طراحی الگوریتم:

مرجع: • Introduction to Algorithms ⟶ (CLRS) edition 3rd 2009

• Fundamentals of computer Algorithm (Horowitz)

۱- مقدمه و نقش الگوریتم ها

۲- مرور رشد توابع O و o و Ω و Θ

۳- روش های حل روابط بازگشتی ← در روش divide & conquer : ۱) درخت بازگشت ۲) قضیه اصلی

۴- تحلیل الگوریتم ها و اثبات درستی (مرتب سازی heap sort quick sort - اثبات درستی و تحلیل زمانی)
• کمترین زمان
• بدترین حالت
• حالت متوسط

۵- مرتب سازی زمان خطی ← الگوریتم مرتب سازی که زمانش براساس تعداد اعدادش درجه ۱ است.
counting sort ←
radix sort
Bucket sort

۶- پیدا کردن میانه و k امین عدد در بین n عدد مساله selection

۷- تحلیل سرشکن Amortized Analysis یک عمل هزینه ای را روی n عمل یکی ای سرشکن کنیم هر عمل ۲ می شود ← برای ارائه جدول استفاده می شود.

۸- طراحی الگوریتم: روش تقسیم و حل divide & conquer: مساله ضرب ماتریس ها (روش استراسن)

۹- روش برنامه ریزی پویا: Dynamic programming ← • longest common subsequence
داشت بیشترین اشتراکشان چقدر است.
حافظه اضافه می کند
optimal substructure ← از جواب بهینه زیرمسائل

• مثلث بندی بهینه چندضلعی (جمع طول قطرها min)

PAPCO

۱۰- روش حریصانه (greedy) • رویا ← چند حالت بیشتان min

• یکی را جلو می‌کند زمان بهتر از رویا ← همه را جلو نمی‌کند

• کدگذاری هافمن

• کوله پشتی fractional

۱۱- جستجو فضای حالت‌ها: ۸ وزیر ← هیچ ۲ وزیری یکدیگر را تهدید نکنند

• روش عقب‌گرد back tracking انتخاب می‌کند دانه دانه بررسی کردیم یکی عقب انتخاب آلوکیشن کند

آخر را تغییر می‌دهد دوباره جلو می‌آید

• روش شاخه و حد branch & bound

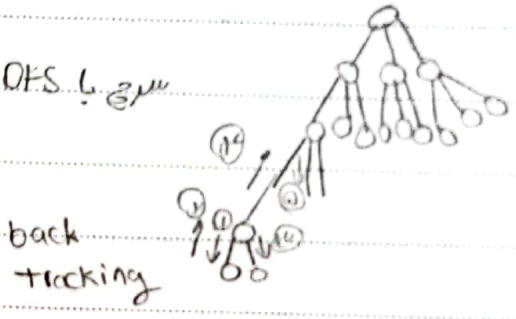
بهتر فضای حالات

↓

بعضی شاخه‌ها لازم نیست بروی

دنبال جواب بهتر ← دانستن یک شاخه بهتر

می‌توان آن شاخه را حرص کرد



۱۲- الگوریتم‌های گراف (مرور) • پیمایش DFS - BFS

• کوتاه‌ترین مسیر

• Dijkstra

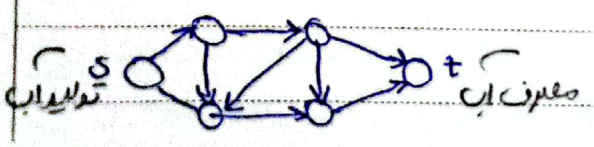
• Floyd-warshal

• درخت پوشا کمینه minimum-spanning-tree

prim kruskal

۱۳- الگوریتم‌های جریان بیشینه maximum flow

بیشترین میزان آب از s به t می‌توان رساند



- نظریه ی پیچیدگی محاسبات مسیر از رأس های گراف هم بگذرد و از هر رأس یا بار ←
همیلتونی ← الگوریتم های موجود زمان نمایی

برج هانوی: اثبات می شود.

تعداد حرکات لازم نمایی است

زمان برج هانوی ← $\Omega(2^n)$

کلاس های مهم انرژی:

کلاس P در زمان چند جمله ای حل می شوند

NP

NP-hard

NP-complete

بارم بندی: میان ترم ۷ نمره

پایان ترم ۱۵ نمره

تمرین ۳ نمره

جلسه دوم:

مقدمه: جایگاه الگوریتم: اهمیت الگوریتم

الگوریتم چیست؟

تعریف الگوریتم: دنباله ای از دستورات با شروع و پایان مشخص و ترتیب مشخص برای حل مسئله

دنباله ای از دستورات که با ترتیب مشخص باید اجرا شود و شروع و پایان مشخص دارد و یک مسئله را حل می کند.

مسأله ← یک زوج مرتب است تشکیل شده از توصیف ورودی و خروجی مسأله

(توصیف خروجی مسأله و توصیف ورودی مسأله)

مسأله مرتب ساری ← (دنباله از جنس n خروجی و یک مجموعه n تایی از اعداد : ورودی)

عدد ورودی به طوری که بزرگتر هر دو عدد پشت هم

$a_i \leq a_{i+1}$

PAPCO

CamScanner

تعریف دنباله: $\langle a_1, a_2, \ldots, a_n \rangle$

ورودی مسأله $\xrightarrow{\text{الگوریتم}}$ خروجی مسئله

اهمیت طراحی الگوریتم: مدام ما با مسائل جدید روبرو می‌شویم ← باید الگوریتم بدهیم.
یا الگوریتم های قبلی ← خاطر سازی بزرگ ورودی ← باید بهبود پیدا کنند.

مسائل دنیای واقعی ← برای بعضی از آن ها الگوریتم نداریم که باید طراحی شود.

اهمیت الگوریتم خوب یا بد یا...

مسأله مرتب سازی: فرض کنیم می‌خواهیم 1M عدد را مرتب کنیم

تعداد اعداد $n = 10^6$

کامپیوتر A $\rightarrow 10^9$ ins/sec سرعتش (instruction)

زمان الگوریتم مرتب‌سازی A $\rightarrow n^2 \rightarrow$ یعنی اگر n عدد داشته باشیم n^2 تا دستور باید اجرا کنیم تا مرتب شود

کامپیوتر B $\rightarrow 10^7 \frac{ins}{sec}$

زمان الگوریتم مرتب سازی B $\rightarrow n \log_2 n$

n^2 ← نشان دهنده تعداد دستورهاست.

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

$$\frac{10^6 \log_2 10^6}{10^7} = \frac{20}{10} = 2$$ زمان حل مسئله مرتب سازی در B

$6 \log_2 10 =$

کامپیوتری که 100 برابر سریع‌تر زمان حل مسئله 500 برابر کند تر است

← علاوه بر کامپیوتر مورد استفاده همان قدر توانایی طراحی الگوریتم مهم است ←
مخصوصاً وقتی سایز ورودی بزرگ است

الگوریتم ۱ هفته ⟶ با تعداد ورودی های بزرگ
الگوریتم ۲ " ۲ هفته این ۲ برابر هم زیاد می شود.

← اول تحلیل بعد طراحی الگوریتم

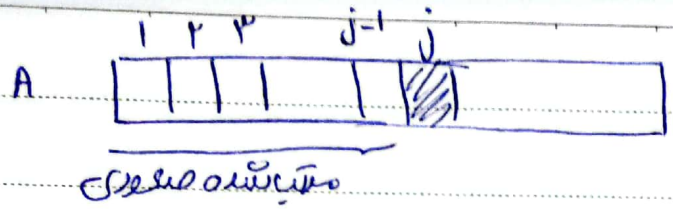
تحلیل زمانی و اثبات درستی الگوریتم insertion sort.

الگوریتم با توصیف مشخص بدهیم یک خروجی که مطابق خروجی مشخص شده است بدهد

آرایه ← تعداد اعضای آرایه

Insertion_Sort(A , n)

۱- ارائه الگوریتم با pseudo code است.
۲-
۳- بیان الگوریتم با شبه کد است.
۴- pseudo_code
۵- Insertion_Sort →
۶- یک روش incremental است
۷- که یک روش حل مساله است ← ورودی های
۸- مسئله یکی یکی داده شود.
یک عدد ← خودش
دو عدد ← مقایسه با اولی
سه عدد ← سومی را با دومی مقایسه اگر کوچکتر بود با اولی و...



اضافه کردن خونه j ام

تا زمانی که خانه های قبل خانه j کوچکتر است خانه ی j را عقب تر می آوریم ← تا در نهایت از اولی هم کوچکتر است و خانه ی یک قرار می گیرد

هنگام کپی و شیفت دادن برای خراب نشدن خانه ها ← خانه ی j را در متغیر key می ریزیم

j از ۲ شروع می کنیم تا n ← هنگام درج خانه ی j با قبلی ها مقایسه می کنیم.

← یک حلقه j از ۲ تا n ← for

← یک حلقه دیگر ← تا وقتی از قبلی ها کوچکتر است عقب می آوریم ← while با تکرار شرطی

```
Insertion_Sort(A, n)
for j=2 to n do
    key ← A[j]
    i = j-1                 while شمارنده حلقه
    while i>0 and key < A[i] do
        A[i+1] = A[i]
        i = i-1
    A[i+1] = key
```

بعد از ارائه الگوریتم باید درستی الگوریتم را بررسی کنیم correctness : با استفاده از استقرا اثبات می کنیم

حال باید ثابت کنیم الگوریتم درست کاری کند ← پایه ی استقرا ← فرافرض داخلی ← الگوریتم بازگشتی

الگوریتم غیربازگشتی ← اگر نتیجه ای در گام قبلی درست است ← ثابت کنیم برای گام بعدی هم درست است.

PAPCO

CamScanner

ثابت حلقه: loop invariant ← گزاره‌ای که در طول اجرای حلقه‌ی الگوریتم درست است و از درستی آن درستی الگوریتم را می‌توان نتیجه گرفت.

insertion sort ← چه‌طوری از آرایه کمک می‌گیرد به مرتب سازی ← در مرحله j ام خانه‌ی A[j-1] مرتب شده هستند و همان عناصری هستند که در خانه A[j-1] بوده‌اند همان خانه‌های A[1-j] اولیه ما فقط ترتیبشان صعودی است.

اثبات درستی: الگوریتم تمام می‌شود خانه‌های 1 تا n مرتب شده صعودی هستند.

ثابت حلقه: در ابتدای گام j ام حلقه for الگوریتم insertion_sort، زیرآرایه‌ی A[1 ..... j-1] شامل همان عناصر اولیه‌ی ورودی هستند که در زیرآرایه A[1 ... j-1] بوده‌اند ولی به صورت مرتب شده صعودی

الگوریتم درست ← مبتنی بر یک سری پایه‌ها هستند

پایه: حالت ابتدایی حلقه، حلقه تازه می‌خواهد شروع شود

حالت پایه: (قبل از شروع حلقه‌ی for) ← j=2

ثابت حلقه برای حالت پایه: در ابتدای گام j=2 ام حلقه‌ی for الگوریتم Insertion-Sort زیرآرایه‌ی A[1 ... 1] شامل همان عناصر اولیه ورودی هستند که در زیرآرایه‌ی A[1 ... 1] بوده‌اند ولی به صورت مرتب شده‌ی صعودی A[1 ... 1] ← که این گزاره درست است و پایه برقرار است.

فرض می‌کنیم برای j درست است ثابت می‌کنیم برای j+1 درست است. ← باید در گام j+1 ام ثابت کرد خانه 1 تا j مرتب به صعودی هستند.

اعداد 1 تا j ام اگر شیفت پیدا کنند راست ← ترتیب عوض نمی‌شود و هر خانه 1 خانه جلو می‌رود.

* اعداد از بازه‌ی 1 تا j خارج نمی‌شوند.

* جای خالی که بین آن‌ها می‌ماند با خانه j یا key پر می‌شود ← یا خانه‌ی اول آرایه یا جایی که کلید از خانه‌ی i ام بزرگ‌تر مساوی است. کلید از خانه‌ی قبلی خودش بزرگ‌تر مساوی و از خانه بعدی کوچک‌تر است.

for $1 \leq i \leq j-2$ $\quad a_i \leq a_{i+1}$

$a_1 \; a_2 \; \ldots \; a_i \leq key \leq a_{i+1} \; a_{j-1}$

شیفت ترتیب را عوض نمی‌کند.

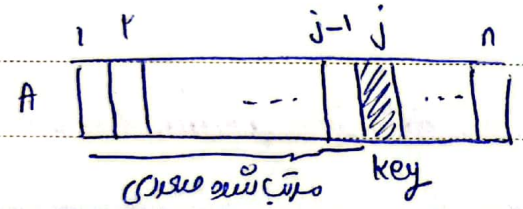
اثبات درستی الگوریتم Insertion Sort

loop invariant: دیدگی که می‌بینیم قبلی‌ها مرتب شده صعودی اند.

در ابتدای گام j ام حلقه‌ی for الگوریتم insertion sort زیرآرایه‌ی A[1...j-1] شامل همان عناصر ورودی در زیرآرایه‌ی A[1...j-1] هست ولی به صورت مرتب شده صعودی

حالت پایه: $j=2$ ← خانه‌ی اول آرایه A[1...1] درست است مرتب شده است و همان A[1...1] ورودی است.

حالت گذار: فرض می‌کنیم برای j ثابت حلقه برقرار است برای j+1 اثبات می‌کنیم.



گام j ام ← خانه‌ی j را در کلید می‌گذاریم تا زمانی که کلید کوچک‌تر از خانه‌های قبلی است یک شیفت پیدا می‌کند اعداد به سمت راست

چون خانه‌های آرایه که در بازه‌ی 1 تا j-1 هستند حداکثر یک خانه به سمت راست شیفت می‌دهیم پس ① همچنان در بازه‌ی 1 تا j آرایه باقی می‌مانند. key تا وقتی که از خانه‌ی قبل از خود کوچک‌تر است در آرایه به سمت ابتدای آرایه حرکت می‌کند و نهایتاً در جایی در آرایه درج می‌شود که از خانه‌های بعد از خود کوچک‌تر و از خانه‌های قبل از خود بزرگ‌تر یا مساوی است. ②

③ شیفت دادن خانه‌های آرایه 1 تا j-1 ترتیب صعودی را عوض نمی‌کند

← بنابراین در ابتدای گام j+1 حلقه‌ی for همه‌ی 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.

PAPCO

CamScanner

الگوریتم درصورتی درست است که صعودی مطابق مشخصات بدهیم خروجی درستی به ما بدهد

وقتی الگوریتم Insertion Sort تمام می شود (در انتهای حلقه ی for) ثابت حلقه عبارت خواهد بود از: j ← در گام آخر n+1 چون وقتی n تمام شد دوباره j+1 می شود n+1 در حلقه جلو می رفت (j = n+1) در ابتدای گام j=(n+1) ام حلقه ی for

زیر آرایه A[1..n] شامل همان عناصر اولیه زیر آرایه A[1..n] است ولی به صورت مرتب شده صعودی

۱- اثبات می کنیم ثابت حلقه درست است

۲- j را در آخر الگوریتم چک می کنیم ← که درستی الگوریتم را نشان می دهد.

تحلیل زمانی الگوریتم مرتب سازی insertion Sort

← منظور تعداد دستورات پایه است برای حل مسئله

قدرت پردازشی ماشینی که قرار است الگوریتم روی آن پیاده شود هم مهم است ← به آن مدل محاسباتی می گویند

The RAM model

کامپیوتر تک پردازنده ← در هر لحظه ۱ دستور می تواند اجرا کند ← مثل جمع و تقسیم و ضرب

RAM - random access machine

در O(1) می تواند هر جایی دلتری داشته باشد

دستورات ماشین: جمع و تفریق | ضرب و تقسیم | shift

مقایسه | ذخیره و بازیابی | باقی مانده

Sequential نیست

باید ببینیم الگوریتم چند دستور پایه است.

زمان الگوریتم باید فرم تابع باشد ← متغیر تابع چیست

تابع زمان اجرای الگوریتم تابعی از چیست؟

$T(n) = n^2$ n چیست؟

insertion Sort ← ورودی مجموعه ای از n تا عدد است.

① ورودی الگوریتم insertion Sort : مجموعه از n عدد

زمان الگوریتم را برحسب n یعنی تعداد اعداد بیان می کنیم.

الگوریتم دو عدد می گیرد جمع می زند خروجی را به ما می دهد ← اعداد می توانند n بیتی باشند

② الگوریتمی که ۲ عدد n بیتی را با هم جمع می زند ← زمان الگوریتم را برحسب n که تعداد بیت های این ۲ عدد است بیان می کنیم.

ما ورودی را به صورت code شده می گیریم در حافظه به صورت code شده ذخیره می کنیم

در کامپیوتر به صورت بیتی ۰ و ۱ ← ورودی را کد می کنیم ذخیره می کنیم.

۳۲ بیت

① insertion Sort ، فرض این است هر عدد در ۴ بایت جا می شود. و n عدد داریم

هر عدد ۴ بایت (۳۲ بیت) و n عدد داریم

$32n$ bit → طول ورودی (ورودی در مبنای ۲ کد شده)

اندازه مسئله

② دو عدد داریم هر کدام n بیت طول ورودی می شود $2n$ bit

← در هر دو مسئله طول ورودی یک ثابتی ضرب در n

طول ورودی = $c \times n$

طول ورودی را در هر دو مسئله n می گیریم.

طول ورودی را براساس :

size یا اندازه ورودی مساله ←

* زمان الگوریتم را براساس size ورودی مسئله بیان می‌کنیم.

Insertion_Sort(A,n) کارهای داخل دستور jump, compare, store

1- for j=۲ to n do → (n بار تکرار) $c_1 \times n$

۲- * key = A[j] → (n-1 تکرار) $c_2 \times n-1$

۳- i = j-1 → $c_3 \times n-1$

۴- while i>0 and key < A[i] do → $c_4 \times \sum_{j=2}^{n}(t_j+1)$

۵- A[i+1] = A[i] → $c_5 \times \sum_{j=2}^{n} t_j$

۶- i = i-1 → $c_6 \times \sum_{j=2}^{n} t_j$

۷- A[i+1] = key → $c_7 \times n-1$

تعداد تکرار بدنه حلقه while

در اسمبلی ← ماشین یک clock دارد. سرعت یک ماشین یک کاری انجام می‌دهد ← یک دستور ماشین یک تعداد clock (تیک) لازم دارد. ولی تعداد تیک ثابت است. دستورات پایه را به اندازه ثابت فرض می‌کنیم یا حتی ۱ (واحدمان دستورات پایه)

دستورات پایه یا خارج حلقه یا داخل حلقه ← داخل حلقه در تعداد تکرار حلقه ضرب می‌شود

header حلقه یک بار بیشتر از بدنه اش تکرار می‌شود

شد که ← زمان محاوره + دستورات پیش رفته ← ولی باید در ماشین انجام شود

برای for

Store j=۲

مقایسه j و n if j ≤ n

ادامه می‌دهد

else j > n → دستور jump برش

می‌کند به ته حلقه

PAPCO

program counter ← jump ← registor را جابه‌جا می‌کند
pc
مشخص می‌کند کدام دستور باید اجرا شود

* load حافظه را در registor می‌ریزد ← A[j] را داخل یک registor می‌کند
store که رجیستر را در حافظه کلید ذخیره می‌کند

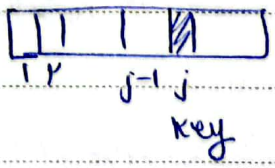
i = j-1 ⟶ j را load کند
سپس decrement کند
سپس i store می‌کند

A[i+1] = key ⟶ load دارد
store

۴ ← چند تا compare برقرار نباشد jump می‌کند ← هزینه while
ثابت است ← c4

تکرار حلقه‌ی while متغیر است ← tj بار تکرار می‌شود به ازای j های مختلف
Σ

tj ⟶ می‌تواند صفر بار باشد



key از همه‌ی خانه‌های قبلی بزرگتر باشد ← tj = 0
اگر از همه‌ی خانه‌های قبلی کوچک‌تر باشد j-1

$0 \leq t_j \leq j-1$
بهترین حالت
بدترین حالت

* بهترین حالت و بدترین حالت کی رخ می‌دهد

PAPCO
CamScanner

بهترین حالت: $t_j = 0$

زمان الگوریتم $T(n) = \overset{\text{ثابت}}{a}n + \overset{\text{ثابت}}{b}$

بدترین حالت $t_j = j-1$

$$\sum_{j=2}^{n} t_j \; (j-1) \longrightarrow$$ جمع از ۱ تا n-1

↓

تکرار حلقه

زمان الگوریتم $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \underbrace{\sum_{j=2}^{n} j}_{\frac{(n+2)(n-1)}{2}} +$

$$(c_6 + c_7) \underbrace{\sum_{j=2}^{n} (j-1)}_{\frac{n(n-1)}{2}} + c_8 \times (n-1)$$

$= a' n^2 + b' n + c'$ تابع درجه ۲

تعداد دستورات باید را جمع زدیم ممکن است تعداد دستورات باید کم باشد یا زیاد

بهترین حالت — بدترین حالت

insertion sort در بهترین حالت درجه ۱ است ورودی را ۲ برابر کنیم زمان تقریباً ۲ برابر می‌شود

در بدترین حالت ۴ برابر می‌شود

بدترین حالت ← الگوریتم دیگر بهتر از این حالت نمی‌شود insertion بهتر از درجه ۲ نیست

← الگوریتم - طور متوسط چقدر زمان می‌برد؟ رفتار میانگین برای ما مهم است

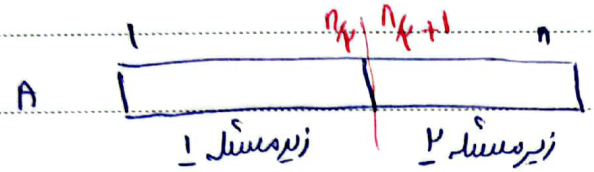
← تحلیل حالت متوسط

insertion Sort ← حلقه داشت - for داشت ← حال الگوریتم دیگر بازگشتی است

الگوریتم merge Sort مبتنی بر divid & conquer

Merge Sort:

مبتنی بر روش divide & conquer است.



مسئله را به ورودی های کوچک تر تقسیم کردن راه های مختلف وابسته به طراح الگوریتم است. از وسط یا سه قسمت یا (۱ و n-1 تا)

← هر کدام الگوریتم های متفاوت با زمان های متفاوت.

۱ و n-1 به الگوریتم insertion Sort می رسیم ← n^2

از وسط نصف کنیم ← mergesort ← $\log n$

Merge_Sort (A, i, j)

اندیس ابتدای آرایه — اندیس انتها

هر بار الگوریتم نصف می شود آن وسط ها آرایه اندیس اول و آخرش ۱ و n نیست

← نیمی از زیر مسئله ها از یک اندیس تا اندیس دیگر شروع می شود ← بهتر است اندیس ابتدای آرایه و انتها را بدهیم

* تا زمانی که مسئله به اندازه کافی کوچک نشود دیگر لازم نیست تقسیم را ادامه دهیم ← موقعی است که نون ها حضور پیدا کنند ← ۱ داده ای دنباله کاری کرد می توان return کرد

```
Merge_Sort (A, i, j)
1- if j > i then
2-     q = ⌊(i+j)/2⌋
3-     Merge_Sort(A, i, q)
4-     Merge_Sort(A, q+1, j)
5-     Merge (A, i, q, j)
6-
7-
8-
```

شرط پایه برای n های بزرگ خیلی تفاوت ایجاد نمی کند

شرط ادامه دادن

گام تقسیم

نیمه اول را مرتب کند (recursive)

ترکیب دو نیمه مرتب شده

```
Merge_Sort (A, i, j)
1- if i < j then
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

گام حل بازگشتی

گام ترکیب

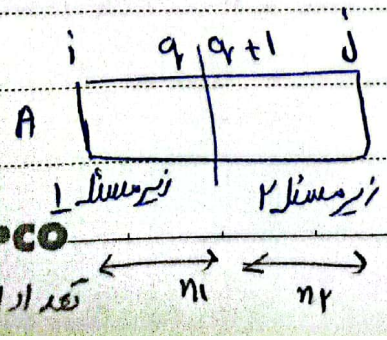
merge Sort از divide Conquer استفاده می کند

Divide & conquer سه گام دارد

۱- گام تقسیم به زیرمسائل

۲- گام حل بازگشتی زیرمسائل

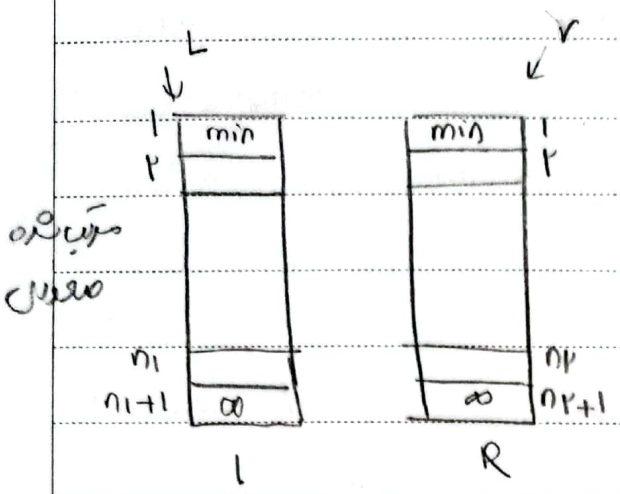
۳- گام ترکیب جواب زیرمسائل



$n_1 = q - i + 1$

تعداد اعداد

merge



این دو L و R مرتب شده صعودی اند ← min مقایسه می‌کنیم هر کدام کوچک‌تر بود در آرایه اصلی قرار می‌گیرد و اشاره‌گر آن یکی جلو می‌رود و این روند ادامه می‌یابد

Merge (A, i, q, j)

1- $n_1 = q - i + 1$, $n_2 = j - q$ → تعداد اعداد نیمه اول و دوم

2- $L[n_1+1] = \infty$, $R[n_2+1] = \infty$ →

3- for l=1 to n_1 do

4- L[l] = A[i+l-1]

5- for r=1 to n_2 do

6- R[r] = A[q+r]

7- l=r=1

کپی کردن نیمه اول و دوم

l=1
L[1]=A[i]

r=1
R[1]=A[q+1]

8- for k=i to j do → از آنجا min ها را در مقصد بریزد

9- if L[l] < R[r] then A[k]=L[l] , l++

10- else A[k] = R[r] , r++

مقایسه min ها

کاربرد ∞ ها ← ممکن است یک آرایه زودتر تمام شود ← تا موقعی که اندیس به انتها نرسیده نیمه کپی کند در مقصد تمام که شد از دیگر کپی کند ← برای طولانی نشدن کد ∞ می‌گذاریم تا آرایه ← خود به خود وارد آرایه دیگر می‌شود.

این الگوریتم درست است یا نه؟

PAPCO

CamScanner

درستی الگوریتم ← دو الگوریتم merge Sort و merge داریم اول باید ثابت کنیم
merge درست است ← ثابت می‌کنیم merge_sort درست است.

اثبات merge . ورودی درست فرض کنیم می‌دهیم خروجی درست می‌دهد

آرایه A که از i تا q
و از q+1 تا j مرتب
شده صعودی باشد

باید از i تا j مرتب شده صعودی
باشد.

اثبات درستی الگوریتم merge . فرض: آرایه‌ی 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.

← اثبات با گزاره لگکس ← ویژگی که در الگوریتم قابل مشاهده است در آرایه‌ی A و L و R ← L به کوچک‌ترین عنصر L اشاره می‌کند که هنوز در مقصد کپی نشده

" " " " " " R " " " r

k تا هر جا که جلو رفته ← تا k عناصر های تا الان کپی شده‌اند min های R و L بوده‌اند و مرتب شده صعودی اند

حلقه‌ی اصلی در خط ۸ و ۹ و ۱۰ هستند ← ثابت حلقه را برای آن‌ها می‌نویسیم

loop invarient: در ابتدای گام k ام حلقه‌ی for در خطوط ۸ تا ۱۰ زیرآرایه‌ی اول آرایه تا قبل k $A[i \dots k-1]$ شامل k-i کوچک‌ترین عناصر آرایه‌ی $L[1 \dots n_1+1]$ و $R[1 \dots n_2+1]$ است به صورت مرتب شده صعودی. همچنین $L[l]$ و $R[r]$ کوچک‌ترین عناصر آرایه‌ی L و R هستند که هنوز در A کپی نشده اند.

در انتهای الگوریتم این گزاره می‌شود: در انتها k = j+1 در ابتدای گام j+1 ام حلقه‌ی for ...
$n = j-i+1$ ← ∞ ها هم کپی نمی‌شوند
← حکم ماست.

پایه‌ی استقرا: $(k=1)$: در ابتدای گام $(k=1)$ ام حلقه‌ی for در خطوط ۸ تا ۱۰ زیرآرایه‌ی $A[1 \dots i-1]$ شامل ۰ کوچک‌ترین عناصر آرایه‌ی $L[1 \dots n+1]$ و $R[1 \dots n+1]$ است به صورت مرتب‌شده صعودی. $L[1]$ و $R[1]$ کوچک‌ترین عناصر آرایه‌ی L و R هستند که در A هنوز کپی نشده اند.

$i=1$ و $r=1$

$r=l=1$

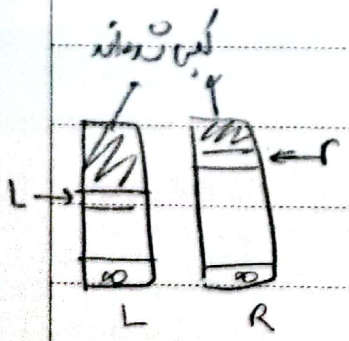
آرایه‌ی بزرگ

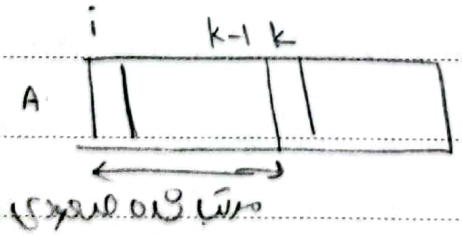
از روی فرض دونیمه مرتب‌شده صعودی اند.

گام گذر: برای k فرض کنیم درست است برای k+1 اثبات کنیم.

فرض می‌کنیم ثابت حلقه برای k برقرار است و برای k+1 اثبات می‌کنیم ← یک گام حلقه‌ی for و انتظار می‌افتد.

نکته‌ها: ۱- k-1 تایی که الان کپی شده اند کوچک‌ترین عناصری هستند که کپی شده اند پس اگر یکی که کپی نشده را کپی کنیم ترتیب صعودی حفظ می‌شود





۲- بین L و r اونی که کوچک‌تر است را کپی می‌کنیم بین همه‌ی باقی مانده‌ها کوچک‌ترین در L و R است هرکدام کپی می‌شود اندیس را یکی جلو تر می‌بریم L و r به min L و R بازهم اشاره می‌کنند.

یک گام جلو رفتیم L و r باز

عدد min ام L و R باشد

اثبات حالت پایه: طبق فرض L[l] و R[r] کوچک ترین عناصر L و R هستند که هنوز کپی نشده اند پس اگر min این دو مقدار در A[k] کپی شود عناصر آرایه ی A[i ... k] همچنان کوچک ترین عناصر L و R هستند که در A کپی شده است و چون قبل از این کپی کردن، آرایه ی A[i ... k-1] شامل کوچک ترین عناصر L و R هستند و مرتب شده صعودی است، با این کپی همچنان مرتب شده صعودی خواهند بود.

چون اونی که کپی می شود از این ها بزرگ تر است

اثبات اندیس i و r کوچک ترین عناصر را اشاره می کنند

پس از کپی کردن min R[r] و L[r] در A[k] اندیس مربوطه ++ می شود پس همچنان R[r] و L[l] min L و R هستند که هنوز در A کپی نشده اند.

در ابتدا درست است در انتها k=j+1 و n=j-i+1 باز هم درست است.

↓

A شامل n/2 تا از عنصر L هستند و خیر آن هم ها
R " " " n/2

merge_sort ← استقرا روی فراخوانی بازگشتی ← الگوریتم n کار می کند برای $\frac{n}{2}$ هم کار کند

استقرا قوی ← باید هر مقدار کمتر n فرض می کنیم درست است برای n. اثبات می کنیم

پایه = 1 ← j=i ← آرایه است نخرده و مرتب شده صعودی هم هست □

برای حالت پایه شرط if i<j برقرار نیست و از الگوریتم خارج می شویم و آرایه ورودی است نخرده باقی می ماند 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 ←

حال می‌خواهیم ثابت کنیم برای n هم درست است

$q = \frac{i+j}{2}$ ← و هر کدام از خط ۲ و ۳ ← i تا q و j تا q+1 درست مرتب می‌شوند ← زیرا کمتر از n تا عنصر دارند ← (هر نیمه حداقل ۱ عنصر دارد آمات

این نیمه‌ی دیگر آن یک عنصر را ندارد ← کمتر از n است)

اگر بیستم merge هم درست است داریم ورودی درست هم می‌دهیم زیرا i تا q و j تا q+1 مرتب شده صعودی اند ← خروجی هم درست است.

تحلیل زمان merge sort و merge :

اول زمان merge :

تحلیل زمانی merge :

و for کپی نیمه اول و دوم

یک for مقایسه R ها و کپی می‌کند

در مقدار

خط ۱ → هزینه ثابت c_1

خط ۲ → c_2

خط ۳ و ۴ → $c_3 \times n_1$ → یکی max باشد

خط ۵ و ۶ → $c_4 \times n_2$

خط ۷ → c_5

خط ۸ و ۹ و ۱۰ → $c_6 \times n$

تا n

به تعداد کل عناصر آرایه تکرار می‌شود

در بدنه‌ی حلقه if را چک می‌کند

که هزینه ثابت است

اگر هزینه if و else فرق داشت

max را بخواهیم max می‌گیریم

" min " " min

کل هزینه merge $\leq c_1 + c_2 + \max\{c_3, c_4\}(\overset{n}{n_1 + n_2}) + c_5 + c_6 n$

$\leq a.n + b$

حد پایین هم: کل هزینه merge $\geq c_1 + c_2 + \min\{c_3, c_4\} \times n + c_5 + c_6 \times n$ ،

⇐ حد پایین و بالا هر دو n است ← رشد این تابع $\Theta(n)$ است

در بهترین حالت یا بدترین حالت یا هر حالتی ← $\Theta(n)$ است و با n رابطه خطی دارد.

↳ میانگینش هم n است.

PAPCO

CamScanner

تحلیل زمانی merge_Sort

Merge_Sort(A, i, j)

		هزینه
1- if $i<j$ then	$\longrightarrow$	c_1
2- $q=\lfloor \frac{i+j}{2} \rfloor$	$\longrightarrow$	c_2
3- Merge_Sort(A, i, q)	$\longrightarrow$	$T(\lceil \frac{n}{2} \rceil)$
4- Merge_Sort(A, q+1, j)	$\longrightarrow$	$T(\lfloor \frac{n}{2} \rfloor)$
5- Merge(A, i, q, j)	$\longrightarrow$	$\Theta(n)$ تابع درجه یک برحسب n

الگوریتم برحسب divid & Conquer / بازگشتی هم هست

برای n تا ← هر زمینه $\frac{n}{2}$

n تا ← نیمه اول سقف $\frac{n}{2}$

نیمه دوم کف $\frac{n}{2}$

$T(n)$ تابع زمان اجرای الگوریتم

$$T(n)=\underbrace{c_1+c_2}+T(\lceil \tfrac{n}{2} \rceil)+T(\lfloor \tfrac{n}{2} \rfloor)+\underbrace{\Theta(n)}_{\text{درجه یک برحسب } n}$$

$$T(n)=T(\lceil \tfrac{n}{2} \rceil)+T(\lfloor \tfrac{n}{2} \rfloor)+\Theta(n)$$

رابطه بازگشتی است. بعد حل می‌کنیم با درخت بازگشت و تئوری بسل جوابش $\Theta(n\log n)$

اگر فقط درجه تابع مهم باشد می‌توان سقف و کف را با $T(\frac{n}{2})$ یکی گرفت

حد پایین $n\log n$ داریم می‌دانیم کمتر از این نمی‌توان حل کرد ← الگوریتم داده ایم بازسازی

n^2 ← تا وقتی به حد پایین نرسیدیم ← کاهش درجه مهم است ← بعد که به حد پایین رسیدیم ← سعی کنیم ضریب را کم کنیم مثلاً $4n\log n$ تبدیل به $2n\log n$

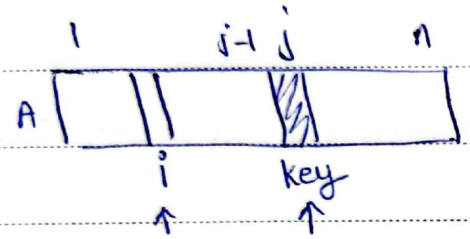
الگوریتم mergeSort بدترین و بهترین حالت:

merge همیشه n است ← بهترین حالت $cn+c_2$ ← میانگینش هم همینه

بدترین حالت $an+b$

merge-Sort ← بدترین و بهترین حالت مساوی است ← هر عددی بهش بدیم عددی را نصف می‌کند ← زمانش $\Theta(n\log n)$ است.

تحلیل حالت متوسط insertion Sort:



while هر بار insertion sort بستگی به مقدار کلید با مقدار خانه‌های ماقبلم دارد ← که چه قدر بیشتر بوده ← که این هم به ورودی بستگی دارد مثلاً آرایه‌ی صعودی بدهند ← key از همه‌ی قبلی‌ها بزرگتر است آرایه نزولی بدهند ← تعداد تکرار حلقه‌ی while $0 \le t_j \le j-1$

بهترین حالت ↓ زمان خطی $T(n) \in \Theta(n)$

بدترین حالت $T(n) \in \Theta(n^2)$

حالت متوسط:

نیاز به احتمالات داریم ← باید دید کدام چه مقادیری را با چه احتمالی می‌گیرد و با هم جمع می‌کنیم.

x متغیر تصادفی

$$\bar{x} = E[x] = \sum_{x} x \cdot pr\{X = x\}$$

امید ریاضی

روی همه‌ی x ها

احتمال X باشد x

X بتواند n تا مقدار بگیرد $\sum_{1}^{n} x \cdot$ احتمال آن مقدار

میانگین یا امید ریاضی insertion Sort ،

زمان به t_j بستگی دارد ← می‌توان میانگین t_j را به دست آورد.

امید ریاضی ترکیب خطی = ترکیب خطی امید ریاضی

$$E[ax+by] = aE[x] + bE[y]$$

$$\sum_{0}^{j-1} t_j \cdot pr$$ امید ریاضی t_j

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

باید احتمال اینکه t_j ← ۰ و ۱ و ۲ و ... باشد

چقدر؟ محاسبات زیاد و سخت ← راه بهتر دارد.

Indicator random variable متغیر تصادفی که فقط می‌تواند ۰ و ۱ باشد

متغیر تصادفی شاخص

رخداد A

① بارون آمدن	pr
⓪ بارون نیامدن	1-pr

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

t_j تعداد تکرار حلقه‌ی while ← تعداد کوچک‌تر بودن کلید از خانه‌های قبلی خودش بستگی دارد. t_j = تعداد اعدادی که کلید از خانه‌های قبلی خود کمتر است.

اعداد داخل آرایه کاملاً تصادفی پخش شده باشند ← n عدد که n! حالت دارد هر ترکیب را با احتمال $\frac{1}{n!}$ ← می‌دهند ← هر عدد آرایه با احتمال یک دوم یا بزرگ‌تر است یا کوچک‌تر

برای هر i متغیر شاخص تعریف می‌کنیم.

$$X_i = \begin{cases} 1 & \text{اگر کلید از } A[i] \text{ کوچک‌تر باشد} \\ 0 & \text{وگرنه} \end{cases}$$

خانه‌های قبل کلید i

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

تعداد دفعات تکرار حلقه

امید ریاضی $E[X_i]$ ← X_i دو مقدار دارد

$1 \times$ احتمال $1 + 0 \times$ احتمال 0 بودن

= احتمال ۱ بودن

$$E[X_i] = pr\{\text{احتمال اینکه کلید از } A[i] \text{ کوچک‌تر باشد}\} \xrightarrow[\text{بخش شده باشند}]{\text{اگر اعداد تصادفی}} = \frac{1}{2}$$

$$E[t_j] = \sum_{i=1}^{j-1} \frac{1}{2} = \frac{j-1}{2} \rightsquigarrow \text{تعداد}$$

حالت میانگین همان بدترین حالت ماست فقط ضریب $\frac{1}{2}$ دارد.

$$E[t_j] = \frac{j-1}{2} \longrightarrow \underset{\text{حالت میانگین}}{T(n)} \in \Theta(n^2)$$

sort شده بودن خانه‌های ۱ تا $j-1$ ← اگر مرتب نشده باشد فرض کنیم و $E[t_j]$ را بدست آوریم حال sort کنیم آیا تعداد خانه‌های بزرگتر از key فرق می‌کرد خیر

احتمال صفر بودن t_j ← $\left(\frac{1}{2}\right)^{j-1}$ ← احتمال ۰ بودن

حالت میانگین ← به حالت توزیع احتمالش بستگی دارد.

هر نفر عددی می‌دهند تصادفی باشد

۱ ۲ ۳ ۴

۲ ۱ ۳ ۴

۲ ۱ ۴ ۳ ← با احتمال ← خانه‌ی کلید از بقیه

۱ ۲ ۴ ۳ مساوی به ما بالا — خانه‌ها با احتمال ۱/۲ کوچک‌تر است.

۲ ۳ ۱ ۴

ن

الگوریتمی داریم که میانگین بدترین حالت یک باشد؟

میانگین کجا می‌تواند باشد ← احتمال بدترین یا بهترین بهتر است بستگی به حالت‌هایی سوق پیدا می‌کند

	بهترین	بدترین	میانگین
quick Sort →	$n\log n$	$\log n^2$	$n\log n$

احتمالشان بیشتر است.

* میانگین همیشه بین دو مساوی بهترین و بدترین حالت.

↳ قابل اتکاتر است ← رفتار میانگین الگوریتم را نشان می‌دهد.

مرور رشد توابع: ← رشد تابع تقریبا همین است ← قوی notation

o , ω , Θ , Ω , O

little o , little omega , big omega , big o

بالا ← تابع رشدش از یک مقداری نمی‌تواند بیشتر باشد

تابع $T(n)$ ← بهترین حالت

$T(n)$ $T(n)$

بدترین حالت	بهترین حالت	بدترین حالت
n^2	n	میانگین

$T(n) \in \Omega(n)$ — حد پایین تابع من n

$T(n) \in O(n^2)$ — حد بالای تابع من n^2

PAPCO

CamScanner

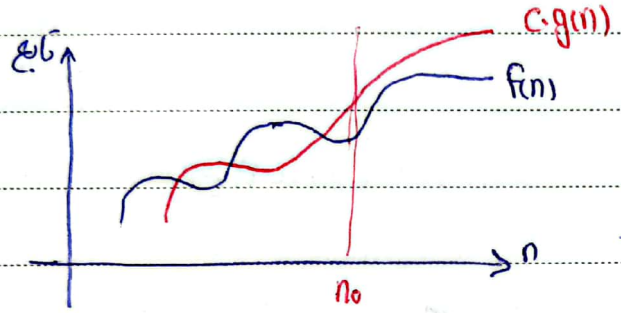
$T(n)$ بهترین حالت $\in \Theta(n)$ $\xRightarrow{\text{معادله}}$ $T(n) \in \Omega(n)$

$T(n)$ حد پایین

$T(n)$ بهترین حالت $\in \Theta(n^2)$

big O , little o : $f(n) \in O(g(n)) \iff \exists\, c>0 , n_0>0 \mid \forall n \geq n_0$

$f(n) \leq c\,g(n)$



$f(n) \in O(g(n))$

* رشد تابع g باید بیشتر یا مساوی f باشد.

رشد تابع چند جمله‌ای یعنی درجه

ولی اگر نباشد یعنی lim اش ∞ می‌شود

قبل n0 مهم نیست کی + کی -

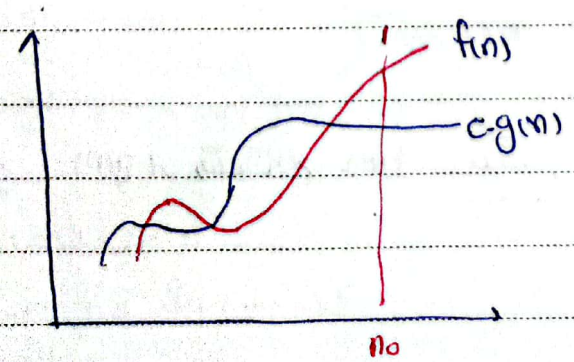
برای n های بزرگتر از n0 کار می‌کنیم

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

کاربرد O . تضمین بدهیم رشد تابع حد بالا دارد بیشتر از این نیست. ولی قطعاً همان نیست ممکن است کمتر باشد.

big Omega:

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



رشد g باید از f کمتر یا مساوی باشد.

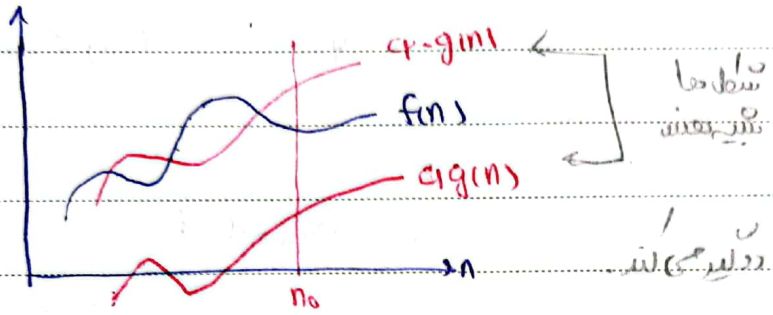
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \begin{cases} 0 & \text{رشد } f > \text{رشد } g \\ c \neq 0 \text{ ثابت} & \text{رشد } f = \text{رشد } g \end{cases}$$

PAPCO

CamScanner

Θ: در واقع and ـــ Ω و O است ـــ وقتی حد بالا و پایین یک تابع مساوی شوند می‌شود Θ تابع

$$f(n) \in \Theta(g(n)) \Longleftrightarrow \exists c_1, c_2 > 0 \ , \ n_0 > 0 \ | \ \forall n > n_0 \quad \underbrace{c_1 \cdot g(n) \leq f(n)}_{\Omega} \leq \underbrace{c_2 \cdot g(n)}_{O}$$



از یک جایی به بعد تابع f بین این دو گیر می‌کند

* باید رشد تابع $g(n)$ برابر رشد تابع $f(n)$ باشد.

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

$f(n) \in \Theta(g(n)) \longrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

حالت جدید:

ادامه رشد توابع: O و Ω و Θ و ω و o

حد بالای اکید

$$f(n) \in o(g(n)) \Longleftrightarrow \underline{\forall c > 0}, \exists n_0 > 0 \ | \ n \geq n_0 \quad f(n) \leq c \cdot g(n)$$

تفاوت با O

به ازای هر c یک n_0 را می‌توان پیدا کرد

← فقط در صورتی شدنی است که رشد g > رشد f

رشد تابع $g(n)$ از رشد تابع $f(n)$ بیشتر است. حد بالای اکید است.

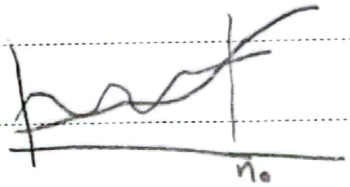
$f(n) \in o(n^2)$

↳ تابع $f(n)$ رشدش به n^2 نمی‌رسد. کمتر از n^2 است

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

* رشد تابع به معنی مقدار نیست

$15n^2 + 100 \in 3n^3$



از یک جایی به بعد مقدارش بیشتر می‌شود.

* $f(n) = 5n^2 + n \longrightarrow$ رشدش برابر جمله‌ای که بیشترین رشد را دارد.

وقتی برای تابعی حد بالا در نظر می‌گیریم ← حد بالای حد بالا می‌تواند حد بالای آن تابع هم باشد.

$5n^2 + n \in O(n^2)$

$n^2 \in O(n^3)$ $\longrightarrow$ $5n^2 + n \in O(n^3)$

برای حد پایین هم برقرار است.

$5n^2 + n \in O(n^2)$

$\in \Omega(n)$

$\in \Omega(\log n)$

← در حد بالا ← حد بالایی که کمترین رشد را دارد و

در حد پایین ← حد پایینی که بیشترین رشد را دارد

← استفاده می‌کنیم ← تا بتوان تعیین تر تابع را بیان کرد

ω:

برای مجموعه ← تعلق استفاده می‌شود

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

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

بزرگ‌تر مساوی هم نیست هم این است برای هر c برقرار باشد.

رشد g کمتر از f(n) است.

$f(n) \quad f(n) + o(f(n))$

همه‌ی توابعی که رشدشان از f(n) کمتر است

رشدش f(n) است.

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

$\omega(g(n)) \longrightarrow$ مجموعه‌ی توابع است که رشدشان بیشتر از g(n) است ← g(n) حد پایین آن است.

$f(n) \in \omega(g(n))$

f رشدش بیشتر از g است.

$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

$f(n) \notin \Omega(g(n))$

صعودی اکید

$g(n)$

$f(n)$

$g(n) \in \Omega(f(n))$

$f(n), g(n) \rightarrow$

$a, b \in R$ $\quad a<b$ $\quad a=b$ $\quad a>b$

ولی برای دو تابع نمی‌توان گفت فقط یکی موارد زیر است

چون ممکن است نشستان هم رابطه نداشته و

متغیر است

مثال

$f(n) \in o(g(n))$

$f(n) \in \Theta(g(n))$ $\quad n^{1+\sin n}$ $\quad e^n$

$f(n) \in \omega(g(n))$

حل روابط بازگشتی:

divide & Conqu ← الگوریتم، رابطه بازگشتی دارد

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

تابع مولد

recergen tree - master teorym

theorym

Recursion Tree ← درخت بازگشتی ← بالاترین سطح

درخت $T(n)$ ← دو مسئله به سایز $T(\frac{n}{2})$

حل می‌کند و $\Theta(n)$ زمان ترکیب و تقسیم

زیرمسائل

The muster theorem

قضیه‌ی اصلی

روش‌های حل روابط بازگشتی

PAPCO

۳ گام روش divide & conquer :

۱- گام تقسیم

۲- گام حل بازگشتی ← $2T(n/2)$

۳- گام ترکیب

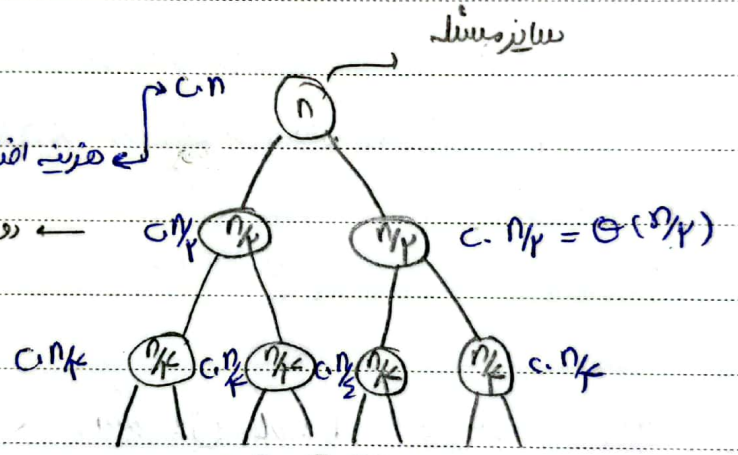
$\Theta(n)$

merge sort ⟷ $\Theta(1)$ گام تقسیم

$\Theta(n)$ گام ترکیب

بازکردن رابطه بازگشتی روی درخت بازگشت

در سطح مسئله به اندازه $T(n)$ و زیرمسئله به اندازه $\frac{n}{2}$ داد و یک $\Theta(n)$ داریم که هزینه اش $n \times$ ثابت $= cn$



هر node دو شاخه می خورد ← هر شاخه سایز نصف می شود تا جایی که به ۱ برسد.

برگ ها ← c' c' c' c'

← به شرط خاتمه بازگشت رسیده ایم در آخر ← حال حل مستقیم می کنیم ← هزینه ثابت داریم که به ما می دهند $T(1) = c'$

← رابطه را از حالت پایه به دست می آوریم

در بقیه از رابطه بازگشتی به دست می آوریم.

* هزینه های کلی ← جمع هزینه هایی که کنار گره ها نوشتیم.

PAPCO

CamScanner

معمولاً سطر به سطر جمع می‌زنیم بعد همه‌ی سطرها را جمع می‌کنیم:

سطر	هزینه
۱	cn
۲	$\frac{cn}{2}+\frac{cn}{2}=cn$
۳	$4\times\frac{cn}{4}=cn$
⋮	
$\log_2^n$ *	cn

هر سطر پایین می‌رویم تعداد گره‌ها دو برابر می‌شود هزینه نصف می‌شود ← ثابت می‌ماند ← هر سطر cn

• باید ببینیم تا کجا ادامه دارد؟ تقسیم بر ۲ ها کی به ۱ می‌رسد

$i=1$	سطر ۱	$\frac{cn}{2^0}$
$i=2$	سطر ۲	$\frac{cn}{2^1}$
$i=3$		$\frac{cn}{2^2}$
⋮		
$i=?$	در برگ‌ها	

→ خروجی 2^k است هر وقت برگ‌ها ۲ برابر می‌شه

سایزش می‌شه $\frac{n}{2^k}=1 \rightarrow k=\log_2^n$

چقدر است $k=i-1$ است

در برگ این عبارت ۱ می‌شود ← $\frac{n}{2^{i-1}}$ سایز مسئله در سطر i ام

$$1=\frac{n}{2^{i-1}} \rightarrow 2^{i-1}=n$$

$$i-1=\log_2^n \rightarrow i=\log_2^n+1$$ در برگ‌ها

شماره سطر یکی بالاتر از برگ‌ها $\log_2^n$ ← تعداد سطرها بدون سطر برگ‌ها $\log^n$ است *

← هزینه‌ی گره‌های غیر برگ برابر است با $\log_2^n \times cn$

هزینه سطر برگ‌ها

تعداد گره‌ها در سطر i ام $= 2^{i-1}$

تعداد برگ ها = $2^{\text{شماره طبقه برگ} - 1} = 2^{\log_2 n + 1 - 1} = 2^{\log_2 n} = n$

هزینه‌ی برگ‌ها = {تعداد برگ} (هزینه‌ی یک برگ) $= nc'$

$\longrightarrow T(n) =$ هزینه‌ی گره‌های غیربرگ + هزینه‌ی برگ‌ها

$$= \underbrace{c \cdot n \log_2 n}_{\text{مشترک بیشتر}} + c' \cdot n \in \boxed{\Theta(n \log_2 n) \leftarrow \text{merge-Sort}}$$

اگر رابطه بازگشتی را بازکنیم همان درخت بازگشتی می‌شود

$$T(n) = 2T(\tfrac{n}{2}) + \Theta(n) \longrightarrow 2\left[2T(\tfrac{n}{4}) + \Theta(\tfrac{n}{2})\right] + \Theta(n)$$

$$= 4T(\tfrac{n}{4}) + 2\Theta(\tfrac{n}{2}) + \Theta(n)$$

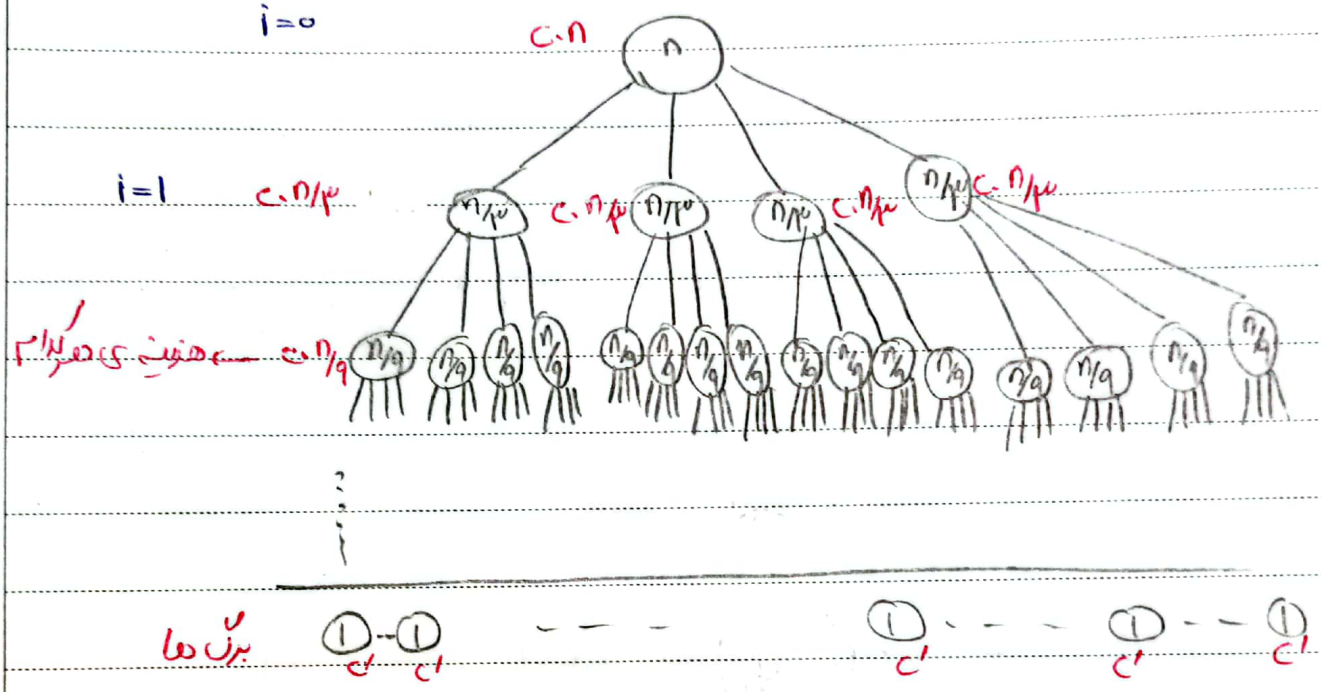
$$= \underbrace{4\left[2T(\tfrac{n}{8}) + \Theta(\tfrac{n}{4})\right]}_{} + 2\Theta(\tfrac{n}{2}) + \Theta(n)$$

هرجا اینجا بعد هزینه‌ی برگ های من می‌شود
ضریب آن هم تعداد برگ هاست.

مثال:

$T(1) = c'$

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



* سایز مسئله می تقسیم بر ۳ می شود تا به ۱ برسد
* تعداد برگ ها لزوماً n نیست بستگی به رابطه بازگشتی دارد.

سطح	هزینه
1	$c.n$
2	$4c\frac{n}{3}$
3	$16c\frac{n}{9}$
⋮	⋮
$\log_3^n$	
$i=\log_3^n+1$	$(\frac{4}{3})^{i-1} \times n = 3^{i-1} c \frac{n}{3^{i-1}}$

• اندازه زیرمسئله در سطر i ام: $\frac{n}{3^{i-1}}$

• در برگ ها سایز مسئله 1 است

$\frac{n}{3^{i-1}} = 1 \longrightarrow n = 3^{i-1}$

$\log_3^n = i-1$

$\boxed{\log_3^n + 1 = i}$

شماره سطح برگ ها

هزینه‌ی گره‌های غیربرگ $= \sum_{i=1}^{\log_3 n} cn\left(\frac{4}{3}\right)^{i-1}$ ← جملات تصاعد هندسی

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

چون مدام $\left(\frac{4}{3}\right)^{i-1}$ زیاد می‌شود همان $\log_3 n$

$$\frac{q^{n+1} - 1}{q - 1}$$

$$3cn\left(n^{\log_3 \frac{4}{3}} - 1\right) \in \Theta\left(n \cdot n^{\log_3 \frac{4}{3}}\right)$$

$\log_3 \frac{4}{3}$ کمتر از ۱ است

$$\Theta\left(n \cdot n^{\log_3 4 - 1}\right)$$

$$\Theta\left(n^1 \cdot n^{1.26 - 1}\right) = \Theta\left(n^{1.26}\right)$$

جمله آخر

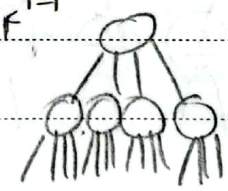
$$\left(\frac{4}{3}\right)^{\log_3 n} - 1$$

$$q^{n+1} = \frac{4}{3}^{\log_3 n}$$

$\log_3 4 = 1.26$

تعداد برگ‌ها 4^{i-1}

تعداد گره‌ها در سطح i ام →



$$4^{i-1} = 4^{\log_3 n + 1 - 1} = 4^{\log_4 n}$$

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

هزینه‌ی برگ‌ها = تعداد برگ × هزینه برگ $= c' n^{\log_3 4} = c' \times n^{1.26}$

کل رابطه بازگشتی → $\Theta\left(n^{1.26}\right)$

۱۵ اسفند:

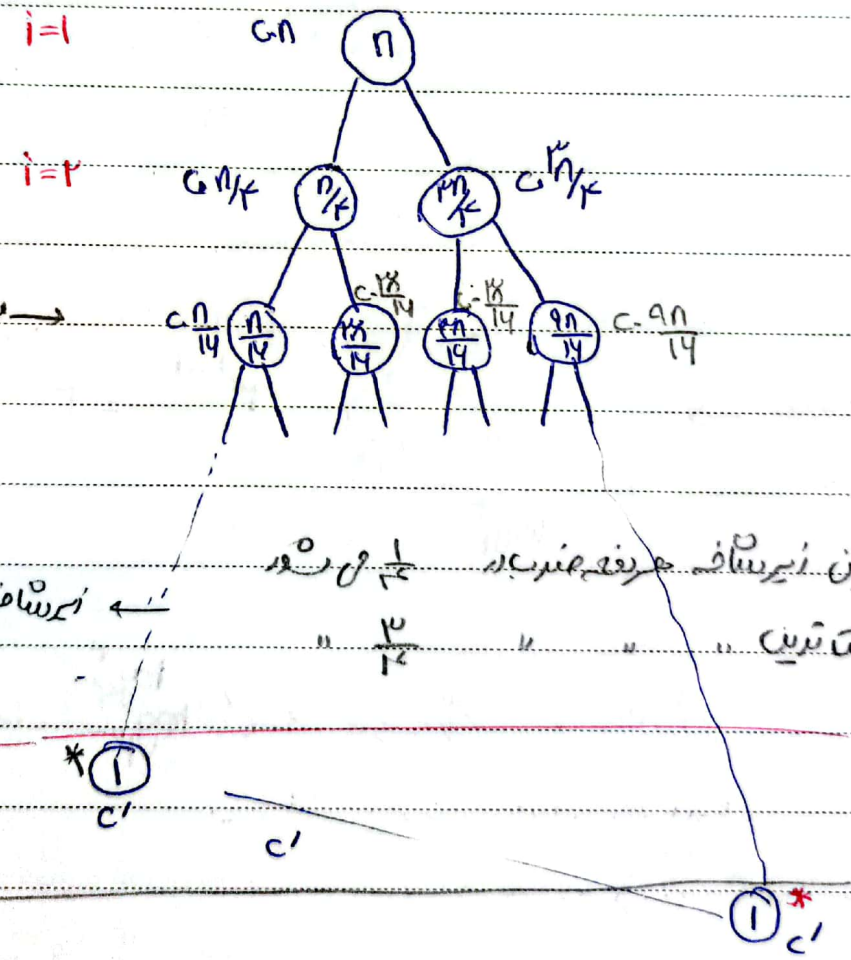
حل روابط بازگشتی در درخت بازگشت ←

مثال: با روش درخت بازگشت می‌خواهیم رابطه زیر را حل کنیم.

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + \Theta(n)$$ در این مسئله با سایز متفاوت

$$T(1) = c'$$

* در فراخوانی بازگشتی در شاخه می‌خورد هر گره ← یکی $\frac{3}{4}$ سایز مسئله و یکی $\frac{1}{4}$ سایز مسئله



i=3 در فراخوانی یکی ←

$\frac{n}{4} \times \frac{n}{4}$

$\frac{n}{4} \times \frac{3n}{4}$

* سمت چپ‌ترین زیرشاخه هر دفعه ضرب در $\frac{1}{4}$ می‌شود

* " راست‌ترین " " " " $\frac{3}{4}$

زیرشاخه سمت چپ اول به ۱ می‌رسد ←

$i = ? \log_4 n$

$i = ? \log_4 n + 1$

← باید هزینه گره‌ها را جمع بزنیم.

هزینه سطر i ام	شماره سطر i
cn	۱
$\frac{cn}{۲}+\frac{۳cn}{۲}=cn$	۲
$\frac{cn}{۱۶}+\frac{۳cn}{۱۶}+\frac{۳cn}{۱۶}+\frac{cn}{۱۶}=cn$	۳
⋮	⋮
cn	۱
⋮	⋮
cn	۱

هزینه‌ی هر گره $\frac{1}{۲}$ و $\frac{۳}{۲}$ از بالا هدایت می‌شود

هزینه فرزندان همان هزینه پدر است ←

جمع هزینه فرزندها = هزینه پدر تا پایین

cn می‌ماند ← تا جایی که به اولین برگ برسیم

سایز مسئله در سطح i ام درست چپ‌ترین گره $\frac{n}{۲^{i-1}}$

سایز مسئله در برگ $=1 \longrightarrow \frac{n}{۲^{i-1}}=۲ \longrightarrow$ ارتفاع برگ $=\log_۲ n+1$

حساب کنیم در ادامه * چه اتفاقی می‌افتد؟ ۱- هزینه پایین را حساب کنیم

۲- حد بالا و حد پایین هزینه را حساب کنیم.

فرض کنیم درخت در * تمام می‌شود فرض کنیم درخت در * تمام می‌شود

* محاسبه‌ی حد پایین هزینه‌ها: در این حالت فرض می‌کنیم ارتفاع کل درخت با ارتفاع سمت چپ‌ترین شاخه مساوی است. هزینه‌ی هر سطح cn ← تعداد سطح‌ها هم $\log_۲ n$

هزینه سطح‌ها $=(\log_۲ n)\cdot cn \longrightarrow cn\cdot\log_۲ n$

PAPCO

CamScanner

هزینه‌ی برگ × تعداد برگ = هزینه‌ی برگ‌ها

هر سطح تعداد گره‌ها چند تاست؟

هر سطح پایین می‌آییم گره‌ها ۲ برابر می‌شود.

تعداد گره‌ها در سطر $= 2^{i-1}$

تعداد برگ‌ها در سطر $= 2^{(\log_2^n + 1) - 1}$

$i = \log_2^n + 1$

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

هزینه برگ‌ها $= c' \sqrt{n}$

هزینه‌ی هر برگ

* تعداد برگ‌ها در نهایت باید n شود زیرا در هر سطح بعد ← فرزندان می‌رسانند و جمع فرزندان برابر پدر است ← در آخر، برگ‌ها ① هستند جمع ①ها باید n شود ← n تا برگ داریم

حد پایین $T(n) = cn\log_2^n + c'\sqrt{n}$

$\in \Theta(n\log^n)$ *

یادآوری $\log_a^n \in \Theta(\log_b^n)$

$\log_a^n = \log_b^n \log_a^b$ ثابت

این دو فقط در یک ثابت فرق دارند.

محاسبه‌ی حد بالای هزینه‌ها:

در این حالت فرض می‌کنیم ارتفاع درخت مساوی ارتفاع سمت راست‌ترین شاخه درخت است.

سایز مسئله در سمت راست‌ترین شاخه $= \left(\frac{3}{4}\right)^{i-1} \times n$

در برگ سایز مسئله $= 1$

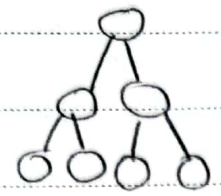
$\left(\frac{3}{4}\right)^{i-1} \times n = 1 \longrightarrow i = \log_{\frac{4}{3}}^n + 1$

* حد پایین اش $n\log^n$ است ← خودش بیشتر یا مساوی $n\log^n$

$T(n) \in \Omega(\underbrace{\log^n \times n}_{n\log^n})$

PAPCO

CamScanner



① $i = \log_{\frac{4}{3}} n$

① $i = \log_{\frac{4}{3}} n + 1$

هزینه‌ی سطح‌ها : $cn \times \log_{\frac{4}{3}} n$

به دلیل اینکه جمع فرزندان ۱ است

تعداد برگ‌ها در ردیف آخر $= 2^{\log_{\frac{4}{3}} n + 1 - 1}$

$= n^{\underbrace{\log_{\frac{4}{3}} 2}_{2.43}}$

$\longrightarrow$ هزینه‌ی برگ‌ها $c' \cdot n^{\log_{\frac{4}{3}} 2}$

$T(n)$ صدا زدن $= cn \log_{\frac{4}{3}} n + c' n^{\log_{\frac{4}{3}} 2} = n \log n + \underbrace{n^{1+\epsilon}}_{\text{جمله غالب}}$ $\quad \epsilon = 1.43$

$\in \Theta(n^{\log_{\frac{4}{3}} 2})$ *

$(\log_a n)^c \in O(n^{\epsilon})$

$0 < \epsilon < 1$

تابع لگاریتمی < تابع چند جمله‌ای < تابع نمایی

هرچه قدر هم بزرگ باشد ولی n^{ϵ} بیشتر است ← هرچه توان کوچک‌تر است توان + باشد در n^{ϵ}

* برای محاسبه دقیق ← اینجا جایی که کم یا زیاد فرض کردیم دقیق فرض کنیم.

① شد $T(n)$ باشد حد بالایش برابر است ← بالاتر را می‌توان اثبات کرد؟؟

* $\rightarrow T(n) \in O(n^{\log_{\frac{4}{3}} 2})$

PAPCO

CamScanner

① تابع تقسیم شده ← زیرمسئله که جمع سایزشان = n شود و $\Theta(n)$ هم n باشد

← چون می‌شود یکی از این دو تا ضرب در یک $\log n$

$\frac{3n}{4} + \frac{n}{4} = n$

درس: حد بالایی $n\log n$ ثابت کنیم $T(n) \in O(n\log n)$

حد بالا $n\log n$ ← ثابت می‌شود یعنی Θ است

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

میانگین برای وقتی که تابع بین دو حالت نوسان می‌کند.

اثبات با استقرا:

پایه برای استقرا و حد حالت گذار $T(1) = c'$

* ← رابطه برای چه n می‌تواند درست باشد ← if $n=2$ $\quad T(2) \leq c \cdot 2\log_2 2 \leq c \cdot 2$

باید $T(2)$ را داشت تا بدانیم به ازای چه c برقرار است

$n=2 \quad T(2) = \underbrace{T(\frac{2}{4})}_{\text{صفر}} + \underbrace{T(\frac{6}{4})}_{\text{رد می‌شود چرا}} + \underbrace{\Theta(n)}_{\text{ثابت}} = c''$ یک ثابت "c که مقدارش مشخص نیست

چون توابع عدد صحیح اند

$c'' \leq 2 \cdot c$

$c \geq \frac{c''}{2} \left(\frac{T(2)}{2}\right)$ ← رابطه * برقرار می‌شود برای $n=2$

اگر این رابطه باشد پایه $(n=2)$ برقرار است.

حالت گذار: استقرای قوی باید استفاده کنیم

فرض: $T(\frac{n}{4}) \leq c(\log^{n/4}) \times \frac{n}{4}$

$T(\frac{3n}{4}) \leq c(\log^{\frac{3n}{4}}) \frac{3n}{4}$

← ثابت کنیم

$T(n) \leq c \cdot n \cdot \log_2^n$

$$T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + \Theta(n) \leq c \cdot \frac{n}{4} \log^{n/4} + c \cdot \frac{3n}{4} \log^{3n/4} + \Theta(n)$$

$$\leq c \frac{n}{4}(\log^n - \log^4) + c \cdot \frac{3n}{4}(\log^3 - \log^n - \log^4) + \Theta(n)$$

$$\leq c n \log^n \underbrace{- c n \log^4 + c \cdot \frac{3n}{4} \log^3 + c''' \cdot n}$$

برای برقراری حکم باید $\leq c n \log^n \Longrightarrow$ جمع این جملات باید منفی باشد

فاکتور از n

$$-c \log^4 + c \cdot \frac{3n}{4} \log^3 + c''' n =$$

$$-2c + \frac{3}{4} \log_2^3 + c''' < 0$$

$$c \geq \frac{c'''}{2 - \frac{3}{4} \log^3}$$

$\Rightarrow$ * با استقرا ثابت کنید $T(n) \in \Theta$ (O, Θ, Ω) درست

* هزینه در هر سطح مستقل کافی است ضرب در تعداد سطوح

← تعداد هفتی

درخت ارث ← حد بالا و پایین

← یک جایی تغییر متغیر لازم است قبل recursion tree حل کنیم بعد تغییر متغیر را حل کنیم

استقرا

قوی: از روی همه‌ی قبلی‌ها بعدی را اثبات می‌کنیم

ضعیف: از روی قبلی n را اثبات می‌کنیم

PAPCO

CamScanner

"c" ← ثابت که ضرب در n می‌شود O(n) را می‌سازد وقتی می‌خواهیم مسئله n را حل کنیم

می‌خواستیم یک c پیدا کنیم که رابطه برقرار باشد ← cهای زیادی وجود دارند که رابطه برقرار است ولی یکی هم باشد کافی است

روش recursion محاسباتش زیاد است روش master یک سری شرایط دارد

روش قضیه‌ی اصلی (برای حل روابط بازگشتی) The master theorem

برای بعضی روابط بازگشتی کار می‌کند که فرم‌های زیر را دارند

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

برای حل مسئله به اندازه n باید a تا مسئله $\frac{n}{b}$ را حل کنی ← زیرمسئله‌ها سایز برابر دارند.

سه حالت:

الف) اگر $\exists \varepsilon > 0$ به طوری که $f(n) \in O(n^{\log_b a - \varepsilon})$ آنگاه $T(n) \in \Theta(n^{\log_b a})$

$n^{\log_b a}$ رشدش از f بیشتر است حتی اگر ε از توانش کم کنیم ← به صورت چندجمله‌ای رشدش بیشتر است ← که همچنان صد بالای f(n) است ← جواب می‌شود $n^{\log_b a}$

قضیه‌ی اصلی f(n) یکی $a^{\log_b n}$ را مقایسه کن

f ? $n^{\log_b a}$

رشد f بیشتر از یا مساوی از یا کمتر است

$n^{\log_b a - \varepsilon} = \frac{n^{\log_b a}}{n^{\varepsilon}}$

ب) $f(n)$ با $n^{\log_b^a}$ با هم رشدشان یکی شد ← یکی را یکبار ضرب در $\log^n$ کن

یا شود رشد $f(n)$

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

شرط منظم بودن

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

آنگاه $T(n) \in \Theta(f(n))$

$f(n)$ رشدش بیشتر $\varepsilon > 0$ اگر وجود داشته باشیم $f(n)$ نه تنها از $n^{\log_b^a}$ بیشتر است بلکه

به توان ε هم اضافه کنیم باز هم بیشتر ← رشد ← صورت چندجمله‌ای بیشتر است

$f(n)$ و $n^{\log_b^a}$ ⟶ رشد $f(n) \leftarrow ② < ①$

① ②

$n^{\log_b^a} \leftarrow ① > ②$

$\log^n \times$ یکی از این دو $\leftarrow ① = ②$

۲) اصل مادر:

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

الف) اگر $f(n) \in O(n^{\log_b^a - \varepsilon})$ ، $\exists \varepsilon$ آنگاه $T(n) \in \Theta(n^{\log_b^a})$

یک ε باشد که

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

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

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

مثال ۱:

$T(n) = 4T(\frac{n}{2}) + \Theta(n)$

$f(n)$ با $n^{\log_b^a}$ باید مقایسه شود

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

$a = 4, b = 2$

حالت الف)

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

بررسی حالت الف)

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

$\in O(n^{2-\varepsilon}) \rightarrow 2 - \varepsilon \geq 1$

$\varepsilon \leq 1$

شرط الف قضیه‌ی اصلی برقرار است پس

* ذهنی هم می‌توان همین‌ها را حساب کرد.

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

$T(n) \in \Theta(n^{\log_2^4})$

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

مثال ۲: $T(n) = \overset{a}{2}T(\frac{n}{\underset{b}{2}}) + n^2$

رشد f(n) بیشتر است ← حالت سوم فرض می‌شود درست است

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

بررسی حالت ج مقدمه ی اصلی

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

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

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

$1+\varepsilon \leq 2 \longrightarrow 0 < \varepsilon \leq 1$

شرط منظم بودن پیدا کردن c:

$af(\frac{n}{b}) = 2f(\frac{n}{2}) = 2(\frac{n}{2})^2 = \frac{n^2}{2} \leq c.f(n) = cn^2$

* اگر رشد f(n) بیشتر است رشد T(n) می‌شود همان f(n)

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

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

$\in \Theta(n^2)$

توضیح:

مشخص است n^2 رشدش بیشتر از $n^{\log_2^2}$ است ← مسلماً ε وجود دارد

ولی باید c را حساب کرد.

مثال ۳: $T(n) = \overset{a}{3}T(\frac{n}{\underset{b}{3}}) + \Theta(n)$

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

حالت ب می‌شود.

$f(n) \in \Theta(n) \in \Theta(n^{\log_b^a} = n^{\log_3^3}) \rightarrow T(n) \in \Theta(n^{\log_b^a} . \log^n)$

$\Theta(n^1 . \log^n)$

PAPCO

CamScanner

مثال ۴:

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

* اثبات قضیه ی اصلی با recursion tree

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

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

رشدش بیشتر

حالت ج قضیه ی اصلی می شود ← بررسی حالت ج

$f(n) \in \Theta(n\log^n) \in \Omega(n^{\log_b^a + \varepsilon}) \in \Omega(n^{\log_2^2 + \varepsilon})$

برای ε های مثبت نفی شد متقابلاً برای ε های ۰ یا منفی می شود که جزو شروط نیست.

باید $\exists \varepsilon > 0$ که ؟

$\nexists \varepsilon > 0 \quad ? \quad \log^n \in \Omega(n^{\varepsilon})$

وجود ندارد

* رشد هر تابع log از هر تابع چندجمله ای کمتر است

← با حالت ج حل نمی شود با توجه به اینکه مطلقاً نفی شود ← رابطه را نمی توان با قضیه اصلی حل کرد

همه ی مسائل را نمی شود با قضیه ی اصلی حل کرد ← حتماً باید زیر مشکل ها سایز برابر باشند ولی باز هم ممکن است حل نشود

extend کردن قضیه ی اصلی که حالت مثال ۴ را هم در بر بگیرد

← با ← می توان extend کرد ← اینکه

$f(n) \in \Theta(n^{\log_b^a} \cdot (\log^n)^k)$ اگر ←

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

$\log^{(k)} n = \underbrace{\log(\log(\log(\cdots)))}_{k \text{ بار } \log \text{ می‌گیریم}}$

← حال می‌توان مثال ۲ را حل کرد.

$f(n)$ رشد از $n^{\log_b^a}$ بیشتر است اما به اندازه یک توانی از $\log^n$

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

$\rightarrow k=1 \quad a=b=2 \quad \rightarrow f(n) = n\log^n \in \Theta(n^{\log_b^a} \cdot (\log^n)^1)$

حالت (ب) قضیه‌ی اصلی (توسعه داده شده): $T(n) \in \Theta(n^{\log_b^a} \cdot (\log^n)^{k+1})$

$\in \Theta(n^{\log_2^2} \cdot (\log^n)^2) \in \Theta(n \cdot (\log^n)^2)$

در recursion محاسبات بیشتری داشتیم. اینجا فقط کافی است که یک سری حالت‌ها را طی کنیم ← بهتر است در صورتی که رابطه اصلی با قضیه‌ی اصلی قابل حل باشد.

تحلیل زمانی - اثبات درستی - بررسی الگوریتم مرتب سازی heap quick sort ← تحلیل زمانی و اثبات درستی.

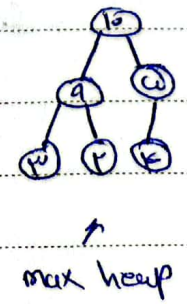
تحلیل الگوریتم های مرتب سازی. (Heap Sort, Quick Sort)

Heap Sort: heap مبنا براساس

max / min heap

min ↓ مقدار فرزندان ≤ پدر

max ↓ مقدار فرزندان ≥ پدر



درخت (باینری) کامل

به جز سطح آخر که سطح آخر از سمت راست پر نباشد

max heap ← ترتیبی بین فرزند چپ و راست نیست.

heap در sort کمک می‌کند ← در max heap عنصر max ریشه است

در مرتب‌سازی صعودی آخرین عنصر بزرگ‌ترین

عنصر ← ریشه heap

هر بار عنصر max را می‌گذاریم در انتها اعداد ←

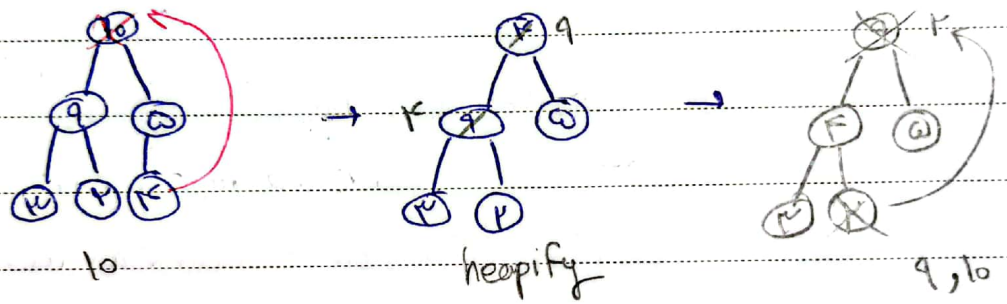
یکی از اعداد مرتب‌شده صعودی است می‌شود.

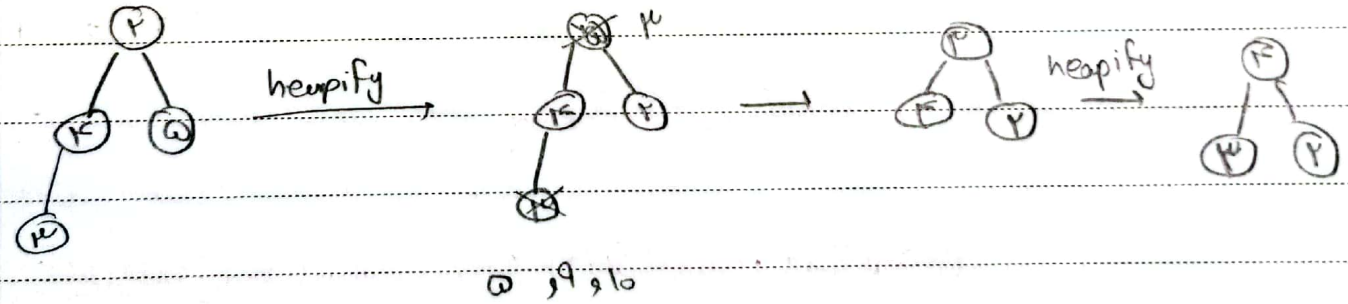
← بجایش سمت راست‌ترین برگ را می‌گذاریم.

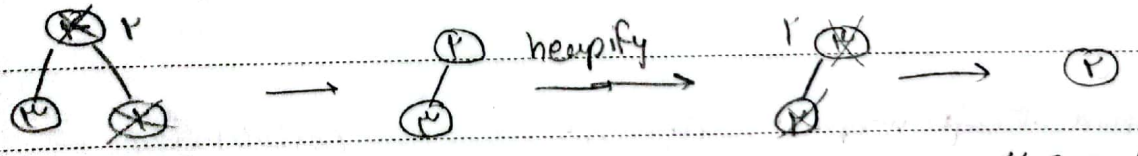
← سمت ۱ heapify می‌کنیم ← سمت

با ۲ فرزند ← max های ریشه ← ریشه هر کدام

عوض شد بین آن و فرزندان max می‌گیریم.







→ ۲,۳,۴,۵,۹,۱۰

ایده: در max heap عنصر max را داریم ← در مرتب شده عنصر آخر عنصر را داریم ← بر می‌داریم با راست‌ترین برگ جابه‌جا می‌کنیم ← heapify می‌کنیم و این کار تکرار می‌شود...

(heap-size) سایز آرایه

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, 1)
```

آرایه را می‌گیرد heap می‌سازد

از وسط آرایه به آخر برگ می‌روند

از وسط به بالا باید heapify را صدا بزنیم

$\frac{n}{2}\log^{n}$

ارجاع اندیسی که heapify

(1) یعنی از ریشه

heapify ← فرض می‌کنیم درخت راست و چپ heap هست ولی کل درخت نیست

heap sort ← distinct بودن اعداد وابسته نیست.

* عنصر آخر آرایه را با اول جابه‌جا می‌کنیم

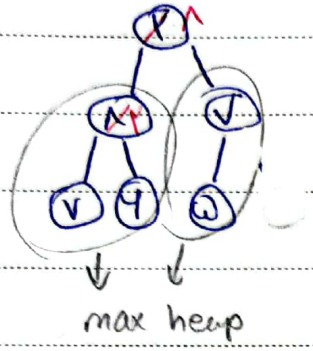
با جابه‌جا کردن با عنصر آخر آرایه دیگر کاری نداریم

⇐ heap size را کم می‌کنیم

۲ ← سایز heap = ۱ می‌شود ← heapify می‌شود خودش وظایفش قائم می‌شود

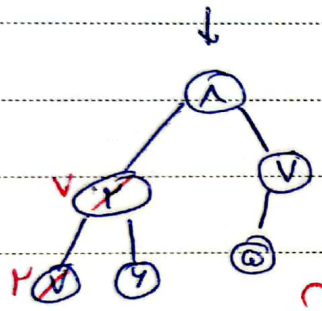
max_heapify ← زیردرخت راست و چپ ← heap است ولی کل درخت مشکل است نباشد

مثال:



← ریشه را با فرزندان چک کنیم اگر بزرگتر از ریشه بیاوریم جای فرزند جابه جا شده دوباره heapify می کنیم.

* درخت heap را می توان در آرایه چید سطح به سطح از بالا به پایین از چپ به راست در آرایه می چینیم



→ A | ۸ | ۷ | ۷ | ۲ | ۶ | ۵ |

ریشه

رابطه پدر فرزندان حفظ می شود.

ریشه: خانه ی ۱

هر خانه ی i ← فرزندان در (فرزند چپ) $2i$ قرار دارند.
(اندیس فرزند راست) $2i+1$

* اگر فرزند راست نداشته باشیم اندیس $2i+1$ از سایز heap بیشتر می شود.

اندازه heap: heap-size = ۶

length = ۷
طول آرایه

اندازه heap حداکثر برابر طول آرایه است.

آرایه بهتر: pointer نداریم ← هزینه حافظه ای کمتر ← heap را در آرایه می بریزیم

به دست آوردن اندیس پدر گره i: $\lfloor \frac{i}{2} \rfloor$

MAX_Heapify(A, i)

آرایه — اندیس ریشه

heapify محل شروع
i=1 → ریشه

* بار اول ورودی که میدیم چون ما الان خودش را صدا می زند

1. .....

2. if L ≤ heapsize → اطمینان از وجود فرزند چپ

```
MAX_Heapify ( A , i )
1. l = 2i , r = 2i+1
2. if l ≤ heap-size and A[l] > A[i]
3.     largest = l
4. else : largest = i
5. if r ≤ heap_size and A[r] > A[largest]
6.     largest = r
7. if largest != i
8.     swap ( A[i] , A[largest] )
9.     MAX_Heapify (A[i], A[largest])
```

فرزند چپ دارد

max بین پدر و راست و فرزند.

هزینه heap است

خط ۱ تا ۸ ← یک مقدار ثابت

خط ۹ ← recursive ← ① رابطه بازگشتی بنویسیم

② در هر سطر هزینه ی ثابت صرف می کند هزینه کل الگوریتم

$\frac{\text{تعداد سطرها}}{\log n} \times \frac{\text{هزینه ثابت}}{\text{ثابت}}$

تحلیل الگوریتم مرتب سازی heap Sort :

```
Heap_Sort(A,n)
1- Build_MAX_Heap(A,n)
2- for i=n to 2 do
3-     swap(A[i], A[1])
4-     heap_size --
5-     MAX_Heapify(A,1)
```

ورودی زیردخت راست و چپ heap خودش نیست.

آرایه ورودی — heapify محل شروع (اندیس ریشه)

```
MAX_Heapify(A, i)
1- l=2i , r=2i+1;
2- if l ≤ heap_size and A[l]>A[i]
3-     largest=l
4- else largest=i
5- if r ≤ heap_size and A[r]>A[largest]
6-     largest=r
7- if largest != i
8-     swap(A[largest], A[i])
9-     MAX_Heapify(A, largest)
```

* اندیس وجود داشته باشد

build_MAX heap ← از heapify استفاده می‌کند تا آرایه‌ی داده‌شده را heapify کند ← باید

نیمی از آرایه را heap کند چون نیمه‌ی دوم به عنوان برگ است و خودش

heap است ← تا به ریشه برسیم و heap کنیم ← کل درخت heap

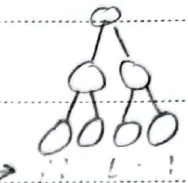
می‌شود.

تعداد عناصر

Build_MAX_Heap(A, n)

1. heap_size = n
2. for $i = \lfloor \frac{n}{2} \rfloor$ to 1 do
3. MAX_Heapify(A, i)

از اینجا heap را می‌سازیم به بالا می‌رویم



برگ‌ها خود heap هستند

max Heapify:

تحلیل زمانی

از ۱ تا ۸ ← هزینه ثابت

تا می‌رسد به ۹

زمان $T(n) \in O(h \times \text{ثابت})$

MAX_Heapify $\in O(h)$ $\in O(\log^n)$

رفت به بالا

است ← درخت طول است اختلاف دو برگ حداکثر ۱ است

$h = \log^n$ ←

روش تحلیل زمانی دیگر برای max_heapify: هزینه MAX_Heapify در $T(n)$ ← تا برسیم تا ۹ که هزینه ثابت است

بدترین حالت زمان اجرای MAX_Heapify

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

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

و ثابت Θ هم هستند

حالت ۲ قضیه‌ی اصلی می‌شود ← می‌شود نتیجه: یکی از این دو ضرب در $\log^n$

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

زمان max heapify ← $\Theta(\log^n)$ نیست بدترین حالتش

$O(\log^n)$ ← این است

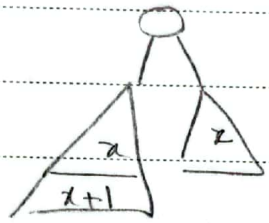
زمان بهترین حالت $\in \Theta(1) \longrightarrow T(n) \in \Omega(1)$

تقریب بزنیم می‌شود $T(n) = c + T(\frac{n}{2})$ ولی درخت heap معلق است

پر نباشد ← ممکن است در زیردرختی بیاندازیم که تعداد عناصرش بیشتر است ←

فرض خواهیم برای زیردرخت بزرگ‌تر می‌شود ← سایز زیردرخت چپ: $\frac{2}{3}$

درخت کامل باینری: تعداد برگ‌ها = ۱ + گره داخلی ← تقریباً نصف گره‌ها برگ‌ها هستند



$n = 2x + 1 + x + 1 = 3x + 2$

$x = \frac{n-2}{3}$ $\qquad 2x+1 = \frac{2}{3}(n-2) + 1 = \frac{2n}{3} - \frac{1}{3}$

← زیردرخت چپ من حداقل $\frac{2}{3}$ است

$T(n) = c + \underbrace{T(\frac{2n}{3})}_{*}$ در حالی که بقیه درخت اول اگر $\frac{2}{3}$ باشد درخت‌های بعدی $\frac{1}{2}$ می‌شود.

$a = 1 \quad b = 3/2 \quad f(n) = c \quad ? \quad n^{\log_{3/2} 1} = 1$

حالت دوم قضیه اصلی ← $T(n) = \Theta(\log^n)$ حالت دوم قضیه اصلی.

* هنگامی که n تقسیم بر عدد بزرگ‌تر از ۱ باشد ← $\log^n$ به ما می‌دهد

Build - MAX - Heap

۱. c_1

۲. هزینه c و تعداد $n/2$ بار $\rightarrow \frac{n}{2} c_2 \rightarrow$ زمان Build MAX-Heap $= O(n\log^n)$

۳. $\frac{n}{2} \times O(\log^n) = O(n\log^n)$

۴.

با تحلیل دقیق‌تر می‌توان نشان داد $O(n)$ است ← چون MAX-Heapify به ارتفاع درخت ربط دارد و برای node های مختلف فرق دارد. از $c_1 \log^n$ ادامه می‌یابد.

PAPCO

CamScanner

تعداد گره‌های با ارتفاع ۱ زیاد است و با ارتفاع $\log^n$ تعدادشان کم است

یک آرایه‌ی دلخواه را می‌توان در $O(n)$ به heap تبدیل کرد

Heap_Sort (A, n)

۱- $O(n)$

۲- $(n-1) \times c_1$

۳- } $(n-1) \times c_2$ → $T(n) \in O(n \log^n)$

۴- heap-sort زمان

۵- $(n-1) \log^n$ بدترین حالت زمان اجراست.

بهترین حالت

Build-MAX-heap for → n $\in \Theta(n)$

MAX-heapify → 1

← هم حد بالا هم پایین n است ← شش $\Theta(n)$ است.

heap-Sort (A, n)

۱- $\Theta(n)$

۲- $c_1(n-1)$ → زمان بهترین حالت $\in \Theta(n)$ heap-Sort

۳- } $c_2(n-1)$

۴-

۵- $n \times \Theta(1)$

بهترین حالت

MAX-Heapify

فقط در صورتی که همه‌ی گره‌ها عناصر با هم مساوی باشند.

برگ را وقتی می‌گذاریم بالا جز کوچک‌ترین اعضای درخت نمی‌تواند از بقیه‌ی عناصر بزرگ‌تر باشد.

اگر عناصر مساوی نباشند چه رخ می‌دهد؟

یکی از بزرگ کوچک‌تر ← پدرش از پدرش کوچک‌تر وقتی برگ بالای درخت می‌نشیند تا پایین درخت می‌آید ← هزینه $n\log^n$ می‌شود و بدترین حالت رخ می‌دهد وقتی عناصر مساوی نیستند

فقط ⊖ heap-Sort.

$$\sum_{i=n}^{1} \log i = \log^{n!} \in \Theta(n\log^n)$$

یک یکی کم می‌شه از آرگمان تأثیری ندارد.

اثبات درستی:

از MAX-Heapify شروع می‌کنیم بعد build ← بعد heapSort

recursive ← با استقرا

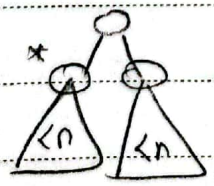
پایه را ۱ می‌گیریم ← $i=1$ ← هزینه ثابت است

اثبات درستی MAX-Heapify:

$n=1$ (heap-size = 1)

اگر دیدیم پس از تعداد 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 زیردرخت چپ و راست آن max heap است درست کار می‌کند.



اگر * max باشد بایستی جابه‌جا می‌شد و چون * بزرگ‌ترین عنصر سمت چپ بوده ← بازهم همه عناصر زیردرخت چپ کوچک‌تر از ریشه خواهند بود

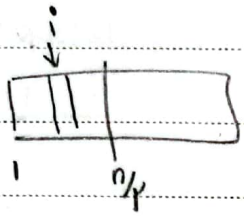
زیردرخت چپ و راست ریشه را ندارند ⇐ $<n$ است

طبق استقرا قوی ← max-heapify درست کار می‌کند

اثبات Build MAX heap :

بازگشتی نیست ← loop invariant

← در طول اجرای حلقه درست است در انتهای حلقه هم درست است.



هرچی که باشد از $\frac{n}{2}$ شروع می‌شود تا 1 ← هرچی هست از اینجا به بعد همه heap هستند (حالت کلی)

برگ‌ها هم خودشان heap هستند

در ابتدای گام i ام از i+1 تا n همه زیردخت‌ها max heap هستند.

اثبات درستی Build MAX heap :

loop invariant : در ابتدای گام i ام حلقه تمام زیردرخت‌های با ریشه‌ی i+1 تا n یک max heap است.

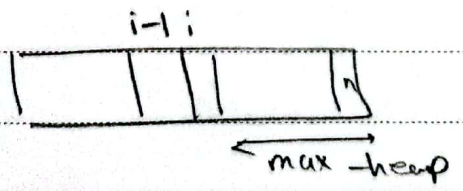
پایه: $i = \frac{n}{2}$ ← از $\frac{n}{2}+1$ تا n همه برگ اند و برگ‌ها max-heap اند.

در این حالت loop invariant برقرار است چون از $i=\frac{n}{2}+1$ تا n همه‌ی زیردرخت‌ها برگ و maxheap هستند.

حالت نگاه: برای i فرض می‌کنیم درست است برای i+1 اثبات می‌کنیم.

حالت نگاه: فرض می‌کنیم برای i ثابت حلقه برقرار است برای i-1 ثابت می‌کنیم که برقرار است.

همه‌ی زیردرخت‌های i+1 به بعد برقرار است.

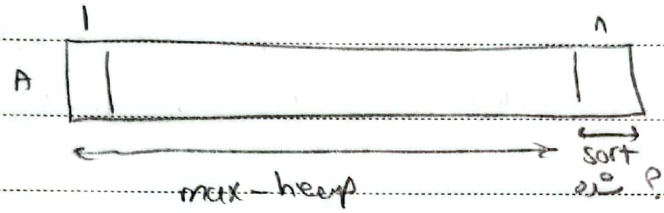


در گام i-1 می‌خواهیم بگوییم i به بعد heap هستند ← زیردرخت چپ و راست i heap هستند ← max heapify را فرض می‌کنیم درست کار می‌کند و قبل صدا زدن max-heapify زیردرخت‌ها heap بوده‌اند وقتی max heapify را صدا می‌زنیم آیا زیردرخت‌ها heap شدن را هم می‌زند؟ خیر.

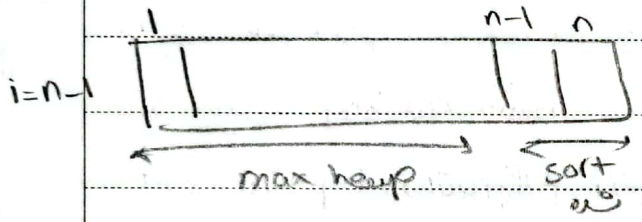
یک درختی که heap است کل زیردرخت ها هم heap خواهند بود.

اثبات درستی heapSort:

loop داریم ← loop invariant ← ویژگی که در طول اجرای حلقه‌ی لوپ برقرار است؟



در سناریو اجرای حلقه شکل زیر چطوری است؟ شده



وقتی i = n است جای n و 1 را عوض می‌کنند heapify می‌کند

در ابتدای گام ام جای 1 را عوض کنیم

از i+1 تا n sort شده است و از 1 تا i ، heap است.

loop invariant: در ابتدای گام ام جلو از i+1 تا n مرتب شده صعودی است و از 1 تا i یک max heap هست.

اثبات جزء

وقتی i=1 است از 2 تا n مرتب شده و 1 max heap است

تحلیل الگوریتم Quick Sort:

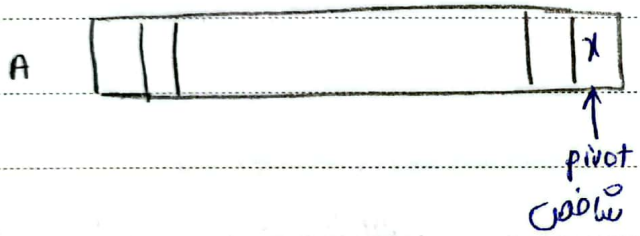
روش divide & conquer ← سه گام دارد: ۱- تقسیم مسئله به زیرمسائل کوچک‌تر

۲- حل بازگشتی زیرمسائل

۳- ترکیب جواب زیرمسائل به دست آوردن جواب مسئله اصلی

Quick sort به روش divide & conquer عمل می‌کند.

آرایه ای از اعداد داریم.



Quick sort هر آرایه را تقسیم می‌کند به دو قسمت مرتب می‌کند و با هم جمع می‌کند

Quick Sort شبیه merge Sort ولی روش تقسیم کردن متفاوت است. یک عدد به اسم شاخص داریم

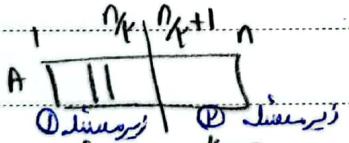
عددهایی که از شاخص کوچک‌تر ← زیرمسئله ۱

زیرمسئله ۲ ← اعداد بزرگ‌تر از شاخص

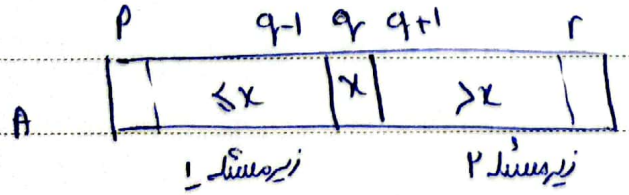
← معمولاً شاخص را عنصر آخر می‌گیرند

← همه‌ی عناصر را با آن مقایسه می‌کنیم

merge Sort



دقیقاً دو زیرمسئله با سایز مساوی



مقایسه اعداد با شاخص و جابه‌جایی کردن ↓ Partition

گام یک روش divide & conquer

← تقسیم مسائل به زیرمسئله

merge Sort	Quick Sort
تقسیم کردن $O(1)$	$O(n)$

PAPCO

CamScanner

گام ۱ (تقسیم) : هزینه Merge Sort > هزینه Quick Sort

بازگشت الگوریتم را ادامه می‌دهیم

p تا q-1

r تا q+1

← حال چگونه ترکیب کنیم؟

در Quick Sort :

← اگر مرتب شود ← کل آرایه مرتب است

نیمه‌ی اول مرتب ← نیمه‌ی اول همه از q کوچک‌تر است

نیمه‌ی دوم مرتب ← نیمه‌ی دوم همه از q بزرگ‌تر است

← نیازی به گام سوم نیست.

۳	۷	۱۰	۱	۵	۱۰

مرتب — مرتب

مرتب نیست

از الگوریتم merge استفاده می‌کنیم.

← شرط خاتمه‌ی بازگشت در Quick Sort ← آرایه‌ی یک‌خانه‌ای است. یا شرط ادامه تا زمانی که $p<r$

اندیس ابتدا — اندیس انتها

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)

$p>r$ ← آرایه پوچ

$p=r$ ← آرایه ۱ای

گام ۱

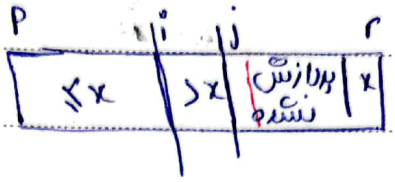
گام ۲

گام ۳ (ترکیب) نداریم.

merge Sort		Quick Sort
$O(1)$ ← گام تقسیم ساده		گام تقسیم پیچیده ← $O(n)$
۲ گام	یکسان	۲ گام
۳ گام ← merge الگوریتم		گام ۳ ندارد ← $O(1)$
$\Theta(n)$		

partition ← مقایسه اعداد با شاخص و جابه‌جایی

Partition



j اولین خانه پردازش نشده

j را با r مقایسه می‌کنیم باید در قسمت دوم اضافه شود ← سر جایش می‌ماند خانه‌ی j ام و j را جلو می‌بریم

اگر کوچک‌تر از شاخص بود باید به قسمت اول بیاید ← اولین خانه‌ی قسمت دوم را با خانه‌ی j ام جابه‌جا می‌کنیم و i را یکی اضافه می‌کنیم و j را هم یکی اضافه می‌کنیم (قسمت دوم شیفت پیدا کرد)

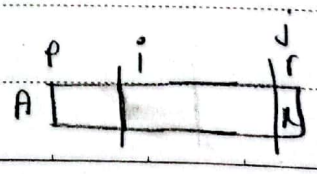
* اول باید طوری set شود که قسمت اول و دوم خالی باشد. i=p j=p

```
Partition(A, p, r)
1-  x = A[r]        ← شاخص
2-  i = j = p   *
3-  for j=p to r-1 do
4-      if A[j] ≤ x then
5-          swap(A[i], A[j])
6-          i++
7-  swap(A[r], A[i])
    return i
```

الگوریتم iterative:

چون j در هر حالت که خانه‌ی j ام از x بزرگ‌تر یا کوچک‌تر باشد i+j می‌شود ← شمارنده حلقه را j می‌گیریم.

بعد از اتمام for:



Quick Sort همان q در الگوریتم

تحلیل زمانی Partition

۱ـ
۲ـ } c_1

۳ـ
۴ـ
۵ـ
۶ـ } $(n-1) \times c_2$

n-1 بار یک کار با هزینه ثابت انجام می‌دهد

Partition(A, 1, n)

↓

$O(n)$

۷ـ
۸ـ } c_3

بهترین حالت:

بدترین حالت:

حالت متوسط:

۳ ⟶ همیشه اجرا می‌شود

به اعداد ربطی ندارد

بهترین حالت: شرط ۴ برقرار نباشد ⟵ خانه‌ی i ام از شاخص کوچک‌تر مساوی

بدترین حالت: شرط ۴ برقرار است. نباشد ⟵ شاخص min باشد

یا آرایه به صورت نزولی باشد.

↓

شاخص maximum است

آرایه به صورت صعودی باشد.

تحلیل الگوریتم Partition:

بهترین حالت: وقتی که x عنصر minimum باشد و شرط $A[j] \le x$ برقرار نیست

بهترین حالت

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

اجرای دستور if

هزینه ثابت

بدترین حالت: ⟵ هزینه ثابت با مقدار بیشتر

$T(n) \in \Omega(n)$ ①

بدترین حالت: شرط if همیشه برقرار باشد.

← x عنصر maximum باشد ← در فراخوانی بازگشتی ← Quick Sort

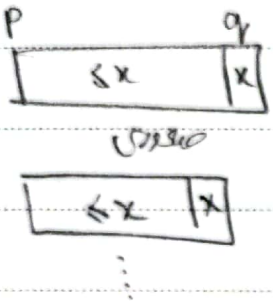
بدترین حالت ← در همه‌ی فراخوانی‌ها

شاخص باید max باشد ←

آرایه مرتب شده صعودی باشد

$$\left(\begin{matrix}\text{بدترین حالت}\\ T(n)\end{matrix}\right) = c_1 + c'_2(n-1) + c_3 \qquad (c'_2 > c_2)$$

$$\in \Theta(n)$$



$T(n) \in O(n)$ ②

آرایه باید مرتب شده صعودی باشد تا در همه‌ی فراخوانی‌ها Partition بدترین حالت باشد.

① ، ② → $T(n) \in \Theta(n)$

در بدترین حالت و بهترین حالت n است ← Partition

میانگینش هم n است. (رشش fix است)

↓

آرایه را صعودی به ما بدهند

بدترین حالت Quick Sort رخ می‌دهد

Quick Sort می‌دهد زیر ابدترین حالت

merge → $\Theta(n)$

partition → $\Theta(n)$

تحلیل زمانی Quick Sort ، Quick-Sort(A, p, r) → T(n) رخ می‌دهد

۱- → c_1

۲- → $\Theta(n)$

۳- → T(تعداد نیمه‌ی اول) = $T(n_1)$ $\qquad n_1 = q - p$

۴- → T(تعداد نیمه‌ی دوم) = $T(n_2)$ $\qquad n_2 = n - n_1 - 1$

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

چون این رابطه بازگشتی بستگی دارد n_1 و n_2 چه کسری از n باشند

فرض کنیم نصف عناصر آرایه کوچک‌تر مساوی x و نصف بزرگ‌تر از x

اگر فرض کنیم $n_1 = n_2 = \frac{n}{2}$ بالانس

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

↓

merge Sort با تعبیر اصلی یا همان رابطه بازگشتی

$\Theta(n\log^n)$

یک مشکلت $n-1$ یک تشکیل میدهد

اگر فرض کنیم $n_1 = n-1$ و $n_2 = 0$ یا $n_1 = 0$ و $n_2 = n-1$ ⟸

$\boxed{x \quad > x}$ $\boxed{< x \quad x}$

$T(n) = \overbrace{\Theta(n) + T(0)}^{\Theta(n)} + T(n-1)$

معمولاً ثابت است

$T(n) = T(n-1) + \Theta(n)$ ⟶ درختی که فقط یک شاخه دارد با n تا node

$= c(n) + c(n-1) + c(n-2) + \dots + c$

$\in \Theta(n^2)$

شد متغیر است بین $n\log^n$ و n^2 ⟵ minmum این تابع همان حالت بالانس است که

$n_1 = n_2 = n/2$ ⟵ بیشترین اختلاف بین n_1 و n_2 ⟵ $T(n) \in \Theta(n^2)$

بدترین حالت

↓

$T(n) \in \Theta(n\log^n)$ بهترین حالت

$T(n) \in \Omega(n\log^n)$ ⟵ اثبات با استقرا

$\in O(n^2)$ رابطه بازگشتی ← ثابت c و n_0

در تعریف Ω را داریم یا خیر.

• در حالت پایه برقرار است $n=2$

• برای همه‌ی مقادیر کمتر n برقرار است

• برای n می‌خواهیم ثابت کنیم.

از کجا معلوم این‌ها بهترین و بدترین حالت هستند؟ قابل اثبات است

← حالت متوسط

← اثبات درستی Partition و QuickSort

اثبات درستی: با استقرا

تابع loop با ثابت حلقه ← استقرا صفیف

تابع بازگشتی ← استقرا قوی

اثبات درستی الگوریتم partition: آرایه آرایه می‌دهیم ← Partition خروجی نهایی به صورت

$\boxed{\leq x \mid x \mid > x}$

می‌شود

خانه‌ی q ← شاخص ← خانه‌ی r آرایه

عناصر کمتر از q ← کوچک‌تر از x

" بزرگتر " ← بزرگ‌تر "

ثابت حلقه: در طول اجرای حلقه‌ی الگوریتم درست است چه چیزی؟

loop invariant: در ابتدای گام j ام حلقه‌ی الگوریتم

$\leq x$	$> x$	پردازش نشده	x
	i	j	r

زیرآرایه‌ی $A[p, i-1]$ شامل عناصر کوچک‌تر مساوی x و زیرآرایه‌ی $A[i+1, j-1]$ شامل عناصر بزرگ‌تر از x و زیرآرایه‌ی $A[j, r-1]$ شامل عناصر پردازش نشده و $x = A[r]$ شاخص است.

← در طول اجرای حلقه درست است.

اثبات با استقرا: $i=j=p$ ← باید ثابت حلقه برقرار باشد

حالت پایه: $i=j=p$

$A[p, p-1]$ پوچ شامل عناصر $\leq x$ و آرایه $A[p, p-1]$ عناصر پوچ

x پوچ اندیس زیر آرایه‌ی $A[p, r-1]$ پردازش نشده و $A[r]$ هم شامل نیست

گام گذار ← فرض می‌کنیم برای j درست است برای $j+1$ اثبات می‌کنیم درست است.

← اثبات: دو حالت: ۱- در ابتدای گام j گذاره درست است

۲- اجرای گام j ام یک بار if شرط برقرار است یک بار if نیست

← loop invariant برای $j+1$ درست است.

اثبات loop invariant ← دستور Partition را نتیجه می‌دهد در حالت آخر $j=r-1$ است ← بعد از حلقه $j=r$ است ← loop invariant برای $j=r$ برقرار است.

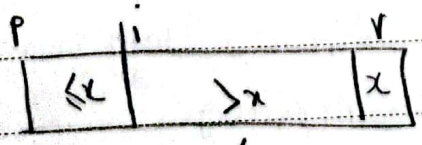
$A[p, i-1]$ شامل عناصر کوچکتر مساوی x

$A[i, r-1]$

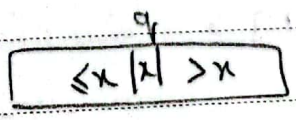
$A[r, r-1]$ پوچ ← عناصر پردازش نشده نداریم

$x=A[r]$

در گام آخر الگوریتم:



← دو دستور ۷ و ۸ ← خانه i و r را swap و return می‌کند.



quick-sort ← برای هر آرایه کمتر n عنصر درست کار می‌کند ← برای n هم درست کار می‌کند.

حالت پایه: n=1

n>1 ← partition اثبات کردیم درست است

شاخص وسط درست طرفین ← هر دو طرف کمتر از n عنصر دارند

چون شاخص خارج آن درست است ← فرض برای هر آرایه n-1 درست کار می‌کند ← ۳ و ۴ هم درست کار می‌کند

← کلاً مرتب شده صعودی می‌شود ← اثبات می‌شود چه

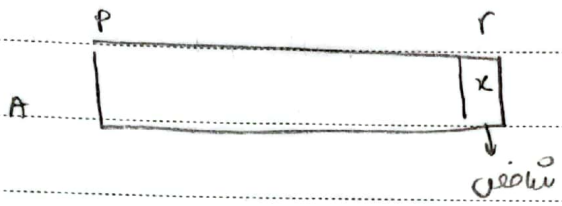
۱۹ فروردین:

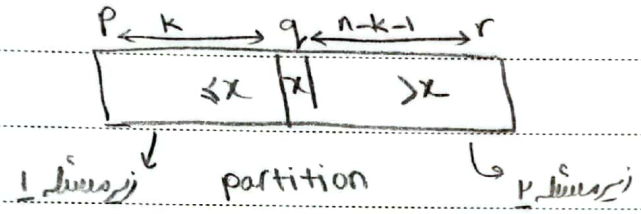
حالت متوسط Quick Sort:

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

زمان

مرور:





$\Theta(n)$ ← زمان partition کردن

* زمانش به k بستگی دارد

balance باشد یعنی طرف ضریبی از n باشد

$\frac{1}{100}n$ ، $\frac{99}{100}n$ △

بهترین حالت

if $k = \frac{n}{2}$ $\quad T(n) = 2T(\frac{n}{2}) + \Theta(n) \in \Theta(n\log n)$

بدترین حالت $\quad T(n) = T(n-1) + \Theta(n) \in \Theta(n^2)$

△ تقریبا پایه ی log کم یا زیاد می شود

حالت متوسط:

سعی تصادفی از الگوریتم ارائه می کنیم بعد آن را تحلیل می کنیم.

سعی تصادفی الگوریتم: Quick Sort

• شاخص را تصادفی انتخاب کنیم.

pivot را به صورت تصادفی انتخاب می کنیم

Randomized ← بخشی از الگوریتم تصادفی باشد.

PAPCO

CamScanner

پارتیشن میانه

* اگر شاخص خود میانه باشد ← نصف عناصر بزرگتر نصف کوچکتر ← بهترین حالت رخ می‌دهد

Randomized_partition (A, p, r)

1- i = RANDOM(p, r) ← یکی از عناصر آرایه را به صورت تصادفی انتخاب می‌کند و جای شاخص می‌گذارد.

انتخاب یک عدد بین p تا r

2- swap(A[i], A[r])

3- return partition(A, p, r)

مدام partition را صدا می‌زنیم.

Randomized_Quick_Sort(A, p, r)

1- if p<r then

2- q = Randomized_partition(A, p, r)

3- Randomized_Quick_Sort(A, p, q-1)

4- Randomized_Quick_Sort(A, q+1, r)

شاخص تصادفی ← تعداد اعداد کوچکتر و بزرگتر میانگین دارد.

محاسبه‌ی میانگین زمان الگوریتم Randomized_Quick_Sort :

$k = n/2$ ← در مرحله اول $n-1$ بار با شاخص مقایسه می‌شوند

بعد دوباره $n/2$ می‌شود $\frac{n}{2} + \frac{n}{2} = n$ تا مقایسه

مرحله بعد $\frac{n}{4} \times 4 = n$ ⇐ $\log^n$

$T(n) = T(n-1) + \Theta(n)$ ← $n-1$ تا مقایسه بعد یک طرف می‌آیند

$n-2$

$n-3$

⋮

تعداد مقایسه‌ها ← $\in \Theta(n^2)$ ← $n-1, n-2, \dots, 1$

PAPCO

partition بندی ← تعداد مقایسه ها عوض می شود.

تعداد مقایسه ها زمان الگوریتم را به ما می دهد + تعدادی فراخوانی

$T(n) = O(n + x)$

تعداد مقایسه ها

تعداد فراخوانی تابع

n ثابت

x فرق می کند ← کافی است میانگین x را حساب کنیم.

$E[T(n)] = O(n + E[x])$

node های درخت تصمیم از order ، n است

به اعداد داخل آرایه اسم می دهیم.

نامگذاری اعداد آرایه $z_1, z_2, z_3, \dots, z_n$ به طوری که z_i : i امین کوچک ترین عدد آرایه است.

$Z_{i,j} = \{z_i, z_{i+1}, \dots, z_j\}$

مجموعه ی اعداد بین z_i و z_j که شامل خود z_i و z_j هم هست.

Quick Sort ، هر عددی با pivot مقایسه می شود

اگر در مجموعه ی $Z_{i,j}$ اولین pivot که از آن انتخاب می شود ← عددی بین z_i و z_j باشد ← z_i در یک partition و z_j در یک بخش دیگر ← z_i و z_j باهم هیچ گاه مقایسه نمی شوند.

z_i و z_j زمانی باهم مقایسه می شوند که اولین شاخص این مجموعه z_i یا z_j باشد.

دو عدد حداکثر یک بار با یک شاخص مقایسه می شود ← دو عدد حداکثر ۱ بار باهم مقایسه می شوند.

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

z_i و z_j باهم مقایسه می شوند اگر و فقط اگر اولین pivot از مجموعه ی $Z_{i,j}$ خود z_i یا خود z_j باشد.

PAPCO

CamScanner

$X_{i,j} = I_{\{z_i \text{ با } z_j \text{ مقایسه شود}\}} = \begin{cases} 1 & \text{اگر } z_i \text{ با } z_j \text{ مقایسه شود} \\ 0 & \text{وگرنه} \end{cases}$

متغیر شاخص تصادفی

متغیری که به رخ دادن یا نشدن می‌دهیم.

رخ دهد ← ۱

رخ ندهد ← ۰

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$$

کل حالت‌های ذوزنی

* هر دو عدد زمانی مقایسه می‌شوند که یکی pivot باشد.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{i,j}]$$

احتمال اتفاق نیفتادن × ۰ + احتمال اتفاق افتادن (مقایسه شدن) × ۱

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} pr\{z_i \text{ با } z_j \text{ مقایسه شود}\}$$

شاخص خارج مجموعه $z_{i,j}$ ← مجموعه را نقطه جابه‌جا کند ولی اگر شاخص درون مجموعه انتخاب شد z_i و z_j ممکن است با هم مقایسه شوند ← اولین انتخاب pivot از این مجموعه z_i و z_j باشد

احتمال اینکه z_i و z_j با هم مقایسه شوند برابر است با احتمال اینکه در اولین باری که در مجموعه‌ی $z_{i,j}$ یک pivot انتخاب می‌شود، آن pivot، z_i یا z_j باشد.

$$pr = \frac{2}{j-i+1}$$

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

$$\leq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} < \sum_{i=1}^{n-1} \underbrace{\sum_{k=1}^{n} \frac{2}{k}}_{\text{حالت کسر}} \leq \sum_{i=1}^{n-1} O(\ln n)$$

کسر ← خرج ↓

$$\int_0^n \frac{1}{x}\, dx = \ln x$$

$$\leq O(n \log n)$$

میانگین تعداد مقایسه ها : $n \log n$

Quick Sort $\xrightarrow{\text{بهترین حالت}}$ $n \log n$

$\xrightarrow{\text{بدترین حالت}}$ n^2

$\xrightarrow{\text{میانگین}}$ $n \log n$

Quick Sort بتوان شاخص را میانه باشد یا نزدیک میانه باشد ← به بهترین حالت می‌رسیم.

شاخص را چگونه انتخاب کنیم که میزان n^2 رخ ندهد.

روش کار الگوریتم های sort ← مقایسه و جابه‌جایی ← می‌خواهیم نشان دهیم کشان $\Omega(n \log n)$ است.

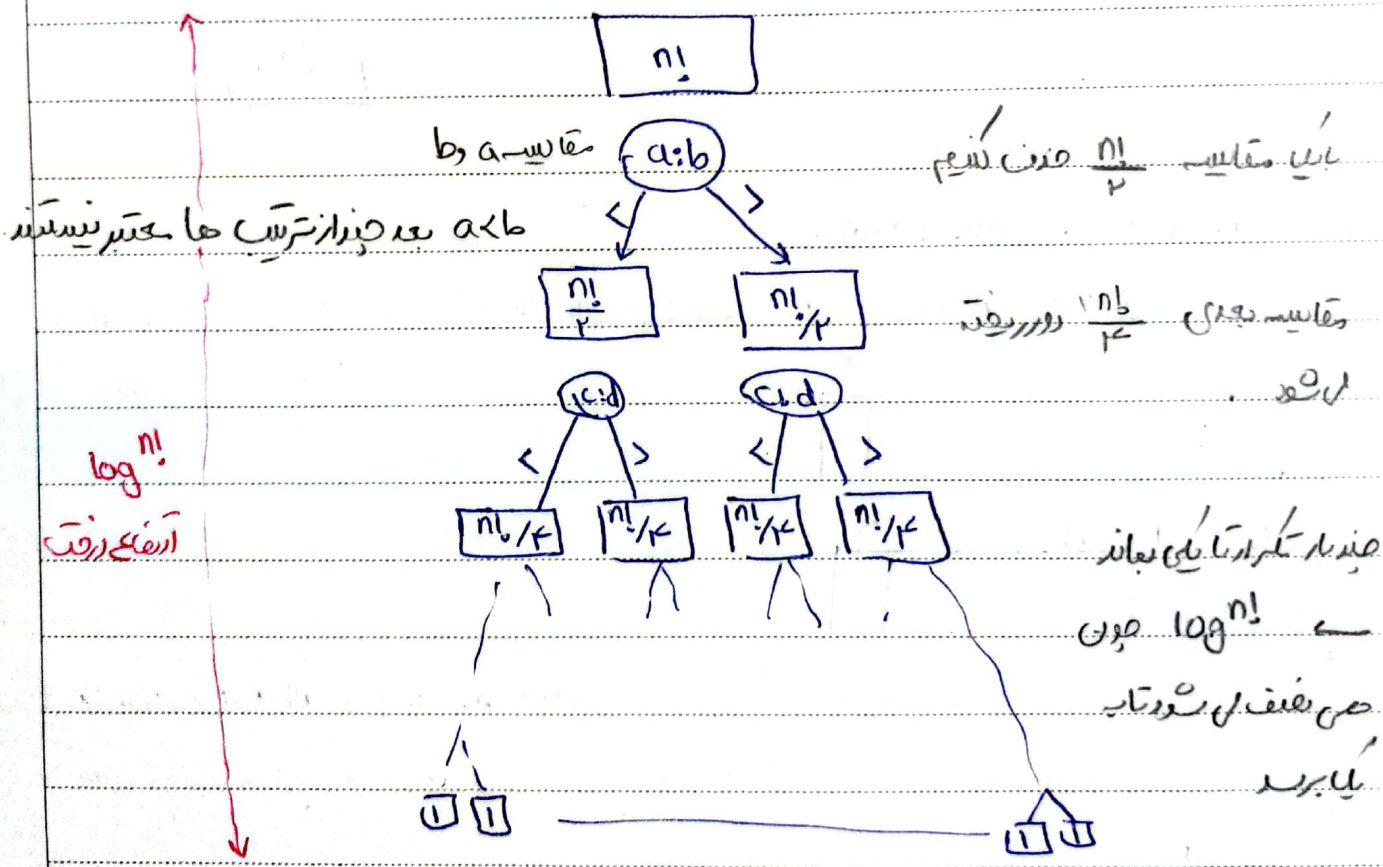
الگوریتم هایی که بر اساس مقایسه نیستند ← می‌توانند کمتر از $n \log n$ باشد.

الگوریتم های مرتب سازی زمان - خطی. linear_time Sorting Algorithms

n تا عدد متمایز ← n! تا ترتیب مختلف دارد ← که یکی ترتیب صعودی است ← دنبال این یکی هستیم.

روش sort کردن: هر ترتیب را چک کنیم که مرتب شده صعودی هست یا نه

در بدترین n! حالت ترتیب هست ← هر بار یک سری ترتیب ها را دور می ریزیم ← تا در آخر به جواب برسیم



هر الگوریتم بر اساس مقایسه این درخت را باید طی کند ← و یک مسیر از ریشه تا برگ این درخت را طی می کند. درخت متوازن ← ارتفاع درخت log n!

$$\boxed{\log n! \in \Theta(n \log n)}$$

ارتفاع درخت حداقل $n \log n$ است.

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

درخت بالانس هر مسیر $n\log^n$ — غیر بالانس یک مسیر بیشتر از $n\log^n$

" کمتر از $n\log^n$

در بدترین حالت تعداد مقایسه‌ها بیشتر یا مساوی $n\log^n$ است. در بهترین حالت ممکن است n باشد.

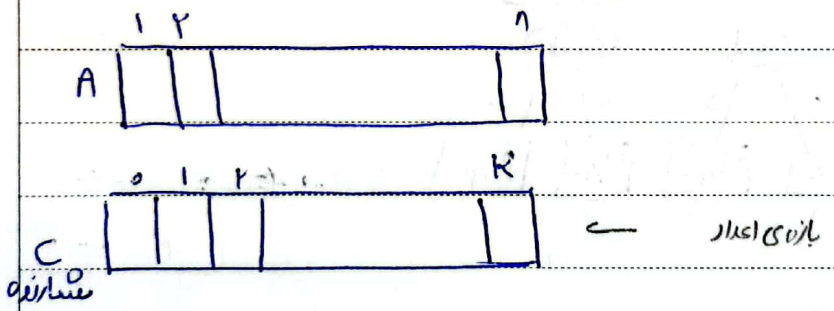
۲۱ فروردین:

الگوریتم کمتر از $n\log^n$ بدون روش غیر مقایسه و جابه‌جایی برای مرتب کردن استفاده کرد.

Sorting in linear time:

1) counting Sort algorithms

اگر n عدد صحیح در بازه‌ی ۰ تا k داشته باشیم. برخی اعداد با k محدود است.

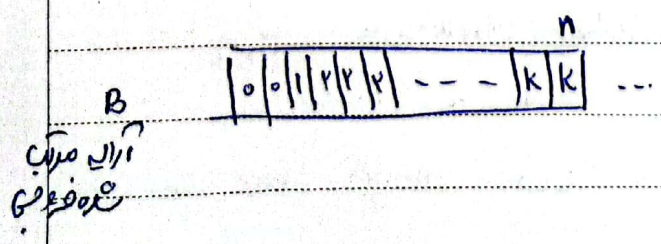


می‌شماریم از اعداد ۰ تا k چند تا داریم

آرایه C را اول صفر می‌کنیم ← هر عددی که در A دیدیم شمارنده را یکی اضافه می‌کنیم

C[A[i]]++

اگر B آرایه‌ی مرتب شده خروجی باشد



PAPCO

CamScanner

Counting Sort کارش کمی متفاوت است ← از آرایه ی A از آخر به اول حرکت می کند
به چپ و راست نگاه می کند چند تا عدد از آن عدد بزرگتر یا کوچکتر است ← ۱

۰	۱	۲		
۲	۱	۳	۷	
۲	۳	۶		

از آخر به اول عددی که می بینیم آخرین عدد از آن عدد است شمارنده خانه اش را نگاه می کند در جای مربوط کپی می کند مثلاً عدد ۲ در خانه ۶ و یکی از شمارنده آن عدد کم می کند

۶ تا عدد داریم کوچکتر ← آخرین ۲ در خانه ی ۶ است
مساوی ۲

```
Counting_Sort (A, n, K)
for i=0 to k do                    مقداردهی اولیه
    C[i]=0
for i=1 to n do                    شمارش
    C[A[i]]++
for i=1 to k do                    محاسبه جمع تجمعی
    C[i]=C[i]+C[i-1]
for i=n downto 1 do                قرار دادن هر عدد A از روی شمارنده در B
    B[C[A[i]]] = A[i]
    C[A[i]]--
```

A[i] می رود در خانه ای از B که شمارنده A[i] نشان می دهد

چرا این روش را استفاده می کنیم؟

Record {
key =
}

بر اساس یک کلید مرتب کنیم. کل record را باید جابه جا کنیم. الگوریتم counting صد A[i] را برمی دارد در موجی B می گذارد ← کلیدها را نمی سازد روش اول کلید اول را می سازد مثلاً ۳ تا ۲ کپی می کند.

A | ۵ | -- | ۹ | ۹ |

← در اینجا اگر آرایه ی ۵ ها را با اندیس نشان می دهیم

PAPCO

با مرتب کردن آرایه هر ۳ ، ۵ بیشتر سر هم قرار می‌گیرند. فرقی نمی‌کند. چون همه ۵ هستند

اما اگر الگوریتم ۵ را به ترتیب ورودی در B می‌گذارد ← ترتیب اعداد مساوی در ورودی را در خروجی حفظ می‌کند ← الگوریتم stable گویند

الگوریتم stable :

ترتیب اعداد مساوی ورودی را در خروجی حفظ می‌کند

counting Sort ، stable است.

تحلیل زمان الگوریتم :

۱- } $\Theta(k)$

۲-

۳- } $\Theta(n)$

۴-

۵- } $\Theta(k)$

۶-

۷- } $\Theta(n)$

۸-

۹-

$\rightarrow \Theta(n+k) \rightarrow$ ظاهر فش درجه ۱ است ، k بر حسب n می‌تواند رشد کم یا زیاد داشته باشد

تابع زمان اجرا ← با اندازه مسئله بیان می‌کنیم

⇐ n تعداد اعداد ، k بازه‌ی اعداد است.

$k \gg n \rightarrow$ آرایه counter بزرگ

نمی‌توان گفت $\Theta(n+k)$ خطی است ← اعداد در بازه ۰ تا ۱۰۰ یا ۰ تا ۱۰۰۰۰۰۰

k عدد ثابت $\Theta(n+k) = \Theta(n)$ عدد ثابت

مثل نمره‌ی دانشجو بین ۰ تا ۲۰

$k=20$ k عدد ثابت

$\Theta(n+20) = \Theta(n)$

PAPCO

CamScanner

← بازه‌ی اعداد ثابت نباشد وقتی n زیاد شود k هم عوض شود.
بازه‌ی اعداد

$k = n \longrightarrow \Theta(n+n) = \Theta(n)$

← اگر بازه‌ی اعداد درجه ۲ تغییر کند وقتی مثلا 1000 شود 0 تا 1000000 ⟸ $\Theta(n^2)$
خواهیم داشت.

$\Theta(n+k)$ شبه چندجمله‌ای است.

if $k \in O(n) \Longrightarrow \Theta(n+k) = \Theta(n)$

بازه‌ی k تا حداکثر n باشد زمان $\Theta(n)$ ← الگوریتم خطی می‌شود

* برای اعداد منفی ← شیفت می‌دهیم اعداد را بعد مرتب می‌کنیم بعد برعکس شیفت می‌دهیم.

اثبات درستی:

loop invariant : ← می‌توان تک تک for ها را ثابت کرد.

11) Radix Sort: الگوریتم

برای مرتب سازی n عدد صحیح d رقمی در مبنای k

روش متداول:

به ارزش رقم ها نگاه می کنیم اینجا یکان - دهگان - صدگان

اون هایی که رقم صدگان کمتر است را می بریم قبل تر

325
351
125
110
204

جابجایی براساس رقم صدگان

125
110
204
325
351

در هر دسته براساس رقم دهگان مرتب می کنیم

110
125
204
325
351

اون هایی که رقم دهگان یکی است رقم یکان را مقایسه می کنیم

در اینجا در هر دسته ۲ مقایسه است.

Radix Sort → از رقم های کم ارزش شروع می کند سمت پر ارزش میرود

یکان	رقم دهگان	صدگان	
325	110	204	110
351	351	110	۱۲۵
125	204	325	۲۰۴
110	325	125	۳۲۵
204	125	351	۳۵۱

→ رقم با ارزش ترتیب را مشخص می کند

* مزیت این روش این است که ساده تر است که دیگر نیازی به دست بندی نیست.

← اینجا stable بودن مهم است ← چرا؟ براساس رقم یکان مرتب کردیم بعد دهگان ← وسط صدگان ← ۲ تا ۳ و ۲ تا ۱ داریم ← مهم است کدام اول یا ۳ جلوتر است زیرا رقم دهگان و یکانش کمتر است که زودتر آمده است ← هر ترتیبی در ورودی همان ترتیب در خروجی است.

← الگوریتم استفاده شده در هر مرحله stable باید باشد.

Radix Sort ← در هر مرحله از counting Sort استفاده می‌کنیم.

Insertion Sort ← اگر while اش < باشد ← stable است.

merge Sort ← merge وقتی دو عدد مساوی اند عدد سمت چپ را اول بگذارد تا stable باشد.

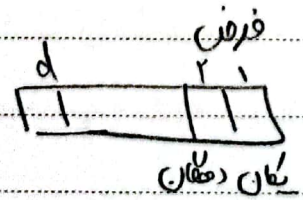
تعداد ارقام‌ها — تعداد اعداد — مبنا

Radix_Sort(A, d, n, k)

1- for i = 1 to d do — رقم اول تا d ام داریم

2- sort A on ith digit using a stable sorting algorithm

↓ می‌تواند counting Sort باشد.



تحلیل زمانی:

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

مبنا k بوده

d بار counting Sort را اجرا می‌کنیم.

PAPCO

• چون مبنا k است ⇐ بازه‌ی هر رقم می‌شود ۰ تا k-1

* d(n+k) ⟵ k,d را طوری تنظیم کنیم که کمترین باشد؟

عدد ۹ در مبنای ۱۰ یک رقم

۹ در مبنای ۲ می‌شود 1001 ← ۴ رقم

$(9)_{10} = (1001)_2$

مبنا را چند بگیریم که ضرب تعداد رقم‌ها در مبنا minimum شود؟

بستگی به مبنای عدد و تعداد رقم‌ها دارد

سوال: ۸-۳-۲ edition 2

n عدد داریم هر عدد در بازه‌ی ۰ تا n^2 است ⟵ اعداد را تک رقمی بگیریم

$\Theta(n+n^2)$ ⟶ روی یک رقم کار می‌کند counting Sort

k

ضرب در تعداد × بازه

چیکار کنیم که از n^2 کمتر شود؟

اثبات درستی: تا هر رقم که رفت از ابتدا تا اونجا اعداد درست مرتب شده‌اند.

۱۴۰۲/۱/۲۶

مرتب سازی زمان خطی

Bucket Sort

Radix-Sort → $\Theta(d(n+k))$ → در چه حالتی خطی می‌شود؟

d باید ثابت باشد ← تعداد رقم‌ها تا یک باند fix داشته باشد مثل تا ۱۰ رقمی

- ۱۰۰ رقمی ← مبنا ۲ کافی می‌تواند تا n رشدش باشد

d: تعداد رقم، n: تعداد اعداد، k: مبنا

counting Sort → k حداکثر رشد n باشد تا خطی شود.

Radix Sort $\underbrace{0023}_{\text{دو رقمی}} - \underbrace{0425}_{\text{سه رقمی}} - 0210$ → $d=4$

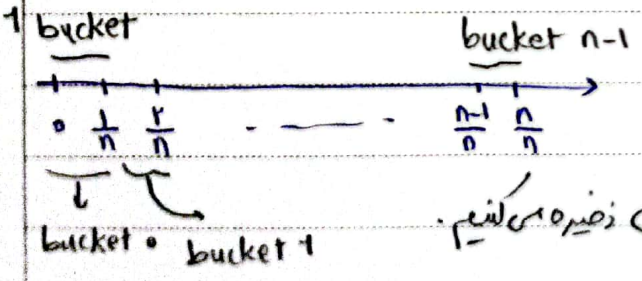
d عدد fix است و به n بستگی ندارد

Bucket Sort:

مرتب سازی n عدد در بازه‌ی ۰ تا ۱ $[0,1)$ ← اعداد حقیقی (اعشاری)

اگر بخواهیم در بازه‌ی $[0,k)$ باشد می‌توان scale کرد

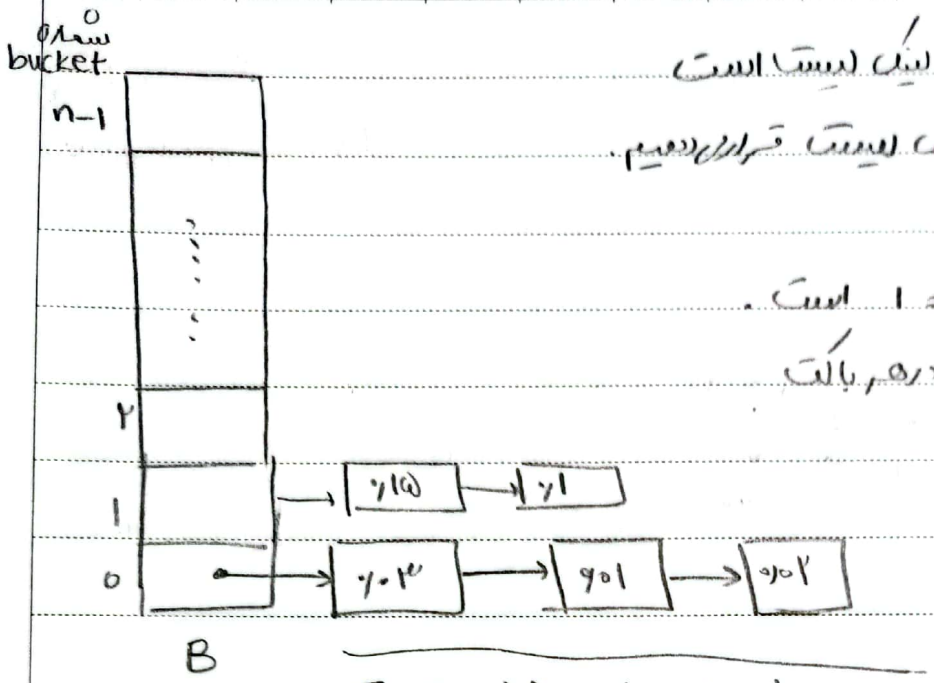
فرض: اعداد در بازه‌ی $[0,1)$ به صورت یکنواخت توزیع شده‌اند با احتمال مساوی پخش شده‌اند - در یک بخش جمع نشده‌اند ← این فرض روی زمان الگوریتم تاثیر می‌گذارد



بازه‌ی $[0,1)$ را به n قسمت تقسیم می‌کنیم ← برای هر bucket یک آرایه در نظر می‌گیریم که اعداد آن bucket را در آن ذخیره می‌کنیم.

PAPCO

هر خانه‌ی این bucket یک لینک لیست است
که عدد داخل باکت را در آن لینک لیست قرار می‌دهیم.



* میانگین تعداد اعداد در باکت‌ها = 1 است.

با احتمال $(\frac{1}{n})^n$ در هر باکت
1 عدد قرار می‌گیرد.

این اعداد در بازه $[0, \frac{1}{n})$ هستند.

بازه اعداد باکت دوم $[\frac{1}{n}, \frac{2}{n})$ ← باکت‌ها را تشکیل دادیم...

← حال همه‌ی باکت‌ها را جدا اعدادشان را sort می‌کنیم ← مثل quick Sort
است در quick Sort ما دو باکت داریم کمتر یا بیشتر از مشخص

نکته: اعداد را چگونه در buckets قرار دهیم؟ آرایه A را بخوانیم هر عدد را ببینیم کجا باید
قرار بگیرد ← اگر مثل quickSort ، partition کنیم n^2 می‌شود
quick Sort ← partition بین ← O(n)

اگر هر عدد را ضرب در n کنیم کف را به دست بیاوریم ← شماره bucket
دست می‌آید

اندیس باکت A[i] = ⌊ A[i] × n ⌋

Bucket_Sort(A , n)

1- for i=1 to n do
2- insert A[i] into Bucket B[⌊ A[i]×n ⌋]

ریختن اعداد در buckets

3- for i=0 to n-1 do
4- sort Bucket B[i]
insertion sort

sort buckets

5- concatenate Buckets B[0]------ B[n-1]

چسباندن باکت‌ها

6-

تحلیل زمانی

دستور داخل For

1- for خاطر for $(n+1)\cdot c_1$

2- $O(1)\times n$ این بیشتر نیست درج می‌کنیم درصورتی که اتصال داشته باشیم می‌شود.

3- $(n+1)\cdot c_2$

4- insertion Sort $\longrightarrow \sum_{i=0}^{n-1} O(n_i^2)$

5- $O(n)$ بیشتر لیست‌ها را به هم وصل می‌کند.

insertion sort → تا زمانی که دکوچک‌تر است می‌رود عقب و درج می‌کند

insertion Sort → بهترین حالت → n
insertion Sort → میانگین حالت → n^2

* توزیع یکنواخت ← در هر باکت تعداد اعداد زیاد نیست ← هر الگوریتم مرتب‌سازی در فقط ۲ خیلی فرقی نمی‌کند.

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

امید ریاضی
میانگین حساب می‌کند

• بهترین حالت: در هر بالت ۱ عدد قرار گیرد

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

• بدترین حالت: همه‌ی اعداد تصادفاً در یک بالت بیافتند

بدترین حالت $n_i = 0$ for $i \neq k$

$n_k = n \quad i = k \quad , \quad 0 \leq k \leq n-1$

بدترین حالت $\rightarrow T(n) = O(n^2)$

• حالت میانگین:

* n ثابت باشد و اعداد عوض نشوند

* اگر n fix باشد $O(n)$ ثابت است

n_i به صورت متغیر تغییر می‌کند

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

$$E[T(n)] = \underbrace{O(n)}_{\text{چون ثابت است}} + \sum_{i=0}^{n-1} O[E[n_i^2]]$$

منظور ثابت ضرب در $E[n_i^2]$ است

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

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

اثبات: $E[n_i^2] = 2 - \frac{1}{n}$

تعریف متغیر شاخص تصادفی $X_{ij} = I\{$ رخ داد $A[i]$ در بالت i ام قرار گیرد $\}$

$$X_{ij} = \begin{cases} 1 & A[j] \text{ در بالت } i \text{ ام قرار گیرد} \\ 0 & o.w \end{cases}$$

$$n_i = \sum_{j=1}^{n} X_{ij}$$

n_i: تعداد اعدادی در بالت i ام

X_{ij}: هر عدد یا در بالت i ام است یا نه

تعداد اعدادی که در بالت i ام قرار می‌گیرند.

PAPCO

CamScanner

$$(n_i)^2 = \left(\sum_{j=1}^{n} x_{ij}\right)^2 \underset{\text{به صورت ضرب در } \sum}{=} \sum_{j=1}^{n}\sum_{k=1}^{n} x_{ij} \cdot x_{ik} = \underbrace{\sum_{j=1}^{n} (x_{ij})^2}$$

وقتی $j=k$ باشد $(x_{ij})^2$ می‌شود

$$+ \underbrace{\sum_{j=1}^{n}\sum_{\substack{k=1\\ j\neq k}}^{n} x_{ij} x_{ik}}$$

وقتی $j\neq k$ باشد

خانه k ام در بالکت i ام

خانه j ام در بالکت i ام

فاصله ۱ که هم خانه k ام هم j ام در بالکت i ام باشد.

$$\xrightarrow{\text{امید ریاضی}} E[n_i^2] = \sum_{j=1}^{n} E[x_{ij}^2] + {}^{*}\sum_{j=1}^{n}\sum_{\substack{k=1\\ k\neq j}}^{n} E[x_{ij} \cdot x_{ik}]$$

x_{ij} و x_{ik} مستقلند

$$\xrightarrow{*} E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n}\sum_{\substack{k=1\\ k\neq j}}^{n} \underbrace{E[x_{ij}]}_{\frac{1}{n}} \cdot \underbrace{E[x_{ik}]}_{\frac{1}{n}}$$

* x_{ij} و x_{ik} باهم مستقلند که خانه j ام در بالکت i ام قرار گیرد

" " " " k "

$E[XY] = E[X]E[Y]$

$$= \underbrace{\sum_{j=1}^{n} \frac{1}{n}}_{1} + \underbrace{\sum_{j=1}^{n}\sum_{\substack{k=1\\ k\neq j}}^{n} \frac{1}{n^2}}_{\frac{n^2-n}{n^2}}$$

تعداد حالات تکراری

$$= 1 + \frac{n^2-n}{n^2} = \boxed{2 - \frac{1}{n}}$$

x_{ij}^2 یا $E[x_{ij}^2]$ مقدارشان یکی است. امید ریاضی $E[x_{ij}]$ احتمال قرار گرفتن خانه j ام در بالکت i ام است $\frac{1}{n}$

$* \; E[x_{ij}] = E[x_{ij}^2] = \frac{1}{n}$

bucket Sort — صورت میانگین زمانش خطی است.

$$E[n_i] = \sum_{j=1}^{n} \underbrace{x_{ij}}_{\frac{1}{n}} = 1$$ → — صورت متوسط در هر بالت ۱ عدد هست.

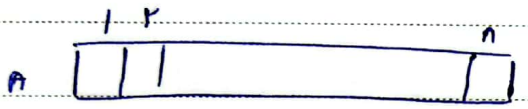
نکته: (خارج از درس)

پیدا کردن میانه و مسئله selection :

میانه ← عنصر k ام کوچک ترین است

پیدا کردن k امین کوچک ترین عدد بین n عدد

$k = \frac{n}{2}$ → مسئله پیدا کردن میانه



① یک راه حل: sort کنیم و عدد k ام آرایه مرتب شده را برگردانیم.

زمانش $O(n\log n)$ است. فرض خاصی هم نداریم مثلاً اعداد در بازه خاصی باشند که از الگوریتم های خطی استفاده کنیم.

حالت خاص: $k=1$ ← مسئله min ← $O(n)$

$k=n$ ← مسئله max ← $O(n)$

② راه حل دیگر: عناصر 1 تا k یکی یکی پیدا کنیم

$O(n.k)$ ← بستگی به k دارد.

③ راه دیگر: یک minheap می سازیم در $O(n)$ انجام می شود و k بار min را استخراج می کنیم

$O(k.\log n)$ ← جواب می شود max این در

$O(n+k\log n)$

* هر عدد ثابت یک کسری از n است ← چه فرقی بین حالت k، fix یا غیر fix ؟

نمرات دانشجوها می دانیم بین 0 تا 20 است ← یک max fix دارد تعداد اعداد ↑

یک سری عدد داده داریم ولی دیگر باند خاصی نمی توان فرض کرد ← عدد بزرگ تر ↑

مسئله ranking . کدام عدد عدد چندم است

← یک درخت binary search با اندازه که تعداد گره های زیرش را در خودش به صورت record داریم

راه حل ۲: ساخت درخت جستجوی دودویی متوازن افزوده

$O(n\log^n) + O(\log^n)$

ساخت درخت — پیدا کردن kامین عدد

درخت متوازن ← در search کردن می گردد یک بار می سازد هی query می زنی

اضافه کردن و حذف کردن داده ها هم در درخت متوازن در $\log^n$ استفاده می شود

$O(k\log^k)$

راه حل ۵: با A[1 ... k] یک درخت جستجوی دودویی متوازن می سازیم. هر عدد جدید که

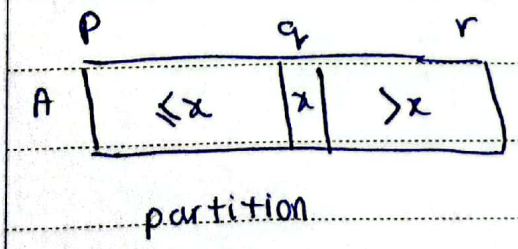
به درخت اضافه می کنیم، عنصر max درخت را حذف می کنیم.

$O(n\log^k)$ به این ترتیب در درخت همواره k تا کوچک ترین عددی که تا به حال دیده ایم ذخیره شده است

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

ایده اصلی:

روش دیگر: آرایه را با یک شاخص partition کنیم با آخرین عدد.



$i = q - p + 1$

رتبه شاخص x

rank x

شاخص: عدد iام

اگر i با k برابر باشد شاخص همان عدد kام می شود.

اگر i با k برابر نباشد عدد kام یا از i کمتر است یا از i بیشتر است ← اگر $k \le i$

در بازه ی p تا q و اگر $k > i$ باشد سمت راست i را می گردیم.

اندیس انتهای آرایه

اندیس ابتدای آرایه — عدد k ام

selection(A, p, r, k)

1- if p==r then return A[p]

2- q=partition(A,p,r)

3- i = q-p+1

4- if k==i then return A[q]

5- if k<i then

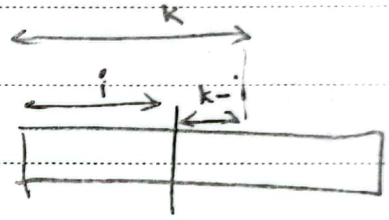
6- return selection(A, p, q-1, k)

7- else return selection(A, q+1, r, k-i)

عدد k ام = عدد k-i ام

در نیمه‌ی دوم است

تا جایی ادامه می‌دهیم که به یک عنصر برسیم و آن عنصر همان عنصر k ام می‌شود



* p==r شد عدد میانه همان عدد k ام است

در قسمت سمت چپ دنبال k ام می‌گردیم ← k کمتر است اندازه آرایه است

سمت چپ k ←

اثبات: استقرایی ← پایه استقرا: برای p=r الگوریتم درست کار می‌کند باشد حال یک k از 1 کمتر باشد و از n هم بیشتر نباشد ←

حالت گذر: برای آرایه کمتر از n تا عنصر k ام را درست برمی‌گرداند

برای آرایه n یا بیش ←

۲:۲۰

ویس

بررسی زمانی الگوریتم selection :

۱- $O(1)$

۲- $O(n)$

۳- $O(1)$

۴- $O(1)$

۵- $O(k)$

۶- } $T(m)$ recusive در نیمه‌ی اول

۷- } " " " دوم

$$T(n) = T(m) + O(n)$$

اندازه نیمه‌ی اول یا دوم است

اندازه نیمه‌ی دوم ، اندازه نیمه‌ی اول) $\longrightarrow \max($

بدترین حالت : $m = n-1$

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

بدترین حالت

درخت باز نشست یا جایگذاری

← هر دفعه (نیمه‌ی) بزرگ‌تر می‌افتیم

بهترین حالت : $m = n/2$

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

بهترین حالت

حالت سوم تقنینی اصلی

← همان دفعه اول یک بار partition کنیم و پیدا شود. پیدا نشد ولی partition خوب عمل کند

← از sort کردن بدتر است ← چون در بدترین حالت n^2 می‌شد ولی در بهترین حالت n است و بهتر از sort است.

حالت میانگین : شاخص به صورت تصادفی انتخاب می‌کنیم.

PAPCO

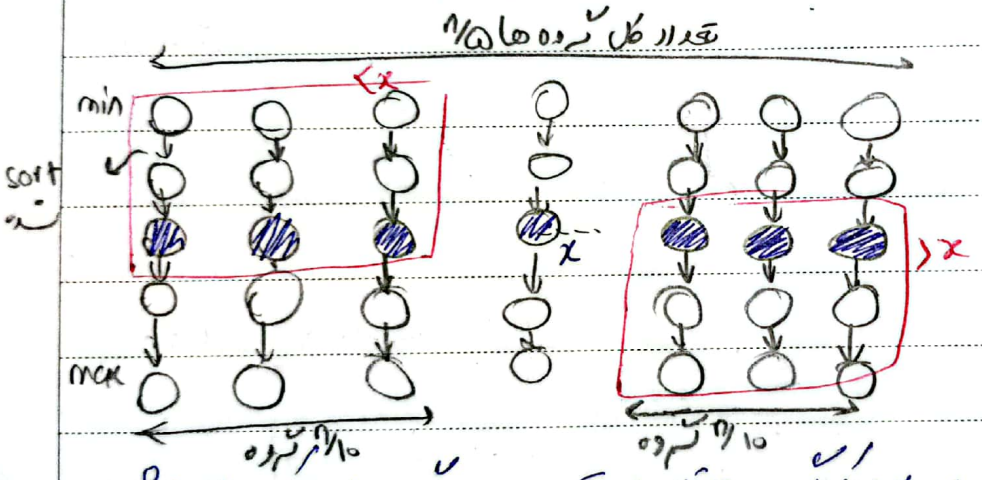
CamScanner

می‌خواهیم کاری کنیم بدترین حالتش $O(n)$ شود ← طوری partition کنیم که شاخص وسط باشد یا balance باشد ← بهترین partition بندی یعنی شاخص میانه باشد یا میانه شود کنیم ← در loop می‌افتیم ← شکستن loop، شاخص یا میانه از زمان آمده بیرون کمتر باشد ← نزدیک میانه هم شاخص باشد خوب است.

balance partition:

بهبود زمان الگوریتم selection:

پیدا کردن شاخص خوب ← اعداد را به گروه‌های ۵ تایی تقسیم می‌کنیم. هر گروه ۵ تایی را sort کنید ← میانه هر گروه عنصر وسطش می‌شود. از بین این میانه، میانه‌ی آن‌ها را پیدا می‌کنیم



میانه میانه‌ها = x ← x در یک گروه ۵ تایی است ← گروه‌هایی که میانه‌ی آن‌ها سمت چپ میانه میانه‌ها می‌گذاریم و بزرگ‌تر باشد سمت راست میانه‌ها می‌گذاریم.

← چند تا عدد از x بیشترند چند تا عدد از x بیشترند؟ میانه partition سمت راست و چپ با شاخص x

تعداد گروه‌های سمت راست میانه میانه‌ها ← $\frac{n}{10}$ است

" " " چپ " " ← $\frac{n}{10}$ " (حالا یکی بیشتر بیشتر)

در گروه‌های سمت چپ " هر گروه ۳ عدد از میانه میانه‌ها کوچک‌ترند ← که تعدادشان $\frac{3n}{10}$ است. ← اعداد بعد میانه هم می‌توانند کوچک‌تر از x باشند ← دل مطمئن نیست ⇐ حداقل

← سایز partition سمت چپ شامل x ← حداقل $\frac{2n}{10}$ است

حداکثر $\frac{7n}{10}$ عدد بزرگتر از x هستند.

حداکثر

$\frac{7n}{10}$ عدد $> x$ ⟸ $\frac{3n}{10}$ عدد حداقل $< x$

در سمت راست هم می‌توان همین استدلال را کرد

حداکثر

$\frac{7n}{10} < x$ ⟸ $\frac{3n}{10} > x$ حداقل

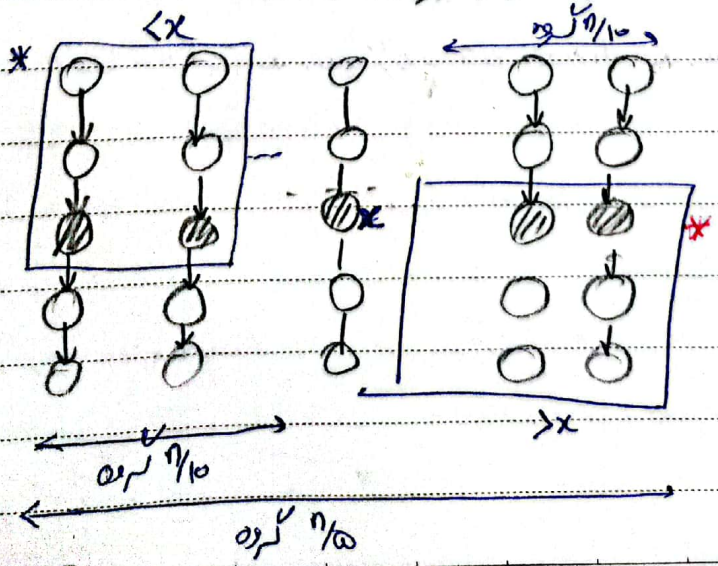
⟸

۱۴۰۲/۲/۱۴

selection پیدا کردن k امین عدد بین n عدد:

شامل را می‌گیرد partition ← اندیس شامل همان است که می‌خواهیم یا نه؟ اگر نه به راست یا چپ می‌رویم.

اگر balance partition شد ← سعی کنیم شامل به میانه نزدیک باشد ← به گروه‌های ۵تایی تقسیم می‌کنیم و هر گروه را sort می‌کنیم ← عنصر وسط می‌شه میانه ← بین میانه‌ها، میانه را پیدا می‌کنیم ← اسمش را x می‌گذاریم ← گروه‌های میانه‌شان کمتر x گروه راست چپ می‌بریم و برعکس. در هر گروه سمت چپ حداقل ۳ عنصر کمتر از x است. در گروه‌های سمت راست حداقل ۳ عنصر از هر گروه بزرگتر از x



* حداقل $\frac{3n}{10}$ عدد از x کمتر هستند ← حداکثر $\frac{7n}{10}$ عدد از x بیشتر اند.

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

PAPCO

CamScanner

← برای بهبود selection ← باید شاخص خوب پیدا کنیم ← از الگوریتم selection به صورت recursive استفاده کنیم

حالت دالبا بیش یعنی آیا چون selection داخلی کوچکتر است.

بهبود الگوریتم selection:

۱- اعداد را به گروه های ۵ تایی تقسیم می کنیم و هر گروه را جداگانه مرتب می کنیم و میانه میانه های گروه ها را جدا می کنیم. ($\frac{n}{5}$ میانه)

۲- به صورت بازگشتی میانه این میانه ها را با فراخوانی selection پیدا می کنیم. فرض کنید این میانه میانه ها x باشد. عنصر $\frac{n}{10}$ را از $\frac{n}{5}$ پیدا کند.

۳- با x آرایه ی ورودی را پارتیشن می کنیم و اندیس x یعنی i را پیدا می کنیم.

۴- اگر $i=k$ آن گاه x را برمی گردانیم

۵- اگر $k<i$ آن گاه به صورت بازگشتی در نیمه ی اول آرایه عدد k ام را می یابیم.

۶- اگر $k>i$ آن گاه به صورت بازگشتی در نیمه ی دوم آرایه عدد $k-i$ ام را می یابیم.

تحلیل زمان این الگوریتم:

۱- $O(n)$ مرتب کردن ۵ تا عدد ← $O(1)$

۲- $T(\frac{n}{5})$ الگوریتم را برای $\frac{n}{5}$ میانه صدا می زنیم

۳- $O(n)$.

۴- $O(1)$ دلیل بدترین حالت این خط اجرا نمی شود

۵- نیمه ای که بیشترین عدد را دارد هر نیمه نهایتا $\frac{7n}{10}$ می تواند داشته باشد

۶- $T(\frac{7n}{10})$

$$T(n) = T(\frac{7n}{10}) + T(\frac{n}{5}) + O(n)$$

بدترین حالت

$T(n) = T(\frac{7n}{10}) + T(\frac{n}{5}) + O(n)$

* با قضیه‌ی اصلی حل نمی‌شود

زیر مسئله‌ها باید اندازه یکسان داشته باشند ولی $\frac{7n}{10} + \frac{n}{5}$

روش حل: ۱- حدس می‌زنیم و با استقرا اثبات می‌کنیم.

۲- درخت بازگشت

$\frac{7}{5}$

$\frac{4n}{10} + \frac{7n}{10} = \frac{9n}{10}$

$T(n) = 2T(\frac{n}{2}) + O(n) \xrightarrow{\text{جواب}} n\log n$

$T(n) = \underbrace{T(\quad) + T(\quad) + \cdots}_{\text{جمع این کسرها } n \text{ شود}} + O(n)$

این هم n باشد → مثل حالت دوم قضیه‌ی اصلی

جواب $n\log n$ می‌شود

ولی اگر کسرها جمعشان n نباشد ولی کمتر باشد ← $f(n)$ غالب می‌شود ← جواب $f(n)$ که اینجا $O(n)$ است می‌شود ← حالت سوم قضیه‌ی اصلی

← اثبات $O(n)$ با استقرا:

$O(n) \rightarrow c_0 n_0 \rightarrow T(n) \leq c \cdot n \quad \text{for } n \geq n_0$

$T(n) \leq c \cdot n \quad \text{for } n \geq n_0$ ← باید این را نشان داد

فرض می‌کنیم برای $\frac{7n}{10}$ و $\frac{n}{5}$ برقرار است ← رابطه بازگشتی می‌نویسیم جواب می‌آید

← در این روش در بدترین حالت $O(n)$ است زمان الگوریتم selection

تئوری این الگوریتم: روی الگوریتم کوچک‌تری میانه را پیدا می‌کنیم.

گروه‌های فرد تقسیم می‌کنند ← چون تحلیل راحت‌تر است ← دلیل اینکه ۳ یا ۵ یا ۷
گروه‌های سه تایی ← در هر گروه دو عضو از ۳ عضو کوچکتر است

حداقل $\frac{n}{3}$ عدد از x کوچکترند ← حداکثر $\frac{2n}{3}$ عدد از x بزرگ‌ترند.

$$T(n) = T\left(\frac{2n}{3}\right) + \underbrace{T\left(\frac{n}{3}\right)}_{\text{پیدا کردن میانه‌ها}} + O(n) \rightarrow$$ جوابش $n\log^n$ می‌شود

← حتماً باید از ۳ بیشتر باشد ← هرچیزی بیشتر از ۳

← همین را می‌توان برای quick sort انجام داد ← balance کنیم ← میانه‌ها با الگوریتم partition

تا بدترین حالت رخ ندهد.

تحلیل سرشکن: Amortized Analysis

n عمل (operation)

n-c تا عمل ← زمان $O(1)$ دارد.

c تا عمل ← زمان $O(n)$ دارد.

* c عدد ثابت است

یک الگوریتم n عمل انجام می‌دهد ← از $O(1)$ کمتر نیست ← n تا هم عمل داریم ← $O(n^2)$

یک تحلیل: هزینه‌ی هر عمل از $O(n)$ بیشتر نیست. n عمل داریم. پس هزینه‌ی کل n عمل از $n \times O(n) = O(n^2)$ بیشتر نیست.

← این تحلیل دقیق نیست ← ما خواهیم حد بالای کمتری را تخمین بزنیم.

تحلیل دقیق‌تر:

$$\text{هزینه‌ی کل} = \underbrace{(n-c)}_{\text{عمل}} \times O(1) + c \times O(n) = O(n)$$

اگر هزینه‌های زیاد را روی c تا عمل هزینه کم پخش کنیم ← هر کس یک ثابتی می‌دهد ← هزینه کم می‌شود.

هزینه زیاد را روی عملیات هزینه کم سرشکن کنیم.

۳ روش تحلیل سرشکن:

۱- aggregate Analysis — تحلیل تجمعی ← * تمرکز ما *

۲- accounting method — روش حسابداری

۳- potential function — تابع پتانسیل

۱- هزینه‌ی سرشکن هر عمل را مساوی

$$\text{هزینه سرشکن هر عمل} = \frac{\text{هزینه کل عملیات}}{n}$$

چند نوع عملیات ← هزینه‌ی سرشکن همه‌ی عملیات‌ها با هم برابر است.

۲- accounting method ، دفتر حساب برای مشتری

← هر عمل یک هزینه‌ی واقعی دارد (add کردن یک عنصر به صف) و یک هزینه سرشکن که ممکن است بیشتر یا کمتر از هزینه‌ی واقعی باشد.

بدهکار شده‌ایم ← سرشکن < واقعی

طلبکار می‌شویم ← سرشکن > واقعی

Data structure: ← میزان طلب یا بدهی را در آن ثبت می‌کنیم

نشان می‌دهد بیشتر یا کمتر مقدار واقعی هزینه شده است

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

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

← اطمینان هزینه‌ی واقعی کمتر از هزینه‌ی سرشکن ← یک upperbound داریم.

یک upperbound برای هزینه واقعی

هزینه‌ی سرشکن هر نوع عمل فرق دارد با بقیه جمع طلب.

$\sum \hat{c}_i - \sum c_i > 0$ ← باید مثبت باشد ← upperbound روی هزینه واقعی داریم

۳- شبیه دومی ← دومی: یک Data Stru- ذخیره می‌کنم برای هر عمل هزینه‌ها را که کمتر داریم ولی در سومی فقط جمع کلش را داریم.

این سه الگوریتم ← پیدا کردن صسالای یک الگوریتم است.

PAPCO

$\sum \hat{c}_i - \sum c_i \geq 0$ تابع پتانسیل در سرعت باید مثبت باشد

Aggregate Analysis:

مثال:

۱- مثال stack ← push-pop - multi pop

۲- " شمارنده باینری

مثال: stack operations

pop() ⟶ عنصر بالای stack را برمی‌دارد ⟶ $O(1)$ هزینه‌ی واقعی

push(x) ⟶ عنصری را بالای stack می‌گذارد ⟶ $O(1)$ " "

multipop(k) ⟶ k تا عنصر بالای stack را ⟶ $O(\min\{k,s\})$

pop می‌کند. اگر k بیشتر از عناصر stack باشد کل عناصر stack را برمی‌گرداند

$\min\{k, s\}$

↓

تعداد عناصر داخل stack

بیشترین مقدار multi pop در n عمل: تا n-1 تا push کنیم و 1 عمل می‌ماند که multipop(n-1) کنیم ← در بدترین حالت $O(n)$ می‌شود

* n-1 عمل push + 1 عمل multipop(n-1)

n تا عمل بدترین حالت هر عمل n-1 است ← امکان ندارد n تا عمل multipop باشد ← $O(n^2)$ غلط است و خیلی تقریب می‌زند.

* چیزی که push نشود را نمی‌توان pop کرد ← یک تعداد push حتماً داریم ← n عمل داریم ← نهایتاً n تا push داریم

نکته ①: تا چیزی در stack push نشود نمی‌توان pop کرد.

نکته ②: تعداد کل push ها حداکثر n تا است

هزینه کل push ها $= O(n)$

با توجه نکته ① ← هزینه کل pop ها $= O(n)$

$\Downarrow$

هزینه کل $= O(n) + O(n) = O(n) \longrightarrow$ هزینه کل n تا عمل $= O(n)$

هزینه سرشکن هر عمل $=$ هزینه کل / n $= O(n)/n = O(1)$

$\downarrow$

می‌توان فرض کرد هر یک عمل $O(1)$ هزینه دارند.

۱۴۰۲/۲/۶

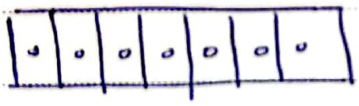
تحلیل سرشکن

برای محاسبه حد بالا های پایین تر

- stack operations
  - push
  - pop
  - multipop

کل push و pop حداکثر $2n$ تعداد است ← $\frac{2n}{n}$ ← تا سرشکن کنیم هر کدام را می‌توان هزینه‌اش را ۲ گرفت

- Binary Counter



		هزینه
0 0 0 0	increment عمل: 0	
0 0 0 1	هزینه ۱ بیت	1
0 0 1 0	هزینه ۲ بیت	2
0 0 1 1	هزینه ۱ بیت	1
0 1 0 0	هزینه ۳ بیت	3

هر بیت که تغییر می‌دهیم یک هزینه‌ی ثابت داشته باشد

* هزینه = تعداد بیت‌هایی که تغییر می‌کند

بدترین حالت تعداد بیت هایی که تغییر می‌کند

0 11 11
1 00 00 ← در بدترین حالت n تا

n تا عمل increment هزینه‌اش چقدر است؟

برای نگه داری عدد n به $\log_2^n$ تا بیت نیاز داریم

در بدترین حالت $\log_2^n - 1$ بیت 1 است و با شمارش $\log_2^n - 1$ صفر می‌شود بیت اول 1 می‌شود ← هزینه حداکثر $\log_2^n$ است.

PAPCO

CamScanner

حداکثر هزینه یک عمل increment $= O(\log_2^n)$

n تا عمل increment داریم $\longrightarrow$ هزینه‌ی n عمل $= n \times O(\log_2^n) = O(n\log_2^n)$

سرشکن انجام نداده ایم $\longleftarrow$ بدترین حالت را حساب کرده ایم.

0 111	2^k-1
1000	2^k

بدترین حالت در 2^k-1 اتفاق می‌افتد

$\longrightarrow$ سه توان دوایی از 2 تعداد بیشتری بیت تغییر می‌کنند

	هزینه‌ها
0000	0
0001	1
0010	2
0011	1
0100	3
0101	1
0110	2
0111	1
1000	4
1001	1
1010	2
1011	1
1100	3

$n/8$ بار تغییر می‌کند

در نصف اندیس‌ها تغییر می‌کند $n/2$

یک در میان تغییر می‌کند

" $n/4$ بار تغییر می‌کند

" n تا عمل جمع n بار تغییر می‌کند

کل تعداد تغییر $= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} = 1\frac{7n}{8}$

کل هزینه‌ی تغییر بیت‌ها $= \sum_{i=0}^{\log_2^n - 1} \frac{n}{2^i} \leq \left(\sum_{i=0}^{\infty} \frac{n}{2^i}\right) \times n$

قاعده هندسی با قدر مطلق $\frac{1}{2}$

$\frac{1}{1-\frac{1}{2}} \times n = 2n$

← کل هزینه تغییر بیت ها نهایتاً $2n$ است - n تا عمل داریم ← هر عمل را نهایتاً می‌توان ۲ گرفت.

می‌توان هر increment را هزینه ۲ گرفت. ← $\frac{\text{هزینه کل}}{\text{تعداد عملیات}} = \frac{2n}{n} = 2$ = هزینه‌ی سرشکن عمل increment

روش های طراحی الگوریتم یا حل مسئله:

۱- روش تقسیم و غلبه — Divide & Conquer

۲- روش برنامه ریزی پویا — Dynamic Programming

۳- روش حریصانه — Greedy

۴- جستجوی فضای حالات — back tracking

brounch & bound

۱- مثل الگوریتم merge sort ← مسئله را به مسائل کوچک‌تر می‌شکنیم بعد جواب‌ها را با هم ترکیب می‌کنیم.

۲- پیرو روش تقسیم و غلبه ← استفاده از حافظه. هر مسئله که حل می‌کنیم جوابش را ذخیره می‌کنیم تا دوباره مسئله را حل نکنیم. پیرو دیگر استفاده از optimal substructure.

۳- پیرو روش دوم ← در روش دوم ← حالت های مختلف زیرمسائل را برای ترکیب بررسی می‌کند ولی در حریصانه ← استدلال می‌کند و یک ترکیب را انتخاب می‌کند ← فقط یک حالت را انتخاب می‌کند.

۴- اگر با سه روش بالا نتوانستیم حل کنیم ← همه‌ی حالات را چک می‌کنیم و جواب مدنظر را برمی‌دارد.

backtracking ← جستجو نظام‌مند

brounch and bound ← جستجو را محدود می‌کند

* حریصانه امکان ندارد بهتر از تقسیم و غلبه باشد.

۱- روش divide & Conquer :

۳ گام دارد این روش: ۱- گام اول: تقسیم مسئله به زیرمسائل کوچکتر

۲- گام دوم: حل بازگشتی زیرمسائل ← فکر کردن ندارد

۳- گام سوم: ترکیب جواب زیرمسائل ← به است آوردن جواب مسئله اصلی

فکری

۱ ← مسئله را به چند زیرمسائلی بشکنند ← زمان الگوریتم براین اساس عوض می‌شه

۳ ← نحوه‌ی ترکیب زیرمسئله‌ها ممکن است راحت و بدیهی نباشد.

مسئله مرتب‌سازی با روش تقسیم غلبه merge Sort - Quick Sort

مسئله مرتب‌سازی :

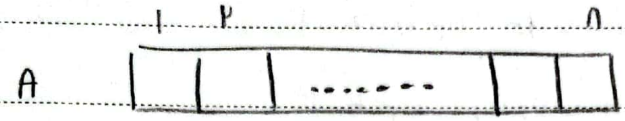
الگوریتم Quick Sort	الگوریتم merge Sort	
با استفاده از partition آرایه دو قسمت احتمالاً نامساوی تقسیم می‌شود $O(n)$	تقسیم آرایه‌ی ورودی به دو قسمت مساوی با اندازه $n/2$ $O(1)$	گام ۱:
Quick-Sort(A, p, q-1) Quick-Sort(A, q+1, r) $T(k) + T(n-k-1)$	merge-Sort(A, p, q) $T(n/2)$ Merge-Sort(A, q+1, r) $T(n/2)$ دو فراخوانی بازگشتی	گام ۲:
—	Merge(A, p, q, r) $O(n)$ ↓ بدون بررسی اعداد نصف کردن merge لازم دارد	
$O(n^2)$ بدترین حالت $O(n\log^n)$ بهترین حالت	$O(n\log^n)$ → بهترین - بدترین - میانگین	

PAPCO

CamScanner

مثل sort به جز partition کردن چه کاری می‌توان انجام داد. میتوانی بر

divide & Conquer باشد.



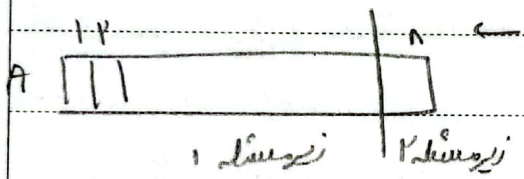
مرتب‌سازی n عدد:

۱- مقایسه n عدد مثل insertion Sort _ bubble Sort

۲- نصف کنیم و مرتب کنیم ← merge Sort

۳- همه را با هم شناختن مقایسه ← Quick Sort

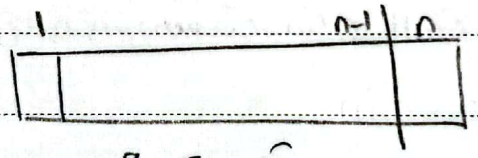
۴- وسط تا مساوی ← یکی ۱ دانه و یکی n-1 تا ← گام تقسیم



زیرمسئله ۱ را حل کنیم یک آرایه مرتب شده n-1 به ما می‌دهد

بعد ۱ دانه را باید با n-1 ترکیب کنیم.

حل بازگشتی: گام ۲



آرایه مرتب شده

گام ۳: insertion Sort (بازگشتی)

گام ۳- مانند while الگوریتم insertion Sort

که جای خالی A[n] را در A[1---n-1] پیدا می‌کند

بعد درج می‌کند.

سینتاکس

Recursive insertion Sort(A, n)

چون هر فرم یکی از آخر می‌زاریم کنار ← داشتن سایز کافی است ← نیازی به از بین ابتدا و انتها نیست.

```
1- if n > 1 then
2-     Recursive- insertion Sort (A, n-1)
3-     key = A[n]
4-     i = n-1
5-     while i > 0 and key < A[i]
6-         A[i+1] = A[i]
7-         i = i - 1
8-     A[i+1] = key
```

۲ گام فراخوانی

گام تقسیم

۵ گام ترتیب

insertion Sort ← while

کلید را جا به آخر می‌آوریم.

* آرایه از ۱ شروع می‌شود.

تحلیل زمانی

1- $O(1)$

2- $T(n-1)$

3- $O(1)$

4- $O(1)$

5- $O(n)$

6- $O(n-1) \times c$

7- $O(n-1) \times c$

8- $O(1)$

$O(n)$ بدترین حالت

while حالت n-1 بار اجرا می‌شود

$\longrightarrow T(n) = T(n-1) + O(n)$

در بدترین حالت ← با جایگذاری یا درخت بازگشت حل می‌شود. $\longrightarrow O(n^2)$

بهترین حالت: while اجرا نشد $T(n) = T(n-1) + O(1)$

جواب: $O(n)$

PAPCO

CamScanner

میانه میانه‌ها ← quick sort مرحله اول $O(n)$ تا زمان می‌برد در نتیجه پیدا کردن میانه در $O(n)$ زمان الگوریتم را زیاد می‌کند ← الگوریتم هیچ در بهترین حالت quickSort قرار می‌گیرد که $n\log^n$ است.

راه حل دیگر ← ۳ قسمت کنیم:

۱- تقسیم آرایه به سه قسمت مساوی با اندازه $\frac{n}{3}$. $O(1)$

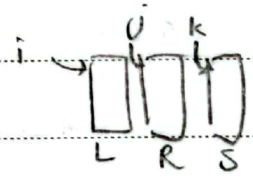
۲- سه فراخوانی بازگشتی $3T(n/3)$

۳- پس ۳ تا min می‌گیریم هر کدام min بود اندیس آن آرایه جلو می‌رود

$i=j=k=1$ $\qquad$ $i=j=k=1$

$\min\{L[i], R[j], S[k]\}$ $\qquad$ هر کدام min بود اندیسش جلو می‌رود و در مقصد کپی می‌شود

مثل merge با این تفاوت که ۳ تا اندیس i و j و k داریم و min این ۳ تا آرایه در مقصد کپی می‌شود. $\longrightarrow O(n)$



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

$T(n)=T(\frac{n}{3})+O(n) \xrightarrow[\text{قضیه اصلی}]{} O(n\log^n)$

به k قسمت تقسیم کنیم:

۱- $O(k)$ ← اگر $\frac{n}{2}$ بخش کنیم هر قسمت ۲ عدد ← $O(k)$

۲- $kT(\frac{n}{k})$

۳- $O(n\times k)$ ← تا k ← min بگیریم $O(k)$ ← n بار هم اینکار را بکنیم

$O(nk)$ ←

ولی روش بهتر: min heap ← min را برداریم

برای کدام آرایه عنصر بعدی را داخل heap بریزیم

کنار هر عدد

$O(\log^k)$

آرایه اش را هم می‌گذاریم

PAPCO

CamScanner

$$T(n) = kT\left(\frac{n}{k}\right) + O(k) + O(n\log^k)$$

$O(n\log^k)$ جمله غالب می‌شود.

k را چند بذاریم کمترین جواب را می‌دهد

k ثابت ← همان جواب قبلی

k را مثلاً $\frac{n}{2}$ بذاریم ←

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

$$O(n\log^n)$$

اگر $k = n\log^n$

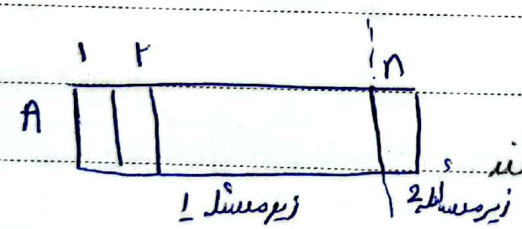
$$n\log^n T\left(\frac{n}{n\log^n}\right) +$$

؟؟

۱۴۰۲/۲/۱۱

Divide & Conquer:

مسئله مرتب سازی



bubble Sort را می‌توان با تقسیم و غلبه انجام داد

اولین عنصر را برمی‌دارد مقایسه می‌کند جابه‌جا می‌کند

max ها به آخر آرایه می‌روند

مثل partition ، max را پیدا کنیم ← بذاریم ← بعد حل بازگشتی می‌شود حل n-1 تای اول.

۱- پیدا کردن max و انتقال آن به انتهای آرایه تقسیم

۲- فراخوانی الگوریتم به طور بازگشتی برای n-1 عدد اول آرایه (حل بازگشتی)

۳- ← ← گام ترکیب ندارد

```
Bubble_Sort(A, n)
1- if n>1 then                                  O(1)
2-     for i=1 to n-1 do
3-         if A[i] > A[i+1] then                O(n)
4-             swap(A[i], A[i+1])
5-     Bubble_Sort(A, n-1)                      T(n-1)
6-
```

۲ تا تقسیم

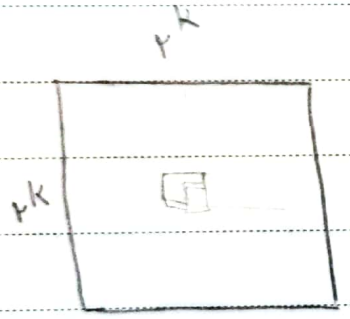
۲ تا

۳ تا

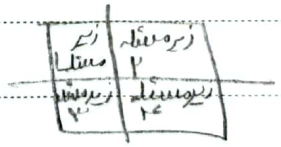
$T(n) = T(n-1) + O(n) \rightsquigarrow O(n^2)$

مساله‌ی موزائیک کردن یک زمین مربع $2^k \times 2^k$

موزائیک ها



راه حل: هر بعد را نصف کنیم ← ۴ زیرمسئله تولید می‌شود



موزائیک مساحتش ۳ — مساحت زمین $2^k \times 2^k$ ← کامل موزائیک نمی‌شود.

$2^k \bmod 3 = 1$ ← خالی می‌ماند که موزائیک نمی‌شود.

مساحت زمین در زیرمسائل ۱و۲و۳و۴ $= 2^{k-1} \times 2^{k-1} = 4^{k-1}$ ← زیرمسائل هم کامل موزائیک نمی‌شود

↓

هزینه مسئله یک خانه‌اش خالی می‌ماند.

در هر فراخوانی باید بگوییم خانه‌ی خالی را کجا بیندازد. مثلاً زیرمسئله ۳ ← سمت راست

top × right
bottom × left

← یک Flag که ۴ مقدار دارد می‌توان مشخص کرد

زمین حالت پایه ۲ ۲ اگر قرار است سمت راست پایین خالی بماند اینجا خالی می‌کنیم

اگر بخواهیم موزائیک کردن زمین مربع $2^k \times 2^k$ با موزائیک

۱- اگر $k=1$ بود آنگاه

* برای خانه خالی flag می‌خواهیم.

حالت پایه

$O(1)$

flag: TR
TL
BR
BL

اگر flag=TR

اگر flag=TL

اگر flag=BR

اگر flag=BL

۲- وگرنه $(k>1)$،

اگر flag=BR آنگاه:

حل بازگشتی زیرمسئله ۱ با flag=BR

حل بازگشتی زیرمسئله ۲ با flag=BL

حل بازگشتی " ۳ " " flag=TR

" " " " ۴ " flag=BR

اگر flag=BL آنگاه: "

" flag=TR " : "

" flag=TL " : "

* بالا بیاید اگر

باید را خیر خانه

$T(N/4)$

$T(N/4)$

$T(N/4)$

$T(N/4)$

* N که اندازه مسئله دوست.

* خانه‌های هر موزائیک را با یک رنگ یا یک عدد پر می‌کنیم

$N = 2^k \times 2^k$

$n = 2^k$

* ادامه صفحه بعد

۳- قرار دادن موزائیک آخر در وسط زمین ← ترکیب.

* باید جنس مسئله و زیرمسئله یکی باشد در تقسیم و غلبه

← تقسیم ← روش متفاوت از وسط نصف کنیم

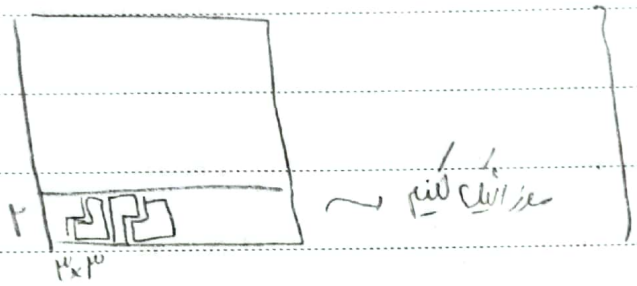
خانه‌های خالی را جمع و ترکیب کنیم.

وقتی دو قسمت کنیم یکی زوج نیست ← این تقسیم خوبی نیست

زیرمسئله‌ها موازنگی زمین مستطیلی می‌شود یا ۴ قسمت کنیم یا صورت مسئله را

تغییر دهیم. مسائل را موازنگی کردن زمین مستطیل تغییر دهیم.

مسئله مستطیل



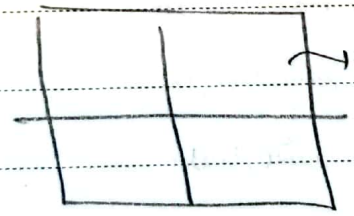
recursive

مساوی نیم

بر ۳ بخش پذیر نیست طول

مستطیل ۲ تا باقی می‌ماند

$N = 2^k \times 2^k$



$\rightarrow 2^{k-1} \times 2^{k-1} = \frac{N}{4}$

$T(N) = 4T(\frac{N}{4}) + O(1)$

$n^{\log_4 4} > 1 \rightsquigarrow = O(N) = O(n \times n)$

براساس پارامتر n:

* استاندارد این است که براساس سایز مسئله انجام دهیم. اندازه ورودی $2^k \times 2^k$

$\cap$

N

← این تابع خطی است یعنی باید

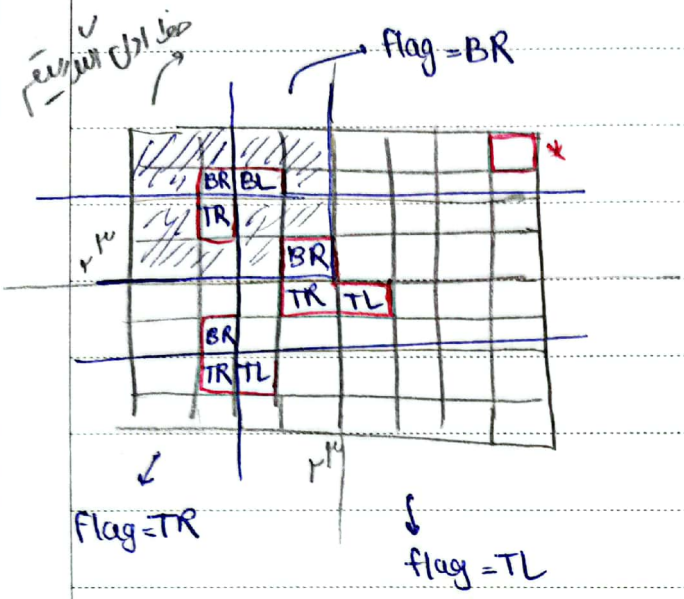
براساس N بزرگ کار انجام داد.

$K \rightarrow$ نمایی $\quad N \rightsquigarrow$ خطی

$T(n) = 4T(n/2) + O(1) = O(n^2)$: n برای سایز

طبق قضیه‌ی اصلی

مثال:



* ورودی flag = TR

زمان بندی مسابقات ورزشی:

* n تا تیم فوتبال داریم می‌خواهند روبه‌روی هم مسابقه بدهند. هر تیم در یک روز ۱ بازی می‌تواند بکند. جدولی طراحی کنید تا در کمترین تعداد روز کل بازی‌ها انجام شود.

n تا تیم

$\frac{n \times (n-1)}{2}$ ← تعداد بازی‌ها

n تیم ورزشی باید با هم مسابقه بدهند. هر تیم با هر تیم باید مسابقه دهد. هر تیم در یک روز حداکثر یک مسابقه می‌تواند بدهد. جدول زمان بندی مسابقات در کمترین تعداد روز؟

تکنیک حل مسئله: property های مسئله را بشناسیم

← در یک روز چند تا مسابقه حداکثر می‌تواند انجام شود؟ $\frac{n}{2}$

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

حداکثر تعداد بازی در یک روز $= \frac{n}{2}$

تعداد روزها از n-1 نمی‌تواند کمتر باشد.

* حداقل n-1 روز لازم داریم.

برای زمانی است که تعداد تیم‌ها زوج باشد ← اگر فرد باشد هر روز ۱ تیم استراحت می‌کند.