Subject:

Year. Month. Date. ( )

«طراحی الگوریتم»

کتاب مرجع: «An Introduction to Algorithms» ⟵ CLRS, Second Edition

فصل 1، 2 ← مقدمه، الگوریتم و تحلیل الگوریتم.

فصل 3 ← رشد توابع O, Ω, Θ

فصل 4 ← روابط بازگشتی

فصل 5، 6، 7 ← HeapSort, QuickSort

فصل 8 ← مرتب سازی در زمان خطی

فصل 9 ← مسئله Selection, Median

← روش حل مسئله «تقسیم و حل»

فصل 15 ← روش حل مسئله Dynamic Programming

فصل 16 ← روش حل مسئله حریصانه Greedy

فصل 22 ← گراف ها و الگوریتم های پایه

فصل 23 ← پیدا کردن درخت پوشای کمینه

فصل 24 ← پیدا کردن کوتاه ترین مسیر یا مسیر با وزن بیشینه در یک گراف. Single Source

فصل 25 ← پیدا کردن کوتاه ترین مسیر بین هر دو زوج. All-pairs

فصل 33 ← هندسه محاسباتی Computational Geometry

Subject:
Year. Month. Date. ( )

✓ الگوریتم‌ها در منابع مصرف می‌کنند: - پردازنده (زمان اجرا) - حافظه

ما معمولاً بیشتر به **زمان اجرای** الگوریتم‌ها توجه می‌کنیم.

مثال: فرض کنید دو الگوریتم مرتب‌سازی با زمان اجرای مقابل

Insertion Sort $\rightarrow c_1 \cdot n^2$

Merge Sort $\rightarrow c_2 \cdot n \log_2^n$

برایشان داریم که n تعداد ورودی‌های مسئله است.

فرض کنید Insertion Sort روی ماشین A با سرعت اجرای 10^9 دستور در ثانیه اجرا می‌شود و الگوریتم Merge Sort روی ماشین B با سرعت اجرای 10^7 دستور در ثانیه اجرا می‌شود.

همچنین فرض کنید الگوریتم Insertion Sort توسط یک برنامه‌نویس با $c_1 = 2$ پیاده‌سازی شده است و الگوریتم Merge Sort توسط برنامه‌نویس دیگر با $c_2 = 50$ پیاده‌سازی شده است.

(شرایط اجرای الگوریتم Merge Sort بسیار بدتر از الگوریتم Insertion Sort است!)

حال زمان اجرای دو الگوریتم مرتب‌سازی فوق را بدست می‌آوریم.

$$t_1 = \frac{2 \times (10^6)^2}{10^9} = 2000\,s$$

$$t_2 = \frac{50 \times 10^6 \times \log_2^{10^6}}{10^7} \simeq 100\,s$$

اگر ورودی مسئله برابر با $10,000,000$ باشد الگوریتم Insertion Sort عمل مرتب‌سازی را در $2,5$ روز و Merge Sort عمل مرتب‌سازی را در 20 دقیقه انجام می‌دهد!!

« تمرین 3-2-1 و مسئله 1-1-1 »

Subject:

Year. Month. Date. ( )

الگوریتم Insertion Sort

شبه کد مرتب سازی درجی:

```
Insertion_Sort (A)
1. for j ← 2 to length(A) do
2.     key ← A[j]
3.     i ← j - 1
4.     while i > 0 and A[i] > key do
5.         A[i+1] ← A[i]
6.         i ← i - 1
7.     A[i+1] ← key
```

الگوریتم مرتب سازی درجی اولین عنصر آرایه ورودی را مرتب شده در نظر می گیرد و از دومین عنصر آرایه شروع می کند. در هر مرحله تمت مرتب شده آرایه را از آخر پیمایش می کند و هر عنصر به عنصر بعدی خود منتقل می دهد. با این عمل جایی به سمت آخر را آنقدر ادامه می دهد تا جای عنصر جدید را پیدا کند.

اثبات درستی:

! برای اثبات درستی یک الگوریتم از روش اثبات استقرایی استفاده می کنیم. برای منظور یک گزاره با نام «ثابت حلقه» در نظر می گیریم. این گزاره باید به گونه ای باشد که با درستی آن، درستی الگوریتم نتیجه شود.

پس سه مرحله زیر را انجام می دهیم.

1. ابتدا باید بررسی نماییم گزاره ثابت حلقه در ابتدای اولین گام حلقه درست باشد.
2. در اجرای یک گام با فرض درست بودن ثابت حلقه در ابتدای گام بعد نیز ثابت حلقه درست باشد.
3. در انتهای حلقه درستی الگوریتم را نتیجه بگیریم (با درستی قسمت های 1 و 2، درستی این قسمت نیز نتیجه می شود)

✓ در قسمت بعد به منظور حلقه در گام درست در نظر گرفته شده و درستی گام بعد اثبات شود.

Subject:
Year. Month. Date. ( )

ثابت حلقه: «در ابتدای گام j ام حلقه (حلقه for در خطوط ۱ تا ۷) زیرآرایه $A[1 \dots j-1]$ شامل عناصر اولیه $A[1], \dots, A[j-1]$ بصورت مرتب شده می باشد.»

حالت اولیه: به ازای $j=2$ داریم:

«در ابتدای گام دوم حلقه (حلقه for در خطوط ۱ تا ۷) زیرآرایه $A[1 \dots 1]$ شامل عنصر اولیه $A[1]$ بصورت مرتب شده است» ✓

پس ثابت حلقه در ابتدای اولین گام حلقه درست است.

حالت نگهداری: در اینجا مانند استقراء عمل می کنیم. یعنی ثابت کنیم اگر ثابت حلقه در گام j ام حلقه درست باشد، آنوقت ثابت حلقه در گام $j+1$ ام حلقه نیز درست است.

پس فرض می کنیم گزاره زیر درست است (فرض استقراء):

«در ابتدای گام j ام حلقه (حلقه for در خطوط ۱ تا ۷) زیرآرایه $A[1 \dots j-1]$ شامل عناصر اولیه $A[1], \dots, A[j-1]$ بصورت مرتب شده می باشد» ✓

اکنون می خواهیم ثابت کنیم گزاره زیر درست است:

«در ابتدای گام $j+1$ ام حلقه (حلقه for در خطوط ۱ تا ۷) زیرآرایه $A[1 \dots j]$ شامل عناصر اولیه $A[1], \dots, A[j]$ بصورت مرتب شده می باشد» ✓

طبق فرض استقراء در زیرآرایه $A[1 \dots j]$ عناصر $A[1], \dots A[j-1]$ مرتب شده می باشند. اگر در Insertion Sort از عنصر $j-1$ ام شروع می کنیم، عنصر j را یکی به جلو می بریم و سپس عمل فوق را برای عنصر $j-2$ را الی آخر ادامه می دهیم. این عمل تا جایی ادامه می یابد که محل درست عنصر j ام مشخص شود. بنابراین زیرآرایه $A[1 \dots j]$ نیز بصورت مرتب شده در می آید.

بنابراین حکم استقراء ثابت می شود.

Subject:

Year. Month. Date. ( )

حالت نهایی: در حالت نهایی باید ببینیم ثابت حلقه چه وضعی پیدا می‌کند با فرض اینکه ورودی شده n باشد، داریم:

«زیر آرایه $A[1..n]$ شامل عناصر اولیه $A[1], \dots, A[n]$ بصورت مرتب شده است»

بنابراین طبق توضیحات ارائه شده در شبه کد ثابت حلقه در پایان حلقه نیز برقرار است.

⇐ بنابراین درستی الگوریتم Insertion Sort اثبات می‌شود. □

تحلیل الگوریتم‌ها

برای تحلیل الگوریتم‌ها باید بدانیم الگوریتم ما قرار است روی چه ماشینی اجرا شود. یعنی مدل محاسباتی خود را بدانیم. اگر ماشین ما برای مثال دستورات مرتب سازی داشته باشد، در اینصورت عمل مرتب سازی را بستگی به ورودی ندارد و در زمان ثابتی انجام می‌شود.

مدل محاسباتی که معمولا استفاده می‌شود، مدل RAM (Random Access Memory) است که در آن یک پردازنده با دستورات محاسباتی و منطقی ساده و معمولی با یک حافظه با دسترسی تصادفی در ارتباط است که دستورات برنامه یا الگوریتم را بصورت ترتیبی (نه موازی) اجرا می‌کند.

✓ در تحلیل الگوریتم‌ها یک تابع بر حسب سایز ورودی شده (اندازه ورودی). سایز ورودی شده بر حسب ماهیت مسئله می‌تواند متفاوت باشد. مثلا در یک جا می‌تواند تعداد ورودی‌ها باشد و در جای دیگر تعداد بیت‌های ورودی‌های شده.

Subject:
Year. Month. Date. ( )

تحلیل Insertion Sort : حلقه for حلقه Header حداقل یکی بیشتر از بدنه حلقه اجرا می‌شود. بنابراین Header حلقه for به تعداد n بار و بدنه حلقه به تعداد n-1 بار اجرا می‌شود. (n سایز شده، تعداد ورودی‌ها، طول آرایه ورودی است)

اکنون هر خط الگوریتم مرتب‌سازی درجی را تحلیل می‌کنیم.

	هزینه		تعداد	
1.	c_1	x	n	هر بار اجرای Header حلقه در زمان ثابتی اجرا می‌شود که با ضریب c_1 نشان داده شده است.
2.	c_2	x	$(n-1)$	دستور assignment خط دوم هم نیز در زمان ثابتی اجرا می‌شود.
3.	c_3	x	$(n-1)$	دستور assignment در خط سوم هم در زمان ثابت صورت می‌پذیرد.
4.	c_4	x	$\sum_{2}^{n} t_j$	حلقه while هر بار به تعداد متناظری اجرا می‌شود. فرض کنیم t_j
5.	c_5	x	$\sum (t_j - 1)$	کل زمان اجرای حلقه while در یک گام حلقه for به ازای j باشد.
6.	c_6	x	$\sum (t_j - 1)$	while حلقه Header نیز یکبار بیشتر از بدنه آن اجرا می‌شود.
7.	c_7	x	$(n-1)$	

بهترین حالت : اعداد به صورت صعودی به آرایه ورودی مرتب شده باشند. در اینصورت حلقه while اجرا می‌شود $t_j = 1$, $(t_j - 1) = 0$ پس از ضرب و جمع ثابت‌ها در نهایت یک تابع برحسب n به صورت خطی بدست می‌آید

$T(n) = a \cdot n + b \longrightarrow$ "linear"

بدترین حالت : اعداد به آرایه ورودی به صورت نزولی مرتب شده‌اند. در اینصورت حلقه while هر بار به تعداد j بار اجرا می‌شود، $t_j = j$ می‌شود.

در اینصورت پس از ضرب و جمع ثابت‌ها یک تابع درجه 2 برحسب n بدست می‌آید.

$T(n) = an^2 + bn + c \longrightarrow$ " Quadratic "

6

Subject:
Year. Month. Date. ( )

✓ الگوریتم Insertion Sort به روش افزایشی عمل مرتب‌سازی را انجام می‌دهد.

✓ روش افزایشی یا Incremental یک روش حل مسئله است.

مثال: مسئله Convex Hull یا پوسته محدب که تعدادی نقطه در صفحه داریم و می‌خواهیم یک چند ضلعی محدب روی این نقاط بسازیم به طوری که حداقل محیط را داشته باشد.
با روش Incremental می‌توان این مسئله را حل کرد. فرض می‌کنیم برای تعدادی نقطه چند ضلعی محدب‌شان را بدست آورده‌ایم. سپس یک نقطه اضافه می‌کنیم و چند ضلعی محدب قبلی را update می‌کنیم.

✓ روش دیگری برای حل مسائل، روش تقسیم و حل یا Divide & conquer می‌باشد.

✓ یک الگوریتم یا approach براساس تقسیم و حل سه گام دارد.

گام اول: **تقسیم**، تقسیم مسئله به تعدادی زیرمسئله از نوع همان مسئله‌ها ولی با اندازه کوچکتر.
گام دوم: **حل بازگشتی**، زیرمسائل را حل شده فرض می‌نماییم و زیرمسائل با کوچکترین اندازه را حل می‌کنیم (پایه بازگشت).
گام سوم: **ترکیب**، ترکیب جواب زیرمسائل برای بدست آوردن جواب مسئله اولیه.

مثال: می‌خواهیم مسئله مرتب‌سازی یک آرایه با روش تقسیم و حل، حل نماییم. جواب مورد نظر الگوریتم Merge Sort است. در این روش آرایه را از وسط نصف می‌کنیم و عمل مرتب‌سازی را برای هر یک از دو زیرآرایه جدید با روش Merge Sort ادامه می‌دهیم. سپس باید این دو آرایه مرتب شده را به گونه‌ای باهم ادغام کنیم که باز هم مرتب شده باقی بماند.

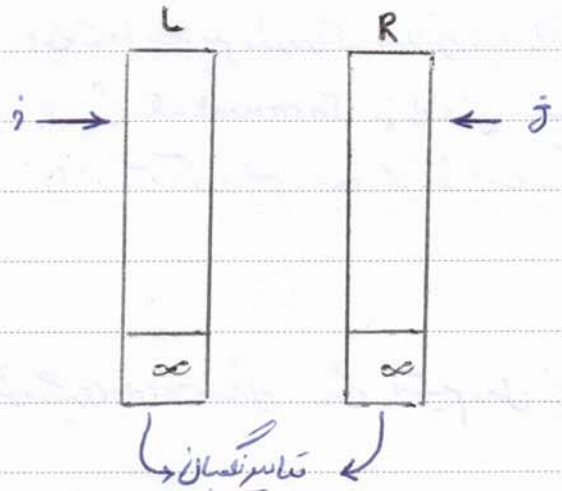
Subject:

Year. Month. Date. ( )

گام تقسیم ← کردن آرایه را به دو قسمت تقسیم کرد و آنقدر ادامه می‌دهیم تا به آرایه‌های جدید را برسد.

گام حل یا بازگشت ← زیرآرایه‌ها را با استفاده از تقسیم‌بندی مرتب شده فرض می‌کنیم و شرط خاتمه‌دهم زمانی است که طول زیرآرایه به 1 برسد. ($n=1$)

گام ترکیب ← در اینجا با استفاده از الگوریتم Merge، دو زیرآرایه مرتب شده را با یکدیگر ادغام می‌کنیم که مرتب شده باقی بماند.



شبه‌کد Merge Sort

```
Merge (A, p, q, r)          // زیرآرایه p-q و q+1-r مرتب‌شده هستند
1. n1 ← q-p+1
2. n2 ← r-q
3. ▶ Create arrays L[1..n1+1] and R[1..n2+1]
4. for i ← 1 to n1 do
5.       L[i] ← A[p+i-1]
6. for i ← 1 to n2 do
7.       R[i] ← A[q+i]
8. L[n1+1] ← ∞
9. R[n2+1] ← ∞
```

Subject:

Year. Month. Date. ( )

```
10. i ← 1
11. j ← 1
12. for k ← p to r do
13.     if L[i] ≤ R[j] then
14.         A[k] ← L[i]
15.         i ← i+1
16.     else
17.         A[k] ← R[j]
18.         j ← j+1
```

اثبات درستی: برای اثبات درستی الگوریتم بالا باز هم با یک ثابت حلقه استدلال می‌کنیم.

ثابت حلقه: در ابتدای گام kام از حلقه خطوط 12 تا 18 داریم:

زیرآرایه $A[p \dots k-1]$ شامل $k-p$ کوچکترین عناصر L و R بصورت مرتب شده است.

$L[i]$ و $R[j]$ کوچکترین عناصر آرایه خودشان هستند که هنوز کپی نشده‌اند.

حالت اولیه: $k=p$ ✓

حالت نگهداری: فرض می‌کنیم که در گام kام گزاره درست است. می‌باید با استفاده از فرض نشان دهیم که در گام $k+1$ نیز درست است.

برای این منظور باید حلقه for را در گام k ام یکبار اجرا کرد.

با بررسی عملکرد الگوریتم مشاهده می‌کنیم که درستی حالتهای دیگر اثبات می‌شود.

Subject:
Year. Month. Date. ( )

حالت نهایی: باید انتهای آخرین گام حلقه یعنی بعد از آن را بررسی نماییم تا همچنان ثابت حلقه درست باشد.

تحلیل زمان اجرا:

1, 2 → که این خطوط هزینه ثابت دارند.

8-11 → که این خطوط نیز هزینه ثابت دارند.

5. $n_1 \cdot c_1$
6. $n_2 \cdot c_2$

5-7 → $c(n_1 + n_2)$
↳ $\max\{c_1, c_2\}$

در خطوط 13 الی 18 چه قسمت if اجرا شود چه قسمت else آن یک هزینه ثابت مثل c_6 دارد.

13-18 → $c_6 \cdot n$

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

پس هزینه ادغام دو آرایه با تعداد عناصر آن‌ها رابطه متناسب است.

شبه کد مرتب‌سازی ادغام:

```
MergeSort(A, p, r)
1. if p < r then
2.    q ← ⌊(p+r)/2⌋          → گام تقسیم (1)
3.    MergeSort(A, p, q)      ┐
4.    MergeSort(A, q+1, r)    ┘ → گام حل (2)
5.    Merge(A, p, q, r)       → گام ترکیب (3)
```

Subject:

Year. Month. Date. ( )

تحلیل:

1, 2 → هزینه ثابت

3 → $T(n/2)$

4 → $T(n/2)$

5 → $\Theta(n)$

$$\Rightarrow T(n) = 2\,T(n/2) + \Theta(n)$$

! فرم کلی تابع زمانی الگوریتم حل مسائلی که بوسیله تقسیم و حل و بصورت بازگشتی حل می‌شوند بصورت زیر است.

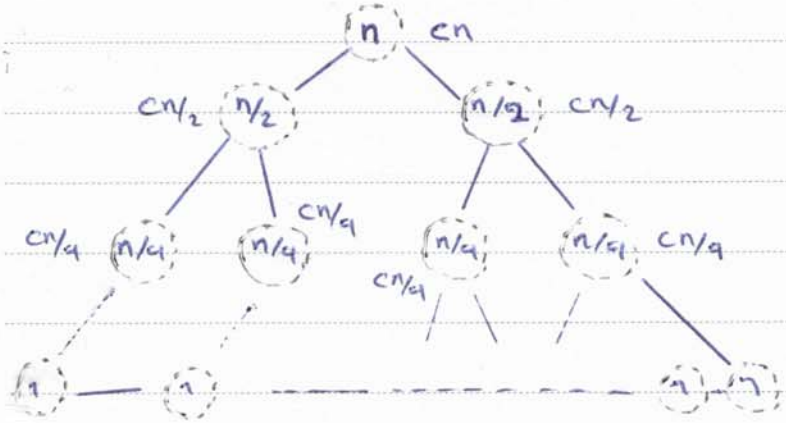
$$T(n) = a\,T(n/b) + D(n) + C(n)$$

(a) → «تعداد زیرمسائل»

(b) → «اندازه یک زیرمسئله»

D(n) → «هزینه گام تقسیم»

C(n) → «هزینه گام ترکیب»



✓ برای بدست آوردن زمان اجرای MergeSort یک درخت Binary در نظر می‌گیریم.

هزینه ثابت $\Theta(n)$ را نیز باید در نظر بگیریم.

$\Theta(n)$ را می‌توان بصورت cn نمایش داد.

✓ تا سطح آخر درخت به برگ‌ها، هزینه هر رأس cn می‌باشد.

برای هر برگ نیز هزینه‌ای ثابت داریم که هزینه حل شده با کوچکترین سایز است. (n)

تعداد برگ‌ها $\times$ $T(1)$

درخت باینری کامل n تا برگ دارد. بنابراین ارتفاع آن درخت $\log_2^n$ است.

هزینه کل برابر است با: $T(n) = \log_2^n \times cn + c'n \longrightarrow \Theta(n \log n)$

تمرین 2.3-7 ، 2.3-7

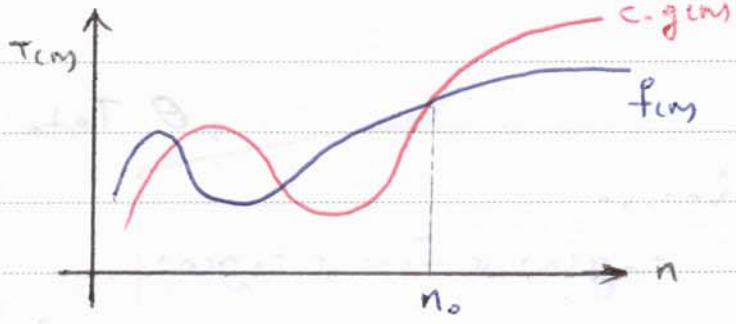
Subject:
Year. Month. Date. ( )

«توابع رشد»

در این قسمت می خواهیم ببینیم یک الگوریتم با تابع زمان مشخص برحسب n، زمانی که n به سمت بی نهایت می رود، از کدام تابع چگونه است.
به توابع رشد مجانبی تابع را بررسی می کنیم (asymptotic)

Big oh O :

$$f(n) = O(g(n)) \iff \exists c>0,\ n_0>0 : \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)$$



✓ برای یک چندجمله‌ای با درجه n یک تابع با درجه n یا درجه بالاتر برای آن چندجمله‌ای است.

$f(n) = O(g(n))$

$$\lim_{n \to \infty} f(n)/g(n) = \text{const}$$

(که عددی ثابت و غیر صفر باشد)

$f(n) = 2n^2 + n + 3$

$f(n) = O(n^2) = O(n^2 \log n)$

$= O(n^3) \neq O(n \log n)$

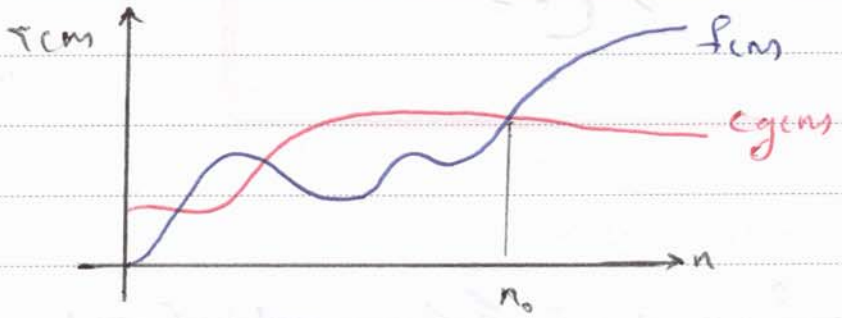
Subject:

Year. Month. Date. ( )

$f(n) = \Omega(g(n)) \iff \exists c>0, n_0>0$

$\forall n \geq n_0 : f(n) \geq c g(n)$

: Ω Omega



✓ برای یک چندجمله‌ای با درجه n همه تابعی با درجه کمتر می‌تواند Ω آن چندجمله‌ای باشد.

$\lim_{n\to\infty} f(n)/g(n) = \infty$ یا const (مثبت ضرب نشود)

$f(n) = 2n^2 + n + 3$

$f(n) = \Omega(n^2) = \Omega(n \log n)$

$\neq \Omega(n^3)$

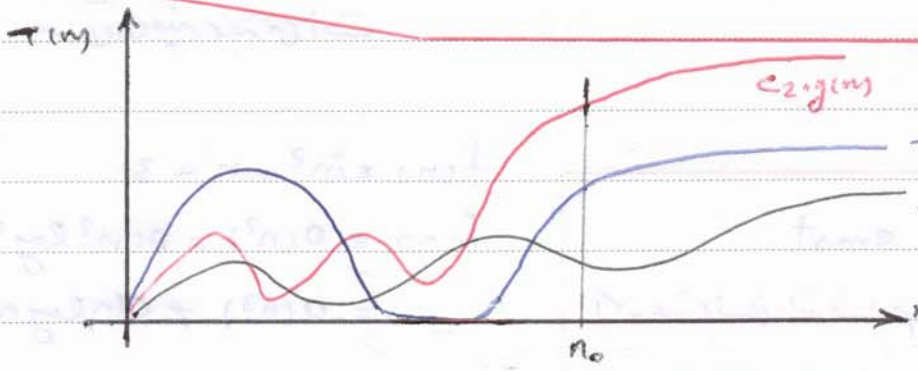
✓ Ω می‌گوید رشد تابع از یک حدی بدتر نمی‌شود دیگه!

✓ O می‌گوید رشد تابع از یک حدی بهتر نمی‌شود دیگه!

: Θ Teta

$f(n) = \Theta(g(n)) \iff \exists c_1, c_2 > 0, n_0 > 0$

$\forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$



$$\lim_{n \to \infty} f(n)/g(n) = \text{Constant} \neq 0 \iff f(n) = \Theta(g(n))$$

o Little oh :

$$f(n) = o(g(n)) \iff \forall c > 0, \exists n_0 > 0 \quad \forall n \geq n_0 : f(n) < c\,g(n)$$

در معادل o می‌گوییم برای هر c بزرگتر از 0 فقط یک n_0 باید وجود داشته باشد.

مثال: $f(n) = 2n^2 + n + 3 \rightsquigarrow f(n) = o(n^2 \log n) \neq o(n^2)$

$$f(n) = o(g(n)) \iff \lim_{n \to \infty} f(n)/g(n) = 0$$

ω Little omega :

$$f(n) = \omega(g(n)) \iff \forall c > 0, \exists n_0 > 0 \quad \forall n \geq n_0 : f(n) > c \cdot g(n)$$

مثال: $f(n) = 2n^2 + n + 3 \rightsquigarrow f(n) = \omega(n \log n) = \omega(n) \neq \omega(n^2)$

$$f(n) = \omega(g(n)) \iff \lim_{n \to \infty} f(n)/g(n) = \infty$$

خواص

ترگذاری: خاصیت ترگذاری برای هر پنج نماد فوق برقرار است.

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

بازتابی: خاصیت بازتابی برای O، Ω و Θ برقرار است.

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

تقارنی: خاصیت تقارنی برای Θ لزوماً برقرار است.
خاصیت تقارنی برای O، Ω و ω لزوماً برقرار نیست.

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

پادتقارنی:

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$$
$$f(n) = o(g(n)) \iff g(n) = \omega(f(n))$$

تشابه با عملگرهای مقایسه ای :

$f(n) = O(g(n)) \quad \rightsquigarrow \quad a \leq b$

$f(n) = \Omega(g(n)) \quad \rightsquigarrow \quad a \geq b$

$f(n) = \Theta(g(n)) \quad \rightsquigarrow \quad a = b$

$f(n) = o(g(n)) \quad \rightsquigarrow \quad a < b$

$f(n) = \omega(g(n)) \quad \rightsquigarrow \quad a > b$

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \iff f(n) = \Theta(g(n))$$

تمرین : 3.1-8 . 3.1-2

روابط ریاضی پرکاربرد :

$x \in \mathbb{R}: \quad x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$

$n \in \mathbb{N}: \quad \lceil n/2 \rceil + \lfloor n/2 \rfloor = n$

$n \in \mathbb{R}, \ a, b \in \mathbb{N}, \ a, b > 0 \quad \lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil$

$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor$

Subject :

Year . Month . Date . ( )

نامساوی رشد :

$n^n > n! > 2^n > n^a > \log n$ برای n های بسیار بزرگ داریم :

$n! = O(n^n)$, $n! = \omega(2^n)$

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

تابع Iteration

$$f^{(i)}(n) \begin{cases} \text{if } i=0 \\ f(f^{(i-1)}(n)) \quad \text{if } i>0 \end{cases}$$

$f(n) = 2n$

$f^{(i)}(n) = 2^i \cdot n$

$f^{(2)}(n) = f(f^{(1)}(n)) = f(f(f^0(n))) = 2^2 n$

تمرین : 3.2-3

Subject:
Year. Month. Date. ( )

«روابط بازگشتی»

سه روش کلی برای تبدیل روابط بازگشتی به روابط بدون بازگشتی وجود دارد.

- روش جایگذاری
- درخت بازگشت
- قضیه اصلی

روش جایگذاری

در این روش ابتدا جواب مسئله را حدس می‌زنیم و فرض کنیم که جواب مسئله را داریم و سپس با استفاده از استقراء پیش می‌رویم.

مثال: $T(1) = c \quad , \quad T(n) = 2T(n/2) + cn \quad ; \quad n > 1$

✓ ابتدا باید یک حدس در مورد جواب بزنیم. اگر امکان پیش‌بینی جواب نبود، می‌توانیم با استفاده از درخت بازگشت جواب را حدس بزنیم.

در مورد مثال قبل حدس می‌زنیم جواب بصورت $n\log n$ باشد! این حدس و فرض به واقع حکم استقراء است که باید ثابت شود.

حدس $\Rightarrow T(n) = O(n\log n)$

حکم استقراء $\Rightarrow T(n) \leq d \cdot n \cdot \log n$

حال با استفاده از روش استقراء قوی حکم را اثبات می‌کنیم یا به عبارتی از فرض به حکم برسیم.

فرض $\rightarrow T(n/2) \leq d \cdot n/2 \cdot \log_2^{n/2}$

فرض $\rightarrow T(n) \leq 2d\, n/2 \log_2^{n/2} + c.n$

$\Rightarrow T(n) \leq dn(\log_2^n - \log_2^2) + cn$

$\Rightarrow T(n) \leq dn\log_2^n + cn - dn$

حکم

برای آنکه رابطه حکم برقرار باشد این عبارت باید یا صفر باشد یا منفی باشد.

$cn - dn \leq 0 \Rightarrow c - d \leq 0 \Rightarrow d \geq c$

بنابراین d پیدا شد که رابطه حکم به ازای آن برقرار باشد.

Subject:
Year. Month. Date. ( )

حال پس از آنکه نشان دادیم یک O خاصی وجود دارد که حکم بازگشتی آن برقرار است، پس باید نشان دهیم که شرایط استقراء هم برقرار است.

مثال:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1$$

باز هم باید ابتدا یک حدس در مورد جواب بدست بیاوریم. بنابراین جواب را بصورت زیر حدس می‌زنیم!

$$T(n) = O(n)$$

حدس $T(n) = O(n) \Rightarrow T(n) \leq cn$ $\longrightarrow$ حکم $(\leq cn+b)$

فرض $\begin{cases} T(\lceil n/2 \rceil) \leq c\lceil n/2 \rceil \\ T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \end{cases}$

$$\Rightarrow T(n) \leq c\lceil n/2 \rceil + c\lfloor n/2 \rfloor + 1$$

$$T(n) \leq cn + 1$$

! که بدلیل هیچ مقداری برای c رابطه حکم $T(n) \leq cn$ برقرار نیست. بنابراین احتمالاً یا در فرض غلط برده است یا حکم را اشتباه از روی فرض نوشتیم.

! اگر در آخرین مرحله از روی مشروحات برحسب راست رابطه علاوه بر حکم عبارت اضافی دیگری وجود داشت، باید نکات زیر را مد نظر بگیریم.

- اگر درجه عبارات اضافی کمتر از درجه عبارت حکم بود، باید درجه حکم جلوی باز درجه کمتر اضافه شود.
- اگر درجه عبارات اضافی مساوی یا درجه عبارت حکم بود، درجه حدس را اشتباه حدس زده‌ایم و باید درجه حدس را بالا ببریم. معمولاً در این حالت یک ثابت کنار c می‌آید به حدس اضافه شود.
- اگر هم درجه عبارت اضافی بیشتر از درجه حکم باشد، درجه حدس را باید بیشتر کنیم.

Subject:
Year. Month. Date. ( )

مثال: $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$

فرض کنیم حدس ما بصورت زیر باشد: $T(n) = O(n)$

حدس $\Rightarrow T(n) \leq cn$ حکم

فرض $\begin{cases} T(\lceil n/2 \rceil) \leq c\lceil n/2 \rceil \\ T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \end{cases} \Rightarrow T(n) \leq c\lceil n/2 \rceil + c\lfloor n/2 \rfloor + n$

$\Rightarrow T(n) \leq \underline{cn} + \underline{n}$

بنابراین حدس ما اشتباه بوده است، نه نظر می‌رسد که با یک فاکتور لگاریتمی به حدس اضافه کنیم.

حدس $T(n) = O(n\log n) \Rightarrow T(n) \leq c \cdot n\log n$ حکم

فرض $\begin{cases} T(\lceil n/2 \rceil) \leq c\lceil n/2 \rceil \log\lceil n/2 \rceil \\ T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor \end{cases}$

فرض $\Rightarrow T(n) \leq c\lceil n/2 \rceil \log^{\lceil n/2 \rceil} + c\lfloor n/2 \rfloor \log^{\lfloor n/2 \rfloor} + n$

$T(n) \leq c\lceil n/2 \rceil \log\lceil n/2 \rceil + c\lfloor n/2 \rfloor \log\lceil n/2 \rceil + n$

22

$T(n) \leq cn\log\lceil n/2 \rceil + n$ (?)

Subject:

Year. Month. Date. ( )

« روش تغییر متغیر »

$$T(n) = 2T(\sqrt{n}) + \log_2 n$$

$n = 2^k \rightsquigarrow T(2^k) = 2T(2^{k/2}) + k$

$S(k) = T(2^k) \rightsquigarrow S(k) = 2S(k/2) + k$

$\Rightarrow S(k) = O(k \log k)$

$\Rightarrow T(2^k) = O(k \log k)$

$\Rightarrow T(n) = O(\log n \times \log^{\log n})$

تمرین: تمرین های 6-4.1, 6-4.1

درخت بازگشت

در این روش درخت بازگشتی مسئله را رسم می‌کنیم. رسم درخت را تا کوچکترین سایز مسئله (معمولا کوچکترین سایز 1 است) ادامه می‌دهیم. هزینه هر مرحله را جمع می‌کنیم.

Subject:

Year. Month. Date. ( )

$T(n) = 3T(n/4) + \Theta(n^2)$:حل

$k=0$; cn^2 n

$k=1$; $3c(n/4)^2$ $n/4$ $n/4$ $n/4$

$k=2$; $9c(n/16)^2$

1 1

تعداد سطرها $= \log_4^n + 1$ ، تعداد برگ‌ها $= 3^{\log_4^n}$

$k = \log_4^n$

انتهای سطر قبل شروع شده

$$T(n) = \sum_{k=0}^{\log_4^n - 1} cn^2 (3/16)^k + c \times 3^{\log_4^n}$$

$$T(n) = cn^2 \frac{1-(3/16)^{\log_4^n}}{1-3/16} + cn^{\log_4^3}$$

$$T(n) \leq \sum_{k=0}^{\infty} cn^2 (3/16)^k + cn^{\log_4^3} = cn^2 \frac{1}{1-3/16} + cn^{\log_4^3}$$

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

24

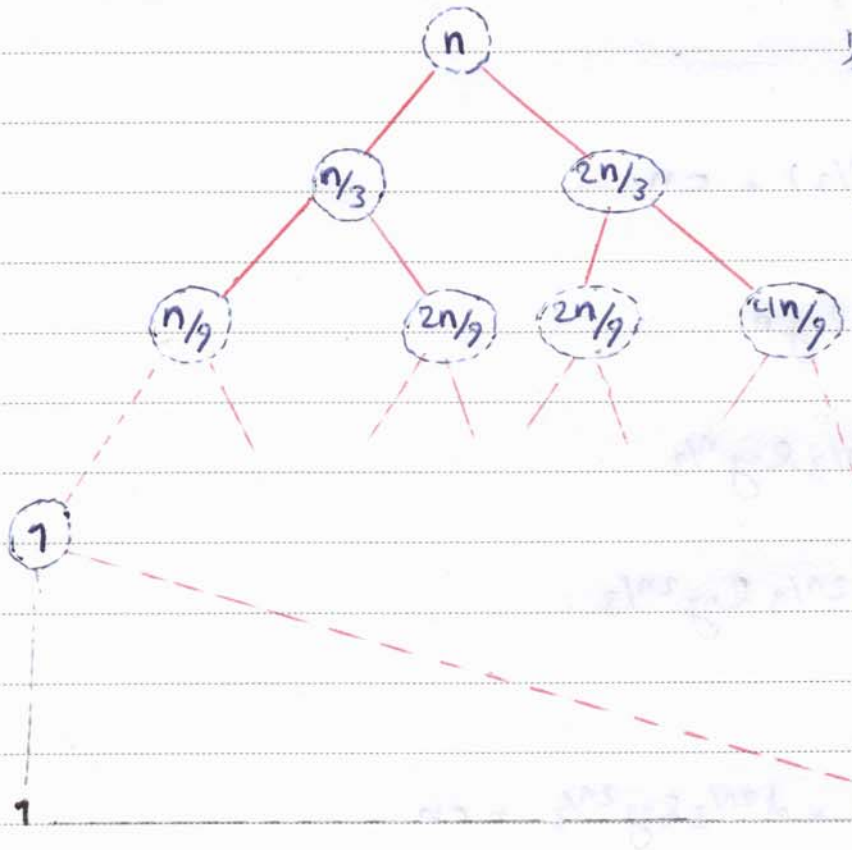
PAPCO

Subject :

Year . Month . Date . ( )

مثال: $T(n) = T(n/3) + T(2n/3) + cn$

الگوریتم فوق در برابر به مسئله‌ای است که یک مسئله با سایز $n/3$ و یک مسئله با سایز $2n/3$ را حل می‌کند و سپس جواب‌ها را با هزینه cn با هم ترکیب می‌کند.



! درخت مسئله فوق یک درخت کامل نیست. زیرا همه‌ی زیرشاخه‌ها زودتر به سایز 1 می‌رسند.

بیشترین به کم‌ترین سطر با سایز 1، هزینه هر سطر cn است.

✓ برای راحتی کار فرض می‌کنیم که درخت مسئله کامل باشد.

$k = \log_{3/2}^{n}$ ، تعداد برگ‌ها $= 2^{\log_{3/2}^{n}} = n^{\log_{3/2}^{2}}$

$T(n) \leq cn \log_{3/2}^{n} + n^{\log_{3/2}^{2}} \times c \longrightarrow T(n) = O(n^{\log_{3/2}^{2}})$

به دلیل آنکه فرض کردیم درخت کامل است، شاید افزایش زیاد برآورد کردیم. اما می‌توانیم حد بالای مناسب‌تری را ارائه دهیم.

Subject:

Year. Month. Date. ( )

✓ حدس می‌زنیم که بهترین زمانی الگوریتم مرتب‌سازی ادغامی $O(n\log n)$ باشد!! پس با استقرا از استقرای مرتبه حدس خود را بررسی می‌کنیم.

! برای اینکه یک حدس از مرتبه زمانی الگوریتم بدست آوریم، می‌توانیم یک بار هم فرض کنیم درست باشد این حدس گره بالایی 1 کامل باشد و سایر گره‌ها را در نظر بگیریم پس یک حد پایین هم بدست آوریم.

$$T(n) = T(n/3) + T(2n/3) + cn$$

حکم $\Rightarrow$ $T(n) \leq dn\log n$

فرض $\Rightarrow$ $\begin{cases} T(n/3) \leq d\,n/3 \log n/3 \\ T(2n/3) \leq d\,2n/3 \log 2n/3 \end{cases}$

$$\Rightarrow T(n) \leq d\,n/3 \log n/3 + d\,2n/3 \log 2n/3 + cn$$

$$T(n) \leq dn/3 \log n - dn/3 \log 3 + 2dn/3 \log 2n - 2dn/3 \log 3 + cn$$

$$T(n) \leq dn\log n - dn\log 3 + 2dn/3 + cn$$

$$\Rightarrow d \geq \frac{c}{\log 3 - 2/3}$$

≤ 0

Subject:

Year. Month. Date. ( )

تمرین: تمرین‌های 4.2-5 ، 4.2-9

روش قضیه اصلی

قضیه اصلی را فقط برای روابطی با فرم زیر می‌توان بکار برد.

$T(n) = a\,T(n/b) + f(n) \quad ; \quad a \geq 1,\ b > 1$

برای این منظور باید دو تابع

$n^{\log_b^a}$? $f(n)$

را با هم مقایسه کنیم که بر این اساس سه حالت پیش می‌آید.

(I) ⟵ if $f(n) = O(n^{\log_b^a - \varepsilon})$ for same $\varepsilon > 0$

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

(II) ⟵ if $f(n) = \Theta(n^{\log_b^a})$

Then $T(n) = \Theta(n^{\log_b^a} \cdot \log n)$

(III) ⟵ if $f(n) = \Omega(n^{\log_b^a + \varepsilon})$ for same $\varepsilon > 0$

and $a f(n/b) \leq c f(n)$ for same $0 < c < 1$

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

Subject:

Year. Month. Date. ( )

مثال:

$$T(n) = 9T(n/3) + n$$

$a = 9 \quad , \quad b = 3 \quad , \quad f(n) = n$

$$n^{\log_b^a} = n^{\log_3^9} = n^2$$

آنرا با n^2 و $f(n)$ را با n^2 مقایسه کنیم که رشد n^2 از $f(n)$ بیشتر است و حالت اول قضیه است. بنابراین باید رابطه اول قضیه اصلی را چک کنیم.

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

$$\Rightarrow 2 - \varepsilon \geq 1 \Rightarrow \varepsilon \leq 1 \quad \rightsquigarrow \quad 0 < \varepsilon \leq 1$$

پس شرایطی برای ε وجود دارد که رابطه اول برقرار است.

$$\Rightarrow T(n) = \Theta(n^{\log_3^9}) = \Theta(n^2)$$

مثال:

$$T(n) = T(2n/3) + 1$$

$a = 1 \quad , \quad b = 3/2 \quad , \quad f(n) = 1$

$$n^{\log_b^a} = n^{\log_{3/2}^1} = n^0 = 1$$

چون $f(n)$، 1 هم درجه هستند بنابراین حالت دوم قضیه اصلی است.

$$f(n) = 1 = \Theta(n^{\log_b^a}) = \Theta(n^{\log_{3/2}^1}) = \Theta(1)$$

$$T(n) = \Theta(f(n) \cdot \log n) = \Theta(\log n)$$

Subject:

Year. Month. Date. ( )

مثال: $T(n) = 3T(n/4) + n\log n$

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

$n^{\log_b^a} = n^{\log_4^3}$

به این حالت رسیدیم $n^{\log_4^3}$ کوچکتر شد؟ $f(n)=n\log n$ است. پس حالت سوم را نمی‌توانیم.

$f(n) = n\log n = \Omega\left(n^{\log_b^a + \varepsilon}\right)$

اگر توان n بیشتر از 1 شود بزرگتر می‌تواند جدای این $n\log n$ باشد.

$\rightsquigarrow \log_4^3 + \varepsilon \geq 1 \Rightarrow \varepsilon \geq 1 - \log_4^3 \approx 0.2$

$af(n/b) \leq c f(n)$

شرط دوم را هم بررسی کنیم.

$\Rightarrow 3(n/4)\log(n/4) \leq cn\log n \Rightarrow \frac{3}{4} \leq c < 1$

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

لذا خواهیم داشت:

مثال: $T(n) = 2T(n/2) + n\log n$

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

$n^{\log_2^2} = 1$

لذا باید حالت سوم را بررسی کنیم.

Subject:

Year. Month. Date. ( )

$$f(n) = n \lg n = \Omega\left(n^{\log_2^2 + \varepsilon}\right) = \Omega(n^{1+\varepsilon})$$

$$\Rightarrow T(n) = \Theta(n \cdot \lg^2 n)$$

$$f(n) = \Theta\left(n^{\log_b^a} \cdot \lg^k n\right)$$

$$\Rightarrow$$

$$T(n) = \Theta\left(n^{\log_b^a} \cdot \lg^{k+1} n\right)$$

تمرین:

تمرین: 4.3-1

4-1

4-2

4-4

Subject :

Year . Month . Date . ( )

« الگوریتم های مرتب سازی »

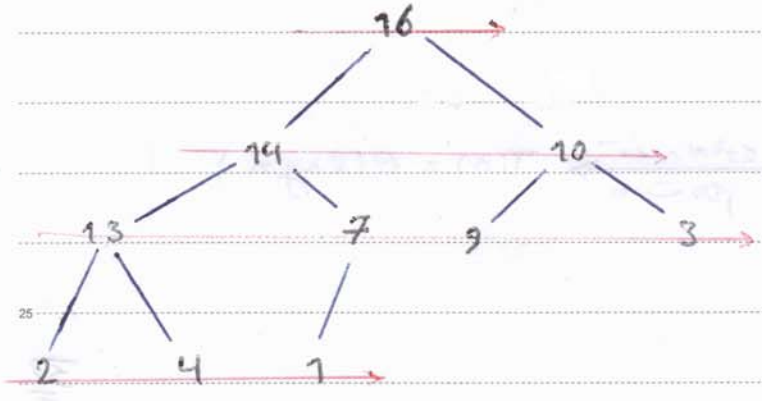
Heap Sort

Heap

- max ← یک درخت کامل که شامل هر گره بزرگتر مساوی با شامل فرزندان است.
- min ← یک درخت کامل که شامل هر گره کوچکتر مساوی با شامل فرزندان است.

✓ Heap را می توان با استفاده از یک ساختمان درخت پیاده سازی کرد.

✓ چون درخت Heap یک درخت کامل است، می توان آن را با استفاده از یک آرایه پیاده سازی نمود و خط به خط سطرهای Heap را در آرایه ذخیره نمود.



	0	1	2	3	4	5	6	7	8	9	10
A		16	14	10	13	7	9	3	2	4	1

parent(i) → $\lfloor i/2 \rfloor$

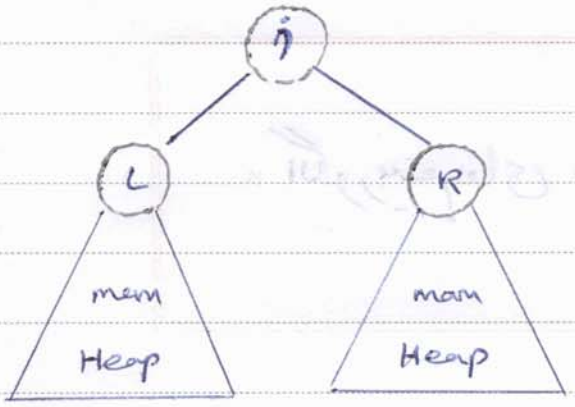
Left(i) → $2i$

Right(i) → $2i+1$

Subject:

Year. Month. Date. ( )

تبدیل به Heap:



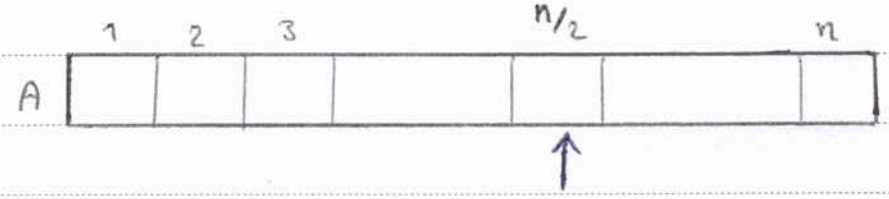

```
Max-Heapify (A, i)
1. l ← Left(i) = 2i
2. r ← Right(i) = 2i+1
3. if l ≤ Heap-Size(A) and A[l] > A[i] Then
4.     largest ← l
5. else largest ← i
6. if r ≤ Heap-Size(A) and A[r] > A[largest] Then
7.     largest ← r
8. if largest ≠ i Then
9.     [exchange A[i] ↔ A[largest]
10.    [Max-Heapify (A, largest)
```

$$T(n) = T(2n/3) + \Theta(1) \xrightarrow[\text{حالت بدتر}]{\text{قضیه اصلی}} T(n) = \Theta(\log n)$$

Subject :

Year . Month . Date . ( )

روش ساختن Heap :

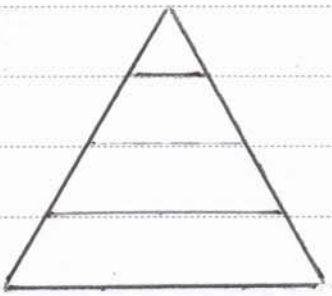


✓ Heap کردن آرایه را از جلوترین پدر شروع می‌کنیم ($n/2$)

✓ عناصر بعد از $n/2$ تشکیل یک Heap تک عنصری هستند.

```
Build-max-Heap(A)
1. HeapSize(A) ← Length(A)
2. for i ← ⌊Length(A)/2⌋ downto 1 do
3.     Max-Heapify(A, i)
```

✓ الگوریتم بالا حداکثر از Order زمانی $O(n\lg n)$ می‌باشد. می‌خواهیم حد بالای کمتری را هم برای الگوریتم بالا پیدا کنیم.



h	تعداد Heapify شدن	تعداد
$\lfloor \lg n \rfloor$		
۰	۱	$n/16$
۱	۲	$n/8$
۲	۳	$n/4$

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{h+1}} \times O(h)$$

$$\Rightarrow T(n) = O\left(n/2 \sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h\right) = O(n)$$

Subject :

Year . Month . Date . ( )

6.5 – 8 تمرین

34

Subject:

Year. Month. Date. ( )

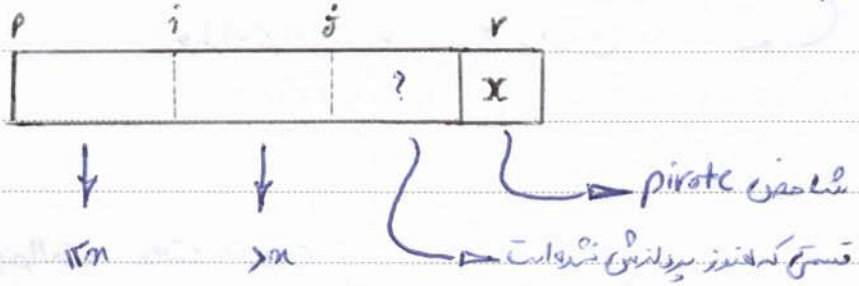
Quick Sort

عمل مرتب سازی را با استفاده از Divide & Conquer انجام میدهد Quick Sort

در این روش مرتب سازی یک عنصر از آرایه را انتخاب میکنیم سپس همه عناصر بزرگتر از آن را به سمت راست آن و همه عناصر کوچکتر از آن را به سمت چپ آن منتقل میکنیم. به همین ترتیب بقیه را Sort میکنیم. هزینه گام ترکیب نیز برابر با صفر خواهد بود زیرا پس از مرتب کردن دو قسمت نیازی به ترکیب نداریم.

✓ برای گام اول (تقسیم) باید آرایه را partition بندی کنیم.

برای این منظور از روش زیر عمل میکنیم.




```
PARTITION ( A, P, r)
1.  x ← A[r]
2.  i ← P-1
3.  for j ← p to r-1 do
4.      if A[j] ≤ x Then
5.          i ← i+1
6.          A[i] ↔ A[j]
7.  A[i+1] ↔ A[r]
8.  return i+1
```

✓ اگر عنصر j ام از شاخص x کوچکتر بود عنصر j ام با عنصر i ام عوض می شود و اشاره گر عنصر i ام یکی جلو می رود.

✓ در پایان عنصر x با عنصر سمت بزرگترها جابجا می شود و i+1 برگردانده می شود.

Subject:

Year. Month. Date. ( )

Quick Sort بصورت زیر عمل می‌کند

```
QUICK_SORT(A, P, r)
1.  if p<r Then
2.      q ← PARTITION(A,P,r)
3.      QUICK-SORT(A, P, q-1)
4.      QUICK-SORT(A, q+1, r)
```

تحلیل زمان اجرا:

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

$\Theta(n)$ ← هزینه پارتیشن کردن

k ← طول یک زیرلیست تقسیم شده است

بدترین حالت: حالت نامتوازن

$k=1$

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

$T(n) = T(n-2) + \Theta(n)$ ، $T(n) = T(n-4) + \Theta(n-2) + \Theta(n) = \Theta(n) + \Theta(n-2) + \cdots + \Theta(n)$

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

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

بهترین حالت: حالت متوازن

$k = n/2$

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

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

در حالت متوسط هم با استفاده از آنالیز ریاضی ثابت می‌شود که زمان اجرای الگوریتم Quick sort برابر با $\Theta(n \lg n)$ است.

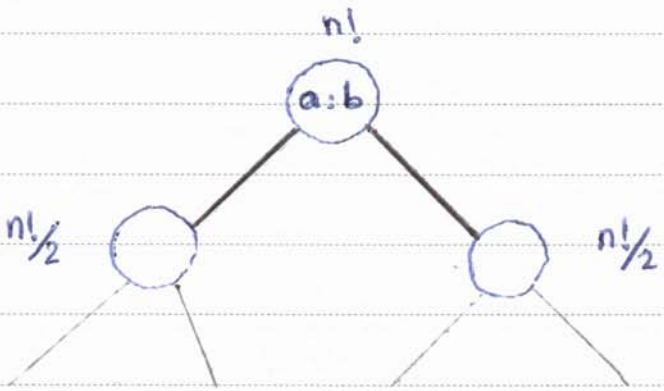
Subject:

Year. Month. Date. ( )

❗ زمان اجرای الگوریتم های مرتب سازی که منحصرا مقایسه استفاده می کنند، هیچگاه بهتر از $n \lg n$ نمی شود.

✓ برای اثبات قضیه فوق از «درخت تصمیم» استفاده می کنیم.

اگر n عدد داشته باشیم به تعداد $n!$ جایگشت مختلف ممکن است که فقط یکی از آنها ترتیب مورد نظر ما است.
در درخت تصمیم براساس اینکه کدام عنصر کوچکتر از دیگری است، تعداد این ترتیب ها را کاهش می دهیم.



درخت فوق یک درخت باینری است که $n!$ برگ دارد.
برای پیدا کردن جواب باید از ریشه شروع کنیم و باطی کردن یک مسیر به برگ مورد نظر برسیم.

$\lg n!$ ⟹ حداقل ارتفاع درخت

$$\lg n! = \Theta(n \lg n)$$

❗ زمان اجرای الگوریتم های مرتب سازی براساس مقایسه $\Omega(n \lg n)$ می باشد.

Subject :

Year . Month . Date . ( )

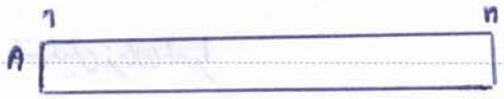
تمرین: 7-3

Subject:
Year. Month. Date. ( )

«مرتب سازی در زمان خطی»

Counting Sort

در Counting Sort فرض بر این است که n عدد صحیح در بازه ۰ تا K داریم.



در این روش یک آرایه کمکی به نام C هم داریم که ابتدا آن را با صفر مقداردهی می‌کنیم. سپس آرایه اصلی را پیمایش می‌کنیم و به ازای هر مقدار در آرایه اصلی، مقدار خانه متناظر با آن در آرایه C را یکی افزایش می‌دهیم.

سپس آرایه C را به صورت تجمعی جمع می‌زنیم تا اندیس شروع هر مقدار در آرایه مرتب شده مشخص شود.

! برای مرتب کردن عناصر آرایه اصلی، آرایه اصلی از آخر به اول پیمایش می‌کنیم و عناصر آن را مرتب می‌کنیم تا عمل Sorting به صورت «Stable» عمل کند.

Subject:

Year. Month. Date. ( )

```
COUNTING-SORT (A, B, k)
1.  for i ← 0 to k do
2.      C[i] ← 0
3.  for i ← 1 to length(A) do
4.      C[A[i]] ← C[A[i]] + 1
5.  for i ← 1 to k do
6.      C[i] ← C[i] + C[i-1]
7.  for i ← length(A) downto 1 do
8.      B[C[A[i]]] ← A[i]
9.      C[A[i]] ← C[A[i]] - 1
```

تحلیل زمان اجرا:

$\Theta(k)$ ⟵ هزینه گام اول، صفر کردن آرایه C

$\Theta(n)$ ⟵ هزینه گام دوم، مقدار دهی

$\Theta(k)$ ⟵ هزینه گام سوم، مقدار دهی

$\Theta(n)$ ⟵ هزینه گام آخر، مقدار چهارم

$\Theta(n+k)$ ⟵ «هزینه کل»

! اگر مقدار عدد k زیاد باشد، الگوریتم فوق چندان خوب نیست.

Radix Sort

در radix sort فرض براین است که n عدد صحیح d رقمی در مبنای k داریم.
در این روش اعداد ابتدا براساس ارقام کم‌ارزش‌تر sort می‌شوند و سپس براساس ارقام با ارزش‌تر sort می‌شوند.

✓ در هر مرحله عمل sorting باید بصورت stable باشد. می‌توان در هر مرحله از روش counting sort استفاده کرد.

"RADIX _ SORT"

```
1. for i ← 1 to d do
2.     use a stable sort for array A on digit i
```

✓ d ممکن است ثابت باشد، مثلاً به n بستگی داشته باشد یا نداشته باشد.

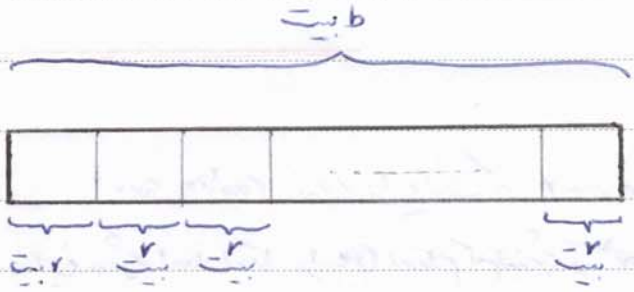
زمان اجرا ⟶ $\Theta(d(n+k))$

اگر تعداد ارقام ثابت باشد $d = const$
اگر مرتب‌سازی خطی باشد ⟶

$$\Longrightarrow T(n) = \Theta(n)$$

تمرین: یک الگوریتم radix sort ارائه کنید که از ارقام پرارزش به کم‌ارزش (از سمت چپ به راست) عمل مرتب‌سازی را انجام دهد.

تاثیر مبنا روی زمان مرتب سازی:



فرض کنیم n عدد با مبنای 2 باشند.

$$d = \lceil b/r \rceil \quad , \quad k = 2^r$$

$$\Theta\left(b/r \,(n + 2^r)\right)$$

در واقع باید رابطه بالا را بهینه کرد.

تمرین: 8.3-4

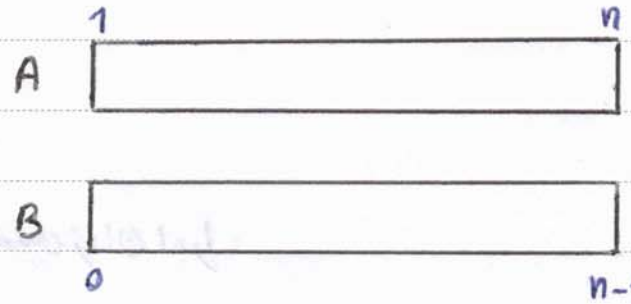
Subject:

Year. Month. Date. ( )

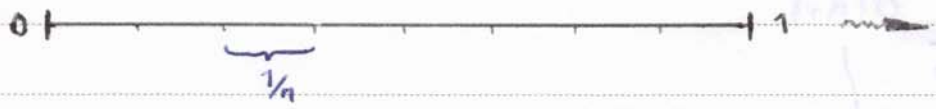
Bucket Sort

در Bucket Sort فرض بر این است که n عدد حقیقی در بازه [0,1] داریم که بطور یکنواخت در این بازه توزیع شده اند.

✓ اگر اعداد در بازه دیگری هم باشند می توان آنها را به بازه [0,1] مقیاس (scale) کرد.



✓ بازه عضو i ام را به n زیر بازه مساوی هر یک با طول $\frac{1}{n}$ تقسیم می کنیم



در این روش مرتب سازی از یک آرایه B استفاده می کنیم که عضوهای آن به یک لیست اشاره می کند.
هر عضو در آرایه اصلی A در یکی از این بازه ها قرار می گیرد. شماره اندیس مربوط به بازه بصورت زیر است.

$\lfloor n \cdot A[i] \rfloor$ ⟵ اندیس باکت $A[i]$

هر عضو به لیست باکت مربوط اضافه می شود. اعداد موجود در هر Bucket کوچکتر از اعداد موجود در Bucket بعدی خود هستند (Partitioning)

سپس هر لیست را با Insertion Sort مرتب می کنیم و نهایتاً لیست ها را بهم متصل می نماییم.

Subject:

Year. Month. Date. ( )

```
"BUCKET_SORT"
1.  n ← length[A]
2.  for i ← 1 to n do
3.      insert A[i] into List B[⌊nA[i]⌋]
4.  for i ← 0 to n-1 do
5.      sort list B[i] with insertion sort
6.  concatenate the lists B[0], B[1], ..., B[n-1]
```

تحلیل زمان اجرا:

n_i ⟵ تعداد اعداد بکت $B[i]$

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

$\Theta(n)$ ← هزینه اعمال لیست‌ها

$O(n_i^2)$ ← هزینه insertion Sort در لیست

$$E[T(n)] = \Theta(n) \quad \longleftarrow \text{"expected running Time"}$$

← امید ریاضی

Subject:

Year. Month. Date. ( )

اثبات:

X_{ij} ⟵ متغیر تصادفی شاخص

اگر عضو رخ دهد 1 در غیر اینصورت صفر

$X_{ij} = I\{A[j] \text{ falls in bucket } i\}$ اگر $A[j]$ در باکت i قرار بیفتد، متغیر تصادفی X_{ij} برابر با 1 خواهد شد.

$i = 0, 1, \ldots, n-1$

$j = 1, 2, \ldots, n$

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

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

متغیر n فقط n_i است، n ثابت است. چون می‌دانیم امید ریاضی بصورت خطی است.

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

ثابت $O()$ notation بصورت یک ضرب ضمنی با ضریب ثابت است که ضریب ثابت هم از E بیرون می‌آید و E داخل $O()$ می‌رود.

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

$$n_i = \sum_{j=1}^{n} X_{ij} \quad \longrightarrow \quad E[n_i^2] = E\left[\sum_{j=1}^{n} X_{ij} \times \sum_{j=1}^{n} X_{ij}\right]$$

Subject:

Year. Month. Date. ( )

$$E[n_i^2] = E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] = E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\sum_{1\le k\le n,\ (k\neq j)} X_{ij}\cdot X_{ik}\right]$$

✓ امید ریاضی متغیر تصادفی برابر است با حاصلجمع حاصلضرب های هر یک از مقادیر متغیر در احتمال وقوع آن مقدار.

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

$$\sum_{j=1}^{n} E[X_{ij}^2] = \sum_{j=1}^{n} 1/n$$ ← یک عدد ثابت Constant

$$E[X_{ij}\cdot X_{ik}] = E[X_{ij}]\cdot E[X_{ik}] = 1/n \times 1/n = 1/n^2$$

✓ متغیرهای مربوط به X_{ij} و X_{ik} متغیرهای مستقل هستند و لذا امید ریاضی حاصلضرب آنها برابر است با حاصلضرب امیدهای ریاضی تک تک آنها.

$$E[n_i^2] = \sum_{j=1}^{n} 1/n + \sum_{1\le k,j\le n}\sum E[X_{ij}\cdot X_{ik}] = 2 - 1/n$$

$$\Rightarrow E[T(n)] = \Theta(n) + \sum_{j=0}^{n-1} O(2-1/n) = \Theta(n) \quad \square$$

Subject :
Year . Month . Date . ( )

« Selection »

می‌خواهیم در بین n عدد، k امین کوچکترین عدد را پیدا کنیم.

راه حل اول : ۱- اعداد را مرتب می‌کنیم ⟵ $\Theta(n\lg n)$

2- k امین عدد را برگردانیم ⟵ $\Theta(1)$

حالات خاص :

k = 1 (min) ⟵ n-1 مقایسه	}	⟵ 2n-2
k = n (Max) ⟵ n-1 مقایسه		

پیدا کردن max , min : از روش تقسیم و غلبه استفاده می‌کنیم. آرایه را نصف می‌کنیم و min , max را در هر قسمت پیدا می‌کنیم سپس ترکیب می‌نماییم.

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

اگر بصورت خطی هم بشماریم max , min را پیدا کنیم،
هزینه باز هم $\Theta(n)$ خواهد شد.

⟶ $T(n) = \Theta(n) = 2n-2$

✓ به عنوان راه حل دیگر می‌توانیم اعداد را به $n/2$ تا دسته‌ی تقسیم کنیم. با $n/2$ تا مقایسه در هر دسته یک عدد min, یک عدد max پیدا می‌شود. سپس min ها را با هم، max ها را با هم مقایسه می‌کنیم تا min, max کل بدست آید.

$$T(n) = 2(n/2 - 1) + n/2 = 3/2\, n - 2$$

مسئله Selection

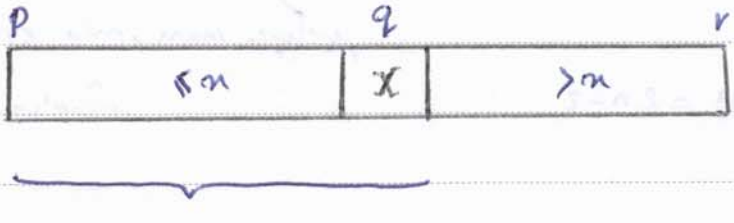
در حالت کلی برای حل مسئله Selection بصورت زیر عمل می‌کنیم.

مانند الگوریتم Quick Sort ابتدا آرایه را partition می‌کنیم. سپس با توجه به شماره‌ی pivote, عدد k حالت زیر پیش می‌آید.

① $q - p + 1 = k$ ⟵ عضو مورد نظر پیدا شده است

② $k < q - p + 1$ ⟵ در نیمه‌ی اول دنبال kامین کوچکترین عنصر می‌گردیم

③ $k > q - p + 1$ ⟵ در نیمه‌ی دوم دنبال kامین کوچکترین عنصر می‌گردیم



Subject:

Year. Month. Date. ( )

! در الگوریتم partition کردن شاخص را بصورت تصادفی انتخاب می‌کنیم ⟵ الگوریتم تصادفی

```
RANDOMIZED-SELECTION(A, p, r, k)
1.  if p = r then
2.      return A[p];
3.  q ← Randomized-partition(A, p, r)
4.  i ← q - p + 1
5.  if k = i then
6.      return A[q]
7.  if k < i then
8.      ret Randomized-Selection(A, p, q-1, k)
9.  else
10     ret Randomized-Selection(A, q+1, r, k-i)
```

✓ اگر با استفاده از امید ریاضی زمان اجرای الگوریتم تصادفی بالا را تحلیل نماییم، مانند حالتی می‌شود که شاخص pivot لزوماً وسط آرایه انتخاب شده باشد

n/2	x	n/2

$$T(n) = T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n)$$

هزینه‌ی پارتیشن کردن

هزینه‌ی یک شاخص انتخاب را نشان می‌دهیم

Subject:

Year. Month. Date. ( )

! اگر بتوانیم یک الگوریتم برای partition کردن ارائه دهیم که همیشه آرایه را بصورت Balance پارتیشن کند، را بصورت زمان اجرای الگوریتم Quick Sort همیشه $O(n\lg n)$ و زمان اجرای الگوریتم تند (Selection) همیشه $\Theta(n)$ خواهد شد.

✓ برای الگوریتم پارتیشن کردن Balance به عنصر «میانه» را پیدا کنیم و به عنوان شاخص pivote در نظر بگیریم.

◎ برای این منظور بصورت زیر عمل می‌نماییم. (الگوریتم Select جدید ترکیب شده با الگوریتم پارتیشن کردن)

1. اعداد را به $\lfloor n/5 \rfloor$ گروه 5 تایی بعلاوه حداکثر یک گروه کمتر از 5 تقسیم می‌کنیم.

2. میانه هر $\lceil n/5 \rceil$ گروه را پیدا می‌کنیم (با استفاده از الگوریتم Insertion Sort)

3. با استفاده از تابع Select و بصورت بازگشتی میانه x ، $\lceil n/5 \rceil$ میانه تند را پیدا می‌کنیم.

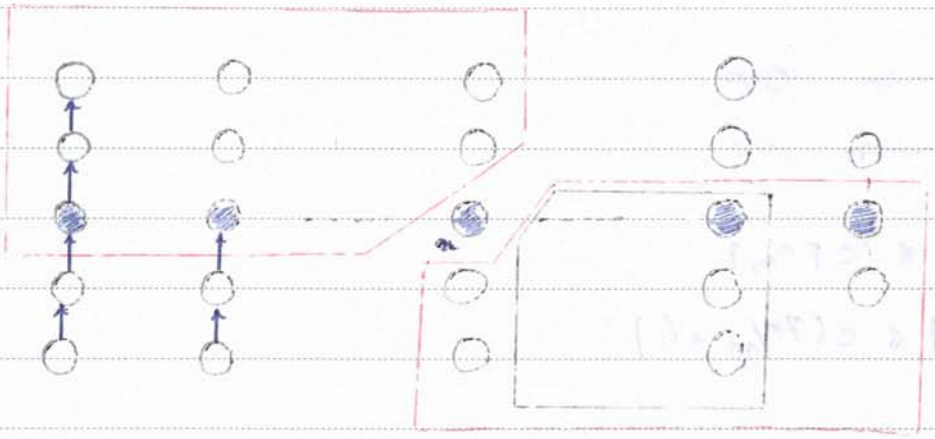
4. با استفاده از x آرایه ورودی را پارتیشن می‌کنیم. (بازاء اینکه x، از این کوچکترین باشد)

✓ میانه میانه‌ها الزاماً میانه کل نیست ولی باعث خوبی آرایه پارتیشن شده با این شاخص متوازن است.

5. اگر $k = i$ آنگاه x را برگردان وگرنه Select را بصورت بازگشتی برای پیدا کردن kامین عدد میانه‌ی آرایه فراخوانی کن، اگر $i > k$ باشد وگرنه الگوریتم Select را بصورت بازگشتی برای پیدا کردن $(i-k)$امین عدد روی نیمه دوم فراخوانی کن.

50

تحلیل:



$\lceil \frac{n}{5} \rceil$ ← «تعداد گروه‌ها»

تعداد اعداد بزرگتر از x حداقل برابر است با:

$$3\left(\left\lceil \frac{1}{2}\lceil n/5 \rceil \right\rceil - 2\right) \geq 3n/10 - 6$$

تعداد اعداد کوچکتر از x حداکثر برابر است با:

$$n - (3n/10 - 6) = 7n/10 + 6$$

زمان اجرا:

1. گام ← $\Theta(\lceil n/5 \rceil)$
2. گام ← $\Theta(\lceil n/5 \rceil)$
3. گام ← $T(\lceil n/5 \rceil)$
4. گام ← $\Theta(n)$
5. گام ← $T(7n/10 + 6)$

$$T(n) \begin{cases} \Theta(1) \\ T(\lceil n/5 \rceil) + T(7n/10+6) + \Theta(n) \end{cases}$$

✓ چون در حالتی که partition ها تقریبا Balanced هستند، می‌توان حدس زد که زمان اجرا، همان زمان اجرای حالت Balanced باشد.
سپس درستی حدس خود را بررسی می‌نماییم.

$T(n) = \Theta(1)$ ⟹ ثابت

$T(n) \leq cn$ ⟹ خطی

فرض $\begin{cases} T(\lceil n/5 \rceil) \leq c\lceil n/5 \rceil \\ T(7n/10 + 6) \leq c(7n/10 + 6) \end{cases}$

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \qquad (\lceil n \rceil < n + 1) \\ &\leq cn/5 + c + 7c/10\, n + 6c + an \\ &= 9c/10\, n + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

$\Longrightarrow \quad -cn/10 + 7c + an \leq 0$

$\Rightarrow \quad n_0 = 140 \;,\; n > 140 \;\rightsquigarrow\; c \geq 20a$

$$\rightsquigarrow \begin{cases} \Theta(1) & n \leq 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + \Theta(n) & n > 140 \end{cases}$$

تمرین 8.4-4 , 9.1-1
9.3-6 , 9.3-8

Subject :

Year . Month . Date . ( )

«روش تقسیم و حل»

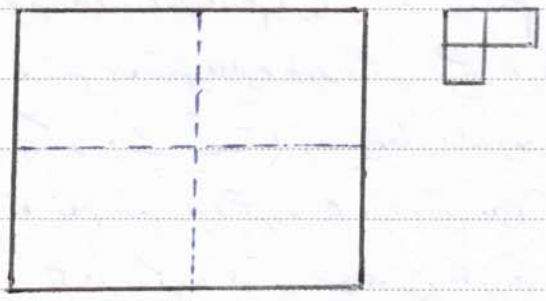
روش های کلاسیک حل مسئله:

- روش تقسیم و حل — Divide & Conquer
- روش برنامه ریزی پویا — Dynamic Pragramming
- روش حریصانه — Greedy
- جستجوی کامل فضای حالت

روش تقسیم و حل:

1- تقسیم به زیر مسائل
2- حل بازگشتی زیر مسائل
3- ترکیب زیر مسائل

مثال: یک سانت مربع شکل داریم می خواهیم آن را با کاشی های شکل زیر بپوشانیم

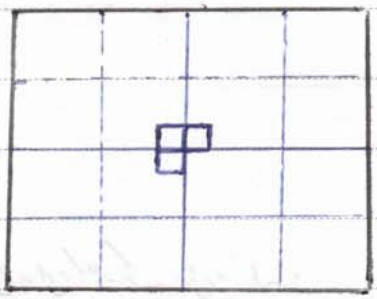


زیر مسائل (Sub problem) باید یک نسخه کوچکتر از همان مسئله اصلی باشند (سایز کوچکتر)

Subject:

Year. Month. Date. ( )

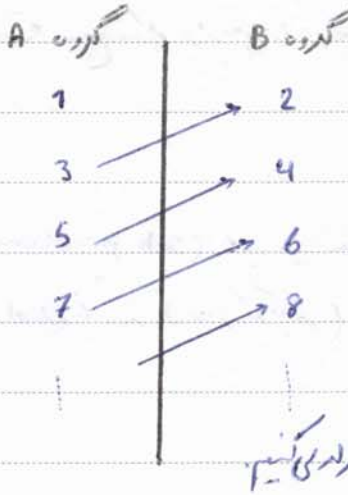
✓ پس از پر شدن کردن تمرین مسئله یک مربع کوچک واحد در تمرین مسئله خالی می‌ماند. بنابراین برای اینکه گام ترکیب به درستی انجام شود، باید در تمرین مسئله به گونه‌ای عمل نماییم که حتی تا آخر الگوریم خالی در تمرین مسائل، در کنار یکدیگر قرار گیرند.



* پروژه: برنامه‌ای بنویسید که مسئله بالا را حل کند (ابعاد مربع بزرگ همیشه 2^k می‌باشد)

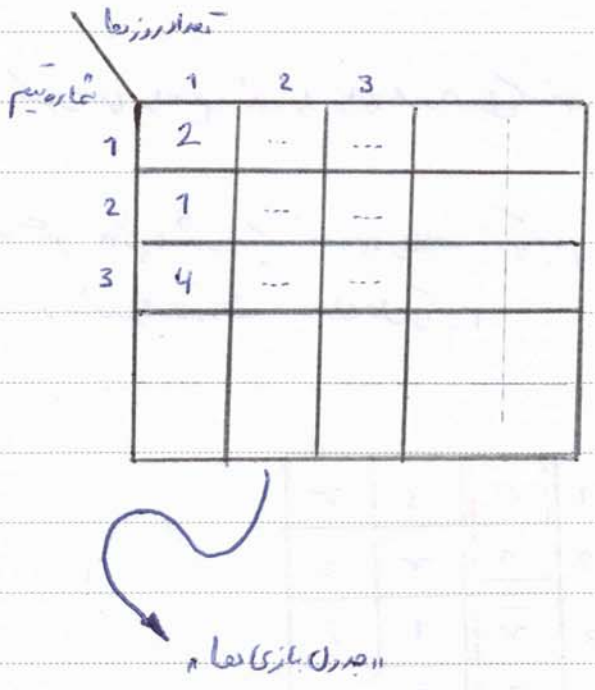
مسئله بازی‌های فوتبال: n تیم فوتبال، دو به دو باید مسابقه دهند

هر تیم در هر روز حداکثر فقط یک بازی می‌تواند انجام دهد.



✓ تیم‌ها را به دو دسته تقسیم می‌کنیم. روز اول تیم‌های شماره فرد با هم بازی می‌کنند. روز دوم هر تیم با تیم بالایی خود در گروه دیگر بازی می‌کند (یک شیفت) و همین طور ادامه می‌دهیم تا تمام تیم‌های گروه A با تمام تیم‌های گروه B بازی کرده باشند. سپس همین الگوریتم را داخل A و B تکرار می‌کنیم.

54



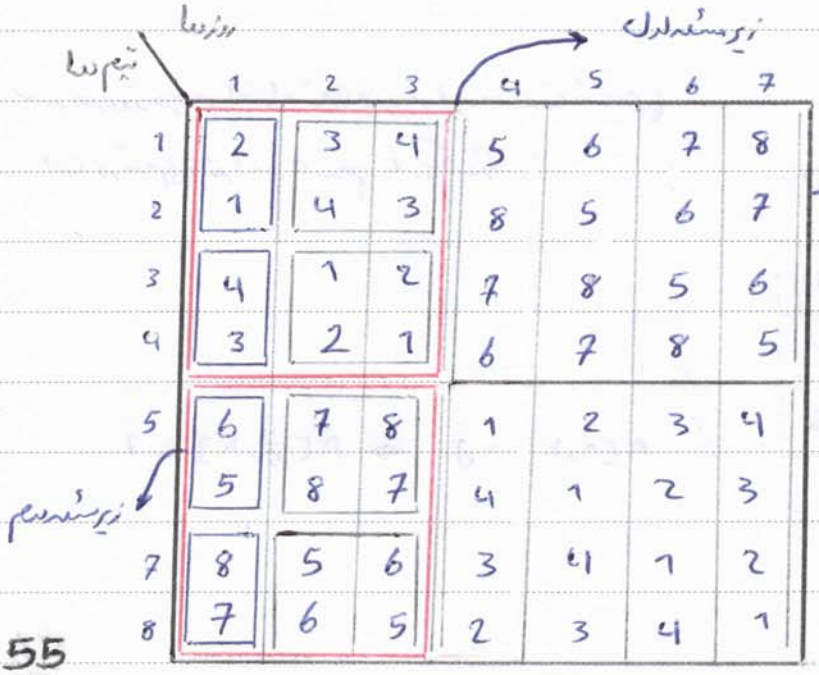
«مربع لاتین»

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

شیفت‌ها را روی مربع لاتین نشان می‌دهیم

❗ اگر n زوج باشد، n تیم، بازی‌ها در $n-1$ روز انجام می‌شود.
اگر n فرد باشد، n تیم، بازی‌ها در n روز انجام می‌شود. (یک تیم بازی اضافه‌ای کنیم و ...)

✔ برای $n=8$ هفته بازی‌های فوتبال را با الگوریتم فوق حل می‌کنیم.



نقش‌های لازم برای ترکیب جواب زیرمسئله اول و زیرمسئله دوم

اکنون می خواهیم مسئله را برای $n=6$ حل کنیم.

✓ اگر n فرد باشد یک تیم بازی اضافه می کنیم و مسئله را بصورت معمول حل می کنیم.

تیم‌ها \ روزها	1	2	3	4	5
1	2	3	-		
2	1	-	3		
3	-	1	2		
4	5	6	-		
5	4	-	6		
6	-	4	5		

1	2	3	7
2	1	7	3
3	7	1	2
7	3	2	1

✓ اگر n زوج ولی $n/2$ فرد باشد، مرجع لاتین بدست نمی آید. در این حالت یک تیم اضافه می کنیم.

✓ در هر روز در تیم استراحت می خورند (در مقابل یک تیم) این دو تیم می توانند با هم بازی کنند.

تیم‌ها \ روزها	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

$A[i,k] = j \rightarrow A[j,k] = i$

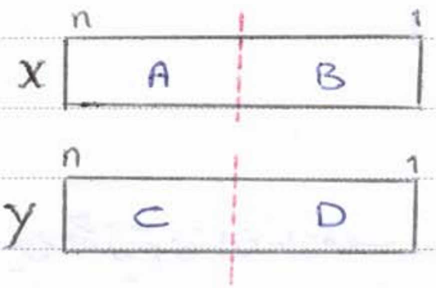
زمان اجرا: $T(n) = T(n/2) + \Theta(n^2)$

هزینه حل یک مربع با ابعاد $n/2$ به همان صورت. مربع دیگر از روی جواب همین بدست می‌آید

هزینه حل مربعی با ابعاد $n/2 \times n/2$ از روی جواب اولی

قضیه اصلی ⟶ $T(n) = \Theta(n^2)$

مثال: مسئله ضرب دو عدد n بیتی



حاصلضرب دو عدد n بیتی، یک عدد 2n بیتی می‌شود.

Sub problem ⟶ ضرب دو عدد $n/2$ بیتی در یکدیگر

$$X * Y = B * D + (AD + CB)\,2^{n/2} + AC \times 2^{n}$$

هزینه ضرب‌ها برای دو عدد n بیتی ⟶ $4 \times$ هزینه ضرب دو عدد $n/2$ بیتی

$M(n) = 4M(n/2)$

$M(1) = 1$

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

هزینه گام تقسیم هم ثابت است

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

$= \Theta(n^2)$

هزینه گام ترکیب

Subject:

Year. Month. Date. ( )

✓ اگر مستقیم مسئله را حل می‌کردیم از order زمانی $\Theta(n^3)$ بود، الگوریتم تقسیم و حل از order زمانی $\Theta(n^2)$ شد. لذا می‌خواهیم این زمان را بهتر کنیم.

✓ علت اینکه order زمانی الگوریتم قبلی بصورت $\Theta(n^3)$ شد به خاطر ضرب 4 سوئه بازگشتی بود.
لذا اگر بتوانیم ضرب کمتری انجام دهیم، زمان الگوریتم بهبود می‌یابد.

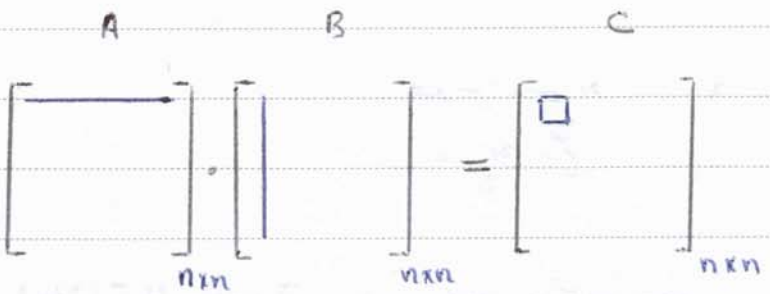
$$AD + BC = (A-B)(D-C) + AC + BD$$

$$T(n) = 3T(n/2) + \Theta(n) = \Theta(n^{\log_2 3})$$

❗ از همین الگوریتم می‌توان برای ضرب در چند جمله‌ای استفاده کرد.

مثال: ضرب ماتریس‌ها

$C = A \times B$



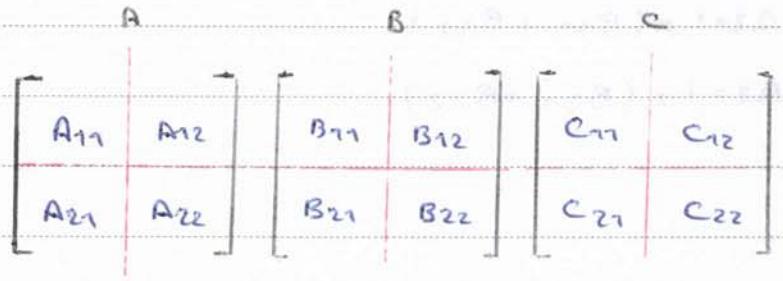
ضرب $\rightarrow n^2 \times n = \Theta(n^3)$

جمع $\rightarrow n \times n = \Theta(n^2)$

Subject:

Year. Month. Date. ( )

✓ برای حل شده بصورت بازگشتی ماتریس را به چهار قسمت تقسیم می‌کنیم.



Sub Problem ⟶ ضرب دو ماتریس با ابعاد $n/2 \times n/2$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

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

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

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

$$T(n) = 8\,T(n/2) + \Theta(n^2) \longrightarrow T(n) = \Theta(n^3)$$

هزینه گام ترکیب (جمع دو ماتریس نها)

✓ در اینجا نیز می‌خواهیم با استفاده از تکنیک بازگشتی زمان بهتری بدست آوریم.

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

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

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

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

$$Q_5 = (A_{11} + A_{12}) \times B_{22}$$
$$Q_6 = (-A_{11} + A_{21}) \times (B_{11} + B_{12})$$
$$Q_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

تعداد ضرب‌ها ⟵ 7

$$C_{11} = Q_1 + Q_4 - Q_5 + Q_7$$
$$C_{12} = Q_2 + Q_4$$
$$C_{21} = Q_3 + Q_5$$
$$C_{22} = Q_1 + Q_3 - Q_2 + Q_6$$

$$T(n) = 7T(n/2) + \Theta(n^2)$$

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

Strassen

Subject:

Year. Month. Date. ( )

«Dynamic Programming»

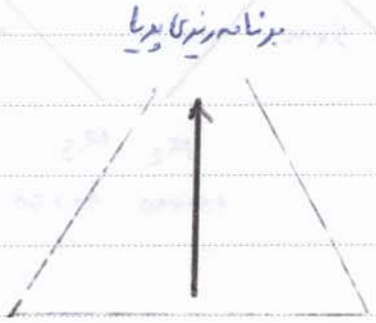
خصوصیات مسائلی که با برنامه ریزی پویا حل می‌شوند:

1- مسئله بهینه سازی باشد (Minimum & Maximum)

2- جواب بهینه مسئله باید از جواب های بهینه زیرمسائل بدست آید (optimal Substructure)

3- درصدی که مسئله بصورت تقسیم و حل، حل گردد و زیرمسائل تکراری زیادی داشته باشد، این روش بهتر است

روش تقسیم و حل در مسئله از بالا به پایین باز می‌شود و مسئله از بالا به پایین حل می‌شود. در این روش زیر مسائل تکراری دوبار، حل می‌شوند.



در روش مسئله از پایین به بالا حل می‌شود و زیرمسائل تکراری فقط یکبار حل می‌شوند. (به همان دلیل برای نگهداری جواب زیرمسائل در نظر گرفته می‌شود)

Subject:

Year. Month. Date. ( )

مثال: می خواهیم n ماتریس را به هم ضرب کنیم.

$$M_1 \times M_2 \times M_3 \times \cdots\cdots \times M_n$$
$$d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4 \quad d_n \times d_{n+1}$$

می خواهیم چگونه ای پرانتز گذاری ها را انجام دهیم که تعداد ضرب ها کمینه باشد.

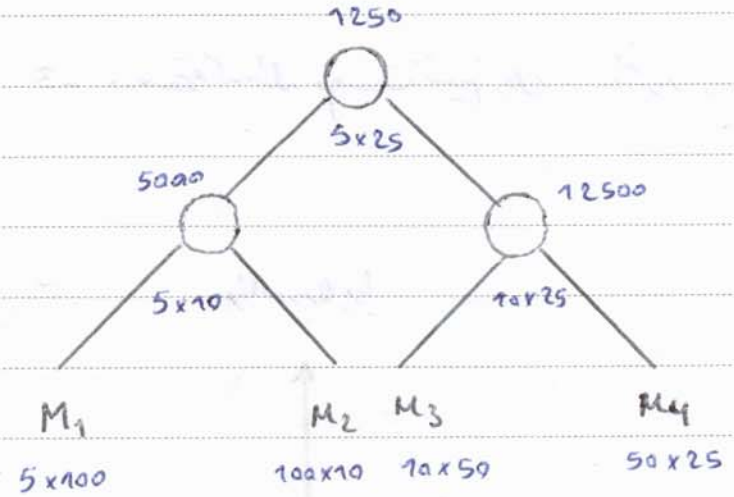
$$M_1 \cdot M_2 = M$$
$$n \times m \quad m \times p \quad n \times p$$

فرض کنیم می خواهیم چهار ماتریس به صورت زیر را به هم ضرب کنیم.

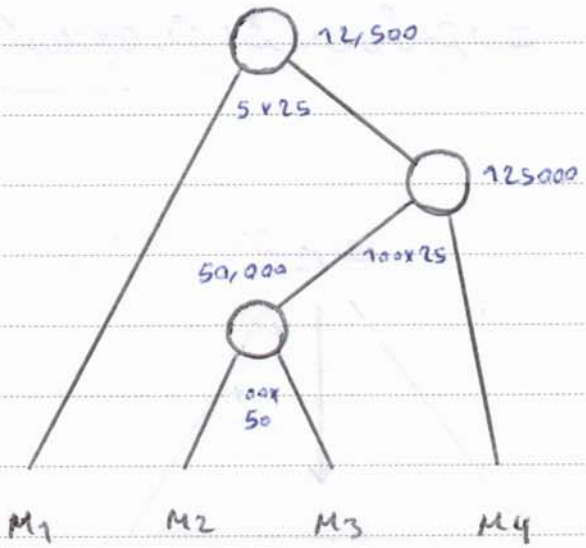
تعداد ضرب ها $= nmp$

$$M_1 \times M_2 \times M_3 \times M_4$$
$$5 \times 100 \quad 100 \times 10 \quad 10 \times 50 \quad 50 \times 25$$

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



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



تعداد ضرب ها = 18,750

تعداد ضرب ها = 187,500

Subject:

Year. Month. Date. ( )

✓ اکنون می خواهیم مسئله فوق را به روش بهینه سازی پویا حل نماییم.
بنابراین ابتدا باید شرایط مسائل قابل حل با برنامه ریزی پویا را بررسی نماییم

← مسئله فوق که یک مسئله بهینه سازی است.

← برای اینکه ببینیم مسئله بصورت optimal Substruct می باشد یا نه، فرض می کنیم مسئله بصورت بهینه حل شده است و ضرب آخر، ضرب بین M_k و M_{k+1} است.

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

تعداد ضرب های M_n تا M_1 (تعداد کل ضرب ها) = تعداد ضرب های M_k تا M_1 + تعداد ضرب های M_n تا M_{k+1} + $d_1 \times d_{k+1} \times d_{n+1}$

! بهترین قسمت در برنامه ریزی پویا پیدا کردن رابطه optimal Substructure می باشد که باید رابطه بین جواب بهینه مسئله و جواب بهینه زیر مسائل را پیدا کنیم. در واقع با پیدا کردن این رابطه مسئله حل شده است.

✓ اگر بخواهیم مسئله فوق را به روش دیگری حل کنیم، باید تمام حالات ممکن را چک کنیم. بدین منظور باید ضرب ها را به دو قسمت تقسیم کنیم و حالت بهینه را برای هر قسمت حساب کنیم. این کار را به ازای تمام حالات ممکن برای دو قسمت کردن انجام دهیم.

$$T(n) = \sum_{i=1}^{n-1} \left(T(i) + T(n-i) \right)$$

$$T(n) = \Theta\left(\frac{1}{n+1} \binom{2n}{n} \right) = \Theta(n^n)$$

«عدد کاتالان» ←

Subject:
Year. Month. Date. ( )

✓ مسائلی که با برنامه‌ریزی پویا حل می‌شوند، دارای خاصیتی هستند که به خاطر آن این روش overlap زیادی دارد. لذا می‌توانیم به جای اینکه مسئله را بصورت تقسیم و حل، حل کنیم و سائل تکراری زیادی را حل کنیم، از پایین شروع کنیم و روش را بسازیم.

(1) $M_1 \times (M_2 \times M_3 \times M_4 \times M_5) \longrightarrow M_2 \times (M_3 \times M_4 \times M_5)$

(2) $(M_1 \times M_2) \times (M_3 \times M_4 \times M_5)$

← گام اول: مسئله را باید بصورت General تعریف کنیم.

مسئله $M_{ij} = M_i \times M_{i+1} \times \cdots \times M_j$
$d_{i-1} d_i \quad d_i d_{i+1} \quad d_{j-1} d_j$

$C_{ij} \longrightarrow M_{ij}$ هزینه مسئله

$$C_{ij} = C_{i,k} + C_{k+1,j} + d_{i-1} \cdot d_k \cdot d_j$$ ← با فرض اینکه آخرین ضرب بین k و $k+1$ می‌باشد.

$$C_{ij} = \text{Min} \{ C_{i,k} + C_{k+1,j} + d_{i-1} \cdot d_k \cdot d_j \}$$

هزینه را برای کوچکترین زیرمسئله یعنی یک تا یک در نظر می‌گیریم.

$C_{ii} = 0$

ضرب یک ماتریس در یکدیگر هیچ هزینه‌ای ندارد.

64

PAPCO

Subject:

Year. Month. Date. ( )

✔ می خواهیم مثال زیر را با روش فوق حل کنیم.

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

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

	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

*

ناتمان داده شده نیاز برای نگهداری جداب زیرمسائل

یک آرایه دوبعدی است که c_{ij}ها را نگهداری

می‌کند.

$$c_{1,2} = \min\{c_{1,k} , c_{k+1,2} , d_0 d_k d_2\}$$

$$1 \le k < 2 \Rightarrow k = 1$$

$$c_{1,2} = c_{1,1} + c_{2,2} + d_0 d_1 d_2 = 5000$$

$$\rightarrow c_{2,3} = c_{2,2} + c_{3,3} + d_1 d_2 d_3 = 50000 \qquad (k=2)$$

$$\rightarrow c_{3,4} = c_{3,3} + c_{4,4} + d_2 d_3 d_4 = 12500 \qquad (k=3)$$

$$\rightarrow c_{1,3} = \min\{c_{1,k} + c_{k+1,3} + d_0 d_k d_3\}$$

$$c_{1,3} = \begin{cases} k=1 \rightarrow c_{1,1} + c_{2,3} + d_0 d_1 d_3 = 75000 \\ k=2 \rightarrow c_{1,2} + c_{3,3} + d_0 d_2 d_3 = 7500 \rightarrow \text{Min} \end{cases}$$

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

$$c_{2,4} = \begin{cases} k=2 \rightarrow c_{2,2} + c_{3,4} + d_1 d_2 d_4 = 37500 \rightarrow \text{Min} \\ k=3 \rightarrow c_{2,5} + c_{4,4} + d_1 d_3 d_4 = 175000 \end{cases}$$

$$c_{1,4} = \min \{ c_{1,k} + c_{k+1,4} + d_0 d_k d_4 \}, \quad 1 \le k < 4$$

$$c_{1,4} = \begin{cases} k=1 \rightarrow c_{1,1} + c_{2,4} + d_0 d_1 d_4 = 50000 \\ k=2 \rightarrow c_{1,2} + c_{3,4} + d_0 d_2 d_4 = 18750 \\ k=3 \rightarrow c_{1,3} + c_{4,4} + d_0 d_3 d_4 = 13750 \rightarrow \text{Min} \end{cases}$$

جواب $\rightarrow \left(\left(M_1 \times M_2 \right) \times M_3 \right) \times \left(M_4 \right)$

Subject:

Year. Month. Date. ( )

«LCS»

مثال: می خواهیم در دنباله‌های a و b دنباله‌ای پیدا کنیم که بیشترین طول را هم داشته باشد و بیشترین طول را هم داشته باشد.

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

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

منظور از زیر دنباله تعدادی از عناصر است که ترتیب آنها هم است ولی الزاماً نباید پشت سر هم باشند.

Longest Common Subsequence ⟶ LCS

✓ چون دنبال زیر دنباله‌ای با بیشترین طول هستیم پس مسأله از نوع بهینه سازی است.

✓ برای پیدا کردن Substructure هم ابتدا فرض می‌کنیم مسئله حل شده است. بنابراین فرض می‌کنیم زیر دنباله‌ای بصورت زیر پیدا شده است.

$\vec{c}$: $c_1, c_2, \dots, c_k$

یک راه حل این است که ببینیم اگر $a_n = b_m$ بود، بیاییم دنبال زیر دنباله مشترک بزرگترین $a_1, \dots, a_{n-1}$ و $b_1, \dots, b_{m-1}$ را پیدا کنیم، بعد $a_n = b_m$ را به آن Concate کنیم.

پس حالت کلی حل مسئله را به شکل زیر بیان می‌کنیم.

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

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

اگر $a_i = b_j$ ⟶ $|L_{ij}| = |L_{i-1,j-1}| + 1$

اگر $a_i \neq b_j$ ⟶
- (I) LCS حرف آخر $= a_i \neq b_j$ ⟶ $L_{i,j-1}$
- (II) LCS حرف آخر $= b_j \neq a_i$ ⟶ $L_{i-1,j}$
- (III) LCS حرف آخر $\neq a_i \neq b_j$ ⟶ $L_{i-1,j-1}$

Subject:

Year. Month. Date. ( )

اگر $a_i = a_j \rightarrow |L_{ij}| = |L_{i-1,j-1}| + 1$

اگر $a_i \neq b_j \rightarrow |L_{ij}| = \max\{|L_{i,j-1}|, |L_{i-1,j}|, |L_{i-1,j-1}|\}$

✓ از آنجایی که $|L_{i-1,j-1}| \leq |L_{i-1,j}|$ و $|L_{i-1,j-1}| \leq |L_{i,j-1}|$ بنابراین: $|L_{i-1,j-1}|$

$a_i = b_j \longrightarrow |L_{i,j}| = |L_{i-1,j-1}| + 1$

$a_i \neq b_j \longrightarrow |L_{i,j}| = MAX\{|L_{i-1,j}|, |L_{i,j-1}|\}$

ریز ترین زیر مسئله:

اگر $i = 0$ یا $j = 0$ $\rightarrow$ $|L_{ij}| = 0$

$|L_{n,m}|$ $\rightarrow$ جواب نهایی

$(n+1) \times (m+1)$ $\rightarrow$ تعداد

← اکنون بی خواسیم مثال زیر را به روش تدت حل کنیم.

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

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

68

$\vec{a}$ \ $\vec{b}$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	1	← 1	← 1	← 1
2	0	↑ 1	↙ 1	2	← 2	← 2
3	0	↖ 1	↖ 2	↙ 2	↙ 2	↙ 2
4	0	↖ 1	↖ 2	↙ 2	↙ 2	↙ 2
5	0	↑ 1	↑ 2	↙ 2	↖ 3	← 3
6	0	↑ 1	↑ 2	↙ 2	↑ 3	↖ 4
7	0	↑ 1	↑ 2	↖ 3	↙ 4	↑ 4
8	0	↖ 1	↖ 2	↑ 3	↙ 3	↑ 4

$L_{1,1} = |L_{0,0}| + 1$

چون $a_{11} = b_1$

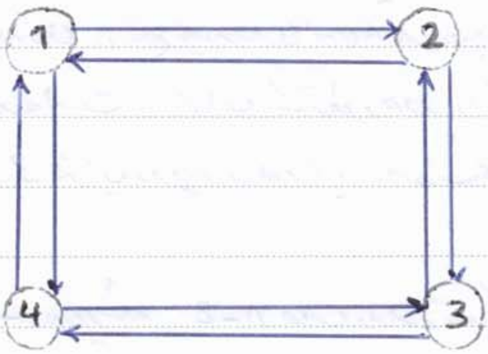
✓ به ازای یک حرف از رشته a می‌گیریم آن را به ترتیب با تمام حروف b تا بیستی می‌کنیم و بین یک حرف بیشتر از a را الی آخر.

✓ در مرحله آخر هم باید جواب را از روی ساختمان داده بنویسیم

$L_{n,m} = 4$ ← طول بزرگترین زیر دنباله مشترک

✓ برای پیدا کردن خود زیر دنباله باید فلش‌ها را دنبال کنیم.
فلش‌هایی که به صورت ↖ هستند، نشون دهنده‌ی قسمت از دنباله مشترک هستند.

مثال: پیداکردن طولانی‌ترین مسیر در یک گراف (Longest Path)

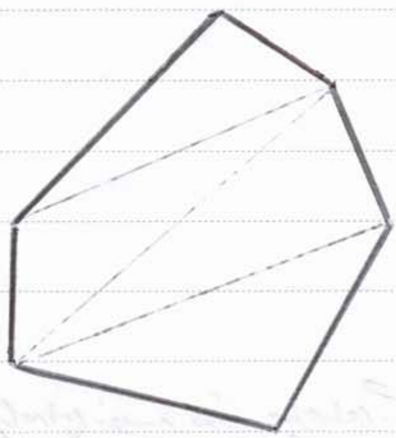


طولانی‌ترین مسیر از ۱ به 2 مسیر ۱ → 4 → 3 → 2 می‌باشد
بنابراین جواب مسئله اصلی از دو جواب بهینه زیرمسئله بدست نمی‌آید و لذا نمی‌توان آن را از روی جواب زیرمسئله به روش برنامه‌سازی پویا حل کرد.

«طولانی‌ترین مسیر ساده از ۱ به 3»

1 → 2 → 3

مثال: مسئله مثلث‌بندی چندضلعی محدب «Trinangulation»



✓ چندضلعی محدب (convex polygon) یک چندضلعی است که زوایای داخلی آن همگی کمتر از 180° می‌باشند.

✓ هزینه مثلث‌بندی، مجموع طول قطرهایی است که در مثلث‌بندی استفاده شده است.

✓ برای حل مسئله فوق به روشی غیر از روش بازگشتی می‌توانیم کل فضای حالات را با استفاده از روش بازگشتی بررسی نماییم. در هر مرحله، چندضلعی محدب را به دو قسمت تقسیم می‌کنیم و دو قسمت را جداگانه مثلث‌بندی می‌کنیم.

Subject:

Year. Month. Date. ( )

$\frac{n(n-3)}{2}$ ⟵ «تعداد کل قطرها»

✓ مسئله فوق از نوع بهینه‌سازی می‌باشد. همچنین برای پیدا کردن optimal substruc فرض می‌کنیم مسئله حل شده است. بنابراین یک قطر وجود دارد که چندضلعی محدب را به دو چندضلعی محدب دیگر تقسیم می‌کند و هر یک از این دو چندضلعی باید به صورت بهینه مثلث‌بندی شده باشند.

! جواب هر زیرمسئله $n-2$ حالت دارد و تعداد کل زیرمسائل نیز n^2 است و بنابراین $O(n^2)$ است.

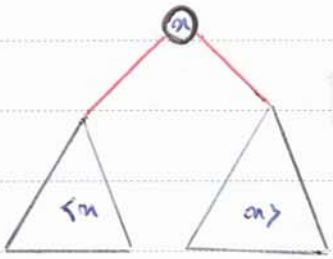
✓ برای حل مسئله فوق لازم دو رأس مجاور از چندضلعی را در نظر می‌گیریم (i و j) یکبار i را ثابت می‌گیریم و از $k = i+2$ تا $j-1$ حالات مختلف را بررسی می‌کنیم و یکبار هم j را ثابت می‌گیریم و از $j-2$ تا $i+1$ حالات مختلف را بررسی می‌کنیم. (برای جلوگیری از حالات تکراری و بدست آوردن حالات تکراری کمتر، تمام اضلاع و قطرهای چندضلعی را بررسی نمی‌کنیم)

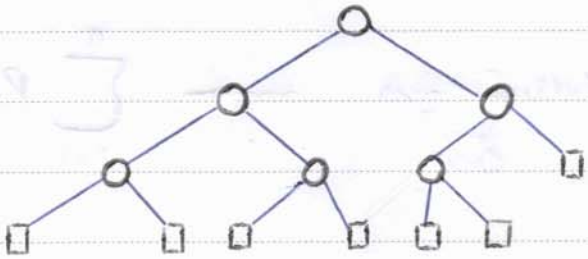
تمرین: برنامه‌ای بنویسید که یک چندضلعی بگیرد و آن را بصورت بهینه مثلث‌بندی کند.

Subject:
Year. Month. Date. ( )

مسئله پیدا کردن درخت جستجوی دودویی بهینه

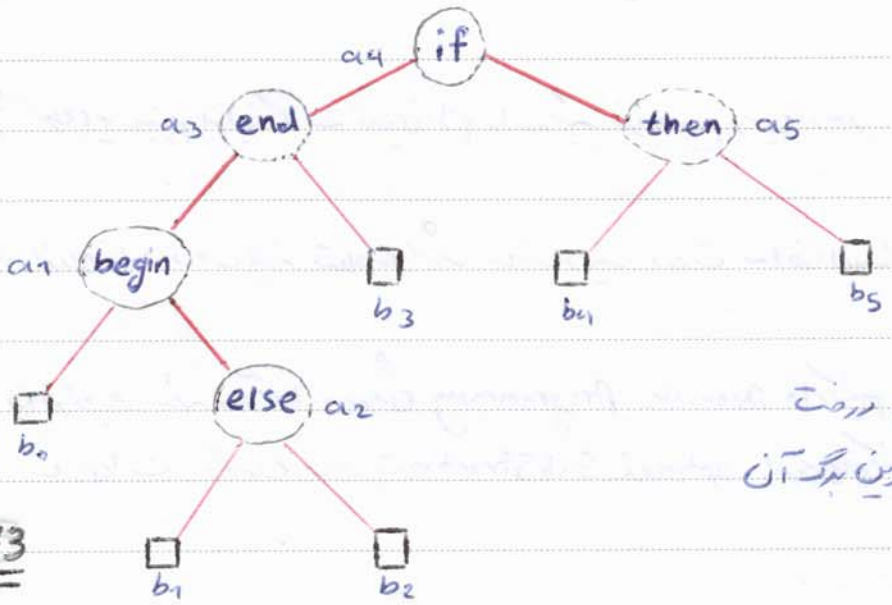
Optimal Binary Search Tree (OBST)





برگ‌ها: null های pointer هستند. جستجو اگر دایره‌ها (Node های دارای داده) خاتمه یابد، جستجو موفق بوده است.
و اگر به برگ‌ها خاتمه یابد، جستجو ناموفق بوده است.

فرض کنیم می‌خواهیم با دستورات if زبان پاسکال کار کنیم. تعداد کلیدهای مثل $a_1 a_2 a_3 \dots a_n$ داریم که می‌دانیم ترتیب Sort آنها بصورت $a_1 < a_2 < a_3 \dots < a_n$ است.



عمق ← فاصله یک Node تا ریشه درخت
ارتفاع ← فاصله یک Node تا عمیق‌ترین برگ آن

جستجوی موفق ← تعداد مقایسه‌ها = عمق گره + 1

جستجوی ناموفق ← تعداد مقایسه‌ها = عمق گره

✓ تعداد کلمات بزرگتر از یک کلمه مشخص یا کوچکتر از آن برای یک ترکیبی تکرار مشخص هستند. (q_i)

هزینه جستجوی های موفق ← $\sum_{i=1}^{n} p_i[depth(a_i)+1]$

هزینه جستجوی های ناموفق ← $\sum_{i=0}^{n} q_i \cdot depth(a_i)$

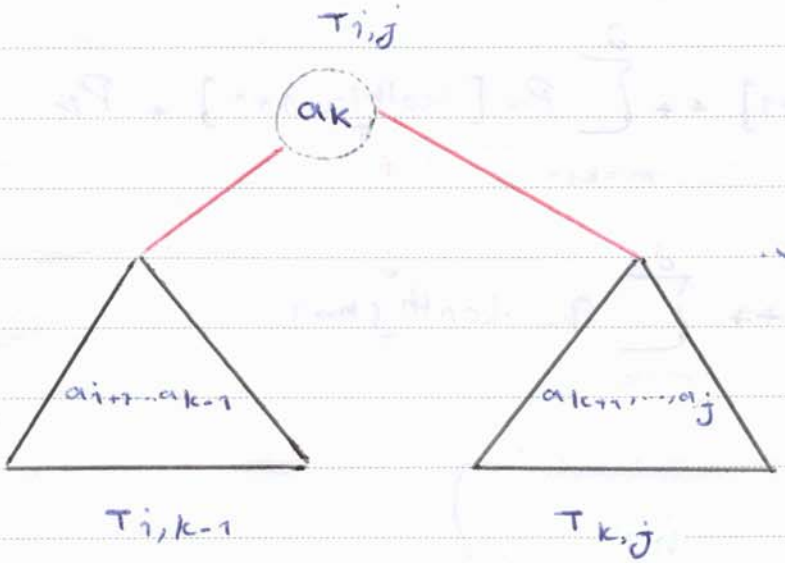
✓ اکنون می‌خواهیم در هزینه بالا را کمینه کنیم.

$$q_0 + \sum_{i=1}^{n}(q_i + p_i) = 1$$ ← «مجموع احتمالات»

! اکنون باید ببینیم برای این کلمات کلیدی کدام را در ریشه قرار دهیم و سپس در هر SubTree باز کدام را ریشه قرار دهیم.

✓ لزومی ندارد درخت بهینه متوازن باشد. ساختار بهینه درخت به احتمال‌ها وابسته است.

✓ می‌خواهیم مسئله فوق را به روش Dynamic Programming حل کنیم. ثابت شده که بطور بدیهی از نوع بهینه سازی است. اکنون خاصیت optimal SubStructure را بررسی می‌کنیم.



← جواب زیر مسائل در یک زیر درخت لزما باید خود پیوسته باشند. در اینصورت تعداد کل جستجوها یکی بیشتر از تعداد جستجوهای زیر درخت هاست.

مسئله T_{ij}:

$P_{i+1} \quad P_{i+2} \quad \dots \quad P_j$

$a_{i+1} < a_{i+2} < \dots < a_j$

$b_i \quad b_{i+1} \quad \dots \quad b_j$

$q_i \quad q_{i+1} \quad \dots \quad q_j$

C_{ij}:

$$C_{ij} = \sum_{m=i+1}^{j} P_m \left[depth(a_m) + 1 \right] + \sum_{m=i}^{j} q_m \cdot depth(b_m)$$

برای هر $i \le m \le j$، a_m در زیردرخت ریشه باشد.

✓ باید رابطه مناسب برای برنامه ریزی پویا (رابطه بین مسئله و زیرمسائل) را پیدا کنیم. با فرض اینکه ریشه a_k باشد، جستجوهای موفق را به دو دسته a_k و زیردرخت سمت راست و زیردرخت سمت چپ تقسیم می‌کنیم و جستجوهای ناموفق را هم به دو دسته زیردرخت سمت راست و زیردرخت سمت چپ تقسیم می‌کنیم. (ریشه فقط جستجوی موفق دارد)

Subject:

Year. Month. Date. ( )

$$C_{ij} = \sum_{m=i+1}^{k-1} P_m \left[depth_T(a_{m+1}) \right] + \sum_{m=k+1}^{j} P_m \left[depth_T(a_{m+1}) \right] + P_k$$

$$+ \sum_{m=i}^{k-1} q_m \cdot depth_T(b_m) + \sum_{m=k}^{j} q_m \cdot depth_T(b_m)$$

زیر مسیر چپ

زیر مسیر راست

$$C_{ij} = \sum_{m=i+1}^{k-1} P_m \cdot \left[depth_{T_1}(a_m) + 1 \right] + \sum_{m=i}^{k-1} q_m \, depth_{T_1}(b_m)$$

$$+ \sum_{m=k}^{j} q_m \cdot depth_{T_2}(b_m) + \sum_{m=k+1}^{j} P_m \left[depth_{T_2}(a_m) + 1 \right]$$

$$+ P_k + \sum_{m=k+1}^{j} P_m + \sum_{m=i+1}^{k-1} P_m$$

$$+ \sum_{m=k}^{j} q_m + \sum_{m=i}^{k-1} q_m$$

Subject:

Year. Month. Date. ( )

$$W_{ij} = q_i + \sum_{m=i+1}^{j} (P_m + q_m)$$

$$C_{ij} = C_{i,k-1} + C_{k,j} + W_{ij}$$

$$\longrightarrow \quad C_{i,j} = \min_{i<k\leq j} \{ C_{i,k-1} + C_{k,j} + W_{ij} \}$$

✓ در واقع هزینه کل، هزینه ثابت متناظر با ریشه (W_{ij}) بعلاوه هزینه زیردرخت های چپ و راست است.

✓ یک تابع (آرایه دو بعدی) برای نگهداری C_{ij} کافیست (ساختمان داده مورد نیاز)

✓ سپس باید کوچکترین زیرشده ممکن را مشخص کنیم که یا یک است یا صفر. $(T_{i-1,i}, T_{i,j})$

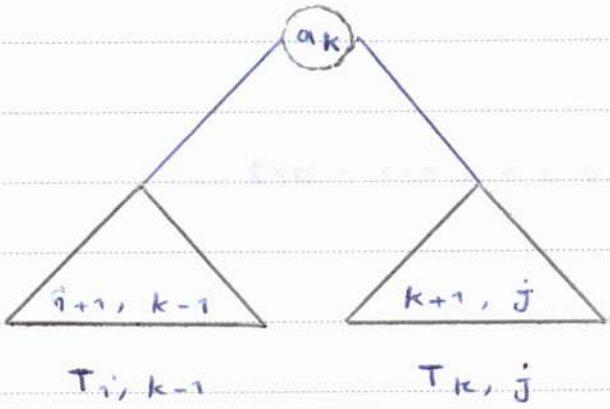
(می توانیم کوچکترین شده را مستقیم نقط $T_{i,i}$ در نظر بگیریم)

$C_{i,i} = 0$ (هزینه بدون ریشه!)

Subject:

Year. Month. Date. ( )

$a_{i+1}, \ldots\ldots, a_j$: OBST لمسألة



$$C_{ij} = \min_{i<k\leq j} \{ C_{i,k-1}, C_{k,j} \} + W_{ij} \quad , \quad C_{ii} = 0$$

مثال :

$a_1 < a_2 < a_3 < a_4 < a_5$

P_i : 0.15 , 0.10, 0.05 , 0.10 , 0.2

q_i : 0.05 , 0.10 , 0.05 , 0.05 , 0.05 , 0.10

	0	1	2	3	4	5
0	0	0.25 K=1				
1		0	0.2 K=2			
2			0			
3				0		
4					0	
						0

« C »

	0	1	2	3	4	5
0	0.05	0.25				
1		0.10	0.2			
2			0.05			
3				0.05		
4					0.05	
5						0.10

« W »

$$C_{0,1} = \min_{0 \le k < 1} \{ C_{0,0} + C_{1,1} \} + w_{0,1} = 0 + 0 + 0/25 = 0/25$$

$$C_{1,2} = C_{1,1} + C_{2,2} + w_{1,2} = 0 + 0 + 0/2 = 0/2$$

$(k=2)$

تمرین: بادانستن اینکه $r_{i,j-1} \le r_{ij} \le r_{i+1,j}$ ثابت کنید که الگوریتم ارائه شده برای OBST بصورت $O(n^2)$ است. (راهنمایی: از تعداد ریشه یابهای k است)

Subject :

Year . Month . Date . ( )

«روش حریصانه»

- در الگوریتم‌های حریصانه ما سعی می‌کنیم در هر گام بهترین قدم را برای رسیدن به جواب برداریم.
- روش برنامه‌سازی حریصانه بسیار شبیه به روش برنامه‌سازی Dynamic است. در روش برنامه‌سازی پویا ما بین چند انتخاب، همگی آن‌ها را بررسی می‌کنیم ولی در روش Greedy سعی می‌کنیم فقط یک انتخاب انجام دهیم که آن انتخاب ما را به بهترین جواب می‌رساند.

✓ اگر انتخاب‌های ما در روش Greedy منجر به بهینه‌ترین جواب شود، الگوریتم حریصانه ما یک الگوریتم حریصانه بهینه است.

✓ روش حریصانه یا Greedy برای حل مسائل بهینه‌سازی بکار می‌رود.
مسئله‌ای که با روش حریصانه حل می‌شود باید خاصیت optimal substructure را نیز داشته باشد
یعنی جواب بهینه مسئله اصلی از جواب بهینه زیرمسائل بدست آید.
در روش حریصانه برخلاف روش برنامه‌سازی پویا که چندین انتخاب ممکن بررسی می‌شوند، فقط یک انتخاب
انجام می‌شود که معمولاً به جواب بهینه منتهی می‌شود. (Greedy choice)

Subject:

Year. Month. Date. ( )

مسئله خردکردن اسکناس با سکه‌ها:

اسکناس : 28

سکه‌ها : 10, 5, 2, 1

می خواهیم یک اسکناس را با کمترین تعداد سکه‌ها خرد کنیم

انتخاب حریصانه ← در هر مرحله سکه‌های درشت‌تر را انتخاب کنیم.

$28 = 2\times 10 + 1\times 5 + 1\times 2 + 1\times 1$

اسکناس : 15

سکه‌ها : 11, 5, 1

روش حریصانه ← $15 = (1)\times 11 + (4)\times 1$

جواب بهینه ← $15 = (3)\times 5$

تمرین: بررسی کنید که الگوریتم‌های حریصانه در چه صورتی جواب بهینه می‌دهند؟؟

Subject:

Year. Month. Date. ( )

مسئله زمانبندی کارها : فرض است که تعدادی کار با زمان اجرایهای متفاوت روی یک پردازنده اجرا شوند. ترتیبی از اجرا را می خواهیم مشخص کنیم که متوسط زمان پاسخ بهینه شود.

t_1 ← زمان اجرای کار اول — t_1 ← زمان پاسخ t_1 (کار اول)

t_2 ← " " دوم " " — t_1+t_2 ← " t_2 (" دوم)

t_3 ← " " سوم " " " — $t_1+t_2+t_3$ ← " t_3 (" سوم)

انتخاب حریصانه ← کارها را براساس ترتیب صعودی زمان اجرایشان انجام می دهیم.

(کاری که زمان کمتری می برد، زودتر اجرا می شود. اگر کاری که زمان بیشتری دارد، اول اجرا شود، در اینصورت در مقام زمان پاسخ همه یک عدد بزرگ تکرار شده است)

✓ برای اینکه اثبات کنیم انتخاب حریصانه ما بهینه است، بصورت زیر عمل می کنیم

(1) زمان بندی : $T_1, T_2, \ldots, \underbrace{T_a, T_b, \ldots}_{\beta}, T_n$

(2) زمان بندی : $T_1, T_2, \ldots, \underbrace{T_b, T_a, \ldots}_{\beta}, T_n$

α

فرض ← $t_a \leq t_b$

زمان پاسخ زمانبندی (1) :

مجموع زمان پاسخ ها $= \alpha + (T_A + t_a) +$

$(T_A + t_a + t_b) + \beta$

Subject:
Year. Month. Date. ( )

زمان پاسخ دهی زمانبندی 2:

$$\text{مجموع زمان پاسخ ها} = \alpha + (T_A + t_b) + (T_A + t_b + t_a) + \beta \quad (2)$$

$t_a < t_b \implies$ مجموع زمان پاسخ های (1) $<$ مجموع زمان پاسخ های (2)

به طور استقرایی نشان داده می شود که اگر در ترتیب یکسان داشته باشیم که قطعه های بیشتری کمتر داشته باشد، در آن ترتیبی که عنصر با زمان کمتر جلوتر آمده است، زمان پاسخ دهی کمتری دارد.

مسئله تالار عروسی: فرض کنید یک تالار عروسی داریم (D_i!) و در هر روز تعدادی پیشنهاد برای اجاره سالن داریم که هر یک برای ساعاتی سالن را می خواهند.
می خواهیم به گونه ای در هر روز تعدادی پیشنهاد را قبول کنیم که بیشترین سود را ببریم.

مسئله انتخاب فعالیت ها:

$\{A_i\}$ ⟵ تعداد فعالیت ها

$A_i = (s_i, f_i)$ می خواهیم به گونه ای A_i را انتخاب کنیم که بازه های آنان تداخل نداشته باشند.

PAPCO

Subject:

Year. Month. Date. ( )

انتخاب حریصانه 1 : فعالیت هایی را که طول کمتری دارند را ابتدا انتخاب کنیم.

« این انتخاب الزاماً جواب بهینه نمی دهد »

انتخاب حریصانه 2 : فعالیت هایی که با بقیه کمترین تداخل را دارند، انتخاب می کنیم.

« باز هم لزوماً بهترین جواب حاصل نمی شود »

✓ انتخاب حریصانه 3 : فعالیتی را انتخاب می کنیم که با فعالیت هایی که تا به حال انتخاب شده، تداخل ندارد و f_i کمترین را دارد.

ابتدا فعالیت با f_i کمینه انتخاب می شود، تمام فعالیت های دارای تداخل حذف می گردند و در بین باقی مانده ها، باز آن فعالیتی انتخاب می شود که f_i کمینه بود و الی آخر

★ **مسئله کوله پشتی** : یک کوله پشتی به وزن M داریم و می خواهیم وسایلی به وزن صحیح M با ارزش های مختلف جمع آوری کنیم. می خواهیم به گونه ای این کوله پشتی را پر کنیم که حداکثر ارزش را جمع آوری کرده باشیم.

نسخه 0-1 : در این نسخه از مسئله کوله پشتی یک وسیله برداشته می شود یا برداشته نمی شود ← « معاده »

نسخه کسری : در این نسخه از مسئله کوله پشتی می توان کسری از شئ را برداشت ← « سفت »

مسئله ساده: الگوریتمی برای حل آن با زمان چندجمله‌ای وجود دارد. (P)

مسئله سخت: الگوریتمی چندجمله‌ای برای آن ارائه نشده است و هنوز هم اثبات نشده چنین الگوریتمی وجود ندارد.

NP_Complete

✓ طبق نظریه پیچیدگی محاسباتی، مسائل به چندین کلاس P، NPC و ... تقسیم می‌شوند.

مسائل NPC در زمان چندجمله‌ای تاکنون قابل حل نیستند. مسائل این کلاس بصورت open هستند و هر یک به دیگری قابل تبدیل است. بدین معنی که با حل یکی از مسائل NPC به پیچیدگی زمان چندجمله‌ای، همه مسائل این کلاس نیز در این زمان حل خواهند شد.

* الگوریتم حریصانه برای حل کوله‌پشتی Fractional:

انتخاب حریصانه ← در هر مرحله واحدهای با ارزش با وزن کمتر را درون کوله‌پشتی قرار می‌دهیم
اشیاء را به ترتیب نزولی $\frac{\text{ارزش}}{\text{وزن}}$ مرتب کرده، در کوله‌پشتی قرار می‌دهیم.

✓ آخرین شیء اگر بصورت کامل در کوله‌پشتی جا نشد، کسری از آن در کوله‌پشتی جای گیرد.

مثال:

$M = 50$

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

اشیاء انتخابی ← 1,2

$10 + 20 = 30$ وزن

$60 + 100 = 160$ ارزش

انتخاب آخر ← $60 + 100 + 2/3\ 120 = 240$

$M = 50$

Subject :

Year . Month . Date . ( )

مسئله کوله پشتی نسخه 0-1 : در این حالت کار به برداشتن کسری از اشیاء نیستیم و فقط می‌توانیم یک شیء را برداریم یا برنداریم.

✓ در این حالت روش حریصانه پاسخی نمی‌دهد. زیرا در آخرین مرحله اگر کوله جا نداشته باشد، نمی‌توان کسری از شیء را برداشت و لذا از فضای باقی مانده کوله پشتی هم نمی‌توان استفاده کرد.

✓ یک روش برای حل مسئله فوق، جستجوی کامل فضای حالت به روش Backtracking می‌باشد.
روش دیگر استفاده از Counter برای حل مسئله است. یک شمارنده n وقتی باینری که معرف آن یک حالت را مشخص می‌کند.

← روش دیگر برای حل مسئله فوق، استفاده از برنامه‌ریزی پویایی باشد.

با اشیاء 1 تا i ام و کوله به پشتی به گنجایش j را در نظر گرفته $P_{i,j}$ مسئله

$$C_{i,j} = \text{Max}\{ V_i + C_{i-1,j-w} \ , \ C_{i-1,j} \} = \text{Max} \begin{cases} \text{شیء } i \text{ ام انتخاب شود} \\ \text{شیء } i \text{ ام انتخاب نشود} \end{cases}$$

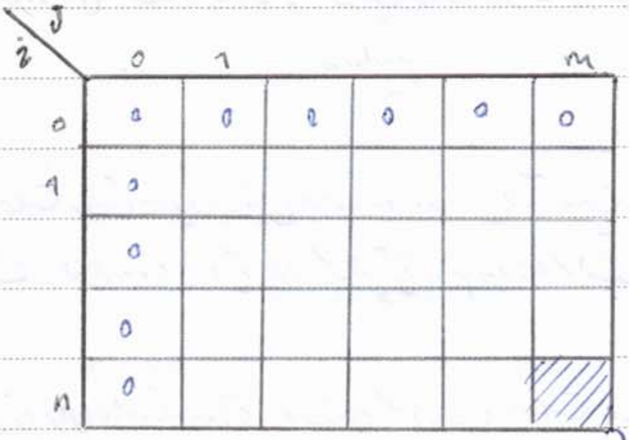
ارزش کالای i ام

برای حل مسئله اصلی در نظر بگیرید با بهترین چینه حل شده باشد، لذا خاصیت Optimal Substruct هم برقرار است.

$C_{0,j} = 0$ ⟵ شیء ای نداریم

$C_{i,0} = 0$ ⟵ کوله جا ندارد

ساختمان داده مورد نیاز یک آرایه دو بعدی است.



حالات ترکیب شدن اشیاء درون کوله پشتی و حالات وزن آن 2^n است. زمان حل مسئله $O(n2^n)$ می‌شود.

اگر وزن اشیاء بصورت صحیح باشد برای وزن کوله پشتی m حالت داریم و ماتریس زیر کافی خواهد بود.

جواب ← $C_{n,m}$

$$T(n) = O(n \times m \times Const)$$

$T(n)$: زمان اجرا — $n \times m$: خانه‌های ماتریس — $Const$: هزینه پر کردن یک خانه

در این حالت زمان اجرا برابر است با زمان پر کردن خانه‌ها

! $O(n \times m)$ را نمی‌توان یک زمان چندجمله‌ای در نظر گرفت زیرا زمان اجرا باید تابعی از اندازه مسئله باشد. در اینجا اندازه مسئله n است و زمان اجرا برحسب m و n چندجمله‌ای است و اگر $M = \Omega(2^n)$ باشد زمان اجرا دیگر چندجمله‌ای از n نخواهد بود.

در این حالت تابع را شبه‌چندجمله‌ای (چندجمله‌ای برحسب n و پارامترهای دیگر) خوانند و الگوریتم ارائه شده هم دارای پیچیدگی زمانی شبه‌چندجمله‌ای خواهد بود.

« Pseudo Polynomial »

Subject:
Year. Month. Date. ( )

«مسئله زمانبندی کارها»

تعدادی کار با زمان اجرای واحد داریم. هر کار دارای یک Deadline است، و مدتی که از تاریخ آن بگذرد هر کار شامل جریمه‌ای می‌شود.

می‌خواهیم کارها را با حداقل جریمه زمان‌بندی کنیم.

$d_1, d_2, \ldots, d_n$ → زمان‌های انقضاء

$\{T_1, T_2, \ldots, T_n\}$

$w_1, w_2, \ldots, w_n$ → جریمه‌ها

✗ **انتخاب حریصانه (1)** ← انتخاب کارهایی با تاریخ Deadline نزدیک‌تر و زودتر.

برای این انتخاب حریصانه مثال نقض وجود دارد. برای مثال چند کار که زمان انقضای نزدیک، جریمه کمی دارند و تعدادی کار که زمان انقضای دیرتر ولی جریمه بسیار سنگین‌تری دارند.

T_1	T_2	T_3	T_4
1	2	3	3
1	2	100	100

ترتیب مثال نقض ← T_1, T_2, T_3, T_4

✓ **انتخاب حریصانه (2)** ← کاری که جریمه بیشتری دارد، الویت داشته باشد

الویت داشتن کار بیشتر به معنی زودتر انجام شدن نیست. ممکن است کاری با جریمه بیشتر دیرتر از زمان Deadline خود انجام شود زیرا اگر مدتی عقب‌تر صبر بنشیند، ممکن است جای کارهای دیگری را بگیرد که Deadline صبرتری دارند.

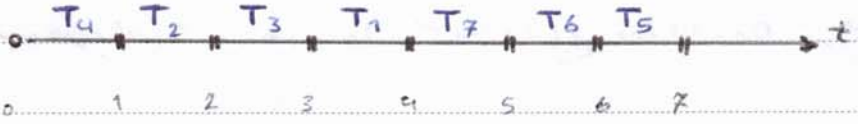
Subject:

Year. Month. Date. ( )

- اگر slot زمانی دقیقاً قبل از سررسید باشد، نزدیکترین slot قبل از آن انتخاب می‌شود.
- اگر به کاری جریمه تعلق بگیرد، در آخرین slot خالی زمانبندی می‌شود.
- اگر هیچ slot خالی قبل از سررسیدش وجود نداشته باشد به اینصورت به کار جریمه تعلق می‌گیرد.

مثال:

«کارها»	1	2	3	4	5	6	7
«d_i»	4	2	4	3	1	4	6
«w_i»	7	6	5	4	3	2	1



فرض کنیم شرط دیگری هم به مسئله اضافه شده است که می‌گوید اگر زمان Deadline یک کار گذشت، به‌ازای هر واحد زمانی که از Deadline آن می‌گذرد به اندازه یک w_i به جریمه آن اندازه خواهد شد. برای حل مسئله در این حالت درست به مانند قبل عمل می‌کنیم، پس اگر قرار شد به کاری جریمه تعلق بگیرد، آن کار در آخرین slot خالی زمانبندی می‌شود.

مسئله زمانبندی کارها حالت‌های مختلفی دارد و در بعضی موارد بسیار پیچیده می‌شود. مثلاً زمان هر کار می‌تواند متفاوت باشد و یا کارهایی چند خط زمانی (چند پردازنده) انجام دارد.

Subject:
Year. Month. Date. ( )

«مسئله کدگذاری Huffman»

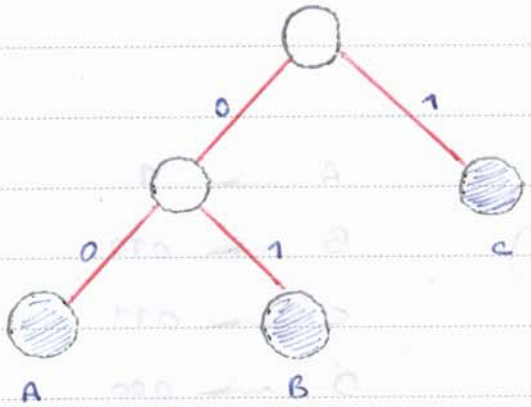
فرض کنید یک فایل متنی داریم که تعدادی حرف در آن آمده است. می‌خواهیم به گونه‌ای به این حروف کد دهیم که حجم فایل کمتر شود. ایده اصلی این است که می‌خواهیم به حروفی با تکرار بیشتر، کدی با تعداد بیت کمتر اختصاص دهیم.

بنابراین ورودی مسئله تعدادی حرف است که هریک دارای یک فرکانس تکرار هستند.

در این کدگذاری هنگام عمل Decode کردن کمی پیچیدگی وجود دارد: در کدگذاری اسکی هر حرف دارای کدی با تعداد ثابت بیت است و لذا Decode کردن آن هم بسیار راحت است ولی در کدگذاری Huffman چون طول تعداد بیت‌های آن متغیر است، تشخیص خود کد سخت است.

✓ برای حل مشکل فوق کدهای ما باید خاصیت پیشوندی داشته باشند. یعنی هیچ کد نباید پیشوندی از کدهای دیگر باشد.

برای پیدا کردن کدهای فوق درخت باینری (درخت Huffman) آن را می‌سازیم.



- برگ‌های درخت فوق حروف ما هستند.
- برای پیدا کردن کد یک حرف مسیری از ریشه به سمت برگ آن حرف را طی می‌کنیم و با توجه به چپ و راست رفتن در هر مرحله، کد هر حرف مشخص می‌شود.

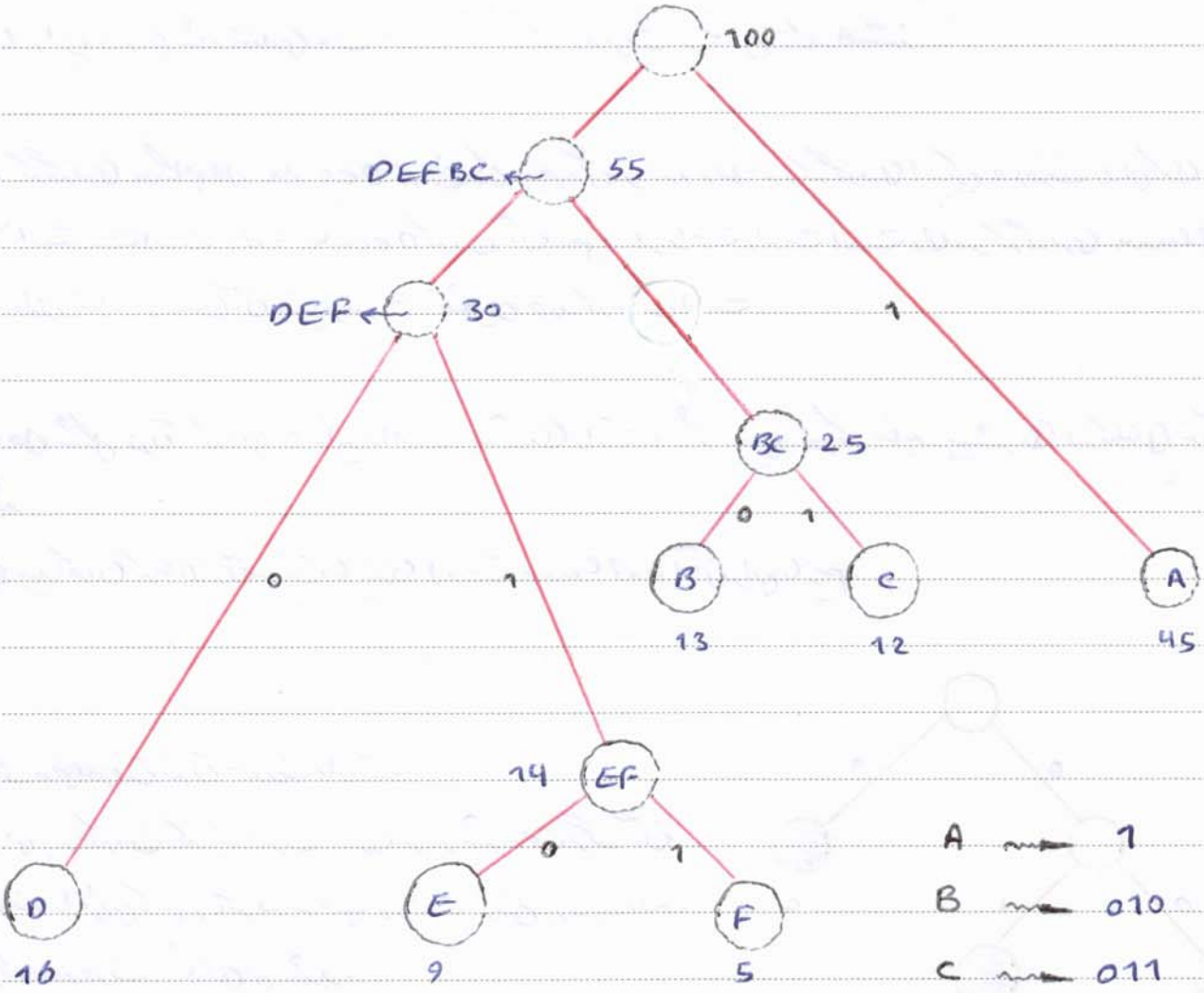
✓ برای ساختن این درخت، از پایین شروع می‌کنیم! به هر حرف یک فرکانس کمتر را یافته و Merge می‌کنیم و یک گره جدید می‌سازیم که فرکانس آن، مجموع فرکانس‌های دو حرف است. این عمل Merge را آنقدر ادامه می‌دهیم تا به ریشه برسیم.

Subject:

Year. Month. Date. ( )

مثال:

حروف	A	B	C	D	E	F
تکرار	45	13	12	16	9	5



A ~ 1

B ~ 010

C ~ 011

D ~ 000

E ~ 0010

F ~ 0011

ASCII بٹ = 8 × 100 = 800

Huffman بٹ = 224

92

Subject:
Year. Month. Date. ( )

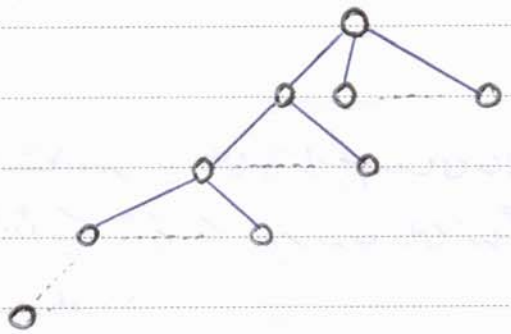
«جستجوی فضای حالات»

روش عقب گرد Backtracking

در روش Backtrack از یک حالت شروع می‌کنیم و گیری انتخاب‌ها انجام می‌دهیم. اگر به جواب نرسیدیم یک مرحله در انتخاب‌ها به عقب برمی‌گردیم و در آن انتخاب به اندازه مرحله بعد، انتخاب دیگری را انجام می‌دهیم

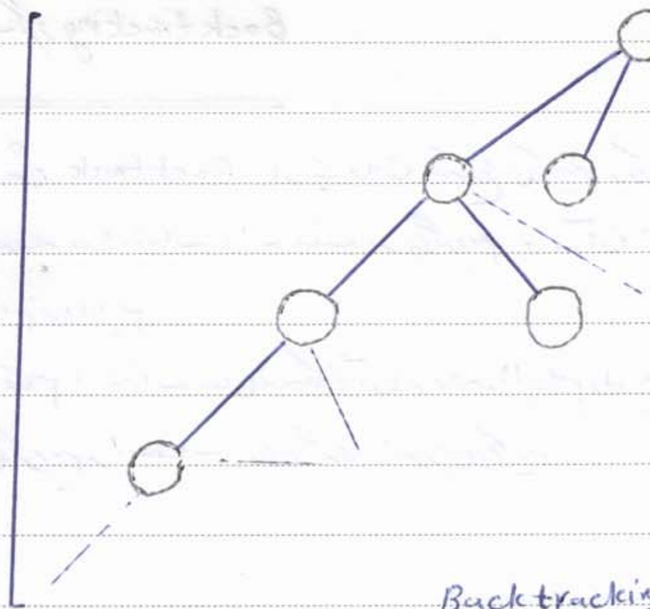
اگر تمام انتخاب‌های ممکن آن مرحله را انجام دادیم و به جواب نرسیدیم، یک مرحله مجدداً به عقب برمی‌گردیم و انتخاب‌های دیگر را بررسی می‌کنیم.

می‌توانیم برای روشن شدن شیوه حرکت را به نظر می‌گیریم که بچه‌های آن انتخاب‌های ممکن هستند.



مثال: مسئله 8 وزیر، یک صفحه شطرنج داریم. می‌خواهیم 8 وزیر را در آن بگونه‌ای قرار دهیم که هیچ دو وزیری یکدیگر را نزنند.

یک راه حل ابتدایی و خام این است که فرض کنیم هر وزیر را می‌توانیم در 64 خانه ممکن قرار دهیم. بنابراین در درخت شده ما، هر Node دارای 64 تا child می‌باشد.



برای حل شدن باید درخت بالا را بصورت Backtracking پیمایش کنیم. یک مسیر را تا رسیدن به برگ می‌رویم. اگر برگ مورد نظر جواب نبود، یک مرحله به بالا باز می‌گردیم و وارد بچه بعدی می‌شویم .....

✓ با استفاده از ایده‌هایی می‌توانیم تعداد حالات را کمتر کنیم
مثلاً وقتی یک وزیر در خانه‌ای قرار گرفت، در سطر و ستون و قطرهای آن نمی‌توان وزیر دیگر قرار داد.

مثال: مسئله پر کردن کوله پشتی: فرض کنید تعدادی شیء با وزن مشخص داریم. می خواهیم یک کوله پشتی را به گونه‌ای با این اشیاء پر کنیم که کوله پشتی حجم کامل پر گردد.

i ⟵ شماره‌ی شیء ، m ⟵ گنجایش کوله پشتی ، n ⟵ تعداد اشیاء

KS(i , m)

```
1.  if m=0 Then                          ✓ Order زمانی تابع الگوریتم برابر
2.      return true                      بهترین حالتی می باشد.
3.  if i>n or m<0 Then
4.      return false                     T(n) = 2T(n-1) + c
5.  if KS(i+1, m - weight[i]) Then       ⟹ T(n) = O(2^n)
6.      print i
7.      return true
8.  else                                 ✓ شمردن را با استفاده از یک شمارنده
9.      return KS(i+1, m)                با تعدی n بیت نیز می توان حل کرد.
```

```
1.  for i=0 to 2^n - 1 do
2.      w=0
3.      b = binary representation of i        ⟵ T(n) = O(n2^n)
4.      for j=1 to n do
5.          if b[j]=1 Then w = w + w[j]
6.      if w=M Then print b
```

Subject:

Year. Month. Date. ( )

Subject:

Year. Month. Date. ( )

« گراف ها »

یک گراف بصورت $G(V,E)$ مشخص می‌شود که V مجموعه رئوس (Vertex, Node) و E مجموعه یال‌ها بصورت زوج مرتب‌های (u,v) می‌باشد.

✓ در گراف‌های جهت‌دار، یال‌ها دارای جهت می‌باشند و در یال (u,v) ، (v,u) متمایز از هم می‌باشند.
در گراف‌های بدون جهت زوج مرتب‌های (u,v) و (v,u) یک یال را نشان می‌دهند.

Directed Graph ⟹ Digraph

✓ درجه (Degree) یک رأس در یک گراف بدون جهت، تعداد یال‌های متصل به آن رأس است.
در گراف‌های جهت‌دار درجه ورودی (indegree) و درجه خروجی (outdegree) تعریف می‌شود.

✓ مسیر (Path) در یک گراف شامل تعدادی رأس است که بطور متوالی با یال‌ها کنار هم متصل‌اند.
مسیر غیرساده، مسیری است که خودش را قطع می‌کند.
سیکل یک مسیری است که ابتدا و انتهای آن بر هم منطبق‌اند. (ساده و غیرساده)

✓ گراف همبند (Connected) گرافی است که بین هر دو رأس دلخواه آن، حداقل یک مسیر وجود داشته باشد.

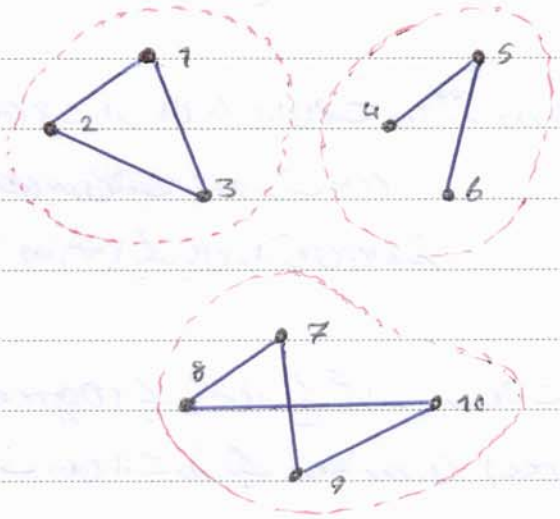
✔ یک گراف جهت دار همبند است، هرچند اگر جهت آن حذف شود همبند باشد.

✔ اجزای همبند (Connected component) : زیرگراف هایی از یک گراف غیرهمبند می باشند که خود به تنهایی همبند می باشند.

✔ زیرگراف (subgraph) : زیرمجموعه ای از یال ها و رئوس گراف اصلی است.

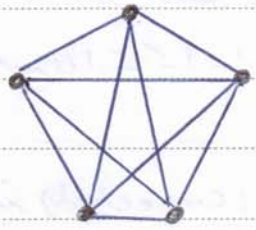
$G(V,E) \longrightarrow G'(V', E')$

$V' \subseteq V$, $E' \subseteq E$



زیرگراف های همبند یک گراف ⟵ غیرهمبند.

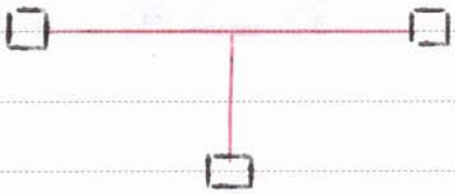
✔ گراف کامل (complete) گرافی است که بین هر دو رأس دلخواه آن یک یال وجود دارد.



K_5

K_n ⟵ یک گراف کامل با n رأس

✓ یک فوق گراف (Hypergraph) گرافی است که هر یال آن می تواند بیش از دو رأس را بهم متصل نماید.



✓ یک درخت، یک گراف همبند بدون دور است (Free tree ← هر Node حداقل هم است)

* قضیه : گزاره های زیر معادل یکدیگر اند.

1- G یک درخت آزاد است.
2- هر دو رأس G توسط یک مسیر ساده واحد بهم متصل اند.
3- G همبند است ولی اگر یک یال برداشته شود، نیز همبندی نمی شود.
4- G همبند است و $|E| = |V| - 1$ می باشد.
5- G حلقه ندارد و $|E| = |V| - 1$ می باشد.
6- G حلقه ندارد ولی اضافه کردن یک یال به G، حلقه ایجاد می کند.

اثبات : می خواهیم نشان دهیم اگر یک گراف، n رأس داشته باشیم و اگر 3 برای آن برقرار باشد، آنگاه 4 هم برای آن برقرار است.

$|V| = n$

گراف را به دو زیر گراف همبند تقسیم می کنیم که با یک یال بهم متصل شده اند.

$G(V,E)$:

③ ⟹ ④

اثبات با استقرا و از استقرا صعودی پذیرد.

طبق فرض استقرا ④ ⟹ ③ برای G_1 و G_2 برقرار است.

فرض می‌کنیم ③ برای G برقرار است، اثبات می‌کنیم که ④ برای G نیز برقرار است.

اثبات می‌کنیم که ③ برای G_1 و G_2 برقرار است. چون G همبند است، پس G_1 و G_2 نیز همبند می‌باشند. چون ③ برای G برقرار است، پس بدیهی است که از G_1 و G_2 بردارهیم، G_1 و G_2 غیر همبند می‌شوند.

③ برای G_1 و G_2 برقرار است و فرض کرده بودیم ④ ⟹ ③ برای G_1 و G_2 برقرار است. بنابراین ④ برای G_1 و G_2 برقرار است.

و خواهیم داشت:

$$|E_1| = |V_1| - 1$$
$$|E_2| = |V_2| - 1$$

$$|E_1| + |E_2| + 1 = |V_1| + |V_2| - 1$$

$$|E| = |V| - 1$$

Subject:
Year. Month. Date. ( )

ساختمان داده‌های گراف

ماتریس همسایگی: در این روش یک ماتریس دوبعدی $n \times n$ که n تعداد رئوس گراف است، در نظر گرفته می‌شود و به ازای یک سطر (یک رأس)، رئوس مجاور با آن (متن‌ها) چک می‌خورند.

لیست همسایگی: در این روش یک آرایه یک بعدی با طول n که n تعداد رئوس گراف است، در نظر می‌گیریم و رئوس مجاور با هر رأس (هر اندیس آرایه) به صورت یک لینک لیست به خانه مربوطه آن در آرایه وصل می‌شود.

- حافظه مورد نیاز برای روش ماتریس همسایگی $\Theta(|V|^2)$ و حافظه مورد نیاز در روش لیست همسایگی $\Theta(|V|+|E|)$ است. بنابراین اگر ماتریس ما sparse باشد، روش لیست همسایگی بهتر است اگر sparse نباشد، روش ماتریس مناسب‌تر است.

- زمان پیدا کردن یک رأس مجاور در روش ماتریس همسایگی $O(1)$ و در روش لیست همسایگی $O(n)$ است ← درجه رأس

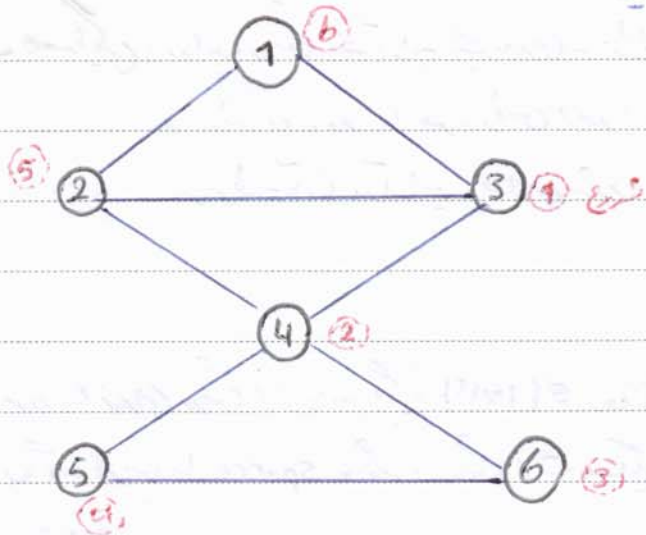
الگوریتم‌های پیمایش گراف

در الگوریتم باید به صورت برای پیمایش گراف مدام جلو برویم.

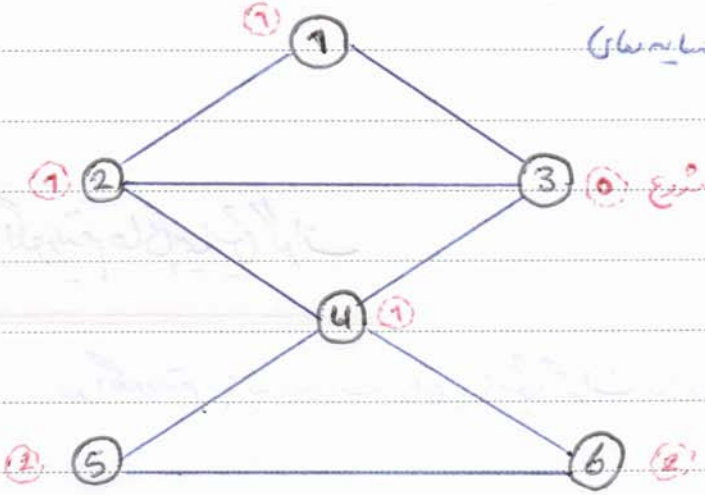
DFS ⟵ Depth First Search

BFS ⟵ Breadth First Search

✔ در روش DFS از یک Node شروع می‌کنیم و از طریق یکی از یال‌هایی که تا به حال visit شده‌اند، به Node دیگری می‌رویم که تا حالا visit شده است. سپس بر روی Node جدید همین کار را تکرار می‌کنیم.
مجددا به یک Node می‌رویم که تا به حال visit شده است.
در نهایت Node که دیگر نتوانستیم به Node جدید برویم، به Node قبلی آن برمی‌گردیم و مجددا" تکرار می‌کنیم.
شرط خاتمه این است که به گره شروع برگردیم و جایی دیگر نتوانیم برویم.



- پیاده‌سازی این روش با استفاده از Stack ممکن است.



✔ در روش BFS از یک Node شروع می‌کنیم و تمام همسایه‌های آن Node را visit می‌کنیم.
سپس مجددا" تمام همسایه‌های Node های visit شده را visit می‌کنیم.

- پیاده‌سازی این روش با استفاده از Queue ممکن است.

کد DFS: ✓

```
DFS(v)
    Mark[v] = VISITED
    foreach w in L[v] do
        if Mark[w] = UNVISITED then
        DFS(w)
```

راس شروع

لیست مجاورت

زمان اجرا $T(n) = \Theta(|V| + |E|)$

کد BFS: ✓

BFS(v)

! درصورتی که گراف ما غیرهسته باشد، الگوریتم BFS یا DFS را باید روی تمام اجزای هسته (زیرگراف ها) اعمال کنیم.

Subject:
Year. Month. Date. ( )

! الگوریتم BFS ، مسیری با کمترین تعداد یال‌ها را به ما می‌دهد. (؟)

کوتاه‌ترین مسیرها در گراف

All Pairs ⟵ پیدا کردن کوتاه‌ترین مسیر بین دو دو گره در گراف $G(V, E)$

Single-Source ⟵ پیدا کردن کوتاه‌ترین مسیر بین یک گره مشخص در گراف $G(V, E)$

گراف وزن‌دار: در گراف‌های وزن‌دار به هر یال اندیسی نسبت داده می‌شود

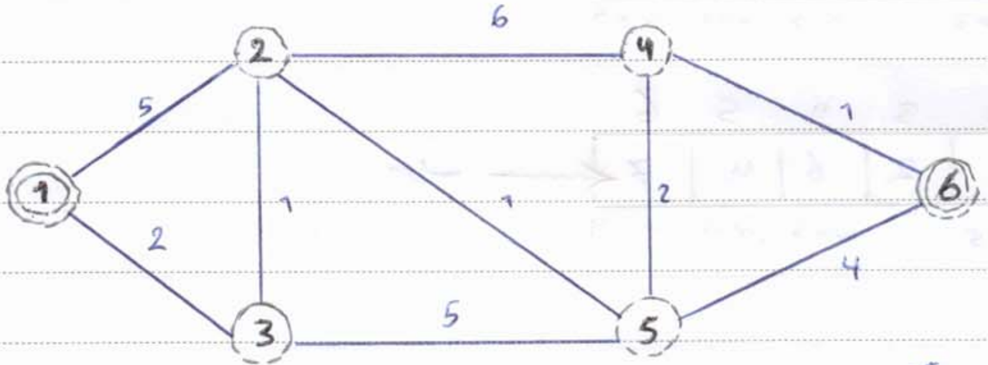
طول مسیر در گراف وزن‌دار: طول مسیر در یک گراف وزن‌دار، به صورت مجموع وزن یال‌های مسیر تعریف می‌شود.

پیدا کردن کوتاه‌ترین مسیر

الگوریتم Dijkstra مربیانه Single Source

الگوریتم Floyd پویا All pairs

: Dijkstra



در ابتدا یک آرایه مثل D در نظر می‌گیریم
و فاصله هر راس تا راس مورد نظر خود را در آن نگهداری می‌کنیم

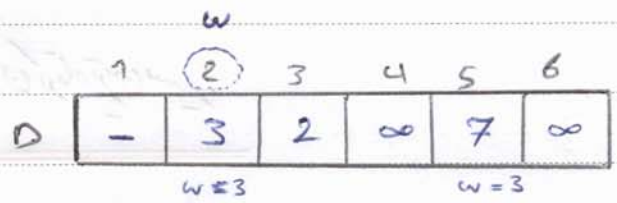
	1	2	3 (w)	4	5	6
D	–	5	2	∞	∞	∞

↑ min (زیر ستون 3) — حالت اولیه

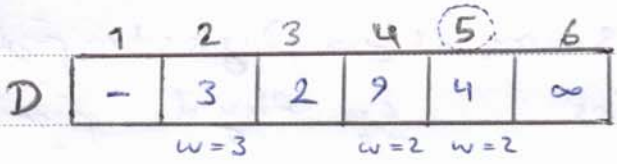
سپس در هر مرحله آرایه بالا را update می‌کنیم. ← کمترین مجاورت (همسایگی)

$$D[v] = Min\{D[v] , D[w] + C[w,v]\}$$

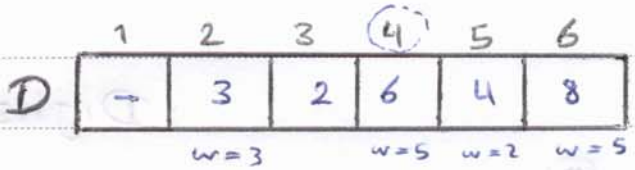
و فقط آنهایی به‌روز می‌شوند که از w یک یال مستقیم به آنها وجود دارد.



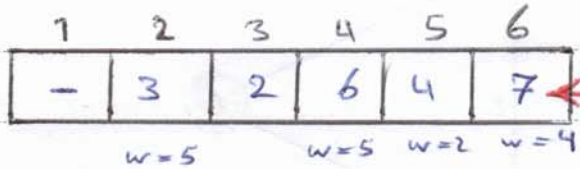
(1)



! min را بین آنهایی انتخاب می‌کنیم که تاحالا انتخاب نکرده‌ایم.



(3)



جواب

(4)

✓ در الگوریتم فرق، فرق یال‌ها مثبت است.

راس شروع

```
Digkstra ( G , s)
1.  S = {1}
2.  for i = 2 to n do
3.      D[i] = C[1, i]
4.  for i = 1 to n-1 do
5.      choose a vertex w in V-S such that D[w] is minimum
6.      Add w to S
7.      foreach vertex v in V-S do
8.          D[v] = min{ D[v], D[w] + C[w,v] }
```

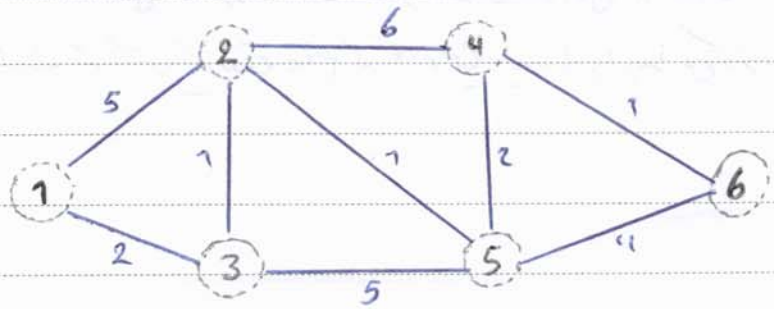
✓ C ماتریس همسایگی رئوس فرق یال‌های گراف است.

تمرین: مثالی که نشان دهد الگوریتم Dijkstra برای یال‌های منفی کار نمی‌کند.

← Order زمانی الگوریتم Dijkstra بصورت فوق $O(n^2)$ می‌باشد.
ولی اگر از Heap به جای آرایه D استفاده کنیم و بدلیل هزینه به روز کردن هزینه همسایه‌های w، همسایه را از Heap خارج و سپس وارد کنیم، در اینصورت Order زمانی الگوریتم فوق $O(|E|\lg|V|)$ خواهد شد.

✓ برای پیدا کردن خود سیستم از یالهایی که قبلا نوشتیم استفاده می‌کنیم.

الگوریتم Flogd



این الگوریتم کوتاه‌ترین مسیر بین تمام رئوس را پیدا می‌کند.

✓ ماتریس C ماتریس همسایگی است.
ماتریس A هم یک کپی از ماتریس C است که آن را update می‌کنیم.

```
«Floyd»
1.  for i=1 to n do
2.      for j=1 to n do
3.          A[i,j] = C[i,j]
4.  for i=1 to n do
5.      A[i,i] = 0
6.  for k=1 to n do
7.      for i=1 to n do
8.          for j=1 to n do
9.              if A[i,k] + A[k,j] < A[i,j] then
10.                 A[i,j] = A[i,k] + A[k,j]
```

کپی آرایه C در A

✓ order زمانی الگوریتم فوق با در ساختمان داده‌ای $O(n^3)$ است. و دلیل این زمان اضافی نسبت به این است که الگوریتم فوق عام‌تر کوتاه‌ترین مسیرها را پیدا می‌کند.

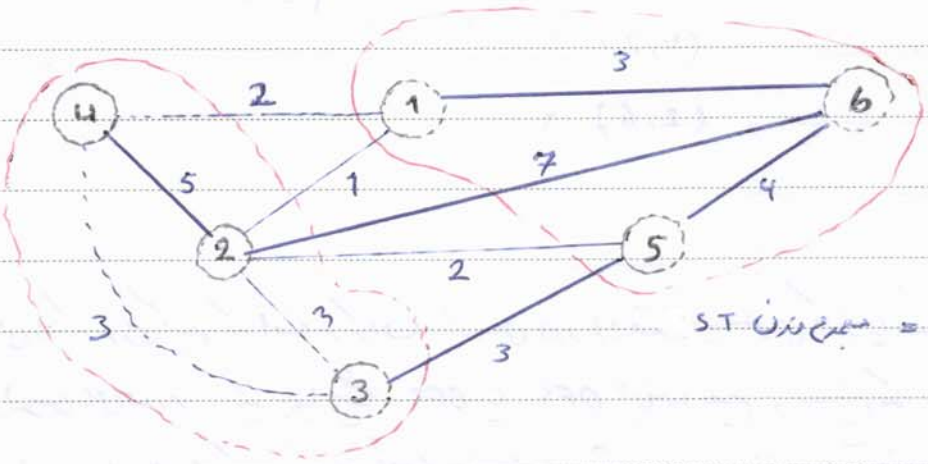
پیدا کردن درخت پوشای کمینه

Minimum Spanning Tree (MST)

به زبان ساده‌تر گراف اولیه گراف اصلی هستیم که می‌خواهیم یک درخت پوشا باشد و مجموع وزن یال‌های آن هم حداقل باشد.

دو الگوریتم هیصانه برای حل این مسئله وجود دارد (prim و kruskal)

قضیه: اگر گراف همبند $G(V,E)$، مجموعه رئوس V را به دو زیرمجموعه V_1 و V_2 افراز کنیم ($V_1 \cap V_2 = \emptyset$, $V_1 \cup V_2 = V$)، آنگاه یال (u,v) که $u \in V_1$ و $v \in V_2$ و دارای کمترین وزن است، جزو درخت پوشای کمینه G است.



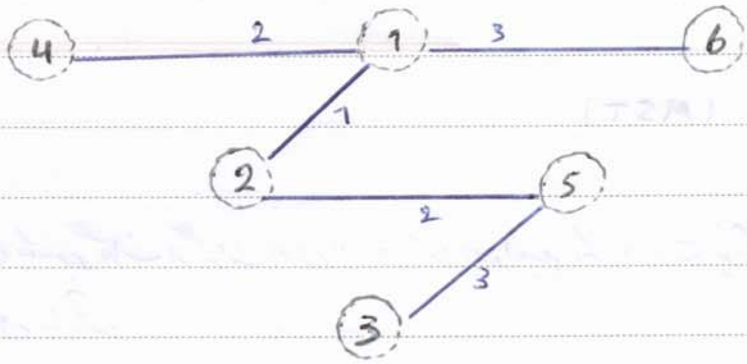
مجموع وزن ST = 5 + 3 + 7 + 4 + 3 = 22

الگوریتم Kruskal:

1- یال‌های گراف را براساس وزن آنها بصورت صعودی Sort می‌کنیم.

2- یالی که کمترین وزن را دارد به MST اضافه می‌کنیم، به شرطی که حلقه ایجاد نشود.
اگر اضافه کردن یال حلقه ایجاد کند، این یال از لیست حذف می‌شود.

مثال:



MST مجموع وزن = $1+2+1+2+3=9$

✓ به محض اینکه درخت پوشا حاصل شد، عمل اضافه کردن یال‌ها را متوقف می‌کنیم و جواب را اعلام می‌کنیم

(1,2) ✓	(2,3) ✗
(3,5) ✓	(1,6) ✓
(2,5) ✓	(6,5) ✗
(1,4) ✓	(4,2) ✗
(4,3) ✗	(2,6) ✗

! برای اینکه بررسی کنیم که اضافه کردن یک یال دور (حلقه) ایجاد می‌کند یا نه، باید روی گراف از یکی از رئوس یال جدید، یک پیمایش BFS یا DFS انجام دهیم و ببینیم که به رأس دیگر یال جدید می‌رسیم یا نه. (برای اینکه بین دو رأس حلقه ایجاد شود باید قبلاً بین آن دو رأس یک مسیر وجود داشته باشد)

✓ Order زمانی الگوریتم فوق بصورت $O(|E||V|)$ می‌باشد.

برای الگوریتم Kruskal راه‌های بهتری وجود دارد (در قسمت پیدا کردن حلقه با Data struc.) زمان آن را به $O(|E|lg|E|)$ کاهش داد.

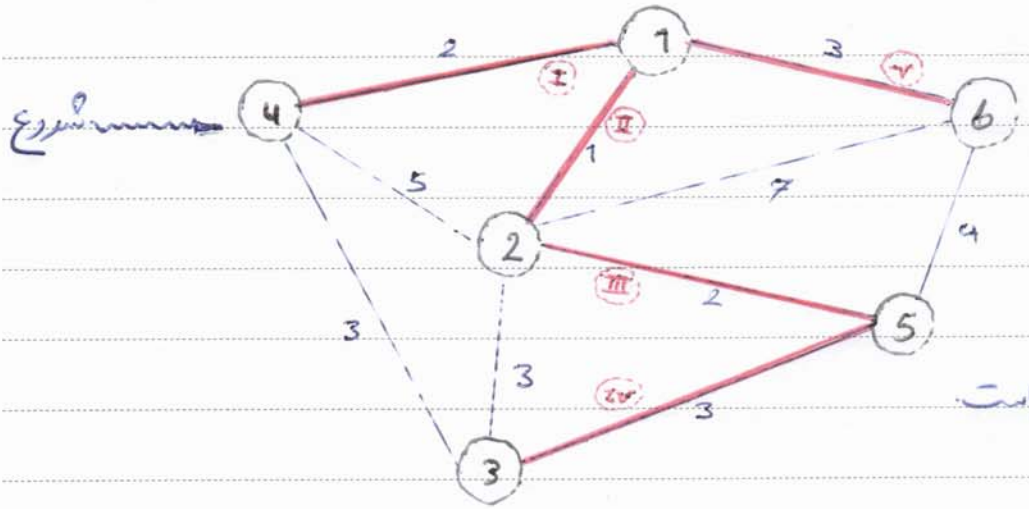
الگوریتم Prim

1- یک رأس دلخواه مثل u را از گراف $G(V,E)$ انتخاب کن: $S = \{u\}$

2- بین یال‌های S و $V-S$ یالی که کمترین وزن را دارد به MST اضافه کن و رأس طرف $V-S$ این یال را به S اضافه کن

گام 2 را تا مدتی که $|V|$ رأس به S منتقل شوند، تکرار کن

مثال:



اعداد بیرنگ نشان‌دهنده ترتیب اعمال الگوریتم است.

✓ در روش Prim نیازی نیست تا با اضافه کردن یال جدید، چک کنیم که حلقه تشکیل می‌شود یا نه، زیرا در حالت ما همواره یکی از S است و یال جدیدی که اضافه می‌شود بین S و $V-S$ است، لذا S همیشه درخت باقی می‌ماند.

Subject:

Year. Month. Date. ( )

✓ اگر الگوریتم فوق را با استفاده از Min-Heap پیاده‌سازی کنیم (یال‌ها را در Heap نگه داریم)، زمانی الگوریتم prim بصورت $O(|E|\lg|E|)$ خواهد شد.