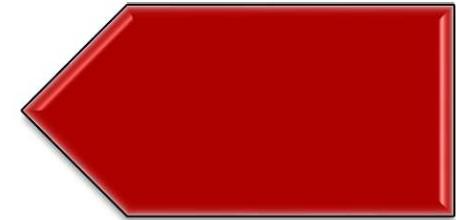


- ۱- مرتبه اجرایی (پیچیدگی اجرایی) Θ, Ω, O تحلیل الگوریتم مرتب سازی درجی
- ۲- رابطه های بازگشته ساختمان داده ها توابع بازگشته
- ۳- روش تقسیم و حل الگوریتم جستجوی دودویی، الگوریتم Quick Sort ، الگوریتم Merge Sort
- ۴- روش پویا برنامه نویسی پویا شامل سری فیبوناچی - ضرب زنجیره ای ماتریس ها - پیدا کردن درخت جستجو
- ۵- روش حریصانه الگوریتم های زمانبندی - الگوریتم کد هافمن - الگوریتم کوله پشتی
- ۶- روش عقبگرد - شاخه و قید
- ۷- الگوریتم های گراف الگوریتم کوتاهترین مسیرها
- ۸- مسائل p و np



۱. حافظه مصرفی

۲. زمان مصرفی

پیچیدگی
زمانی



الگوریتمی بهتر است که فضا و زمان کمتری را بخواهد

زمان اجرای یک الگوریتم با افزایش اندازه ورودی (n) زیاد می شود و زمان اجرا با تعداد دفعاتی که عملیات اصلی انجام می شود تناسب دارد.

float sum (float list[], int n)	برای تعاریف توابع و متغیرها زمانی صرف نمی شود	0
{	یک بلاک باز است که زمانی برای آن صرف نمی شود.	0
float s=0 ; int i;	یک واحد زمانی را برای عمل انتساب $S=0$ صرف می کند	1
for (i = 0; i<n; i++)	یک واحد زمانی $i=0$ و n واحد برای عمل $i \leq n$ صرف می کند	$n+1$
 s = s + list[i];	به تعداد اجرای حلقه این دستور اجرا می شود	n
return s;	برای دستور return هم یک واحد صرف می شود.	1
}	یک بلاک بسته است که زمانی برای آن صرف نمی شود.	0
		$2n+3$

پیچیدگی اجرایی

پیچیدگی یک الگوریتم، تابعی است که مدت زمان اجرای استفاده شده توسط الگوریتم را بر حسب تعداد داده‌های ورودی n اندازه نشانه گذاری می‌گیرد.

به عنوان مثال اگر $n=10$ باشد

$O(n)$

پیچیدگی زمانی کمتری نسبت به

$O(n^2)$

notation	name
$O(1)$	constant
$O(n)$	linear
$O(\log n)$	logarithmic
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential
$O(n!)$	factorial

- مثلا الگوریتمی برای مرتب سازی n عدد

بر اساس اینکه این n عدد در چه زمانی مرتب بشود میگوییم این الگوریتم دارای چه ویژگی است.

هر چقدر پیچیدگی اجرایی کمتر باشد آن الگوریتم بهتر است.

پیچیدگی اجرایی

Constant : $O(1)$ یا ثابت در این الگوریتم به هیچ عنوان مقدار تعداد داده های ورودی اهمیت ندارد.

Linear : $O(n)$ یا خطی

Logarithmic : $O(\log n)$ یا لگاریتمی

Quadratic : $O(n^2)$ یا توان ۲

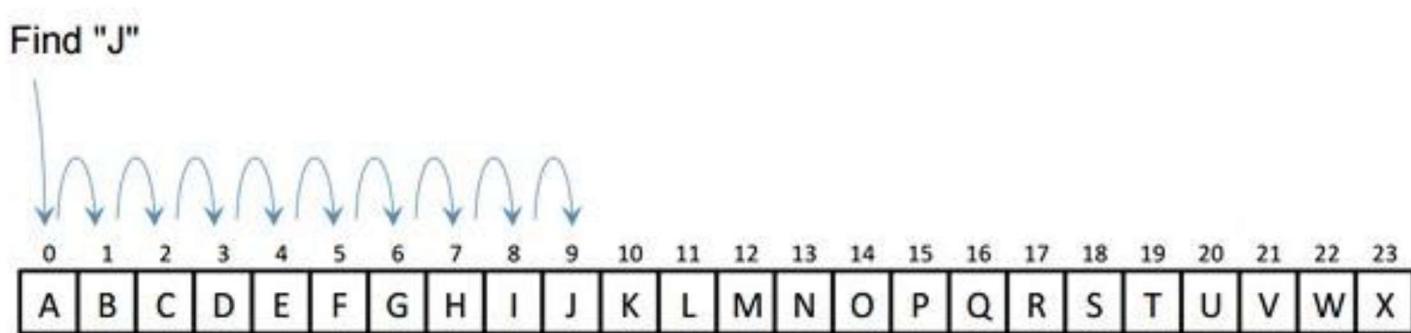
Polynomial : $O(n^c)$ یا چند جمله ای (c یک عدد ثابت است)

exponential : $O(c^n)$ یا نمایی (c یک عدد ثابت است)

Factorial : $O(n!)$ یا فاکتوریل

مسئله جستجو خطی

- یکی از الگوریتم‌هایی که برای جستجوی یک سری داده وجود دارد جستجوی خطی **search linear** است.
- این الگوریتم کلیه عناصر درون یک لیست را یکی یکی بررسی می‌کند تا آرگومان جستجو پیدا شود.



Search 23	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>2</td><td>5</td><td>8</td><td>12</td><td>16</td><td>23</td><td>38</td><td>56</td><td>72</td><td>91</td></tr></table>	0	1	2	3	4	5	6	7	8	9	2	5	8	12	16	23	38	56	72	91	
0	1	2	3	4	5	6	7	8	9													
2	5	8	12	16	23	38	56	72	91													
23 > 16	<table border="1"><tr><td>L=0</td><td>1</td><td>2</td><td>3</td><td>M=4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>H=9</td></tr><tr><td>take 2nd half</td><td>2</td><td>5</td><td>8</td><td>12</td><td>16</td><td>23</td><td>38</td><td>56</td><td>72</td><td>91</td></tr></table>	L=0	1	2	3	M=4	5	6	7	8	H=9	take 2 nd half	2	5	8	12	16	23	38	56	72	91
L=0	1	2	3	M=4	5	6	7	8	H=9													
take 2 nd half	2	5	8	12	16	23	38	56	72	91												
23 > 56	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>L=5</td><td>6</td><td>M=7</td><td>8</td><td>H=9</td></tr><tr><td>take 1st half</td><td>2</td><td>5</td><td>8</td><td>12</td><td>16</td><td>23</td><td>38</td><td>56</td><td>72</td><td>91</td></tr></table>	0	1	2	3	4	L=5	6	M=7	8	H=9	take 1 st half	2	5	8	12	16	23	38	56	72	91
0	1	2	3	4	L=5	6	M=7	8	H=9													
take 1 st half	2	5	8	12	16	23	38	56	72	91												
Found 23, Return 5	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>L=5, M=5</td><td>H=6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>2</td><td>5</td><td>8</td><td>12</td><td>16</td><td>23</td><td>38</td><td>56</td><td>72</td><td>91</td></tr></table>	0	1	2	3	4	L=5, M=5	H=6	7	8	9		2	5	8	12	16	23	38	56	72	91
0	1	2	3	4	L=5, M=5	H=6	7	8	9													
	2	5	8	12	16	23	38	56	72	91												

مسئله جستجو دودویی

الگوریتم جستجوی دودویی تکنیکی است برای یافتن یک مقدار عددی از میان مجموعه‌ای از اعداد مرتب.

این متده محدوده جستجو را در هر مرحله به نصف کاهش می‌دهد، بنابراین هدف مورد نظر یا به زودی پیدا می‌شود و یا مشخص می‌شود که مقدار مورد جستجو در فهرست وجود ندارد.

در این روش عنصر مورد نظر با خانه وسط آرایه مقایسه می‌شود اگر با این خانه برابر بود جستجو تمام می‌شود اگر عنصر مورد جستجو از خانه وسط بزرگتر بود جستجو در بخش بالایی آرایه و در غیر این صورت جستجو در بخش پایینی آرایه انجام می‌شود(فرض کرده ایم آرایه به صورت صعودی مرتب شده است) این رویه تا یافتن عنصر مورد نظر یا بررسی کل خانه‌های آرایه ادامه می‌یابد.

مسئله جستجو یا Search

به عنوان مثال ما ده عدد داریم و می خواهیم در این ده عدد دنبال یک عدد بگردیم و ببینیم آن عدد جز این ده

عدد هست یا خیر؟

• در مسائل جستجو یا Search اگر الگوریتم به این صورت باشد که عنصری را که دنبال آن هستیم با تک

تک عناصر دیگر مقایسه کنیم می شود جستجوی خطی.

• در جستجوی خطی در بدترین حالت شما در آخرین مقایسه به عنصر مدنظرتان خواهید رسید. یعنی n

عنصرتان را پیمایش می کنید پس Order آن $O(n)$ خواهد بود.

• اگر جستجو دودویی باشد Order آن می شود $O(\log n)$

مسئله جستجو یا Search

اگر ۱۶ عدد داشته باشیم و هر جستجو یک ثانیه وقت بگیرد به روش خطی $O(16)$ که همان ۱۶ ثانیه است

زمان می برد تا جستجوی ما به نتیجه برسد.

ولی در همین مثال با روش دودویی $(\log_2 16) = 4$ در نتیجه $O(\log 16)$ که ۴ ثانیه است زمان کمتری

پس میبینیم در بدترین حالت که مقایسه با عنصر آخر انجام شود روش دو دویی که ۴ ثانیه است زمان کمتری

برای جستجو در نظر می گیرد تا ۱۶ ثانیه

- پس الگوریتمی بهتر عمل می کند که دارای Order کمتری باشد.

- از نماد O برای استفاده بدترین حالت ممکن استفاده می گردد.

مرتبه اجرایی چند جمله

مرتبه اجرایی چند جمله‌ای ها خیلی ساده پیدا می‌شوند.

در صورتی که داشته باشیم:

$$f(n) = n^m + n^{m-1} + \dots + n^2 + n + c \Rightarrow f(n) = O(n^m)$$

یعنی جمله اثرگذار در این رابطه چون این جمله بزرگ‌تر از جمله‌های دیگر است.

مرتبه اجرایی چند جمله

مرتبه اجرایی چند جمله‌ای ها خیلی ساده پیدا می‌شوند.

در صورتی که داشته باشیم:

$$f(n) = n^m + n^{m-1} + \dots + n^2 + n + c \Rightarrow f(n) = O(n^m)$$

یعنی (n^m) جمله اثرگذار در این رابطه چون این جمله بزرگ‌تر از جمله‌های دیگر است.

$$f(n) = 5n^2 - 3n + 4 \Rightarrow O(n^2)$$

یعنی (n^2) جمله اثرگذار در این رابطه چون این جمله بزرگ‌تر از جمله‌های دیگر است.

$$f(n) = n - 6n^8 + n^2 \Rightarrow O(n^8)$$

یعنی (n^8) جمله اثرگذار در این رابطه چون این جمله بزرگ‌تر از جمله‌های دیگر است.

مقایسه مراتب اجرائی

$$O(1) < O(\lg n) < O(n) < O(n \lg n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$

$$< O(n^4) \quad < O(5^n)$$

$$O(1) < O(\lg n) < O(n) < O(n \lg n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$

الگوریتمی بنویسید که عمل سرچ را انجام دهد، سرچ را می توانیم با دستور $O(n)$ بنویسیم
یا با دستور $O(\lg n)$

با کدام دستور عمل سرچ بهتری انجام می شود؟

$$O(1) < O(\lg n) < O(n) < O(n \lg n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$

الگوریتمی بنویسید که عمل سرچ را انجام دهد، سرچ را می توانیم با دستور $O(n)$ بنویسیم یا با دستور $O(\lg n)$

با کدام دستور عمل سرچ بهتری انجام می شود؟
با $O(\lg n)$ چون زمان کمتری را جهت سرچ کردن مطرح خواهد کرد.

$$n! + 2^n + 1000n^{10} \Rightarrow o(n!)$$

یعنی $(n!)$ جمله اثرگذار در این رابطه چون این جمله بزرگ‌تر از جمله‌های دیگر است.

مرتبه اجرایی حلقه های ساده

```
for (i=a ; i<=b ; i=i+k )
    s;
```

$$\frac{b-a+1}{k}$$

```
for (i=b ; i>=a ; i=i-k )
    s;
```

```
for ( i=1 ; i<=n ; i=i+1 )
    s;
```

$$\rightarrow \frac{n-1+1}{1} = n$$

```
for ( i=3 ; i<=n ; i=i+2 )
    s;
```

$$\rightarrow \frac{n-3+1}{2} = \frac{n}{2} - 1$$

```
for ( i=9 ; i<3n+4 ; i=i+5 )
    s;
```

$$\rightarrow \frac{3n+4-9}{5} = \frac{3n}{5} - 1$$

تعداد اجرا هر سه حلقه برابر n خواهد بود.

مرتبه اجرا یا پیپیدگی اجرایی هر سه حلقه برابر $O(n)$ و خطی خواهد بود.

مرتبه اجرایی لگاریتمی

```
for ( i=a ; i<=b ; i=i*k )
    s;
```

$$\log_k^b - \log_k^a + 1$$

```
for ( i=b ; i>=a ; i=i/k )
    s;
```

```
for ( i=1 ; i<=8 ; i=i*2 )
    s;
```

→ $\log_2^8 - \log_2^1 + 1 = 4$

مرتبه اجرایی این دستور ثابت است و برابر $O(1)$ هست.

```
for ( i=27 ; i<=n ; i=i*3 )
    s;
```

→ $\log_3^n - \log_3^{27} + 1 = \log_3^n - 2$

مرتبه اجرایی $\log_3 n$ که همان لگاریتمی هست

مرتبه اجرایی حلقه های تو در
تو

```
for ( i=1 ; i<=n ; i++ ) → n
  for ( j=1 ; j<=n ; j++) → n
    s;
```

$$n^2$$

از ۲ میرود تا n شمارنده آن ۴ تا ۴ تا اضافه می شود.

```
for ( i=2 ; i<=n ; i=i+4 ) →
  for ( j=n ; j>3 ; j=j-2 )
    s;
```

از n به ۳ میرود شمارنده آن ۲ تا ۲ تا کم می شود.

$$\rightarrow \frac{n-2+1}{4} \times \frac{n-3}{2} \Rightarrow O(n^2)$$

چون علامت = ندارد $+ 1$ را نمی گذاریم.

```
for ( i=1 ; i<=n ; i=i*2 )
  for ( j=1 ; j<=n ; j++)
    s;
```

$$\rightarrow (lg n + 1) \times n \Rightarrow O(n \lg n)$$

مرتبه اجرایی حلقه های پشت
سر هم

```
for ( i=1 ; i<=n ; i++)  
{  
    s;  
}  
  
for ( j=1 ; j<=m ; j++)  
{  
    s;  
}
```

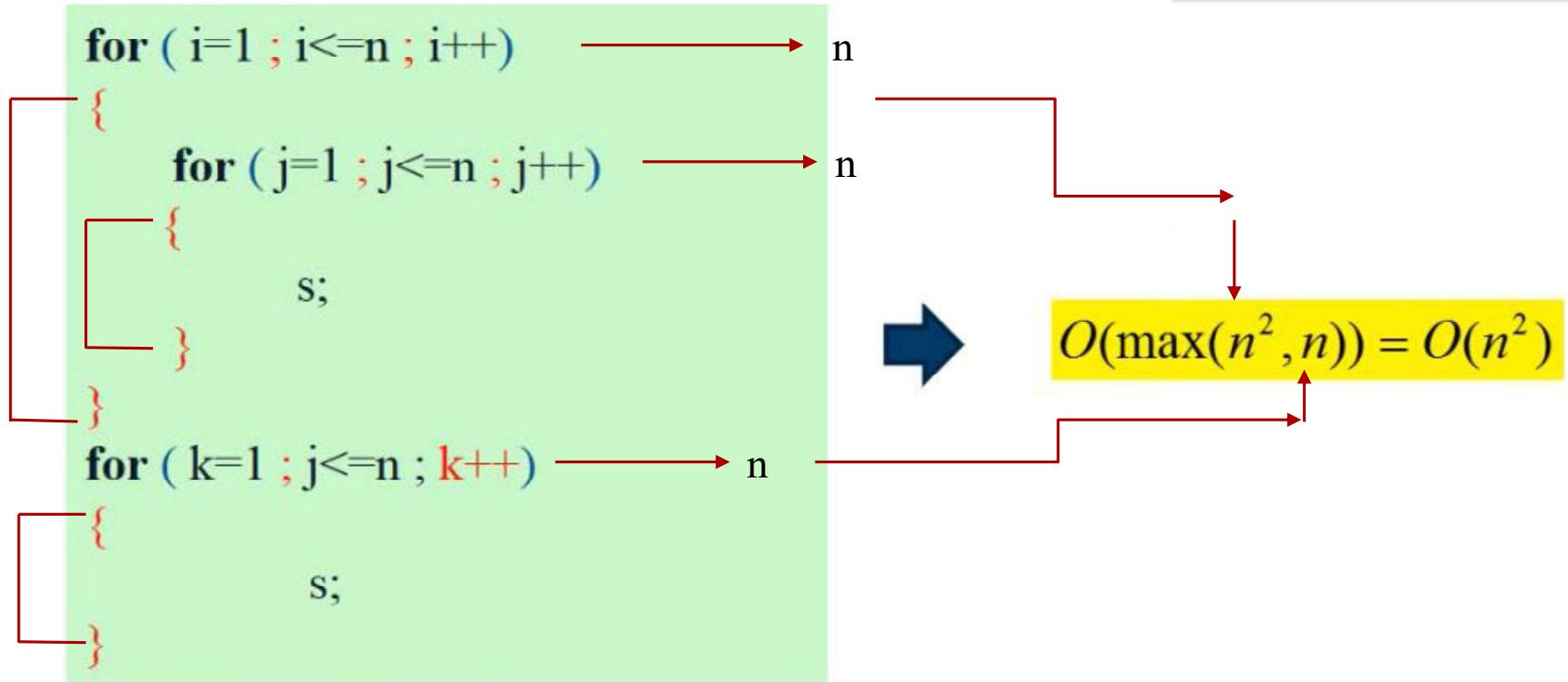
از ۱ تا n دستور s را انجام میدهد.

$$\rightarrow O(\max(n,m))$$

از ۱ تا m دستور s را انجام میدهد.

$$O(n + m) \text{ یا}$$

ترکیب حلقه های تو در تو
پشت سر هم



مثال

```
x=0;  
for ( i=1 ; i <=n ; i++ )  
{  
    for ( j=1 ; j <=n ; j++ )  
        x++;  
    j=1;  
    while ( j < n )  
    {  
        x++;  
        j = j*2;    logn  
    }  
}
```

n

n

$$n(n + \lceil \log n \rceil) = n^2 + n\lceil \log n \rceil$$

مرتبه اجرایی حلقه های تو در
تو وابسته

```
for ( i=1 ; i<=n ; i++ )  
    for ( j=1 ; j<=i ; j++ )  
        S;
```

i	1	2	3	...	n
تعداد تکرار	1	2	3	...	n

این فرمول حفظ شود

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2} \Rightarrow O(n^2)$$

نمادهای پیچیدگی اجرایی

O

اوی بزرگ

Ω

امگا بزرگ

θ

تتا

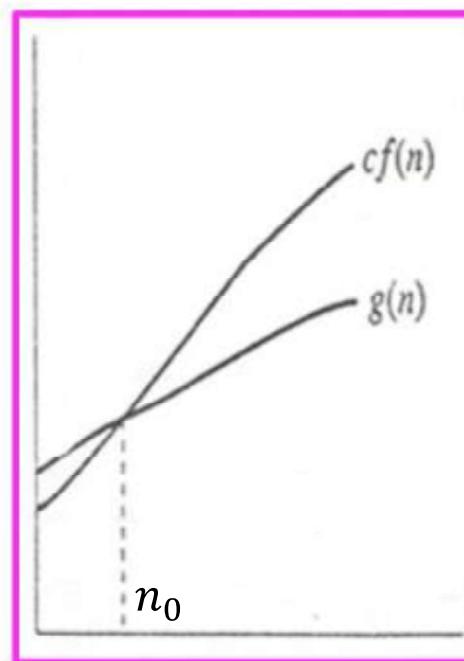
اوی بزرگ

$$g(n) \in O(f(n))$$
 عبارت

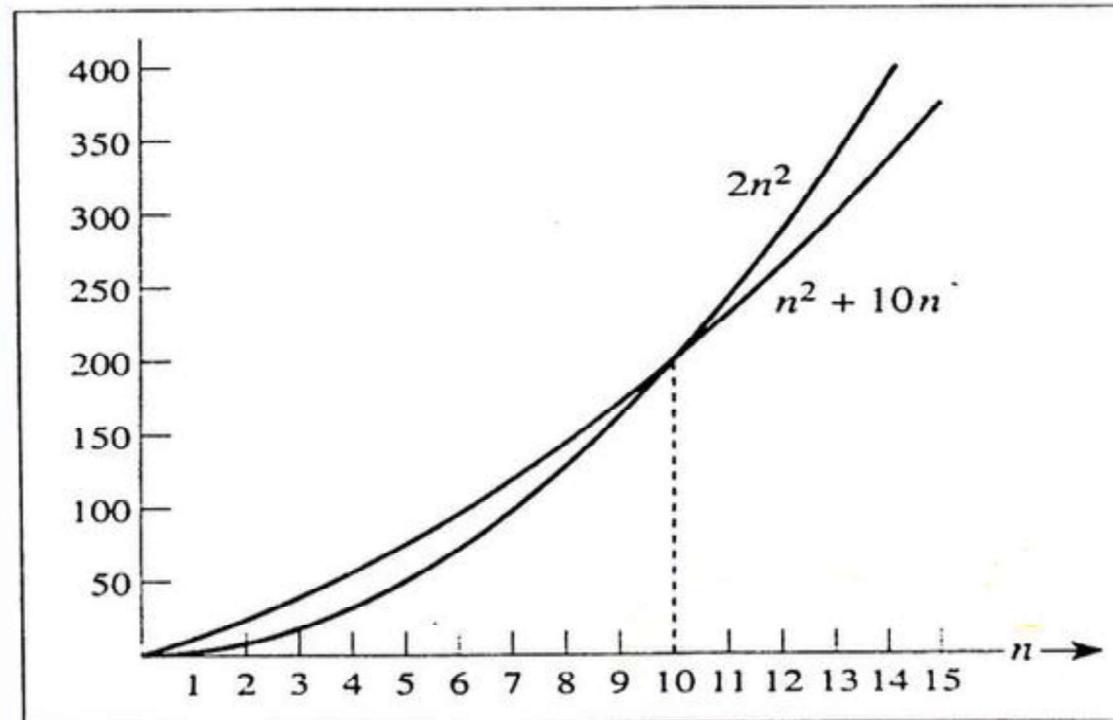
معنی: برای تابع پیچیدگی مفروض $O(f(n))$ ، $f(n)$ به مجموعه ای از توابع اشاره دارد که برای آنها ثابت‌های c و n_0 وجود دارند، بطوریکه برای همه $n \geq n_0$ داریم:

$$g(n) \leq cf(n)$$

یک عدد ثابت



مثال اوی بزرگ



$$n^2 + 10n \in O(n^2)$$

$$n^2 + 10n \leq 2n^2$$

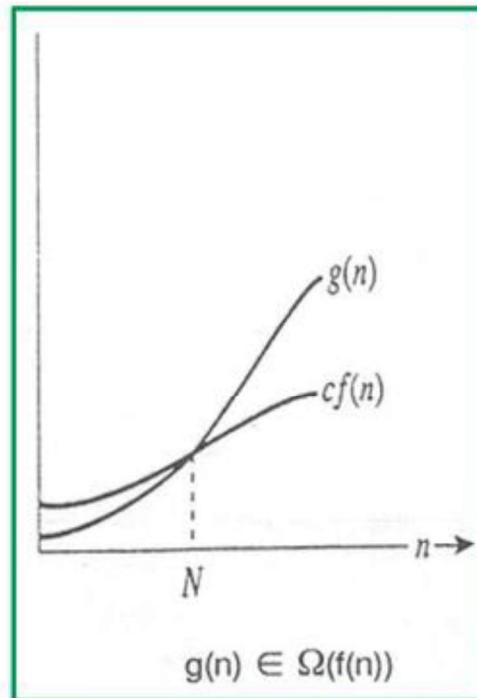
$$\begin{aligned}n^2 + 10n &= 2n^2 \\ \Rightarrow 10n &= n^2 \\ \Rightarrow n &= 10\end{aligned}$$

مثال امگای بزرگ

امگا بزرگ

عبارت $g(n) \in \Omega(f(n))$

یعنی: برای تابع پیچیدگی مفروض $f(n)$ ، $\Omega(f(n))$ به مجموعه ای از توابع اشاره دارد که برای آنها ثابت‌های c و n_0 وجود دارند، بطوریکه برای همه $n \geq n_0$ داریم:



مثال امگای بزرگ

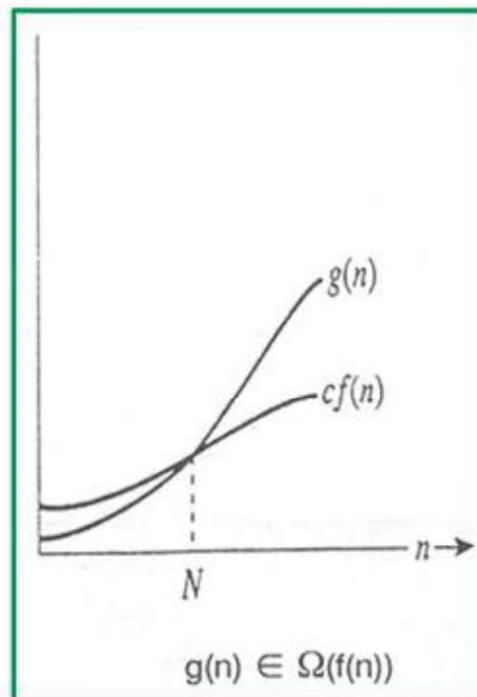
امگا بزرگ

عبارت $g(n) \in \Omega(f(n))$

یعنی: برای تابع پیچیدگی مفروض $f(n)$ ، $\Omega(f(n))$ به مجموعه ای از توابع اشاره دارد که برای آنها ثابت‌های c و n_0 وجود دارند، بطوریکه برای همه $n \geq n_0$ داریم:

$$n^5 \in \Omega(f(n^2))$$

$$n^5 \geq n^2$$



مثال تا بزرگ

عبارت $g(n) \in \theta(f(n))$

يعني : $g(n) \in \Omega(f(n))$ و $g(n) \in O(f(n))$

$$n^2 \in \Theta(f(n^2+6))$$

هر دو هم رتبه هستند

$$n^2 = n^2 + 6$$

چون لگاریتم ها هم رتبه هم هستند.

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

مثال

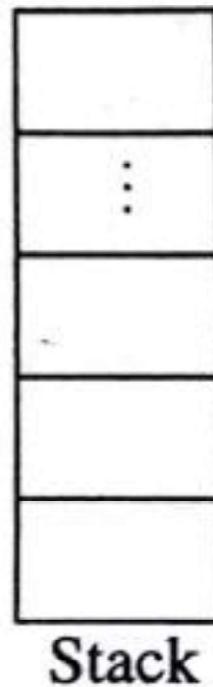
$$O(n^2) = \{\lg n, n, 7n^2 + 4, n \lg n, n^2, 5n^2 - 6, \dots\}$$

$$\Omega(n^2) = \{n^2, 5n^2 - 6, n^2 \lg n, n^3, n^6, 7n^2 + 4, \dots\}$$

$$\theta(n^2) = \{n^2, 5n^2 - 6, 7n^2 + 4, \dots\}$$

- ❖ آخرین عنصری که وارد می شود اولین عنصری است که خارج می شود.
- ❖ LIFO (Last Input First Output)
- ❖ ذخیره آدرس تابع بازگشت در توابع بازگشتی

n آخر پشتہ



Stack : Array [1..n] of items;

دامنه تغیرات top : 0 .. n ; : top

مقدار اولیه top := 0 ;

شرط حالی بودن پشتہ : if top = 0

شرط پر بودن پشتہ : if top = n

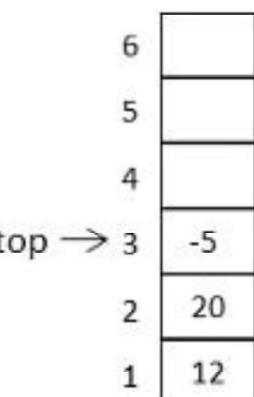
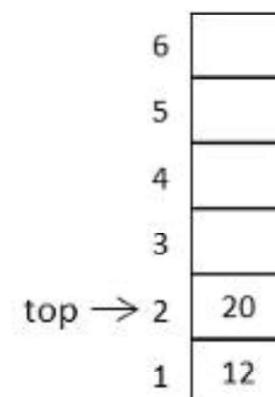
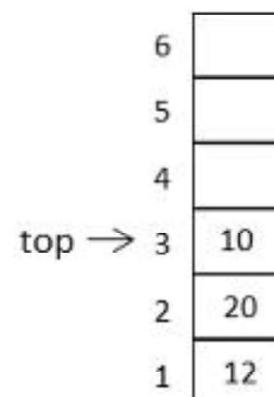
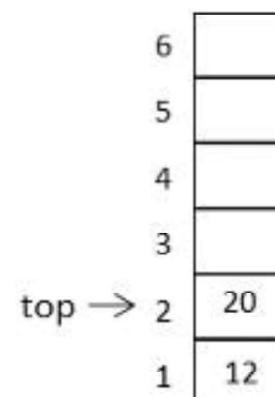
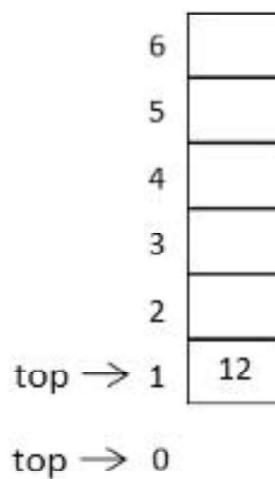
زیربرنامه حذف از پشته

```
procedure pop(var K:items);
begin
    if top=0 then
        stackempty
    else begin
        K:=stack[top]
        top:=top-1;
    end;
end;
```

زیربرنامه درج در پشته

```
procedure push (K:items);
begin
    if top = n then
        stackfull
    else begin
        top:=top+1;
        stack[top]:=K;
    end;
end;
```

Push(12);push(20);push(10);push(-2);pop(A);pop(B);push(-5)



پشته

زیربرنامه حذف از پشته

```
items pop( )
{
    if (top == -1)
        stackempty( );
    else
        return stack [top --];
}
```

زیربرنامه درج در پشته

```
void push (items k)
{
    if (top == n-1)
        stackfull( );
    else
        stack[++top] = k;
}
```

پشتہ

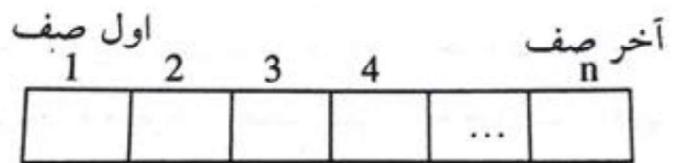
q:Array [1..n] of items;

Front , rear : 0 .. n دامنه تغییرات:

Front = rear = 0; مقدار اولیه :

شرط حالی بودن صفحه: if Front = rear

شرط پر بودن صف : if rear = n



صف معمولی (Queue)

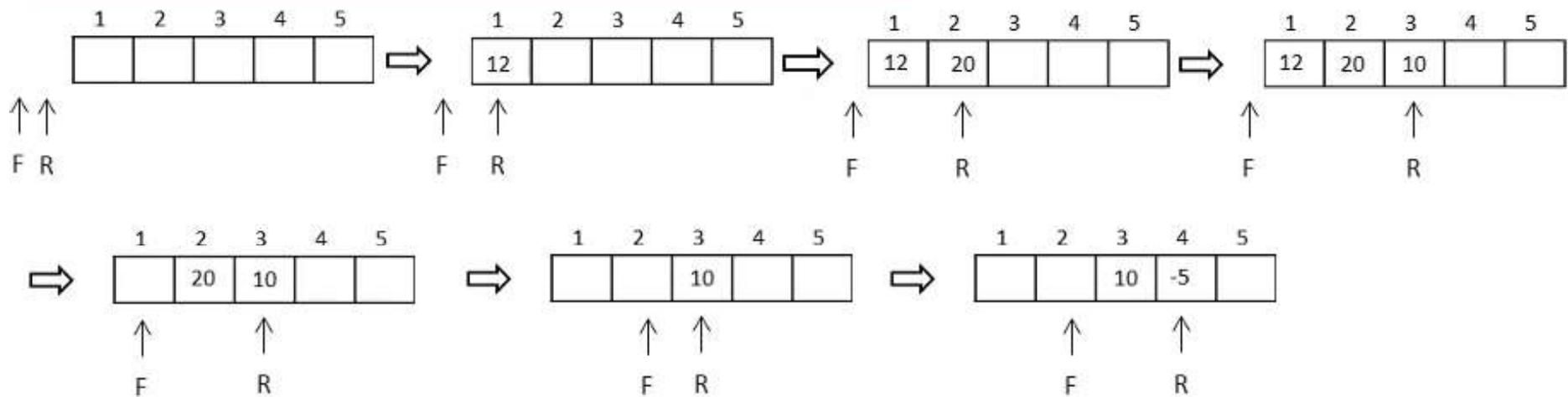
زیربرنامه درج در صف

```
procedure addq (K:items);
begin
    if rear = n then
        queuefull
    else begin
        rear:=rear+1;
        q[rear]:=K;
    end;
end;
```

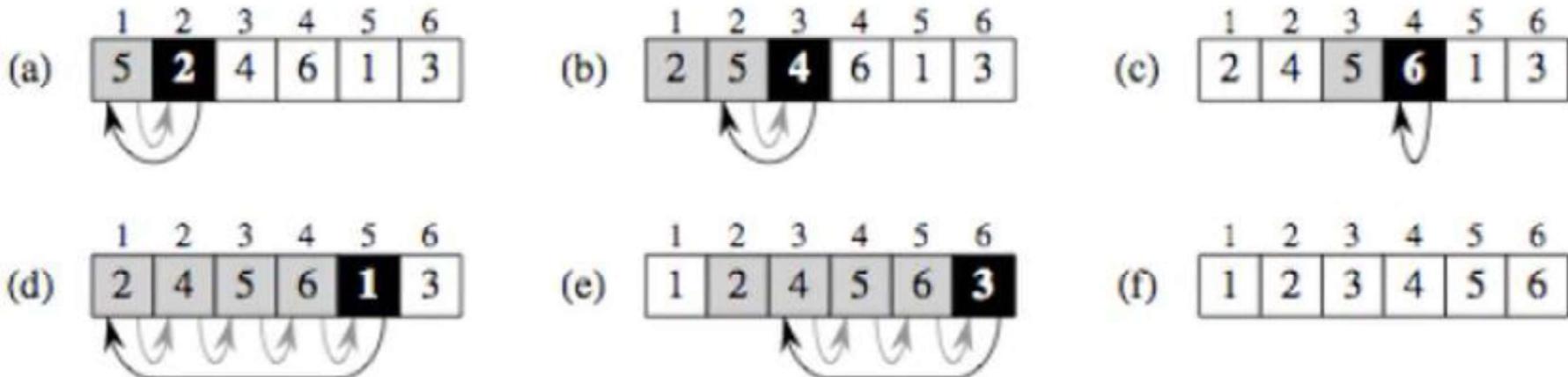
زیربرنامه حذف از صف

```
procedure delq (var K:items);
begin
    if front = rear then
        queueempty
    else begin
        front=front+1;
        k:=q[front];
    end;
end;
```

addq(12);addq(20);addq(10);delq(A);delq(B);addq(-5)



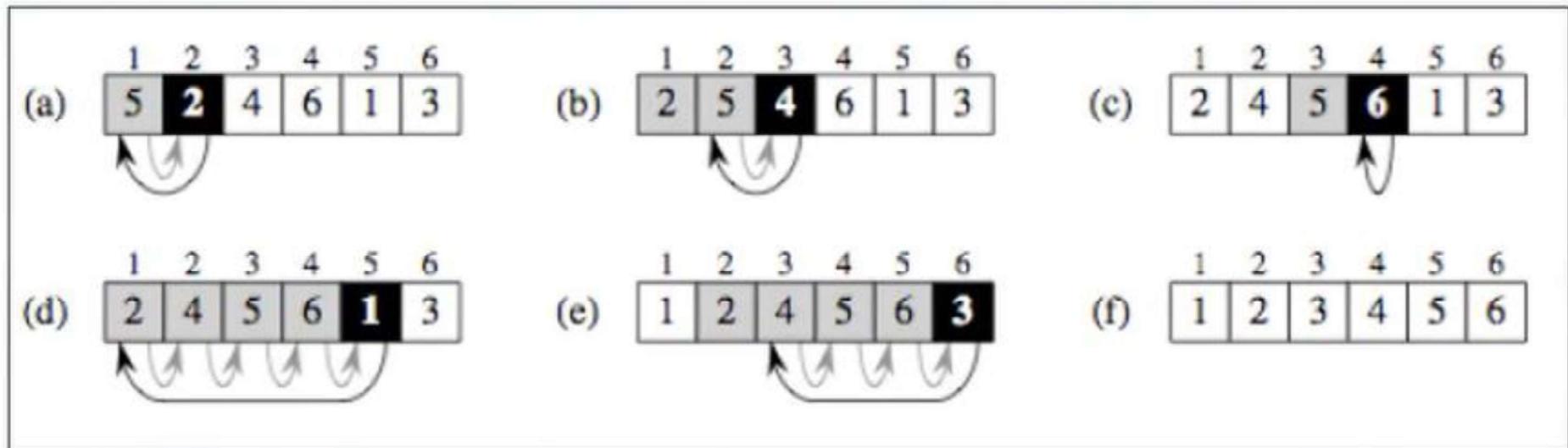
مرتب سازی درجی



دادن یک ایده برای موضوع:

- فرض کنید شما این ۶ عدد موجود را به عنوان کارت هایی در نظر می گیرید.
- کارت دوم و اول را مقایسه می کنیم اگر کارت دوم کوچکتر بود به سمت چپ شیفت می دهیم.
- کارت سوم را با کارت دوم مقایسه می کنیم اگر کارت سوم کوچکتر بود دوباره به سمت چپ شیفت می دهیم.
- کارت چهارم را با کارت سوم مقایسه می کنیم می بینیم که از کارت سوم و بقیه کارت ها بزرگتر هست پس سر جای خودش باقی می ماند.

مرتب سازی درجی



- کارت پنجم را با کارت چهارم مقایسه می کنیم که از کارت چهارم و بقیه کارت ها کوچکتر هست هست پس به اندازه تمام اعداد بزرگتر از خودش به سمت چپ شیفت داده می شود.
- کارت ششم را با کارت پنجم مقایسه می کنیم که از کارت پنجم و بقیه کارت ها کوچکتر هست پس به اندازه تمام اعداد بزرگتر از خودش به سمت چپ شیفت داده می شود.

مرتب سازی درجی

اعداد را یکی یکی در محل درست درج می کنیم:

- 8, 5, 3, 21, 12, 11, 6, 2
- 5, 8, 3, 21, 12, 11, 6, 2
- 3, 5, 8, 21, 12, 11, 6, 2
- 3, 5, 8, 21, 12, 11, 6, 2
- 3, 5, 8, 12, 21, 11, 6, 2
- 3, 5, 8, 11, 12, 21, 6, 2
- 3, 5, 6, 8, 11, 12, 21, 2
- 2, 3, 5, 6, 8, 11, 12, 21

- بهترین حالت در الگوریتم مرتب سازی درجی این هست که اعداد Sort شده یا مرتب شده باشد.
- بدترین حالت این هست که اعداد به صورت معکوس از بزرگ به کوچک sort شده باشد.

الگوریتم

INSERTION-SORT(A)

```

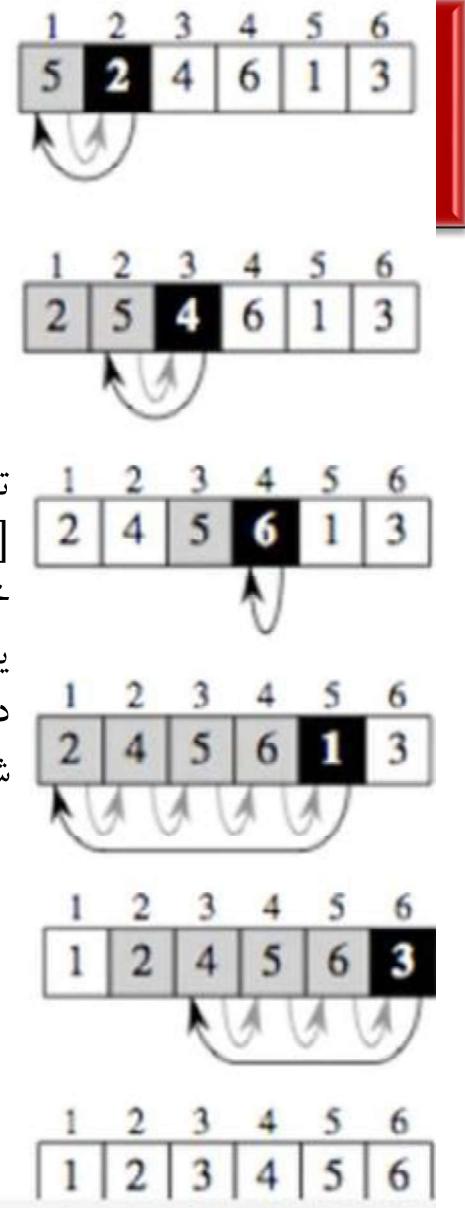
1   for  $j \leftarrow 2$  to  $\text{length}[A]$ 
2     do  $key \leftarrow A[j]$ 
3       ▷ Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$ .
4        $i \leftarrow j - 1$ 
5       while  $i > 0$  and  $A[i] > key$ 
6         do  $A[i + 1] \leftarrow A[i]$ 
7          $i \leftarrow i - 1$ 
8        $A[i + 1] \leftarrow key$ 

```

شمارنده j از ۲ تا طول آرایه را چک می کند
عنصر j که عدد ۲ هست می شود key ما
تا وقتی i بزرگتر از صفر ($i=2$) و $A[i]=5$
بزرگتر از key هست خانه ای که ۵ در آن قرار دارد به اندازه
یک خانه به جلو می رود و خانه ای که ۴ در آن قرار دارد یک واحد به سمت چپ
شیفت داده می شود

یک کامنت هست که می گوید درج کن عنصر را به آرایه مرتب شده از شماره ۱ تا $j-1$
یعنی هر محله که i تغییر می کند (که در اینجا ۱ شما ۲ است) از ۱ تا $j-1$ (که در اینجا منظور
از خانه شماره ۱ است) آرایه شما مرتب شده هست.

با دستور `while` ما عمل `shift` را انجام می دهیم.



INSERTION-SORT(A)

```

1  for  $j \leftarrow 2$  to  $\text{length}[A]$ 
2    do  $key \leftarrow A[j]$ 
3      ▷ Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$ .
4       $i \leftarrow j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6        do  $A[i + 1] \leftarrow A[i]$ 
7         $i \leftarrow i - 1$ 
8       $A[i + 1] \leftarrow key$ 

```

<i>cost</i>	<i>times</i>
c_1	n
c_2	$n - 1$
0	$n - 1$
c_4	$n - 1$
c_5	$\sum_{j=2}^n t_j$
c_6	$\sum_{j=2}^n (t_j - 1)$
c_7	$\sum_{j=2}^n (t_j - 1)$
c_8	$n - 1$

تحليل الگوریتم

- حلقه اول از مقدار ۲ تا طول آرایه می‌رود پس اگر طول آرایه را n در نظر بگیریم تکرار حلقه از رابطه $n-2+1 = n-1$ به دست می‌آید، اما یکبار هم j مقداری بالاتر از طول آرایه می‌گیرد و متوجه می‌شود که شرط دیگر برقرار نیست، پس در واقع این حلقه n بار تکرار می‌شود.
- دستورات داخل حلقه مثل دستور ۲ و ۴ و ۸، $n-1$ بار اجرا می‌شود.
- در دستور ۵ حلقه while چون خیلی وابسته به این است که عنصری که داریم چقدر شیفت احتیاج دارد و تعداد اجراهای به عدد مقدار شیفت وابسته است، ما یک t_j تعریف می‌کنیم. t_j تعداد مقایسات لازم را نشان می‌دهد برای اینکه عنصر j ام سر جای خودش قرار بگیرد.
وقتی مقدار $j=2$ باشد مقدار t_2 هست ، وقتی مقدار $j=3$ باشد مقدار t_3 هست و در نهایت ما تمام این t ها را از ۲ تا n با هم جمع می‌کنیم.
- دو دستور ۶ و ۷ مقدار (t_{j-1}) را به خود می‌گیرند چون یکبار هم دستور while مقداری بالاتر از طول آرایه می‌گیرد و متوجه می‌شود که شرط دیگر برقرار نیست پس دو دستور ۶ و ۷ مقدار (t_{j-1}) را می‌گیرند.

INSERTION-SORT(A)

```

1  for  $j \leftarrow 2$  to  $\text{length}[A]$ 
2    do  $key \leftarrow A[j]$ 
3      ▷ Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$ .
4       $i \leftarrow j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6        do  $A[i + 1] \leftarrow A[i]$ 
7         $i \leftarrow i - 1$ 
8       $A[i + 1] \leftarrow key$ 
```

<i>cost</i>	<i>times</i>
c_1	n
c_2	$n - 1$
0	$n - 1$
c_4	$n - 1$
c_5	$\sum_{j=2}^n t_j$
c_6	$\sum_{j=2}^n (t_j - 1)$
c_7	$\sum_{j=2}^n (t_j - 1)$
c_8	$n - 1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n t_j - 1.$$

تحليل الكوريتم

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^{n-2+1} t_j + (c_6 + c_7) \sum_{j=2}^n t_j - 1.$$

Best Case:

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5(n - 1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

آرایه مرتب است و $t_j=1$

Worst Case:

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2}\right. \\ &\quad \left.- \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

آرایه مرتب معکوس است و $t_j=j$

تحليل الگوریتم

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n t_j - 1.$$

Best Case:

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5(n - 1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

آرایه مرتب است و $tj=1$

$$\begin{aligned} &c_1 n + c_2 n + c_4 n + c_8 n - c_2 - c_4 - c_8 + c_5 n - c_5 \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

تحليل الگوریتم

اگر $j=1$ باشد از فرمول $\frac{n(n+1)}{2}$ ولی چون j از ۲ تا n هست فرمول به صورت

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n t_j - 1.$$

Worst Case:

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + (c_6 + c_7) \left(\frac{n(n-1)}{2} \right) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} \right. \\ &\quad \left. - \frac{c_7}{2} + c_8 \right) n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

آرایه مرتب معکوس است و $t_j=j$

INSERTION-SORT(A)

```

1  for  $j \leftarrow 2$  to  $\text{length}[A]$ 
2    do  $key \leftarrow A[j]$ 
3      ▷ Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$ .
4       $i \leftarrow j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6        do  $A[i + 1] \leftarrow A[i]$ 
7         $i \leftarrow i - 1$ 
8       $A[i + 1] \leftarrow key$ 
```

<i>cost</i>	<i>times</i>
c_1	n
c_2	$n - 1$
0	$n - 1$
c_4	$n - 1$
c_5	$\sum_{j=2}^n t_j$
c_6	$\sum_{j=2}^n (t_j - 1)$
c_7	$\sum_{j=2}^n (t_j - 1)$
c_8	$n - 1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n t_j - 1.$$

فاکتوریل

فاکتوریل (به فرانسوی: Factorielle) هر عدد طبیعی در ریاضیات از حاصل ضرب آن عدد در تمام اعداد طبیعی کوچکتر از آن بدون صفر به دست می‌آید. فاکتوریل عددی مانند $n!$ می‌نویسند و «إن فاکتوریل» می‌خوانند. همچنین طبق قرارداد، فاکتوریل صفر همیشه برابر با یک است.

$$n! = n * (n - 1) * (n - 2) * (n - 3) * \dots * 3 * 2 * 1$$

فرمول سادهٔ محاسبهٔ فاکتوریل

$$0! = 1$$

$$1! = 1$$

$$2! = 2 * 1 = 2$$

$$3! = 3 * 2 * 1 = 6$$

$$4! = 4 * 3 * 2 * 1 = 24$$

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

توابع بازگشتی

یک رابطه‌ی بازگشتی برای تعداد ضرب‌ها در تابع فاکتوریل:

```
fact(n)
{
    if (n==0)
        return 1;
    else
        return n*fact(n-1);
}
```

$$T(n) = \begin{cases} 0 & n = 0 \\ 1 + T(n - 1) & n >= 1 \end{cases}$$

- ❖ می‌خواهیم در مورد توابع بازگشتی و روش‌های حل توابع بازگشتی در این مبحث توضیح دهیم.
- ❖ این تابع فاکتوریل مقدار پارامتر ورودی که در اینجا n هست را حساب می‌کند.
- ❖ اگر $n=0$ باشد مقدار ۱ برگردانده می‌شود چون فاکتوریل عدد ۰ برابر ۱ هست.
- ❖ اگر مقدار $n=5$ باشد مقدار فاکتوریل ما می‌شود $5 * \text{fact}(4)$

توابع بازگشتی

یک رابطه‌ی بازگشتی برای تعداد ضرب‌ها در تابع فاکتوریل:

```
fact(n)
{
    if (n==0)
        return 1;
    else
        return n*fact(n-1);
}
```

اگر فاکتوریل ، را بخواهیم نشان دهیم تعداد ضرب‌های مورد نیاز ، است.

$$T(n) = \begin{cases} 0 & n = 0 \\ 1 + T(n-1) & n >= 1 \end{cases}$$

❖ یعنی ما داخل تابع فاکتوریل فاکتوریل را صدا میزنیم.

❖ یعنی داخل یک تابع بازگشت می کنیم به خود تابع.

❖ در این تابع بازگشتی یک رابطه برای ضرب رابطه بنویس که مشخص کننده تعداد ضرب‌ها باشد. چون فاکتوریل ، یک هست و نیازی به عمل ضرب نیست.

❖ به ازای به دست آوردن ضرب فاکتوریل اعداد بزرگتر یا مساوی n (مثلا عدد ۵) می شود ($n * fact(n-1)$)
 $5 * (4 * 3 * 2 * 1)$

توابع بازگشته

```
void f(int n)
{
    if (n>=2)
    {
        f (n-1);
        print "***";
        f(n-2);
        print "****";
        f(n-1);
    }
}
```

: تعداد ستاره های چاپ شده $T(n)$

$$T(n) = 2T(n-1) + T(n-2) + 7$$

- ❖ ما تابع n از نوع int داریم.
 - ❖ اگر $n \geq 2$ باشد دوباره تابع خودش را با مقدار $(n-1)$ صدا میزنند.
 - ❖ سپس سه ستاره را چاپ می نماید.
 - ❖ سپس دوباره $(n-2)$ صدا زده می شود.
 - ❖ سپس چهار ستاره را چاپ می نماید.
 - ❖ سپس دوباره $(n-1)$ صدا زده می شود.
- یعنی در داخل تابع f ما سه مرتبه فراخوانی بازگشتی را داریم.

توابع بازگشته

```
void f(int n)
{
    if (n>=2)
    {
        f (n-1);
        print "***";
        f(n-2);
        print "****";
        f(n-1);
    }
}
```

: تعداد ستاره های چاپ شده $T(n)$

$$T(n) = 2T(n-1) + T(n-2) + 7$$

- ❖ $T(n)$ تعداد ستاره های چاپ شده هست.
- ❖ برای این هست که تابع با مقدار $n-1$ دو بار صدای $2T(n-1)$ زده شده است.
- ❖ $T(n-2)$ برای این هست که تابع با مقدار $n-2$ دو بار صدای زده است.

$$\begin{aligned} T(n-1) + 3 + T(n-2) + 4 + T(n-1) &= \\ 2T(n-1) + T(n-2) + 7 \end{aligned}$$

مسئله خرگوش ها

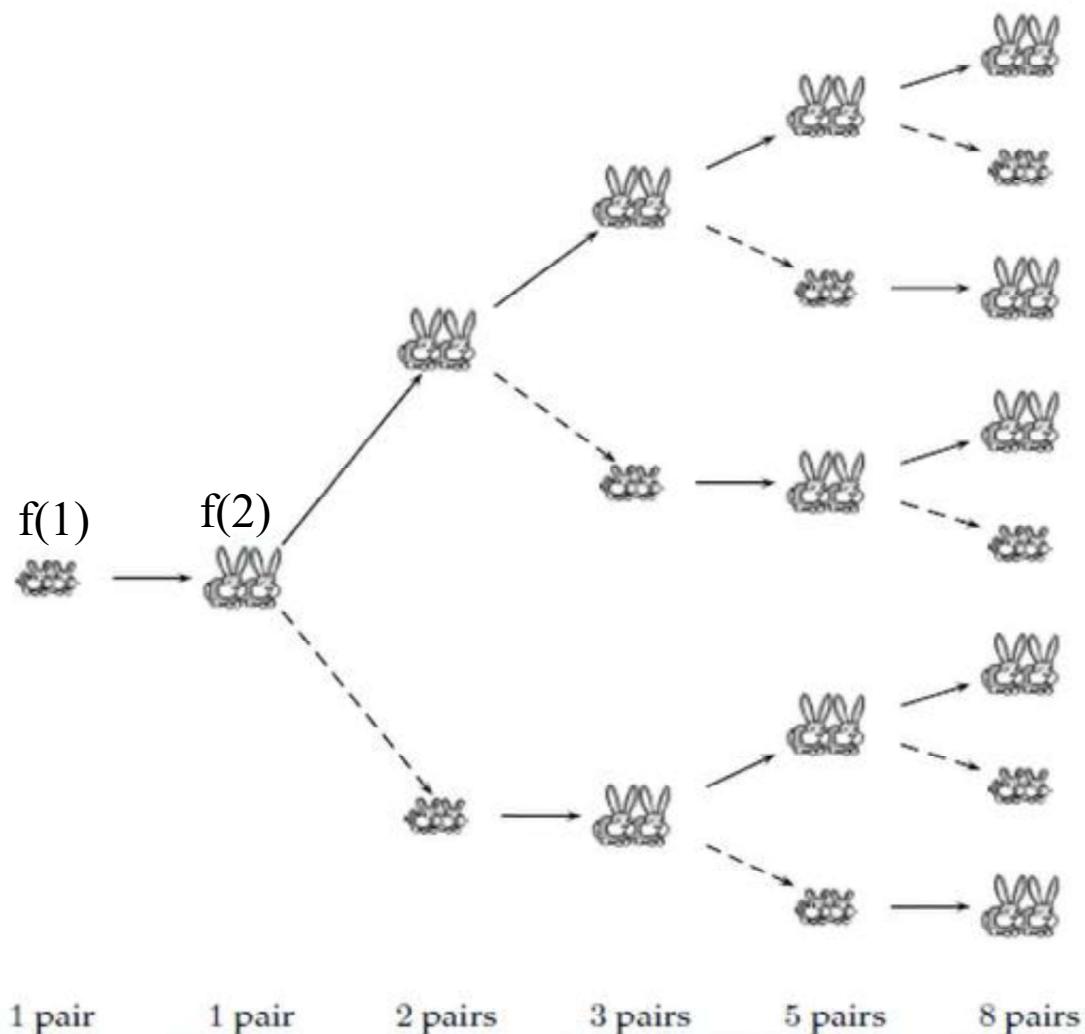
در یک جزیره یک جفت خرگوش نر و ماده نوزاد وجود دارد و مدل رشد جمعیت خرگوش ها به صورت زیر است:

- (الف) خرگوش ها یک ماه پس از تولد به سن بلوغ می رسند.
- (ب) یک ماه پس از رسیدن به سن بلوغ و از آن به بعد همه ماهه هر جفت خرگوش بالغ، یک جفت خرگوش دیگر تولید می کند.
- (ج) خرگوش ها هرگز نمی میرند.
- رابطه بازگشتی برای تعداد خرگوش ها در شروع ماه ۱۱ام :

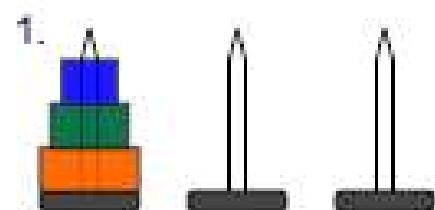
مسئله خرگوش ها

تعداد خرگوش ها در شروع ماه n ام $f(n)$

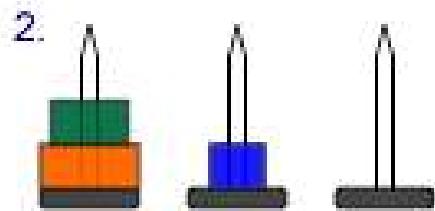
$$\begin{cases} f(n) = f(n - 1) + f(n - 2), n \geq 3 \\ f(1) = 1, f(2) = 1 \end{cases}$$



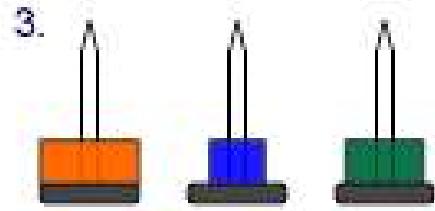
برج هانوی



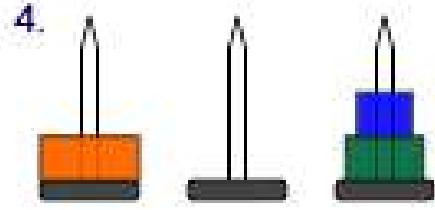
From Tower A to Tower B



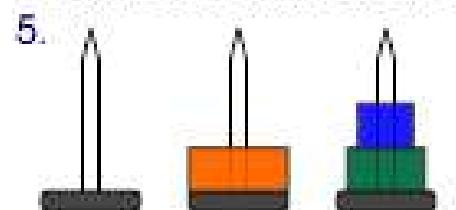
From Tower A to Tower C



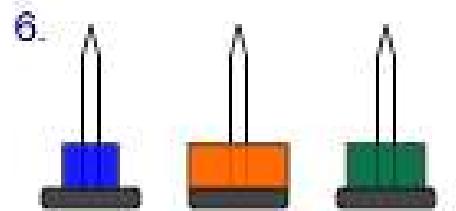
From Tower B to Tower C



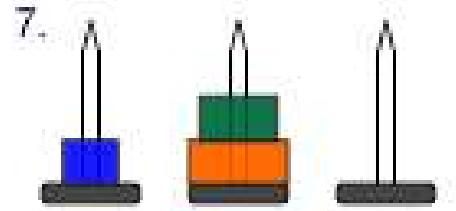
From Tower A to Tower B



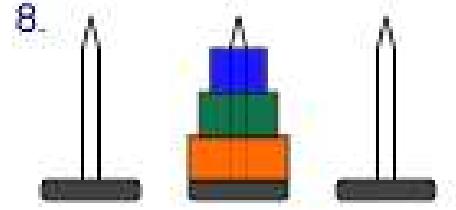
From Tower C to Tower A



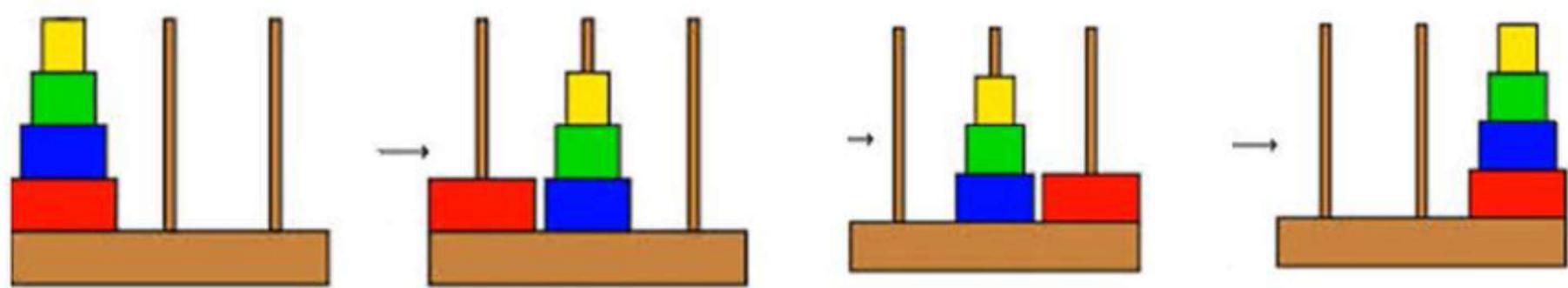
From Tower C to Tower B



From Tower A to Tower B



برج هانوی



$$a_n = a_{n-1} + 1 + a_{n-1}$$

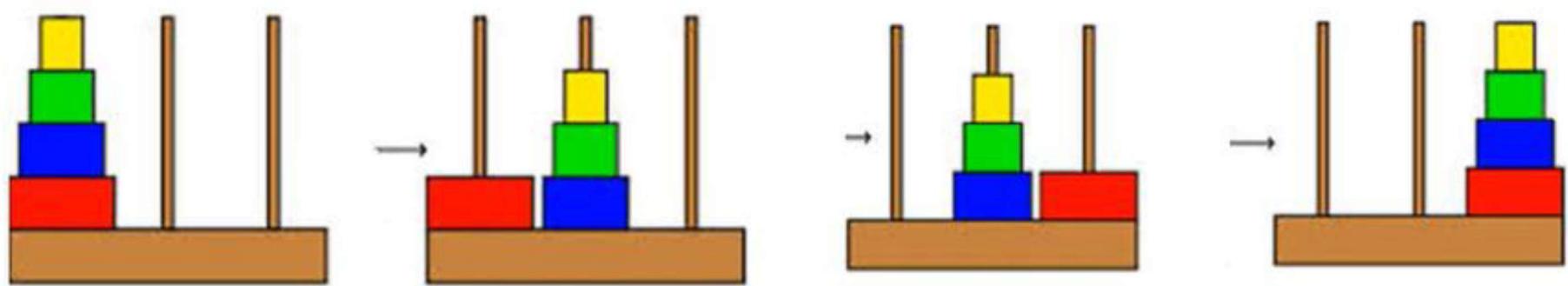
$$a_n = 2a_{n-1} + 1$$

$$a_1 = 1$$

$$\Rightarrow a_n = 2^n - 1$$

- ❖ سه تا برج داریم در برج اول ۴ مهره داریم.
- ❖ هدفمان این هست که این ۴ مهره را بیاوریم در برج آخر.
- ❖ مهره ها از بزرگ به کوچک چیده شدند.
- ❖ برای حرکت مهره ها از لوله وسطی کمک می گیریم.

برج هانوی



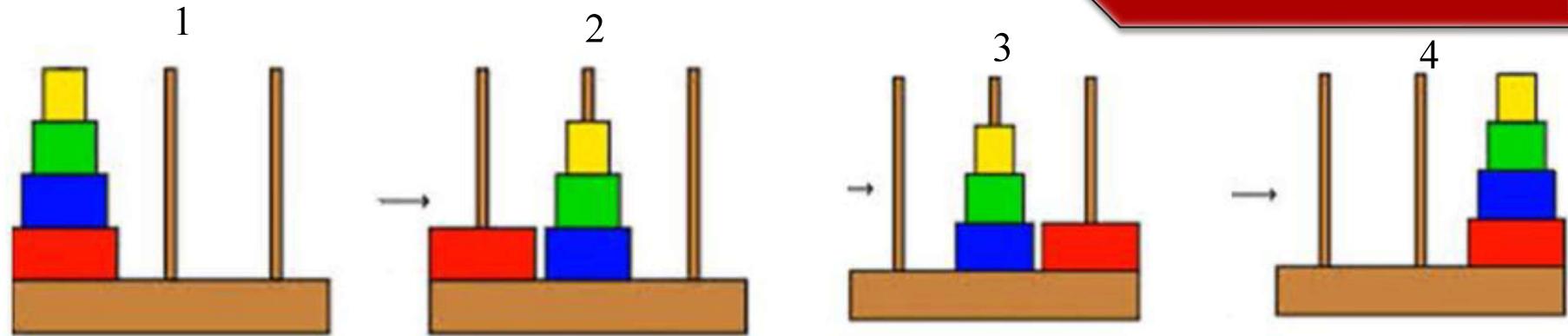
$$a_n = a_{n-1} + 1 + a_{n-1}$$

$$\begin{aligned} a_n &= 2a_{n-1} + 1 \\ a_1 &= 1 \end{aligned}$$

$$\Rightarrow a_n = 2^n - 1$$

- ❖ مهره بزرگ روی مهره کوچک نباید قرار بگیرد.
- ❖ هر بار فقط یک مهره را می توانیم جا به جا کنیم.
- ❖ مهره های آبی و سبز و زرد را با هم به لوله وسط انتقال می دهیم.
- ❖ سپس مهره قرمز را به لوله آخر انتقال می دهیم.

برج هانوی



$$a_n = a_{n-1} + 1 + a_{n-1}$$

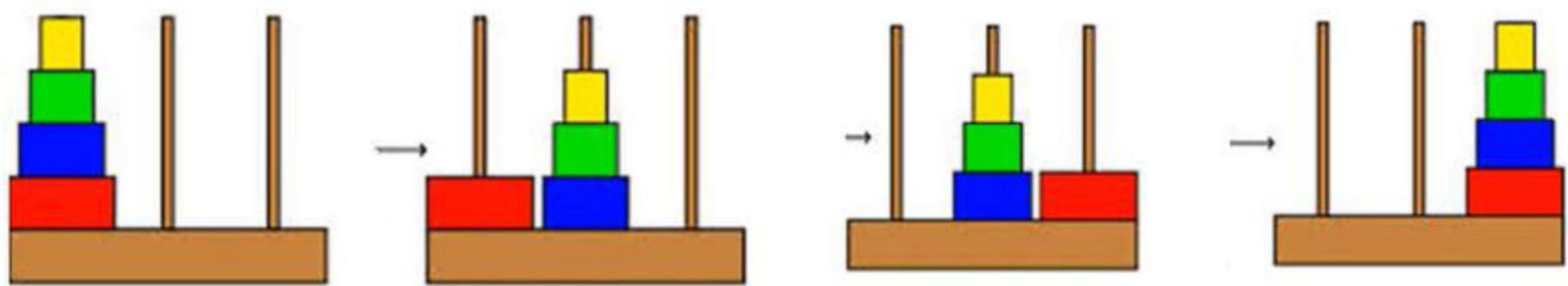
$$a_n = 2a_{n-1} + 1$$

$$a_1 = 1$$

$$\Rightarrow a_n = 2^n - 1$$

- ❖ در نهایت مهره های آبی و سبز و زرد را با هم به لوله سوم انتقال می دهیم.
- ❖ تعداد حرکت های مورد نیاز شکل ۱ هست.
- ❖ تعداد حرکت های مورد نیاز شکل ۲ هست.
- ❖ $+1$ به این معناست که یک مهره قرمز را به لوله سوم انتقال دادیم (شکل ۳)
- ❖ در انتها مجدد a_{n-1} مهره به لوله ۳ انتقال پیدا خواهد کرد.(شکل ۴)

برج هانوی



$$a_n = a_{n-1} + 1 + a_{n-1}$$

$$a_n = 2a_{n-1} + 1$$

$$a_1 = 1$$

$$\Rightarrow a_n = 2^n - 1$$

❖ اگر یک مهره داشته باشیم $a_n = 1$ می شود یعنی با یک حرکت مهره لوله اول را به لوله سوم انتقال می دهیم.

❖ به عنوان مثال اگر ۴ تا مهره داشته باشیم تعداد انتقال های مهره های ما برابر

$$a_n = 2^n - 1 = a_4 = 2^4 - 1 = 15$$

روش های حل رابطه های
بازگشتی

۱- تکرار با جایگذاری

۲- درخت بازگشت

۳- قضیه اصلی

۴- رابطه های بازگشتی همگن

مثال روش تکرار با جایگذاری

$$T(n) = n + T(n - 1)$$

$$T(1) = 1$$

$$\begin{aligned} T(n) &= n + T(n - 1) \\ &= n + (n - 1) + T(n - 2) \\ &= n + (n - 1) + (n - 2) + T(n - 3) \\ &\dots \\ &= n + (n - 1) + (n - 2) + \dots + 2 + T(1) \\ &= n + (n - 1) + (n - 2) + \dots + 2 + 1 \end{aligned}$$

$\sum_{i=1}^n i = \frac{n(n+1)}{2}$

این رابطه بازگشتی نشان دهنده جمع اعداد ۱ تا n

مثال

```
f(n)
{
    if(n>0)
    {
        f(n-1); T= (n-1)
        print(n); T= 1
        f(n-1); T= (n-1)
    }
}
```

این رابطه شبیه به رابطه برج هانوی است

مرتبه زمانی الگوریتم مقابل :

$$\begin{aligned}
 T(n) &= 2T(n-1) + 1 \longrightarrow 2T(n-1-1)+1 \\
 &= 2(2T(n-2) + 1)) + 1 = 2^2 T(n-2) + 2 + 1 \\
 &= 2^2 (2T(n-3) + 1)) + 2 + 1 = 2^3 T(n-3) + 2^2 + 2 + 1 \\
 &\dots \qquad \qquad \qquad \text{انقدر این رابطه را ادامه می دهیم تا به شرط توقف که } n=0 \text{ است بررسیم} \\
 &= 2^n T(n-n) + 2^{n-1} + \dots + 2^2 + 2 + 1 \\
 &= 2^{n-1} + \dots + 2^2 + 2 + 1 \\
 &= 2^n - 1
 \end{aligned}$$

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

مثال

$$\begin{cases} T(n) = \frac{1}{n} + T(n-1) \\ T(1) = 1 \end{cases}$$

$$\begin{aligned} T(n) &= \frac{1}{n} + T(n-1) \\ &= \frac{1}{n} + \frac{1}{n-1} + T(n-2) \\ &= \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + T(n-3) \\ &\dots \\ &= \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \dots + \frac{1}{2} + T(1) \\ &= \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \dots + \frac{1}{2} + \frac{1}{1} \\ &\approx \ln n \end{aligned}$$

روش درخت بازگشت (recursion tree)

به کمک این روش می‌توان رابطه‌های بازگشتی را حدس یا حل کرد.

در این روش نحوه جای‌گذاری یک عبارت بازگشتی و نیز مقدار ثابتی را که در هر سطح از آن عبارت به دست می‌آید نشان داده می‌شود. با جمع کردن مقادیر ثابت تمام سطوح، جواب بدست می‌آید.

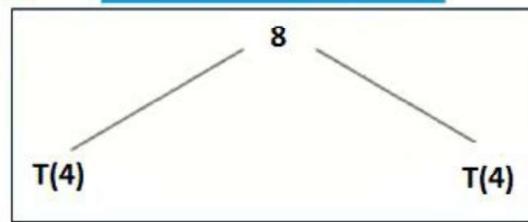
درخت‌های بازگشت زمانی که رابطه بازگشتی، زمان اجرای یک الگوریتم تقسیم و حل را توصیف می‌کند مفید هستند.

رسم درخت بازگشتی

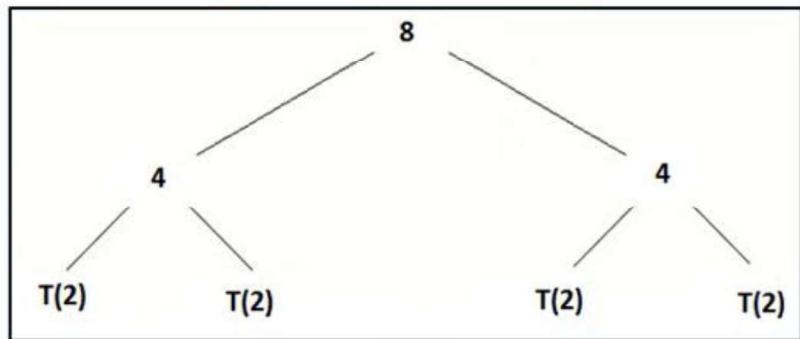
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(1) = 1$$

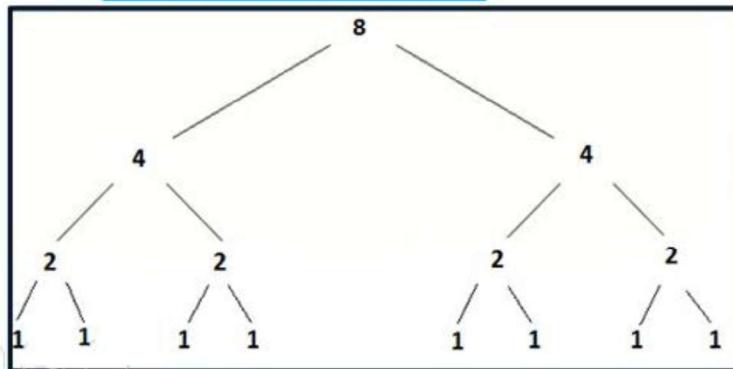
$$T(8) = 2T(4) + 8$$



$$T(4) = 2T(2) + 4$$



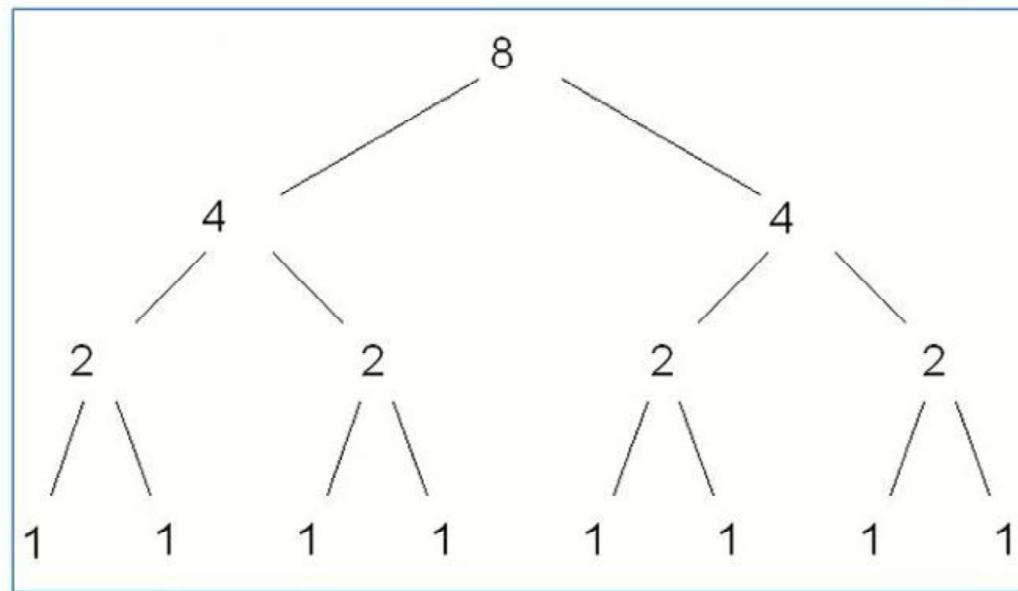
$$T(2) = 2T(1) + 2$$



رسم درخت بازگشتی

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

$$T(1) = 1$$



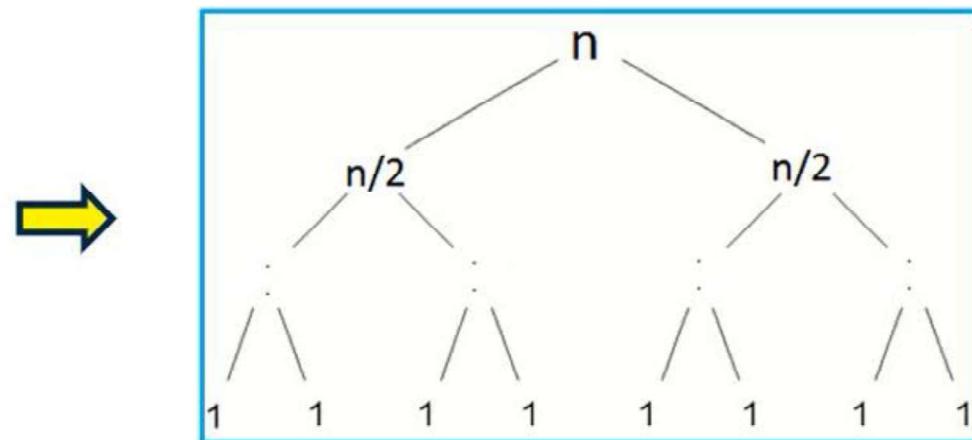
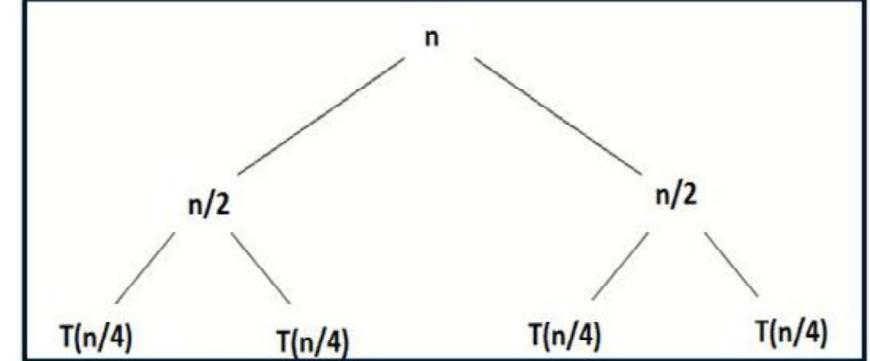
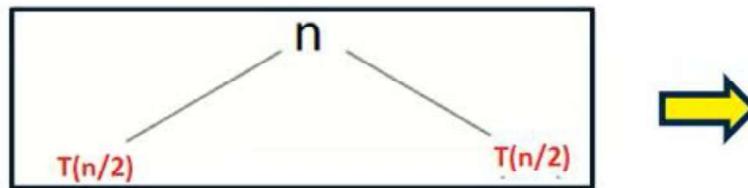
هزینه:

$$(8) + (4 + 4) + (2 + 2 + 2 + 2) + (1 + 1 + 1 + 1 + 1 + 1 + 1 + 1) = 4 \times 8 = 32$$

مثال

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

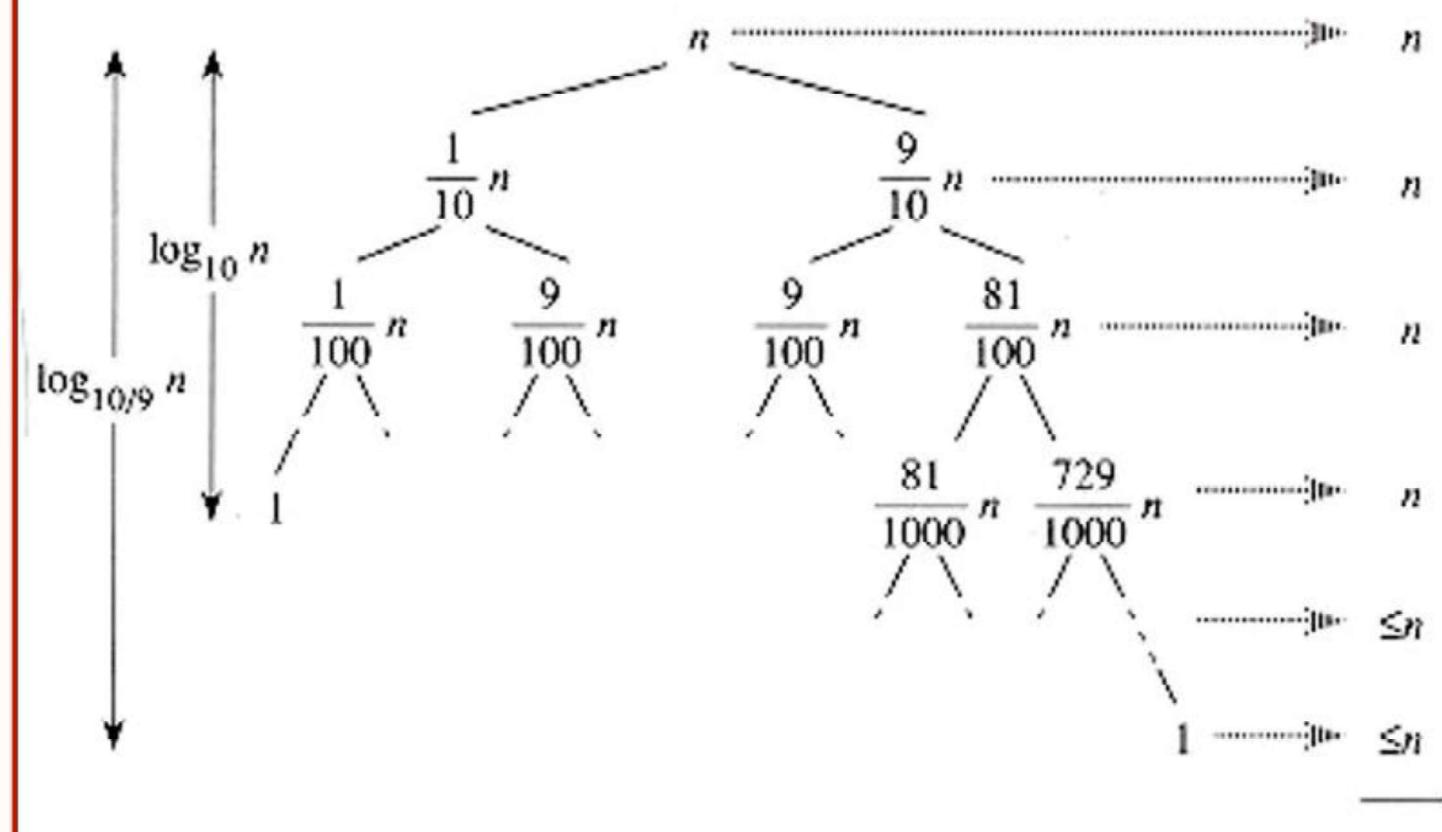
حل رابطه بازگشتی :



$$(lg n + 1) \times n = n \lg n + n$$

مثال

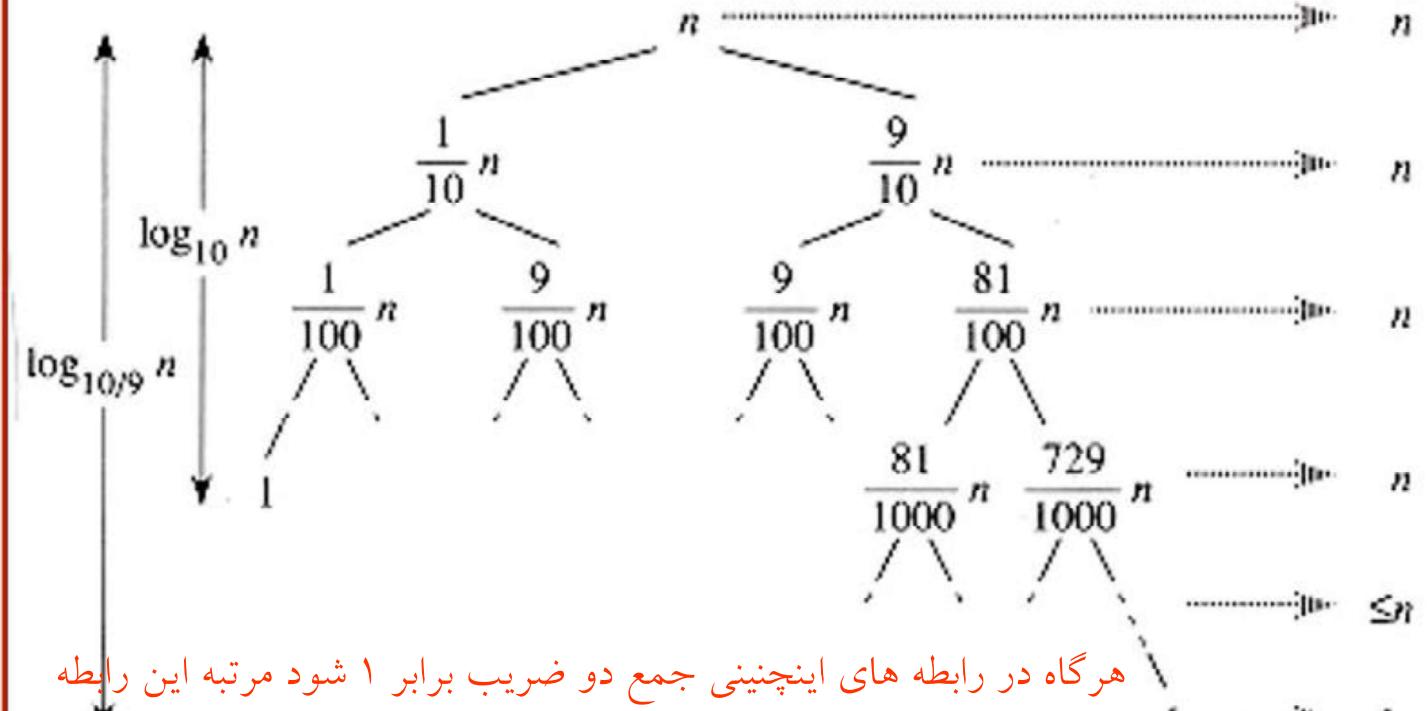
$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn$$



$\Theta(n \lg n)$

مثال

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn$$

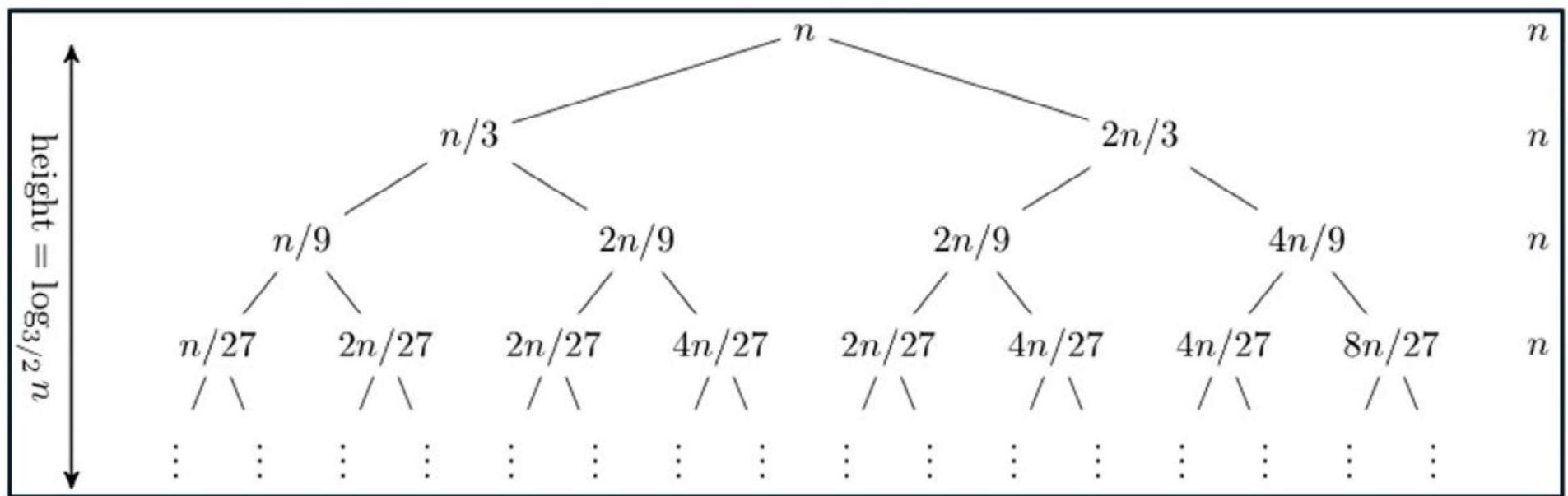


اگر جمع ضریب ها کمتر از 1 شود مرتبه می شود n

$\Theta(n \lg n)$

مثال

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$



فرمول

$$T(n) = T\left(\frac{n}{a}\right) + T\left(\frac{n}{b}\right) + cn$$



$$n \sum_{i=0}^h \left(\frac{1}{a} + \frac{1}{b}\right)^i$$

ارتفاع درخت: حداقل مقدار بین ارتفاع سمت چپ (\log_b^n) و ارتفاع سمت راست (\log_a^n)

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$



$$\begin{aligned} T(n) &= n \sum_{i=0}^h \left(\frac{1}{10} + \frac{9}{10}\right)^i \\ &= n \sum_{i=0}^h (1)^i = \theta(n \lg n) \end{aligned}$$

مثال

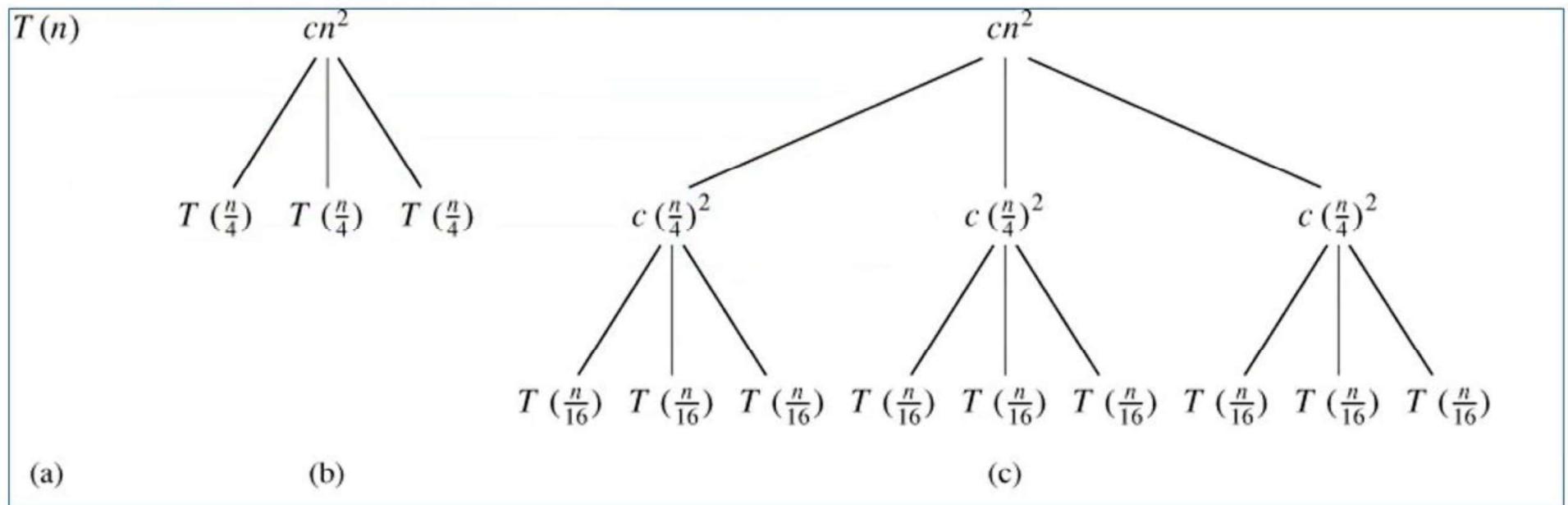
$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$$

$$T(n) \leq n \sum_{i=0}^{\log_{10/7} n} \left(\frac{1}{5} + \frac{7}{10}\right)^i = n \sum_{i=0}^{\log_{10/7} n} \left(\frac{9}{10}\right)^i$$

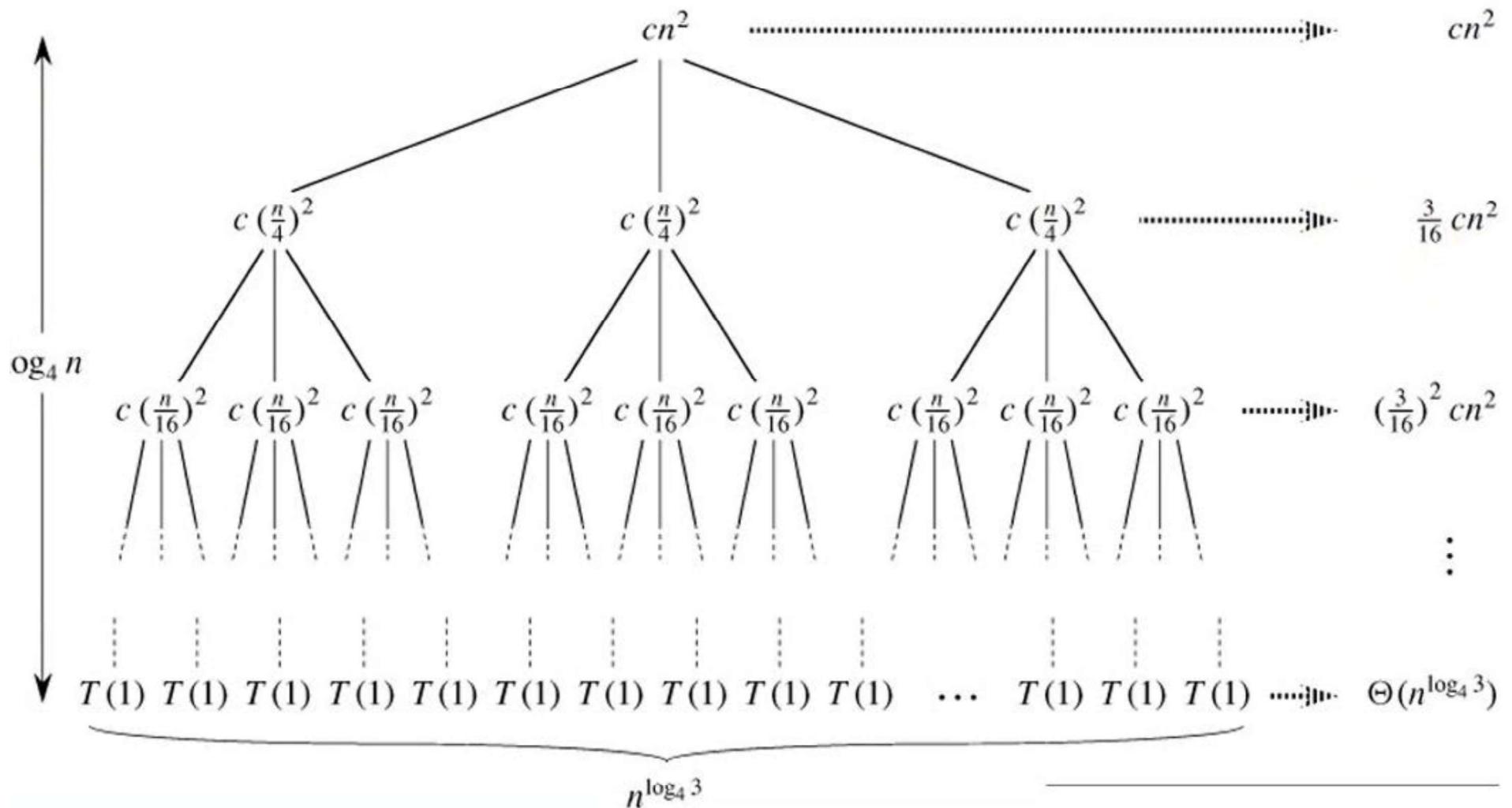
اگر جمع ضریب ها کمتر از 1 شود مرتبه می شود n

مثال

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2$$



مثال



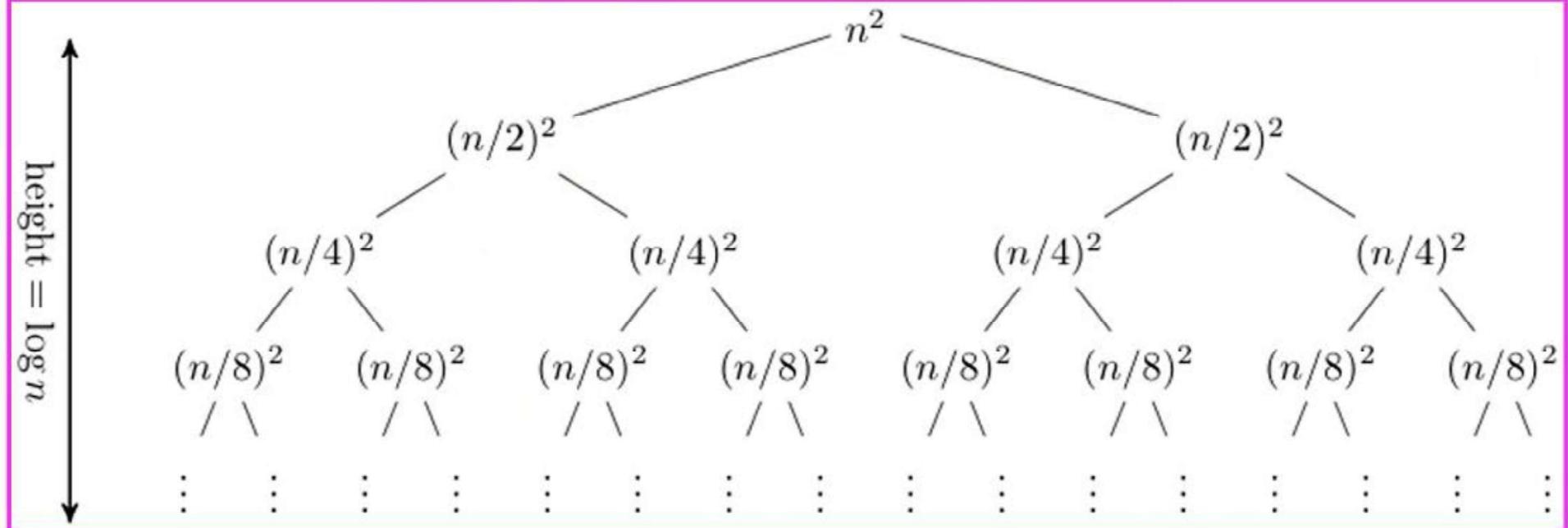
مثال

$$\begin{aligned}
 T(n) &= cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
 &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}).
 \end{aligned}$$

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
 &= O(n^2).
 \end{aligned}$$

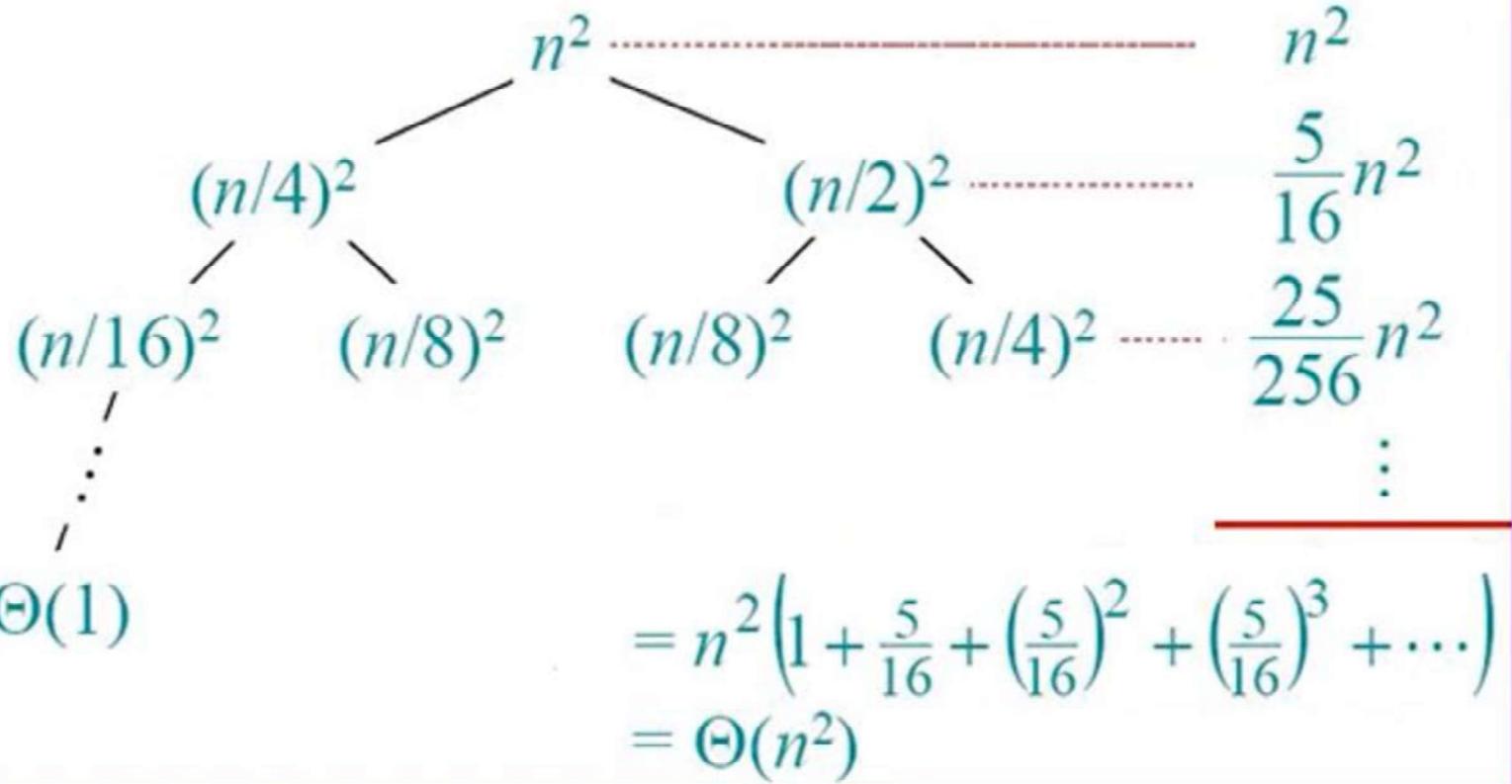
مثال

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



مثال

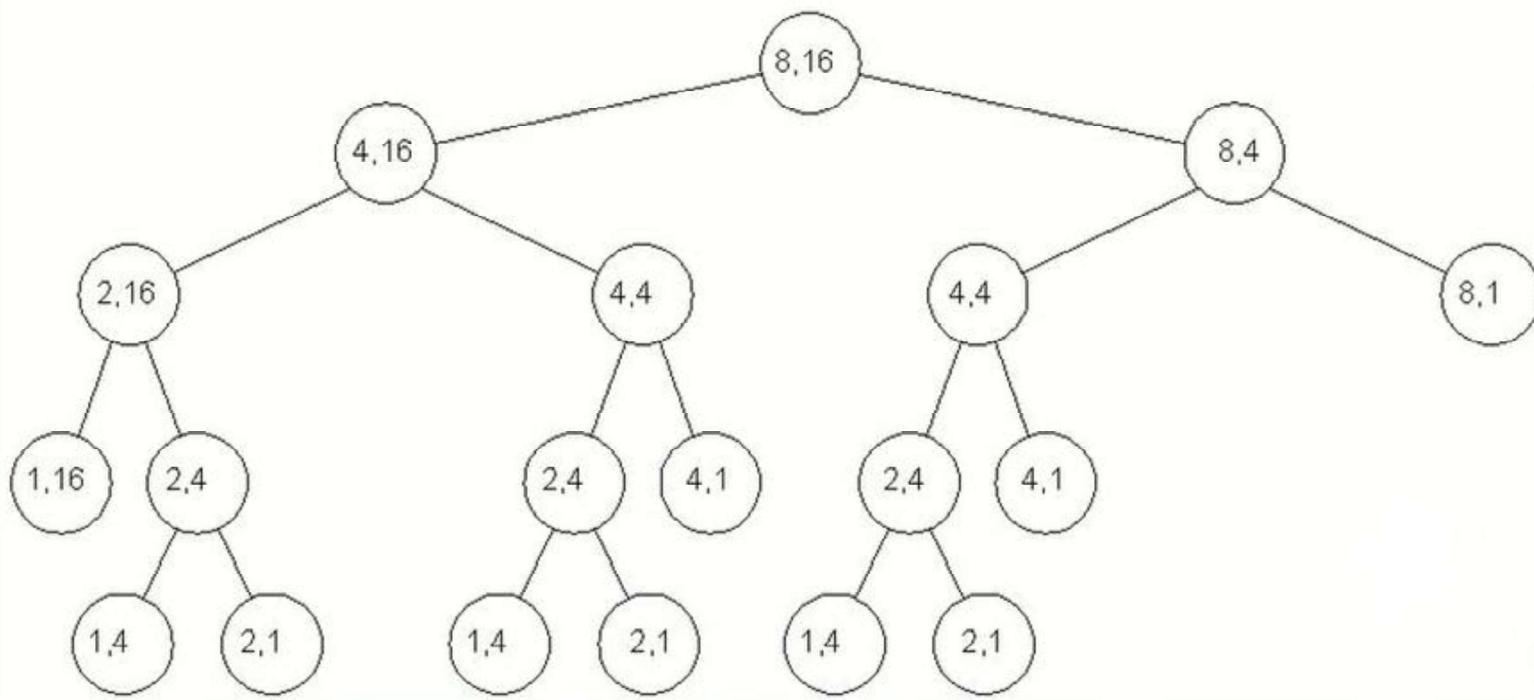
$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2$$



رابطه بازگشتی دو متغیره

$$T(n,k) = T\left(\frac{n}{2}, k\right) + T\left(n, \frac{k}{4}\right) + kn$$

رسم درخت بازگشت : $T(8,16)$



ارتفاع:

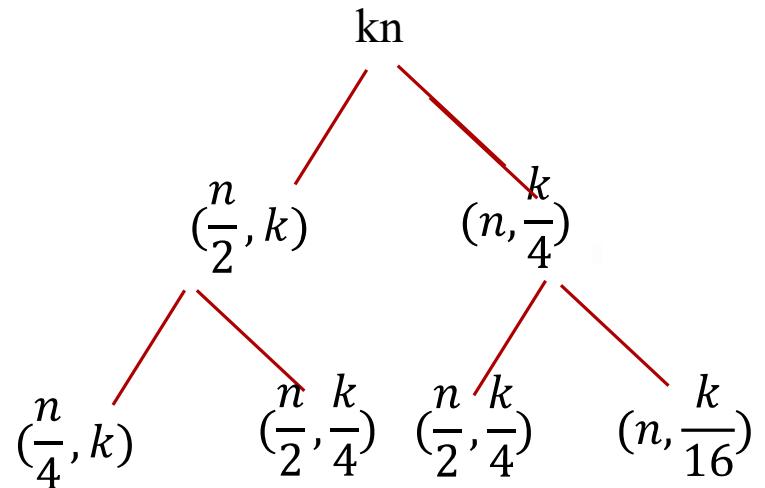
$$\log_2 n + \log_4 k - 1$$

$$T(n, k) = T\left(\frac{n}{2}, k\right) + T\left(n, \frac{k}{4}\right) + kn$$

$$= nk + \left(\frac{n}{2} \times k + n \times \frac{k}{4} \right) + \left(\frac{n}{4} \times k + \frac{n}{2} \times \frac{k}{4} + \frac{n}{2} \times \frac{k}{4} + n \times \frac{k}{16} \right) + \dots$$

$$= nk + \frac{3}{4} nk + \frac{9}{16} nk + \dots$$

$$= nk \left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \dots \right) \approx \Theta(nk)$$



رابطه بازگشتی دو متغیره

$$T(n, k) = T\left(n_1, \left\lfloor \frac{k}{2} \right\rfloor\right) + T\left(n_2, \left\lfloor \frac{k}{2} \right\rfloor\right) + nk$$

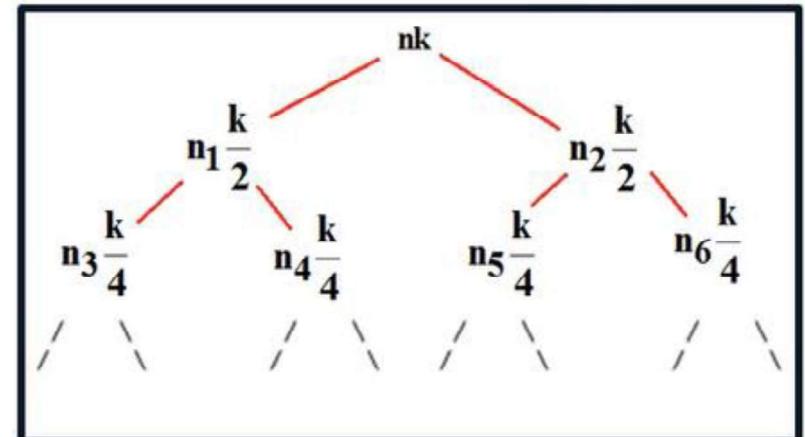
$$T(1, k) = T(k, 1) = 1$$

$$nk + \left(n_1 \frac{k}{2} + n_2 \frac{k}{2}\right) + \left(n_3 \frac{k}{4} + n_4 \frac{k}{4} + n_5 \frac{k}{4} + n_6 \frac{k}{4}\right) + \dots$$

$$= nk + (n_1 + n_2) \frac{k}{2} + (n_3 + n_4 + n_5 + n_6) \frac{k}{4} + \dots$$

$$= nk + n \frac{k}{2} + (n_1 + n_2) \frac{k}{4} + \dots$$

$$= nk + n \frac{k}{2} + n \frac{k}{4} + \dots = nk \left(1 + \frac{1}{2} + \frac{1}{4} + \dots\right) \approx nk$$



$$(n_1 + n_2 = n)$$

$$(n_3 + n_4 = n_1, n_5 + n_6 = n_2)$$

قضیه اصلی

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

اگر داشته باشیم: $a \geq 1, b > 1$

$$T(n) = \begin{cases} \Theta(n^{\log_b^a}) & f(n) < n^{\log_b^a} \\ \Theta(f(n) \cdot \lg n) & f(n) = n^{\log_b^a} \\ \Theta(f(n)) & f(n) > n^{\log_b^a} \end{cases}$$

مثال

$$T(n) = 4T\left(\frac{n}{2}\right) + \lg n$$

$$a = 4$$

$$b = 2$$

$$f(n) = \lg n$$

$$\lg n < n^{\log_2^4}$$

$$\theta(n^2)$$

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

مثال

$$n \lg n > n^{\log_4^3}$$

$$\theta(n \lg n)$$

مثال

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

$$1 = n^{\log_{3/2} 1}$$

$$\theta(\lg n)$$

نکته

در قضیه اصلی، اگر $\frac{f(n)}{n^{\log_b^a}} < n^\varepsilon$ باشد، یعنی $f(n)$ به صورت چند جمله‌ای از $n^{\log_b^a}$ زرگتر نباشد،

اگر $f(n)$ از مرتبه $T(n) = n^{\log_b^a \cdot \lg^k n}$ باشد، آنگاه مرتبه $f(n)$ برابر است با:

$$n^{\log_b^a \cdot \lg^{k+1} n}$$

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

مثال

$$\frac{n \lg n}{n} = \lg n < n^\varepsilon$$

$$a = 2$$

$$b = 2$$

$$f(n) = n \lg n$$

$$T(n) = \Theta(n \lg^2 n)$$

مثال

$$T(n) = T\left(\frac{2n}{3}\right) + (\lg n)^2$$

$$T(n) = \Theta(n^0 (\lg n)^{2+1}) = \Theta((\lg n)^3)$$

$$T(n) = T\left(\frac{2n}{3}\right) + (\lg n)^2$$

مثال

مرتبه اجرایی رابطه بازگشته مشخص کنید.

فرض: $n = 2^m$

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

$$T(2^m) = S(m) \Rightarrow T\left(2^{\frac{m}{2}}\right) = S\left(\frac{m}{2}\right)$$

$$S(m) = S\left(\frac{m}{2}\right) + 1 \quad \Rightarrow$$

$$S(m) = \Theta(\lg m)$$

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

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

مثال

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

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

$$\theta(m \lg m)$$

$$\theta(\lg n \cdot \lg \lg n)$$

$$T(n) = 4T(\sqrt{n}) + 1$$

$$n = 2^m$$

مثال

$$T(2^m) = 4T(2^{m/2}) + 1$$

$$S(m) = 4S\left(\frac{m}{2}\right) + 1$$

$$S(m) = \theta(m^{\log_2 4}) = \theta(m^2)$$

$$T(n) = \theta(\lg n)^2$$

رابطه های بازگشتی همگن

برای حل روابط بازگشتی همگن مرتبه دوم با ضرایب ثابت، ابتدا معادله مشخصه آن را پیدا می کنیم.

جواب بعد از حل این معادله با فرض داشتن:

$$c_1 r_1^n + c_2 r_2^n : r_2, r_1$$

$$c_1 r^n + c_2 n r^n : r$$

مثال

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

$$T(0) = 2$$

$$T(n) - T(n-1) - 2T(n-2) = 0$$

$$T(1) = 7$$

$$r^2 - r - 2 = 0 \Rightarrow r_1 = 2, r_2 = -1$$

$$T(n) = c_1 2^n + c_2 (-1)^n$$
 جواب کلی:

$$T(0) = 2 \Rightarrow 2 = c_1 2^0 + c_2 (-1)^0 \Rightarrow c_1 + c_2 = 2$$

$$T(1) = 7 \Rightarrow 7 = c_1 2^1 + c_2 (-1)^1 \Rightarrow 2c_1 - c_2 = 7$$

$$\Rightarrow c_1 = 3, c_2 = -1$$

$$T(n) = 3 \times 2^n - (-1)^n$$

مثال

$$T(n+2) = 4T(n+1) - 4T(n)$$

$$T(n+2) - 4T(n+1) + 4T(n) = 0$$

$$r^2 - 4r + 4 = 0 \Rightarrow r_1 = 2, r_2 = 2$$

$$T(n) = c_1 2^n + c_2 n 2^n$$
 جواب کلی:

$$T(0) = 1 \Rightarrow c_1 \times 2^0 + c_2 \times 0 \times 2^0 = 1$$

$$T(1) = 3 \Rightarrow c_1 \times 2^1 + c_2 \times 1 \times 2^1 = 3$$

$$\Rightarrow c_1 = 1$$

$$\Rightarrow 2c_1 + 2c_2 = 3$$

$$c_1 = 1, c_2 = \frac{1}{2}$$

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

مثال

$$T(n) = 3T(n-1) + 4T(n-2)$$

$$T(0) = 0, T(1) = 1$$

$$r^2 - 3r - 4 = 0$$

$$T(n) = c_1 4^n + c_2 (-1)^n$$

$$T(n) = \theta(4^n)$$

روش تقسیم و حل

(Divide-and-Conquer)

مراحل :

- ۱- تقسیم نمونه ای از یک مسئله به یک یا چند نمونه کوچکتر
- ۲- حل نمونه های کوچکتر
- ۳- ترکیب حل نمونه های کوچکتر برای بدست آوردن حل نمونه اولیه (در صورت نیاز)

دلیل اینکه می گوئیم "در صورت نیاز" این است که در بعضی الگوریتم ها مانند جستجوی دودویی نمونه فقط به یک نمونه کوچکتر کاهش می یابد و نیازی به ترکیب حل ها نیست.

هنگام طراحی الگوریتم های تقسیم و حل معمولًاً آن را به صورت یک روال بازگشتی می نویسند.

در صورت امکان باید در مورد زیر از روش تقسیم و حل پرهیز کرد:

نمونه ای با اندازه n به دو یا چند نمونه تقسیم می شود که اندازه آن ها نیز تقریبا n است. افزایش در این حالت به یک الگوریتم زمانی نمایی منجر می شود.

مثال

الگوریتمی هر ورودی مسئله به اندازه n را به 2 بخش کم و بیش مساوی تقسیم می کند. زیر مسئله ها را به صورت بازگشتی حل و سپس با هزینه خطی حاصل این دو را با هم ترکیب کرده و جواب مسئله را به دست می آورد.

رابطه بازگشتی :

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + O(n) \\ \Rightarrow T(n) &= O(n \lg n) \end{aligned}$$

جستجوی دودویی

پیدا کردن عدد ۱۰ در ارایه مرتب:

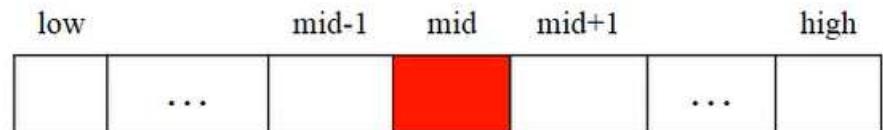
۱	۲	۳	۴	۵	۶	۷	۸	۹
5	9	10	20	35	50	60	70	75



جستجوی دودویی

الگوریتم جستجوی دودویی

```
bsearch (a[ ] , x , low , high ){  
    if (low <=high ) {  
        mid = ( low+high ) / 2;  
        if ( x < a[mid] )  
            bsearch( a , x , low , mid-1 );  
        else if ( x > a[mid] )  
            bsearch (a , x , mid+1 , high );  
        else  
            return mid;  
    }  
    return -1;  
}
```



$$T(n) = T\left(\frac{n}{2}\right) + 1 \rightarrow O(\lg n)$$

6

مثال

1	4	4	7	7	8	11	19	21	23	24	30
---	---	---	---	---	---	----	----	----	----	----	----

mid

1	4	4	7	7	8	11	19	21	23	24	30
---	---	---	---	---	---	----	----	----	----	----	----

mid

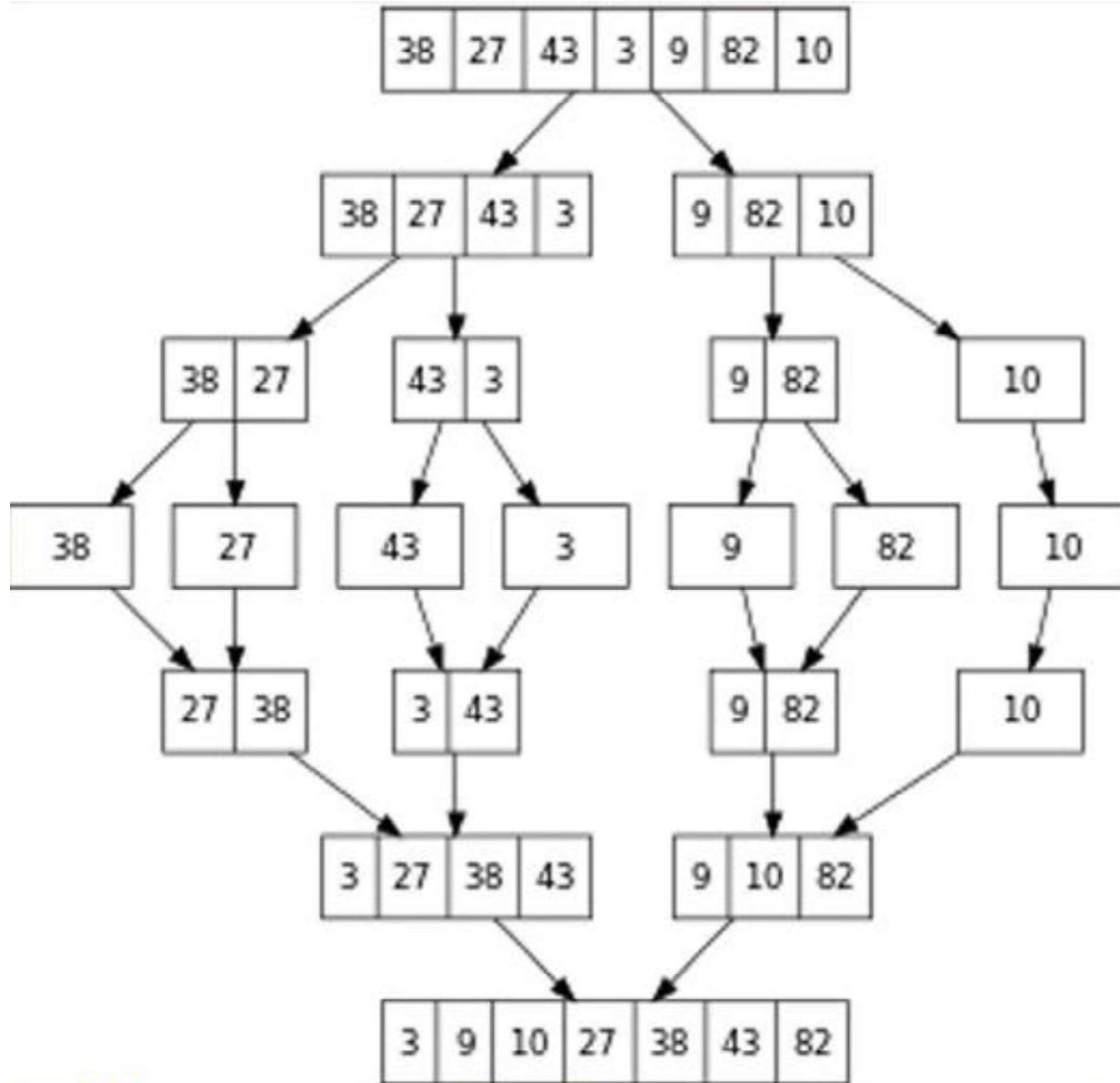
1	4	4	7	7	8	11	19	21	23	24	30
---	---	---	---	---	---	----	----	----	----	----	----

mid

1	4	4	7	7	8	11	19	21	23	24	30
---	---	---	---	---	---	----	----	----	----	----	----

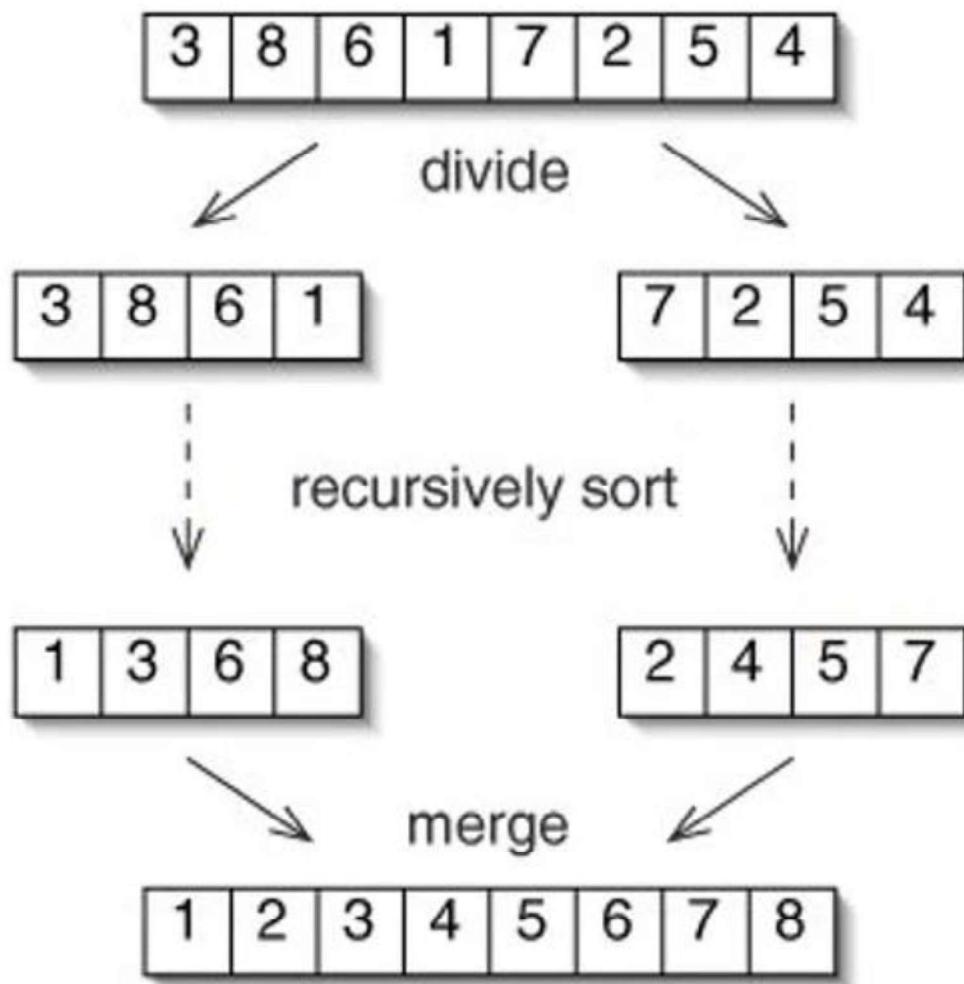
↑

مرتب سازی ادغامی (Merge Sort)



مرتب سازی ادغامی

(Merge Sort)



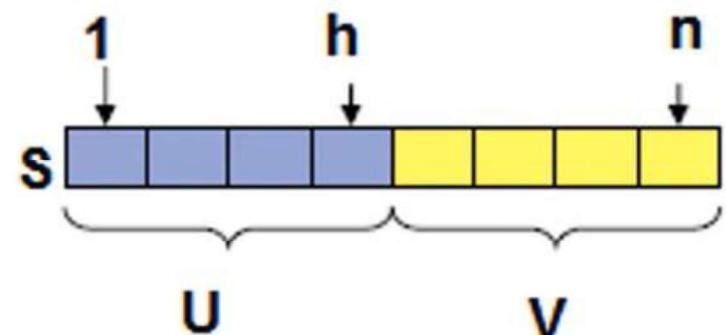
الگوریتم

mergesort (S[] , n)

```

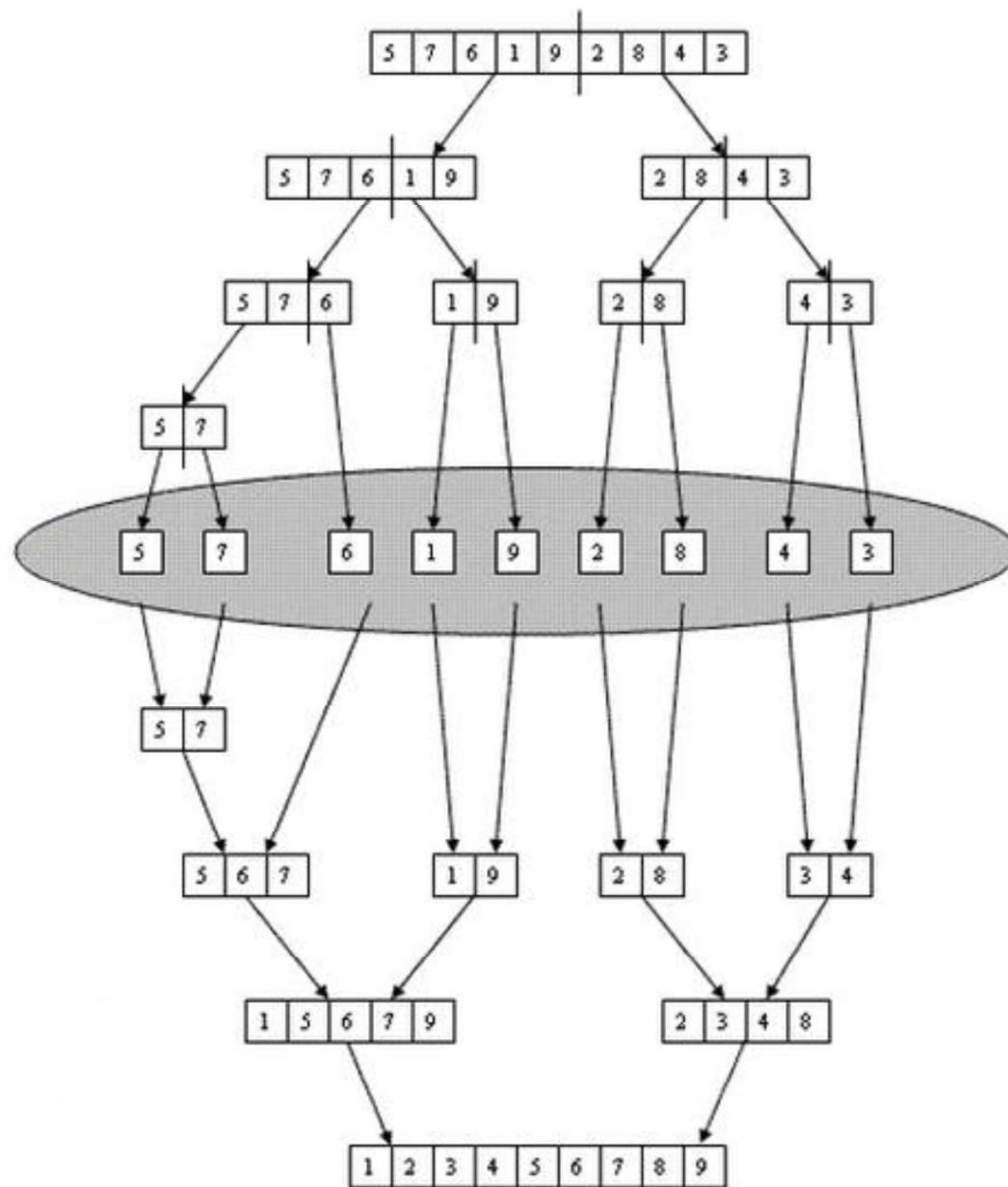
{
    h = [n/2];
    m = n - h;
    if (n > 1)
    {
        copy S[1..h] to U[1..h];
        copy S[h + 1..n] to V[1..m];
        mergesort ( U , h );
        mergesort ( V , m );
        merge ( U , h , V , m , S);
    }
}

```



$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + n - 1 \\
 \Rightarrow T(n) &= n \lg n - (n - 1) \\
 &\in \theta(n \lg n)
 \end{aligned}$$

الگوریتم



بھار 1403

پریسا مرادی

101

مرتب سازی سریع (Quick Sort)

مرتب سازی سریع (Quick sort)

QuickSort (A , p , r)

```
{  
    if ( p < r )  
    {  
        q = Partition (A , p , r) ;  
        QuickSort (A , p , q-1) ;  
        QuickSort (A , q+1, r ) ;  
    }  
}
```



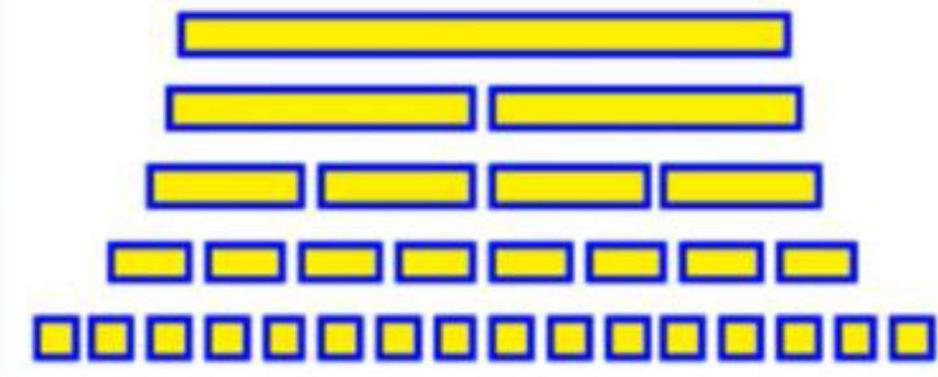
مرتب سازی سریع
(Quick Sort)

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

مرتب سازی سریع (Quick Sort)

عملکرد مرتب سازی سریع

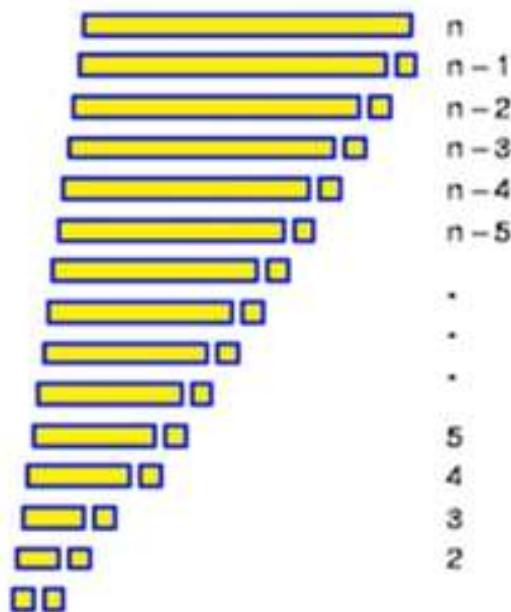
حالت خوب و میانگین :



$$T(n) = T(n/2) + T(n/2) + O(n) \rightarrow O(n \log n)$$

بدترین حالت مرتب سازی سریع

در صورتی که اولین عنصر یا آخرین عنصر را به عنوان محور انتخاب کنیم، مرتب سازی سریع برای یک آرایه **مرتب**، بدترین عملکرد را خواهد داشت.



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

ضرب ماتریس های استراسن

استراسن الگوریتمی را ارائه داد که پیچیدگی آن، از لحاظ ضرب (و همچنین از لحاظ جمع و تفریق) بهتر از پیچیدگی درجه سوم است.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 10 & 8 \end{bmatrix}$$

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} A_{00} * B_{00} + A_{01} * B_{10} & A_{00} * B_{01} + A_{01} * B_{11} \\ A_{10} * B_{00} + A_{11} * B_{10} & A_{10} * B_{01} + A_{11} * B_{11} \end{bmatrix}$$

strassen (n , A , B , C)

{

if ($n \leq \text{threshold}$)

 compute $C = A \times B$ using the standard algorithm;

else {

 partition A into four submatrices $A_{11}, A_{12}, A_{21}, A_{22}$;

 partition B into four submatrices $B_{11}, B_{12}, B_{21}, B_{22}$;

 compute $C = A \times B$ using Strassen's Method;

}

}

مقدار threshold نقطه ای است که در آن احساس می شود که استفاده از الگوریتم ضرب استاندارد بهتر از فراخوانی بازگشتی روال Strassen است.

برنامه نویسی پویا (Dynamic Programming)

در روش پویا ابتدا نمونه های کوچکتر را حل کرده و نتایج را ذخیره می کنیم و بعداً هرگاه به یکی از آن ها نیاز پیدا شد به جای محاسبه دوباره، کافی است آن را بازیابی کنیم.

در برنامه نویسی پویا از آرایه‌ای (جدولی) استفاده می شود.

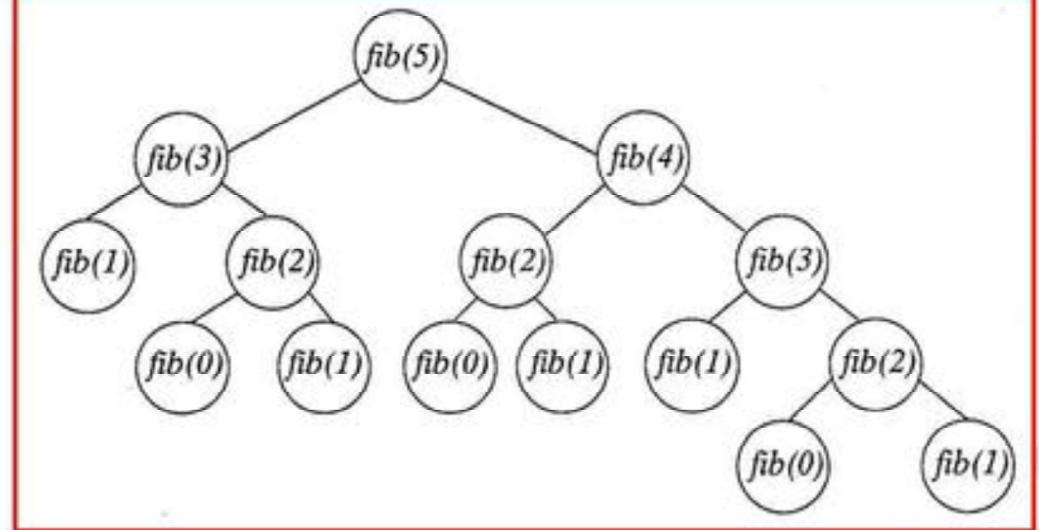
برنامه نویسی پویا از این لحاظ که نمونه را به نمونه های کوچکتر تقسیم می کند مشابه روش تقسیم و حل است.

تعدادی از الگوریتم هایی که به روش برنامه نویسی پویا حل می شوند:

- ۱- دنباله فیبوناچی
- ۲- ضریب دو جمله ای
- ۳- ضرب زنجیره ای ماتریس ها
- ۴- درخت های جستجوی دودویی بهینه
- ۵- کوله پشتی صفر و یک
- ۶- فلوبید
- ۷- فروشنده دوره گرد

الگوریتم بازگشتی برای محاسبه جمله n ام دنباله فیبوناچی

```
fib ( n){  
    if (n <= 1)  
        return n;  
    else  
        return fib(n-1) + fib(n-2);  
}
```



تعداد جملات محاسبه شده توسط الگوریتم بازگشتی بالا، برای تعیین جمله n ام فیبوناچی، بزرگتر از $2^{n/2}$ است.

با توجه به درخت، الگوریتم برای تعیین $\text{fib}(n)$ تعداد جملات زیر را محاسبه می‌کند:

n	0	1	2	3	4	5	6	...
تعداد جملات محاسبه شده	1	1	3	5	9	15	25	...

```

f[0] = 0;
f[1] = 1;
for (i = 2; i <= n; i++)
    f[i] = f[i-1] + f[i-2];
    
```

تعداد جملات محاسبه شده برای محاسبه فیبوناچی n :

$$k(n) = k(n-1) + k(n-2) + 1$$

مسئله کوله پشتی صفر و یک

(0-1 knapsack)

دزدی وارد یک جواهر فروشی شده و می خواهد قطعه هایی که دارای ارزش و وزن معینی هستند را طوری در کوله پشتی خود قرار دهد که بیشترین سود حاصل شود.

البته وزن قطعه ها از یک حد مشخص نباید بیشتر شود، چون کوله پشتی پاره خواهد شد.
اگر قطعه ها به گونه ای باشند که یا انتخاب می شوند و یا نه، به آن مسئله کوله پشتی صفر و یک می گویند. و اگر دزد بتواند هر کسری از قطعه ها را بردارد، به آن مسئله کوله پشتی کسری می گویند.

کوله پشتی صفر و یک : شمش های طلا و نقره

کوله پشتی کسری: کیسه های حاوی خاک طلا و نقره .

روش حل کوله پشتی:

صفر و یک : پویا

کسری : حریصانه

روش پویا برای حل مسئله کوله پشتی 0/1

$$P[n][w] = \begin{cases} \text{maximum}(P[n-1][w], P_n + P[n-1][w - w_n]) & w_n \leq W \\ P[n-1][W] & w_n > W \end{cases}$$

وزن item_i با w_i و ارزش آن با P_i نشان داده می شود.

تنها عناصر مورد نیاز در سطر (n-1) ام، آنها یی هستند که برای محاسبه $P[n][w]$ به کار می روند. که عبارتند از: $P[n-1][w]$ و $P[n-1][w - w_n]$

مثال: با فرض $w = 30$ ، سود بهینه را بدست آورید.

قطعه	ارزش(دلار)	وزن(پوند)
1	50	5
2	60	10
3	140	20

باید $P[3][30]$ را بدست آوریم.

$$P[n][w] = \max(P[n-1][w], P_n + P[n-1][w - w_n])$$

$$P[3][30] = \max \begin{cases} P[2][30] \\ P_3 + P[2][30 - w_3] = 140 + P[2][10] \end{cases}$$