برنامه نویسی پویا
Dynamic Programming

مطالب مورد بحث

- مقدمه
- بررسی چند مساله نمونه
  - زنجیره ضرب ماتریس ها
  - ویرایش رشته ها
  - فروشنده دوره گرد
  - ترکیب k از n
  - کوله پشتی صفر و یک

مقدمه

- یادآوری روش تقسیم و حل:
  - برای حل یک مساله، آن را به تعدادی زیرمساله کوچکتر تقسیم می کردیم. آن زیرمسائل را حل کرده و با ترکیب جواب آنها، جواب مساله اصلی را بدست می آوردیم.
    - تقسیم و حل، یک روال بالا به پایین است.
  - در تقسیم و حل، اگر زیرمسائل بهم مربوط باشند، معمولا کارآیی خیلی پایین می آید
    - چون بطور blind، به حل تمام زیرمسائل می پردازیم و ممکن است یک زیرمساله را بارها حل کنیم. مثال: الگوریتم فیبوناتچی مبتنی بر روش تقسیم و حل، با مرتبه زمانی نمایی
  - اگر زیرمسائل بهم مرتبط نباشند، معمولا الگوریتم قابل قبولی بدست می آید. مثال: جستجوی دودویی

مقدمه

- در روش پویا
  - برای حل یک مساله باید تعدادی زیرمساله کوچکتر را حل کنیم. برای حل هریک از این زیرمسائل نیز باید تعدادی زیرمساله کوچکتر را حل کنیم. (چند سطح از مسائل)
  - از پایین ترین سطح شروع می کنیم و تمام زیرمسائل آن را حل می کنیم و جواب آنها را ذخیره می کنیم.
  - تا وقتی مساله اصلی حل نشده:
    - به سطح بالاتر می رویم و تمام مسائل آن را حل می کنیم.
    - در حل این مسائل، از جواب مسائل تمام سطح های قبل استفاده می کنیم (عدم نیاز به محاسبه مجدد و حل مکرر).
  - نهایتا به مساله اصلی می رسیم و با حل آن کار تمام می شود.
  - یک روش پایین با بالا است. چرا؟؟؟؟؟؟؟؟؟

مقدمه

- ویژگی های اغلب مسائلی که با این روش حل می شوند
  - برای حل یک نمونه از مساله باید تعدادی نمونه دیگر با اندازه کوچکتر را حل کرد. (یک خاصیت بازگشتی در تعریف مساله می توان مشاهده کرد)
  - معمولا موضوع بهینه سازی مطرح است. یعنی باید بهترین جواب از بین جواب های ممکن را پیدا کنیم.
  - چنانچه موضوع بهینه سازی مطرح باشد، باید اصل بهینگی صدق کند.

مقدمه

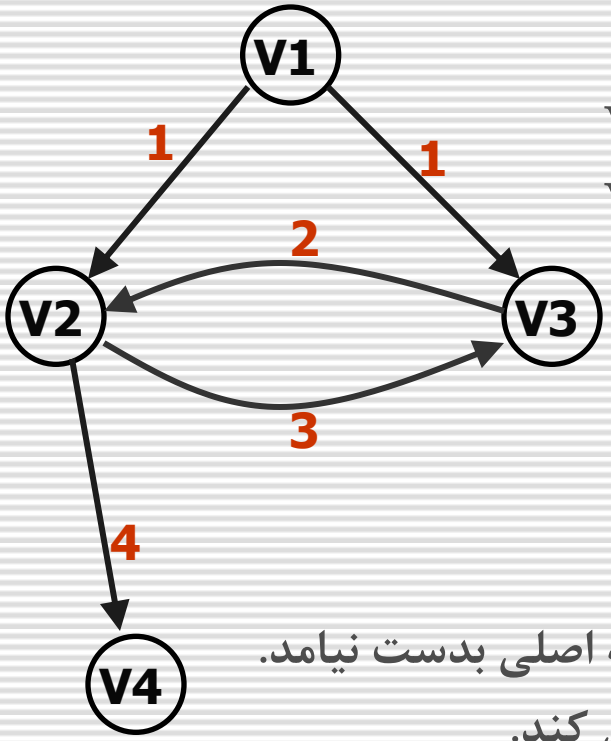
- اصل بهینگی (principle of optimality) صدق می کند.
  - اگر بدست آوردن جواب بهینه مساله اصلی، مستلزم بدست آوردن جواب بهینه هر یک از زیرمسائل باشد، آنگاه اصل بهینگی در مورد مساله صدق می کند.
  - به بیان دیگر، جواب بهینه مساله اصلی، شامل جواب بهینه زیرمسائل باشد.

مقدمه

- مثال: پیدا کردن کوتاهترین مسیر بین دو شهر A و C که از شهر B بگذرد.
  - باید ابتدا کوتاهترین مسیر بین A و B را پیدا کرده و سپس کوتاهترین مسیر بین B و C را نیز پیدا کنیم و با ترکیب آنها کوتاهترین مسیر بین A و C را بدست آوریم.
  - اصل بهینگی صدق می کند
  - یک توجیه دیگر: اگر X، جواب بهینه، (یعنی کوتاهترین مسیری باشد که از A شروع می شود و از B می گذرد و به C می رسد) باشد، آنگاه الزاما قسمتی از X که از A شروع می شود و به B می رسد، کوتاهترین مسیر بین A و B است.

مقدمه

- مثال: پیدا کردن بلندترین مسیر بدون حلقه از V1 به V4
- جواب مساله: [V1, V3, V2, V4]
- اگر مساله را به دو زیرمساله تقسیم کنیم
  - پیدا کردن بلندترین مسیر بدون حلقه از V1 به V3
  - پیدا کردن بلندترین مسیر بدون حلقه از V3 به V4
  - جواب زیرمساله اول: [V1, V2, V3]
  - جواب زیرمساله دوم: [V3, V2, V4]
  - ترکیب جوابهای بهینه زیرمسائل:
    - [V1, V2, V3, V2, V4]
    - حلقه دارد ← این جواب مساله اصلی نیست.
  - با ترکیب جواب بهینه زیرمسائل، جواب بهینه مساله اصلی بدست نیامد.
  - در نتیجه اصل بهینگی در مورد این مساله صدق نمی کند.



مطالب مورد بحث

- زنجیره ضرب ماتریس ها
- ویرایش رشته ها
- فروشنده دوره گرد
- ترکیب k از n
- کوله پشتی صفر و یک

زنجیره ضرب ماتریس ها

- ضرب چند ماتریس

$$M = M1 * M2 * M3$$

- دو ماتریس در صورتی قابل ضرب هستند که تعداد ستونهای ماتریس اول با تعداد سطرهای ماتریس دوم برابر باشد.
- ضرب ماتریس ها شرکت پذیر است.
- فقط می توانیم ماتریس های مجاور را در هم ضرب کنیم.
- اگر M1 دارای i سطر و j ستون و M2 دارای j ستون و k سطر باشد، آنگاه ماتریس M1*M2، دارای i سطر و k ستون خواهد بود.

زنجیره ضرب ماتریس ها

- مرتبه زمانی الگوریتم معمولی برای ضرب دو ماتریس n*n برابر $O(n^3)$ می باشد ← تعداد ضرب ها زیاد
- اگر بخواهیم زنجیره ای از ماتریس ها را در هم ضرب کنیم، می توانیم این کار را به ترتیب های مختلفی انجام دهیم
- مثال: M1*M2*M3
  - ترتیب اول: M1*(M2*M3)
  - ترتیب دوم: (M1*M2)*M3
- در ترتیب های مختلف، تعداد اعمال ضرب صورت گرفته ممکن است تفاوت قابل توجهی داشته باشند.

زنجیره ضرب ماتریس ها

- مثال: محاسبه M1*M2*M3*M4

M1	M2	M3	M4
10*20	20*50	50*1	1*100

- یک ترتیب ممکن M1 * (M2 * (M3 *M4))
  - X = M3*M4 ← تعداد ضرب : 50*1*100 = 5000 حاصل: ماتریس 50*100
  - Y = M2 * (X) ← تعداد ضرب: 20*50*100 = 100000 حاصل: ماتریس 20*100
  - M1 * Y ← تعداد ضرب: 10*20*100 = 20000 حاصل: ماتریس 10*100
  - مجموع تعداد ضرب های انجام شده: 125000 ضرب
- یک ترتیب ممکن (M1 * (M2 * M3)) * M4
  - X = M2*M3 ← تعداد ضرب : 20*50*1 = 1000 حاصل: ماتریس 20*1
  - Y = M1 * (X) ← تعداد ضرب: 10*20*1 = 200 حاصل: ماتریس 10*1
  - Y*M4 ← تعداد ضرب: 10*1*100 = 1000 حاصل: ماتریس 10*100
  - مجموع تعداد ضرب های انجام شده: 2200 ضرب

زنجیره ضرب ماتریس ها

$M = M_1 * M_2 * \dots * M_n$

$M = M_1 * (M_2 * \dots * M_n)$

$M = (M_1 * M_2) * (M3 * \dots * M_n)$

$M = (M_1 * M_2 * M_3) * (M4 * \dots * M_n)$

...

$M = (M_1 * \dots * M_{n-1}) * M_n$

$M = (M_1 * M_2 * \dots * M_i) * (M_{i+1} * M_{i+2} * \dots M_n)$

تعداد ترتیب های مختلف:

$$F(n) = \sum_{1}^{n-1} F(i).F(n-i)$$

زنجیره ضرب ماتریس ها

- تعداد ترتیب های مختلف: اعداد کاتالان
  - رشد بسیار سریع
  - F(5)=14
  - F(10)=4862
  - F(15)=2674460

زنجیره ضرب ماتریس ها

- صورت مساله: تعیین ترتیب بهینه ضرب زنجیره ای n ماتریس
- یعنی ترتیبی که اگر ماتریس ها را بر اساس آن در هم ضرب کنیم، تعداد اعمال ضرب ساده (ضرب عنصر در عنصر)، حداقل ممکن باشد.
- آیا اصل بهینگی صدق می کند؟ بله
- بطور مثال، اگر ترتیب بهینه M1*M2*M3*M4 به شکل

(M1 * ((M2 * (M3 * M4))

باشد، آنگاه ترتیب بهینه M2*M3*M4 به شکل زیر می باشد.

M2 * (M3 * M4)

جواب بهینه مساله اصلی، شامل جوابهای بهینه زیرمسائل می باشد.

زنجیره ضرب ماتریس ها

- گام اول در حل مسائل به روش پویا
  - تشخیص یک خاصیت بازگشتی در مساله
  - بعبارت دیگر، تشخیص رابطه بین جواب مسائل با جواب زیرمسائل کوچکتر

زنجیره ضرب ماتریس ها

- M1*M2*...*Mn
- برای محاسبه ضرب فوق، یعنی ضرب n ماتریس در هم
- تعدادی ضرب ماتریس در ماتریس داریم
- تعدادی ضرب ماتریس در ماتریس در ماتریس داریم
- تعدادی ضرب ماتریس در ماتریس در ماتریس در ماتریس داریم.
- ....
- در تمام این مراحل، ترتیب ضرب ماتریسها، باید بهینه باشد.

زنجیره ضرب ماتریس ها

- راه حل:
  - ابعاد ماتریس M_i : $r_{i-1} * r_i$ که $i=1,\ldots,n$
  - $m_{i,j}$: تعداد حداقل اعمال ضرب لازم برای محاسبه

$$M_i * M_{i+1} * \ldots * M_j \qquad , 1 \le i \le j \le n$$

اصلاح

  - اگر $i=j$ باشد $m_{i,j}=0$
  - در غیر اینصورت

$$m_{i,j} = \min_{i \le k \le j-1} (m_{i,k} + m_{k+1,j} + r_{i-1} * r_k * r_j)$$

- باید $m_{1,n}$ را بدست آوریم (حداقل تعداد اعمال ضرب لازم)

زنجیره ضرب ماتریس ها

- از یک آرایه دو بعدی برای ذخیره $m_{i,j}$ استفاده می کنیم.
- مراحل انجام کار (حل زیرمسائل کوچکتر)
  - مرحله 0: تعداد 0 عمل ضرب ماتریسی ← قرار دادن صفر در $m_{i,i}$ که $i=1,...,n$
  - مرحله 1: تعداد 1 عمل ضرب ماتریسی ← محاسبه $m_{i,i+1}$ ، $i=1,..,n-1$
  - مرحله 2: تعداد 2 عمل ضرب ماتریسی ← محاسبه $m_{i,i+2}$ ، $i=1,..,n-2$
  - ...
  - مرحله n-1: تعداد n-1 عمل ضرب ماتریسی ← محاسبه $m_{1,n}$

زنجیره ضرب ماتریس ها

- مثال: محاسبه M1*M2*M3*M4

M1	M2	M3	M4
10*20	20*50	50*1	1*100

m

0	10000	1200	2200
	0	1000	3000
		0	5000
			0

حداقل تعداد ضرب های لازم

زنجیره ضرب ماتریس ها

- الگوریتم

```
int minCost(int n) {
  for(i=1; i<=n; i++)  m[i][i]=0;

  for(step=1; step<n; step++)
    for(i=1; i<=n-step; i++) {
      j = i+step;
      m[i][j] = min{ m[i][k] + m[k+1][j] + r[i-1]*r[k]*r[j] };
    }                 i ≤ k ≤ j-1
  return m[1][n];
}
```

زنجیره ضرب ماتریس ها

- یک حلقه برای m[i][i]=0 ← O(n)
- دوحلقه تودرتو
  - در داخل حلقه مینیمم گیری می شود؟ بین چند عنصر؟
  - بین i + 1 – (j-1) عنصر
  - j-1-i+1 = i+step-1-i+1 = step
  - پس در داخل حلقه بین step عنصر مینیمم گیری می شود.

$$\sum_{step=1}^{n-1}[(n-step)*step]=\frac{n*(n-1)(n+1)}{6}$$

  - مرتبه زمانی حلقه های متداخل: $O(n^3)$

زنجیره ضرب ماتریس ها

- مرتبه زمانی کل الگوریتم $O(n^3)$
- الگوریتم قبل، فقط تعداد مینیمم اعمال ضرب لازم را محاسبه می کرد، درحالیکه هدف ما پیدا کردن ترتیب بهینه است. با تغییر کمی می توان این قابلیت را به الگوریتم افزود که علاوه بر محاسبه تعداد بهینه، ترتیب بهینه را هم تعیین نماید.

مطالب مورد بحث

- زنجیره ضرب ماتریس ها
- **ویرایش رشته ها**
- فروشنده دوره گرد
- ترکیب k از n
- کوله پشتی صفر و یک

ویرایش رشته ها

- دو رشته X و Y داریم

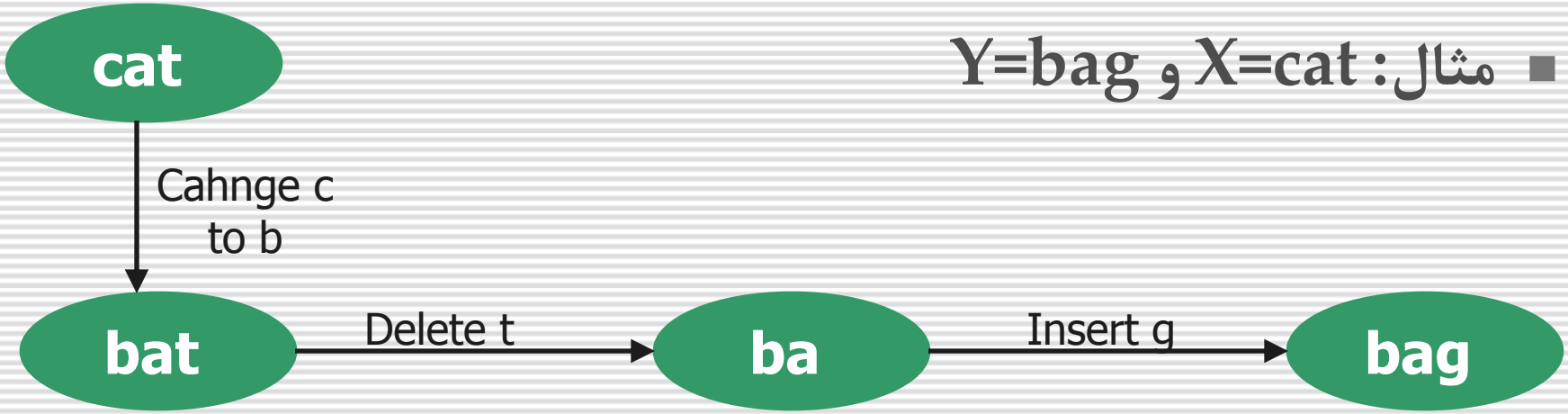
$$X = x_1\ x_2\ x_3\ \dots\ x_n$$

$$Y = y_1\ y_2\ y_3\ \dots\ y_m$$

- هدف: با اجرای یک دنباله از اعمال ویرایشی، رشته X را به رشته Y تبدیل کنید، بطوریکه مجموع هزینه این اعمال حداقل گردد.
- اعمال ویرایشی مجاز
  - delete: حذف یک کاراکتر از رشته
  - insert: درج یک کاراکتر در رشته
  - change: تعویض یک کاراکتر در رشته با یک کاراکتر دیگر
  - هر یک از این اعمال هزینه ای دارد.

ویرایش رشته ها

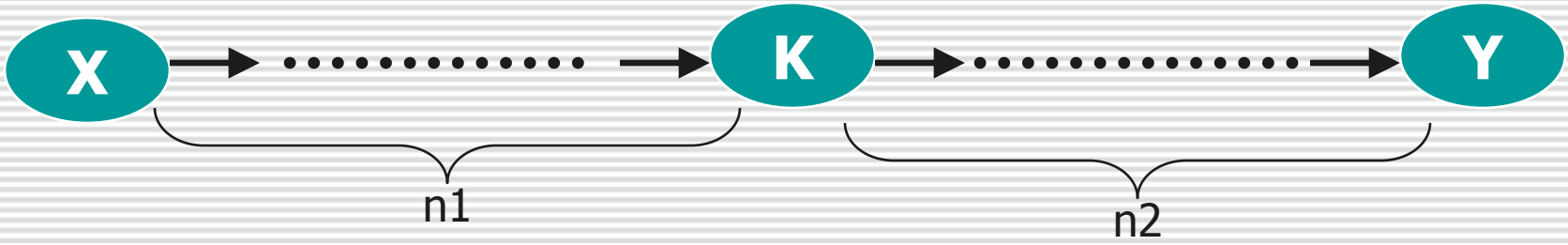
- مثال: X=cat و Y=bag



- آیا این راه حل بهینه است؟
  - به هزینه هر یک از اعمال ویرایشی بستگی دارد.
  - $D(x_i)$: هزینه حذف کاراکتر x_i
  - $I(x_i)$: هزینه درج کاراکتر x_i
  - $C(x_i,y_j)$: هزینه تعویض کاراکتر x_i با کاراکتر y_j

ویرایش رشته ها

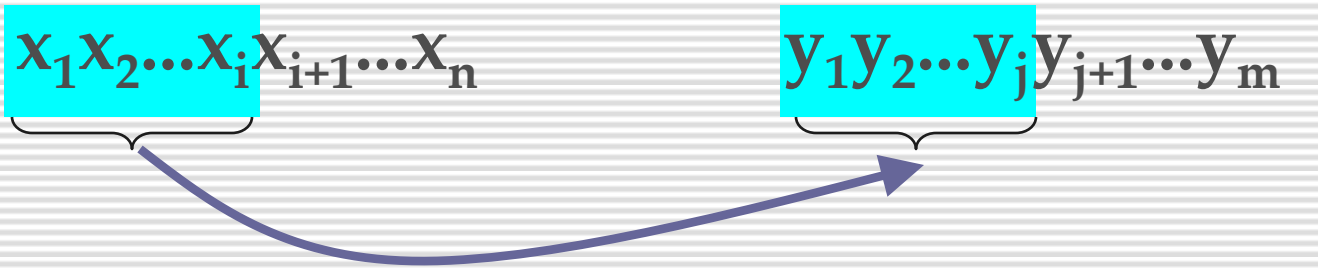
- اصل بهینگی؟



- فرض کنیم دنباله ای شامل n عمل ویرایشی داریم که با حداقل هزینه، X را به Y تبدیل می کند. می توانیم فرض کنیم که در حین این کار ابتدا با n1 عمل ویرایشی اول، X به K تبدیل می شود و سپس با استفاده از n2 عمل ویرایشی دوم، K به Y تبدیل می شود. در اینصورت واضح است که n1 عمل ویرایشی اول، برای تبدیل X به K، یک جواب بهینه می باشند(حداقل هزینه را دارد) و n2 عمل ویرایشی دوم نیز، برای تبدیل K به Y، یک جواب بهینه می باشند. (n=n1+n2)

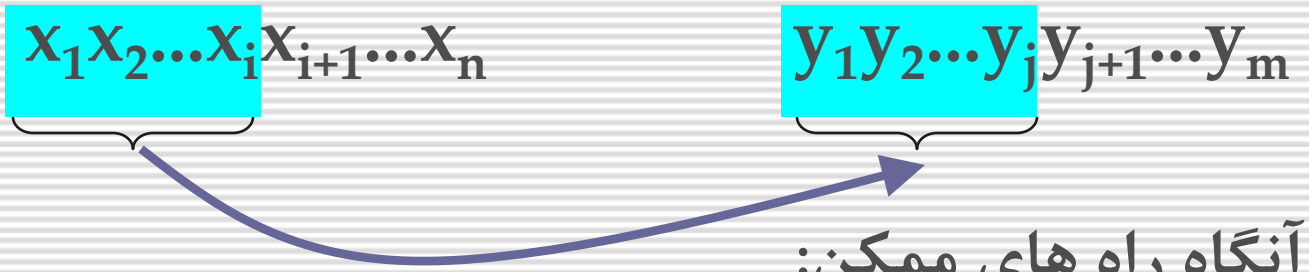
ویرایش رشته ها

- نماد cost(i,j): حداقل هزینه برای تبدیل پیشوند i کاراکتری X به پیشوند j کاراکتری از Y
  - یعنی هزینه تبدیل $x_1x_2x_3...x_i$ به $y_1y_2y_3..y_j$



- با توجه به مفهوم این نماد:
  - اگر i=j=0 ← cost(i,j) = cost(0,0) = 0
  - اگر i>0 و j=0 ← cost(i,j) = cost(i,0) = cost(i-1, 0) + D(xi)
  - اگر j>0 و i=0 ← cost(i,j) = cost(0,j) = cost(0,j-1) + I(Yj)

ویرایش رشته ها



- اگر $i>0$ و $j>0$ آنگاه راه های ممکن:
  - $cost(i,j) = cost(i-1,j) + D(xi)$
  - $cost(i,j) = cost(i,j-1) + I(yj)$
  - $cost(i,j) = cost(i-1,j-1) + C(xi,yi)$, if xi != yj
  - $cost(i,j) = cost(i-1,j-1)$, if xi == yj

ویرایش رشته ها

- اگر i>0 و j>0 ← سه روش کلی
  - ممکن است هزینه این روش ها یکسان نباشد.
  - چون می خواهیم حداقل هزینه را پیدا کنیم، پس اگر i>0 و j>0 آنگاه

```
cost(i,j) = MIN {
                cost(i-1,j) + D(xi)   ,
                cost(i,j-1) + I(yj)   ,
                if(xi!=yj) cost(i-1,j-1) + C(xi,yj)
                else cost(i-1,j-1)
}
```

ویرایش رشته ها

- فرمول کلی:

$$
cost(i,j) = \begin{cases}
0 & \text{if } i = j = 0 \\
cost(i-1,0) + D(xi) & \text{if } j = 0,\ i > 0 \\
cost(0,j-1) + I(yj) & \text{if } i = 0,\ j > 0 \\
MIN \begin{cases}
cost(i-1,j) + D(xi) \\
cost(i,j-1) + I(yj) \\
\text{if}(xi != yj)\ cost(i-1,j-1) + C(xi,yj) \\
\text{else } cost(i-1,j-1)
\end{cases} & \text{if } i > 0,\ j > 0
\end{cases}
$$

ویرایش رشته ها

- هدف: اگر رشته X دارای n کاراکتر و رشته Y دارای m کاراکتر باشد، باید cost(n,m) را پیدا کنیم.
  - البته cost(n,m) یک مقدار است، علاوه بر آن باید دنباله اعمال را نیز پیدا کنیم.
- از آرایه دو بعدی cost استفاده می کنیم و عناصر آن را محاسبه می کنیم تا نهایتا مقدار cost[n][m] را بدست آوریم.
  - سطر اول را محاسبه می کنیم cost[0][j] و $0 \le j \le m$
  - ستون اول را محاسبه می کنیم cost[i][0] و $0 \le i \le n$
  - بقیه عناصر را سطر به سطر محاسبه می کنیم.
  - آرایه (n+1)*(m+1) عنصری

ویرایش رشته ها

- مثال: تبدیل abc به cd با فرض اینکه هزینه تمام اعمال ویرایشی به ازای هر کاراکتر برابر ۱ می باشد.
- حداقل هزینه: cost(3,2)=3

0	1	2
1	1	2
2	2	2
3	2	3

- دنباله اعمال ویرایشی لازم:
  - delete(a)
  - change(b,c)
  - change(c,d)
  - البته در این مثال، دنباله ویرایشی بهینه یکتا نمی باشد.

ویرایش رشته ها

- محاسبه مرتبه زمانی
  - در این الگوریتم باید (n+1)*(m+1) عنصر آرایه cost را محاسبه کنیم.
  - محاسبه هر عنصر، به تعداد ثابتی عمل نیاز دارد که با مرتبه زمانی O(1) قابل انجام است. یعنی حجم اعمال لازم برای محاسبه هر عنصر آرایه cost، مستقل از n و m است.
  - در نتیجه، مرتبه زمانی الگوریتم O(n.m) می باشد.

مطالب مورد بحث

- زنجیره ضرب ماتریس ها
- ویرایش رشته ها
- **فروشنده دوره گرد**
- ترکیب k از n
- کوله پشتی صفر و یک

فروشنده دوره گرد

- TSP: Traveling Salesperson Problem
  - یک فروشنده برای فروش اجناس خود، باید به تعدادی شهر برود و در نهایت هم به شهر خودش بازگردد. فروشنده می خواهد کوتاهترین مسیر را پیدا کند بطوری که از هر شهر فقط یک بار بگذرد.
  - استفاده از یک گراف جهت دار وزن دار برای مدل کردن مساله
    - گره های گراف: شهر ها
    - یالهای گراف: جاده های ارتباطی بین شهرها
    - وزن یالها: طول مسیر بین دو شهر یا هزینه جابجایی بین دو شهری که گره های ابتدا و انتهای یال را مشخص می کنند.

فروشنده دوره گرد

- گردش (tour)
  - برای گراف جهت دار وزن دار G با n گره، یک گردش، مسیری است از یک گره به خود آن گره، بطوری که از هر یک از گره های دیگر، دقیقا یک بار بگذرد.
- مساله TSP در واقع
  - پیدا کردن گردش بهینه (optimum tour) می باشد. یعنی گردشی که دارای کمترین هزینه ممکن باشد (البته به فرض اینکه حداقل یک گردش وجود داشته باشد)

فروشنده دوره گرد

- اگر n شهر (n گره) وجود داشته باشد (فرض: گراف کاملا متصل)
  - باید در هر گام یک شهر را انتخاب کرده و به آنجا برویم.
  - ابتدا در گره مبدأ قرار داریم که این گره را با v1 نشان می دهیم.
  - در گام ۱ (شروع حرکت) n-1 گره قابل انتخاب می باشد. کدام؟
  - در گام ۲، n-2 گره قابل انتخاب است. کدام؟
  - ...
  - در گام n-1، 1 گره قابل انتخاب است.
  - در نتیجه، تعداد کل حالات ممکن (تعداد گردشهای ممکن) برابر است با:

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

  - در واقع (n-1)! گردش مختلف وجود دارد که باید از بین آنها کوتاهترین را انتخاب کنیم.

فروشنده دوره گرد

- الگوریتم brute-force: بررسی تمام حالات ممکن
  - (n-1)! گردش مختلف وجود دارد که باید طول هر کدام از آنها را محاسبه کنیم و سپس کوتاهترین گردش را انتخاب کنیم.
  - برای محاسبه طول هر گردش، از آنجا که هر گردش شامل n یال است، باید طول n یال را با هم جمع کنیم.
    - واضح است که جمع n عدد با مرتبه زمانی O(n) انجام می شود.
  - پس، محاسبه طول (n-1)! گردش، با مرتبه زمانی O(n.(n-1)!) قابل انجام است یعنی با مرتبه زمانی O(n!)
  - سپس باید کوتاهترین گردش را پیدا کنیم، که این هم با مرتبه زمانی O((n-1)!) قابل انجام است.
  - در نتیجه مرتبه زمانی الگوریتم brute-force، برابر O(n!) می باشد.

فروشنده دوره گرد

- این مرتبه زمانی فقط برای n های خیلی کوچک قابل قبول است.
- مثلا اجرای الگوریتم brute-force برای n=20، بر روی یک کامپیوتر معمولی، ماهها طول می کشد.
- $20! = 2432902008176640000 \approx 10^{18}$
- آیا می توان الگوریتم بهتری، مبتنی بر برنامه نویسی پویا ارائه کرد؟
- آیا اصل بهینگی در مورد مساله صادق است؟
  - بله

فروشنده دوره گرد

- V: مجموعه تمام گره ها
- v_1: گره مبدأ
- هدف: پیدا کردن کوتاهترین مسیری که از هر یک از گره های مجموعه V-{v1} یک بار می گذرد و سپس به v1 ختم می شود.
- فرض کنیم به روشی، تشخیص داده ایم که اولین گره مسیر (بعد از v_1) گره v_k می باشد. آنگاه برای تشخیص گره بعدی،
- باید کوتاهترین مسیری را پیدا کنیم که از هر یک از گره های مجموعه V-{v1, vk} یک بار بگذرد و به v1 ختم شود.
- ...
- مساله خاصیت بازگشتی دارد.

فروشنده دوره گرد

- A را بعنوان مجموعه گره های انتخاب نشده در نظر می گیریم که در هر گام، گره بعدی را باید از آن مجموعه انتخاب کنیم.
  - واضح است که A زیرمجموعه V است.
- برای نمایش گراف، از ماتریس مجاورت W استفاده شده است.
  - W[i][j]: وزن یالی که از گره v_i به گره v_j می رود.

فروشنده دوره گرد

- نماد D[vi][A] : طول کوتاهترین مسیری که از vi شروع می شود و از هر یک از گره های مجموعه A یک بار می گذرد و به v1 ختم می شود.
- استخراج رابطه بازگشتی

$$D[v_i][A] = \begin{cases} \underset{v_j \in A}{MIN}\{W[i][j] + D[v_j][A - \{v_j\}]\} & \text{if } A != \varnothing \\ W[i][1] & \text{if } A = \varnothing \end{cases}$$

- با توجه به مفهوم این نماد،
  - هدف اصلی بدست آوردن $D[v_1][V-\{v_1\}]$ می باشد.
  - البته این مقدار، فقط طول کوتاهترین گردش را مشخص می کند، سپس باید خود مسیر گردش را مشخص نمود.

فروشنده دوره گرد

- محاسبه مرتبه زمانی
  - اگر از یک آرایه دو بعدی برای ذخیره مقادیر عناصر D استفاده کنیم.
  - این آرایه دارای n سطر و 2^n ستون خواهد بود.
    - چرا 2^n: چون مجموعه n عضوی V، دارای 2^n زیرمجموعه می باشد.
  - پس بطور کلی تعداد عناصری که باید محاسبه کنیم برابر است با

$$n \cdot 2^n$$

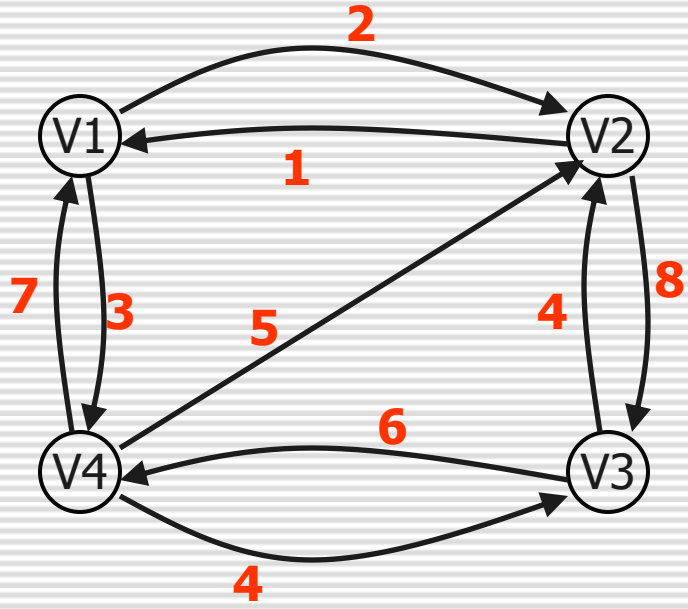
  - البته در عمل برخی عناصر را محاسبه نمی کنیم.
  - از طرفی در محاسبه هر عنصر، نیاز به مینیمم گیری می باشد که این مینیمم گیری با مرتبه زمانی O(n) انجام می شود.
  - در نتیجه، مرتبه زمانی الگوریتم ارائه شده: $O(n^2.2^n)$

فروشنده دوره گرد

- در نتیجه، مرتبه زمانی الگوریتم ارائه شده: $O(n^2.2^n)$
- مرتبه زمانی خیلی بد است، اما خیلی بهتر از الگوریتم brute-force می باشد.
- الگوریتم brute-force ← $O(n!)$
- الگوریتم پویا ← $O(n^2.2^n)$
- مثلا برای n=20 داریم:

- $n! = 20! = 2432902008176640000 \approx 10^{18}$
- $n^2.2^n = 419430400 \approx 10^9$

فروشنده دوره گرد



مثال:

باید به محاسبه عناصر D برای A های مختلف بپردازیم.

گام ۱: محاسبه حالات لازم برای زیرمجموعه های 0 عضوی

D[v2][Φ] , D[v3][Φ] , D[v4][Φ]

گام ۲: محاسبه حالات لازم برای زیرمجموعه های 1 عضوی

D[v2][v3], D[v2][v4], D[v3][v2], D[v3][v4], D[v4][v2], D[v4][v3]

گام ۳: محاسبه حالات لازم برای زیرمجموعه های 2 عضوی

D[v2][v3,v4], D[v3][v2,v4], D[v4][v2,v3]

گام ۴: محاسبه حالات لازم برای زیرمجموعه های 3 عضوی

D[v1][v2,v3,v4] ← جواب نهایی مساله

فروشنده دوره گرد

ماتریس **W**: وزن یالها

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

$D[v2][\Phi] = W[2][1] = 1$

$D[v3][\Phi] = W[3][1] = \infty$

$D[v4][\Phi] = W[4][1] = 7$

==========

$D[v2][v3] = W[2][3] + D[3][\Phi] = 8 + \infty = \infty$

$D[v2][v4] = W[2][4] + D[4][\Phi] = \infty + 7 = \infty$

$D[v3][v2] = W[3][2] + D[2][\Phi] = 4+1=5$

$D[v3][v4] = W[3][4] + D[4][\Phi] = 6+7=13$

$D[v4][v2] = W[4][2] + D[2][\Phi] = 5+1=6$

$D[v4][v3] = W[4][3] + D[3][\Phi] = 4 + \infty = \infty$

فروشنده دوره گرد

D[v2][v3,v4] = MIN { W[2][3] + D[v3][v4] , W[2][4]+D[v4][v3] }

= MIN { 21 , ∞ } = 21

D[v3][v2,v4] = MIN { W[3][2]+D[v2][v4] , W[3][4]+D[v4][v2] }

= MIN {∞ , 12 } = 12

D[v4][v2,v3] = MIN { W[4][2]+D[v2][v3] , W[4][3]+D[v3][v2] }

= MIN {∞ , 9 } = 9

==========

D[v1][v2,v3,v4] = MIN { W[1][2] + D[v2][v3,v4] ,

W[1][3] + D[v3][v2,v4] ,

W[1][4] + D[v4][v2,v3] }

= MIN { 23 , ∞ , 12} = 12

طول کوتاهترین مسیری که از **v1** شروع شود و از هر یک از گره های **v2** و **v3** و **v4** دقیقا یک بار بگذرد و دوباره به **v1** ختم شود.

فروشنده دوره گرد

- **اما خود مسیر را چگونه بدست آوریم؟**
- **عدد ۱۲ از کدام term بدست آمد؟ از** W[1][4] + D[v4][v2,v3]
  - **این ترم معادل حرکت از v1 به v4 است. پس در اولین گام از v1 باید به v4 برویم.**
- **اما مقدار** D[v4][v2,v3] **از کدام term بدست آمد؟**
  - از W[4][3]+D[v3][v2] که معادل حرکت از v4 به v3 است. پس بعد از v4 باید به v3 برویم.
- اما مقدار D[v3][v2] از کدام term بدست آمد؟
  - **از** W[3][2] + D[2][Φ] **که معادل حرکت از v3 به v2 است. پس باید از v3 به v2 برویم.**
- **مقدار** D[2][Φ] **چطور بدست آمد؟ این مقدار برابر بود با** W[2][1] **یعنی حرکت از v2 به v1**
- **در نتیجه مسیر** **V1 → V4 → V3 → V2 → V1**

مطالب مورد بحث

- زنجیره ضرب ماتریس ها
- ویرایش رشته ها
- فروشنده دوره گرد
- ترکیب k از n
- کوله پشتی صفر و یک

ترکیب k از n

- ترکیب k مؤلفه از n شیء

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

- روش اول: طبق فرمول بالا، ابتدا هر یک از مقادیر !n و !k و !(n-k) را محاسبه کنیم و سپس با استفاده از این مقادیر، مقدار کسر فوق را حساب کنیم.
- مشکل: اگر k و n کوچک نباشند، محاسبه فاکتوریل برای آنها مشکل می شود.

ترکیب k از n

n	Number of digits in n!
10	7
50	65
100	158
200	375
500	1135
1000	2568
10000	35660

ترکیب k از n

- راه حل: از فرمول زیر استفاده کنیم که نیاز به محاسبه فاکتوریل ندارد و اصلا ضرب و تقسیم ندارد.

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

- در ادامه برای راحتی کار $\binom{n}{k}$ را با comb(n,k) نشان می دهیم.

ترکیب k از n

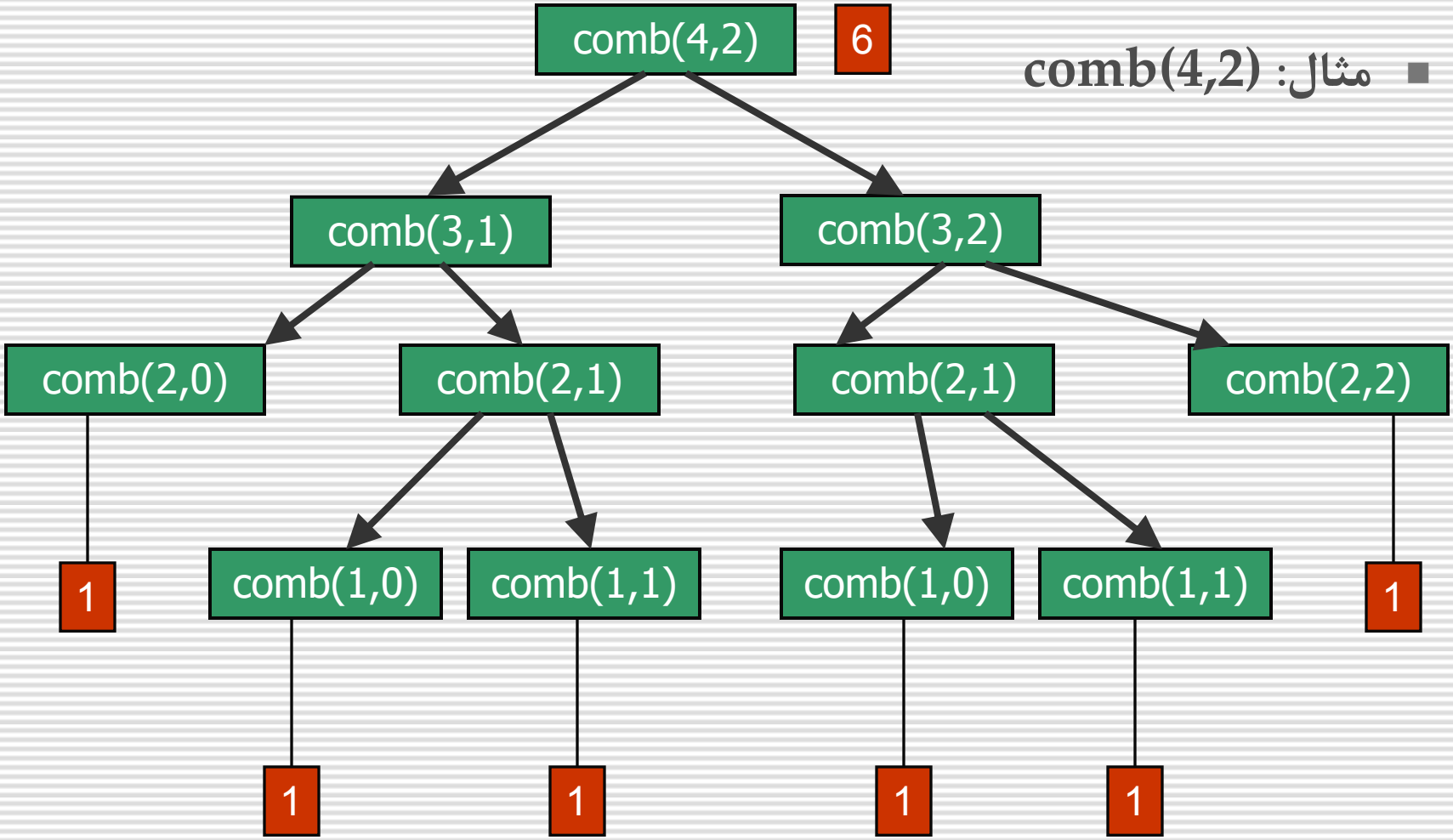
- یک الگوریتم ساده با استفاده از روش تقسیم و حل
  - دقیقا مبتنی بر رابطه بازگشتی مورد نظر

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

- مشکل:
  - هر مساله را به دو مساله کوچکتر که تقریبا به اندازه مساله اصلی هستند تقسیم می کند.
  - محاسبه مجدد عناصر
  - در نتیجه مرتبه زمانی بد می باشد. در واقع از مرتبه زمانی $O(2^n)$ می باشد.

ترکیب k از n

- مثال: comb(4,2)



ترکیب k از n

- آیا این مساله را می توان به روش پویا، حل کرد؟
  - آیا اصل بهینگی صدق می کند.
  - آیا اصلا موضوع بهینه سازی مطرح است؟

- برای ارائه یک راه حل مبتنی بر روش پویا، در اولین گام، نیازمند یک رابطه بازگشتی می باشیم که جواب مسائل بزرگتر را بر حسب جواب مسائل کوچکتر، بیان نماید.
- این رابطه را داریم.

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

ترکیب k از n

- برای انجام کار
  - از آرایه دو بعدی C استفاده می کنیم که C[i][j] معادل comb(i,j) می باشد.
  - بنابراین برای محاسبه comb(n,k) باید مقدار عنصر C[n][k] را حساب کنیم.
  - عناصر آرایه را محاسبه می کنیم تا نهایتا C[n][k] بدست آید.

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

ترکیب k از n

- محاسبه عناصر به چه ترتیبی باشد؟
  - روش پایین به بالا
  - عناصر آرایه به دو گروه قابل تقسیم می باشند.
    - عناصری که برای محاسبه آنها به مقدار عناصر دیگر نیازی نیست. C[i][i]=1
    - عناصری که برای محاسبه آنها به مقدار دو عنصر دیگر نیاز است.
  - پس در گام اول عناصر مستقل را محاسبه می کنیم.
  - سپس در گام دوم، با استفاده از مقدار عناصر مستقل، عناصر وابسته را محاسبه می کنیم.
  - عناصر مستقل در واقع عناصر ۱-ساز می باشند.

ترکیب k از n

- مثال: comb(4,2)
- گام اول: محاسبه عناصر ۱-ساز
  - C[i][j]=1 اگر i=j (قطر اصلی)
  - C[i][0]=1 ← عناصر ستون اول

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

ترکیب k از n

- مثال: comb(4,2)
- گام دوم: محاسبه عناصر وابسته
  - C[i][j] = C[i-1][j-1] + C[i-1][j]
  - مقدار هر عنصر: مقدار همسایه بالا و سمت چپ + مقدار همسایه بالایی

	0	1	2
0	1		
1	1	1	
2	1	2	1
3	1	3	3
4	1	4	6

جواب مساله برابر است با مقدار C[4][2]

comb(4,2) = C[4][2] = 6

ترکیب k از n

- آرایه C یک آرایه (k+1) * (n+1) است.
- برخی عناصر محاسبه نشدند. (سلولهای زرد در جدول زیر)
- برخی عناصر محاسبه شدند اما محاسبه آنها ضروری نبود، یعنی مقدار محاسبه شده مورد استفاده قرار نگرفت. (سلولهای قرمز در جدول زیر)

	0	1	2
0	1		
1	1	1	
2	1	2	1
3	1	3	3
4	1	4	6

ترکیب k از n

- الگوریتم مبتنی بر روش پویا، بدون محاسبه عناصر اضافه

```
int comb(int n, int k) {
    for(i=0; i<=k; i++)                C[i][i] = 1;
    for(i=1; i<=n; i++)                C[i][0] = 1;

    for(i=2; i<=n; i++)
        for(j=max(1, k+i-n); j<=min(i-1,k); j++)
            C[i][j] = C[i-1][j-1] + C[i-1][j];

    return C[n][k];
}
```

ترکیب k از n

محاسبه comb(7,4)

	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	15
7	1	7	21	35	35

ترکیب k از n

- محاسبه مرتبه زمانی
  - تعداد عناصر محاسبه شده آرایه C برابر است با (n-k+1)(k)
  - برای محاسبه هر عنصر،
    - برای عناصر مستقل، محاسبه خاصی نیاز نیست ← O(1)
    - برای عناصر وابسته، باید مقدار دو عنصر دیگر را با هم جمع کنیم. ← O(1)
  - در نتیجه الگوریتم ارائه شده از مرتبه زمانی O(n.k) می باشد.
  - آیا این استدلال قابل قبول است؟؟؟؟
    - آیا واقعا جمع دو عنصر آرایه را با O(1) قابل انجام است.
    - مثلا مقدار comb(100,50) تقریبا برابر 10^{30} است. پس ...
    - یادآوری: در اول بحث گفتیم : " اگر k و n کوچک نباشند، محاسبه فاکتوریل برای آنها مشکل می شود"

مطالب مورد بحث

- زنجیره ضرب ماتریس ها
- ویرایش رشته ها
- فروشنده دوره گرد
- ترکیب k از n
- کوله پشتی صفر و یک

کوله پشتی صفر و یک

- شبیه مساله کوله پشتی غیر صفر و یک می باشد با این تفاوت که یک شیء را یا باید بطور کامل انتخاب کنیم یا اصلا انتخاب نکنیم.
- انتخاب کسری از شیء مجاز نمی باشد.
- برای کوله پشتی غیر صفر و یک، الگوریتمی مبتنی بر روش حریصانه ارائه شد.
- این الگوریتم برای کوله پشتی صفر و یک قابل استفاده نمی باشد.
- مثال نقض: M=20

(P1 , P2 , P3) = (25 , 24 , 15)

(W1 , W2 , W3) = (18 , 15 , 10)

جواب کوله پشتی غیر صفر و یک : (0, 1, 1/2) ارزش: ۳۱.۵

کوله پشتی صفر و یک

- جواب حاصل از الگوریتم حریصانه برای این مثال:
  - (0 , 1 , 0) ارزش: ۲۴
- جواب بهینه برای این مثال: (0 , 1 , 0) ارزش: ۲۴
- در نتیجه الگوریتم حریصانه ارائه شده، برای کوله پشتی صفر و یک مناسب نیست.

کوله پشتی صفر و یک

- از یک منظر دیگر، مساله کوله پشتی صفر و یک معادل است با
  - انتخاب زیرمجموعه ای از اشیا که مجموع ارزش آنها ماکزیمم شود و مجموع وزن آنها از ظرفیت کوله پشتی تجاوز نکند.
- از این منظر:
  - یک مجموعه n عضوی دارای 2^n زیرمجموعه است ← یک راه حل: بررسی تمام حالات ممکن
  - روش brute-force: مرتبه زمانی $O(n.2^n)$
  - اما با استفاده از روش پویا، بدنبال روش بهتری هستیم.
  - آیا اصل بهینگی صدق می کند؟

کوله پشتی صفر و یک

- تعریف نماد P[i][w] که i>0 و w>0
- P[i][w] برابر است با حداکثر ارزش کوله پشتی وقتی که برای پر کردن آن، فقط از i شیء اول استفاده کنیم و ضمنا کوله پشتی را حداکثر تا ظرفیت w پر کنیم.
- اگر کوله پشتی را با انتخاب اشیا از بین i شیء اول پر کنیم، طوری که مجموع وزن اشیا از w بیشتر نشود و ضمنا مجموع ارزش اشیا حداکثر شود، حداکثر ارزش بدست آمده را با P[i][w] نشان می دهیم.
- P[i][w] فقط ارزش ماکزیمم را بیان می کند، نه اینکه این ارزش به ازای چه انتخابهایی بدست آمده است.

کوله پشتی صفر و یک

- طبق مفهوم P[i][w] خواهیم داشت

$$P[i][w]=\begin{cases}\max\{P[i-1][w],\ P[i-1][w-w_i]+P_i\} & \text{if } w \geq w_i\\ P[i-1][w] & \text{if } w < w_i\end{cases}$$

- مساله اصلی پیدا کردن P[n][M] است.
  - البته این فقط بیشترین ارزش ممکن برای پر کردن کوله پشتی می باشد و باید X_i ها را هم محاسبه کنیم.

کوله پشتی صفر و یک

- روش کار: استفاده از یک آرایه دو بعدی (n+1)*(M+1) برای P و محاسبه تمام عناصر آن تا نهایتا P[n][M] را بدست آوریم.
- واضح است که P[0][i]=0
- پس ترتیب پر کردن عناصر جدول:

  گام اول: P[0][i]=0 برای i=1, 2, …, W

  گام دوم: از بالا به پایین، سطر به سطر عناصر را حساب می کنیم.
- مثال: M=5 و n=3

(P1 , P2 , P3) = (5, 7, 4)

(W1 , W2 , W3) = (1, 3, 2)

کوله پشتی صفر و یک

- مرتبه زمانی: باید مقدار (n+1)*(M+1) عنصر را حساب کنیم. محاسبه هر عنصر با O(1) قابل انجام است ← محاسبه کل عناصر با O(n.M)
- اگر M خیلی بزرگ باشد، مثلا $M>2^n$ باشد آنگاه مرتبه زمانی این الگوریتم از الگوریتم brute-force هم بدتر خواهد شد ← قابل قبول نیست.
- روش دیگر: با توجه به رابطه صفحه قبل، واضح است که برای محاسبه هر عنصر P نهایتا، به دو عنصر دیگر P نیاز داریم. بنابراین فقط عناصر لازم را محاسبه می کنیم. این الگوریتم بیشتر مبتنی بر روش تقسیم و حل است تا روش پویا.

کوله پشتی صفر و یک

- برای بدست آوردن P[n][M] باید P[n-1][M] و همچنین P[n-1][M-w_n] را بدست آوریم و ...
- یک درخت دودویی
- اجرای مثال قبل با استفاده از این روش
- مثال: M=5 و n=3

(P1 , P2 , P3) = (5, 7, 4)
(W1 , W2 , W3) = (1, 3, 2)

- نحوه پیدا کردن X_i ها ؟
- با این کار مرتبه زمانی الگوریتم $O(2^n)$ خواهد شد.