算法 设计与分析

第三章　动态规划

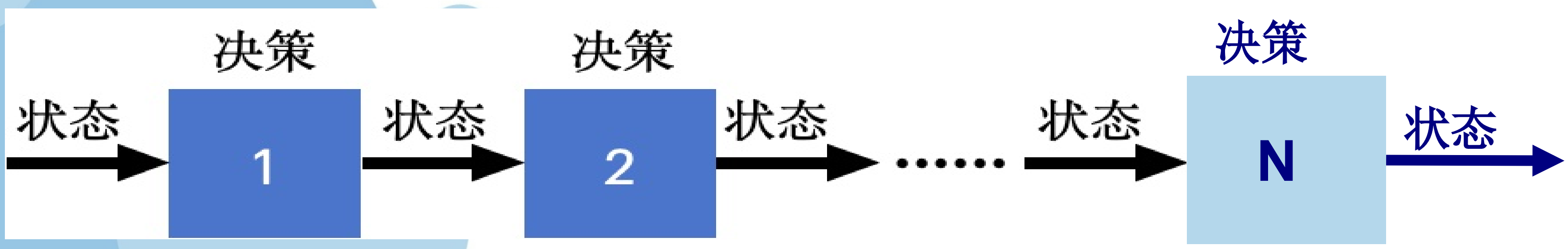
Dynamic Programming(DP)

算法 设计与分析

动态规划

动态规划是运筹学的一个分支，由美国数学家贝尔曼（R. Bellman）等人在1957年提出，是解决多阶段决策过程最优化的一种数学方法，在工程技术、经济、工业生产、军事以及自动化控制等领域用途广泛。

把多阶段问题转换为一系列的相互联系的单阶段问题，逐个加以解决。所以，DP实际上是一种数学方法，是求解某类问题的方法，严格意义上不是一种算法。



Chapter 3

本章内容

引言：理解动态规划

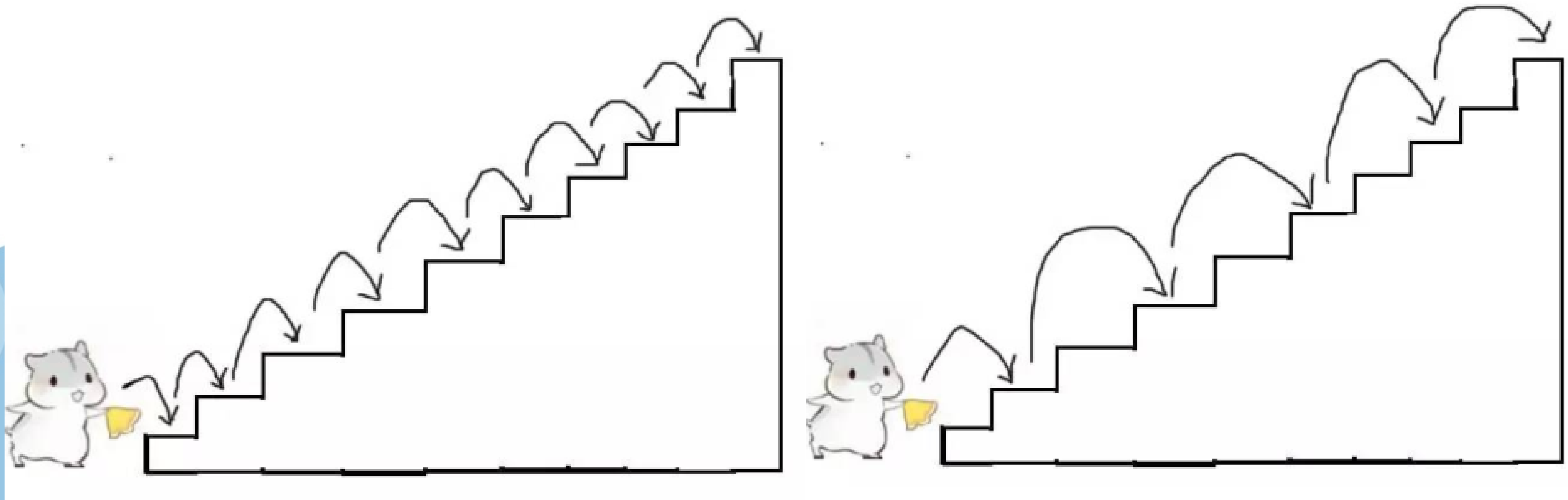
3.1 动态规划概述

3.2 动态规划经典算法

- 整数分解
- 最大连续子序列和
- 最长递增子序列(合唱队形问题）
- 最长公共子序列
- 编辑距离
- 矩阵连乘问题（最佳次序）
- 0-1背包问题

引 言

【问题描述】有一座高度是10级台阶的楼梯，从下往上走，每跨一步只能走1级或者2级台阶（可交叉进行，如1211212）。编写程序求出一共有多少种走法。



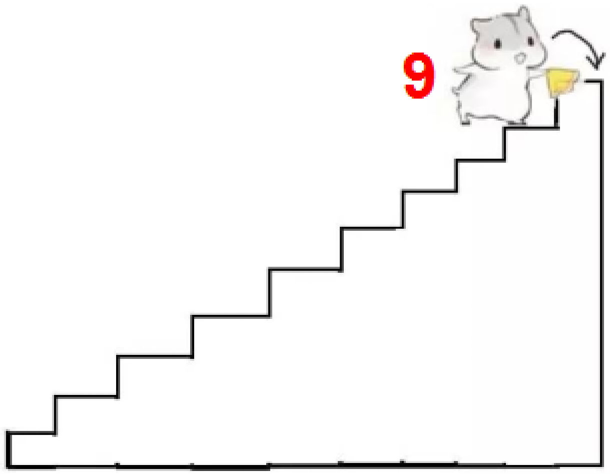
引 言

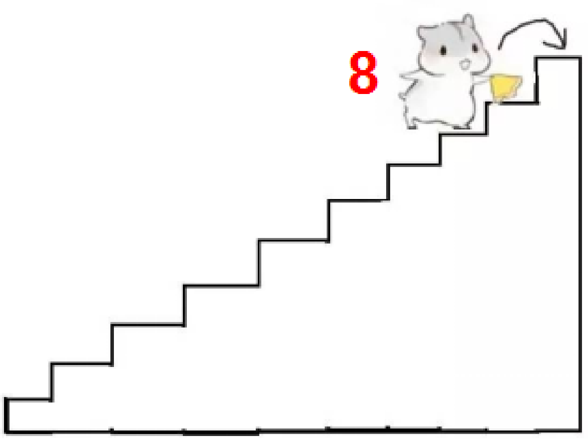
【问题分析】

什么情况下，能一下就到第10级台阶?

✓ 在第9级台阶处，走一步结束。

✓ 在第8级台阶处，走两步结束。





3 引 言——理解动态规划

用F(i)表示从第0级台阶走到第i级台阶的方案数，

则$F(10)=F(9)+F(8)$

$$\begin{cases} F(1)=1 \qquad F(2)=2 & \text{边界条件} \\ F(n)=F(n-1)+F(n-2) \end{cases}$$

斐波那契数列

从斐波那契数列理解动态规划

$$\mathrm{Fib}(n)=\begin{cases} n & \text{若}n=1\text{或}n=2 \\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2) & \text{其它情况} \end{cases}$$

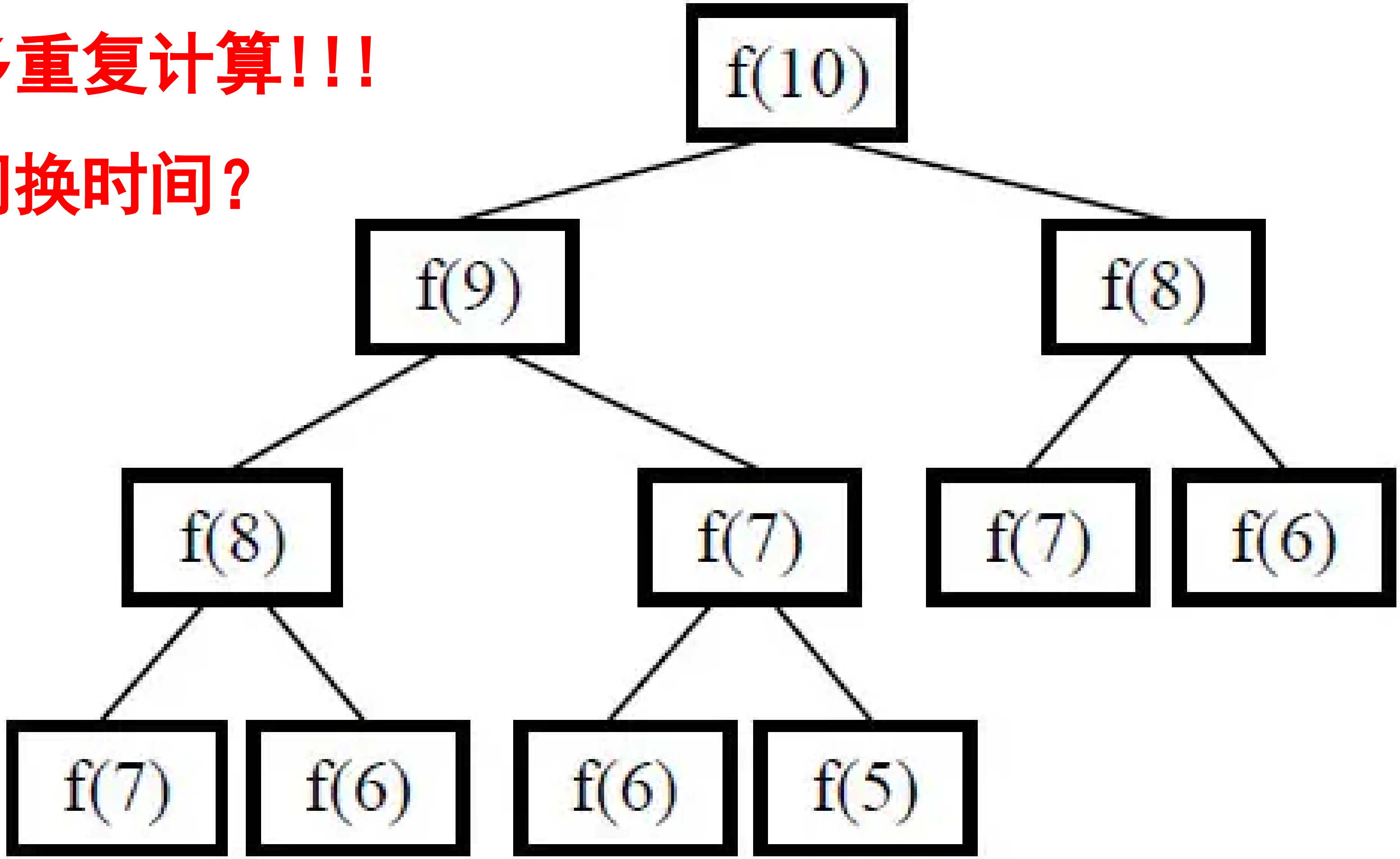
```
long long Fib1(int n)
{  if (n==1 || n==2)
       return n;
   else
       return  Fib(n-1) + Fib(n-2);
 }
```

求解斐波那契数列——递归算法

3 从斐波那契数列理解动态规划

太多重复计算!!!

空间换时间?



为避免重复计算，设计一个dp数组。dp[i]存放Fib(i)的值，首先设置dp[1]=1和dp[2]=2，再让i从3到n循环以计算dp[3]到dp[n]的值，最后返回dp[n]即结果。对应算法如下：

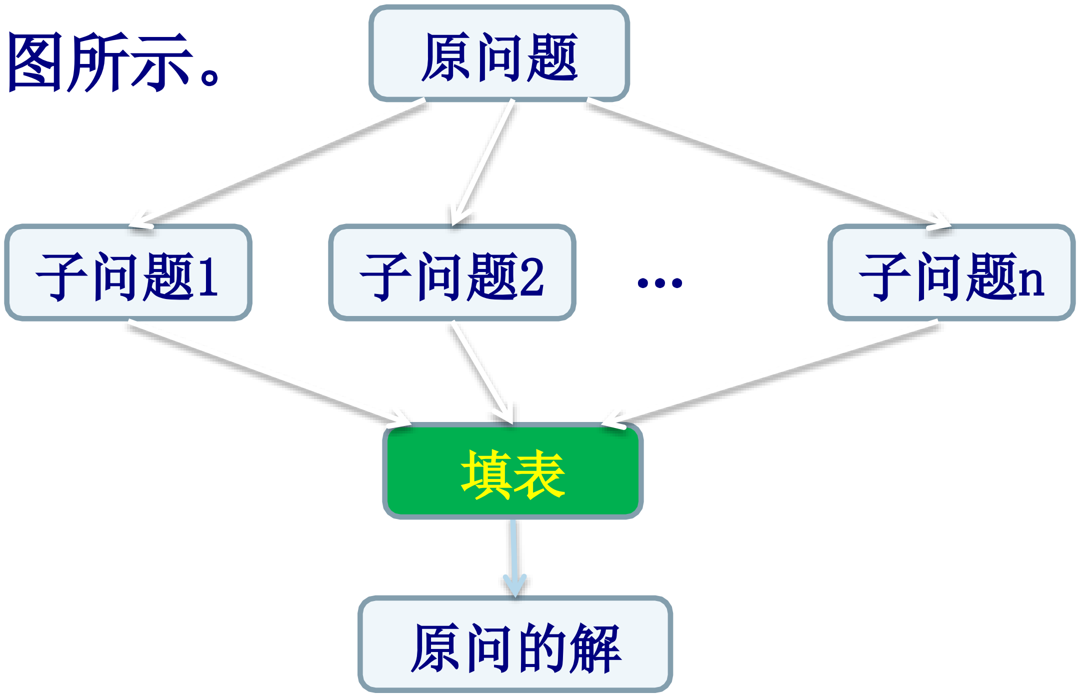
求解斐波那契数列——动态规划算法

```
int dp[MAX+1]={0};            //所有元素初始化为0
long long Fib2(int n)
{
   dp[1]=1; dp[2]=2;
   for (int i=3;i<=n;i++)
        dp[i]=dp[i-1]+dp[i-2];

   return dp[n];
}
```

3 从斐波那契数列理解动态规划

动态规划法也称填表法，数组dp（表）称为动态规划数组，其基本求解过程如下图所示。



台阶数	1	2	3	4	5	6	7	8	9
走法数	1	2	3	5	8				

3 动态规划与备忘录法的区别

备忘录方法是动态规划算法的变形，也用一个表格来保存已解决的子问题的答案，在下次需要解决此问题时，只需查看该子问题的解答，而不必重新计算。

与动态规划算法不同的是，备忘录方法的递归方式是自顶向下的，而动态规划算法则是自底向上递推的。因此，备忘录方法的控制结构与直接递归方法的控制结构相同，但为每个解过的子问题建立了备忘录以备需要时查看，避免了相同子问题的重复求解。

3 从钞票问题理解动态规划

【问题描述】假设你身上带了足够的1、5、10、20、50、100元面值的钞票。现在您的目标是凑出某个金额 M，用到尽量少的钞票张数。比如凑666元，你的方案是什么？

如果某国家的钞票面额分别是1、5、11，如何凑15元的面额，用张数尽量少的钞票？

3 从钞票问题理解动态规划

【问题分析】： 钞票面值1，5，11

M=15时，

✓如果取11，接下来需要面对 M=4 的情况；

✓如果取5，接下来需要面对 M=10 的情况；

✓如果取1，接下来面对 M=14 的情况；

重大发现，有相同的问题形式：给定 M，凑出 M 所用的最少钞票张数是多少张？

因此，用f(n)来表示“凑出n所需的最少钞票数量”。

3 从钞票问题理解动态规划

取11的代价（钞票总数）：$f(15) = f(4)+1 = 4+1 = 5$

取5的代价（钞票总数）：$f(15) = f(10)+1 = 2+1 = 3$

取1的代价（钞票总数）：$f(15) = f(11)+1 = 4+1 = 5$

重要启示：$f(n)$只和$f(n-1), f(n-5), f(n-11)$有关，即

$$f(n)=\min\{ f(n-1), f(n-5), f(n-11)\} + 1$$

3 从钞票问题理解动态规划

```c
scanf("%d",&n);
f[0]=0;

for(i=1;i<=n;i++)
{
    c1=c2=c3=INF;
    if(i-1>=0) c1=f[i-1];
    if(i-5>=0) c2=f[i-5];
    if(i-11>=0) c3=f[i-11];
    cost=min(c1,c2,c3);
    f[i]=cost+1;
    printf("i=%d, cost=%d\n",i,f[i]);
}
```

```
15
i=1, cost=1
i=2, cost=2
i=3, cost=3
i=4, cost=4
i=5, cost=1
i=6, cost=2
i=7, cost=3
i=8, cost=4
i=9, cost=5
i=10, cost=2
i=11, cost=1
i=12, cost=2
i=13, cost=3
i=14, cost=4
i=15, cost=3

----------------
Process exited
```

3.1 动态规划概述

- 递归：程序自己调用自己的技术。
- 分治：分而治之，把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

  大问题可分解为若干规模较小的相同子问题；
  子问题可解或容易解；
  子问题的解容易合并为原问题的解。
  子问题相互独立，不互相包含；

3 3.1 动态规划概述

- 递归：程序自己调用自己的技术。
- 分治：分而治之，把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

  大问题可分解为若干规模较小的相同子问题；
  子问题可解或容易解；
  子问题的解容易合并为原问题的解。
  子问题相互独立，不互相包含；

3 3.1 动态规划概述

- 动态规划算法与分治法类似，也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。
- 但是，适合于用动态规划求解的问题，经分解得到子问题往往不互相独立。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。

3 3.1 动态规划概述

适用于动态规划求解问题的3个性质：

（1）最优子结构：问题的最优解，包含子问题的最优解。或者说子问题的最优解构成原问题的最优解。基本条件。

（2）无后效性（马尔科夫性）：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

（3）有重叠子问题（重叠子结构）：子问题之间不独立，一个子问题在下一阶段决策中可能被多次使用到。提体现DP优势所在。

3.1 动态规划概述

动态规划求解问题基本步骤：

① 分析最优解的性质（最优子结构）。

② 递归定义最优解（状态转移方程或递归方程）。

③ 以自底向上或自顶向下（备忘录法）的记忆化方式计算出最优值。

④ 根据计算最优值时得到的信息，构造问题最优解。

3.1 动态规划概述

【例】求解组合数 C_n^m $\qquad \dfrac{n!}{(n-m)!m!}$

组合数-递归算法 $\quad C_n^m = C_{n-1}^{m-1} + C_{n-1}^m$

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

3.1 动态规划概述

组合数-动态规划算法

- 步骤1：分析最优子结构。子问题最优解包含原问题最优解，且子问题具有重叠性。

- 步骤2：建立递归关系

$$\begin{cases} C_n^m = C_{n-1}^m + C_{n-1}^{m-1}, & n > m > 0; \\ C_n^m = 1, & m = 0 \text{ 或 } m = n \end{cases}$$

可以用C[i][j](i=1~n, j=1~m)来记录C_i^j，即用一张表来记录重复子问题的结果。

3.1 动态规划概述

- **步骤3：计算最优值（填表）**

$$\begin{cases} C_n^m = C_{n-1}^m + C_{n-1}^{m-1}, & n > m > 0; \\ C_n^m = 1, & m = 0\text{或}m = n \end{cases}$$

如求解 C_5^3

。

	j=0	j=1	j=2	j=3
i=0				
i=1	1	0	0	0
i=2	1	0	0	0
i=3	1	0	0	0
i=4	1	0	0	0
i=5	1	0	0	0

？ ？

3 3.1 动态规划概述

步骤4：算法描述及分析

```
int ComB(int n, int m )
{  int C[n+1][m+1]={0},i,j;

   for (i=1;i<=n; i++)  C[i][0]=1;

   for (i=1;i<=n;i++)
       for (j=1 ;j<=m; j++)
           if(i==j)    C[i][j]=1;
           else if(i>j)   C[i][j]=C[i-1][j-1]+C[i-1][j];

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

3 3.1 动态规划概述

算法分析：

相对于递归算法O(C_n^m)的时间复杂度，动态规划算法解决该组合数问题时，时间复杂度减少为O(n²)。

同时，递归可能要重复计算子问题，而动态规划用填表技术避免重复计算。

3 3.1 动态规划概述

组合数-备忘录法

```
int dp[MAX][MAX]={0};
int ComB(int n,int m)
{   if(dp[n][m]) return dp[n][m];
    if  (m==0 || n==m)
    {       dp[n][m]=1;      return 1;}
    else
    {
            if(dp[n-1][m-1]==0)  dp[n-1][m-1]=ComB(n-1,m-1);
            if(dp[n-1][m]==0)  dp[n-1][m]=ComB(n-1,m);
            return dp[n-1][m-1] + dp[n-1][m];
    }
}
```

3　3.2 DP示例—整数分解

【问题描述】求将正整数n无序拆分成最大数为k（称为n的k拆分）的拆分方案个数，要求所有的拆分方案不重复。

比如，设n=5，k=5，对应的拆分方案有7种：

① 5=5
② 5=4+1
③ 5=3+2
④ 5=3+1+1
⑤ 5=2+2+1
⑥ 5=2+1+1+1
⑦ 5=1+1+1+1+1

3 3.2 DP示例—整数分解

动态规划法：设f(n，k)为n的k拆分方案个数,那么

（1）当n=1或k=1时，显然f(n，k)=1。

（2）当n<k时，有f(n，k)=f(n，n)。

（3）当n=k时，其拆分方案为将正整数n无序拆分成最大数为n-1的拆分方案+将n拆分成1个n（n=n）的拆分方案，后者仅仅一种，所以有

$$f(n, n)=f(n, n-1)+1。$$

3 3.2 DP示例—整数分解

（4）当n>k时，根据拆分方案中是否包含k，可以分为两种情况：

①拆分中包含k的情况：即一部分为单个k，另外一部分为$\{x_1, x_2, \cdots, x_i\}$，后者的和为n-k，后者中可能再次出现k，因此是（n-k）的k拆分，所以这种拆分方案个数为f(n-k，k)。

$$n = k + \underbrace{x_1 + x_2 + \cdots + x_i}_{f(n-k,\ k)}$$

3 3.2 DP示例—整数分解

② 拆分中不包含k的情况：则拆分中所有拆分数都比k小，即n的（k-1）拆分，拆分方案个数为f(n，k-1)。

$$n = \underbrace{x_1 + x_2 + \cdots + x_i}_{\text{最大数为}k-1}$$

$$\Downarrow$$

$$f(n, k-1)$$

因此，f(n，k) = f(n，k-1) + f(n-k，k)

3 3.2 DP示例—整数分解

状态转移方程：

$$f(n, k)=\begin{cases} 1 & \text{当}n=1\text{或者}k=1 \\ f(n, n) & \text{当}n<k \\ f(n, n-1)+1 & \text{当}n=k \\ f(n, k-1) + f(n-k, k) & \text{当}n>k \end{cases}$$

```
int fun(int n, int k)                    //递归求解算法
{  if (n==1 || k==1)
        return 1;
   else if (n<k)
        return fun(n,n);
   else if (n==k)
        return fun(n,n-1)+1;
   else
        return fun(n,k-1)+fun(n-k,k);
}
```

3.2 DP示例—整数分解

但由于子问题重叠，存在重复计算！

动态规划：设置数组dp，用dp[i][j]存放f(i, j)。初始化dp的所有元素为特殊值0。

3 3.2 DP示例—整数分解

```
int dp[MAXN+1][MAXN+1]={0};  //动态规划数组
int Split(int n, int k)                  //求解算法
{  for (int i=1;i<=n;i++)
      for (int j=1;j<=k;j++)
      {  if (i==1 || j==1)
            dp[i][j]=1;
         else if (i<j)
            dp[i][j]=dp[i][i];
         else if (i==j)
            dp[i][j]=dp[i][j-1]+1;
         else
            dp[i][j]=dp[i][j-1]+dp[i-j][j];
      }
   return dp[n][k];
}
```

3

3.2 DP示例—整数分解

Split()算法计算dp[5][5]的过程：

（1）dp[2][2]=dp[2][1]+1=1+1=2

（2）dp[2][3]=dp[2][2]=2

（3）dp[3][2]=dp[3][1]+dp[1][2]=1+1=2

（4）dp[5][2]=dp[5][1]+dp[3][2]=1+2=3

（5）dp[5][3]=dp[5][2]+dp[2][3]=3+2=5

（6）dp[5][4]=dp[5][3]+d[1][4]=5+1=6

（7）dp[5][5]=dp[5][4]+1=6+1=7

k

n	1	2	3	4	5
1	1	1	1	1	1
2	1	2	2	2	2
3	1	2	3	3	3
4	1	3	4	5	5
5	1	3	5	6	7

3 2 DP示例—整数分解_备忘录方法

$$f(n,k)=\begin{cases}1 & \text{当}n=1\text{或者}k=1\\ f(n,n) & \text{当}n<k\\ f(n,n-1)+1 & \text{当}n=k\\ f(n-k,k)+f(n,k-1) & \text{其他情况}\end{cases}$$

若dp[n][k]在之前的递归中已被求解，则无需重复递归求解。

```
int dp[MAXN][MAXN];
int dpf(int n, int k)                    //求解算法
{  if (dp[n][k]!=0) return dp[n][k];
   if (n==1 || k==1)
   {  dp[n][k]=1; return dp[n][k]; }
   else if (n<k)
   {  dp[n][k]=dpf(n, n); return dp[n][k]; }
   else if (n==k)
   {  dp[n][k]=dpf(n, k-1)+1; return dp[n][k]; }
   else
   {  dp[n][k]=dpf(n, k-1)+dpf(n-k, k); return dp[n][k]; }
}
```

3 3.2 DP示例—最大连续子序列和

【问题描述】给定一个有n（$n \geq 1$）个整数的序列，要求求出其中最大连续子序列的和。

例如

序列（-2，11，-4，13，-5，-2）的最大子序列和为20

序列（-6，2，4，-7，5，3，2，-1，6，-9，10，-2）的最大子序列和为16

规定一个序列最大连续子序列和至少是0，如果小于0，其结果为0。

3 3.2 DP示例—最大连续子序列和

（-6，2，4，-7，5，3，2，-1，6，-9，10，-2）

【问题求解1】

- ✓ 这个子序列必然是以正数开头的，因为如果以负数开头，那么去掉这个负数，可得到一个更优解。
- ✓ 一个子序列，如果一旦和为负数，那么去掉这个子序列，便能得到了一个更优解。

```
int MaxSubSequence(int A[], int n)
{
    int ThisSum,MaxSum,j;
    ThisSum = MaxSum =0;
    for(i = 1;i <= n;i++)
    {
        ThisSum += A[i];

        if(ThisSum > MaxSum)
            MaxSum = ThisSum;
        else if(ThisSum < 0)
            ThisSum = 0;
    }
    return MaxSum;
}
```

3 3.2 DP示例—最大连续子序列和

【问题求解2】 对于含有n个整数的序列a，设

$$b_j = \text{MAX}\{a_i + a_{i+1} + \cdots + a_j\} \quad (1 \leqslant i \leqslant j) \quad (1 \leqslant j \leqslant n)$$

表示$a[1..j]$的前j个元素范围内的最大连续子序列和，则b_{j-1}表示$a[1..j-1]$的前$j-1$个元素范围内的最大连续子序列和。

b[j]的含义：

$$\left.\begin{array}{l}\text{a[1\textasciitilde j]的连续子序列和} \\ \text{a[2\textasciitilde j]的连续子序列和} \\ \vdots \\ \text{a[j-1\textasciitilde j]的连续子序列和} \\ \text{a[j\textasciitilde j]的连续子序列和}\end{array}\right\} \boxed{\text{MAX}}$$

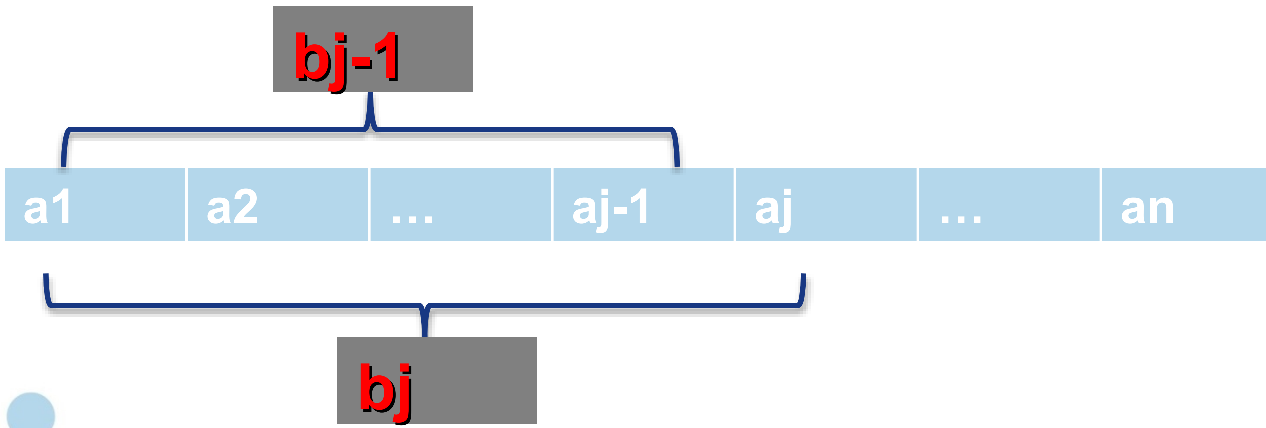
3

3.2 DP示例—最大连续子序列和

b[j]的含义：

a[1~j]的连续子序列和
a[2~j]的连续子序列和
⋮
a[j-1~j]的连续子序列和
a[j~j]的连续子序列和

MAX



发现：

bj-1>0时bj=bj-1+aj

bj-1≤0时放弃前面选取的元素，从aj开始重新选取，bj=aj。

3 3.2 DP示例—最大连续子序列和

当b_{j-1}>0时，$b_j=b_{j-1}+a_j$，当$b_{j-1}\leqslant 0$时，放弃前面选取的元素，从a_j开始重新选起，$b_j=a_j$。**用一维动态规划数组dp存放b，对应的状态转移方程如下：**

构造状态数组（表格）

$$dp[0]=0 \qquad \text{边界条件}$$

$$dp[j]=MAX\{dp[j-1]+a_j,\ a_j\} \qquad 1\le j\le n$$

构造问题的解

序列a的最大连续子序列和等于：
dp[j]（1≤j≤n）中的**最大者其下标为maxj,**即dp[max$j$]为dp数组中的最大值。在dp数组中，从该位置**maxj向前找，找到第一个dp值小于或等于0的元素dp[k],则a序列中从第k+1~maxj位置的元素和构成了该序列的最大连续子序列的和。**

3 3.2 DP示例—最大连续子序列和

dp[0]=0　　边界条件

dp[j]=MAX{dp[j-1] +a_j, a_j}　　$1 \leqslant j \leqslant n$

```
//问题表示
int n=6;
int a[]={0，-2，11，-4，13，-5，-2};      //a数组不用下标为0的元素
//求解结果表示
int dp[MAXN];
void maxSubSum()                         //求dp数组
{
    dp[0]=0;
    for (int j=1;j<=n;j++)
        dp[j]=max(dp[j-1]+a[j], a[j]);
}
```

边界条件

其他dp[j]

3 3.2 DP示例—最大连续子序列和

若a序列为（ -2，11，-4，13，-5，-2 ）

则a[]={0，-2，11，-4，13，-5，-2}，dp[0]=0(a数组下标为0的空间不存元素，即a[0]=0;求结果过程如下：

状态方程

（1）dp[1]=max(dp[0]+a[1],a[1])=max(0+(-2),-2)=-2

(2)dp[2]=max(dp[1]+a[2],a[2])=max(-2+11,11)=11

(3)dp[3]=max(dp[2]+a[3],a[3])=max(11-4,-4)=7

(4)dp[4]=max(dp[3]+a[4],a[4])=max(7+13,13)=20

(5)dp[5]=max(dp[4]+a[5],a[5])=max(20-5，-5)=15

(6)dp[6]=max(dp[5]+a[6],a[6])=max(15-2,-2)=13

问题的解

其中，dp[4]最大即maxj=4;

从maxj=4开始向前扫描，找到第一个dp[k]小于等于0即dp[1],所以对于序列a的最大连续子序列和为20，即a2~a4

3 3.2 DP示例—最大连续子序列和

```
//问题表示
int n=6;
int a[]={0, -2, 11, -4, 13, -5, -2};//a数组不用下标为0的元素
int dp[MAXN];
void maxSubSum()                    //求dp数组
{  dp[0]=0;
   for (int j=1;j<=n;j++)
      dp[j]=max(dp[j-1]+a[j], a[j]);
}
```

3 3.2 DP示例—最大连续子序列和

```
void dispmaxSum()                              //输出结果
{   int maxj=1;
    for (int j=2;j<=n;j++)                     //求dp中最大元素dp[maxj]
      if (dp[j]>dp[maxj]) maxj=j;
    for (int k=maxj;k>=1;k--)                  //找前一个值小于等于0者
      if (dp[k]<=0) break;
        printf("    最大连续子序列和: %d\n",dp[maxj]);
    printf("所选子序列: ");
    for (int i=k+1;i<=maxj;i++)
       printf("%d ",a[i]);
    printf("\n");
}
```

3 3.2 DP示例—最大连续子序列和

【算法分析】maxSubSum()的时间复杂度为O(n)。

分治法求解求解最大连续子序列和问题

【算法分析】设求解序列a[0..n-1]最大连续子序列和的执行时间为$T(n)$，第（1）、（2）两种情况的执行时间为$T(n/2)$，第（3）种情况的执行时间为O(n)，所以得到以下递推式：

$$
\begin{array}{ll}
T(n)=1 & \text{当}n=1 \\
T(n)=2T(n/2)+n & \text{当}n>1
\end{array}
$$

容易推出，$T(n)=O(n\log_2 n)$。

Chapter 3

3.2 DP示例—最长递增子序列LIS

（Longest Increasing Subsequence）

【问题描述】给定一个无序的整数序列A[1..n]，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}，

或{2，5，6，8，9}，

或{1，5，6，8，9}，结果为5。

3.2 DP示例—最长递增子序列

【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：

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

$$dp[i]=\max(dp[i],\ dp[j]+1) \qquad \text{若}\ a[i]>a[j],\ 0 \le i \le n-1,\ \boxed{0 \le j \le i-1}$$

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。

3.2 DP示例—最长递增子序列

//问题表示

序列 a[]={2，1，5，3，6，4，8，9，7}；

初始的dp[i]=1($0\leqslant i\leqslant n-1$)

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

3.2 DP示例—最长递增子序列

//问题表示
序列 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：
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

3.2 DP示例—最长递增子序列

//问题表示
序列 a[]={2，1，5，3，6，4，8，9，7}；
初始的dp[i]=1（$0 \leqslant i \leqslant n-1$）
递推求解状态方程：
同理：计算出dp[7]=5
dp[8]=4

下标	0	1	2	3	4	5	6	7	8
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

3.2 DP示例—最长递增子序列

```
//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]);}
```

3.2 DP示例—最长递增子序列

【算法分析】solve()算法中含两重循环，时间复杂度为O(n^2)。

【问题】：如何根据dp[]求出在a数组中的最长递增子序列？

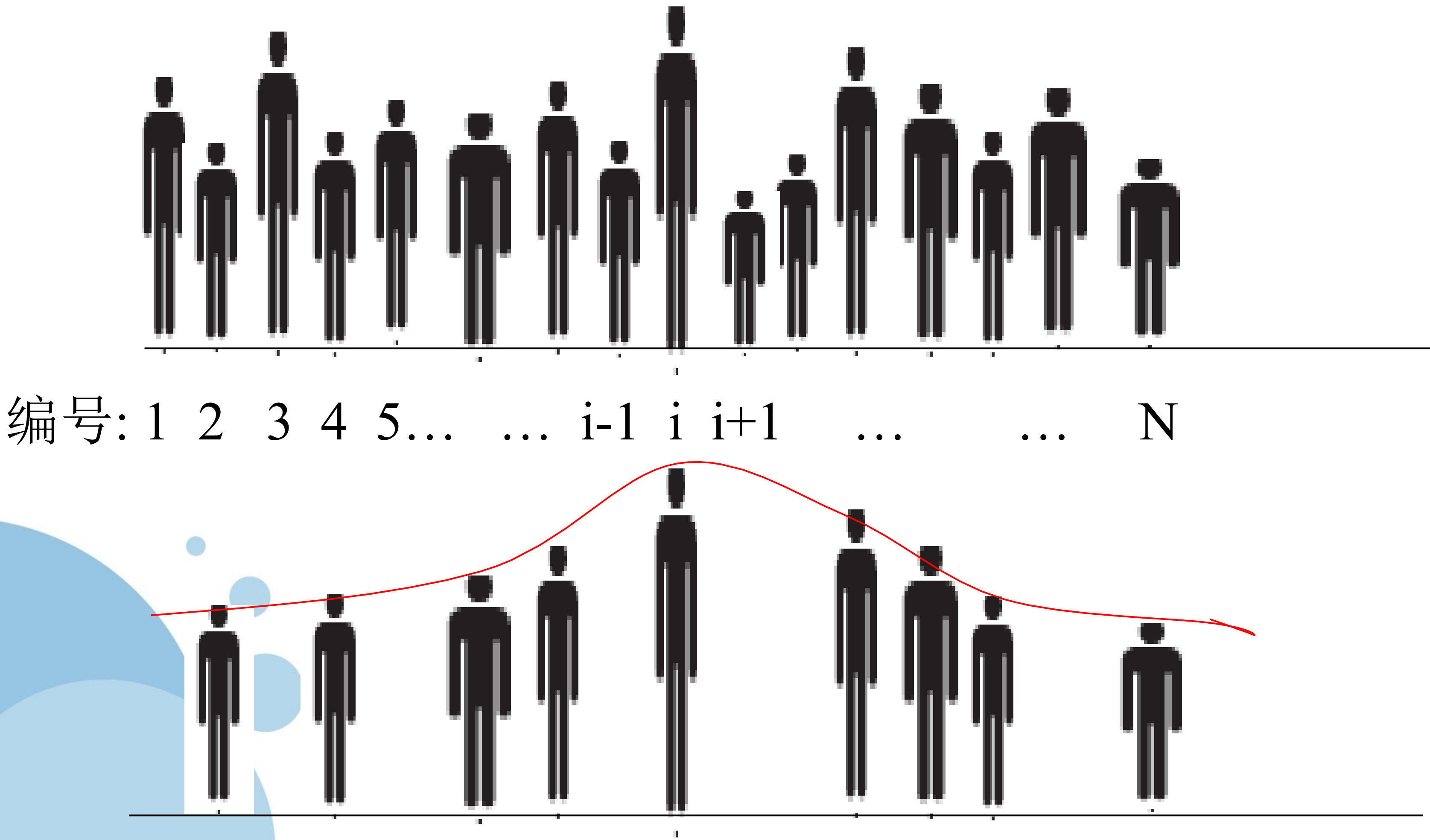
2 DP示例—合唱队形

【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，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T_1, T_2，...，T_K，且满足

$$T_1 < ... < T_i, \quad 且 T_i > T_{i+1} > ... > T_K (1 <= i <= K)。$$

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。

Chapter 3 2 DP示例—合唱队形



Chapter 3.2 DP示例—合唱队形

【问题分析】

寻找一个同学，其左边同学的身高递增序列+其右边同学的身高递减序列是最长的。

原问题转换为求最长递增序列长度和最长递减序列长度，两者相加再减1，即可得到整个合唱队形的长度。具有最优子结构的性质。

Chapter 3 2 DP示例—合唱队形

【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

2 DP示例—合唱队形

【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。

-

2 DP示例—合唱队形

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

Chapter 3

3.2 DP示例—合唱队形

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[5](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

2 DP示例—合唱队形

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

Chapter 3

3.2 DP示例—合唱队形

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

Chapter 3 2 DP示例—合唱队形

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

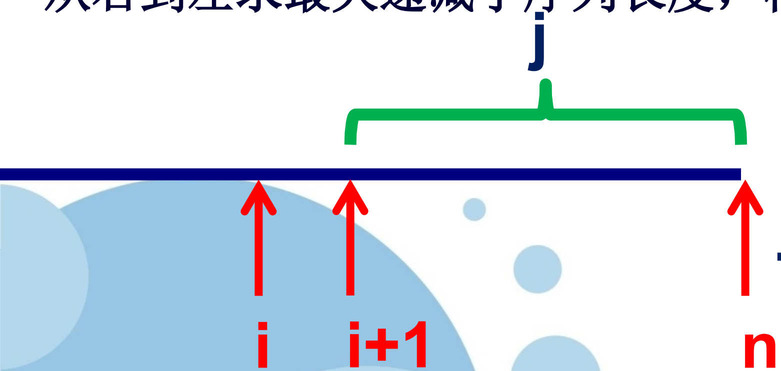
a数组用于存放身高值

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f2[i]



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

Chapter 3

2 DP示例—合唱队形

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f2[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

2 DP示例—合唱队形

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f2[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

Chapter 3

2 DP示例—合唱队形

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f2[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

2 DP示例—合唱队形

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f2[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

Chapter 3.2 DP示例——合唱队形

【例】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。

第一步：从左到右求最大上升子序列的长度，得到表a

	176	163	150	180	170	130	167	160
f1[i]	1	1	1	2	2	1	2	2

第二步：从右到左求最大上升子序列的长度，得到表b

	176	163	150	180	170	130	167	160
f2[i]	4	3	2	4	3	1	2	1

第三步：将两表中对应元素相加后减1，得到表ans。

	176	163	150	180	170	130	167	160
ans[i]	4	3	2	5	4	1	3	2

Chapter 3

2 DP示例—合唱队形

```
int* ans  ChorusRank(int n, int a[100])
{// 从左到右求最大上升子序列
  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;
    }
//从右到左求最大下降子序列
  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;
      }
 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
}
```

2 DP示例—合唱队形

【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$O(n^2)$，空间复杂度为$O(n)$。

Chapter 3

3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

【问题描述】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。求两序列A和B的最长公共子序列。子序列是指字符串的若干字符（可能不连续，但前后关系不能变）构成的序列。

A=（a，b，c，b，h，a，d，a，f）

B=（b，a，c，h，d，d，b）

3.2 DP示例—最长公共子序列LCS（Longest Common Subsequence）

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

A=（*a*，*b*，*c*，*b*，*d*，*a*，*b*）

B=（*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）。

3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

【问题求解】 若设A=（a_0，a_1，…，a_{m-1}）（含m个字符），

B=（b_0，b_1，…，b_{n-1}）（含n个字符），

设Z=（z_0，z_1，…，z_{k-1}）（含k个字符）为它们

证明有以下性质：

如何把问题划分成子问题？

- 如果$a_{m-1}=b_{n-1}$，则$z_{k-1}=a_{m-1}=b_{n-1}$，且（z_0，z_1，…，z_{k-2}）是（a_0，a_1，…，a_{m-2}）和（b_0，b_1，…，b_{n-2}）的一个最长公共子序列。
- 如果$a_{m-1}\neq b_{n-1}$且$z_{k-1}\neq a_{m-1}$，则（z_0，z_1，…，z_{k-1}）是（a_0，a_1，…，a_{m-2}）和（b_0，b_1，…，b_{n-1}）的一个最长公共子序列。
- 如果$a_{m-1}\neq b_{n-1}$且$z_{k-1}\neq b_{n-1}$，则（z_0，z_1，…，z_{k-1}）是（a_0，a_1，…，a_{m-1}）和（b_0，b_1，…，b_{n-2}）的一个最长公共子序列。

3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

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）

$(x_0, x_1, \dots, x_{i-1})$

‖

$(y_0, y_1, \dots, y_{j-1})$

⇒ dp[i][j]=dp[i-1][j-1]+1

例如：

a	b	c	x	y
				‖
		b		y

Chapter 3 3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

定义二维动态规划数组dp，其中dp[i][j]为子序列（a_0，a_1，…，a_{i-1}）和（b_0，b_1，…，b_{j-1}）的最长公共子序列的长度。

情况2：$a[i-1] \neq b[j-1]$（当前两个字符不同）

（x_0，x_1，…，x_{i-2}，x_{i-1}）

dp[i][j-1]

dp[i-1][j]

（y_0，y_1，…，y_{j-2}，y_{j-1}）

dp[i][j]=MAX(dp[i][j-1]，dp[i-1][j])

3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

dp[i][j]为子序列（a_0，a_1，…，a_{i-1}）和（b_0，b_1，…，b_{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	a[i-1]=b[j-1]
dp[i][j]=MAX(dp[i][j-1]，dp[i-1][j])	a[i-1]≠b[j-1]

显然，dp[m][n]为最终结果。

3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

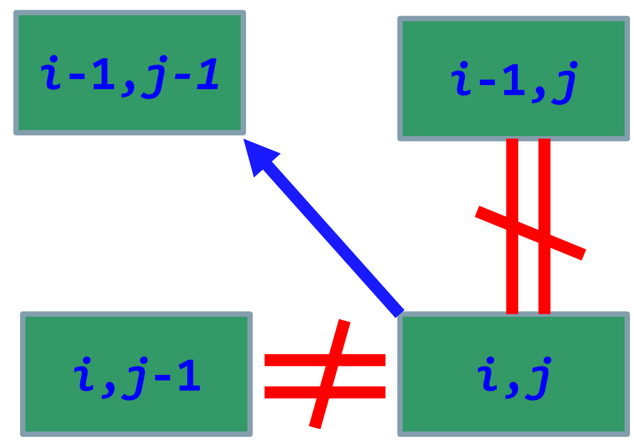
那么如何由dp求出LCS呢？ dp ⇨ LCS

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

$dp[i][j] \neq dp[i][j-1]$（左边）并且$dp[i][j] \neq dp[i-1][j]$（上方）值时：

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

将$a[i-1]$添加到LCS中。



3.2 DP示例—最长公共子序列LCS
(Longest Common Subsequence)

i-1,*j*-1	*i*-1,*j*
i,*j*-1	*i*,*j*

- 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

Y

		a	c	b	b	a	b	d	b	b	
X		0	1	2	3	4	5	6	7	8	9
a	0	0	0	0	0	0	0	0	0	0	0
b	1	0	1	1	1	1	1	1	1	1	1
c	2	0	1	1	2	2	2	2	2	2	2
b	3	0	1	2	2	2	2	2	2	2	2
d	4	0	1	2	3	3	3	3	3	3	3
b	5	0	1	2	3	3	3	3	4	4	4
	6	0	1	2	3	4	4	4	4	5	5

（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

Y

X			a	c	b	b	a	b	d	b	b
		0	1	2	3	4	5	6	7	8	9
a	0	0	0	0	0	0	0	0	0	0	0
b	1	0	1	1	1	1	1	1	1	1	1
c	2	0	1	1	2	2	2	2	2	2	2
b	3	0	1	2	2	2	2	2	2	2	2
d	4	0	1	2	3	3	3	3	3	3	3
b	5	0	1	2	3	3	3	3	4	4	4
	6	0	1	2	3	4	4	4	4	5	5

（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

Y

X			a	c	b	b	a	b	d	b	b
		0	1	2	3	4	5	6	7	8	9
a	0	0	0	0	0	0	0	0	0	0	0
b	1	0	1	1	1	1	1	1	1	1	1
c	2	0	1	1	2	2	2	2	2	2	2
b	3	0	1	2	2	2	2	2	2	2	2
d	4	0	1	2	3	3	3	3	3	3	3
b	5	0	1	2	3	3	3	3	4	4	4
	6	0	1	2	3	4	4	4	4	5	5

（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"

3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

```
#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
```

Chapter 3

3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

```
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a、b的所有字符
      {  if (a[i-1]==b[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$O(m \times n)$

3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

```
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(a[i-1]);  //subs中添加a[i-1]
       i--; j--; k--;
     }
}
```

3.2 DP示例—最长公共子序列LCS (Longest Common Subsequence)

【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($m \times n$)。

3.2 DP示例—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。

如何找到两个字符串a[0..m-1]和b[0..n-1]的编辑距离呢？

3.2 DP示例—编辑距离问题

【应用1】一篇公众号文章由于疏忽，写错位了一段内容，需要修改。但是公众号文章最多只能修改 20 个字，且只支持增、删、替换操作（编辑距离问题，哈哈），于是某人就用算法求出了一个最优方案，只用了 16 步就完成了修改。

【应用2】拼写检查可以根据一个拼错的字和其他正确的字的编辑距离，判断哪一个（或哪几个）是比较可能的字。

3.2 DP示例—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。

Chapter 3

3.2 DP示例—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）。

3.2 DP示例—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]（$1 \leqslant i \leqslant m$）与$b$[0..$j$-1]（$1 \leqslant j \leqslant n$）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）。

3.2 DP示例—编辑距离问题

两种特殊情况（边界条件）

A=（A，B，C，D，A，B），B=（）；如何将A变成B?

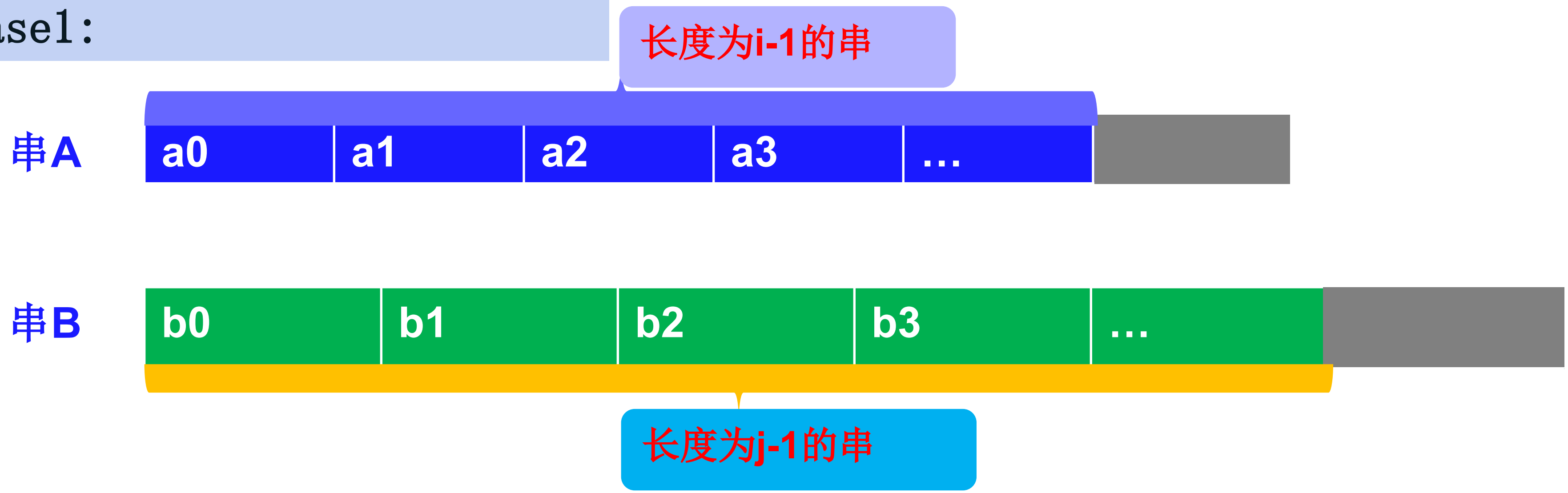
- 当*B*串空时，要删除*A*中全部字符转换为*B*，即 dp[i][0]=i（删除*A*中 i个字符，共i次操作）；

A=（），B=A，B，C，D，A，B） ；如何将A变成B?

- 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）。

3.2 DP示例—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][
[$i-1$][$j-1$]。

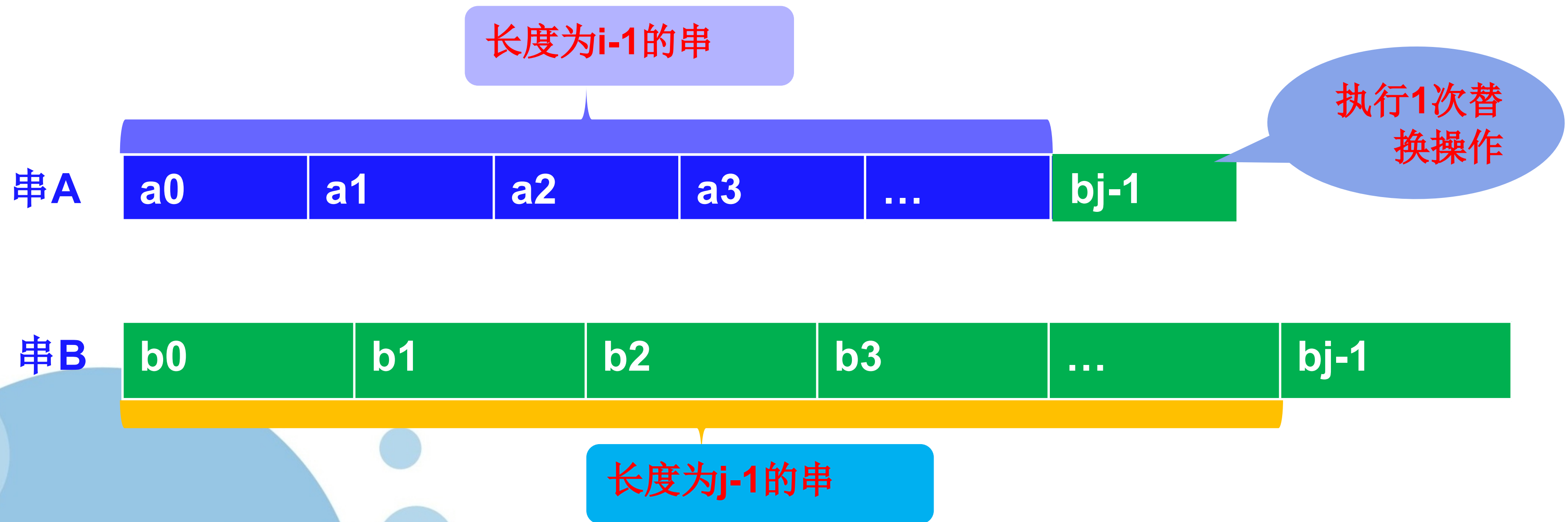
长度为i的A串变换为长度为j的B串的最优编辑距离

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

3.2 DP示例—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:替换



- 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：

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

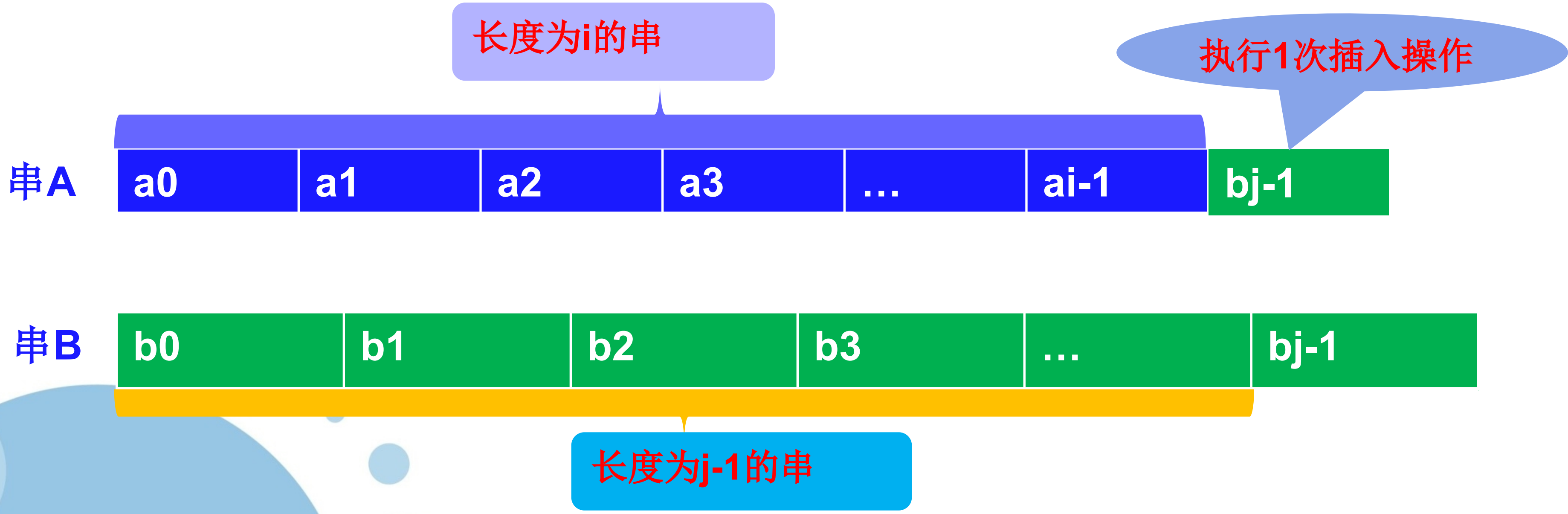
3.2 DP示例—编辑距离问题

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：插入



- 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：

$$dp[i][j]=dp[i][j-1]+\boxed{1}$$

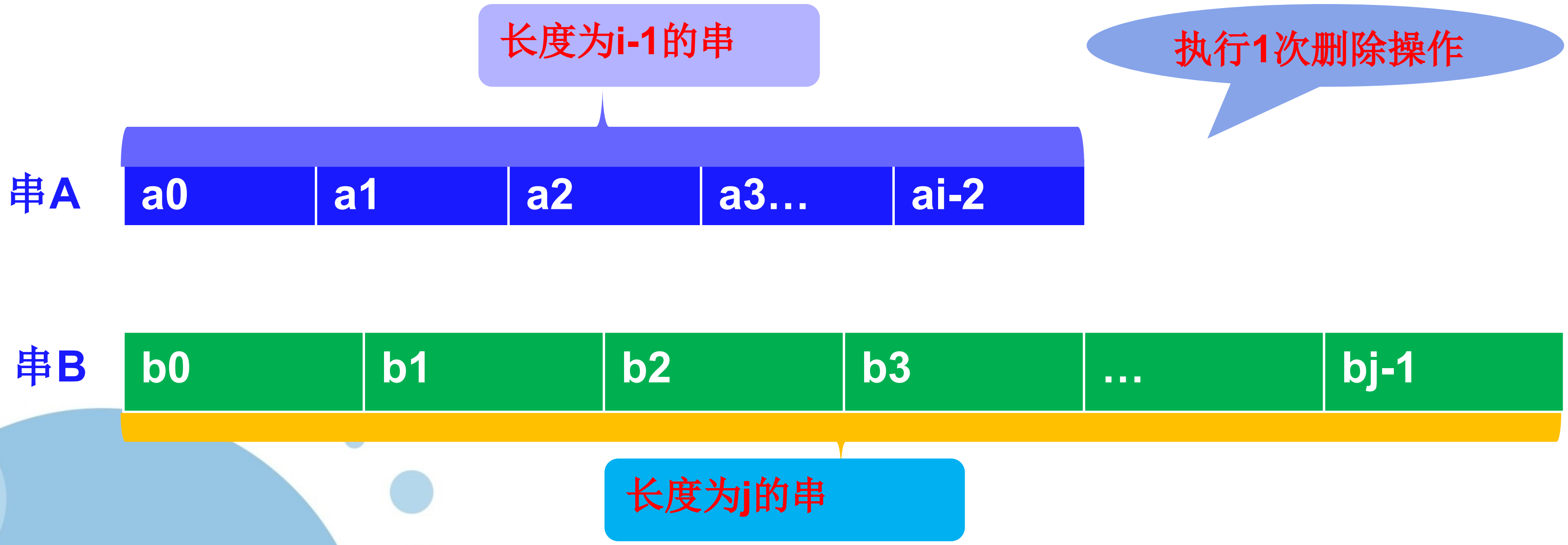
3.2 DP示例—编辑距离问题

构造状态转移方程dp[i][j]

case2:

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

操作3：删除



- 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：

$$dp[i][j]=dp[i-1][j]+\boxed{1}$$

3.2 DP示例—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]。

构造状态转移方程dp[i][j]
case2:

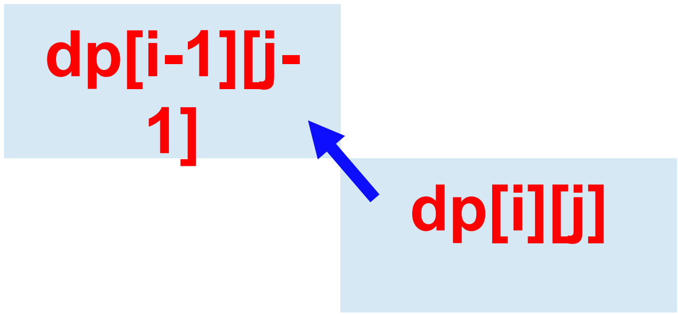
当$a[i-1]\neq b[j-1]$时：可以通过如下三种操作之一进行转换：

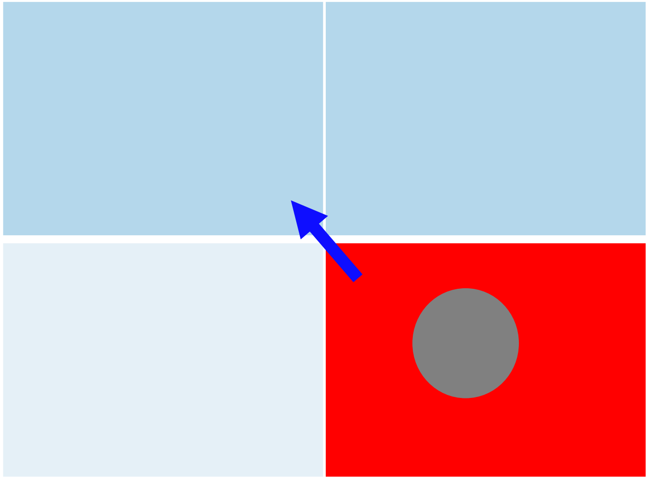
dp[i][j]=min(dp[i][j]=dp[i-1][j-1]+1,
dp[i][j]=dp[i][j-1]+1,
dp[i][j]=dp[i-1][j]+1)

Chapter 3

3.2 DP示例—编辑距离问题

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]。

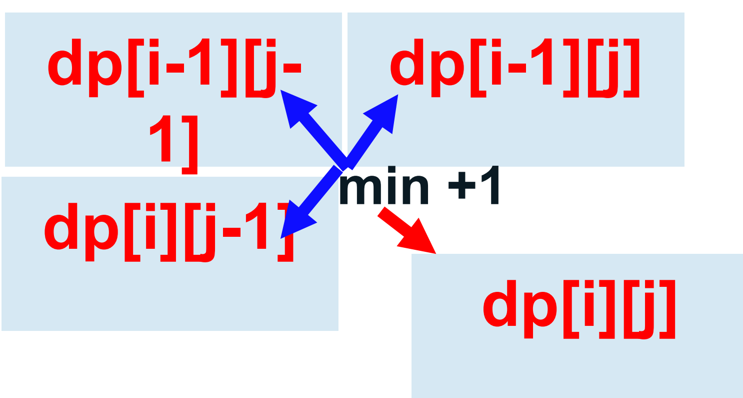


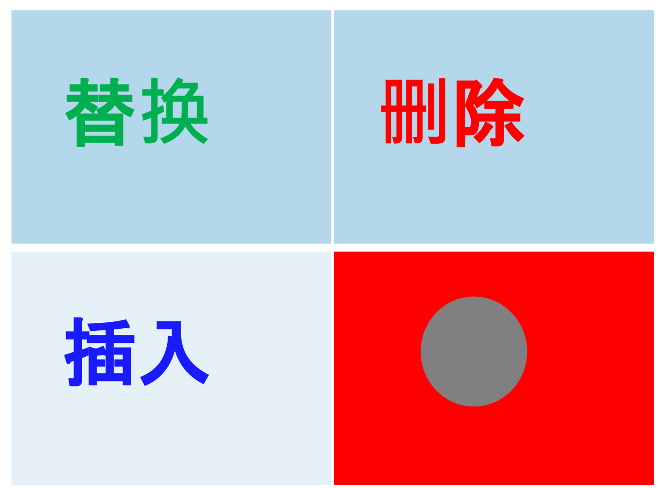


Chapter 3 3.2 DP示例—编辑距离问题

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——示例

例如A="sfdqxbw",B="gfdgw"

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

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

dp[i][j]	0 (B 为空串)	1	2	3	4	5
A 为空串 0	0	1	2	3	4	5
i → 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;
     }
}
```

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

3.2 DP示例—矩阵连乘问题

- 给定n个矩阵 $\{A_1, A_2, \ldots, A_n\}$， 其中 A_i与 A_{i+1} 是可乘的， $i = 1,2,\ldots,n-1$ 。考察这n个矩阵的连乘积：

$$A_1 A_2 \ldots A_n$$

- 由于矩阵乘法满足结合律，所以计算矩阵的连乘可以有许多不同的计算次序。这种计算次序可以用加括号的方式来确定。

- 若一个矩阵连乘积的计算次序完全确定，也就是说该连乘积已完全加括号，则可以依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积。

3.2 DP示例—矩阵连乘问题

完全加括号的矩阵连乘积可递归地定义为：

（1）单个矩阵是完全加括号的；

（2）矩阵连乘积A是完全加括号的，则A可表示为2个完全加括号的矩阵连乘积B和C的乘积并加括号，即 $A=(BC)$

每一种完全加括号对应于一个矩阵连乘积的计算次序，而矩阵连乘积的计算次序与其计算量有密切的关系。

3.2 DP示例—矩阵连乘问题

计算两个矩阵乘积的标准算法：

```
void matrixMultiply(double a[][], double b[][], double c[][],
              int ra, int ca, int rb, int cb)
{
      if(ca!=rb)
        throw new IllegalArgumentException("矩阵不可乘");

      for( int i=0;i<ra;i++)
          for( int j=0;j<cb;j++)
          {
            int sum=a[i][0]*b[0][j];
           for(int k=1;k<ca;k++)
           sum=sum+a[i][k]*b[k][j];
          c[i][j]=sum;
          }
}
```

注： ra，ca 和 rb，cb分别表示矩阵A和B的行数和列数。

Chapter 3

3.2 DP示例—矩阵连乘问题

通过矩阵乘积标准算法可知：若矩阵A是 $p\times q$ 矩阵，B是 $q\times r$矩阵，则乘积C=AB是 $p\times r$ 矩阵，总共需要 pqr 次数乘得到。

这样可以计算每一种完全加括号方式的计算量，如

设有四个矩阵 A, B, C, D ，它们的维数分别是：

$A = 50\times 10$ $B = 10\times 40$ $C = 40\times 30$ $D = 30\times 5$

总共有五种完全加括号的方式

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

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

16000, 10500, 36000, 87500, 34500

不同计算顺序的差别

设矩阵A_1，A_2和A_3分别为10×100，100×5和5×50的矩阵，现要计算$A_1A_2A_3$ 。

若按$((A_1A_2)A_3)$来计算，则需要的数乘次数为10×100×5 + 10×5×50 = 7500

若按$(A_1(A_2\ A_3))$来计算，则需要的数乘次数为100 ×5 ×50+ 10×100×50 = 75000

两种计算顺序的计算量竟然相差10倍！

3.2 DP示例—矩阵连乘问题

给定n个矩阵 $\{A_1, A_2, \cdots, A_n\}$，其中Ai与Ai+1是可乘的，i=1，2…，n-1。如何确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的数乘次数最少。

1. 穷举法：列举出所有可能的计算次序，并计算出每一种计算次序相应需要的数乘次数，从中找出一种数乘次数最少的计算次序。

算法复杂度分析：

对于n个矩阵的连乘积，设其不同的计算次序为P(n)。

由于每种加括号方式都可以分解为两个子矩阵的加括号问题：(A1...Ak)(Ak+1...An)可以得到关于P(n)的递推式如下：

$$P(n)=\begin{cases} 1 & n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n>1 \end{cases} \Rightarrow P(n)=\Omega(4^n/n^{3/2})$$

P(n)随n的增长呈指数增长。因此，穷举法不是一个有效算法。

2.动态规划

- 将矩阵连乘积$A_iA_{i+1}\dots A_j$ 简记为A[i:j] ，这里i≤j

- 考察计算A[i:j]的最优计算次序。设这个计算次序在矩阵Ak和Ak+1之间将矩阵链断开，i≤k<j，则其相应完全加括号方式为 $(A_iA_{i+1}\dots A_k)(A_{k+1}A_{k+2}\dots A_j)$

- **计算量：** A[i:k]的矩阵连乘计算量+

  A[k+1:j]矩阵连乘的计算量+

  计算A[i:k]的结果矩阵和A[k+1:j]结果矩阵相乘的计算量

3.2 DP示例—矩阵连乘问题

1. 分析最优解的结构

- 特征：计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的。
- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

3.2 DP示例—矩阵连乘问题

1. 分析最优解的结构

- 将$A_iA_{i+1}\cdots A_j$的矩阵连乘问题记为A[i：j]。
- 设A[1：n] 的一个最优解是在A_k和A_{k+1}处断开的，即A[1：n] = (A[1：k] A[k+1：n])，则A[1：k]和A[k+1：n]也分别是其最优解。
- 否则，若有A[1：k]的一个计算次序的计算量更少的话，则用此计算次序替换原来的次序，则得到A[1：n]一个更少计算量。矛盾。同理A[k+1：n]也是最优解。

2. 建立递归关系

- 令m[i][j]，1≤i≤j≤n，为计算A[i:j] 的**最少数乘次数**，则原问题为m[1][n]。
- 当i = j时，A[i:j]为单一矩阵， m[i][j] = 0;
- 当i<j时，利用最优子结构性质有：

$$m[i][j] = \min_{i\le k<j}\{m[i][k] + m[k+1][j] + p_{i-1}p_kp_j\}$$

其中矩阵A_i，1≤i≤n的维数为$p_{i-1}\times p_i$。

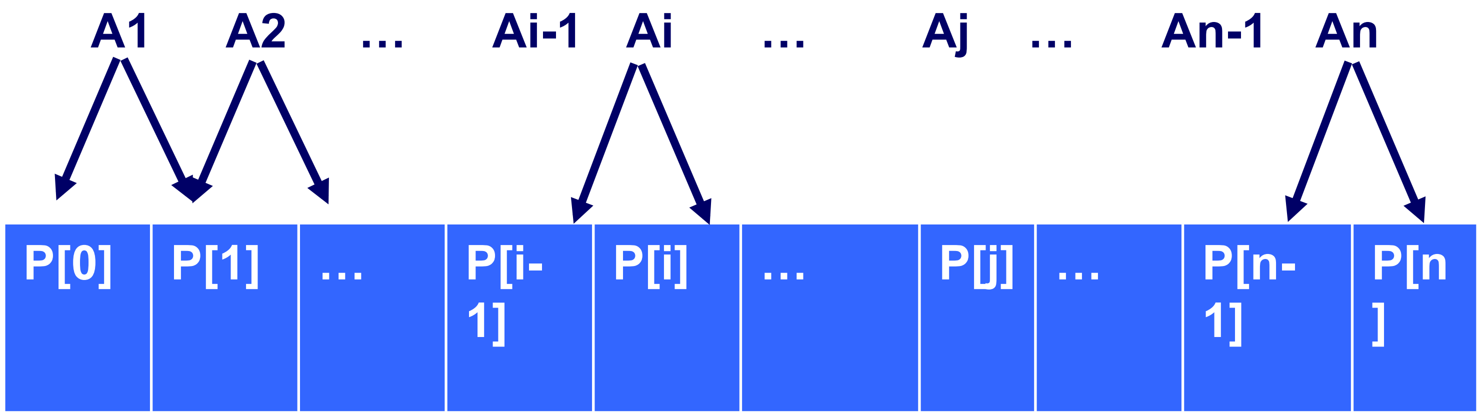
- 可以递归地定义m[i, j]为：

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

k **的位置只有** $j-i$**种可能，即{i,i+1,...,j-1}**。

Chapter 3

3.2 DP示例—矩阵连乘问题



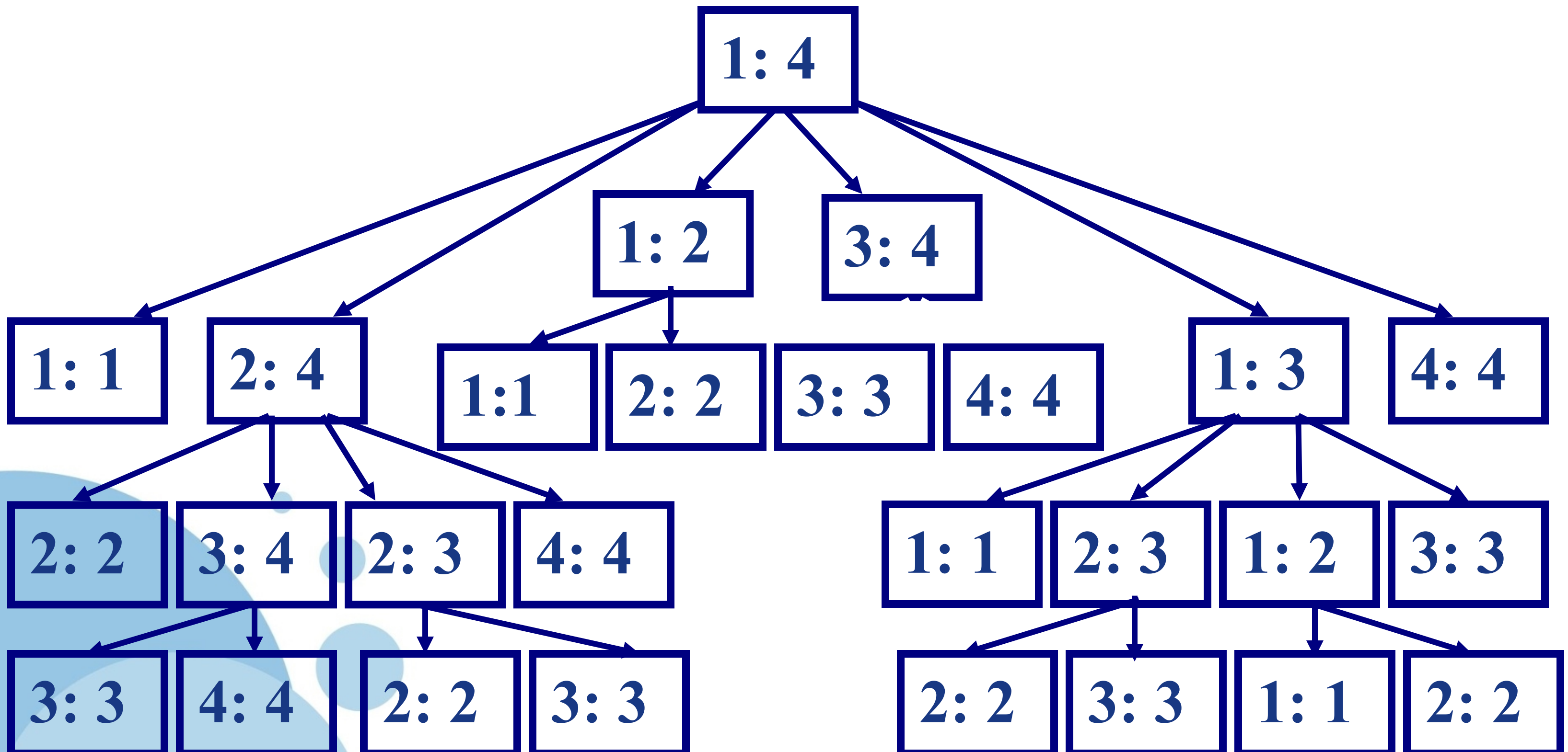
其中矩阵A_i，$1 \leqslant i \leqslant n$的维数为$p_{i-1} \times p_i$。

只需数组P[n+1]即P[0..n]就可存放各矩阵的行列数。

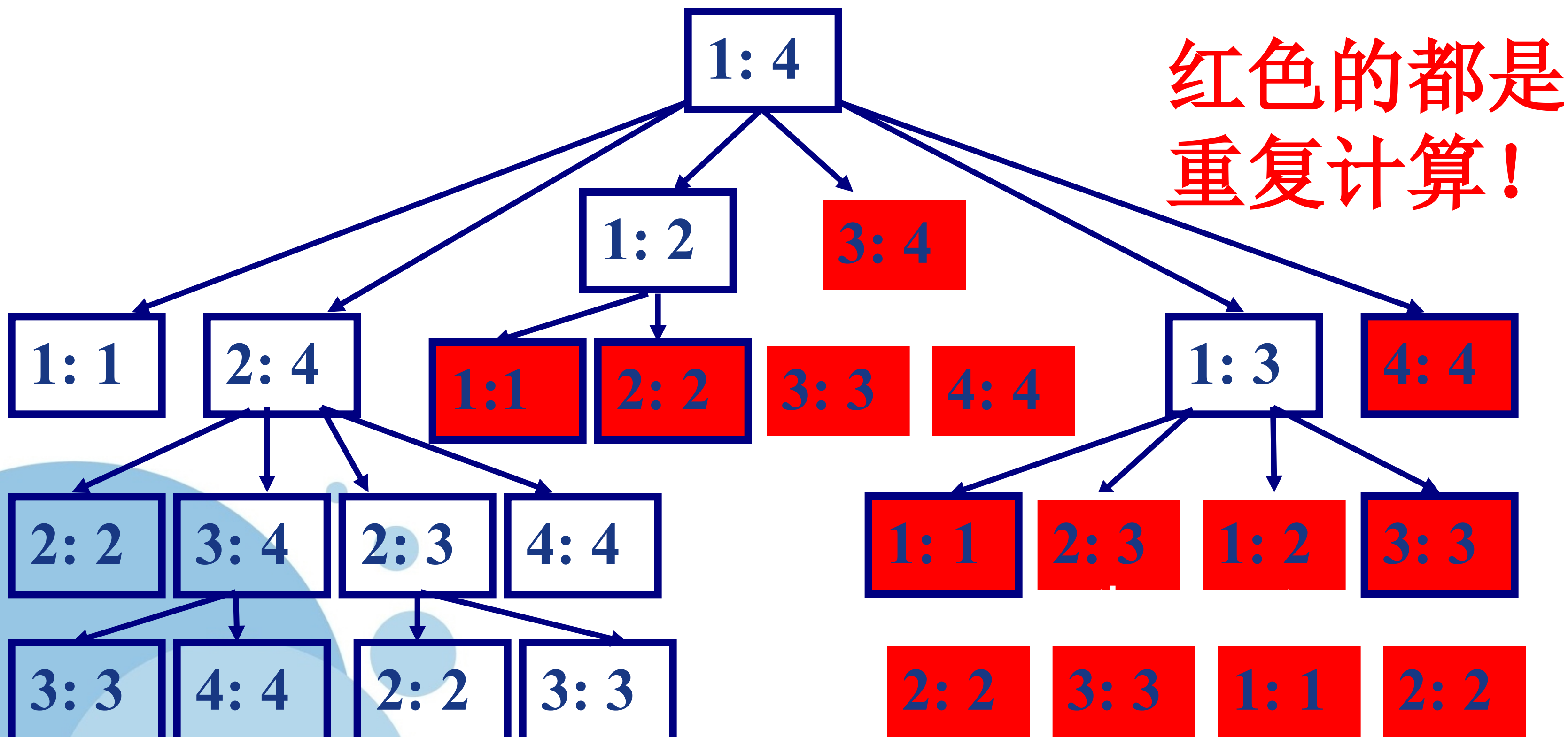
注意：因为这n个矩阵可乘，故后一个矩阵的行数就是一个矩阵的列数。 P[0]为A_1的行数, P[1]为A_2的行数，也是A_1的列数, 其余类推。最后的P[n]是A_n的列数。

3.2 DP示例—矩阵连乘问题

- 矩阵连乘为题的递归**自顶向下**地执行，如A[1: 4]计算：



矩阵连乘问题的递归执行中有大量重复计算：

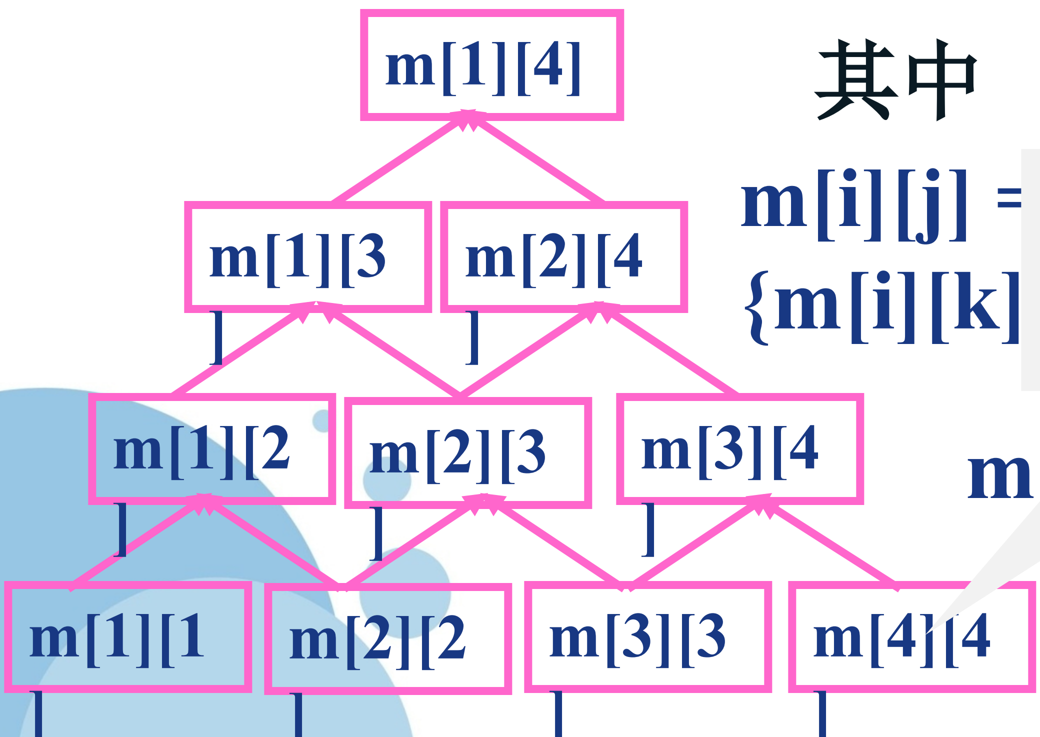


3. 计算最优值

用动态规划算法解此问题，可依据其递归式以自底向上的方式进行计算。在计算过程中，保存已解决的子问题答案。每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算，最终得到多项式时间的算法。

自底向上的计算

例如对于$A_1A_2A_3A_4$，依据递归式以自底向上的方式计算出各个子问题，其过程如下：



其中

$m[i][j] =$ … $\{m[i][k]$ … $p_j\}$

最简单情况是单个矩阵的计算。

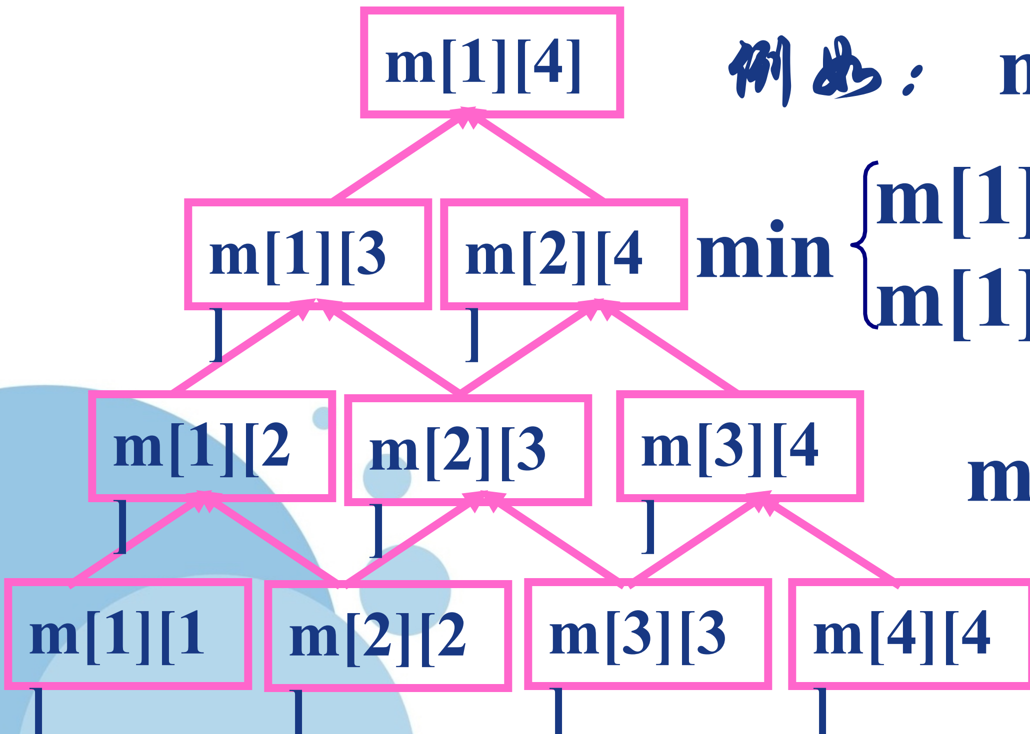
$m[i][i+1] = p_{i-1}p_ip_{i+1}$

$m[i][i] = 0$

Chapter 3

3.2 DP示例—矩阵连乘问题

例如对于$A_1A_2A_3A_4$，依据递归式以自底向上的方式计算出各个子问题，其过程如下：



例如： $m[1][3] =$

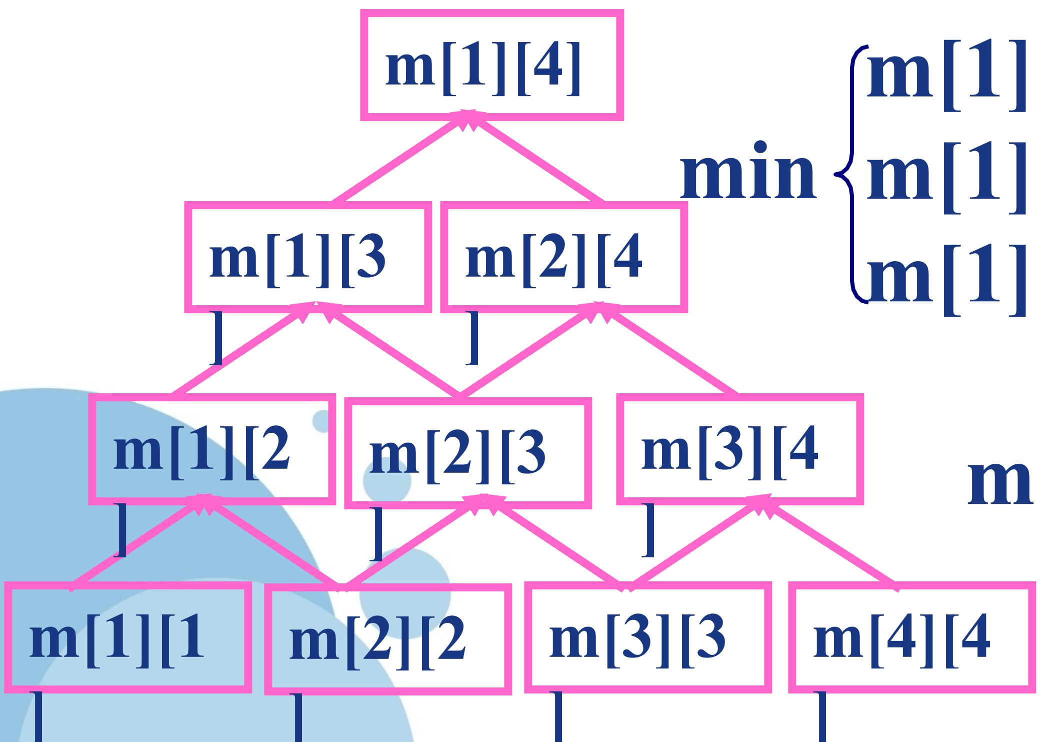
$$\min\begin{cases} m[1][1]+m[2][3]+p_0p_1p_3 \\ m[1][2]+m[3][3]+p_0p_2p_3 \end{cases}$$

$$m[i][i+1] = p_{i-1}p_ip_{i+1}$$

$$m[i][i] = 0$$

Chapter 3 3.2 DP示例—矩阵连乘问题

例如对于$A_1A_2A_3A_4$，依据递归式以自底向上的方式计算出各个子问题，其过程如下：



例如：

$$m[1][4] = \min\begin{cases} m[1][1]+m[2][4]+p_0p_1p_4 \\ m[1][2]+m[3][4]+p_0p_2p_4 \\ m[1][3]+m[4][4]+p_0p_3p_4 \end{cases}$$

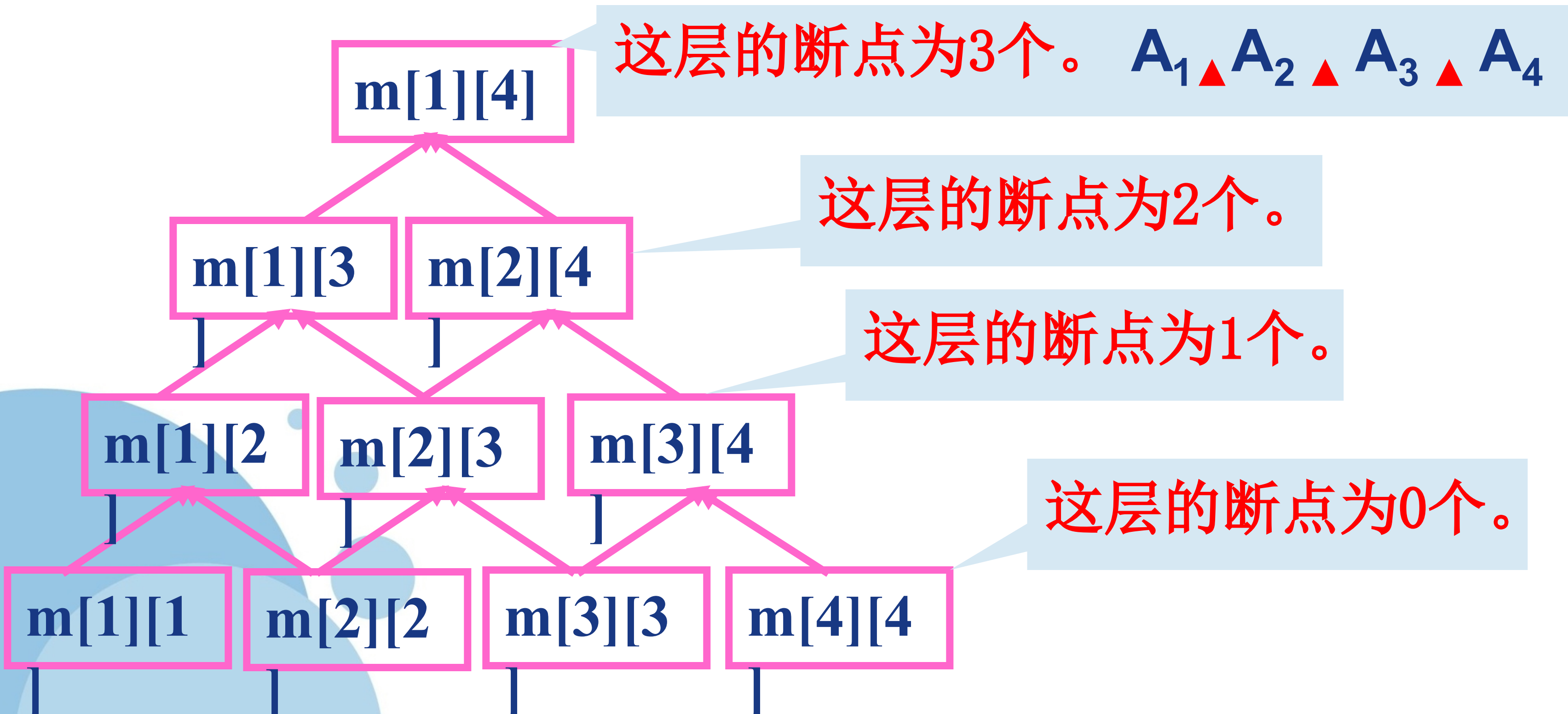
$$m[i][i+1] = p_{i-1}p_ip_{i+1}$$

$$m[i][i] = 0$$

3.2 DP示例—矩阵连乘问题

自底向上的计算

- 例如对于$A_1A_2A_3A_4$，依据递归式以自底向上（动态规划）的方式计算出各个子问题，其过程如下：



矩阵连乘算法

MatrixChain(形参表)

{

初始化；

自底向上地计算每一个m[i][j]并将结果填入表中。

}

初始化是将m[i][i]，即对角线元素，赋值为0。

底是m[i][i]，即对角线元素。最顶层是m[1][n]。

矩阵连乘算法的数据结构

- n（连乘的矩阵个数）
- P[n+1]：记录A1~An矩阵的行数和An的列数
- 算法需要两个二维数组：
  - 二维矩阵m[n][n]。其每个元素m[i][j]，1≤i≤j≤n 为A[i:j] 的最少数乘次数。
  - 二维矩阵s[n][n]，其元素s[i][j]，1≤i≤j≤n 为计算A[i:j] 的断点最佳位置。

矩阵连乘算法

```
void matrixChain(int n, int p[], int m[][], int s[][])
{

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

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

m[i][j]

先将对角线元素m[i][i]赋值为 0。

3.2 DP示例—矩阵连乘问题

```
void matrixChain(int n,int p [], int m [][], int s[][])
{ …
 for (int r = 2; r <= n; r++)
   for (int i = 1; i <= n – r +1; i+
     int j = i + r – 1;//j为长度为
     m[i][j]=m[i+1][j]+p[i–1]*p[i]*p[j]; //k = i;
     s[i][j]=i;
     for (int k = i+1; k < j; k++) {
       int t = m[i][k]+m[k+1][j]+p[i–1]*p[k]*p[j];
       if (t < m[i][j]) {
          m[i][j] = t;
          s[i][j] = k;}
       }
    }
}
```

控制连乘矩阵的链长度r=2~n，对(i, j)间的每个断点k，计算m[i, j] = m[i: k]+m[k+1: j]+p[i–1]*p[k] *p[j]，并记下较小的m[i][j]及相应的断点k。

```
for (int r = 2; r <= n; r++)
  for (int i = 1; i <= n – r +1; i++) {
    int j = i + r – 1; //j为长度为r的链的最后一个相乘矩阵编号
    m[i][j]=m[i+1][j]+p[i–1]*p[i]*p[j];  //k==i;
    s[i][j]=i;
    for (int k = i+1; k < j; k++) {
      int t = m[i][k]+m[k+1][j]+p[i–1]*p[k]*p[j];
      if (t < m[i][j]) {
        m[i][j] = t;
```

当r > 2，每对(i, j)中的断点k有r – 1个，越往高层断点数目越多。这样自底向上完成整个m[i][j]的计算。

```
}
```

r=2 j=i+r-1

i=1

i=2

i=3

i=4

i=5

	1	2	3	4	5	6
1	0 {A1}	15750= {A1}A2	A1{A2A3} {A1A2}A3	A1{A2A3A4} {A1A2}{A3A4} {A1A2A3}A4	11875	15125
2	0	0 {A2}	2625= {A2}A3	A2{A3A4} {A2A3}A4	A2{A3A4A5} {A2A3}{A4A5} {A2A3A4}A5	10500
3	0	0	0 {A3}	750= {A3}A4	A3{A4A5} {A3A4}A5	A3{A4A5A6} {A3A4}{A5A6} {A3A4A5}A6
4	0	0	0	0 {A4}	1000= {A4}A5	A4{A5A6} {A4A5}A6
5	0	0	0	0	0 {A5}	5000= {A5}A6
6	0	0	0	0	0	0 {A6}

r=3 j=i+r-1

i=1

i=2

i=3

i=4

	1	2	3	4	5	6
1	[illegible]	[illegible]	[illegible]	A1{A2A3A4} {A1A2}{A3A4} {A1A2A3}A4	11875	15125
2	0	[illegible]	[illegible]	[illegible] A4	A2{A3A4A5} {A2A3}{A4A5} {A2A3A4}A5	10500
3	0	0	0 [illegible]	750= [illegible]	[illegible] }A5	A3{A4A5A6} {A3A4}{A5A6} {A3A4A5}A6
4	0	0	0	0 [illegible]	1000= [illegible]	[illegible]
5	0	0	0	0	0 {A5}	5000= {A5}A6
6	0	0	0	0	0	0 {A6}

MatrixChain的运行举例

MatrixChain用矩阵m[i][j]存放子问题A[i: j]的最小计算量;
s[i][j]存放子问题A[i: j] 相应的断点位置在第s[i][j]个矩阵处。

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

m[i][j]

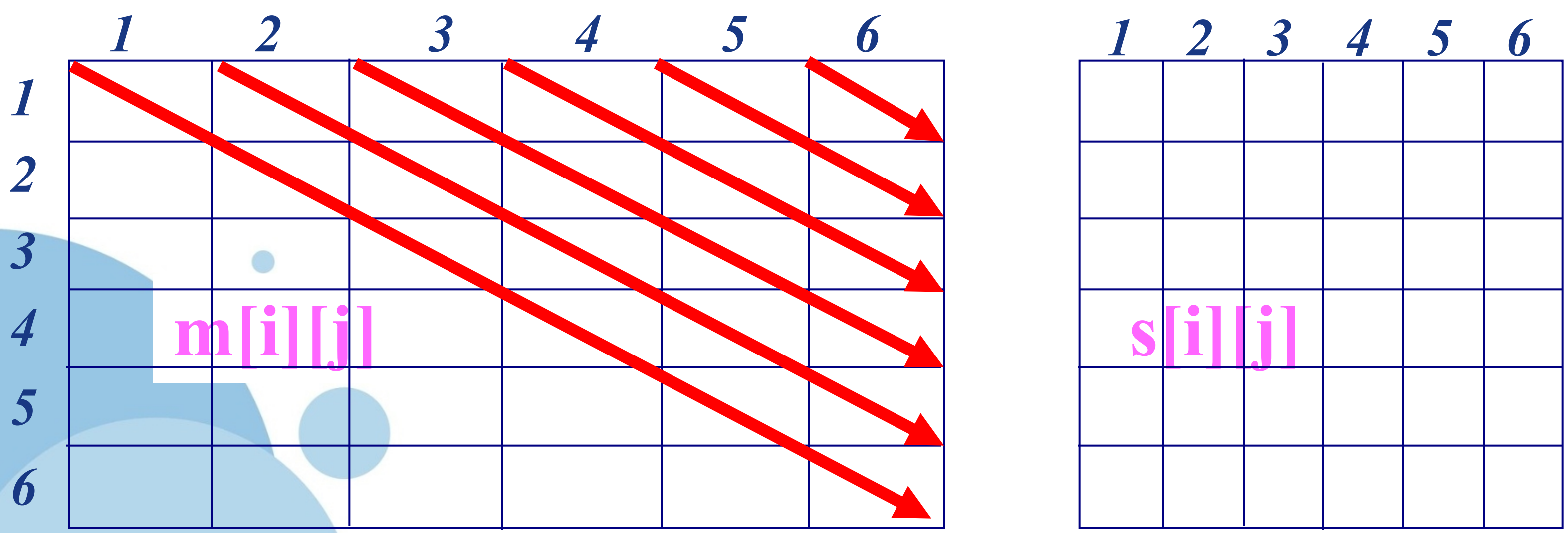
	1	2	3	4	5	6

s[i][j]

Chapter 3 3.2 DP示例—矩阵连乘问题

MatrixChain的运行举例

MatrixChain将如下面红色箭头所示的过程逐个计算子问题A[i: j] 。



MatrixChain的运行举例

- 设要计算矩阵连乘积$A_1A_2A_3A_4A_5A_6$，其维数分别为$30\times35, 35\times15, 15\times5, 5\times10, 10\times20, 20\times25$，即$p_0=30, p_1=35, p_2=15, p_3=5, p_4=10, p_5=20, p_6=25$。

m[i][j]

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

s[i][j]

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

MatrixChain的运行举例

执行for (int i = 1; i <= n; i++) m[i][i] = 0后将对角线元素全部置零，即子问题A[i][i] = 0。

m[i][j]

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

s[i][j]

	1	2	3	4	5	6

MatrixChain的运行举例

- 当r=2，对每个i，完成相邻矩阵数乘次数计算，即m[i
  [i–1]*p[i]*p[j]，并在s[i][j]中添入了相应的断点。
- $p_0=30, p_1=35, p_2=15, p_3=5, p_4=10, p_5=20, p_6=25$。

m[i][j]

	1	2	3	4	5	6
1	0	15750				
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1				
2			2			
3				3		
4					4	
5						5
6						

MatrixChain的运行举例

- 当链长r = 3，i = 1时，即计算A[1:3]断点k有两个：
- 对断点k = 1，计算A[1:1]A[2:3]有m[1][3] = m[2][3] + p[0]*p[1]*p[3] = 2625 +30*35*5 = 7875

i\j	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

m[i][j]

i\j	1	2	3	4	5	6
1		1	1			
2			2			
3				3		
4					4	
5						5
6						

s[i][j]

MatrixChain的运行举例

- 当r=3，i=1时，即计算A[1:3]断点k有两个：
- 对断点k = 2，计算A[1:2]A[3:3]有m[1][3] = m[1][2] + m[3][3] + p[0]*p[2]*p[3] = 15750 >7875。m[1][3]仍为7875。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2			
3				3		
4					4	
5						5
6						

MatrixChain的运行举例

- 当r＝3，i＝2时，即计算A[2:4]断点k有两个：
- 对断点k＝2，计算A[2:4]即计算A[2:2]A[3:4]有 m[2][4] = m[2][2] m[3][4] + p[1]*p[2]*p[4] = 750 +35*15*10 = 6000。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	6000		
3			0	750		
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	2		
3				3		
4					4	
5						5
6						

MatrixChain的运行举例

- 当r＝3，i＝2时，即计算A[2:4],断点k有两个：
- 对断点k＝3，计算A[2:3]A[4:4]有m[2][4]＝m[2][3]＋m[4][4]＋p[1]*p[3]*p[4]＝2625＋0＋35*5*10 ＝4375 ＜ 6000。m[2][4]改为4375，断点改为3。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	6000 4375		
3			0	750		
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	2 3		
3				3		
4					4	
5						5
6						

Chapter 3

3.2 DP示例—矩阵连乘问题

MatrixChain的运行举例

- 当r＝3，i＝3时，即计算A[3:5]断点k有两个：
- 对断点k＝3，计算A[3:3]A[4:5]有m[3][5] = m[3][3] + m[4][5] + p[2]*p[3]*p[5] = 1000 +15*5*20 = 2500。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	
5					0	5000
6						0

s[i][j]

1	2	3	4	5	6
	1	1			
		2	3		
			3	3	
				4	
					5

MatrixChain的运行举例

- 当r＝3，i＝3时，即计算A[3:5],断点k有两个：
- 对断点k＝4，计算A[3:4]A[5:5]有m[3][5]＝m[3][4]＋m[5][5]＋p[2]*p[4]*p[5]＝750＋0＋15*10*20＝3750 ＞ 2500。m[3][5]仍为2500，断点仍为3。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	3		
3				3	3	
4					4	
5						5
6						

MatrixChain的运行举例

- 当r = 3，i = 4时，即计算A[4:6]断点k有两个：
- 对断点k = 4，计算A[4:4]A[5:6]有m[4][6] = m[4][4] + m[5][6] + p[3]*p[4]*p[6] = 5000 +5*10*25 = 6250

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	6250
5					0	5000
6						0

s[i][j]

1	2	3	4	5	6
	1	1			
		2	3		
			3	3	
				4	4
					5

MatrixChain的运行举例

- 当r = 3，i = 4时，即计算A[4:6]断点k有两个：
- 对断点k = 5，计算A[4:5]A[6:6]有m[4][6] = m[4][5] +m[6][6]+ p[3]*p[5]*p[6] = 1000+0+5*20*25 = 3500 ＜6250。m[4][6]改为3500，断点改为5。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	6250 3500
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	3		
3				3	3	
4					4	4 5
5						5
6						

MatrixChain的运行举例

- 类似的，当r=4、5、6时，可计算出相应的m[i][j]及其相应的断点s[i][j]，如下图中所示：

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

s[i][j]

1	2	3	4	5	6
	1	1	3	3	3
		2	3	3	3
			3	3	3
				4	5
					5

MatrixChain的运行举例

由m[1][6]知此矩阵连乘的最小数乘量为1 5125。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

4.构造最优解

算法matrixChain 记录了构造最优解所需的全部信息。

s[i][j]=k表明计算矩阵链A[i:j]的最佳方式在矩阵A_k和A_{k+1}之间断开，即最优的加括号方式为

(A[i:k])(A[k+1:j])。

因此，从s[1][n]记录的信息可知计算A[1:n]的最优加括号方式为 (A[1:$\boxed{s[1][n]}$])(A[$\boxed{s[1][n]+1}$:n])。

而A[1:s[1][n]]的最优加括号方式为：

(A[1:s[1][s[1][n]]])(A[s[1][s[1][n]]+1:s[1][n]])

同理可以确定A[s[1][n]+1:n]的最优加括号方式在s[s[1][n]+1][n]处断开。

照此递推下去，最终可以得到A[1:n]的最优完全加括号方式，即构造出问题的一个最优解。

3.2 DP示例—矩阵连乘问题

下面算法traceback按算法matrixChain计算出的s输出计算A[i:j]的最优计算次序:

```
void traceback(int s[][],int i,int j)
{
    if(i==j) return;
    traceback(s,i,s[i][j]);
    traceback(s,s[i][j]+1,j);
    System.out.println("Multiply A"+i+","+s[i][j]+
                      "and A"+(s[i][j]+1)+","+j);
}
```

要输出A[1:n]的最优计算次序只需调traceback(s,1,n)即可。

由s[1][6] = 3、s[1][3]=1、s[4][6]=5可知矩阵连乘的最优计算次序为：$(A_1(A_2A_3))((A_4A_5)A_6)$。

算法复杂性分析

- 算法matrixChain的主要计算取决于程序中对r、i和k的三重循环。循环体内的计算量为O(1)，1≤ r、i、k≤n，三重循环的总次数为$O(n^3)$。因此该算法时间复杂性的上界为$O(n^3)$，比直接递归算法要有效得多。
- 算法使用空间显然为$O(n^2)$。

3.2 DP示例—0/1背包问题

【问题描述】有n个重量分别为{w_1，w_2，…，w_n}的物品，它们的价值分别为{v_1，v_2，…，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。

3.2 DP示例—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 $v_1*x_1+v_2*x_2+\cdots+v_n*x_n$（其中xi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x1、x2、…、xn的值。假设按i=1，2，…，n的次序来确定xi的值，对应n次决策即n个阶段。

3.2 DP示例—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（$x_1, x_2, \cdots, x_n$）解向量X（$x_1, x_2, \cdots, x_n$）

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的次序来确定x_i的值，对应n次决策即n个阶段。

3.2 DP示例—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：若背包当前剩余容量为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

3.2 DP示例—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} …，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} …，x_n） *当前背包剩余容量为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} …，x_n） *当前背包剩余容量为r-wi*

【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（$1 \leqslant r \leqslant W$），

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。

$X\ (\underbrace{x_1,\ x_2,\ \ x_{i-1}}_{\text{已决策}},\ \boxed{x_i,}\ x_{i+1}\ \cdots,\ x_n)$

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

3.2 DP示例—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。

$X\ (\underbrace{x_1,\ x_2,\ x_{i-1}}_{\text{已决策}},\ \boxed{x_i},\ x_{i+1}\ \ldots,\ x_n)$

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

3.2 DP示例—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】

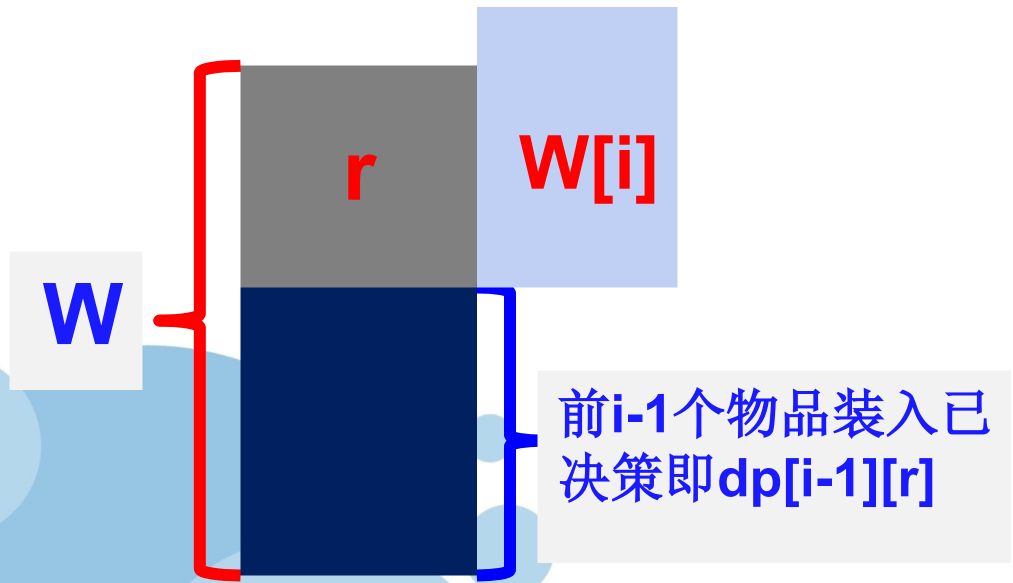
- dp[i][0]=0（背包不能装入任何物品，总价值为0） 边界条件dp[i][0]=0（1≤i≤n）－边界条件
- dp[0][r]=0（没有任何物品可装入，总价值为0）
  边界条件dp[0][r]=0（1≤r≤W）－边界条件

这样， dp[n][W]便是0/1背包问题的最优解。

3.2 DP示例—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

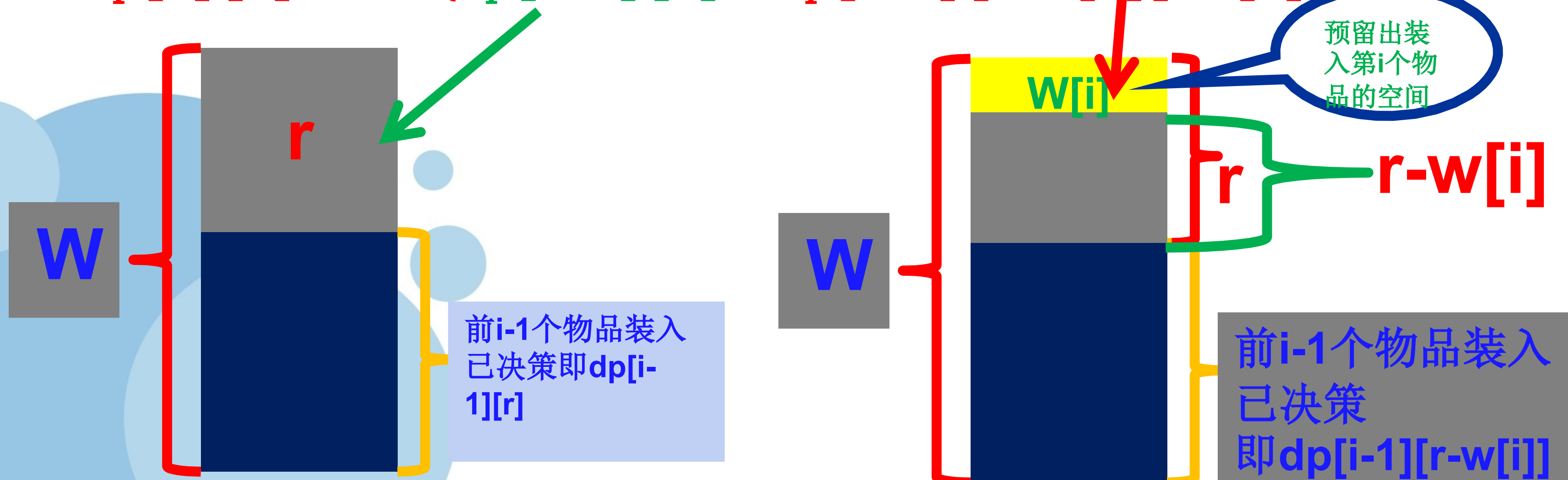
3.2 DP示例—0/1背包问题

【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]}



3.2 DP示例—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

3.2 DP示例—0/1背包问题

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	边界条件
1	0											
2	0											
3	0											
4	0											
5	0											

i

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

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	0	6	6	6	6	6	6	6	6	6
2	0										
3	0										
4	0										
5	0										

i

边界条件

边界条件

3.2 DP示例—0/1背包问题

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] (i)	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											

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

求解dp过程：

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;

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											

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:

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

i r

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

3.2 DP示例—0/1背包问题

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。

求出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	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	6	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

边界条件

3.2 DP示例—0/1背包问题

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

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	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

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

x=（1，1，0，0，1），
装入物品总重量为8，
总价值为15

Chapter 3 3.2 DP示例—0/1背包问题

```
//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
```

```
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]);
   }
}
```

3.2 DP示例—0/1背包问题

```
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--;
   }
}
```

3.2 DP示例—0/1背包问题

【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($n\times W$)。

Chapter 3

3.3 本章小结

动态规划(多阶段决策法，填表法）≠ 穷举 ≠ 分治

- 适合动态规划求解的问题：
  - ✓ 具有最优子结构：原问题的最优解包含子问题的最优解
  - ✓ 有重叠子问题：子问题之间不独立，一个子问题在下一阶段决策中可能被多次使用到。
  - ✓ 无后效性：某阶段状态一旦确定，就不受这个状态以后决策的影响

Chapter 3

3.3 本章小结

- 动态规划求解问题步骤：

① 分析问题的最优子结构，将大问题转换为小问题（状态转移）

② 递归的定义最优解（状态转移方程或递归方程，确定dp含义）。

③ 以自底向上或自顶向下（备忘录法）的记忆化方式计算出最优值。

④ 根据计算最优值时得到的信息，构造问题最优解。