فصل اول

کارایی، تحلیل و مرتبه الگوریتم ها

کارایی، تحلیل و مرتبه الگوریتم ها

- تکنیک ، روش مورد استفاده در حل مسائل است.
- مسئله ، پرسشی است که به دنبال پاسخ آن هستیم.
- بکار بردن تکنیک منجر به روشی گام به گام (الگوریتم) در حل یک مسئله می شود.
- منظور از سریع بودن یک الگوریتم، یعنی تحلیل آن از لحاظ زمان و حافظه.

- نوشتن الگوریتم به زبان فارسی دو ایراد دارد:

۱- نوشتن الگوریتم های پیچیده به این شیوه دشوار است.

۲- مشخص نیست از توصیف فارسی الگوریتم چگونه می توان یک برنامه کامپیوتری ایجاد کرد.

کارایی، تحلیل و مرتبه الگوریتم ها

تحلیل الگوریتم ها

- برای تعیین میزان کارایی یک الگوریتم را باید تحلیل کرد.

تحلیل پیچیدگی زمانی

- تحلیل پیچیدگی زمانی یک الگوریتم ، تعیین تعداد دفعاتی است که عمل اصلی به ازای هر مقدار از ورودی انجام می شود.

کارایی، تحلیل و مرتبه الگوریتم ها

- $T(n)$ را پیچیدگی زمانی الگوریتم در حالت معمول می گویند.
- $W(n)$ را تحلیل پیچیدگی زمانی در بدترین حالت می نامند.
- $A(n)$ را پیچیدگی زمانی در حالت میانگین می گویند.

تحلیل پیچیدگی زمانی برای حالت معمول برای الگوریتم(جمع کردن عناصرآرایه)

عمل اصلی: افزودن یک عنصر از آرایه به sum.

اندازه ورودی: n، تعداد عناصر آرایه.

$$T(n) = n$$

تحلیل پیچیدگی زمانی برای حالت معمول برای الگوریتم(مرتب سازی تعویضی)

عمل اصلی: مقایسه S [j] با S[i] .

اندازه ورودی: تعداد عناصری که باید مرتب شوند.

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

تحلیل پیچیدگی زمانی دربدترین حالت برای الگوریتم(جست و جوی ترتیبی)

عمل اصلی: مقایسه یک عنصر آرایه با x.

اندازه ورودی: n , تعداد عناصر موجود در آرایه.

$$W(n) = n$$

تحلیل پیچیدگی زمانی در بهترین حالت برای الگوریتم(جست وجوی ترتیبی)

عمل اصلی: مقایسه یک عنصر آرایه با x.

اندازه ورودی: n , تعداد عناصر آرایه.

$$B(n) = 1$$

کارایی، تحلیل و مرتبه الگوریتم ها

مثال ۱: برای مثال جمع عناصر یک آرایه ، تعداد کل مراحل برنامه را بنویسید.

```
float  sum  (float  list[  ], int  n)
{
  float  s=0 ;       int i;
    for (i = 0;  i<n;  i++)
              s = s + list[i];
       return    s;
  }
```

0
0
1
0
n+1
n
1
0
2n+3

کارایی، تحلیل و مرتبه الگوریتم ها

نکته : هنگام محاسبه تعداد دفعاتی که یک دستور درون حلقه‌ها اجراء می‌گردد می‌توان از فرمولهای زیر استفاده کرد :

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} kf(i) = k\sum_{i=1}^{n} f(i)$$

K عدد ثابتی است.

$$\sum_{i=1}^{n} i = 1+2+3+\ldots+n = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

a_1 جمله اول، a_n جمله آخر و n تعداد جملات است : $\text{جمع تصاعد عددی} = \frac{n(a_1 + a_n)}{2}$

راه دیگر Trace کردن برنامه است.

مثال ۲: دستور اصلی در تکه برنامه زیر چند بار اجرا میشود؟ تعداد کل گامهای برنامه چند است؟

```
for  i: = 1 to m do
    for j: = 1 to n do
            x: = x+1;
```

راه حل اول:

تعداد اجرای دستور به دلیل وجود حلقه های مستقل برابر nm است.

راه حل دوم:

$$\text{تعداد اجرای دستور اصلی} = \sum_{i=1}^{m}\sum_{j=1}^{n} 1 = \sum_{i=1}^{m} n = n\left(\sum_{i=1}^{m} 1\right) = nm$$

$$\text{تعداد کل مراحل برنامه} = (m + 1) + m(n + 1) + mn$$

مثال ۳: دستور اصلی در تکه برنامه زیر چند بار اجرا میشود؟

```
for  j: = 1 to n do
    for  i: = 1 to j do
                x: = x+1;
```

j	تغییرات i	تعداد اجرا شدن دستور اصلی
1	1	1 بار
2	1,2	2 بار
3	1,2,3	3 بار
	.....	
n	1,2,3,...,n	n بار

$$x: = x+1; \text{ تعداد اجرا شدن دستور } = 1+2+3+\ldots+n = \frac{n(n+1)}{2}$$

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

کارایی، تحلیل و مرتبه الگوریتم ها

مثال ۳: دستور اصلی در تکه برنامه زیر چند بار اجرا میشود؟

```
i:=n;
while (i>1) do begin
        x:=x+1;
         i: = i div 2;
end;
```

اگر n=16 باشد:

i	شرط i >1	تعداد اجرا شدن دستور اصلی
16	درست	۱ بار
8	درست	۱ بار
4	درست	۱ بار
2	درست	۱ بار
1	غلط	-
		جمعاً ٤ بار

کارایی، تحلیل و مرتبه الگوریتم ها

اگر n=14 باشد:

i	شرط i>1	تعداد اجرا شدن دستور اصلی
14	درست	۱ بار
7	درست	۱ بار
3	درست	۱ بار
1	غلط	-
		جمعاً ۳ بار

دستور اصلی به تعداد $\lfloor \log_2^n \rfloor$ بار اجرا می‌شود.

کارایی، تحلیل و مرتبه الگوریتم ها

برخی مسائل برای همه موارد یک تابع پیچیدگی دارند مثل الگوریتم جمع عناصر یک آرایه :

```
A : Array [1 .. n]  of Integer;
S := 0;
For I := 1  To  n  do
    S := S + A[i] ;
```

در برنامه فوق عمل اصلی S := S + A[i] به تعداد n بار اجرا شده و همواره T(n) = n می‌باشد. ولی در الگوریتمی مثل جستجوی خطی (ترتیبی) تابع پیچیدگی برای حالات مختلف ممکن است متفاوت باشد.

```
A : Array [1 .. n] of Integer;
For i := 1  to  n  do
    if (x = A[i])  {
            write ('Yes');
              exit ( ) ; → خروج از برنامه
    }
write ('No');
```

در برنامه فوق عمل اصلی شرط if (x = A[i]) می‌باشد.

در بدترین حالت عدد x ، در خانه آخر قرار دارد و یا اصلاً در آرایه نیست که در این حالت باید n بار عمل اصلی آزمایش کردن، انجام گیرد. برای بدترین حالت به جای نماد T(n) از نماد W(n) استفاده می‌کنیم که W مخفف Worst یعنی بدترین است. پس در برنامه فوق W(n) = n می‌باشد.

در بهترین حالت عدد x در اولین خانه قرار دارد و تنها به یک عمل if مورد نیاز است. بهترین حالت را با B نمایش می‌دهیم که مخفف Best است لذا در برنامه فوق داریم : B(n) = 1

برای تحلیل برنامه فوق در حالت متوسط که آن را با A (مخفف Average) نشان می‌دهیم باید به صورت زیر عمل کنیم. ابتدا فرض می‌کنیم x در آرایه A وجود داشته باشد. بدیهی است احتمال آنکه x در خانه i ام باشد برابر $\frac{1}{n}$ است و تعداد دفعات آزمایش کردن در این حالت برابر i است. لذا :

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

پس به طور متوسط نیمی از عناصر آرایه باید جستجو شود.

حال موردی را در نظر می‌گیریم که x ممکن است اصلاً در آرایه نباشد. فرض کنید احتمال وجود x در آرایه برابر p است. پس احتمال آنکه x در یکی از خانه‌های آرایه (مثل خانه i ام) باشد برابر $\frac{p}{n}$ است. احتمال آنکه x در آرایه نباشد برابر 1-p است. اگر x در خانه i ام باشد دستور اصلی i بار اجرا می‌شود و اگر x در آرایه نباشد دستور اصلی n بار اجرا می‌شود، لذا :

$$A(n)=\left(\sum_{i=1}^{n}\frac{p}{n}\times i\right)+(1-p)n=\frac{p}{n}\times\frac{n(n+1)}{2}+n(1-p)=n(1-\frac{p}{2})+\frac{p}{2}$$

توجه کنید که اگر p = 1 باشد همان حالت قبلی $A(n)=\frac{n+1}{2}$ بدست می‌آید و اگر $p=\frac{1}{2}$ باشد $A(n)=\frac{3n}{4}+\frac{1}{4}$ می‌شود و این بدان معناست که حدود $\frac{3}{4}$ آرایه به طور میانگین باید جستجو شود.

مرتبه الگوریتم

- الگوریتم ها یی با پیچیدگی زمانی ازقبیل n و100n را الگوریتم های زمانی خطی می گویند.

- مجموعه کامل توابع پیچیدگی را که با توابع درجه دوم محض قابل دسته بندی باشند، $\theta(n^2)$ می گویند.

- مجموعه ای ازتوابع پیچیدگی که با توابع درجه سوم محض قابل دسته بندی باشند، $\theta(n^3)$ نامیده می شوند.

- برخی از گروه های پیچیدگی متداول در زیر داده شده است:

$$\theta(\lg n) < \theta(n) < \theta(n \lg n) < \theta(n^2) < \theta(n^3) < \theta(2^n)$$

- برای یک تابع پیچیدگی مفروض $f(n)$، $O(f(n))$ "O" بزرگ" مجموعه ای از توابع پیچیدگی $g(n)$ است که برای آن ها یک ثابت حقیقی مثبت c و یک عدد صحیح غیر منفی N وجود دارد به قسمی که به ازای همه ی $n >= N$ داریم:

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

- برای یک تابع پیچیدگی مفروض $f(n)$ ، $\Omega(f(n))$ مجموعه ای از توابع پیچیدگی $g(n)$ است که برای آن ها یک عدد ثابت حقیقی مثبت c و یک عدد صحیح غیر منفی N وجود دارد به قسمی که به ازای همه ی $n >= N$ داریم:

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

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 مثبت c و d و عدد صحیح غیر منفی N وجود دارد به قسمی که :

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

- برای یک تابع پیچیدگی $f(n)$ مفروض، $o(f(n))$ "o کوچک" عبارت ازمجموعه کلیه توابع پیچیدگی $g(n)$ است که این شرط را برآورده می سازند : به ازای هرثابت حقیقی مثبت c ،یک عدد صحیح غیر منفی N وجود دارد به قسمی که به ازای همه ی $n >= N$ داریم:

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

ویژگی های مرتبه

۱- $g(n) \in O(f(n))$ اگروفقط اگر $f(n) \in \Omega(g(n))$.

۲- $g(n) \in \theta(f(n))$ اگروفقط اگر $f(n) \in \theta(g(n))$.

۳- اگر $b > 1$ و $a > 1$، در آن صورت:

$$\log_a n \in \theta(\log_b n)$$

۴- اگر $b > a > 0$،در آن صورت:

$$a^n \in o(b^n)$$

۵- به ازای همه ی مقادیر $a > 0$ داریم :

$$a^n \in o(n!)$$

۶- اگر $c >= 0$، $d > 0$ ، $g(n) \in o(f(n))$ و $h(n) \in \theta(f(n))$ باشد، در آن صورت:

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

۷- ترتیب دسته های پیچیدگی زیر را در نظربگیرید:

$\theta(\lg n)\ \theta(n)\ \theta(n \lg n)\ \theta(n^2)\ \theta(n^j)\ \theta(n^k)\ \theta(a^n)\ \theta(b^n)\ \theta(n!)$

که در آن $2 < j < k$ و $1 < a < b$ است. اگر تابع پیچیدگی $g(n)$ در دسته ای واقع در طرف چپ دسته ی حاوی $f(n)$ باشد، در آن صورت:

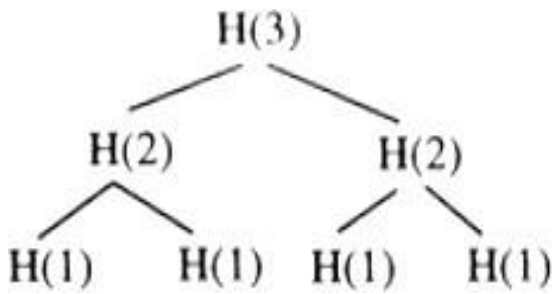
$$g(n) \in o(f(n))$$

کارایی، تحلیل و مرتبه الگوریتم ها

مثال: مرتب اجرایی الگوریتم بازگشتی برج های هانوی را بدست آورید.

```
void Hanoi (int n, A, B, C)
{
    if (n==1) Move a disk from A to C;
    else {
            Hanoi (n - 1, A, C, B);
            Move a disk from A to C;
            Hanoi (n-1, B, A, C);
    }
}
```

روش اول



روش دوم

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

$$T(n)=2T(n-1)+1=2(2T(n-2)+1)+1=2^2T(n-2)+2+1$$
$$=2^3T(n-3)+2^2+2+1=\dots$$
$$=2^{n-1}T(1)+2^{n-2}+2^{n-3}+\dots+2+1 \qquad \left(\frac{t_1(q^n-1)}{q-1}\right) \quad \text{جمع تصاعد هندسی}$$

$$\frac{1\times(2^n-1)}{2-1}=2^n-1 \quad \Rightarrow \quad T(n)=O(2^n)$$

کارایی، تحلیل و مرتبه الگوریتم ها

تمرین: مرتبه اجرایی تابع زیر کدام مورد است؟

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

۱) $O(\log n)$ ۲) $O(n)$ ۳) $O(2^{\frac{n}{2}})$ ۴) $O(2^n)$

روش برهان خلف و استقراء

برهان خلف : جهت اثبات حکم p، ابتدا فرض می‌کنیم نقیض آن یعنی $\sim p$ درست باشد. سپس به کمک قوانین و احکام اثبات شده قبلی به نتیجه‌ای برخلاف فرض اولیه یا قوانین اثبات شده قبلی می‌رسیم و نتیجه می‌گیریم که $\sim p$ نادرست است لذا p اثبات می‌شود.

استقرای ضعیف : می‌خواهیم ثابت کنیم حکم $A(n)$ به ازای $n \geq n_1$ درست است (n عدد صحیح است).

مراحل کار به صورت زیر است :

۱ـ **(پایه استقرا)** ثابت می‌کنیم $A(n)$ به ازای $n = n_1$ درست می باشد.

۲ـ **(فرض استقرا)** فرض می‌کنیم $A(n)$ به ازای عدد صحیح $k \geq n_1$ درست است.

۳ـ **(حکم استقرا)** ثابت می‌کنیم به ازای عدد $k + 1$ نیز درست خواهد بود.

روش برهان خلف و استقراء

استقرای قوی :

۱ـ **(پایه استقرا)** ثابت می‌کنیم $A(n)$ به ازای $n = n_1$ درست می‌باشد.

۲ـ **(فرض استقرا)** فرض می‌کنیم حکم $A(n)$ به ازای **همه مقادیر صحیح** i $(n_1 \leq i < k)$ درست است.

۳ـ **(حکم استقرا)** ثابت می‌کنیم $A(n)$ به ازای عدد صحیح k نیز درست خواهد بود.

فصل دوم

الگوریتم های بازگشتی

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.

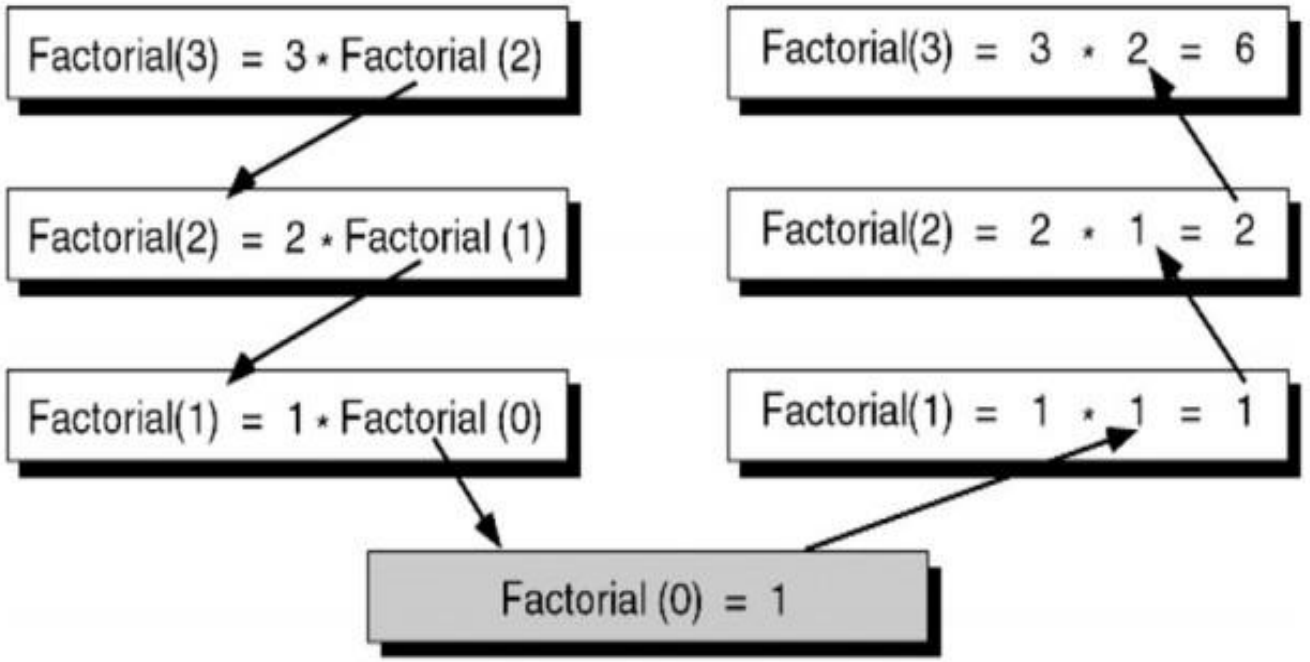
برای اینکه بتوان از روش بازگشتی برای حل یک مساله استفاده نمود، مساله باید قابلیت خرد شدن به زیرمساله هایی از همان نوع مساله اصلی و اندازه کوچکتر را داشته باشد.

الگوریتم های بازگشتی

راه حل بازگشتی یک مساله شامل دو مرحله کلی است:

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

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



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

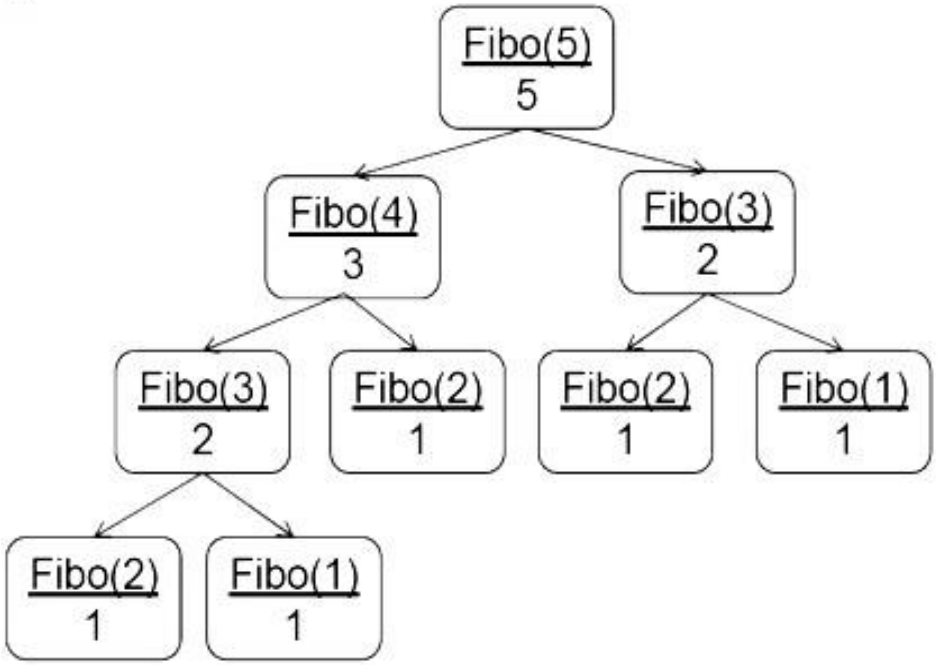
محاسبه زمان اجرای الگوریتم بازگشتی

- زمان حل زیرمساله
- زمان شکستن مساله به زیرمسائل
- زمان لازم برای ادغام جواب

الگوریتم های بازگشتی

محاسبه جمله n ام سری فیبوناچی

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



الگوریتم های بازگشتی

$$
\begin{aligned}
&Fib(5)\\
&= Fib(4) + Fib(3)\\
&= Fib(3) + Fib(2) + Fib(3)\\
&= Fib(2) + Fib(1) + Fib(2) + Fib(3)\\
&= Fib(1) + Fib(0) + Fib(1) + Fib(2) + Fib(3)\\
&= Fib(1) + Fib(0) + Fib(1) + Fib(1) + Fib(0) + Fib(3)\\
&= Fib(1) + Fib(0) + Fib(1) + Fib(1) + Fib(0) + Fib(2) + Fib(1)\\
&= Fib(1) + Fib(0) + Fib(1) + Fib(1) + Fib(0) + Fib(1) + Fib(0) + Fib(1)
\end{aligned}
$$

مشکل: محاسبه تکراری جملات

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

الگوریتم های بازگشتی

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

الگوریتم های بازگشتی

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.

مثال: بازگشتی زیر را به روش استقرای ریاضی حل کنید.

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

الگوریتم های بازگشتی

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

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

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

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

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

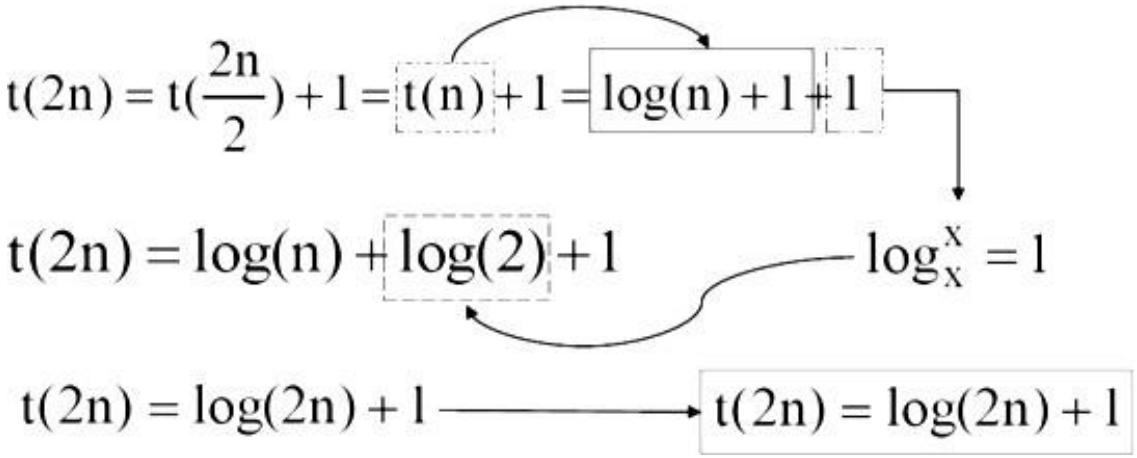
با استفاده از استقرا اثبات می کنیم که این حل درست است.

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

الگوریتم های بازگشتی



مثال: بازگشتی زیر را به روش استقرای ریاضی حل کنید.

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

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

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

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

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

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

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:

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

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

حل معادلات بازگشتی به روش جایگذاری و تکرار

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

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

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

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

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

$$t(1) = 1$$

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

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

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

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

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

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

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

فصل سوم

روش تقسیم و حل

روش تقسیم و حل

- استراتژی بکار رفته توسط ناپلئون
- نمونه ای از یک مساله را به صورت بازگشتی به تعدادی نمونه کوچکتر تقسیم کن تا زمانی که راه حل نمونه های کوچکتر به سادگی قابل تعیین باشند.
- رهیافت بالا به پایین که توسط روتین های بازگشتی به کار می رود.

روش تقسیم و حل

اگر X با عنصر وسط برابر است خارج شو، در غیر این صورت:

۱- **تقسیم** آرایه به دو زیر آرایه با اندازه ای تقریبا برابر نصف اندازه آرایه اولیه. اگر X کوچکتر از عنصر وسط می باشد، آرایه سمت چپ را انتخاب کن. اگر X بزرگتر از عنصر وسط می باشد، آرایه سمت راست را انتخاب کن.

۲- **حل** زیر آرایه به صورت بازگشتی با تعیین این که آیا X در آن زیر آرایه قرار دارد یا خیر، مگر این که اندازه زیر آرایه به اندازه کافی کوچک باشد.

۳- راه حل آرایه را با توجه به راه حل زیر آرایه تعیین کن.

روش تقسیم و حل

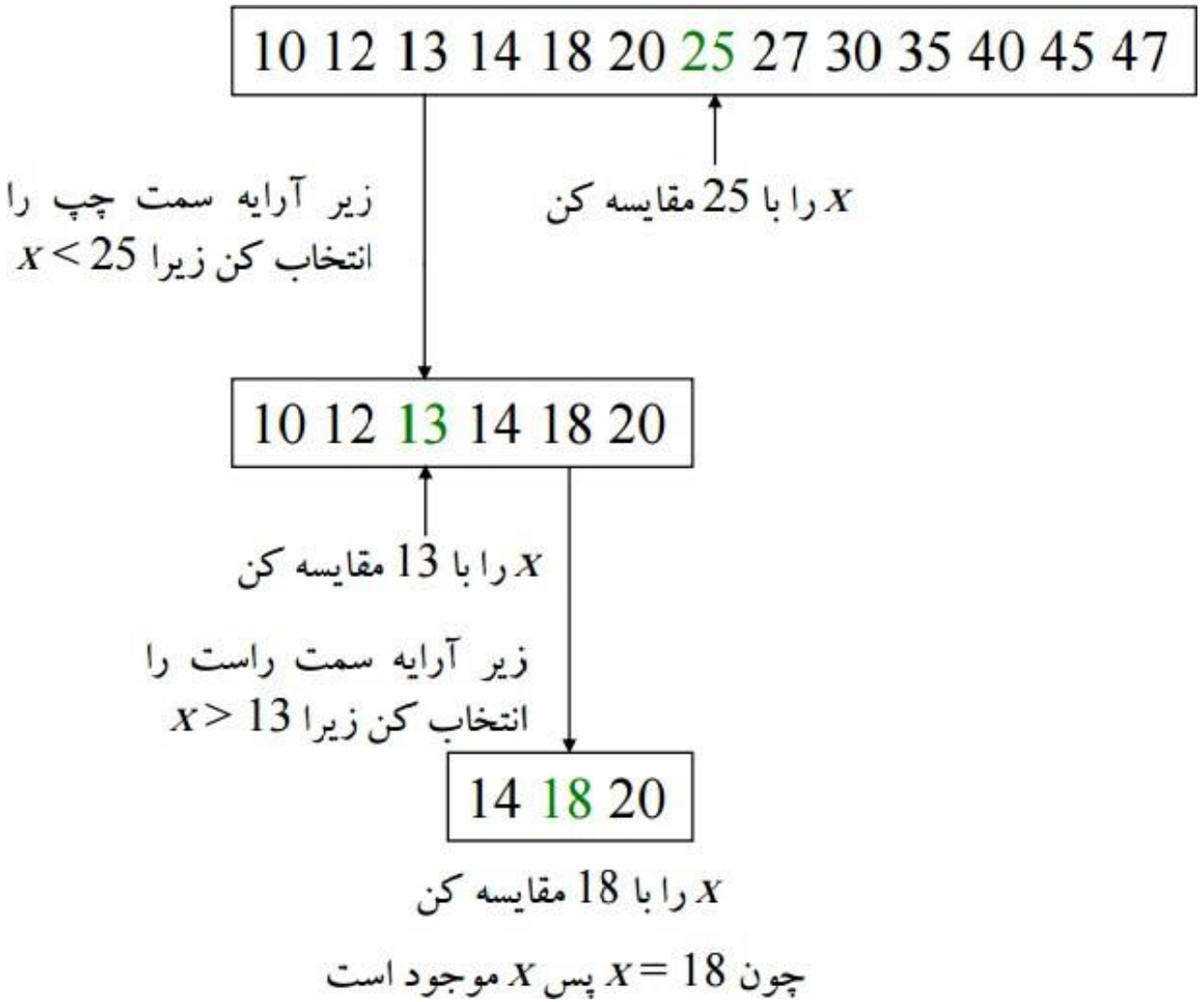
- فرض کنید $x = 18$ و آرایه به صورت زیر باشد:

10 12 13 14 18 20 25 27 30 35 40 45 47

↑

عنصر وسط

روش تقسیم و حل



جستجوی دودویی (بازگشتی)

- مساله: تعیین کنید که آیا X در آرایه مرتب شده S به اندازه n وجود دارد یا خیر.
- ورودی ها: عدد صحیح و مثبت n، آرایه مرتب S که از ۱ تا n اندیس گذاری شده است، کلید X.
- خروجی ها: $location$، موقعیت X در S (اگر X در S نباشد برابر صفر می باشد)

الگوریتم جستجوی دودویی

```
index location ( index low, index high)
{
    index mid,
    if ( low > high)
        return 0;
    else {
        mid = ⌊( low + high) ⌋ / 2;
        if ( x == S[mid])
            return mid,
        else if ( x < S[mid])
            return location ( low, mid – 1);
        else
            return location ( mid + 1, high);
    }
}
```

پیچیدگی زمانی :بدترین حالت

- عمل اصلی: مقایسه x با $S[mid]$
- اندازه ورودی: تعداد عناصر آرایه، n
- پیچیدگی زمانی:

$$\begin{cases} W(n) = W(\frac{n}{2}) + 1 & for \quad n > 1, \quad n \quad a \quad power \quad of \quad 2 \\ W(1) = 1 \end{cases}$$

- حل: $W(n) = \lg n + 1$

و اگر n به توانی از دو محدود نباشد:

$$W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$$

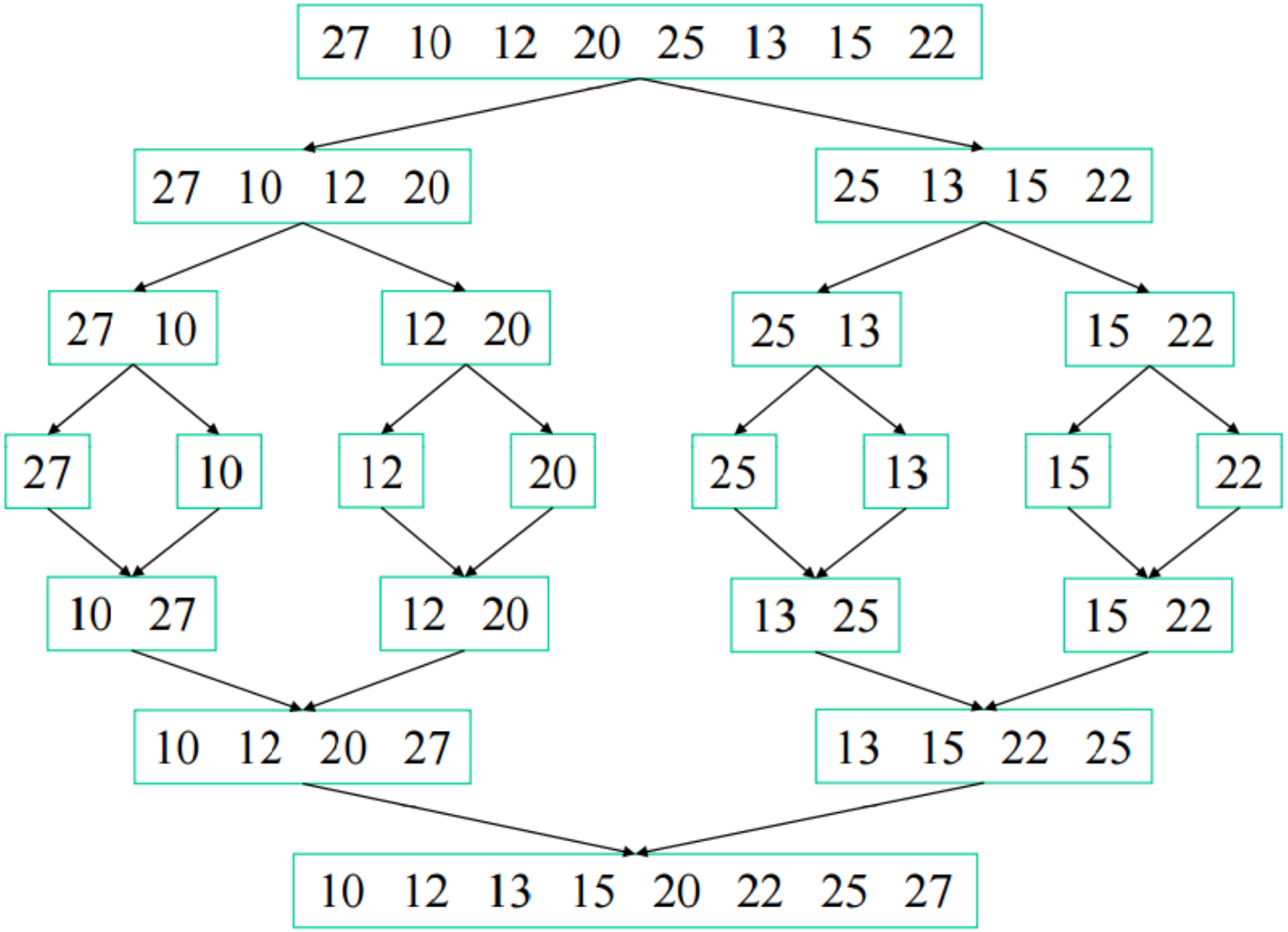
مرتب سازی ادغامی

- **تقسیم** آرایه به دو زیر آرایه به اندازه $n / 2$ عنصر.

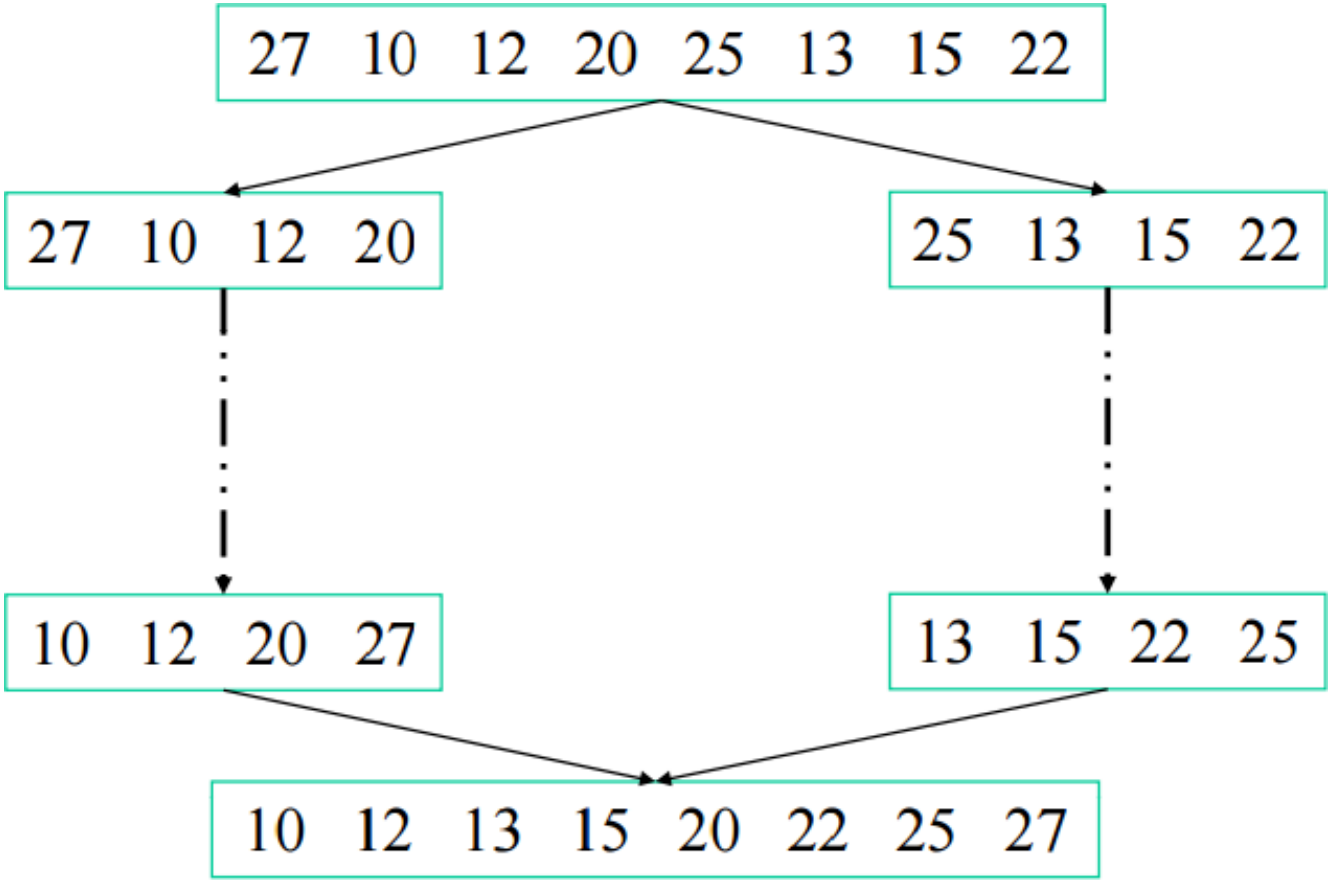
- **حل** هر یک از زیر آرایه ها با مرتب سازی آن. اگر زیر آرایه به اندازه کافی کوچک نبود، برای حل آن به روش بازگشتی عمل می کنیم.

- **ترکیب** راه حل های زیر آرایه ها با ادغام آن ها در یک آرایه مرتب.

روش تقسیم و حل



روش تقسیم و حل



الگوریتم مرتب سازی ادغامی

Merge sort

Problem: Sort n keys in nondecreasing order.

Inputs: positive integer n, array of keys S indexed from 1 to n.

Outputs: the array S containing the keys in nondecreasing order.

```
void mergesort ( int n, keytype S[ ] )
{
    const int h=⌊n/2⌋, m= n– h;
    keytype U[1..h], V[1..m];
    if (n> 1) {
        copy S[1] through S[h] to U[1] through U[h];
        copy S[h+ 1] through S[n] to V[1] through V[m];
        mergesort(h, U);
        mergesort(m, V);
        merge(h, m, U, V, S);
    }
}
```

الگوریتم ادغام

Merge

Problem: merge two sorted array into one sorted array.

Inputs: positive integer *h and m*, array of sorted keys *U* indexed from 1 to *h*, array of sorted keys *V* indexed from 1 to *m*.

Outputs: the array *S* containing the keys in nondecreasing order.

```
void merge ( int h, int m, const keytype U[],
                           const keytype V[],
                                 keytype S[])
{
   index i, j, k;
   i = 1; j = 1; k = 1;
   while ( i <= h && j <= m) {
      if ( U[i] < V[j]) {
         S[k] = U[i];
         i++;
      }
      else {
         S[k] = V[j];
         j++;
      }
      k++;
   } // end of while
   if ( i > h)
      copy V[j] thtough V[m] to S[k] through S[h + m];
   else
      copy U[i] thtough U[h] to S[k] through S[h + m];
}
```

پیچیدگی زمانی ادغام: بدترین حالت

- عمل اصلی: مقایسه $U[i]$ با $V[j]$
- اندازه ورودی: h و m، تعداد عناصر موجود در هر یک از دو آرایه ورودی
- پیچیدگی زمانی:

$$W(h, m) = h + m - 1$$

- یعنی زمانی که هنگام خروج از حلقه به دلیل مقایسه تمام عناصر یکی از آرایه ها، آرایه دیگر فقط یک عنصر مقایسه نشده داشته باشد.

پیچیدگی زمانی مرتب سازی ادغامی: بدترین حالت

- عمل اصلی: مقایسه ای که در *merge* انجام می شود.
- اندازه ورودی: n، تعداد عناصر آرایه S.
- پیچیدگی زمانی:

$$W(n) = W(h) + W(m) + h + m - 1$$

- اگر n توانی از ۲ باشد:

$$\begin{cases} W(n) = 2W(\frac{n}{2}) + n - 1 & for \quad n > 1, \quad n \quad a \quad power \quad of \quad 2 \\ W(1) = 0 \end{cases}$$

- حل: (مثال ۱۹ از پیوست ۲)

$$W(n) = n\lg n - (n - 1) \in \Theta(n\lg n)$$

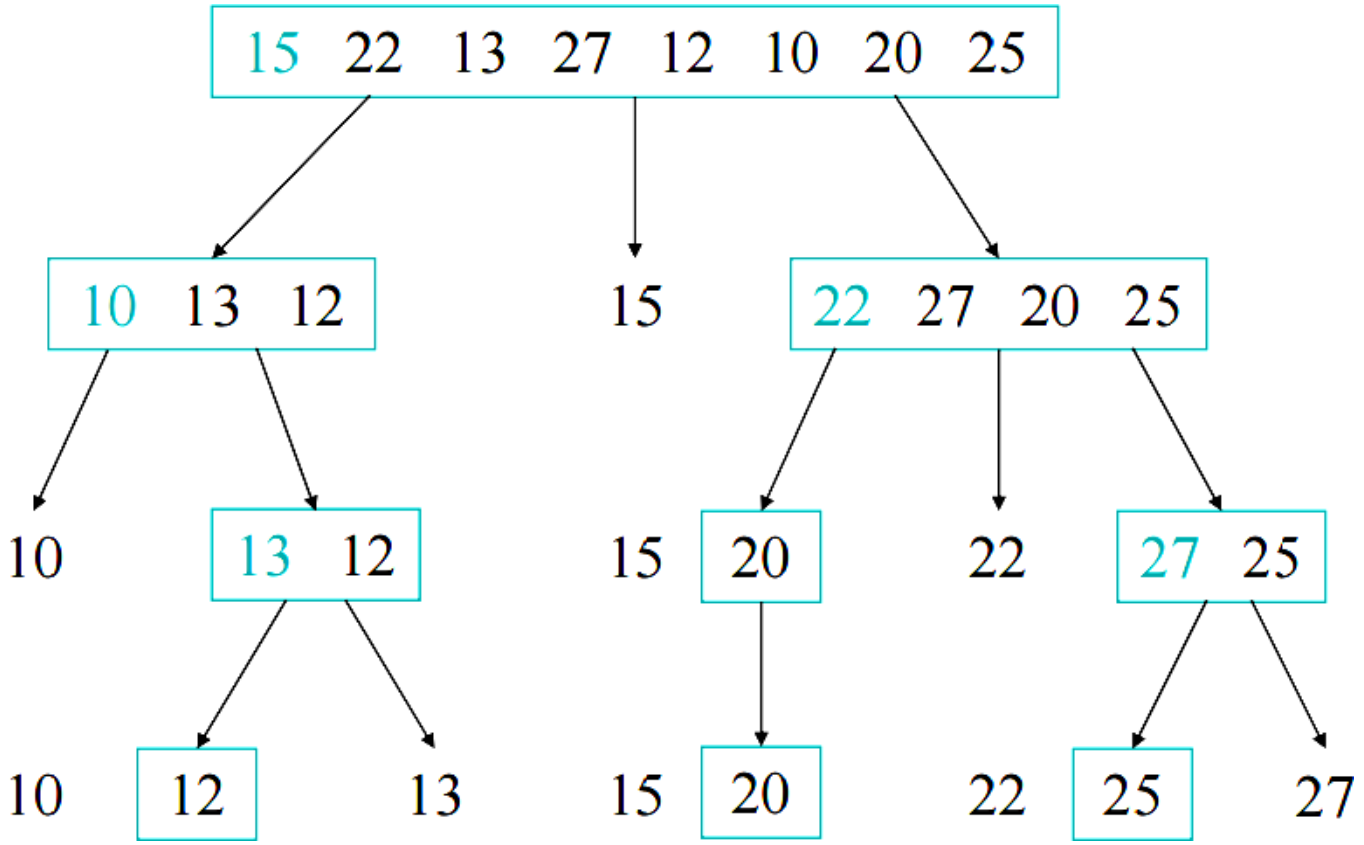
- و اگر n به توانی از ۲ محدود نباشد:

$$W(n) = W(\lfloor \frac{n}{2} \rfloor) + W(\lceil \frac{n}{2} \rceil) + n - 1 \Rightarrow W(n) = \theta(n\lg n)$$

مرتب سازی سریع

- مراحل:
  - انتخاب **عنصر محوری** (معمولا عنصر اول)
  - تقسیم آرایه به دو بخش به طوری که عناصر کوچکتر از عنصر محوری در سمت چپ و عناصر بزرگتر از آن در سمت راست آن قرار بگیرند.
  - مرتب سازی هر بخش به صورت بازگشتی

روش تقسیم و حل



روش تقسیم و حل

Quick Sort

```
procedure Partition (Var x: Arraylist; left, right : integer; Var pivotpoint : integer
Var i,j,pivot : integer;
begin
        i := left ;     j := right+1;     pivot := x[left];
        repeat
                repeat
                        i := i + 1;
                until   x[i] >= pivot;
                repeat
                        j := j - 1;
                until  x[j] <= pivot;
                if i<j   then   swap (x[i], x[j]);
        until   i>= j;
       swap (x[left], x[j]);
       pivotpoint := j;
end;
{ * * * * * * * * * * * * * * * * * * * }
procedure Quicksort (Var  x: Arraylist; left, right : integer) ;
var    pivotpoint : integer;
begin
        if (left  < right)    then   begin
                partition (x , left, right, pivotpoint); {سر لولا در محل واقعی خود قرار می‌گیرد
                QuickSort(x, left, pivotpoint - 1); {مرتب‌سازی زیر لیست چپ}
                QuickSort (x, pivotpoint+1, right); { مرتب‌سازی زیر لیست راست }
        end;    {if}
end;
```

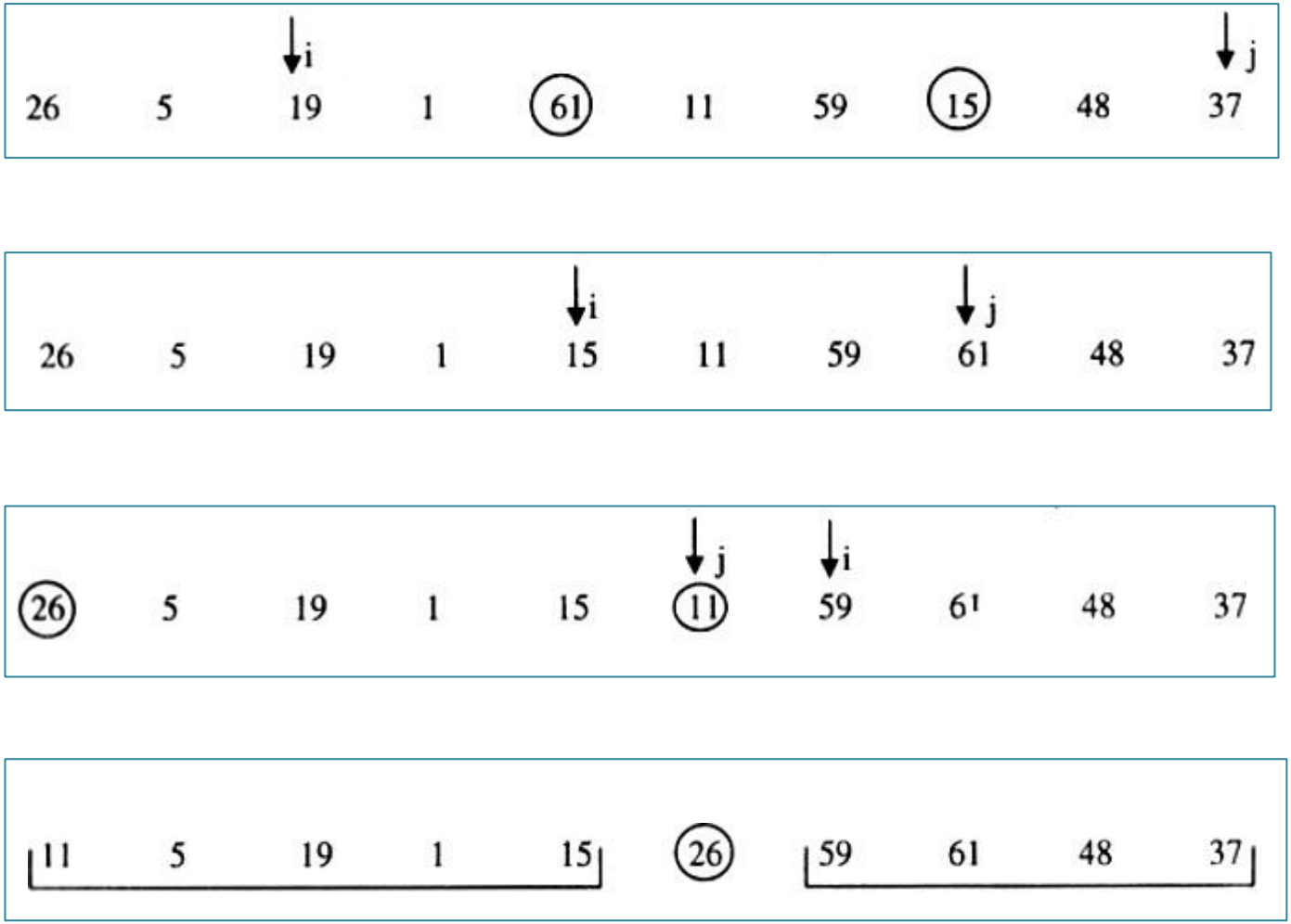
روش تقسیم و حل

پردازه Partition آرایه را به گونه‌ای افراز می‌کند که عنصر محوری (اولین عنصر سمت چپ آرایه) در مکان درست خود قرار گیرد و همچنین مکان قرارگیری آن در آرایه توسط متغیر خروجی pivotpoint برگردانده می‌شود.

فرض کنید آرایه اولیه به صورت زیر باشد انجام یک مرحله از الگوریتم فوق به صورت زیر است:

x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]
26	5	37	1	61	11	59	15	48	19

روش تقسیم و حل



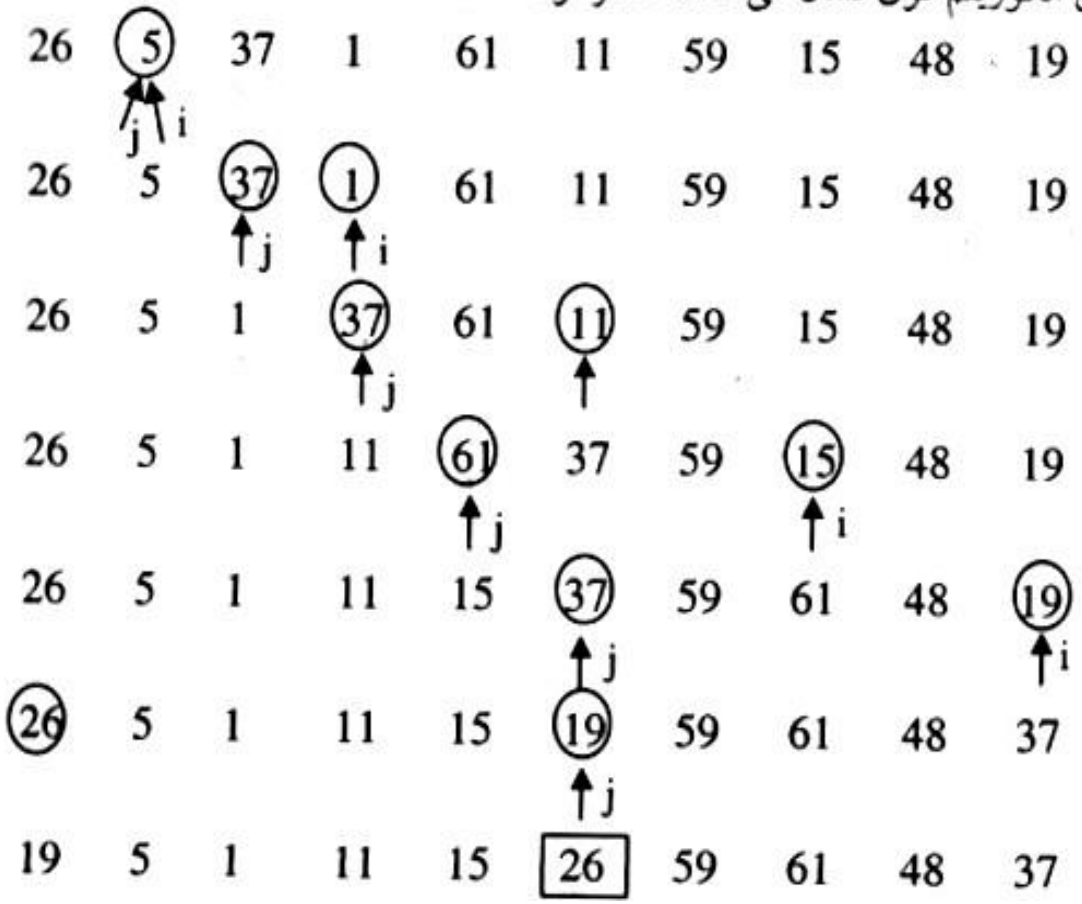
روش تقسیم و حل

[26	5	37	1	61	11	59	15	48	19
[11	5	19	1	15]	(26)	[59	61	48	37
[1	5]	(11)	[19	15]	26	[59	61	48	37
(1)	(5)	11	[19	15]	26	[59	61	48	37]
1	5	11	(15)	(19)	26	[59	61	48	37]
1	5	11	15	19	26	[48	37]	(59)	[61
1	5	11	15	19	26	37	48	59	[61
1	5	11	15	19	26	37	48	59	61

```
procedure partition (Var  x: Arraylist; left, right : integer; Var pivotpoint : integer)
Var  i,j, pivot : integer;
begin
   j := left;      pivot := x[left];
   for  i := left +1    to   right   do
      if  x[i] < pivot  then  begin
            j := j + 1 ;
            swap (s[i], s[j]);
        end;      -
  swap (x[left], x[j]);
  pivotpoint := j;
end;
```

روش تقسیم و حل

نمونه زیر مراحل کار افراز را مطابق الگوریتم فوق نشان می‌دهد. عنصر لولا عدد 26 است.



روش تقسیم و حل

26	5	37	1	61	11	59	15	48	19
19	5	37	1	61	11	59	15	48	26
19	5	26	1	61	11	59	15	48	37
19	5	15	1	61	11	59	26	48	37
19	5	15	1	26	11	59	61	48	37
19	5	15	1	11	26	59	61	48	37

پیچیدگی زمانی زمان بخشی (همه حالات)

- عمل اصلی: مقایسه $S[i]$ با $pivotitem$
- اندازه ورودی: $n = high - low + 1$ (اندازه زیر آرایه)
- پیچیدگی زمانی: از آنجا که هر یک از عناصر (به جز اولی) یک بار مقایسه می شوند:

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

پیچیدگی زمانی (بدترین حالات)

- عمل اصلی: مقایسه $S[i]$ با $pivotitem$ در رویه $partition$
- اندازه ورودی: n، اندازه آرایه S
- پیچیدگی زمانی:

$$T(n) = \underbrace{T(0)}_{\substack{Time\ to\ sort \\ left\ subarray}} + \underbrace{T(n-1)}_{\substack{Time\ to\ sort \\ right\ subarray}} + \underbrace{n-1}_{\substack{Time\ to \\ partition}}$$

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

- حل:

$$T(n) = n(n-1)/2 \in \Theta(n^2)$$

پیچیدگی زمانی (حالت متوسط)

- عمل اصلی: مقایسه $S[i]$ با $pivotitem$ در رویه $partition$
- اندازه ورودی: n، اندازه آرایه S
- پیچیدگی زمانی:

$$A(n)=\sum_{p=1}^{n}\overbrace{\frac{1}{n}}^{\Pr obability\ pivotpo\text{int}\ is\ p}\underbrace{[A(p-1)+A(n-p)]}_{\substack{Average\ time\ to\\ sort\ subarrays\ when\\ pivotpo\text{int}\ is\ p}}+\underbrace{n-1}_{\substack{Time\ to\\ partition}}$$

- حل:

$$A(n)\approx 1.38(n+1)\lg n\in\Theta(n\lg n)$$

	بهترین حالت	حالت متوسط	بدترین حالت
مرتب‌سازی سریع	O (n logn)	O (n logn)	$O(n^2)$
مرتب‌سازی ادغام	O (n logn)	O (n logn)	O (n logn)

الگوریتم ضرب ماتریس ها به روش استراسن

- ضرب ماتریس ها طبق تعریف:
  - تعداد ضرب ها: $T(n) = n^3$
  - تعداد جمع ها: $T(n) = n^3 - n^2$
- الگوریتم استراسن برای ضرب ماتریس ها (1969)
  - پیچیدگی بهتر از درجه سوم (جمع و ضرب)

روش تقسیم و حل

- برای محاسبه:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

- تعریف می کنیم:

$$m_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$
$$m_2 = (a_{21} + a_{22})b_{11}$$
$$m_3 = a_{11}(b_{12} - b_{22})$$
$$m_4 = a_{22}(b_{21} - b_{11})$$
$$m_5 = (a_{11} + a_{12})b_{22}$$
$$m_6 = (a_{21} - a_{11})(b_{11} + b_{22})$$
$$m_7 = (a_{12} - a_{22})(b_{21} + b_{22})$$

- آنگاه :

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

روش تقسیم و حل

- تقسیم ماتریس ها:

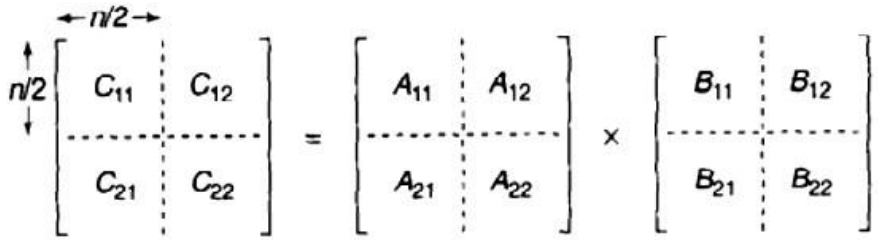


Figure 2.4 • The partitioning into submatrices in Strassen's algorithm.

- محاسبه M ها، مثلا:

$$M_1 = (A_{11} + A_{22})\,(B_{11} + B_{22})$$

روش تقسیم و حل

- وقتی $n = 4$

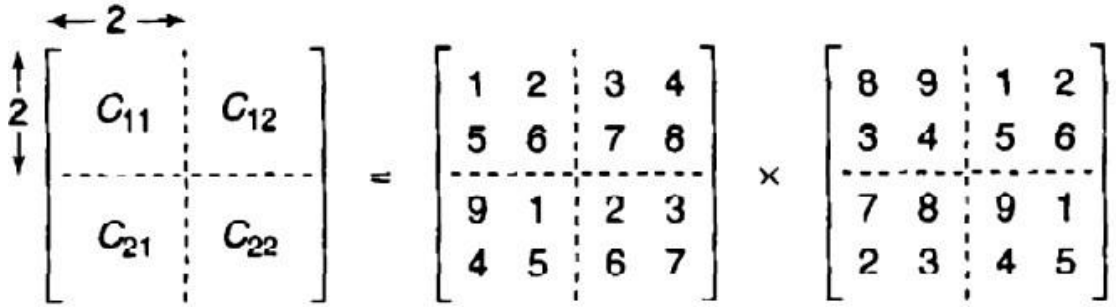


Figure 2.5 ● The partitioning in Strassen's algorithm with $n = 4$ and values given to the matrices.

- محاسبه M ها، مثلا:

$$M_1 = \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}$$

پیچیدگی زمانی (همه حالات - ضرب)

- عمل اصلی: یک ضرب ساده
- اندازه ورودی: n، تعداد سطرها و ستون های ماتریس ها
- پیچیدگی زمانی:

$$\begin{cases} T(n) = 7T(\frac{n}{2}) & for \quad n>1, \quad n \quad a \quad power \quad of \quad 2 \\ T(1) = 1 \end{cases}$$

- حل:

$$T(n) = n^{\lg 7} \approx n^{2.81} \in \Theta(n^{2.81})$$

پیچیدگی زمانی (همه حالات - جمع و تفریق)

- عمل اصلی: یک جمع یا تفریق ساده
- اندازه ورودی: n، تعداد سطرها و ستون های ماتریس ها
- پیچیدگی زمانی:

$$\begin{cases} T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2 & for \quad n>1, \quad n \quad a \quad power \quad of \quad 2 \\ T(1) = 0 \end{cases}$$

- حل:

$$T(n) = 6n^{\lg 7} - 6n^2 \approx 6n^{2.81} - 6n^2 \in \Theta(n^{2.81})$$

عمل	روش استاندارد	روش استراسن
ضرب	n^3	$n^{2.81}$
جمع / تفریق	$n^3 - n^2$	$6n^{2.81} - 6n^2$

محاسبه با اعداد صحیح بزرگ

- نمایش اعداد صحیح بزرگ: جمع و عملیات خطی دیگر
  - استفاده از آرایه ای از اعداد صحیح، در هر خانه یک رقم
  - ذخیره علامت در بالاترین محل آرایه
  - الگوریتم های زمان خطی:
    - جمع
    - تفریق
    - $u \times 10^m$
    - u divide 10^m
    - u rem 10^m

ضرب اعداد صحیح بزرگ

- تقسیم یک عدد صحیح n-رقمی به دو عدد صحیح که هر کدام تقریبا دارای $n/2$ رقم می باشند. مثلا:

  - $567{,}832 = 567 \times 10^3 + 832$
  - $9{,}423{,}723 = 9423 \times 10^3 + 723$

- به طور کلی:

$$\underset{n \text{ digits}}{u} = \underset{\lceil n/2 \rceil \text{ digits}}{x} \times 10^m + \underset{\lfloor n/2 \rfloor \text{ digits}}{y}$$

که $m = \lfloor n/2 \rfloor$

الگوریتم ضرب اعداد صحیح بزرگ

```
Large_int   prod (Large_int   u, Large_int   v)
  {
  Large_int   x,y,w,z;
   int  n,m ;
   n = Maximum (number of digits in u, number of digits in v)
   if (u == 0 || v == 0)    return  0;
   else  if(n <= threshold)
        return    u × v;
    else   {
         m = ⌊n / 2⌋ ;
         x = u   divide   10^m ;     y = u   rem   10^m ;
        w = v  divide   10^m ;  z = v  rem  10^m;
        return   prod (x,w) × 10^2m + (prod(x,z) + prod(w,y)) × 10^m + prod (y,z);
    }
}
```

روش تقسیم و حل

- برای ضرب نمودن دو عدد صحیح n-رقمی

$$u = x \times 10^m + y$$
$$v = w \times 10^m + z$$

- حاصل ضرب برابر است با:

$$uv = xw \times 10^{2m} + (xz + wy) \times 10^m + yz$$

- مثال:

$$567{,}832 \times 9{,}423{,}723 = (567 \times 10^3 + 832)(9423 \times 10^3 + 723) =$$
$$567 \times 9423 \times 10^6 + (567 \times 723 + 9423 \times 832) \times 10^3 + 832 \times 723$$

روش تقسیم و حل

زمانی که نباید از تقسیم و حل استفاده کرد.

- یک نمونه به اندازه n به دو یا چند نمونه تقسیم شود به طوری که اندازه هر یک از این نمونه ها تقریبا برابر اندازه نمونه اصلی باشد. ← نمایی
  - مثال: دنباله فیبوناچی
  - استثناء : مساله برج های هانوی (تمرین ۱۷)

- یک نمونه به اندازه n تقریبا به n نمونه با اندازه های n/c تقسیم شود به طوری که c یک عدد ثابت باشد. ← $\Theta(n^{\lg n})$

فصل سوم

برنامه نویسی پویا

علت ناکارآمدی تقسیم و حل

- بعد از تقسیم ...
  - نمونه های کوچکتر غیر مرتبط هستند، مانند مرتب سازی ادغامی
  - نمونه های کوچکتر مرتبط هستند، مانند فیبونانچی
    - حل نمونه های مشترک به طور مکرر
- برنامه نویسی پویا
  - رهیافت پایین به بالا (bottom-up)
  - با استفاده از یک آرایه (جدول) برای ذخیره کردن راه حل نمونه های کوچکتر

مراحل

- ارایه یک خاصیت بازگشتی برای حل نمونه ای از مساله
- حل نمونه ای از مساله به روش پایین به بالا با حل نمونه های کوچکتر در ابتدا

مثال : رابطه بازگشتی $T(n) = T(n-1) + 3$ با حالت خاص $T(1) = 1$ را حل کنید.

روش اول (بالا به پائین) :

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

روش دوم (پائین به بالا) :

$$T(1) = 1$$
$$T(2) = T(1) + 3 = 1 + 3 = 4$$
$$T(3) = T(2) + 3 = 4 + 3 = 7$$
$$T(4) = T(3) + 3 = 7 + 3 = 10$$
$$T(5) = T(4) + 3 = 10 + 3 = 13$$
$$\vdots$$
$$T(n) = 3n - 2$$

مثال : سری فیبوناچی

روش پویا

```
int   fib(int  n)
{
  int i, 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];
}
```

روش تقسیم و غلبه

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

مثلاً الگوریتم پویای فوق را می‌توان به صورت زیر نیز انجام داد :

```
int   fib (int n)
{
   int  f1, f2, f;
   f1 = 0;     f2 = 1;
        if(n == 0)  return  f1;
        else  if(n ==1) return  f2;
   for (i=2;  i<= n; i++)
   {
        f = f1 + f2;
        f1 = f2;
        f2 = f;
   }
   return f;
}
```

پس مراحل ایجاد یک الگوریتم پویا شامل مراحل زیر است :

۱ـ ایجاد یک خاصیت بازگشتی برای حل نمونه‌ای از مسأله

۲ـ حل نمونه‌ای از مسأله با روش جزء به کل از طریق حل نمونه‌های کوچکتر

مثال : ضریب دو جمله ای

- تعریف

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad for \quad 0 \le k \le n$$

- تعریف بازگشتی

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 \quad k = 0 \quad or \quad k = n \end{cases}$$

پیاده سازی فرمول قبل با تقسیم و حل

```
int  bin(int  n, int k)
{
   if (k ==0  || n == k)   return 1 ;
   else  return  bin (n-1 , k-1) + bin (n-1, k);
}
```

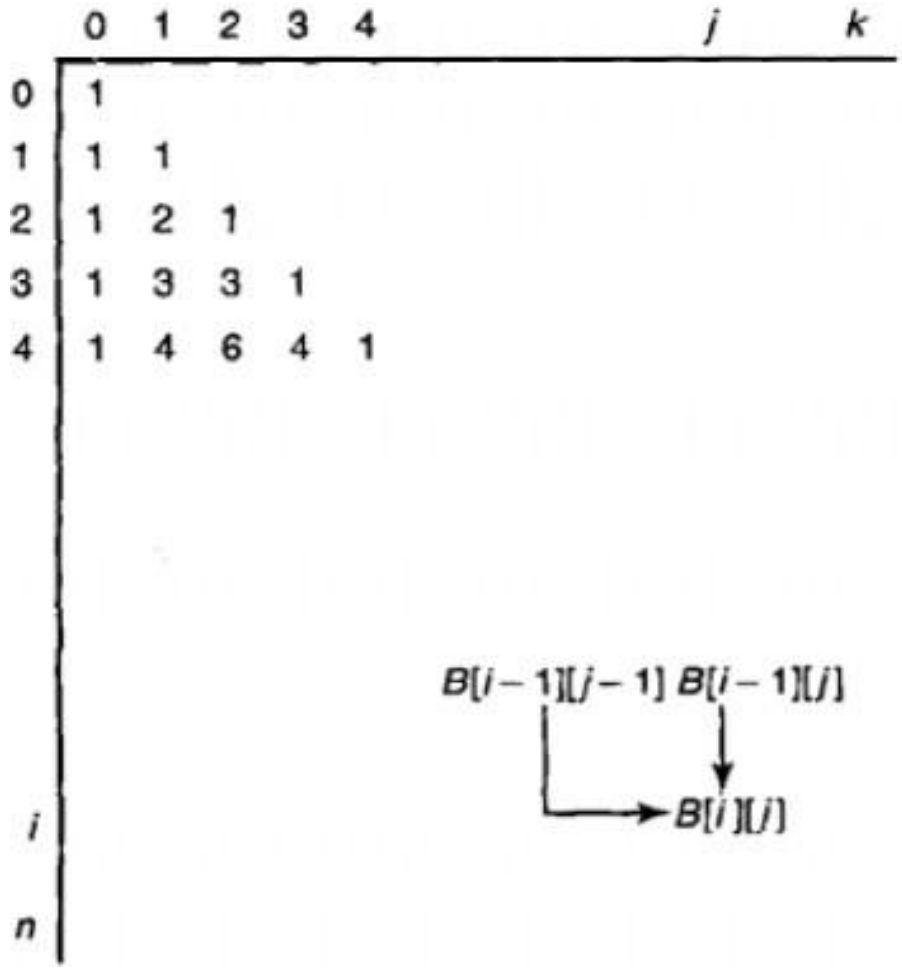
$$2\binom{n}{k}-1$$

استفاده از برنامه نویسی پویا

- استفاده از یک آرایه B به منظور ذخیره کردن ضرایب
- مراحل:
  - بنا نهادن یک خاصیت بازگشتی

$$B[i][j]=\begin{cases} B[i-1][j-1]+B[i-1][j] & 0<j<i \\ 1 & j=0 \quad or \quad j=i \end{cases}$$

  - حل یک نمونه مساله به روش پایین به بالا با محاسبه نمودن سطرهای B به طور متوالی با شروع از سطر اول

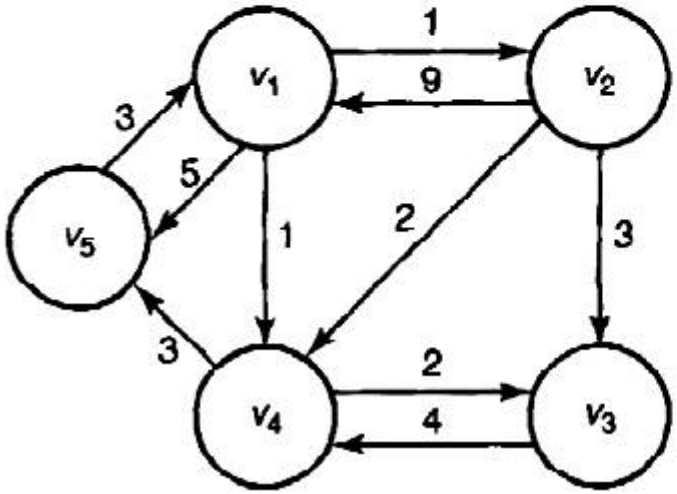

0 1 2 3 4 j k
0 1
1 1 1
2 1 2 1
3 1 3 3 1
4 1 4 6 4 1
i
n
B[i − 1][j − 1] B[i − 1][j]
B[i][j]

منظور از B[i][j] عبارت $\binom{i}{j}$ و $\binom{n}{k}$ = B[n][k] است. برنامه‌ای که ماتریس فوق را ایجاد می‌کند به صورت زیر است:

```
int   bin2 (int  n , int  k)
{
   int  i,j ;   int  B[0 .. n] [0 .. k];
   for (i=0;  i <= n ; i ++)
     for (j=0; j <=minimum(i,k); j++)
     {
          if (j == 0 || j == i)
                B[i][j] = 1 ;
          else  B[i][j] = B[i-1][j-1] + B[i-1][j];
     }
  return B[n][k];
}
```

الگوریتم فلوید برای محاسبه کوتاه ترین مسیر ها

- اصطلاحات:
  - گراف : رئوس، یال ها (کمان ها)
  - گراف جهت دار (دایگراف)
  - گراف وزن دار



اصطلاحات

- مسیر
- دور
- گراف دور دار/ بدون دور
- مسیر ساده
- طول یک مسیر

مساله کوتاه ترین مسیر

- مساله بهینه سازی
  - می تواند بیش از یک راه حل کاندیدا وجود داشته باشد
  - هر راه حل کاندیدا دارای یک مقدار می باشد.
  - راه حل، یک راه حل کاندیدا می باشد که دارای مقدار بهینه می باشد.

- الگوریتم brute force (الگوریتمی که تمام حالت های ممکن را در نظر می گیرد) دارای پیچیدگی زمانی به صورت فاکتوریل می باشد:

$$(n-2)(n-3)\ldots 1 = (n-2)!$$

نمایش گراف

- ماتریس مجاورتی

$$W[i][j] = \begin{cases} \text{وزن یال} & \text{اگر یالی بین } V_i \text{ و } V_j \text{ باشد} \\ \infty & \text{اگر یالی بین } V_i \text{ و } V_j \text{ نباشد} \\ 0 & \text{اگر } i=j \text{ باشد} \end{cases}$$

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

الگوریتم

- ایجاد دنباله ای از $n+1$ ماتریس $D^{(k)}$ به طوری که $0 \le k \le n$

$$D^{(0)}, D^{(1)}, D^{(2)}, \ldots, D^{(n)}$$

$D^{(k)}[i][j]$ = طول کوتاه ترین مسیر از V_i به V_j که فقط از رئوس موجود در مجموعه $\{V_1, V_2, \ldots, V_k\}$ به عنوان رئوس میانی استفاده می کند.

$$D^{(0)} = W$$
$$D^{(n)} = D$$

مثال

- محاسبه $D[2][5]$

$D^0[2][5] = length[v_2, v_5] = \infty$

$D^1[2][5] = \text{minimum}(length[v_2, v_5], length[v_2, v_1, v_5]$
$= \text{minimum}(\infty, 14) = 14$

$D^2[2][5] = D^1[2][5] = 14$

$D^3[2][5] = D^2[2][5] = 14$

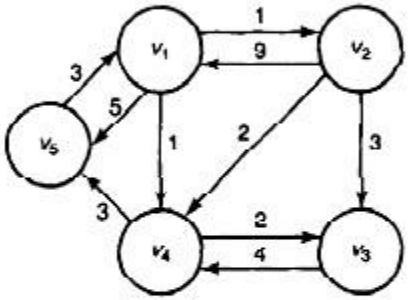


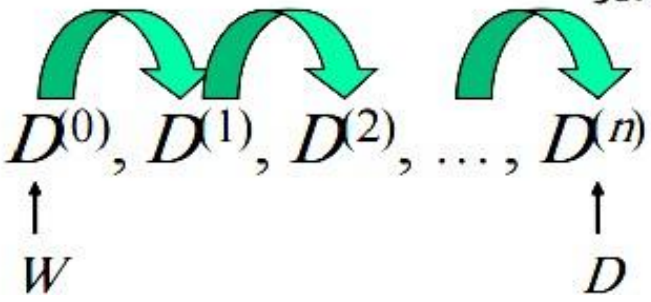
Figure 3.2 • A weighted, directed graph.

$D^4[2][5] = \text{minimum}(length[v_2, v_1, v_5], length[v_2, v_4, v_5],$
$length[v_2, v_1, v_4, v_5], length[v_2, v_3, v_4, v_5])$
$= \text{minimum}(14, 5, 13, 10) = 5$

$D^5[2][5] = D^4[2][5] = 5$

برنامه نویسی پویا برای مساله کوتاه ترین مسیر

- باید راهی برای محاسبه $D = D^{(n)}$ از روی $W = D^{(0)}$ به دست آوریم:
  - ارایه یک خاصیت بازگشتی برای محاسبه کردن $D^{(k)}$ از $D^{(k-1)}$
  - حل نمونه ای از مساله به روش پایین به بالا بازاء $k = 1$ تا n و ایجاد دنباله زیر:



الگوریتم فلوید

$D^{(0)} = W;$

for ($k = 1$; $k <= n$, k++)

compute $D^{(k)}$ from $D^{(k-1)}$;

محاسبه $D^{(k)}$ از روی $D^{(k-1)}$

- **حالت ۱:** حداقل یک نمونه از کوتاه ترین مسیرها از V_i به V_j که فقط از رئوس موجود در مجموعه $\{V_1, V_2, \ldots, V_k\}$ به عنوان رئوس میانی استفاده می کند، از V_k عبور نکند. در این صورت:

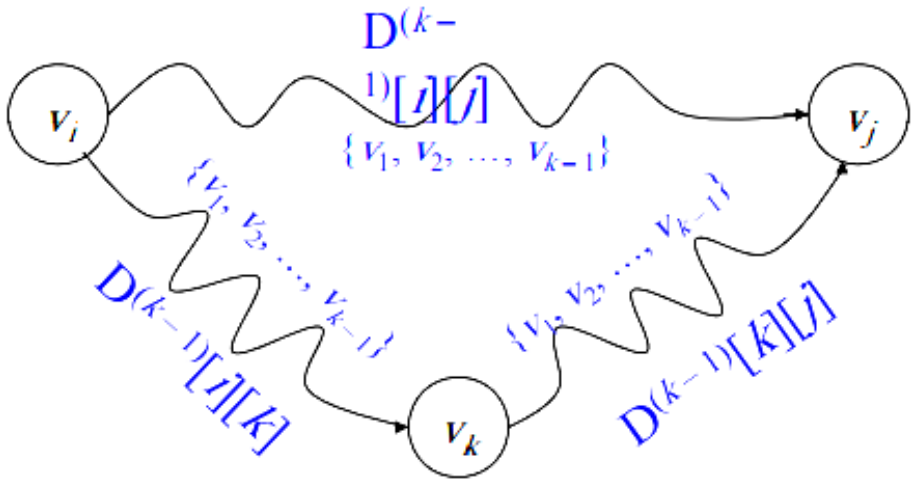
$$D^{(k)}[i][j] = D^{(k-1)}[i][j]$$

- **حالت ۲:** تمام نمونه های کوتاه ترین مسیر از V_i به V_j که فقط از رئوس موجود در مجموعه $\{V_1, V_2, \ldots, V_k\}$ به عنوان رئوس میانی استفاده می کند، از V_k عبور کنند. در این صورت:

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$$

ارائه خاصیت بازگشتی: محاسبه $D^{(k)}$ از روی $D^{(k-1)}$

$$D^{(k-1)} \Rightarrow D^{(k)}$$



$$D^{(k)}[i][j] = minimum(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

مثال

- محاسبه $D^{(2)}[5][4]$

$$D^{(1)}[2][4] = \text{minimum}(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4])$$
$$= \text{minimum}(2, 9 + 1) = 2$$

$$D^{(1)}[5][2] = \text{minimum}(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2])$$
$$= \text{minimum}(\infty, 3 + 1) = 4$$

$$D^{(1)}[5][4] = \text{minimum}(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4])$$
$$= \text{minimum}(\infty, 3 + 1) = 4$$

$$D^{(2)}[5][4] = \text{minimum}(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4])$$
$$= \text{minimum}(4, 4 + 2) = 4$$

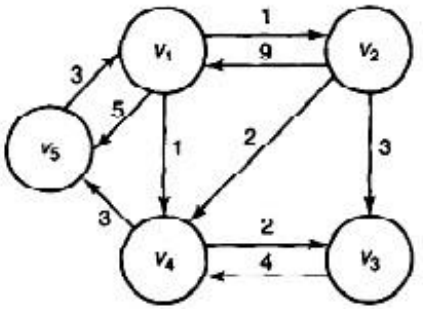


Figure 3.2 • A weighted, directed graph.

محاسبه $D^{(k)}$ از روی $D^{(k-1)}$

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        D(k)[i][j] = min(D(k-1)[i][j], D(k-1)[i][k] + D(k-1)[k][j]);
```

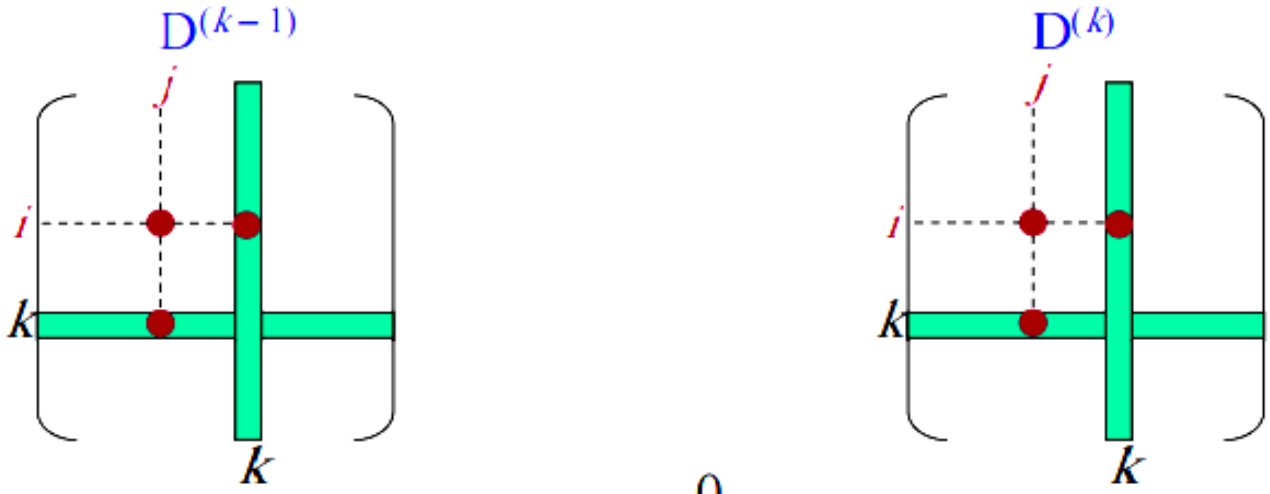
الگوریتم فلوید

$D^{(0)} = W;$

```
for (k = 1; k <= n; i++)
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            D^(k)[i][j] = min(D^(k-1)[i][j], D^(k-1)[i][k] + D^(k-1)[k][j]);
```

الگوریتم فلوید

$$D^{(k)}[i][j] = minimum(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$



$$D^{(k)}[k][j] = minimum(D^{(k-1)}[k][j], \overbrace{D^{(k-1)}[k][k]}^{0} + D^{(k-1)}[k][j]) = D^{(k-1)}[k][j]$$

$$D^{(k)}[i][k] = minimum(D^{(k-1)}[i][k], D^{(k-1)}[i][k] + \overbrace{D^{(k-1)}[k][k]}^{0}) = D^{(k-1)}[k][j]$$

پیچیدگی زمانی برای همه حالات

- عمل اصلی: دستور واقع در حلقه for-j
- اندازه ورودی: n، تعداد رئوس گراف
- پیچیدگی زمانی:

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$$

برنامه نویسی پویا و مسایل بهینه سازی

- مراحل:

– ارائه یک خاصیت بازگشتی برای بدست آوردن راه حل بهینه نمونه ای از مساله

– محاسبه مقدار راه حل بهینه به روش پایین به بالا

– ساختن یک راه حل بهینه به روش پایین به بالا

اصل بهینگی

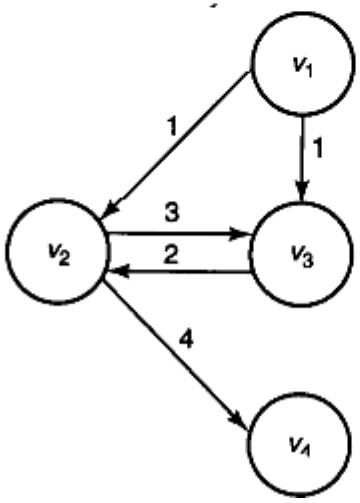
- یک راه حل بهینه برای یک نمونه از مساله همواره شامل راه حل های بهینه برای تمامی زیر نمونه ها می باشد.

- اطمینان می دهد که راه حل بهینه یک نمونه از مساله می تواند با ترکیب راه حل های بهینه زیر نمونه ها حاصل شود.

- قبل از استفاده از برنامه نویسی پویا برای بدست آوردن راه حل، لازم است که نشان دهیم اصل بهینگی برقرار است.

مساله طولانی ترین مسیرها

- طولانی ترین مسیر از V_1 به V_4 مسیر $[V_1, V_3, V_2, V_4]$ می باشد.
- زیر مسیر $[V_1, V_3]$ بهینه نیست.



ضرب زنجیره ای ماتریس ها

- برای ضرب نمودن یک ماتریس $i * j$ در یک ماتریس $j * k$ تعداد ضرب های ابتدایی برابر با $i * j * k$ می باشد.
- ضرب ماتریس ها دارای خاصیت شرکت پذیری می باشد، بدین معنا که ترتیب های متفاوتی برای ضرب چند ماتریس در یکدیگر وجود دارد که هر یک منجر به تعداد متفاوتی از ضرب های ابتدایی می شود.
- مثال:

$$\begin{array}{cccccccc} A & \times & B & \times & C & \times & D \\ 20\times2 & & 2\times30 & & 30\times12 & & 12\times8 \end{array}$$

درخت های جستجوی دودویی بهینه

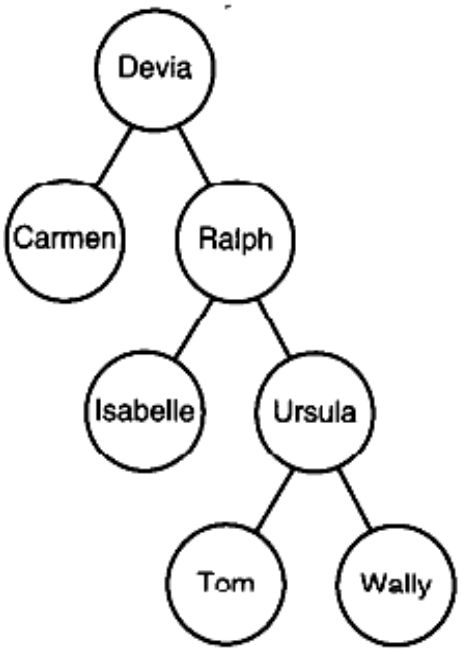
درخت جستجوی دودویی یک درخت دودویی از عناصر (کلید ها) است، که از یک مجموعه مرتب حاصل می شود، به طوری که

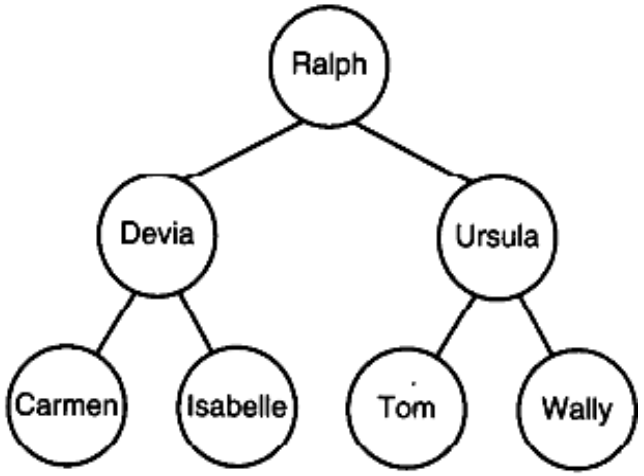
۱. هر گره دارای یک کلید می باشد.

۲. کلیدهای واقع در زیر درخت چپ یک گره، کوچکتر یا مساوی کلید آن گره می باشند.

۳. کلیدهای واقع در زیر درخت راست یک گره، بزرگتر یا مساوی کلید آن گره می باشند.

مثال ها


Devia
Carmen
Ralph
Isabelle
Ursula
Tom
Wally


Ralph
Devia
Ursula
Carmen
Isabelle
Tom
Wally

درخت متوازن

- عمق (سطح) یک گره: تعداد لبه های موجود در مسیر منحصر بفرد از ریشه به آن گره.
- عمق یک درخت: حداکثر عمق تمامی گره ها در آن درخت.
- درخت دودویی متوازن: اگر عمق دو زیر درخت از هر گره بیش از یک واحد اختلاف نداشته باشند.
- درخت جستجوی دودویی بهینه: زمان متوسط برای مکان یابی یک کلید کمینه است.

متوسط زمان جستجو

- زمان جستجو: تعداد مقایسه های انجام شده برای مکان یابی یک کلید
- زمان جستجو برای key برابر است با:

$$depth(key) + 1$$

- متوسط زمان جستجو:

$$\sum_{i=1}^{n} c_i \, p_i$$

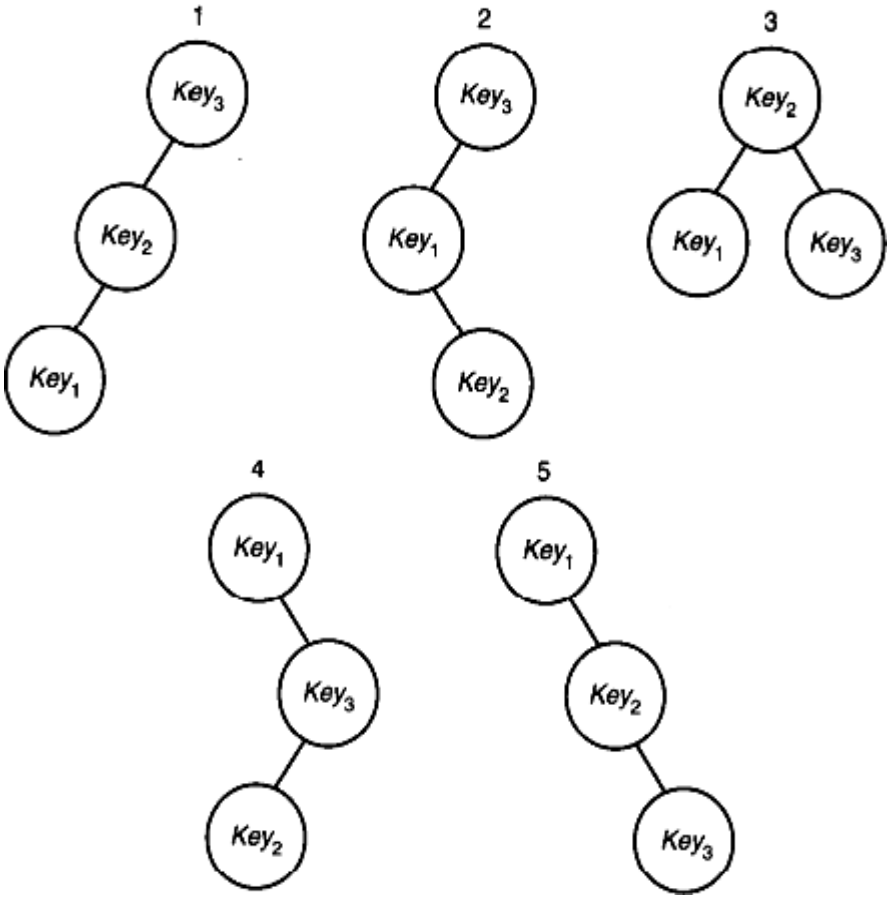
که در آن:

n تعداد کلید ها،

p_i احتمال آنکه key_i کلید مورد جستجو باشد،

c_i تعداد مقایسه های مورد نیاز برای یافتن key_i می باشد.

مثال



- $P_1 = 0.7$
- $P_2 = 0.2$
- $P_3 = 0.1$

1. $3(0.7) + 2(0.2) + 1(0.1) = 2.6$
2. $2(0.7) + 3(0.2) + 1(0.1) = 2.1$
3. $2(0.7) + 1(0.2) + 2(0.1) = 1.8$
4. $1(0.7) + 3(0.2) + 2(0.1) = 1.5$
5. $1(0.7) + 2(0.2) + 3(0.1) = 1.4$

مساله فروشنده دوره گرد (*TSP*)

- تور(دور هامیلتونی): مسیری از یک راس به خودش که از هر یک از رئوس دیگر دقیقا یک بار عبور می کند.
- تور بهینه: مسیری با مشخصات بالا با طول حداقل.
- الگوریتم *Brute force* از مرتبه فاکتوریل می باشد:

$$(n\text{-}1)(n\text{-}2)\cdots 1 = (n\text{-}1)!$$

- اصل بهینگی برقرار است(؟).

یک مثال

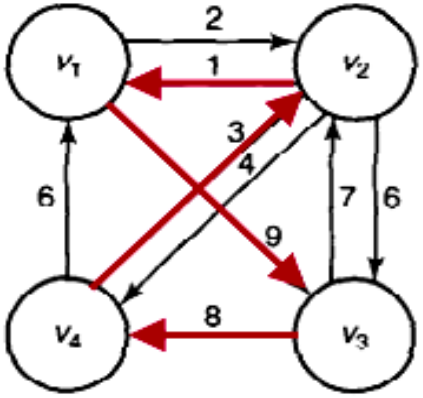


Figure 3.16 ● The optimal tour is $[v_1, v_3, v_4, v_2, v_1]$.

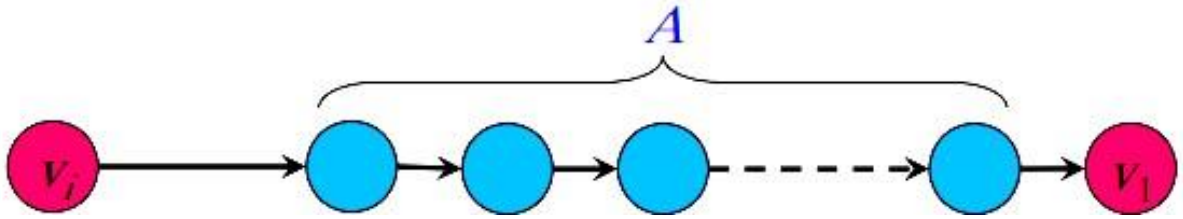
$$length[V_1, V_2, V_3, V_4, V_1] = 22$$

$$length[V_1, V_3, V_2, V_4, V_1] = 26$$

$$length[V_1, V_3, V_4, V_2, V_1] = 21$$

آماده سازی

- V = مجموعه تمام رئوس $V = \{v_1, v_2, \cdots, v_n\}$
- A = زیر مجموعه ای از V
- $D[v_i][A]$ = طول کوتاهترین مسیر از v_i به v_1 که از تمام رئوس مجموعه A دقیقا یک بار عبور می کند.



مثال

- محاسبه $D[v_2][A]$ برای گراف زیر

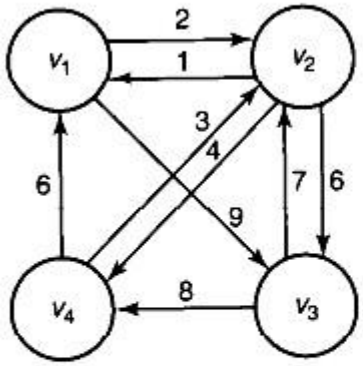


Figure 3.16 ● The optimal tour is $[v_1, v_3, v_4, v_2, v_1]$.

اگر $A=\{v_3\}$، آنگاه:

$$D[v_2][A] = length[v_2, v_3, v_1] = \infty$$

اگر $A=\{v_3, v_4\}$، آنگاه:

$$D[v_2][A] = \text{minimum}(\ length[v_2, v_3, v_4, v_1], length[v_2, v_4, v_3, v_1]\ )$$

$$= \text{minimum}(20, \infty) = 20$$

الگوریتم

- طول یک تور بهینه =

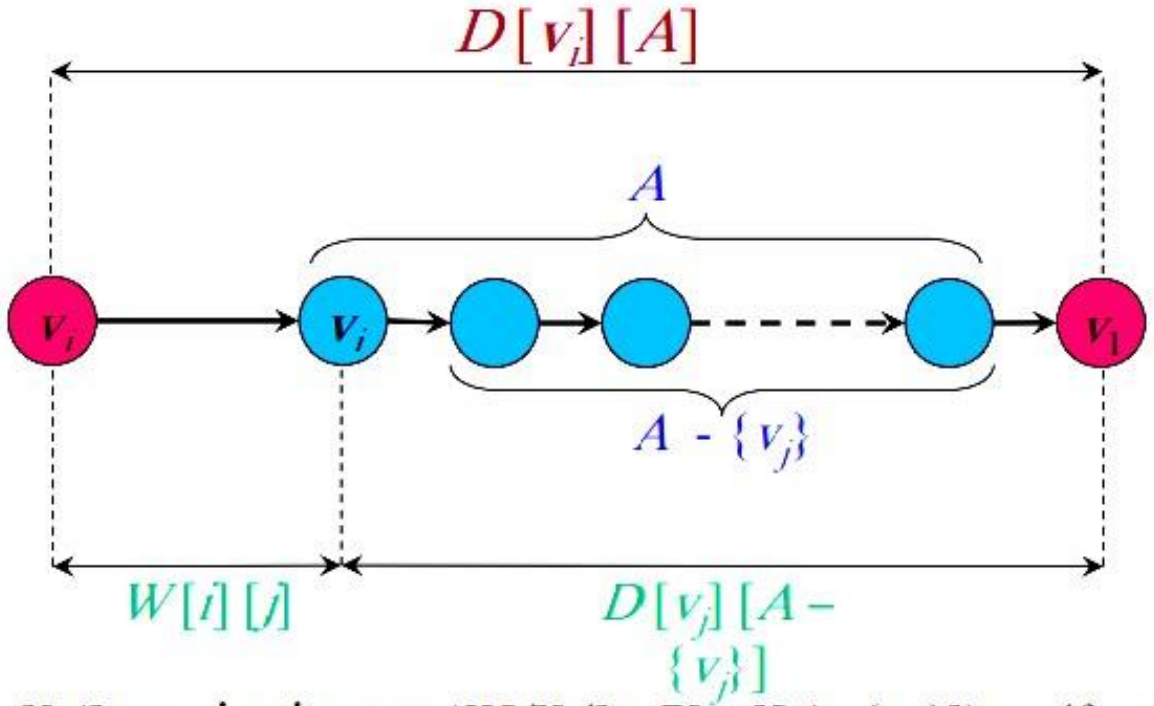
$$D[v_1][V-\{v_1\}] = \underset{2\le j\le n}{\text{minimum}}(W[1][j] + D[v_j][V-\{v_1, v_j\}])$$

- به طور کلی به ازای $i \neq 1$ و $v_i \notin A$

$$D[v_i][A] = \min_{j:v_j \in A} imum(W[i][j] + D[v_j][A-\{v_j\}]) \quad if \quad A \neq \phi$$

$$D[v_i][\phi] = W[i][1]$$

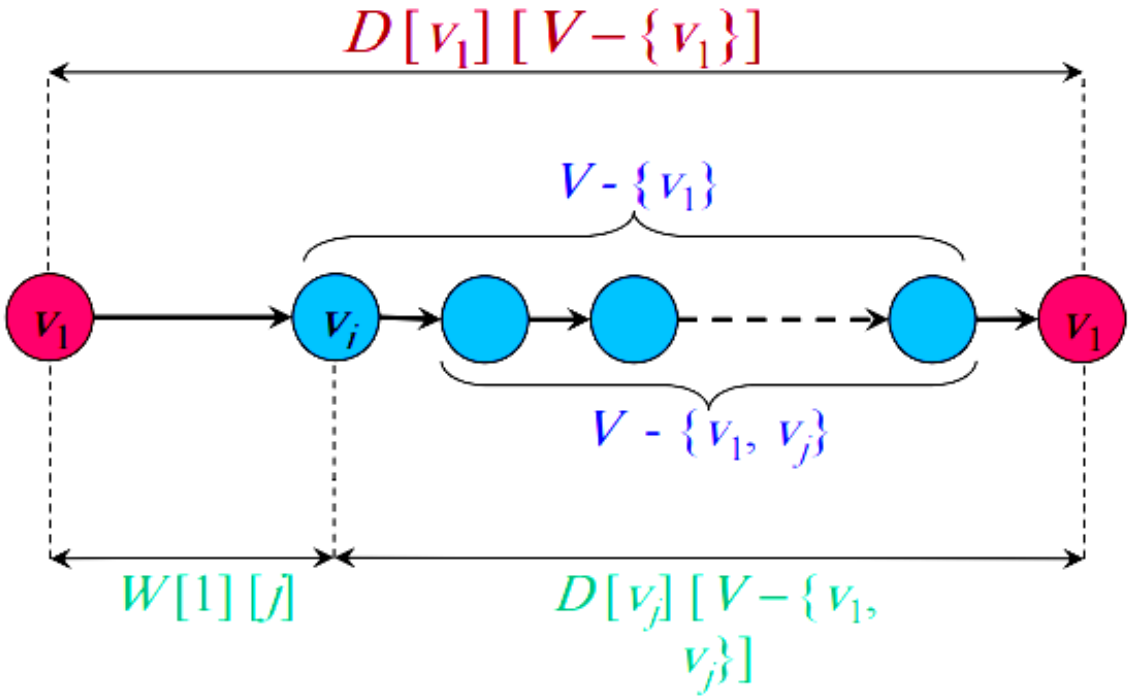
محاسبه طول مسیر بهینه ($i \neq 1$ و $v_i \notin A$)



$$D[v_i][A] = \min_{j:v_j \in A} imum(W[i][j] + D[v_j][A - \{v_j\}]) \quad if \quad A \neq \phi$$

$$D[v_i][\phi] = W[i][1]$$

محاسبه طول تور بهینه



$$D[v_1][V-\{v_1\}] = \underset{2\le j\le n}{\text{minimum}}(W[1][j] + D[v_j][V-\{v_1, v_j\}])$$

مثال

- محاسبه تور بهینه برای گراف زیر

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

مثال فروشنده دوره گرد

	$\varnothing$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
1	-	-	-	-	-	-	-	
2	1	-			-	-		-
3	∞		-		-		-	-
4	6			-		-	-	-

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

$$D[v_2][\phi] = 1$$

$$D[v_3][\phi] = \infty$$

$$D[v_4][\phi] = 6$$

مثال فروشنده دوره گرد

	∅	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
1	-	-	-	-	-	-	-	
2	1	-			-	-		-
3	∞	8	-		-		-	-
4	6	4		-		-	-	-

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

$$D[v_3][\{v_2\}] = \underset{j:v_j \in \{v_2\}}{\text{minimum}}(W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$

$$= W[3][2] + D[v_2][\phi] = 7 + 1 = 8$$

$$D[v_4][\{v_2\}] = \underset{j:v_j \in \{v_2\}}{\text{minimum}}(W[4][j] + D[v_j][\{v_2\} - \{v_j\}])$$

$$= W[4][2] + D[v_2][\phi] = 3 + 1 = 4$$

مثال فروشنده دوره گرد

	$\varnothing$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
1	-	-	-	-	-	-	-	
2	1	-	∞		-	-		-
3	∞	8	-		-		-	-
4	6	4	∞	-		-	-	-

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

$$D[v_2][\{v_3\}] = \underset{j, v_j \in \{v_3\}}{\text{minimum}}(W[2][j] + D[v_j][\{v_3\} - \{v_j\}])$$

$$= W[2][3] + D[v_3][\phi] = 6 + \infty = \infty$$

$$D[v_4][\{v_3\}] = \underset{j, v_j \in \{v_3\}}{\text{minimum}}(W[4][j] + D[v_j][\{v_3\} - \{v_j\}])$$

$$= W[4][3] + D[v_3][\phi] = \infty + \infty = \infty$$

مثال فروشنده دوره گرد

	∅	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
1	-	-	-	-	-	-	-	
2	1	-	∞	10	-	-		-
3	∞	8	-	14	-		-	-
4	6	4	∞	-		-	-	-

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

$$D[v_2][\{v_4\}] = \underset{j, v_j \in \{v_4\}}{\text{minimum}}(W[2][j] + D[v_j][\{v_4\} - \{v_j\}])$$
$$= W[2][4] + D[v_4][\phi] = 4 + 6 = 10$$

$$D[v_3][\{v_4\}] = \underset{j, v_j \in \{v_4\}}{\text{minimum}}(W[3][j] + D[v_j][\{v_4\} - \{v_j\}])$$
$$= W[3][4] + D[v_4][\phi] = 8 + 6 = 14$$

مثال فروشنده دوره گرد

	∅	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
1	-	-	-	-	-	-	-	
2	1	-	∞	10	-	-		-
3	∞	8	-	14	-		-	-
4	6	4	∞	-	∞	-	-	-

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

$$D[v_4][\{v_2, v_3\}] = \underset{j: v_j \in \{v_2, v_3\}}{\text{minimum}}(W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}])$$

$$= \text{minimum}(W[4][2] + D[v_2][\{v_3\}], W[4][3] + D[v_3][\{v_2\}])$$

$$= \text{minimum}(3 + \infty, \infty + 8) = \infty$$

مثال فروشنده دوره گرد

	$\varnothing$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
1	-	-	-	-	-	-	-	
2	1	-	∞	10	-	-		-
3	∞	8	-	14	-	12 4	-	-
4	6	4	∞	-	∞	-	-	-

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

$$D[v_3][\{v_2, v_4\}] = \underset{j: v_j \in \{v_2, v_4\}}{\text{minimum}}(W[3][j] + D[v_j][\{v_2, v_4\} - \{v_j\}])$$

$$= \text{minimum}(W[3][2] + D[v_2][\{v_4\}], W[3][4] + D[v_4][\{v_2\}])$$

$$= \text{minimum}(7+10, 8+4) = 12$$

مثال فروشنده دوره گرد

	$\varnothing$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
1	-	-	-	-	-	-	-	
2	1	-	∞	10	-	-	20 3	-
3	∞	8	-	14	-	12 4	-	-
4	6	4	∞	-	∞	-	-	-

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

$$D[v_2][\{v_3, v_4\}] = \underset{j: v_j \in \{v_3, v_4\}}{\text{minimum}}(W[2][j] + D[v_j][\{v_3, v_4\} - \{v_j\}])$$

$$= \text{minimum}(W[2][3] + D[v_3][\{v_4\}], W[2][4] + D[v_4][\{v_3\}])$$

$$= \text{minimum}(6 + 14, 4 + \infty) = 20$$

مثال فروشنده دوره گرد

	$\varnothing$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
1	-	-	-	-	-	-	-	21 (3)
2	1	-	∞	10	-	-	20 (3)	-
3	∞	8	-	14	-	12 (4)	-	-
4	6	4	∞	-	∞	-	-	-

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

$$D[v_1][\{v_2, v_3, v_4\}] = \underset{j\, v_j \in \{v_2, v_3, v_4\}}{\text{minimum}}(W[1][j] + D[v_j][\{v_2, v_3, v_4\} - \{v_j\}])$$

$$= \text{minimum}(W[1][2] + D[v_2][\{v_3, v_4\}],$$
$$W[1][3] + D[v_3][\{v_2, v_4\}],$$
$$W[1][4] + D[v_4][\{v_2, v_3\}])$$
$$= \text{minimum}(2+20, 9+12, \infty+\infty) = 21$$

برنامه نویسی پویا

```
void travel(int n,
            const number W[][],
            index P[][],
            number& minlength)
{
    index i, j, k;
    number D[1..n][subset of V − {v1}];
    for (i = 2; i <= n, i++)
      D[i][∅] = W[i][1];
    for (k = 1; k <= n − 2; k++)
      for (all subsets A ⊆ V - {v1} containing k vertices)
        for (i such that i ≠ 1 and vi is not in A){
          D[i][A] = minimum( W[i][j] + D[j][A − {vj}]);
                    j : vj ∈ A
          P[i][A] = value of j that gave the minimum;
        }
    D[1][V − {v1}] = minimum ( W[1][j] + D[j][V − {v1, vj}]);
                     2 ≤ j ≤ n
    P[1][V − {v1}] = value of j that gave the minimum;
    minlength = D[1][V − {v1}];
}
```

- بازاء هر $n \geq 1$

$$\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1}$$

$$k\binom{n}{k} = n\binom{n-1}{k-1}$$

$$\sum_{k=1}^{n} k\binom{n}{k} = \sum_{k=1}^{n} n\binom{n-1}{k-1} = n\sum_{k=0}^{n-1}\binom{n-1}{k} = n2^{n-1}$$

پیچیدگی زمانی برای همه حالات

- عمل اصلی: دستورالعمل اجرا شده برای هر مقدار از V_j
- اندازه ورودی: n، تعداد رئوس در گراف
- پیچیدگی زمانی:

$$T(n) = \sum_{k=1}^{n-2} (n-1-k)k\binom{n-1}{k}$$

$$(n-1-k)\binom{n-1}{k} = (n-1)\binom{n-2}{k}$$

$$\Rightarrow T(n) = (n-1)\sum_{k=1}^{n-2} k\binom{n-2}{k}$$

$$T(n) = (n-1)(n-2)2^{n-3} \in \Theta(n^2 2^n)$$

- پیچیدگی حافظه:

$$M(n) = 2 \times n2^{n-1} = n2^n \in \Theta(n2^n)$$

آیا چنین الگوریتمی می تواند مفید باشد؟

- رالف و نانسی در حال رقابت برای تصاحب یک موقعیت شغلی
- مسابقه برای تصاحب موقعیت شغلی در شرکت:
  - هرکسی که سریعتر سفر به ۲۰ شهر مشخص را انجام دهد و به شرکت برگردد برنده است. بین هر دو شهر یک جاده وجود دارد.
  - رالف : الگوریتم غیر هوشمند

$(20-1)!\ \mu s = 19!\ \mu s = 3857$ سال

  - نانسی: الگوریتم برنامه نویسی پویا

$(20-1)(20-2)2^{20-3}\ \mu s = 45$ ثانیه

$20 * 2^{20} = 20,971,520$ عنصر آرایه

اگر تعداد شهر ها 60 باشد: الگوریتم برنامه نویسی پویا نیز سال ها وقت لازم دارد.

فصل چهارم

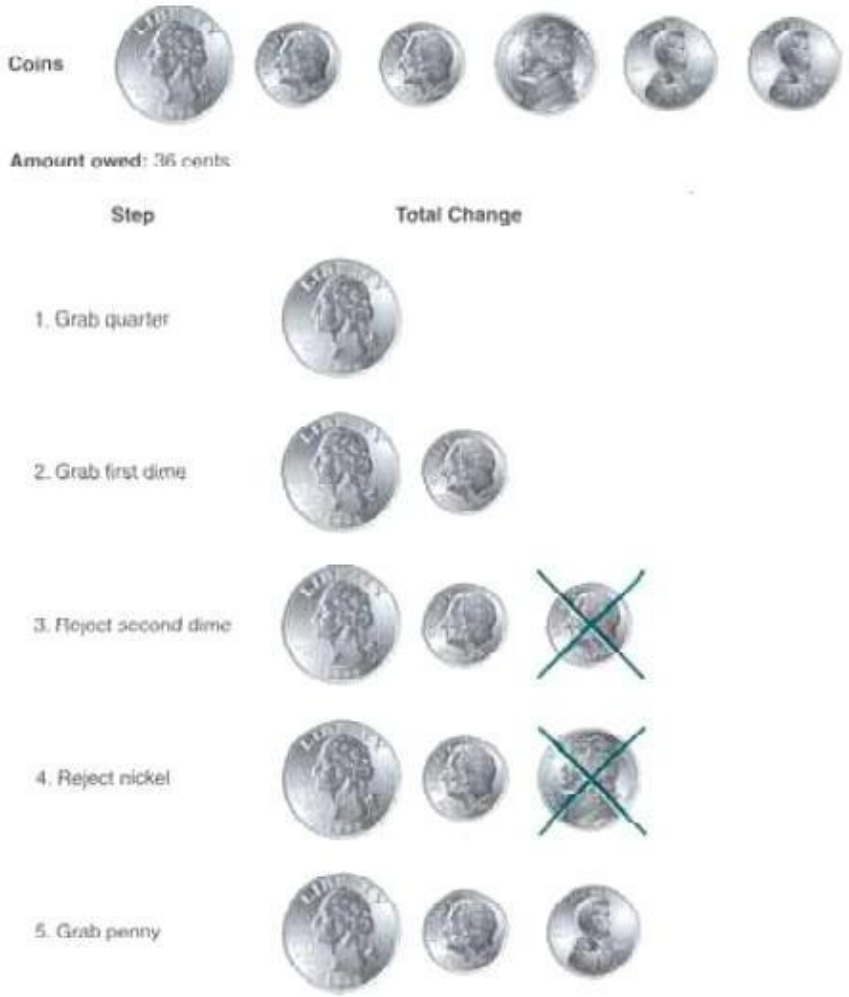
روش حریصانه

روش حریصانه

- رهیافت اسکروج
- عناصر داده ای را به ترتیب انتخاب کن، هر بار «بهترین» انتخاب را انجام بده، بدون توجه به انتخاب های قبلی و انتخاب هایی که در آینده انجام خواهند گرفت.
- اغلب در مسائل بهینه سازی بکار می رود. (مانند برنامه نویسی پویا)
- باید مشخص شود که با دنباله ای از راه حل های بهینه محلی، یک راه حل بهینه سراسری بدست می آید.

روش حریصانه

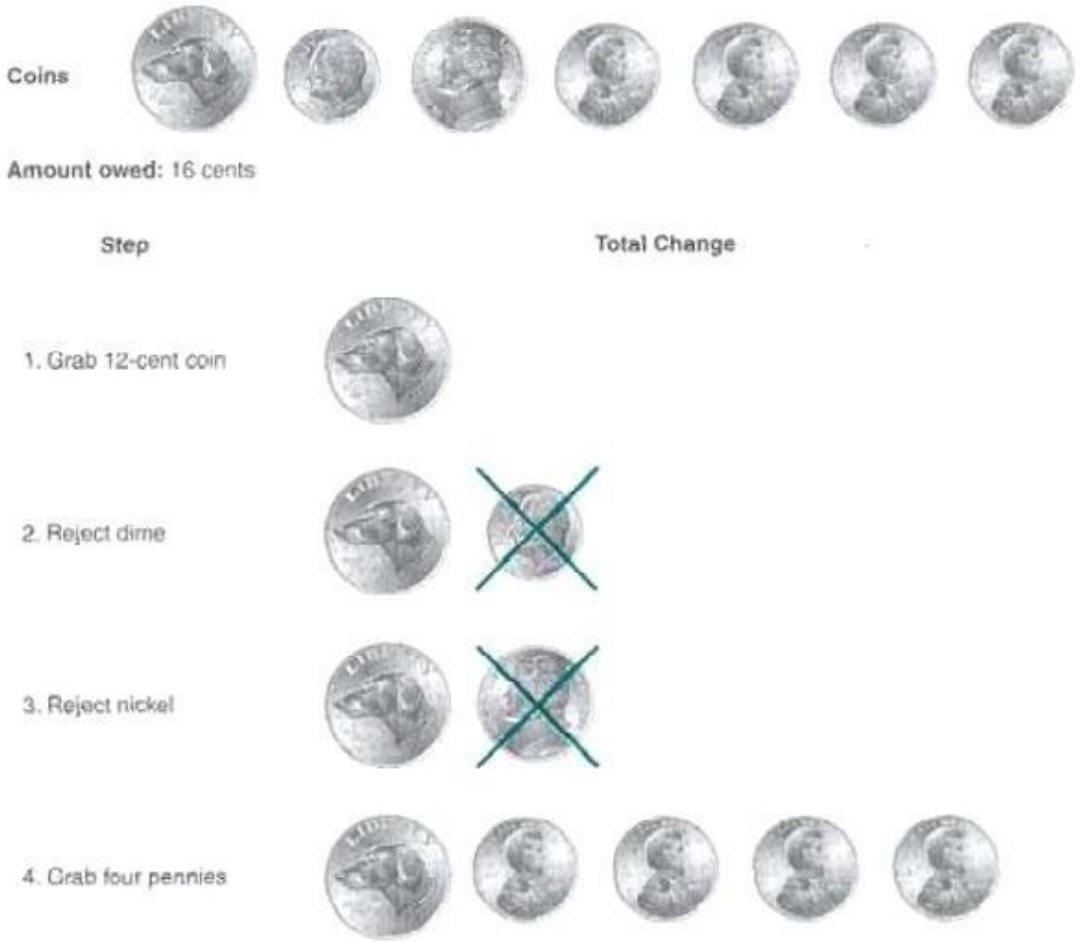
مثال: باقیمانده پول



الگوریتم

```
while (there are more coins and the instance is not solved){
  grab the largest remaining coin;                    // selection procedure
  If (adding the coin makes the change exceed the
                  amount owed)                        // feasibility check
     reject the coin;
  else
     add the coin to the change;
  If (the total value of the change equals the
                  amount owed)                        // solution check
     the instance is solved;
}
```

روش حریصانه



شکست
رهیافت
حریصانه

اضافه کردن یک عنصر به مجموعه

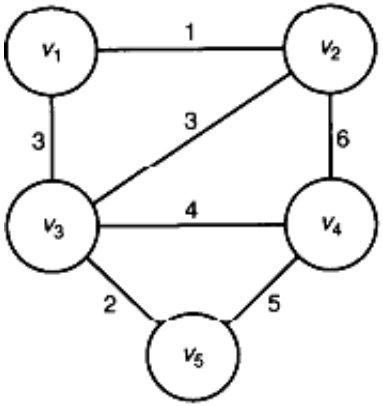
- **روال انتخاب**، عنصر بعدی را که باید به مجموعه اضافه شود، انتخاب می کند. انتخاب براساس یک ملاک حریصانه انجام می شود که برخی شرایط بهینه محلی را در زمان انتخاب برآورده می سازد.
- **بررسی امکان سنجی**، تعیین می کند که آیا مجموعه جدید برای رسیدن به حل نمونه عملی است یا خیر.
- **بررسی راه حل**، بررسی می کند که آیا مجموعه جدید، راه حلی برای نمونه مساله می باشد یا خیر.

درخت های پوشای کمینه (*MST*)

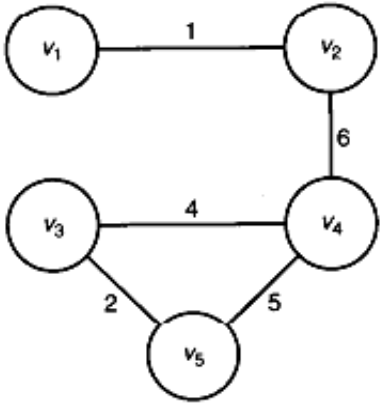
- برخی اصطلاحات:
  - گراف بدون جهت
  - مسیر
  - گراف همبند
  - دور ساده
  - گراف بدون دور
  - درخت (درخت آزاد)
  - درخت ریشه دار

روش حریصانه

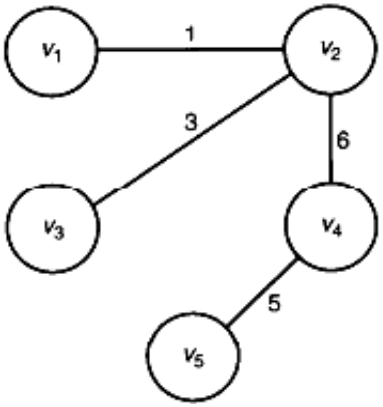
(a) A connected, weighted, undirected graph G.



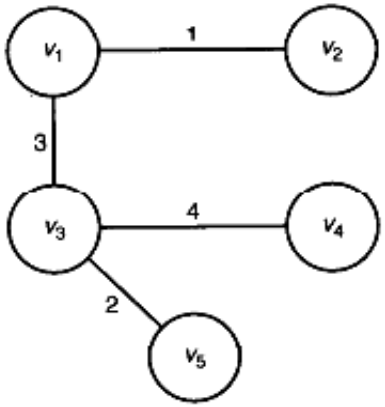
(b) If (v_4, v_5) were removed from this subgraph, the graph would remain connected.



(c) A spanning tree for G.



(d) A minimum spanning tree for G.



درخت پوشای کمینه

- درخت پوشا
- درخت پوشای کمینه
- تعریف رسمی گراف بدون جهت

تعریف:

یک **گراف بدون جهت** G شامل یک مجموعه متناهی و غیر تهی V می باشد که عناصر آن را رئوس گراف G می نامیم، به همراه مجموعه E که شامل مجموعه ای از زوج رئوس (یال) در V می باشد.

$$G=(V, E)$$

یافتن درخت پوشای کمینه

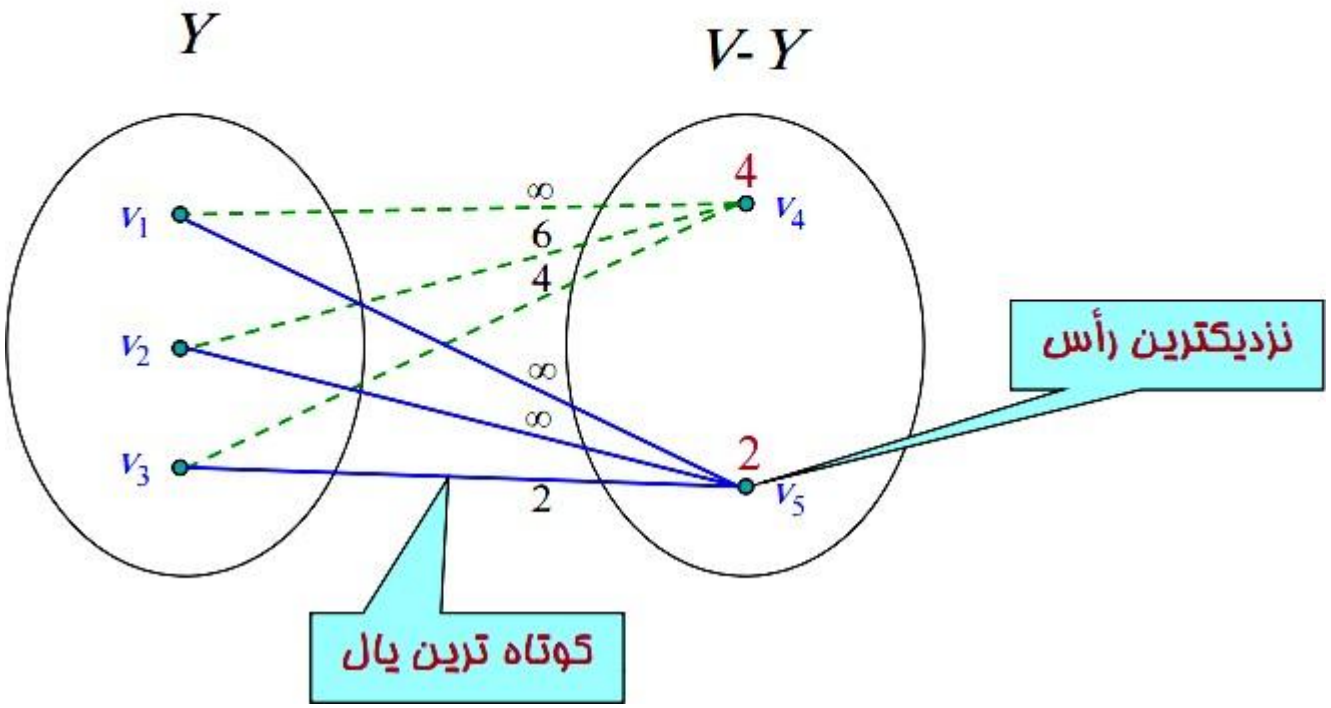
- برای یافتن $T=(V, F)$ کمینه برای $G=(V, E)$

```
F = ∅                                              // Initialize set of
                                                   // edges to empty.
while (the instance is not solved){

  select an edge according to some locally
  optimal consideration;                           // selection procedure

  if (adding the edge to F does not create a cycle)
  add it;                                          // feasibility check

  if (T = (V,F) is a spanning tree)                // solution check
  the instance is solved;
}
```

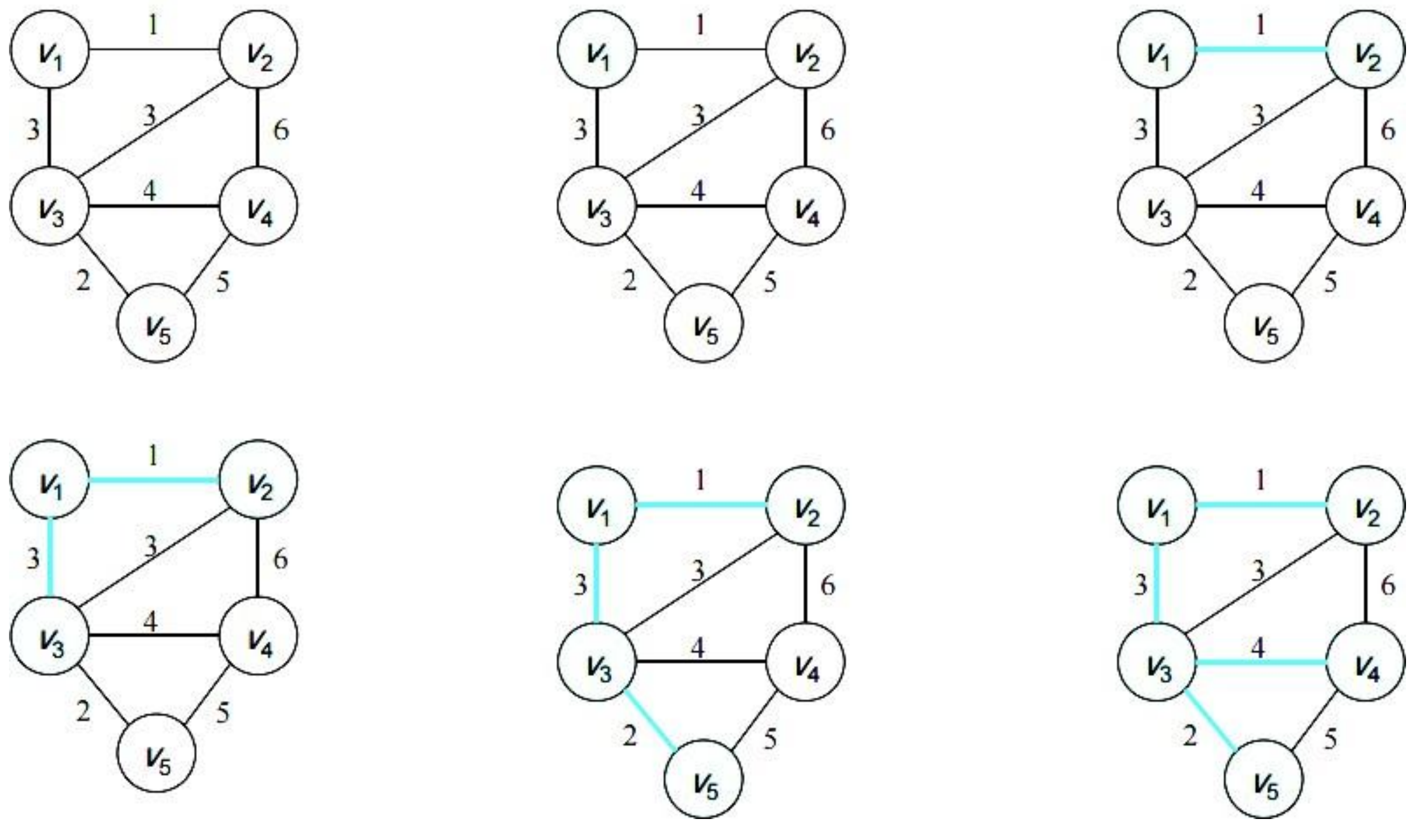
الگوریتم پریم

```
F = ∅;                                  // Initialize set of edges
                                        // to empty.
Y = {v1};                               // Initialize set of vertices to
                                        // contain only the first one.

while (the instance is not solved){

  select a vertex in V − Y that is      // selection procedure and
  nearest to Y;                         // feasibility check

  add the vertex to Y;
  add the edge to F;

  if (Y == V)                           // solution check
      the instance is solved;
}
```

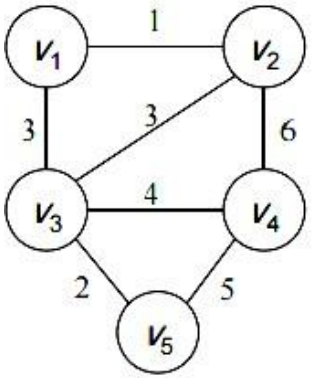
یافتن نزدیک ترین رأس

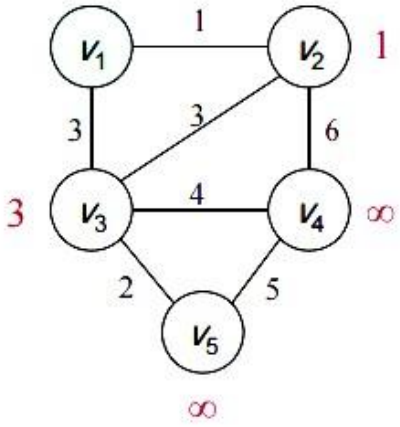


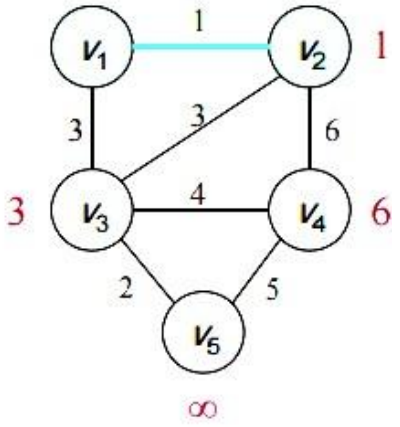
یک مثال

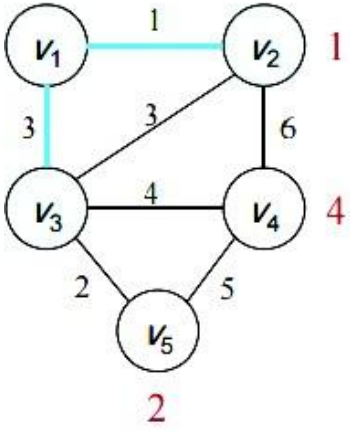

v_1 v_2 v_3 v_4 v_5 1 3 3 6 4 2 5
v_1 v_2 v_3 v_4 v_5 1 3 3 6 4 2 5
v_1 v_2 v_3 v_4 v_5 1 3 3 6 4 2 5
v_1 v_2 v_3 v_4 v_5 1 3 3 6 4 2 5
v_1 v_2 v_3 v_4 v_5 1 3 3 6 4 2 5
v_1 v_2 v_3 v_4 v_5 1 3 3 6 4 2 5

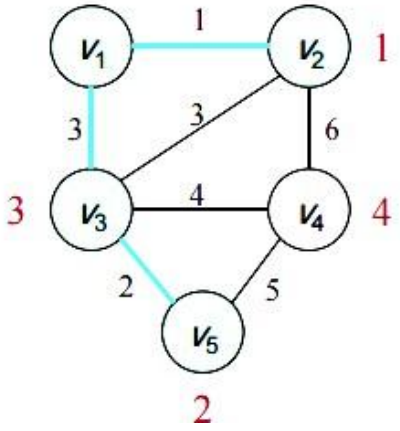
یک مثال

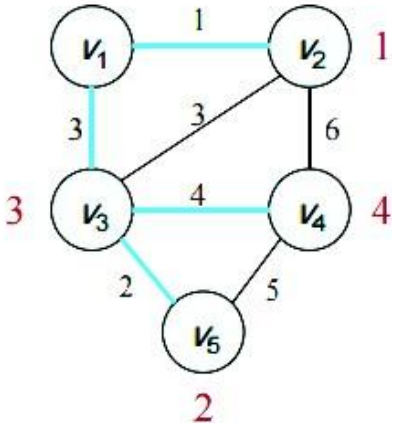

1
3
3
6
4
2
5
v_1
v_2
v_3
v_4
v_5


1
3
3
6
4
2
5
v_1
v_2
v_3
v_4
v_5
1
3
∞
∞


1
3
3
6
4
2
5
v_1
v_2
v_3
v_4
v_5
1
3
6
∞


1
3
3
6
4
2
5
v_1
v_2
v_3
v_4
v_5
1
4
2


1
3
3
6
4
2
5
v_1
v_2
v_3
v_4
v_5
1
3
4
2


1
3
3
6
4
2
5
v_1
v_2
v_3
v_4
v_5
1
3
4
2

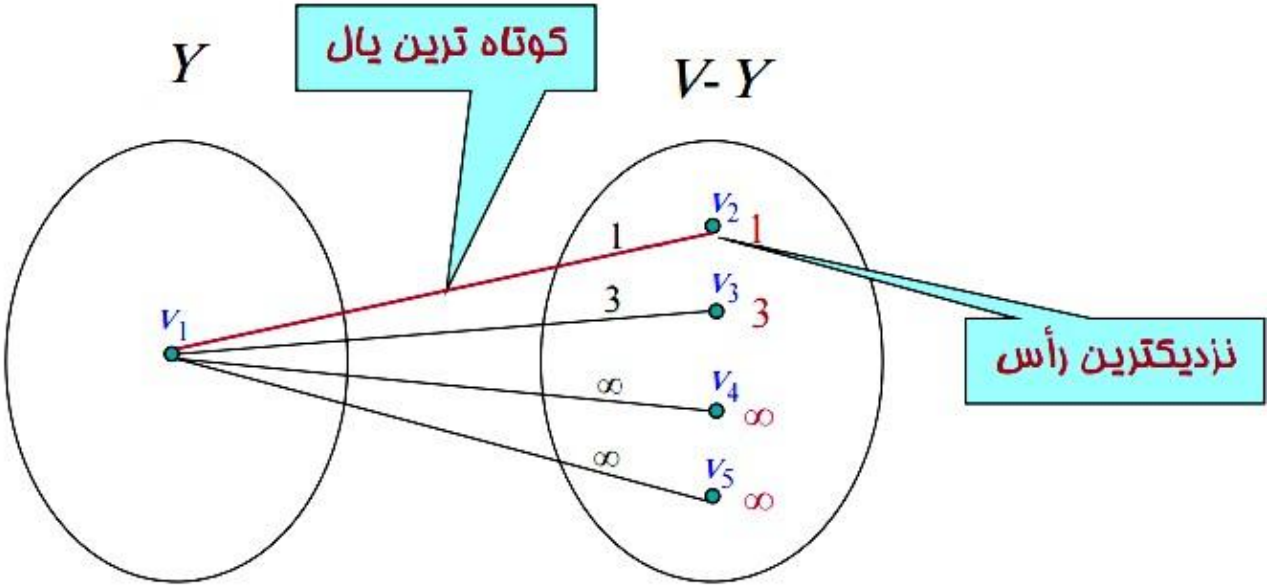
ماتریس مجاورتی

$$W[i][j] = \begin{cases} \text{weight on edge} & \text{if there is an edge between } v_i \text{ and } v_j \\ \infty & \text{if there is no edge between } v_i \text{ and } v_j \\ 0 & \text{if } i = j. \end{cases}$$

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

مثال: اجرای الگوریتم پریم

$F = \emptyset$

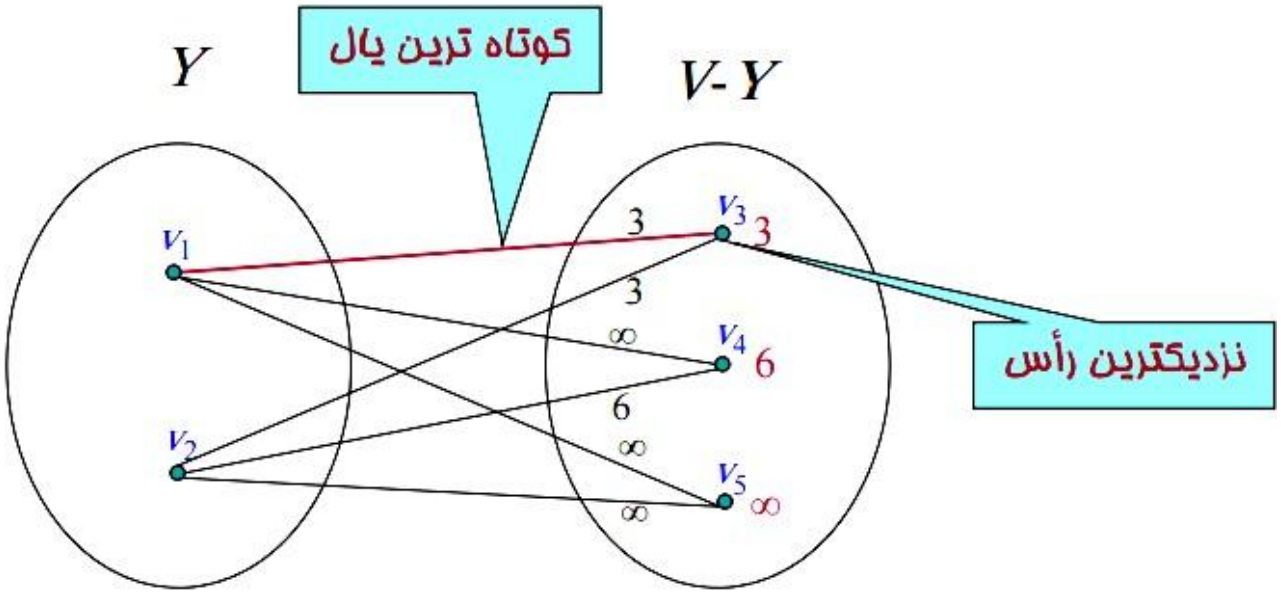


	2	3	4	5
nearest	1	1	1	1

	2	3	4	5
distance	1	3	∞	∞

مثال: اجرای الگوریتم پریم

$F = \{\{v_1, v_2\}\}$

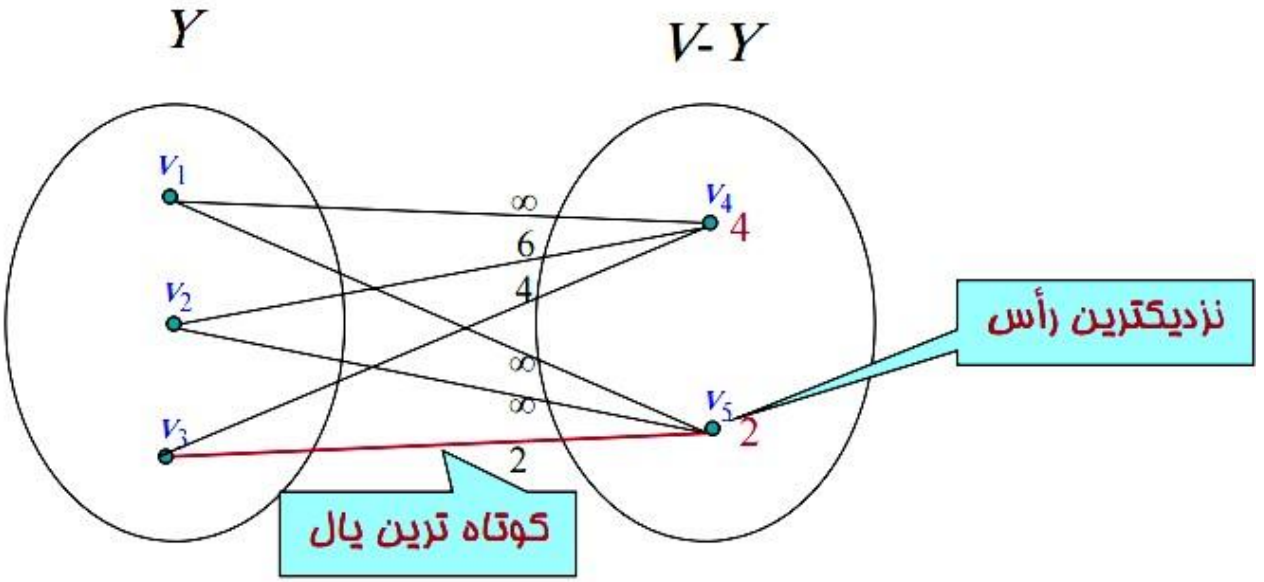


	2	3	4	5
nearest	1	1	2	1

	2	3	4	5
distance	-1	3	6	∞

مثال: اجرای الگوریتم پریم

$F = \{\{v_1, v_2\}, \{v_1, v_3\}\}$

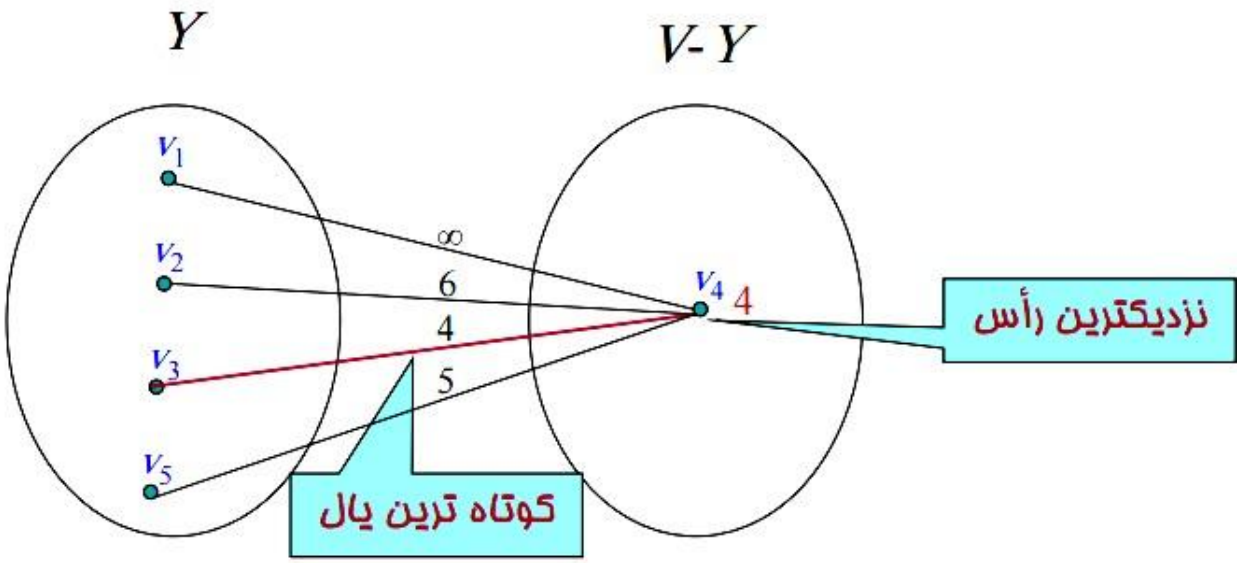


	2	3	4	5
nearest	1	1	3	3

	2	3	4	5
distance	-1	-1	4	2

مثال: اجرای الگوریتم پریم

$F = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_5\}\}$

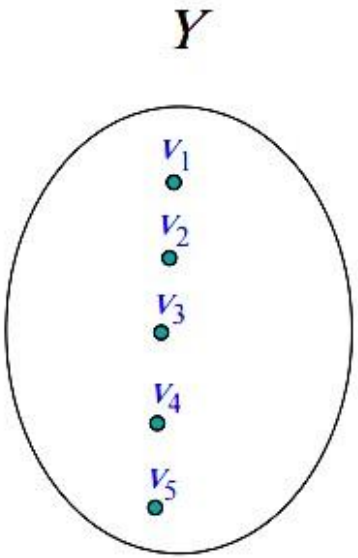


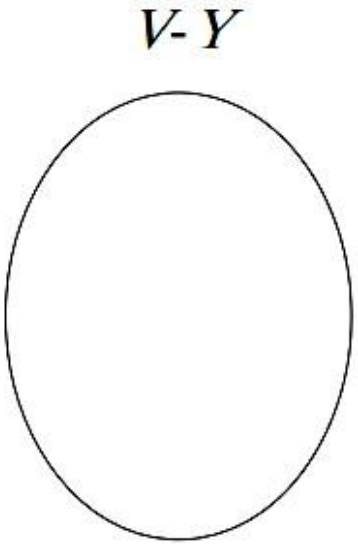
	2	3	4	5
nearest	1	1	3	3

	2	3	4	5
distance	-1	-1	4	-1

مثال: اجرای الگوریتم پریم

$$F = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_5\}, \{v_3, v_4\}\}$$





	2	3	4	5
nearest	1	1	3	3

	2	3	4	5
distance	-1	-1	-1	-1

پیچیدگی زمانی برای همه حالات

- عمل اصلی: در حلقه *repeat* دو حلقه وجود دارد که هر یک از آنها $(n-1)$ بار تکرار می شود، اجرای دستورات داخل هریک از آنها را می توان به عنوان یک بار اجرای عمل اصلی در نظر گرفت.
- اندازه ورودی: n، تعداد رئوس
- پیچیدگی زمانی:

چون حلقه *repeat* به تعداد $(n-1)$ بار تکرار می شود، پیچیدگی زمانی برابر است با:

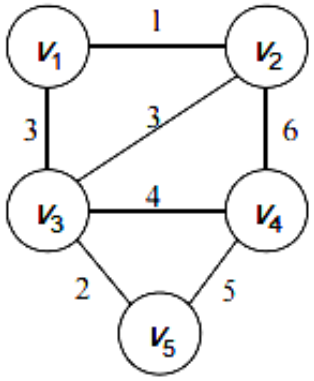
$$T(n) = 2(n-1)(n-1) = \Theta(n^2)$$

الگوریتم کروسکال

```
F = ∅;                                          // Initialize set of
                                                // edges to empty.
create disjoint subsets of V, one for each
vertex and containing only that vertex;

sort the edges in E in nondecreasing order;


while (the instance is not solved){

  select next cdge;                             // selection procedure

  if (the edge connects two vertices in         // feasibility check
                      disjoint subscts){
      merge the subsets;
      add the edge to F;
  }
  if (all the subsets are merged)               // solution check
     the instance is solved;
}
```

یک مثال

۱. مرتب سازی یال ها بر حسب طول



(v_1, v_2) 1

(v_3, v_5) 2

(v_1, v_3) 3

(v_2, v_3) 3

(v_3, v_4) 4

(v_4, v_5) 5

(v_2, v_4) 6

روش حریصانه

۲. ایجاد مجموعه های مجزا

(v_1, v_2) 1

(v_3, v_5) 2

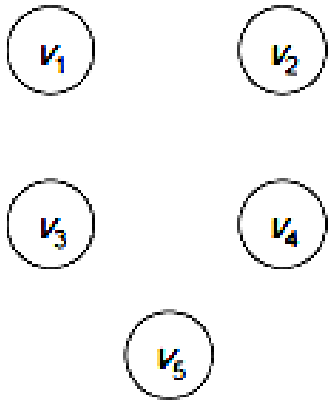
(v_1, v_3) 3

(v_2, v_3) 3

(v_3, v_4) 4

(v_4, v_5) 5

(v_2, v_4) 6



روش حریصانه

۳. انتخاب یال (v_1, v_2)

(v_1, v_2) 1

(v_3, v_5) 2

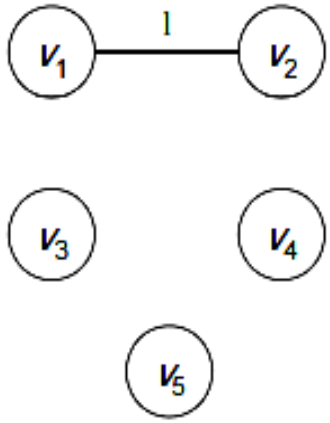
(v_1, v_3) 3

(v_2, v_3) 3

(v_3, v_4) 4

(v_4, v_5) 5

(v_2, v_4) 6



روش حریصانه

۴. انتخاب یال (v_3, v_5)

(v_1, v_2) 1

(v_3, v_5) 2

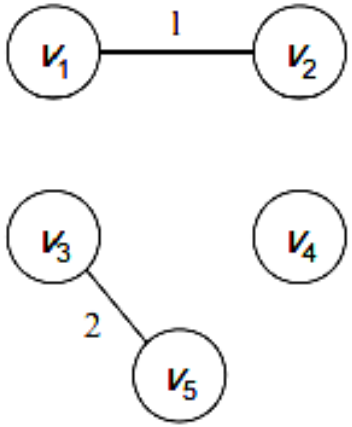
(v_1, v_3) 3

(v_2, v_3) 3

(v_3, v_4) 4

(v_4, v_5) 5

(v_2, v_4) 6



روش حریصانه

5. انتخاب یال (v_1, v_3)

(v_1, v_2) 1

(v_3, v_5) 2

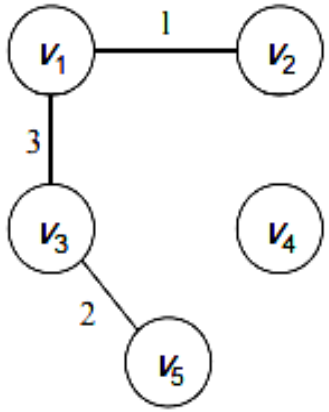
(v_1, v_3) 3

(v_2, v_3) 3

(v_3, v_4) 4

(v_4, v_5) 5

(v_2, v_4) 6



روش حریصانه

6. انتخاب یال (v_2, v_3)

(v_1, v_2) 1

(v_3, v_5) 2

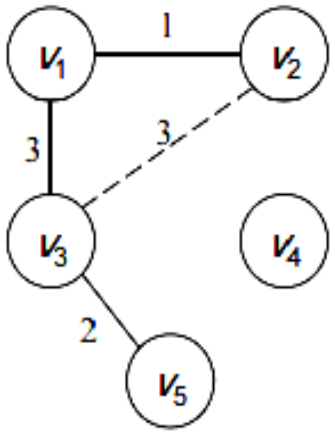
(v_1, v_3) 3

(v_2, v_3) 3

(v_3, v_4) 4

(v_4, v_5) 5

(v_2, v_4) 6



روش حریصانه

7. انتخاب یال (v_3, v_4)

(v_1, v_2) 1

(v_3, v_5) 2

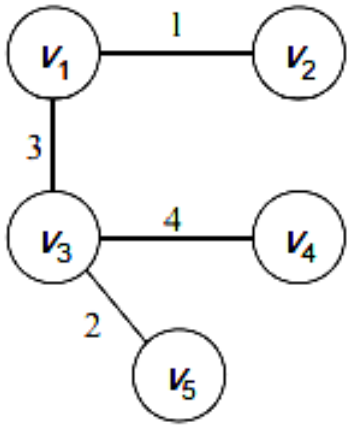
(v_1, v_3) 3

(v_2, v_3) 3

(v_3, v_4) 4

(v_4, v_5) 5

(v_2, v_4) 6



انواع داده ای و عملیات

- انواع داده ای:
  - index *i*,
  - set_pointer *p*, *q*,
- عملیات:
  - *initial*(*n*): *n* زیرمجموعه مجزا ایجاد می کند که هر یک حاوی دقیقا یکی از اندیس های ۱ تا *n* می باشد.
  - *p* = *find*(*i*): باعث می شود که *p* به مجموعه حاوی اندیس *i* اشاره کند.
  - *merge*(*p*, *q*): دو مجموعه ای را که *p* و *q* به آنها اشاره می کنند، ادغام می کند.
  - *equal*(*p*, *q*): اگر *p* و *q* هر دو به یک مجموعه اشاره کنند *true* برمی گرداند.

پیچیدگی زمانی در بدترین حالت

- عمل اصلی: یک دستورالعمل مقایسه
- اندازه ورودی: n، تعداد رئوس و m تعداد یال ها
- تحلیل:
  - زمان لازم برای مرتب سازی یال ها: $W(m) \in \Theta(m \lg m)$
  - زمان در حلقه while: $W(m) \in \Theta(m \lg m)$
  - زمان لازم برای ایجاد n مجموعه مجزا: $T(n) \in \Theta(n)$
  - در کل:

$$W(m, n) \in \Theta(m \lg m) = \Theta(n^2 lgn) \text{ (when } m = n(n\text{-}1)/2)$$

مقایسه الگوریتم های پریم و کروسکال

- الگوریتم پریم: $T(n) \in \Theta(n^2)$
- الگوریتم کروسکال: $W(m, n) \in \Theta(m \lg m) = \Theta(n^2 lgn)$
- در هر انتهای بازه زیر

$$n - 1 \leq m \leq n(n-1)/2$$

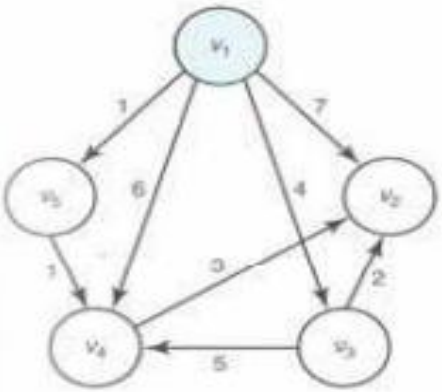
- اگر تعداد یال ها نزدیک به کرانه پایینی باشد، الگوریتم کروسکال $\Theta(nlgn)$ است و باید سریعتر از پریم باشد.
- اگر تعداد یال ها نزدیک به کرانه بالایی باشد، الگوریتم کروسکال $\Theta(n^2 lgn)$ است ، یعنی الگوریتم پریم باید سریعتر باشد.

الگوریتم دیکسترا برای کوتاه ترین مسیر تک مبدا

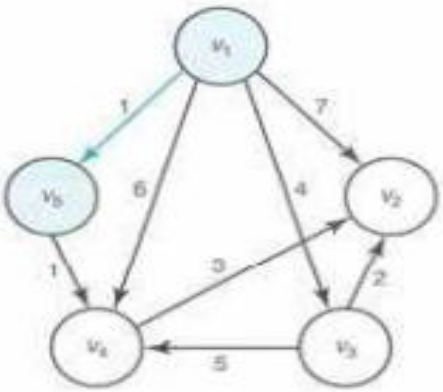
```
Y = {v1};
F = ∅;

while (the instance is not solved){

  select a vertex v from V − Y, that has a    // selection
  shortest path from v1, using only vertices  // procedure and
  in Y as intermediates;                      // feasibility check

  add the new vertex v to Y;
  add the edge (on the shortest path) that touches v to F;

  if (Y == V)
    the instance is solved;                   // solution check
}
```

روش حریصانه

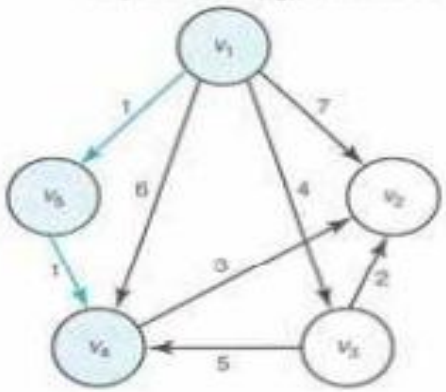
Compute shortest paths from v_1.



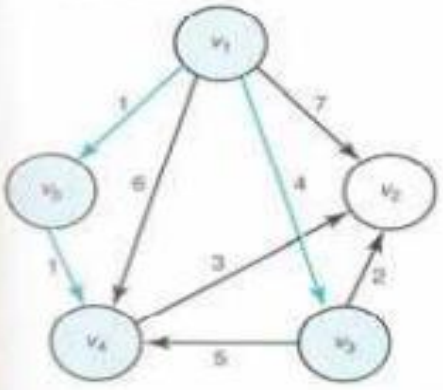
1. Vertex v_5 is selected because it is nearest to v_1.



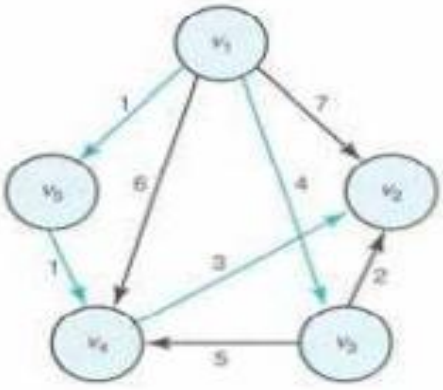
2. Vertex v_4 is selected because it has the shortest path from v_1 using only vertices in $\{v_5\}$ as intermediates.



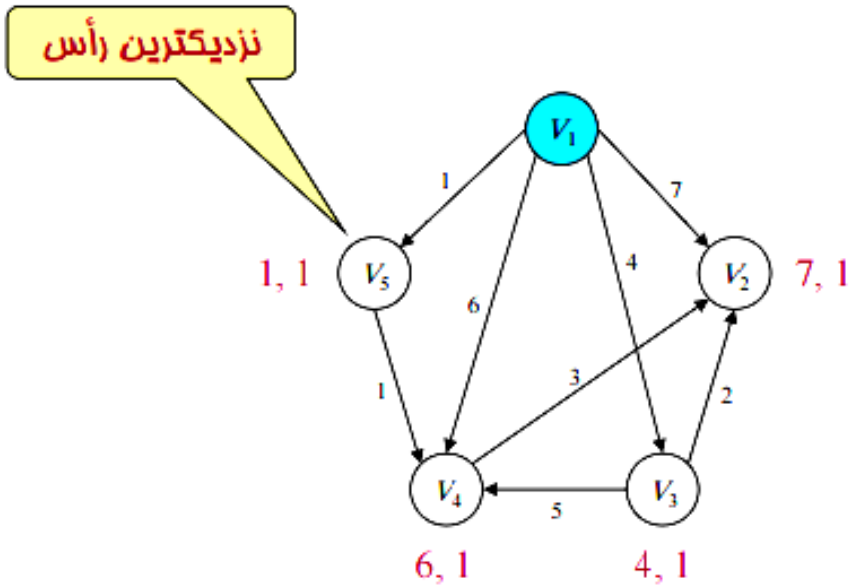
3. Vertex v_3 is selected because it has the shortest path from v_1 using only vertices in $\{v_4, v_5\}$ as intermediates.



4. The shortest path from v_1 to v_2 is $[v_1, v_5, v_4, v_2]$.



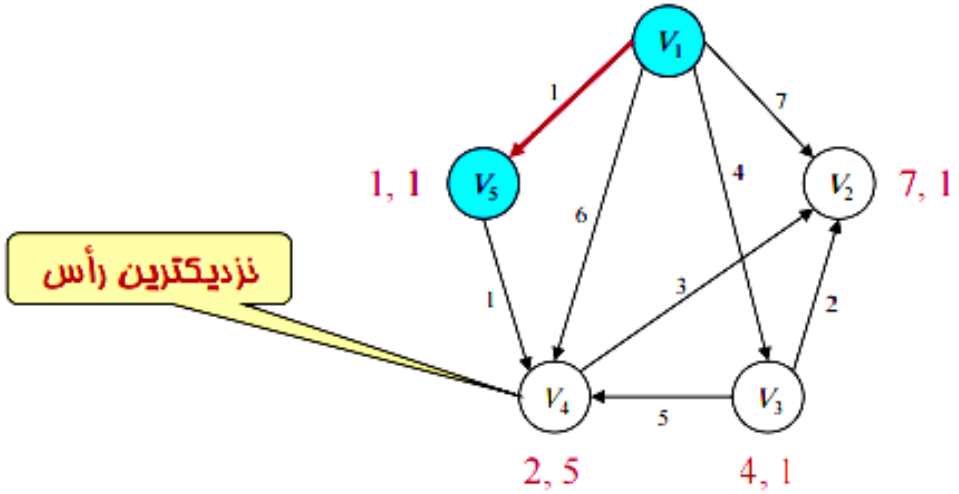
مثال: الگوریتم دیکسترا



	2	3	4	5
touch	1	1	1	1

	2	3	4	5
length	7	4	6	1

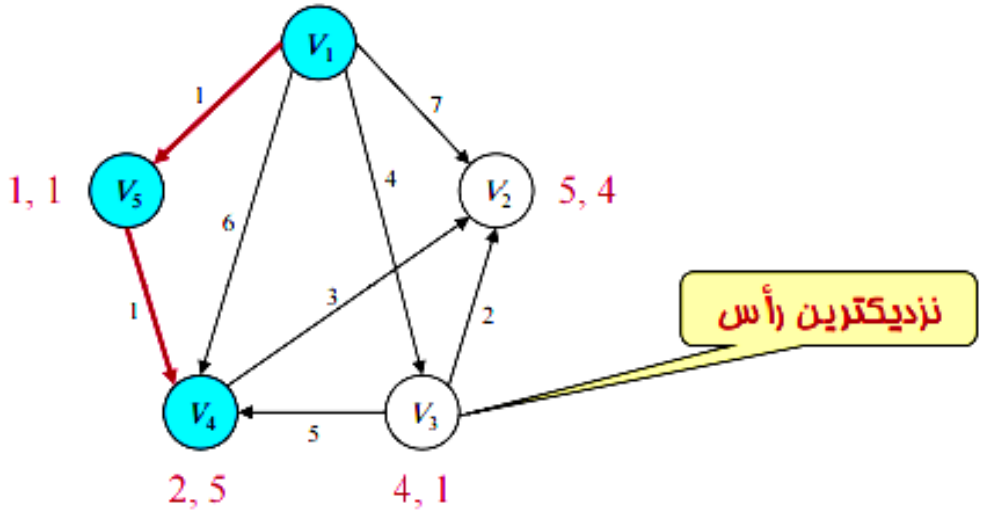
مثال: الگوریتم دیکسترا



	2	3	4	5
touch	1	1	5	1

	2	3	4	5
length	7	4	2	-1

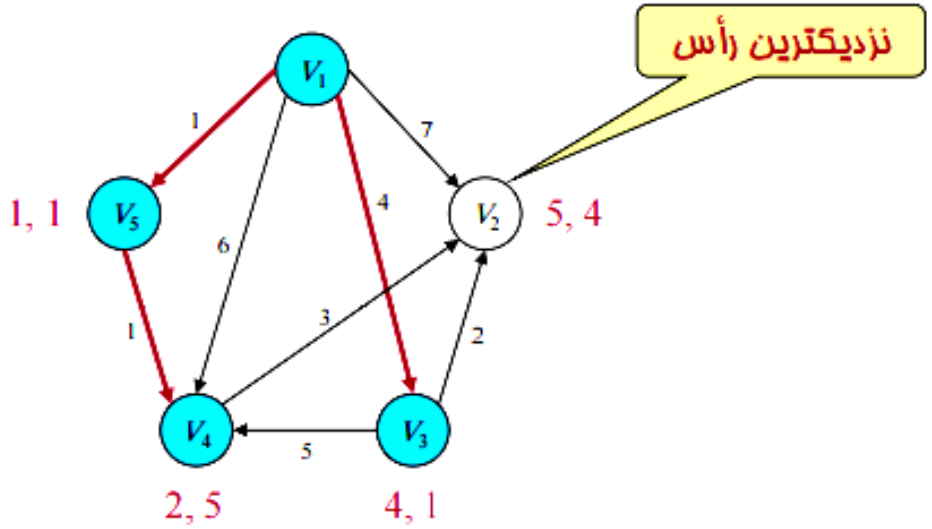
مثال: الگوریتم دیکسترا



	2	3	4	5
touch	4	1	5	1

	2	3	4	5
length	5	4	-1	-1

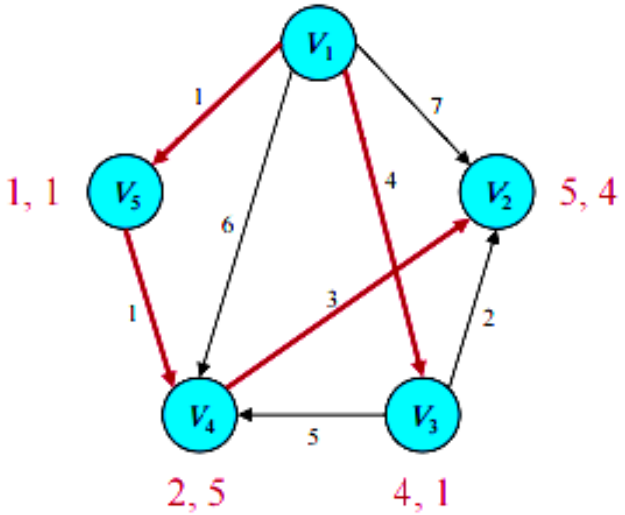
مثال: الگوریتم دیکسترا



	2	3	4	5
touch	4	1	5	1

	2	3	4	5
length	5	-1	-1	-1

مثال: الگوریتم دیکسترا



	2	3	4	5
touch	4	1	5	1

	2	3	4	5
length	-1	-1	-1	-1

آرایه های کمکی

- $Touch\,[i]$ =

اندیس راس v در Y به طوری که یال $<v, v_i>$ آخرین یال روی کوتاهترین مسیر فعلی از V_1 به V_i که فقط از رئوس Y به عنوان رئوس میانی استفاده می کند، می باشد.

- $Length\,[i]$ =

طول کوتاهترین مسیر فعلی از V_1 به V_i که فقط از رئوس Y به عنوان رئوس میانی استفاده می کند، می باشد.

پیچیدگی زمانی برای همه حالات

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