جلسه چهارم. تحلیل جستجوی خطی و باینری

- **آشنایی با جستجو در آرایه**
- **جستجوی باینری و خطی**
- **تحلیل مرتبه زمانی آن‌ها در سه حالت بهترین، متوسط و بدترین حالت**
- **استفاده از تکنیک unfolding و روش درختی**
- **استفاده از میانگین وزنی و میانگین احتمالاتی**

یک آرایه را در نظر بگیرید؛

یک آرایه‌ای از اعداد صحیح و البته غیر مرتب.

۸	۲	۴	۵	۱	۶	۳

برای پیدا کردن یک عدد من باید همه‌ی اعداد را ببینم. چون دقیقا نمیدانم عدد مد نظرم کجاست ممکن در ابتدای آرایه باشد و یا در انتهای آرایه. در بهترین سناریو و از شانس خوب ممکن است با اولین مقایسه آن را بیابم ممکن هم هست از شانس بدم پس از بررسی همه‌ی عناصر در آخرین المان به جواب برسم. اما به طور متوسط که بیشتر اوقات رخ خواهد داد حدودا نیمی از آرایه را مقایسه باید بکنم تا به جواب برسم.

برای مثال برای پیدا کردن عدد ۶ در آرایه ما از تابع پایتون زیر استفاده می‌کنیم؛

```
def find (A, x):
    # find x in array A
     for  index,  item  in
enumerate(A):
                        #   print
(index,item)
            if item == x:
                return (index)
    return -1
```

در این تابع اگر find در آرایه‌ی A عدد x را پیدا کند index آن را بر می‌گرداند و در غیر اینصورت ۱- را بر خواهد گرداند. پس میتوان گفت مرتبه‌ی زمانی یا پیچیدگی زمانی یا time complexity این برنامه از مرتبه‌ی n هست یا به مجموعه‌ی BigO تعلق دارد.

بعبارتی اگر شما تابع ریاضی این شبه کد مثلا g(x) را بدست بیاورید و با کمک حد گیری این تابع را با تابع f(x) = x مقایسه کنید خواهید یافت حدودا از یک مرتبه خواهند بود. یعنی :

باشد یعنی

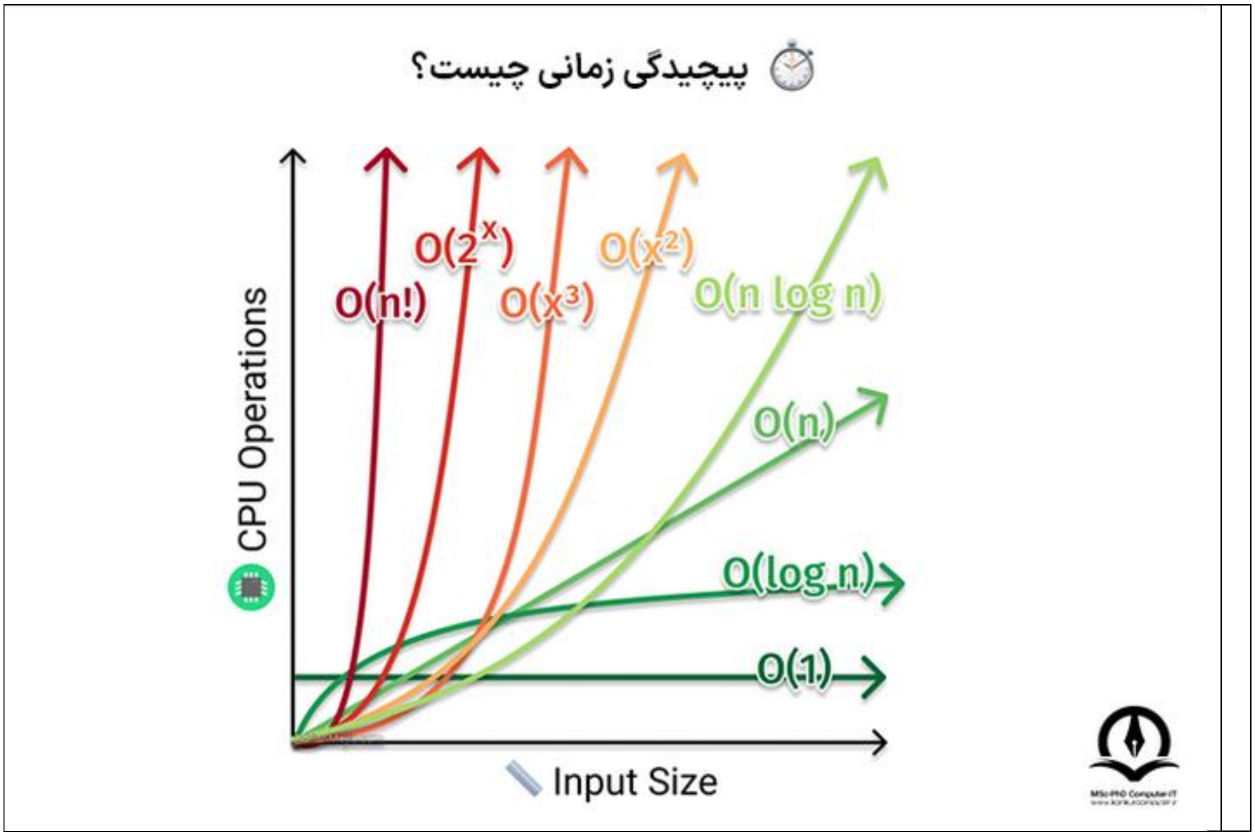
$$g(x) \in O(n)$$

این عبارت به ما می‌گوید در بدترین حالت الگوریتم من به طور خطی عمل می‌کند. و میدانیم توابع خطی از سرعت نسبتا قابل قبولی برخوردار است.

سوال، اگر آرایه مرتب بود آیا الگوریتمی برای پیدا کردن بهتر عنصر در آن وجود دارد؟ الگوریتمی که از مرتبه زمان خطی یا $O(n)$ بهتر باشد؟

بله. الگوریتم جستجوی باینری از مرتبه $O(\log_2 n)$ هست.

بگذارید قبل از معرفی الگوریتم یک مقایسه بین توابع مختلف در بی نهایت داشته باشیم.



همانطور که مشاهده می‌کنید logn پایین‌تر از n قرار دارد. اما سرعتی به مراتب بالا تر دارد برای مثال عددی اگر x=1024 ورودی داشته باشم تعداد دستورات برای تابع $f(x) = x$ و $g(x) = \log_2 x$ چقدر است؟ برای f(x) = 1024 و برای g(x) که یک تابع لگاریتمی در مبنا دو است g(x) = 10 هست.

برگردیم به سوال زیر؛

سوال، اگر آرایه مرتب بود آیا الگوریتمی برای پیدا کردن بهتر عنصر در آن وجود دارد؟ الگوریتمی که از مرتبه زمان خطی یا $O(n)$ بهتر باشد؟

بله الگوریتم جستجوی دودویی و یا binary search الگوریتمی است که می‌تواند عدد مد نظر را در ساختار مرتب شده بسیار سریع بیابد برای مثال اگر من ۱۰۲۴ عدد داشتم با ۱۰ مقایسه میتوانست بگوویید دقیقا عدد کدام هست و اگر نیست بگوید وجود ندارد. اما چگونه؟

یک دیکشنری را در نظر بگیرید یک دیکشنری بزرگ که از قضا index گذاری نشده اما مرتب کلمات از A تا Z کلمات را شامل شده است، می‌خواهید یک کلمه همانند Jaguar را در آن پیدا کنید. شما وسط دیکشنری را باز می‌کنید و به حرف m برخورد می‌کنید و یک سوال می‌پرسید J اولین حرف Jaguar از m جلوتر است یا عقب‌تر؟ عقب‌تر است پس به سادگی نصف دیکشنری یعنی از حرف M تا Z را حذف کردید و می‌دانید کلمه Jaguar در نیمه اول کتاب است. بار دیگر این کار را می‌کنید.

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T

U	V	W	X	Y
Z				

با یک مقایسه تمام هر چند هزار حروف از خود m تا آخرین کلمه z حذف می‌گردد بعبارتی اگر ۱۰۲۴ لغت داشتیم که نصفش از A تا m باشد و نصف دیگر از m تا z بسادگی نصف آن‌ها را حذف کردیم و از ۵۱۲ مقایسه‌ی عبث جلوگیری ککردیم.

دوباره این موضوع را تکرار می‌کنیم. اما این بار برای نیمه اول کتاب یعنی بخش باقی مانده.

A	B	C	D	E
F	G	H	I	J
K	L			

اینبار شما که کتاب را در دست دارید وسط نیمه‌ی اول کتاب را باز خواهید کرد؛ مثلا به حرف F بر میخورید؛ بار دیگر می‌پرسید آیا J که اول کلمه Jaguar هست از F بزرگتر هست؟ بله همینطور هست پس با قبل از آن کاری نداریم و جستجو را بین F و M ادامه خواهیم داد و صفحات کمتر از F را دور میریزیم. دوباره فضای حلمان نصف شد.

	G	H	I	J
K	L			

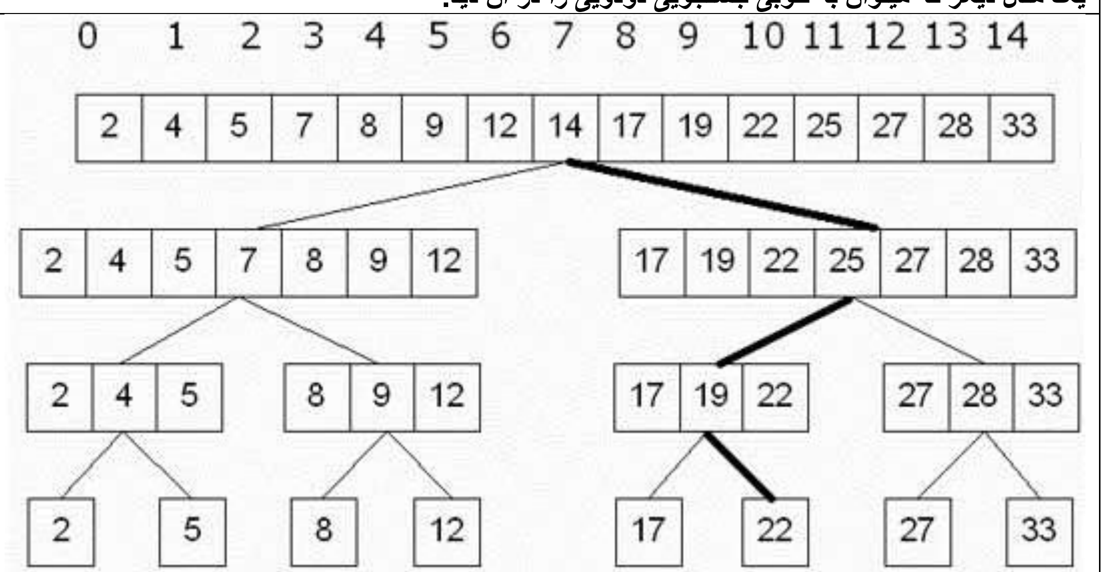
این روال تکرار می‌شود. بار بعد I انتخاب می‌شود. و با J مقایسه می‌گردد و G و H حذف می‌شود. پس؛

J	K	L		

دوباره برای آرایه‌ی بالا K انتخاب شده و با J مقایسه می‌شود و میدانیم J کوچک‌تر از آن است؛ تمام شد به مجموعه کلمات J رسیدیم و حال در مجموعه‌ی J بدنبال Jaguar می‌گردیم. اگر تعداد کم به ترتیب مقایسه می‌کنیم و اگر تعداد زیاد دوباره روی حروف دوم کلمات جستجوی دودویی را انجام میدهیم.

الگوریتم پایتون آن به شکل زیر آمده است:

یک مثال دیگر که میتوان به خوبی جستجویی دودویی را در آن دید.



```python
def binarySearch (arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it
        # can only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid + 1, r, x)

    else:
        # Element is not present in the array
        return -1
if __name__ == '__main__':
    arr = [ 2, 3, 4, 10, 40 ]
    x = 10

    # Function call
    result = binarySearch(arr, 00, len(arr)-1, x)

    if result != -1:
        print ("Element is present at index % d" % result)
    else:
        print ("Element is not present in array")
```

توضیح الگوریتم. در این الگوریتم از تکنیک بازگشتی استفاده شده است یعنی در خود آن تابع همان تابع صدا زده شده است یعنی آن تابع به کمک خودش حل می‌شود. L کوچک یا l اندیکس چپ‌ترین عنصر آرایه مدنظر و r اندیکس راست ترین عنصر آرایه را نشان می‌دهد.

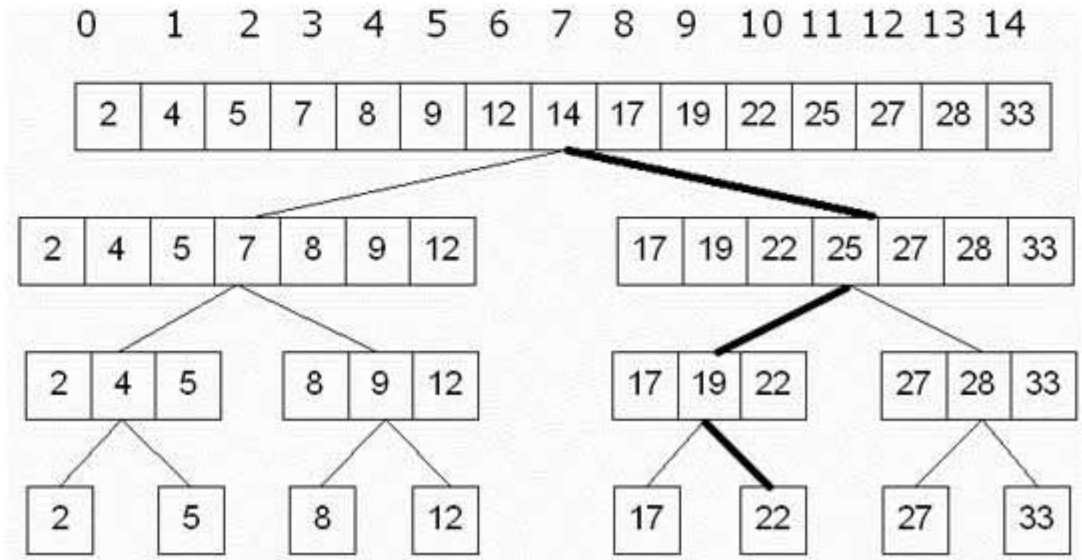
$$mid = \left\lceil \frac{l+r}{2} \right\rceil$$

و آنچه که در بالا آمده است یعنی

$$mid = l + \left\lceil \frac{r-l}{2} \right\rceil = \left\lceil \left(\frac{r}{2}\right) - \left(\frac{l}{2}\right) + l \right\rceil = \left\lceil \frac{l+r}{2} \right\rceil$$

فرقی با سخنان ما ندارد.

- **اگر عنصر در وسط آرایه یا A[mid] یافته شد کار تمام است وگرنه یا از آن بزرگتر است یا کوچکتر**
- **اگر کوچک تر بود مساله اصلی به زیر مساله مشابه‌ای اینبار با زیر آرایه‌ی چپ شکسته است**
- **اگر بزرگتر هم بود مساله‌اصلی به زیر مساله مشابه‌ای اینبار با نصف عناصر آرایه که سمت راست mid هستند ادامه خواهد یافت.**



حال چگونه این الگوریتم را به دنیای ریاضی ببریم و یا به طور رسمی چگونه پیچیدگی زمانی آن را محاسبه کنیم؟

تکنیک بازگشتی، توابع بازگشتی ، توابع متناظر ریاضی دارند که وابستگی حل کل مساله به زیر مسائل آن را نشان می‌دهد. برای مثال شما اینجا به جای حلقه‌ی for درخت دارید. محاسبات تکراری در قطعه قطعه کردن مساله یتان و دوباره حل کردن آن نهفته شده است **حتی در بسیاری از جاها شما می‌توانید حلقه‌های for را با روش بازگشتی جایگزین بنویسید و بالعکس یعنی تابع بازگشتی را هم می‌توانید به روش غیر بازگشتی یا iteratiove بنویسید.**

برای این توابع یعنی بازگشتی شمارش حلقه‌ها کارساز نیست؛ و شما بایستی توابع بازگشتی را ببینید مساله اصلی شما را به چند زیر مساله می‌شکنند و در هر زیر مساله چه بخشی از ورودی‌اصلی شما را دریافت می‌کند.

برای مثال برای الگوریتم جستجوی دودویی به شیوه بازگشتی چیزی که برای ما جالب است فلسفه‌ی حل به شیوه‌ی بازگشتی است؛ چرا که در هر تابع اتفاق زیادی نمی‌افتد چند if تودر تو و محاسبه‌ی یک میانه بنام mid که زمان خاصی ایجاد نمی‌کند اما آنچه که جالبسست حل مساله با دادن ساختار درختی به ورودی و شکاندن مساله به زیر مسائل است؛

در الگوریتم جستجوی دودویی

1. **میانه حساب میشود که هزینه چندانی ندارد**
2. **مقدار میانه با عدد درخواستی ما مقایسه میشود این هم هزینه چندانی ندارد**
3. **اگر جواب را یافته باشیم return x**
4. **وگرنه نصف ورودی دور ریخته می‌شود و تنها نصف آن دوباره به تابع داده می‌شود.بعبارتی:**

$$f(n) = f\left(\frac{n}{2}\right) + cost\ in\ each\ node\ of\ the\ tree!$$

تعجب نکنید هزینه‌ی هر نود درخت یعنی باقی دستورات که در یک گام اجرا می‌شود؛ یا در تابع *binary search* به جز توابع بازگشتی هست: محاسبه‌ی میانه!!! که از مرتبه‌ی یک هست.

پس تابع کلی الگوریتم جستجوی دودویی بدست آمد

$$f(n) = f\left(\frac{n}{2}\right) + O(c) \text{یا} O(1)$$

حال روش حل این تابع هم بسیار ساده است ؛ روش حل آن با تکنیک unfolding و یا روش درخت است؛ ایندو در جزوه جلسه سوم آمده است؛ و حل الگوریتم بالا بسیار ساده و برعهده دانشجو است.

تمرین‌های دانشجو
1. در بدترین حالت این الگوریتم از مرتبه *log n* هست چرا که به *log n* مقایسه نیاز دارد تا جواب را به شما بدهد همچنین در بهترین حالت هم با یک مقایسه جواب حاضر خواهد بود؛ پس در بهترین حالت *O(1)* پاسخ هست. حال حالت متوسط حالتی است که به طور معمول یا میانگین رخ می‌دهد. بعبارتی در کنار *worst case* و *best case* حالتی دیگر به نام *Average case* قرار دارد. از شما می‌خواهم *Average case* را بدست آورید برای الگوریتم جستجوی باینری چقدر است؟ (از تکنیک امید ریاضی استفاده بفرمایید؛ میانگین مقایسه ها) جواب همه جا هست تحقیق بفرمایید و جواب درست را با چند خط توضیح که کارتان را نشان می‌دهد ارسال نمایید.
2. با کمک تکنیک *unfolding* جستجوی دودویی را بفرمایید از چه مرتبه زمانی است؟
3. با کمک تکنیک درختی جستجوی دودویی را بفرمایید از چه مرتبه زمانی است؟

جواب ها:

جواب اول:

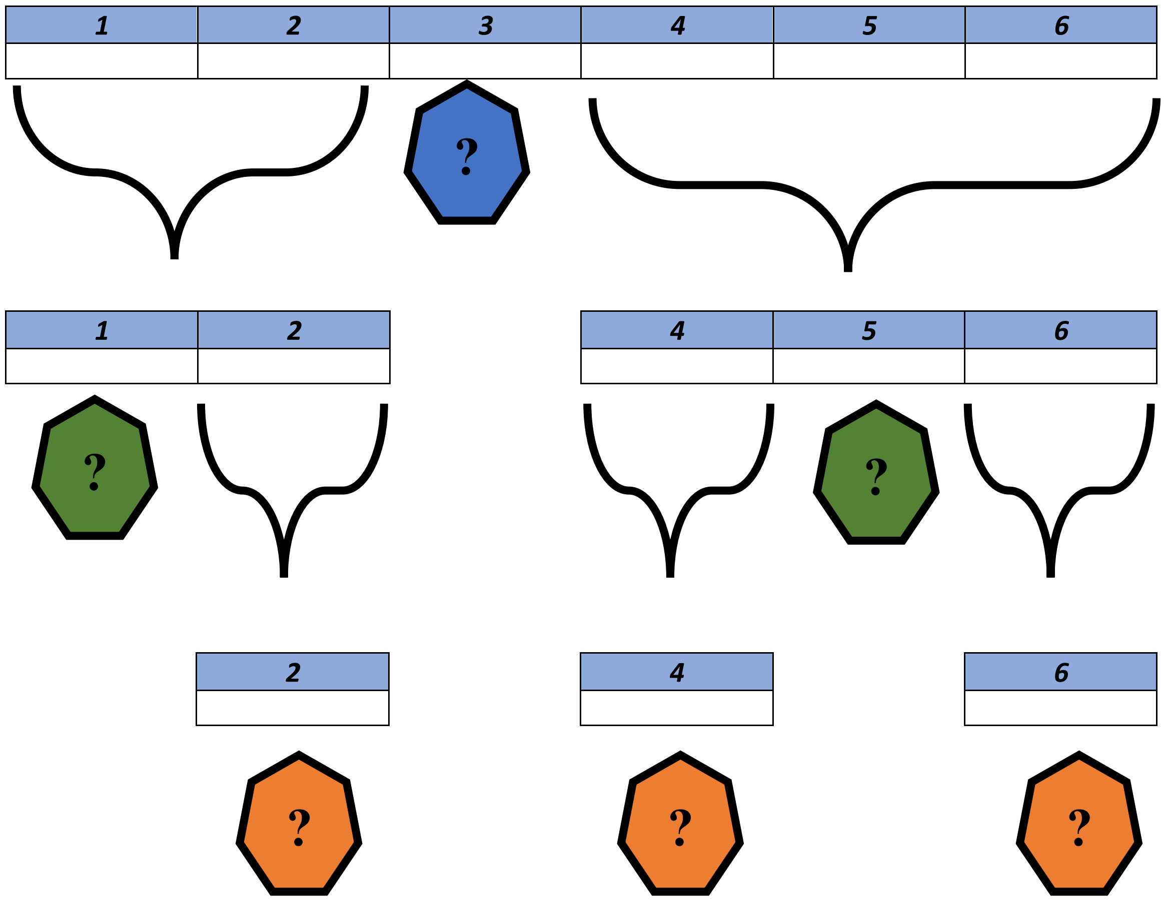
برای جستجوی خطی با آرایه نامرتب بگذارید اول حساب کنیم: برای جستجوی خطی با آراسه‌ی نامرتب داریم حالت میانگین به شکل زیر محاسبه می‌شود:

توضیح	**تعداد مقایسه**	**احتمال**
یا با یک مقایسه به جواب می‌رسیم	۱	$\frac{1}{n}$
یا با دو مقایسه به جواب می‌رسیم	۲	$\frac{1}{n}$
یا با سه مقایسه به جواب می‌رسیم	۳	$\frac{1}{n}$
	همینطور ادامه خواهد داشت ...	
یا با *k* مقایسه به جواب می‌رسیم	***K***	$\frac{1}{n}$
یا با *n* مقایسه به جواب می‌رسیم	***n***	$\frac{1}{n}$

$$\frac{1}{n} * 1 + \frac{1}{n} * 2 + \cdots + \frac{1}{n} * n = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n(n+1)}{n} \in O(n)$$

و خوب برای جستجوی دودویی چطور؟

برای این جستجو ؛ ساختار سیگما که برای مدلسازی حلقه‌های *for* به کار می‌رفت به ساختار درخت جایش را داده است. اما رویکردمان همان است آرایه‌ی ما *n* عنصر دارد و من باید نشان دهم هر خانه چند مقایسه خواهد داشت و میانگین بین مقایسه‌ها جواب من خواهد بود. این بار از فضای امید ریاضی استفاده نمی‌کنم و از میانگین وزن دار استفاده می‌کنم اما بگذارید به ترتیب پیش برویم تا متوجه کلیت امر سویم:



با توجه به شکل فوق من تونستم تعداد مقایسه ها برای هر خانه را بدست آورم برای خانه‌ی با ایندکس سه یک مقایسه برای خانه با ایندکس ۱ و ۵ دو مقایسه و برای خانه با ایندکس‌های دیگر اگر مقایسه‌ای باشد ۳ مقایسه خواهد بود. بهترین حالت یک مقایسه ، بدترین حالت ۳ مقایسه و حالت میانگین میانگین همه مقایسه هاست البته مقایسه‌ی وزن دار

1	2	3	4	5	6
2	3	1	3	2	3

برای مثال ما

$$\frac{\sum wi}{n} = \frac{1*1+2*2+3*3 = 1+4+9}{6} = {}^{14}/_{6}$$

و حال برای حالت کلی؛

- چند تعداد عنصر با یک مقایسه : ۱ عنصر
- چند تعداد عنصر با ۲ مقایسه : ۲ عنصر
- چند تعداد عنصر با ۳ مقایسه : ۴ عنصر
- چند تعداد عنصر با ۴ مقایسه : ۸ عنصر
- و الا آخر برای k مقایسه: 2^{k-1} عنصر وجو خواهد داشت.

کل عناصر n هست فرض های زیر را ببینید:

So We have Geometric Series Level for Dividing elements	We have n element in array
$1, 2, 4, 8$	15
۱ عنصر یک مقایسه‌ای ، دو عنصر دو مقایسه‌ای ، ۴ عنصر سه مقایسه‌ای و الی آخر	
$1, 2, 4, 5, 16, 32$	63
$1, 2, 4, 5, 16, 32,$ باقیمانده	75

نکته: روابط مختلفی می‌توان بین تعداد عناصر، تعداد مقایسه‌ها به طور کلی یا به تفکیک دسته بدست آورد که جالب ترین آنها در جدول زیر آمده است:

Levels		Number of element	
1,2,4,8		15	
الگوی جالب اول			
level	pattern	Num of comparison	Num of element by level
1	$[Log2\ (1)] + 1 = 1$	1	1
2	$[Log2\ (2)] + 1 = 2$	2	2
3	$[Log2\ (4)] + 1 = 3$	3	4
4	$[Log2\ (8)] + 1 = 4$	4	8
			Sum : 15
الگوی جالب دوم			
	Number of max comparison		Number of element
	$[Log2(15)] + 1 = 4$		15

پی به راحتی می‌توانیم بنویسم:

$$1 * 1 + 2 * 2 + 4 * 3 + 8 * 4 + \cdots + 2^{k-1} * k$$

خوب این تعداد مقایسه‌ها هست که بعدا تقسیم بر n خواهد شد ؛

$$\frac{1 * 1 + 2 * 2 + 4 * 3 + 8 * 4 + \cdots + 2^{k-1} * k}{n}$$

که k در اینجا عناصری هستند که بیشترین مقایسه را دارند و تعداد آن‌ها 2^{k-1} که اگر کل آرایه n باشد طبق آنچه در نکته آمده $[Log2(n)] + 1 = k$ مقایسه خواهد داشت؛ و اگر در 2^{k-1} هم k بدست آمده را قرار دهیم :

$$2^{k-1} * k = 2^{log2(n)} * log2(n)$$

پس در فرمول :

$$\frac{1*1+2*2+4*3+8*4+\cdots+n*log2(n)}{n}$$

و ما کران بالا و ***bigO*** میخواهیم پس :

$$\frac{1*1+2*2+4*3+8*4+\cdots+n*log2(n)}{n}$$
$$\leq \frac{n*log2(n)+n*log2(n)+n*log2(n)+n*log2(n)+\cdots+n*log2(n)}{n}$$
$$= log2(n)+log2(n)+log2(n)+log2(n)+\cdots+log2(n)$$
$$\approx\ log2(n)*log2(n)=2log2(n)$$

پس میانگین وزن دار به مجموعه‌ی $O(2log2(n))$ دارد و میدانیم $O(log2(n))$ و $(2log2(n))$ از یک مرتبه هستند پس به خواسته‌ی خود رسیدیم.

جواب دوم :

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

$$T\left(\frac{n}{2}\right)=\ T\left(\frac{n}{4}\right)+\ c\ ,T\left(\frac{n}{4}\right)=T\left(\frac{n}{8}\right)+c$$
$$\rightarrow T(n)=T\left(\frac{n}{4}\right)+2c\rightarrow T(n)=T\left(\frac{n}{8}\right)+3c$$

بسیار خوب الگویی به دست آمد اگر روال بالا را برای k بار تکرار کنم چه خواهد شد؟

$$T(n)=T\left(\frac{n}{2^k}\right)+kc$$

در حالتی که من آرایه‌ای به طول یک داشته باشم چند مقایسه نیاز است؟

$$T(1)=1$$

حال

$$n=\ 2^k\rightarrow\ \log_2 n=k$$

و جایگزاری نهایی ما را به جواب مد نظر می‌رساند؛

$$T(n)=T(1)+\log_2 n*c$$

پس :

$$T(n)\in O(log\,n)$$

جواب تمرین سوم.

$$\mathbf{1+1+1+\cdots+1} = \boldsymbol{log2(n)*1} = \boldsymbol{log2(n)}$$