جزوه کلاسی درس

تحلیل و طراحی الگوریتم ها

استاد خرسند

هدف درس : آشنایی با انواع الگوریتم ها و تحلیل آنها

کتاب درس:

Algorithmics
Theory and practice
G.Brassard
P.Bratley

منابع درس:

1. *Intruduction to algorithms*
   Second edition
   T.H.German
   C.E.leiserson
   R.L.Rivest
   C.Stein
2. *Algorithms*
   R.Sedgewiek
3. *The art of computer programming*
   D.Knuth

فصل 1 .....

انواع الگوریتم ها

- Greedy
- Divide and conquer
- Dynamic programming
- Exploring graphs

الگوریتم Algorithm :

یک فرآیند گام به گام که اگر به ترتیب خاصی دنبال شود ، کار به خصوصی انجام پذیرد. هر الگوریتم ویژگی های زیر را دارد:

- حداقل یک خروجی یا نتیجه دارد.
- هر یک از مراحل آن بدون ابهام و شدنی است.
- شرط خاتمه آن معلوم است.

مانند الگوریتم اخذ گواهینامه رانندگی ، الگوریتم استفاده از یک سایت کامپیوتری یا الگوریتم حل یک معادله درجه دو .

برنامه Program :

همان الگوریتم است که در چارچوب قواعد یک زبان برنامه نویسی نوشته شده است.

مراحل نوشتن یک برنامه :

برای نوشتن یک برنامه الگو های مختلفی وجود دارد. یکی از الگو های ساده به شرح زیر است.

- Requirment (درک نیازها ، شناخت مسئله و فراهم سازی نیازمندیها)
- Design (طراحی)
- Analysis (تحلیل)
- Refinment and coding (پالایش و برنامه نویسی ، بررسی نهایی نیازمندیها و در نهایت پس از تایید شروع به کدنویسی)
- Verification(بازبینی)

که مرحله آخر خود شامل سه عمل زیر است.

- Program testing
- Program debugging
- program Proving (اثبات درستی)

نکته : اثبات درستی یعنی اینکه برنامه را به صورت یک قضیه کلی تلقی کنیم و ثابت کنیم درست است.

در این درس بیشتر در مراحل Design و Analysis پرداخته و آنها را مورد بررسی قرار می دهیم.

در فاز آنالیز معمولا الگوریتم نوشته شده را به دو شکل تحلیل می کنیم:

1. مطمئن شویم درست است.
2. الگوریتم را با الگوریتم های مشابه مقایسه کنیم. فرضا مشخص کنیم میان الگوریتم های ممکن برای یک صورت مساله کدام الگوریتم حافظه مصرفی کمتری دارد ، یا کدام الگوریتم زمان مصرفی کمتری خواهد داشت.

مثال: الگوریتمی طراحی کنید که با کمترین زمان مصرفی دو عدد صحیح را در هم ضرب کند.

```
Function russe ( A,B )
        X[1] ← A
        Y[1] ← B
        i ← 1
        While X[1] > 1 do
                X[ i+1] ← X[i] div 2
                Y[ i+1] ← Y[i] + Y[i]
                i ← i+1
        end
        p ← 0
        while  i > 0  do
                if  X[i]  is odd
                        then  p ← p + Y[i]
                i ← i -1
        end
End
```

اجرای الگوریتم فوق با مقادیر :

A= 45 , B= 19

1	2	3	4	5	6	
45	22	11	5	2	1	X

1	2	3	4	5	6	
19	38	76	152	304	608	Y

1 2 4 8 16 32

i = 1 2 3 4 5 6 5 4 3 2 1 0
p = 0 608 760 830 855

مثال : الگوریتمی طراحی کنید که در آرایه n عنصری A ، ماکزیمم و مینیمم را تعیین کند.

```
1)  Procedure  Smaxmin( A,n,max,min )
2)      Max,min ← A(1)
3)      For  i ← 2  to  n  do
4)              If  A(i) > max
5)                      Then  max ← A(i)
6)              If  A(i) < min
7)                      Then  min ← A(i)
8)      End
9) End
```

اجرای الگوریتم با آرایه مقابل

1	2	3	4	5	
10	20	5	17	35	A

Trace table

n	**max**	**min**	**i**
5	10	10	2
-	20	-	3
-	-	5	4
-	-	-	5
-	35	-	6

تحلیل زمان مصرفی:

برای تعین زمان مصرفی یک الگوریتم از ایده ساده زیر استفاده می کنیم. دستورات را شماره گذاری می کنیم . تعداد دفعات اجرای هر دستور را پیدا می کنیم و در زمان مصرفی هر دستور ضرب می کنیم . عددهای حاصل را با هم جمع می کنیم تا زمان مصرفی الگوریتم بدست آید. توجه داشته باشید درباره الگوریتم صحبت می کنیم نه برنامه.

1- شماره دستورالعمل	2- تعداد دفعات تکرار	3- زمان مصرفی هر اجرا	2*3
1	1		
2	2		
3	n		
4	n-1		
5	n-1حداکثر		
6	n-1		
7	n-1 حداکثر		
8	n-1		
9	1		
	مجموع = شاخص زمان مصرفی		مجموع = زمان مصرفی

به نظر می رسد عناصر ستون 3 را به درستی نمی توان بدست آورد. زیرا بستگی به سخت افزار ، روش کاری کامپیوتر ، سیستم عامل و ... دارد.

برای حل این مشکل ستون 3 را از بحث خارج می کنیم و مجموع عناصر ستون 2 را به عنوان شاخص زمان مصرفی در نظر می گیریم. البته این شاخص دقیق نیست زیرا زمان مصرفی دستورات مختلف با هم متفاوت است.

تمرین: الگوریتم فوق را دوباره بنویسید . سعی کنید که دستورات اضافی آن را کم کنید.

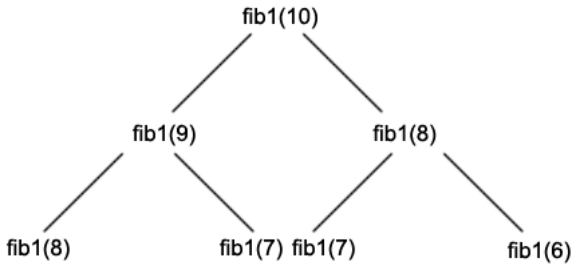
مثال: الگوریتمی طراحی کنید که n امین عدد دنباله فیبوناچی را تولید کند.

$$F_n = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ (F_{n-1}) - (F_{n-2}) & if \quad n >= 2 \end{cases}$$

Golden Ratio : $F(n) = \frac{1}{\sqrt{5}}\left[\Phi^n - (-\Phi)^{-n}\right]$ و $\Phi = \frac{\left(1+\sqrt{5}\right)}{2}$

الگوریتم ها:

```
Function  fib1(n)
        If  n = 0  then  return(0)
        If  n = 1  then return(1)
        If  n >= 2  then
                Return ( fib1(n-1) + fib1 (n-2))
End
```




```
Function fib2(n)
        i ← 1
        j ← 0
        for  k ← 1  to  n  do
                j ← i + j
                i ← j - i
        end
        return (j)
end

function fib3(n)
        i ← 1
        j ← 0
        k ← 0
        h ← 1
        While  n > 0  do
                If   n   is odd
                        Then [ t ← j.h
                                j ← i.h + j.k + t
                                i ← i.k + t]
                t ← h^2
                h ← 2.h.k + t
                k ← k^2 + t
                n ← n div 2
        end
        Return (j)
End
```

تمرین : الگوریتم fib3 و fib2 و fib1 را از نظر منطقی مقایسه کنید. به این نتیجه می رسید که دنباله فیبوناچی تعریف دیگری نیز باید داشته باشد و ممکن است جمله عمومی آن متفاوت باشد.

مثال: زمان مصرفی الگوریتم زیر را فرموله نمایید:

```
Function  Euclid (m,n)
        While  m> 0  do
                t ← n  mod  m
                n ← m
                m ← t
        end
        Return (n)
End
```

حل:

Trace table

n	m	t
32	24	8
24	8	0
8	0	

نکته : این الگوریتم برای به دست آوردن بزرگترین مقسوم علیه مشترک دو عدد است.

تمرین : مطمئن شوید الگوریتم فوق را متوجه شده اید.

محاسبه زمان مصرفی:

حلقه while مثل for نیست. پس نمی توانیم بفهمیم که چندبار اجرا می شود. سراغ مفهوم حلقه می رویم. سعی می کنیم مفهوم حلقه را فرموله نماییم و از روی آن تعداد دفعات گردش حلقه را مشخص می کنیم.

فرض کنید m_i, n_i مقادیر m و n در پایان گردش i ام حلقه باشند. فرض کنید حلقه حداکثر k بار اجرا شود. بنابراین I می تواند مقادیر 1و2و...و k را داشته باشد. بحث را در جهتی دنبال می کنیم که برای k یک سقف بدست آوریم. به این ترتیب در ابتدای ورود به الگوریتم مقادیر m و n با m_0 و n_0 نشان داده می شود. حال به سراغ الگوریتم می رویم. سعی می کنیم بین مقادیر مختلف m و n ارتباط برقرار کنیم.

$$\begin{matrix} n_1 = m_0 \\ m_1 = n_0 \quad \text{mod} \quad m_0 \end{matrix} \rightarrow \begin{matrix} n_2 = m_1 \\ m_2 = n_1 \quad \text{mod} \quad m_1 \end{matrix} \Rightarrow \begin{cases} n_i = m_{i-1} \\ m_i = n_{i-1} \quad \text{mod} \quad m_{i-1} \\ i = 1,2,3,....,k \end{cases}$$

فراموش نکنید که باید برای k مقداری را بدست آوریم. به این منظور گاهی مجبوریم از معلومات ریاضیمان استفاده کنیم. در این مثال از این قضیه استفاده می کنیم.

(n mod m) < n/2

در نتیجه:

$$m_i = n_{i-1} \quad \text{mod} \quad m_{i-1} < \frac{n_{i-1}}{2} = \frac{m_{i-1}}{2}$$

$$* \qquad m_i < \frac{m_{i-2}}{2}$$

$m_k = 0$ است اما m_{k-1} یک باقیمانده صحیح و مثبت است پس حتما بزرگتر و مساوی یک است. به همین دلیل از m_{k-1} شروع کردیم.
فرض کرده ایم حلقه k بار اجرا می شود، در پایان بار K ام ، m_k مساوی صفر می شود.
به این ترتیب از رابطه * نتیجه می شود.

$$1 \le m_{k-1} < \frac{m_{k-3}}{2} < \frac{m_{k-5}}{2^2} < \frac{m_{k-7}}{2^3} < \cdots < \frac{m_0}{2^d}$$

نکته: K تعداد دفعات گردش حلقه است. اگر فرد باشد k = 2d +1 و اگر زوج باشد k = 2d در نظر می گیریم.

$$\Rightarrow \frac{m_0}{2^d} > 1 \Rightarrow m_0 > 2^d \Rightarrow 2^d < m_0 \Rightarrow d < \log \frac{m_0}{2}$$

پس d حداکثر می تواند $\log \frac{m}{2}$ باشد ، پس :

$$\begin{cases} k = 2d \Rightarrow k = 2d < 2.\log m_0 \\ k = 2d+1 \Rightarrow k = 2d+1 < (2.\log m_0)+1 \end{cases} \quad \Rightarrow \quad k \in O(\log m)$$

برای حلقه های تکراری شرطی ، رابطه منطقی حاکم بر حلقه را پیدا می کنیم ، این رابطه یا روابط را به سمتی می بریم که تعداد دفعات گردش حلقه معلوم شود. در مثال 2 این کار نسبتا ساده اتفاق افتاد. در سایر موارد نیز به همین صورت عمل می کنیم.

ساختمان داده Data structure :

برای نوشتن هر برنامه باید به دو مورد اصلی توجه کنیم:

1. الگوریتم
2. Data structure

که این دو مورد بر روی هم تاثیر متقابل دارند. یعنی تغییر در یکی سبب تعریف دیگری می شود. به مثال زیر توجه کنید.
مثال: الگوریتمی طراحی کنید که اعداد زیر را به صورت صعودی مرتب کند.

10 , 5 , 25 , 15 , 7

برای طراحی الگوریتم اگر عددهای فوق در آرایه باشند الگوریتم سورت کردن به یک شکل تعریف می شود و اگر عددهای فوق در پشته یا هر چیز دیگری باشد الگوریتم سورت کردن به شکل دیگری می باشد.

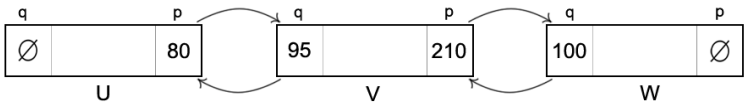
انواع Data structure :

1. آرایه:

ساختاری است که تمام عناصرش از یک نوع است و خانه های متوالی حافظه را هم اشغال می کند.

2. لیست پیوندی Link list :

```
Structure R
    {
        int a;
        Struct R*p;
        Struct R*q;
    } u,v,w;
    u.p = &v;
    u.q = null;
    v.p = &w;
    w.p = null;
    w.q = &v;
    v.q = &u;
```



3. گراف Graph :

مجموعه ای از دوتای ها شامل راس و یال که هر ضلع متکی به دو راس است.

G = (V,E)

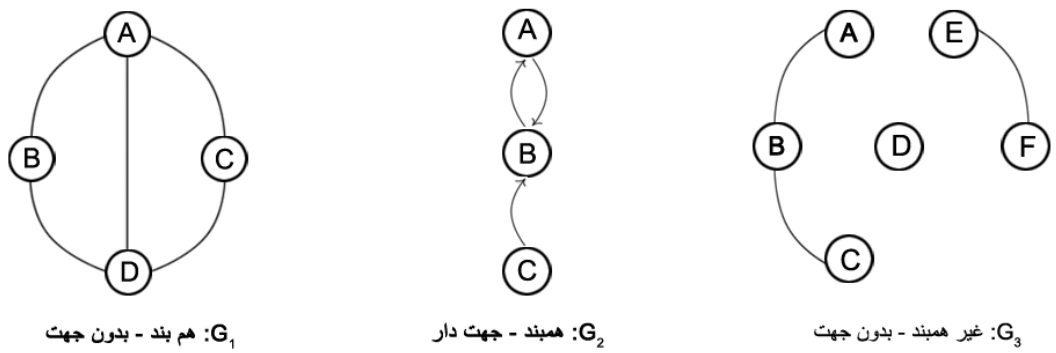
V: Vertex (راس)

E: Edge (یال- ضلع)

گراف دو نوع می باشد: بدون جهت undirected و جهت دار directed

هر ضلع بدون جهت مانند این است که دو طرفه جهت دار است. به همین خاطر جهت ها را مشخص نکرده ایم.

مثال:



G_1: هم بند - بدون جهت G_2: همبند - جهت دار G_3: غیر همبند - بدون جهت

در یک گراف بدون جهت زوج رئوس ، زوج مرتب نیستند ، بنابراین (v_0,v_1) با (v_1,v_0) با هم یکسانند.

$V(G_1)$ = { A , B , C , D }

$E(G_1)$ = { (A , B) (B , D) (A , C) (C , D) (A , D) }

در یک گراف جهت دار هر لبه با زوج مرتب $< v_0,v_1 >$ نشان داده می شود. که پیکانی از v_0 به v_1 ترسیم می شود.

بنابراین $< v_0,v_1 >$ و $< v_1,v_0 >$ دو لبه متفاوت را نشان می دهند.

$V(G_2) = \{ A , B , C \}$
$E(G_2) = \{ < C , B > < A , B > < B , A > \}$

$V(G_3) = \{ A , B , C , D , E , F \}$
$E(G_3) = \{ (A , B) (B , C) (E , F) \}$

گراف حالت خاصی از لیست پیوندی است و از نمایش های مربوط به لیست پیوندی استفاده می کند . اما گراف نمایش های مخصوص به خود را دارد مانند لیست همجواری.

چند تعریف :

- **گراف هم بند connected :**

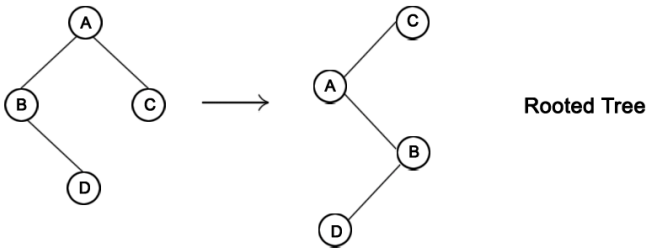
یک گراف را هم بند گویند ، هرگاه بین هر دو راس گراف یک مسیر وجود داشته باشد.

- **مسیر Path :**

یک مسیر یک دنباله از رئوس گراف است به طوریکه بین هر دو راس متوالی از این دنباله یک ضلع وجود داشته باشد.

- **درخت Tree :**

گرافی که n راس داشته باشد ، همبند باشد و دور نداشته باشد یعنی n-1 ضلع داشته باشد ، ساختار درختی پیدا می کند.
معمولا در درخت ها ، یک عنصر خاص را ، به نام ریشه در نظر گرفته و درخت حاصل را Rooted Tree گویند.



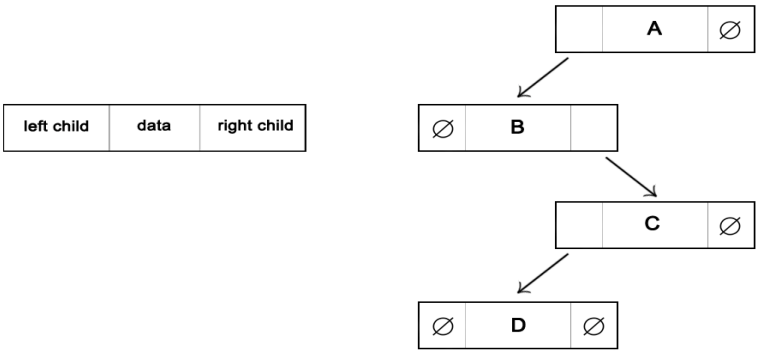
می توانیم برای ریشه درخت ، وظایف دیگری هم در نظر بگیریم . مثلا بگوییم ریشه درخت عنصری باشد که توسط آن می توانیم وارد درخت شویم .

نکته : از ریشه به هر عنصر ، فقط و فقط یک مسیر وجود دارد.

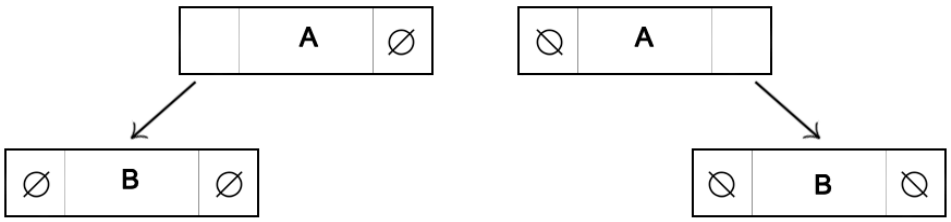
معمولا در درخت ارتباط عناصر را ، ارتباط ولی به فرزند در نظر می گیریم ، یعنی هر parent به فرزندانش اشاره می کند . یا می توانیم بگوییم هر عنصر به نسل بعدی اشاره می کند. معمولا برای مطالعه درخت ها به شکل خاصی از آنها به نام درخت دودویی اشاره می کنیم.

- **درخت دودویی Binary Tree :**

در درخت دودویی هر عنصر دو فرزند متمایز ، به نام فرزند چپ و فرزند راست دارد. هر گره در درخت دودویی به شکل زیر است.



دو Data structure را با هم مساوی گویند ، هرگاه یکی کپی دیگری باشد.(یعنی دارای شکل ساختاری یکسان و محتوای یکسانی باشد)



این دو درخت نا مساوی هستند

- **عمق درخت :**

ماکزیمم سطح های عنصرهای درخت را ، عمق یا ارتفاع درخت گویند.

- **درخت دودویی پر Full Binary Tree :**

یک درخت دودویی به عمق k را پر گویند هر گاه هر عنصر آن دو فرزند داشته باشد.

در یک درخت دودویی پر به عمق k گزاره های یر برقرار است:

1. تعداد عناصر سطح n ام برابر 2^{k-1} است.
2. تعداد عناصر درخت برابر $2^k - 1$ است.

برای اثبات گزاره 1 از استقرا بر روی i استفاده می کنیم.

ابتدای استقرا:

برای $i = 1$ گزاره 1 برقرار است.

فرض استقرا:

فرض کنید برای $i = p$ گزاره 1 برقرار باشد ، یعنی در سطح p ام ، 2^{p-1} عنصر دارد. حال ثابت می کنیم حکم استقرا نیز برقرار است. به این منظور کافی است به تعریف درخت توجه کنیم . یعنی هر عنصر آن دو فرزند دارد . پس وقتی در سطح p ام ، 2^{p-1} عنصر داشته باشیم ، در سطح بعدی ، $p + 1$ ام دو برابر سطح قبل عنصر داریم یعنی :

$$2.2^{p-1} = 2^p$$

پس در سطح $P + 1$ ام نیز گزاره 1 برقرار است.

برای اثبات گزاره 2 از گزاره 1 استفاده می کنیم:

برای یک درخت دودویی به عمق k ، طبق گزاره 1 تعداد عناصر برابر است با :

$$2^0 + 2^1 + \cdots + 2^{k-1} = 2^k - 1$$

- **درخت دودویی کامل Complete Binary Tree :**

یک درخت n عنصری را کامل گویند هرگاه عناصرش متناظر به عناصر n , ... , 1 از درخت دودویی پر باشد. در یک درخت دودویی پر ، عناصر را شماره گذاری می کنیم ، ریشه شماره 1 را دارد و در هر سطح شماره ها از چپ به راست افزایشی اند.

درختی را درخت کامل به عمق K گویند دو شرط را داشته باشد:

1. برگ های آن در سطح k یا $k - 1$ باشد.
2. تمام گره های سطح k از منتهی الیه سمت چپ آغاز شده باشد.

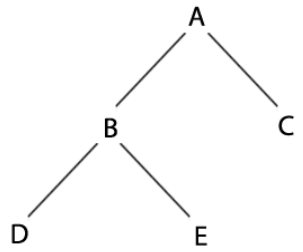
از تعریف فوق نتیجه می شود ، هر درخت پر ، کامل است. بعلاوه میان عناصر درخت دودویی کامل روابط زیر برقرار است.

1. Lchild (i) = 2i
2. Rchild (i) = 2i +1
3. Parent (i)n = [i/2]

بر همین اساس است که هر درخت دودویی کامل را با آرایه نشان می دهیم ، هر آرایه را به یک درخت دودویی کامل تبدیل می کنیم. بعبارت دیگر در الگوریتم ها اگر ضروری باشد ، آرایه را با درخت کامل عوض می کنیم و یا درخت کامل را با آرایه عوض می کنیم.

1	2	3	4	5
A	B	C	D	E

$\Rightarrow$



اگر یک درخت دودویی کامل به عمق k داشته باشیم ، این درخت حداکثر $2^k - 1$ عنصر دارد و حداقل عنصر در آن 2^{k-1} است.معمولا تعداد عناصر درخت را با n نشان می دهیم.

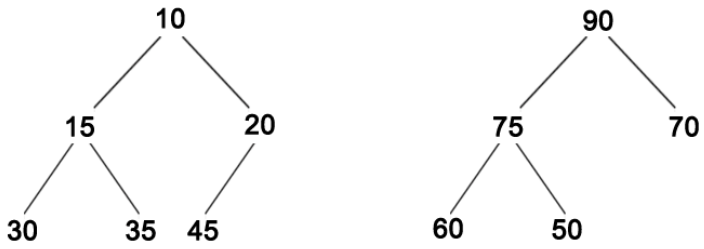
$$n = 2^k - 1 \rightarrow n + 1 = 2^k \rightarrow k = \log_2^{n+1}$$

اگر یک درخت n عنصری داشته باشیم که کامل هم باشد ، عمق این درخت از $\log n$ بدست می آید (تابعی از $\log n$ است). به عکس ، اگر آرایه ای n عنصر داشته باشد ، از روی آن درختی با عمق $\log n$ بدست می آید. اینجاست که باید تصمیم بگیریم برای الگوریتم هایمان ، آرایه را به شکل درخت نگاه کنیم و یا برعکس درخت را به شکل آرایه.

در یک درخت دودویی پر با n عنصر عمق درخت برابر $\log(n+1)$ است اما در یک درخت کامل با n عنصر عمق درخت برابر است با $\left\lfloor \log_2^n \right\rfloor + 1$.

4. Heap :

Heap یک درخت دودویی کامل است که میان عناصرش یک رابطه ترتیبی وجود دارد ، این رابطه ترتیبی معمولا به شکل های زیر است. (در درخت اصلی و هر یک از زیر درخت هایش ، ریشه از فرزندانش کوچکتر (بزرگتر) است.)



Min Heap **Max Heap**

مهمترین کاربرد Heap در سیستم های عامل است . معمولا سیستم های عامل برای مدیریت حافظه از Heap استفاده می کنند. عددهای داخل Heap اندازه بلاک های مختلف حافظه است.

Heap یک Data structure کاربردی است.

الگوریتم های مربوط به Heap :

در این بخش با الگوریتم های ساده ای آشنا می شویم ، که آرایه را به Heap تبدیل می کند ، در Heap جای عناصر را عوض می کند و یا در نهایت یک Heap را سورت می کند.

الگوریتم Make-Heap :

```
Procedure make-heap ( T [1 .. n] )
        For   i ← ( n div 2 )  down to  1  do
                Sift-down ( T , I )
        End
End
```

الگوریتم sift-down :

```
Procedure sift-down ( T[1..n] , i )
        K ← i
        Repeat
                j ← k
                if  2j ≤ n   and  T[2j] > T[k]
                        then  k ← 2j
                if  2j < n   and  T[2j + 1] > T[k]
                        then  k ← 2j + 1
                exchange ( T[j]  and  T[k])
        until  j = k
end
```

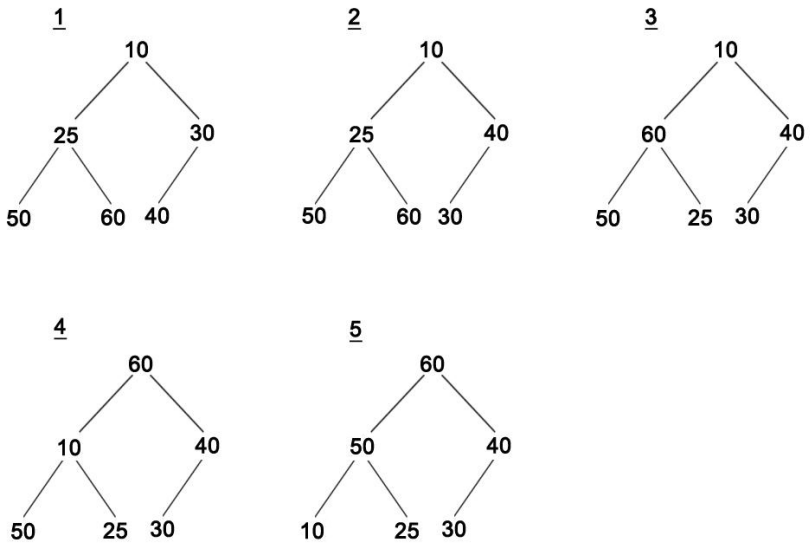
نکته : الگوریتم sift-down در درخت T ، زیر درختی با ریشه i را به Heap تبدیل می کند.

الگوریتم Make-heap را با آرایه زیر آزمایش می کنیم.

1	2	3	4	5	6
10	25	30	50	60	40

n	i	k	j
6	3	3	3
		6	6
	2	2	2
		5	5
	1	1	1
		2	2
		4	4

n	i	
6	3	Sift_down (T,3)
	2	Sift_down (T,2)
	1	Sift_down (T,1)



الگوریتم Heap-sort :

```
Procedure Heap-sort ( T[ 1 .. n] )
        Make-heap( T )
        For  i ← n  step-1 to  2  do
                Exchange ( T[1]  and  T[ i ] )
                Sift-down ( T[ 1 .. i-1] , 1)
        End
End
```

تحلیل الگوریتم های فوق :

به دو روش این کار را انجام می دهیم. ابتدا به صورت کلی زمان مصرفی الگوریتم ها را پیدا می کنیم.

زمان مصرفی Make heap = $n/2 \times$ زمان مصرفی sift down

زمان مصرفی Sift down = تعداد دفعات گردش حلقه داخلی آن

تعداد دفعات گردش حلقه = عمق درخت

در یک درخت دودویی کامل با n عنصر و عمق k ، حداکثر تعداد عناصر برابر با یک درخت دودویی پر با همان عمق است. به عبارت دیگر عمق این درخت ، حداکثر $\log (n+1)$ است.

زمان مصرفی Make heap $= n/2 \times \log_2^{n+1} \Rightarrow O(n \log n)$

حال سعی می کنیم همین موضوع را با دقت بیشتری بررسی کنیم.

مجددا می خواهیم زمان مصرفی Make-heap را تعیین کنیم. باید به سراغ sift-down برویم. چون زمان مصرفی Make-heap به صورت (زمان مصرفی sift-down $\times$ n/2) تعیین می شود.

در الگوریتم sift-down زمان مصرفی را حلقه Repeat-until می سازد که برای هر ورودی تعداد دفعات گردش حلقه متفاوت است.

فرض کنید ، n حداکثر تعداد دفعات گردش حلقه Repeat-until باشد. سعی می کنیم برای n یک سقف محاسبه پذیر پیدا کنیم.

پس اگر بتوانیم برای n یک سقف قابل قبول پیدا کنیم ، زمان مصرفی Make-heap بدست می آید. به حلقه Repeat-until توجه کنید. شرط خاتمه این حلقه به مقادیر j و k بستگی دارد.در هر اجرای حلقه ، j جدید بر اساس k ای که در اجرای قبل حلقه محاسبه شده

است ، به دست می آید. بنابر این به مقدار j توجه می کنیم. فرض کنید j_t مقدار j پس از اجرای دستور $j \leftarrow k$ در t امین گردش حلقه باشد.

$j_1 = i$ $\quad$ t = 1,2, … , m

$j_2 = 2j_1$ یا $2j_1+1$ درابتدای گردش دوم حلقه

$j_3 = 2j_2$ یا $2j_2+1$ در ابتدای گردش سوم حلقه

توجه نمایید فرض کرده ایم ، حلقه حداکثر n بار می چرخد. از طرفی ما می دانیم j_1 همان مقدار i است . رابطه اخیر را برای دفعات مختلف اجرای حلقه داریم:

$j_t \geq 2.j_{t-1}$ $\quad$ t = 2,3, … , m

$n \geq j_m \geq 2j_{m-1} \geq 2^2.j_{m-2} \geq \cdots \geq 2^{m-1}.j_1$

$\Rightarrow n \geq 2^{m-1}.i \Rightarrow 2^{m-1} \leq n/i \Rightarrow m-1 \leq \log(n/i)$

حداکثر تعداد دفعات گردش حلقه Repeat-until (سقف m) :

$m \leq 1+\log(n/i)$

حال سعی می کنیم بر اساس این رابطه ، برای الگوریتم Make-heap عددی را بدست آوریم. یعنی سقف زمانی بدست آوریم . به الگوریتم Make-heap توجه کنید. حلقه آن به ازای مقادیر مختلف i اجرا می شود. یک بار i برابر n/2 است ، بار بعدی (n/2)-1 تا 1 . در هر بار سراغ sift-down می رویم ، حلقه sift-down بیشترین تعداد دفعات گردش حلقه را داشته باشد ، به این ترتیب زمان مصرفی Make-heap با رابطه زیر تعریف می شود:

$$\sum_{i=1}^{\lceil n/2 \rceil}(1+\log(n/i))$$

بحث را در جهت محاسبه $\sum$ فوق دنبال می کنیم.

$$\sum_{i=1}^{\lceil n/2 \rceil}(1+\log(n/i)) = \lfloor n/2 \rfloor + \sum_{i=1}^{\lceil n/2 \rceil}\log(n/i) \qquad \text{(الف}$$

به این منظور هم از تعریف لگاریتم استفاده می کنیم ، هم از تعریف درخت کامل.

یک درخت کامل را در نظر بگیرید ، در بدترین حالت این درخت پر می شود . مطابق روش گفته شده ، در سطح k+1 ام عناصر از 2^k تا $2^{k+1}-1$ شماره گذاری شده اند. رابطه (الف) را در سطح k+1 ام می نویسیم ، در بدترین حالت که درخت ما به درخت پر تبدیل می شود.

$$A = \sum_{i=2^k}^{2^{k+1}-1}\log(n/i) = \log\left(\frac{n}{2^k}\right) + \log\left(\frac{n}{2^k+1}\right) + \cdots + \log\left(\frac{n}{2^{k+1}-1}\right)$$

همانطور که مشاهده می شود ، از عددهای لگاریتم می گیریم که صورت آنها n است اما مخرج کسر آنها از 2^k یکی یکی افزایش پیدا کرده اند تا ($2^{k+1}-1$). یعنی از عددهای لگاریتم می گیریم که کوچک می شوند. به این ترتیب میان log های فوق اولین جمله از بقیه بزرگتر است ، بنابراین می توان نتیجه گرفت :

$$A \leq 2^k.\log\left(\frac{n}{2^k}\right) \qquad \text{(ب}$$

حال این موضوع را تعمیم میدهیم . یعنی اینکه فرض می کنیم ، درختی که میسازیم ، عمق d را داشته باشد. فرض کنید ، در هر سطح آن رابطه (ب) برقرار باشد ، به این ترتیب اگر رابطه (الف) را در کل این درخت در نظر بگیریم ، نتیجه خواهیم گرفت :

$$\sum_{i=1}^{\lceil n/2 \rceil}\log(n/i) < \sum_{k=0}^{d}2^k.\log\left(\frac{n}{2^k}\right)$$

حال سعی می کنیم باز هم برای این رابطه ، سقف معنی دار پیدا کنیم . طبق تعریف درخت دودویی پر برای عمق d داریم:

$$d = \lfloor \log(n/2) \rfloor \Rightarrow d \le \log(n/2)$$
$$\Rightarrow d + 1 \le \log n$$

با استفاده از ویژگی های لگاریتم ، رابطه(ج) را می توانیم ساده نماییم.

تمرین: از رابطه $d + 1 \le \log n$ نتیجه بگیرید : $d - 1 > \log(n/8)$

راهنمایی: به طرفین رابطه فوق 2- را اضافه نمایید.

تمرین : از نتیجه تمرین قبل و رابطه ج نتیجه بگیرید :

$$\sum_{k=0}^{d} 2^k . \log\left(\frac{n}{2^k}\right) \le 3.n$$

پس به این ترتیب از تمرین های اخیر نتیجه گرفتیم زمان مصرفی الگوریتم Make-heap حداکثر $3.n + \lfloor n/2 \rfloor$است.

تمرین: درستی رابطه ب و ج را توضیح دهید.

چند الگوریتم دیگر:

الگوریتم alter-heap :

این الگوریتم در در خت T مقدار عنصری با شماره i را برابر v قرار می دهد و heap را دوباره می سازد.

```
Procedure alter-heap ( T[ 1..n ] , i , v)
        X ← T[ i ]
        T[ i ] ← v
        If   v < x   then sift-down ( T, i )
                        Else  percolate ( T , i )
End
```

الگوریتم percolate :

```
Procedure  percolate ( T[ 1..n ] , i )
K ← i
Repeat
        J ← k
        If    j > 1  and  T[ j div 2 ] < T[ k ]  then   k ← j div 2
        Exchange  T[ j ] and T[ k ]
Until  j = k
End
```

الگوریتم find-max :

```
Function  find-max ( T[ 1..n ] )
        Return T[1]
```

الگوریتم insert-node :

```
Procedure  insert-node ( T[1..n] , v )
T[ n + 1 ] ← v
Percolate ( T[ 1 .. n +1 ] , n + 1 )
End
```

نمادهای مجانبی Asymptotic Notations :

همانطور که مشاهده شده ، برای پیدا کردن زمان مصرفی یک الگوریتم نتوانستیم به عدد و رقم دقیقی برسیم. یعنی آمدیم تعداد دفعات اجرای هر یک از دستورات را شمردیم و عملا از آن به عنوان زمان مصرفی استفاده کردیم. در صورتی که می دانستیم ، زمان مصرفی یک دستور با زمان مصرفی دستور دیگر ، طبیعتا متفاوت است ، حتی اگر ظاهری یکسان داشته باشند.

یکی از راه های که بتوانیم دقت کارمان را با این نوع عددها بالا ببریم ، استفاده از نماد های مجانبی است. نماد های مجانبی سبب می شوند برای این نوع عبارات ، کف یا سقف یا هر دو را تعریف کنیم.

در این بخش با سه نماد O و Ω و θ آشنا می شویم . کار این نماد ها ارائه سقف یا کف و یا هر دو است. به عبارت دیگر برای زمان مصرفی حداکثر مقدار یا حداقل مقدار و یا هر دو را تعریف می کنیم.

نماد O :

فرض کنید f تابعی دلخواه از $N \to R^*$ باشد. منظور از O(f(n)) مجموعه توابعی است که سقف آنها تحت شرایطی بر حسب f(n) تعریف می شود. به عبارت دیگر:

$$O(f(n)) = \left\{ t : N \to \mathbb{R}^+ \middle| (\exists c \in \mathbb{R}^*)(\exists n_0 \in \mathbb{N})(\forall n \ge n_0)[t(n) \le c \cdot f(n)] \right\}$$

نکته : $\mathbb{R}^* = \mathbb{R}^+ \cup \{0\}$

مثال : نشان دهید تابع $t(n) = 5n - 9$ در گروه $O(n^2)$ قرار دارد.

$$t(n) = 5n - 9 \in O(n^2)$$

حل:

به این منظور باید سراغ تعریف تابع رویم و نشان دهیم C و n_0 ای وجود دارد که شرایط فوق برایش برقرار است .

طبق تعریف داریم : $[5n - 9 \le c \cdot n^2] \Rightarrow c \cdot n^2 - 5n + 9 \ge 0$

حال باید C و n_0 را تعیین کنید تا رابطه برقرار باشد. این کار را با روش سعی و خطا می توان انجام داد. به عنوان مثال برای رابطه بالا به ازای $n_0 = 5$ و $c = 5$ رابطه برقرار است . پس به این ترتیب این تابع در گروه $O(n^2)$ قرار می گیرد.

یک نکته مهم در مورد سقف ها این است که تا جایی که امکان دارد واقعی تعریف شوند. به عبارت دیگر تا جایی که ممکن باشد سقف را پایین بیاوریم.

حال می خواهیم بررسی کنیم که آیا $t(n) \in O(n)$ هست یا نه ؟

برای این منظور باز هم از تعریف استفاده می کنیم.

$$(\exists c \in \mathbb{R}^*)(\exists n_0 \in \mathbb{N})(\forall n \ge n_0)[5n - 9 \le c \cdot n]$$

$$5n - 9 \le c \cdot n \Rightarrow 5n - c \cdot n \le 9 \Rightarrow (5 - c) \cdot n \le 9 \Rightarrow n \le \frac{9}{5 - c}$$

پس تابع t(n) در گروه $O(n)$ قرار گرفت و سقف پایین آمد.

مثال: نشان دهید رابطه "$\in O$" خاصیت تعدی (transitive) دارد.

$$f(n)\in O(g(n))\cap g(n)\in O(h(n))\xrightarrow{?} f(n)\in O(h(n))$$

حل:

ثابت می کنیم:

$$\left(\exists c\in\mathbb{R}^+\right)\left(\exists n_2\in\mathbb{N}\right)\left(\forall n\ge n_2\right)\left(g(n)\le c_2\cdot h(n)\right)$$

$n_3\ge\max(n_1,n_2)$ و $c_3\ge c_1\cdot c_2$ پس

$$f(n)\le c_1\cdot g(n)\le \underbrace{c_1\cdot c_2}_{c_3}\cdot h(n)$$

$$\left(\exists c_3\in\mathbb{R}^+\right),c_3=\left(c_1\cdot c_2\right)\left(\exists n_3\in\mathbb{N}\right)\left(\forall n\ge n_3\right)$$

مثال: فرض کنید f و g دو تابع دلخواه به شرط $f,g:\mathbb{N}\to\mathbb{R}^*$. نشان دهید :

$$O\left(f(n)\right)=O\left(g(n)\right)\qquad iff\qquad f(n)\in O\left(g(n)\right)\cap g(n)\in O\left(f(n)\right)$$

حل:

فرض کنید $O\left(f(n)\right)=O\left(g(n)\right)$ باشد ، چون نمادهای O تعریف مجموعه ای دارند لذا وقتی دو مجموعه با هم مساوی باشند یعنی یکی زیر مجموعه دیگری است و بلعکس .

$$1)O\left(f(n)\right)\subseteq O\left(g(n)\right)$$

$$2)O\left(g(n)\right)\subseteq O\left(f(n)\right)$$

رابطه اول نشان دهنده آن است که هر تابعی که سقفش با f(n) تعریف می شود ، سقفش با g(n) هم تعریف می شود.

رابطه اول نشان دهنده آن است که هر تابعی که سقفش با g(n) تعریف می شود ، سقفش با f(n) هم تعریف می شود.

طبق رابطه اول هر تابعی که سقفش با f(n) تعریف می شود ، سقفش با g(n) هم تعریف می شود ، یکی از این توابع خود f(n) است.

$$f(n)\in O\left(f(n)\right)\to f(n)\in O\left(g(n)\right)$$

طبق رابطه دوم هر تابعی که سقفش با g(n) تعریف می شود ، سقفش با f(n) هم تعریف می شود ، یکی از این توابع خود g(n) است.

$$g(n)\in O\left(g(n)\right)\to g(n)\in O\left(f(n)\right)$$

تمرین : دو تابع f و g مثال بزنید بطوریکه $f(n)\notin O\left(g(n)\right)$ و $g(n)\notin O\left(f(n)\right)$ ، برد هر دو تابع اعداد طبیعی است.

$$f,g:\mathbb{N}\to\mathbb{N}^+$$

مثال : فرض کنید $f,g:\mathbb{N}\to\mathbb{R}$. نشان دهید:

$$O\left(f(n)+g(n)\right)=O\left(\max\left(f(n),g(n)\right)\right)$$

حل:

برای اینکه نشان دهیم دو مجموعه با هم مساویند باید نشان دهیم هر کدام زیر مجموعه دیگری است.

$$1)\ O\left(f(n)+g(n)\right)\subseteq O\left(\max\left(f(n),g(n)\right)\right)$$

اثبات :

$$O\left(f(n)+g(n)\right)=\left\{t\mid:\mathbb{N}\to\mathbb{R}^*\middle|\left(\exists c_1\in\mathbb{R}^*\right)\left(\exists n_1\in\mathbb{N}\right)\left(\forall n\ge n_1\right)\qquad\left[t(n)\le c_1\left(h(n)+g(n)\right)\right]\right\}$$

$$if \qquad h(n)=\max\left(f\ (n),g(n)\right) \qquad \forall n\geq n_1 \qquad t(n)\leq c_1\left(f\ (n)+g(n)\right)$$
$$t(n)\leq c_1\left(h(n)+g(n)\right)$$
$$t(n)\leq \underbrace{2c_1}_{c_2}\cdot h(n)$$

2) $O\left(\max\left(f\ (n),g(n)\right)\right)\subseteq O\left(f\ (n)+g(n)\right)$

از این ویژگی یک نتیجه مهم می گیریم :

O(max) = O(بیشترین درجه)

به مثال زیر توجه کنید.

مثال: آیا $O\left(5n^3-10n^2+15n-25\right)\in O\left(n^3\right)$.

حل:

$$O\left(5n^3-10n^2+15n-25\right)=O\left(\max\left(5n^3-10n^2+15n,25\right)\right)$$
$$O\left(5n^3-10n^2+15n\right)=O\left(\max\left(5n^3-10n^2,15n\right)\right)$$
$$O\left(5n^3-10n^2\right)=O\left(\max\left(5n^3,10n^2\right)\right)$$
$$O\left(5n^3\right)=O\left(n^3\right)$$

مثال: فرض کنید $f,g:\mathbb{N}\to\mathbb{R}$ باشد ، $\lim\limits_{n\to\infty}\dfrac{f\ (n)}{g(n)}$ یک عدد مثبت باشد ، آنگاه $O\left(g(n)\right)=O\left(f\ (n)\right)$

مثال:

تعریف حد:

$$\forall\varepsilon_0 \qquad \exists n_0>0 \qquad \forall n\geq n_0\to\left|\frac{f\ (n)}{g(n)}-l\right|<\varepsilon$$
$$-\varepsilon<\frac{f\ (n)}{g(n)}-l<\varepsilon \qquad \to \qquad l-\varepsilon<\frac{f\ (n)}{g(n)}<l+\varepsilon \qquad \to\left(l-\varepsilon\right)g(n)<f\ (n)<\left(l+\varepsilon\right)g(n)$$
$$\underbrace{\left(l-\varepsilon\right)g(n)<f\ (n)}_{g(n)\in O(f\ (n))} \quad , \quad \underbrace{f\ (n)<\left(l+\varepsilon\right)g(n)}_{f\ (n)\in O(g\ (n))} \qquad \Rightarrow O\left(f\ (n)\right)=O\left(g(n)\right)$$

نماد Ω :

$\Omega\left(f\ (n)\right)$ مجموع تمام توابعی را نشان می دهد که کف آنها بر اساس f(n) تعریف شده است. در این تعریف هم c,n_0 مقادیر دلخواهی هستند که تعریف می شوند:

$$\Omega\left(f\ (n)\right)=\left\{t:\mathbb{N}\to\mathbb{R}^+\ \middle|\ \left(\exists c\in\mathbb{R}^*\right)\left(\exists n_0\in\mathbb{N}\right)\left(\forall n\geq n_0\right)\left[t(n)\geq c\cdot f\ (n)\right]\right\}$$

بلافاصله از تعریف فوق نتیجه می شود:

$$f,g:\mathbb{N}\to\mathbb{R}^*$$
$$\underbrace{f\ (n)\in O\left(g(n)\right)}_{\searrow} \qquad iff \qquad g(n)\in\Omega\left(f\ (n)\right)$$
$$\left(\exists c_1\right)\left(\exists n_1\right)\left(\forall n\geq n_1\right)\left[f\ (n)\leq c_1\cdot g(n)\right]$$
$$g(n)\geq\underbrace{\frac{1}{c_1}}_{c_2}\cdot f\ (n)$$

نماد θ:

مجموعه توابعی که هم کف آنها و هم سقف آنها بر حسب f(n) تعریف می شود.

$$\theta(f(n)) = \left\{ t : \mathbb{N} \to \mathbb{R}^* \middle| \left(\exists c_1, c_2 \in \mathbb{R}^+\right)\left(\exists n_0 \in \mathbb{N}\right)\left(\forall n \ge n_0\right)\left[c_2 \cdot f(n) \le t(n) \le c_1 \cdot f(n)\right] \right\}$$

برای مثال برای تابع $f(n) = 5n + 5$ رابطه $f(n) \in \theta(n)$ برقرار است.

همچنین برای تابع $g(n) = 10n^2 - 5n + 90$ رابطه $g(n) \in \theta(n^2)$ برقرار است.

مثال: فرض کنید $f, g : \mathbb{N} \to \mathbb{R}^+$ و $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l$ ثابت کنید : $f(n) \in \theta\left(g(n)\right)$

حل:

براحتی و با استفاده از تعریف حد داریم:

$$\forall \varepsilon_0 \qquad \exists n_0 > 0 \qquad \forall n \ge n_0 \to \left| \frac{f(n)}{g(n)} - l \right| < \varepsilon$$

$$-\varepsilon < \frac{f(n)}{g(n)} - l < \varepsilon \qquad \to \qquad l - \varepsilon < \frac{f(n)}{g(n)} < l + \varepsilon \qquad \to \left(l - \varepsilon\right) g(n) < f(n) < \left(l + \varepsilon\right) g(n)$$

$$\Rightarrow f(n) = \theta\left(g(n)\right)$$

به طور کلی برای θ می توان عبارت زیر را در نظر گرفت:

$$\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

تمام توابعی که سقف و کف آنها با f(n) تعریف می شوند ، توابعی هستند که کف آنها با f(n) و سقف آنها با f(n) تعریف می شوند.

اسقرای ساختاری Constructive Induction :

تابع f(n) را در نظر بگیرید. فرض کنید از ما بخواهند این تابع را با نماد مجانبی نشان بدهیم ، یعنی بگوییم بشکل O و Ω یا θ تابع مفروض است. به این منظور از یک راه ساده استفاده می کنیم.

ادعایی را مطرح می کنیم و سعی می کنیم آن ادعا را ثابت کنیم. چون معمولا با توابعی سروکار داریم که متغییر های اصلی آنها از نوع طبیعی هستند ، مانند تعداد دفعات گردش حلقه ، تعداد عناصر آرایه و ... بنابراین ادعایمان را باید برای هر مقدار n ثابت کنیم و همین جاست که پای استقرا به وسط می آید.

اما این استقرا را باید از آخر به اول بیان نمود. یعنی از ادعایمان به عنوان نتیجه استقرا یا حکم استقرا استفاده کنیم و فرض استقرا را از روی آن بسازیم. نتیجه بگیریم از فرض استقرا به حکم استقرا می رسیم. سپس ابتدای استقرا را بیان می کنیم و باز هم نتیجه می گیریم از ابتدای استقرا به فرض استقرا می توانیم برسیم. حال اگر مراحل فوق را کنار هم قرار دهیم به نظر می رسد یک استقرا از آخر به اول ساخته ایم. در هر مرحله از کار اگر اثباتهایمان دچار مشکل شود ادعای اولیه خودمان را تغییر خواهیم داد ، یا شرطی را پیدا می کنیم که تحت آن شرط ادعایمان برقرار باشد. توجه داشته باشید ، تابع f(n) ازهمان ابتدا معلوم نیست تابعی باشد که برایش نماد O یا Ω یا θ به سادگی تعریف شود. به عبارت دیگر طبیعت روش فوق مبنی بر سعی و خطا می باشد.

جملات فوق روش استقرای ساختاری را نشان می دهند. یعنی استقرایی که بر اساس ساختار تابع f(n) تعریف می شود.

مثال : تابع f را بشکل زیر در نظر بگیرید ، آنرا با نماد مجانبی نشان دهید.

$$f : \mathbb{N} \to \mathbb{N}$$

$$f(n) = \begin{cases} 0 & n = 0 \\ n + f(n-1) & O.W \end{cases}$$

حل:

به صورت مسئله نگاه می کنیم ، نمی دانیم از نماد O استفاده کنیم یا Ω یا θ. سعی می کنیم با بررسی تابع نمادی را حدس بزنیم و در قدم بعد ادعایمان را در چارچوب نوشته فوق ثابت نماییم. گاهی اوقات حدس اولیه خودمان به سادگی به دست می آید اما معلوم نیست همیشه اینطور باشد.

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

$$f(n) \in \theta(?)$$

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

$$f(n) = n + f(n-1) = n + (n-1) + f(n-1) = n + (n-1) + (n-2) + f(n-3) = \cdots \qquad \Rightarrow f(n) \in O\left(n^2\right)$$

ادعا می کنیم f(n) به شکل $O\left(n^2\right)$ است. چون تابع را می شناسیم ادعایمان را دقیقتر بیان می کنیم. ادعا می کنیم:

$$f(n) = an^2 + bn + c$$

این ادعای ما به عنوان حکم اسقرا یا آخر اسقرا بیان می شود. باید ثابت کنیم به ازای هر n ادعایمان برقرار است. به ادعای فوق Partially Specified Induction Hypothesis گویند که آنرا با HI نشان می دهیم و چون به n بستگی دارد ، آنرا HI(n) می گوییم.

این عبارت این موضوع را می رساند که هنوز استقرایی ثابت نشده است، ادعای را مطرح کرده ایم که شاید برای مقادیری از n برقرار باشد . باید نشان دهیم ادعایمان برای هر مقدار از n برقرار است. سراغ HI(n-1) می رویم. آنرا از روی HI(n) می سازیم.

$$HI(n) : f(n) = an^2 + bn + c \Rightarrow HI(n-1) : f(n-1) = a(n-1)^2 + b(n-1) + c$$

حال نشان می دهیم از HI(n-1) ، یعنی فرض اسقرا به HI(n) یعنی حکم استقرا تحت شرایط مسئله می توانیم برسیم .توجه نمایید برای آنکه HI(n) برقرار باشد یا باید a,b,c را تعیین کنیم و یا برایش محدودیت های را بدست آوریم.

فرض می کنیم HI(n-1) برقرار باشد. می خواهیم ثابت کنیم تحت شرایط مسئله به HI(n) می رسیم.

$$f(n) = n + f(n-1) = n + a(n-1)^2 + b(n-1) + c$$

$$\Rightarrow an^2 + n(1 - 2a + b) + (a - b + c) = an^2 + bn + c$$

برای آنکه HI(n) برقرار باشد باید جای f(n) بتوانیم $an^2 + bn + c$ را قرار دهیم. یعنی رابطه اخیر مساوی $an^2 + bn + c$باشد ، فراموش نکنیم مجهولاتمان a,b,c است. برای پیدا کردن آنها می توانیم طرفین تساوی فوق را باهم ، هم ارز بگیریم و یا جملات تساوی فوق را به یک طرف بیاوریم فاکتورگیری کنیم و تمام ضرایب را متحد با صفر قرار دهیم.

$$\begin{cases} 1 - 2a + b = b \to a = 1/2 \\ a - b + c = c \to a = b \to a = b = 1/2 \end{cases}$$

پس به این ترتیب نشان دادیم اگر a,b برابر 1/2 باشند ، از HI(n-1) به HI(n) می رسیم.

$$f(n) = \frac{1}{2}n^2 + \frac{1}{2}n + c$$

سراغ شرط می رویم. طبق صورت مسئله F(0)=0 است. از طرف دیگر طبق آنچه تا به حال بدست آوردیم F(0)=c است. چون باید این دو مقدار مساوی باشند پس به این ترتیب c=0 شد و با این حساب f(n) تابعی است که بشکل زیر باید باشد.

$$f(n) = \frac{1}{2}n^2 + \frac{1}{2}n \qquad c_2 \cdot n^2 \le \frac{1}{2}n^2 + \frac{1}{2}n \le c_1 \cdot n^2$$

مثال: تابع t را بااستفاده از نماد مجانبی نشان دهید.

$$t:\mathbb{N}^+ \to \mathbb{R}^+$$

$$t(n)=\begin{cases} a & n=1 \\ b\cdot n^2+n\cdot t(n-1) & OW \end{cases} \qquad a,b\in\mathbb{R}^+$$

حل:

$$\begin{aligned} t(n) &= b\cdot n^2+n\cdot t(n-1) \\ &= b\cdot n^2+n\left(b\cdot(n-1)^2+(n-1)\cdot t(n-2)\right) \\ &= b\cdot n^2+b\cdot n\cdot(n-1)^2+n\cdot(n-1)\cdot t(n-2) \\ &= b\cdot n^2+b\cdot n\cdot(n-1)^2+n\cdot(n-1)\cdot\left(b\cdot(n-2)^2+(n-2)\cdot t(n-3)\right) \\ &= b\cdot n^2+b\cdot n\cdot(n-1)^2+b\cdot n\cdot(n-1)\cdot(n-2)^2+n\cdot(n-1)\cdot(n-2)\cdot t(n-3) \end{aligned}$$

تابع t(n) از n! حتما بزرگتر است. نتیجه می شود تابع t(n) کف n! دارد. این موضوع را باید برای هر مقدار n ثابت کنیم.

$$HI(n): t(n)\in\Omega(n!)$$

ادعای فوق را بصورت ساده تر بیان می کنیم.

$$HI(n): \exists u\in\mathbb{R}^+ \qquad \exists t(n)\geq u\cdot n!$$

از روی HI(n) فوق ، HI(n-1) را بصورت زیر در نظر می گیریم:

$$HI(n-1): \exists u\in\mathbb{R}^+ \qquad t(n-1)\geq u\cdot(n-1)!$$

ثابت می کنیم تحت شرایط مسئله از HI(n-1) به HI(n) می رسیم. یعنی اگر HI(n-1) برقرار باشد ، HI(n) هم برقرار است.

$$t(n)=bn^2+nt(n-1)\geq bn^2+n\cdot u\cdot(n-1)!=bn^2+un!\geq un!$$

همانطور که اشاره می شود عبارت فوق از $(u\cdot n!)$ بزرگتر است، پس از HI(n-1) به HI(n) رسیدیم. برای u هم شرطی پیدا نکردیم. یعنی برای هر مقدار u از HI(n-1) به HI(n) می رسیم.

حال سراغ ابتدای استقرا می رویم. طبق صورت مسئله t(1)=a است. طبق ادعای فوق $t(1)\geq u\cdot 1!$ است. از این دو رابطه نتیجه می شود :

$$a\geq u$$

a را صورت مسئله به ما داده است. بنابر این کافی است u را کوچکتر یا مساوی a در نظر بگیریم ، u هر چه باشد ادعای ما ثابت است.

حال سراغ سقف این تابع می رویم . با توجه به شکل t(n) حدسی را مطرح می کنیم . سعی می کنیم حدس را بشکل بهتری مطرح نماییم.

$$HI(n): t(n)\in O(n!)$$

ادعای فوق را بشکل ساده تر زیر بیان می کنیم.

$$HI(n): \exists v\in\mathbb{R}^+ \qquad \exists t(n)\leq v\cdot n!$$

از روی HI فوق ، HI(n-1) را بشکل زیر تعریف می کنیم.

$$HI(n-1): \exists v\in\mathbb{R}^+ \qquad \exists t(n-1)\leq v\cdot(n-1)!$$

نشان می دهیم تحت شرایط مسئله از HI(n-1) به HI(n) می رسیم.

$$t(n)=bn^2+nt(n-1)\leq bn^2+n\cdot v\cdot(n-1)!\leq v\cdot n!$$

چون bn^2 مثبت است پس نمی توانیم نتیجه فوق را بگیریم . حال باید ادعایمان را تصحیح کنیم. علت آنکه از HI(n-1) به HI(n) نتوانستیم برسیم ، وجود جمله bn^2 است. سعی می کنیم این جمله را از بین ببریم به این منظور به ادعایمان یک جمله منفی اضافه می کنیم. شکل جدید ادعای ما بصورت زیر خواهد شد.

$$HI(n): t(n)\leq v\cdot n!-w\cdot n, w>0$$

مجددا سراغ HI(n-1) می رویم:

$$HI(n-1): t(n-1)\leq v\cdot(n-1)!-w\cdot(n-1)$$

باید نشان دهیم اگر HI(n-1) برقرار باشد ، HI(n) هم برقرار می شود.

$$* \quad t(n) = bn^2 + n \cdot t(n-1)$$
$$\leq bn^2 + n\left(v \cdot (n-1)! - w\,(n-1)\right)$$

برای حذف کردن bn^2 ، جمله wn منفی را اضافه کردیم که از مرتبه 1 بود ولی با قرار دادن در HI(n-1) آنرا تبدیل به مرتبه 2 کرده و در * از bn^2 کم کردیم.

$$= bn^2 + v \cdot n! - wn \cdot (n-1)$$
$$= v \cdot n! + n\left(bn - w\,(n-1)\right)$$
$$= v \cdot n! + n\left(n(b-w) + w\right)$$

برای آنکه رابطه اخیر بشکل زیر در بیاید ، کافی است داشته باشیم:

$$t(n) \leq v \cdot n! - wn$$
$$(b-w)n + w \leq -w$$
$$bn - wn + w \leq -w$$
$$w \geq \frac{bn}{n-2} \rightarrow n \geq 3 \rightarrow n = 3 \rightarrow w \geq 3b$$

به نظر می رسد مقدار w رابدست آورده ایم ، پس اگر $w \geq 3b$ برای هر مقدار n,w مورد قبول است.

$w = 3b$ انتخاب می کنیم. سعی می کنیم v را بدست آوریم. بعلاوه به این نکته توجه کنید که برای پیدا کردن w ، $n \geq 3$ است. یعنی برای n=1 و n=2 ، w فوق برقرار نیست.

سراغ t می رویم ، طبق فرض فوق :

$$n = 1 \rightarrow \quad \underbrace{t(1) = a}_{A} \rightarrow \quad \underbrace{t(1) \leq v \cdot 1! - 3b}_{B}$$
$$A, B \Rightarrow a \leq v - 3b \rightarrow \quad v \geq a + 3b$$
$$\left.\begin{array}{l} t(2) = 4b + 2t(1) = 4b + 2a \\ t(2) \leq v \cdot 2! - 6b \end{array}\right\} \Rightarrow 4b + 2a \leq 2v - 6b \rightarrow v \geq a + 5b$$

با توجه به مثبت بودن مقادیر a و b ، اگر v را بزرگتر و مساوی $a + 5b$ در نظر بگیریم ، HI(n) برای n=1 و n=2 هم برقرار است. به این ترتیب ثابت کردیم

$$t(n) \leq (a + 5b)n! - 3b \cdot n$$

به این ترتیب سقف این تابع نیز پیدا شد.

مثال: الگوریتمی طراحی کنید که در یک آرایه n عنصری ، عناصر ماکزیمم و می نیمم را تعیین نمایید. زمان مصرفی این الگوریتم را بدست آورید. در مقایسه با الگوریتم های مشابه ، زمان مصرفی الگوریتم خود را بررسی نمایید.

```
1) Procedure Smaxmin( A,n,max,min )
2)     Max,min ← A(1)
3)     For i ← 2 to n do
4)         If A(i) > max
5)             Then max ← A(i)
6)         If A(i) < min
7)             Then min ← A(i)
8)     End
9) End
```

این الگوریتم در فصل یک معرفی شد.

1- شماره دستورالعمل	2- تعداد دفعات تکرار
1	1
2	2
3	n
4	n-1
5	n-1حداکثر
6	n-1
7	n-1 حداکثر
8	n-1
9	1
	مجمـوع = شـاخص زمان مصرفی

به نظر می رسد که در خط 4 تا 7 باید تغییراتی اعمال شود به این صورت که :

```
4)        If  A(i) > max
5)              Then  max ← A(i)
6)        else If  A(i) < min
7)              Then  min ← A(i)
```

تا اینجای کار دو راه حل برای یک صورت مسئله بدست آوردیم . حال باید زمان مصرفی و مقدار حافظه مصرفی این دو الگوریتم را بدست آوریم.

$\sum(n)$ شاخص زمان مصرفی است. پس پیدا کردن این عدد در مورد الگوریتم ساده فوق دچار مشکل می شود. (منظور دستورات 5 و 7 است). از طرفی دیگر نماد های مجانبی را مطالعه کردیم ، پس می آییم برنامه را ساده می کنیم.

در هر الگوریتم دستوراتی را به عنوان دستورات شاخص انتخاب می کنیم . به جای آنکه بر روی کل الگـوریتم تعـداد دفعـات اجـرای تمـام دستورات الگوریتم را حساب کنیم ، این کار را برای دستورات شاخص انجام می دهیم.

دستور شاخص دستوری است که اجرای آن به عنوان نماینده اجرای بخشی از الگوریتم قابل در نظر گرفتن است.

معمولا دستورات شرطی دستورات شاخص خوبی هستند.

با توضیحات فوق ، مقایسه بین عناصر را به عنوان دستور شاخص (Barometer) انتخاب می کنیم.

زمان مصرفی الگوریتم اول :

$$T_1(n) = 2(n-1)$$

و زمان مصرفی الگوریتم دوم :

$$T_2(n) = \begin{cases} 1(n-1) \\ 2(n-1) \end{cases}$$

که $1(n-1)$ برای بهترین حالت و $2(n-1)$ در بد ترین حالت می باشد.

پس مشاهده می شود که هر دو در گروه $O(n)$ هستند.

با این که الگوریتم اولی و دومی از نظر زمان مصرفی تفاوت زیادی دارند ، اما هر دو در گروه $O(n)$ قرار دارند. پس اگر مقیاس اندازه گیـر بیان O باشد ، این دو از نظر زمان مصرفی یکسان هستند.

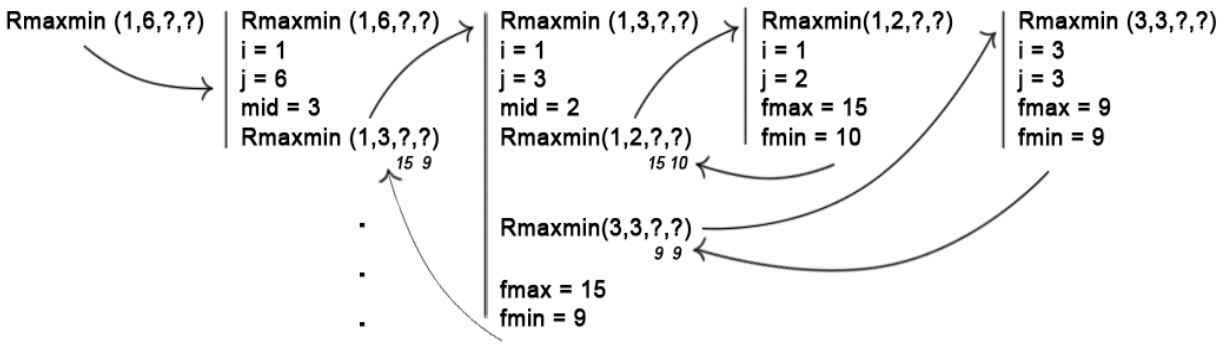
```
Procedure  Rmaxmin (i,j ,fmax ,fmin )
        Case:
                :i=j:    fmax,fmin←A(i)
                :i=j-1:  if  A(i)<A(j)
                                Then[ fmax ← A(j), fmin ← A(i) ]
                                Else[ fmax ← A(i), fmin ← A(j) ]
                :else:   mid ← [ (i+j )/2 ]
                         Rmaxmin (i , mid , gmax, gmin )
                         Rmaxmin ( mid +1 , j , hmax , hmin)
                         Fmax ← max (gmax , hmax)
                         Fmin ← min( gmin,hmin )
        End
End
```

پس به نظر می رسد در مورد این الگوریتم باید روش دیگری غیر از Barometer اتخاذ کرد.

ابتدا کار این الگوریتم بازگشتی را مرور می کنیم . این الگوریتم در آرایه A ، از اندیس i تا j عناصر max و min را تعیین می کند. آرایه A از 1 تا n است ، که global این الگوریتم است.

در اولین بازخوانی i را 1 می دهیم و j را n .

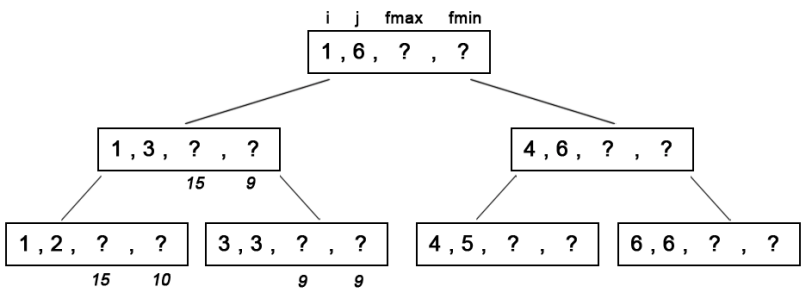
1	2	3	4	5	6	
10	15	9	25	7	16	A



تمرین: اجرای الگوریتم فوق را با دست دنبال نمایید. Page ها را به طور کامل با ذکر جزئیات بنویسید. رفت و آمد بین page ها و ترتیب بازخوانی ها را به خوبی متوجه شوید.

درخت بازخوانی بازگشتی:

معمولا ساختن page ها از نظر بیان جزئیات بسیار مناسب است. اما در مواردی کار بسیار طولانی می شود . در چنین حالت هایی شکل ساده تر page ها در چارچوب یک درخت به کار می آید. در این درخت هرpage با یک عنصر درخت جایگزین می شود. برای مثال این نوع درخت برای مثال بالا به شکل زیر خواهد بود:



تمرین:درخت فوق را مجددا بسازید. مطمئن شوید مراحل کار را متوجه شده اید.

تعیین زمان مصرفی الگوریتم های بازگشتی:

برای الگوریتم های بازگشتی ابتدا سعی می کنیم مفهوم الگوریتم را متوجه شویم ، سپس سعی می کنیم با استفاده از فرمول های ریاضی مفهوم الگوریتم را بیان کنیم . معمولا مفهوم الگوریتم به ما کمک لازم را می نماید.

طبق مفهوم الگوریتم زمان مصرفی الگوریتم به n (تعداد عناصر آرایه) بستگی دارد . n کم یا زیاد شود زمان مصرفی کاهش یا افزایش می یابد. بنا بر این فرض می کنیم T(n) زمان مصرفی الگوریتم برای حالتی باشد که آرایه ورودی n عنصر دارد. سعی می کنیم برای T(n) با توجه به شکل الگوریتم تعاریف مناسب بنویسیم. در چنین مواردی ساختار case الگوریتم ها به ما کمک می کند.

فرض کنید در هر حالت تعداد اجرای دستورات شاخص را شمارش می کنیم .(به نظر می رسد راه دیگری نداریم)

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ T\left(\lfloor n/2\rfloor\right)+T\left(\lceil n/2\rceil\right)+2 & O.W\end{cases}$$

به اینگونه توابع Recurrence Function گویند. یعنی نشان می دهند دستور Barometer چند بار اجرا شده است.

به این ترتیب زمان مصرفی الگوریتم بازگشتی را نیز به دست آوردیم. اما چیزی که می ماند آن است که این تابع با توابع T_1 و T_2 قبلی قابل مقایسه نیست. به همین دلیل برای این توابع بازگشتی باید فرآیند حل را مطرح نمود. یعنی یک معادله بازگشتی را باید حل کنیم. برای حل از یک روش ساده می توان شروع نمود تا روش های کاملا ریاضی.

معمولا در ابتدای کار شکل معادله را ساده می کنیم و سپس معادله را حل می کنیم. برای ساده کردن شکل معادله قراری را می گذاریم که با آن قرار شکل معادله ساده شود. در این مثال حذف توابع کف و سقف کار ما را ساده می کند.

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ 2\cdot T(n/2)+2 & O.W\end{cases}$$

فرضی مطرح می کنیم که با آن فرض شکل معادله ساده شود.

برای حل از روش جایگذاری (Replacement) استفاده می کنیم.

$$\begin{aligned}T(n)=2\cdot T(n/2)+2\rightarrow \quad T(n)&=2^2\cdot T(n/4)+(2+2^2)\\ &=2^2\cdot\left(2\left(T(n/8)\right)+2\right)+(2+2^2)\\ &=2^3\cdot T(n/8)+(2+2^2+2^3)+\cdots\\ &=2^{k-1}\cdot T\left(\frac{n}{2^{k-1}}\right)+(2+2^2+2^3+\cdots+2^{k-1})\\ &=2^{k-1}\cdot T(2)+(2+2^2+2^3+\cdots+2^{k-1})\\ &=\frac{n}{2}+(2+2^2+2^3+\cdots+2^{k-1})\\ &=\frac{n}{2}+2\sum_{i=0}^{k-2}2^i=\frac{n}{2}+2\left(\frac{2^{k-1}-1}{2-1}\right)\\ &=\frac{n}{2}+2\left(\frac{n}{2}-1\right)=\frac{n}{2}+n-2=\frac{3n}{2}-2\end{aligned}$$

به این ترتیب می توانیم میان الگوریتم های فوق با دقت مناسب ، الگوریتمی با زمان مصرفی کمتر انتخاب کنیم.

مجددا می خواهیم عملیات فوق را تکرار کنیم با یک فرض جدید . و آن اینکه عناصر آرایه هم integer هستند.بنابراین ابتدا باید T(n) را مجددا بنویسیم و سپس حل مشابه فوق را تکرار کنیم.

در چنین حالتی ، مقایسه عناصر مانند مقایسه i و j هاست ، که در تعریف T(n) باید آنرا در نظر بگیریم.

$$T(n)=\begin{cases}1 & n=1\\ 3 & n=2\\ T\left(\lfloor n/2\rfloor\right)+T\left(\lceil n/2\rceil\right)+4 & OW\end{cases}$$

مثال: یک داستان قدیمی ویتنامی عمر دنیا را بر اساس یک اسطوره ، به شرط زیر تعریف می نماید.
دو میله الماس نشان وجود دارد ، روی میله اول 64 مهره وجود دارد به این صورت که مهره کوچک در بالا و مهره بزرگ در پایین. به کمک میله کمکی سوم مهره ها را از میله اول به دوم منتقل میکنیم ، به طوری که هیچ گاه حلقه بزرگتر روی کوچکتر قرار نگیرد.
بر اساس این داستان عمر دنیا 500000 میلیون سال می باشد.
برنامه ای بنویسید که این کار را انجام دهد.

```
Procedure  Hanoi(m,I,j)
      If  m > 0
            Then  [Hanoi (m-1,I,6-i-j)
                   Write (i→j)
                   Hanoi(m-1,6-i-j,j)
End
```

این الگوریتم m مهره را از ستون i به j منتقل می کند. در اولین بازخوانی این الگوریتم را با Hanoi(64,1,2) صدا کرده ایم. برای سادگی کار ، m را 3 در نظر بگیرید ، گردش کار الگوریتم را بشرح زیر دنبال می کنیم.

$$H(3,1,2)\nearrow\begin{vmatrix}H\\ m=3\\ i=1\\ j=2\\ H(2,1,3)\nearrow\end{vmatrix}\begin{vmatrix}H\\ m=2\\ i=1\\ j=3\\ H(1,1,2)\nearrow\end{vmatrix}\begin{vmatrix}H\\ m=1\\ i=1\\ j=2\\ H(0,1,3)\nearrow\\ 1\to 2\\ H(0,3,2)\\ \vdots\end{vmatrix}\begin{vmatrix}H\\ m=0\\ i=1\\ j=3\\ \hookleftarrow\end{vmatrix}$$

تمرین: مراحل کاری مثال فوق را با ذکر جزئیات کامل کنید. مطمئن شوید رفت و آمد میان page ها را یادگرفته اید.

نکته : از فرمول (6-i-j) برای پیدا کردن شماره ستون کمکی استفاده می کنیم.

نکته: یکی از مهمترین کارها در برنامه نویسی مدل سازی است. فرضا در این مثال برج ها را با چه نشان می دهیم . جابه جای مهره ها را چگونه انجام می دهیم ، ستون ها را چگونه تعیین می کنیم و بطور کلی منطق کاری الگوریتم چگونه است.

نکته: در این الگوریتم جابه جایی مهره ها توسط دستور write شبیه سازی شده است. هرگاه write اتفاق بیافتد ، یک مهره جا به جه شده است. توجه نمایید در هر page یک write وجود دارد. این الگوریتم خوانایی الگوریتم های بازگشتی را به خوبی نشان می دهد.

تعیین زمان مصرفی:

ابتدا باید روش تعیین زمان را مشخص کنیم. از راه شماره گذاری دستورات به نتیجه نمی رسیم. بنابراین سراغ انتخاب Barometer و تحلیل الگوریتم می رویم. دستور Barometer دستوری است که اجرای آن نشان می دهد که بخشی از الگوریتم اجرا

شده است . در الگوریتم فوق ، در هر page یکبار دستور write اجرا می شود ، بنابراین می توان فرض نمود نماینـده اجـرای یـک page ، دستور write است. پس Baromete را همان دستور write در نظر می گیریم.
به شکل بازگشتی الگوریتم توجه کنید. تعداد page های که ساخته می شود به m یعنی تعداد مهره ها بستگی دارد. اگر m زیاد یا کم شـود تعداد page ها هم زیاد یا کم می شود. پس می تواند فرض شود T(m) زمان مصرفی الگوریتم برای حالتی است که m مهره داریم.
برای تعریف T(m) سراغ شکل الگوریتم میرویم . چون الگوریتم بازگشتی است پس T(m) هـم بازگـشتی مـی شـود. بـرایش مـی تـوانیم تعریف زیر را از روی الگوریتم نتیجه بگیریم.

$$T(m)=\begin{cases}2\cdot T(m-1)+1 & m>1\\ 1 & m=1\end{cases}$$

زمان مصرفی الگوریتم بر اساس تابع فوق که یک Recurrence Function است ، بیان می شود.
این نوع توابع نشان می دهند دستور Barometer چندبار اتفاق افتاده است.

نکته : گاهی اوقات در تعریف توابع ، شرط اولیه را مشاهده نمی کنیم ، اما می دانیم اگر تابع بازگشتی مقدار اولیه نداشته باشد ، محاسباتش تمام نمی شود. بنابراین مقدار اولیه را بایستی به ترتیبی در تعریف تابع منظور کنیم . یک راه ساده آن است که الگوریتم را برای مقادیر اولیـه با دست دنبال کنیم و ببینیم آیا عدد مناسبی بدست می آید یا نه؟

حال می توانیم درستی تابع فوق را آزمایش نماییم . به m مقادیر مختلف می دهیم ، ببینیم تعداد جـا بـه جـای هـا بـا تعریـف مـسئله Hanoi سازگار است .
پس به نظر می رسد این تابع با تعریف مسئله برج Hanoi سازگار است.

حل یک Recurrence Function :

همانطور که در مثال های قبل دیدیم ، پس از تعریف یک رابطه بازگشتی باید جواب آن را بدست آوریم . جواب آن تابعی است که در Recurrence Function صدق می کند. این جواب معمولا به شکل یکی از توابـع سـاده ریاضـی بیـان مـی شـود. در مـسئله فـوق جواب به شکل $\left(T(m)=2^m-1\right)$ است. جواب یک Recurrence Function باید درآن صدق کند.
روش های مختلفی برای حل معادلات Recurrence Function وجود دارد. جلسه قبل روش جایگزینی را که یـک روش سـاده بـود بـه کار گرفتیم. اما این روش دامنه کاربرد محدودی دارد ، به همین دلیل باید سراغ روش های جایگزین دیگری برویم که بوسـیله آن روش هـا برای بسیاری از الگوریتم ها می توانیم معادلات مربوطه را حل نماییم.
به ازای هر الگوریتم می توانیم یک Recurrence Function بدست آوریـم . بنـابراین تنـوع Recurrence Function هـا بـه انـدازه تعداد الگوریتم ها است. در ادامه گروه خاصی از Recurrence Function را معرفی و حل می کنیم. البتـه ایـن نـوع از Recurrence Function ها پاسخ گوی بسیاری از الگوریتم های دوره کارشناسی و کارشناسی ارشد می باشد.

Recurrence Functions:

1. معادلات همگن Homogeneous Recurrence :

یک Recurrence Function همکن به شکل عمومی زیر است:

$$a_0\cdot t_n+a_1\cdot t_{n-1}+\cdots+a_k\cdot t_{n-k}=0 \qquad *$$

که در آن a_i ها ضرایب ثابت و t_i ها مجهولات مسئله هستند.

همانطور که مشاهده می شود در معادله فوق ، عوامل غیر خطی مانند $(t_i)^2, (t_i)^3, t_i \cdot t_i$ وجود ندارد. جمله ای هم نداریم که t_i نداشته باشد و عدد باشد. به همین خاطر به این معادلات همگن گوییم.

برای اینکه ببینیم با این معادلات چقدر آشنا هستیم ، به مسئله برج هانوی باز می گردیم. به جای T(m) ، t_m قرار می دهیم.

معادله آن الگوریتم به شکل زیر تبدیل می شود که یک معادله همگن نمی باشد.

$$t_m - 2 \cdot t_{m-1} = 1$$

نکته دیگری که باید به این نوع معدلات اضافه نمود، آن است که مجهولات آن بهم مرتبط است. یعنی مقدار t_m به مقدار t_{m-1} بستگی دارد و مقدار t_{m-1} به t_{m-2} و الی آخر .

یعنی درست است که مجهولات معادله $t_n, t_{n-1}, \cdots, t_{n-k}$ می باشد ، اما باید به ارتباط بین آنها توجه داشت . یعنی مجهولات از هم مستقل نیستند.

یک معادله همگن مانند معادله * جواب های مختلفی دارد. در اینجا یکی از جواب ها را بدست می آوریم. فرض کنید $t_n = x^n$ باشد ، یعنی اگر x را پیدا کنیم ، تمام مجهولات به دست می آید.

برای پیدا کردن x از یک ایده ساده پیروی می کنیم. چون t_n جواب معادله * است ، در معادله باید صدق کند.

$$a_0 \cdot x^n + a_1 \cdot x^{n-1} + \cdots + a_k \cdot x^{n-k} = 0$$

$$x^{n-k}\left(a_0 \cdot x^k + a_1 \cdot x^{k-1} + \cdots + a_k\right) = 0$$

اگر x صفر باشد ، تمام t_i ها صفر می شوند که در تحلیل الگوریتم ها مفهومی ندارد. بنابراین باید داشته باشیم :

$$a_0 \cdot x^k + a_1 \cdot x^{k-1} + \cdots + a_k = 0$$

به این معادله ، معادله مشخصه یا Charactristic Equation برای معادله * گویند. از حل این معادله ، مقادیر مجهول یعنی x بدست می آید. طبق قضیه اساسی جبر یک چند جمله ای درجه K ، K ریشه حقیقی یا موهومی دارد. فرض کنید این ریشه ها $r_1, r_2, \ldots, r_k$ باشد ، به این ترتیب t_n می تواند مقادیر r_k^n یا ... یا r_2^n یا r_1^n را داشته باشد.

حال به همگن بودن * توجه می کنیم. جون معادله * همگن است ، یعنی به ازای مقادیر r_1^n مقدارش صفر میشود ، به ازای r_2^n هم مقدارش صفر می شود و ... و به ازای r_k^n هم مقدارش صفر می شود ، بنابراین از ایده ای مشابه با حل معادلات دیفرانسیل استفاده می کنیم. ترکیب خطی جواب های یک معادله همگن ، جوابی برای معادله همگن است . به عبارت دیگر t_n بشکل زیر خواهد بود:

$$t_n = c_1 \cdot r_1^n + c_2 \cdot r_2^n + \cdots + c_k \cdot r_k^n$$

برای تعیین t_n ، باید مقادیر $r_1, r_2, \ldots, r_k$ را بدست آوریم و هم ضرایب $c_1, c_2, \ldots, c_k$.

مثال: معادله زیر را حل کنید.

$$t_n - 3t_{n-1} - 4t_{n-2} = 0 \qquad t_0 = 0, t_1 = 1 \qquad n \geq 2$$

همانطور که مشاهده می شود ، این معادله از نوع معادله * است ، که در آن K = 2 است. معادله مشخصه آن از درجه K خواهد بود و بشکل زیراست.

$$x^2 - 3x - 4 = 0 \rightarrow \begin{cases} r_1 = -1 \\ r_2 = 4 \end{cases}$$

$$\left.\begin{array}{l} n = 0 \rightarrow t_0 = c_1 \cdot 1 + c_2 \cdot 1 = 0 \\ n = 1 \rightarrow t_1 = -c_1 + 4c_2 = 1 \end{array}\right\} \rightarrow \begin{cases} c_1 = -1/5 \\ c_2 = 1/5 \end{cases}$$

$$(x-4)(x+1) = 0 \rightarrow t_n = \frac{1}{5}\left[4^n - (-1)^n\right]$$

مثال:

$$t_n = t_{n-1} + t_{n-2} \qquad n \geq 2$$
$$\begin{cases} t_0 = 0 \\ t_1 = 1 \end{cases}$$

حل:

$$t_n = t_{n-1} - t_{n-2} \Rightarrow t_n - t_{n-1} - t_{n-2} = 0$$
$$x^2 - x - 1 = 0 \rightarrow \begin{cases} r_1 = \dfrac{1+\sqrt{5}}{2} \\ r_2 = \dfrac{1-\sqrt{5}}{2} \end{cases}$$
$$t_n = c_1 \cdot r_1^n + c_2 \cdot r_2^n \Rightarrow \qquad \begin{cases} n = 0 \rightarrow t_0 = c_1 + c_2 = 0 \\ n = 1 \rightarrow t_1 = c_1 \cdot r_1 + c_2 \cdot r_2 = 1 \end{cases} \Rightarrow \begin{cases} c_1 = \dfrac{1}{\sqrt{5}} \\ c_2 = \dfrac{-1}{\sqrt{5}} \end{cases}$$
$$t_n = \frac{1}{\sqrt{5}}\left[r_1^n - r_2^n \right] \qquad \text{یک جمله عمومی دیگر فیبوناچی}$$

نکته : همانطور که مشاهده می شود ، پیدا کردن $r_1, r_2, \ldots, r_k$ بر اساس حل معادله مشخصه ، امکان پـذیر اسـت. بنـابراین ، شـکل معادلـه مشخصه درپیدا کردن این مقدار اهمیت دارد. به یکی از حالت های خاص شکل معادله توجه کنید.

فرض کنید معادله مشخصه یک ریشه مکرر از مرتبه m داشته باشد. اگر این ریـشه را r بنـامیم . بـه ایـن ترتیـب در معادلـه مشخـصه ضـریب $(x-r)^m$ بوجود می آید. این موضوع سبب می شود که بر اساس r برای t_n ، جواب های زیر بدست آید.

به هر حال توجه داشته باشید معادله مشخصه از مرتبه k است یعنی k ریشه دارد. برخی از آنها مکررند، برخی از آنها غیر مکـرر. ریـشه هـای غیر مکرر برای t_n مقادیر جدید فوق را بوجود می آورد.

$$t_n = r^n, t_n = n \cdot r^n, \ldots, t_n = n^{m-1} \cdot r^n$$

مثال : معادله رو به رو را حل کنید:

$$\begin{cases} t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0 \qquad n \geq 3 \\ t_0 = 0 \\ t_1 = 1 \\ t_2 = 2 \end{cases}$$

حل:

معادله مشخصه بشکل زیر می باشد:

$$x^3 - 5x^2 + 8x - 4 = 0 \qquad \Rightarrow (x-1)(x-2)^2 = 0 \rightarrow \begin{cases} r_1 = 1 \\ r_2 = 2 \end{cases}$$

$$t_n = r_1^n, r_2^n, n \cdot r_2^n$$
$$t_n = c_1 \cdot r_1^n + c_2 \cdot r_2^n + c_3 \cdot n \cdot r_2^n$$
$$\begin{cases} n = 0 \rightarrow t_0 = c_1 + c_2 = 0 \\ n = 1 \rightarrow t_1 = c_1 \cdot r_1 + c_2 \cdot r_2 + c_3 \cdot r_2 = 1 \\ n = 2 \rightarrow t_2 = c_1 \cdot r_1^2 + c_2 \cdot r_2^2 + c_3 \cdot 2r_2^2 = 2 \end{cases} \qquad \begin{cases} c_1 = -2 \\ c_2 = 2 \\ c_3 = -1/2 \end{cases}$$

2. معادلات غیر همگن Inomogeneous Recurrence :

یک معادله غیر همگن انواع و اقسام شکل ها را دارد ، در این جا به حل نوع خاصی از معادلات غیر همگن می پردازیم.

$$a_0 \cdot t_n + a_1 \cdot t_{n-1} + \cdots + a_k \cdot t_{n-k} = b^n \cdot p(n) \qquad **$$

همانطور که مشاهده می شود ، سمت چپ این معادله مانند معادله * است. مشابه معادله همگن مجهولات $t_n, t_{n-1}, \cdots, t_{n-k}$ می باشند. $a_0, a_1, \ldots, a_k$ مقادیر ثابت هستند. B یک مقدار ثابت است و p(n) یک چندجمله ای از درجه d .

نکته: تمام جملات در معادله * مجهول را داشتند ، تنها جمله ای که مجهول نداشت ، مقدار صفر را داشت. معادله ** به علت وجود $b^n \cdot p(n)$ یک معادله غیر همگن است.

ایده حل این نوع معادلات ، یک ایده بسیار ساده است. روش کار آن است که معادله ** را به معادله * تبدیل کنیم. جواب معادله همگن جواب مسئله خواهد بود. اما می ماند اینکه چطور این کار را انجام دهیم. یک معادله داریم که باید شکلش را عوض کنیم یعنی تبدیلش کنیم به معادله همگن. راهی نمی ماند غیر از اینکه این معادله را با خودش جمع ، تفریق و یا جایگزینی های لازم را انجام دهیم تا بشکل همگن تبدیل شود.

ابتدا با دو مثال این روش را تمرین می کنیم ، سپس یک فرمول کلی ارائه می دهیم.

مثال : معادله روبرو را حل کنید.

$$t_n - 2t_{n-1} = 3^n$$

حل:

$$b = 3, p(n) = 1, d = 0$$

با یکسری عملیات ریاضی ساده سمت راست معادله فوق را صفر می کنیم ، توجه نمایید سمت راست ، مخالف صفر است.پس سراغ نکاتی می رویم ، مانند اینکه این رابطه به ازای هر n برقرار است ، بنابراین جای n می توانیم (n+1) بگذاریم.

1) $\quad t_{n+1} - 2t_n = 3^{n+1}$

حال معادله اصلی را در 3 ضرب می کنیم.

2) $\quad 3t_n - 6t_{n-1} = 3^{n+1}$

معادلات 1 و 2 را از هم کم می کنیم.

$$\begin{array}{l} t_{n+1} - 2t_n = 3^{n+1} \\ \underline{-(3t_n - 6t_{n-1} = 3^{n+1})} \\ t_{n+1} - 5t_n + 6t_{n-1} = 0 \end{array}$$

اگر بخواهیم این معادله را با معادله * مقایسه کنیم به این نکته باید توجه کرد. جمله اول این معادله t_n بود نه t_{n-1} . بنابراین در اینجا به نظر می رسد که k = 2 است.

Charactristic Equation : $\quad x^2 - 5x + 6 = 0$

$$(x-2)(x-3) = 0 \rightarrow \begin{cases} r_1 = 2 \\ r_2 = 3 \end{cases}$$

با پیدا شدن r_1 و r_2 مشابه معادله همگن معادله را حل می کنیم ، یعنی اکر دو مقدار اولیه صورت مسئله به ما داده بود ، می توانستیم c_1 و c_2 را بدست آوریم. نکات دیگری هم قابل طرح است. همواره در پیدا کردن معادله مشخصه دو بخش را می توانیم مشاهده کنیم. یک بخش متناظر به سمت چپ معادله اولیه مسئله است و بخش دیگر متناظر به تغییرات است. در مثال فوق این موارد به خوبی مشاهده می شود.

مثال: معادله فوق را به یک معادله همگن تبدیل نمایید و آنرا حل کنید.

$$t_n - 2t_{n-1} = (n+5)3^n$$

حل:

به جای n ، n+1 می گذاریم.

$$t_{n+1} - 2t_n = (n+6)3^{n+1}$$

معادله اصلی را در 3 ضرب می کنیم.

$$3t_n - 6t_{n-1} = (n+5)3^{n+1}$$

پس به نظر می رسد با توان n+1 نمی توانیم سمت راست را صفر کنیم. پس سعی می کنیم با توان n+2 سمت راست را صفر کنیم.

معادله اصلی را در 9 ضرب می کینم.

$$1) \qquad 9t_n - 18t_{n-1} = (n+5)3^{n+2}$$

در همان معادله اصلی جای n ، n+2 می گذاریم.

$$2) \qquad t_{n+2} - 2t_{n+1} = (n+7)3^{n+2}$$

این بار در معادله اصلی جای n ، n+1 قرار می دهیم و در 6- ضرب می کنیم.

$$2) \qquad -6t_{n+1} + 12t_n = -2(n+6)3^{n+2}$$

معادلات 1 و 2 و 3 را با هم جمع می کنیم. که حاصل معادله زیر می شود.

$$t_{n+2} - 8t_{n+1} + 21t_n - 18t_{n-1} = 0$$

با این کار یک معادله همگن ساخته شد که معادله مشخصه آن به این صورت است.

$$x^3 - 8x^2 + 21x - 18 = 0$$

$$(x-2)(x-3)^2 = 0$$

همانطور که مشاهده می شود ، این معادله مشخصه دو بخش دارد. یک بخش آن متناظر به معادله همگن و بخش دیگر آن ناشی از تغییرات است.

به طور کلی برای معادله ** معادله مشخصه به شکل زیر خواهد بود:

$$(a_0 \cdot x^k + a_1 \cdot x^{k-1} + \cdots + a_k)(x-b)^{d+1} = 0$$

مثال:

$$\begin{cases} t_n - 2t_{n-1} = 1 \\ n \geq 1 \\ t_0 = 0 \end{cases}$$

حل:

$$\begin{cases} b=1 \\ p(n)=1 \\ d=0 \end{cases} \rightarrow (x-2)(x-1)=0 \rightarrow \begin{cases} r_2=2 \\ r_1=1 \end{cases}$$

$$\rightarrow t_n = c_1 \cdot 1^n + c_2 \cdot 2^n$$

$$\left.\begin{array}{l} n=0 \rightarrow t_0 = c_1 + c_2 = 0 \\ n=1 \rightarrow t_1 = c_1 + 2c_2 = 1 \end{array}\right\} \rightarrow c_1 = -1, c_2 = 1$$

روش دوم : فرض کنید ، تعداد مقادیر اولیه از تعداد ضرایب c_i کمتر باشد. یک راه مانند فوق است که معلوم نیست همیشه قابل استفاده باشد . راه دیگر آن است که جواب t_n را در معادله قرار دهیم و سعی کنیم ضرایب c_1 و c_2 را بدست آوریم.

$$t_n - 2t_{n-1} = 1 \rightarrow \begin{cases} t_n = c_1 + 2^n c_2 \\ t_n = 2t_{n-1} + 1 \end{cases}$$

$$t_1 = 2t_0 + 1 = 1 \rightarrow c_1 + 2^n c_2 = 2\left(c_1 + c_2 \cdot 2^{n-1}\right) + 1$$

$$c_1 + 2^n c_2 = 2c_1 + 2^n c_2 + 1 \Rightarrow c_1 = -1$$

پس c_2 هر مقداری می تواند باشد.

مثال:

$$t_n - 2t_{n-1} = n$$

حل:

$$\begin{cases} p(n)=1 \\ d=1 \\ b=1 \end{cases} \rightarrow (x-2)(x-1)^2 = 0 \rightarrow t_n = c_1 \cdot 2^n + c_2 \cdot 1^n + c_3 \cdot n \cdot 1^n$$

در این صورت مسئله t_0 داده نشده است. پس اگر بخواهیم c_1, c_2, c_3 را بدست آوریم ، چاره ای نداریم غیر از آنکه t_n را در معادله قرار دهیم و برای c_1, c_2, c_3 ، یا مقداربدست آوریم و یا شرط پیدا کنیم. اگر به این روش مسئله را حل کنیم ، $c_1 > 0, c_2 = -2, c_3 = -1$ خواهد شد.

به طور کلی یک معادله همگن را بشکل زیر در نظر می گیریم:

$$a_0 \cdot t_n + a_1 \cdot t_{n-1} + \cdots + a_k \cdot t_{n-k} = b_1^n \cdot p_1(n) + b_2^n \cdot p_2(n) + \cdots \qquad ***$$

که در آن $p_1(n)$ از درجه d_1 و $p_2(n)$ از درجه d_2 و ...

در این حالت معادله مشخصه بشکل زیر خواهد بود:

$$(a_0 \cdot x^k + a_1 \cdot x^{k-1} + \cdots + a_k)(x-b_1)^{d_1+1}(x-b_2)^{d_2+1} \cdots = 0$$

مثال:

$$\begin{cases} t_n - 2t_{n-1} = n + 2^n \\ t_0 = 0 \end{cases}$$

حل:

$$\begin{cases} b_1 = 1 \\ p_1(n) = n \\ d_1 = 1 \end{cases} \qquad \begin{cases} b_2 = 2 \\ p_2(n) = 1 \\ d_2 = 0 \end{cases}$$

$$(x-2)(x-1)^2(x-2) = 0 \rightarrow (x-2)^2(x-1)^2 = 0$$
$$\Rightarrow t_n = c_1 \cdot 1^n + c_2 \cdot n \cdot 1^n + c_3 \cdot 2^n + c_4 \cdot n \cdot 2^n$$

$$\begin{cases} t_1 = 2t_0 + 1 + 2 = 3 \\ t_2 = 2t_1 + 2 + 2^2 = 12 \\ t_3 = 2t_2 + 3 + 2^3 = 35 \end{cases} \Rightarrow \begin{cases} n = 0 \rightarrow t_0 \\ n = 1 \rightarrow t_1 \\ n = 2 \rightarrow t_2 \\ n = 3 \rightarrow t_3 \end{cases} \Rightarrow \begin{cases} c_1 = -2 \\ c_2 = -1 \\ c_3 = 2 \\ c_4 = 1 \end{cases}$$

تغییر متغییر:

در حد معلومات دوره کارشناسی ، دو روش فوق معمولا روش های اصلی حل معادلات است. در مورد برخی از معادلات این روش ها را نمی توانیم بکار ببریم. اما گاهی اوقات با یک تغییربه شکل ساده که همان تغییر متغییر ریاضی است ، شکل معادله را به شکل همگن یا غیر همگن فوق الذکر در می آوریم. در چنین مواردی ایده تغییر متغییر ، به کار برده می شود. که این ایده محدود به مثال های گفته شده در چند صفحه بعد حتما نمی باشد.

مثال:

$$T(n) = 4T(n/2) + n \qquad n > 1$$

حل:

$$if \quad n = 2^k \rightarrow t_k \equiv T(n) \equiv T(2^k) \\ \rightarrow T(n/2) \equiv T(2^{k-1}) \equiv t_{k-1} \Big\} \rightarrow t_k = 4t_{k-1} + 2^k \rightarrow t_k - 4t_{k-1} = 2^k$$

$$\begin{cases} p_1(k) = 1 \\ d = 0 \\ b = 2 \end{cases} \rightarrow (x-4)(x-2) = 0 \rightarrow t_k = c_1 \cdot 4^k + c_2 \cdot 2^k$$

باید دقت نمود هر تغییر متغییری باید با تغییر متغییر وارون همراه باشد . چون بالاخره جواب t_n را می خواهیم. در این مثال ، تغییر متغییر وارون را به سادگی می توانیم بکار بگیریم . کافی است در رابطه قبل همه چیز را بر حسب n بنویسیم.

$$T(n) = c_1 \cdot n^2 + c_2 \cdot n$$
$$T(n) \in O(n^2 \,|\, n = 2^k)$$

مثال:

$$\mathrm{T(n)} = 4\mathrm{T}(\mathrm{n}/2) + n^2$$

حل:

$$if \quad n = 2^k \rightarrow T(n) \equiv T(2^k) \equiv t_k \\ \rightarrow T(n/2) \equiv T(2^{k-1}) \equiv t_{k-1} \Big\} \rightarrow t_k - 4t_{k-1} = 4^k$$

$$\begin{cases} P_1(k) = 1 \\ d^1 = 0 \\ b_1 = 4 \end{cases} \rightarrow (x-4)(x-4) = 0 \rightarrow (x-4)^2 = 0 \Rightarrow t_k = c_1 \cdot 4^k + c_2 \cdot k \cdot 4^k$$

باز هم در این مثال تغییر متغییر وارون را می توانیم به سادگی انجام دهیم.

$$T(n) = c_1 \cdot n^2 + c_2 \cdot \log_2^n \cdot n^2$$
$$T(n) \in O(n^2 \log n \,|\, n = 2^k)$$

مثال: معادله روبرو را حل کنید.

$$\begin{cases} T(n) = n \cdot T^2(n/2) & n > 1 \\ T(1) = 6 \end{cases}$$

حل:

ظاهرا این معادله غیر خطی است. بنابراین به نظر می رسد با تغییراتی مشابه تغییرات فوق قابل حل نباشد. اما معمولا در تغییر متغییر از ساده ترین تغییرات شروع می کنیم.

$$\left.\begin{array}{l} n = 2^k \rightarrow T(n) \equiv T(2^k) \equiv t_k \\ \quad\rightarrow T(n/2) \equiv T(2^{k-1}) \equiv t_{k-1} \\ \quad\rightarrow T(1) \equiv T(2^0) \equiv t_0 \end{array}\right\} \rightarrow \begin{cases} t_k = 2^k \cdot t_{k-1}^2 \\ t_0 = 6 \end{cases}$$

در برخی موارد تغییرات معادله به شکل ساده امکان پذیر نیست ممکن است لازم شود که به طرفین معادله تابعی را اعمال کنیم . یکی از این توابع $\log_2$ است. به 2^k هم نزدیک است بعلاوه لگاریتم حاصلضرب به مجموع لگاریتم ها تبدیل می شود.

حال از طرفین رابطه فوق لگاریتم می گیریم ، نتیجه می شود:

$$\log t_k = \log 2^k + \log t_{k-1}^2$$
$$= k + 2 \log t_{k-1}$$
$$\log t_0 = \log 6 = \log 2 + \log 3 = 1 + \log 3$$

حال یک تغییر متغییر جدید بکار می بریم.

$$v_k = \log t_k \Rightarrow \begin{cases} v_k = k + 2v_{k-1} & \Rightarrow v_k - 2v_{k-1} = k \\ v_0 = \log t_0 = 1 + \log 3 \end{cases}$$

$$\begin{cases} b_1 = 1 \\ p_1(k) = k \Rightarrow (x-2)(x-1)^2 = 0 \\ d_1 = 1 \end{cases} \rightarrow v_k = c_1 \cdot 2^k + c_2 \cdot 1^k + c_3 \cdot k \cdot 1^k$$

$$\begin{cases} v_1 = 1 + 2v_0 = 1 + 2 + 2\log 3 = 3 + 2\log 3 \\ v_2 = 2 + 2v_1 = 8 + 4\log 3 \end{cases}$$

$$\begin{cases} k = 0 \rightarrow v_0 = c_1 + c_2 = 1 + \log 3 \\ k = 1 \rightarrow v_1 = 2c_1 + c_2 + c_3 = 3 + 2\log 3 \\ k = 2 \rightarrow v_2 = 4c_1 + c_2 + 2c_3 = 8 + 4\log 3 \end{cases} \rightarrow \begin{cases} c_1 = 3 + \log 3 \\ c_2 = -2 \\ c_3 = -1 \end{cases}$$

$$v_k = (3 + \log 3)2^k - k - 2$$

توجه کنید که باید t_n را پیدا کنیم.بنابراین باید با جایگذاری وارون به t_n برسیم.

$$v_k = \log t_k \Rightarrow t_k = 2^{v_k} \Rightarrow \begin{cases} t_k = 2^{(3+\log 3)2^k - k - 2} \\ n = 2^k \end{cases} \Rightarrow t_n = 2^{(3+\log 3)n - \log_2^n - 2} = 2^{(3+\log 3)} \cdot 2^{-\log_2^n} \cdot 2^{-2}$$

$$= 2^{(3+\log 3)} \cdot \frac{1}{2^{\log_2^n}} \cdot \frac{1}{4} = 2^{(3+\log 3)} \cdot \frac{1}{n} \cdot \frac{1}{4}$$

$$\Rightarrow t_n = 2^{(3+\log 3)} \cdot \frac{1}{4n}$$

به این ترتیب در عملیات وارون باید بالاخره $T(n)$ را پیدا کنیم که ممکن است به دقت بیشتری احتیاج باشد.

تمرین: مسئله 2.3.13 کتاب را بررسی کنید . حالات مختلف آنرا بررسی کنید. نتیجه بگیرید تمام مثال های که مطرح کردیم حالت خاصی از آن است.

مثال: معادله روبرو را حل کنید.

$$\begin{cases} t_n = \dfrac{1}{4 - t_{n-1}} \\ t_1 = 1/4 \end{cases}$$

حل:

$$T(n) = \overbrace{\frac{a_n}{b_n}}^{(A)} = \frac{1}{4 - \dfrac{a_{n-1}}{b_{n-1}}} = \frac{b_{n-1}}{4b_{n-1} - a_{n-1}}$$

از تساوی فوق نتیجه می شود صورت های دو کسر و مخرج ها با هم برابرند

$$\begin{cases} a_n = b_{n-1} \\ b_n = 4b_{n-1} - a_{n-1} \end{cases} \quad (B) \quad \rightarrow b_n = 4b_{n-1} - b_{n-2} \rightarrow b_n - 4b_{n-1} + b_{n-2} = 0$$

$$\rightarrow x^2 - 4x + 1 = 0 \rightarrow \begin{cases} r_1 = 2 + \sqrt{3} \\ r_2 = 2 - \sqrt{3} \end{cases} \Rightarrow b_n = c_1(2+\sqrt{3})^n + c_2(2-3)^n \qquad (1)$$

بحث را در جهتی دنبال می کنیم که c_1, c_2 را پیدا کنیم. سراغ مقادیر اولیه می رویم. طبق رابطه (A) :

$$T(1) = t_1 = \frac{1}{4} = \frac{a_1}{b_1} \rightarrow \begin{cases} a_1 = 1 \\ b_1 = 4 \end{cases} \quad (C)$$

چون مقادیر با اندیس منفی را تعریف نکرده ایم بنابراین باید برای b_0 مقداری انتخاب کنیم تا محاسبات به جواب برسد. بنابراین اگر لازم باشد برای a_0 هم باید شرطی در نظر بگیریم.

طبق رابطه (1):
$$\left.\begin{array}{ll} n = 0 & b_0 = c_1 \cdot 1 + c_2 \cdot 1 = 1 \\ n = 1 & b_1 = c_1(2+\sqrt{3}) + c_2(2-\sqrt{3}) = 4 \end{array}\right\} \rightarrow c_1 = \frac{2+\sqrt{3}}{2\sqrt{3}}, c_2 = 1 - c_1$$

به این ترتیب به نظر می رسد که به مقدار a_0 نیاز داریم. سراغ صورت مسئله می رویم ببینیم شرطی به ما می دهد یا نه؟ اگر شرطی به ما ندهد باید مقدار a_0 را انتخاب کنیم.

$$(C) \qquad \rightarrow b_1 = 4b_0 - a_0 = 4a_0 \rightarrow a_0 = 0$$

پس با استفاده از رابطه (1) ، b_n را بدست می آوریم و با استفاده از رابطه (B) مقدار a_n را بدست می آوریم و به این ترتیب T(n) بدست می آید.

الگوریتم های حریصانه Greedy Algorithm :

در این فصل با الگوریتم های سروکار داریم که در آنها معمولا مقدار کمیت باید max یا min شود. فرضا بگوییم سودمان حداکثر شود و یا ضررمان حداقل شود.

عنوان Greedy از همین مفهوم می آید . می توانیم بگوییم هدفی را می خواهیم max کنیم و یا هدفی را می خواهیم min کنیم. برای رسیدن به این نوع هدف ها ، محدودیت های وجود دارد که آن محدودیت ها در هر مسئله شکل های مختلفی دراد.

این محدودیت ها طبیعت نامساوی دارند . یا بشکل کوچکتر تعریف می شوند یا بشکل بزرگتر.الگوریتم کاری این روش بشکل زیر است . آنرا با یک مثال توضیح می دهیم.

```
Function Greedy (C:set)  :set
    S ←∅
    while  not solution(S)  and  C ≠ ∅ do
        X ← an element of C maximizing select(X)
        C ← C \ {x}
        If  feasible(S ∪{X })  then  S ← S ∪{X }
    If  solution(S)  then return S
                Else return " there are no solutions "
```

درخت پوشای کمین Minimum Spanning Tree :

زیر گرافی از یک گراف را در نظر بگیرید. این زیر گراف می تواند هر تعدادی ضلع یا راس داشته باشد. می تواند طبیعتی درختی داشته باشد یعنی دور نداشته باشد. اگر n راس دارد n-1 ضلع داشته باشد. زمانی این درخت ها را پوشا گویند که شامل تمام رئوس گراف باشند.

درخت پوشای کمین درختی است که وزن آن min باشد ، درخت پوشای کمین الزاما یکتا نیست (برای یک گراف). دو الگوریتم کلاسیک برای پیدا کردن درخت پوشای کمین وجود دارد.

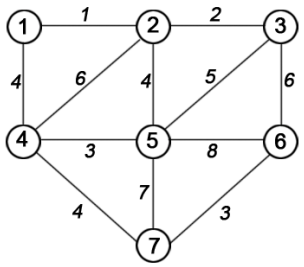
الگوریتم Kruskal :

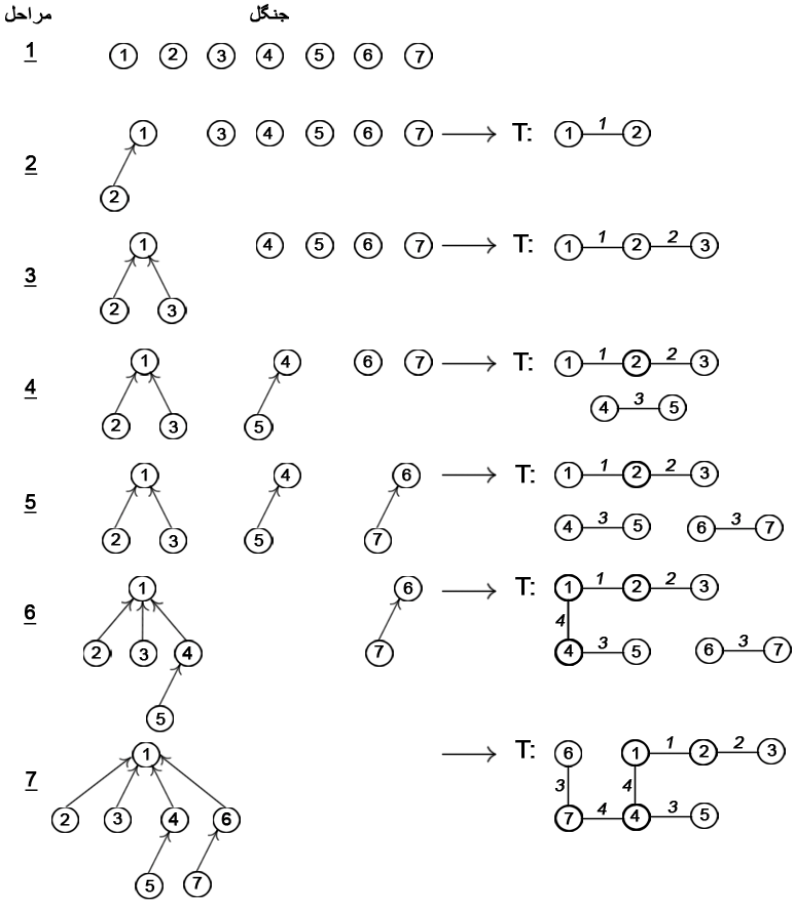
```
Function Kruskal (G =< N ,A >: graph; length : A → ℝ* ) : set of edges
    {initialization}
    Sort A by increasing length
    n ← #N
    T ← ∅ { will contain the edges of the minimum spanning tree
    Initialize n sets, each containing one distinct element of N }
    {greedy loop}
    Repeat
        {u,v} ← shotest edge not yet considered
        ucomp ← find(u)
        vcomp ← find(v)
        If  ucomp ≠ vcomp  then
            Merge(ucomp,vcomp)
            T ← T ∪{{u,v }}
    Until  #T = n-1
    Return T
```

در الگوریتم فوق T خروجی الگوریتم است. جنگل ابزاری است برای ساختن T ، بنابراین با T اشتباه نشود. توجه داریم که T یک درخت است ، پس دور ندارد ، اگر n راس داشته باشد ، n-1 ضلع دارد و همبند است.

نکته : الگوریتم find ورودی خود را می گیرد و از درخت بالا می رود تا به ریشه برسد.

مثال:





مراحل	u	v	ucomp	Vcomp
1	1	2	1	2
2	2	3	1	3
3	4	5	4	5
4	6	7	6	7
5	1	4	1	4
6	2	5	1	1
7	4	7	1	6

تحلیل زمان مصرفی الگوریتم kruskal :

زمان مصرفی sort(A) برابر $O(a \cdot \log a)$ است. همچنین حلقه repeat-until ، n-1 بار می چرخد . پس یک O(n) برای این گردش داریم. برای انتخاب ضلعی با کمترین هزینه نیز زمانی داریم که اگر درخت minheap برای وزن ها داشه باشیم ، زمان این کار log a است. زمان مصرفی برای یافتن ریشه نیز O(log N) است.

زمان مصرفی برای داخل حلقه : $O(n \log a) \leftarrow (n-1)(\log a + \log a)$

پس یک O(a log a) داریم برای sort(A) و یک O(n log a) نیز برای داخل حلقه. حال برای کل این الگوریتم این دو را با یکدیگر جمع می کنیم. (ماکزیمم را پیدا می کنیم) . در اینجا اگر n بزرگتر از a باشد برای کل الگوریتم n log a و اگر n کوچکتر از a باشد a log a برای کل الگوریتم است.

الگوریتم prim :

Function prim ($G =< N, A >: graph : length : A \rightarrow \mathbb{R}^*$):set of edges

{initialization}

T $\leftarrow \varnothing$ { will contain the edges of the minimum spanning tree }

While $B \neq N$ **do**

Find {u,v} of minimum length such that $u \in N \setminus B$ and $v \in B$

T $\leftarrow T \cup \{\{u,v\}\}$

B $\leftarrow B \cup \{u\}$

Return T

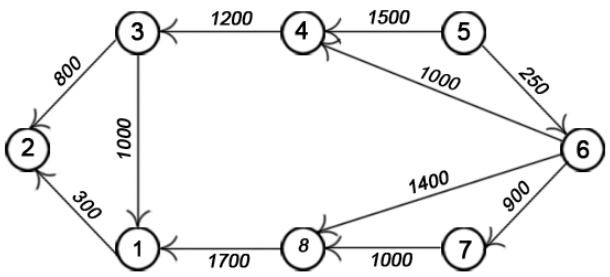
نکته : در این الگوریتم در هر مرحله یک درخت داریم. در حالی که در الگوریتم kruskal در هر مرحله جنگل داشتیم که در آخر تبدیل به درخت می شد.

زمان مصرفی الگوریتم prim :

حلقه while به تعداد رئوس گراف (N) بار می چرخد. و تنها یک log a یا log n برای انتخاب ضلع (u,v) با کمترین هزینه داریم . بنابراین زمان مصرفی این الگوریتم در گروه n log a یا n log n قرار می گیرد.

کوتاه ترین مسیر Shortest Path :

الگوریتمی طراحی کنید که کوتاه ترین فاصله بین یک راس مشخص و سایر رئوس گراف را پیدا کند.



مثال: به جای گراف فوق ، گراف درس های رشته کامپیوتر را در نظر بگیرید . می خواهیم کوتاه ترین فاصله بین درس مبانی کامپیوتر و سایر دروس را پیدا کنیم.

نمایش گراف ها:

بطور معمول به دو شکل کلاسیک می توانیم گراف را نشان دهیم :

1. استفاده از ماتریس
2. استفاده از لیست همجواری

در ماتریس بجای اعداد (0و1) وزن اضلاع را می نویسیم . ضلعی که وجود نداشته باشد ، با وزن خیلی زیاد نشان داده می شود ، این وزن آنقدر زیاد است که در انتخابها هیچ موقع انتخاب نمی شود. معمولا این وزن را با علامت کلی ∞ نشان می دهند. عناصر قطری ماتریس را با صفر نشان می دهیم. پیش فرضمان آن است که وزن منفی نداریم . اما این موضوع کاربرد دیگری هم دارد. ماتریس فوق را می توانیم یک ماتریس اسپارس در نظر بگیریم.

```
Procedure shortest-path ( v,cost,dist,n )
    For  i ← 1  to  n  do
        S(i) ← 0 , Dist(i) ← Cost(v,i)
    End
    S(v) ← 1 , Dist(v)=0
    For  num ← 2  to  n-1  do
        Choose  u  such that
            Dist(u) = min { Dist (w) }
                S(w)=0
                S(u) ← 1
        For  all  w  with S(w)=0 do
            Dist(w) ← min { Dist (w),Dist(u)+Cost (u,w) }
        End
    End
End
```

این الگوریتم کوتاه ترین فاصله بین راس دلخواه v و سایر رئوس گراف را تعیین می کند. Cost ماتریس مربوط به گراف است و Dist آرایه است که خروجی الگوریتم را نشان می دهد. Dist(i) کوتاه ترین فاصله بین راس v و راس i را نشان می دهد. آرایه Dist ، n عضو دارد که n تعداد رئوس گراف است. آرایه S یه عنوان یک آرایه از بیت ها به کار می رود که در فرآیند الگوریتم کاربرد دارد . v یک راس دلخواه از گراف است یعنی می تواند هر راسی باشد. برای مثال فوق v را 5 انتخاب می کنیم تا گردش کاری الگوریتم را بهتر ببینیم.

	S
1	0
2	0
3	0
4	0 1
5	0 1
6	0 1
7	0 1
8	0 1

	1	2	3	4	5	6	7	8
Dist	∞	∞	∞	1500	0	250	∞	∞
	∞	∞	∞	1250	0	250	1150	1650
	∞	∞	∞	1250	0	250	1150	1650
	∞	∞	2450	1250	0	250	1150	1650
	3350	∞	2450	1250	0	250	1150	1650

آرایه S ، یک آرایه از بیت هاست ، از این آرایه برای علامت گذاری استفاده می کنیم. هر راس را که ببینیم بیتش را یک می کنیم تا دوباره آن را نبینیم.

انتخاب v برابر 5 به این معنی است که از 5 راه می افتیم ، سراغ ارزانترین راس می رویم که همان راس 6 است . حال می توانیم ، برا اساس انتخاب u ، تمام مسیرهای ممکن را به روز درآوریم.

Dist(1): min { Dist(1),Dist(6)+Cost(6,1) }= ∞
Dist(2): min { Dist(2),Dist(6)+Cost(6,2) }= ∞
Dist(4): min { Dist(4),Dist(6)+Cost(6,4) }= 1250
Dist(7): min { Dist(7),Dist(6)+Cost(6,7) }= 1150
Dist(8): min { Dist(8),Dist(6)+Cost(6,8) }= 1650

در این الگوریتم ، ایده اصلی آن است که در هر مرحله ، ارزانترین ضلع انتخاب شود. مسیری که از این اضلاع ساخته می شود یک مسیر مینیمم نسبی خواهد بود.

وقتی می توانیم کار را تمام کنیم که در مسیر، تمام رئوس را لحاظ کرده باشیم . یعنی تمام رئوس در آرایه S ، یک شده باشند. استفاده از آرایه Dist ، این خاصیت را دارد که همواره از مقادیر بعدی ، در به روز درآوردن آرایه Dist استفاده می کنیم.

تحلیل زمان مصرفی:

زمان مصرفی حلقه اول برابر $O(n)$ است. داخل حلقه دوم یک زمان برای انتخاب u و یک زمان نیز برای حلقه بعدی داریم . حلقه دومی $O(n^2)$ است. زمان مصرفی این الگوریتم در گروه $O(n^2)$ است.

مثال های کلاسیک برای روش گریدی:

- نقشه ایران را در نظر بگیرید . می خواهیم با کمترین تعداد رنگ نسبتا اصلی نقشه را رنگ آمیزی کنیم بطوریکه استان های مجاور رنگ های متفاوت داشته باشند.
- مسئله فروشنده دوره گرد. منطقه ای را در نظر بگیرید ، فروشنده ای می خواهد با طی کردن جاده ها ، تمام شهر های منطقه را ببیند ، بطوریکه در کل حداقل مسافت را پیموده باشد. یعنی از هر ضلع گراف فقط و فقط یکبار عبور کند.

الگوریتم Quicksort :

```
Procedure  Quicksort ( p,q )
        If   p < q
                Then [ j ← q+1
                        Partition( p,j )
                        Quicksort ( p,j-1 )
                        Quicksort ( j+1,q )]
End
Procedure  partition ( m,p )
        v ← A(m)
        i ← m
        Loop
                Loop
                        i ← i +1
                Until  A(i) ≥ v
                Loop
                        P ← p-1
                Until  A(p) ≤ v
                If  i < p
                        Then  interchange ( A(i),A(p) )
                Else  exit
        Forever
        A(m) ← A(p)
        A(p) ← v
End
```

الگوریتم Quick sort ، آرایه A(p:q) را بهطور صعودی مرتب می کند. اگر بخواهیم الگوریتم Quick sort را به راه بیاندازیم ، در یک برنامه اصلی آن را صدا می کنیم. جای p یک می دهیم . جای q ، n قرار می دهیم. به این ترتیب Quick sort را راه انداخته ایم که آرایه n عنصری را مرتب می کند. فرض کرده ایم ، آرایه A ، گلوبال این الگوریتم است. بخش اصلی این الگوریتم ، الگوریتم partition است. که کار آن را بشکل زیر می توان بیان نمود.

در آرایه A ، عنصر اول را به عنوان عنصر افراز کننده در نظر می گیرد. از دو اندیس i,p استفاده می کنیم. اندیس i ازابتدای آرایه به سمت انتهای آرایه حرکت می کند و اندیس p از انتهای آرایه به سمت ابتدای آن.

با این دو اندیس عناصر آرایه را با v یعنی عنصر افراز کننده مقایسه می کنیم. آنهایی که از عنصر افراز کننده بزرگتر هستند به انتهای آرایه منتقل می کنیم و آنهایی که از عنصر افراز کننده کوچکتر هستند به ابتدای آرایه . در نهایت عنصر افراز کننده را با یک مقایسه به محل جدیدش منتقل می کنیم. محل جدید این خاصیت را دارد که تمام عناصر قبل از آن در آرایه از عنصر افراز کننده کوچکتر و تمام عناصر بعد از آن از عنصر افراز کننده بزرگتر هستند. به عبارت دیگر آرایه سورت نمی شود بلکه عنصر افراز کننده در محل خودش قرار می گیرد. یعنی partition یک عنصر را برداشت و در محل خودش در آرایه مرتب قرار داد. حال اگر Quick sort به دفعات لازم خودش را صدا زند ، آرایه A مرتب خواهد شد.

مثال:

	1	2	3	4	5	6	7	8	9	10
A	65	70	60	80	45	90	50	75	55	∞

در ابتدای الگوریتم Quick sort در خانه $n+1$ ام از آرایه A ، یک عدد بزرگ قرار می دهیم . این عدد از تمام عناصر آرایه بزرگتر است. در مثال فوق ، به طور مثال 91 پاسخ گوی نیاز ما است. اما چون مقدارش را نمی دانیم از نماد ∞ استفاده می کنیم.

1	2	3	4	5	6	7	8	9	10
45	55	60	50	65	90	80	75	70	∞

m	**p**	**v**	**i**
1	10	65	**1**
	9		**2**
	8		**3**
	7		**4**
	6		**5**

تحلیل زمان مصرفی:

در این الگوریتم partition بخش اصلی است. بنابراین بارومتر در داخل partition انتخاب می شود . پس سعی می کنیم برای partition زمان مصرفی پیدا کنیم. دستور بارومتر را باید انتخاب کنیم.

مقایسه بین عناصر را به عنوان بارومتر انتخاب می کنیم.

n +1

n

	#	

n - 2

	#		#		#	

$$(n+1)+(n)+(n-2)+(n-6)+\cdots \qquad \Rightarrow O(n^2)$$

در نتیجه زمان مصرفی این الگوریتم در گروه $O(n^2)$ قرار می گیرد.

حال کمی واقع بینانه تر دنبال تعیین زمان مصرفی می رویم. تصاعد حسابی فوق به نظر خیلی ناقص است. اگر کامل بود $O(n^2)$ بدست می آمد. در واقع $O(n^2)$ سقفی است که نمی تواند اتفاق بیافتد.

فرض کنید T(n) زمان مصرفی Quick sort برای مرتب کردن آرایه n عنصری باشد. برای T(n) ، می توانیم رابطه زیر را در یک وضعیت واقع بینانه بنویسیم.

$$T(n) = n+1+\frac{1}{n}\sum_{k=1}^{n}\big(T(k-1)+T(n-k)\big) \qquad *$$

در اولین بازخوانی partition ، $n+1$ مقایسه بین عناصر اتفاق می افتد . عنصر افراز کننده یا در خانه اول قرار می گیرد ، یا در خانه دوم و یا بالاخره در خانه n ام. مقابل $\sum$ ، هر یک از این n حالت نوشته شده است و ضریب $\frac{1}{n}$ میانگین را به ما می دهد.

رابطه * ، تابع بازگشتی متناظر به الگوریتم Quick sort است که باید آنرا حل کنیم. می توانیم از روش های گفته شده در فصل دوم استفاده کنیم و یا آنکه بطور موردی آنرا حل کنیم. در این جا روش موردی را انتخاب می کنیم. قبل از هر کار شرایط اولیه را از روی الگوریتم بدست می آوریم.

$$\begin{cases} T(1)=0 \\ T(0)=0 \end{cases}$$

حال به سراغ رابطه می رویم.

$$T(n)=n+1+\frac{1}{n}\left(2\cdot T(0)+2\cdot T(1)+\cdots+2\cdot T(n-1)\right)$$

$$n\cdot T(n)=n\cdot(n+1)+2(T(0)+T(1)+\cdots+T(n-1))$$

رابطه فوق به ازای هر مقدار n برقرار است. جای n ، n-1 قرار می دهیم. بشکل زیر تبدیل می شود.

$$(n-1)\cdot T(n-1)=(n-1)n+2\left(T(0)+T(1)+\cdots+T(n-2)\right)$$

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

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

طرفین معادله را بر n(n+1) تقسیم می کنیم. نتیجه می شود :

$$\frac{T(n)}{n+1}=\frac{T(n-1)}{n}+\frac{2}{n+1}$$

این رابطه ، یک رابطه بازگشتی است . با مقداردهی می توانیم برایش تقریبی بدست آوریم.

$$=\frac{T(n-2)}{n-1}+\frac{2}{n}+\frac{2}{n+1}=\frac{T(n-3)}{n-2}+\frac{2}{n+1}+\frac{2}{n}+\frac{2}{n+1}=\frac{T(1)}{2}+\frac{2}{3}+\cdots+\frac{2}{n+1}$$

$$=2\cdot\sum_{k=3}^{n+1}\frac{1}{k}$$

$$\sum_{k=3}^{n+1}\frac{1}{k}\le\int_3^{n+1}\frac{1}{x}\cdot dx=\ln(n+1)-\ln(3)<\ln(n+1)$$

$$\Rightarrow\frac{T(n)}{n+1}<\ln(n+1)\Rightarrow T(n)<(n+1)\cdot\ln(n+1)\Rightarrow T(n)\in O\left(n\cdot\log_2^n\right)$$