设计与分析

第四章 动态规划
【二、三】

动态规划法求解最长递增子序列问题

【问题描述】

给定一个无序的整数序列$a[0..n-1]$，求其中最长递增子序列的长度。

例如，$a[]=\{2，1，5，3，6，4，8，9，7\}$，$n=9$，其最长递增子序列为{1，3，4，8，9}，结果为5。

动态规划法求解最长递增子序列问题

【问题求解】设计动态规划数组为一维数组dp，dp[i]表示a[0..i]中以a[i]结尾的最长递增子序列的长度。对应的状态转移方程如下：

$$dp[i]=1 \qquad 0\le i\le n-1$$

$$dp[i]=\max(dp[i], dp[j]+1) \quad 若a[i]>a[j],\ 0\le i\le n-1,\ 0\le j\le i-1$$

求出dp后，其中最大元素即为所求。

这里的dp[j]是指对于所有的前j+1个元素的最长递增子序列的长度。
即：当检视a[i]时，就看看对于每一个dp[j],在有了a[i]后，是否会增加递增子序列的长度。

动态规划法求解长递增子序列示例

下标	0	1	2	3	4	5	6	7	8
a[]	2	1	5	3	6	4	8	9	7

初始的dp[i]=1(0≤i≤n-1)

递推求解状态方程：

dp[1]=1(因为a[1](1)<a[0](2),所以dp[1]保持初值

a[2](5)>a[0](2):dp[2]=max(dp[2],dp[0]+1)=2

a[2](5)>a[1](1):dp[2]=max(dp[2],dp[1]+1)=2

最终：dp[2]=2

a[3](3)>a[0](2):dp[3]=max(dp[3],dp[0]+1)=2

a[3](3)>a[1](1): dp[3]=max(dp[3],dp[1]+1)=2

a[3](3)<a[2](5): dp[3] =2(因为a[3](3)<a[2](5),所以dp[3]维持原值 最终dp[3]=2

a[4](6)>a[0](2):dp[4]=max(dp[4],dp[0]+1)=2

a[4](6)>a[1](1): dp[4]=max(dp[4],dp[1]+1)=2

a[4](6)>a[2](5): dp[4]=max(dp[4],dp[2]+1)=3

a[4](6)>a[3](3):dp[4]=max(dp[4],dp[3]+1)=3

最终dp[4]=3

动态规划法求解长递增子序列示例

下标	0	1	2	3	4	5	6	7	8
a[]	2	1	5	3	6	4	8	9	7

递推求解状态方程：

a[5](4)>a[0](2):dp[5]=max（dp[5],dp[0]+1)=2

a[5](4)>a[1](1): dp[5]=max(dp[5],dp[1]+1）=2

a[5](4)<a[2](5): 此次dp[5]不更新

a[5](4)>a[3](3): dp[5]=max(dp[5],dp[3]+1)=3

a[5](4)<a[4](2):此次dp[5]不更新

最终： dp[5]=3

a[6](8)>a[0](2):dp[6]=max（dp[6](1),dp[0]+1)=2

a[6](8)>a[1](1): dp[6]=max(dp[6](2),dp[1]+1）=2

a[6](8)>a[2](5): dp[6]=max(dp[6](2）,dp[2]+1）=3

a[6](8)>a[3](3): dp[6]=max(dp[6]（3）,dp[3]+1)=3

a[6](8)<a[4](6):dp[6]=max(dp[6]（3）,dp[4](3)+1)=4

a[6](8)<a[5](4):dp[6]=max(dp[6]（4）,dp[5](3)+1)=4

最终： dp[6]=4

下标	0	1	2	3	4	5	6	7	8
a[]	2	1	5	3	6	4	8	9	7

递推求解状态方程：

同理：计算出dp[7]=5

dp[8]=4

dp

1	1	2	2	3	3	4	5	4

Max(dp)=5

所以该序列最长递增子序列长度为5

求解最长递增子序列问题

```
//问题表示
int a[]={2, 1, 5, 3, 6, 4, 8, 9, 7};
int n=sizeof(a)/sizeof(a[0]);
//求解结果表示
int ans=0;
int dp[MAX];
void solve(int a[], int n)
{  int i, j;
   for(i=0;i<n;i++)
   {  dp[i]=1;
      for(j=0;j<i;j++)
      {  if (a[i]>a[j])
            dp[i]=max(dp[i], dp[j]+1);
      }
   }
   ans=dp[0];
   for(i=1;i<n;i++)
     ans=max(ans, dp[i]);}
```

求解最长递增子序列问题

【算法分析】solve()算法中含两重循环，时间复杂度为O(n^2)。

【练习题】：如何根据dp[]求出在a数组中的最长递增子序列？（所有可能的）

合唱队形问题

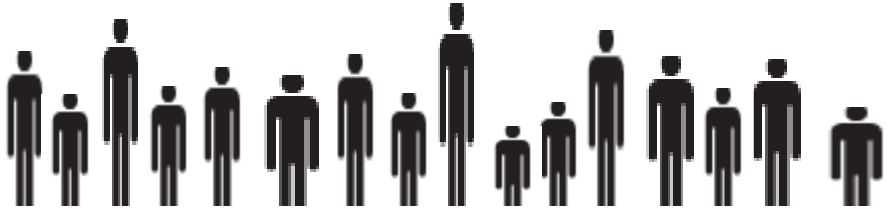
- 建立在线性结构或图结构的基础之上的**线性动态规划算法**，可解决的问题具有**线性递推**的特点，通常以某一结点为状态，**向两个方向：即由前向后或者由后向前线性遍历每个状态，从中找出具有最优值的状态从而得到问题的解。**

1. 【问题描述】

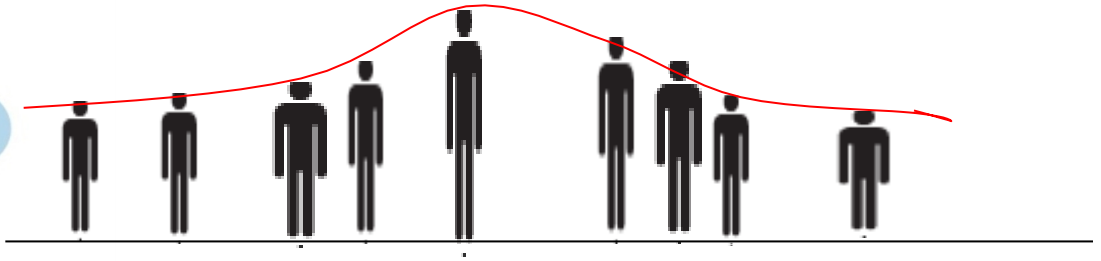
N位同学站成一排，音乐老师要请其中的(N-K)位同学出列，而不改变其他同学的位置，使得剩下的K位同学排成**合唱队形**。

合唱队形问题

- 1. 问题描述合唱队形要求：设K位同学从左到右依次编号为1，2，...，K，他们的身高分别为T1,T2，...，TK，则他们的身高满足T1<...<Ti,且Ti>Ti+1>...>TK(1<=i<=K)。当给定队员人数N和每个学生的身高T[i]时，计算需要多少学生出列，可以得到最长的合唱队形。如下图所示：



编号: 1 2 3 4 5… …… i-1 i i+1….. N



合唱队形问题-----分析最优子结构

- 2. 分析最优子结构
- 假设第i位同学为最高个，则对其左边序列求最长递增序列长度，对其右边序列求最长递减序列长度，然后两者相加再减1（因为第i位同学被重复计算了一次）即可得到整个合唱队形的长度。

- 由此可见，原问题的解即最长的合唱队形取决于*最大上升子序列*和*最大下降子序列的长度*，具有最优子结构的性质。由于在某一时刻，它是单向的从一个状态线性递推到下一个状态，属于线性动态规划问题。

合唱队形问题

————————建立递归关系

3. 建立递归关系

当组成最大上升子序列时，得到递归方程：

$f1(i)=max\{f1(j)+1\}$ $(j<i, Tj<Ti)$

$f1(1)=1$;//递归出口

设f2(i)为后面N-i+1位排列的最大下降子序列长度，用同样的方法可以得到递归方程：

$f2(i)=max\{f2(j)+1\}$ $(i<j, Tj<Ti)$

$f2(N)=1$; //边界值

合唱队形问题
———— 计算最优值

4. 计算最优值

首先用数组a保存所有人的身高，第一遍正向扫描，从左到右求最大上升子序列的长度，然后反向扫描，从右到左求最大下降子序列的长度，然后依次枚举由前i个学生组成的最大上升子序列的长度和由后N-i+1个学生组成的最大下降子序列的和，则N次枚举后得到N个合唱队形的长度，取其中的最大值，然后用学生总数n减去该最大值即可得到原问题的解。

例3-3：已知8个学生的身高：176、163、150、180、170、130、167、160，计算他们所组成的最长合唱队形的长度。

合唱队形问题 ------示例

下标	1	2	3	4	5	6	7	8
a	176	163	150	180	170	130	167	160

a数组用于存放身高值

根据递归方程

f1[1]=1;//递归出口

f1[i]=max{f1[j]+1} (j<i,1<=j<=i-1,a[j]<a[i]

从左到右求最大递增子序列长度，构造f1[i]

初始所有的f1[i]=1

f1[1]=1

计算f1[2](j从1~i-1即2-1=1）计算f1[2]

因为a[j](a[1](176)并不小于a[2](163)所以f1[2]维持初值，f1[2]=1

计算f1[3](j从1~i-1即3-1=2）计算f1[3]

因为a[j](a[1](176)并不小于a[3](150)所以f1[3]维持初值1

因为a[j](a[2](163)并不小于a[3](150)所以f1[3]维持初值1，f1[3]=1

计算f1[4](j从1~i-1即4-1=3）计算f1[4]

a[1]<a[4](180) f1[4]=max(f1[4](1),f1[1](1)+1)=2

a[2]<a[4](180) f1[4]=max(f1[4](2),f1[2](1)+1)=2

a[3]<a[4](180) f1[4]=max(f1[4](2),f1[3](1)+1)=2, f1[4]=2

合唱队形问题

------示例

下标	1	2	3	4	5	6	7	8
a	176	163	150	180	170	130	167	160

a数组用于存放身高值

根据递归方程

f1[1]=1;//递归出口

f1[i]=max{f1[j]+1} (j<i,1<=j<=i-1,a[j]<a[i]

从左到右求最大递增子序列长度，构造f1[i]

计算f1[5](j从1~i-1即5-1=4）计算f1[5]

a[1]不<a[5](170) f1[5]不变，维持原值1f1[5]=1;

a[2]<a[5](170) f1[5]=max(f1[5](1),f1[2](1)+1)=2

a[3]<a[5](170) f1[5]=max(f1[4](2),f1[3](1)+1)=2

a[4]不<a[5](170) f1[5]不变，维持f1[5]=2;最终：f1[5]=2

计算f1[6](j从1~i-1即6-1=5）计算f1[6]

因为j从1~5所有的a[j]均大于a[6](130),所以最终f1[6]=1

合唱队形问题

------示例

下标	1	2	3	4	5	6	7	8
a	176	163	150	180	170	130	167	160

a数组用于存放身高值

根据递归方程

f1[1]=1;//递归出口

f1[i]=max{f1[j]+1} (j<i,1<=j<=i-1,a[j]<a[i]

从左到右求最大递增子序列长度，构造f1[i]

计算f1[7](j从1~i-1即7-1=6）计算f1[7]

a[1]不<a[7](167) f1[7]不变，维持原值1f1[7]=1;

a[2]<a[7](170) f1[5]=max(f1[7](1),f1[2](1)+1)=2

a[3]<a[7](167) f1[7]=max(f1[7](2),f1[3](1)+1)=2

a[4]不<a[7](167) f1[7]不变，维持f1[7]=2

a[5]不<a[7](167) f1[7]不变，维持f1[7]=2

a[6]<a[7](167) f1[7]=max(f1[7](2),f1[6](1)+1)=2

所以最终f1[7]=2

合唱队形问题

------示例

下标	1	2	3	4	5	6	7	8
a	176	163	150	180	170	130	167	160

a数组用于存放身高值

根据递归方程

f1[1]=1;//递归出口

f1[i]=max{f1[j]+1} (j<i,1<=j<=i-1,a[j]<a[i]

从左到右求最大递增子序列长度，构造f1[i]

计算f1[8](j从1~i-1即8-1=7）计算f1[8]

a[1]不<a[8](160) f1[8]不变，维持原值1f1[8]=1;

a[2]不<a[8](160) f1[8]不变，维持原值1f1[8]=1;

a[3]<a[8](160) f1[8]=max(f1[8](1),f1[3](1)+1)=2

a[4]不<a[8](160) f1[8]不变，维持f1[8]=2

a[5]不<a[8](160) f1[8]不变，维持f1[8]=2

a[6]<a[8](160) f1[8]=max(f1[8](2),f1[6](1)+1)=2

a[7]不<a[8](160) f1[8]不变，维持f1[8]=2,最终：f1[8]=2

	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
f1[i]	1	1	1	2	2	1	2	2

合唱队形问题

------示例

下标	1	2	3	4	5	6	7	8
a	176	163	150	180	170	130	167	160

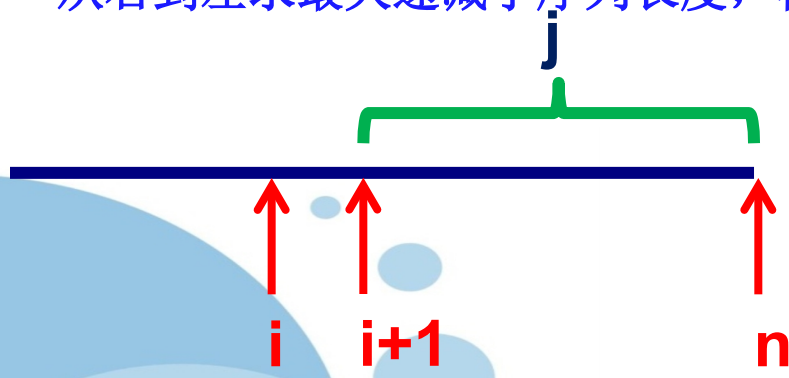
a数组用于存放身高值

设f2(i)为后面N-i+1位排列的最大下降子序列长度，用同样的方法可以得到递归方程：（即倒着从n~i求最大递增子序列）

f2(N)=1; //边界值

f2[i]=max{f2[j]+1} (i<j,i+1<=j<=n,a[j]<a[i])

从右到左求最大递减子序列长度，构造f1[i]



$$f2[i]=\begin{cases} f2[i]; \text{若} a[i]<a[j] \\ Max(f2[i],f2[j]+1); \text{若} a[i]>a[j] \end{cases}$$

合唱队形问题

------示例

下标	1	2	3	4	5	6	7	8
a	176	163	150	180	170	130	167	160

a数组用于存放身高值

设f2(i)为后面N-i+1位排列的最大下降子序列长度，用同样的方法可以得到递归方程：（即倒着从n~i求最大递增子序列）

f2(N)=1; //边界值

f2[i]=max{f2[j]+1} (i<j,i+1<=j<=n,a[j]<a[i])

从右到左求最大递减子序列长度，构造f1[i]

初始所有的f2[i]=1

从i=8开始，（j从i+1~8即j从8+1~n(8)）循环不执行,f2[8]=1

计算f2[7]:j从8~8

因为a[7]>a[8] f2[7]=max(f2[7],f2[8]+1)=2;f2[7]=2

计算f2[6]:j从7~8

a[6](130)不>a[7]

a[6](130)不>a[8],f2[6]维持初值； f2[6]=1

合唱队形问题

------示例

下标	1	2	3	4	5	6	7	8
a	176	163	150	180	170	130	167	160

a数组用于存放身高值

设f2(i)为后面N-i+1位排列的最大下降子序列长度，用同样的方法可以得到递归方程：（即倒着从n~i求最大递增子序列）

f2(N)=1; //边界值

f2[i]=max{f2[j]+1} (i<j,i+1<=j<=n,a[j]<a[i])

从右到左求最大递减子序列长度，构造f1[i]

计算f2[5]:j从6~8

a[5](170) >a[6], f2[5]=max(f2[5],f2[6]+1)=2

a[5](170) >a[7],f2[5]=max(f2[5],f2[7]+1)=3

a[5](170) >a[8],f2[5]=max(f2[5],f2[8]+1)=3,最终f2[5]=3

计算f2[4]:j从5~8

a[4](180) >a[5],f2[4]=max(f2[4],f2[5]+1)=4

a[4](180) >a[6],f2[4]=max(f2[4],f2[6]+1)=max(4,2)=4

a[4](180) >a[7],f2[4]=max(f2[4],f2[7]+1)=max(4,3)=4

a[4](180) >a[8],f2[4]=max(f2[4],f2[8]+1)=max(4,2)=4

最终f2[4]=4

合唱队形问题

------示例

下标	1	2	3	4	5	6	7	8
a	176	163	150	180	170	130	167	160

a数组用于存放身高值

设f2(i)为后面N-i+1位排列的最大下降子序列长度，用同样的方法可以得到
递归方程：（即倒着从n~i求最大递增子序列）
f2(N)=1; //边界值
f2[i]=max{f2[j]+1} (i<j,i+1<=j<=n,a[j]<a[i])
从右到左求最大递减子序列长度，构造f1[i].

计算f2[3]:j从4~8
a[3](150)不大于a[4]（180） f2[3]维持初值1;
a[3](150)不大于a[5] （170） f2[3]维持初值1;
a[3](150) >a[6],f2[3]=max(f2[3],f2[6]+1)=2;
a[3](150)不大于a[7] f2[3]维持值2;
a[3](150)不大于a[8] f2[3]维持值2; 最终f2[3]=2
计算f2[2]:j从3~8
a[2](163)>a[3],f2[2]=max(f2[2],f2[3]+1)=3;
a[2](163）不大于a[4],f2[2]维持值3; a[2](163）不大于>a[5],f2[2]维持值3;
a[2](163)>a[6],f2[2]=max(f2[2],f2[6]+1)=max（3,2）=3;
a[2](163)不大于>a[7],f2[2]维持值3;
a[2](163)>a[8],f2[2]=max(f2[2],f2[8]+1)=max(3,2)=3;最终f2[2]=3

合唱队形问题

------示例

下标	1	2	3	4	5	6	7	8
a	176	163	150	180	170	130	167	160

a数组用于存放身高值

设f2(i)为后面N-i+1位排列的最大下降子序列长度，用同样的方法可以得到
递归方程：（即倒着从n~i求最大递增子序列）
f2(N)=1; //边界值
f2[i]=max{f2[j]+1} (i<j,i+1<=j<=n,a[j]<a[i])
从右到左求最大递减子序列长度，构造f1[i].

计算f2[1]:j从2~8
a[1](176)>a[2],f2[1]=max(f2[1],f2[2]+1)=max(1,3+1)=4;
a[1](176）>a[3]; f2[1]=max(f2[1],f2[3]+1)=max(4,2+1)=4;
a[1](176)不大于>a[4],f2[1]维持值4;
a[1](176)>a[5],f2[1]=max(f2[1],f2[5]+1)=max（4,4）=4;
a[1](176)>a[6],f2[1]=max(f2[1],f2[6]+1)=max（4,2）=4;
a[1](176)>a[7],f2[1]=max(f2[1],f2[7]+1)=max（4,3）=4;
a[1](176)>a[8],f2[1]=max(f2[1],f2[8]+1)=max（4,2）=4;
最终f2[1]=4

合唱队形问题

计算最优值

4. 计算最优值

先从左到右求最大上升子序列的长度，通过比较8个同学的身高，得到f1表如3-3（a）所示，然后从右到左求最大下降子序列的长度，通过比较8个同学的身高，得到f2如表3-3（b）所示，最后将两个表中对应的元素相加减1（两表中的值相加，第i位同学被重复计算了一次），得到最终问题的解表如3-3（c）所示：

表3-3合唱队形问题动态规划表

(a) 最大上升子序列的长度表f1

	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
f1[i]	1	1	1	2	2	1	2	2

(b) 最大下降子序列的长度表f2

	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
f2[i]	4	3	2	4	3	1	2	1

合唱队形问题

程序描述及算法分析

5. 程序描述及算法分析

(c) 最终结果表ans

	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
ans	4	3	2	5	4	1	3	2

由此可见，当i取4，即以身高为180的同学为中心位置，所形成的合唱队形的长度最长为5，需要8-5=3个人出列，最终形成的合唱队形为：176,180,170,167,160。

【算法3.3 用动态规划求解合唱队形问题】

```
{/*功能：从n个同学中取出k个，求他们所组成的合唱队形
  输入：队员人数n和每个学生身高a[i]
  输出：最长的合唱队形的长度ans*/
```

3 合唱队形问题 ----------算法描述及算法分析

```
int* ans  ChorusRank(int n, int a[100])
{
 for (int i = 1; i <= n; i ++)
  {  int f1 [maxn]; int f2 [maxn];
    f1[i] = 1;
     for (int j = 1; j < i; j ++)
         if ( a[j] < a[i] && f1[i] < f1[j] + 1 ) f1[i] = f1[j] + 1;// 从左到右求最大上升子序列
    }

 for (int i = n; i >= 1; i --)
   {  f2[i] = 1;
     for (int j = i + 1; j <= n; j ++)
       if ( a[j] < a[i] && f2[i] < f2[j] + 1 ) f2[i] = f2[j] + 1; //从右到左求最大下降子序列
     }
int ans = 0;
for (int i = 1; i <= n; i ++)
 if ( ans < f1[i] + f2[i]-1 )
 ans = f1[i] + f2[i] - 1; //枚举中间最高值
return ans;   //返回最长合唱队形的长度
}
```

算法分析

由于解决该问题时使用了两次动态规划方法来求解，即第一次求最大上升子序列的长度，第二次求最大下降子序列的长度，再枚举中间最高的一个人所在队形的长度。算法实现所需的时间复杂度0(n^2)，空间复杂度为0(n)。

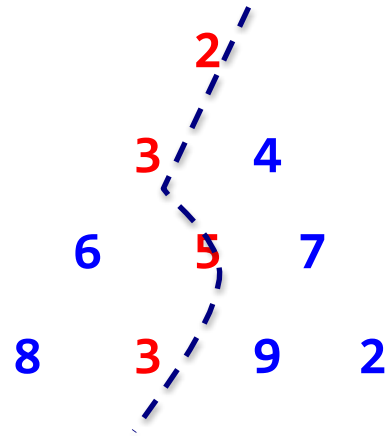
动态规划法求解三角形最小路径问题

【问题描述】给定高度为n的一个整数三角形，找出从顶部到底部的最小路径和，只能向下移动到相邻的整数结点。

首先：输入n，接下来的1～n行，第i行输入i个整数，

输出：分为2行，第一行为最小路径，第2行为最小路径和。

例如，下图是一个n=4的三角形，输出的路径是2 3 5 3，最小路径和是13。



动态规划法求解三角形最小路径问题

【问题求解】将三角形采用二维数组a存放，前面的三角形对应的二维数组如下：

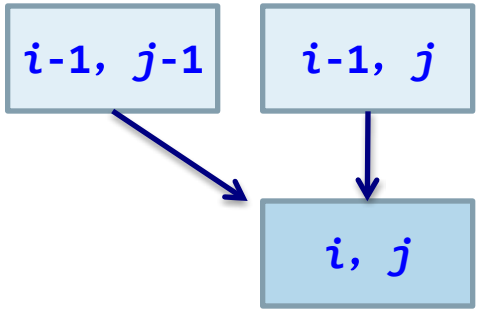
2

3 4

6 5 7

8 3 9 2

从顶部到底部查找最小路径，那么结点（i，j）的前驱结点只有（i-1，j-1）和（i-1，j）两个：

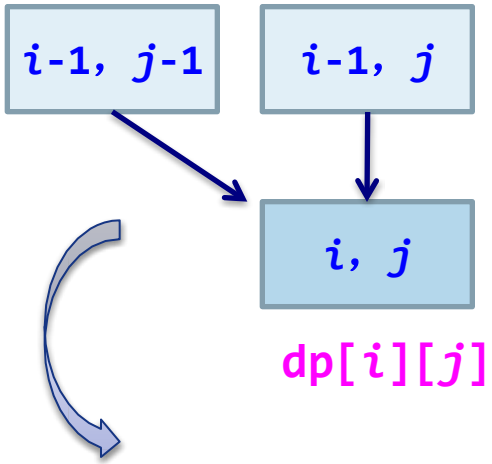


动态规划法求解三角形最小路径问题

dp[i][j]表示从顶部a[0][0]查找到（i，j）结点时的最小路径和。

- **一般情况：**

dp[i-1][j-1]　dp[i-1][j]



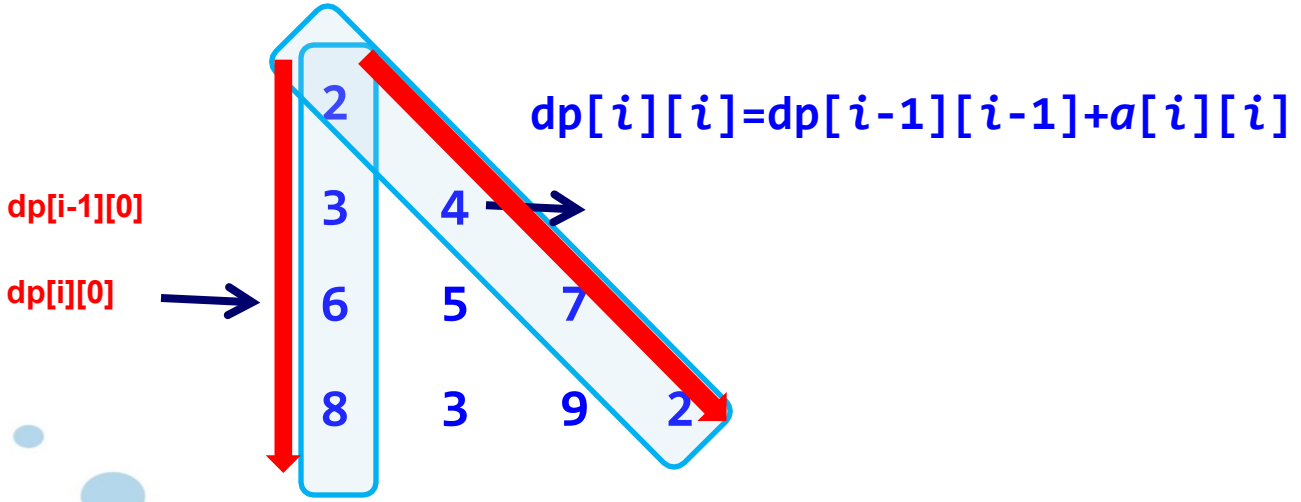
dp[i][j]

dp[i][j]=min(dp[i-1][j-1]，dp[i-1][j])+a[i][j]

动态规划法求解三角形最小路径问题

dp[i][j]表示从顶部a[0][0]查找到（i，j）结点时的最小路径和。

- **特殊情况：两个边界，即第1列和对角线，达到它们中结点的路径只有一条而不是常规的两条。**



dp[i][0]=dp[i-1][0]+a[i][0]

动态规划法求解三角形最小路径问题

状态转移方程如下：

$dp[0][0]=a[0][0]$　　顶部边界

$dp[i][0]=dp[i-1][0]+a[i][0]$　　考虑第1列的边界，$1\le i\le n-1$

$dp[i][i]=dp[i-1][i-1]+a[i][i]$　　考虑对角线的边界，$1\le i\le n-1$

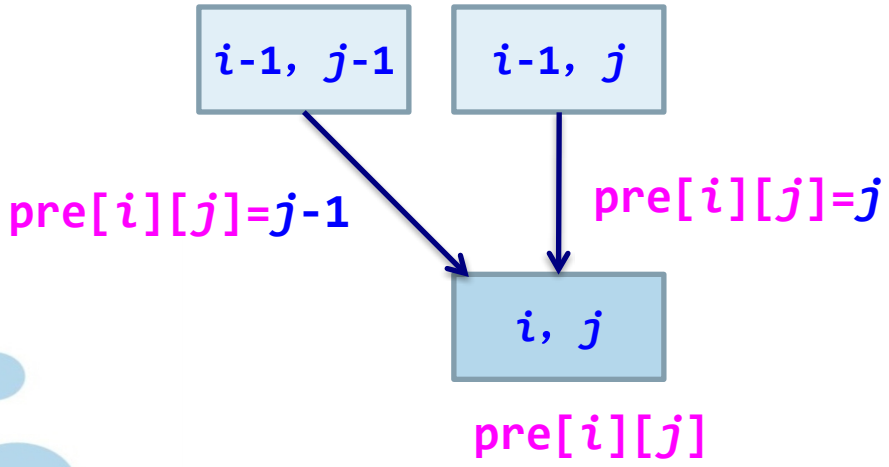
$dp[i][j]=\min(dp[i-1][j-1], dp[i-1][j])+a[i][j]$　　$i>1$的其他有两条达到路径的结点

最后求出的最小路径和ans=min(dp[n-1][j])（$0\le j<n$）。

动态规划法求解三角形最小路径问题

找到的最短路径即dp[i][j]的路径输出问题

用pre[i][j]表示查找到（i，j）结点时最小路径上的前驱结点，由于前驱结点只有两个，即（i-1，j-1）和（i-1，j），用pre[i][j]记录前驱结点的列号即可。



在求出ans后，通过pre[n-1][k]反推求出反向路径，最后正向输出该路径。

动态规划法求解三角形最小路径问题

```
//问题表示
int a[MAXN][MAXN];
int n;
//求解结果表示
int ans=0;
int dp[MAXN][MAXN];//计从a[0][0]出发到a[i][j]的最短路径
int pre[MAXN][MAXN];//计走到a[i][j]的直接前驱结点的列号
```

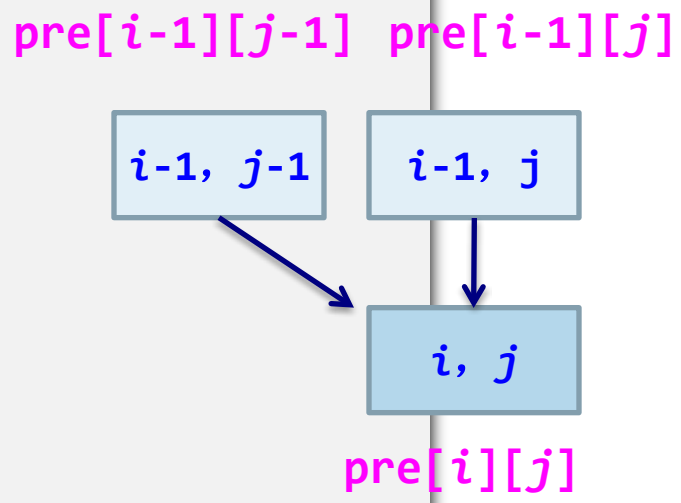
动态规划法求解三角形最小路径问题

```
int Search()                    //求最小和路径ans
{  int i, j;
   dp[0][0]=a[0][0];
   for(i=1;i<n;i++)             //考虑第1列的边界
   {  dp[i][0]=dp[i-1][0]+a[i][0];
     // pre[i][0]=i-1;//记录的是上一行的行号，列号仍为0
                              pre[i][0]=0;//应该是这个才对
   }
   for (i=1;i<n;i++)                    //考虑对角线的边界
   {  dp[i][i]=a[i][i]+dp[i-1][i-1];
      pre[i][i]=i-1;
   }
```

动态规划法求解三角形最小路径问题

求最小和路径上每个结点的前驱结点

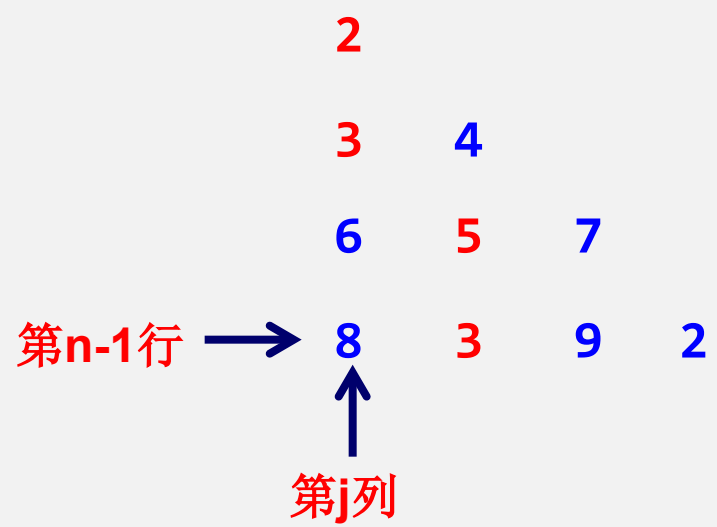
```
for(i=2;i<n;i++)                    //考虑其他有两条达到路径的结点
{   for(j=1;j<i;j++)
    {   if(dp[i-1][j-1]<dp[i-1][j])
        {   pre[i][j]=j-1;
            dp[i][j]=a[i][j]+dp[i-1][j-1];
        }
        else
        {   pre[i][j]=j;
            dp[i][j]=a[i][j]+dp[i-1][j];
        }
    }
}
```



动态规划法求解三角形最小路径问题

返回最小和路径最后行的列号k

```
ans=dp[n-1][0];
int k=0;
for (j=1;j<n;j++)          //求出最小ans和对应的列号k
{  if (ans>dp[n-1][j])
   {  ans=dp[n-1][j];
      k=j;
   }
}
return k;
}
```



动态规划法求解三角形最小路径问题

```
void Disppath(int k)              //输出最小和路径
{  int i=n-1;
   vector<int> path;              //存放逆路径向量path
   while (i>=0)                   //从（n-1,k）结点反推求出反向路径
   {    path.push_back(a[i][k]);
        k=pre[i][k];              //最小路径在前一行中的列号
        i--;                      //在前一行查找
   }
   vector<int>::reverse_iterator it;          //定义反向迭代器
   for (it=path.rbegin();it!=path.rend();++it)
      printf("%d ",*it);          //反向输出构成正向路径
   printf("\n");
}
```

动态规划法求解三角形最小路径问题

```
int main()
{  int k;
   memset(pre, 0, sizeof(pre));
   memset(dp, 0, sizeof(dp));
   scanf("%d", &n);                          //输入三角形的高度
   for (int i=0;i<n;i++)                     //输入三角形
      for (int j=0;j<=i;j++)
         scanf("%d", &a[i][j]);
   k=Search();                               //求最小路径和
   Disppath(k);                              //输出正向路径
   printf("%d\n", ans);                      //输出最小路径和
   return 0;
}
```

【算法分析】Search()算法中有i从2到n-1、j从1到i-1的两重循环，容易求出时间复杂度为$O(n^2)$。

动态规划求解最长公共子序列问题

【问题描述】字符序列的子序列是指从给定字符序列中随意地（不一定连续）去掉若干个字符（可能一个也不去掉）后所形成的字符序列。

x_{i_j}

孩子有多像爸爸？最长公共子序列问题

假设爸爸的基因序列为X={x0,x1,x2,…xm-1},孩子对应的基因序列Y={y0,y1,y2,…yn-1}那么怎么找到他们有多少相似的基因呢？

如果按照严格递增的顺序，从爸爸的基因序列X中取一些值，组成序列Z={x_{i1},x_{i2},x_{i3},…x_{ik}},其中下标{i_1，i_2，i_3，…i_k}是一个严格递增的序列，Z中元素个数就是该子序列的长度。

那么爸爸X和孩子Y的公共子序列是指该序列既是爸爸X的子序列也是孩子Y的子序列。

孩子有多像爸爸？

即是找到爸爸X和孩子Y的最长公共子序列问题.

给定两个序列X={x0,x1,x2,…xm-1} 和Y={y0,y1,y2,…yn-1}找出X和Y的一个最长的公共子序列。

子序列：例如，X=（a，b，c，b，d，a，b），Y=（b，c，d，b）是X的一个子序列。

X=（a，b，c，b，d，a，b）

Y=（　b，c，　d，　b）

给定两个字符序列A和B，如果字符序列Z既是A的子序列，又是B的子序列，则称序列Z是A和B的公共子序列。该问题是求两序列A和B的**最长公共子序列**（LCS）。

动态规划求解最长公共子序列问题

【问题求解】若设$X=(x_0, x_1, \ldots, x_{m-1})$（含$m$个字符），$Y=(y_0, y_1, \ldots, y_{n-1})$（含$n$个字符），设$Z=(z_0, z_1, \ldots, z_{k-1})$（含$k$个字符）为它们的最长公共子序列。不难证明有以下性质：

如何把问题划分成子问题？

- 如果$x_{m-1}=y_{n-1}$，则$z_{k-1}=x_{m-1}=y_{n-1}$，且$(z_0, z_1, \ldots, z_{k-2})$是$(x_0, x_1, \ldots, x_{m-2})$和$(y_0, y_1, \ldots, y_{n-2})$的一个最长公共子序列。
- 如果$x_{m-1}\neq y_{n-1}$且$z_{k-1}\neq x_{m-1}$，则$(z_0, z_1, \ldots, z_{k-1})$是$(x_0, x_1, \ldots, x_{m-2})$和$(y_0, y_1, \ldots, y_{n-1})$的一个最长公共子序列。
- 如果$x_{m-1}\neq y_{n-1}$且$z_{k-1}\neq y_{n-1}$，则$(z_0, z_1, \ldots, z_{k-1})$是$(x_0, x_1, \ldots, x_{m-1})$和$(y_0, y_1, \ldots, y_{n-2})$的一个最长公共子序列。

动态规划求解最长公共子序列问题

定义二维动态规划数组dp，其中dp[i][j]为子序列（x_0，x_1，…，x_{i-1}）和（y_0，y_1，…，y_{j-1}）的最长公共子序列的长度。

每考虑字符x[i]或y[j]都为动态规划的一个阶段（共经历约m×n个阶段）。

情况1：x[i-1]=y[j-1]（当前两个字符相同）

（x_0，x_1，…，x_{i-1}）

‖

（y_0，y_1，…，y_{j-1}）

⇒ dp[i][j]=dp[i-1][j-1]+1

例如：

a b c x y

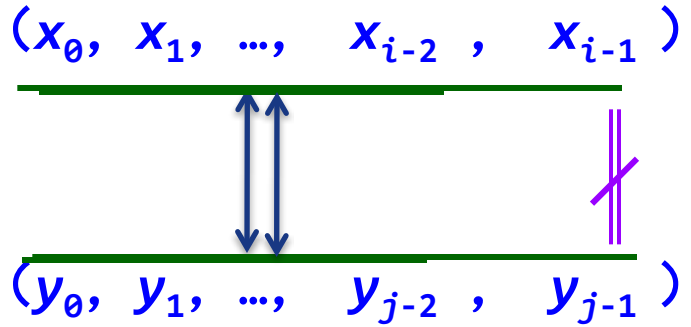
‖

b y

动态规划求解最长公共子序列问题

定义二维动态规划数组dp，其中dp[i][j]为子序列（x_0，x_1，…，x_{i-1}）和（y_0，y_1，…，y_{j-1}）的最长公共子序列的长度。

情况2：x[i-1]≠y[j-1]（当前两个字符不同）



dp[i-1][j]　　dp[i][j-1]

→ dp[i][j]=MAX(dp[i][j-1], dp[i-1][j])

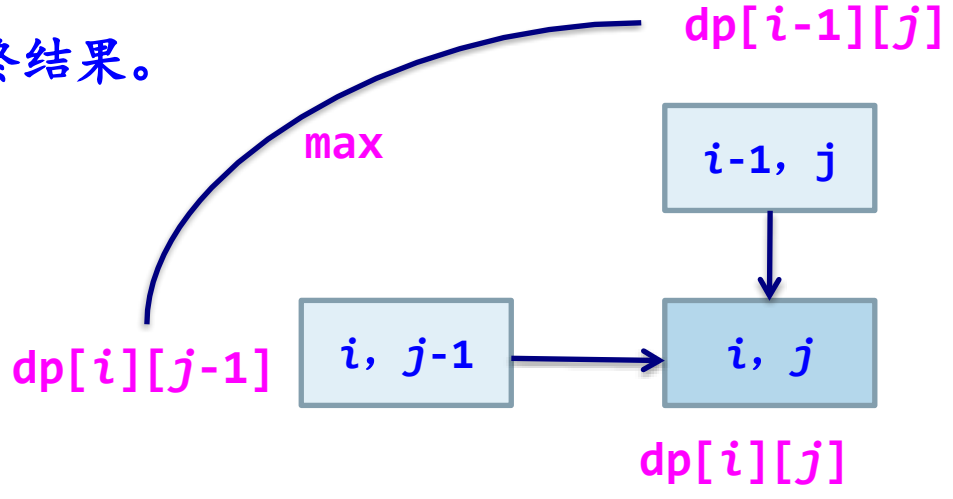
动态规划求解最长公共子序列问题

dp[i][j]为子序列（x_0，x_1，…，x_{i-1}）和（y_0，y_1，…，y_{j-1}）的最长公共子序列的长度。

对应的状态转移方程如下：

dp[i][j]=0	i=0或j=0—边界条件
dp[i][j]=dp[i-1][j-1]+1	x[i-1]=y[j-1]
dp[i][j]=MAX(dp[i][j-1], dp[i-1][j])	x[i-1]≠y[j-1]

显然，dp[m][n]为最终结果。



动态规划求解最长公共子序列问题

那么如何由dp求出LCS（最长公共子序列）呢？

dp ⇨ LCS

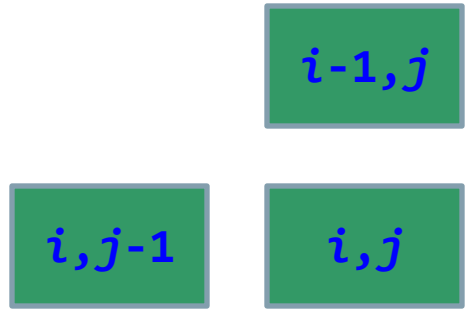
dp[i][j]=0	i=0或j=0—边界条件
dp[i][j]=dp[i-1][j-1]+1	a[i-1]=b[j-1]
dp[i][j]=MAX(dp[i][j-1], dp[i-1][j])	a[i-1]≠b[j-1]

dp[i][j] ≠ dp[i][j-1]（左边）并且

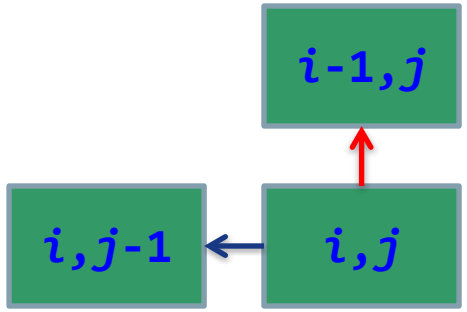
dp[i][j] ≠ dp[i-1][j]（上方）值时：

a[i-1]=b[j-1]

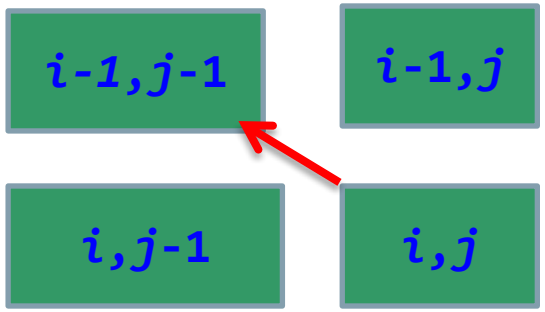
将a[i-1]添加到LCS中。



动态规划求解最长公共子序列问题

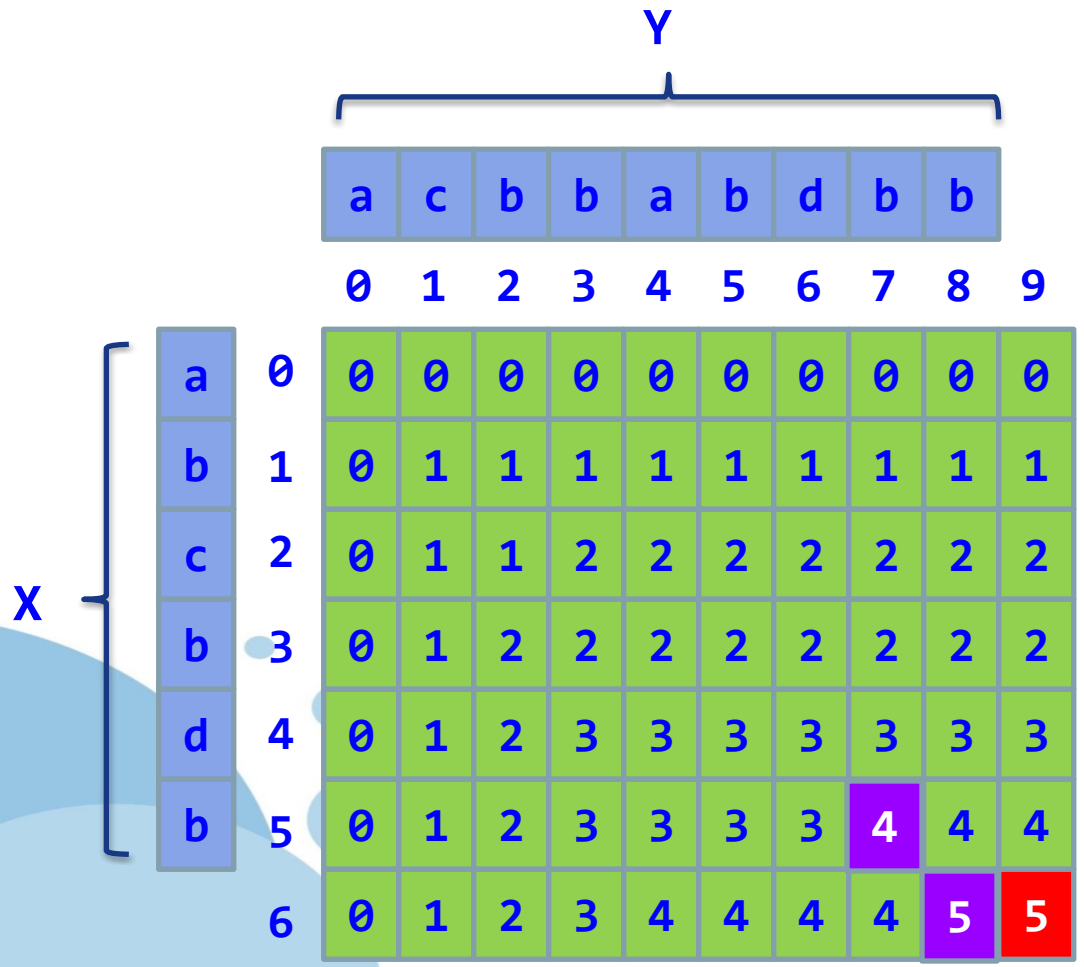


- dp[i][j]=dp[i][j-1]：与左边相等 ⇨ j--
- dp[i][j]=dp[i-1][j]：与上方相等 ⇨ i--
- 与左边、上方都不相等：a[i-1]或者b[j-1]属于LCS ⇨ i--，j--



例如，X=（a，b，c，b，d，b），m=6，Y=（a，c，b，b，a，b，d，b，b），n=9。

（1）求出dp



（2）dp[6][9]=5开始

LCS：d b

i=6，j=9

↓ 与左边相等，j--

i=6，j=8

↓ 与左、上方不等，i--，j--

i=5，j=7

↓ 与左、上方不等，i--，j--

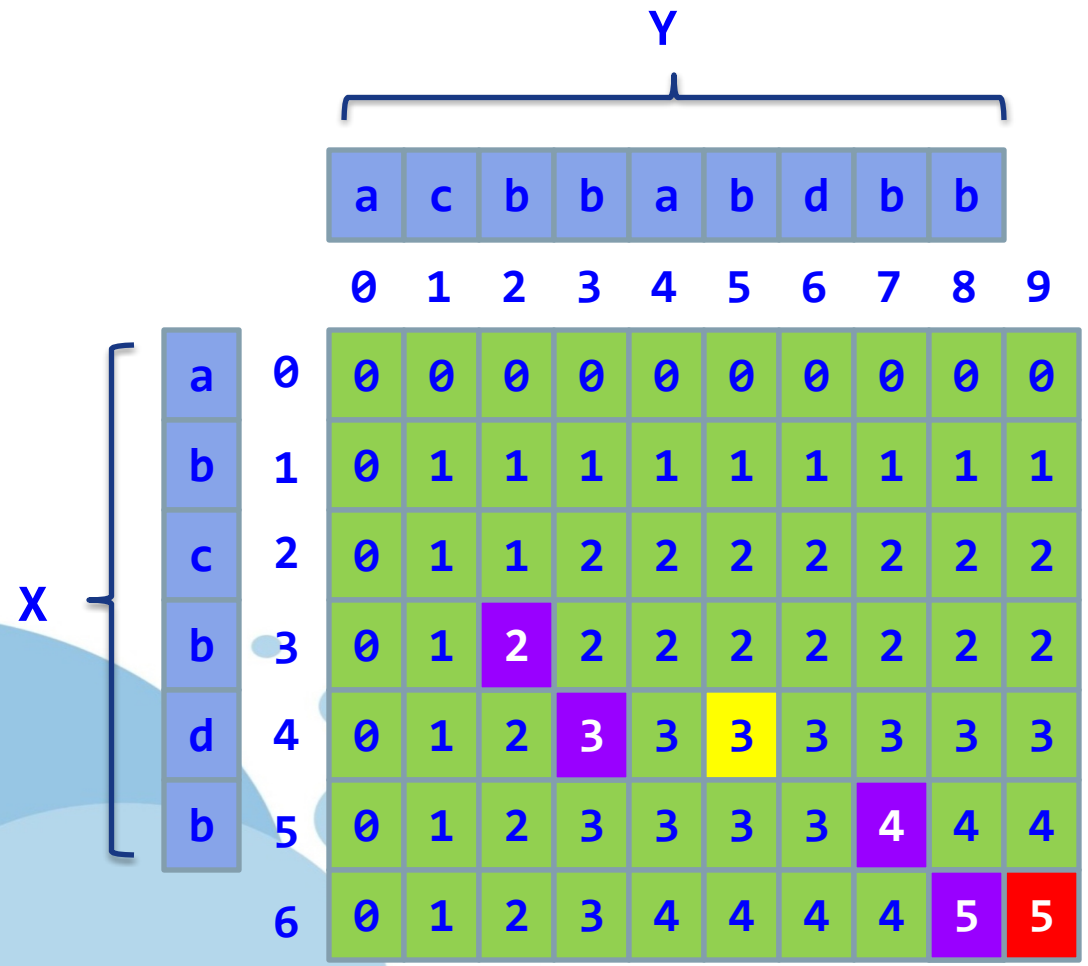
i=4，j=6

↓ 与左边相等，j--

i=4，j=5

那么如何由dp求出LCS呢？例如，X=（a，b，c，b，d，b），m=6，Y=（a，c，b，b，a，b，d，b，b），n=9。

（1）求出dp



（2）dp[6][9]=5开始

LCS：c b d b

i=4，j=5

↓ 与左边相等，j--

i=4，j=4

↓ 与左边相等，j--

i=4，j=3

↓ 与左、上方不等，i--，j--

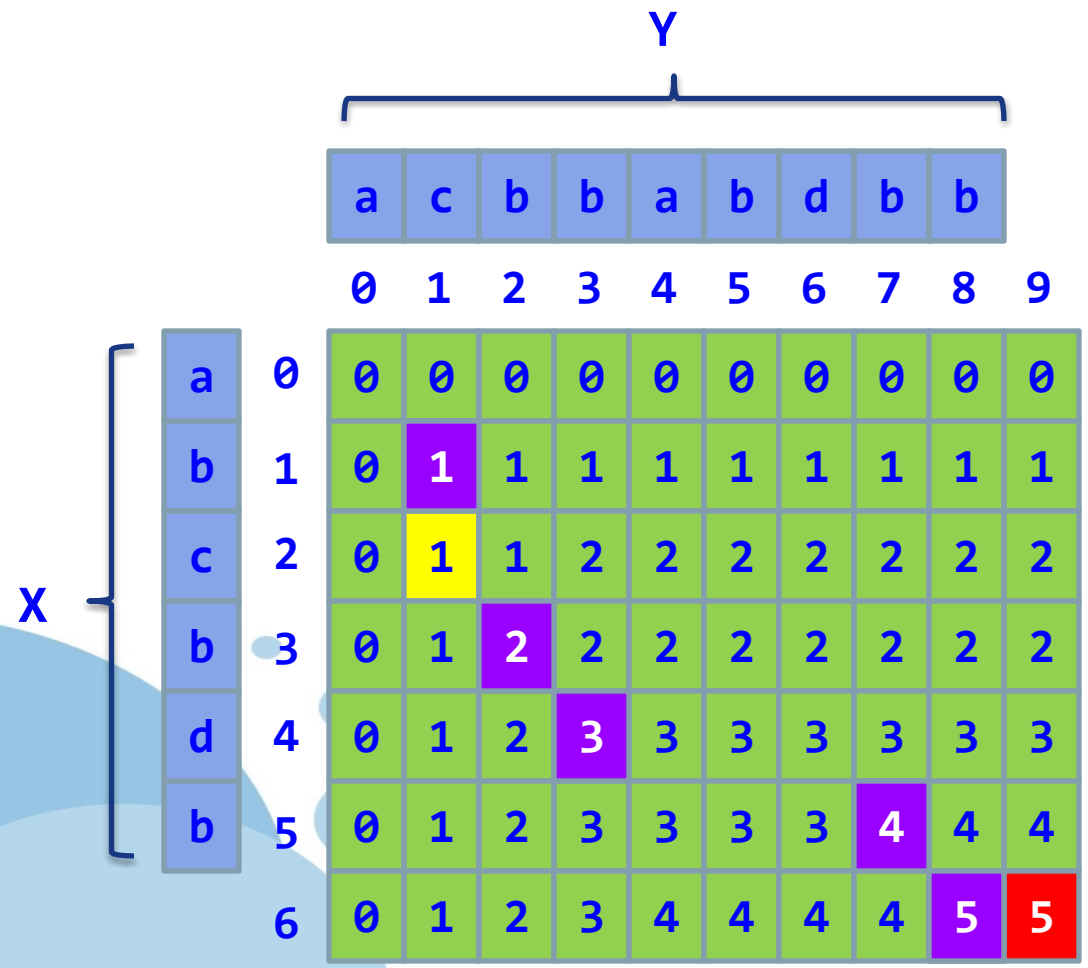
i=3，j=2

↓ 与左、上方不等，i--，j--

i=2，j=1

那么如何由dp求出LCS呢？例如，X=（a，b，c，b，d，b），m=6，Y=（a，c，b，b，a，b，d，b，b），n=9。

（1）求出dp



（2）dp[6][9]=5开始

LCS：a c b d b

i=2，j=1

↓ 与上方相等，i--

i=1，j=1

↓ 与左、上方不等，i--，j--

i=0，j=0

结果LCS="acbdb"

动态规划求解最长公共子序列问题

```
#define MAX 51                          //序列中最多的字符个数
//问题表示
int m, n;
string a, b;                            //求解结果表示
int dp[MAX][MAX];                       //动态规划数组
vector<char> subs;                      //存放LCS
```

动态规划求解最长公共子序列问题

```
void LCSlength()                          //求dp
{  int i, j;
   for (i=0;i<=m;i++)                     //将dp[i][0]置为0，边界条件
      dp[i][0]=0;
   for (j=0;j<=n;j++)                     //将dp[0][j]置为0，边界条件
      dp[0][j]=0;
   for (i=1;i<=m;i++)
      for (j=1;j<=n;j++)                  //两重for循环处理x、y的所有字符
      {  if (x[i-1]==y[j-1])              //情况(1)
            dp[i][j]=dp[i-1][j-1]+1;
         else                             //情况(2)
            dp[i][j]=max(dp[i][j-1], dp[i-1][j]);
      }
}
```

时间复杂度为O(*m*×*n*)

动态规划求解最长公共子序列问题

```
void Buildsubs()                          //由dp构造从subs
{  int k=dp[m][n];                        //k为a和b的最长公共子序列长度
   int i=m;
   int j=n;

   while (k>0)                            //在subs中放入最长公共子序列(反向)
     if (dp[i][j]==dp[i-1][j])
        i--;
     else if (dp[i][j]==dp[i][j-1])
        j--;
     else                                 //与上方、左边元素值均不相等
     {  subs.push_back(x[i-1]);           //subs中添加a[i-1]
        i--; j--; k--;
     }
}
```

动态规划求解最长公共子序列问题

【算法分析】 LCSlength算法中使用了两重循环，所以对于长度分别为m和n的序列，求其最长公共子序列的时间复杂度为$O(m \times n)$。空间复杂度为$O(m \times n)$。

DNA基因鉴定—编辑距离问题

人类的DNA有四个基本字母{A,C,G,T}构成，包含了多达30亿个字符。如果两个人的DNA序列相差0.1%，仍然意味着有300万个位置不同，我们通常看到的DNA亲子鉴定报告上结论有：相似度99.99%，不排除亲子关系。

如何判断两个基因的相似度呢？生物学给出了“编辑距离”的概念。

【编辑距离】是指将一个字符串变换为另一个字符串所需要的最小编辑操作序列。

如何找到两个字符串a[0..m-1]和b[0..n-1]的编辑距离呢？

DNA基因鉴定—编辑距离问题

例如：X=（A，B，C，D，A，B），Y=（B，D，C，A，B）如何找出编辑距离？

- 穷举法：会有很多种对齐方式，暴力穷举法是不可取的。
- 考虑能否把原问题转化为规模更小的子问题？
- 该问题具有最优子结构性质（证明略）可采用动态规划法进行求解。

DNA基因鉴定—编辑距离问题

【问题描述】设*A*和*B*是两个字符串。现在要用最少的字符操作次数（编辑距离最小），将字符串*A*转换为字符串*B*。这里所说的字符编辑操作共有3种：

（1）删除一个字符（delete）；

（2）插入一个字符（insert）；

（3）将一个字符替换另一个字符（replace）。

动态规划法求解编辑距离问题

【问题求解】设字符串A、B的长度分别为m、n，分别用字符串a、b存放。

设计一个动态规划二维数组dp，其中dp[i][j]表示：将a[0..i-1]（1≤i≤m）与b[0..j-1]（1≤j≤n）的最优编辑距离（即a[0..i-1] 转换为b[0..j-1]的最少操作次数）。

动态规划法求解编辑距离问题

两种特殊情况（边界条件）

- 当B串空时，要删除A中全部字符转换为B，即

  dp[i][0]=i（删除A中i个字符，共i次操作）；
- 当A串空时，要在A中插入B的全部字符转换为B，即

  dp[0][j]=j（向A中插入B的j个字符，共j次操作）。

动态规划法求解编辑距离问题

构造状态转移方程dp[i][j]

对于串A和串B均非空的情况，

Case1：当$a[i-1]=b[j-1]$时，这两个字符不需要任何操作，即dp[i][j]=dp[i-1][j-1]。

Case2：当$a[i-1]\neq b[j-1]$时，**以下3种操作都可以达到目的：**

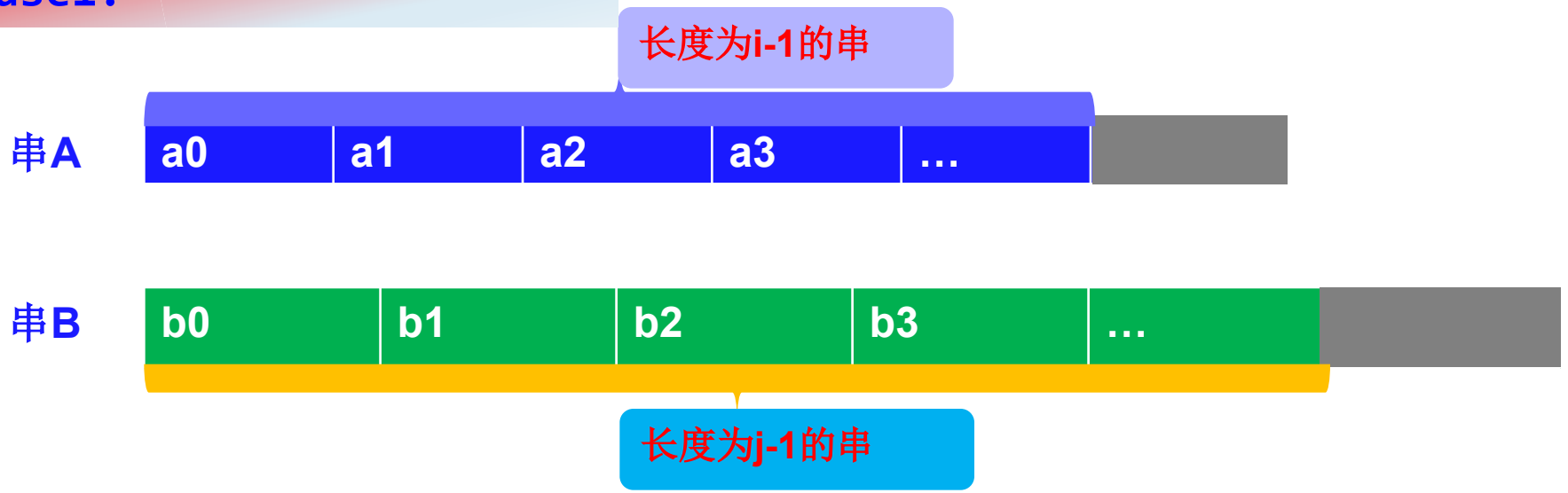
- 将$a[i-1]$替换为$b[j-1]$，有：dp[i][j]=dp[i-1][j-1]+1（一次替换操作的次数计为1）。
- 在$a[i-1]$字符后面插入$b[j-1]$字符，有：dp[i][j]=dp[i][j-1]+1（一次插入操作的次数计为1）。
- 删除$a[i-1]$字符，有：dp[i][j]=dp[i-1][j]+1（一次删除操作的次数计为1）。

此时dp[i][j]取3种操作的最小值。

动态规划法求解编辑距离问题

构造状态转移方程dp[i][j]:

Case1:



当$a[i-1]=b[j-1]$时，这两个字符不需要任何操作，即dp[i][j]=dp[i-1][j-1]。

长度为i的A串变换为长度为j的B串的最优编辑距离

长度为i-1的A串变换为长度为j-1的B串的最优编辑距离

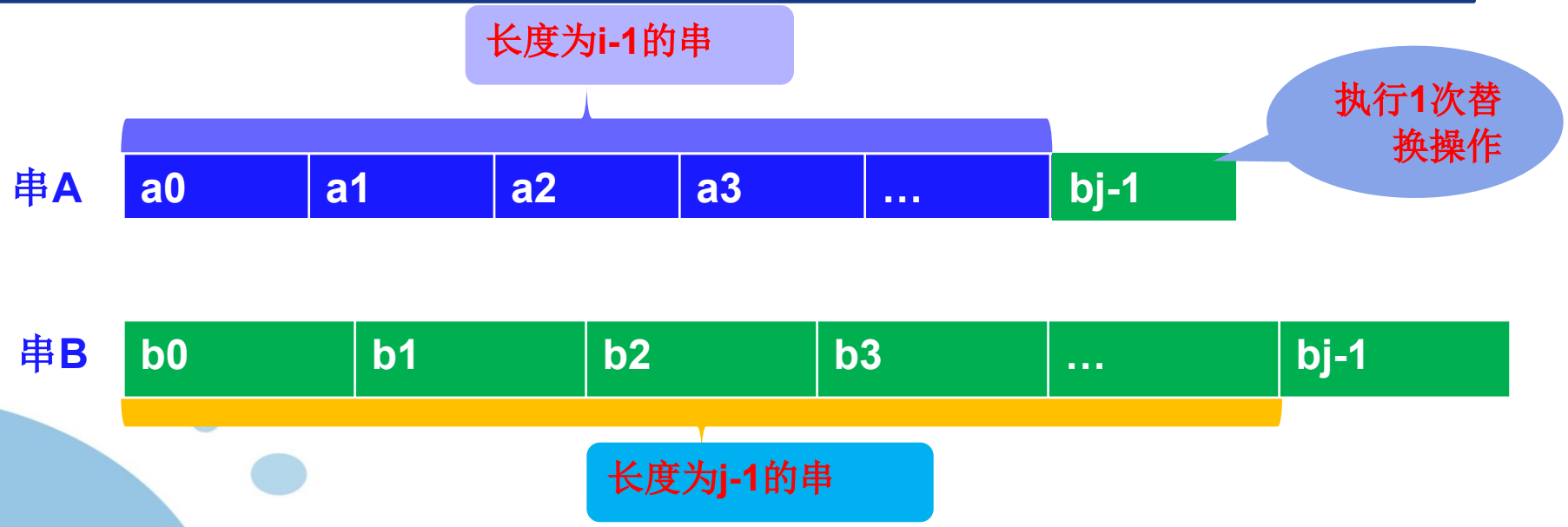
动态规划法求解编辑距离问题

构造状态转移方程dp[i][j]

case2:

当$a[i-1]\neq b[j-1]$时：可以通过如下三种操作之一进行转换：

操作1：



- 执行一次替换操作，
- 之后长度为i 的A串转换为长度为j的B串的编辑距离问题转化为：
  将长度为i-1 的A串转换为长度为j-1的B串的编辑距离问题。即：

$$dp[i][j]=dp[i-1][j-1]+1$$

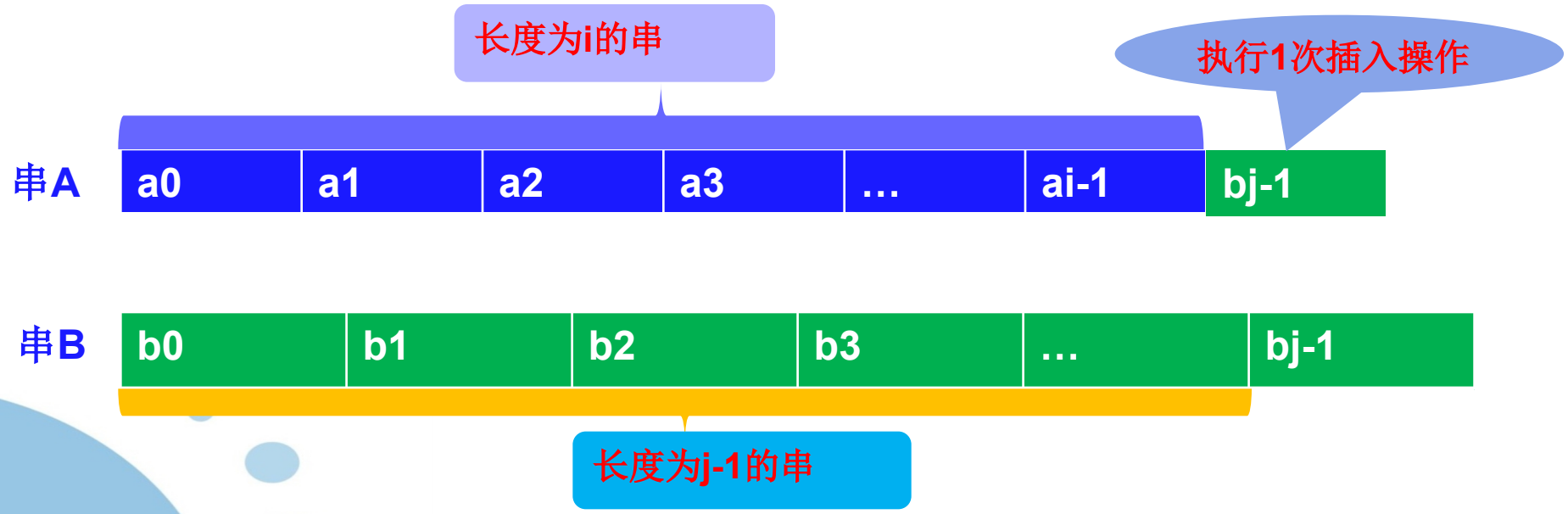
动态规划法求解编辑距离问题

构造状态转移方程dp[i][j]

case2:

当$a[i-1]\neq b[j-1]$时：可以通过如下三种操作之一进行转换：

操作2：



- 执行一次插入操作，在A串末尾插入b[j-1],
- 之后长度为i 的A串转换为长度为j的B串的编辑距离问题转化为：
  将长度为i 的A串转换为长度为j-1的B串的编辑距离问题。即：

$$dp[i][j]=dp[i][j-1]+1$$

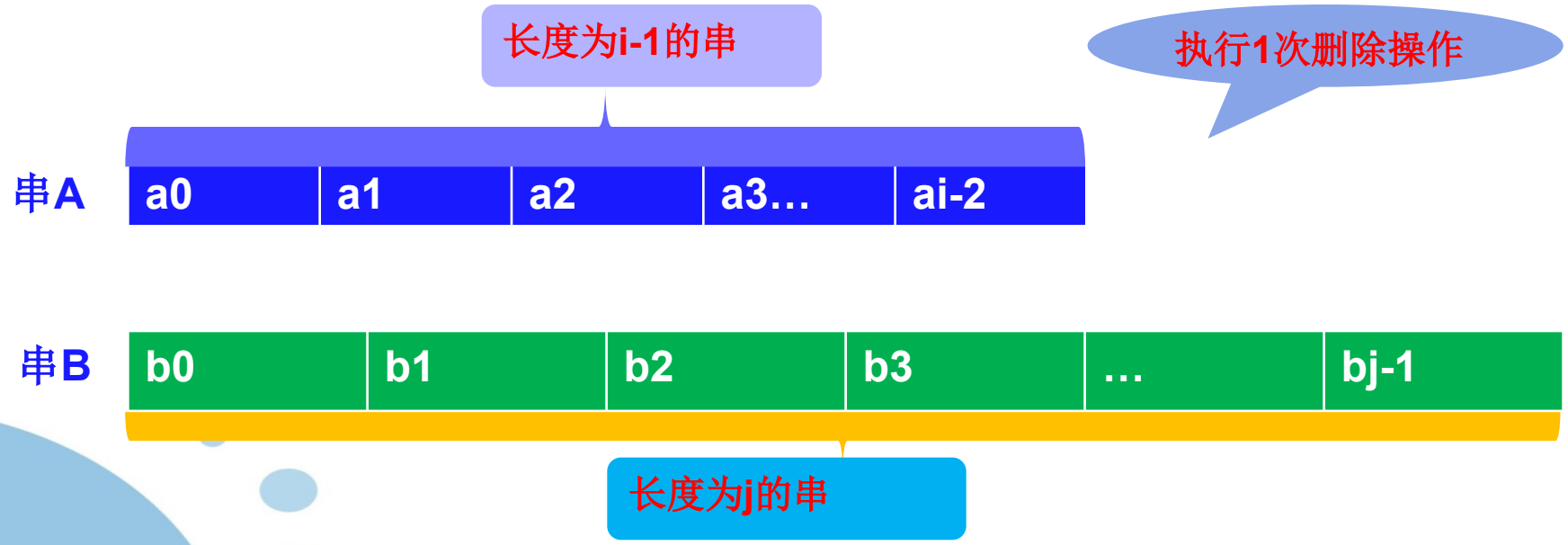
动态规划法求解编辑距离问题

构造状态转移方程dp[i][j]

case2:

当$a[i-1]\neq b[j-1]$时：可以通过如下三种操作之一进行转换：

操作3：



- 执行一次删除操作，将A串末尾的a[i-1]删除。
- 之后长度为i 的A串转换为长度为j的B串的编辑距离问题转化为：
  将长度为i-1 的A串转换为长度为j的B串的编辑距离问题。即：

$$dp[i][j]=dp[i-1][j]+1$$

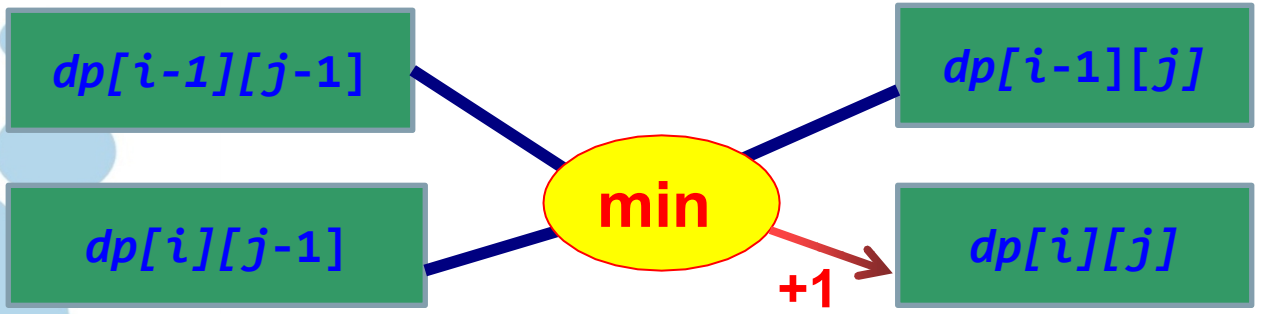
动态规划法求解编辑距离问题

构造状态转移方程dp[i][j]

case2:

当$a[i-1]\neq b[j-1]$时：可以通过如下三种操作之一进行转换：

dp[i][j]=min(dp[i][j]=dp[i-1][j-1]+1,
dp[i][j]=dp[i][j-1]+1,
dp[i][j]=dp[i-1][j]+1)



动态规划法求解编辑距离问题

状态转移方程dp[i][j]

- 当a[i-1]=b[j-1]时dp[i][j]=dp[i-1][j-1]
- 当a[i-1]≠b[j-1]时：

```
dp[i][j]=min(dp[i-1][j-1]+1,
             dp[i][j-1]+1,
             dp[i-1][j]+1)
```

最后得到的dp[m][n]即为所求。即将A串转换为B串的编辑距离最优值

动态规划法求解编辑距离问题——示例

例如A=“sfdqxbw”,B=“gfdgw”

	0	1	2	3	4	5	6
串A	s	f	d	q	x	b	w

	0	1	2	3	4
串B	g	f	d	g	w

动态规划法求解编辑距离问题——示例

构造dp[i][j]

串A	0	1	2	3	4	5	6
	s	f	d	q	x	b	w

串B	0	1	2	3	4
	g	f	d	g	w

i →

dp[i][j]	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	3	2	2	3
5	5	5	4	3	3	3
6	6	6	5	4	4	4
7	7	7	6	5	5	4

动态规划法求解编辑距离问题_算法描述

```
//问题表示
string a="sfdqxbw";
string b="gfdgw";
//求解结果表示
int dp[MAX][MAX];

void solve()                    //求dp
{  int i, j;
   for (i=1;i<=a.length();i++)
      dp[i][0]=i;               //把a的i个字符全部删除转换为b
   for (j=1; j<=b.length(); j++)
      dp[0][j]=j;               //在a中插入b的全部字符转换为b

   for (i=1; i<=a.length(); i++)
     for (j=1; j<=b.length(); j++)
     {  if (a[i-1]==b[j-1])
          dp[i][j]=dp[i-1][j-1];
        else
          dp[i][j]=min(min(dp[i-1][j], dp[i][j-1]), dp[i-1][j-1])+1;
     }
}
```

动态规划法求解编辑距离问题_算法分析

【算法分析】solve()算法中有两重循环，对应的时间复杂度为O(*mn*)。

购物街——0/1背包问题

娱乐节目——购物街，舞台模拟大型购物超市，有很多的商品，每个嘉宾都有一个购物车，嘉宾在规定的时间内尽情地将购物车装满，购物车中装的货物的价值（价格）最高的嘉宾获胜。

【问题描述】假设有n个商品和一个购物车，每个商品i对应的价值为vi,重量（体积）为wi,购物车的容量为W。且假设每个商品只有1件，要么装入，要么不装人，不可拆分。

【求解的问题】在购物车不超重的情况下，如何选取商品装入购物车，使所装入的物品总价值最大？最大价值是多少？装入了哪些物品？

动态规划法求解0/1背包问题

【问题描述】有n个重量分别为$\{w_1, w_2, \ldots, w_n\}$的物品，它们的价值分别为$\{v_1, v_2, \ldots, v_n\}$，给定一个容量为$W$的背包。

设计从这些物品中选取一部分物品放入该背包的方案，每个物品*要么选中要么不选中*，要求选中的物品不仅能够放到背包中，而且重量和为W具有最大的价值。

动态规划法求解0/1背包问题

【问题求解】对于可行的背包装载方案，背包中物品的总重量不能超过背包的容量。

最优方案是指所装入的物品价值最高，即$v_1*x_1+v_2*x_2+…+v_n*x_n$（其中$x_i$取0或1，取1表示选取物品$i$）取得最大值。

在该问题中需要确定x_1、x_2、…、x_n的值。假设按i=1，2，…，n的次序来确定x_i的值，对应n次决策即n个阶段。

动态规划法求解0/1背包问题

【STEP1】问题划分阶段：将整体问题划分成若干个阶段（阶段一定是有序的）

设置一个解向量X（x_1，x_2，…，x_n）解向量X（x_1，x_2，…，x_n）含义：n个物品，每一个对应一个x_i=0，

x_i=0代表对应的物品i不放入背包，

x_i=1代表对应的物品i放入背包

假设按i=1，2，…，n来确定x_i的值，对应n次决策即n个阶段。

的次序

动态规划法求解0/1背包问题

【STEP1】问题划分阶段：将整体问题划分成若干个阶段（阶段一定是有序的）

（假设按i=1，2，…，n的次序来确定x_i的值，对应n次决策即n个阶段。）

例如：若背包当前容量为r，（前提背包当前容量r>=wi）

若x_1=0 ⟶问题转化为其余物品（x_2，…，x_n）背包容量为r的问题，此时解向量为（0，x_2，…，x_n），当前背包容量为r

若x_1=1　问题转化为其余物品（x_2，…，x_n）

背包容量为r-w1的问题，此时解向量为（1，x_2，…，x_n），当前背包剩余容量为r-w1

动态规划法求解0/1背包问题

【STEP1】问题划分阶段：将整体问题划分成若干个阶段（阶段一定是有序的）

依次确定解向量中的每个分量，推广到一般。。。。

决策第i个物品的情况,当前背包剩余容量为r:

Case1:若（即第i个物品的重量）wi>r,则不装入第i个物品即： $X(x_1, x_2, x_{i-1}, 0, x_{i+1} \ldots, x_n)$

Case2：若（即第i个物品的重量）wi<=r,则

x_i

- 0：背包中不装入物品i，问题转换为：
  $X(x_1, x_2, x_{i-1}, 0, x_{i+1} \ldots, x_n)$ 当前背包容量为r
- 1：背包中不装入物品i，问题转换为：
  $X(x_1, x_2, x_{i-1}, 1, x_{i+1} \ldots, x_n)$ 当前背包容量为r-wi

动态规划法求解0/1背包问题

【STEP2】状态描述及状态变量：

设置二维动态规划数组dp，dp[i][r]表示当前背包（剩余）容量为r（1≤r≤W），

此时已装入了（1~i-1中的某些物品后），现在考虑第i个物品的决策后，背包装入物品的最优价值。

$X(x_1, x_2, \ldots x_{i-1}, \boxed{x_i}\ x_{i+1} \ldots, x_n)$

已决策

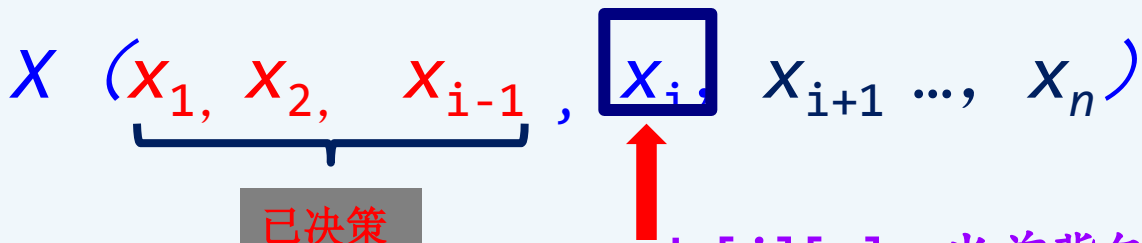
dp[i][r]：当前背包可用（剩余）容量为r，决定物品i的决策以使背包价值达到最大价值

动态规划法求解0/1背包问题

【STEP3】确定状态转移公式（方程）：

设置二维动态规划数组dp，即由第i-1个物品决策后形成的第i-1个状态（阶段）如何决策第i个状态（阶段）

dp[i][r]表示当前背包（剩余）容量为r（1≤r≤W），此时已装入了（1~i-1中的某些物品后），现在考虑第i个物品的决策后，背包装入物品的最优价值。



动态规划法求解0/1背包问题

设置二维动态规划数组dp，dp[i][r]表示背包剩余容量为r（1≤r≤W），已考虑物品1、2、…、i（1≤i≤n）时背包装入物品的最优价值。显然对应的状态转移方程如下：

【边界条件】

- dp[i][0]=0（背包不能装入任何物品，总价值为0）

  边界条件dp[i][0]=0（1≤i≤n）—边界条件

- dp[0][r]=0（没有任何物品可装入，总价值为0）

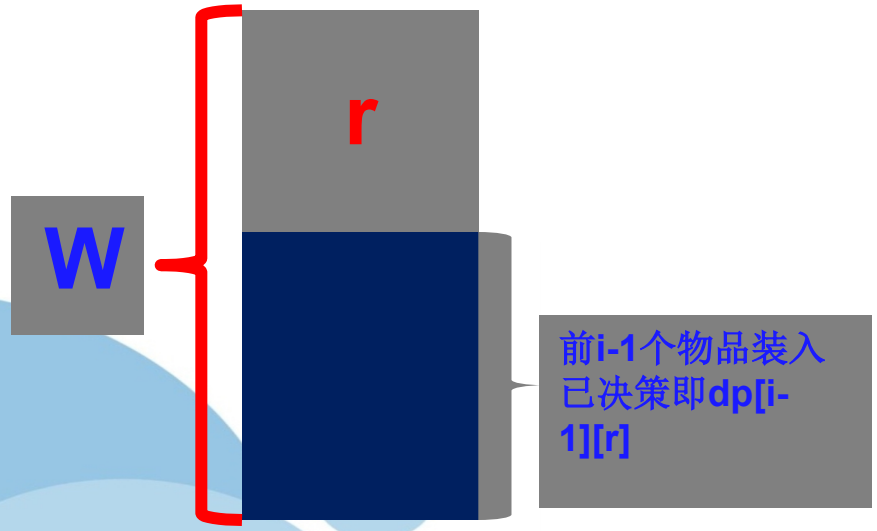
  边界条件dp[0][r]=0（1≤r≤W）—边界条件

这样， dp[n][W]便是0/1背包问题的最优解。

动态规划法求解0/1背包问题

【状态方程的递归公式】

Case1：若r<wi即当前背包剩余容量r<物品i的重量时：dp[i][r]=dp[i-1][r]（当r<w[i]时，物品i放不下对应的xi=0）



dp[i][r]= MAX{dp[i-1][r], dp[i-1][r-w[i]]+v[i]}

否则在不放入和放入物品i之间选最优解

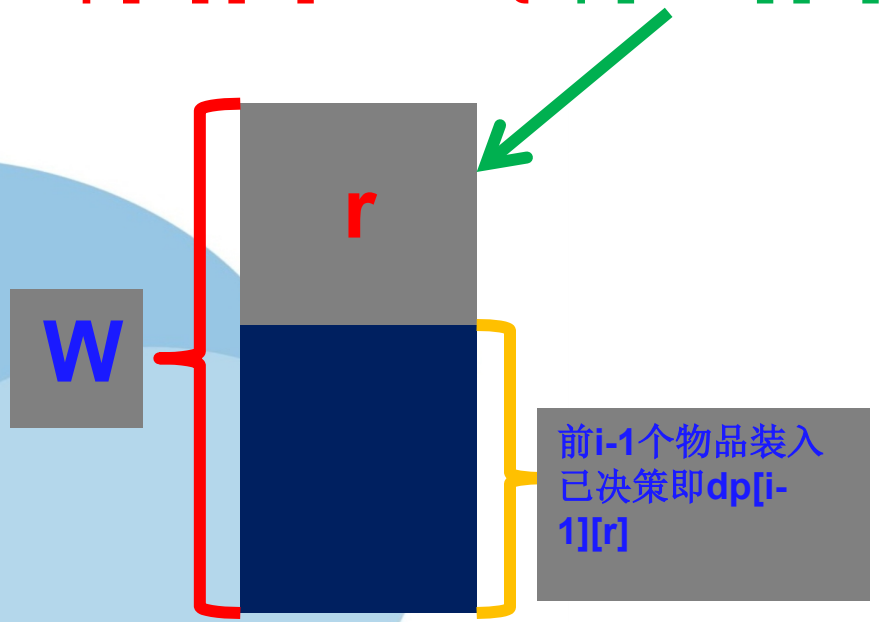
动态规划法求解0/1背包问题

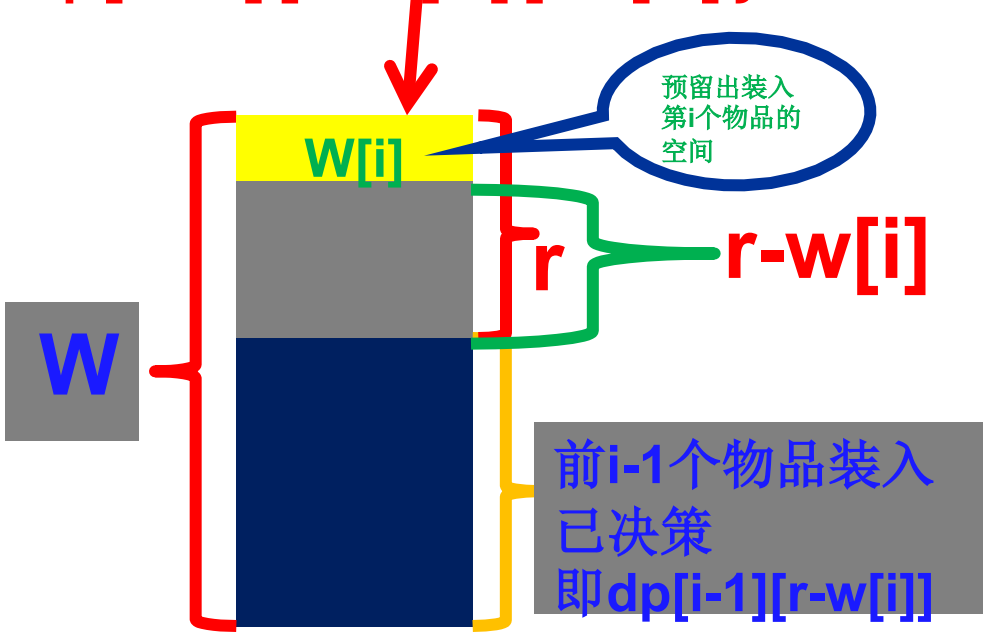
【状态方程的递归公式】

Case2：若r>=wi即当前背包剩余容量r>=物品i的重量时：对于物品i的决策有两种情况：

- 物品i不被装入则dp[i][r]=dp[i-1][r];
- 物品i被装入则dp[i][r]=dp[i-1][r-w[i]]+v[i]

dp[i][r]= MAX{dp[i-1][r], dp[i-1][r-w[i]]+v[i]}





动态规划法求解0/1背包问题

当dp数组计算出来后，推导出解向量x的过程十分简单，从dp[n][W]开始：

（1）若dp[i][r]≠dp[i-1][r]，若dp[i][r]=dp[i-1][r-w[i]]+v[i]，置x[i]=1，累计总价值maxv+=v[i]，递减剩余重量r=r-w[i]。

（2）若dp[i][r]=dp[i-1][r]，表示物品i放不下或者不放入物品i，置x[i]=0。

dp[i][r]=dp[i-1][r]　当r<w[i]时，物品i放不下

dp[i][r]= MAX{dp[i-1][r], dp[i-1][r-w[i]]+v[i]}

否则在不放入和放入物品i之间选最优解

动态规划法求解0/1背包问题示例

例如：n=5,W={2,2,6,5,4},V={6,3,5,4,6}(下标从1开始），背包初始容量为10。

W	2	2	6	5	4
V	6	3	5	4	6

r

Dp[i][r]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										
5	0										

i

边界条件

边界条件

求解dp过程：

i r

dp[1][1]=dp[0][1]=0;因为w[1]=2>r(1),所以物品1不能装入当前剩余容量为1的背包里。

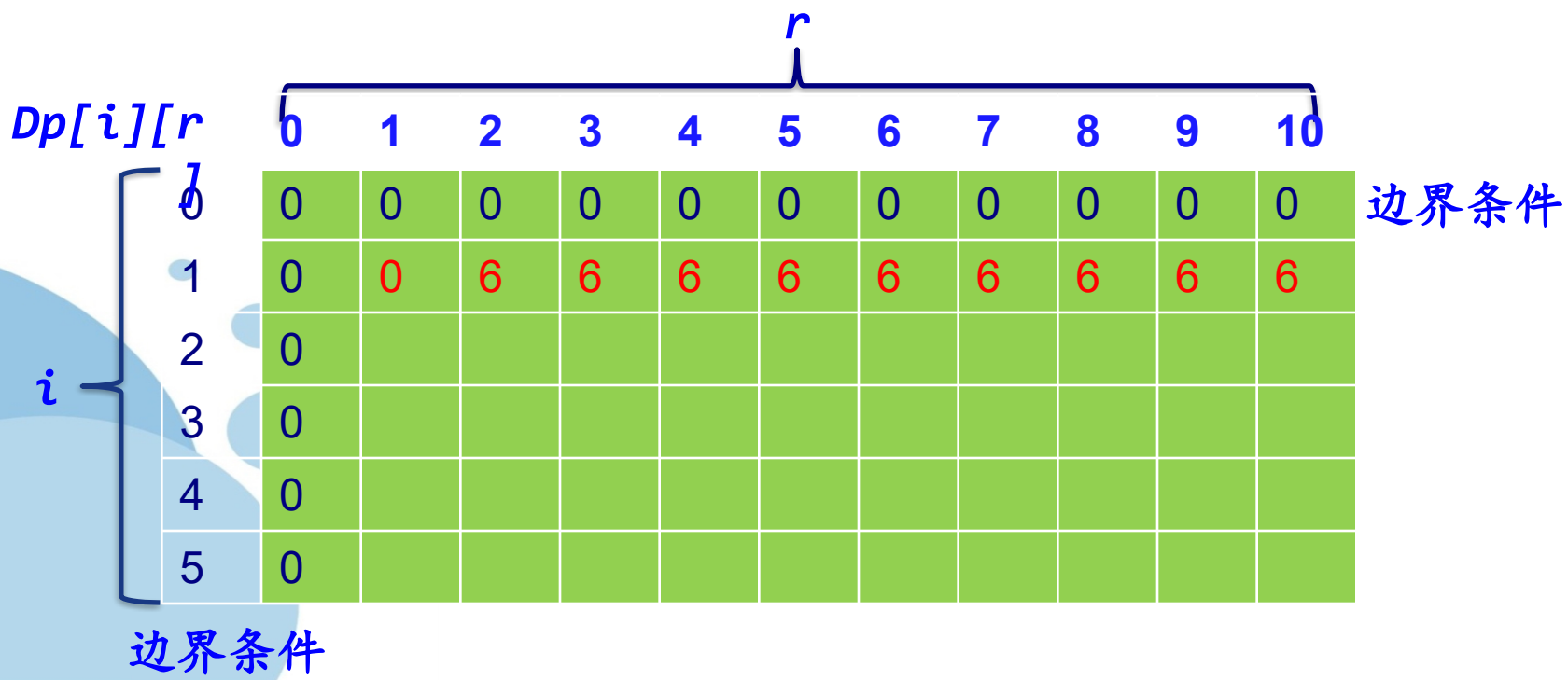
dp[1][2]=max(不装入物品1：dp[0][2],

装入物品1：dp[0][2-2]+v[1])=max（0,0+6）=6

dp[1][3]=max(不装入物品1：dp[0][3],

装入物品1：dp[0][3-2]+v[1])=max（0,0+6）=6

同理dp[1][4].....dp[1][10]=6



求解dp过程：　W[2]=2,v[2]=3

i　r

dp[2][1]=dp[1][1]=0;因为w[2]=2>r(1),所以物品2不能装入当前剩余容量为1的背包里。

dp[2][2]=max(不装入物品2：dp[1][2](6),

装入物品2：dp[1][2-2]+v[2])=max（0,0+3）=6

dp[2][3]=max(不装入物品2：dp[1][3],

装入物品2：dp[1][3-2]+v[2])=max（0,0+3）=6

dp[2][4] =max(不装入物品2：dp[1][4],

装入物品2：dp[1][4-2]+v[2])=max（6,6+3）=9

因为后面dp[1][3~10]均为6，所有dp[2][5~10]=9

r

Dp[i][r]	0	1	2	3	4	5	6	7	8	9	10	
0	0	0	0	0	0	0	0	0	0	0	0	边界条件
1	0	0	6	6	6	6	6	6	6	6	6	
2	0	0	6	6	9	9	9	9	9	9	9	
3	0											
4	0											
5	0											

i

求解dp过程：

i r

W[3]=6,v[3]=5

dp[3][1]=dp[2][1]=0;因为w[3]=6>r(1),所以物品3不能装入当前剩余容量为1的背包里。

dp[3][2]=dp[2][2]=6;因为w[3]=6>r(2)

dp[3][3]=dp[2][3]=6; dp[3][4]=dp[2][4]=9; dp[3][5]=dp[2][5]=9;

dp[3][6]=max(不装入物品3：dp[2][6](9),
装入物品3：dp[2][6-6]+v[3])=max（9,0+5）=9;

dp[3][7]=max(不装入物品3：dp[2][7](9),
装入物品3：dp[2][7-6]+v[3])=max（9,0+5）=9;

dp[3][8]=max(不装入物品3：dp[2][8](9),
装入物品3：dp[2][8-6]+v[3])=max（9,6+5）=11;

dp[3][9]=max(不装入物品3：dp[2][9](9),
装入物品3：dp[2][9-6]+v[3])=max（9,6+5）=11;

dp[3][10]=max(不装入物品3：dp[2][10](9),
装入物品3：dp[2][10-6]+v[3])=max（9,9+5）=14;

r

Dp[i][r]	0	1	2	3	4	5	6	7	8	9	10	
0	0	0	0	0	0	0	0	0	0	0	0	边界条件
1	0	0	6	6	6	6	6	6	6	6	6	
2	0	0	6	6	9	9	9	9	9	9	9	
3	0	0	6	6	9	9	9	9	11	11	14	
4	0											
5	0											

i

求解dp过程：

W[4]=5,v[4]=4

i r

dp[4][1]=dp[3][1]=0;因为w[4]=5>r(1),所以物品4不能装入当前剩余容量为1的背包里。

dp[4][2]=dp[3][2]=6;因为w[4]=6>r(2)

dp[4][3]=dp[3][3]=6; dp[4][4]=dp[3][4]=9;

dp[4][5]=max(不装入物品4：dp[3][5](9),
装入物品4：dp[3][5-5]+v[4])=max（9,0+4）=9;

dp[4][6]=max(不装入物品4：dp[3][6](9),
装入物品4：dp[3][6-5]+v[4])=max（9,0+4）=9;

dp[4][7]=max(不装入物品4：dp[3][7](9),
装入物品4：dp[3][7-5]+v[4])=max（9,6+4）=10;

dp[4][8]=max(不装入物品4：dp[3][8](11),
装入物品4：dp[3][8-5]+v[3])=max（11,6+4）=11;

dp[4][9]=max(不装入物品4：dp[3][9](11),
装入物品4：dp[3][9-5]+v[4])=max（11,9+4）=13;

dp[4][10]=max(不装入物品4：dp[3][10](14),
装入物品4：dp[3][10-5]+v[4])=max（14,9+4）=14;

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0	0	6	6	9	9	9	10	11	13	14

求解dp过程：

i r

W[5]=4,v[5]=6

dp[5][1]=dp[4][1]=0;因为w[5]=4>r(1),所以物品5不能装入当前剩余容量为1的背包里。

dp[5][2]=dp[4][2]=6;因为w[5]=4>r(2)

dp[5][3]=dp[4][3]=6;

dp[5][4]=max(不装入物品5：dp[4][4]=9,装物品5：dp[4][4-4]+6)=9;

dp[5][5]=max(不装入物品5：dp[4][5](9),
装入物品5：dp[4][5-4]+v[5])=max（9,0+6）=9;

dp[5][6]=max(不装入物品5：dp[4][6](9),
装入物品5：dp[4][6-4]+v[5])=max（9,6+6）=12;

dp[5][7]=max(不装入物品5：dp[4][7](10),
装入物品5：dp[4][7-4]+v[5])=max（10,6+6）=12;

dp[5][8]=max(不装入物品5：dp[4][8](11),
装入物品5：dp[4][8-4]+v[5])=max（11,9+6）=15;

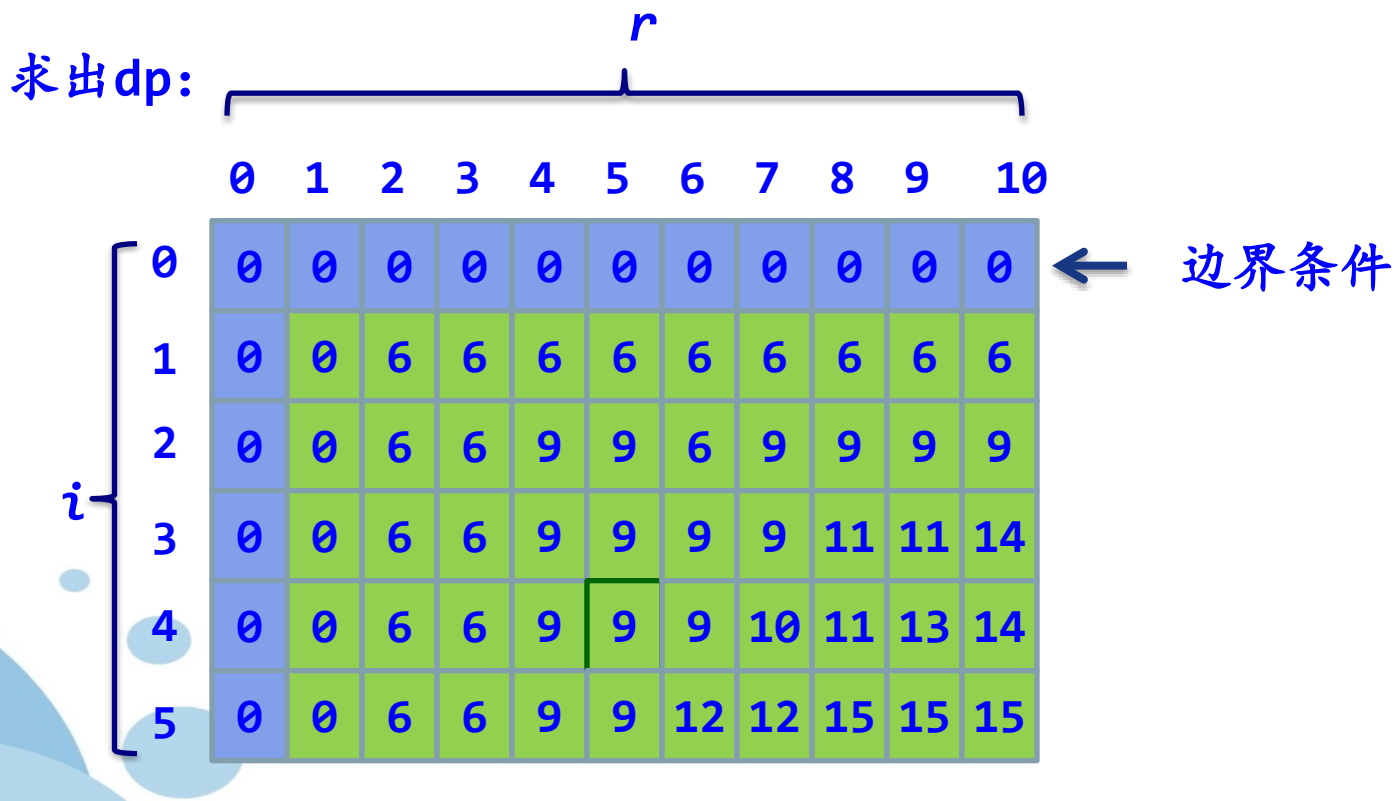
dp[5][9]=max(不装入物品5：dp[4][9](13),
装入物品5：dp[4][9-4]+v[5])=max（13,9+6）=15;

dp[5][10]=max(不装入物品5：dp[4][10](14),

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0	0	6	6	9	9	9	10	11	13	14
5	0	0	6	6	9	9	12	12	15	15	15

动态规划法求解0/1背包问题示例

例如，某0/1背包问题为，$n=5$，$w=\{2, 2, 6, 5, 4\}$，$v=\{6, 3, 5, 4, 6\}$（下标从1开始），$W=10$。

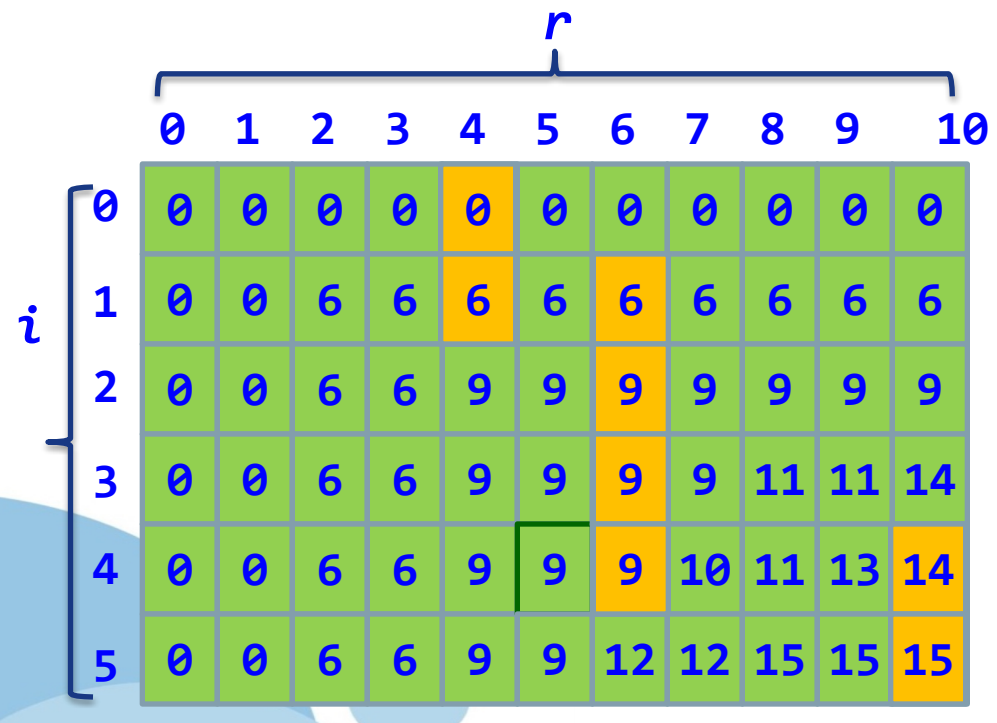


动态规划法求解0/1背包问题示例

回推求最优解的过程：

$w=\{2, 2, 6, 5, 4\}$，$v=\{6, 3, 5, 4, 6\}$

$i=5$，$r=W=10$，从dp[5][10]开始



$x=(1, 1, 0, 0, 1)$，
装入物品总重量为8，
总价值为15

dp[5][10] ≠dp[4][10]
⇨ x[5]=1，r=r-w[5]=6

i=i-1=4，dp[4][6]=dp[3][6]
⇨ x[4]=0

i=i-1=3，dp[3][6]=dp[2][6]
⇨ x[3]=0

i=i-1=2，dp[2][6]≠dp[1][6]
⇨ x[2]=1，r=r-w[2]=4

i=i-1=1，dp[1][4]≠dp[0][4]
⇨ x[1]=1，r=r-w[1]=2

动态规划法求解0/1背包问题算法描述

```
//问题表示
int n=5, W=10;                          //5种物品，限制重量不超过10
int w[MAXN]={0, 2, 2, 6, 5, 4};         //下标0不用
int v[MAXN]={0, 6, 3, 5, 4, 6};         //下标0不用

//求解结果表示
int dp[MAXN][MAXW];
int x[MAXN];
int maxv;                               //存放最优解的总价值
```

动态规划法求解0/1背包问题算法描述

```
void Knap()                          //动态规划法求0/1背包问题
{  int i, r;
   for (i=0;i<=n;i++)                //置边界条件dp[i][0]=0
      dp[i][0]=0;
   for (r=0;r<=W;r++)                //置边界条件dp[0][r]=0
      dp[0][r]=0;

   for (i=1;i<=n;i++)
   {  for (r=1;r<=W;r++)
        if (r<w[i])
           dp[i][r]=dp[i-1][r];
        else
           dp[i][r]=max(dp[i-1][r], dp[i-1][r-w[i]]+v[i]);
   }
}
```

动态规划法求解0/1背包问题算法描述

```
void Buildx()                          //回推求最优解
{  int i=n, r=W;
   maxv=0;
   while (i>=0)                        //判断每个物品
   {
      if (dp[i][r]!=dp[i-1][r])
      {  x[i]=1;                       //选取物品i
         maxv+=v[i];                   //累计总价值
         r=r-w[i];
      }
      else
        x[i]=0;                        //不选取物品i
      i--;
   }
}
```

动态规划法求解0/1背包问题算法分析

【算法分析】Knap()算法中含有两重for循环，所以时间复杂度为$O(n\times W)$，空间复杂度为$O(n\times W)$。