مرجع ⇐ CLRS (3rd edition)

fundamentals of computer Algorithm (horowitz)

سرفصل ها :

* مقدمه و نقش الگوریتم‌ها

* مرور رشد توابع

* حل روابط بازگشتی
  - درخت بازگشت
  - قضیه اصلی

* مرتب‌سازی Heap sort ، Quick sort ، اثبات درستی، تحلیل زمانی

* مرتب‌سازی زمان خطی
  - counting sort
  - radix sort
  - Bucket sort

* پیدا کردن میانه و k امین عدد در بین n عدد . مسئله selection

* تحلیل سرشکن Amortized Analysis

تحلیل

* الگوریتم‌های مقایسه‌ای بهتر از nlogn نمی‌شه

* روش تقسیم و حل (divided & conquere)

مساله ضرب ماتریس‌ها (روش استراسن)

* برنامه ریزی پویا dynamic programming

Longest common subsequence

مثلث بندی بهینه چندضلعی

* روش حریصانه (greedy)

کدگذاری هافمن

کوله پشتی fractional

* روش جست و جوی فضای حالات

روش عقب گرد (back tracking)

↳ DFS

روش شاخه و حد (branch & bound)

↳ درخت فضای حالات رو یک جوری شاخه هاییش را حذف کنیم.

* مرور الگوریتم‌های گراف

- پیمایش گراف ← BFS / DFS
- کوتاه‌ترین مسیر ← dijkstra / floyd warshal
- درخت پوشای کمینه ← prim / krushal

* الگوریتم جریان بیشینه (maximum flow)

* نظریه پیچیدگی محاسبات
  - کلاس P
  - کلاس NP
  - کلاس NP-hard
  - کلاس NP-complete

بارم‌بندی:

میان‌ترم ← 7 ، تمرین ← 3 ، پایان‌ترم ← 10

جلسه دوم - 1401,11,24

مقدمه: جایگاه الگوریتم

الگوریتم ← دنباله ای از دستورات که با ترتیب مشخصی باید اجرا شود و مسئله را حل کند.

مسئله ← یک زوج مرتب است که تشکیل شده است از ورودی مسئله و توصیف خروجی مسئله.

مثال: مسئله: مرتب سازی   ورودی: یک مجموعه تشکیل شده از n عدد.

خروجی: دنباله ای از همین n عدد ورودی که $a_i \leq a_{i+1}$

* برای برخی از مسائل دنیای واقعی الگوریتم نداریم و باید خودمان طراحی کنیم.

بررسی اهمیت طراحی الگوریتم خوب:

فرض کنید که یک مسئله مرتب سازی است که تعداد عددها، 10^6 است.

دو نوع کامپیوتر داریم که یکی ضعیف تر از دیگری است.

گروه A :
توانایی کامپیوتر = 10^9 ins/sec
زمان الگوریتم = n^2

گروه B :
توانایی کامپیوتر = 10^7 ins/sec
زمان الگوریتم = $n\log n$

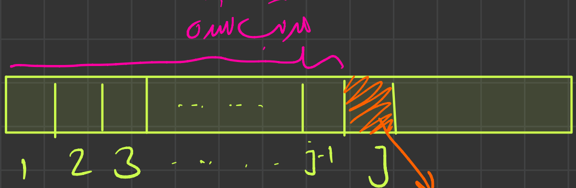
$$\text{زمان حل مسئله} = \frac{\text{زمان اجرای الگوریتم}}{\text{سرعت اجرای کامپیوتر}}$$

$$\text{زمان حل مسئله } A = \frac{(10^6)^2}{10^9} = 10^3 \text{ sec}$$

$$\text{زمان حل مسئله } B = \frac{10^6 \log 10^6}{10^7} \simeq 2 \text{ sec}$$

تحلیل زمانی و اثبات درستی الگوریتم insertion sort :

روش incremental است . در روش incremental فرض می‌کنیم دونه دونه ورودی به ما داده شده .



Key = مقدار اولیه خانه j

این خانه را باید از چپ به راست دونه دونه با خانه‌ها مقایسه شود و جایی که درجای درست قرار گیرد.

```
Insertion_Sort(A, n):        // A: آرایه , n: تعداد
    for (j=2 to n) do:
        key = A[j]
        i = j-1
        while (i>0 and key < A[i]):
            A[i+1] = A[i]
            i = i-1
        A[i+1] = key
```

Loop_invarient ← گزاره ای است که در طول اجرای حلقه الگوریتم درست است و از درستی آن درستی الگوریتم را می توان نتیجه گرفت.

ثابت حلقه

ویژگی مسئله insertion sort ← در مرحله jام ، خانه های (1 تا j-1) مرتب شده محفوظ اند.

ثابت حلقه : در ابتدای دور jام حلقه for الگوریتم insertion sort ، زیر آرایه $A[1,\ldots,j-1]$ شامل همان عناصر اولیه ورودی هستند که در زیر آرایه $A[1,\ldots,j-1]$ بوده اند ولی به صورت مرتب شده محفوظ.

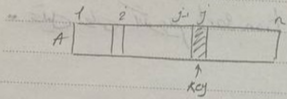
حالت پایه: (قبل از شروع حلقه for) $j=2$

← ثابت حلقه را برای $j=2$ بازنویسی می‌کنیم: در ابتدای گام ۲ ام حلقه for الگوریتم insertion sort زیرآرایه $A[1,\dots,1]$ شامل همان عناصر اولیه ورودی هستند که در زیرآرایه $A[1,\dots,1]$ بوده‌اند ولی به صورت مرتب شده صعودی

← این پایه درست است

فرض می‌کنیم گام j درست

- تحلیل حالت متوسط Insertion sort



بدترین حالت ← $j-1 \geq t_j \geq 0$ بهترین حالت

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

t_j: تعداد تکرار حلقه while

* یادآوری: متغیر تصادفی X و $\bar{X} = E[X] = \sum_{x} x \cdot P(X = x)$

امید ریاضی

$$E[t_j] = \sum_{k=0}^{j-1} k \cdot P(t_j = k)$$

- متغیر تصادفی شاخص (Indicator random variable)

$$I\{A\} = \begin{cases} 1 & \text{اگر رخداد } A \text{ رخ دهد} \\ 0 & \text{وگرنه} \end{cases}$$

$$X_i = \begin{cases} 1 & \text{اگر کلید از } A[i] \text{ کمتر باشد} \\ 0 & \text{وگرنه} \end{cases} \Rightarrow t_j = \sum_{i=1}^{j-1} X_i$$

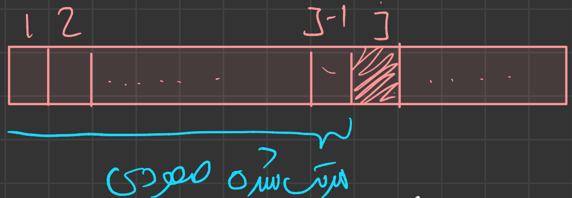
$$\Rightarrow E[t_j] = \sum_{i=1}^{j-1} E[X_i] \quad , \quad E[X_i] = P(\text{کلید از } A[i] \text{ کوچکتر باشد}) = \frac{1}{2}$$

$$E[t_j] = \frac{j-1}{2}$$

جلسه سوم :

ادامه insertion sort :

حالت نگهدار : فرض می‌کنیم برای j ثابت حلقه برقرار است و برای j+1 اثبات می‌کنیم



چون خانه‌های آرایه که در بازه 1 تا j هستند حداکثر یک خانه به سمت راست شیفت می‌دهیم، پس همچنان در بازه 1 تا j آرایه باقی می‌ماند.

key تا وقتی که از خانه‌های قبل از خود کوچک‌تر است در آرایه به سمت ابتدای آرایه حرکت می‌کند و نهایتاً در جایی از آرایه درج می‌شود که از خانه‌های بعد از خود کوچک‌تر و از خانه‌های قبل از خود بزرگ‌تر است.

شیفت دادن خانه‌های آرایه 1 تا j-1 ترتیب صعودی را عوض نمی‌کند.

بنابراین در ابتدای گام j+1 ام حلقه for عناصر زیرآرایه A[1,...,j] همان عناصر اولیه ورودی زیرآرایه A[1,...,j] است ولی به صورت مرتب شده صعودی پس با استقرا ثابت حلقه اثبات شد.

وقتی الگوریتم insertion sort تمام می‌شود (در انتهای حلقه for)

ثابت حلقه عبارت خواهد بود از : $(j = n+1)$

در ابتدای گام $j = n+1$ ام حلقه for زیرآرایه $A[1, \dots, n]$ شامل همان

عناصر اولیه زیر آرایه $A = [1, \dots, n]$ است ولی به صورت مرتب‌شده صعودی

تحلیل زمانی الگوریتم مرتب‌سازی insertion sort :

* The RAM model ⟶ تک پردازنده است. دستورات ماشین شامل

جمع و تفریق ، ضرب و تقسیم ، مقایسه و ذخیره و بازیابی و شیفت دادن .

از دستورات پایه‌ای ما هستند

تابع زمانی اجرای الگوریتم‌ها چه‌طور است ؟

ورودی insertion sort ⟵ مجموعه‌ای از n عدد .

⟵ زمان الگوریتم را برحسب n یعنی تعداد اعداد بیان می‌کنیم $T(n) = n^2$

الگوریتمی که 2 عدد n بیتی را باهم جمع می‌کند . زمان الگوریتم را برحسب n

که تعداد بیت‌های این 2 عدد است ، بیان می‌کنیم .

طول ورودی ⟵ size یا اندازه ورودی مسئله .

آرایه — تعداد

Insertion_Sort(A, n):

for(j=2 to n) do: → $\begin{cases} j=2 \text{ or } j+=1 \\ j<n \end{cases}$ → دستورات پایه ; $n \cdot c_1$

key = A[j] → $(n-1)c_2$

i = j-1 → $(n-1).c_3$

while(i>0 and key < A[i]): $\left(\sum_{j=2}^{n} (t_j+1)\right)c_4$

تعداد تکرار بدنه حلقه while

A[i+1] = A[i] → $\left(\sum_{j=2}^{n} t_j\right).c_5$

i = i-1 → $\left(\sum_{j=2}^{n} t_j\right)c_6$

A[i+1] = key → $(n-1).c_7$

این اعداد مختلف می‌تواند باشد.

اگر خانه j از خانه j-1 بزرگتر باشد $t_j=0$ و اگر خانه j از همه اعداد قبل خودش کوچکتر باشد، $t_j=j-1$

بهترین حالت: $t_j=0$ → زمان الگوریتم: $T(n)=a\times n+b$

بدترین حالت: $t_j=j-1$

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

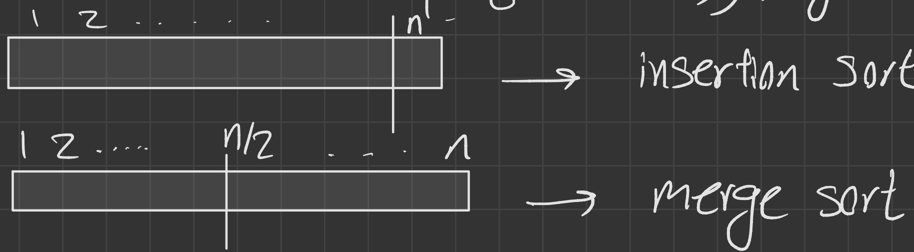
$$\rightarrow T(n)=a'n^2+b'n+c'$$

بدترین حالت تحلیل زمانی تضمین می‌کنه که الگوریتم بدتر از این نمیشه.

merge sort :

مبتنی بر روش divide and conquere است.

تفاوت merge sort با insertion sort در نحوه تقسیم بندی کردن آرایه به مسئله‌های کوچک‌تر است. در insertion sort، عنصر آخر را جدا می‌کنیم ولی در merge sort از وسط نصف می‌کنیم.



زمان اجرای الگوریتم insertion در بدترین حالت $O(n^2)$ است.

اما در merge در بدترین حالت $O(n\log n)$ خواهد بود.

شرط پایان را معمولا یک عنصر یا دو عنصر می‌گذاریم.

اندیس ابتدای آرایه

اندیس انتهای آرایه

```
merge_sort(A, i, j):
    if(j > i):
        q = ⌈(i+j)/2⌉
        merge_sort(A, i, q)
        merge_sort(A, q+1, j)
        merge(A, i, q, j)
```

جلسه چهارم:

گام های divide and conquere :

۱_ گام تقسیم ← مسئله را چجوری به زیر بخش های کوچک تر تقسیم کنیم

← ایده

2_ گام حل بازگشتی

3_ گام ترکیب

اندیس ابتدای آرایه

اندیس انتهای آرایه

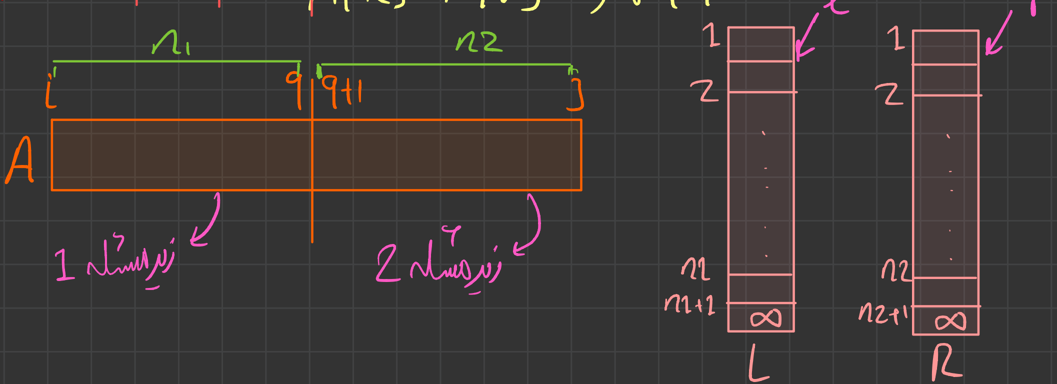
```
  merge_sort(A,i,j):
1     if(j>i):
2         q = ⌈(i+j)/2⌉
3         merge_sort(A,i,q)
4         merge_sort(A,q+1,j)
5         merge(A,i,q,j)
```

گام تقسیم (2)

گام حل بازگشتی (3، 4)

گام ترکیب (5)

```
Merge(A,i,q,j):
    n1 = q-i+1  ,  n2 = j-q
    L[n1+1] = ∞  ,  R[n2+1] = ∞
    for (l = 1 to n1):
        L[l] = A[i+l-1]
    for (r = 1 to n2):
        R[r] = A[q+r]
    l = r = 1
    for (k = i to j):
        if (L[l] < R[r]):
            A[k] = L[l]  ,  l++
        else:
            A[k] = R[r]  ,  r++
```



اثبات درستی الگوریتم merge :

فرض: آرایه ورودی A از اندیس i تا q ، و از $q+1$ تا j مرتب شده صعودی است.

حکم: در انتهای الگوریتم merge آرایه A از اندیس i تا j مرتب شده صعودی است.

ثابت حلقه: در ابتدای گام k ام حلقه for در 8 تا 10 الگوریتم زیرآرایه $A[i, \dots, k-1]$ شامل $k-i$ کوچکترین عناصر آرایه $L[1, \dots, n_1+1]$, $R[1, \dots, n_2+1]$ است به صورت مرتب شده صعودی

همچنین $\boxed{R[r] , L[l]}$ کوچکترین عناصر آرایه R,L هستند که در A کپی نشده اند.

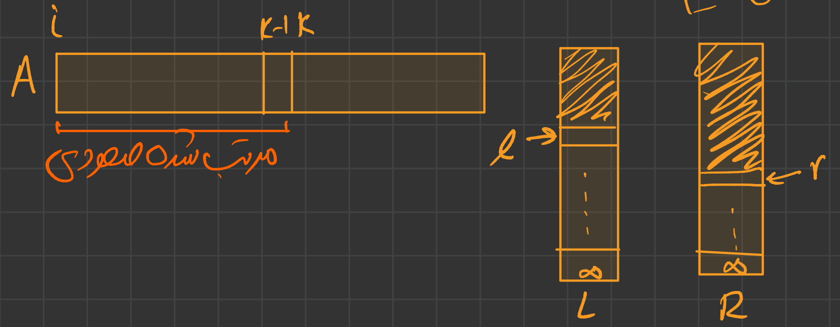
درنهایت همان بی‌نهایت‌ها هستند که کپی نشده اند.

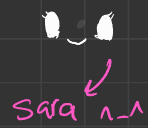
پایه استقرا: $(k=i) \xrightarrow{k=i} l=r=1$

در ابتدای گام k ام حلقه for در 8 تا 10 الگوریتم زیرآرایه $A[i, \dots, i-1]$ شامل صفر کوچکترین عناصر آرایه $L[1, \dots, n_1+1]$, $R[1, \dots, n_2+1]$ است به صورت مرتب شده صعودی

همچنین $R[1] , L[1]$ کوچکترین عناصر آرایه R,L هستند که در A کپی نشده اند.

اثبات گام گذار: فرض می‌کنیم ثابت حلقه برای K برقرار است. و برای $K+1$ اثبات می‌کنیم





اثبات: طبق فرض $L[l]$, $R[r]$ کوچک‌ترینِ عناصر L و R هستند که هنوز کپی نشده اند پس آن min این دو مقدار در $A[k]$ کپی شود عناصر $A[i, \dots, k]$ همچنان کوچک‌ترینِ عناصر R و L هستند که در A کپی شده است و چون قبل از این کپی کردن، آرایهِ $A[i, \dots, k-1]$ شامل کوچک‌ترینِ عناصر L و R هستند و مرتب شده صعودی است، با اینِ کپی همچنان مرتب شده صعودی خواهند بود.

پس از کپی min $R[r]$, $L[l]$ در $A[k]$ اندیس مربوط ++ می‌شود

پس همچنان $R[r]$ ، $L[l]$ مینیمم L و R هستند که هنوز در A کپی نشده اند.

اثبات درستی merge-sort :

پایه استقرا : $(i=j) \leftarrow n = i - \cancel{j} + 1 = 1$

برای حالت پایه شرط $(i<j)$ if برقرار نیست و از الگوریتم خارج می‌شویم و آرایه ورودی دست نخورده باقی می‌ماند که چون فقط یک عنصر دارد. مرتب شده صعودی است. پس پایه برقرار است.

فرض استقرا : merge-sort هر آرایه با کمتر از n عنصر را درست مرتب می‌کند

اثبات ←

تحلیل زمانی merge , merge-sort :

```
Merge(A,i,q,j):
    n1 = q-i+1 , n2 = j-q              → c1  هزینه ثابت
    L[n1+1] = ∞ , R[n2+1] = ∞          → c2  هزینه ثابت
    for (l = 1 to n1):
        L[l] = A[i+l-1]                → c3 n1
    for (r = 1 to n2):
        R[r] = A[q+r]                  → c4 n2
    l = r = 1                          → c5
    for (k = i to j):
        if (L[l] < R[r]):
            A[k] = L[l] , l++
        else:                          → c6 . n
            A[k] = R[r] , r++
```

کل هزینه merge $\leq c_1 + c_2 + \max\{c_3, c_4\}(n_1+n_2) + c_6 n$

$\Rightarrow$ کل هزینه مرج $\geq a.n + b \longrightarrow \boxed{\Theta(n)}$

هم در بدترین حالت و هم در بهترین حالت از مرتبه n است.

جلسه پنجم شنبه : (1401,12,8)

Subject:
Year. ۱۴۰۱ Month. ۱۲ Date. ۸

طراحی الگوریتم

• جلسه پنجم ۱۴۰۱/۱۲/۸:

ادامه تحلیل Merge sort:

Merge_sort(A, i, j)

```
if i<j then                     c1
    q = ⌊(i+j)/2⌋               c2
    Merge-sort(A, i, q)         T(⌈n/2⌉)
    Merge-sort(A, q+1, j)       T(⌊n/2⌋)
    Merge(A, i, q, j)           ∈ Θ(n)
```

⇒ تابع زمان: $T(n) = C_1 + C_2 + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$

$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$ } دارای رشد یکسان

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

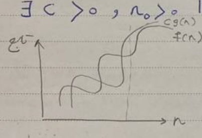
• مرور رشد توابع: Θ, Ω, O, o, ω

n^2 بدترین حالت — $T(n)$ — n بهترین حالت

یک مثال:

$T(n) \in \Omega(n)$
$T(n) \in O(n)$ } → $T(n)$ بهترین حالت $\in \Theta(n)$ ، $T(n)$ بدترین حالت $\in \Theta(n^2)$

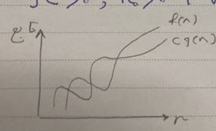
$\exists c > 0, n_0 > 0 \mid \forall n \geq n_0 \quad f(n) \leq c\,g(n) \iff f(n) \in O(g(n))$ —



که رشد تابع g(n) بیشتر یا مساوی رشد تابع f(n)

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \begin{cases} \infty & \text{بیشتر} \\ \underset{\text{ثابت}}{c} \neq 0 & \text{مساوی} \end{cases}$$

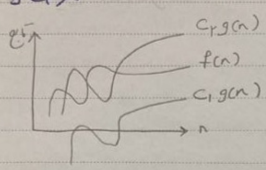
$\exists c > 0, n_0 > 0 \mid \forall n \geq n_0 \quad f(n) \geq c\,g(n) \iff f(n) \in \Omega(g(n))$ —



که رشد تابع g(n) کمتر یا مساوی رشد تابع f(n)

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \begin{cases} 0 & \text{کمتر} \\ \underset{\text{ثابت}}{c} \neq 0 & \text{مساوی} \end{cases}$$

$\exists c_1 > 0, n_0 > 0 \mid \forall n \geq n_0 \quad f(n) \geq c_1 g(n) \iff f(n) \in \Theta(g(n))$ —
$\exists c_2 > 0 \quad f(n) \leq c_2 g(n)$



که رشد تابع g(n) و f(n) یکسان باشد

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

جلسه ششم: ۱۴۰۱/۱۲/۱۳

ادامه رشد توابع:

Little o:

$$f(n) \in o(g(n)) \iff \forall c>0, \exists n_0>0 \mid \forall n \geq n_0, f(n) < c.g(n)$$

رشد تابع $g(n)$ از رشد تابع $f(n)$ بیشتر است

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

$$\text{if } f(n) = 5n^2+n \in \boxed{O(n^2)} \Rightarrow f(n) \in O(n^3)$$

حد بالای $f(n)$

$$\in \boxed{\Omega(n)} \Rightarrow f(n) \in \Omega(\log n)$$

حد پایین

Little omega:

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

* $f(n) \in \Theta(f(n)) + o(f(n))$ ^^

$$\begin{cases} f(n) = n^{1+\sin n} \\ g(n) = n \end{cases}$$

* می‌تونه رشد توابع نسبت به هم ثابت نباشه ⇐

حل روابط بازگشتی :

روش ها:

- درخت بازگشتی (Recursion tree)
- قضیه اصلی (The master thearm)

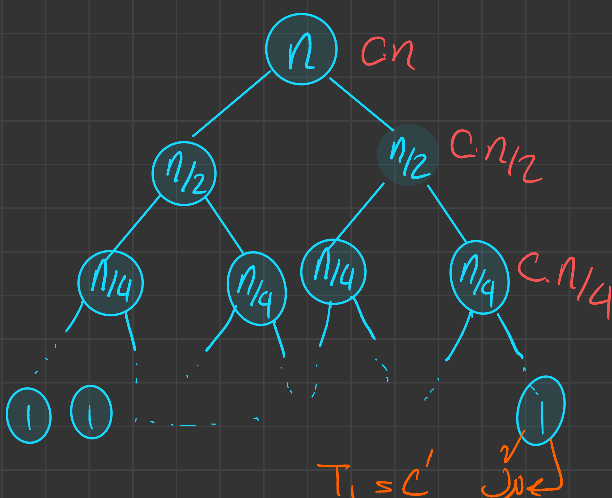
در مسئله divide & conquer ۳ گام داریم :

1. گام تقسیم
2. حل بازگشتی $2T(n/2)$
3. گام ترکیب

$$T(n) = \boxed{2T(n/2)} + \underbrace{\Theta(n)}_{\text{هزینه گام تقسیم و ترکیب}}$$

$2T(n/2)$: هزینه حل بازگشتی

درخت بازگشتی :



$T(n) = \sum$ هزینه هر سطر

هزینه	سطر
$c.n$	1
$cn = c.n/2 + cn/2$	2
$cn = 4 \times cn/4$	3
⋮	⋮
cn	$\log_2^n$

در چه سطری به برگ می رسیم؟

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

$\boxed{\text{برگ در سطر } i = \log_2^n + 1 \text{ قرار دارد.}}$

هزینه گره‌های غیربرگ $= \log_2^n \times cn = c.n.\log_2^n$

هزینه گره‌های درخت $= n \times c' = c'n$

تعداد گره‌ها در سطر i ام $= 2^{i-1}$

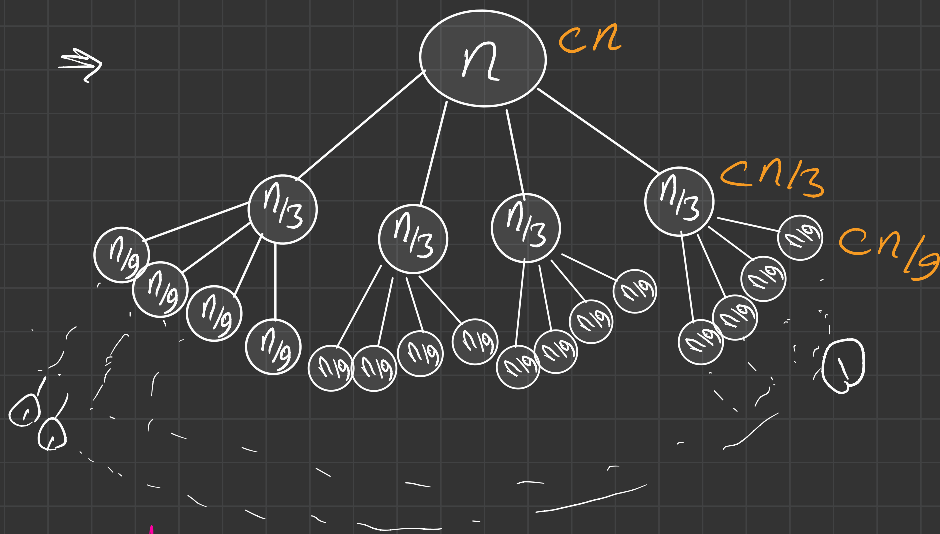
تعداد برگ ها $= 2^{((\log_2^n + 1) - 1)} = n$

$T(n) = cn\log_2^n + c'n \in \Theta(n\log n) \Leftarrow +$

Merge Sort هزینه الگوریتم

$T(n) = 4T(n/3) + \Theta(n) \quad , \quad T(1) = c'$

درخت بازگشت $\Rightarrow$



هزینه	سطر
cn	1
$4cn/3$	2
$16cn/9$	3
⋮	⋮
	$\log_3^n$
1	$\log_3^n + 1$

اندازه زیرمسئله در هر سطر $= n/3^{i-1}$

شماره سطر برگ‌ها $\Rightarrow 1 = \frac{n}{3^{i-1}} \rightarrow i = \log_3^n + 1$

هزینه گره‌های غیربرگ $= \sum_{i=1}^{\log_3^n} c.n.\left(\frac{4}{3}\right)^{i-1} =$

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

از خاصیت $a^{\log_c^b} = b^{\log_c^a}$

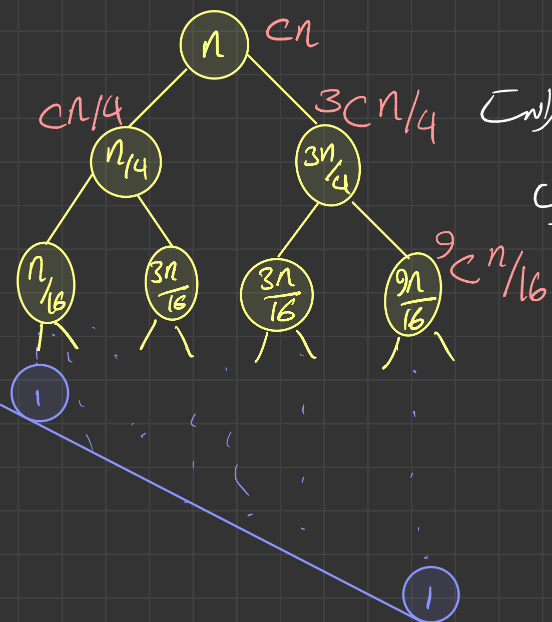
$\Rightarrow$ هزینه گره‌های غیربرگ $\in \Theta\left(n.n^{\log_3^4}\right)$

$$\text{تعداد گره ها در سطر } i \text{ ام} = 4^{i-1} \Rightarrow \text{تعداد برگ‌ها} = 4^{(\log_3^n + 1) - 1} = 4^{\log_3^n} = n^{\log_3^4}$$

$$\text{هزینه برگ ها} = \text{تعداد برگ‌ها} \times \text{هزینه هر برگ} = c' n^{\log_3^4}$$

جلسه هفتم : (1401 ,12 ,15)

حل رابطه $T(n)=T(n/4)+T(3n/4)+\Theta(n)$ با استفاده از درخت بازگشت :



* زیرشاخه سمت چپ زودتر از زیرشاخه های راست تمام میشود و این درخت یک حالت اریب دارد.

سطر	هزینه
1	cn
2	$cn/4 + 3n/4 = cn$
3	cn
⋮	//
⋮	//
⋮	//

تا جایی هزینه cn باقی میماند که به برگ ها برسیم و کم کم هزینه ها متفاوت میشه تا همه زیرشاخه ها به 1 برسند.

سایز مسئله در سطر i ام در چپ‌ترین شاخه $= n/4^{i-1}$

سایز مسئله در برگ $= 1 \longrightarrow 1 = n/4^{i-1} \rightarrow \boxed{\log_4^n + 1 = i}$

تعداد سطرهای چپ‌ترین شاخه درخت.

محاسبه حد پایین هزینه‌ها: در این حالت فرض می‌کنیم که ارتفاع کل درخت با ارتفاع سمت چپ‌ترین شاخه مساوی است.

هزینه	سطر
cn	1
cn	2
	$\vdots$
cn	$\log_4^n$

هزینه سطرها $= \log_4^n \times cn = cn\log_4^n$

هزینه برگ‌ها $= c'\boxed{\sqrt{n}} \longrightarrow$ تعداد برگ‌ها در درختی که باینری است و ارتفاع درخت $\log_4^n$ است.

تعداد گره‌ها در سطر i ام $= 2^{i-1}$

تعداد برگ‌ها در این حالت $= 2^{(\log_4^n+1)-1} = 2^{\log_4^n} = n^{\log_4^2} = \boxed{\sqrt{n}}$ ✓

$T(n)$ حد پایین $= c.n.\log_4^n + c'\sqrt{n} \in \Theta(n\log n)$

محاسبه حد بالای هزینه‌ها: در این حالت فرض می‌کنیم که ارتفاع درخت مساوی ارتفاع سمت راست‌ترین شاخه درخت است.

سایز مساله در سمت راست ترین شاخه $= \left(\frac{3}{4}\right)^{i-1} \cdot n$

سایز مساله در برگ ها برابر یک است $\rightarrow \quad 1 = \left(\frac{3}{4}\right)^{i-1} n \rightarrow i = \log_{4/3}^{n} + 1$

هزینه سطرها $= c.n.\log_{3/4}^{n}$

هزینه برگ ها $= c'.n^{\log_{4/3}^{2}}$

$T(n)$ حد بالای $= cn\log_{4/3}^{n} + c'n^{\log_{4/3}^{2}} \in \Theta(n^{\log_{4/3}^{2}})$

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

$T(n) \in O(n^{\log_{4/3}^{n}})$

حال میخواهیم ثابت کنیم که $T(n) \in \Theta(n\log n)$

باید ثابت کنیم که $\Leftarrow \quad T(n) \in O(n\log n)$

$\exists\, c>0, n_0>0 \mid \forall\, n \geq n_0,\ T(n) \leq c.n\log n$

روش درخت بازگشت

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + \Theta(n)$$

$$T(1) = c' \qquad T(2) = c''$$

ثابت $T(n) \in O(n \lg n)$

$$\exists c > 0,\ n_0 > 0 \mid \forall n \geq n_0 \quad T(n) \leq c \cdot n \cdot \log_2 n$$

$n = 2$

$$T(2) \leq c \cdot 2 \cdot \log_2 2$$

$$\leq 2 \cdot c \longrightarrow \boxed{c \geq \frac{T(2)}{2}}$$

اگر → پایه برقرار است

$n = 2$

فرض استقراء
$$\begin{cases} T\left(\frac{n}{4}\right) \leq c \cdot \frac{n}{4} \lg \frac{n}{4} \\ T\left(\frac{3n}{4}\right) \leq c \cdot \frac{3n}{4} \lg \frac{3n}{4} \end{cases}$$

$$-2c + \frac{3}{4}\log_2 3 + c''' \leq 0 \rightarrow \boxed{c \geq \frac{c'''}{2 - \frac{3}{4}\lg 3}}$$

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + \Theta(n)$$

$$\leq c\frac{n}{4}\lg\frac{n}{4} + c \cdot \frac{3n}{4} \cdot \lg\frac{3n}{4} + \Theta(n)$$

$$\leq c\frac{n}{4}(\lg n - \lg 4) + c\frac{3n}{4}(\lg 3 + \lg n - \lg 4) + \Theta(n)$$

$$\leq c \cdot n \lg n - c \cdot n \lg 4 + c\frac{3n}{4}\lg 3 + c''' \cdot n$$

$$-c \cdot n \lg 4 + c\frac{3n}{4}\lg 3 + c''' \cdot n \leq 0$$

$$\leq c \cdot n \lg n$$

Subject:
Year. Month. Date.

هزینه برگ ها $= c'.n^{\log_{\frac{4}{3}}2}$

$T(n)$ حد بالای $= c\,n\log_{\frac{4}{3}} n + c'\,n^{\log_{\frac{4}{3}}2} \in \Theta(n^{\log_{\frac{4}{3}}2})$

$\Longrightarrow T(n) \in \Omega(n\lg n) \quad , \quad T(n) \in O(\log_{\frac{4}{3}}2)$

← به دست آوردن رشد $T(n)$:

حدس: $T(n) \in O(n\lg n)$

$\exists c>0 , n_0>0 \mid \forall n \geq n_0 , T(n) \leq c.n.\lg n$

با استفاده از استقرا حدس خود را اثبات می کنیم:

اگر

$n=2 \rightarrow T(2) \leq c.2.\log_2 2 \Longrightarrow \underbrace{c''}_{T(2)} < 2.c \rightarrow c \geq \frac{T(2)}{2}$

پایه برقرار است

حالت کلی:

فرض:

$T(\frac{n}{4}) \leq c.\frac{n}{4}.\lg\frac{n}{4}$

$T(\frac{3n}{4}) \leq c.\frac{3n}{4}\lg\frac{3n}{4}$

حکم:

$T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + \theta(n) \leq$

$\frac{cn}{4}\lg\frac{n}{4} + c\times\frac{3n}{4}\lg\frac{3n}{4} + \theta(n)$

$\leq \frac{cn}{4}(\lg n - \log 4) + \frac{c3n}{4}(\log 3 + \lg n - \lg 4) + \theta(n)$

$\leq c.n.\lg n - c\lg 4 + \frac{3cn}{4}\log 3 + c'''\times n \leq 0$

$\Longrightarrow -2c + \frac{3}{4}\log\frac{3}{4} + c''' \geq 0$

PAPCO

قضیه اصلی (The master theorem):

برای حل بعضی از روابط فقط کار می‌کند

← مسائلی با فرم $T(n) = aT(n/b) + f(n)$ و $(b > 1)$

قضیه اصلی سه حالت دارد: $f(n) \;?\; n^{\log_b^a}$

الف) اگر وجود داشته باشد $\varepsilon > 0$ به طوری که $f(n) \in \Theta(n^{\log_b^a - \varepsilon})$

آنگاه $T(n) \in \Theta(n^{\log_b^a})$

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

ج) اگر $\exists\, \varepsilon > 0$ که $f(n) \in \Omega(n^{\log_b^a + \varepsilon})$, $\exists\, 0 < c < 1$ که

$af(\frac{n}{b}) \leqslant c.f(n)$ آنگاه $T(n) \in \Theta(f(n))$

جلسه هشتم؟ (1401, 12, 20)

مثال 1: فرض کنید $T(n) = 4 \cdot T(\frac{n}{2}) + \Theta(n)$

$f(n) \in \Theta(n)$ $\quad a = 4 \quad b = 2$

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

حالت اول → $n < n^2$

بررسی حالت الف ←

$$f(n) \in \Theta(n^1) \in \Theta(n^{\log_2^4 - \varepsilon})$$

$$\in \Theta(n^{2-\varepsilon}) \Rightarrow 2 - \varepsilon \geqslant 1 \Rightarrow \boxed{\varepsilon \leqslant 1}$$

یک ε ای وجود دارد.

شرط الف برقرار است پس: $T(n) = \Theta(n^{\log_2^4}) = \Theta(n^2)$

مثال 2: $T(n) = 2T(n/2) + n^2$

$f(n) = n^2 \quad ? \quad n^{\log_2^2} = n^1$

بررسی حالت ج ←

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

$$\in \Omega(n^{1+\varepsilon})$$

$$\rightarrow 1 + \varepsilon \leqslant 2 \rightarrow \boxed{\varepsilon \leqslant 1}$$

$$af\left(\frac{n}{b}\right) \to 2f\left(\frac{n}{2}\right) = 2\left(\frac{n}{2}\right)^2 = \frac{n^2}{2} \overset{?}{\leq} c.f(n) = cn^2$$

$$\boxed{\frac{1}{2} \leq c \leq 1}$$

$$\Rightarrow T(n) \in \Theta(f(n)) \to T(n) \in \Theta(n^2)$$

مثال 3: $T(n) = 3T\left(\frac{n}{3}\right) + \Theta(n)$

$$f(n) = \Theta(n) \quad ? \quad n^{\log_b^a} = n^{\log_3^3} = n^1$$

$f(n) \in \Theta(n) \in \Theta(n^{\log_b^a}) \in \Theta(n^{\log_3^3}) \in \Theta(n^1)$ $\leftarrow$ بررسی حالت ب

$$T(n) \in \Theta(n^{\log_b^a} . \log n) = \Theta(n^1 . \log n) = \Theta(n \log n)$$

مثال 4: $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$

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

$$f(n) = n\log n \quad ? \quad n^{\log_b^a} = n^{\log_2^2} = n^1$$

$f(n) = n\log n \in \Omega(n^{\log_b^a + \varepsilon}) \in \Omega(n^{\log_2^2 + \varepsilon})$ $\leftarrow$ بررسی حالت ج

$$\in \Omega(n^{1+\varepsilon})$$

$\exists \varepsilon > 0$? $\longrightarrow$ $\log n \in \Omega(n^{\varepsilon})$ وجود ندارد ε مثبتی که

این مسئله را نمی‌توان با روش قضیه اصلی حل کرد!

تعمیم یافته قضیه اصلی سه حالت تفسیر می‌کند.

قضیه اصلی سه حالت دارد: $f(n) \; ? \; n^{\log_b^a}$

الف) اگر وجود داشته باشد $\varepsilon > 0$ به طوری که $f(n) \in \Theta(n^{\log_b^a - \varepsilon})$

آنگاه $T(n) \in \Theta(n^{\log_b^a})$

ب) اگر $f(n) \in \Theta(n^{\log_b^a} \cdot (\log n)^k)$ آنگاه $T(n) \in \Theta(n^{\log_b^a} \cdot (\log n)^{k+1})$

ج) اگر $\exists \varepsilon > 0$ که $f(n) \in \Omega(n^{\log_b^a + \varepsilon})$, $\exists\, 0 < c < 1$ که

$af(\frac{n}{b}) < c \cdot f(n)$ آنگاه $T(n) \in \Theta(f(n))$

حل مثال ۹ با روش تعمیم یافته: $T(n) = 2T(\frac{n}{2}) + n\log n$

$$f(n) = n\log n \quad ? \quad n^{\log_b^a} \cdot (\log n)^k$$

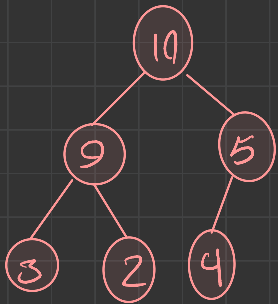
$$f(n) = n\log n \in \Theta(n^{\log_b^a}) \cdot (\log n)^1 \longrightarrow k = 1$$

$\Longleftarrow$ طبق حالت ب تعمیم یافته قضیه اصلی

$$T(n) \in \Theta(n^{\log_b^a} \cdot (\log n)^{k+1}) \in \Theta(n^{\log_2^2} \cdot (\log n)^{1+1}) = \Theta(n (\log n)^2)$$

تحلیل الگوریتم های مرتب سازی : * heap sort   * Quick sort

یادآوری max heap ← فرزندان هیچ ترتیبی ندارند ، صرفا پدر بزرگتر مساوی فرزندانش است



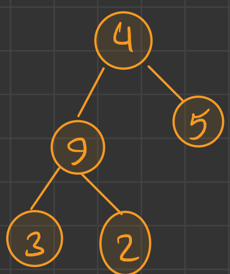
تابع heapify ← بین پدر و فرزندانش max می‌گیره و بزرگترین رو می‌زاره جای پدر

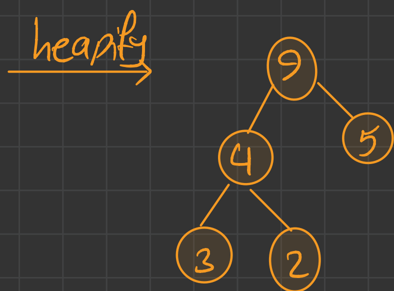
برای مرتب سازی دونه دونه ریشه را حذف می‌کنیم و درخت را heapify می‌کنیم

← آخرین عنصر سمت راست ترین شاخه درخت در ریشه قرار می‌گیرد.

1 :

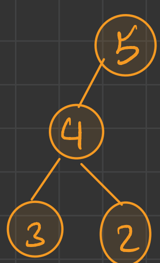
					10





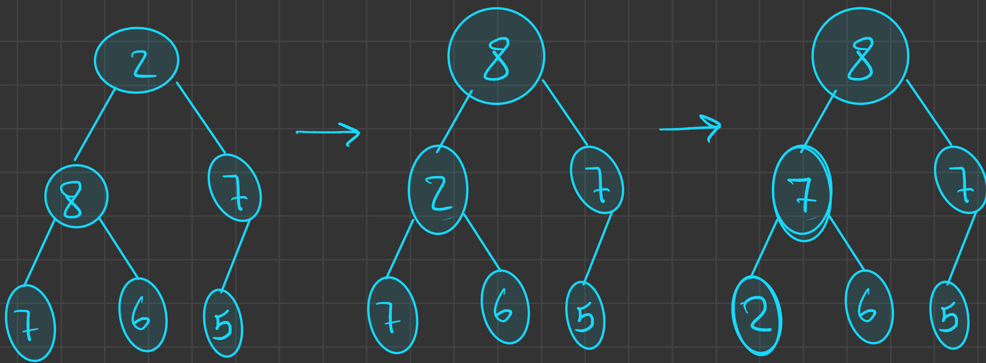
2 :

				9	10



			5	9	10	:3
		4	5	9	10	:4
	3	4	5	9	10	:5
2	3	4	5	9	10	:6

```
Heap_Sort(A,n):
    Build_max_Heap(A,n)
    for (i=n to 2) do:
        Swap(A[1],A[i])
        heap_size --
        max_heapify(A,i)
```

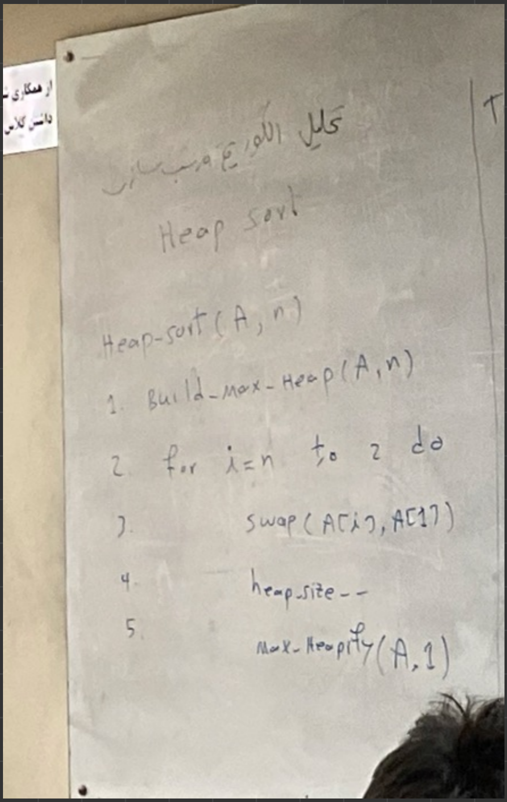


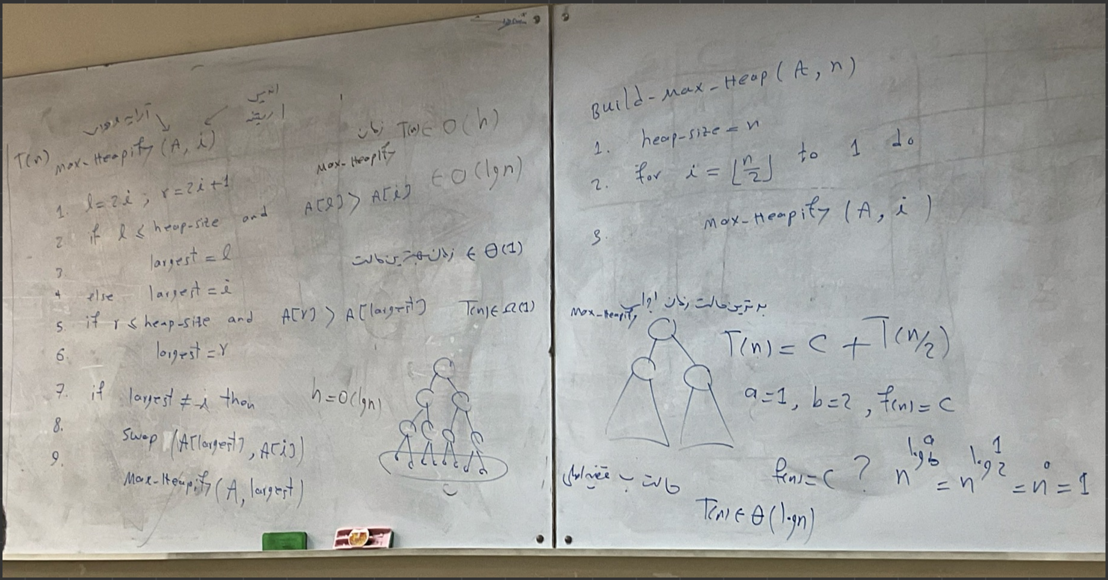
heap اندازه : heap_size = 6

اندازه آرایه : Length = 7

8	7	7	2	6	5	1

جلسه نهم: (1401,12,22)





Subject:
Year. Month. Date. ( )

```
if r ≤ heap-size and A[r] > A[largest]
    largest = r

if largest ≠ i then
    Max-Heapify(A, largest)
    swap(A[i], A[largest])
```

Build-Max-Heap(A, n)

```
heap-size = n
for i = ⌊n/2⌋ to 1 do
    max-Heapify(A, i)
```

زمان Max-Heapify : $T(n) \in O(h) \in O(\lg n)$

بصورت بازگشتی : بهترین حالت زمان اجرا

$T(n) = c + T(\frac{n}{2})$ ⟶

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

$f(n) = c \; ? \; n^{\log_b a} = n^{\log_2 1} = n^0 = 1$ حالت ۲ قضیه اصلی

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

$n = (2m+1) + (m) + 1 = 3m + 2$

$m = \frac{n-2}{3}$

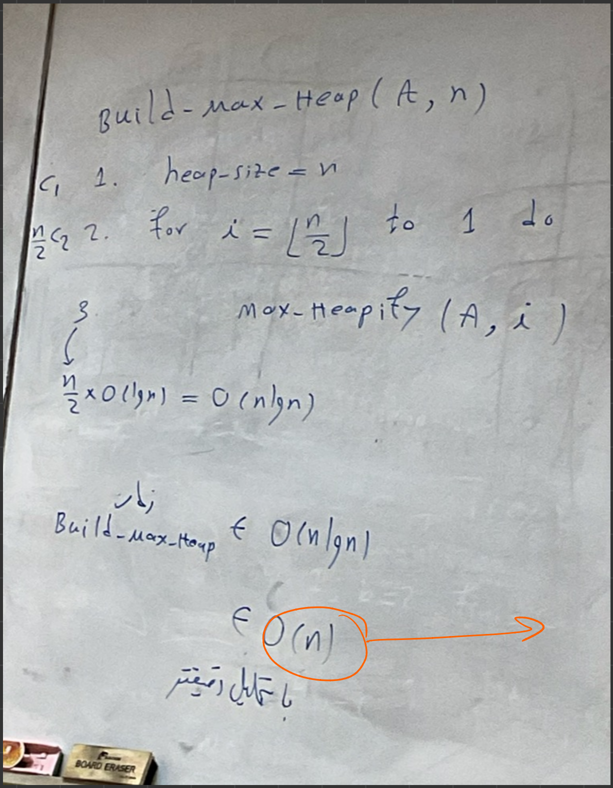
$T(n) = c + T(\frac{2n}{3})$

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

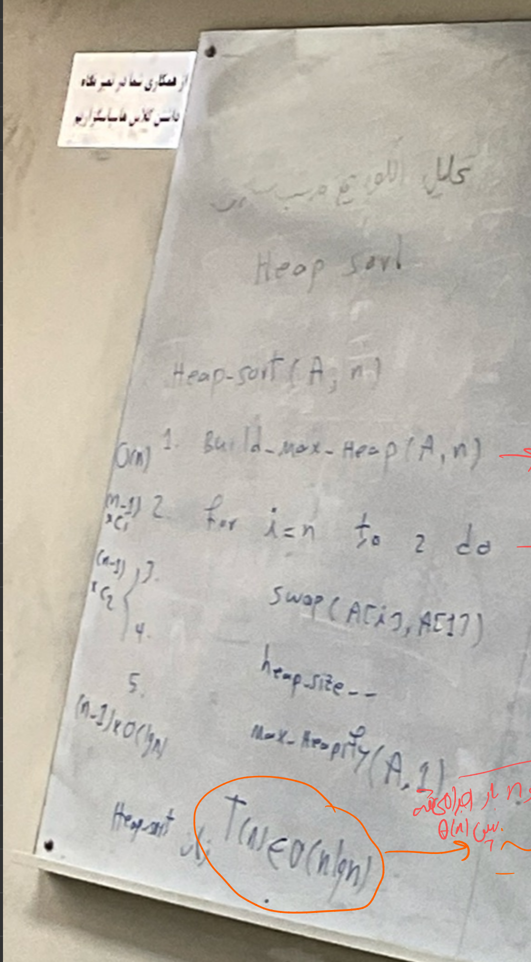
$f(n) = c \; ? \; n^{\log_{3/2} 1} = n^0 = 1$

حالت دوم قضیه اصلی ⟶ $T(n) \in \Theta(\log n)$

تحلیل زمانی Build_Max_heap :



چون زمان اجرای max_heapify
به ارتفاع درخت بستگی دارد. و تعداد
ارتفاع درخت های که زیاد هستند کم و
تعداد شاخه های کم ارتفاع زیاد هست پس می توان ثابت کرد که $O(n)$ است.



$\rightarrow \Theta(n)$ بهترین حالت

$\rightarrow \Theta(n)$

در بهترین حالت که هیچ کاری نکنه $\Theta(1)$ است و n بار اجرا میشه پس $\Theta(n)$

$T(n) \in O(n \lg n) \rightarrow$ در بدترین حالت است

بهترین حالت:

* بهترین حالت build max heap $\rightarrow \in \Theta(n) \Leftarrow$ هم حد بالا هم حد پایین n است.

اثبات درستی heapify : ($n = 1$, heap-size $= 1$)

الگوریتم پس از تعدادی مقایسه بدون تغییر آرایه تمام می‌شود پس آرایه ورودی که یک عنصر دارد،
بدون تغییر می‌ماند که max heap هم هست.

حالت گذار ← فرض برای هر درخت که کمتر از n عنصر دارد و زیر درخت چپ و راست آن max heap
است درست کار می‌کند.

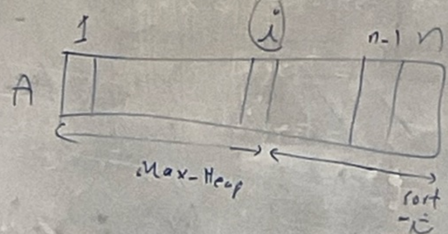
اثبات درستی Build-max-heap :

loop-invariant : در ابتدای iامین حلقه تمام زیر درخت‌ها با ریشه $i+1$ تا n یک max-heap
است.

پایه : ($i = n/2$) در این حالت loop invariant برقرار است. چون $i = n/2 + 1$ تا n همه
زیر درخت‌ها برگ هستند و max-heap هستند.

حالت گذار : فرض می‌کنیم برای i تا آخر حلقه برقرار است برای $i-1$ ثابت می‌کنیم که برقرار
است

وقتی
اعداد با

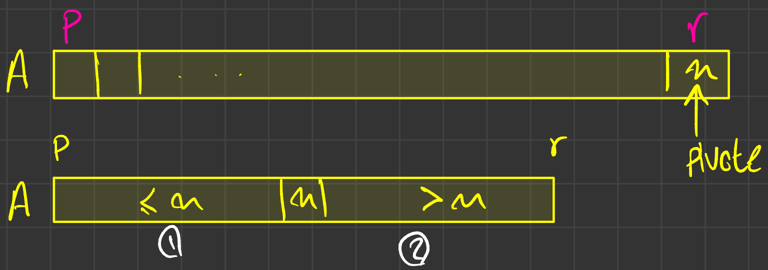


loop invariant : «در ابتدای گام iام حلقه
از i+1 تا n مرتب‌شده معکوس است و از 1 تا i
به Max-Heap است»

جلسه دهم : (1402, 1, 19)

تحلیل الگوریتم Quick sort :

* Quick sort مانند Merge sort از روش divided & conquore استفاده میکند



گام 1 : روش Partition تقسیم مساله به زیر مساله

گام 2 : فراخوانی بازگشتی و حل زیر مسئله ① ②

گام 3 ← ندارد

```
Quick_sort (A,p,r):
    if(p<r) :
        q=partition(A,p,r)
        Quick_sort(A,p,q-1)
        Quick_sort(A,q+1,r)
```

if(p<r) : → c_1

q=partition(A,p,r) → $\Theta(n)$

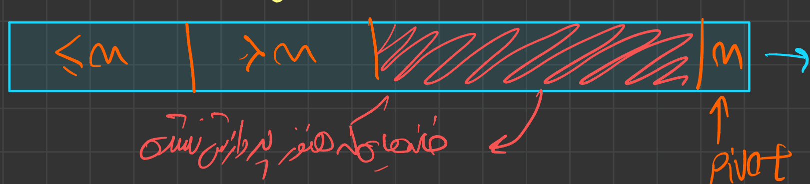
$T(n_1)$ ← Quick_sort(A,p,q-1)

$T(n_2)$ ← Quick_sort(A,q+1,r)

→ زمان اجرای این دو شرط مشخص نیست چون این دو بستگی دارند به آرایه ورودی

$n_1 = q - p$

$n_2 = n - n_1 - 1$



→ آرایه تابع Partition

تحلیل زمانی الگوریتم partition :

```
Partition(A,p,r):
    x=A[r]
    i=j=p
    for j=p to r-1 :
        if(A[j] <= x):
            swap(A[i],A[j])
            i++
    swap(A[r],A[i])
    return i
```

$x=A[r]$, $i=j=p$ $\rightarrow c_1$

for ... $\rightarrow c_2(n-1)$

swap, return $\rightarrow c_3$

بهترین حالت این است که if را اجرا نکنیم

که آرایه نزولی مرتب باشد این شرط برقرار نمی‌شود.

$c_2(n-1)$

بقیه دستورات همیشه اجرا می‌شوند.

$T(n) = c_1 + c_2(n-1) + c_3 \in \Theta(n)$

بهترین حالت $\rightarrow T(n) \in \Omega(n)$

بدترین حالت این است که if همیشه اجرا شود. آرایه صعودی مرتب باشد.

$T(n) = c_1 + c_2'(n-1) + c_3 \in \Theta(n) \quad (c_2' > c_2)$

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

* بدترین حالت در الگوریتم quick sort زمانی رخ می‌دهد که آرایه ورودی از اول مرتب باشد.

تحلیل زمانی الگوریتم Quick sort

$T(n) = c_1 + \Theta(n) + T(n_1) + T(n_2)$

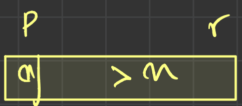
* اگر فرض کنیم که $n_1 = n_2 = n/2$

بهترین حالت

$$\boxed{T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)}$$

$\rightarrow$ رابطه بازگشتی merge sort

* اگر فرض کنیم : $n_2 = 0 , n_1 = n-1$ یا $n_2 = n-1 , n_1 = 0$



$$T(n) = \Theta(n) + T(0) + T(n-1) = T(n-1) + \Theta(n)$$ بدترین حالت

$$= \underbrace{c.n + c.(n-1) + \dots + c}_{cn} \in \Theta(n^2)$$

$$\Rightarrow \boxed{\begin{aligned} T(n) &\in \Omega(n \log n) \\ &\in O(n^2) \end{aligned}}$$

اثبات درستی الگوریتم partition :

← در طول اجرای حلقه درسته.

ثابت حلقه ← در ابتدای گام j ام حلقه الگوریتم زیرآرایه $A[p, i-1]$ شامل عناصر $\leq n$ و

زیرآرایه $A[i, j-1]$ شامل عناصر $> n$ ، زیرآرایه $A[j, r-1]$ شامل عناصر

پردازش نشده، و $n = A[r]$ مشخص است

حالت پایه : $i = j = p$ ← حالت پایه با جایگذاری در ثابت حلقه درست است.

حالت گذار : برای j فرض می‌کنیم درست است برای $j+1$ ثابت می‌کنیم که درست است

← دو حالت باید در نظر بگیریم. برای شرط $if(A[j] \leq n)$

جلسه یازدهم: (1402,1,19)

ادامه تحلیل Quick sort :

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

برای پارتیشن کردن

بهترین حالت زمانی رخ می‌دهد که دو طرف تقسیم یک کسری از n باشد و
بدترین حالت زمانی رخ می‌دهد که یک طرف یک ثابت باشد مثل k باشد و آن طرف $n-k$ باشد.

نسخه تصادفی الگوریتم Quick sort : (Randomized)

pivot را به صورت تصادفی انتخاب می‌کنیم.

```
Randimized_partition(A,p,r):
    i = RANDOM(p,r)
    swap(A[i],A[r])
    return partition(A,p,r)
```

```
randomized_Quick_sort(A,p,r):
    if(p<r):
        q=Raandomized_partition(A,p,r)
        randomized_Quick_sort(A,p,q-1)
        randomized_Quick_sort(A,q+1,r)
```

محاسبه میانگین زمان الگوریتم Quick Sort Randomized :

$$T(n) = O(n+X) \qquad E(T(n)) = O(n+E(X))$$

n: تعداد فراخوانی توابع

X: تعداد مقایسه‌ها

E: امید ریاضی

نام گذاری اعداد آرایه ← $z_1, z_2, z_3, \ldots, z_n$

به طوری که z_i ، i امین کوچک ترین عدد آرایه است.

مجموعه ی اعداد بین z_i و z_j ⟵ $Z_{i,j} = \{z_i, z_{i+1}, \ldots, z_j\}$

نکته: هر دو عدد حداکثر یک بار با هم مقایسه می شوند.

نکته: z_i ، z_j با هم مقایسه می شوند اگر و فقط اگر اولین pivot از مجموعه $Z_{i,j}$ ، خود z_i یا z_j باشد.

$$X_{i,j} = I\{z_i, z_j \text{ مقایسه شوند}\} = \begin{cases} 1 & \text{اگر } z_i, z_j \text{ با هم مقایسه شوند} \\ 0 & \text{در غیر این صورت} \end{cases}$$

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j} \qquad E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{i,j}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \overbrace{P\{z_i, z_j \text{ مقایسه شوند}\}}^{\frac{2}{j-i+1}}$$

* احتمال این که z_i با z_j مقایسه شوند برابر است با احتمال این که در اولین باری که در مجموعه $Z_{i,j}$ یک pivot انتخاب شوند، آن pivot ، z_i یا z_j باشد.

$$k = j - i \rightarrow \quad E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \leq \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \leq \sum_{i=1}^{n-1} O(\ln n) \leq O(n \log n)$$

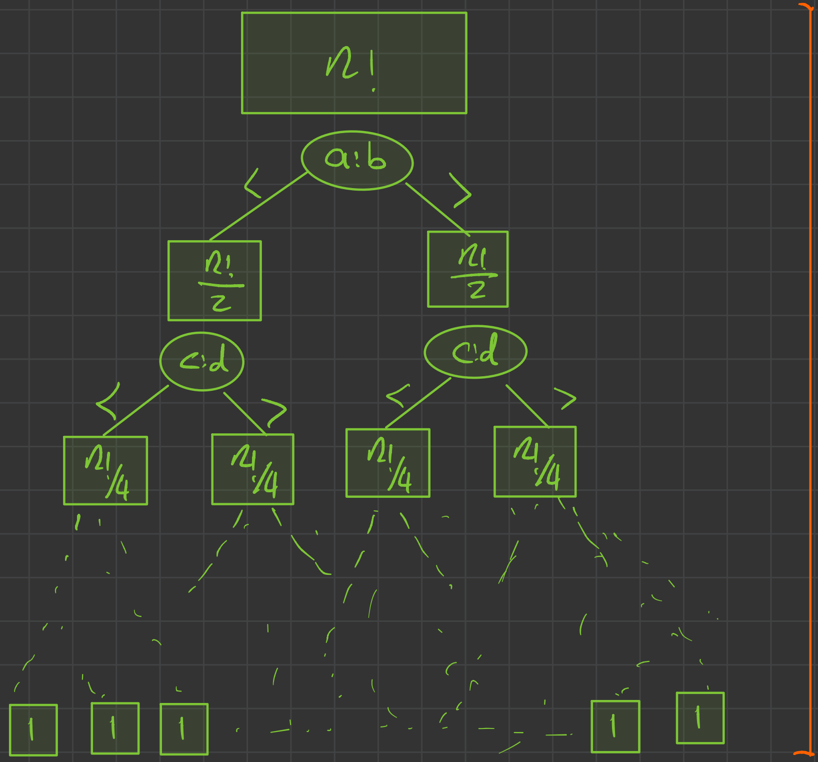
حالت گسسته $\int \frac{dn}{n}$ است

$\leq \ln n$

الگوریتم‌های مرتب‌سازی زمان خطی:
(Linear-time sorting algorithm)

n عدد متمایز، دارای $n!$ ترتیب مختلف است که یکی از این $n!$ ترتیب مختلف، ترتیب صعودی است.

* دقیقا برای هر ترتیبی که a بعد از b است همان مقدار ترتیب داریم که a قبل از b است.



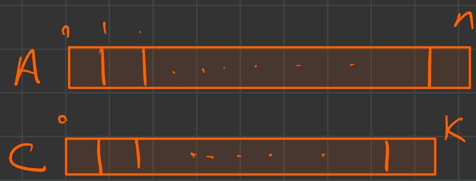
$\log(n!) \in \theta(n\log n)$

* هر الگوریتم مرتب‌سازی مبتنی بر مقایسه در بدترین حالت دارای پیچیدگی زمانی $\Omega(n\log n)$ است.

جلسه دوازدهم: (1402,1,21)

Sorting in Linear time:

* counting sort algorithm:



اگر n عدد صحیح در بازه 0 تا k داشته باشیم:

- یک آرایه k خانه ای می سازیم.
- روی آرایه اصلی یک بار کامل حرکت می کنیم ← A
- به ازای هر بار مشاهده یک عدد، خانه مربوط به آن عدد در آرایه k تایی را افزایش می دهیم.
- در نهایت به ترتیب از روی آرایه C، آرایه مرتب شده A را می سازیم.

* الگوریتم counting-sort کمی از این الگوریتم متفاوت است.

```
Counting_sort(A,n,k):
    for (i=0 to k):
        C[i]=0
    for (i=1 to n):
        C[A[i]]++
    for (i=1 to K):
        C[i] = C[i]+C[i-1]
    for (i=n down to 1):
        B[C[A[i]]] = A[i]
        C[A[i]--]
```

$\Theta(k)$ ← مقدار دهی اولیه شمارنده

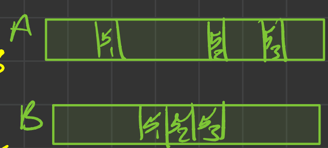
$\Theta(n)$ ← شمارش

$\Theta(k)$ ← محاسبه جمع تجمعی

$\Theta(n)$ ← قرار دادن هر عدد از A در جای مناسب B

تحلیل زمانی الگوریتم $= \Theta(n+k)$

ترتیب اعداد مساوی در ورودی را در خروجی حفظ می کند → الگوریتم stable



* اگر مقدار k ثابت باشد، الگوریتم counting sort، خطی است.

* اگر بازه k فیکس نباشد و درجه یک برحسب n باشد، باز هم خطی است $\Theta(n+n) = \Theta(n)$

* اگر بازه k فیکس نباشد و درجه دو برحسب n باشد، $\Theta(n^2)$ خواهد شد.

* چون k بازه‌اش معلوم نیست و شکل تحلیل زمانی‌اش $\Theta(n+k)$ است و شکل خطی می‌باشد پس ممکن است به چندجمله‌ای می‌رسند

* الگوریتم Radim sort :

- مرتب‌سازی n عدد صحیح d رقمی در مبنای k

صدگان	دهگان	یکان
3	2	5
3	5	1
1	2	5
1	1	0
2	0	4

مرتب‌سازی براساس رقم صدگان →

1	2	5
1	1	0
2	0	4
3	2	5
3	5	1

هر دسته را دهگان را مقایسه می‌کنم →

1	1	0
1	2	5
2	0	4
3	2	5
3	5	1

→ روش متداول

* radim sort برعکس عمل می‌کند.

325, 351, 125, 110, 204 → 110, 351, 204, 325, 125 → 204, 110, 325, 125, 351 → 110, 125, 204, 325, 351

* مزیت radim sort نسبت به روش متداول این است که گروه‌بندی نیاز ندارد.

* این مرتب‌سازی رقم‌ها باید stable باشد.

* الگوریتم‌های بازگشتی معمولا ترتیب را بهم می‌زنند پس stable نیست.


```
Radix_sort(A,d,n,k):
    for (i=1 down to d):
        sort A on i_th digit using a stable sorting algorithm (like
            counting sort)
```

مبنای K ← بازه رقم‌ها $0-(K-1)$

تحلیل زمانی ⇒ $\Theta(d(n+K))$

مثال: n عدد داریم، هر عدد در بازه $0-n^2$ است. ← اگه کل عدد رو یک رقم بگیریم:

counting sort $\Theta(n+n^2)$

← چی کار کنیم که کمتر از n^2 بشه؟!

← سوال 8.3.4 کتاب - edition 2

8.3.4 _ CLRS

Show how to sort n integers in the range 0 to n^3 -1 in O(n) time

first run through the list of integers and convert each one to base n then radix sort them. each number will have at most $\log_n(n^3) = 3$ digits so there will only need to be taken on, so we can use counting sort to each digit in O(n) time.

جلسه سیزدهم : (1402 ,1 ,26)

الگوریتم Bucket sort:

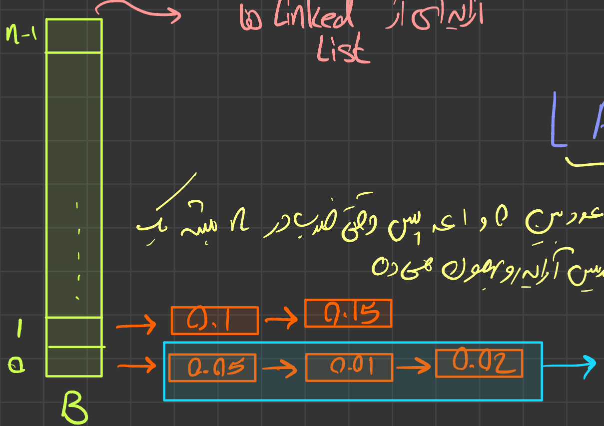
مرتب سازی n عدد در بازه صفر تا یک

فرض: اعداد در بازه صفر تا یک به صورت یکنواخت توزیع شده اند ← روی زمان الگوریتم تاثیر می‌ذارد

آرایه‌ای از Linked List ها

$\lfloor A[i] * n \rfloor$: اندیس bucket(A[i])

یه عدد بین 0 و 1 ه پس وقتی ضرب در n بشه یک اندیس آرایه رو بهمون می‌ده



هر bucket رو جداجدا sort می‌کنیم

```
Bucket_sort(A,n):
    for(i=1 to n):                                   ← (n+1)c1
        insert A[i] into Bucket B[⌊A((i)*n)⌋]        ← n×O(1)
    for (i=0 to n-1):                                ← (n+1)c2
        sort Bucket B[i]                             ← insertion sort: Σ_{i=0}^{n-1} O(n_i^2)
    concatenate Bucket B[0] ... B[n-1]               ← O(n)
```

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

بهترین حالت : $n=1$ $\longrightarrow$ $T(n)=O(n)$

بدترین حالت : $n_i=0$ for $i\neq k$, $n_k=n$, $0\leq k<n$ $\longrightarrow$ $T(n)=O(n^2)$

حالت میانگین :

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

$$=O(n)$$

اثبات : $E(n_i^2)=2-1/n$

$$X_{ij}=I\{\text{در باکت } i \text{ ام قرار بگیرد } A[j]\}=\begin{cases}1 & \text{اگر } A[j] \text{ در باکت } i \text{ ام قرار بگیرد}\\ 0 & \text{وگرنه}\end{cases}$$

$$n_i=\sum_{j=1}^{n}X_{ij}\rightarrow (n_i)^2=\Big(\sum_{j=1}^{n}X_{ij}\Big)^2=\sum_{j=1}^{n}\sum_{k=1}^{n}X_{ij}\cdot X_{ik}=$$

n_i: تعداد اعدادی که در باکت i ام قرار می‌گیرند

$$\sum_{j=1}^{n}(X_{ij})^2+\underbrace{\sum_{j=1}^{n}\sum_{k=1}^{n}}_{j\neq k}X_{ij}\cdot X_{ik}$$

X_{ij} و X_{ik} مستقل از هم هستند

$$E[n_i^2]=\sum_{j=1}^{n}\boxed{\underbrace{E[X_{ij}^2]}_{=E[X_{ij}]=1/n}}+\sum_{j=1}^{n}\sum_{k=1}^{n}\boxed{\underbrace{E(X_{ij}\times X_{ik})}_{=E[X_{ij}]\cdot E[X_{ik}]}}$$

$$E[n_i^2] = \underbrace{\cancel{\sum_{j=1}^{n} \frac{1}{n}}}_{1} + \underbrace{\sum_{j=1}^{n} \sum_{k=1}^{n} \frac{1}{n} \cdot \frac{1}{n}}_{\frac{n^2-n}{n^2}} = 2 - \frac{1}{n}$$

$$E[n_i] = \sum_{j=1}^{n} E[X_{ij}] = \sum_{j=1}^{n} \frac{1}{n} = 1 \longrightarrow$$

پس به طور متوسط در هر bucket یک عدد است.

جلسه چهاردهم : (1402 , 1 , 28)

پیدا کردن میانه مسئله selection :

← پیدا کردن k امین کوچک‌ترین عدد بین n عدد.

میانه → if $k = n/2$

راه حل 1 : Sort کنیم و عدد k ام آرایه مرتب شده را برگردانیم → $\Omega(n\log n)$

حالات خاص : → $k=1$ → minimization → $O(n)$

→ $k=n$ → maximization → $O(n)$

راه حل 2 : عناصر 1 تا k را یکی یکی پیدا کنیم. → $O(kn)$

راه حل 3 : یک min heap می‌سازیم $O(n)$ و k بار کل عنصر کوچک‌تر را استخراج کنیم

heapify → $O(\log n)$

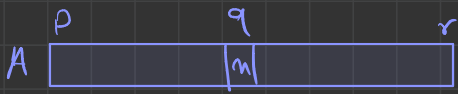
$(n + k\log n)$

راه حل 4 : ساخت درخت جستجوی دودویی افزوده

$\underbrace{O(n\log n)}_{\text{ساخت درخت}} + \underbrace{O(\log n)}_{\text{پیدا کردن k امین عدد}}$

راه حل 5 : با $A[1:k]$ یک درخت جستجوی دودویی متوازن می‌سازیم. هر عدد جدیدی که به درخت اضافه می‌کنیم عنصر max درخت را حذف می‌کنیم. به این ترتیب در درخت همواره k تا کوچک‌ترین عددی که تا به حال دیده‌ام ذخیره شده است.

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



راه حل نهایی:

رتبه مشخص m = $i = q-p+1$

```
selection(A,p,r,k):
    if (p==r):
        return A[p]
    q = partition(A,p,r)
    i = q-p+1
    if (k==i):
        return A[q]
    if (k<i):
        return selection(A,p,q-1,k)
    else:
        return selection(A,q+1,r,k-i)
```

$\to O(1)$, $\to O(n)$, $\to O(1)$, $\to O(m)$, $\to O(m)$

m اشاره می‌شود که اندازه کار در quick sort داشتیم

بدترین حالت: $m = n-1$

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

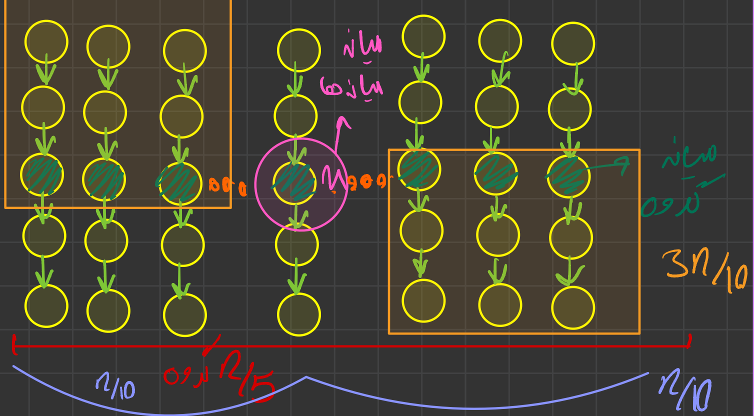
بهترین حالت: $m = n/2$

$$T(n) = \quad T(n) = T(n/2) + O(n) = O(n)$$

$3n/10$

بهبود زمان الگوریتم selection:

* حداقل $3n/10$ کمتر از m
* حداکثر $7n/10$ بزرگتر از m
* حداقل $3n/10$ بزرگتر از m
* حداکثر $7n/10$ کوچکتر از m



(1402,2,4) جلسه پانزدهم :

بهبود الگوریتم :

$O(n)$
۱- اعداد را به گروه های ۵ تایی تقسیم می‌کنیم و هر گروه را جداگانه مرتب می‌کنیم
میانه گروه‌ها را جدا می‌کنیم.

2- به صورت بازگشتی میانه این میانه‌ها را با الگوریتم selection پیدا می‌کنیم. فرض کنید این میانه میانه‌ها m باشد. ← $T(n/5)$

$O(n)$
3- با m آرایه ورودی را پارتیشن می‌کنیم و اندیس m یعنی i را پیدا می‌کنیم.
4- اگر k=i آنگاه m را برمی‌گردانیم
5- اگر k<i آنگاه به صورت بازگشتی در نیمه اول آرایه عدد kام را می‌یابیم
6- اگر k>i آنگاه به صورت بازگشتی در نیمه دوم آرایه عدد (k-i)ام را می‌یابیم
$T(\frac{7n}{10})$

حالت سوم $T(n)=aT(\frac{n}{b})+O(n)$ ← کمتر از n

در بدترین حالت
$$T(n)=\boxed{T\left(\frac{7n}{10}\right)+T\left(\frac{n}{5}\right)}+O(n)=O(n)$$

* اگر جمع سرهای T ها n بشود و O(n) هم داشته باشیم ، جواب $n\log n$ می‌شود
example : $2T(n/2)+O(n)$ $O(n\log n)$

تحلیل سرشکن: (Amortized Analysis)

← یک الگوریتم که n عمل (operation) انجام می‌دهد. و c عدد ثابت

← $(n-c)$ تا عمل ← زمان $O(1)$

← c تا عمل ← زمان $O(n)$

یک تحلیل:

هزینه هر عمل از $O(n)$ بیشتر است. n عمل داریم پس هزینه کل n عمل از $n\times O(n)=O(n^2)$ بیشتر نیست

تحلیل دقیق‌تر:

$$\text{هزینه کل} = (n-c)\times O(1) + c\times O(n) = O(n)$$

3 روش تحلیل سرشکن:

1. aggregate analysis ← $\text{هزینه سرشکن} = \dfrac{\text{هزینه کل عملیات}}{n}$
2. accounting method
3. potentied function

هر عمل یک هزینه واقعی c_i و یک هزینه سرشکن $\hat{c}_i$ دارد

$$\underbrace{\sum \hat{c}_i}_{\text{مجموع هزینه‌ها سرشکن}} \geq \underbrace{\sum c_i}_{\text{مجموع هزینه‌های واقعی}}$$

← شبیه 2 است ولی در 2 همه هزینه‌ها را نگه می‌داریم

ولی در 3 فقط هزینه کل را نگه می‌داریم.

مثال) stack operations :

$pop() \rightarrow O(1)$

$push() \rightarrow O(1)$

$Multi\text{-}pop(k) \rightarrow \boxed{O(\min(k,S))}$

k: تعداد عناصری که باید pop شود.

S: تعداد عناصر داخل استک

در بدترین حالت $O(n)$

نکته ۱) تا چیزی در stack، push نشود، نمی‌توان pop کرد.

نکته ۲) تعداد کل pushها، حداکثر n است. ← هزینه کل pushها = $O(n)$

* n تا multipop نمی‌توانیم داشته باشیم زیرا باید چیزی در stack باشد که pop میشه

بلکه حتی در بدترین حالت n-1 push خواهیم داشت و یک Multi-pop

هزینه push $= (n-1) \times O(1) = O(n)$

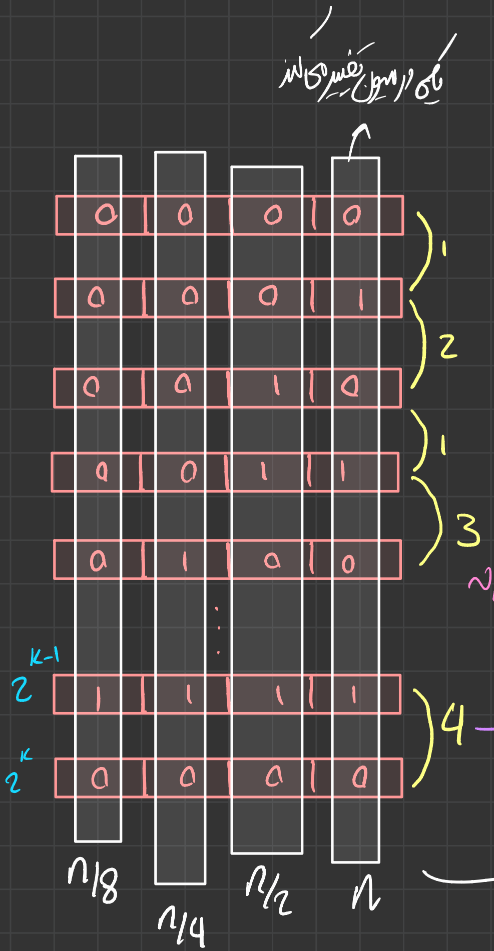
هزینه pop $= 1 \times O(n) = O(n)$

هزینه کل $= O(n) + O(n) = O(n)$

هزینه سرشکن هرعملیات $= \dfrac{O(n)}{n} = 1$

جلسه شانزدهم : (1402,2,9)

مثال : Binary Counter



حداکثر هزینه یک عمل increament ← (log n)

n عمل increament داریم ← هزینه n عمل

$n\,O(\log n) \rightarrow O(n\log n)$

مثلا این counter از 0 تا 15 را می‌شمارد پس 4 بیت داریم که می‌تونه تغییر کنه ← $\log_2 16 = 4$

4 → بدترین حالت

کل هزینه تغییرها →

$$\sum_{i=0}^{\log_2^n -1} \frac{n}{2^i} < n \underbrace{\sum_{i=0}^{\infty} \frac{1}{2^i}}_{2n} \qquad \frac{1}{1-\frac{1}{2}} = 2$$

$$\text{هزینه سرشکن عمل increment} = \frac{\text{هزینه کل}}{\text{تعداد عملیات}} = \frac{2n}{n} = \boxed{2} \checkmark$$

* بدترین حالت از 2^{k-1} به 2^k رخ می‌دهد چون در آن حالت k بیت با هم تغییر میکنند.

روش های طراحی الگوریتم :

1) روش تقسیم و غلبه (Divide & conquer)

بهتر از تقسیم و غلبه

2) روش برنامه ریزی پویا (Dynamic programming)

بهتر از برنامه ریزی پویا

3) روش حریصانه (Greedy)

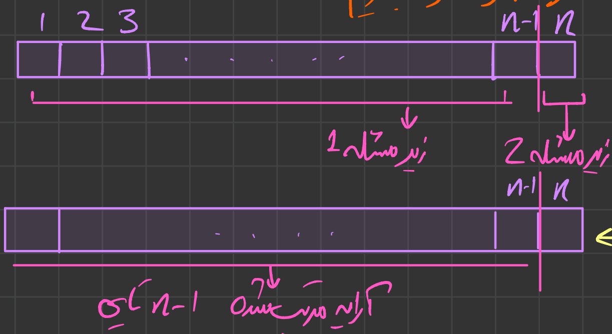
4) جست و جوی فضای حالات (backtracking , bronch & bound)

همه حالات رو چک کن!

روش divide & conqure :

	merge sort	Quick sort
گام اول	تقسیم آرایه ورودی به دو قسمت مساوی با اندازه $n/2$ $O(1)$	با استفاده از partition آرایه به دو قسمت احتمالا نامساوی تقسیم می‌شود. $O(n)$
گام دوم	merge-sort(A,p,q) merge-sort(A,q+1,r) $T(n/2)+T(n/2)$	Quick-sort(A,p,q-1) Quick-sort(A,q+1,r) $T(k)+T(n-k+1)$
گام سوم	merge(A,p,q,r) $O(n)$	———
	$O(n\log n)$	$O(n\log n)$ $O(n^2)$ → در بدترین حالت

مسئله مرتب سازی با این تفاوت که عنصر n را کنار بگذاریم :



گام 1 : تقسیم

گام 2 : حل بازگشتی

گام 3 : ترکیب

مانند while الگوریتم insertion-sort که
جایگاه A[n] را در A[1.. n-1] پیدا می‌کند
و درج می‌کند

```
Recursive_insertion_sort(A,n):
    if (n>1):
        Recursive_insertion_sort(A,n-1)
        key = A[n]
        i=n-1
        while (i>0 and key <A[i]):
            A[i+1] = A[i]
            i=i-1
        A[i+1]= key
```

if (n>1): → $O(1)$ — گام 2

Recursive_insertion_sort(A,n-1) → $T(n-1)$; n-1 → گام 1

key … A[i+1]= key → $O(n)$ — گام 3

$$T(n) = T(n-1) + O(n) = \underbrace{O(n^2)}_{\text{در بدترین حالت}}$$

در بدترین حالت .

جلسه هجدهم : (1402,2,16)

مسئله مسابقات ورزشی :

n تیم ورزشی باهم مسابقه می دهند. هر تیم در هر روز حداکثر 1 مسابقه می دهند.

کل مسابقه $\Rightarrow \frac{n(n-1)}{2}$

* حداقل $(n-1)$ روز لازم است

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

روش تقسیم و غلبه:

گروه X

گروه A

گروه B

گروه Y

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

شیفت

شیفت

* اگر تیم‌هامان فرد بود، یک تیم مجازی اضافه می‌کنیم، که زوج شود. و تیمی که با تیم مجازی بازی می‌کند. در واقع وضعیت آن تیم استراحت است.

اگر بعد از تقسیم تیم‌های گروه فرد شد:

①

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

گروه ۱ →

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

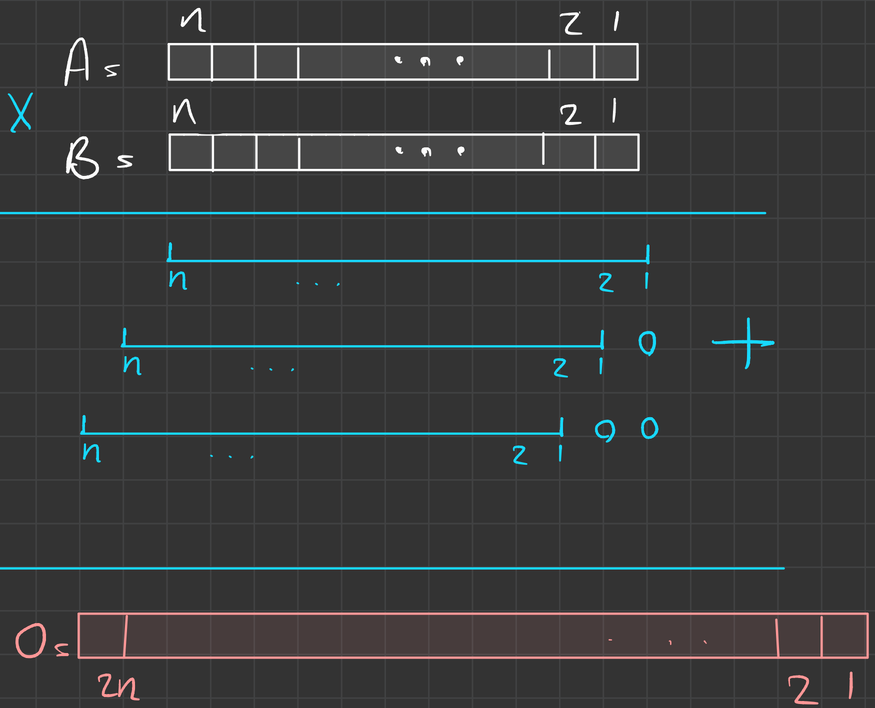
⇒

تیم \ روز			
	2	3	—
	1	—	3
	—	1	2

②

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

مسئله ضرب 2 عدد n بیتی :

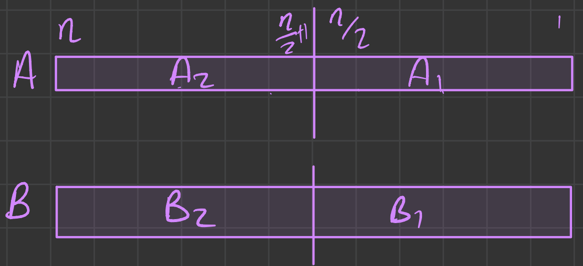


این روش یک روش تقسیم و غلبه است :

n تا زیرمسئله داریم که هر کدام ضرب یک بیت در n بیت است.

$$\Theta(n^2) = n^2 + \frac{n(n-1)}{2} \quad \left\{ \begin{array}{l} n^2 \text{ تا ضرب بیتی} \\ \frac{n(n-1)}{2} \text{ تا جمع} \end{array} \right.$$

اگه از وسط نصف اش کنیم:



$$A = A_2 \times 2^{n/2} + A_1$$

$$B = B_2 \times 2^{n/2} + B_1$$

$$C = A \times B = (A_2 \times 2^{n/2} + A_1) \times (B_2 \times 2^{n/2} + B_1)$$

$$= \underbrace{A_1 B_1}_{1} + \underbrace{A_2 B_2}_{2} \times \underbrace{2^{n}}_{\text{هزینه } n} + (\underbrace{A_1 B_2}_{3} + \underbrace{B_1 A_2}_{4}) \underbrace{2^{n/2}}_{\text{هزینه } n/2}$$

هزینه n

* 4 مسئله به سایز $n/2$ باید حل بشود

$$T(n) = 4T(n/2) + O(1) + O(n) = \Theta(n^2)$$

$O(n) \rightarrow$ هزینه ترکیب

$O(1) \rightarrow$ هزینه تقسیم

جلسه بیستم : (1402,2,23)

مسئله ضرب ماتریس ها : $M_1 \times M_2 \times M_3 \times \dots \times M_n$

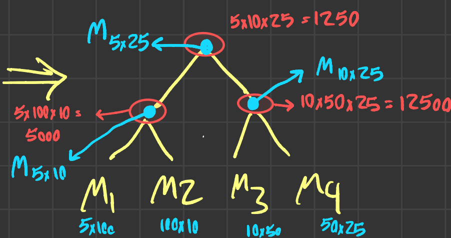
ما نمی‌توانیم جای ضرب ها را تغییر داد ولی می‌توانیم ترتیب ضرب ها را تغییر دهیم.

این ترتیب ضرب ها در میزان محاسبه ما تاثیر داره

$$\overset{1}{M}_{4\times100} \times \overset{2}{M}_{100\times10} \times \overset{3}{M}_{10\times50} \times \overset{4}{M}_{50\times24}$$

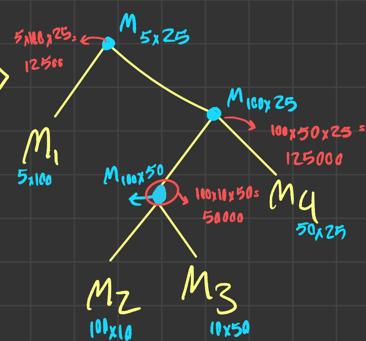
$((M_1 \times M_2) \times (M_3 \times M_4))$ ⟹

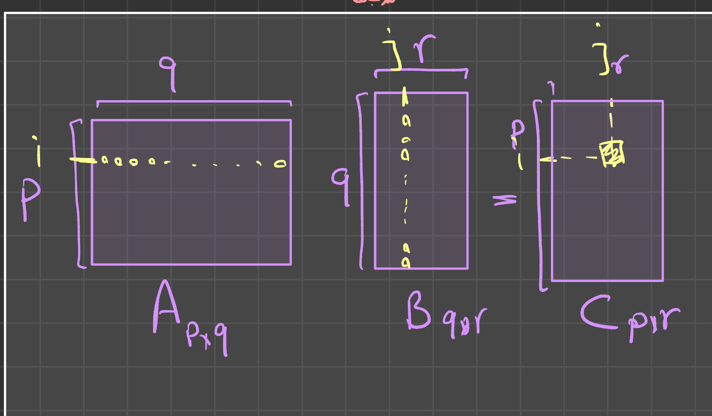
تعداد کل ضرب‌ها $= 18750$



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

تعداد کل ضرب‌ها $= 187500$





تعداد کل ضرب ها $= p\times q\times r$

تعداد پرانتزگذاری های n ماتریس $\Rightarrow$ $P(n)=\sum_{k=1}^{n-1} P(k)P(n-k) = \frac{1}{n+1}\binom{2n}{n}$

عدد کاتالان ^_^

$\in \Omega(2^n)$

یکی از راه‌های حل پرانتزگذاری، روش حریصانه است، که میاد سعی می‌کنه بیشترین انتخاب کلی را انجام بده و بیشترین dimention رو اول حل می‌کنه.

جواب حریصانه $\Leftarrow$ $((M_1\times M_2)\times(M_3\times M_4))$

جواب حریصانه لزوما بهینه نیست

Dynamic Programming:

$P_{i,j}$ مسئله :

$$\Big(\overbrace{\underbrace{\big(\underset{d_{i-1}\times d_i}{M_i}\times \underset{d_i\times d_{i+1}}{M_{i+1}}\times \cdots \times \underset{d_{k-1}\times d_k}{M_k}\big)}_{\text{زیرمسئله ①}}}^{d_i\times d_k}\times\overbrace{\underbrace{\big(\underset{d_k\times d_{k+1}}{M_{k+1}}\times \cdots \times \underset{d_{j-1}\times d_j}{M_j}\big)}_{\text{زیرمسئله ②}}}^{d_k\times d_j}\Big)$$

$C_{i,j}$ تعداد ضرب‌های انجام‌شده در پرانتزگذاری :

$$C_{i,j}=\min\{C_{i,k}+C_{k+1,j}+d_{i-1}\times d_k\times d_j\}$$

$C_{i,j}$: تعداد ضرب ها در مسئله $P_{i,j}$

$C_{i,k}$: تعداد ضرب‌ها در زیرمسئله ①

$C_{k+1,j}$: تعداد ضرب ها در زیرمسئله ②

$d_{i-1}\times d_k\times d_j$: تعداد ضرب‌ها در آخرین ضرب

$i\le k<j$

کوچک‌ترین زیرمسئله $\rightarrow$ $P_{i,i}$: M_i $\Rightarrow$ $C_{i,i}=0$

جلسه بیست و یکم : (1402,2,25)

حل مسئله پرانتزگذاری در ضرب ماتریس‌ها با روش DP :

مثال : $M_1 \times M_2 \times M_3 \times M_4$ $\quad n=4$

$5\times100 \quad 100\times10 \quad 10\times50 \quad 50\times20$

i \ j	1	2	3	4
1	0	5000 k=1	7500 k=2	13750 k=3
2		0	50000 k=2	37500 k=2
3			0	12500 k=3
4				0

$(i=1, j=2) \longrightarrow$

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

k فقط می‌تواند 1 باشد $\Rightarrow$ $= C_{1,1} + C_{2,2} + d_0 d_1 d_2 =$

$$0 + 0 + 5\times100\times10 = \boxed{5000} \checkmark$$

$(i=2, j=3) \longrightarrow$

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

$$2 \leq k \Rightarrow = C_{2,2} + C_{3,3} + d_1 d_2 d_3 =$$

$$\boxed{0 + 0 + 100 \times 10 \times 50 = 50000}$$

$(i=3, j=4) \longrightarrow$

$$\underset{(k=3)}{C_{3,4}} = C_{3,3}, C_{3,4}, d_2 d_3 d_4 =$$

$$= 0 + 0 + 10 + 50 \times 25 = 12500$$

$(i=1, j=3) \longrightarrow$

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

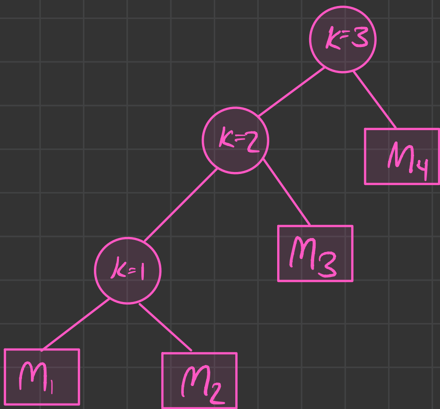
$$= \min \begin{cases} k=1 & C_{1,1} + C_{2,3} + d_0 d_1 d_3 = \\ & 0 + 50000 + 50 \times 100 \times 50 = 75000 \\ k=2 & C_{1,2}, C_{3,3} + d_0 d_2 d_3 = \\ & 5000 + 0 + 5 \times 10 \times 50 = 7500 \end{cases}$$

$(i=2, j=4) \longrightarrow$

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

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

$$(i=1, k=4) \rightarrow = \min \begin{cases} k=1: C_{1,1}+C_{2,4}+d_0d_1d_4 = 0+37500+5\times100\times25 = 50000 \\ k=2: C_{1,2}+C_{3,4}+d_0d_2d_4 = 5000+12500+5\times10\times25 = \boxed{18750} \\ k=3: C_{1,3}+C_{4,4}+d_0d_3d_4 = 7500+0+5\times50\times25 = 13750 \end{cases}$$



جواب بهینه $\Rightarrow ((M_1\times M_2)\times M_3)\times(M_4)$

زمان حل یک زیر مسئله × تعداد زیر مسائل = زمان الگوریتم مبتنی بر DP

تعداد مقایسه‌ها

$$\text{زمان} = \frac{n\times(n+1)}{2}\times O(n) = O(n^3)$$

$$\text{تحلیل سرشکن} = n\times1 + (n-1)2 + (n-2)\times3 + \dots + 1\times n =$$

$$\sum_{i=0}^{n-1}(n-i)\times(i+1) \Rightarrow$$ اگه جواب این رو هم حساب کنم باز به n^3 می‌رسیم.

مسئله ۶ : (LCS) : Longest common Sub-Sequence

$\vec{a}$: $a_1, a_2, a_3, \ldots, a_{n-1}, a_n$

$\vec{b}$: $b_1, b_2, b_3, \ldots, b_{n-1}, b_n$

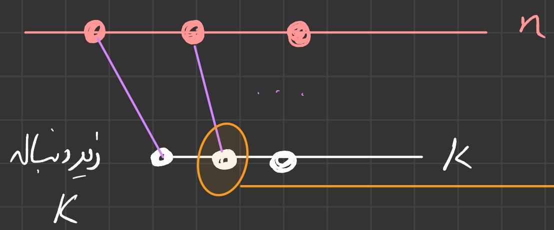
$\vec{a}$: 1, 2, 5, 3, 4, 6 $\Rightarrow$ یک زیردنباله $\vec{a}$: 1, 5, 3, 4

$\vec{b}$: 2, 3, 5, 4, 1

$$LCS(\vec{a}, \vec{b}) = \begin{cases} 2, 3, 4 \\ 2, 5, 4 \end{cases}$$

* این الگوریتم برای شباهت سنجی استفاده می‌شود. هر چقدر LCS آن‌ها بزرگ‌تر باشد، شبیه‌تر هستند.

- اگر بخواهیم همه حالت‌ها را چک کنیم $\Leftarrow$ $n\binom{n}{k}$



اگر این نشود بقیه زیردنباله را بررسی نمی‌کنیم.

جلسه بیست و سوم: (1402,3,6)

روش حریصانه (Greedy):

D.P	Greedy
چند انتخاب داریم که بین n تا Min/Max می‌گیریم	بین انتخاب های ممکن یکی را انتخاب می‌کنیم (greedy choice)

مسئله خرد کردن اسکناس با سکه ها:

سکه ها: $1, 3, 4$ و 5 ریالی

15 ریال اسکناس $= 3 \times 5$ ریالی

8 ریال اسکناس $= 2 \times 8$ ریالی

انتخاب حریصانه ⇐ انتخاب سکه های بزرگتر

7 ریال اسکناس $= \boxed{5+2+1}$ → حریصانه

$= \boxed{4+3}$ → بهینه

⚠ حریصانه ممکن است بهینه نباشد.

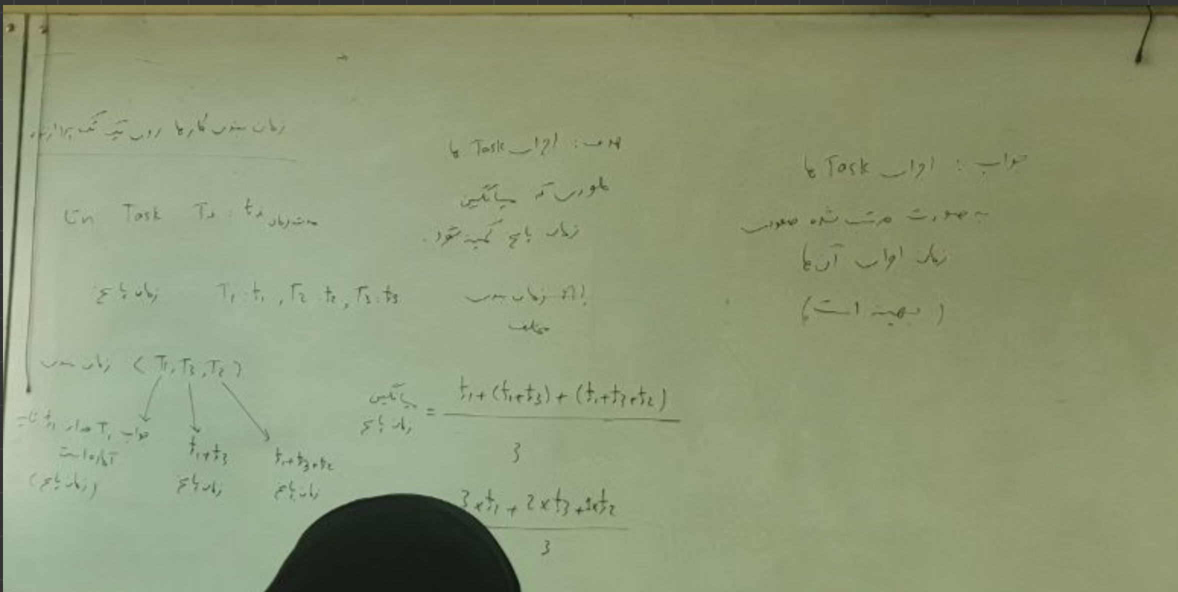
اسکناس M: $w_1, w_2, \dots, w_n$

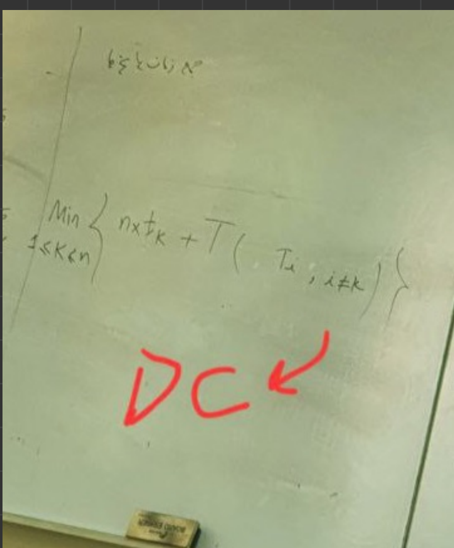
$$\min \begin{cases} ((M-w_n), w_1, \dots, w_n) + 1 \\ (M, w_1, \dots, w_{n-1}) \end{cases}$$

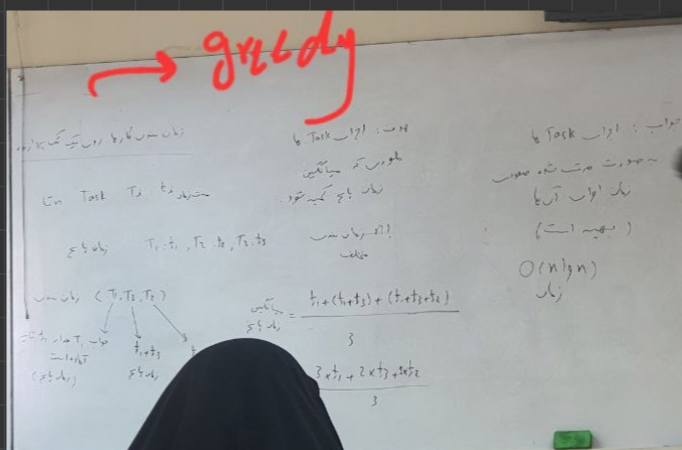
⇐ روش DP

زمان‌بندی کارها روی یک پردازنده:

$\text{task } i\text{n} \Rightarrow T_1, T_2, \ldots, T_n$







به نام خدا

زمان بندی کارها با جریمه تأخیر

n تا کار باید انجام دهیم

هر کار در زمان واحد انجام می‌شود (مدت زمان = 1)

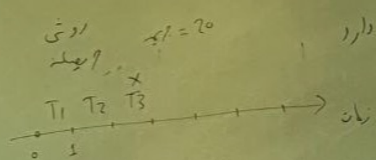
هر کار یک سررسید d_i دارد

اگر قبل از سررسیدش زمان‌بندی نشود

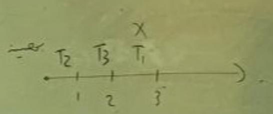
جریمه P_i به آن تعلق می‌گیرد.

هدف: انجام کارها به طوری که کمترین جریمه به ما تعلق بگیرد.

انتخاب حریصانه ①: کاری که سررسید زودتر دارد اول انجام دهیم. (بهینه نیست)

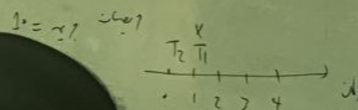


مثال نقض
$$\begin{cases} T_1 = (1, 10) \\ T_2 = (2, 20) \\ T_3 = (2, 20) \end{cases}$$



جریمه = 10

انتخاب حریصانه ②: کاری که جریمه بیشتر دارد اول انجام شود (بهینه نیست)



مثال نقض
$$\begin{cases} T_1 = (1, 10) \\ T_2 = (2, 20) \end{cases}$$

T_1, T_2

مثال)

انتخاب حریصانه ③ : کارها را به ترتیب نزولی جریمه‌ها
(بهینه است)
زمان‌بندی می‌کنیم. هر کاری (دقیقاً)
قبل از سررسید خودش زمان‌بندی می‌شود.
اگر محل (دقیقاً قبل از سررسید) پر بود
به سمت چپ محور زمان حرکت می‌کنیم و
کار را در اولین محل خالی زمان‌بندی می‌کنیم.
اگر هیچ جای خالی پیدا نشد آنگاه به این کار
جریمه تعلق می‌گیرد و بنابراین آن را در آخرین محل
ممکن زمان‌بندی می‌کنیم.

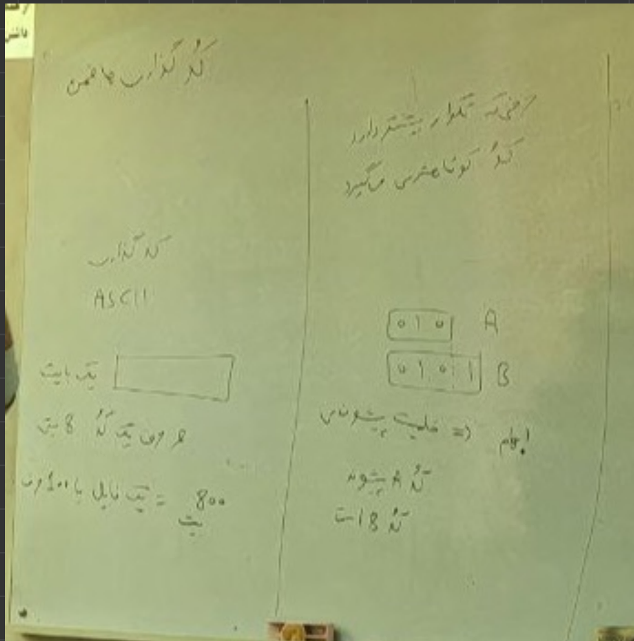
* هزینه‌ها با لذت‌خیال
بیشتر نمیشه!

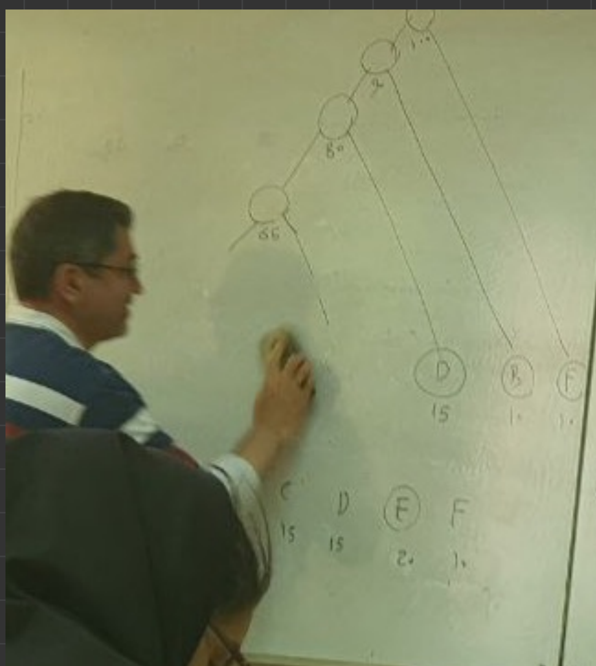
	✓	✓	✓	✓	×	×	✓
کارها	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
p_i	70	60	50	40	30	20	10

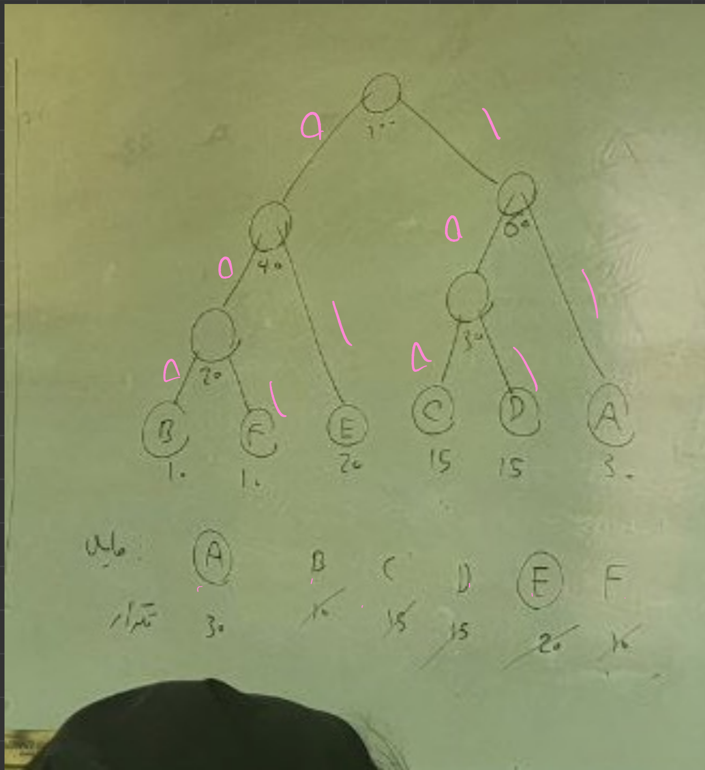
0–1	1–2	2–3	3–4	4–5	5–6	6–7
T_4	T_2	T_3	T_1	T_7	$\times$ T_6	$\times$ T_5

$$\text{جریمه کل} = \underbrace{30}_{\text{برای } T_5} + \underbrace{20}_{\text{برای } T_6} = 50$$

مثال)
...ترتیب نزولی جریمه‌ها
...می‌کنیم. هر کاری (دقیقاً)
...سید خودش زمان‌بندی می‌شود.
...دقیقاً قبل از سررسید پر بود
... محور زمان حرکت می‌کنیم و
...در اولین محل خالی زمان‌بندی می‌کنیم.
...ی خالی پیدا نشد آنگاه به این کار
...یرد و بنابراین آن را در آخرین محل
...بندی می‌کنیم.


کدگذاری هافمن
ASCII
010 A
0101 B
800


80
D
15
B
10
F
10
C D E F
15 15 20 10



$\underbrace{6 \times 8}_{\text{اسکی}}$

جلسه بیست و چهارم: (1402,3,8)

مسئله کوله پشتی:

کوله پشتی گنجایش M دارد، n شی داریم که وزن w_i و ارزش v_i دارند.

محدودیت: $\sum\limits_{\text{برای اشیا داخل کوله}} w_i \le M$

هدف: $\max \sum\limits_{\text{برای اشیا داخل کوله}} v_i$

نسخه های مسئله کوله پشتی:
- حالت صفر و یکی اشیا
- حالت کسری (Fractional) ← راحت تره!

حالت کسری:

انتخاب حریصانه ⇐ اشیا به ترتیب نزولی $\frac{v_i}{w_i}$ انتخاب می‌شوند.

مثال: $M=16$, $n=4$

i	v_i	w_i	v_i/w_i
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

شی 3 - $\frac{9}{10}$ از 10 وزن
شی 2 - وزن 5
شی 1 - وزن 2

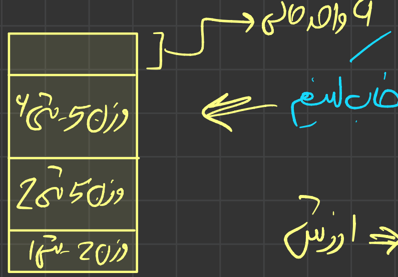
ارزش کل $= 40+30+\frac{9}{10}\times 50$

$= 70+45 = \boxed{115}$ ✓ بهینه

زمان حل مسئله $= O(n\log n)$

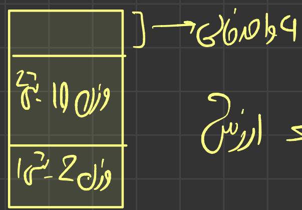
حالت صفر و یکی مسئله :

مثال قبل با استفاده از حالت صفر و یک :



روش حریصانه ⇐ اشیا با ارزش بالا را انتخاب کنیم

ارزش ⇐ $40+30+10=\boxed{80}$ ✓



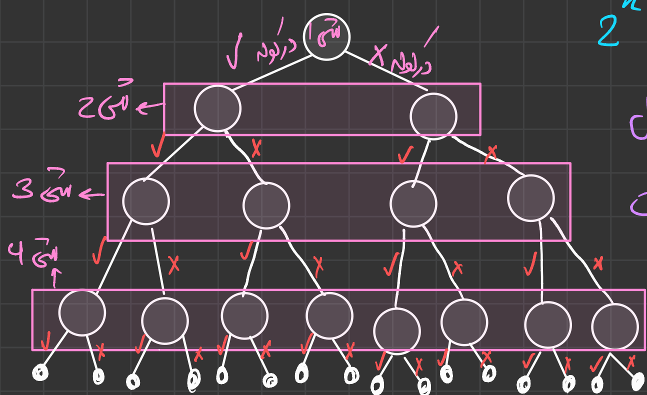
حالت بهینه ⇐

ارزش $= 40+50 = \boxed{90}$ ✓

* حالت صفر و یک مسئله کوله پشتی NP-hard است.

حل مسئله در زمان نمایی : ⇐ بررسی همه حالت ها با استفاده از درخت حالت.

n شی ⇐ تعداد برگ ها $= 2^n$



پیمایش DFS درخت فضای حالات معادل

شمارش یک شمارنده باینری n رقمی است

⇓

0	0	0	0
0	0	0	1
0	0	1	0

⋮

برای اشیاء با بیت 1 (انتخاب شده)

$\sum w_i \leq M$

$50 = 90\$$

$\sum v_i$

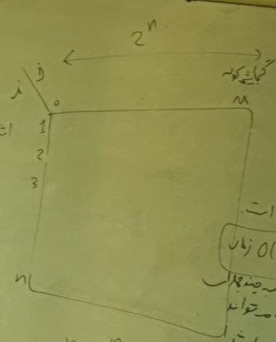
$O(n \times 2^n)$

مساله کوله پشتی

حل کوله پشتی 0-1 با روش D.P.

شئ nام یا در کوله قرار میگیرد یا نمیگیرد.

$$\text{Max}\begin{cases} v_n + K(M-w_n,\ n-1) & \text{اگر شئ } n \text{ در کوله قرار گیرد} \\ K(M,\ n-1) & \text{اگر شئ } n \text{ در کوله قرار نگیرد} \end{cases}$$



اگر وزن اشیاء و گنجایش کوله اعداد صحیح باشند $(M+1)$ ستون کافی است.

زمان $O(n.M)$

شرح چنین M میتواند 2^n باشد.

زیر مساله $n \times 2^n$

زمان حل هر زیرمساله $O(1)$

زمان $= O(1) \times n \times 2^n = O(n.2^n)$

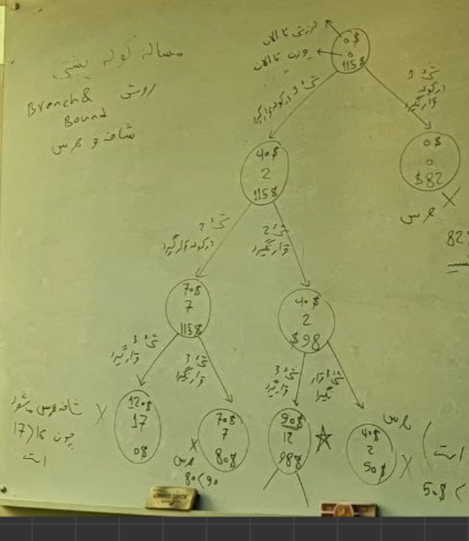
مثال $M=16,\ n=4$

i	v_i	w_i	v_i/w_i
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5 X
4	\$10	5	\$2

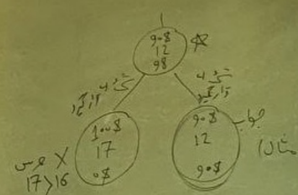
مساله کوله پشتی

روش Branch& Bound

شاخه و هرس



$82\$ < 90\$$ نقد نیست



مثال $M=16,\ n=4$

i	v_i	w_i	v_i/w_i
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5 X
4	\$10	5	\$2

(این جواب بالا) جواب حالت 0-1 است

$50\$ < 90\$$

جواب بهینه $= 115\$$ fractional