■ 1- prove that there is no comparison based sorting problem running better than $O(n\log n)$.

* ورودی : یک دنباله از n عدد
* خروجی : یک جایگشت (مرتب‌سازی شده) از دنباله‌ی ورودی.

← یک الگوریتم مرتب‌سازی براساس مقایسه (Comparison based) است اگر از عوامل مقایسه‌ای برای یافتن ترتیب بین اعداد استفاده کند. برای این مدل مرتب‌سازی از "درخت تصمیم‌گیری" استفاده می‌کنیم.
درخت تصمیم‌گیری یک درخت باینری پُر (full binary tree) است که مقایسه بین عناصر را نشان می‌دهد و اجرای این الگوریتم مرتب‌سازی به یافتن یک مسیر از ریشه درخت تصمیم‌گیری تا برگ آن، بستگی دارد. در هر گره (node) داخلی این درخت یک مقایسه بین a_i و a_j صورت گرفته؛ به این صورت که زیردرخت سمت چپ بیانگر $a_i<a_j$ و زیردرخت سمت راست بیانگر $a_i>a_j$ است.

۱- هر یک از $n!$ جایگشتی که از n عدد ورودی داریم، در یکی از برگ‌های درخت تصمیم‌گیری ظاهر می‌شود حتماً.

۲- اگر حداکثر x مقایسه در این الگوریتم انجام شود درنتیجه ماکزیمم ارتفاع درخت تصمیم‌گیری x خواهد بود.
و یک درخت با عمق x، حداکثر 2^x برگ دارد.

با ارجام دو نکته فوق داریم:

$$2^x \geq n! \xrightarrow{\log_2} x \geq \log_2 n! \xrightarrow{\log n! = \Theta(\log n^n) \text{ می‌دانیم}} x \geq \log_2 n! \geq \log_2 n^n$$

$$\Rightarrow x = \Omega(\log_2 n^n) = \Omega(n\log_2 n)$$

□ درنتیجه هر Comparison based Sorting algorithm باید حداقل $n\log n$ مقایسه انجام دهد تا آرایه‌ی ورودی را مرتب کند. پس در زمان بهتر از $n\log n$ انجام نمی‌شود.

2. a) best, average and worst case time complexity of quick sort with the pivot as the last element of the array.

برای این روش در انتخاب pivot، زمان اجرا در حالت کلی : $T(n) = T(n-k-1) + T(k) + \Theta(n)$

که در آن k تعداد عناصر کوچکتر از pivot و (n-k-1) تعداد عناصر بزرگتر از pivot هستند و $\Theta(n)$ مربوط به تابع partition است (که در واقع عملیات Devide را انجام می‌دهد!).

حالا سه وضعیت ممکن است پیش بیاید:

۱- حالت بدتر Worst case : زمانی پیش می‌آید که در تابع partition، همیشه کوچکترین یا بزرگترین عنصر انتخاب شود به عنوان pivot که در این مسئله که pivot همیشه آخرین عنصر آرایه ورودی باشد، می‌توان نتیجه گرفت که بدترین وضعیت وقتی پیش می‌آید که آرایه ورودی از ابتدا مرتب شده باشد (حالا صعودی یا نزولی) پس داریم:

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

که در نهایت، $T(n) = \Theta(n^2)$ خواهد شد!

۲- حالت بهتر Best case : بهترین وضعیت مربوط به زمانی است که تابع partition آرایه را در هر مرحله به دو قسمت تقریباً مساوی تقسیم کند یعنی عنصر میانه را به عنوان pivot انتخاب کند. پس داریم:

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

که در نهایت $T(n) = \Theta(n\log n)$ خواهد شد (بر اساس master theorem)!

۳- حالت میانگین Average case : برای بررسی این حالت باید تمام جایگشت‌های ممکن برای آرایه با n عدد را در نظر گرفته و اردر آن‌ها را یکی یکی بررسی کنیم!

می‌توان از این ایده استفاده کرد که فرض کنیم تابع partition، آرایه را به نسبت ۱ به ۹ تقسیم کند (برگرفته از سایت geeksforgeeks!) پس داریم:

$$T(n) = T(n/10) + T(9n/10) + \Theta(n)$$

که در نهایت $T(n) = \Theta(n\log n)$ خواهد شد.

b) what is a better way of choosing pivot?

برای داشتن عملکرد بهتر، باید یک pivot ثابت در نظر نگیریم چون این کار (یعنی انتخاب fixed pivot) احتمال وقوع بدترین حالت یعنی زمان اجرای $O(n^2)$ را زیاد می‌کند. یک تکنیک بهتر برای انتخاب pivot، "میانه‌ی ۳ عدد" است. می‌دانیم که در این روش، اولین و آخرین عنصر انتخاب نمی‌شود (بر اساس قضیه حد مرکزی، احتمال انتخاب عناصر وسط و میانه به طور نسبی بیشتر از کوچکترین و بزرگترین عنصر است!) پس یعنی احتمال زیاد یکی از عناصر میانی را انتخاب می‌کنیم و در نتیجه به زمان اجرای $O(n\log n)$ می‌رسیم.

همچنین با استفاده از الگوریتمی که میانه عناصر یک آرایه را در $O(n)$ پیدا می‌کند، می‌توانیم با اطمینان بیشتری به زمان اجرای $O(n\log n)$ برسیم. به این صورت که:

$T(n) = O(n)$ (پیدا کردن میانه) $+ O(n)$ (partition زمان اجرای) $+ 2T(n/2)$

که بنابر قضیه اصلی (Master theorem)؛ $T(n) = O(n\log n)$ خواهد شد!

3. K = the number of acorns = $\lceil \log(n) \rceil + 1$: $O(\log n)$

a) n = the number of branches

برای این کار کافیست از شاخه $[\frac{n}{2}]$ اولین acorn را پرتاب کند. دو حالت رخ می‌دهد:

الف) اگر acorn بشکند: همین کار را برای شاخه‌های پایینی درخت تکرار کند. (جواب در بازه‌ی $[0, n/2]$ است)

ب) اگر acorn نشکند: همین کار را برای شاخه‌های بالایی درخت تکرار کند (جواب در بازه‌ی $[\frac{n}{2}, n]$ است)

پس داریم:

$T(n) = \Theta(\log n) \Leftarrow T(n) = T(n/2) + O(1)$ → تست خانه وسط

```
int acorndrop(int n, int k, int s, int e, int arr[]):
    if n == 1 :
        return e.
    elif n == 0;
        return -1  # there is any tree
    elif arr[s+n/2] == 1 : # acorn break open
        acorndrop(n/2, k-1, s, s+n/2, arr)
    else  # arr[s+n/2] == 0  # acorn doesn't break open
        acorndrop(n/2, k, s+n/2, e, arr)
# end of acorndrop
```

e: 1, 1, 1, 1, 1, 1, 1, 0, 0 :s

* (مثلاً آرایه روبرو نشان می‌دهد که acorn در خانه‌ی 2 می‌شکند (s اندیس شروع و e اندیس پایانی آرایه و n تعداد عناصر آرایه و k تعداد acorn) خروجی تابع اندیس مطلوب است

درحقیقت اولین عضو 1 را در آرایه arr پیدا می‌کند و اندیس آن را برمی‌گرداند. $(O(\log n))$

3. b) $O(n)$, $k=1$, n

```
int acorndrop(int arr[], int n):
    for i in range(n):
        if arr[i] == 1:
            return i.
* end of function
```

یعنی باید از خانه صفرم ، acorn را پرت کند. اگر شکست سوال شخصه یعنی می‌رود و اگر شکست مطلوب محقق شود. کاملاً بدیهی است که این تنها راه است وقتی فقط یک acorn داریم! و جواب کمتری (مثلاً $O(\log n)$) وجود ندارد.

3. c) $O(\sqrt{n})$, $k=2$, n.

اگر فرض کنیم اولین acorn را از خانه x بیندازیم دو وضع رخ می‌دهد: ۱- acorn می‌شکند پس باید از خانه صفرم تا $x-1$ را یکی یکی تست کنیم زیرا یک acorn خواهیم داشت. (مثل حالت (3.b))

۲- acorn نمی‌شکند پس مثلاً دومین پرتاب‌مان از خانه $x+(n-1)$ خواهد بود. در حالت کلی اگر در خانه iام، acorn سالم و نشکسته باشیم، پس مقدار پرتاب‌مان می‌شود $x+(x-1)+\dots+(n-i-1)$.

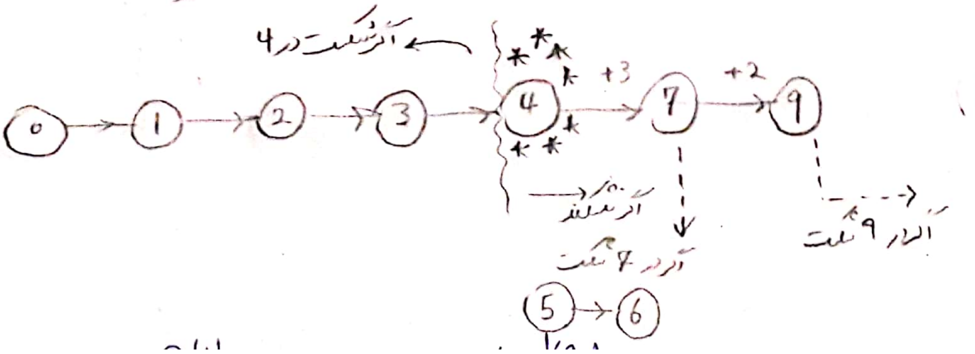
پس در کل تعداد خانه‌هایی که بررسی خواهند شد (مثل در وضع قبل) به این صورت است: $x+(x-1)+(x-2)+\dots+2+1 = \frac{x(x+1)}{2}$.

بدیهی است که: $S=\frac{x(x+1)}{2} >= n \Rightarrow x^2+x \geq 2n \xrightarrow{+\frac{1}{4}} (x+\frac{1}{2})^2 \geq \frac{8n+1}{4}$

$\Rightarrow x \geq \frac{\sqrt{8n+1}-1}{2} \Rightarrow$ تعداد پرتاب در بدترین شرایط $= x = O(\sqrt{n})$

مثلاً اگر $n=10$ باشد و $k=2$ داریم: $x=\frac{\sqrt{81}-1}{2}=4$ یعنی اول از خانه ۴ام (اندیس‌ها را با ۴) شروع می‌کند

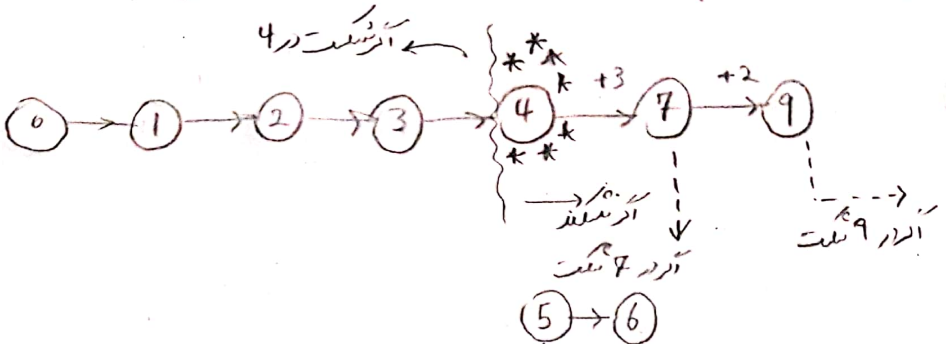
پس:



در بدترین شرایط با $\lceil\sqrt{10}\rceil \cong 4$، پرتاب به مطلوب می‌رسیم.

اگر $n=10$ باشد و $k=2$ داریم: $x = \frac{\sqrt{81}-1}{2} = 4$ یعنی اول از خانه‌ی ۴ام (اولین پرتاب با ۴) شروع می‌کند

پس:



در بدترین شرایط با $\lceil\sqrt{10}\rceil \cong 4$، ۴ پرتاب به مطلوب می‌رسیم.

3. d) $k = O(1) = a$, n , $O(n^{1/a})$

فرض اگر $a=2$: اگر x پرتاب لازم باشد و فرض کنیم که اولین acorn را از شاخه‌ی آخر انداخته باشیم و آن شکسته باشد پس تعداد شاخه‌هایی که می‌توان کاور کنیم، $x-1$ با ۲ acorn کمتر است. پس به شکل جامع‌تر اگر در هر مرحله acorn نشکند، پس از آخر $S(x-1) = \frac{(x-1)x}{2}$ (s در قسمت c یعنی ۲ acorn داریم تعریف شده) شاخه‌ی دیگر را کاور می‌کنیم و در شکل جامع‌تر:

$$P := (S(x-1)+1) + (S(x-1)+S(x-2)+2) + \dots + (S(x-1)+S(x-2)+S(1)+S(0)+x)$$

$$= x + \sum_{i=1}^{x} \frac{i(i-1)}{2} = x + \frac{(x-1)x(x+1)}{6} \geq n \longrightarrow x \cong O(n^{\frac{1}{3}})$$

به طریق مشابه برای $k=a$ داریم $x = O(n^{\frac{1}{a}})$.

3. e. $K = a = \lceil \log(n) \rceil + 1 \longrightarrow\ ?\ O(n^{\frac{1}{a}})$ or $O(\log n)$

واضح است که $O(\log n)$ خواهد شد زیرا در قسمت (d) داشتیم: اگر $K=O(1)=a$ یعنی تعداد acorn ها، static و مستقل از مقدار ورودی (n) باشد، جواب به فرم $O(n^{\frac{1}{a}})$ خواهد بود!

4. این در واقع همان مسئله پیدا کردن ماکزیمم تفاضل دو عدد در یک آرایه است به طوری که عدد بزرگتر همواره بعد از عدد کوچکتر بیاید (نه اینکه آرایه صعودی مرتب شده باشد!) مثلاً ([۱, ۲, ۹۰, ۱۲, ۵۰۰])

● Brute-force : باید تمام حالات چک شوند... ($O(n^2)$)

```
def maxDiff (arr, arr_size):
    max = arr[1] - arr[0]
    for i in range(arr_size):
        for j in range(i+1, arr_size):
            if arr[j]-arr[i] > max :
                max = arr[j] - arr[i]
    return max.
```

● $\rightsquigarrow O(n)$

```
def maxProfit(arr, n):
    max = arr[1] - arr[0]
    min_ele = arr[0]
    for i in range(1, n):
        if (arr[i] - min_ele > max):
            max = arr[i] - min_ele.
        if (arr[i] < min_ele):
            min_ele = arr[i]
    return max.
```

● Divide and Conquer : $O(n \log n)$

```
def MaxProfit (costs, l=0, r):
    if r-l <= 1 :
        return φ;
    mid = (r+l)//2
    res = max(costs[mid:r]) - min(costs[l:mid])
    return max(res, MaxProfit(costs, l, mid), MaxProfit(costs, mid, r))
```