طراحی و تحلیل الگوریتم‌ها

مبحث پنجم

روش‌های طراحی الگوریتم

روش برنامه‌ریزی پویا

Methods of Algorithm Design: Dynamic Programming

کاظم فولادی قلعه

دانشکده مهندسی، دانشکدگان فارابی

دانشگاه تهران

http://courses.fouladi.ir/algorithm

محاسبه‌ی تابع فیبوناچی: روش تقسیم و غلبه

$$f(n) = f(n-1) + f(n-2)$$
$$f(0) = 0$$
$$f(1) = 1$$

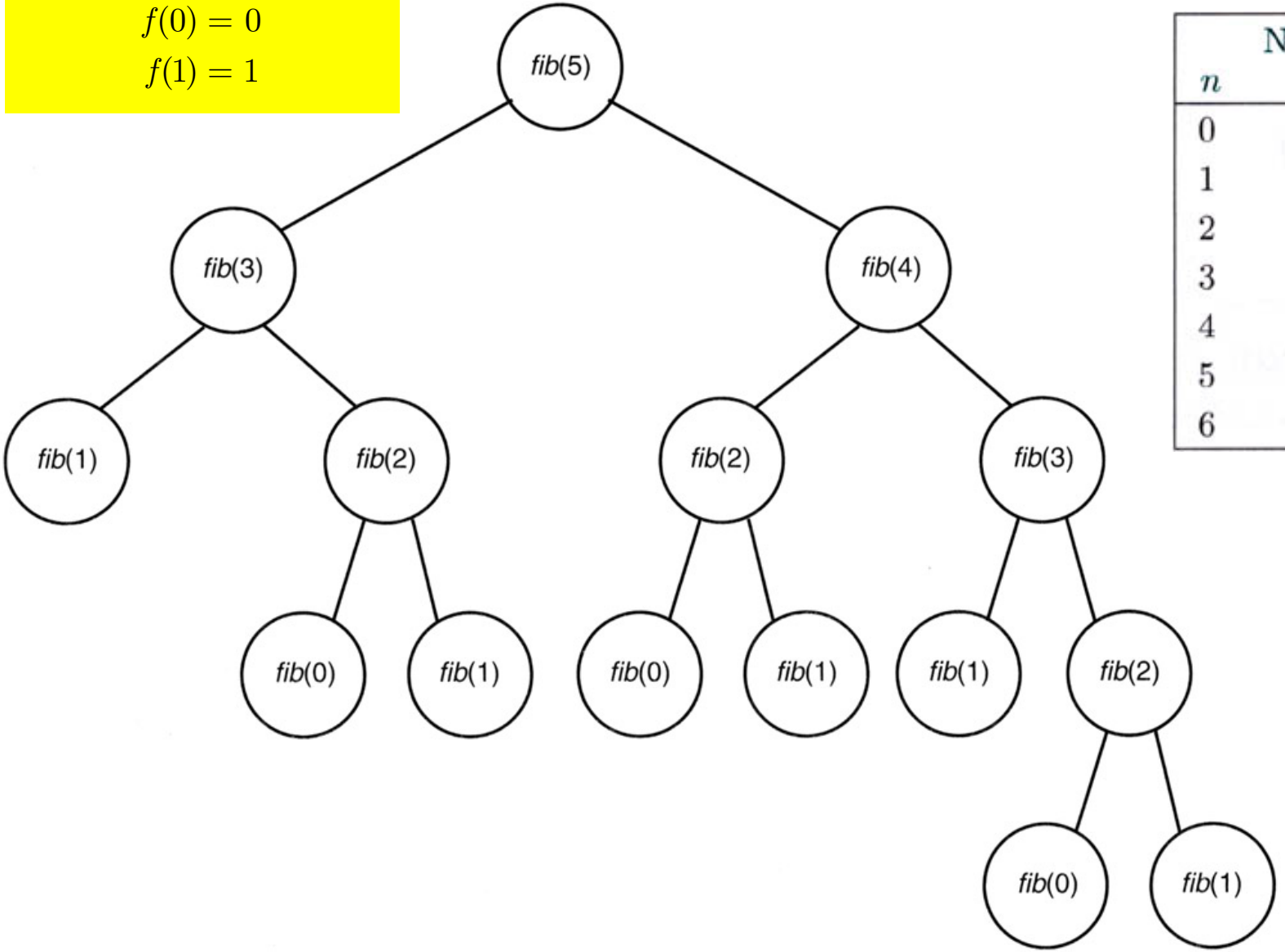
```
FIB-NUMBER(n)
    if n ≥ 0 and n ≤ 1 then
        return n
    else if n > 1 then
        return FIB-NUMBER(n − 1) + FIB-NUMBER(n − 2)
```

روش تقسیم و غلبه به دلیل وجود همپوشانی زیاد بین دو زیرمسئله، کارآمد نیست!

$T(n) \in \Theta(\alpha^n), \alpha > 1$

محاسبه‌ی تابع فیبوناچی: روش تقسیم و غلبه (مثال)

$$f(n) = f(n-1) + f(n-2)$$
$$f(0) = 0$$
$$f(1) = 1$$



n	Number of Terms Computed
0	1
1	1
2	3
3	5
4	9
5	15
6	25

محاسبه‌ی تابع فیبوناچی: روش برنامه‌ریزی پویا

$$f(n) = f(n-1) + f(n-2)$$
$$f(0) = 0$$
$$f(1) = 1$$

```
FIB-NUMBER(n)
    F ← array[0..n]
    F[0] ← 0
    F[1] ← 1
    if n > 1 then
        for i ← 2 to n do
            F[i] ← F[i − 1] + F[i − 2]
    return F[n]
```

از یک جدول برای نگهداری راه‌حل نمونه‌های کوچکتر مسئله استفاده می‌شود تا از محاسبه‌ی تکراری آنها اجتناب شود.

$T(n) \in \Theta(n)$
$S(n) \in \Theta(n)$

محاسبه‌ی تابع فیبوناچی: روش برنامه‌ریزی پویا با بهبود فضای مصرفی

$$f(n) = f(n-1) + f(n-2)$$
$$f(0) = 0$$
$$f(1) = 1$$

```
FIB-NUMBER-IMPROVED(n)
    F_1 ← 0
    F_2 ← 1
    if n = 0 then
        F ← 0
    else if n = 1 then
        F ← 1
    else if n > 1 then
        for i ← 2 to n do
            F ← F_1 + F_2
            F_1 ← F_2
            F_2 ← F
    return F
```

از آنجا که برای محاسبه‌ی مقدار تابع فقط به دو مقدار قبلی آن نیاز است، به جای آرایه از سه متغیر استفاده می‌کنیم.

$T(n) \in \Theta(n)$
$S(n) \in \Theta(1)$

برای حل یک مسئله با روش برنامه‌ریزی پویا:

- **یافتن یک فرمول بازگشتی** که راه‌حل مسئله برای یک نمونه را از روی راه‌حل نمونه‌های کوچکتر همان مسئله مشخص می‌کند.
- **حل** نمونه‌ی مسئله به صورت **پایین به بالا** با شروع از نمونه‌های کوچکتر.

مسائلی با این روش قابل حل هستند که **اصل بهینگی** در مورد آنها صادق باشد:
راه حل بهینه برای هر نمونه از راه حل بهینه‌ی نمونه‌های کوچکتر حاصل می‌شود.

منظور از برنامه‌ریزی، استفاده از **جدول‌بندی** در حل مسئله است.
اصطلاح «پویا» از کاربرد این روش در سیستم‌های کنترل دینامیکی آمده است (Bellman, 1959).

محاسبه‌ی ضریب دوجمله‌ای

تعریف:

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

فرمول بازگشتی:

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

محاسبه‌ی ضریب دوجمله‌ای: روش تقسیم و غلبه

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

```
BINOM(n,k)
    if k = 0 or k = n then
        return 1
    else
        return BINOM(n − 1,k − 1) + BINOM(n − 1,k)
```

زمان اجرا: بر حسب تعداد عمل جمع

اثبات: با استقرای ریاضی

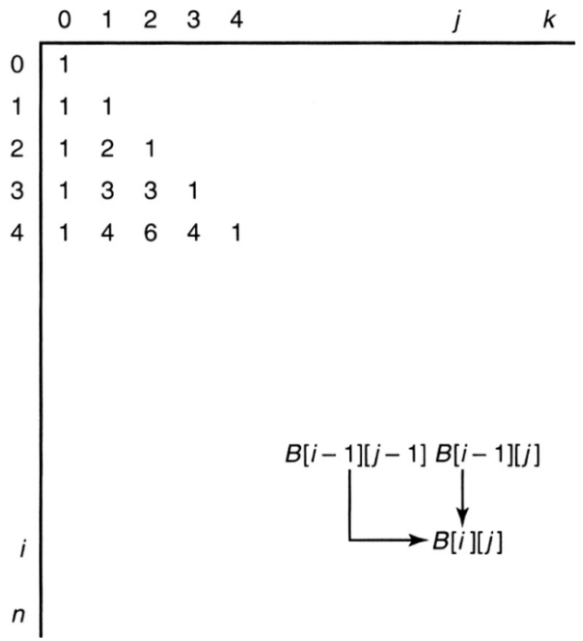
$$T(n) = 2\binom{n}{k} - 1$$

محاسبه‌ی ضریب دوجمله‌ای: روش برنامه‌ریزی پویا

از آرایه‌ی دوبعدی $B[0..n,0..k]$ استفاده می‌کنیم که $B[i,j] := \binom{i}{j}$

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

```
BINOM-DP(n,k)
    B ← array[0..n,0..k]
    for i ← 0 to n do
        for j ← 0 to min(i, k) do
            if j = 0 or j = i then
                B[i,j] ← 1
            else
                B[i,j] ← B[i − 1,j − 1] + B[i − 1,j]
    return B[n,k]
```

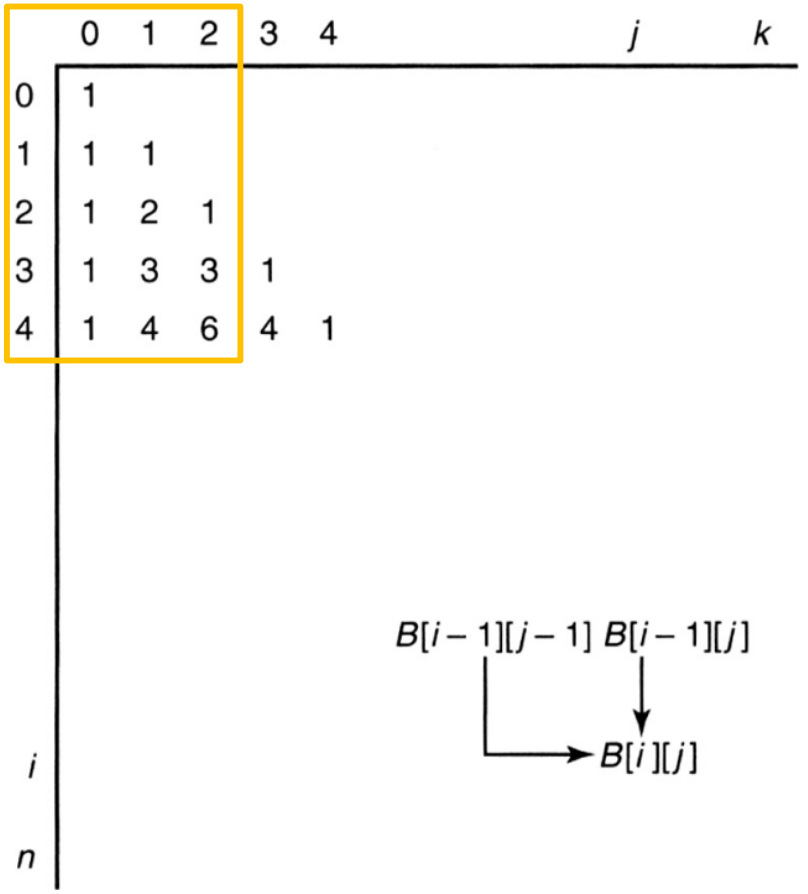


زمان اجرا (بر حسب تعداد عمل جمع): $T(n) \in \Theta(nk)$

فضای مصرفی (جدول): $S(n) \in \Theta(nk)$

محاسبه‌ی ضریب دوجمله‌ای: روش برنامه‌ریزی پویا

- مثال $B[4,2] := \binom{4}{2}$



- B [0,0] = 1

Compute row 1:
- B [1,0] = 1
- B [1,1] = 1

Compute row 2:
- B [2,0] = 1
- B [2,1] = B [1,0] + B [1,1] = 1+1 = 2
- B [2,2] = 1

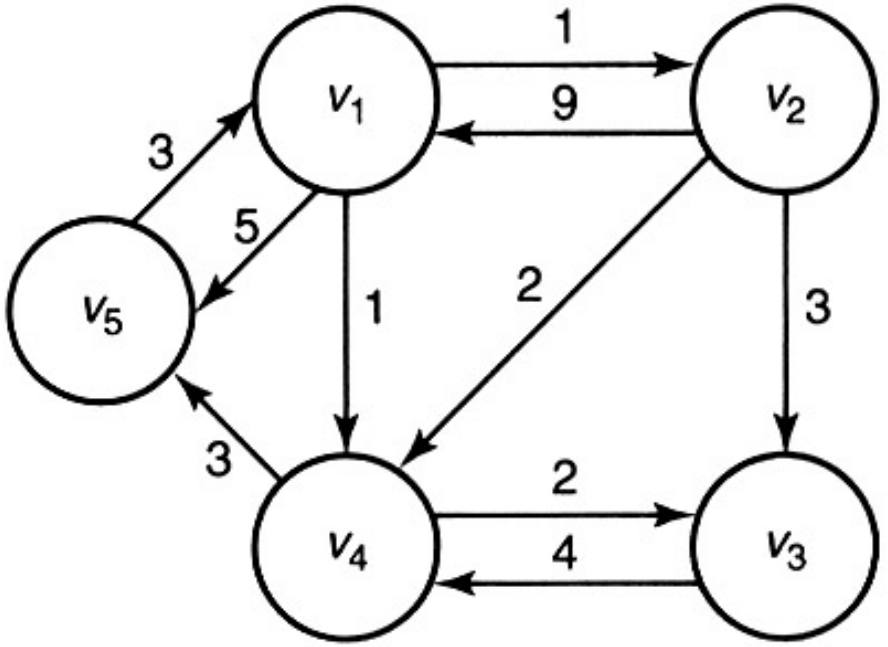
Compute row 3:
- B [3,0] = 1
- B [3,1] = B [2,0] + B [2,1] = 1+2 = 3
- B [3,2] = B [2,1] + B [2,2] = 2+1 = 3

Compute row 4:
- B [4,0] = 1
- B [4,1] = B [3,0] + B [3,1] = 1+3 = 4
- B [4,2] = B [3,1] + B [3,2] = 3+3 = 6

یافتن کوتاه‌ترین مسیر بین هر دو رأس یک گراف

FLOYD-WARSHALL ALGORITHM

هدف: یافتن کوتاه‌ترین مسیر بین هر دو رأس دلخواه از یک گراف جهت‌دار وزن‌دار

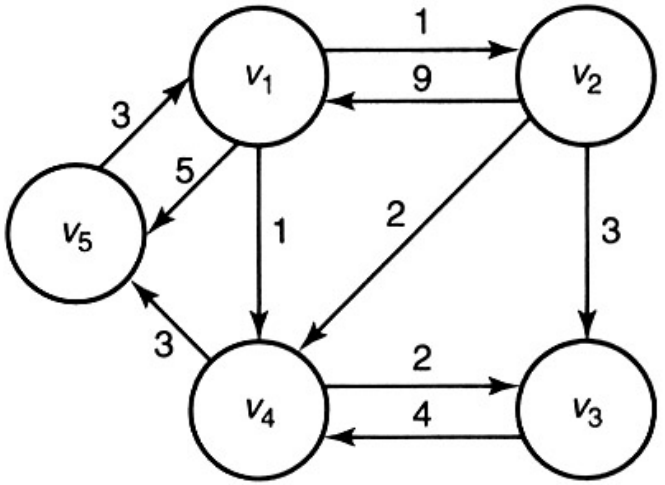


یافتن کوتاه‌ترین مسیر بین هر دو رأس یک گراف

FLOYD-WARSHALL ALGORITHM

هدف: یافتن کوتاه‌ترین مسیر بین هر دو رأس دلخواه از یک گراف جهت‌دار وزن‌دار
ورودی: ماتریس وزن‌های گراف

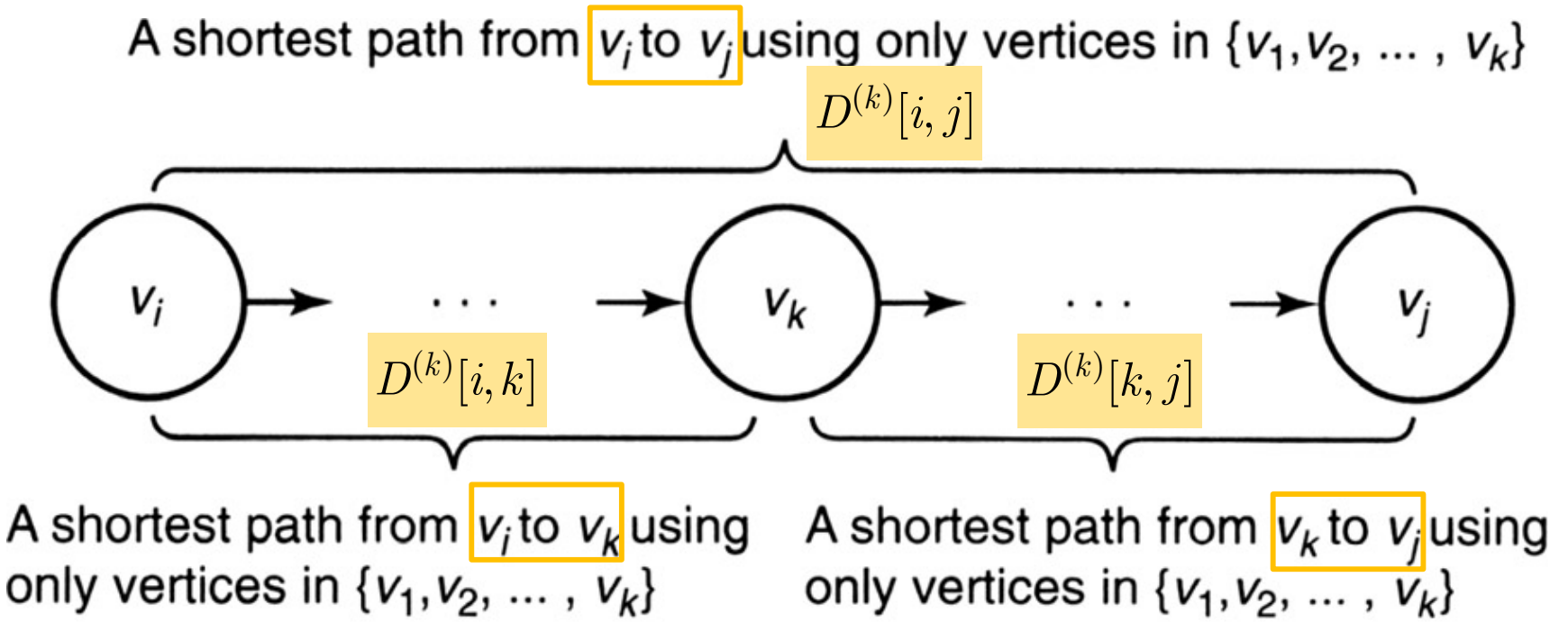
$$W[i,j] = \begin{cases} weight\ on\ edge & ,(v_i, v_j) \in E(G) \\ \infty & ,(v_i, v_j) \notin E(G) \\ 0 & ,i = j \end{cases}$$



	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

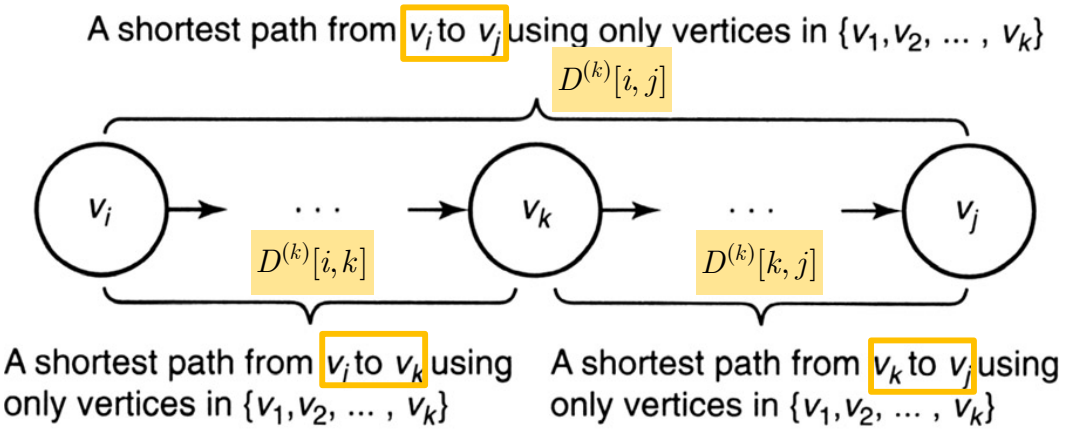
یافتن کوتاه‌ترین مسیر بین هر دو رأس یک گراف



فرمول بازگشتی: کوتاه‌ترین مسیر بین دو رأس

$$D^{(k)}[i,j] = \min_{1 \le k \le n} \{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$

یافتن کوتاه‌ترین مسیر بین هر دو رأس یک گراف



فرمول بازگشتی: کوتاه‌ترین مسیر بین دو رأس

$$D^{(k)}[i,j] = \min_{1 \le k \le n} \{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$

$$\begin{array}{ccccccccc} D^{(0)} & \rightarrow & D^{(1)} & \rightarrow & D^{(2)} & \rightarrow & \cdots \rightarrow & D^{(n-1)} & \rightarrow & D^{(n)} \\ \uparrow & & & & & & & & & \uparrow \\ W & & & & & & & & & D \end{array}$$

یافتن کوتاه‌ترین مسیر بین هر دو رأس یک گراف

هدف: یافتن کوتاه‌ترین مسیر بین دو رأس دلخواه از یک گراف جهت‌دار وزن‌دار

ورودی: ماتریس وزن‌های گراف: W

خروجی: ماتریس کوتاه‌ترین فاصله‌ها: D

فرمول بازگشتی:
کوتاه‌ترین مسیر بین دو رأس

$$D^{(k)}[i,j] = \min_{1 \le k \le n} \{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$

می‌توان کل محاسبات را با استفاده از یک آرایه‌ی D انجام داد
زیرا مقادیر سطر k و ستون k در تکرار k–ام حلقه تغییر نمی‌کند:

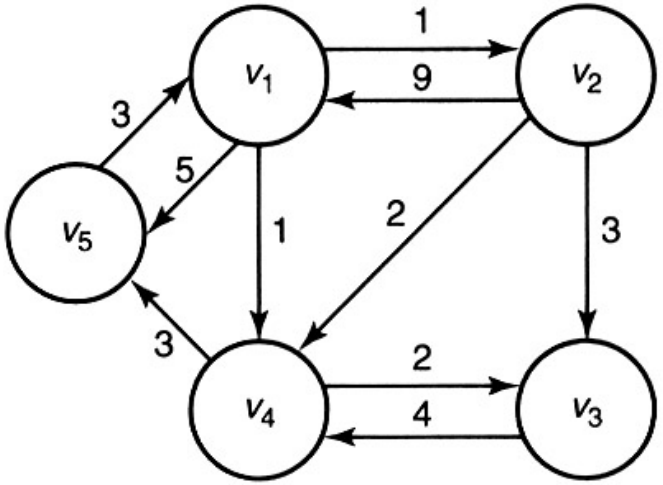
$$D[i,j] \leftarrow \min_{1 \le k \le n} \{D[i,j], D[i,k] + D[k,j]\}$$

```
SHORTEST-PATH(W, n)
    D ← W
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i,j] ← min(D[i,j], D[i,k] + D[k,j])
    return D
```

زمان اجرا: $T(n) \in \Theta(n^3)$

فضای مورد نیاز: $S(n) \in \Theta(n^2)$

یافتن کوتاه‌ترین مسیر بین هر دو رأس یک گراف: مثال



	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

یافتن کوتاه‌ترین مسیر بین هر دو رأس یک گراف

چاپ کوتاه‌ترین مسیر بین هر دو رأس گراف

هدف: یافتن کوتاه‌ترین مسیر بین دو رأس دلخواه از یک گراف جهت‌دار وزن‌دار

ورودی: ماتریس وزن‌های گراف: W

خروجی: ماتریس کوتاه‌ترین فاصله‌ها: D و خود کوتاه‌ترین مسیر

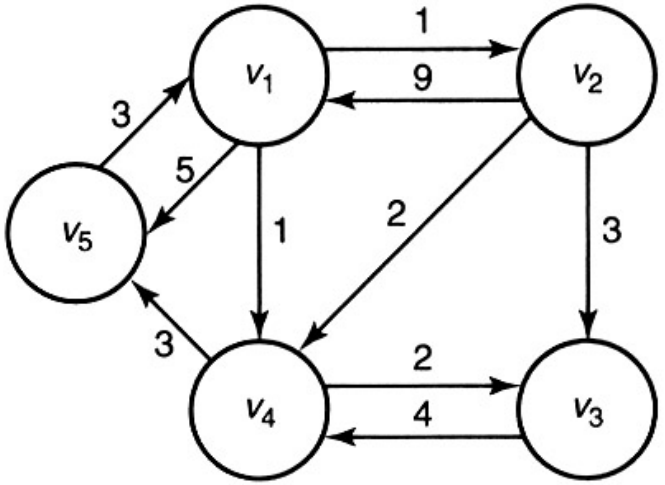
```
SHORTEST-PATH(W, n)
    P ← array[1..n, 1..n]
    P ← [0]
    D ← W
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                if D[i,j] > D[i,k] + D[k,j] then
                    D[i,j] ← D[i,k] + D[k,j]
                    P[i,j] ← k
    return D
```

```
PATH(P, i, j)
    if P[i,j] ≠ 0 then
        PATH(P, i, P[i,j])
        print P[i,j]
        PATH(P, P[i,j], j)
```

$P[i,j]$ بزرگترین اندیس رئوس میانی در کوتاه‌ترین مسیر از i تا j است.

یافتن کوتاه‌ترین مسیر بین هر دو رأس یک گراف: مثال

چاپ کوتاه‌ترین مسیر بین هر دو رأس گراف



	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

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

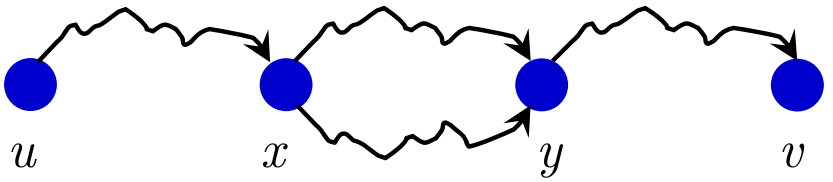
P

```
PATH(P, i, j)
    if P[i,j] ≠ 0 then
        PATH(P, i, P[i,j])
        print P[i,j]
        PATH(P, P[i,j], j)
```

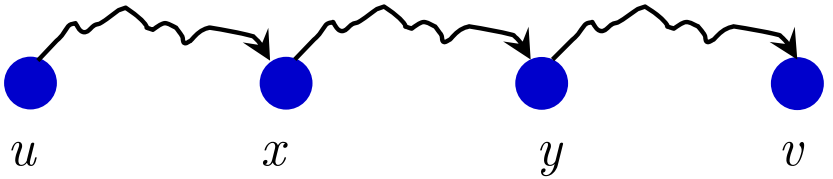
اصل بهینگی

اصل بهینگی:

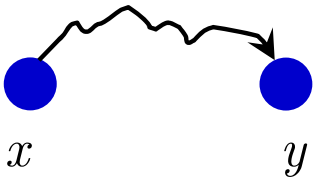
راه حل بهینه برای هر نمونه از راه حل بهینه‌ی نمونه‌های کوچکتر حاصل می‌شود.



اگر این کوتاه‌ترین مسیر بین u و v باشد:

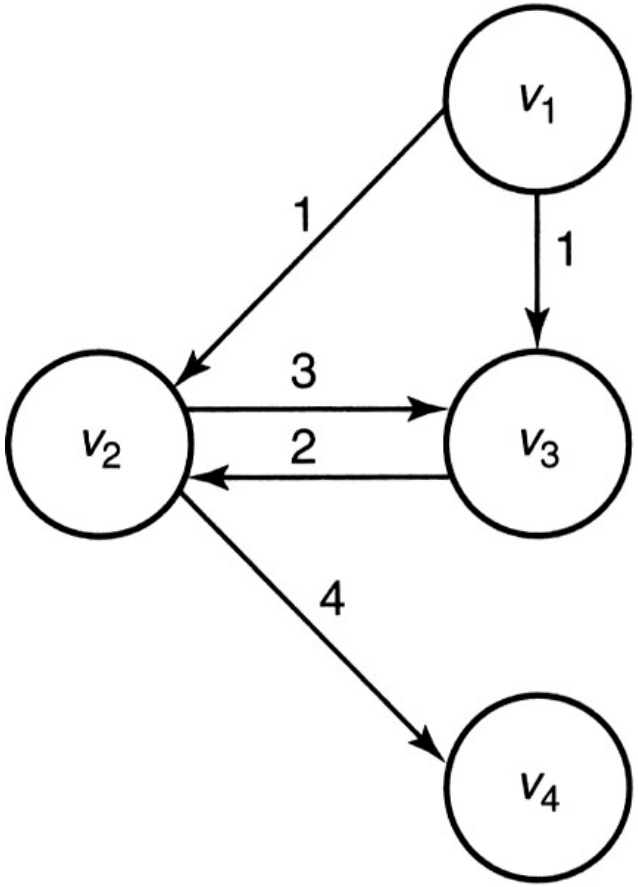


آن‌گاه این کوتاه‌ترین مسیر بین x و y است:



مسئله‌ی طولانی‌ترین مسیر ساده و اصل بهینگی

اگر **اصل بهینگی** در مورد مسئله‌ای برقرار نباشد، نمی‌توان روش برنامه‌نویسی پویا را برای آن به کار برد.



In Figure the optimal (longest) simple path form v_1 to v_4 is $[v_1, v_3, v_2, v_4]$. However, the subpath $[v_1, v_3]$ is not an optimal (longest) path from v_1 to v_3 because

$$length\,[v_1, v_3] = 1 \quad \text{and} \quad length\,[v_1, v_2, v_3] = 4.$$

Therefore, the principle of optimality does not apply. The reason for this is that the optimal paths from v_1 to v_3 and from v_3 to v_4 cannot be strung together to given an optimal path from v_1 to v_4.

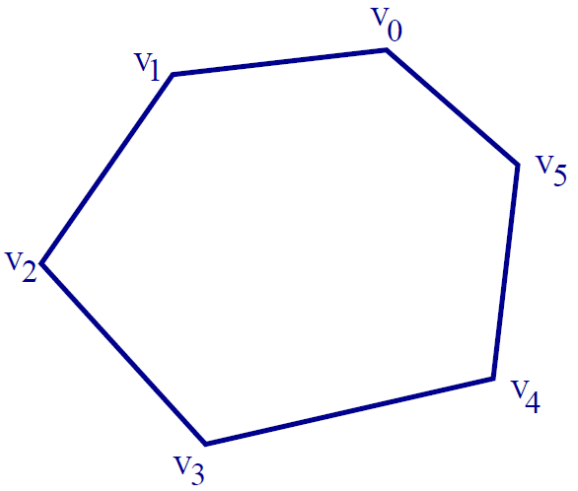
مثلث‌بندی بهینه‌ی یک چندضلعی محدب

CONVEX OPTIMAL POLYGON TRIANGULATION

- **چندضلعی:** یک منحنی بسته متشکل از پاره‌خطها (اضلاع)
- **رأس:** محل برخورد دو ضلع
- **چندضلعی محدب:** یک چندضلعی که پاره‌خط مابین هر دو نقطه‌ی دلخواه آن کاملاً درون همان چندضلعی قرار گیرد.
- ترتیب رئوس را بر خلاف جهت حرکت عقربه‌های ساعت می‌گیریم:

$v_0, v_1, \ldots, v_{n-1}$. **Edges are** $v_i v_{i+1}$, **where** $v_n = v_0$

- **قطر:** پاره خط اتصال‌دهنده‌ی دو رأس غیرمجاور

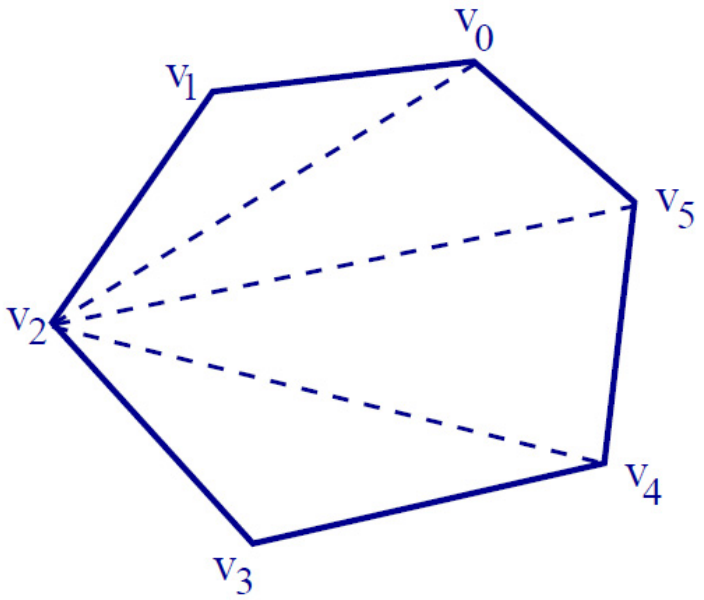


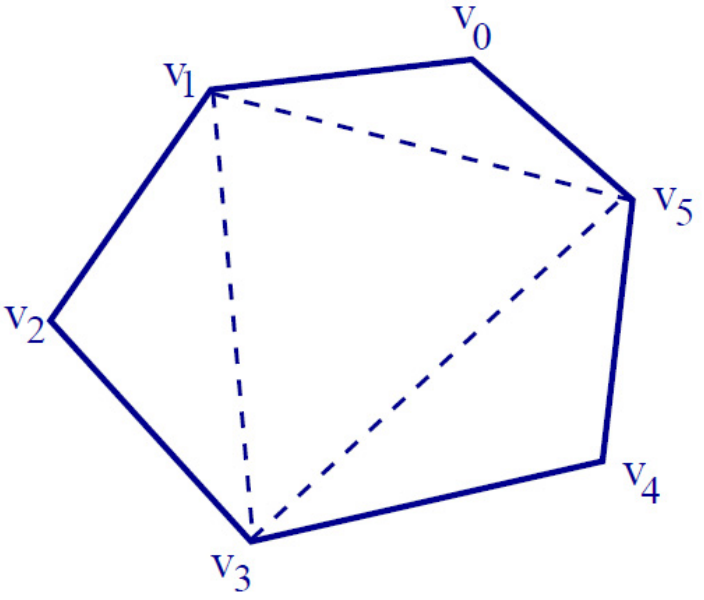
مثلث‌بندی بهینه‌ی یک چندضلعی محدب

CONVEX OPTIMAL POLYGON TRIANGULATION

مثلث‌بندی:

تعیین مجموعه‌ای از قطرها که چندضلعی را به مثلث‌های ناهمپوشان تقسیم می‌کند.





n رأس
n-3 قطر
n-2 مثلث

مثلث‌بندی بهینه:

یک چندضلعی P و تابع وزن w داده شده است. $P = (v_0, \ldots, v_{n-1})$

می‌خواهیم یک مثلث‌بندی با می‌نیمم وزن مجموع را بیابیم.

مثلث‌بندی بهینه‌ی یک چندضلعی محدب: تابع وزن

یک نمونه تابع وزن:

$$w(\triangle v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$$

مثلث‌بندی بهینه‌ی یک چندضلعی محدب: تابع وزن

یک نمونه تابع وزن:

$$w(\triangle v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$$

$m[i, j]$ وزن بهینه‌ی مثلث‌سازی زیرچندضلعی $(v_i, v_{i+1}, \ldots, v_j)$

فرمول وزن یک مثلث‌بندی بهینه:

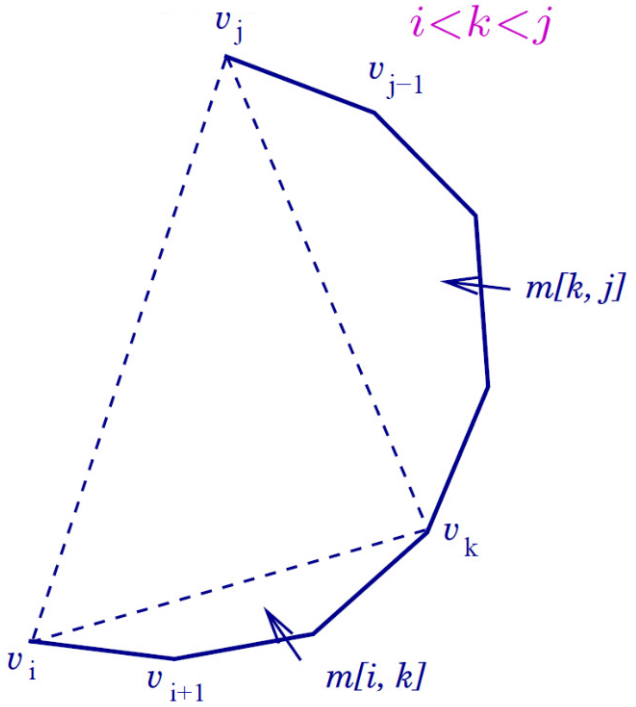
$$m[i, j] = m[i, k] + m[k, j] + w(\triangle v_i v_j v_k)$$

مثلث‌بندی بهینه‌ی یک چندضلعی محدب

فرمول بازگشتی

برای یافتن وزن بهینه‌ی مثلث‌سازی برای یک زیرچندضلعی $(v_i, v_{i+1}, \ldots, v_j)$

$$m[i,j] = \min_{i<k<j} \{m[i,k] + m[k,j] + w(\triangle v_i v_j v_k)\}$$

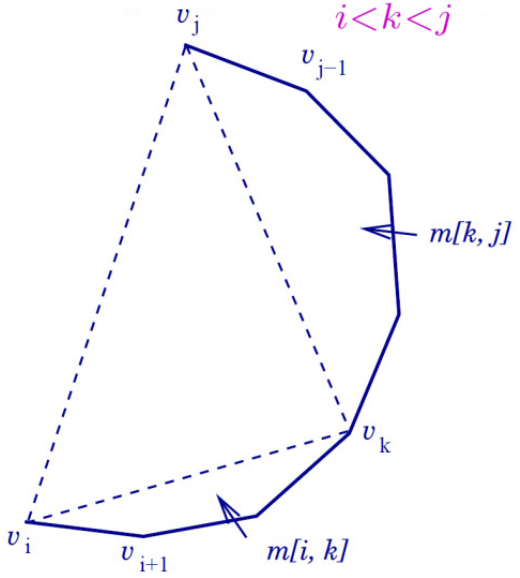


مثلث‌بندی بهینه‌ی یک چندضلعی محدب: شبه‌کد

فرمول بازگشتی

$$m[i,j] = \min_{i<k<j}\{m[i,k] + m[k,j] + w(\triangle v_i v_j v_k)\}$$

برای یافتن وزن بهینه‌ی مثلث‌سازی برای یک زیرچندضلعی $(v_i, v_{i+1}, \ldots, v_j)$

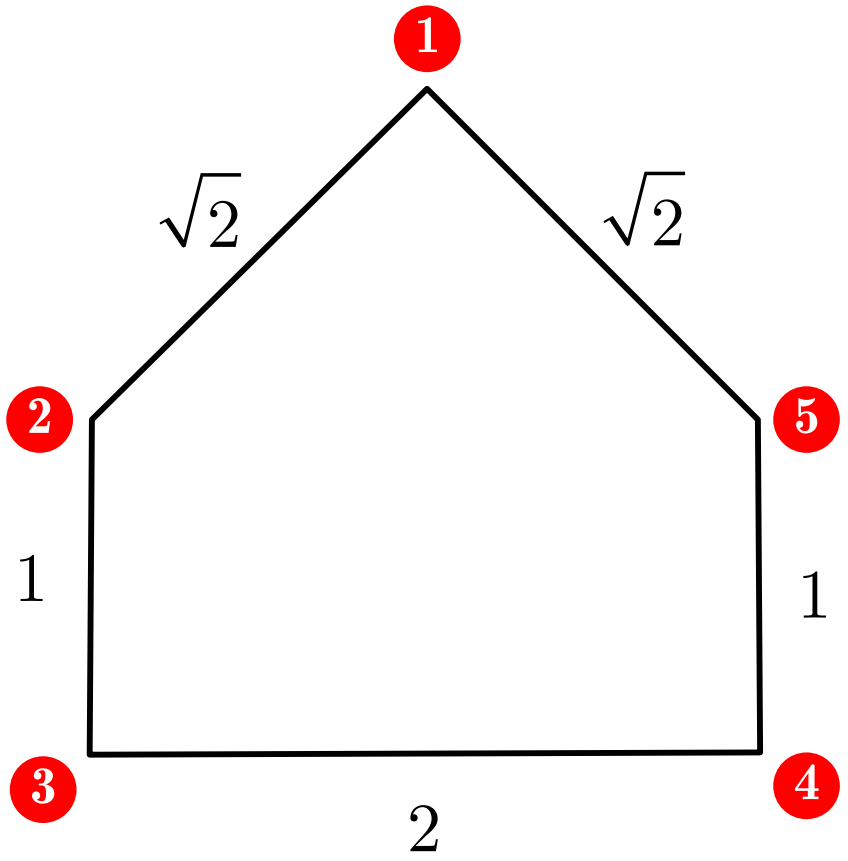


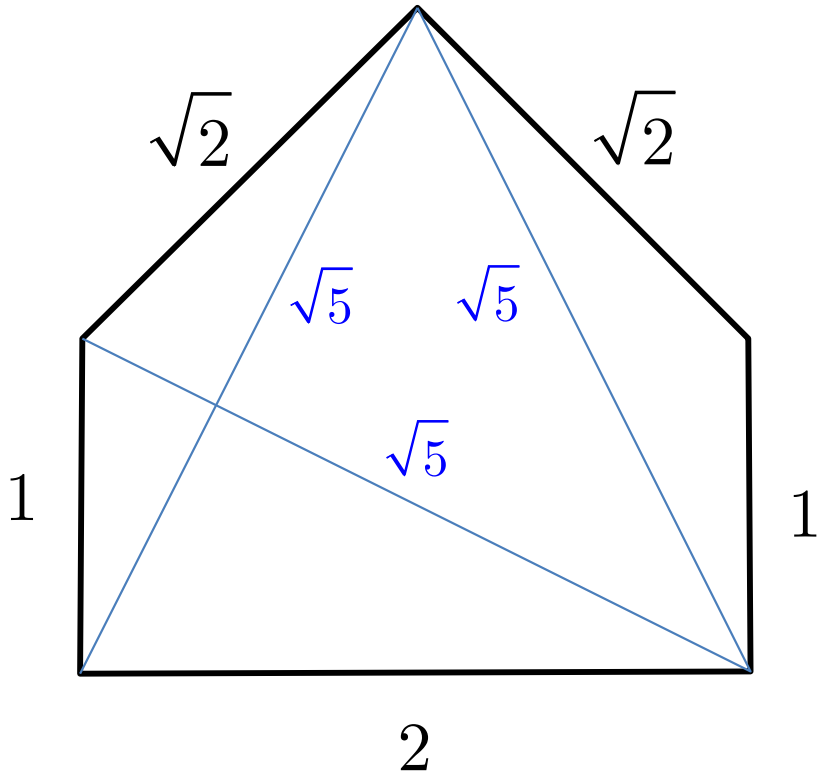

```
TRIANGULATION(P,n)
    m ← array[1..n,1..n]
    for i ← 1 to n do
        m[i, i+1] ← 0
    for d ← 1 to n − 1 do
        for i ← 1 to n − d do
            j ← i + d
            m[i,j] ← ∞
            for k ← i+1 to j − 1 do
                q ← m[i,k] + m[k,j] + w(P,[i,j,k])
                if q < m[i,j] then
                    m[i,j] ← q
    return m[1,n]
```

زمان اجرا:	$T(n) \in \Theta(n^3)$
فضای مورد نیاز:	$S(n) \in \Theta(n^2)$

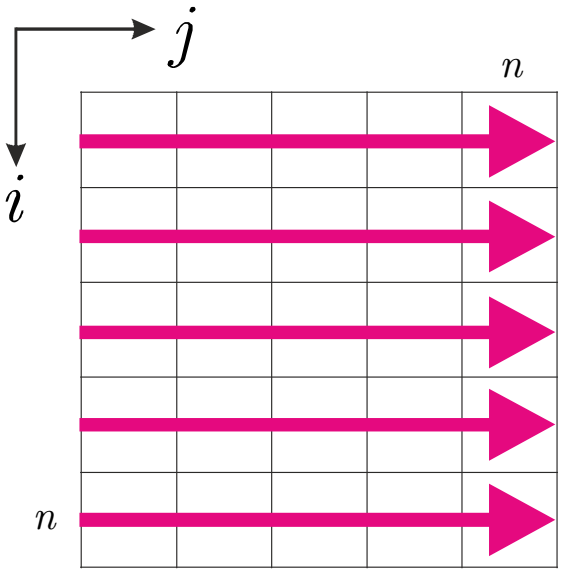
مثلث‌بندی بهینه‌ی یک چندضلعی محدب: مثال

مثلث‌بندی با کم‌ترین وزن را برای چندضلعی زیر بیابید:



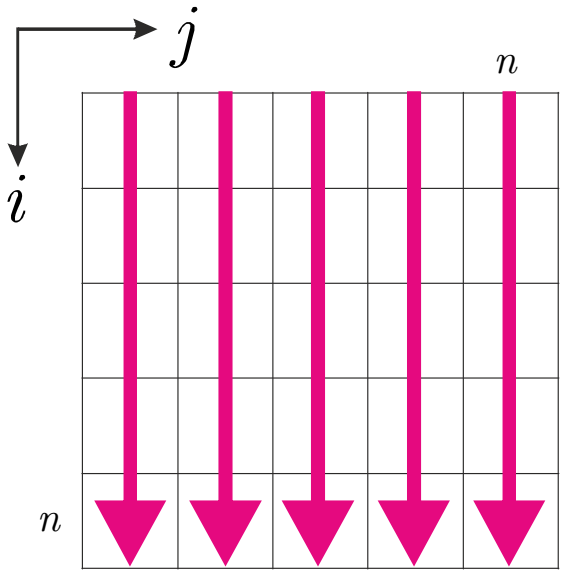


ترتیب پر کردن جدول



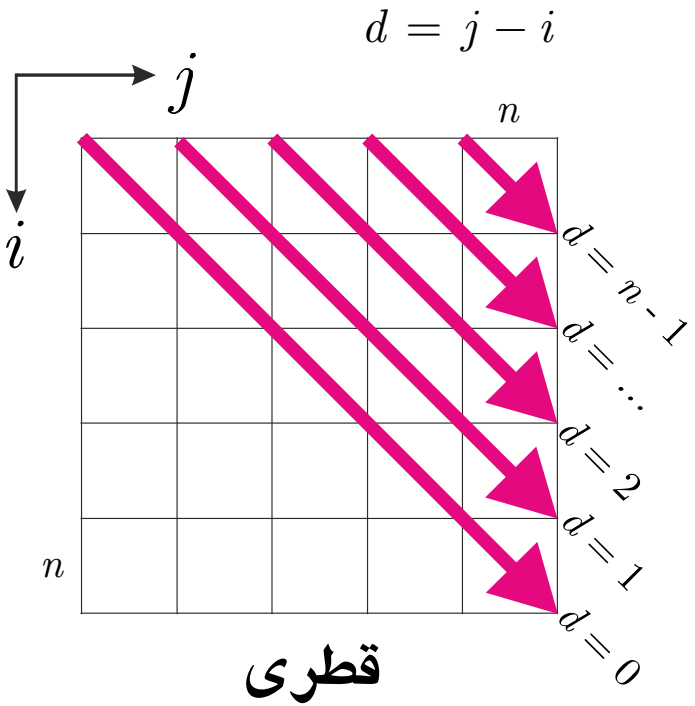
سطری
Row-major

```
for i ← 1 to n do
    for j ← 1 to n do
```



ستونی
Column-major

```
for j ← 1 to n do
    for i ← 1 to n do
```



قطری
Diagonal-major

```
for d ← 0 to n-1 do
    for i ← 1 to n-d do
        j ← i + d
```

ضرب زنجیره‌ای ماتریس‌ها

MATRIX CHAIN PRODUCT

فرض کنید می‌خواهیم حاصلضرب دو ماتریس زیر را به دست آوریم:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

برای یافتن عنصر سطر اول و ستون اول حاصلضرب داریم:

$$\underbrace{1 \times 7 + 2 \times 2 + 3 \times 6}_{\text{3 multiplications}}.$$

چون حاصلضرب ۲×۴ عنصر دارد، مجموعاً ۳×۴×۲ ضرب ساده لازم است:

$$2 \times 4 \times 3 = 24.$$

ضرب زنجیره‌ای ماتریس‌ها

به طور کلی در حاصلضرب دو ماتریس با روش استاندارد

$$\underbrace{\begin{bmatrix} A & & \\ & \ddots & \end{bmatrix}}_{i\times j} \underbrace{\begin{bmatrix} B & & \\ & \ddots & \\ & & \ddots \end{bmatrix}}_{j\times k}$$

$$i \times j \times k$$

ضرب ساده لازم است

ضرب زنجیره‌ای ماتریس‌ها

اگر بیش‌تر از دو ماتریس برای ضرب داشته باشیم،
آن‌ها را می‌توان به حالت‌های مختلفی در هم ضرب کرد
(*خاصیت شرکت‌پذیری ضرب ماتریسی*).

$$A(B(CD))$$
$$(AB)(CD)$$
$$A((BC)D)$$
$$((AB)C)D$$
$$(A(BC))D$$

A	$\times$	B	$\times$	C	$\times$	D
20×2		2×30		30×12		12×8

کدام ترتیب کم‌ترین تعداد ضرب‌ها را دارد؟

ضرب زنجیره‌ای ماتریس‌ها

$$\underset{20\times2}{A} \times \underset{2\times30}{B} \times \underset{30\times12}{C} \times \underset{12\times8}{D}$$

کدام ترتیب کم‌ترین تعداد ضرب‌ها را دارد؟

$A(B(CD))$	$30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 = 3680$
$(AB)(CD)$	$20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 = 8880$
$A((BC)D)$	$2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 = \boxed{1232}$
$((AB)C)D$	$20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 = 10320$
$(A(BC))D$	$2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 = 3120$

ضرب زنجیره‌ای ماتریس‌ها

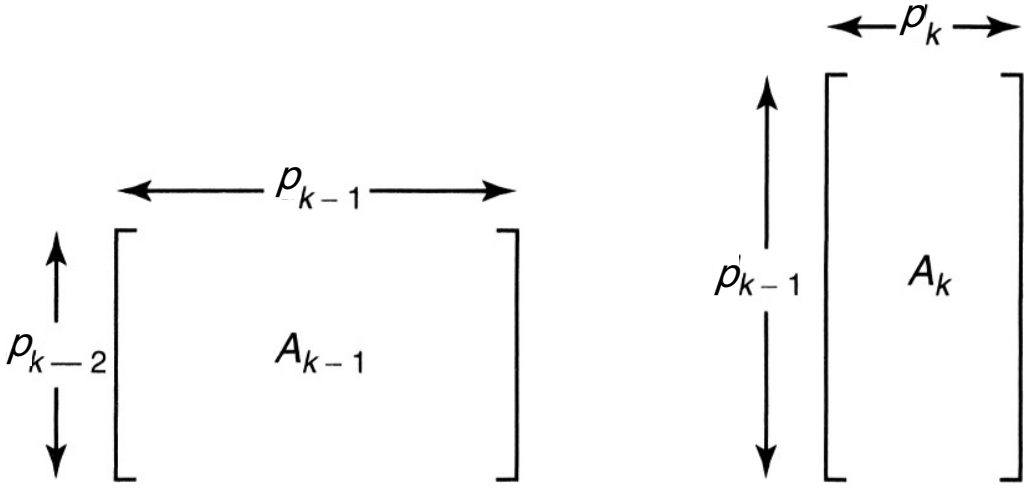
$$A_1 \times A_2 \times A_3 \times \cdots \times A_n$$

زنجیره‌ای از n ماتریس داریم که باید در هم ضرب شوند.

هدف: یافتن ترتیبی از ماتریس‌ها که کم‌ترین تعداد ضرب را داشته باشد.

ابعاد ماتریس‌ها در آرایه‌ی p داده می‌شود.

$$p = \begin{bmatrix} p_0 & p_1 & p_2 & \cdots & p_n \end{bmatrix}$$



ضرب زنجیره‌ای ماتریس‌ها

فرض می‌کنیم:

$m[i,j]$ حداقل تعداد ضرب‌ها در پرانتزگذاری بهینه‌ی ضرب $A_i \dots A_j$ باشد.

در این صورت

$m[1,n]$ جواب مسئله خواهد بود.

اگر جواب بهینه در نقطه‌ی k شکسته شود، داریم:

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

$$(A_i \dots A_k)(A_{k+1} \dots A_j)$$

ضرب زنجیره‌ای ماتریس‌ها

رابطه‌ی بازگشتی:

$$(A_i \dots A_k)(A_{k+1} \dots A_j)$$

$$m[i,j] = \begin{cases} 0 & ,i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & ,i < j \end{cases}$$

ضرب زنجیره‌ای ماتریس‌ها

شبه کد

رابطه‌ی بازگشتی:

$$m[i,j] = \begin{cases} 0 & ,i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & ,i < j \end{cases}$$

```
MATRIX-CHAIN-ORDER(p)
    n ← length(p) − 1
    m ← array[1..n,1..n]
    for i ← 1 to n do
        m[i, i] ← 0
    for d ← 1 to n − 1 do
        for i ← 1 to n − d do
            j ← i + d
            m[i,j] ← ∞
            for k ← i to j − 1 do
                q ← m[i,k] + m[k + 1,j] + p[i − 1]*p[k]*p[j]
                if q < m[i,j] then
                    m[i,j] ← q
                    s[i,j] ← k
    return m[1,n]
```

چاپ ترتیب بهینه

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2      then print "A"_i
3      else print "("
4           PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5           PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6           print ")"
```

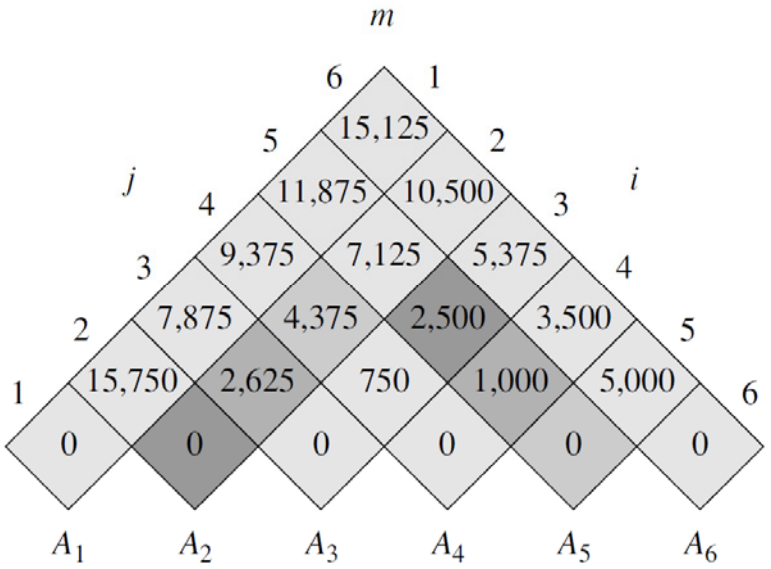
زمان اجرا: $T(n) \in \Theta(n^3)$

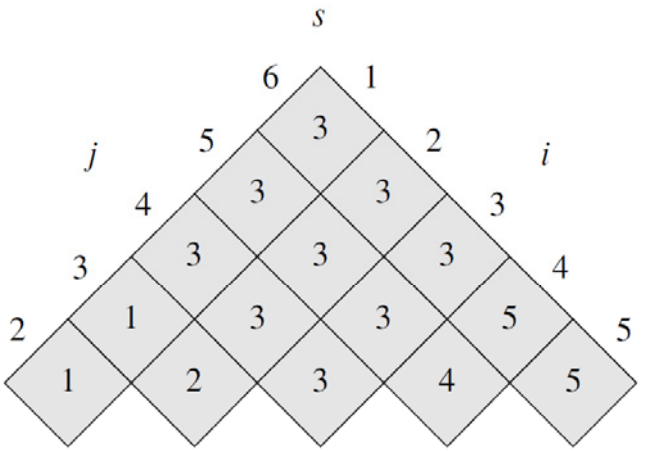
فضای مورد نیاز: $S(n) \in \Theta(n^2)$

ضرب زنجیره‌ای ماتریس‌ها

مثال

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25





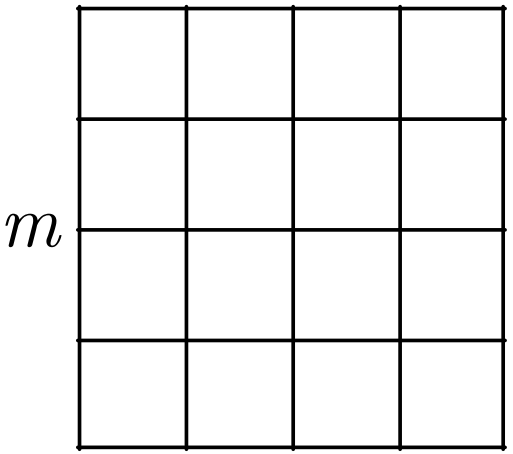
$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$
$$= 7125\,.$$

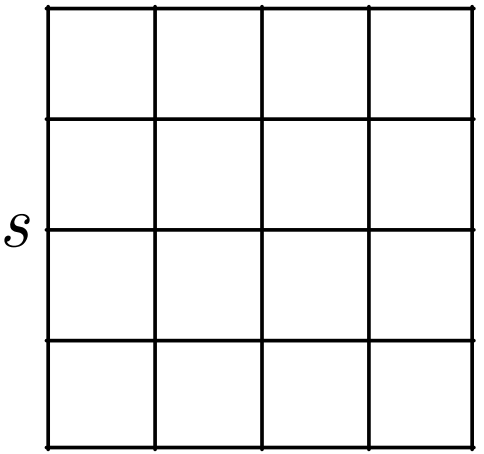
ضرب زنجیره‌ای ماتریس‌ها

مثال

$$\underset{100\times1}{A_1} \times \underset{1\times100}{A_2} \times \underset{100\times1}{A_3} \times \underset{1\times100}{A_4}$$

$$p = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 & p_4 \end{bmatrix} = \begin{bmatrix} 100 & 1 & 100 & 1 & 100 \end{bmatrix}$$





تکنیک به‌خاطرسپاری

MEMOIZAION

تکنیک به‌خاطرسپاری:

نوعی پیاده‌سازی برنامه‌ریزی پویا که از روش بالا به پایین بهره‌برداری می‌کند.

- ساختار کنترلی برای پر کردن جدول، مشابه روش بازگشتی است.
- هر زیرمسئله که حل شد، مقادیر آن در جدول ذخیره می‌شود.
- جدول در ابتدا با مقادیر اولیه، مقداردهی می‌شود.

ضرب زنجیره‌ای ماتریس‌ها با تکنیک به‌خاطرسپاری

```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3        do for j ← i to n
4              do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)
```

```
LOOKUP-CHAIN(p, i, j)
1  if m[i, j] < ∞
2     then return m[i, j]
3  if i = j
4     then m[i, j] ← 0
5     else  for k ← i to j − 1
6                do q ← LOOKUP-CHAIN(p, i, k)
                        + LOOKUP-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
7                   if q < m[i, j]
8                      then m[i, j] ← q
9  return m[i, j]
```

طولانی‌ترین زیردنباله‌ی مشترک

LONGEST COMMON SUBSEQUENCE (LCS)

دنباله‌ی زیر را در نظر می‌گیریم:

$$X = \langle x_1, x_2, \ldots, x_n \rangle$$

یک **زیر دنباله** از یک دنباله، دنباله‌ی حاصل از حذف صفر یا بیشتر عنصر از آن است.

مثال:

$$X = \langle A, B, C, B, D, A, B \rangle$$

$Z_1 = \langle A, C, B, D, B \rangle$ زیردنباله

$Z_2 = \langle A, B, D \rangle$ زیردنباله

$Z_3 = \langle C, D, A \rangle$ زیردنباله

یک دنباله به طول n حداکثر 2^n زیردنباله دارد.

طولانی‌ترین زیردنباله‌ی مشترک

LONGEST COMMON SUBSEQUENCE (LCS)

دو دنباله‌ی زیر را در نظر می‌گیریم:

$$X = \langle x_1, x_2, \ldots, x_m \rangle \qquad Y = \langle y_1, y_2, \ldots, y_n \rangle$$

یک **زیر دنباله‌ی مشترک** دو دنباله، دنباله‌ای است که زیردنباله‌ی هر دو باشد.

مثال:

$$X = \langle A, B, C, B, D, A, B \rangle$$

$$Y = \langle B, D, C, A, B, A \rangle$$

$Z_1 = \langle B, C, A \rangle$ زیردنباله‌ی مشترک

$Z_2 = \langle A, B \rangle$ زیردنباله‌ی مشترک

$Z_3 = \langle B, D, A, B \rangle$ طولانی‌ترین زیردنباله‌ی مشترک

طولانی‌ترین زیردنباله‌ی مشترک

LONGEST COMMON SUBSEQUENCE (LCS)

دو دنباله‌ی زیر را در نظر می‌گیریم:

$$X = \langle x_1, x_2, \ldots, x_m \rangle \qquad Y = \langle y_1, y_2, \ldots, y_n \rangle$$

هدف: یافتن **طولانی‌ترین زیر دنباله‌ی مشترک** دو دنباله است.

طولانی‌ترین زیردنباله‌ی مشترک: ساختار

LONGEST COMMON SUBSEQUENCE (LCS)

دو دنباله‌ی $X = \langle x_1, x_2, \ldots, x_m \rangle$ $Y = \langle y_1, y_2, \ldots, y_n \rangle$

اگر $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ $Y_i = \langle y_1, y_2, \ldots, y_i \rangle$

و طولانی‌ترین زیردنباله‌ی مشترک X و Y، را $Z = \langle z_1, z_2, \ldots, z_k \rangle$ بنامیم،
$Z_i = \langle z_1, z_2, \ldots, z_i \rangle$

آن‌گاه:

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and
$$Z_{k-1} = LCS(X_{m-1}, Y_{n-1}).$$

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies
$$Z = LCS(X_{m-1}, Y).$$

3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies
$$Z = LCS(X, Y_{n-1}).$$

طولانی‌ترین زیردنباله‌ی مشترک: فرمول بازگشتی

LONGEST COMMON SUBSEQUENCE (LCS)

تعریف می‌کنیم:

$$c[i,j] = |LCS(X_i, Y_j)|$$

در این صورت فرمول بازگشتی زیر از ساختار LCS به دست می‌آید:

$$c[i,j] = \begin{cases} 0 & \textbf{if } i \textbf{ or } j = 0 \\ c[i-1,j-1]+1 & \textbf{if } x_i = y_j \\ \max\{c[i,j-1],\ c[i-1,j]\} & \textbf{if } x_i \neq y_j \end{cases}$$

$$X = \langle x_1, x_2, \ldots, x_m \rangle \qquad Y = \langle y_1, y_2, \ldots, y_n \rangle$$

طولانی‌ترین زیردنباله‌ی مشترک: شبه کد

LONGEST COMMON SUBSEQUENCE (LCS)

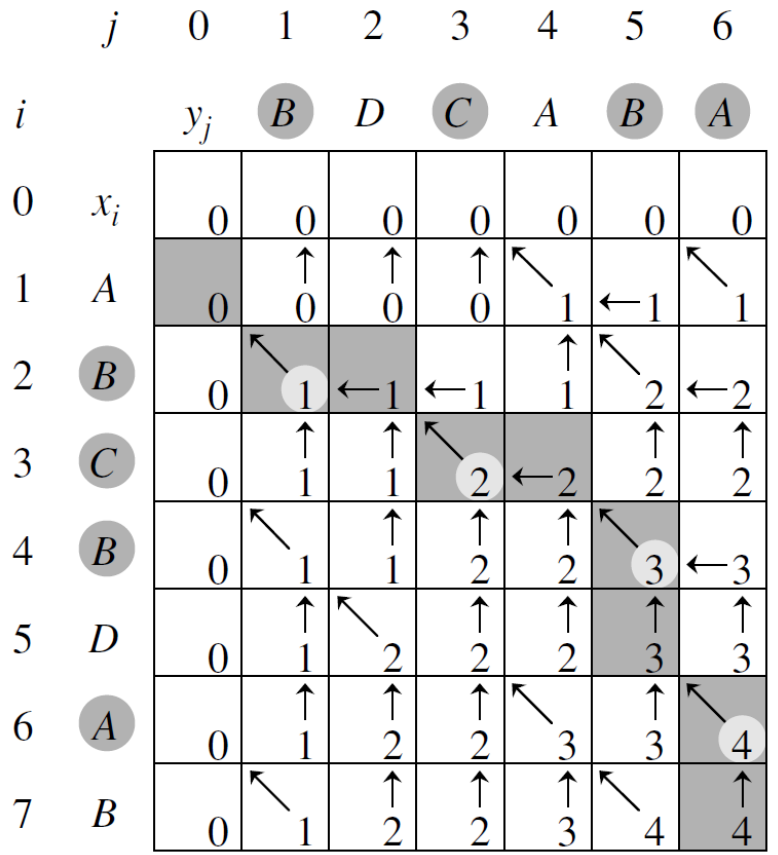
```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4        do c[i, 0] ← 0
 5  for j ← 0 to n
 6        do c[0, j] ← 0
 7  for i ← 1 to m
 8        do for j ← 1 to n
 9              do if x_i = y_j
10                    then c[i, j] ← c[i − 1, j − 1] + 1
11                         b[i, j] ← "↖"
12                    else if c[i − 1, j] ≥ c[i, j − 1]
13                           then c[i, j] ← c[i − 1, j]
14                                b[i, j] ← "↑"
15                           else c[i, j] ← c[i, j − 1]
16                                b[i, j] ← "←"
17  return c and b
```

زمان اجرا:	$T(m,n) \in \Theta(mn)$
فضای مورد نیاز:	$S(m,n) \in \Theta(mn)$

طولانی‌ترین زیردنباله‌ی مشترک: شبه کد

چاپ طولانی‌ترین زیردنباله‌ی مشترک

```
PRINT-LCS(b, X, i, j)
1  if i = 0 or j = 0
2     then return
3  if b[i, j] = "↖"
4     then PRINT-LCS(b, X, i − 1, j − 1)
5          print x_i
6  elseif b[i, j] = "↑"
7     then PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```



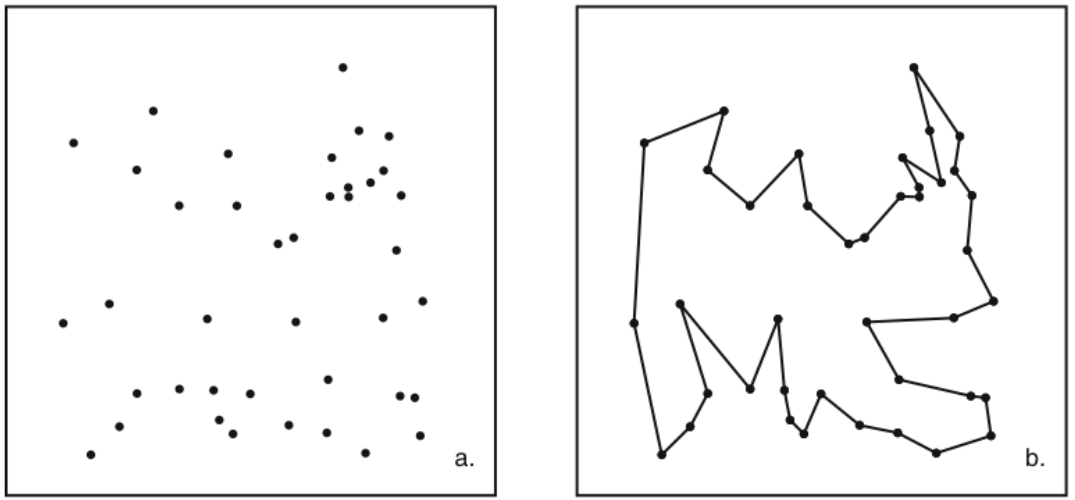
مثال

مسئله‌ی فروشنده‌ی دوره‌گرد

تعریف مسئله

TRAVELLING SALESMAN PROBLEM (TSP)

یک فروشنده‌ی دوره‌گرد می‌خواهد برای فروش کالاهایش
n شهر را یکی پس از دیگری ببیند و دوباره به شهر مبدأ بازگردد؛
با این شرط که طول مسیری که طی می‌کند، حداقل باشد.



مسئله‌ی فروشنده‌ی دوره‌گرد

تعریف انتزاعی مسئله

TRAVELLING SALESMAN PROBLEM (TSP)

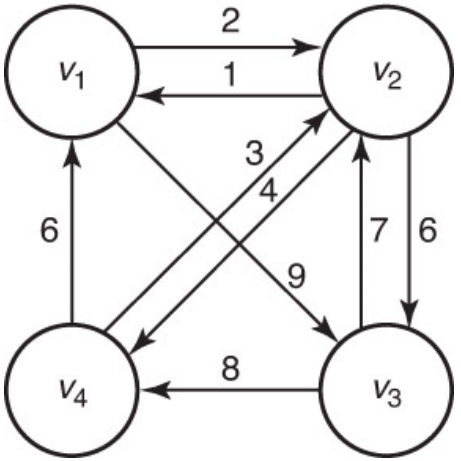
یافتن تور بهینه در یک گراف کامل با n رأس

تور (دور همیلتونی) *Tour (Hamiltonian Cycle)*	مسیری در گراف از یک رأس به خودش که از تمام رئوس گراف عبور کند.
تور بهینه *Optimal Tour*	توری با کوتاه‌ترین طول (طول مسیر = مجموع وزن یال‌های مسیر)

مسئله‌ی فروشنده‌ی دوره‌گرد

مثال

TRAVELLING SALESMAN PROBLEM (TSP)



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

$$length[v_1, v_2, v_3, v_4, v_1] = 22$$

$$length[v_1, v_3, v_2, v_4, v_1] = 26$$

$$length[v_1, v_3, v_4, v_2, v_1] = 21$$

$$\Downarrow$$

$$[v_1, v_3, v_4, v_2, v_1]$$ تور بهینه:

مسئله‌ی فروشنده‌ی دوره‌گرد

الگوریتم ناشیانه

TRAVELLING SALESMAN PROBLEM (TSP)

یافتن تور بهینه در یک گراف کامل با n رأس

تمام تورهای یک گراف کامل با n رأس را بررسی می‌کنیم و کوتاه‌ترین آنها را برمی‌گردانیم.

زمان اجرا: متناسب با تعداد تورهای یک گراف کامل با n رأس

$$(n-1)(n-2)\cdots(1) = (n-1)! \in \Theta(n!)$$

مسئله‌ی فروشنده‌ی دوره‌گرد

اصل بهینگی

TRAVELLING SALESMAN PROBLEM (TSP)

یافتن تور بهینه در یک گراف کامل با n رأس

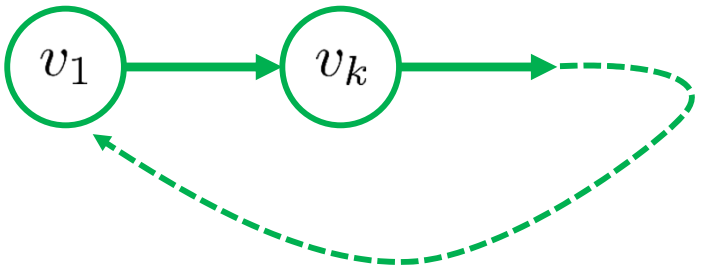
اگر v_k نخستین رأس پس از v_1 روی هر تور بهینه باشد،
زیرمسیر آن تور از v_k به v_1،
کوتاه‌ترین مسیر از v_k به v_1 است
که از هر کدام از رئوس دیگر فقط یک بار می‌گذرد.

$\Downarrow$

اصل بهینگی برقرار است.

$\Downarrow$

می‌توان از روش برنامه‌ریزی پویا برای حل مسئله استفاده کرد.



مسئله‌ی فروشنده‌ی دوره‌گرد

فرمول‌بندی مسئله

TRAVELLING SALESMAN PROBLEM (TSP)

V	مجموعه‌ی رأس‌های گراف
A	یک زیرمجموعه از رأس‌های گراف $A \subseteq V$
$D[v_i, A]$	طول کوتاه‌ترین مسیر از v_i به v_1 که از هر رأس A دقیقاً یک بار می‌گذرد.

$$D[v_i, A] = \min_{v_j \in A}(W[i,j] + D[v_j, A - \{v_j\}]), \quad A \neq \varnothing, i \neq 1, v_i \notin A$$

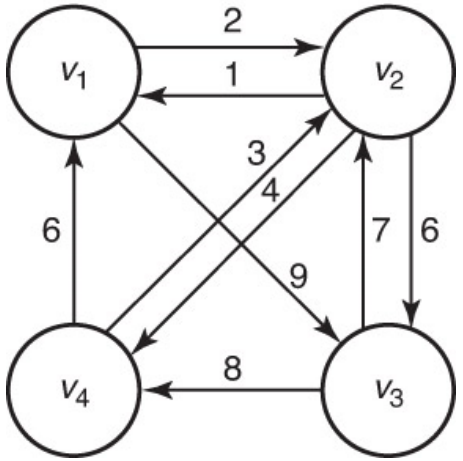
$$D[v_i, \varnothing] = W[i,1]$$

$$\text{minimum tour length} = D[v_1, V - \{v_1\}]$$

مسئله‌ی فروشنده‌ی دوره‌گرد

فرمول‌بندی مسئله : مثال

TRAVELLING SALESMAN PROBLEM (TSP)



$$V = \{v_1, v_2, v_3, v_4\}$$

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

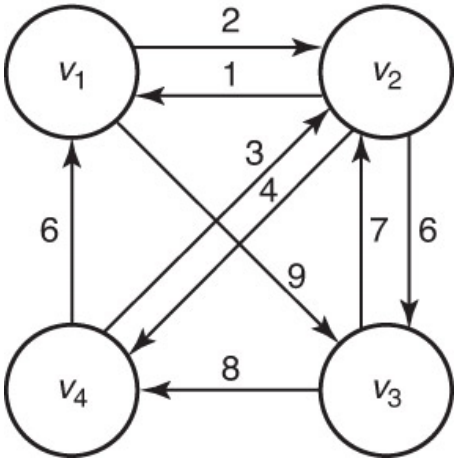
$$\begin{aligned} A &= \{v_3\} \\ &\Downarrow \\ D[v_2, A] &= length[v_2, v_3, v_1] \\ &= \infty \end{aligned}$$

$$\begin{aligned} A &= \{v_3, v_4\} \\ &\Downarrow \\ D[v_2, A] &= \min \left\{ \begin{array}{l} length[v_2, v_3, v_4, v_1] \\ length[v_2, v_4, v_3, v_1] \end{array} \right\} \\ &= \min(20, \infty) \\ &= 20 \end{aligned}$$

مسئله‌ی فروشنده‌ی دوره‌گرد

مثال (۱ از ۴)

TRAVELLING SALESMAN PROBLEM (TSP)



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

ابتدا برای زیرمجموعه‌های $k = 0$ عضوی از رئوس (غیر از مبدأ)

$$
\begin{aligned}
D[v_2, \varnothing] &= 1 \\
D[v_3, \varnothing] &= \infty \\
D[v_4, \varnothing] &= 6
\end{aligned}
$$

سپس برای زیرمجموعه‌های $k = 1$ عضوی از رئوس (غیر از مبدأ)

$$
\begin{aligned}
D[v_3, \{v_2\}] &= \min_{j:v_j \in \{v_2\}} (W[3, j] + D[v_j, \{v_2\} - \{v_j\}]) \\
&= W[3, 2] + D[v_2, \varnothing] \\
&= 7 + 1 = 8
\end{aligned}
$$

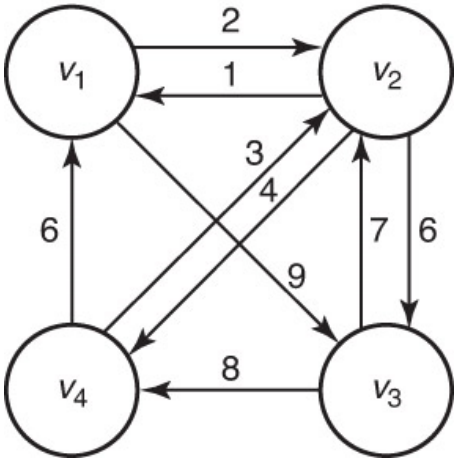
به طور مشابه:

$$
\begin{aligned}
D[v_4, \{v_2\}] &= 3 + 1 = 4 \\
D[v_2, \{v_3\}] &= 6 + \infty = \infty \\
D[v_4, \{v_3\}] &= \infty + \infty = \infty \\
D[v_2, \{v_4\}] &= 4 + 6 = 10 \\
D[v_3, \{v_4\}] &= 8 + 6 = 14
\end{aligned}
$$

مسئله‌ی فروشنده‌ی دوره‌گرد

مثال (۲ از ۴)

TRAVELLING SALESMAN PROBLEM (TSP)



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

سپس برای زیرمجموعه‌های $k = 2$ عضوی از رئوس (غیر از مبدأ)

$$
\begin{aligned}
D[v_4, \{v_2, v_3\}] &= \min_{j:v_j \in \{v_2, v_3\}} (W[4, j] + D[v_j, \{v_2, v_3\} - \{v_j\}]) \\
&= \min \left\{ \begin{array}{l} W[4,2] + D[v_2, \{v_3\}] \\ W[4,3] + D[v_3, \{v_2\}] \end{array} \right\} \\
&= \min(3 + \infty, \infty + 8) = \infty
\end{aligned}
$$

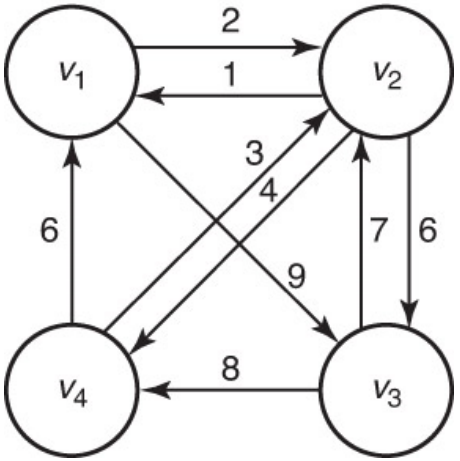
به طور مشابه:

$$
\begin{aligned}
D[v_3, \{v_2, v_4\}] &= \min(7 + 10, 8 + 4) = 12 \\
D[v_2, \{v_3, v_4\}] &= \min(6 + 14, 4 + \infty) = 20
\end{aligned}
$$

مسئله‌ی فروشنده‌ی دوره‌گرد

مثال (۳ از ۴)

TRAVELLING SALESMAN PROBLEM (TSP)



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

سپس برای زیرمجموعه‌های $k = 3$ عضوی از رئوس (غیر از مبدأ)

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

$$= \min \begin{Bmatrix} W[1,2] + D[v_2, \{v_3, v_4\}] \\ W[1,3] + D[v_3, \{v_2, v_4\}] \\ W[1,4] + D[v_4, \{v_2, v_3\}] \end{Bmatrix}$$

$$= \min(2 + 20, 9 + 12, \infty + \infty) = 21$$

طول تور بهینه:

زیرمجموعه‌های $V - \{v_1\}$ ←

$D =$

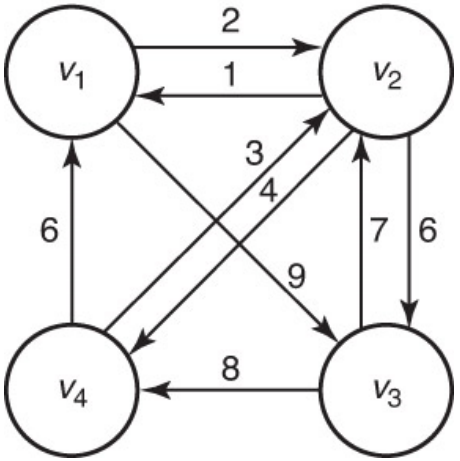
	$\{\varnothing\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
v_1								21
v_2	1		∞	10			20	
v_3	∞	8		14		12		
v_4	6	4	∞		∞			

جهت پر شدن جدول

مسئله‌ی فروشنده‌ی دوره‌گرد

مثال (۴ از ۴)

TRAVELLING SALESMAN PROBLEM (TSP)



3	4	2
$P[1, \{v_2, v_3, v_4\}]$	$P[3, \{v_2, v_4\}]$	$P[4, \{v_2\}]$

اندیس اولین رأس میانی $= P\,[1][\{v_2, v_3, v_4\}] = 3$

اندیس دومین رأس میانی $= P\,[3][\{v_2, v_4\}] = 4$

اندیس سومین رأس میانی $= P\,[4][\{v_2\}] = 2$

تور بهینه :

$$[v_1, v_3, v_4, v_2, v_1].$$

$D =$

	$\{\varnothing\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
v_1								21
v_2	1		∞	10			20	
v_3	∞	8		14		12		
v_4	6	4	∞		∞			

مسئله‌ی فروشنده‌ی دوره‌گرد

شبه کد

TRAVELLING SALESMAN PROBLEM (TSP)

TRAVELLING-SALESMAN-PROBLEM(W, n)

$D \leftarrow$ **array**$[1..n, \mathcal{P}(V-\{v_1\})]$
$P \leftarrow$ **array**$[1..n, \mathcal{P}(V-\{v_1\})]$

for $i \leftarrow 2$ **to** n **do**
    $D[i, \varnothing] \leftarrow W[i, 1]$

for $k \leftarrow 1$ **to** $n-2$ **do**
    for all subsets A $\subseteq V-\{v_1\}$ containing k vertices **do**
        for i such that $i \neq 1$ and $v_i \notin A$ **do**

$$D[i, A] \leftarrow \min_{v_j \in A}(W[i, j] + D[v_j, A - \{v_j\}])$$

$P[i, A] \leftarrow$ value of j that gave the minimum

$$D[v_1, V - \{v_1\}] \leftarrow \min_{2 \le j \le n}(W[1, j] + D[v_j, V - \{v_1\}])$$

$P[v_1, V - \{v_1\}] \leftarrow$ value of j that gave the minimum

$minlength \leftarrow D[v_1, V - \{v_1\}]$

return $minlength$

مسئله‌ی فروشنده‌ی دوره‌گرد

تحلیل زمان اجرا

TRAVELLING SALESMAN PROBLEM (TSP)

TRAVELLING-SALESMAN-PROBLEM(W, n)

$D \leftarrow$ **array**$[1..n, \mathcal{P}(V-\{v_1\})]$
$P \leftarrow$ **array**$[1..n, \mathcal{P}(V-\{v_1\})]$

for $i \leftarrow 2$ **to** n **do**
  $D[i, \varnothing] \leftarrow W[i, 1]$

زمان مصرفی اصلی

for $k \leftarrow 1$ **to** $n-2$ **do**
  for all subsets A $\subseteq V-\{v_1\}$ containing k vertices **do**
    for i such that $i \neq 1$ and $v_i \notin A$ **do**

$$D[i,A] \leftarrow \min_{v_j \in A}(W[i,j] + D[v_j, A-\{v_j\}])$$
$$P[i,A] \leftarrow \text{value of } j \text{ that gave the minimum}$$

$$D[v_1, V-\{v_1\}] \leftarrow \min_{2 \le j \le n}(W[1,j] + D[v_j, V-\{v_1\}])$$
$$P[v_1, V-\{v_1\}] \leftarrow \text{value of } j \text{ that gave the minimum}$$

$minlength \leftarrow D[v_1, V-\{v_1\}]$

return $minlength$

حلقه اول

یک مجموع برای k

$$\sum_{k=1}^{n-2} \dots$$

حلقه دوم

به تعداد زیرمجموعه‌های شامل k رأس:

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

حلقه سوم

بررسی $n-1-k$ رأس
برای هر کدام از این رأس‌ها k عمل جمع

برحسب عمل اصلی: **جمع**

مسئله‌ی فروشنده‌ی دوره‌گرد

تحلیل زمان اجرا : ادامه

TRAVELLING SALESMAN PROBLEM (TSP)

TRAVELLING-SALESMAN-PROBLEM(W, n)

$D \leftarrow$ **array**$[1..n, \mathcal{P}(V-\{v_1\})]$
$P \leftarrow$ **array**$[1..n, \mathcal{P}(V-\{v_1\})]$

for $i \leftarrow 2$ **to** n **do**
  $D[i, \varnothing] \leftarrow W[i, 1]$

زمان مصرفی اصلی

for $k \leftarrow 1$ **to** $n-2$ **do**
  for all subsets A $\subseteq V-\{v_1\}$ containing k vertices **do**
    for i such that $i \neq 1$ and $v_i \notin A$ **do**

$$D[i,A] \leftarrow \min_{v_j \in A}(W[i,j] + D[v_j, A-\{v_j\}])$$
$$P[i,A] \leftarrow \text{value of } j \text{ that gave the minimum}$$

$$D[v_1, V-\{v_1\}] \leftarrow \min_{2 \le j \le n}(W[1,j] + D[v_j, V-\{v_1\}])$$
$$P[v_1, V-\{v_1\}] \leftarrow \text{value of } j \text{ that gave the minimum}$$

$minlength \leftarrow D[v_1, V-\{v_1\}]$

return $minlength$

$$\begin{aligned} T(n) &= \sum_{k=1}^{n-2}(n-1-k)k\binom{n-1}{k} \\ &= (n-1)\sum_{k=1}^{n-2}k\binom{n-2}{k} \\ &= (n-1)(n-2)2^{n-3} \end{aligned}$$

$$\Rightarrow T(n) \in \Theta(n^2 2^n)$$

دو فرمول مورد استفاده :

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

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

مسئله‌ی فروشنده‌ی دوره‌گرد

تحلیل فضای مصرفی

TRAVELLING SALESMAN PROBLEM (TSP)

TRAVELLING-SALESMAN-PROBLEM(W, n)

$D \leftarrow$ **array**$[1..n, \mathcal{P}(V-\{v_1\})]$
$P \leftarrow$ **array**$[1..n, \mathcal{P}(V-\{v_1\})]$ فضای مصرفی اصلی

for $i \leftarrow 2$ **to** n **do**
  $D[i, \varnothing] \leftarrow W[i, 1]$

for $k \leftarrow 1$ **to** $n-2$ **do**
  for all subsets A $\subseteq V-\{v_1\}$ containing k vertices **do**
    for i such that $i \neq 1$ and $v_i \notin A$ **do**

$$D[i, A] \leftarrow \min_{v_j \in A}(W[i, j] + D[v_j, A - \{v_j\}])$$
$$P[i, A] \leftarrow \text{value of } j \text{ that gave the minimum}$$

$$D[v_1, V - \{v_1\}] \leftarrow \min_{2 \le j \le n}(W[1, j] + D[v_j, V - \{v_1\}])$$
$$P[v_1, V - \{v_1\}] \leftarrow \text{value of } j \text{ that gave the minimum}$$

$$minlength \leftarrow D[v_1, V - \{v_1\}]$$

return *minlength*

مجموعه‌ی $V - \{v_1\}$ دارای $n - 1$ عضو و 2^{n-1} زیرمجموعه است.

$$\begin{aligned} S(n) &= 2 \times n2^{n-1} \\ &= n2^n \end{aligned}$$

$$\Rightarrow S(n) \in \Theta(n2^n)$$

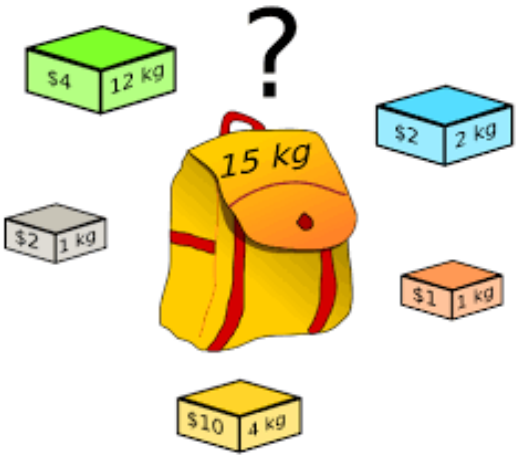
مسئله‌ی کوله‌پشتی 0/1

0/1- KNAPSACK

n قطعه با وزن‌ها و ارزش‌های مختلف داریم.
یک کوله‌پشتی با ظرفیت W موجود است.

می‌خواهیم قطعه‌هایی را انتخاب کنیم که

- کوله‌پشتی تا حد امکان **پر** شود، و
- مجموع ارزش کل ظرف **ماکزیمم** شود.



مسئله‌ی کوله‌پشتی 0/1

0/1-KNAPSACK

$$S = \{item_1, item_2, \ldots, item_n\}$$
$$w_i = weight(item_i)$$
$$p_i = value(item_i)$$
$$W = \text{total weight of knapsack}$$

$$\text{select } A \subseteq S$$

$$\text{maximize} \sum_{item_i \in A} p_i x_i$$

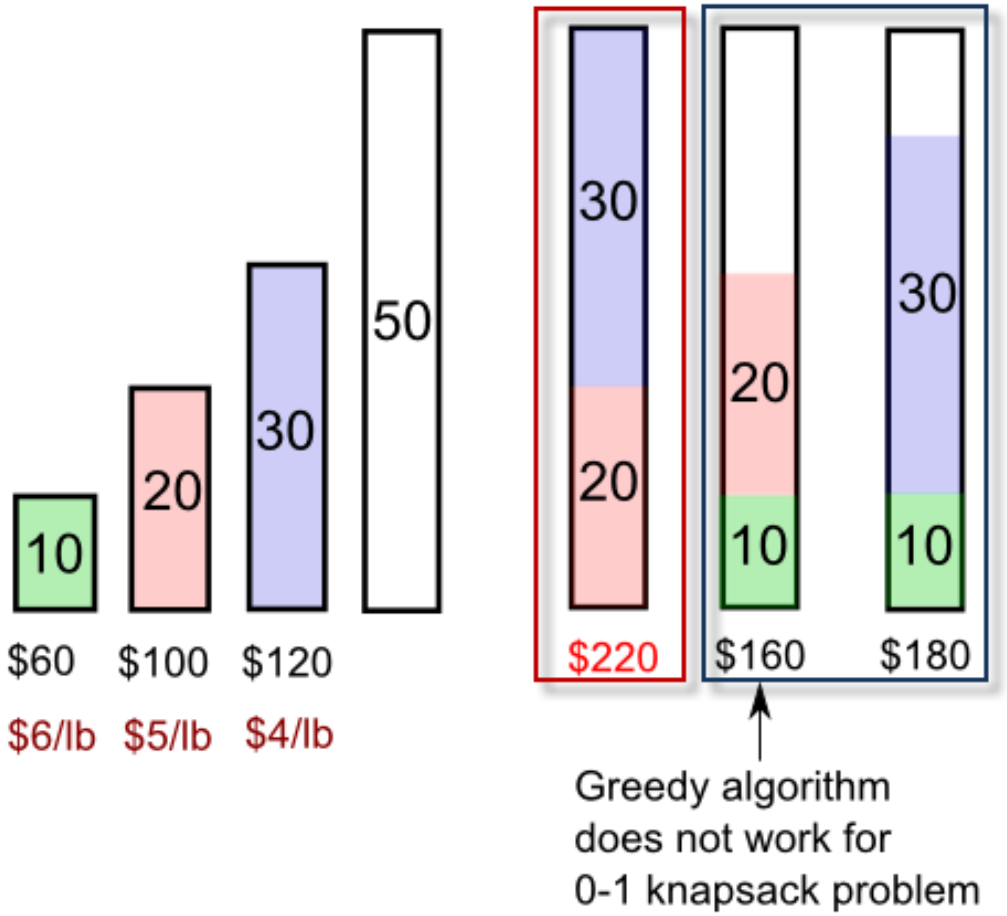
$$\text{subject to} \sum_{item_i \in A} w_i x_i \leq W$$

$$\text{where} \quad x_i \in \{0,1\}, \quad 1 \leq i \leq n$$

مسئله‌ی کوله‌پشتی 0/1

مثال

0/1- KNAPSACK



مسئله‌ی کوله‌پشتی 0/1

الگوریتم ناشیانه

0/1-KNAPSACK

در مسئله‌ی کوله‌پشتی 0/1

همه‌ی زیرمجموعه‌های n قطعه را بررسی می‌کنیم
و آنکه بیشترین منفعت را دارد (بدون تجاوز از قید حداکثر وزن کوله‌پشتی) برمی‌گردانیم.

زمان اجرا: متناسب با تعداد زیرمجموعه‌های یک مجموعه‌ی n عضوی

$$\Theta(2^n)$$

مسئله‌ی کوله‌پشتی 0/1

اصل بهینگی

0/1-KNAPSACK

در مسئله‌ی کوله‌پشتی 0/1

فرض می‌کنیم A یک زیرمجموعه‌ی بهینه از n قطعه باشد.

دو حالت وجود دارد:

- $item_n \notin A$: در این صورت A مساوی است با زیرمجموعه‌ای بهینه از $n-1$ قطعه‌ی نخست.
- $item_n \in A$: در این صورت منفعت کل حاصل از قطعات موجود در A مساوی است

با ارزش قطعه‌ی n به‌اضافه‌ی منفعت بهینه‌ای است که از زیرمجموعه‌ای از $n-1$ قطعه‌ی نخست به‌دست می‌آید (با رعایت این قید که وزن کل آنها نباید از $W - w_n$ بیشتر شود).

⇓

اصل بهینگی برقرار است.

⇓

می‌توان از روش برنامه‌ریزی پویا برای حل مسئله استفاده کرد.

مسئله‌ی کوله‌پشتی 0/1

فرمول‌بندی مسئله

0/1-KNAPSACK

W	حداکثر گنجایش وزنی کوله‌پشتی
n	تعداد قطعه‌ها
p_i	ارزش قطعه i ام
w_i	وزن قطعه i ام
$P[i,w]$	منفعت بهینه‌ی حاصل از انتخاب i قطعه‌ی نخست (با رعایت قید عدم تجاوز وزن کل آنها از W)

$$i > 0, \quad w > 0$$

$$P[i,w] = \begin{cases} \max(P[i-1,w], p_i + P[i-1, w-w_i]) & , w_i \le w \\ P[i-1,w] & , w_i > w \end{cases}$$

$$P[i,0] = P[0,w] = 0$$

حداکثر منفعت = $P[n,W]$

$$P[0..n, 0..W]$$

مسئله‌ی کوله‌پشتی 0/1

شبه‌کد

0/1-KNAPSACK

$$P[i,w] = \begin{cases} \max(P[i-1,w], p_i + P[i-1, w-w_i]) & , w_i \le w \\ P[i-1,w] & , w_i > w \end{cases}$$

$$P[i,0] = P[0,w] = 0$$

```
0/1-KNAPSACK(p, w, W)
    n ← length(p)
    P ← array[0..n, 0..W]
    for w ← 0 to W do
        P[0, w] ← 0
    for i ← 0 to n do
        P[i, 0] ← 0
    for i ← 1 to n do
        for w ← W downto w_i do
            if P[i − 1, w − w_i] + p_i > P[i − 1, w] then
                P[i, w] ← P[i − 1, w − w_i] + p_i
            else P[i, w] ← P[i − 1, w]
    return P[n, W]
```

حداکثر منفعت = $P[n, W]$

مسئله‌ی کوله‌پشتی 0/1

تحلیل زمان اجرا

0/1-KNAPSACK

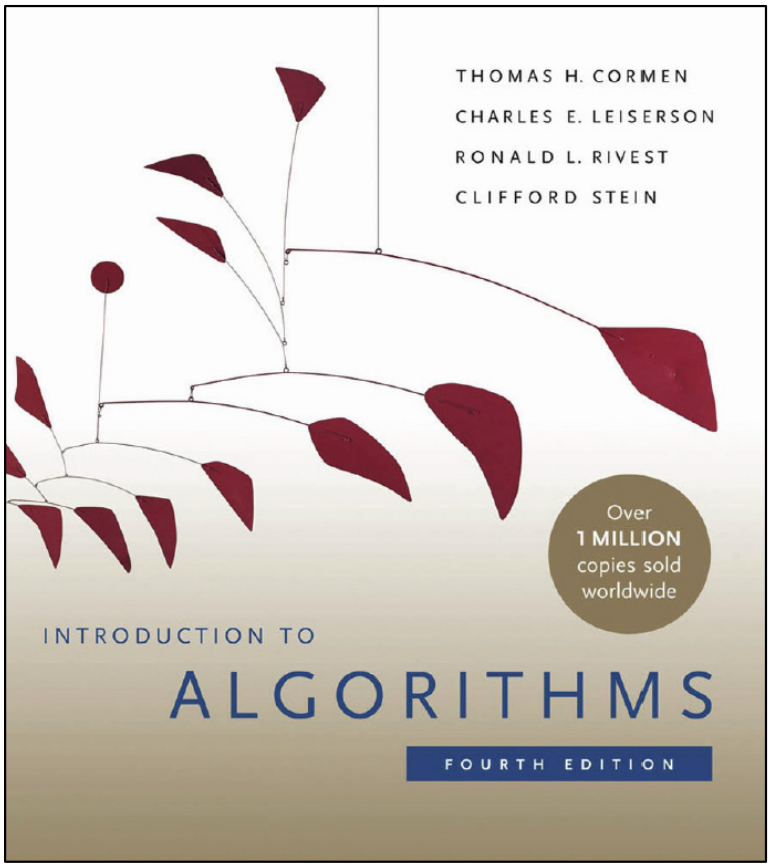
```
0/1-KNAPSACK(p, w, W)
    n ← length(p)
    P ← array[0..n, 0..W]
    for w ← 0 to W do
        P[0, w] ← 0
    for i ← 0 to n do
        P[i, 0] ← 0
    for i ← 1 to n do
        for w ← W downto w_i do
            if P[i − 1, w − w_i] + p_i > P[i − 1, w] then
                P[i, w] ← P[i − 1, w − w_i] + p_i
            else P[i, w] ← P[i − 1, w]
    return P[n, W]
```

تعداد عناصری که محاسبه می‌شوند: $nW \in \Theta(nW)$

چون رابطه‌ای بین n و W وجود ندارد، نمی‌توان فرض کرد که این فرمول خطی است.

(برای یک n خاص می‌توان W به‌دلخواه بزرگ داشت، مثلاً: $W = n!$ که در این صورت زمان الگوریتم حاصل از الگوریتم ناشیانه بدتر است.)

مرجع



T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein,
Introduction to Algorithms,
4th Edition, MIT Press, 2022.

Chapter 14

14 Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. ("Programming" in this context refers to a tabular method, not to writing computer code.) As we saw in Chapters 2 and 4, divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

Dynamic programming typically applies to *optimization problems*. Such problems can have many possible solutions. Each solution has a value, and you want to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

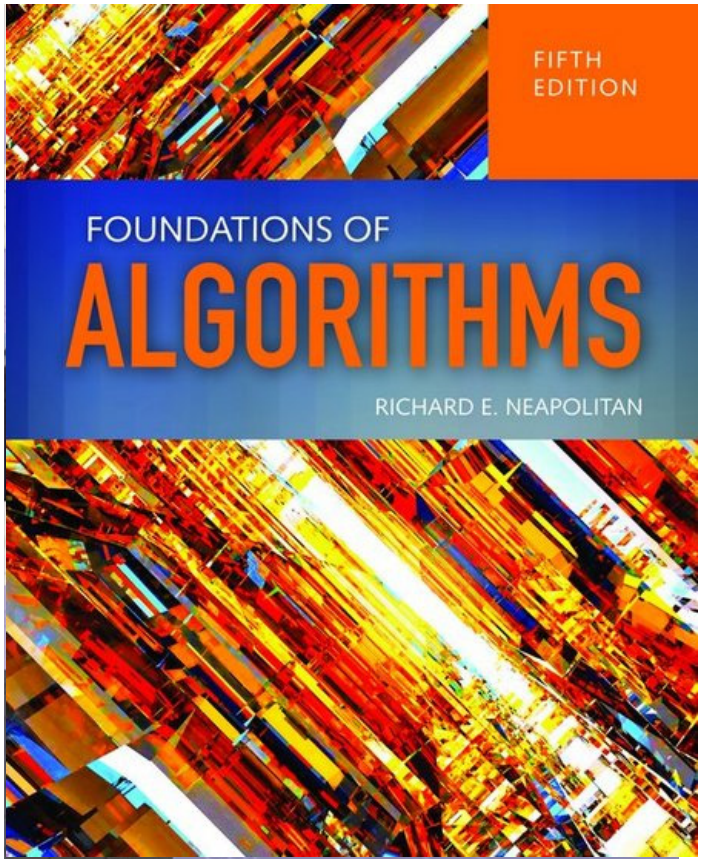
To develop a dynamic-programming algorithm, follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If you need only the value of an optimal solution, and not the solution itself, then you can omit step 4. When you do perform step 4, it often pays to maintain additional information during step 3 so that you can easily construct an optimal solution.

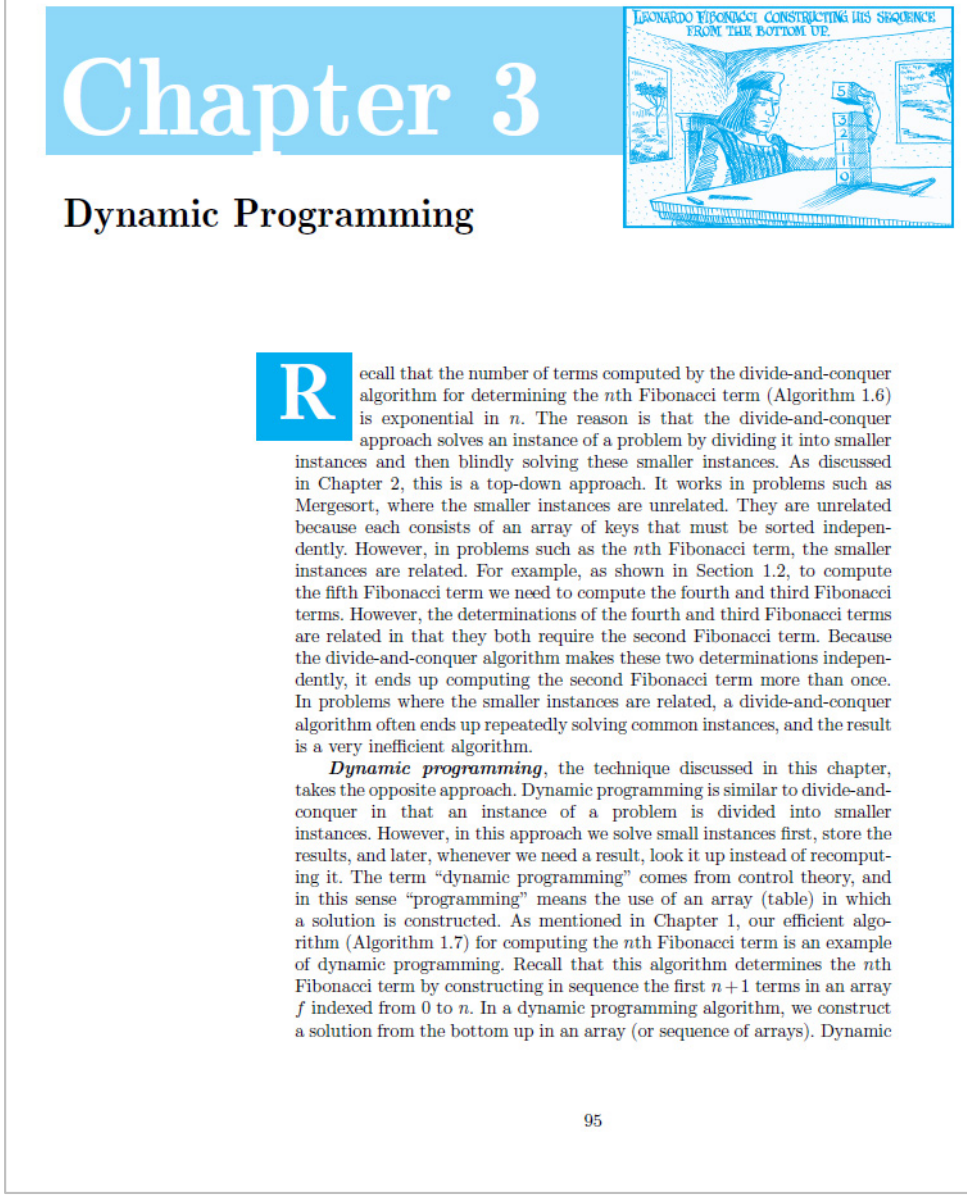
The sections that follow use the dynamic-programming method to solve some optimization problems. Section 14.1 examines the problem of cutting a rod into

مرجع



R.E. Neapolitan,
Foundations of Algorithms,
5th Edition, Jones and Bartlett Publishers, 2015.

Chapter 3



Chapter 3

Dynamic Programming

Recall that the number of terms computed by the divide-and-conquer algorithm for determining the nth Fibonacci term (Algorithm 1.6) is exponential in n. The reason is that the divide-and-conquer approach solves an instance of a problem by dividing it into smaller instances and then blindly solving these smaller instances. As discussed in Chapter 2, this is a top-down approach. It works in problems such as Mergesort, where the smaller instances are unrelated. They are unrelated because each consists of an array of keys that must be sorted independently. However, in problems such as the nth Fibonacci term, the smaller instances are related. For example, as shown in Section 1.2, to compute the fifth Fibonacci term we need to compute the fourth and third Fibonacci terms. However, the determinations of the fourth and third Fibonacci terms are related in that they both require the second Fibonacci term. Because the divide-and-conquer algorithm makes these two determinations independently, it ends up computing the second Fibonacci term more than once. In problems where the smaller instances are related, a divide-and-conquer algorithm often ends up repeatedly solving common instances, and the result is a very inefficient algorithm.

Dynamic programming, the technique discussed in this chapter, takes the opposite approach. Dynamic programming is similar to divide-and-conquer in that an instance of a problem is divided into smaller instances. However, in this approach we solve small instances first, store the results, and later, whenever we need a result, look it up instead of recomputing it. The term "dynamic programming" comes from control theory, and in this sense "programming" means the use of an array (table) in which a solution is constructed. As mentioned in Chapter 1, our efficient algorithm (Algorithm 1.7) for computing the nth Fibonacci term is an example of dynamic programming. Recall that this algorithm determines the nth Fibonacci term by constructing in sequence the first $n+1$ terms in an array f indexed from 0 to n. In a dynamic programming algorithm, we construct a solution from the bottom up in an array (or sequence of arrays). Dynamic

95