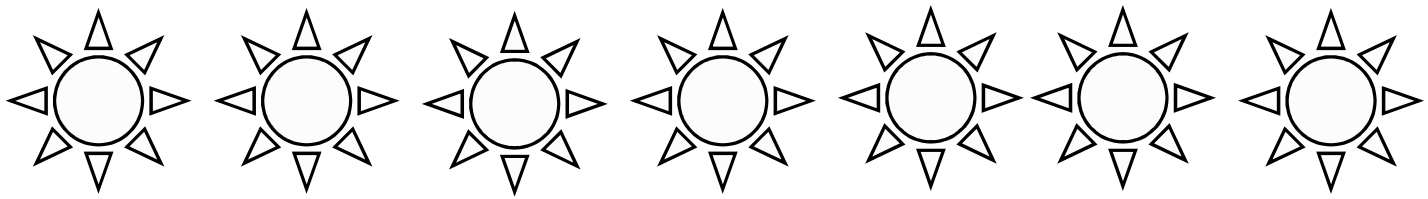
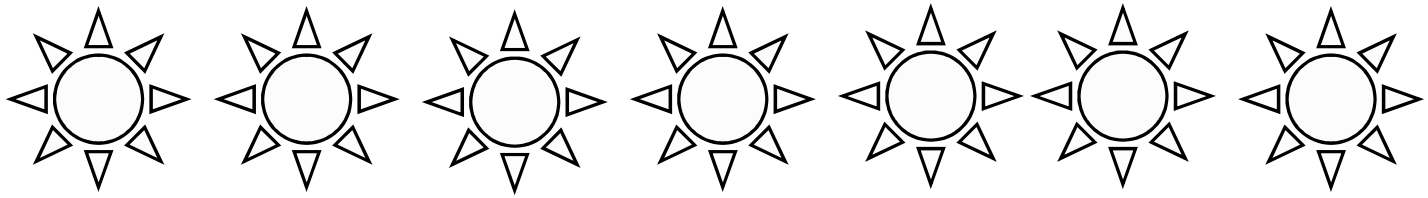


طراحی الگوریتم



2019

M. Damrudi

فصل اول

طراحی الگوریتم

به طور معمول برای حل یک مسأله مشخص بیش از یک الگوریتم وجود دارد.
برخی از این الگوریتم ها کاراتر می باشند و الگوریتمی انتخاب می شود که بیشترین کارایی را داشته باشد.
چه مواردی باید بررسی شود؟

- ✓ آیا الگوریتمی که ارائه می شود درست است یا خیر؟
- ✓ چگونه الگوریتم های مربوط به یک مسأله مقایسه می شوند؟
- ✓ کارایی (efficiency) هر الگوریتم چگونه تعیین می شود؟

مطالعه الگوریتم

مطالعه الگوریتم

۱. طراحی الگوریتم ✓
۲. اثبات درستی یا معتبرسازی الگوریتم
۳. بیان یا پیاده سازی الگوریتم
۴. تحلیل الگوریتم ✓

تعریف الگوریتم

❖ الگوریتم مجموعه‌ای از دستورالعمل‌ها است که اگر به ترتیب دنبال شوند موجب حل مسأله می‌گردند. ترتیب مراحل و شرط خاتمه عملیات باید کاملا مشخص باشد.

❖ خصوصیات هر الگوریتم:

- ✓ ورودی: می تواند داشته باشد یا نداشته باشد.
- ✓ خروجی: باید حداقل یک خروجی داشته باشد.
- ✓ دارای ترتیب (توالی) است.
- ✓ واضح و صریح (بدون ابهام) باشد.
- ✓ محدود است.

❖ طراحی الگوریتم: منظور از طراحی الگوریتم کشف الگوریتم، ایجاد، تعیین اعتبار، آنالیز و ارزیابی الگوریتم برای یک مسأله می باشد.

❖ آنالیز الگوریتم: منظور از آنالیز الگوریتم این است که پارامترهای زمان محاسبه و حافظه مورد نیاز برای محاسبه الگوریتم به دست آید تا بتوان الگوریتم های مختلف رابرای یک مسأله خاص با یکدیگر مقایسه کرد.

تعریف مسأله

❖ مساله: سوالی که به دنبال پاسخ آن هستیم.

1. مرتب سازی یک لیست(S) متشکل از n عدد به ترتیب صعودی

Sorting

Input: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$

Output Permutation (reordering) $\langle a'_1, \ldots, a'_n \rangle$

such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

2. تعیین اینکه آیا عدد x در لیست S متشکل از n عدد وجود دارد یا خیر. در صورت وجود پاسخ بله و در صورت عدم وجود پاسخ خیر خواهد بود.

❖ پارامترهای مساله

1. S و n
2. x، S و n

❖ نمونه مساله: هر انتساب خاصی از مقادیر به پارامترها

❖ راه حل یک نمونه مساله: پاسخ سوال پرسیده شده توسط مساله در آن نمونه مساله

نمونه مساله۱: S={8,13,5,11,7,10} و n=6 راه حل: {5,7,8,10,11,13}

مراحل حل یک مسأله

1. تعریف و بیان مسأله و فهم کامل آن
2. تشخیص و تدوین یک مدل برای حل مسأله (بررسی کلیه روشهای حل مسأله)
3. طراحی الگوریتم بر اساس مدل انتخاب شده
4. بررسی و ارزیابی صحت الگوریتم
5. تحلیل الگوریتم و ارزیابی پیچیدگی آن
6. پیاده سازی الگوریتم با استفاده از زبانهای برنامه سازی
7. تست برنامه
8. مستندسازی

روشهای حل یک مسأله

- تقسیم و حل (Divide & Conquer)
- حریصانه (Greedy)
- برنامه نویسی پویا (Dynamic Programming)
- بازگشت به عقب (Back Tracking)
- شاخه و حد (Branch & Bound)

بیان الگوریتم

1. تشریح الگوریتم توسط یک زبان طبیعی برای الگوریتم های آسان و کوچک.

   به عنوان مثال عبارتی مانند " n را بگیر و در c ضرب کن»

   معایب نوشتن الگوریتم ها به زبان طبیعی:

   - نوشتن و درک الگوریتم های پیچیده مشکل است
   - ترجمه آن به یک زبان برنامه نویسی مشکل است

2. نمایش گرافیکی الگوریتم توسط فلوچارت (برای الگوریتم های آسان و کوچک)
3. نمایش با استفاده از شبه کد (Pseudo Code) که شباهت زیادی به زبان های C و پاسکال دارد.

شبه کد

❖ در شبه کد هرگاه امکانش باشد، مراحل با وضوح بیشتر با استفاده از روابط ریاضی و توضیحات انگلیسی نشان داده می شوند.

```
if ( low ≤ x ≤ high )
  {
    …
  }
exchange x and y ;
```

```
void seqSearch (int n, const keyType s[], keyType x, index& location)
{
   location = 0;
   while(location < n && s[location] != x)
      location++;
   if(location > n)
      location = -1;
}
```

❖ برای هر یک از الگوریتم های زیر ورودی و خروجی تعیین کنید:

- الگوریتم جمع عناصر یک آرایه
- الگوریتم ضرب ماتریس ها
- الگوریتم مرتب سازی عناصر به صورت صعودی (مرتب سازی تعویضی exchange sort)

الگوریتم

❖ الگوریتم جمع عناصر یک آرایه

```
number sum ( int n, const number S[ ] )
{
    index i;
    number result;

    result = 0;
    for ( i = 1; i <= n; i++)
        result = result + S[i];
    return result;
}
```

❖ الگوریتم ضرب ماترس

```
void matrixmult ( int n,
                  const number A[ ][ ],
                  const number B[ ][ ],
                        number C[ ][ ] )
{
    index i, j, k;
    for ( i = 1; i <= n; i++)
        for ( j = 1; j <= n; j++)
            C[ i ] [ j ] = 0;
            for ( k = 1; k <= n; k++)
                C[ i ] [ j ] = C[ i ] [ j ] + A[ i ] [ k ] * B[ k ] [ j ];
}
```

الگوریتم

❖ الگوریتم مرتب سازی عناصر به صورت صعودی
(مرتب سازی تعویضی exchange sort)

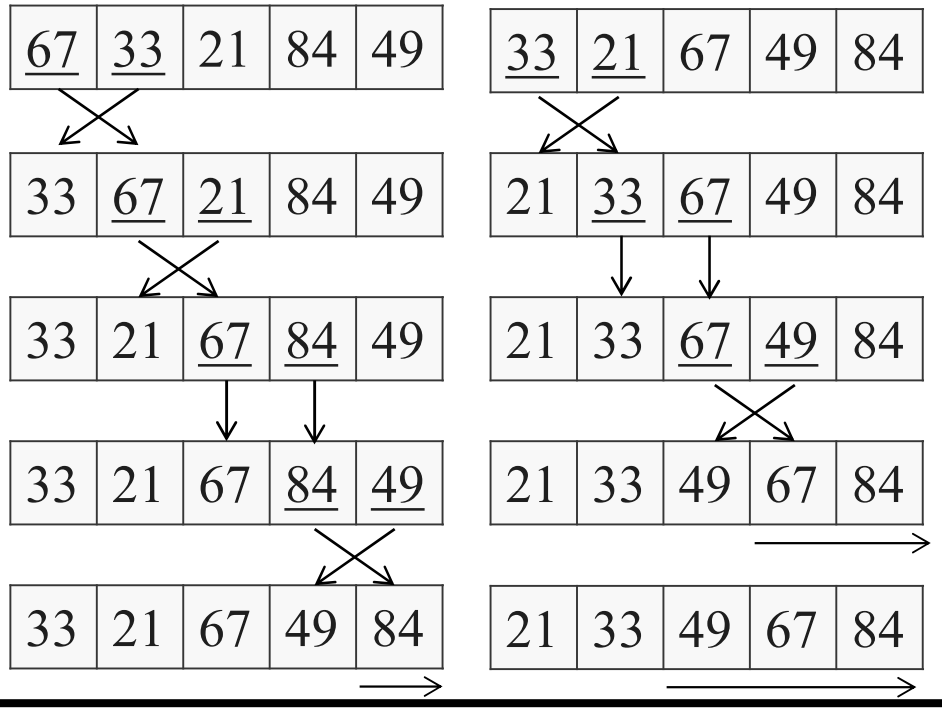
```
void exchangesort ( int n, keytype S[ ] )
{
   index  i, j ;

   for  ( i = 1; i <= n - 1; i++)
       for  ( j = i + 1; j <= n; j++)
           if  ( S[j] < S[i] )
               exchange  S[i] and S[j];
}
```

الگوریتم

Exchange
bubble

جفتهایی از عناصر را که به ترتیب نیستند تعویض می کنند تا چنین جفتهایی وجود نداشته باشند. در مرتب‌سازی حبابی در هر گذر تضمین می‌شود بزرگترین عنصر به انتهای لیست برود و بعضی از عناصر کوچک به سمت ابتدای لیست حرکت کنند. بدترین حالت زمانی است که عناصر لیست به صورت معکوس باشند. در گذر اول n-1 عنصر مقایسه و تعویض می شوند. درگذر بعدی n-2 تعویض صورت می گیرد.



6 5 3 1 8 7 2 4

الگوریتم جستجو ترتیبی

❖ جستجوی عنصری در یک آرایه مرتب (جستجوی خطی)

```
void seqSearch (int n, const keytype S[ ], keytype x, index
&location)
{
  location = 0;
  while (location <= n && S[location] ! = x)
       location++;
   if (location > n )
       location = -1 ;
}
```

الگوریتم جستجو باینری

- x با عنصر وسط لیست S مقایسه می شود. اگر برابر باشد اندیس عنصر در location قرار می گیرد. اگر بزرگتر باشد نیمه راست لیست و اگر کوچکتر باشد نیمه چپ لیست بررسی می شود.

جستجوی عنصری در یک آرایه مرتب

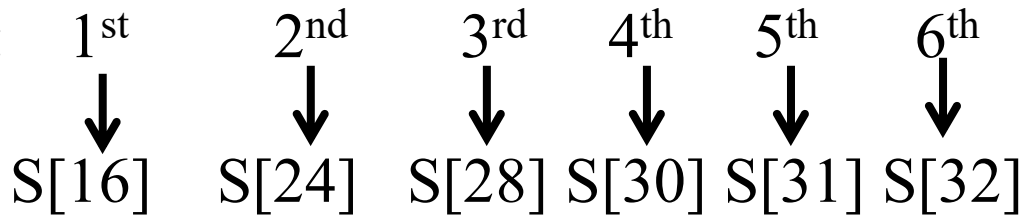
```
void binsearch ( int n,
                 const keytype S[ ],
                 keytype x,
                 index& location )
{
    index low, high, mid;

    low = 1 ;  high = n ;
    location = 0 ;
    while ( low <= high && location == 0) {
        mid = ⌊ ( low + high ) / 2 ⌋ ;
        if ( x == S[mid] )
            location = mid ;
        else if ( x < S[mid] )
            high = mid – 1 ;
        else
            low = mid + 1 ;
    }
}
```

مقایسه الگوریتم های جستجو

❖ با فرض داشتن آرایه ۳۲ عنصری

- Linear (Sequential) search: ۳۲ مقایسه در بدترین حالت
- Binary search:

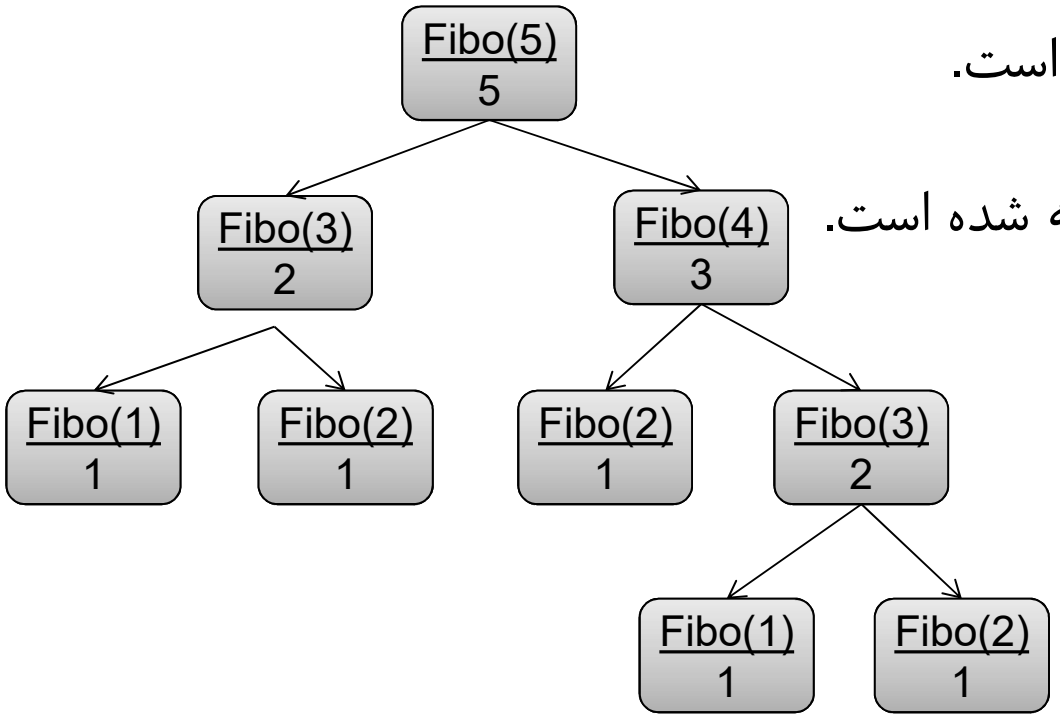


تعداد مقایسه های انجام شده توسط جستجوی ترتیبی (n)	تعداد مقایسه های انجام شده توسط جستجوی دودویی (باینری) (lg n +1)	اندازه آرایه
۱۲۸	۸	۱۲۸
۱۰۲۴	۱۱	۱۰۲۴
۱۰۴۸۵۷۶	۲۱	۱۰۴۸۵۷۶
۴۲۹۴۹۶۷۲۹۴	۳۳	۴۲۹۴۹۶۷۲۹۴

الگوریتم جمله n ام فیبوناچی (بازگشتی)

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

❖ رسم درخت بازگشتی



نشان می دهد الگوریتم ناکارآمد است.

الگوریتم محاسبات تکراری دارد.

مثال: fibo(2) سه مرتبه محاسبه شده است.

n	تعداد جملات محاسبه شده
۱	۱
۲	۱
۳	۳
۴	۵
۵	۹
۶	۱۵

الگوریتم جمله n ام فیبوناچی (بازگشتی)

- $T(n)$: تعداد جملات محاسبه شده در درخت بازگشتی
- با افزودن ۲ واحد به n، تعداد جملات درخت بیش از 2 برابر می شود.

$$T(n) > 2*T(n-2) \quad \text{when } n>=2$$
$$> 2*2*T(n-4)$$
$$> 2*2*2*T(n-6)$$
$$\ldots.$$
$$> \underbrace{2*2*\ldots*2}_{2^{n/2}}*T(0) = 2^{n/2}$$

- $T(n) > 2^{n/2}$

الگوریتم جمله n ام فیبوناچی (تکراری)

❖ برای تعیین جمله nام، هر یک از n-1 جمله نخست را فقط یک بار محاسبه می کند.

❖ الگوریتم کارآمد

```
int fibo2 (int n)
  {
   int  f[0 . . n];
   f[0]=0;
   if (n >0 ) {
       f[1]=1;
       for (i=2; i<=n; i++)
            f[i]=f[i-1]+f[i-2];
    }
     return f[n];
  }
```

مقایسه دو الگوریتم فیبوناچی

n	روش تکراری (n+1)	روش بازگشتی ($2^{n/2}$)	زمان اجرای تکراری	زمان اجرای بازگشتی
40	41	1,048576	41ns	1048 µs
60	61	$1.1*10^{9}$	61ns	1 s
100	101	$1.1*10^{15}$	101ns	13 days
120	121	$1.2*10^{18}$	121ns	36 years
160	161	$1.2*10^{24}$	161ns	$3.8*10^{7}$ years
200	201	$1.3*10^{30}$	201ns	$4*10^{13}$ years

با فرض محاسبه هر جمله در مدت ۱ نانوثانیه

$$1\ \mu s = 10^{-6}\ s$$
$$1\ ns = 10^{-9}\ s$$

تحلیل الگوریتم ها

❖ هدف از تحلیل الگوریتم ها:

✓ بررسی رفتار الگوریتم از نظر زمان اجرا و مقدار حافظه مصرفی قبل از پیاده سازی

✓ مقایسه الگوریتم ها از نظر کارایی

- Time complexity
- Space complexity

❖ عوامل موثر در زمان اجرای یک برنامه

✓ سرعت سخت افزار

✓ نوع کامپایلر (بهینگی کد مقصد)

✓ برنامه نویس (بهینگی کد منبع)

✓ اندازه ورودی

✓ ترکیب دادهای ورودی

✓ پیچیدگی الگوریتم

✓ ...

❖ مهمترین عامل پیچیدگی الگوریتم می باشد که خود تابعی از اندازه ورودی است.

تحلیل پیچیدگیهای زمان و فضا

❖ پیچیدگی فضا (Space complexity):

- ✓ میزان حافظه مورد نیاز از اجرا تا تکمیل الگوریتم می باشد.
- ✓ به حاصل جمع فضای مورد نیاز متغیرها، آرایه ها، پشته ها و کلیه ساختمان داده های مورد نیاز و همچنین فضای ذخیره کد برنامه در حافظه، پیچیدگی فضای الگوریتم گفته می شود. بر حسب n (تعداد ورودی ها) سنجیده شود.

- ✓ پیچیدگی فضای کد زیر چیست؟

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

- ✓ O(1)
- ✓ اگر پیچیدگی فضای یک الگوریتم به n وابسته نباشد، پیچیدگی آن ثابت فرض شده و از مرتبه O(1) در نظر گرفته می شود.

تحلیل پیچیدگیهای زمان و فضا

- پیچیدگی زمان(Time complexity):
  - ✓ محاسبه کارایی بر حسب زمان
  - ✓ مستقل از کامپیوتر، زبان برنامه نویسی و برنامه نویس
  - ✓ تحلیل پیچیدگی زمانی یک الگوریتم، تعیین تعداد دفعاتی است که عملیات اصلی الگوریتم (مقایسه، جمع، ضرب و ...) بر حسب اندازه ورودی انجام می شود.
- برای تحلیل پیچیدگی یک الگوریتم تابعی به نام $T(n)$ در نظر گرفته می شود که در آن n اندازه ورودی می باشد.
- تحلیل پیچیدگی زمانی الگوریتم ها
  - ✓ غیربازگشتی
    - حلقه ها، دستورات کنترلی، حلقه های تو در تو و ...
    - اگر تعداد تکرار دستورات حلقه تکرار به n بستگی نداشته باشد، $T(n)$ (پیچیدگی زمان) مقداری ثابت خواهد بود.
  - ✓ بازگشتی

تحلیل پیچیدگی زمانی

❖ اندازه ورودی

- ✓ در بسیاری از الگوریتم ها یافتن میزانی منطقی از اندازه ورودی آسان است.
  - مثل: جستجوی ترتیبی- محاسبه مجموع عناصر آرایه- مرتب سازی تعویضی- جستجوی دودویی- ضرب ماتریس ها
- ✓ در بعضی از الگوریتم ها بهتر است اندازه ورودی برحسب دو عدد سنجیده شود.
  - اگر ورودی الگوریتم یک گراف باشد، اندازه ورودی برحسب تعداد رئوس و تعداد یال ها سنجیده می شود.

❖ عملیات اصلی

- ✓ عمل اصلی
  - دستور یا مجموعه ای از دستورات به طوری که کل کار انجام شده توسط الگوریتم، تقریبا متناسب با تعداد دفعات اجرای این دستور یا دستورات باشد.
- ✓ مثال: جستجوی ترتیبی و جستجوی دودویی لیستی به طول n عنصر
  - در هر مرحله عنصر x با یک عنصر از S مقایسه می شود.
  - با تعیین اینکه هر یک از الگوریتم ها چند بار این عمل اصلی را انجام می دهند، می توان کارایی این دو الگوریتم را به ازای هر مقدار از n مقایسه نمود.

یافتن max و min در یک آرایه

```
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
```

❖ محاسبه زمان اجرای الگوریتم

۱- شماره دستورالعمل	۲- تعداد دفعات تکرار	۳- زمان مصرفی هر اجرا	۳*۲
۱	1		
۲	2		
۳	n		
۴	n-1		
۵	n-1حداکثر		
۶	n-1		
۷	n-1حداکثر		
۸	n-1		
۹	1		
	مجموع=شاخص زمان مصرفی		مجموع=زمان مصرفی

هزینه زمانی تابع فاکتوریل

```
(1)     int    Factorial( int  n )
        {
(2)             int  fact= 1 ;
(3)             for( int  i=2  ; i<= n ; i ++  )
(4)                   fact*= i ;
(5)             return fact ;
        }
```

سطر	هزینه	تعداد
2	C_1	۱
3	C_2	n
4	C_3	$n-1$
5	C_4	۱

$$T(n) = C_1 + C_2 n + C_3 (n-1) + C_4$$

تحلیل پیچیدگی در حالات مختلف

- در برخی موارد مانند الگوریتم مجموع عناصر آرایه، عمل اصلی همواره به ازای یک نمونه با اندازه ورودی n به یک میزان انجام می شود.
- در برخی موارد دیگر، تعداد دفعات اجرای عمل اصلی نه تنها به اندازه ورودی بلکه به ترکیب و چیدمان مقادیر ورودی نیز بستگی دارد.
  - ✓ مثال: در جستجوی ترتیبی (با اندازه ورودی برابر با n)
    - اگر x در اولین مکان آرایه باشد: تعداد مقایسه ها =۱
    - اگر x در آرایه نباشد: تعداد مقایسه ها= n

- $T(n)$: تعداد دفعاتی که الگوریتم عمل اصلی را به ازای یک نمونه با اندازه ورودی n انجام می دهد.

تحلیل پیچیدگی در حالات مختلف

- ❖ بهترین حالت (Best case) B(n)
  - ✓ حالتی که الگوریتم می تواند سریعتربن زمان اجرا را داشته باشد.
  - ✓ کران پایین زمان اجرا است.
- ❖ بدترین حالت (Worst case) W(n)
  - ✓ حالتی که زمان اجرای الگوریتم هرگز از آن بیشتر نخواهد شد.
  - ✓ کران بالای زمان اجرا است.
  - ✓ جهت مقایسه بین الگوریتم ها معمولا از این تابع استفاده می شود.
- ❖ حالت میانگین (Average case) A(n)
  - ✓ برای تحلیل آن نیاز به در نظر گرفتن توزیع های مختلف مقادیر ورودی می باشد.
  - ✓ محاسبه آن مشکل است.

تحلیل پیچیدگی زمانی

❖ جستجوی خطی(linear search)

✓ عمل اصلی: تعداد مقایسه ✓ اندازه ورودی: تعداد عناصر آرایه n

- بهترین حالت: وقتی که عنصر مورد نظر در اولین خانه باشد B(n)=1
- بدترین حالت: وقتی که عنصر مورد نظر در آرایه موجود نباشد W(n)=n
- حالت متوسط:
  - حالت اول: عنصر مورد نظر قطعا در آرایه وجود دارد. در این صورت احتمال اینکه عنصر در مکان k باشد برابر 1/n باشد.

$$T(n)=\sum_{k=1}^{n}(k\times\frac{1}{n})=\frac{1}{n}\times\sum_{k=1}^{n}k=\frac{1}{n}\times\frac{n(n+1)}{2}=\frac{n+1}{2}$$

  - حالت دوم: عنصر مورد نظر ممکن است در آرایه وجود نداشته باشد.
    - احتمال وجود عنصر مورد نظر در آرایه برابر با p
    - احتمال وجود آن در خانه k ام برابر با p/n است
    - احتمال عدم وجود عنصر در آرایه 1-p

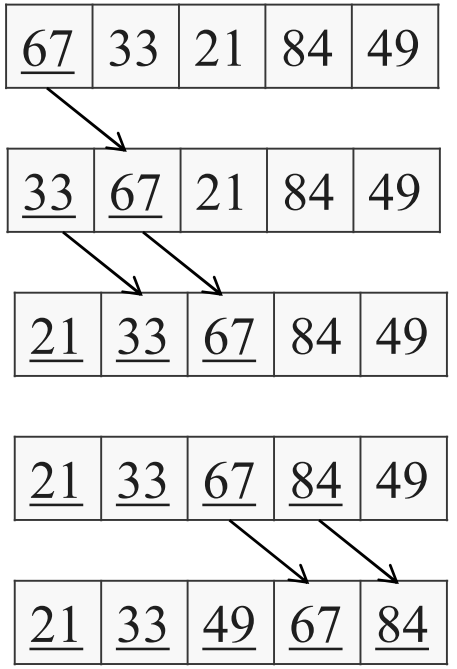
$$T(n)=\sum_{k=1}^{n}(k\times\frac{p}{n})+n(1-p)=\frac{p}{n}\times\frac{n(n+1)}{2}+n(1-p)=n(1-\frac{p}{n})+\frac{p}{2}$$

مرتب سازی درجی

Insertion

عناصر در لیست مرتب طوری درج می شوند که ترتیب لیست حفظ شوند. بدترین حالت در مرتب سازی درج زمانی است که عناصر لیست به صورت معکوس مرتب باشند.

6 5 3 1 8 7 2 4



محاسبه هزینه زمانی مرتب سازی درجی

	هزینه	دفعات اجرا
void ***insertionSort***(int n, int a[])	-	-
{	-	-
for(int j = 2; j <= n; j++)	c_1	n
{	-	-
// Insert a[j] into the sorted sequence a[1..j-1].	0	n - 1
int key = a[j];	c_3	n - 1
int i = j – 1;	c_4	n - 1
while(i > 0 && a[i] > key)	c_5	$\sum_{j=2}^{n} t_j$
{	-	-
a[i+1] = a[i];	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i--;	c_7	$\sum_{j=2}^{n} (t_j - 1)$
}	-	-
a[i+1] = key;	c_8	n -1
}	-	-
}	-	-

کل زمان اجرای مرتب‌سازی درجی

$$T(n) = c_1 n + c_3 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1)$$

پیچیدگی زمانی مرتب سازی درجی

- بهترین حالت (Best case)
  - ✓ زمانی که آرایه از قبل مرتب باشد.
  - ✓ $t_j=1$

کل زمان اجرای مرتب‌سازی درجی

$$T(n) = c_1 n + c_3(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

$$T(n) = c_1 n + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_3 + c_4 + c_5 + c_8)n - (c_3 + c_4 + c_5 + c_8)$$

- هزینه زمانی مرتب سازی درجی در بهترین حالت تابعی خطی و به صورت an+b می باشد.

پیچیدگی زمانی مرتب سازی درجی

❖ بدترین حالت (Worst case)

✓ زمانی که آرایه به ترتیب عکس مرتب باشد.

✓ در اینصورت هر عنصر a[j] باید با تمامی عناصر مرتب شده در زیرآرایه a[1...j-1] مقایسه شود. بنابراین برای $j=2,3,\ldots,n$ داریم $t_j=j$

کل زمان اجرای مرتب‌سازی درجی

$$T(n)=c_1n+c_3(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n}t_j+c_6\sum_{j=2}^{n}(t_j-1)+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

$$T(n)=c_1n+c_3(n-1)+c_4(n-1)+c_5(\frac{n(n+1)}{2}-1)+c_6(\frac{n(n-1)}{2})+c_7(\frac{n(n-1)}{2})+c_8(n-1)$$

$$=(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2})n^2+(c_1+c_3+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8)n-(c_3+c_4+c_5+c_8)$$

❖ هزینه زمانی مرتب سازی درجی در بدترین حالت تابعی درجه دوم و به صورت an^2+bn+c می باشد.

پیچیدگی زمانی مرتب سازی درجی

❖ حالت متوسط (Average case)

✓ فرض می شود a[j] از نصف عناصر a[1..j-1] کوچکتر است.

✓ در اینصورت هر عنصر a[j] باید با نصف عناصر مرتب شده در زیرآرایه a[1...j-1] مقایسه شود. بنابراین برای $j=2,3,\ldots,n$ داریم $t_j=j/2$

کل زمان اجرای مرتب‌سازی درجی

$$T(n) = c_1 n + c_3(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

❖ هزینه زمانی مرتب سازی درجی در بدترین حالت تابعی درجه دوم و به صورت an^2+bn+c می باشد.

❖ حالت متوسط در این الگوریتم مانند بدترین حالت است.

مرتبه رشد (نرخ رشد)

❖ زمانیکه مقدار ورودی (n) بزرگ باشد، بیان دقیق زمان اجرا برحسب n ضروری نیست زیرا جمله با بزرگترین درجه در زمان موثر است.

N	n^2	n^2+n
۱	۱	۲
۱۵	۲۵	۳۰
۱۰	۱۰۰	۱۱۰
۱۰۰	۱۰۰۰۰	۱۰۱۰۰
۱۰۰۰	۱۰۰۰۰۰۰	۱۰۰۱۰۰۰

مرتبه رشد (نرخ رشد)

- مرتبه یک الگوریتم با بزرگتربن جمله تابع پبچبدگی زمانی آن تعیین می شود.
- پیچیدگی و مرتبه اجرایی الگوریتم به n وابسته است و تابعی از n می باشد و با $T(n)$ نشان داده می شود.
- الگوریتم هایی با پیچیدگی زمانی از قبیل n، 100n را الگوریتم های خطی می گویند.
- مجموعه کامل توابع پیچیدگی را که با توابع درجه دوم محض قابل دسته بندی باشند (n^2) گویند.
- تابعی که عضو مجموعه (n^2) باشد، از مرتبه n^2 است.
- مجموعه ای از توابع پیچیدگی که با توابع درجه سوم محض قابل دسته بندی باشند (n^3) گویند.
- گروه های پیچیدگی:

$$\theta(\log n), \theta(n), \theta(n\log n), \theta(n^2), \theta(n^3), \theta(2^n), \ldots$$

نمادهای مجانبی (asymptotic notation)

- برای توصیف زمان اجرای الگوریتم از نمادهای مجانبی استفاده می شود.
- جهت مقایسه الگوریتم های مختلف با یکدیگر از نمادهای مجانبی استفاده می شود.

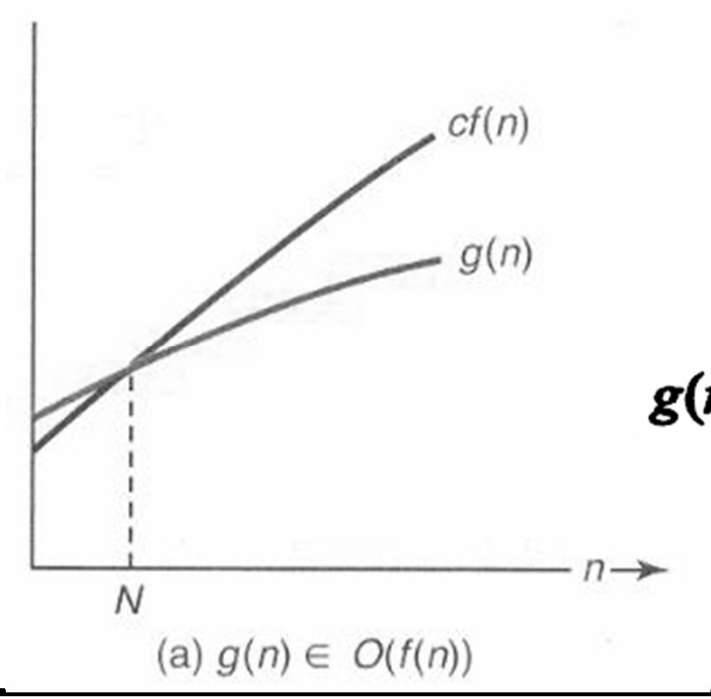
$$O, o, \Omega, \omega, \theta$$

نمادهای مجانبی

❖ Big O (O بزرگ)

✓ O(f(n)) مجموعه ای از توابع پیچیدگی g(n) است که برای آن ها یک ثابت حقیقی مثبت c و یک عدد صحیح غیرمنفی N وجود دارد به قسمی که به ازای همه n>=N داریم:

$g(n) <= c \times f(n)$



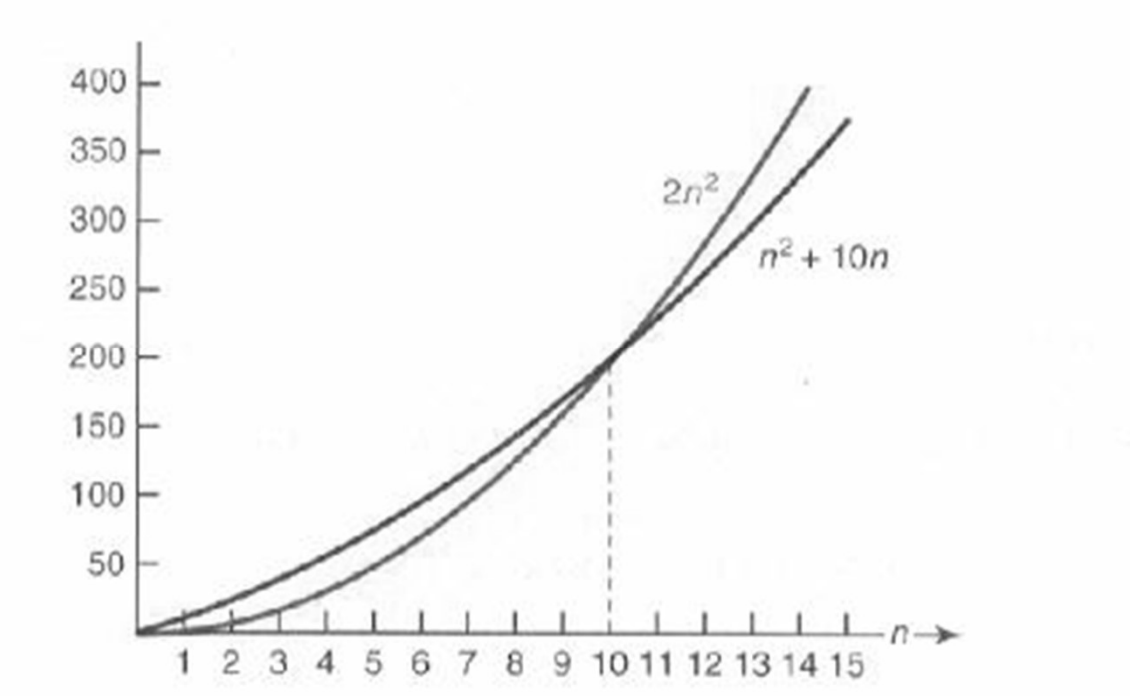
(a) $g(n) \in O(f(n))$

$$g(n) \in O(f(n)) \Leftrightarrow c, N > 0 : \forall n \geq N \; g(n) \leq cf(n)$$

نمادهای مجانبی

❖ Big O (O بزرگ)

$$g(n) <= c \times f(n)$$



✓ O بزرگ یک حد بالای مجانبی (کران بالا) بر روی یک تابع قرار می دهد.

نمادهای مجانبی

- $5n^2 \in O(n^2)$

  $5n^2 \leq 5n^2 \Rightarrow N = 0, c = 5$

- $T(n) = n(n\text{-}1)/2 \in O(n^2)$

  $n(n\text{-}1)/2 \leq n(n)/2 = (1/2)n^2$

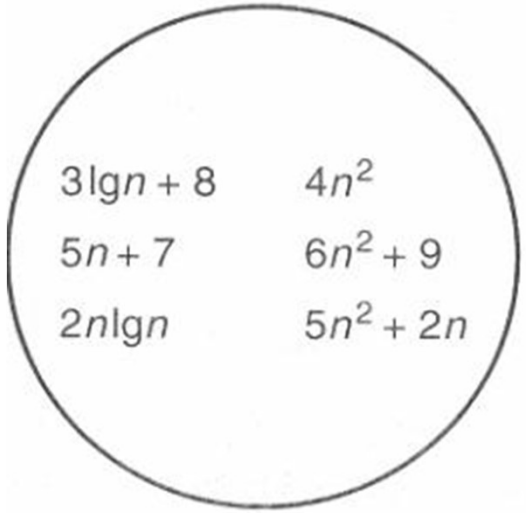
  $\Rightarrow N = 0, c = 1/2$

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

  $N = 10, c = 1$

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

  $N = 1, c = 1$



(a) $O(n^2)$

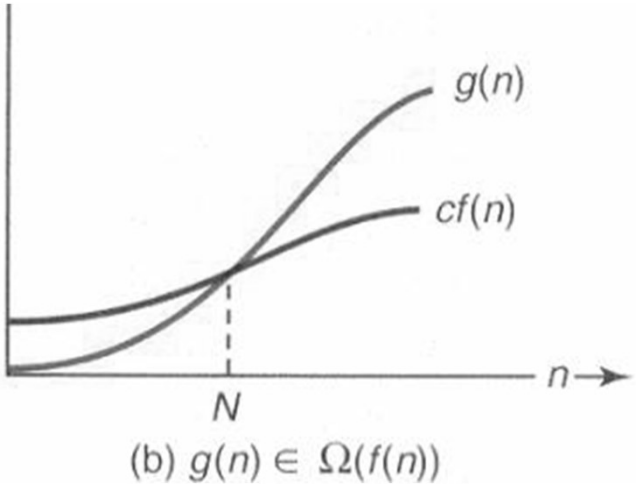
✓ به طور کلی $O(n^2)$ شامل تمام توابعی است که رشدشان کمتر یا مساوی n^2 است.

نمادهای مجانبی

❖ Ω (امگای بزرگ)

✓ برای یک تابع پیچیدگی مفروض f(n)، $\Omega(f(n))$ مجموعه ای از توابع پیچیدگی g(n) است که برای آنها یک عدد ثابت حقیقی مثبت c و یگ عدد صحیح غیرمنفی N وجود دارد طوری که به ازای همه n>=N داریم:

$$g(n) >= c \times f(n)$$



(b) $g(n) \in \Omega(f(n))$

❖ اگر $g(n) \varepsilon \Omega(f(n))$ می گوییم g(n) از امگای f(n) می باشد.

✓ Ω یک حد پایین مجانبی (کران پایین حدی) روی یک تابع قرار می دهد.

نمادهای مجانبی

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

  $5n^2 \geq 1 \times n^2 \Rightarrow N = 0,\ c = 1$

- $n^2 + 10n \in \Omega(n^2)$

  $n^2 + 10n \geq n^2 \Rightarrow N = 0,\ c = 1$

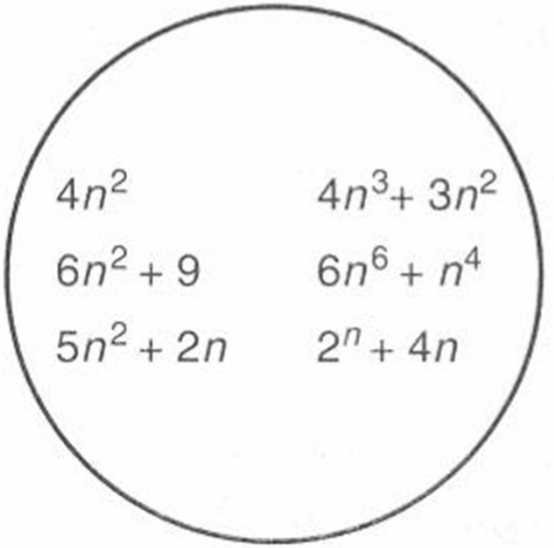
- $T(n) = n(n\text{-}1)/2 \in \Omega(n^2)$

  $n \geq 2 \Rightarrow n - 1 \geq n/2 \Rightarrow$

  $n(n\text{-}1)/2 \geq (n/2)(n/2) = n^2/4$

  $\Rightarrow N = 2,\ c = 1/4$

- $n^3 \in \Omega(n^2)$



(b) $\Omega(n^2)$

✓ به طور کلی $\Omega(n^2)$ شامل تمام توابعی است که رشدشان بیشتر یا مساوی n^2 است.

نمادهای مجانبی

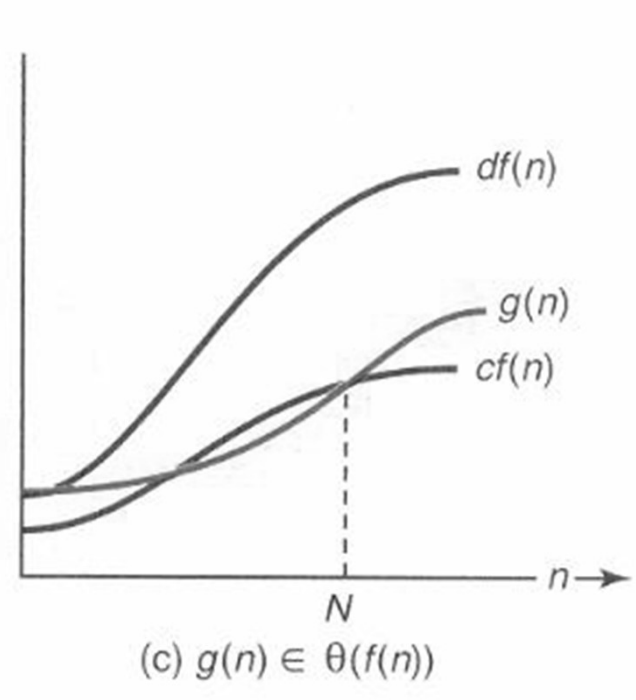
❖ برای یک تابع پیچیدگی مفروض f(n)، داریم:

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

❖ یعنی $\theta(f(n))$ مجموعه ای از توابع پیچیدگی g(n) است که برای آن ها ثابت های حقیقی c و d و عدد صحیح غیر منفی N وجود دارد طوری که:

$$c \times f(n) <= g(n) <= d \times f(n)$$

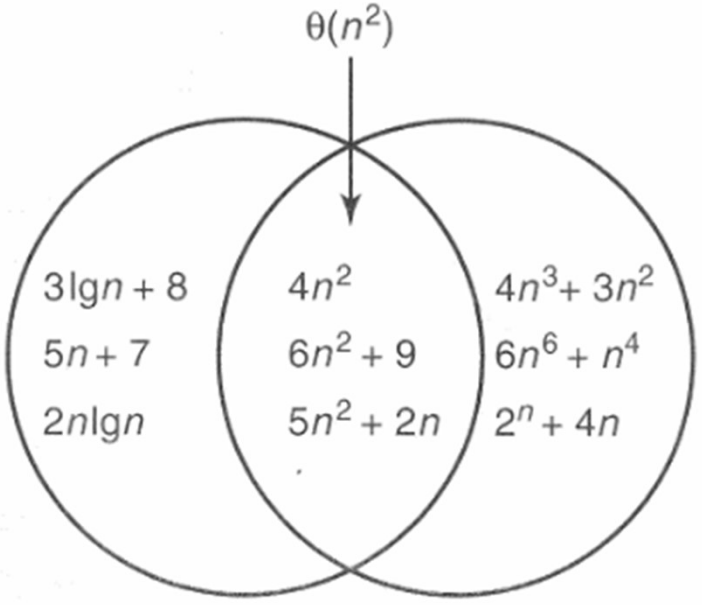
❖ تابع را از بالا و پایین محدود می کند.



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

نمادهای مجانبی

- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- اگر $g(n) \in \Theta(f(n))$ ، می گوییم $g(n)$ از مرتبه (دقیق) $f(n)$ می باشد.



(c) $\theta(n^2) = O(n^2) \cap \Omega(n^2)$

نمادهای مجانبی

❖ Small o (O کوچک)

✓ برای یک تابع پیچیدگی f(n) مفروض، o(f(n)) عبارت است از مجموعه کلیه توابع پیچیدگی g(n) که به ازای هر ثابت حقیقی مثبت c، یک عدد صحیح غیرمنفی N وجود دارد به قسمی که به ازای همه n>=N داریم:

$g(n) < c \times f(n)$

❖ Small ω (ω کوچک)

✓ برای یک تابع پیچیدگی f(n) مفروض، ω(f(n)) عبارت است از مجموعه کلیه توابع پیچیدگی g(n) که به ازای هر ثابت حقیقی مثبت c، یک عدد صحیح غیرمنفی N وجود دارد به قسمی که به ازای همه n>=N داریم:

$c \times f(n) < g(n)$

نمادهای مجانبی

❖ برای درک سریع نمادهای پیچیدگی می توان آنها را به صورت زیر با عملگرهای مقایسه ای هم ارز دانست:

$$g(n) \in O(f(n)) \rightarrow g(n) \leq f(n)$$

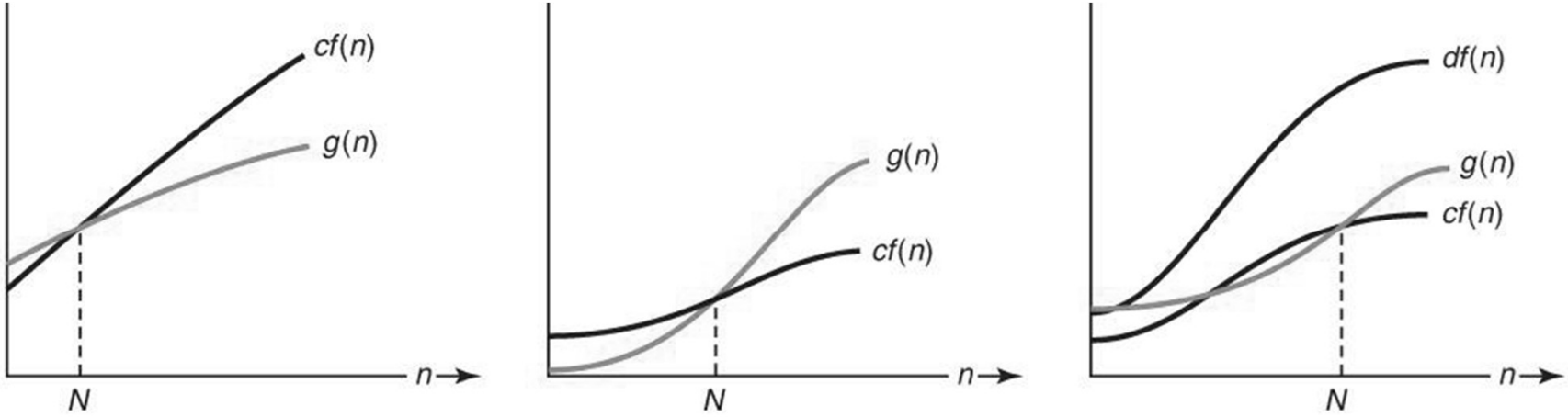
$$g(n) \in \Omega(f(n)) \rightarrow g(n) \geq f(n)$$

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

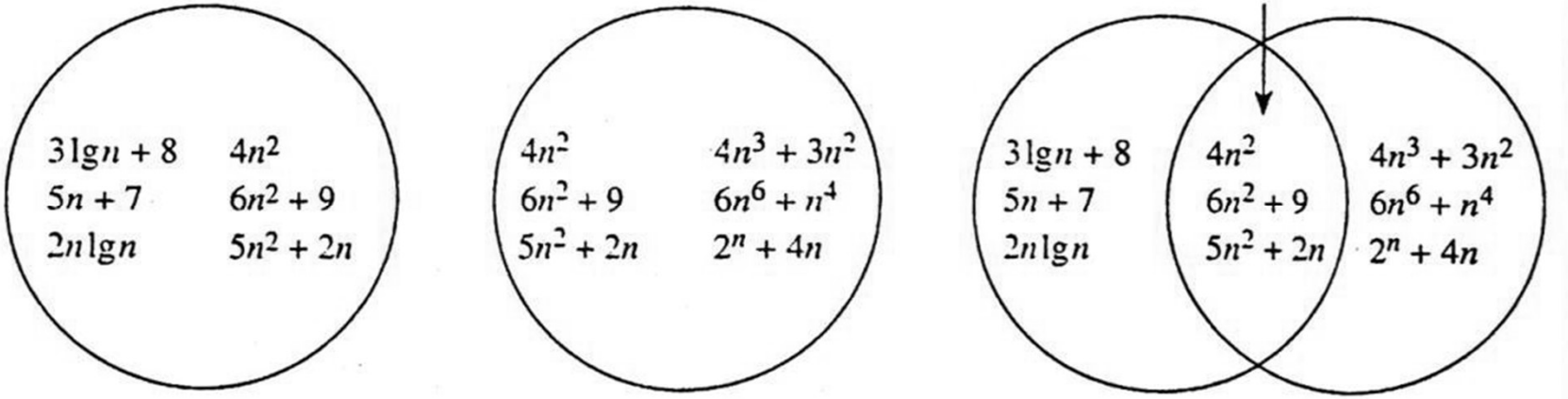
$$g(n) \in o(f(n)) \rightarrow g(n) < f(n)$$

$$g(n) \in \omega(f(n)) \rightarrow g(n) > f(n)$$

نمادهای مجانبی



(a) $g(n) \in O(f(n))$ (b) $g(n) \in \Omega(f(n))$ (c) $g(n) \in \theta(f(n))$



(الف) $O(n^2)$ (ب) $\Omega(n^2)$ (ج) $\theta(n^2) = O(n^2) \cap \Omega(n^2)$

نمادهای مجانبی

❖ تمام عبارات زیر درست هستند.

۱) $5n^2 - 6n = \theta(n^2)$

۲) $n! = O(n^n)$

۳) $2n^2 2^n + n\log n = \theta(n^2 2^n)$

۴) $\sum_{i=0}^{n} i^2 = \theta(n^3)$

۵) $\sum_{i=0}^{n} i^3 = \theta(n^4)$

۶) $n^{2^n} + 6\times 2^n = \theta(n^{2^n})$

۷) $n^3 + 10^6 n^2 = \theta(n^3)$

۸) $6n^3 /(\log n + 1) = O(n^3)$

۹) $n^{1.001} + n\log n = \theta(n^{1.001})$

۱۰) $10n^3 + 15n^4 + 100n^2 2^n = O(n^2 2^n)$

۱۱) $33n^3 + 4n^2 = \Omega(n^2)$

۱۲) $33n^3 + 4n^2 = \Omega(n^3)$

۱۳) $6n^2 + 20n \in O(n^3)$

استفاده از حد برای تعیین مرتبه زمانی

❖ اگر دو الگوریتم با پیچیدگی های f(n) و g(n) داشته باشیم برای مقایسه آنها می توان از حد استفاده نمود.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\begin{cases}c & : \quad g(n)\in\theta(f(n))\\ 0 & : \quad g(n)\in\Omega(f(n))\\ \infty & : \quad g(n)\in O(f(n))\end{cases}$$

و یا به صورت زیر :

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\begin{cases}c & : \quad f(n)\in\theta(g(n))\\ 0 & : \quad f(n)\in O(g(n))\\ \infty & : \quad f(n)\in\Omega(g(n))\end{cases}$$

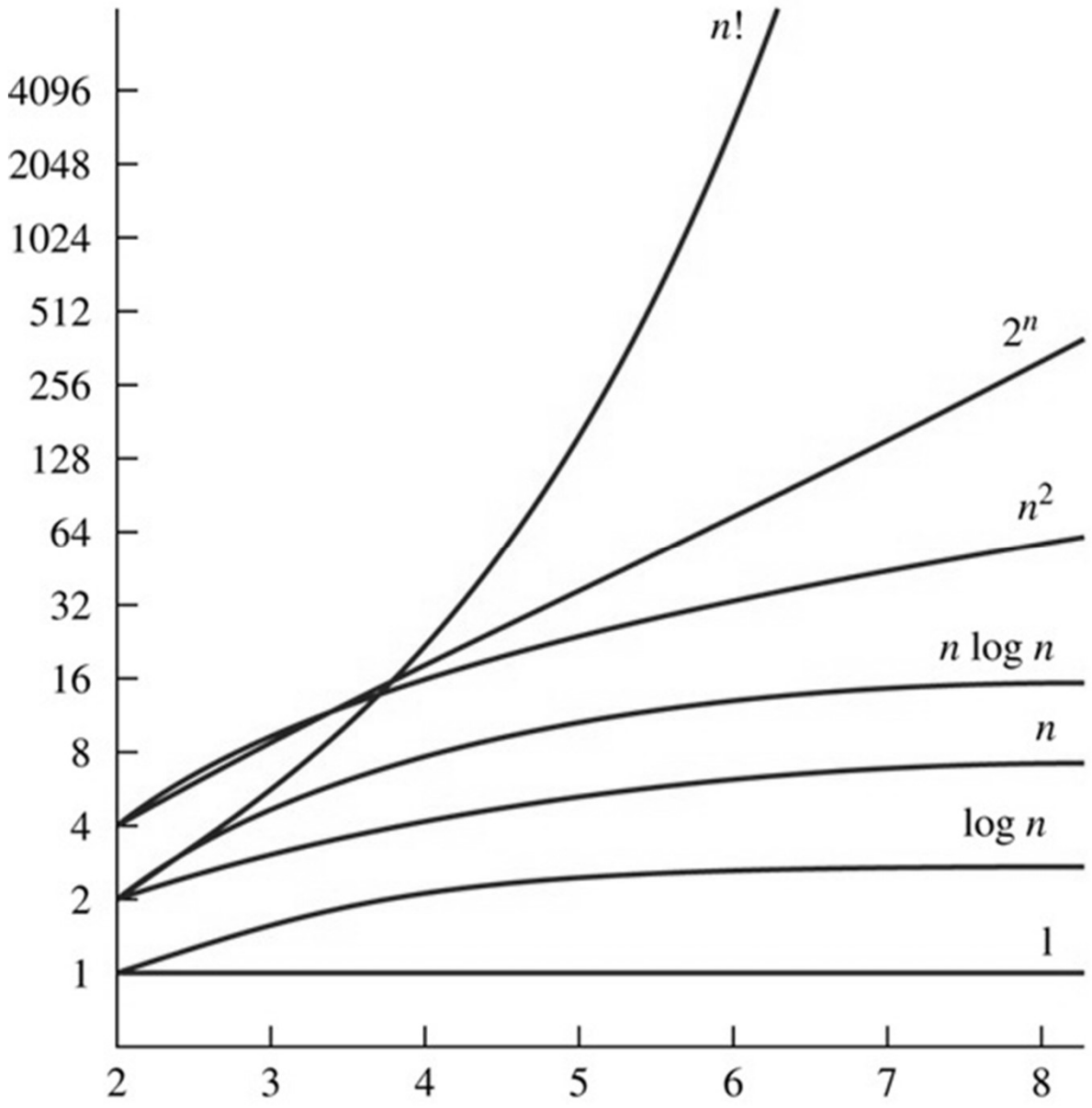
ترتیب توابع رشد

- رشد تابع f(n) از g(n) بیشتر است در صورتی که اگر n به سمت بینهایت میل کند آنگاه f(n) زودتر به بینهایت میل کند.

$$O(1) < O(logn) < O(\sqrt{n}) < O(n) < O(nlogn) < O(n^2) < O(n^2logn) < O(n^3) < O(2^n) < O(3^n) < O(n!) < O(n^n)$$

- در محاسبه O
  - ✓ فاکتورهای ثابت محاسبه نمی شوند.
  - ✓ نرخ رشد در چند جمله ای ها جمله با بیشترین توان است.
  - ✓ توان های بالاتر سریعتر از توان های پایین تر رشد می کنند.
  - ✓ توابع نمایی سریعتر از توابع توانی رشد می کنند.
  - ✓ توابع لگاریتمی کندتر از توابع توانی رشد می کنند.
  - ✓ تمامی توابع لگاریتمی رشدی مشابه دارند.

نمودارهای پیچیدگی



نسبت سرعت رشد

عبارت ریاضی	نسبت سرعت رشد
$g(n)=O(f(n))$	سرعت رشد $f(n)$ ≤ سرعت رشد $g(n)$
$g(n)=\Omega(f(n))$	سرعت رشد $f(n)$ ≥ سرعت رشد $g(n)$
$g(n)=\theta(f(n))$	سرعت رشد $f(n)$ = سرعت رشد $g(n)$

نمادهای مجانبی

❖ خصوصیات نمادها

$g(n) \in O(f(n))$ اگر و فقط اگر $f(n) \in \Omega(g(n))$

$g(n) \in \theta(f(n))$ اگر و فقط اگر $f(n) \in \theta(g(n))$

$$if\ f(n) = a_m n^m + \cdots + a_1 n + a_0\ then\ f(n) = \begin{cases} O(n^m) \\ \Omega(n^m) \\ \Theta(n^m) \end{cases}$$

$$f(n) = \Theta(g(n)) \Leftrightarrow \begin{cases} f(n) = O(g(n)) \\ f(n) = \Omega(g(n)) \end{cases}$$

چند نکته

- هر تابع لگاریتمی بهتر از تابع چندجمله ای و هر تابع چند جمله ای بهتر از هر تابع نمایی و هر تابع نمایی بهتر از هر تابع فاکتوریل عمل می کند.

- الگوریتمی سریعتر است که زمان اجرای بدترین حالت آن رشد کمتری داشته باشد.
  - ✓ الگوریتمی مفید است که کران بالای آن پایین باشد.

- در تعیین مرتبه همواره اجازه حذف جملاتی از مرتبه پایین تر را داریم.

$$5n+3\lg n+10\,n\lg n+7n^2 \in \theta(n^2)$$

- اگر $b>a>0$، در آن صورت : $a^n=O(b^n)$
- به ازای همه مقادیر $a>0$ داریم: $a^n=O(n!)$
- اگر $b>1$، $a>1$ آنگاه داریم: $\log_a n \in \theta(\log_b n)$
- اگر $c>=0$ و $d>0$ و $h(n)\epsilon\theta(f(n)), g(n)\epsilon O(f(n))$ آنگاه

$$c \times g(n) + d \times h(n) \epsilon\, \theta(f(n))$$

تمرین

❖ کدام یک از روابط زیر درست است؟ (مهندسی کامپیوتر – آزاد ۸۱)

1) $O(\log n) < O(\sqrt{n})$

2) $O(\log n) > O(\sqrt{n})$

3) $O(n!) < O(a^n)$

4) $O(\sqrt{n^3}) < O(n)$

فصل دوم

الگوریتم های بازگشتی

- رویکردهای نوشتن الگوریتم های تکراری
  - ✓ تکرار و حلقه
  - ✓ الگوریتم بازگشتی

- الگوریتم بازگشتی
  - ✓ فرایندی تکراری که در آن یک الگوریتم خودش را فراخوانی می کند.

- نحوه اجرای الگوریتم بازگشتی
  - ✓ عمل فراخوانی
    - قرارگیری متغیرهای محلی و مقادیر آنها و آدرس بازگشت در پشته
    - انتقال پارامترها
    - انتقال کنترل برنامه به ابتدای پردازه جدید
  - ✓ بازگشت از یک فراخوانی
    - آدرس بازگشت و ادامه اجرا

الگوریتم های بازگشتی

- مزایا
  - ✓ کدنویسی کوتاه و راحت (سادگی برنامه)

- معایب
  - ✓ فراخوانی های مکرر و نیاز به پشته
  - ✓ معمولا از نظر فضا و زمان بهینه نیستند(مصرف زیاد حافظه)

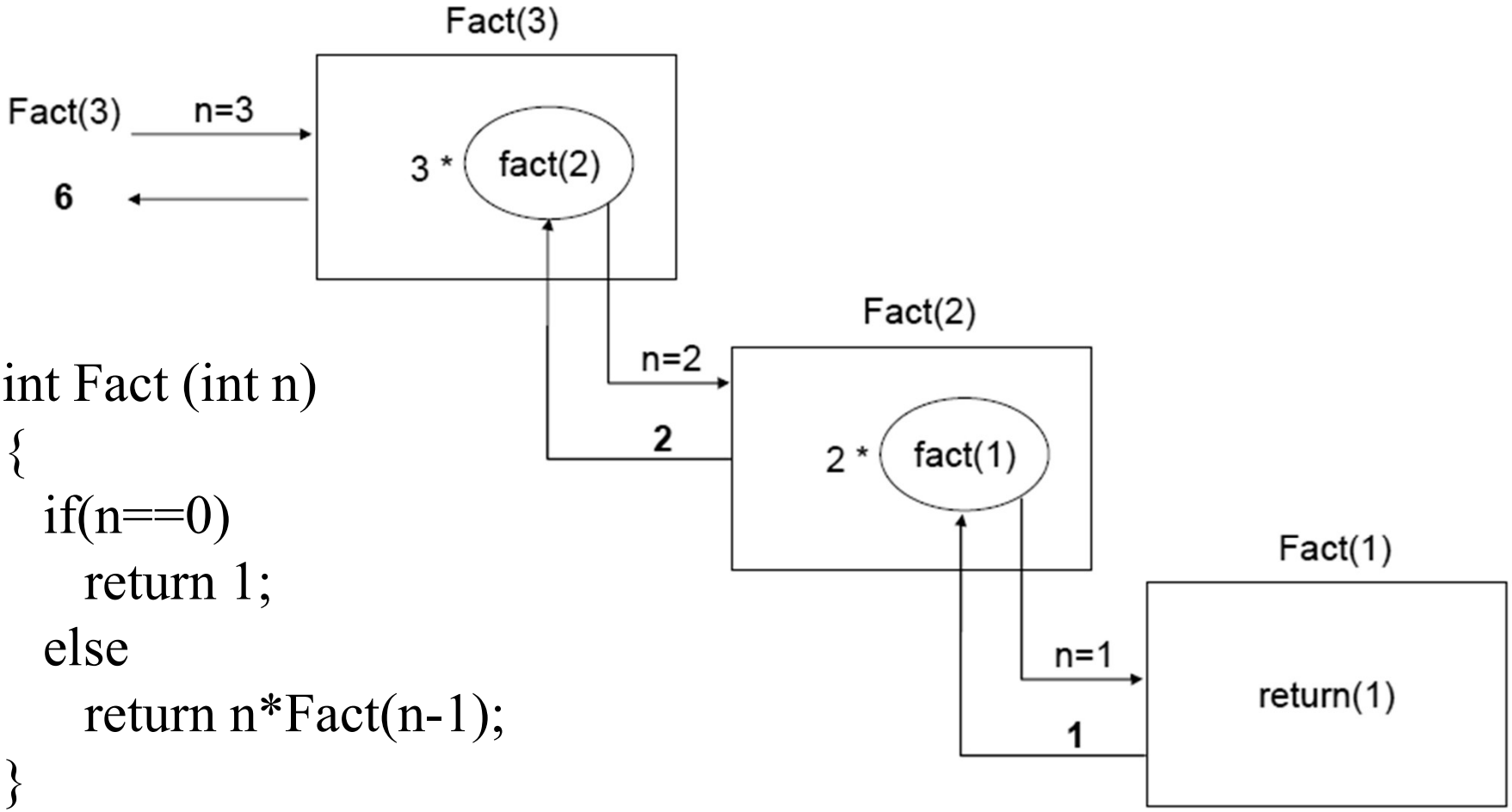
- بعضی از مسائل را هم می توان به صورت بازگشتی و هم غیربازگشتی حل کرد.

- راه حل بعضی از مسائل به طور ذاتی بازگشتی است.

- برای اینکه بتوان از روش بازگشتی برای حل یک مساله استفاده نمود، مساله باید قابلیت خرد شدن به زیرمساله هایی از همان نوع مساله اصلی و اندازه کوچکتر را داشته باشد.

فاکتوریل

$$f(n) = n! = \begin{cases} 1 & if\ n = 1 \\ n \times (n-1) & if\ n > 1 \end{cases}$$

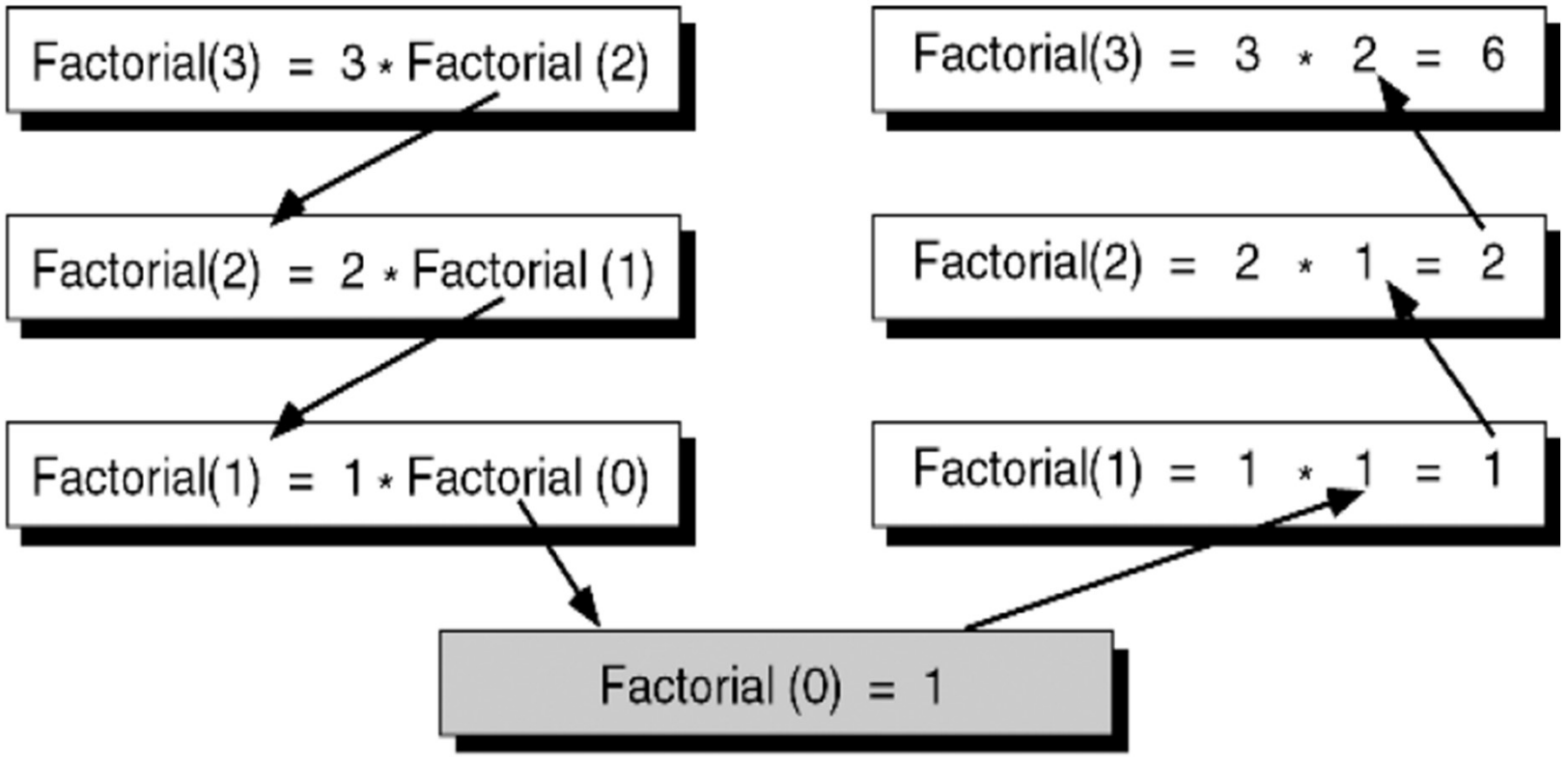



```
int Fact (int n)
{
  if(n==0)
    return 1;
  else
    return n*Fact(n-1);
}
```

فاکتوریل

❖ راه حل بازگشتی یک مساله شامل دو مرحله کلی است:

- ✓ شکستن مساله از بالا به پایین
- ✓ حل مساله از پایین به بالا

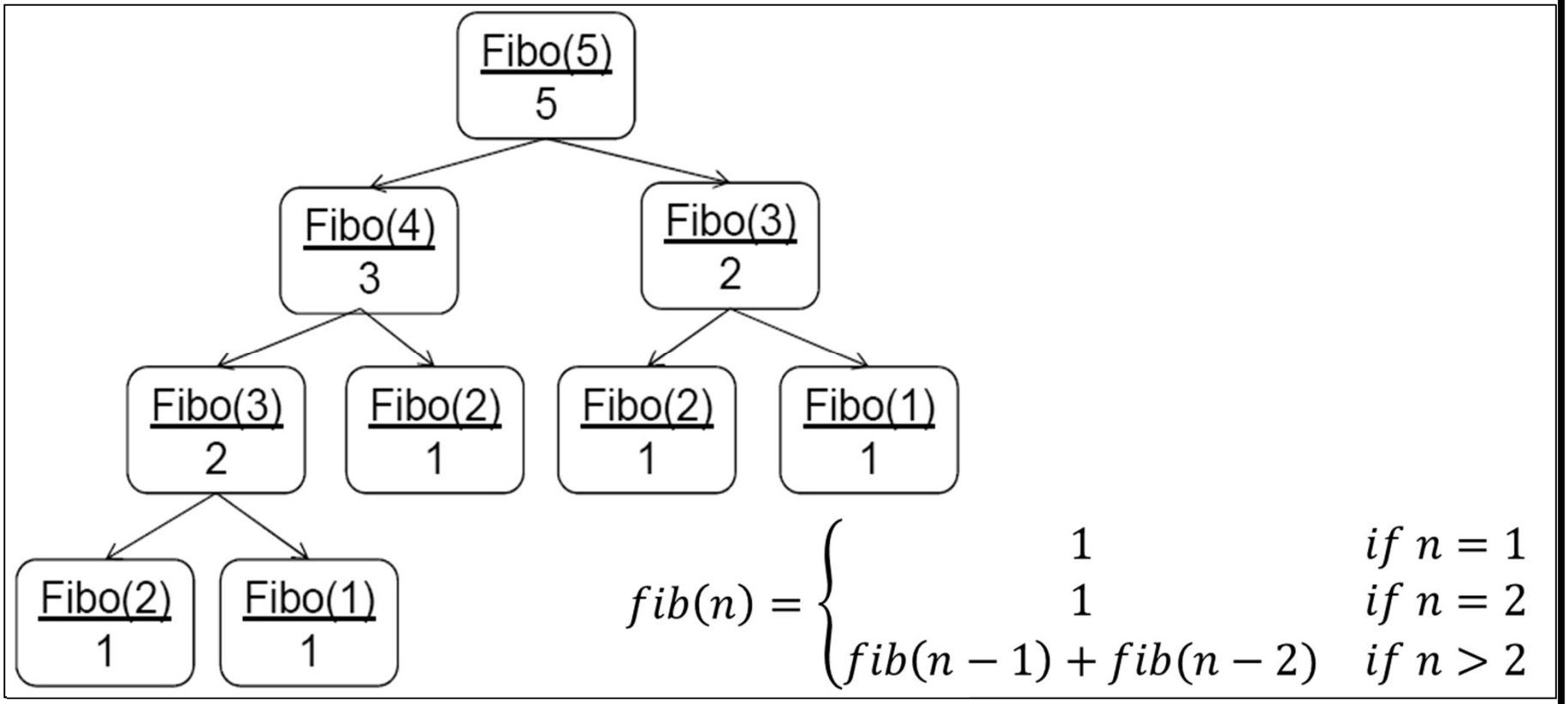


طراحی الگوریتم های بازگشتی

- هر فراخوانی از الگوریتم بازگشتی هم قسمتی از مساله را حل می کند و هم اندازه مساله را کوچک می کند.
- قسمت اصلی راه حل، فراخوانی های بازگشتی است. در هر فراخوانی بازگشتی اندازه مساله کوچکتر می شود.
- عبارتی که مساله را حل می کند، حالت پایه (شرط پایان تکرار) نامیده می شود.
- هر الگوریتم بازگشتی باید یک حالت پایه داشته باشد. بقیه الگوریتم حالت عمومی نام دارد که شامل منطق مورد نیاز برای کاهش اندازه مساله می باشد.
- طراحی الگوریتم بازگشتی
  - مشخص کردن حالت پایه
  - مشخص کردن حالت عمومی (بازگشتی)
  - ترکیب این دو در یک الگوریتم
- محاسبه زمان اجرای الگوریتم بازگشتی
  - زمان حل زیرمساله
  - زمان شکستن مساله به زیرمسائل
  - زمان لازم برای ادغام جواب

محاسبه جمله n ام سری فیبوناچی

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



$$fib(n) = \begin{cases} 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ fib(n-1) + fib(n-2) & if\ n > 2 \end{cases}$$

کارایی الگوریتم بازگشتی فیبوناچی

Fib(5)
= Fib(4) + Fib(3)
= Fib(3) + Fib(2) + Fib(3)
= Fib(2) + Fib(1) + Fib(2) + Fib(3)
= Fib(1) + Fib(0) + Fib(1) + Fib(2) + Fib(3)
= Fib(1) + Fib(0) + Fib(1) + Fib(1) + Fib(0) + Fib(3)
= Fib(1) + Fib(0) + Fib(1) + Fib(1) + Fib(0) + Fib(2) + Fib(1)
= Fib(1) + Fib(0) + Fib(1) + Fib(1) + Fib(0) + Fib(1) + Fib(0) + Fib(1)

- مشکل: محاسبه تکراری جملات
- راه حل: برنامه نویسی پویا (Dynamic programming) و استفاده از آرایه برای ذخیره جملات قبلی

روشهای حل معادلات بازگشتی

- ❖ استفاده از استقرای ریاضی (induction)
- ❖ روش جایگذاری و تکرار
- ❖ استفاده از قضیه اصلی
- ❖ استفاده از درخت بازگشتی
- ❖ حل معادلات بازگشتی خطی با ضرایب ثابت
  - ✓ همگن (استفاده از معادله مشخصه)
  - ✓ غیرهمگن
- ❖ حل معادلات بازگشتی با استفاده از سری مولد
- ❖ حل معادلات بازگشتی به روش تغییر متغیر

حل معادلات بازگشتی با استفاده از استقرای ریاضی

❖ استقرا سه مرحله دارد:

✓ مبنای استقرا: حدس برای شرط اولیه برقرار است.

✓ فرض استقرا: فرض می شود که حدس به ازای n صحیح است.

✓ گام استقرا: باید اثبات شود که حدس برای جمله n+1 نیز برقرار می باشد.

❖ بازگشتی زیر را به روش استقرای ریاضی حل کنید.

$$\begin{cases} t(n) = t(\frac{n}{2}) + 1 \\ t(1) = 1 \end{cases}$$

ابتدا باید یک حل کاندیدا برای معادله فوق پیدا کنیم و سپس درستی این حل کاندیدا را به روش استقرا اثبات نماییم.

با توجه به معادلات می توان گفت حل کاندیدا به صورت $t(n) = \log(n) + 1$ است.

$$t(2) = t(\frac{2}{2}) + 1 = t(1) + 1 = 1 + 1 = 2$$

$$t(4) = t(\frac{4}{2}) + 1 = t(2) + 1 = 2 + 1 = 3$$

$$t(8) = t(\frac{8}{2}) + 1 = t(4) + 1 = 3 + 1 = 4$$

$$t(16) = t(\frac{16}{2}) + 1 = t(8) + 1 = 4 + 1 = 5$$

حل معادلات بازگشتی با استفاده از استقرای ریاضی

- با استفاده از استقرا اثبات می کنیم که این حل درست است.
- مبنا استقرا: برای $n=1$ داریم: $t(1)=1$
- فرض استقرا: فرض کنید برای مقدار دلخواه $n>0$ که n توانی از ۲ است، داشته باشیم: $t(n) = \log(n) + 1$
- گام استقرا: چون رابطه بازگشتی برای n های توانی از ۲ است، عدد بعدی که باید درنظر گرفته شود، $2n$ می باشد: $t(2n) = \log(2n) + 1$

$$t(2n) = t(\frac{2n}{2}) + 1 = t(n) + 1 = \log(n) + 1 + 1$$

$$t(2n) = \log(n) + \log(2) + 1 \qquad \log_x^x = 1$$

$$t(2n) = \log(2n) + 1 \longrightarrow t(2n) = \log(2n) + 1$$

- استقرا اثبات شد بنابراین حل کاندیدا صحیح می باشد.

حل معادلات بازگشتی با استفاده از استقرای ریاضی

❖ بازگشتی زیر را به روش استقرای ریاضی حل کنید.

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

چند جمله را محاسبه می کنیم:

T(0) = 1
T(1) = 1*T(0) = 1*1 = 1
T(2) = 2*T(1) = 2*1 = 2
T(3) = 3*T(2) = 3*2 = 6
T(4) = 4*T(3) = 4*6 = 24
T(k) = K*T(k-1) = k!

حدس می زنیم که: $T(n)=n!$، حدس را با استقرا اثبات می کنیم:

✓ مبنای استقراء: برای $n=0$ ، $T(0)=0!=1$

✓ فرض استقراء: برای مقادیری از n داریم: $T(n) = n!$

باید نشان دهیم: $T(n+1) = (n+1)!$

✓ گام استقراء: $T(n) = n* T(n-1)$

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

$T(n+1) = (n + 1)* n! = (n+1) !$

حل معادلات بازگشتی با استفاده از استقرای ریاضی

❖ بازگشتی زیر را به روش استقرای ریاضی حل کنید.

$$\begin{cases} t(n) = 7t(\frac{n}{2}) \\ t(1) = 1 \end{cases}$$

چند جمله را محاسبه می کنیم:

$$t(2) = 7t(\frac{2}{2}) = 7t(1) = 7$$

$$t(4) = 7t(\frac{4}{2}) = 7t(2) = 7^2$$

$$t(8) = 7t(\frac{8}{2}) = 7t(4) = 7^3$$

$$t(16) = 7t(\frac{16}{2}) = 7t(8) = 7^4$$

نتیجه می شود: $t(n) = 7^{\log n}$

حل معادلات بازگشتی با استفاده از استقرای ریاضی

❖ حل را با استقرا اثبات می کنیم.

✓ مبنای استقراء: برای $n=1$ ، $t(1)=1=7^0=7^{\log 1}$

✓ فرض استقراء: فرض کنید برای هر مقدار دلخواه $n>0$ که n توانی از ۲ است:

$t(n)=7^{\log n}$

✓ گام استقراء: باید نشان داد $t(2n)=7^{\log 2n}$

$2n$ را در رابطه بازگشتی قرار می دهیم:

$$t(2n) = 7t(\frac{2n}{2}) = 7t(n) = 7 \times 7^{\log n} = 7^{1+\log n} = 7^{\log 2+\log n} = 7^{\log(2n)}$$

استقرا ثابت شد. از آنجایی که:

$$7^{\log n} = n^{\log 7} \quad \longleftarrow \quad a^{\log_b c} = c^{\log_b a}$$

$$t(n) = n^{\log 7} \approx n^{2.81}$$

حل معادلات بازگشتی به روش جایگذاری و تکرار

$$\begin{cases} t(n) = t(n-1) + n & \text{for } n > 1 \\ t(1) = 1 \end{cases}$$

❖ روش جایگذاری عکس روش استقرا است:

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

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

$$t(n-2) = t(n-3) + n - 2$$

کلیه مقادیر را در معادله اصلی جایگزین می کنیم.

$$t(2) = t(1) + 2$$

$$t(1) = 1$$

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

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

$$= t(n-3) + n - 2 + n - 1 + n$$

$$= t(1) + 2 + \ldots n - 2 + n - 1 + n$$

$$= 1 + 2 + \ldots n - 2 + n - 1 + n$$

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

$$\Rightarrow T(n) = \frac{n(n+1)}{2} \in O(n^2)$$

حل معادلات بازگشتی به روش جایگذاری و تکرار

$$T(n)=\begin{cases}2 & n=1\\ T(n-1)+5 & n>1\end{cases}$$

T(n) = T(n-۱) + ۵

= T(n-۲) + ۵ + ۵

= T(n-۳) + ۵ + ۵ + ۵

=T(n-۴) + ۵ + ۵ + ۵ + ۵

…

=T(n-k) + ۵k

T(n) = T(n-۱) + ۵

T(n-۱) = T(n-۲) + ۵

T(n-۲) = T(n-۳) +۵

…

T(n-k) = T(n-k-۱)+۵

…

T(۱) = ۲

❖ باید به $T(1)$ برسیم

$$T(n-k)=T(1)\Rightarrow$$

$$n-k=1\Rightarrow k=n-1$$

❖ با جایگذاری مقدار k داریم

$$T(n)=T(n-k)+5k$$

$$T(n)=T(1)+5(n-1)=2+5n-5=5n-3$$

حل معادلات بازگشتی با استفاده از قضیه اصلی

$$T(n) = aT(\frac{n}{b}) + Cn^k$$

اگر a>۱ و b>=۱

$$T(n) = \begin{cases} \theta\left(n^{\log_b^a}\right) & a > b^k \\ \theta\left(n^k \log n\right) & a = b^k \\ \theta\left(n^k\right) & a < b^k \end{cases}$$

❖ مثال

$$T(n) = 4T\left(\frac{n}{2}\right) + n \quad \Rightarrow 1 \quad T(n) \in \Theta\left(n^2\right)$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \quad \Rightarrow 2 \quad T(n) \in \Theta\left(n^2 \lg n\right)$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3 \quad \Rightarrow 3 \quad T(n) \in \Theta\left(n^3\right)$$

حل معادلات بازگشتی با استفاده از قضیه اصلی

❖ حالت کلی

$$T(n) = aT(n/b) + f(n)$$

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \quad \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

حل معادلات بازگشتی با استفاده از درخت بازگشت

- ساخت درخت بازگشت
- تعیین ارتفاع درخت بازگشت بر حسب اندازه ورودی
- تعیین هزینه هر سطح از درخت بازگشت
- محاسبه مجموع هزینه های تمامی سطوح

حل معادلات بازگشتی خطی

❖ بازگشتی خطی همگن درجه k:

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$$

✓ این بازگشتی معادله خطی همگن با ضرایب ثابت (a_i) از مرتبه k ام نام دارد. که نیازمند k شرط اولیه است.

✓ رابطه زیر یک رابطه بازگشتی مرتبه دوم همگن نامیده می شود:

$$T(n)=aT(n-1)+bT(n-2) \qquad n>1$$
$$T(0)=C_0 \quad , T(1)=C_1$$

✓ a و b ثابت هستند.

❖ بازگشتی خطی ناهمگن

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = f(k)$$

این بازگشتی معادله خطی ناهمگن با ضرایب ثابت (a_i) از مرتبه k ام نام دارد.

حل معادلات بازگشتی خطی همگن با استفاده از معادله مشخصه

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$$

❖ جواب معادله به فرم x^n می باشد، پس با قرار دادن $t(n)=x^n$ داریم:

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

❖ حال طرفین را بر x^{n-k} (کوچکترین توان x) تقسیم می کنیم و خواهیم داشت:

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

❖ به این معادله معادله مشخصه رابطه بازگشتی گویند که k جواب $x_1, x_2, \ldots, x_k$ دارد و در حالت کلی نهایی رابطه بازگشتی با توجه به این جوابها به صورت زیر است:

$$t(n) = c_1 x_1^n + c_2 x_2^n + \ \ldots \ + c_k x_k^n$$

❖ که در آنها c_i ها ثابت هستند و می توان آنها را از طریق مقدار اولیه بدست آورد.

حل معادلات بازگشتی خطی همگن با استفاده از معادله مشخصه

❖ مثال:

$$\begin{cases} t(n)-3t(n-1)-4t(n-2)=0 & \text{for } n>1 \\ t(0)=0 \\ t(1)=1 \end{cases}$$

❖ معادله مشخصه رابطه بازگشتی را بدست می آوریم. با قرار دادن $T(n)=x^n$ داریم:

$$x^n-3x^{n-1}-4x^{n-2}=0$$

❖ طرفین را بر x^{n-2} تقسیم می کنیم. معادله مشخصه را حل می کنیم:

$$x^2-3x-4=(x-4)(x+1)=0$$

❖ ریشه های آن عبارتند از:

$$\begin{cases} x-4=0 \rightarrow x=4 \\ x+1=0 \rightarrow x=-1 \end{cases}$$

❖ پس از اعمال قضیه معادله زیر به عنوان حل رابطه بازگشتی بدست می اید:

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

❖ مقادیر ثوابت از طریق اعمال مقادیر اولیه در رابطه فوق بدیست می آید:

$$\begin{cases} t(0)=0=c_1 4^0+c_2(-1)^0 \\ t(1)=1=c_1 4^1+c_2(-1)^1 \end{cases} \longrightarrow \begin{cases} c_1+c_2=0 \\ 4c_1-c_2=1 \end{cases} \longrightarrow \begin{cases} c_1=\frac{1}{5} \\ c_2=\frac{-1}{5} \end{cases}$$

$$t(n)=\frac{1}{5}4^n-\frac{1}{5}(-1)^n$$

حل معادلات بازگشتی خطی ناهمگن

❖ قضیه: بازگشتی ناهمگن زیر را درنظر بگیرید:

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

✓ این شکل بازگشتی می تواند به بازگشتی همگنی تبدیل شود که معادله مشخصه آن به صورت زیر است:

$$(a_0 r^k + a_1 r^{k-1} + \ldots + a_k r^0)(r - b)^{d+1} = 0$$

✓ که d برابر با درجه $p(n)$ اشت.

✓ اگر بیش از یک جمله مانند $b^n p(n)$ در سمت راست وجود داشته باشد، هر یک از آنها در معادله مشخصه ظاهر می شود.

❖ حل بازگشتی

$$a_n = a_n^h + a_n^p$$

حل معادلات بازگشتی خطی ناهمگن

$t_n - ۳t_{n-۱} = ۴^n(۲n+۱)$ for $n>۱$

$t_۰ = ۰$, $t_۱ = ۱۲$

معادله مشخصه: $(r-۳)(r-۴)^{۱+۱} = ۰$

حل بازگشتی: $t_n = c_۱۳^n + c_۲۴^n + c_۳n۴^n$

$$=> t_n = ۲۰(۳^n) - ۲۰(۴^n) + ۸n۴^n$$

حل معادلات بازگشتی با روش تغییر متغیر

❖ با تغییر متغیر می توان رابطه بازگشتی را به یک رابطه ساده تر تبدیل کرد.

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

$$n = ۲^m$$

$$T(۲^m) = T(۲^{m/۲}) + ۱$$

$$T(۲^m) = S(m)$$

$$S(m) = S(m/۲) + ۱$$

با روش جایگذاری حل می شود:

$$S(۱) = c \rightarrow S(m) = c + \lg m$$

در نهایت داریم:

$$m = \lg n \rightarrow T(n) = c + \lg \lg n$$

فصل سوم

روش تقسیم و حل (Divide & Conquer)

- نمونه ای از یک مساله را به صورت بازگشتی به تعدادی نمونه کوچکتر تقسیم می کند (تا زمانی که راه حل نمونه های کوچکتر به سادگی قابل تعیین باشند) و از حل نمونه های کوچکتر به حل نمونه بزرگ می رسد.
- تقسیم (divide): مسئله به تعدادی زیر مسئله تقسیم می شود.
- غلبه (conquer): زیرمسئله ها به صورت بازگشتی حل می شوند.
- ترکیب (combine): در صورت لزوم، ترکیب حل زیرمسئله ها برای یافتن جواب مسئله کلی
- مساله ها با تقسیمات متوالی آنقدر کوچک می شوند تا دیگر مشکلی برای حل آنها نداشته باشیم.
- روش تقسیم و غلبه یک رهیافت بالا به پایین (top-down) است که توسط روتین های بازگشتی به کار می رود.

روش تقسیم و حل (Divide & Conquer)

```
Procedure D&C ( p,q )
if small(p,q) then
        return G(p,q)
else
        m ← divide (p,q)
        return combine(D &C(p,m), D &C(m+1,q))
end if
end.
```

small(p,q): تصمیم گیری در مورد اینکه آیا مساله به اندازه کافی کوچک شده است یا خیر.

G(p,q): مساله کوچک شده را حل می کند.

در هر بار فراخوانی مساله به مسائل کوچکتر تقسیم می شود

combine: جواب تمام زیرمسائل را با هم ترکیب میکند.

تقسیم و حل یافتن مقدار max و min

غیر بازگشتی (تکراری)

```
Maxmin (A , n , max , min)
{
max = min = a[1] ;
for (i = 2 ; i < = n ; i + + )
    {
      if (a[i] > max)
            max = a[i] ;
      if (a[i] < min)
           min = a[i] ;
    }
}
```

```
Maxmin (A , n , max , min)
{
max = min = a[1] ;
for (i = 2 ; i < = n ; i + + )
   {
     if (a[i] > max)
          max = a[i] ;
     else if (a[i] < min)
          min = a[i] ;
   }
}
```

بهینه شده

تقسیم و حل یافتن مقدار max و min

غیر بازگشتی (تکراری)

i	مقایسه انجام شده	نتیجه مقایسه	مقدار Max جدید	مقدار MIN جدید
1	-	-	**22**	**22**
2	30 > 22	T	**30**	22
	30 < 22	F	30	22
3	25 > 30	F	30	22
	25 < 22	F	30	22
4	11 > 30	F	30	22
	11 <22	T	30	**11**
5	32 > 30	T	**32**	11
	32 < 11	F	32	11

۲۲ ۳۰ ۲۵ ۱۱ ۳۲

❖ در الگوریتم غیر بازگشتی بهترین حالت زمانی است که لیست صعودی مرتب باشد و بدترین حالت زمانی است که لیست نزولی مرتب باشد .

تقسیم و حل یافتن مقدار max و min

```
Maxmin (A, i , j , max , min) {
if (i == j)
    {
          max = A[i] ;
          min = A[i];
     }                                  small1(p,q)
else
    if (j == i+ 1)
      if (A[i] < A[j])
          {
            max = A[j] ;
            min = A[i] ;
          }
       else
          {
            max = A[i] ;
            min = A[j] ;
          }                             small2(p,q)
   else
   {
     mid = [(i + j)/2] ;                divide
     Maxmin (A, i , mid , fmax , fmin) ;
     Maxmin (A, mid + 1 , j , gmax , gmin) ;   recursive
     if (fmax> gmax) max = fmax;
     else max=gmax;
     if (fmin < gmin) min = fmin ;      combine
     else min=gmin;   }
}
```

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2 + T(\frac{n}{2}) + T(\frac{n}{2}) & \text{else} \end{cases}$$

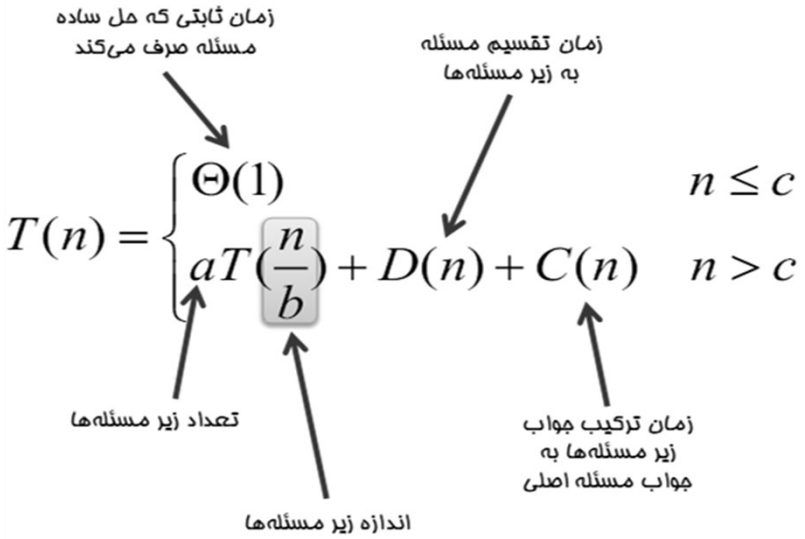
$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2 + 2T(\frac{n}{2}) & else \end{cases}$$

$T(n) = (3/2)\, n - 2$

$T(n)=O(n)$

تقسیم و حل یافتن مقدار max و min

نام الگوریتم	روش حل مساله	بهترین حالت	بدترین حالت	میانگین
max-min	غیر بازگشتی	$2(n-1)$	$2(n-1)$	$2(n-1)$
max-min	غیر بازگشتی بهینه	$(n-1)$	$2(n-1)$	کمتر از $2(n-1)$
max-min	بازگشتی (تقسیم و حل)	$\frac{3n}{2}-2$	$\frac{3n}{2}-2$	$\frac{3n}{2}-2$



تقسیم و حل جستجوی دودویی (binary search)

```
void binsearch ( int n,
                 const keytype S[ ],
                 keytype x,
                 index& location )
{
    index low, high, mid;

    low = 1 ;  high = n ;
    location = 0 ;
    while ( low <= high && location == 0) {
        mid = ⌊ ( low + high ) / 2 ⌋ ;
        if ( x == S[mid] )
            location = mid ;
        else if ( x < S[mid] )
            high = mid – 1 ;
        else
            low = mid + 1 ;
    }
}
```

در بدترین حالت W(n)

عمل اصلی: تعداد مقایسه

اندازه ورودی: تعداد عناصر آرایه

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

حل:

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1 & n > 1 \\ 1 & n = 1 \end{cases}$$

$$T(n)=O(\log n)$$

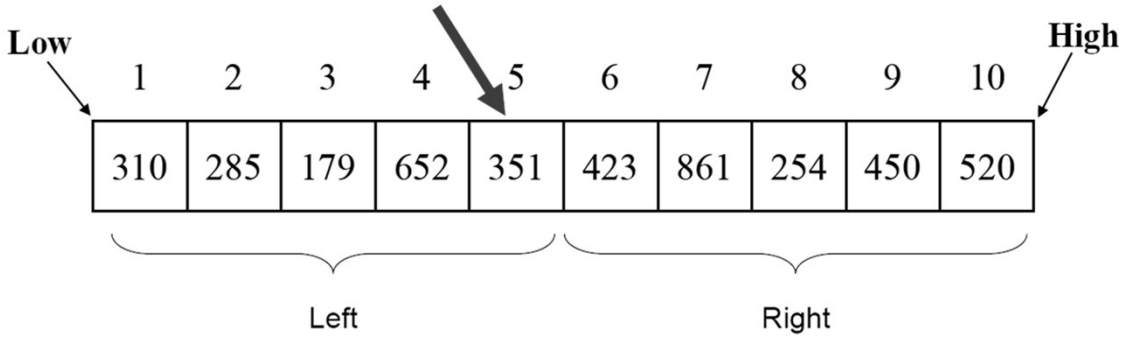
تقسیم و حل مرتب سازی ادغامی (merge sort)

- روش Merge sort به نوع مقادیر ورودی وابستگی ندارد و از استراتژی divide و conquer استفاده می کند. در اکثر پیاده‌سازی‌ها این الگوریتم پایدار(stable) می‌باشد. بدین معنی که این الگوریتم ترتیب ورودی‌های مساوی را در خروجی مرتب شده حفظ می‌کند. این الگوریتم نیازمند فضای اضافی متناسب با تعداد رکوردهایی که باید مرتب شوند می باشد.
- تقسیم (divide): آرایه n عنصری به دو زیر آرایه به اندازه n/2.
- غلبه (conquer): هر یک از زیر آرایه ها با مرتب سازی آن. اگر زیرآرایه به اندازه کافی کوچک نبود، برای حل آن به روش بازگشتی عمل می شود.
- ترکیب (combine): حل زیرآرایه ها با ادغام آن ها در یک آرایه مرتب

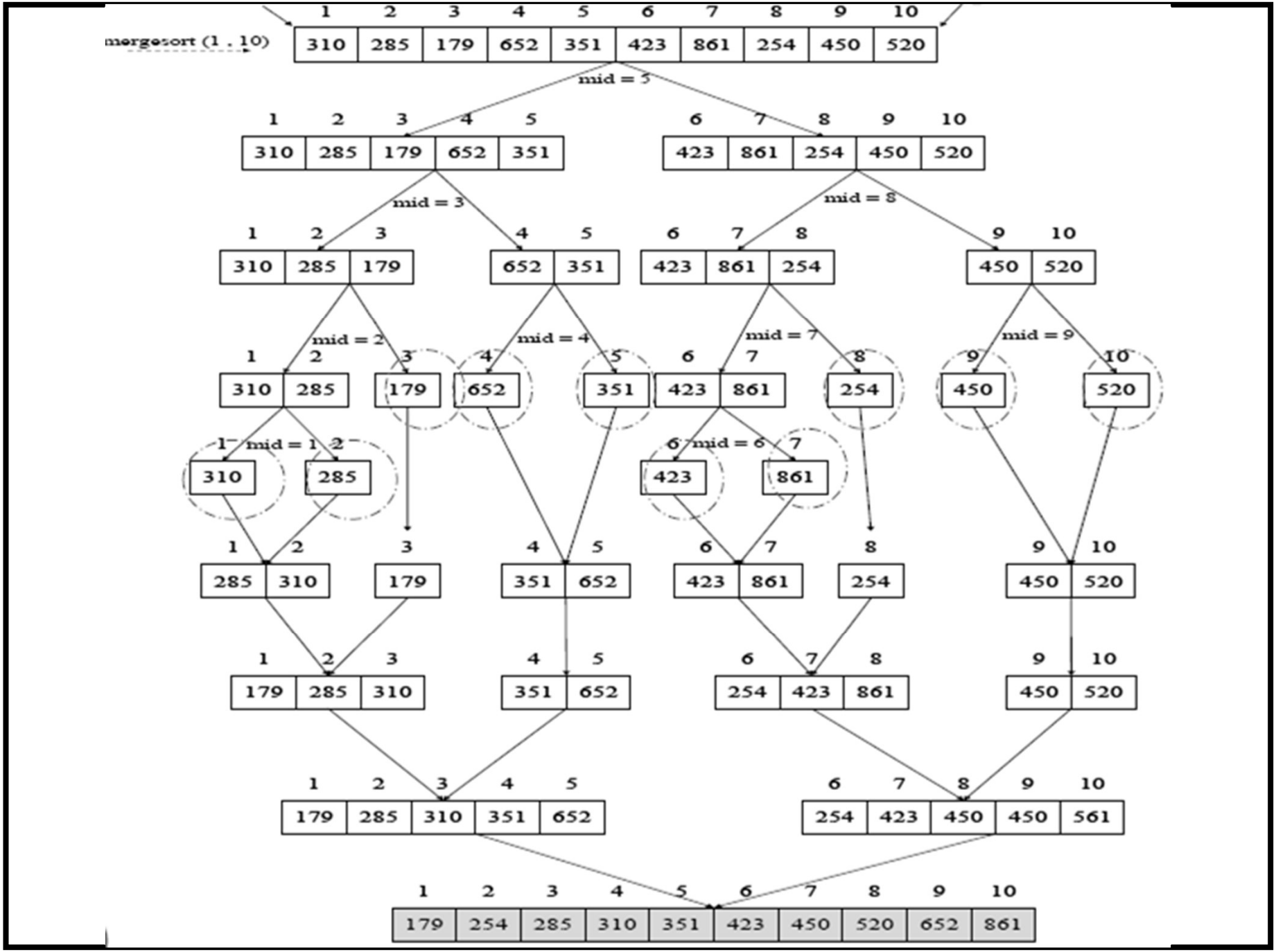
6 5 3 1 8 7 2 4

تقسیم و حل مرتب سازی ادغامی (merge sort)

❖ لیست نامرتبی با n عنصر داریم و می خواهیم لیست را به صورت صعودی مرتب کنیم.




```
Alg mergesort (low , high)
{
  if (low < high)                          // !small(p)
  {
        mid := ⌊(low + high) / 2⌋ ;        // Divide
       mergsort (low , mid) ;              // recursive (left)
       mergesort (mid + 1  ,  high) ;      // recursive (right)
       merge (low , mid , high) ;          // combine
  }
}
```


mergesort (1 , 10)
1 2 3 4 5 6 7 8 9 10
310 285 179 652 351 423 861 254 450 520
mid = 5
1 2 3 4 5
310 285 179 652 351
6 7 8 9 10
423 861 254 450 520
mid = 3
mid = 8
1 2 3
310 285 179
4 5
652 351
6 7 8
423 861 254
9 10
450 520
mid = 2
mid = 4
mid = 7
mid = 9
1 2
310 285
3
179
4
652
5
351
6 7
423 861
8
254
9
450
10
520
1
310
mid = 1
2
285
6
423
mid = 6
7
861
1 2
285 310
3
179
4 5
351 652
6 7
423 861
8
254
9 10
450 520
1 2 3
179 285 310
4 5
351 652
6 7 8
254 423 861
9 10
450 520
1 2 3 4 5
179 285 310 351 652
6 7 8 9 10
254 423 450 450 561
1 2 3 4 5 6 7 8 9 10
179 254 285 310 351 423 450 520 652 861

تقسیم و حل مرتب سازی ادغامی (merge sort)

❖ پیچیدگی محاسباتی الگوریتم ادغام در هر سه حالت (n log n) است.

$$T(n) = \begin{cases} c & n = 1 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + n & \text{else} \end{cases}$$

$$T(n) = n[1 + \log n] = n + n \log n \quad \Rightarrow \quad T(n) \in O(n \log n)$$

فصل چهارم

الگوریتمهای حریصانه(Greedy algorithms)

- الگوریتم حریصانه به ترتیب عناصر را گرفته، هر بار آن عنصری را که طبق معیاری مشخص «بهترین» به نظر می رسد، بدون توجه به انتخابهایی که قبلا انجام داده یا درآینده انجام خواهد داد، برمی دارد.
- الکوریتم حریصانه کار را با یک مجموعه تهی آغاز کرده به ترتیب عناصری به مجموعه اضافه می کند تا این مجموعه، حلی برای نمونه ای از یک مسئله را نشان می دهد.
- غالبا برای حل مسائل بهینه سازی «optimization» به کار می رود.
- دنباله ای از انتخاب ها را پیش رو داریم. در هر مرحله بخشی از جواب به دست می آید. نتیجه نهایی مجموعه ای از داده ها است که ممکن است ترتیب آنها مهم باشد. جواب نهایی تابع هدف را بهینه (ماکسیمم/مینیمم) می نماید. در این روش آینده نگری وجود ندارد و به وضعیت جاری بیشتر توجه می شود. بهینگی ممکن است کلی نباشد، محلی باشد.
- تصمیم اتخاذ شده در مورد انتخاب یا عدم انتخاب یکی از داده های ورودی به عنوان مولفه ای از جواب قطعی و غیر قابل بازگشت است.

الگوریتمهای حریصانه(Greedy algorithms)

❖ اجزای الگوریتم حریصانه

۱- مجموعه کاندید (candidate set): مجموعه جواب هایی است که در هر مرحله از مساله قابل انتخاب است. مجموعه اولیه که مولفه های جواب از بین آنها انتخاب می شود. در هر مرحله از اجرای الگوریتم بخشی از جواب را به دست می آوریم.

۲- تابع انتخاب (selection function): تابعی برای انتخاب مولفه بعدی. معیاری است که در هر مرحله می گوید چه عنصری از مجموعه کاندید انتخاب شود. این انتخاب براساس یک معیار حریصانه که به طور محلی بهترین جواب را در هر لحظه انتخاب می کند، شکل می گیرد.

۳- تابع امکان سنج (feasible function): تابعی است که بررسی می کند آیا عضو انتخاب شده می تواند جزء جواب باشد. بررسی می شود که آیا با انتخاب آن مولفه و کجموعه انتخاب های قبلی امکان رسیدن به جواب وجود دارد یا خیر.

۴- تابع ارزیاب (objective function): با افزودن هر عنصر به مجموعه جواب بررسی می کنیم اگر جواب مساله حاصل شده است. جمع آوری و بازگرداندن جواب مساله وظیفه آن است.

الگوریتمهای حریصانه(Greedy algorithms)

```
function Greedy(c:set)
  S←∅
  while (c≠ ∅ and not solution(S))
        x←selection (c)
        c←c-{x}
       if feasible (S ∪ {x}) then S←S ∪ {x}
  if solution(S) return S
     else return 'no solution'
```

مسئله خرد کردن پول

- هدف برگرداندن باقیمانده پول با حداقل تعداد سکه ها می باشد.
- در ابتدا هیچ سکه ای در مجموعه جواب نداریم.
- سکه با ارزش بیشتر از مجموعه کاندید انتخاب می شود. (تابع انتخاب)
- بررسی می کنیم با افزودن این سکه به بقیه پول ، جمع کل آنها از چیزی که باید باشد بیشتر می شود یا خیر. (تابع امکان سنج)
- اگر با افزودن این سکه بقیه پول از میزان لازم بیشتر نشود، این سکه به مجموعه اضافه می شود.
- بررسی می شود که آیا تمام پول پرداخت شده است یا خیر. (تابع ارزیاب)
- اگر هنوز مقداری مانده بود مجددا از مرحله تابع انتخاب فرایند تکرار می شود.
- این تکرار تا زمانی انجام می شود که با سکه های موجود بقیه پول به طور کامل پرداخت گردد و یا اینکه امکان پرداخت با این سکه ها وجود نداشته باشد.

مسئله خرد کردن پول

```
const c={1,5,10,25,100}
S←∅
s=0
while (s≠ n)
        x←largest item in c such that s+x≤n
        if there is no such item return “no solution”
        S←S ∪ {x}
        s=s+x
Return S
```

مسئله خرد کردن پول

- الگوریتم حریصانه همواره جواب بهینه نمی دهد.
- پرداخت ۱۸ سنت با سکه های ۱، ۶ و ۷ سنتی
  - حریصانه : دو سکه ۷ سنتی و چهار سکه ۱ سنتی
  - بهینه : سه سکه ۶ سنتی
- پرداخت ۱۶ سنت با سکه های ۱۲، ۱۰، ۵ و ۱ سنتی
  - حریصانه : یک سکه ۱۲ سنتی و چهار سکه ۱ سنتی
  - یک سکه ۱۰ سنتی، یک سکه ۵ سنتی و یک سکه ۱ سنتی

مسئله کوله پشتی (Knapsack)

- در این مساله امکان انتخاب کسری از هر شی وجود دارد. اگر کسر $0 \leq x_i \leq 1$ از شی i انتخاب شود ارزش $p_i x_i$ بدست می آید.
- تمام کیسه ها وزنشان بیشتر از گنجایش کوله پشتی است. تعداد کیسه ها n، ارزش هر کیسه p_i، وزن هر کیسه w_i و گنجایش کوله پشتی M است.

- هدف

$$\sum_{i=1}^{n} p_i x_i$$ بیشینه کردن رابطه مقابل

$$\sum_{i=1}^{n} w_i x_i < M$$ به شرط اینکه:

- راه حل های حریصانه:

۱- مرتب کردن اشیا به ترتیب بیشترین ارزش و انتخاب آنها

۲- مرتب کردن اشیا به ترتیب کمترین وزن و انتخاب آنها

۳- مرتب کردن اشیا به ترتیب بیشترین مقدار ارزش واحد وزن (p_i / w_i) و انتخاب آنها

مسئله کوله پشتی (Knapsack)

❖ ظرفیت کوله پشتی ۲۰ می باشد.

وزن	۱۰	۱۵	۱۸
ارزش	۱۵	۲۴	۲۵

❖ جدول زیر نتیجه سه راه حل مختلف بیان شده را نشان می دهد.

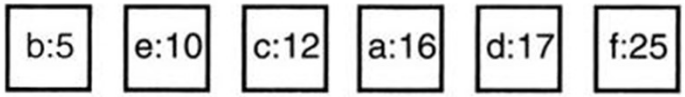
	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1	(1,2/15,0)	20	28.2
2	(0,2/3,1)	20	31
3	(0,1,1/2)	20	31.5

❖ روش سوم (انتخاب بر اساس بیشترین ارزش هر واحد وزن شیئی) جواب بهینه را می دهد.

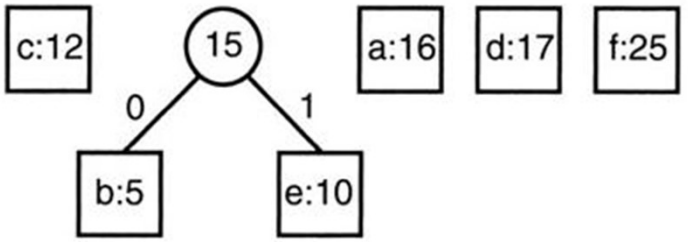
کدینگ هافمن (Huffman code)

- کدگذاری هافمن یک الگوریتم کدگذاری برای فشرده سازی اطلاعات است. برای هر کاراکتر یک کد باینری به طول متغیر استفاده می شود.
- رشته ای که نشان دهنده یک کاراکتر خاص است نمی تواند پیشوند رشته دیگر که نشان دهنده یک کاراکتر دیگر است باشد.
- کاراکتر های با تکرار بیشتر با رشته های بیتی کوتاهتری نسبت به آنهایی که کاربردشان کمتر است نشان داده می شود در نتیجه حجم کمتر اطلاعات ارسال می شود.
- دو کاراکتری که کمترین وزن را دارند ادغام کرده و در لیست قرار می دهیم و دو کاراکتر را از لیست حذف می کنیم.

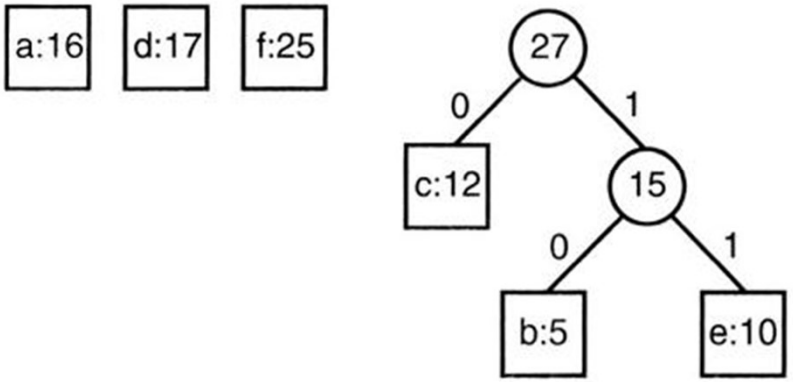
کدینگ هافمن (Huffman code)



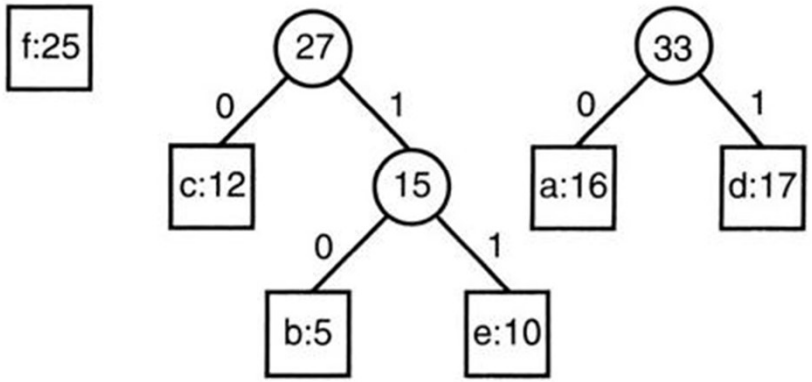
(0)



(1)

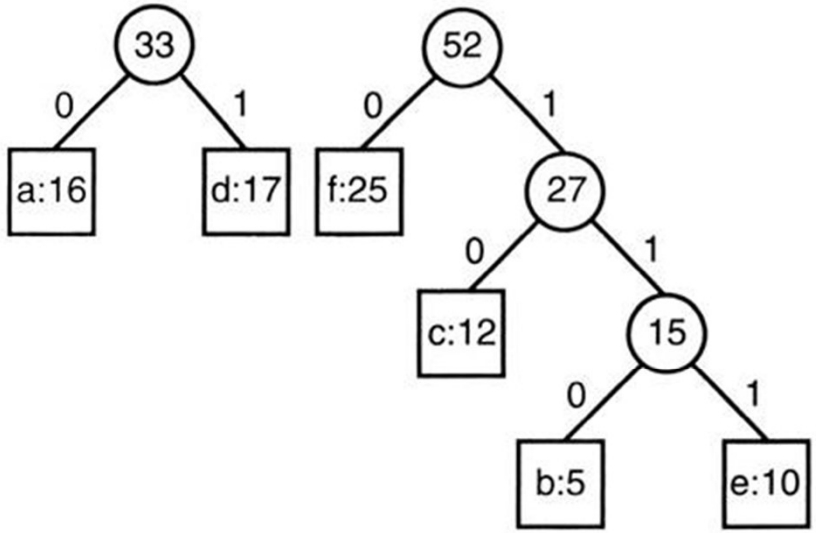


(2)

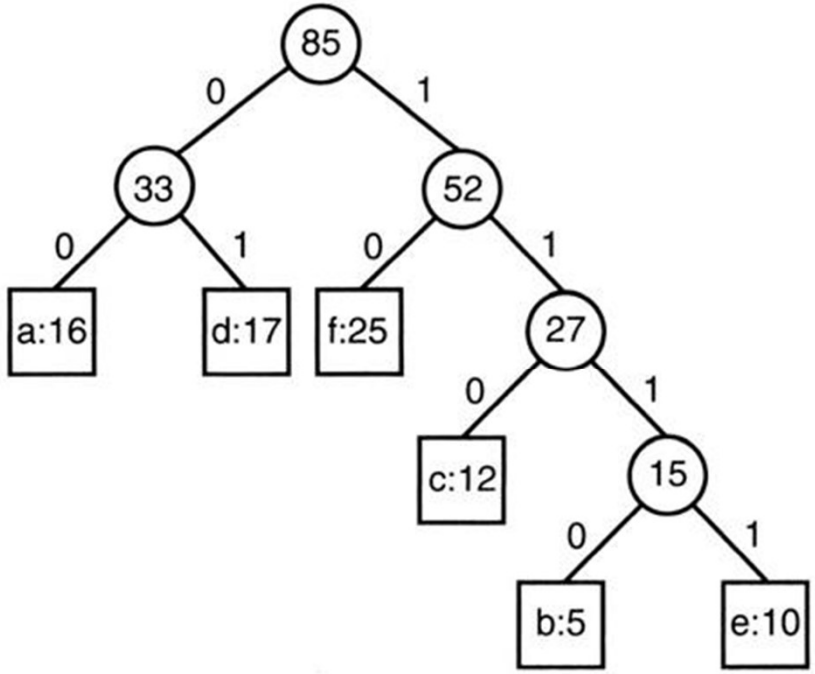


(3)

کدینگ هافمن (Huffman code)



(4)



(5)

الگوریتم های درخت پوشای مینیمم (Minimum Spanning Tree) MST

- مسئله درخت پوشای مینیمم از مسائل بهینه سازی است.
- الگوریتم های درخت پوشای مینیمم در گراف
- الگوریتم پریم prim
- الگوریتم کراسکال kruskal
- نکته:

الگوریتم های کروسکال و پریم همواره درخت های پوشای کمینه را ایجاد می کنند.

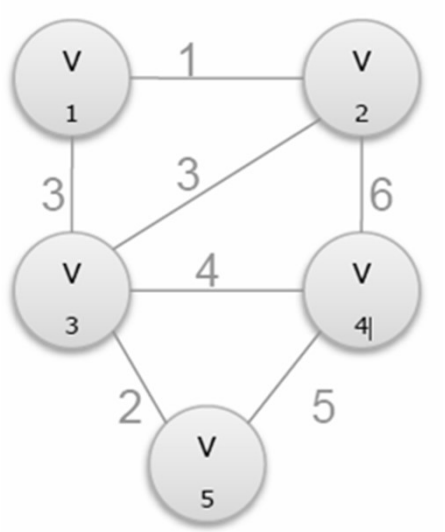
- الگوریتم های پیدا کردن کوتاه ترین مسیر در گراف
- الگوریتم دایجسترا Dijkstra

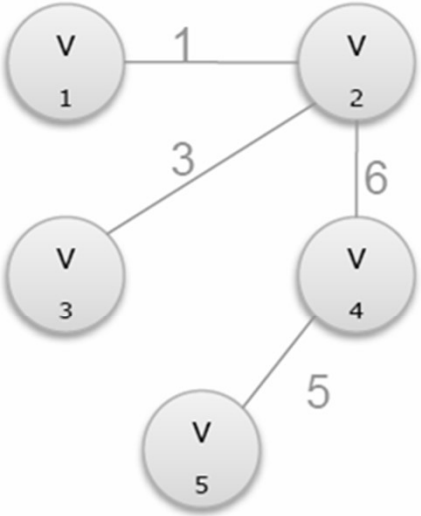
درخت پوشای مینیمم (Minimum Spanning Tree)

- مسئله یافتن درختی در یک گراف که شامل تمام رئوس آن گراف باشد (پوشا) و مجموع وزن یالها در آن حداقل باشد.
- کاربردهای مسئله:
  - چند شهر را با جاده به یکدیگر وصل می کنیم به طوری که مردم بتوانند از هر شهر به شهر دیگر سفر کنند (بین هر دو شهر حداقل یک مسیر وجود داشته باشد) و هزینه حداقل باشد.
- خصوصیات MST
  - شامل n-1 یال می باشد
  - بدون دور است
  - ممکن است برای یک گراف چندین MST وجود داشته باشد ولی هزینه همه آنها یکسان است.

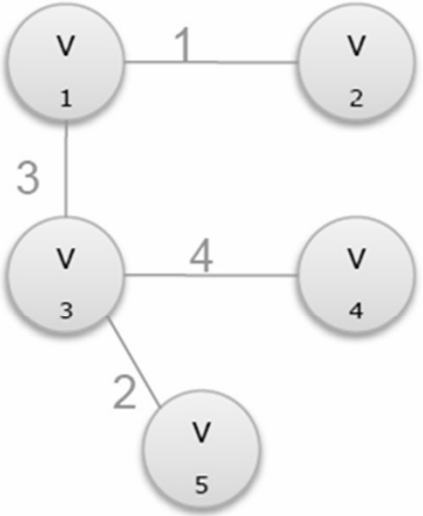
درخت پوشا

❖ نظریه Cayley: برای گراف کامل با n گره تعداد n^{n-2} درخت پوشا وجود دارد.





هزینه ۱۵



هزینه ۱۰

الگوریتم پریم(prim)

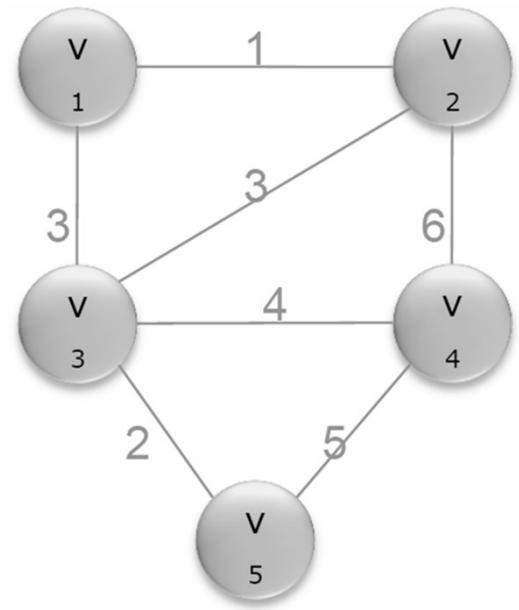
- مبنای کار الگوریتم انتخاب بهترین راس در هر لحظه است.
- در این روش رئوس به دو دسته پردازش شده و پردازش نشده تقسیم می شوند.
- در هر مرحله بررسی می شود کدام راس به مجموعه رئوس پردازش شده ارتباط دارند، کمترین انتخاب شده اضافه می شود.
- یالها به مجموعه T و راس ها به مجموعه P اضافه می شوند.

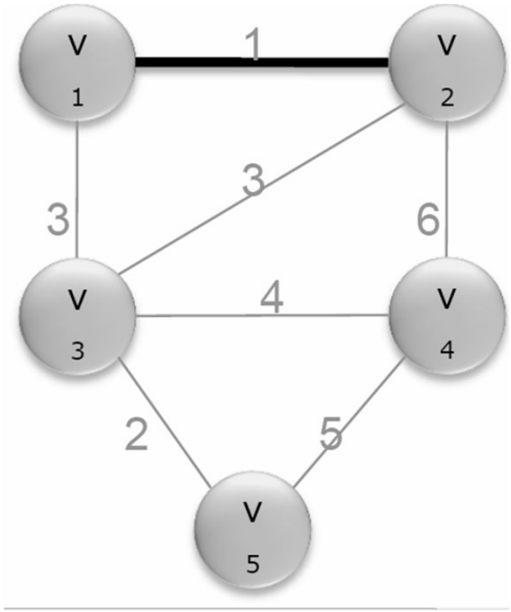
```
Procedure prim (V, E, T)
  T← ∅
  P ← {1}
  while |T|<n-1 do
     select e=(u,v) with minimum cost for each E such that u ∈ P and v ∉ P
     if there is no such edge then exit
     add e to T
     add v to P
  if |T|<n-1 then write 'no spanning tree'
```

الگوریتم پریم(prim)

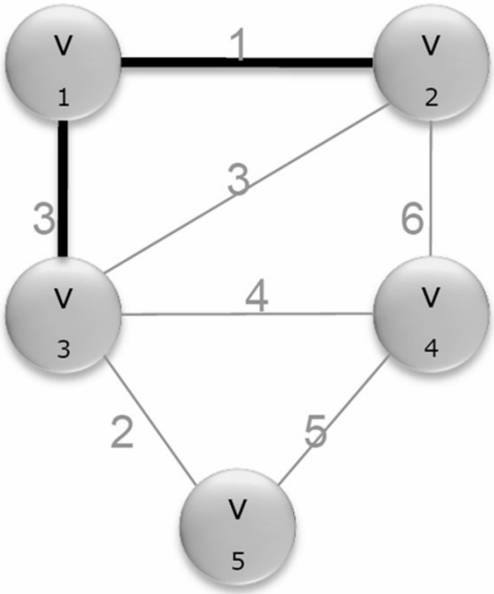
- استفاده از نمایش گراف به صورت ماتریس مجاورت است که درآن به دنبال آرایه‌ای از وزنها باشیم و یال‌های با وزن کمینه را به مجموعه خود بیفزائیم. این روش $o(V^2)$ زمان می‌برد.
- الگوریتم پریم بااستفاده از داده ساختار heap دودوئی ساده و نمایش لیست مجاورت می‌تواند در زمان $O(E \log V)$ اجرا شود که در آن E تعداد یالها و Vتعداد رئوس است.

الگوریتم پریم(prim)

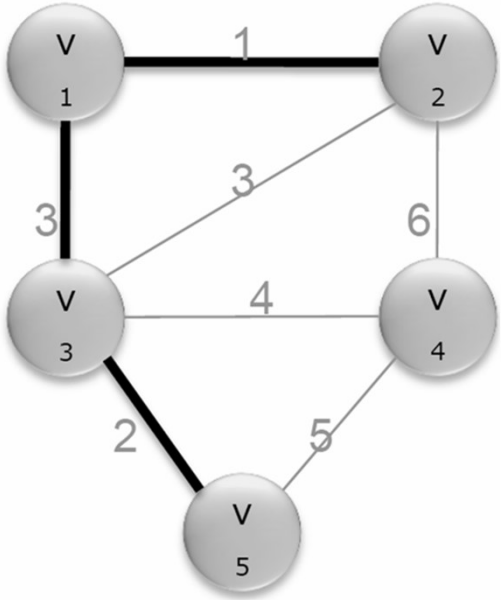




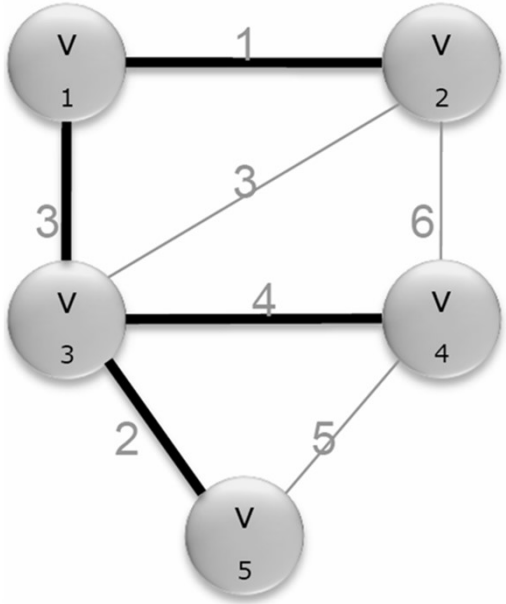
T= {{v1,v2}}
P={v1,v2}



T= {{v1,v2},{v1,v3}}



T= {{v1,v2},{v1,v3},{v3,v5}}



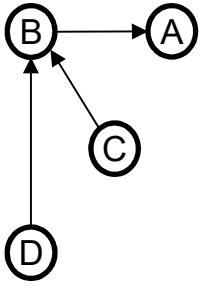
T= {{v1,v2},{v1,v3},{v3,v5},{v3,v4}}

انواع نمایش گراف

ماتریس همجواری adjacency matrix، لیست همجواریadjacency list

❖ ماتریس همجواری

در گراف جهت دار اگر یال وجود داشته باشد آن گاه A(i,j)=1 درغیراینصورت ۰ خواهد بود. اگر از i به j یال نباشد A[i][j]=0 و اگر از i به j یال باشد A[i][j]=1.
ماتریس همجواری گراف های بدون جهت روی قطر اصلی متقارن است اما در گراف های جهت دار ممکن است متقارن نباشد.

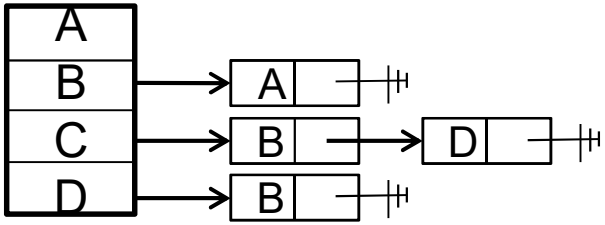


	A	B	C	D
A	0	0	0	0
B	1	0	0	0
C	0	1	0	1
D	0	1	0	0

فضای مصرفی ماتریس همجواری $o(|V|^2)$

❖ لیست همجواری

مشکل ماتریس همجواری این است که این ماتریس اغلب اسپارس است به این معنی که بسیاری از عناصر آن صفر است و نیاز به مقدار زیادی حافظه دارد. به همین دلیل از لیست همجواری استفاده می شود. فضای مصرفی لیست همجواری $o(|V|+|E|)$ است.



الگوریتم کراسکال(Kruskal)

- این الگوریتم در هر لحظه بهترین یال را انتخاب می کند.
- در این الگوریتم لبه ها یکی یکی انتخاب می شوند تا کل درخت ساخته شود.
- در ابتدا مجموعه یالها $T= \emptyset$ ایجاد می شود.
- یالها بر اساس وزن به صورت صعودی مرتب می شوند.
- یال کمترین وزن انتخاب می شود.
- امکان سنجی: اگر یال انتخاب شده حلقه ایجاد نکند و دو مجموعه مجزا را متصل کند، به مجموعه T اضافه می شود.
- تا زمانی که تمام رئوس در یک مجموعه قرار نگرفته اند مراحل تکرار می شوند.
- هزینه پیاده سازی در بهترین حالت $O(E \log V)$ می باشد.

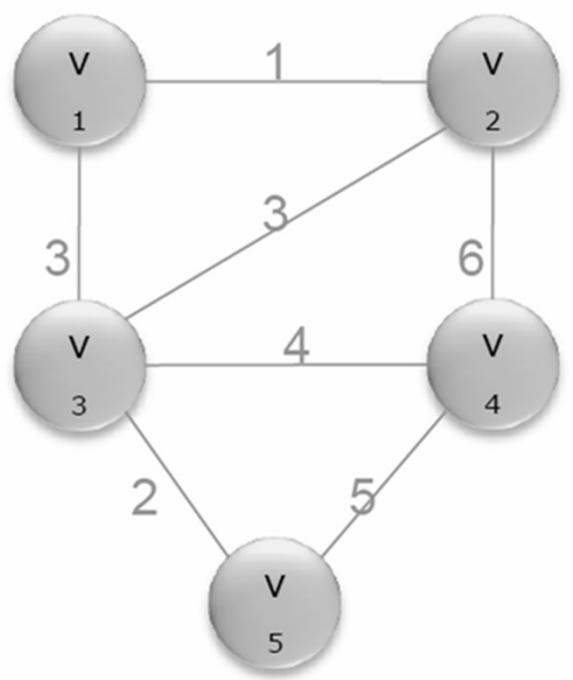
الگوریتم کراسکال(Kruskal)

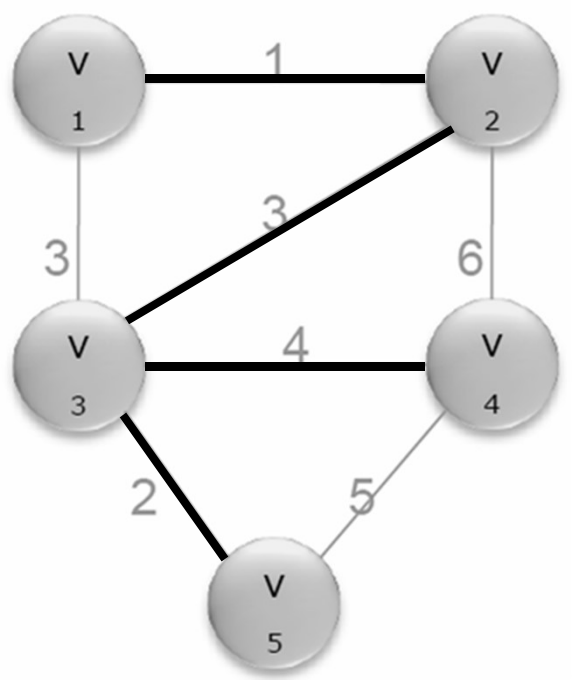
```
Procedure kruskal (V, E, T)
        sort  the edges in E by weight in increasing order
        T← Ø ;
        while( |T|<n-1 and |E|   0) do
             select edge e with least weight  not  yet considered
             Delete e from E
             if e doesn't create cycle in T
                  add e to T
       if |T|<n-1 then write 'no spanning tree'
```

تفاوت پریم(prim) و کروسکال (Kruskal)

- دو الگوریتم prim و kruskal درصورتی دو درخت متفاوت تولید میکنند که چندین یال با هزینه های مساوی داشته باشیم ولی همیشه هزینه درختهای تولید شده برابر و کمینه است.
- در الگوریتم Prim در ابتدا یک گره بوده وکم کم در حین الگوریتم گسترش میابد تا به درخت پوشای کمینه تبدیل شود درحالیکه در الگوریتم kruskal چند درخت (جنگل) وجود دارد که در انتهای الگوریتم درختهای جنگل به هم پیوند خورده تا تبدیل به درخت پوشای کمینه شوند.

الگوریتم کراسکال(Kruskal)





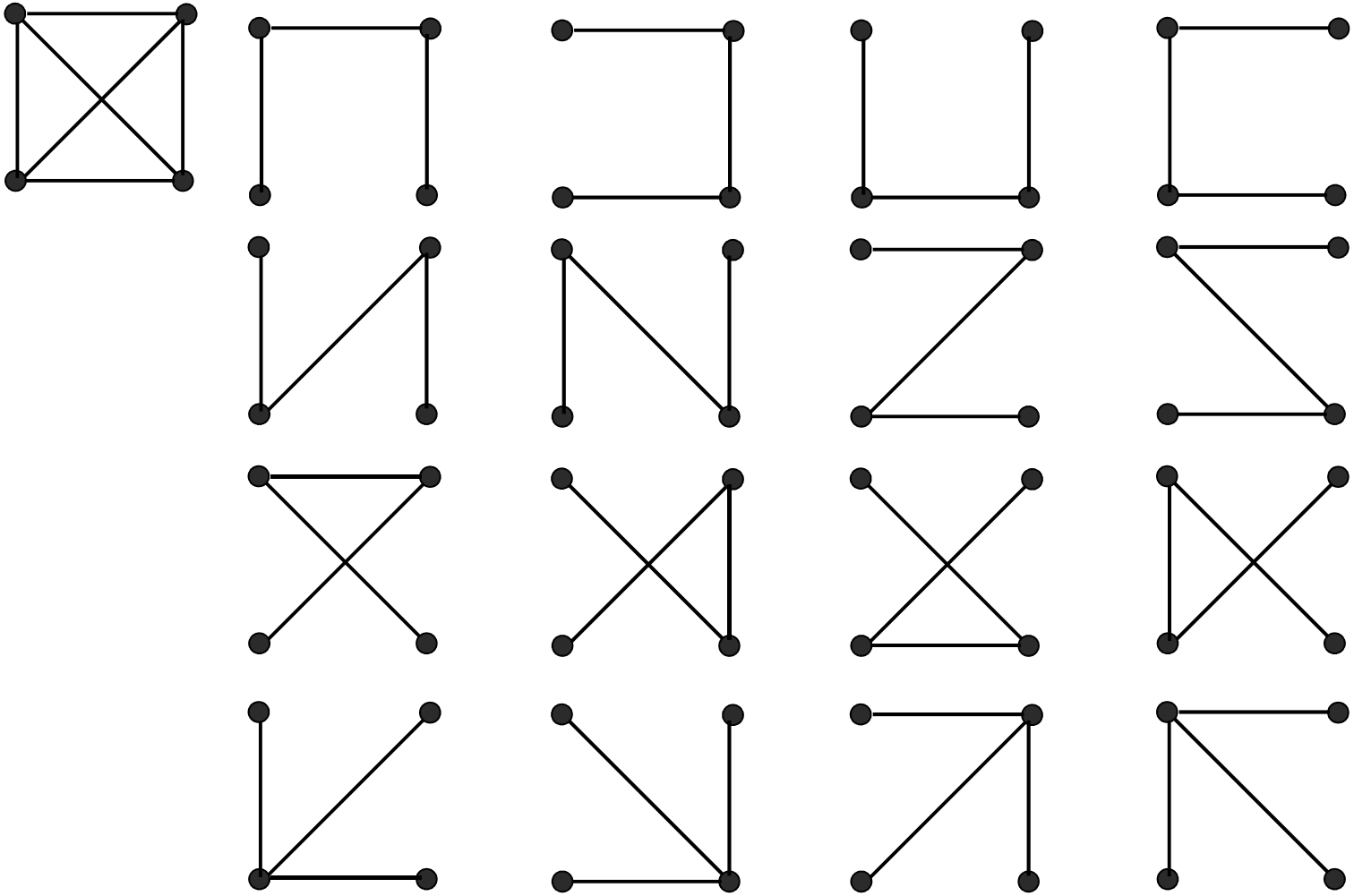
(v1,v2)	(v3,v5)	(v2,v3)	(v1,v3)	(v3,v4)	(v4,v5)	(v4,v2)
1	2	3	3	4	5	6

E= {{~~v1,v2~~},{~~v3,v5~~}, {~~v2,v3~~},{v1,v3},{~~v3,v4~~},{v4,v5} ,{v2,v3}}

T= {{v1,v2},{v2,v3},{v3,v5},{v3,v4}}

تعداد درختهای پوشای K_n

- تعداد درختهای پوشای k_n (گراف کامل با n راس) برابراست با n^{n-2}.
- مثال k_4



کوتاهترین مسیرهای هم بند (Single Source Shortest Path)

SSSP

- الگوریتمهای متعددی برای محاسبه طول کوتاهترین مسیر بین دو گره در گراف وزن دار وجود دارد که در این میان الگوریتم Dijkstra به روش حریصانه طراحی شده است.
- در الگوریتم Dijkstra هدف پیدا کردن طول کوتاهترین مسیر از یک گره مبدا نظیر v به تمام گره های دیگر می باشد. الگوریتم Dijkstra بر روی گرافهای وزنداری قابل اجراست که هزینه یالها نامنفی باشد.
- فرضیات:

طول کوتاهترین مسیر از راس v به راس i	D[i]
تعداد رئوس	n
مبدأ	v
ماتریس هزینه ها	cost
راسی قبل از راس i بر روس کوتاهترین مسیر	P[i]

الگوریتم دایجسترا (Dijkstra)

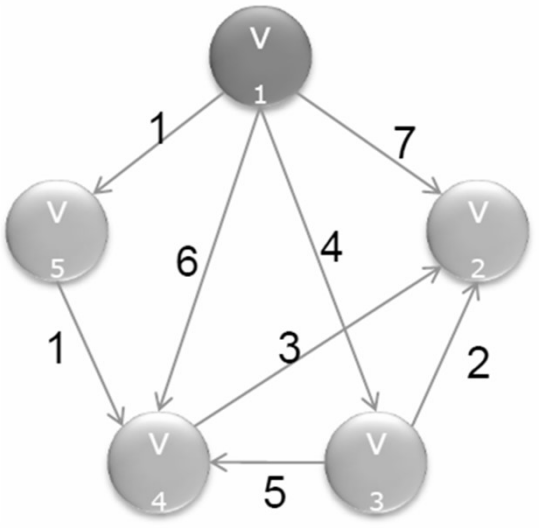
- در این الگوریتم کمترین فاصله مبدا به هر راس کم کم محاسبه شده و در هر مرحله کمتر میشود تا در پایان الگوریتم به کمترین حد خود برسد. در حین اجرای الگوریتم هر راسی که فاصله مبدا به آن بطور کامل محاسبه شده بود و مقدار آن به کمترین حد خود رسیده باشد برچسب دائمی شدن (s[w]=1) خواهد خورد. در ابتدای کار همه رئوس به غیر از مبدا دارای برچسب موقتی هستند (s[w]=0)
- در هر مرحله گره اضافه شده می تواند به عنوان گره واسط در نظر گرفته شود.
- توضیح: الگوریتم دارای ۲ فاز می باشد(مرحله ۲ و ۳)
- مراحل ۲ و ۳ n-2 بار تکرار می شود
- ۲- در هر مرحله از میان رئوسی که هنوز برچسب دائمی شدن نخورده اند راسی که دارای کمترین فاصله نسبت به مبدا است (کمترین D) انتخاب می شود و به آن برچسب دائمی زده می شود.
- ۳- مقدار D مابقی رئوسی که هنوز موقتی هستند با توجه به راس دائمی شده در مرحله ۱ بروز می شود.

الگوریتم دایجسترا (Dijkstra)

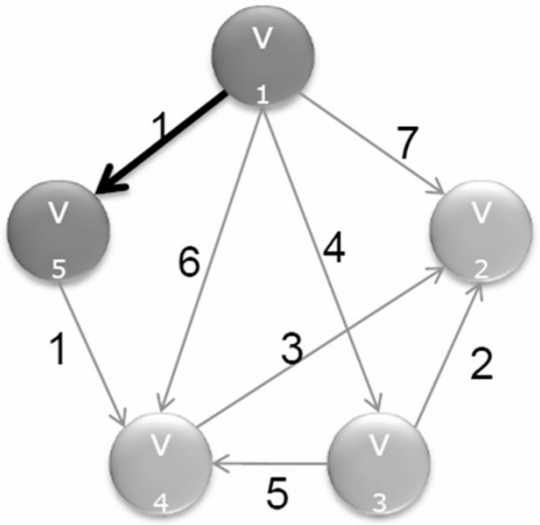
```
Procedure (G, n, v, cost, D)
 for i=1 to n do
      D[i] = cost (v,i)
      p[i] = v
      s[i]=false
 repeat
 s[v] = True
 for i=1 to n-2 do
      select vertex u such that D(u)= min {D(w) | s(w)= false}
       s[u] = true
       for all vertices w in which s(w) = false do
            if D(u) +cost (u,w) <D(w) then
                 D(w)=D(u) + cost (u,w)
                 P[w] = u
            end if
       repeat
 repeat
```

❖ این الگوریتم را با مرتبه زمانی $o(n^2)$ می توان پیاده سازی کرد.

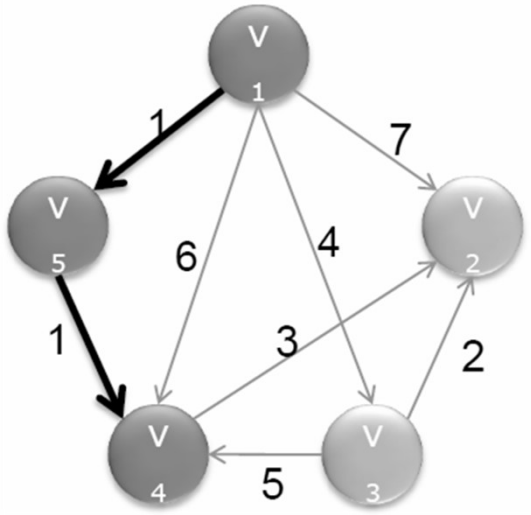
الگوریتم دایجسترا (Dijkstra)



Vertex	S	D	P
1	1	0	1
2	0	7	1
3	0	4	1
4	0	6	1
5	0	(1)	1

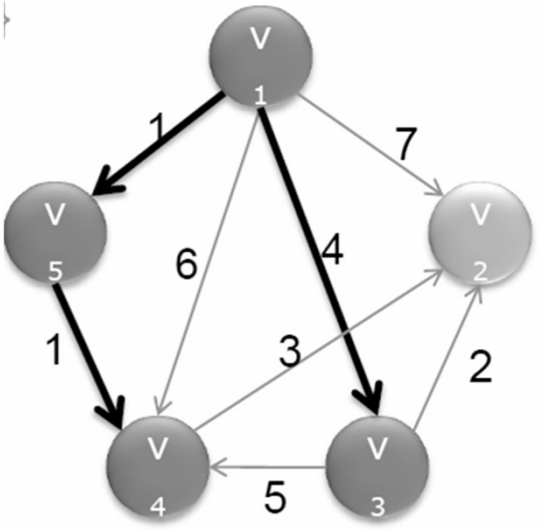


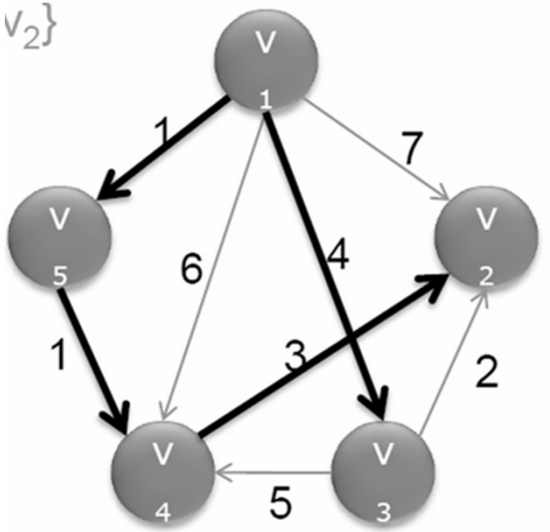
Vertex	S	D	P
1	1	0	1
2	0	7	1
3	0	4	1
4	0	(2)	5
5	1	1	1



Vertex	S	D	P
1	1	0	1
2	0	5	4
3	0	(4)	1
4	1	2	5
5	1	1	1

الگوریتم دایجسترا (Dijkstra)





Vertex	S	D	P
1	1	0	1
2	0	(5)	4
3	1	4	1
4	1	2	5
5	1	1	1

W ها	D[2]	D[3]	D[4]	D[5]
1	7	4	6	(1)
1,5	7	4	(2)	1
1,5,4	5	(4)	2	1
1,5,4,3	(5)	4	2	1

فصل پنجم

برنامه سازی پویا(Dynamic Programming)

- در روش تقسیم و حل ابتدا از مسئله اصلی شروع کرده و آن را به زیر مسائل کوچکتر تقسیم و سپس از انتها به ابتدا مسائل را حل می کنیم. در واقع برای شکستن مسائل الگوی بالا به پایین (top-down) و در ترکیب جوابها الگوی پایین به بالا رعایت می گردد که منطبق بر الگوریتم های بازگشتی است. اما اگر در روش تقسیم و حل لازم باشد زیر مسأله خاصی چندین مرتبه حل می شود که این تکرار کارآیی الگوریتم را پایین می آورد.
- اگر زیرمسأله ای یک بار حل شود و جواب آن نگهداری شود، می توان در مراحل بعدی از آن استفاده کرد و این اساس کار روش حل مسائلی است که به روش برنامه سازی پویا حل می شوند. در این روش از کوچکترین مسائل شروع و همه آنها حل می شوندو جواب آنها نگهداری می شوند. سپس به سطح بعدی رفته و کلیه مسائل اندکی بزرگتر حل می شوند و کار تا جایی ادامه می یابد که مسأله اصلی حل شود. برای حل هریک از مسائل هر سطح می توان از حل کلیه سطوح پایین تر که لازم باشد استفاده کرد. از روش برنامه سازی پویا زمانی میتوان استفاده کرد که اصل بهینگی برقرار باشد. این اصل بر این اساس است که برای حل مسئله به صورت بهینه از حل بهینه زیر مسائل آن میتوان استفاده کرد.

برنامه سازی پویا(Dynamic Programming)

- برنامه سازی پویا در مقایسه با روش تقسیم و حل : درخت حل مسئله را از پایین به بالا (bottom-up) ساخته و نتایج در یک جدول نگهداری می شوند تا در موقع لزوم بتوان از آنها استفاده کرد و دوباره آنها را حل نکرد.
- نکته: نوعی روش برنامه سازی پویا وجود دارد که در آن زیر فضای حل مسئله از بالا به پایین است ولی زیر مسائل حل شده در جدولی نگهداری می شوند تا از حل زیر مسائل تکراری پرهیز شود که این روش به نام روش به خاطرسپاری (memorized) شناخته می شود.
- اصل بهینگی: انتخاب بهینه نهایی به انتخابهای بهینه اولیه بستگی دارد.
- برای ارائه الگوریتم به روش برنامه سازی پویا ۴ مرحله را باید طراحی کرد:

۱- تعریف تابعی که حل تابع منجر به حل مسئله شود.

۲- بیان شرایط مرزی

۳- بیان جواب مسئله بر حسب تابع

۴- تعریف بازگشتی تابع

برنامه سازی پویا(Dynamic Programming)

❖ مثال:

روش بازگشتی

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

روش برنامه نویسی پویا

```
int fibo2 (int n)
  {
   int  f[0 . . n];
   f[0]=0;
   if (n >0 ) {
       f[1]=1;
       for (i=2; i<=n; i++)
            f[i]=f[i-1]+f[i-2];
   }
    return f[n];
  }
```

برنامه سازی پویا(Dynamic Programming)

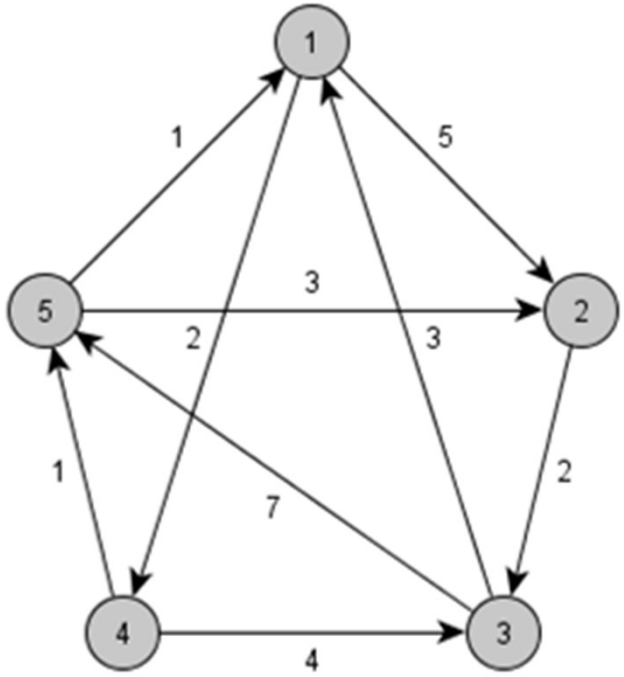
- مسئله همه کوتاهترین مسیرها All Pairs Shortest Path (ASPS) که هدف آن پیدا کردن طول کوتاهترین مسیر بین همه زوج رئوس گراف می باشد.
- دراین روش تمام مسیرهای ممکن از هر رأسی به هر رأس دیگر که از رأس k بگذرد یا نگذرد را محاسبه کرده تا کوتاهترین مسیر بدست آید و خود k از ۱ تا n تغییر می کند.
- وقتی k=n شد تمام حالات ممکن برای کوتاهترین مسیر محاسبه شده است. جواب D(i,j) وقتی k=0 است یعنی زمانی که از هیچ راسی عبور نمی کند. مستقیما از راس i به j رفته که همان cost(i,j) می باشد.
- در صورتی که از i به j مسیر وجود نداشته باشد ارزش یالهای آن :است.
- الگوریتم مورد بحث الگوریتم floyd است.
- در فلوید ماتریس وزن ها را بدون درنظر گرفتن واسط بدست می آوریم. از سطر اول ماتریس شروع می کنیم و با اضافه کردن هر راس آن را می توان به عنوان یال واسط گرفت.

برنامه سازی پویا(Dynamic Programming)

```
procedure APSP_Floyd(cost,n,A)
for i=1 to n do
    for j=1 to n do
        D[i,j]=cost[i,j]
        p[i,j] =0
    repeat
repeat
for k=1 to n do
     for i=1 to n do
          for j=1 to n do
               if D[i,k]+D[k,j]<D[i,j] then
                    D[i,j] =D[i,k]+D[k,j]
                    p[i,j] =k
               end if
          repeat
     repeat
repeat
```

- مرتبه زمانی این الگوریتم $O(n^3)$ است.

فلوید (Floyd)



$$D_0 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & 3 & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 5 & 7 & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 7 & 2 & 14 \\ 5 & 0 & 2 & 7 & 9 \\ 3 & 8 & 0 & 5 & 7 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_5 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 2 & 4 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

عدد کاتلان (Catalan) و مسائل مرتبط

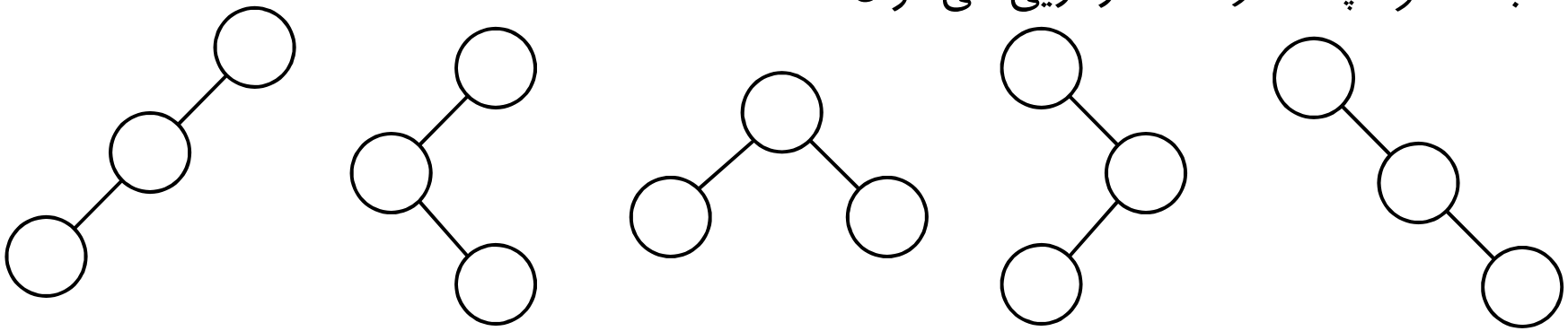
- اعداد کاتالان در ریاضیات ترکیبی، یک سری از اعداد طبیعی هستند که در مسائل شمارشی متنوع که معمولاً اشیا به صورت بازگشتی تعریف شده را در بر می‌گیرند، رخ می‌دهند. این اعداد به افتخار ریاضیدان بلژیکی شارل کاتالان (۱۸۱۴-۱۸۹۴) اعداد کاتالان نامیده می‌شوند.
- n اُمین عدد کاتالان عبارت است از:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} \qquad \text{for } n \geq 0.$$

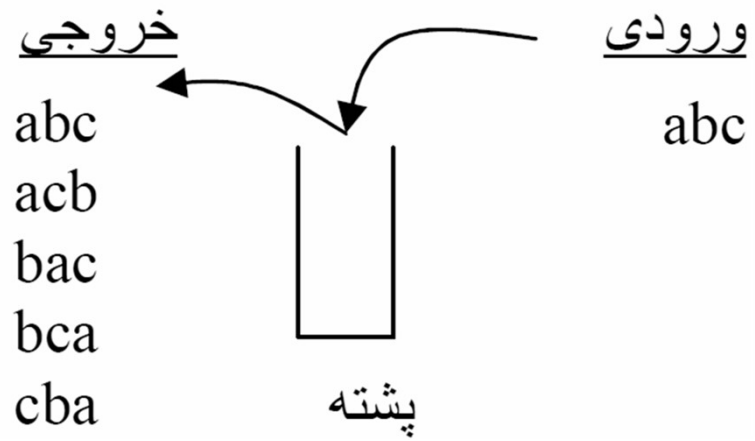
- چند عدد اولیه کاتالان عبارتند از:
- ۱, ۲, ۵, ۱۴, ۴۲, ۱۳۲, ۴۲۹, ۱۴۳۰, ۴۸۶۲, ۱۶۷۹۶, ۵۸۷۸۶, ۲۰۸۰۱۲, ۷۴۲۹۰۰, ۲۶۷۴۴۴۰, ۹۶۹۴۸۴۵, ۳۵۳۵۷۶۷۰, ۱۲۹۶۴۴۷۹۰, ۴۷۷۶۳۸۷۰۰, ۱۷۶۷۲۶۳۱۹۰, ۶۵۶۴۱۲۰۴۲۰, ۲۴۴۶۶۲۶۷۰۲۰, ۹۱۴۸۲۵۶۳۶۴۰, ۳۴۳۰۵۹۶۱۳۶۵۰, ۱۲۸۹۹۰۴۱۴۷۳۲۴, ۴۸۶۱۹۴۶۴۰۱۴۵۲

عدد کاتلان (Catalan) و مسائل مرتبط

❖ با n گره چند درخت دودویی می توان ساخت:



❖ به چند طریق می توان n ورودی را به کمک یک پشته در خروجی چاپ کرد به قسمی که فقط عملیات read، write، push و pop انجام شود.



عدد کاتلان (Catalan) و مسائل مرتبط

❖ به چند طریق می توان n+1 ماتریس را در هم ضرب کرد.

$$1-\left(\left(\left(m_1\times m_2\right)\times m_3\right)m_4\right)$$

$$2-\left(\left(m_1\times\left(m_2\times m_3\right)\right)\times m_4\right)$$

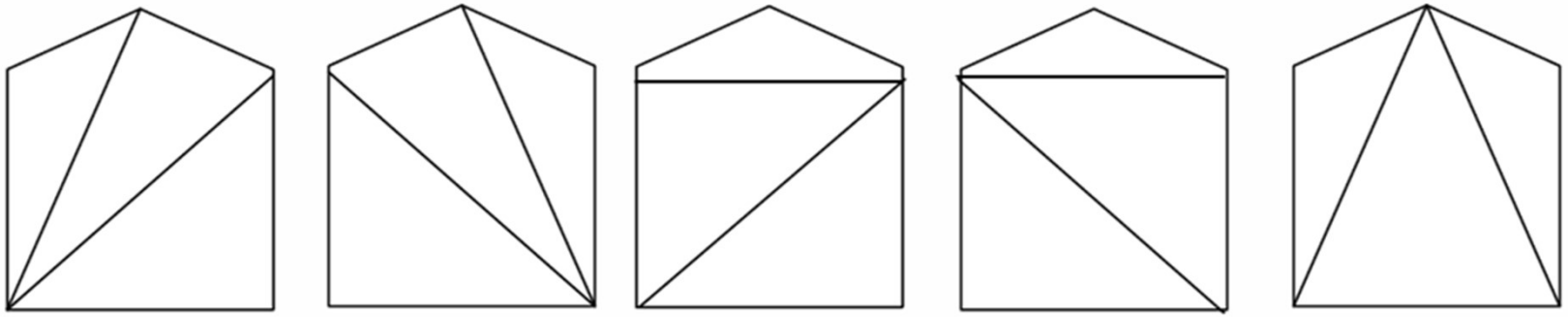
$$3-\left(\left(m_1\times m_2\right)\times\left(m_3\times m_4\right)\right)$$

$$4-\left(m_1\times\left(\left(m_2\times m_3\right)\times m_4\right)\right)$$

$$5-\left(m_1\times\left(m_2\times\left(m_3\times m_4\right)\right)\right)$$

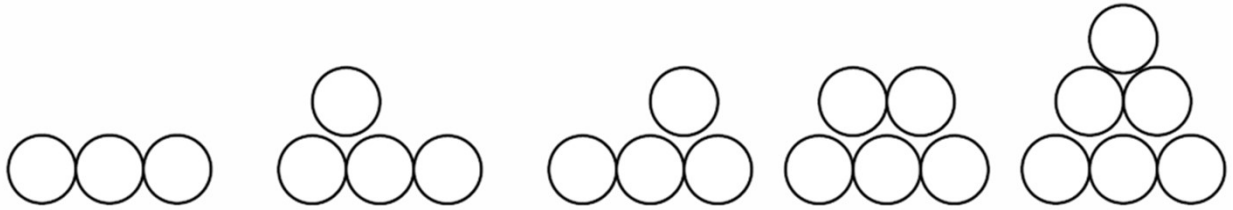
❖ به چند طریق می توان یک n+2 ضلعی محدب را مثلث بندی کرد؟

نکته: مثلث بندی منجر به رسم بیشترین تعداد اقطار میشود به قسمی که اقطار یکدیگر را قطع نکنند.

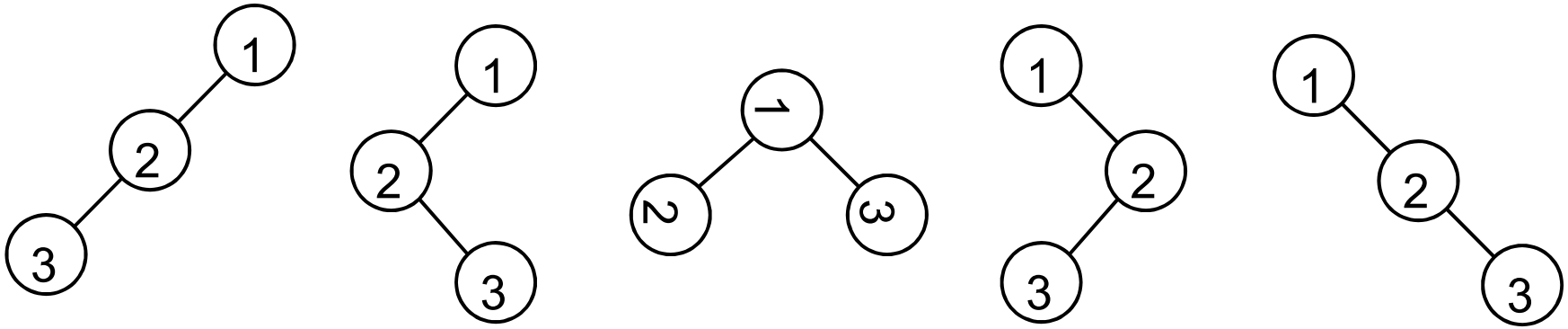


عدد کاتلان (Catalan) و مسائل مرتبط

❖ به چند طریق می توان n سکه را در قاعده قرار داد و روی آن تعداد دلخواه سکه چید.



❖ اگر پیمایش preorder درخت دودویی برابر با 1,2,3,…,n باشد چند inorder می توان برای آن تصور کرد.



❖ قابل اثبات است که جواب همگی این مسائل همگی برابر با عدد معروفی به نام عدد کاتالان میباشد.

فروشنده دوره گرد (Traveling Salesman Problem)

TSP

- دور هامیلتونی در یک گراف دوری است که از همه رئوس دقیقا یکبار بگذرد. در مسئله فروشنده دوره گرد هدف پیدا کردن دور همیلتونی با هزینه مینیمم در یک گراف وزن دار ورودی می باشد.
- فرض کنید گراف ورودی بصورت $G(V,E)$ می باشد که در آن $V=\{1,2,...,n\}$ است و $c_{i,j}$ نشاندهنده هزینه یال (i,j) باشد. همچنین فرض کنید راس آغازین راس شماره ۱ باشد.
- مراحل الگوریتم:

۱- طول کوتاهترین مسیری که از راس i شروع شده و از کلیه رئوس مجموعه S دقیقاً یکبار بگذرد و به راس ۱ ختم شود = $g(i,S)$ $(i\notin S\ ,\ 1\notin S)$

۲- $g(i,\varnothing)=c_{i,1}$ (بدیهی)

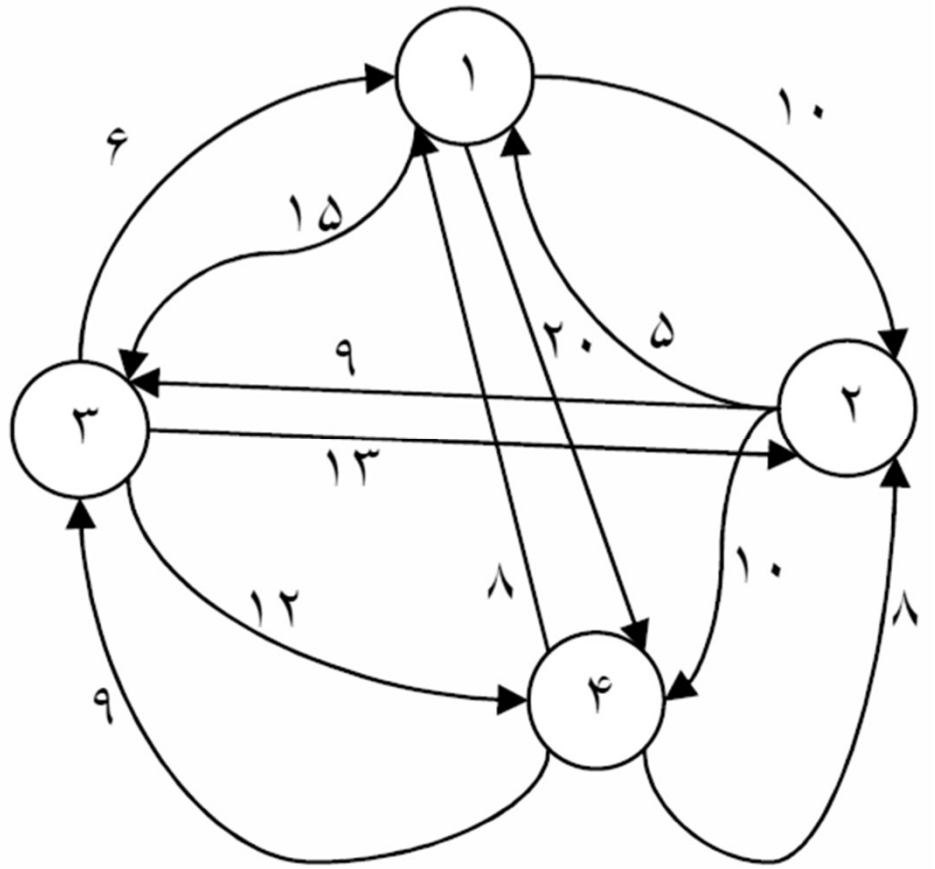
۳- جواب = $g(1,\ V-\{1\})$

۴- $g(i,S)=\min\{c_{i,j}+g(j,\ S-\{j\})\ \}$ $(j\in S,\ 1\notin S,\ i\notin S)$

TSP

فروشنده دوره گرد (Traveling Salesman Problem)

❖ در گراف زیر دور هامیلتونی با هزینه کمینه را پیدا کنید.



❖ ماتریس هزینه ها

c	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

فروشنده دوره گرد

$$|S|=0 \Rightarrow \begin{cases} g(2,\varnothing)=c_{2,1}=5 \\ g(3,\varnothing)=c_{3,1}=6 \\ g(4,\varnothing)=c_{4,1}=8 \end{cases}$$

$$|S|=1 \Rightarrow \begin{cases} g(2,\{3\})=c_{2,3}+c_{3,1}=15 \\ g(2,\{4\})=c_{2,4}+c_{4,1}=18 \\ g(3,\{2\})=c_{3,2}+c_{2,1}=18 \\ g(3,\{4\})=c_{3,4}+c_{4,1}=20 \\ g(4,\{2\})=c_{4,2}+c_{2,1}=13 \\ g(4,\{3\})=c_{4,3}+c_{3,1}=15 \end{cases}$$

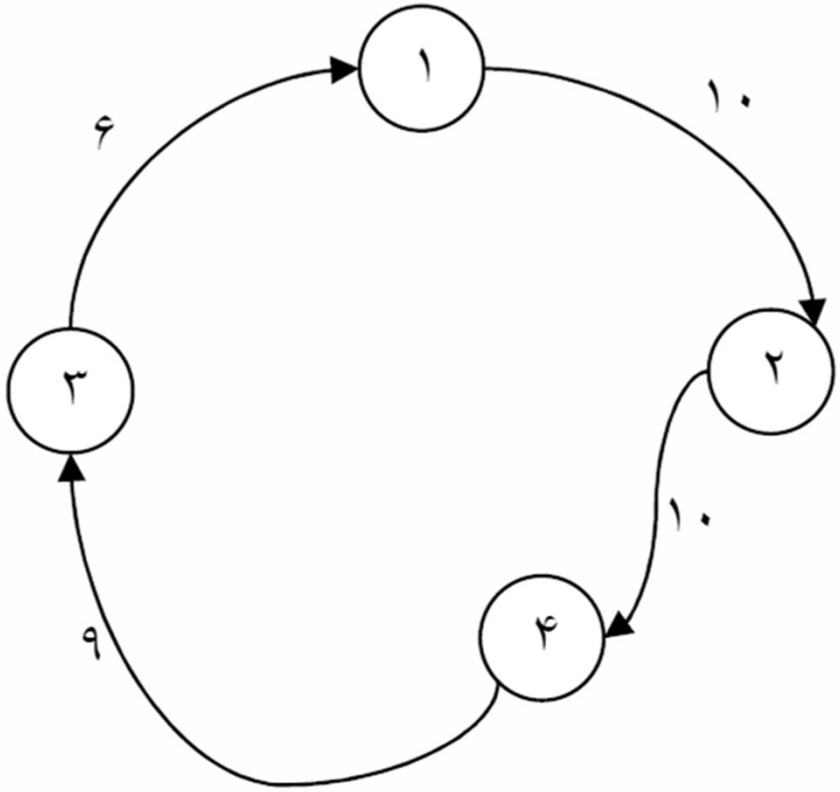
$$|S|=2 \Rightarrow \begin{cases} g(2,\{3,4\})=\min\{c_{2,3}+g(3,\{4\}),\underbrace{c_{2,4}+g(4,\{3\})}_{\min}\}=25 \\ g(3,\{2,4\})=\min\{c_{3,2}+g(2,\{4\}),c_{3,4}+g(4,\{2\})\}=25 \\ g(4,\{2,3\})=\min\{c_{4,2}+g(2,\{3\}),c_{4,3}+g(3,\{2\})\}=23 \end{cases}$$

$$|S|=3 \Rightarrow \begin{cases} g(1,\{2,3,4\})=\min\{\underbrace{c_{1,2}+g(2,\{3,4\})}_{\min},c_{1,3}+g(3,\{2,4\}),c_{1,4}+g(4,\{2,3\})\}=35 \end{cases}$$

TSP

فروشنده دوره گرد (Traveling Salesman Problem)

$$\Rightarrow 35 = c_{1,2} + \underbrace{g(2,\{3,4\})}_{c_{2,4}+\underbrace{g(4,\{3\})}_{c_{4,3}+c_{3,1}}} = c_{1,2} + c_{2,4} + c_{4,3} + c_{3,1}$$

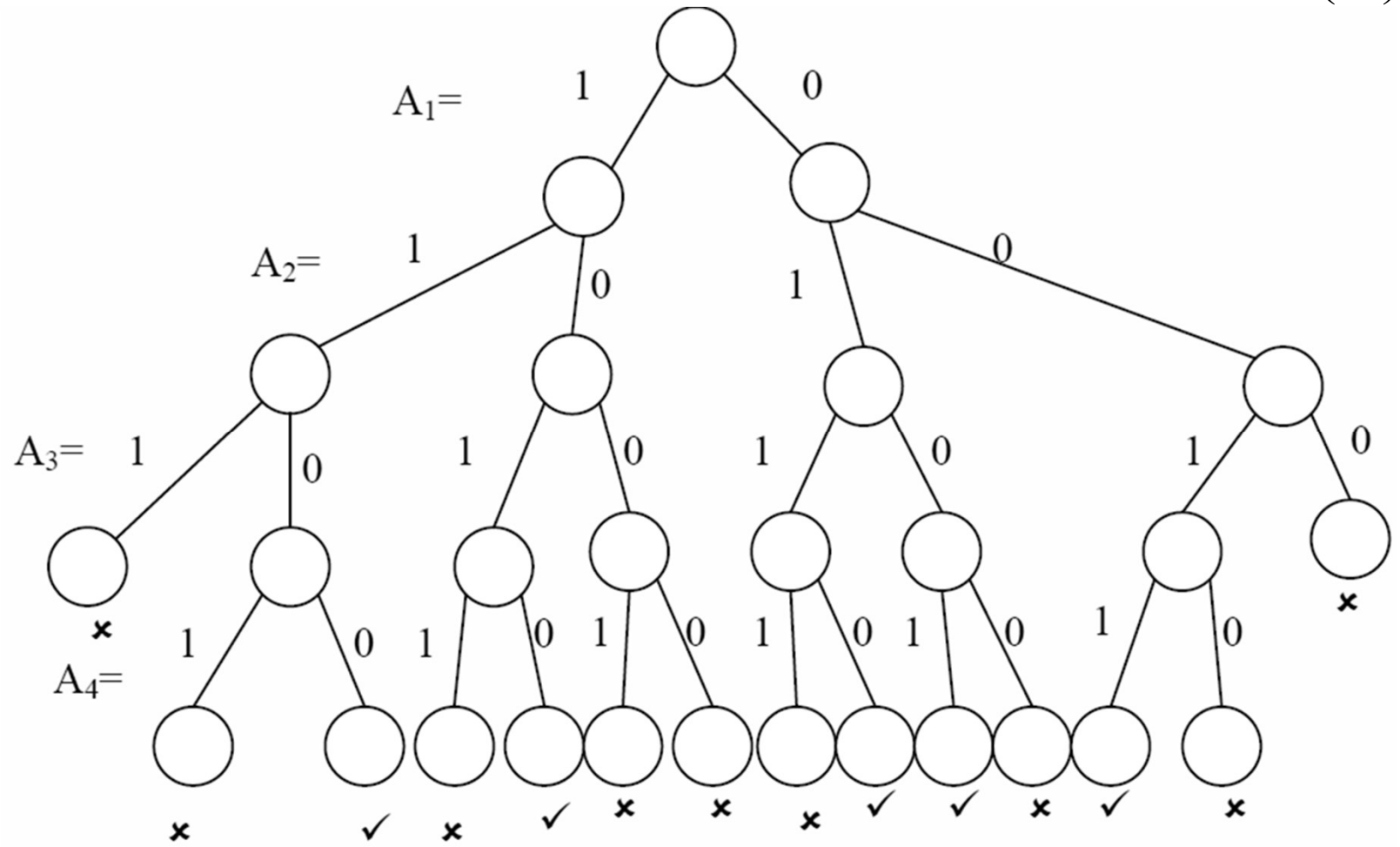


روش عقبگرد(Backtracking)

- در بعضی شرایط در هنگام حل مسئله نمیتوان از روش خاصی استفاده کرد و از این رو لازم می شود که فضای حالات بطور کامل جستجو شده تا جواب مسئله مشخص شود. در این شرایط از روش خاصی به نام روش پیجویی به عقب یا روش عقبگرد استفاده می شود.
- در این روش درخت فضای حالت (درخت تصمیم) به روش "عمقی" depth-first جستجو می شود به این معنی که تا زمانی که از اشتباه بودن یک مسیر مطمئن نشده ایم آن مسیر را ادامه داده و در صورت اطمینان از اشتباه بودن آن یک مرحله به عقب بازگشته و دوباره کار را ادامه می یابد. این روند ادامه می یابد تا به جواب نهایی برسیم.
- این روش یک روش غیرهدفمند می باشد که در حل مسئله باید تمام حالات را درنظر گرفت.
- برای حل مسائل تصمیم گیری مفید است. مسائل تصمیم گیری جزء مسائلی هستند که پیچیدگی محاسباتی بالایی(نمایی-فاکتوریل) دارند از این لحاظ به مسائل NP-complete معروف هستند. مسائلی که راه حل چند جمله ای برای آنها یافت نشده است.

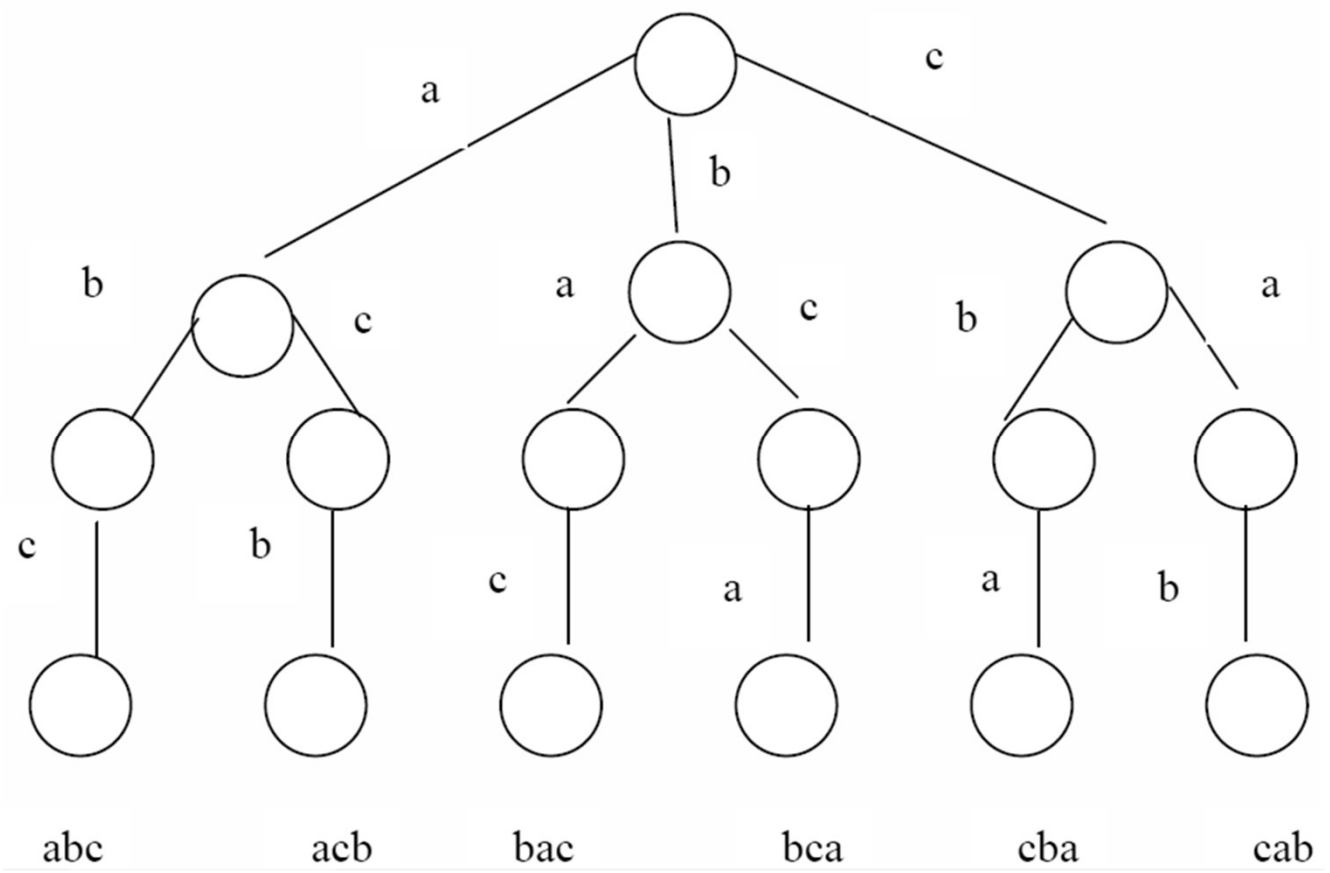
روش عقبگرد(Backtracking)

❖ اعداد ۴ بیتی را پیدا کنید که تعداد یکهای آنها دقیقًا ۲ تا باشد. (مرتبه زمانی $O(2^n)$)



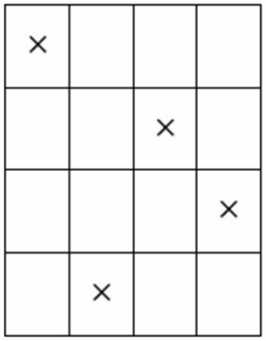
روش عقبگرد(Backtracking)

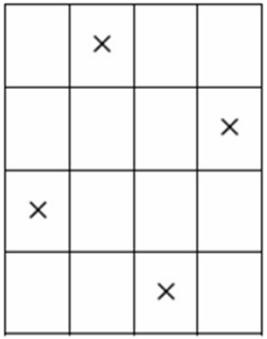
❖ مجموعه ای با n>=1 عضو وجود دارد. تمام ترکیبات ممکن آن نیاز است (مرتبه زمانی O(n!))



روش عقبگرد(Backtracking)

- مساله n وزیر: هدف قرار دادن n مهره وزیر در صفحه شطرنج n*n به گونه ای است که هیچیک از وزیران دیگری را تهدید نکند. فرض است در هیچ سطر و ستونی بیش از یک وزیر قرار نگیرد.
- با این فرض تمام جایگشت های ممکن تولید می شود یعنی همان درخت حل مسئله یا فضای حالت بوجود می آید سپس در هرمرحله بررسی می شود که مسیر فعلی در درخت با این نحوه ای چیدمان وزیرها آیا دارای شرایط مسئله می باشد یعنی وزیرها یکدیگر را تهدید می کنند یا خیر. برای مثال برای n=4 مسئله دارای دو جواب بصورت زیر است. (مرتبه زمانی O(n!))
- اگر n برابر۵ باشد ۵ جواب برای مسئله وجود دارد. برای n=8، ۹۲ جواب وجود دارد.





روش انشعاب و تحدید(Branch & Bound)

B&B

- یک روش غیرهدفمند اما هوشمند است. در این روش برخلاف روش backtracking همه حالات بررسی نمی شود.
- سه تفاوت عمده B&B و Backtrack:
- در روش backtracking درخت فضای حالت به صورت DFS است ولی در B&B کلیه فرزندان گره فعلی ساخته می شود سپس از بین آنها یکی بسته به شرایط انتخاب می شود و تا حدی شبیه BFS است.
- در روش B&B یک تابع محدود کننده وجود داردکه از گسترش نابجا و بیش از حد شاخه های درخت جلوگیری می کند (تابع تخمین). در موقع لزوم شاخه را قطع کرده و مسیر فعلی را ادامه نمی دهد. در backtracking چنین معیاری وجود ندارد.
- روش B&B در مقابل backtracking هوشمندانه تر عمل می کندو در هنگام انشعاب شاخه جهت گسترش شاخه ای که احتمال بیشتری برای تولید دارد را انتخاب می کند.
- مرتبه زمانی backtracking و B&B فرقی ندارد اما زمان واقعی آن کمتر است.