动态规划（dp）

简述

思想：通过不重复的计算来得到某个问题的最优解，解决各种**离散优化问题**，这些问题中往往有很多解，每个解都会有一个值，我们称这种解为该问题的最佳解之一。（可能有多个最优解都达到了最优值，比如说找零问题）

> **Important**
>
> - **刻画最优解的结构特性**
>
> P(X) 例如:P(n)，P(n,w)
>
> - **将问题划分为子问题**
>
> P(X) =  （f（P（X-A1），...，P(X-Ad)））
>
> 通常而言  是max{} 或者 min{}
>
> 比如说:
>
> 斐波那契数：F(n) = F(n-1) + F(n-2)
>
> 找零问题：M(n) = min{1 + M(n-di)}
>
> - **自底而上计算**
>
> P(X) =  （f（P（X-A1），...，P(X-Ad)））
>
> 从子问题算到最后的问题上
>
> - **注意初值**

与其他算法策略的区别

贪婪策略： 逐步建立一个解决方案，每一步都"目光短浅"地选择优化一些局部目标（选取的是局部的最优解）

分治策略： 将一个问题分解为多个**不相交**的子问题，独立解决每个子问题，并将子问题的解结合起来形成原问题的解。

动态规划： 将一个问题分解称一系列**相互存在重叠**的子问题，并不断由子问题的解形成越来越大的问题的解

一些列子

> **Note**
>
> 直接总结算法的思路，因为题目的背景都是上课讲过的，这里就是再次回顾一下解决这些问题的算法思路。

EX1.最长单调子序列

令S[1]S[2]S[3]....S[n]表示输入的序列

令L(i) (1 ≤ i ≤ n)表示**以S[i]结束**的**最长**单调递增子序列的长度

-子序列的最后一项为S[i]

-此时**只需考虑**前i项 S[1]S[2]....S[i]的**以S[i]结束**的**最长**单调子序列即可

总目标：max{L(1),...L(n)}（每一个子序列总会有一个“最后一项”的）

目标函数的递归关系：

$$L(i)=\begin{cases}\max\limits_{1\le j<i且S[j]<S[i]} L(j)+1, & 若存在j使得1\le j<i且S[j]<S[i]\\ 1, & 否则\end{cases}$$

```
输入：序列
输出：最长单调子序列长度Len
1.for i = 1 to n do
2.    L(i) ← 1
3.    for j = 1 to i-1 do
4.       if S[j] < S[i] and L[j] ≥ L[i] do
5.          L[i] ← L[j] + 1 (,P(i) ← j)
6.Len ← max{L(1),...,L(n)}  //假设最大值为L(k)
7.i ← 1 //回溯
8.j ← k
9.do
10.     T(i) ← S[j],i ← i+1, j ← P(j)
11.until j = 0;
12.output Len以及T的反序
```

然后设置一个标记函数P[i]去回溯每一个位置

比如现在S = 1,8,2,9,3,10,4,5

i	S	L[i]		P[i]
1	1	1	1	0
2	8	2	**1**,8	1
3	2	2	**1**,2	1
4	9	3	1,**8**,9	2
5	3	3	1,**2**,3	3
6	10	4	1,8,**9**,10	4
7	4	4	1,2,**3**,4	5
8	5	5	1,2,3,**4**,5	7

EX2.最大子段和问题

描述：就是一段序列，然后我们要求里面和最大的子段

-例如

所以我们的**目标函数**是：

$$\max\left\{0,\ \max_{1\le i\le j\le n}\left\{\sum_{k=i}^{j} a_k\right\}\right\}$$

这个可以看成灰度图：

思路：

-蛮力法?

不太可行，会有$n^2/2$个不同的子段和

-Kadane算法

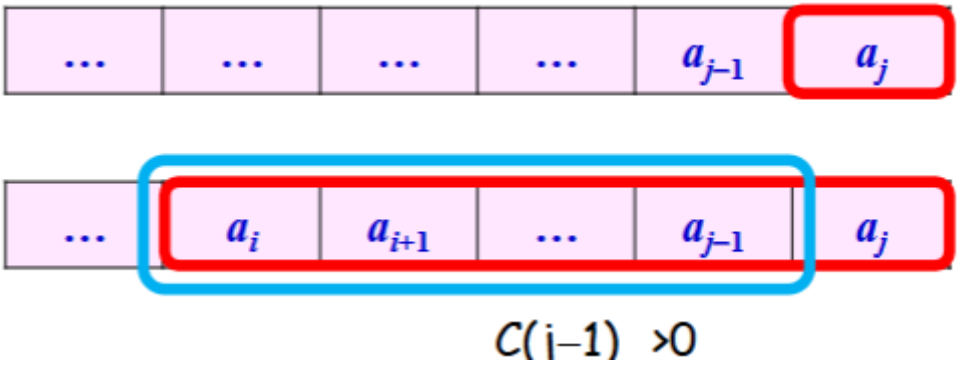
令C(j)表示必须以元素aj结尾（因此长度至少为1）的最大子段和（允许为负数值）

$$即C(j)=\max_{1\le j<i}\{\sum_{k=1}^{j} a_k\}$$

此时就会有**两种**情况：

1.长度为1，即仅仅包含a_j

2.长度大于1



所以可以得出C(j)的递推公式：

$$C(j)=\begin{cases} a_j + C(j-1) & \text{if C(j-1)>0} \\ a_j & \text{else} \end{cases}$$

```
Alhorithm MaxSum(A,n)
输入：数组/序列A，长度n
输出：最大子段和
1.current_sum ← 0
2.best_sum ← 0
3.for j = 1 to n do
4.    if current_sum > 0 then #如果现在的和大于零
5.       current_sum ← current_sum + aj
6.    else current_sum ← aj    #如果和小于等于零就重新开始赋值
7.    if current_sum > best_sum then #更新最大值
8.       best_sum ← current_sum
9. return best_sum
```

> **Note**
>
> 如果还要得到具体的子段，还需要进行标记。
>
> 在第6行的时候更新start位置，在第5行的时候更新end的位置。

EX3.0-1背包问题

背包问题是一个很典型的组合优化问题

假设共有n种物品，其中第i种物品的价值为vi，重量为wi（假定vi和wi都是整数）

确定要从这n种物品中选择哪些种、每种选择多少，将其装入背包里，使得这些物品的总重量不超过给定的限制**W**，并且总价值尽可能大

0-1背包问题描述：

对于每种物品，当在考虑是否将其装入背包时，要么全部装入背包，要么全部不装入背包，而不能只装入物品i的一部分

> **Important**
>
> 凡是论及“背包问题”时，将“体积”和“重量”视作同一个属性，“收益”和“价值”含义相同

贪婪策略的话，没法得到最优解。

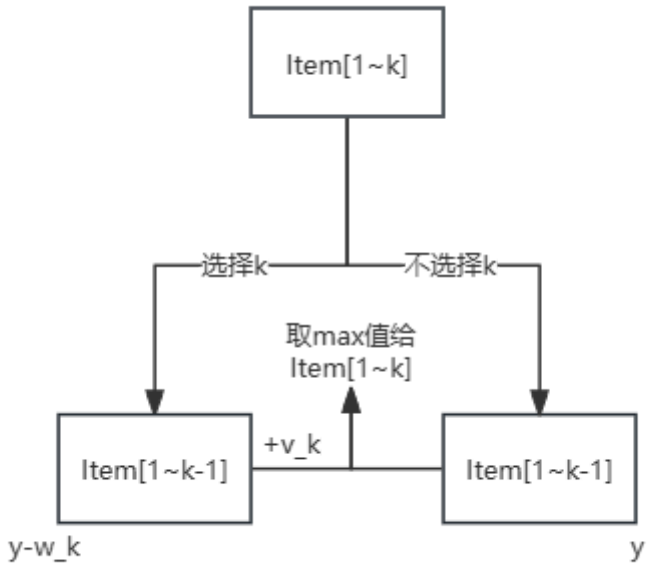
所以用动态规划：

考虑子问题P(k,y)——只使用前k种物品，总重量不超过y

令F(k,y)表示仅使用前k种物品且背包容量为y时的最优解的总价值

> **Note**
>
> 即子问题P(k,y)问题的最优解
>
> 原问题的目标为F(n,W)

于是在选择的时候会有**两种**情况：



得到递推式：

$$F(k,y)=\begin{cases}0 & \text{if } k=0 \text{ or } y=0\\ F(k-1,y) & \text{if } w_k>y\\ \max\{F(k-1,y),v_k+F(k-1,y-w_k)\} & \text{otherwise}\end{cases}$$

填起来的表格长这样：

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	40
{1,2,3,4,5}	0	1	6	7	7	18	22	28	29	34	35	40

+18

+22

伪代码：

```
输入：n,w1,...,wn,v1,...,vn,W                时间复杂度：O(nW)
1.for y = 0 to W              #重量上限为0，赋值为0
2.  F(0,y) ← 0
3.for k = 1 to n              #什么物品都没有放，赋值为0
4.  F(k,0) ← 0
5.for k = 1 to n
6.  for y = 1 to W
7.      if(wk > y)            #如果该物品的重量超过上限
8.          F(k,y) ← F(k-1,y)
9.      else                  #如果该物品的重量没有超过上限
10.         F(k,y) ← max{F(k-1,y),vk+F(k-1,y-wk)} #递推关系
11.return F(n,W)
```

> **Note**
>
> 时间复杂度进一步降低，可参考：
>
> https://zhuanlan.zhihu.com/p/30959069

EX4.投资问题

> **Tip**
>
> 可以和背包问题类比起来看

问题描述：

有m元，n项可能的投资项目，fi(x)表示将x元投入第i个项目获得的利益

> **Note**
>
> x是非负整数
>
> f_i(x)值非负
>
> f_i(x)关于x是不减函数

目标：将投资的利益**最大化**

即：

$$max\{f_1(x_1)+f_2(x_2)+...+f_n(x_n)\}$$

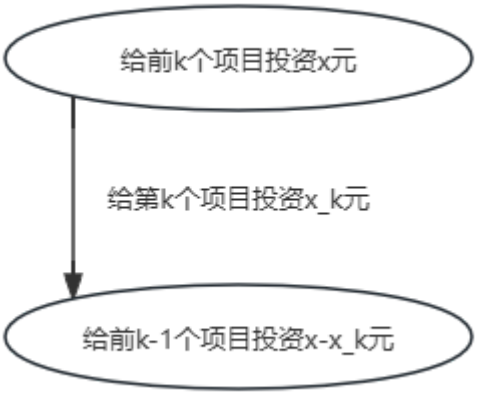
$$s.t.\ x_1+x_2+...+x_n=m$$

思路:

用F_k(x)表示前k个项目进行共计x元的投资所能取得的最大收益

x_k(x)表示在F_k(x)中投资给项目k的钱数

考虑"**最后一个**"项目



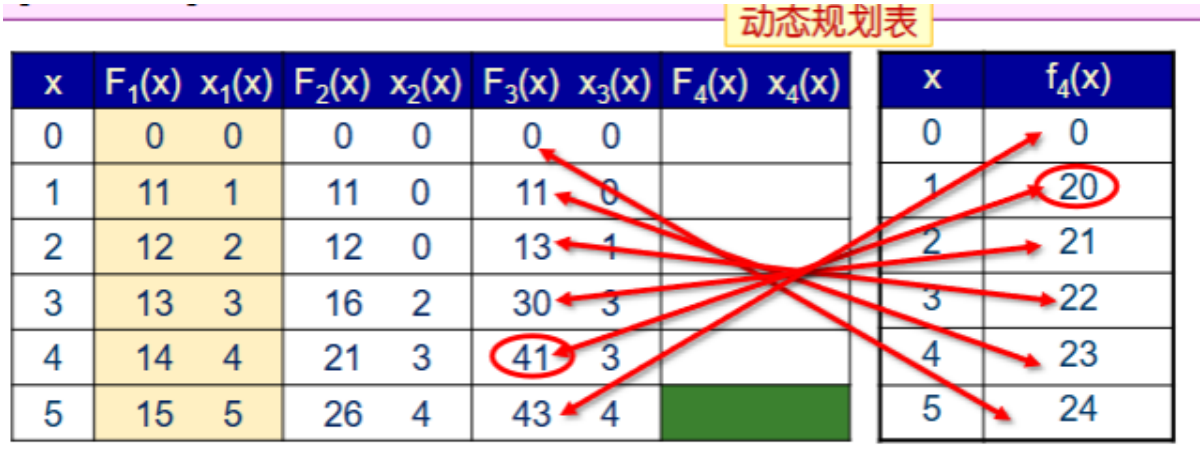
递推关系式:

$$F_k(x) = \max_{0 \le x_k \le x} \{f_k(x_k) + F_{k-1}(x - x_k)\}, 0 \le x \le m, 2 \le k \le n$$

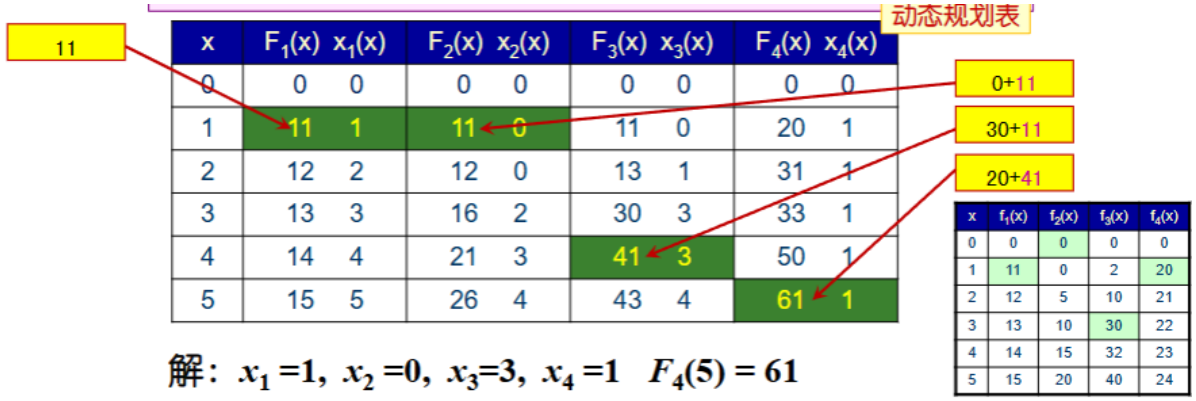
$$F_1(x) = f_1(x), 0 \le x \le m$$

填表:

(填入F_4时的情况)



(完整的填表)



> **Tip**
>
> **从前往后**依次填表，然后用递推关系进行判断
>
> 与背包问题相似，但是背包问题的初值的处理方法和投资问题还是不太一样的。
>
> 背包问题是重量和价值为0均赋值为0，这样方便后面去运算；**但是**投资问题是投资金额为0全部赋值为0，初始值为f_1的投资回报。

```
1.for y = 1 to m
2.  F1(y) ← f1(y)    #只投资第一个物品，赋初值
3.for k = 2 to n     #从投资第二件物品开始计算收益
4.  for y = 1 to m
5.      Fk(y) ← max_0≤xk≤y{fk(xk) + Fk-1(y-xk)} #递推关系
6.return Fn(m)
```

时间复杂度:

计算 F_k(y) (2 ≤ k ≤ n,1 ≤ y ≤ m) 时候需要y+1次加法和y次比较

加法次数	$\sum_{k=2}^{n}\sum_{y=1}^{m}(y+1)=\frac{1}{2}(n-1)m(m+3)$
比较次数	$\sum_{k=2}^{n}\sum_{y=1}^{m}y=\frac{1}{2}(n-1)m(m+1)$
时间复杂度	$O(nm^2)$

> **Note**
>
> **序列的比较**

EX5.最长公共子序列（LCS）

描述:

子序列和子串不一样，序列只要相对的顺序不变就行。然后找到两个序列之间最长的子序列。

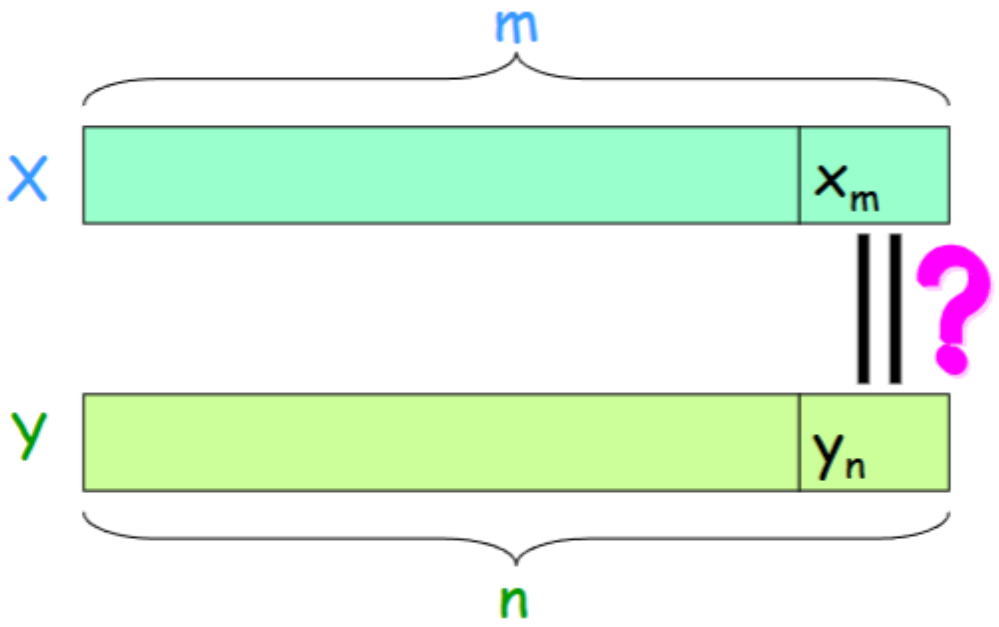
> **Note**
>
> 序列的长度指的是序列的项数
>
> 空序列是任意两个序列的公共子序列
>
> 任一个序列和空序列的最长公共子序列都是空序列

最长公共子序列可能不唯一

应用:

度量两个序列的相似程度

刻画方式:



LCS的最优子结构（定理）：

三种情况：

- 定理（LCS的最优子结构）
  - 设序列 $X=x_1x_2...x_m$，$Y=y_1y_2...y_n$，$Z=z_1z_2...z_k$ 是 X 和 Y 的任一个LCS

    使用反证法，否则必然可以在 Z 的末尾添加 $x_m=y_n$
    - 1. 若 $x_m=y_n$，则 $z_k=x_m=y_n$，且 $Z[1...k-1]$ 是 $X[1...m-1]$ 和 $Y[1...n-1]$ 的（一个）LCS
    - 2-1. 若 $x_m\neq y_n$，则 $z_k\neq x_m$ 蕴涵 Z 是 $X[1...m-1]$ 和 Y 的（一个）LCS
    - 2-2. 若 $x_m\neq y_n$，则 $z_k\neq y_n$ 蕴涵 Z 是 X 和 $Y[1...n-1]$ 的（一个）LCS

      若（$z_k=x_m$ 且 $z_k=y_n$）不成立，则必然有（$z_k\neq x_m$ 或 $z_k\neq y_n$）成立

最后两个相等、不相等（x与z相等，y与z相等）

> **Important**
>
> 如果用递归的思想会很慢，因为会有很多重复计算的地方
>
> 所以考虑按特定的次序计算 m x n个 len (i,j)

伪代码：

```
1.for i = 1 to m do len(i,0) ← 0          时间复杂度：O(mn)
2.for j = 0 to n do len(0,j) ← 0
3.for i = 1 to m do
4.   for j = 1 to n do
5.       if X[i] = Y[j] then  #如果两个相等
6.           len(i,j) ← len(i-1,j-1) + 1
7.           b(i,j) ← 1       //往左上方走
8.       else if len(i-1,j) ≥ len(i,j-1) then #比较左边和上边的子序列长度，取较长的那段子
序列
9.           len(i,j) ← len(i-1,j)
10.          b(i,j) ← 2       //往上走
11.      else len(i,j) ← len(i,j-1) #这里同样取较长的那段子序列
12.          b(i,j) ← 3       //往左走
13.return len(m,n) 以及 b
```

具体填表:

（整张表从左上往右下填写，然后比较最后一个字段）

■ 示例：X[1...4] = acbd，Y[1...4] = abcd

len(i,j)	j=0		1 a		2 b		3 c		4 d	
i=0	0	""	0	""	0	""	0	""	0	""
1 a	0	""	1	a	1	a	1	a	1	a
2 c	0	""	1	a	1	a	2	ac	2	ac
3 b	0	""	1	a	2	ab	2	ab	2	ab
4 d	0	""	1	a	2	ab	2	ab	3	abd

■ 示例：X= a, b, c, d, a, c, e; Y= b, a, d, c, a, b, e

len	i=0	1 a	2 b	3 c	4 d	5 a	6 c	7 e
j=0	0	0	0	0	0	0	0	0
1 b	0	0 ↑	1 ↖	1 ←	1 ←	1 ←	1 ←	1 ←
2 a	0	1 ↖	1 ↑	1 ↑	1 ↑	2 ↖	2 ←	2 ←
3 d	0	1 ↑	1 ↑	1 ↑	2 ↖	2 ↑	2 ↑	2 ↑
4 c	0	1 ↑	1 ↑	2 ↖	2 ↑	2 ↑	3 ↖	3 ←
5 a	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	3 ↑
6 b	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑	3 ↑
7 e	0	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↖

时间复杂度与空间复杂度：O(mn)

> **Note**
>
> 数组b的作用就是用来回溯最长子序列具体的内容是什么。
>
> 回溯算法就是遇到左上的方向，然后记录字母，其他的根据方向继续遍历。

（回溯）构造LCS的算法

- 例：X: a,b,c,d,a,c,e; Y: b,a,d,c,a,b,e

len	0	1 a	2 b	3 c	4 d	5 a	6 c	7 e
0	0	0	0	0	0	0	0	0
1 b	0	0	1 ↖	←	←	1 ←	1 ←	1 ←
2 a	0	1 ↖	1 ↑	1 ↑	1 ↑	2 ↖	2 ←	2 ←
3 d	0	1 ↑	1 ↑	1 ↑	2 ↖	2 ↑	2 ↑	2 ↑
4 c	0	1 ↑	1 ↑	2 ↖	2 ↑	2 ↑	3 ↖	3 ←
5 a	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	3 ↑
6 b	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑	3 ↑
7 e	0	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↖

b
a
c
e

（具体的伪代码就用老师ppt里的图来表示了）

- 可以使用 $b(i, j)$ 找到具体的LCS
- 从 $b(m, n)$ 开始，回溯到某个 $b(0, j)$ 或 $b(i, 0)$ 为止

```
1.  i ← m, j ← n, k ← 1
2.  while ( i ≠ 0 and j ≠ 0 )
3.     if b(i, j) = 1 then                    //↖
4.        i ← i − 1, j ← j − 1
5.        LCS[k] ← X[i], k ← k + 1
6.     if b(i, j) = 2 then i ← i − 1          //↑
7.     if b(i, j) = 3 then j ← j − 1          //←
8.  for k = len(m, n) downto 1
9.     output LCS[k]
```

EX6. 最短公共超序列（SCS）

定义：

设X和Y是两个序列。如果X和Y都是Z的子序列，那么称Z是X和Y的公共超序列（common supersequence）

> **Note**
>
> 1.X和Y肯定有一个最短的公共超序列
>
> 2.任一个序列和空序列的最短公共超序列是序列自身

递推关系式：

令 *len[i,j]* 表示 *X[1..i]* 和 *Y[1..j]*的最短公共超序列的长度，于是会有如下的公式。

$$len[i,j]=\begin{cases} j & \text{if } i=0 \\ i & \text{if } j=0 \\ len[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j \\ \min\{len[i,j-1]+1, \\ \quad len[i-1,j]+1\} & \text{if } i,j>0 \text{ and } x_i\neq y_j \end{cases}$$

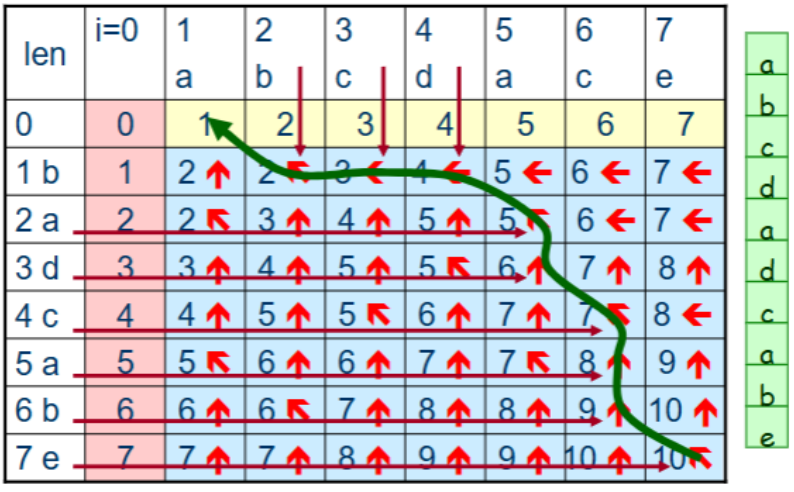
> **Tip**
>
> 和最长公共子序列的思路是有相似之处的，比如从前往后找，然后都是比较最后一个是否一样，**但是**处理的方法略有不同，最长公共子序列是找前面最长的然后直接赋值，最短公共超序列就要从前面找最小还要加一（因为是超序列嘛，要把两个序列中的内容都涵盖进去）

伪代码：

```
1.for i=0 to n do   #赋初值，和空序列进行比较
2.  len[i,0] = i
3.for j-0 to m do   #赋初值，和空序列进行比较
4.  len[0,j] = j
5.for i=1 to n do
6.  for j=1 to m do
7.      if(X[i] = Y[j]) then    #两个字段相同
8.          len[i,j] ← len[i-1,j-1] + 1
9.      else                    #两个字段不相同
10.         len[i,j] ← min{len[i-1,j]+1,len[i,j-1]+1} #找最小
11.return len[n,m]
```

填表：

- 示例：X: a, b, c, d, a, c, e; Y: b, a, d, c, a, b, e



回溯算法：

```
1.for i=0 to n do   #赋初值，和空序列进行比较
2.  c[i,0] = i
3.for j-0 to m do   #赋初值，和空序列进行比较
```

```
4.  c[0,j] = j
5.for i=1 to n do
6.  for j=1 to m do
7.      if(X[i] = Y[j]) then c[i,j] ← c[i-1,j-1]+1, b[i,j] ← 1
8.      else                        #两个字段不相同
9.          c[i,j] ← min{c[i-1,j]+1,c[i,j-1]+1} #找最小
10.         if(c[i,j] = c[i-1,j]+1) then b[i,j] ← 2 #记录来源
11.         else b[i,j] ← 3
12.p ← n, q ← m, k ← 1  #回溯超序列
13.while(p≠0 or q≠0)
14.     if(b[p,q] = 1) then {SCS{k}←y[p],k←k+1,p←p-1,q←q-1}
15.     if(b[p,q] = 2) then {SCS{k}←x[p],k←k+1,p←p-1}
16.     if(b[p,q] = 3) then {SCS{k}←y[q],k←k+1,q←q-1}
#最后再反向输出SCS[]就得到了最短公共超序列
```

LCS和SCS之间的联系:

- 定理:

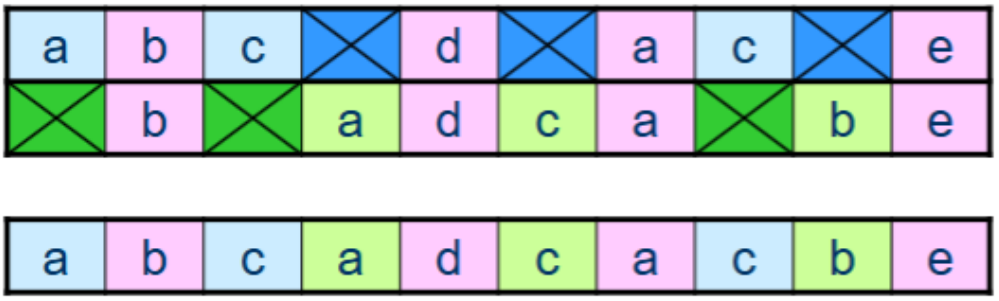
  假设 $|X| = m, |Y| = n, |LCS| = L, |SCS| = K$

  则有 $L + K = m + n$

- 示例:

  X: a b c d a c e; Y: b a d c a b e

  LCS: b d a e; SCS: a b c a d c a c b e



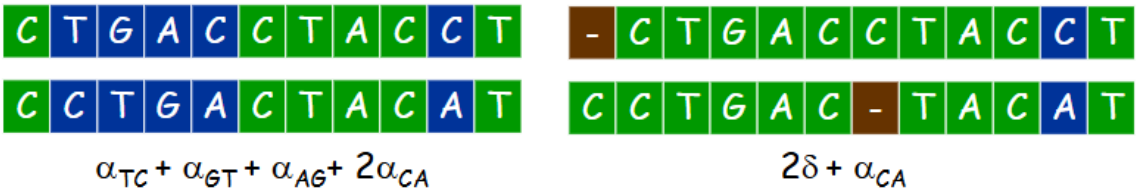
EX7.序列对齐与编辑距离

多相似：错误匹配和缺漏。

编辑距离:

就是给缺漏一个惩罚值，错误匹配一个惩罚值。

- 缺漏的惩罚 δ；错误匹配的惩罚 α_{pq}
  - 为叙述上的方便，定义 $\alpha_{pp}=0$
- 总开销 = 缺漏的惩罚总值 + 错误匹配的惩罚总值



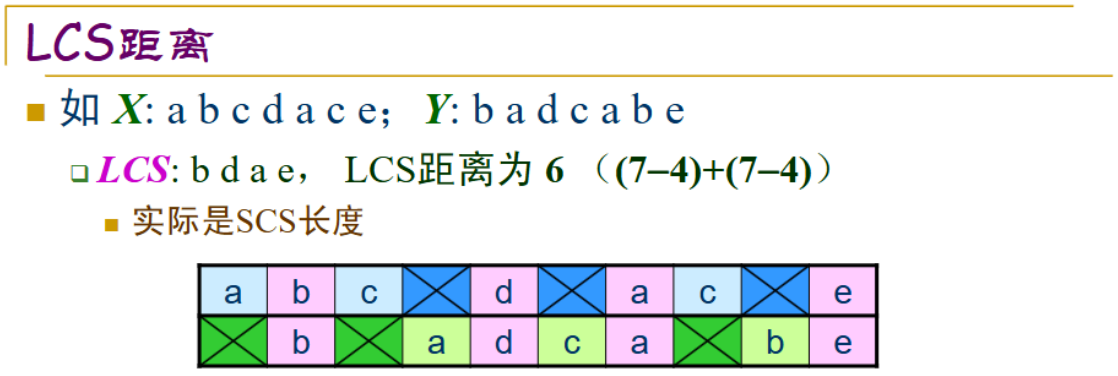
$\alpha_{TC}+\alpha_{GT}+\alpha_{AG}+2\alpha_{CA}$　　$2\delta+\alpha_{CA}$

Levenshtein距离

- Levenshtein 的原始定义：给定两个序列 S_1 和 S_2，通过一系列字符编辑（插入、删除、替换）等操作，将 S_1 转变成 S_2
- 完成这种转换所需要的最少的编辑操作个数称为 S_1 和 S_2 的编辑距离
- 在 Levenshtein 的原始定义中，插入、删除、替换操作中的每一个都具有单位成本，因此 Levenshtein 距离等于字符串转换的最小操作数
- 示例：vintner 转变成 writers，编辑距离 ≤ 5：

(操作最少的编辑操作个数)

> **Note**
>
> 用序列对齐去找



序列对齐：解的结构

- 定义 $OPT(i, j)$ 为将字符串 $x_1 x_2 \dots x_i$ 和 $y_1 y_2 \dots y_j$ 对齐的最小总开销
  - 情形1：最优方案选择将 x_i 和 y_j 进行匹配（可能会形成错误匹配）
    - 此时最小总开销为：x_i 和 y_j（可能形成的）错误匹配的惩罚（可能为0）+ 将字符串 $x_1 x_2 \dots x_{i-1}$ 和 $y_1 y_2 \dots y_{j-1}$ 对齐的最小总开销
  - 情形2-1：最优方案选择让 x_i 产生缺漏（无匹配）
    - 此时最小总开销为：一次缺漏惩罚 + 将字符串 $x_1 x_2 \dots x_{i-1}$ 和 $y_1 y_2 \dots y_j$ 对齐的最小总开销
  - 情形2-2：最优方案选择让 y_j 产生缺漏（无匹配）
    - 此时最小总开销为：一次缺漏惩罚 + 将字符串 $x_1 x_2 \dots x_i$ 和 $y_1 y_2 \dots y_{j-1}$ 对齐的最小总开销

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

LCS：$\delta = 0$；
$\alpha_{pq} = \infty$（$p \neq q$），$\alpha_{pp} = 0$

SCS：$\delta = 1$；
$\alpha_{pq} = \infty$（$p \neq q$），$\alpha_{pp} = 0$

伪代码：（序列对齐：算法）

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α)

1.  for i = 0 to m do
2.        OPT[0, i] = i×δ
3.  for j = 0 to n do
4.        OPT [j, 0] = j×δ
5.  for i = 1 to m do
6.    for j = 1 to n do
7.       OPT [i,j] = min { α[xi, yj] + OPT [i − 1, j − 1],
                           δ + OPT [i − 1, j],
                           δ + OPT [i, j − 1] }
8.  return OPT [m, n]
```

时间和空间复杂度：O(mn)

编辑距离（Edit Distance）

- 更一般的定义将非负权重函数 $w_{ins}(x)$、$w_{del}(x)$ 和 $w_{sub}(x, y)$ 与操作相关联
- 缺漏的惩罚不再简单地是 $\delta = 1$

图编辑距离（graph edit distance）

- 图编辑距离（GED）用以度量两个图之间的相似性（或相异性）
- 六种操作
  - 插入一个具有新标号的顶点
  - 删除一个（通常是孤立的）顶点
  - 替换顶点，即更改给定顶点的标号
  - 插入边
  - 删除边
  - 边替换，即更改给定边的标号

> **Note**
>
> 编辑距离是对LCS和SCS的一个应用，缺漏是SCS带来的，LCS是匹配上的，从而得到了（在该题下简化的）编辑距离。

EX8.矩阵链乘积（真有点难）

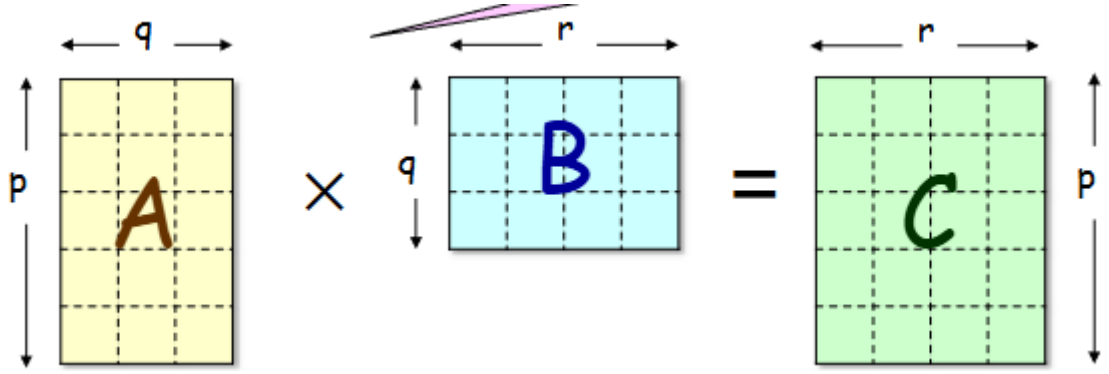
背景:

- 假定给定了矩阵的序列A1,A2,....,An
  - 其中Ai为Pi-1Pi阶矩阵
  - 于是Ai-1和Ai都是可以进行乘法的
- 目的是计算它们的链乘积A1A2...An
  - 然而每次**只**能是两个矩阵相乘得到第三个矩阵
- 由于矩阵乘法满足**结合律**，因此*无论计算的过程是什么样的*，其最终结果都是一样的

> **Note**
>
> **但是**不同的计算过程的效率可能是**不同**的

先来看一下两个矩阵相乘的复杂度

- 采用“教科书算法”



于是我们可以看到，总的元素乘法次数为pqr

```
输入：矩阵A_pXq 和 B_qXr (维数分别是pXq和qXr)
输出：矩阵C_pXr = A·B
MATRIX-MULTIPLY(A_pXq,B_qXr)
1.for i ← 1 to p
2.  for j ← 1 to r
3.      C(i,j) ← 0
4.      for k ← 1 to q
5.          C(i,j) ← C(i,j) + A(i,k)·B(k,j)
6.return C
```

问题定义：

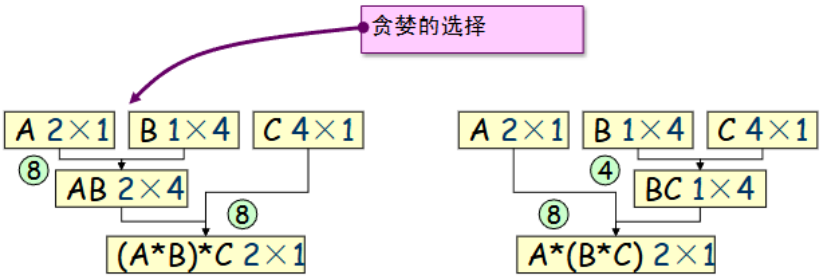
- 给定了矩阵的序列A1,A2,...An,其中Ai的阶数为Pi-1XPi
- 试确定矩阵相乘的次序（即，加括号的方法）使得**矩阵元素相乘的总次数最少**

> **Note**
>
> **不是真的计算乘积**，而是它们相乘的次数

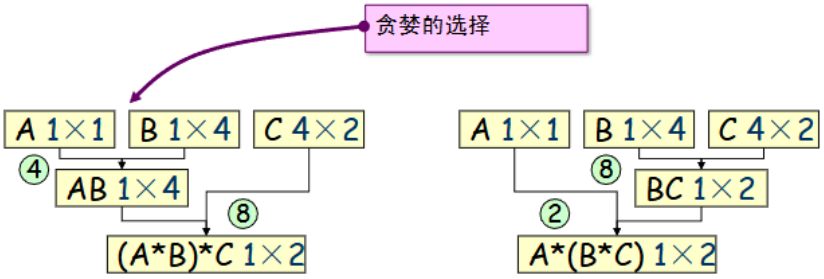
思路：

也是先考虑各种方法：

- 蛮力法？x 卡特兰数，数据量大了不可能找到解 时间复杂度为O(4^n)
- **每次选取最多元素乘法次数**的两个矩阵相乘？x
  - 反例：



- **每次选取最少元素乘法次数**的两个矩阵相乘？x
  - 反例：



- 找递推式：
  - 考虑**最后一次**乘法
    - 它将矩阵序列划分为两部分
  - 例子：
    1. $(A_1(A_2(A_3A_4)))$
    2. $(A_1((A_2A_3)A_4))$
    3. $((A_1A_2)(A_3A_4))$
    4. $((A_1(A_2A_3))A_4)$
    5. $(((A_1A_2)A_3)A_4)$
  - 将Ai·Ai+1·...·Aj的乘积记为Ai..j （j ≥ i）
    - 其中共有 l = j - i + 1 个矩阵

- 令m(i,j)表示计算Ai..j的最优方式的元素乘法总次数
- 当 j = i 时，m(i,j) = 0
- 假设计算Ai..j（j > i）的最后一次矩阵乘法是

$$A_{i..j} = A_{i..k} * A_{k+1..j}$$

其中 i ≤ k ≤ j

> **Note**
>
> Ai..k的阶数为Pi-1 X Pk, Ak+1..j的阶数为Pk Pi-1 X Pk
>
> 这次矩阵乘法的**开销**是Pi-1 X Pk X Pk
>
> 如果**在固定k的前提下**希望将总的元素乘法总次数降到最少，那么之前的每一次乘法就应该按照**最优方式**计算Ai..k和Ak+1..j
>
> （因为每次都是最优方式才能让最后的解也为最优解）
>
> 之前计算的Ai..k和Ak+1..j的最优方式的开销分别是m(i,k)和m(k+1,j)
>
> 于是总开销的最小可能就是
>
> $$m(i,k) + m(k+1,j) + P_{i-1} * P_k * P_j$$

但是我们怎么知道Ai..j的最优方案的最后一次矩阵乘法发生在**哪里**？即，k的值为多少

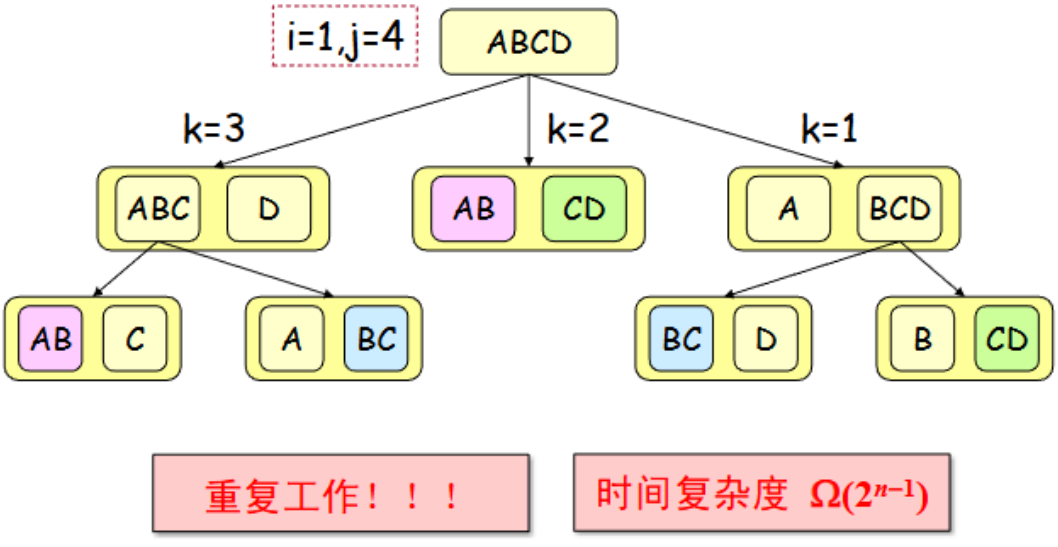
无论如何，最后一次乘法**必然会发生**在某个Ak“后面”

于是就把所有的 i ≤ k < j **都**试试看，然后选择其中“**最少**”的

最后得到递推式：

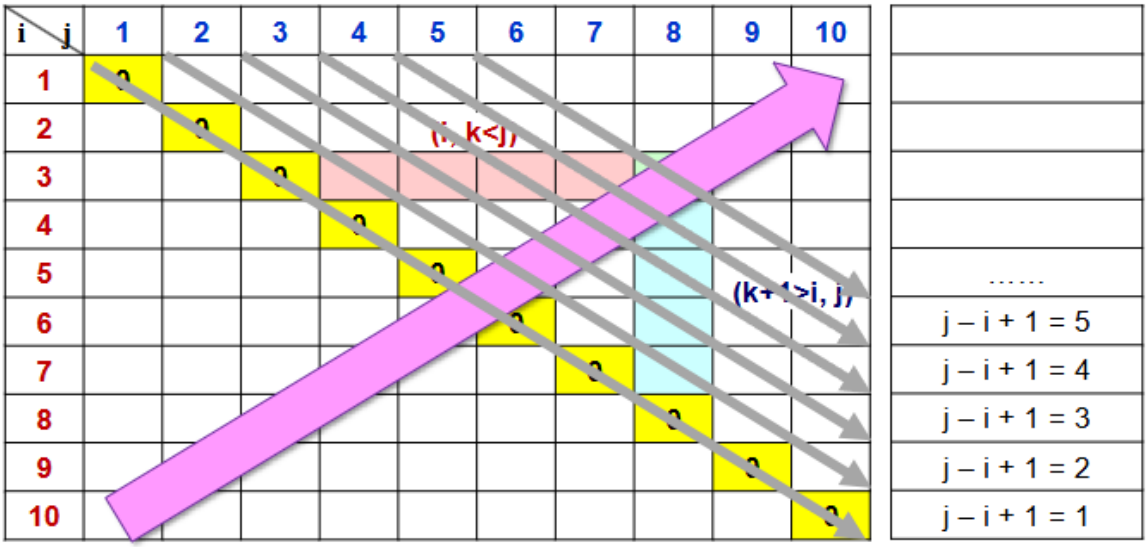
$$m(i,j) = \begin{cases} 0 & \text{if i} = \text{j} \\ \min\limits_{i \le k < j} \{m(i,k) + m(k+1,j) + P_{i-1} * P_k * P_j\} & \text{if i} < \text{j} \end{cases}$$

如果采用**递归**的方式：



因此采用**动态规划**：

$$m(i,j)=\begin{cases}0 & \text{if } i=j\\ \min\limits_{i\le k<j}\{m(i,k)+m(k+1,j)+P_{i-1}P_kP_j\} & \text{if } i<j\end{cases}$$



填写**最优开销表**m和**划分表**s

> **Note**
>
> s(i,j) = 0
>
> s(i,j) (j > i) 表示计算Ai..j 的最优方法的最后一次矩阵乘法发生的位置，即之前提及的k值

伪代码（重要）：

```
MATRIX-CHAIN-ORDER
输入：序列P0，P1,P2,...,Pn
输出：最优开销表m和划分表s
1.for i = 1 to n
2.    m(i,i) ← 0, s(i,i) ← 0 //将斜对角赋值为0
3.for l = 2 to n
4.    for i = 1 to n - l + 1 //从左上往右下角填写
5.        j ← i + l - 1
6.        m(i,j) ← ∞
7.        for k = i to j - 1
8.            q ← m(i,k) + m(k+1,j) + P(i-1)·P(k)·P(j)
9.            if(q < m(i,j))
10.               m(i,j) ← q, s(i,j) ← k
11.return m 和 s
```

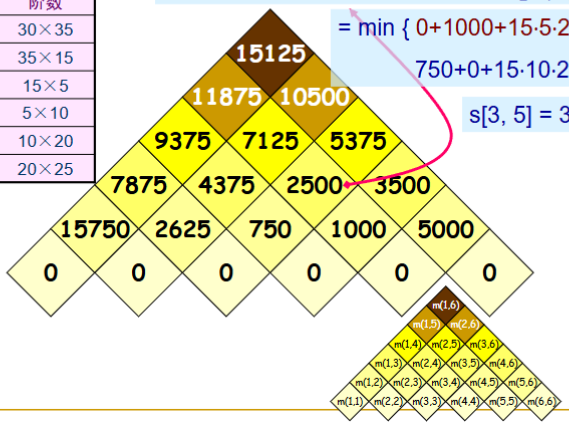
> **Important**
>
> **在复习的时候要注意好，i，j，k是如何遍历的，然后动态规划的填表时怎么进行的**

一些图示：

示例

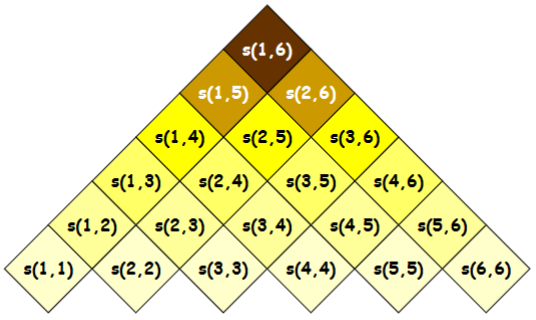
$m[3, 5] = \min \{ m[3, 3] + m[4, 5] + P_2P_3P_5,$
$m[3, 4] + m[5, 5] + P_2P_4P_5\}$
$= \min \{ 0+1000+15\cdot5\cdot20,$
$750+0+15\cdot10\cdot20\}$
$s[3, 5] = 3$

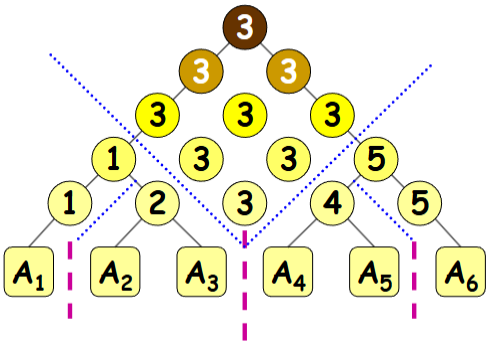
矩阵	阶数
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25



8

示例

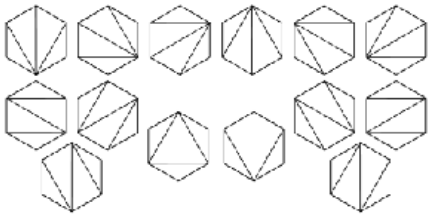




MORE:

更为有效的算法

- 1984年，Hu and Shing 发表了一个 $n \cdot \log n$ 复杂度的算法
- 基本思想：将矩阵链乘积问题转化（或称归约）为将凸多边形划分为不相交三角形的问题

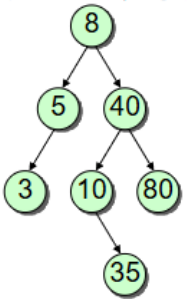


- 而后，他们设计了一个 $n \cdot \log n$ 时间的划分算法

EX9.最优二叉查找树

二叉查找树:

- 如果二叉树的任一顶点的键值都大于其非空左子树的所有顶点的键值，而小于其非空右子树的所有顶点的键值，则称其为一棵二叉查找树（Binary Search Tree，BST）
- 对一棵二叉查找树进行中序遍历，所得的键值序列一定是递增有序的

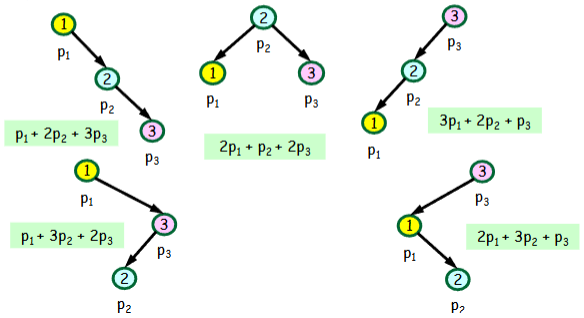


问题描述

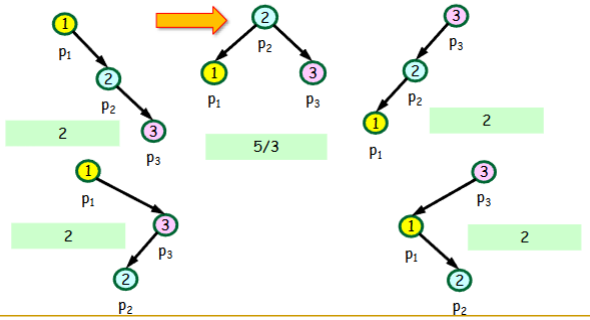
- 给定 n 个不同的键值 $k_1, k_2, \ldots, k_n$，每个键值 k_i 被访问的概率为 p_i
- 假设 $k_1 < k_2 < \ldots < k_n$
- $p_1 + p_2 + \ldots + p_n = 1$

- 应如何构建二叉查找树以最小化其成功查找的期望开销（比较次数）？

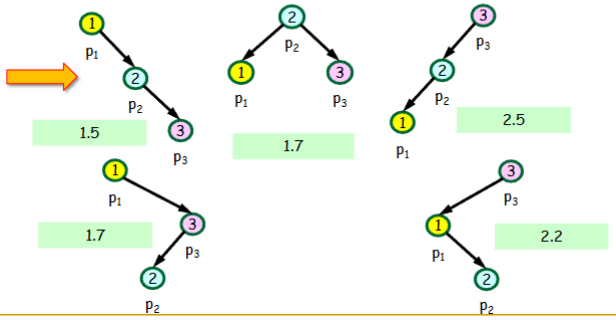
- 例1：键值为1、2和3，查找概率分别为 p_1、p_2 和 p_3。所有可能的5棵二叉查找树及其成功查找的期望开销（比较次数）为



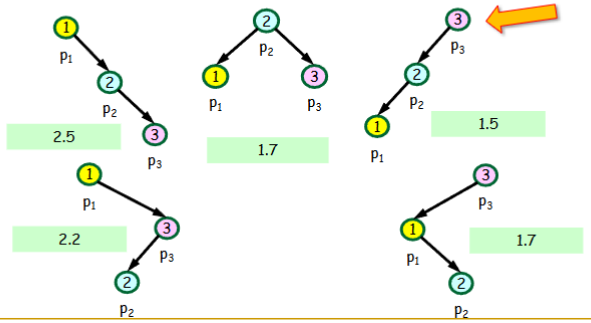
- 当 $p_1=p_2=p_3=1/3$ 时，第二棵树是最优的
  - 事实上，平衡树就是所有键值的查找概率都相等时的特例



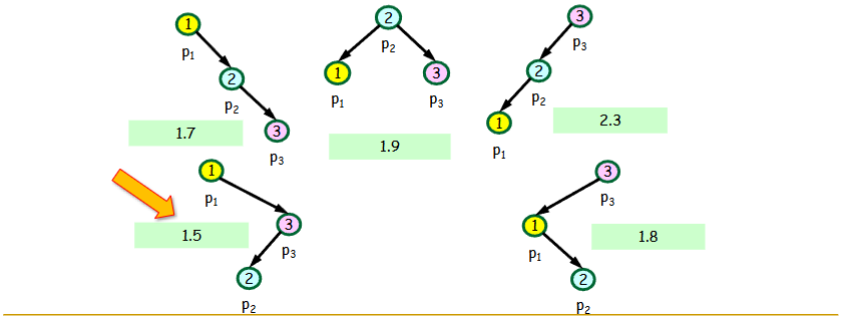
- 当 $p_1=0.6$、$p_2=0.3$、$p_3=0.1$ 时，第一棵树是最优的



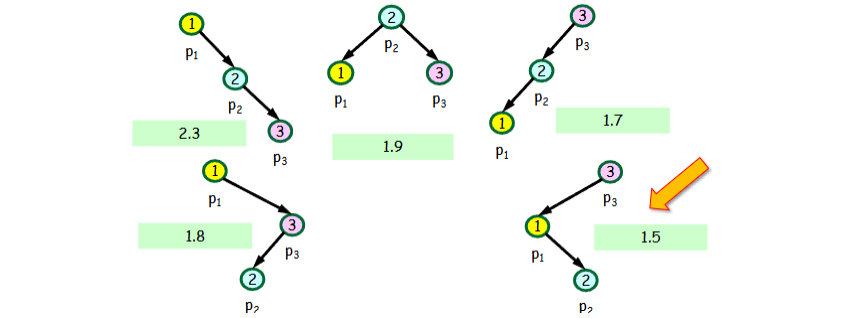
- 当 $p_1=0.1$、$p_2=0.3$、$p_3=0.6$ 时，第三棵树是最优的



- 当 p_1=0.6、p_2=0.1、p_3=0.3 时，第四棵树是最优的



- 当 p_1=0.3、p_2=0.1、p_3=0.6 时，第五棵树是最优的



通过上面这个例子我们可以看到，我们**不是**试图去构造一棵平衡树，而是要优化道路的**加权**长度

而且，此时**内部节点也包含键值**，并因"查找"而**固定**了树中节点的中序次序，因此和Huffman编码的问题背景和使用条件不同

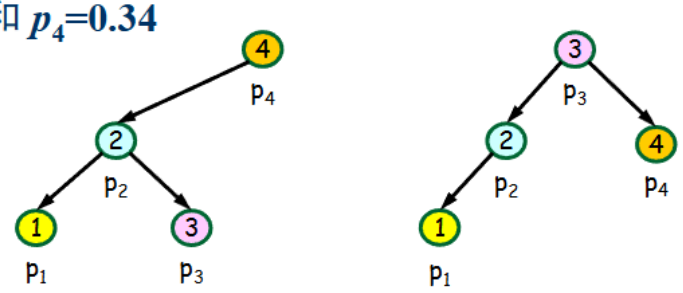
基本原则：**具有更大访问概率的键值的顶点应该更接近于根**

思路：

1.蛮力法？x 还是卡特兰数，得不出解

2.贪婪策略？将查找概率最大的键值作为根？x 反例：

- 反例：键值为1、2、3和4，查找概率分别为p_1=0.22、p_2=0.22、p_3=0.22 和 p_4=0.34



- 左侧树（查找概率最大的键值作为根）成功查找的期望开销为2.1，而右侧树成功查找的期望开销为2

3.递推关系：

记T(i,j)表示对应于键值ki,...,kj（及其对应的查找概率）的最优二叉查找树

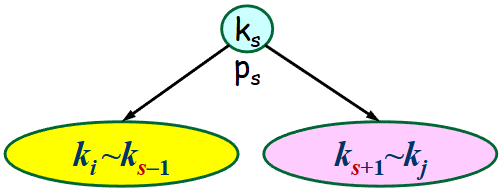
- 此时**不**要求pi+..+pj = 1

并令C(i,j)表示此时成功查找的（最优）期望开销（比较次数）

i > j时，T(i,j)为空树，C(i,j) = 0

考虑T(i,j)的**根**

如果根的键值时ks(i ≤ s ≤ j)，那么其左子树键值为ki~ks-1,右子树键值为ks+1~kj



于是此时

C(i , j) = ps X 1 +

cost_left + (pi + … + ps-1) X 1 +

cost_right + (ps + … + pj) X 1

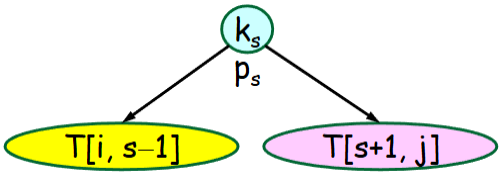
（cost_left 和 cost_right 是暂定名它们彼此的计算是独立）

C(i , j) = (pi + …. + pj) + cost_left + cost_right

（cost_left 和 cost_right 希望和达到最优（最小）值）

回过来

如果根的键值是ks,那么其左子树**一定**是T(i,s-1)，右子树**一定**是T(s,j) (i ≤ s ≤ j)



此时

C(i , j) = C(i , s - 1) + C(s+1, j) + (pi + … + pj)

之后就是对于**所有可能**的s计算其最小值

得到**递推关系**:

$$C(i,j) = \min_{(i \le s \le j)} \{C(i,s-1) + C(s+1,j) + (P_i + \ldots + P_j)\}$$

初值 i > j时，C(i , j) = 0

令 s(k) = p1+…+ pk , s(0) = 0 (s(k)表示第k个数后的所有概率总和)

- 可以用O(n)时间计算每一个s(k)

则可以使用s(j) - s(i-1) 计算（pi+…+pj）

最优二叉查找树 —— 问题扩展

- 问题描述
  - 给定 n 个不同键值 $k_1, k_2, \ldots, k_n$，每个键值 k_i 被访问的概率为 p_i
    - 假设 $k_1 < k_2 < \ldots < k_n$
  - 补充键值 $x_0 = -\infty$ 和 $x_{n+1} = +\infty$，查找概率都是0
  - 待查找键值落入开区间 (k_i, k_{i+1}) 的概率为 q_i
  - 于是有 $(p_1 + p_2 + \ldots + p_n) + (q_0 + q_1 + \ldots + q_n) = 1$
- 应如何构建二叉查找树以最小化其期望查找开销？
  - 包括查找成功和查找不成功的情况

EX10.跳棋棋盘

考虑一个nXn的方格棋盘

第i行第j列的方格的开销为c(i,j)

下图为5x5的棋盘示例 c(1,3) = 5

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

现在有一个棋子只能向左前方、正前方或者右前方跳一个，然后找总开销最小的方案。

定义函数q（i，j）为

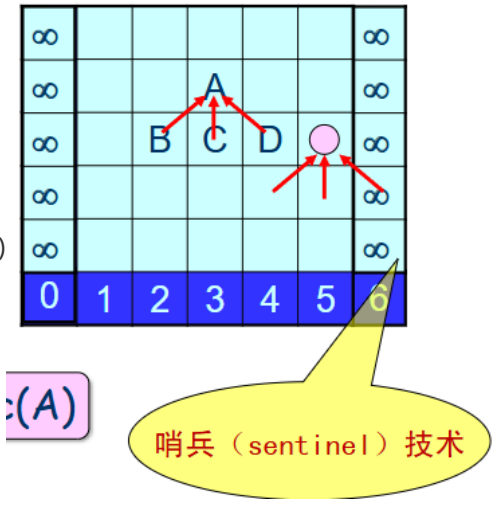
q（i，j） = 到达方格（i，j）的最小总开销

目标就是计算

$$min_{1 \le j \le n}\{q(n, j)\}$$

很容易就能得到：

q(A) = min{q(B),q(C),q(D)} + c(A)



伪代码:

```
ComputeShortestPathArrays //把每一条格的开销计算出来
1.for x = 1 to n
2.    q(1,x) ← ∞    //给初始的第一行赋值
3.q(1,(n+1)/2) ← c(1,x) //中间起始位置附上对应的成本
4.for y = 1 to n    //把最左和最右的两列赋值成∞，为了后面方便计算
5.    q(y,0) ← ∞
6.    q(y,n+1) ← ∞
7.for y = 2 to n
8.    for x = 1 to n
9.        m ← min{q(y-1,x-1),q(y-1,x),q(y-1,x+1)}
10.       q(y,x) ← m + c(y,x)
11.       if m = q(y-1,x-1) then  //记录道路
12.          p(y,x) ← -1
13.       else if m = q(y-1,x) then
14.          p(y,x) ← 0
15.       else
16.          p(y,x) ← 1
```

```
ComputeShortestPath
1.Call ComputeShortestPathArrays
2.minIndex ← 1 //总开销最小的列号
3.min ← q(n,1) //总开销的最小值
4.for i = 2 to n
5.    if q(n,1) < min then
6.       min ← q(n,i)
7.       minIndex ← i
8.Call PrintPath(n,minIndex)

PrintPath(y,x)      //自上而下的打印各行的列号
1.print(x)
2.print("<-")
3.if(y = 2) then
4.  print(x+p(x,y))
5.else
6.  Call PrintPath(y-1,x+p(y,x))
```

总结

- ❶ 刻画最优解的结构特性
  - P(X)
  - 例如 P(*n*), P(*n*, *w*)
- ❷ 将问题划分为子问题

$$P(\mathbf{X}) = \phi(f(P(\mathbf{X}-\mathbf{A}_1), \ldots, P(\mathbf{X}-\mathbf{A}_d)))$$

- 通常而言，ϕ 是 max{} 或者 min {}

- 示例
  - 斐波那契数：$F(n) = F(n-1)+F(n-2)$
  - 找零问题：$M(n) = \min \{ 1+M(n-d_i) \}$
  - 最长单调子序列：$L(i) = \max \{ L[j]+1 \ (1 \le j<i, S[j]<S[i]) \}$
  - 最大子段和：$C[j]=a_j$ or $a_j+C[j-1]$
  - 背包问题：$K(n, w) = \max \{ K(n-1, W-w_n) + v_n, K(n-1, W) \}$
  - 投资问题：$F(k, x) = \max \{ f(k, x_k) + F(k-1, x-x_k) \}$
  - LCS
    - $Len[X_{i-1}, Y_{j-1}] = len[X_{i-1}, Y_{j-1}]+1$ or $\max\{len[X_{i-1}, Y_j], len[X_i, Y_{j-1}] \}$
  - SCS
    - $Len[X_{i-1},Y_{j-1}] = len[X_{i-1},Y_{j-1}]+1$ or $\min\{len[X_{i-1},Y_j]+1, len[X_i,Y_{j-1}]+1 \}$
  - 编辑距离
    - $M[i, j] = \min(\alpha[x_i, y_i] + M[i-1, j-1], \delta + M[i-1, j], \delta + M[i, j-1])$
    - 矩阵链乘积
      - $C[i, j] = \min_{i \le k < j} \{ C[i, k] + C[k+1, j] + m_{i-1}m_k m_j \}$
    - 最优二叉查找树
      - $C(i, j) = \min_{i \le s \le j} \{ C(i, s-1) + C(s+1, j) + (p_i + \ldots + p_j) \}$
    - 跳棋棋盘
      - $q(A) = \min(q(B), q(C), q(D)) + c(A)$

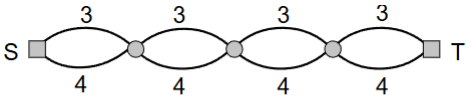
❸ 自底而上计算

$M(A_1)$	$M(A_2)$	....	$M(A_i)$	....	$M(A_n)$

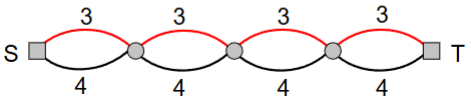
❹ 注意初值

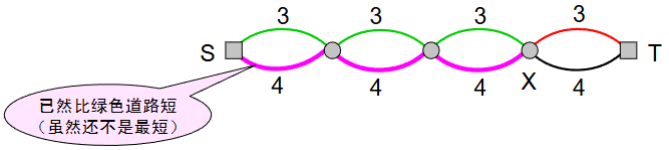
动态规划的基本要素:

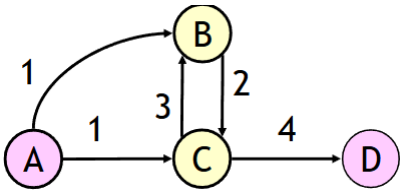
- 一个最优化多步决策问题适合用动态规划法求解有两个要素：**最优字结构特性**和重叠子问题。
- 最优子结构
  - 一个最优决策序列的任何子序列本身一定是相对于子序列的初始和结束状态的最优的决策序列
  - 一个问题的最优解总是包含所有子问题的最优解
  - 但**不是**说：如果有所有子问题的最优解，就可以**随便**把它们组合起来得到一个最优的解决方案
- 例如1、5找零问题
  - 凑成8元的最优解是5+1+1+1
  - 凑成9元的最优解是5+1+1+1+1
  - 但是凑成17元的最优解并**不是**5+1+1+1+5+1+1+1+1
  - 然而，的确**有一种方法**可以把凑成17元的问题的最优解分解为**子问题的最优解的组合**（例如，凑成15元+凑成2元）
  - 所有，找零问题满足最优子结构
- 但**不是**所有问题都满足最优子结构



- 【例】求总长模10的最短道路



- 

- 再如：**最长简单道路问题**（出租车敲竹杠问题）
- 右图中，从A到D的最长简单道路是ABCD 

- 但是，子道路AB**不是**从A到B的最长简单道路（ABC更长）
- 这个问题**不满足**最优性原则
- 因此，最长简单道路问题**不能**用动态规划方法解决