HW1

مریا اورکی نیا / 98310772

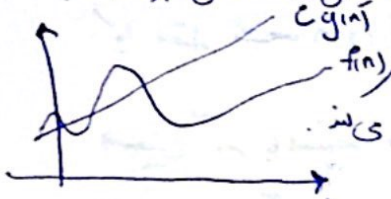
①

Big O)

یک معیار برای مقایسه رشد توابع Big O می‌باشد. Big O نمایش می‌دهد که حالتی ترین تابعی در الگوریتم با یک ورودی مشخص اجرا می‌شود چقدر است.

Big O مفهوم حد بالا را می‌رساند و نشان می‌دهد زمان اجرای الگوریتمی از حد بالای آن بیشتر نخواهد شد.

- برای اطمینان از زمان اجرای الگوریتم از Big O استفاده می‌شود. در واقع اطمینان می‌دهد زمان اجرا از حد بالا بیشتر نیست.

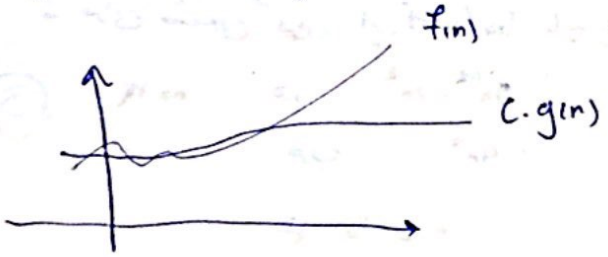


- چون اطمینان را مشخص می‌کند، در واقع زمان اجرای الگوریتم را در بدترین حالت مشخص می‌کند.

worst - case

- برای بیان دشواری مسئله به کار می‌رود.

Big Omega (Ω)



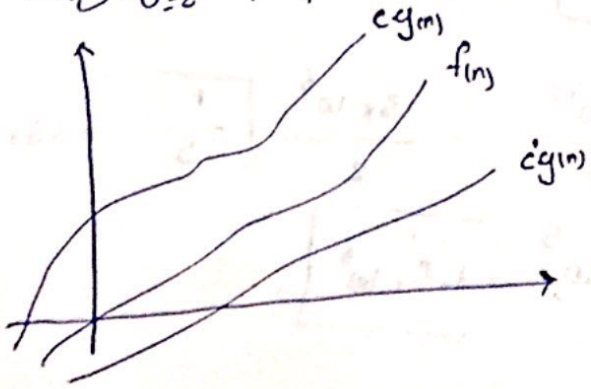
- حد پایین زمان اجرای الگوریتم را مشخص می‌کند.
- یک مقایسه امیدبخش برای الگوریتم می‌باشد.
- اعلام می‌کند در بهترین حالت الگوریتم در چه زمانی اجرا می‌شود.
- در بهترین حالت زمان اجرا از چه مقداری بیشتر است.

Best - case

Θ)

Θ رشد دقیق یک الگوریتم و مقایسه منطقی آن در حالات متوسط است. مفهوم حد بالا و پایین را می‌رساند. حد میانی

- رشد دقیق الگوریتم
- حالات میانی
- احتمال رخداد حالت میانی بیشتر است



average - case

۱) space-complexity & پیچیدگی مکانی، مجموع منابعی مورد استفاده توسط یک الگوریتم / برنامه من جمله فضای اشغال شده توسط ورودی برنامه می باشد.

* رابطه بین زمان مسئله (time -complexity) ، اندازه مسئله (Space-complexity)

- برنامه نویسی معمولا محدودیت زمانی به وجود می آید (البته زمان الگوریتم مهم است) . یعنی با کند تر شدن الگوریتم، برنامه ما به کل از کار می افتد و صرفا زمان طولانی تری اجرای شود. امکان اجرای الگوریتم در اما با زیاد شدن پیچیدگی زمانی ممکن است بنا به اندازه مسئله و منابع موجود دیگر یک مقدار ... a وجود نداشته باشد.

همچنین اگر با استفاده از فضای بیشتر ممکن است الگوریتم سریع تری داشته باشیم که باید بررسی شود Trade-off الگوریتم ما به چه صورت است . آیا می ارزد این حجم از فضا خالی برگشت شود یا نه .

- برای مشخص کردن این Tradeoff باید نیازهای خود و منابع در اختیار را سود مقایسه نمود هیم.

③

$9n^4 + 2n^2$	جمع $n = 1K$
$4n^2$	ضرب
جمع = 5 ضرب	

$T_{(n)}$ { $(9n^4 + 2n^2) \times C_s$ ⟶ sum , $n = 10^3$

$T'_{(n)}$ { $4n^2 \times C_p$ ⟶ product

$5C_s = C_p$

$$\Rightarrow T_{(n)} = \left(9\times(10^3)^4 + 2(10^3)^2\right)\times C_s = (9\times10^{12} + 2\times10^6)C_s = C_s\times10^6(9\times10^6+2)$$

$$T'_{(n)} = 4n^2 \times 5C_s = 20n^2 C_s$$

$$\Rightarrow T_{(n)} = T'_{(n)} \Rightarrow C_s\times10^6(9\times10^6+2) = 20n^2 C_s$$

کم تاثیر

$$\Rightarrow 20n^2 = 9\times10^{12} \Rightarrow n = \sqrt{\frac{9\times10^{12}}{20}} = \frac{3\times10^6}{2}\sqrt{\frac{1}{5}} = 15\times10^5\sqrt{\frac{1}{5}}$$

$$\Rightarrow n = 15\times10^5\sqrt[5]{\frac{1}{5}} \simeq 15\times10^5 = \boxed{1.5\times10^6}$$

a) Bnary Search.

فرض ① : لیست ورودی مرتب‌شده است

a.1)

```
Binary_Search (A, L, R, x)
    if R >= 1
        mid = L + (R-L)/2
        if A[mid] = x
            return mid.
        else if A[mid] > x   // left half
            return Binary_Search (A, L, mid-1, x)
        else   // Right half
            return Binary_Search (A, mid+1, R, x)
    else
        return -1   //not found
```

Worst_case : وقتی x اول یا آخر آرایه باشد $O(n\log n)$

Best_case : وقتی x عنصر وسط باشد $O(1)$

Average_case : $o(\log n)$

فرض ② : لیست ورودی مرتب‌شده نیست.

a.2)

```
linear_Search (A, x)
    for i in A
        if A[i] == x
            return i
```

Best_case : عضو اول $O(1)$

worst_case : عضو آخر $O(n)$

Avg_case : $O(n)$

اگر لیست مرتب شده باشد، linear_Search بهتر است چون $O(n)$ است. اگر اول لیست را سورت کنیم سپس Binary_search بزنیم بهترین حالت $O(n\log n)$ می‌شد که گفته است.

همچنین می‌توان با استفاده از یک مرحله hashing با $O(1)$ حل کرد.

b)

```
Kth_Smallest (A, K)
    L, H = 1, Len(A)
    quick_sort (A, L, H)
    return A [n-K+1]
```

time	Best	worst	Avg
1	1	1	1
1	1	1	1
	$O(n^2)$	$O(n^2)$	$O(n \log n)$
1	1	1	1
	$O(n^2+3)$	$O(n^2+3)$	$O(n \log n+3)$

ابتدا باید آرایه را مرتب کنیم و بعد K امین عدد از ست چه پاسخ ما است. از quick sort استفاده کردیم چون در Avg case سرعت مطلوبی دارد. با توجه به الگوریم quick sort ، الگوریم Kth-Smallest ما به شرح ذیل است:

$O(n^2) + 3 = O(n^2)$; Best case

$O(n^2) + 3 = O(n^2)$; worst case

$O(n \log n) + 3 = O(n \log n)$; Average case

c)

از Bucket sort استفاده کردیم. این الگوریتم را با پیاده سازی کردیم چون نمی خواستیم از Counter استفاده کنیم. Counter ماشه dict یا همان HashMap حساب می شود.

```
def topKfreq(nums, K)
    bucket = [[] for _ in range(len(nums)+1)]   # Import items in a bucket
    Count = Counter(nums).items()               # find frequent of each element
                                                # and store it in a dict

    for num, freq in Count: bucket[freq].append(num)

    Ilist = list(chain(*bucket))
    return Ilist[::-1][:K]
```

Input: [1, 1, 1, 2, 2, 3], K = 2

output: [1, 2]

O(n)

worst-case : O(n) ← چون بدی nums ساختیم و Bucket ساختیم. بعد لیست را flatten کردیم (یک لیست چند بعدی را تک بعدی کردیم) که زمان O(n) می برد و در نهایت خروجی را برگرداندیم.

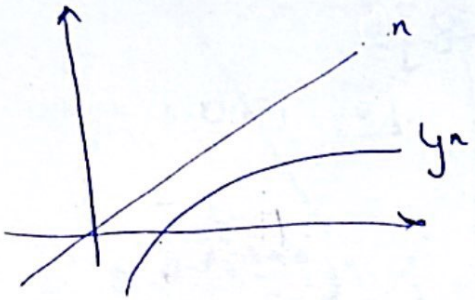
Input 2: [4, 5, 5, 2, 1, 1, 1, 5, 2, 3, 3, 5, 2], K=3

Output 2: [1, 5, 2]

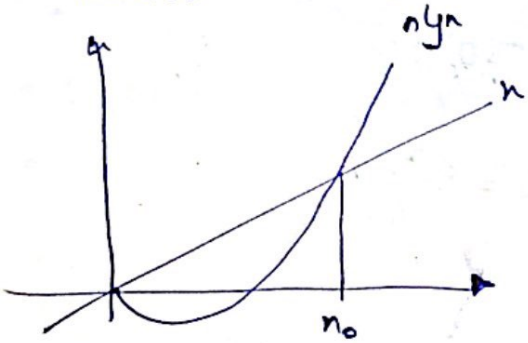
Best-case : O(n)

Avg-case : O(n)

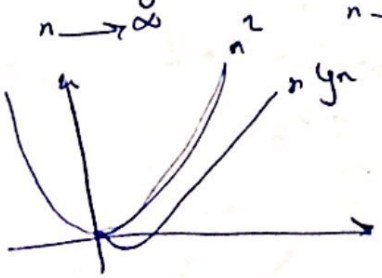
a. $\lim_{n\to\infty} \frac{n}{\lg n} \xrightarrow{HOP} \lim_{n\to\infty} \frac{1}{\frac{1}{n}} = \infty \Rightarrow$ $y_n \in O(n)$ T (5)



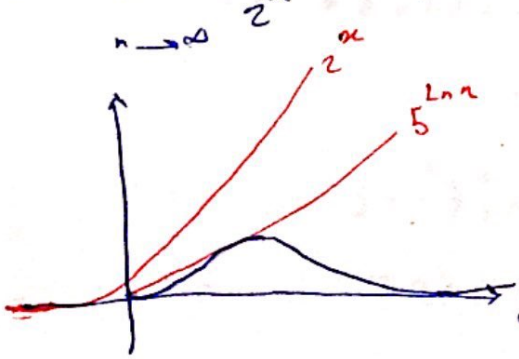
b. $\lim_{n\to\infty} \frac{n\lg n}{n} = \lim_{n\to\infty} \lg n = \infty \Rightarrow$ $n \in O(n \lg n)$ T



c. $\lim_{n\to\infty} \frac{n^2}{n\lg n} = \lim_{n\to\infty} \frac{n}{\lg n} \xrightarrow{(\alpha)} \infty \Rightarrow$ $n\lg n \in O(n^2)$ T



d. $\lim_{n\to\infty} \frac{5^{\ln n}}{2^n} \xrightarrow{HOP} \lim_{n\to\infty} \frac{\ln 5\, n^{\ln 5 - 1}}{\ln^2 2 \cdot 2^n} \xrightarrow{\text{خودار کشیدم}}$



$\lim_{n\to\infty} \frac{5^{\ln n}}{2^n} = 0$

$\Rightarrow 2^n \in \Omega(5^{\ln n})$

$T_{(1)} = T_{(2)} = T_{(3)} = 1$

a) $T_{(n)} = T(\sqrt{n}) + c$

$n = 2^m$, $m = \log_2^n$

$\Rightarrow T'_{(m)} = T_{(2^m)} = T(2^{m/2}) + c$

$\Rightarrow T'_{(m)} = T'_{(m/2)} + c$

master theoreme $f_{(m)} = c$

$a = 1 \quad b = 2$

$m^{\log_2^1}$

$\Rightarrow f_{(m)} \in \theta(m^{\log_2^1})$

case 2 $\Rightarrow T'_{(m)} \in \theta(m^1 \log m)$

$m = \log n \Rightarrow$

$T_{(n)} = \theta(\log \log n)$

b. $T_{(n)} = 2T(\sqrt{n}) + \log n$

$\Rightarrow n = 2^m$, $\log_2^n = m$

$\Rightarrow T_{(n)} = T_{(2^m)} = 2T(2^{m/2}) + m$

$\Rightarrow T'_{(m)} = T_{(2^m)}$

$T'_{(m)} = 2T'(m/2) + m$

$f_{(m)} = m \qquad a = 2$

$b = 2$

$\Rightarrow m^{\log_2^2} = m \Rightarrow f_{(m)} = \theta(m)$

case 2 (master theorem) $\Rightarrow T'_{(m)} = \theta(m \log m)$

$m = \log n \Rightarrow T_{(n)} = \theta(\log n \cdot \log \log n)$

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

$n = 2^m \rightarrow \log n = m$

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

$\Rightarrow T'(m) = 2T'(m/2) + m \lg m^{-1}$

$a = 2$
$b = 2$

$f(m) = m \lg m^{-1}$

$\log_2^2 = 1$

master theorem case 2 extended?

$p = -1$
$k = 1$
$a = b^k$
$2 = 2^k$

$\Rightarrow T(n) = \Theta(n^{\log_b^a} \log^2 n)$

$T'(m) \in \Theta(m^1 \times \log\log m)$

$\Rightarrow$

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

---

d. $T(n) = 3T(n/3) + \frac{n}{\lg n}$

$a = 3$
$b = 3$

$f(n) = n \lg n$

master theorem case 2 extend

$p = -1$
$k = 1$
$a = b^k$

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

---

f. $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

$n = 2^m \rightarrow \lg n = m$

$T(n) = T(2^m) = 2^{m/2} T(2^{m/2}) + 2^m$

$\Rightarrow T(2^m) = 2^{m/2}\left(T(2^{m/2}) + 2^{m/2}\right)$

$= 2^{m/2}\left(T(2^{m/2}) + 2^{m/2}\right)$

$\div 2^m$

$\frac{T(2^m)}{2^m} = \frac{1}{2^{m/2}}\left(T(2^{m/2}) + 2^{m/2}\right)$

$T'(m)$

$\Rightarrow T'(m) = \frac{T(2^{m/2})}{2^{m/2}} + 1$

$\Rightarrow T'(m) = T'(m/2) + 1$

$a = 1$
$b = 2$

$f(m) = 1$

$\log_b^a = 0 \rightarrow f(m) \in \Theta(m^0)$

master theorem case 2

$\Rightarrow f(m) \in \Theta(1)$

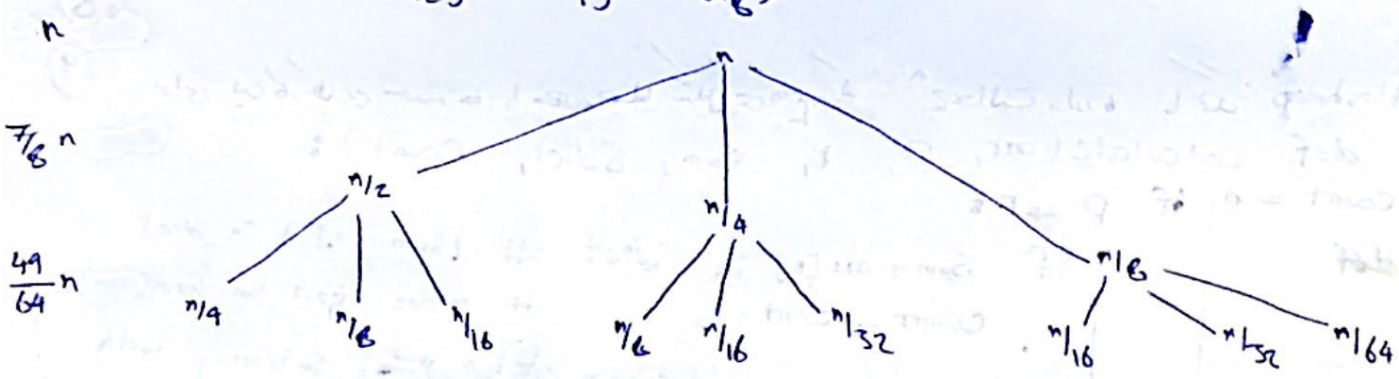
$\Rightarrow T'(m) \in \Theta(m^0 \times \log m)$

$\Rightarrow T'(m) \in \Theta(\log m)$

$m = \lg n$

$\Rightarrow \boxed{T(n) \in \Theta(\lg \lg n)}$

e. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$



$\left(\frac{7}{8}\right)^i n$

$\frac{n}{2^i} = 1 \longrightarrow i = \log_2 n$

why 2?

Because $n/2$ has a Larger Subtree than $n/4$ and $n/8$

This we use i

$$n + \frac{7}{8}n + \frac{49}{64}n + \cdots + \left(\frac{7}{8}\right)^{\log_2 n} n = S_n$$

$$S_n = \sum_{j=0}^{\log_2 n} \left(\frac{7}{8}\right)^j n$$

$$S_n = n + \frac{7}{8}n + \cdots + \left(\frac{7}{8}\right)^{\log_2 n} n$$

$$\frac{8}{7} S_n = \frac{8}{7} + \underbrace{n + \frac{7}{8}n + \cdots + \left(\frac{7}{8}\right)^{\log_2 n - 1} n}_{S_{n-1}}$$

$$\Rightarrow S_n = n + 1 + \frac{7}{8}n + \cdots + \left(\frac{7}{8}\right)^{\log_2 n - 1} n \Rightarrow S_n = S_{n-1} + 1$$

Substitute

$$S_n = S_{n-1} + 1$$
$$S_n = S_{n-2} + 2$$
$$\vdots$$
$$S_n = S_{n-K} + K$$

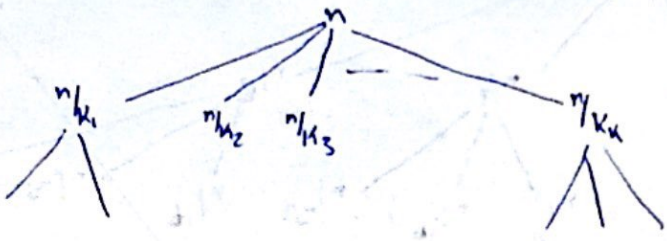
$K = n$

$$S_n = S_0 + K \Rightarrow S_n = n$$

$$\Rightarrow S_n \in \Theta(n)$$

$$T(n) = \underbrace{\sum_{j=0}^{\log_2 n} \left(\frac{7}{8}\right)^j n}_{S_n} \Rightarrow T(n) \in \Theta(n)$$

$$T(n) = T\left(\frac{n}{k_1}\right) + T\left(\frac{n}{k_2}\right) + \cdots + T\left(\frac{n}{k_k}\right) + c_n$$ (7)



اگر k_i وجود داشته باشد $k_i \in \{[k_1 \cdots k_k] - \{k_0\}\}$ یا به عبارتی k_i کوچک ترین بین k ها باشد در این صورت ارتفاع زیر درخت آن از همه بزرگ تر است. پس ارتفاع را مشخص می کند. در نتیجه ارتفاع درخت از این راه حساب می شود

$$\frac{n}{b_i^h} = 1 \longrightarrow h = \log_{b_i} n \Rightarrow h = \frac{\log n}{\log b_i} = \underbrace{\left(\frac{1}{\log b_i}\right)}_{\text{عدد ثابت}} \log n$$

$$\Rightarrow \boxed{T(n) \in O(n \log_{b_i} n) = O(n \log n)}$$

- برای اینکه کمترین مقدار هزینه را داشته باشیم باید درخت متوازن باشد تا زیر مسئله ها جوابی شوند.

$$T(n) = kT(n/k) + c_n$$

master theorem $\longrightarrow$ $a = k$, $b = k$, $f(n) \in O(n)$ $\Rightarrow f(n) \in \Theta(n^{\log_k k})$

case 2 $\longrightarrow$ $T(n) \in \Theta(n^{\log_k k} \log n)$

$$\Rightarrow \boxed{T(n) \in \Theta(n \log n)}$$

* ازای هر k ثابت است.

```
def Kth_element(arr1, m, arr2, n, K):
    # if K is out of bound
    if K > (m+n) or K < 1:
        return -1
    if m > n
        return(arr2, n, arr1, m)  # m has to be smaller than n

    if m == 0:  # if arr1 is empty, K-th element arr2
        return arr2[K-1]
    if K == 1:  # min of first elements
        return min(arr1[0], arr2[0])

    i, j = min(m, K//2), min(n, K//2)

    if (arr1[i-1] > arr2[j-1]):  # find K-j element
                                 # because we found j
        return Kth_element(arr1, m, arr2[j:], n-j, K-j)
    else:
                                 # find K-i element
                                 # because we found i
        return Kth_element(arr1[i:], m-i, arr2, n, K-i)
```

- این روش شبیه راهی مانند Binary-search است. اگر عناصر میانی هر arr1, arr2 را مقایسه کنیم این شانس را به ما اجازه کوچک کردن مسئله را می دهند. اگر هم انتهای میانه arr1, arr2 از K کمتر باشند دیگر نیازی نیست نیمه چپ آرایه ها را مقایسه کنیم. و اگر میانه arr1 از میانه arr2 کمتر باشد دیگر نیازی به مقایسه میانه چپ آرایه ۱ نیست. این مقایسه ها هر سری آرایه در حال مقایسه را ½ می کند. مانند Binary search

- همچنین اگر بجای میانه آرایه ها را به 2 بخش K و n-K (K و m-K) تقسیم کنیم از پیچیدگی زمانی بهتری برخوردار خواهیم بود.

- این الگوریتم از پیچیدگی زمانی $O(\lg K)$ برخوردار است. از آنجا که K بین ۱ و m+n است در بهترین حالت $O(1)$ و در بدترین حالت $O(\lg m+n)$

```
def calculate(arr, p, r, Sum, Sdict, Count):
    if p == r:
        if Sum + arr[r] in Sdict  # [Sum]+1 in dict
            count = count + 1     # means it can form
                                  # a new subarray with
        elif Sum in Sdict         # last element
            count = count + 1
        return
    calculate(arr, p+1, r, Sum, Sdict, count)
    calculate(arr, p+1, r, Sum+arr[p], Sdict, counter)

        # check if with addiction of first elem a new subarray
        # is possible

counter = 0
Sdict = {arr[i]: 1 for i in range(0, Len(arr)-1)}
calculate(arr, 0, Len(arr)-1, Sum, Sdict, count)
```

- در یک dict مقدار هر اعضا آرایه را ۱ می‌گذاریم. با صدا زدن تابع مسئله به 2^n زیرمسئله تبدیل می‌شود چون از dict استفاده می‌کنیم هزینه $O(1)$ دارد. Sum با جمع کردن خودش با ابتدا و انتها اولین جمع‌های زیرمجموعه‌های ممکن را می‌سازد. در واقع این الگوریتم تمامی subarray ممکنه [ ] و arr به صورت بازگشتی می‌سازد و شروط را بررسی می‌کند. چون 2^n زیرمجموعه داریم و هزینه $O(1)$ نداریم (به خاطر dict) پس از $O(2^n)$ می‌باشد. در واقع اگر درخت بازگشتی را رسم کنیم تعداد برگ‌ها همان تعداد زیرمسئله و پیچیدگی زمانی الگوریتم ما است.

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