پردازش می‌کنیم $O(n)$

$$O(n)+O(n\log n)=O(n.\log n)$$

⊗ مسأله‌ی فعالیت (Activity selection)

n تا فعالیت داریم: $A_i = (s_i , f_i)$

s_i: زمان شروع ، f_i: زمان خاتمه



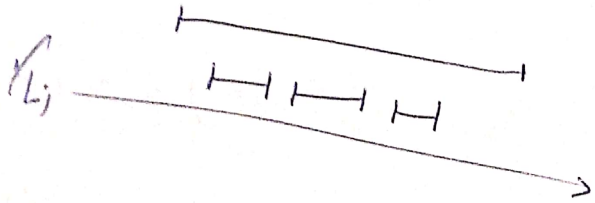
هدف انتخاب بیشترین فعالیت به طوری که زمان آنها با هم همپوشانی نداشته باشند.

✓ انتخاب حریصانه ① فعالیتی انتخاب شود که کمترین f_i را دارد و فعالیت‌های همپوشان با آن حذف می‌شوند.

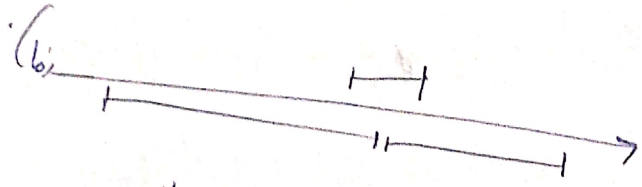
✓ انتخاب حریصانه ② فعالیتی انتخاب شود که کمترین زمان را دارد.

✓ انتخاب حریصانه ③ فعالیتی که s_i کمتر را دارد.

* مثال نقض ③:

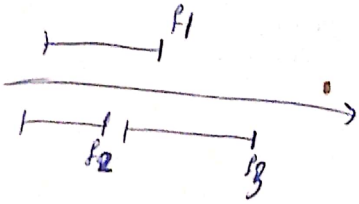


* مثال نقض ②:



* بررسی حریصانه‌ی ①: اولی که f_i کمتری دارد، از بهترین باشد حالت نمی‌شود

* آیا بهینه است؟ بله اثبات می‌کنیم که



۹۷۱

۹۴

* الگوریتم حریصانه‌ی بهینه: فعالیت ها را براساس زمان f_i آنها مرتب می‌کنیم. فعالیت که کمترین f_i را دارد انتخاب می‌شود و فعالیت هایی که با آن overlap دارند حذف می‌شوند و در لیست مرتب شده فعالیت بعدی انتخاب می‌شود.

زمان الگوریتم:

مرتب سازی $O(n.\lg n)$

بعد از مرتب سازی، برای هر فعالیت داخل لیست مرتب شده به ترتیب مرتب شده پردازش می‌شود، برای هر فعالیت باید زمان s آن را با آخرین f فعالیت های انتخاب شده مقایسه کنیم، اگر $s \geq f$ بود فعالیت انتخاب می‌شود وگرنه حذف می‌شود. که $O(1)$ زمان می‌برد و برای کل فعالیت ها $O(n)$.

$$\text{زمان کل} = O(n.\lg n) + O(n) = O(n.\lg n)$$

* حریصانه‌ی فعالیتی که کمترین overlap را دارد ← هم بهینه است.

مسأله‌ی کوله پشتی (Knap sack)

یک کوله با گنجایش M کیلو داریم.

n تا شیء داریم که هر شیء O_i دارای یک وزن w_i و ارزش V_i است.

می‌خواهیم کوله را با اشیاء با ارزش طوری پر کنیم که مجموع ارزش اشیاء داخل کوله بیشینه شود.

جمع وزن اشیاء داخل کوله نباید از M بیشتر شود:

$$\sum_{O_i \text{ های داخل کوله}} W_i \leq M$$

هدف: $\max \sum_{\text{اشیاء داخل کوله}} V_i$

(*) این مساله ۲ نسخه دارد:

(۱) نسخه‌ی ۱-۰ : هر شیء را یا باید تماماً در کوله قرار دهیم یا انتخاب نشود.

← مساله‌ی سخت است و تا به حال الگوریتم کم‌تر از نمایی داده نشده است.

(۲) نسخه‌ی کسری: می‌توان از یک شیء کسری از آن را داخل کوله پشتی گذاشت.

← مساله‌ی حریصانه است و آسان

حل نسخه‌ی کسری:

انتخاب حریصانه: اشیاء که $\frac{V_i}{W_i}$ بیشتری دارند اولویت دارند.

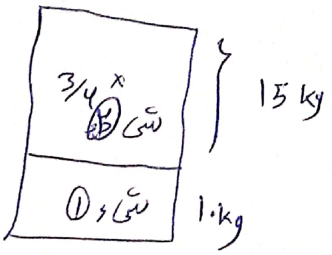
این انتخاب حریصانه بهینه است.

زمان: زمان مرتب سازی $O(n.\log n)$

مثال:

شیء	وزن	ارزش	ارزش/وزن
①	10	60	6
②	20	100	5
③	30	120	4

$M=25$



ارزش حاصل $= 60 + \frac{3}{4} \times 100 = 135$

مسأله‌ی زمان‌بندی کارها به همراه جریمه‌ی تأخیر:

n تا کار داریم که هر کاری یک سررسید d_i دارد که اگر بعد از آن انجام شود یک جریمه‌ی P_i دارد.

هدف: زمان‌بندی کارها به طوری که کمترین جریمه تعلق گیرد.

$T_i = (d_i , P_i)$

زمان سررسید — جریمه

نکته: طول هر کاری ۱ است.

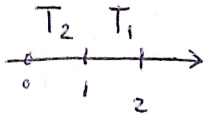
انتخاب های حریصانه:

۱) به ترتیب نزولی جریمه انجام شود:

T	d	P
T_1	2	0
T_2	1	5

مثال نقض

جواب بهینه اصلی



جریمه = 0

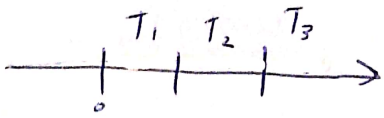
ولی با روش ۱ جریمه = 5

۲) به ترتیب صعودی سررسیدها انجام شود:

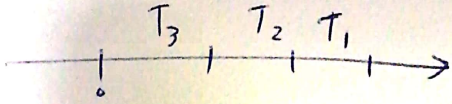
T	d	P
T_1	1	5
T_2	2	10
T_3	2	10

مثال نقض

با روش ۲ جریمه = 10



جریمه = 5



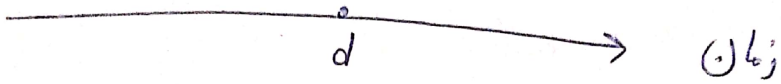
که جواب بهینه است

۳) به ترتیب نزولی نسبت جریمه به سررسید.

این هم مثال نقض دارد.

* انتخاب جریمه‌ای بهینه:

اولویت زمان‌بندی با کارهایی است که جریمه سنگین‌تر دارند.



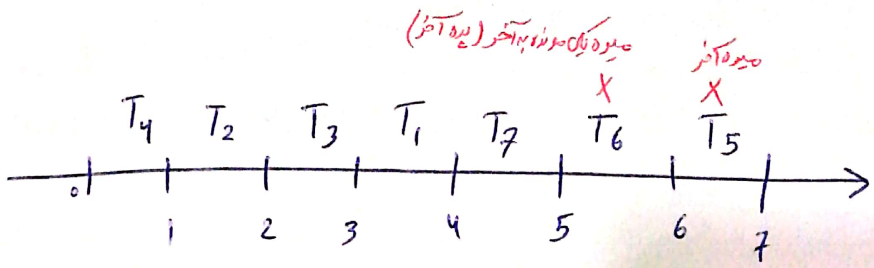
* هر کاری دقیقاً قبل از سررسید خودش بهتر است که زمان‌بندی شود. اگر یک کاری زمان قبل‌تر از سررسیدش پر شده باشد به عقب‌تر حرکت می‌کنیم و اولین جای خالی که پیدا شود زمان‌بندی می‌شود. اگر هیچ جای خالی قبل از سررسیدش پیدا نشود به آن کار جریمه تعلق می‌گیرد.

* کاری که به آن جریمه تعلق می‌گیرد کجا زمان‌بندی شود؟ چرا؟

بهترین جا برای این کار آخرین جا است. چون هرچه زودتر ببندش فقط جای بقیه را پر می‌کند که ممکنه اون کار اونجا قبل سررسیدش باشد.

مثال:

کار	1	2	3	4	5	6	7
سررسید d	4	2	4	3	1	4	6
جریمه P	70	60	50	40	30	20	10



جریمه = 30 + 20 = 50

تحلیل زمانی:

مرتب سازی $O(n.\log n)$ + پیدا کردن محل خالی قبل از سرریز $O(n^2)$

$$T(n) = O(n^2)$$

* مسأله‌ی کدگذاری هافمن: (کدگذاری حریصانه و متغییر)

کدگذاری ASCII
8 بیت = یک بایت = یک حرف
تعداد ⑧ بیت یکسان برای هر حرف

800 بیت = 100 حرف

← اگر تکرار حروف در یک متن، متفاوت باشد.

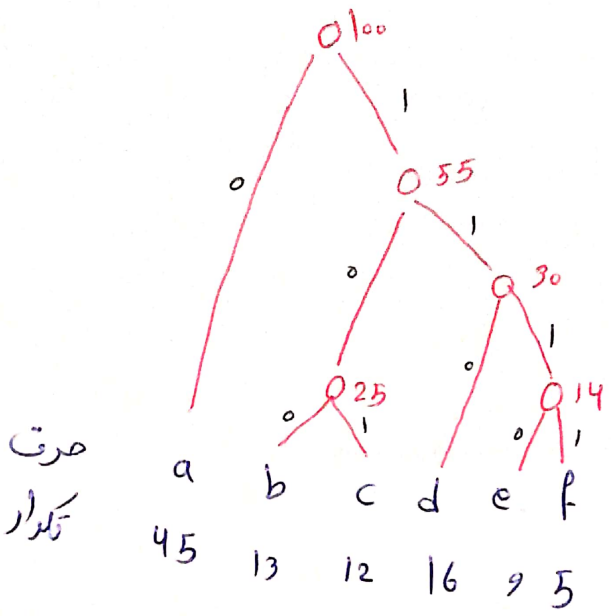
مثلاً از A تا 20
از B تا 30
از C تا 10
⋮

⇐ حرفی که تکرار بیشتر دارد کد کوتاهتری داشته باشد.

↓ انتخاب حریصانه

↓ ایده اصلی کدگذاری هافمن

⊛ کدگذاری هافمن:



* دو تا کوچکترین می‌گیریم می‌چسبونیم از برگ‌ها درخت رو می‌سازیم ← به بالا سمت چپ ⓪ می‌دهیم به بالا سمت راست ①
می‌دهیم و برچسب می‌زنیم.

(درخت کدگذاری هافمن)

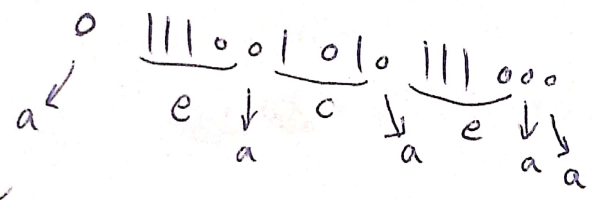
در درخت هافمن از ریشه تا حرف مورد نظر برچسب یال هایی که مشاهده می کنیم می شود کد مرتفر برای آن حرف :

حرف	کد	تکرار
a	0	45
b	100	13
c	101	12
d	110	16
e	1110	9
f	1111	5

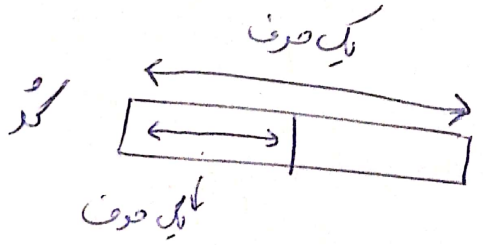
چون هر حرفی که کمترین تکرار رو داره اول انتخاب میشه در عمق بیشتری است و کد طولانی تری دارد.

مهم: روش هافمن بهینه است یعنی فایل گذاری شده کمترین تعداد بیت کد را دارد.

بازگشایی کد هافمن :



از اول از ریشه شروع می کنیم و هادو به یه حرفی رسوندیم دوباره از ریشه شروع می کنیم و...



توجه: این کدگذاری ابهام ندارد.

* خاصیت پیشوندی ندارد (کد یک حرف پیشوند کد یک حرف دیگر نیست)

* چون در کدگذاری هافمن همه ی کد حروف در برگ ها هستند پس خاصیت پیشوندی و ابهام در بازگشایی کد وجود ندارد.

روش جستجوی حالات :

بررسی همه‌ی حالات برای حل یک مساله (در صورتی که با روش های قبلی نتوان مساله را حل کرد)

مثال یک : (مساله‌ی ۸ وزیر)

۸ وزیر در صفحه شطرنج باید قرار بدهیم به طوری که همدیگر را تهدید نکنند.

* جستجوی منظم فضای حالات (back tracking)

۶۴ خانه داریم ۸ خانه برای ۸ وزیر انتخاب کنیم

کل حالات $\binom{64}{8}$

اما با کمی تامل می‌توان فهمید که می‌توان تعداد حالات را کاهش داد چون دو وزیر نمی‌توانند در یک سطر قرار بگیرند و هر سطر را به یک وزیر نسبت می‌دهیم.

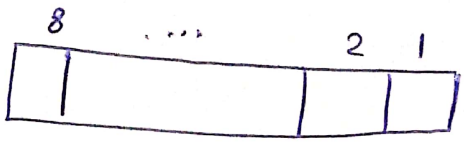
وزیر اول ۸ انتخاب (سطر اول) 8

وزیر دوم ۸ انتخاب (سطر دوم) 8

⋮

وزیر ۸ام ۸ خانه‌ی سطر ۸ام. 8

کل حالات : 8^8



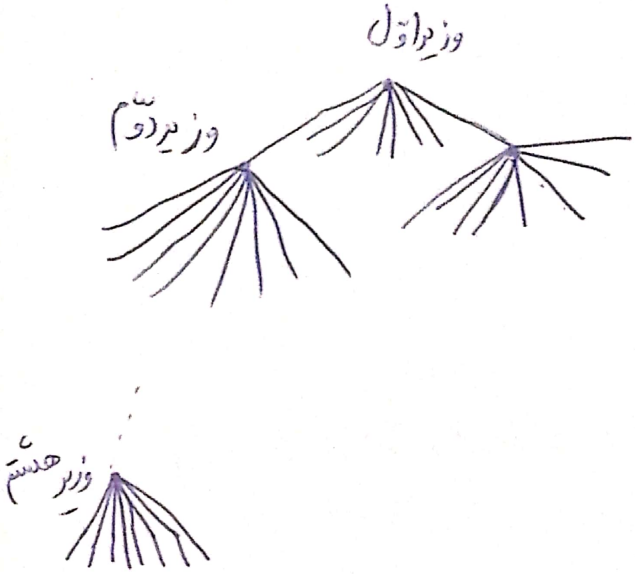
شماره‌ی خانه

۸ رقم برای ۸ وزیر

هر خانه شماره‌ی از ۰ تا ۷ بشمارد (مبنای ۸)

* این شمارش را می‌توان روی درخت فضای حالات به صورت پیمایش DFS نشان داد.

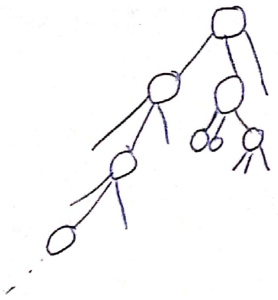
⊛ یک درخت 8 تایی با 8 سطح



✓ بهبودهای بیشتری می‌توان داد دوباره!

حل به روش شاخه و حد

روش Branch & Bound برای هرس کردن درخت و محدود کردن فضای حالات و سرعت بخشیدن جستجوی جواب:

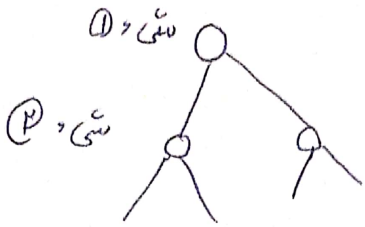


مسأله‌ی کوله پشتی

هر شیء یا در کوله قرار می‌گیرد یا قرار نمی‌گیرد پس ۲ حالت وجود دارد.

n شیء داریم پس 2^n حالت داریم،

درخت فضای حالت



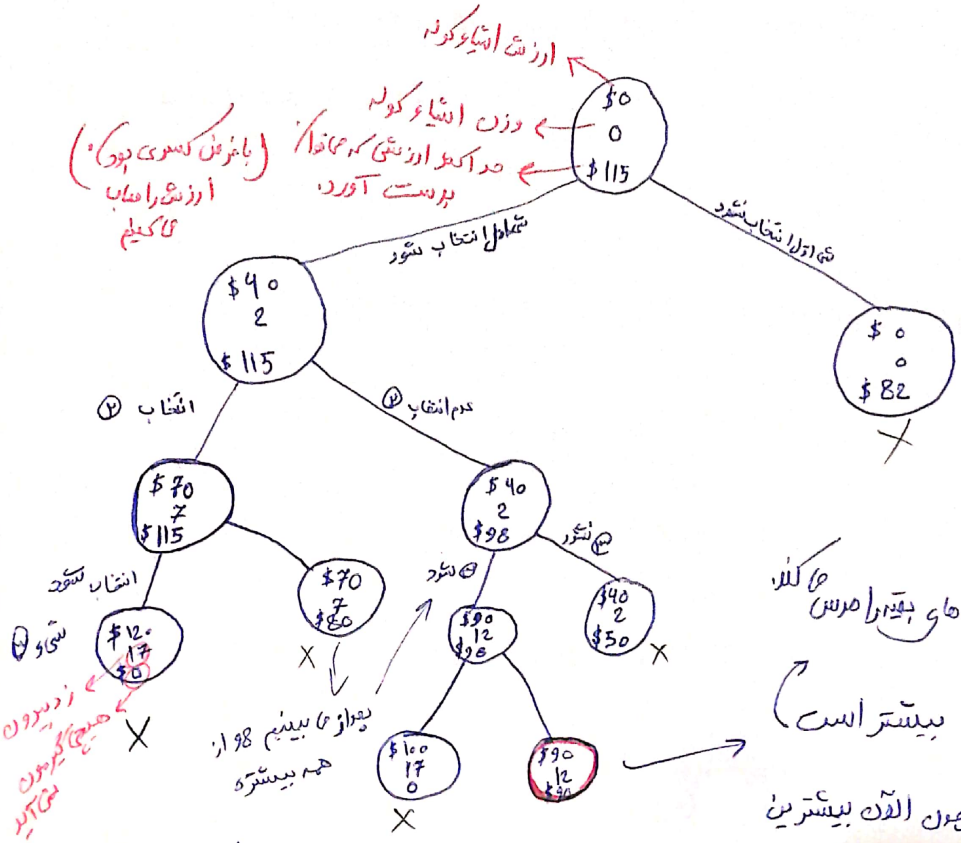
مثال: فرض کنیم اشیاء و ارزش‌های مقابل را داریم:

i	V_i	W_i	V_i/W_i
۱	۴۰ $	۲	۲۰ $
۲	۳۰ $	۵	۶ $
۳	۵۰ $	۱۰	۵ $
۴	۱۰ $	۵	۲ $

ظرفیت کوله $\boxed{W_{\text{کوله}} = 16}$

* جواب کوله پشتی کسری یک حد بالا برای جواب کوله پشتی صفر-یک است، چون هر حجم باقی مانده را با بیشترین ارزش پر می‌کند.

$$\text{جواب کسری} = 40 + 30 + \frac{9}{10} \times 50 = 115\ \$$$



روش مشاهده وحد:

در هر مرحله برای ما ارزش را در انتخاب می‌کنیم.

[تغییر دادن order تاثیر نداره ولی زمانش کمتر می‌کند]

همه‌ی گره‌های بقیه را هرس می‌کند.

تقدم این گره از امید بقیه گره‌ها بیشتر است

جواب این است چون الان بیشترین

(یا همان 90 دلار برای این گره است و اشیا هم تمام شد پس این بهتر از همه‌ی شاخه‌هاست.)

⊗ مبحث: پیچیدگی محاسبات:

مسأله‌ها از نظر زمانی که برای حل احتیاج دارند یکسان نیستند.

مسأله‌ی sort در $O(n.\log n)$ حل می‌شود!

مسأله‌ی برج هانوی به زمان نمایی نیاز دارد!!

مسأله‌ی که لااقل تا به حال در زمان چندجمله‌ای حل نشده است

tractable = * مسایلی که در زمان چندجمله‌ای یعنی $O(n^c)$ حل می‌شوند

↳ مثل sort

Intractable = * مسایلی که به زمان فوق چندجمله‌ای برای حل نیاز دارند

↳ مثل برج هانوی

⊗ مسایلی هستند که بلا تکلیف هستند و معلوم نیست که tractable هستند یا نه مثل مسأله‌ی کوله‌پشتی

⊗ برخی مسایل هم اصلاً با کامپیوتر حل نمی‌شوند، هر چقدر هم که زمان داشته باشیم

مثل مسأله‌ی توقف تورینگ

$$\boxed{\text{کلاس } P = \text{tractable}}$$

Polynomial solvable Problems

مسایلی که در زمان چندجمله‌ای حل می‌شوند

کلاس NP : مسایلی که روی ماشین غیرقطعی در زمان چندجمله‌ای حل می‌شوند.

تعریف دوم کلاس NP : مسایلی که در زمان چندجمله‌ای Verify می‌شوند.

Verify : با داشتن یک جواب بالقوه بررسی کنیم که آیا این جواب بالقوه جواب درستی بوده یا نه؟

Certificate = جواب بالقوه = گواه

مثال: مساله کوله‌پشتی ۱-۰ :

n شیء و گنجایش M

آیا یک انتخاب از اشیاء وجود دارد که از گنجایش بیشتر نباشد و جمع ارزش آنها بیشتر یا مساوی k باشد؟

* جواب بالقوه این مساله : یک مجموعه از اشیاء انتخاب شده!

چطور Verify کنیم؟

$$\sum W_i \leq M \rightarrow O(n)$$

$$\sum V_i \geq k \rightarrow O(n)$$

* بنابراین مساله‌ی کوله‌پشتی در زمان $O(n)$ Verify می‌شود.

پس مساله‌ی کوله‌پشتی ۱-۰ $\in NP$

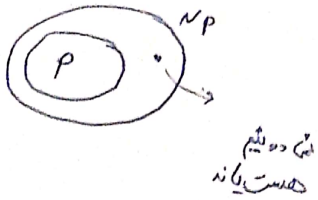
* هر مساله‌ای که در زمان چندجمله‌ای حل شود در زمان چند جمله‌ای هم Verify می‌شود. پس:

$$P \subseteq NP$$

* مساله‌ی باز

$P = NP$ ؟

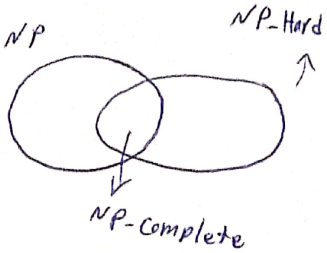
از سال ۱۹۷۱ باز مونده



$NP\text{-}Complete = NP \cap NP\text{-}hard$ (غیر رسمی)

مسایلی سخت‌تر از NP-complete توی مسایل NP. نیست و با حدس دیگه مساله‌ی NP نمی‌مانند که در چندجمله‌ای حل نشود.

NP-Complete
NP-Hard



تعریف غیررسمی NP-hard: مسایلی که حداقل به سختی مسایل NP هستند.

⊛ تعریف رسمی NP-Hard:

$A \in NP\text{-}Hard$ اگر بتوان همه‌ی مسایل NP را در زمان چندجمله‌ای به مساله A تبدیل کرد به طوری که از جواب A در زمان چندجمله‌ای جواب مسایل NP محاسبه شود.

$\forall \Theta \in NP$:

بتوان $\Theta \propto_p A$

تبدیل در زمان چندجمله‌ای

$\Rightarrow A \in NP\text{-}Hard$

آن هایی که خارج از NP اند اصلاً در زمان چندجمله‌ای حل نمی‌شوند

تمرکزمون رو میگذاریم روی NP-Complete ← اگر حل بشه همه مسایل NP می‌شوند (زمان چندجمله‌ای)

قضیه‌ای داریم که می‌گوید هر مساله‌ی NP را می‌توان در زمان نمایی حل کرد.

مساله‌ی توقف ترادینگ کلاً بیرون NP و توی NP-Hard است.