算法设计与分析

主讲教师：邵蓥侠
联系方式：shaoyx@bupt.edu.cn
个人主页：https://shaoyx.github.io/

第3章 动态规划

学习要点:

理解动态规划算法的概念。

掌握动态规划算法的基本要素。

（1）最优子结构性质

（2）重叠子问题性质

掌握设计动态规划算法的步骤。

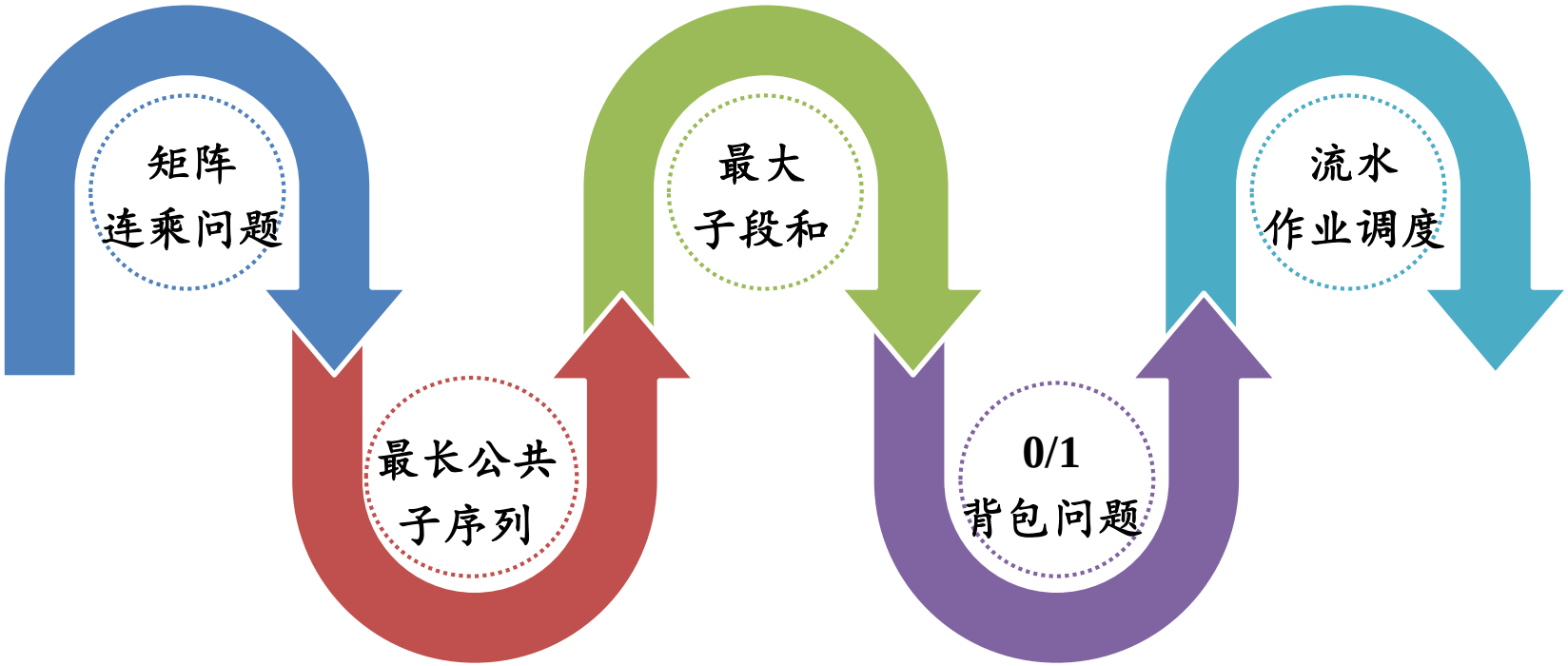
（1）找出最优解的性质，并刻划其结构特征。

（2）递归地定义最优值。

（3）以自底向上的方式计算出最优值。

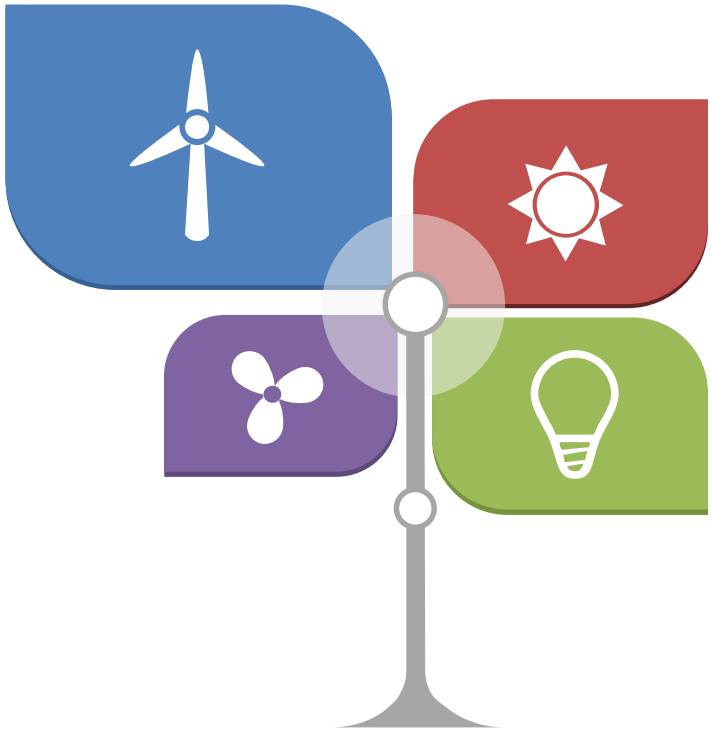
（4）根据计算最优值时得到的信息，构造最优解。

通过应用范例学习动态规划算法设计策略



动态规划算法的基本思想

- 聪明的遍历方法
- 与分治法对比学习



最优化原理

在实际生活中，有一类问题的活动过程可以分成若干个阶段，而且在任一阶段后的行为依赖于该阶段的状态，与该阶段之前的过程如何达到这种状态的方式无关。

- 这类问题的解决是多阶段的决策过程。
- 是处理分段过程最优化问题的一类非常有效的方法。
- 在50年代，贝尔曼（Richard Bellman)等人提出了最优化原理，并利用这个原理产生了动态规划算法从而解决了这类问题。

最优化原理

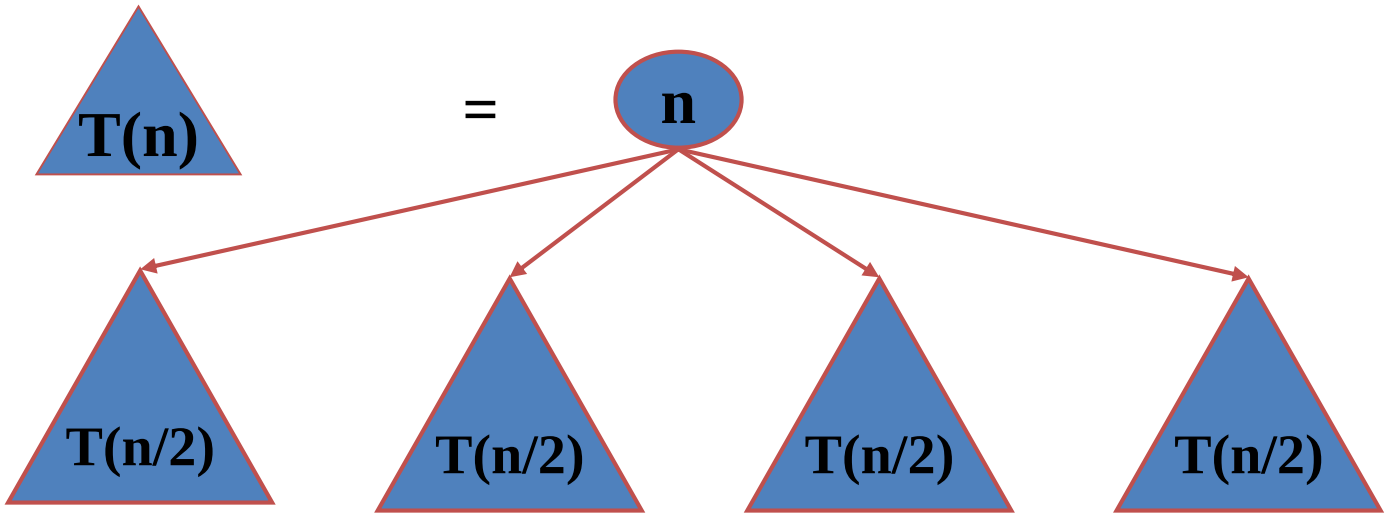
- 多阶段过程的最优决策序列应当具有性质：
  - 无论过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。

问题必须具备的性质

- 最优子结构性质
  - 原问题的最优解包含了其子问题的最优解，即原问题可以由子问题的最优解组合而成，这就使得问题可以拆分成若干个子问题。
- 子问题重叠性质
  - 每次产生的子问题并不总是新问题，有些子问题被反复计算多次。

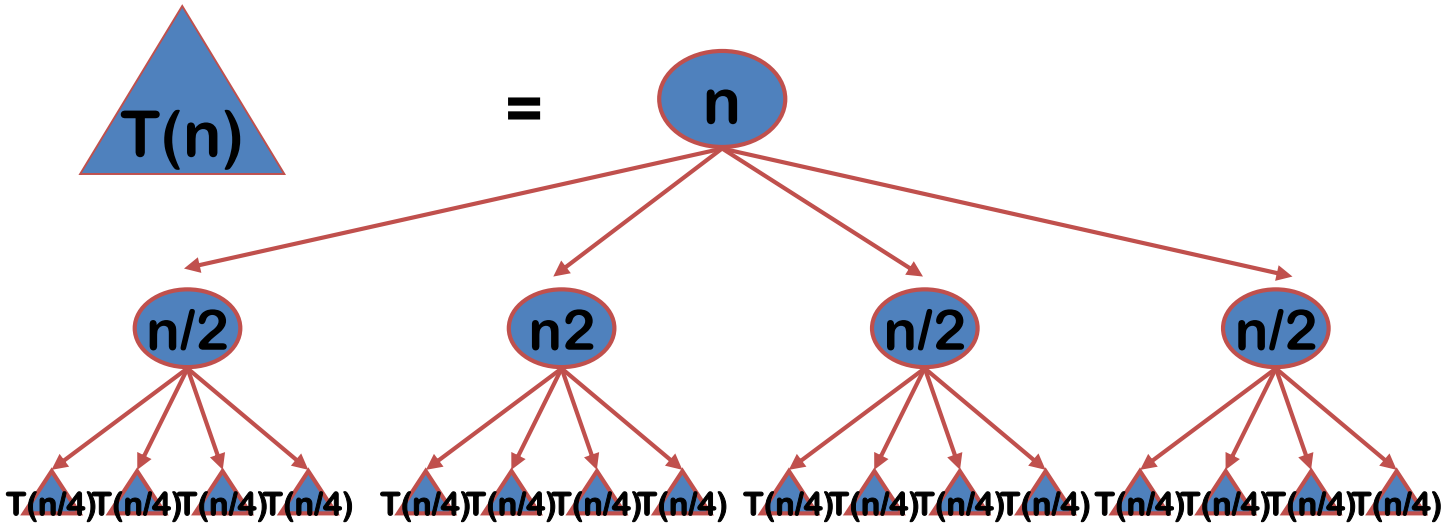
算法总体思想

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题。



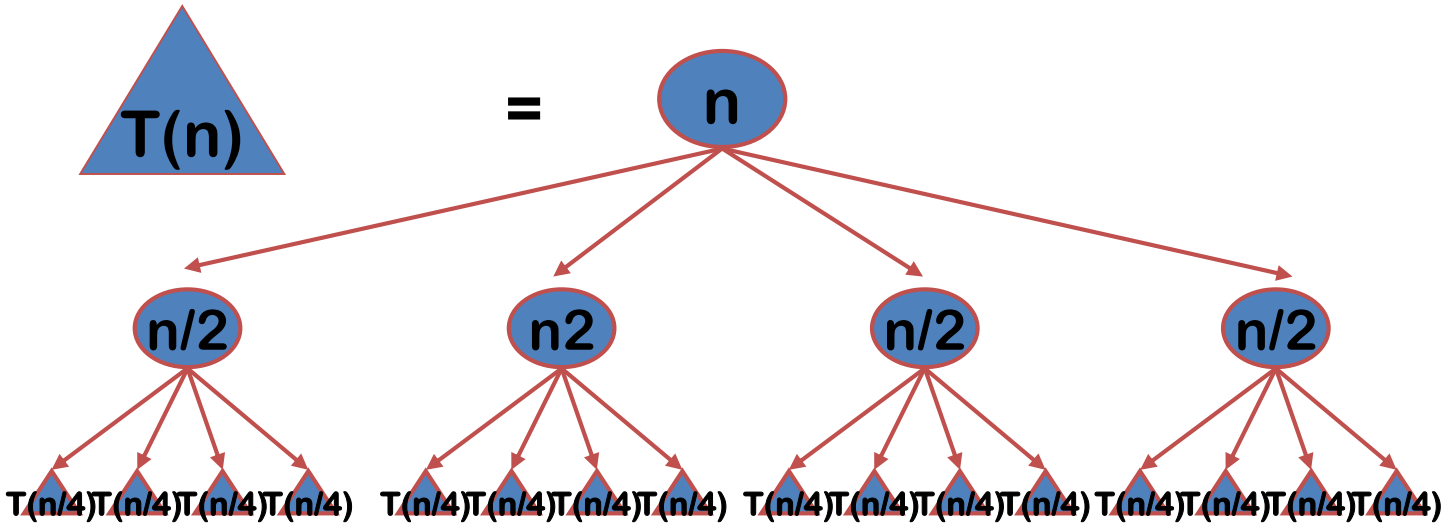
算法总体思想

- 但是经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次，时间复杂性呈指数增长。

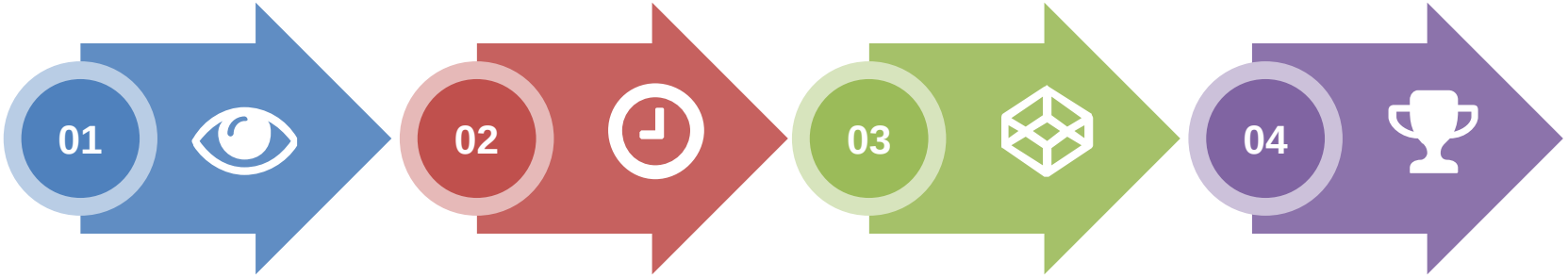


算法总体思想

- 如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。



动态规划基本步骤



找出最优解的性质，并刻划其结构特征。

递归地定义最优值。

以自底向上的方式计算出最优值。

根据计算最优值时得到的信息，构造最优解。

聪明的遍历

自底向上的求解

矩阵连乘问题

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

完全加括号的矩阵连乘积

- 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 $A=(BC)$

- 设有四个矩阵 $A1, A2, A3, A4$ ，它们的维数分别是：

$$A1=10\times100 \qquad A2=100\times5 \qquad A3=5\times50 \qquad A4=50\times30$$

- 总共有五种完全加括号的方式

矩阵连乘积 $A_1A_2A_3A_4$ 的不同的加括号方式及其对应计算量分别如下：

$A1 = 10 \times 100$　　$A2 = 100 \times 5$　　$A3 = 5 \times 50$　　$A4 = 50 \times 30$

(1) $(A_1(A_2(A_3A_4)))$ ： 5×50×30+100×5×30+10×100×30 = 52500

(2) $(A_1((A_2A_3)A_4))$ ： 100×5×50+100×50×30+10×100×30 = 205000

(3) $((A_1A_2)(A_3A_4))$ ： 10×100×5+5×50×30+10×5×30 = 14000

(4) $((A_1(A_2A_3))A_4)$ ： 100×5×50+10×100×50+10×50×30 = 90000

(5) $(((A_1A_2)A_3)A_4)$ ： 10×100×5+10×5×50+10×50×30 = 22500

如何求最优解?

- **最优解** == 以最少的数乘次数计算出矩阵连乘的乘积

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…，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。

◆直观思考一：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：

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

◆**直观思考二：计算量小的优先计算**

n个矩阵的连乘积共有n-1次乘法计算。首先在n-1次乘法计算中选择乘法计算量最小的两个矩阵优先计算；然后再在剩下的n-2次乘法计算中选择计算量最小的两个矩阵进行计算，依此往下。

该解决策略在某些情况下可以得到最优解，但是在很多情况下得不到最优解。如上例的第（5）种完全加括号方式就是采用该策略，但它得到的显然不是最优解。

◆直观思考三：矩阵维数大的优先计算

在n个矩阵的n+1个维数序列 $p_0, p_1, p_2, \cdots, p_n$ 中选择维数的最大值（设为 p_i），并把相邻两个含有维数 p_i 的矩阵优先进行计算；然后在剩下的n个维数序列中再次选择维数的最大值，进行相应矩阵的乘法运算；依此往下。

该解决策略与上一种策略相似，在有些情况下可以得到最优解，但是在有些情况下得不到最优解。如上例的第（3）种完全加括号方式就是采用该策略，显然它得到的是最优解。

但当4个矩阵 A_1, A_2, A_3, A_4 的维数改为50×10，10×40，40×30和30×5时，可以验证，采用该策略得到的就不是最优解。

以上两种策略实质都是基于某种贪心选择的贪心算法，这种算法不一定能得到问题的全局最优解。在下一章我们将详细讨论此种策略。

◆思考四：分治法

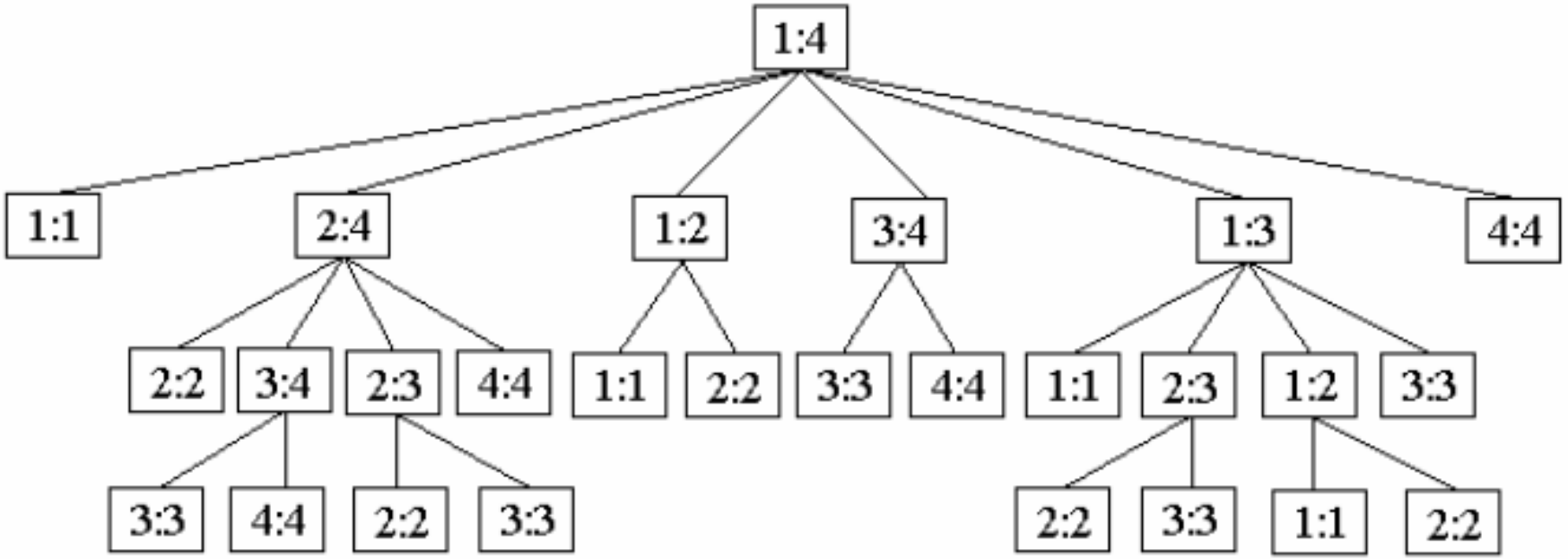
设n个矩阵连乘积 $A_1A_2\cdots A_n$ 的计算次序在矩阵 A_k 和 A_{k+1} 之间将矩阵链断开，1≤k<n， 则其相应的完全加括号方式为 $((A_1\cdots A_k)(A_{k+1}\cdots A_n))$ 。

由于完全加括号方式是一个递归定义，因此可以递归地计算A[1:k]和A[k+1:n]，然后将计算结果相乘得到A[1:n]。据此可设计如下递归算法**recurmatrixChain**：

```
int recurMatrixChain(int i, int j)
{
    if (i == j) return 0;
    int u = recurMatrixChain(i, i) + recurMatrixChain(i+1, j) + p[i-1]*p[i]*p[j];
    s[i][j] = i;
    for (int k = i+1; k < j; k++) {
      int t = recurMatrixChain(i, k)+ recurMatrixChain(k+1, j) + p[i-1]*p[k]*p[j];
      if (t < u) {
        u = t;
        s[i][j] = k;
      }
    }
    return u;
}
```

◆思考 为何分治法的效率如此低下？

下图是用算法**recurmatrixChain(1，4)**计算A[1:4]的递归树。



从上图可以看出，许多子问题被重复计算，这是分治法效率低下的根本原因。

该递归算法的计算时间T(n)可递归定义如下：

$$T(n)\geq\begin{cases}O(1) & n=1\\ 1+\sum_{k=1}^{n-1}(T(k)+T(n-k)+1) & n>1\end{cases}$$

当n>1时，$T(n)\geq 1+(n-1)+\sum_{k=1}^{n-1}T(k)+\sum_{k=1}^{n-1}T(n-k)=n+2\sum_{k=1}^{n-1}T(k)$

以上递归式，可用数学归纳法证明 $T(n)\geq 2^{n-1}=\Omega(2^n)$

即该算法的计算时间T(n)有指数下界。这样对该问题而言，分治法是一个可行方法，但不是一个有效算法。

矩阵连乘问题

思考五：该问题可用分治思想解决，并存在大量计算冗余，能否用动态规划求解？

将矩阵连乘积 $A_iA_{i+1}...A_j$ 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A_k和A_{k+1}之间将矩阵链断开，i⩽k<j，则其相应完全加括号方式为 $(A_iA_{i+1}...A_k)(A_{k+1}A_{k+2}...A_j)$

计算量：A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k]和A[k+1:j]相乘的计算量

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。
- 如何证明？——反证法

2. 建立递归关系

- 设计算A[i:j]，1≤i≤j≤n，**所需要的最少数乘次数m[i,j]**，则原问题的最优值为m[1,n]
- 当i=j时，A[i:j]=Ai，因此，m[i,i]=0，i=1,2,…,n
- 当i<j时，$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j$

这里 A_i 的维数为 p_{i-1} ´ p_i

- **由最优子结构性质，可以递归地定义m[i,j]为：**

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

k **的位置只有** $j-i$ **种可能**

使用自底向上的方式解决重复计算问题

- 先解子问题，
- 合并子问题，
- 得到全解

3. 计算最优值

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

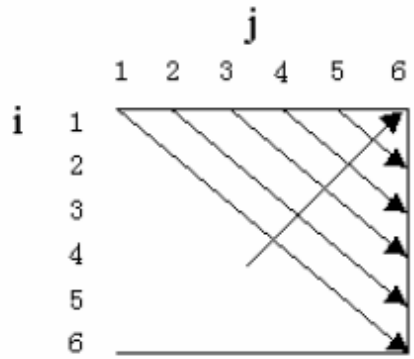
1. 计算$A_1, A_2, A_3, A_4, A_5, A_6$
2. 计算$(A_1A_2), (A_2A_3), (A_3A_4), (A_4A_5), (A_5A_6)$
3. 计算$(A_1A_2A_3)$, …, $(A_4A_5A_6)$
4. 计算$A[1:4], A[2:5], A[3:6]$
5. 计算$A[1:5], A[2:6]$
6. 计算$A[1:6]$

1. 计算$A_1, A_2, A_3, A_4, A_5, A_6$
2. 计算$(A_1A_2), (A_2A_3), (A_3A_4), (A_4A_5), (A_5A_6)$
3. 计算$(A_1A_2A_3)$, ..., $(A_4A_5A_6)$
4. 计算$A[1:4], A[2:5], A[3:6]$
5. 计算$A[1:5], A[2:6]$
6. 计算$A[1:6]$

	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

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

S[i][j]的含义？



(a) 计算次序

j

i	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

(b) m[i][j]

j

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

(c) s[i][j]

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375 \end{cases}$$

用动态规划法求最优解

```
void MatrixChain(int *p，int n，int **m，int **s)
{
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
      for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
        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;}
        }
      }
}
```

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

S[i][j]的含义？

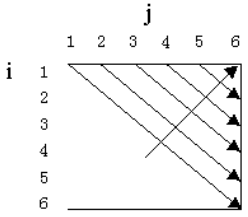
用动态规划法求最优解

```
void MatrixChain(int *p，int n，int **m，int **s)
{
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
      for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
        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;}
        }
      }
}
```

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

算法复杂度分析：

算法matrixChain的主要计算量取决于算法中对r，i和k的3重循环。循环体内的计算量为O(1)，而3重循环的总次数为$O(n^3)$。因此，算法的计算时间上界为$O(n^3)$。算法所占用的空间显然为$O(n^2)$。



(a) 计算次序

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

(b) m[i][j]

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

计算最优值

- 对于1≤i≤j≤n不同的有序对(i,j)对应于不同的子问题。因此，不同子问题的个数最多只有

$$\binom{n}{2}+n=\Theta(n^2)$$

- 由此可见，在递归计算时，许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。
- 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

4. 构造最优值

- 事实上，**matrixChain**已记录了构造最优解所需要的全部信息。s[i][j]中的数k表明计算矩阵链A[i:j]的最佳方式应在矩阵Ak和Ak+1之间断开，即最优的加括号方式应为(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:s[1][n]])(A[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][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，最终可以确定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。

```
void traceback (int i, int j, int * * s)
{
    if (i == j) return;
     traceback(s, i, s[i][j]);
     traceback(s, s[i][j] + 1, j);
     cout<<"Multiply A “<< i << ", " << s[i][j];
     cout<<" and A " << (s[i][j] + 1) << ", " << j << endl;
}
```

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

(b) m[i][j]

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

要输出A[1:n]的最优计算次序只要调用**traceback(1，n，s)**即可。对于上面所举的例子，通过调用**traceback(1，6，s)**，可输出最优计算次序

$((A_1(A_2A_3))((A_4A_5)A_6))$

动态规划算法的基本要素

一、最优子结构

- 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**。
- 在分析问题的最优子结构性质时，**所用的方法具有普遍性：首先假设**由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。
- 利用问题的最优子结构性质，以**自底向上**的方式递归地从子问题的最优解逐步构造出整个问题的最优解。**最优子结构是问题能用动态规划算法求解的前提**。

同一个问题可以有多种方式刻划它的最优子结构，有些表示方法的求解速度更快（空间占用小，问题的维度低）

比较下面两个算法段

```
void MatrixChain(int *p, int n, int **m, int **s)
{
        for (int i = 1; i <= n; i++)
        {
            m[i][i] = 0;
        }
        for (int r = 2; r <= n; r++)
        {
            for (int i = 1; i <= n - r+1; i++)
            {
                int j=i+r-1;
                m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
                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;
                    }
                }
            }
        }
}
```

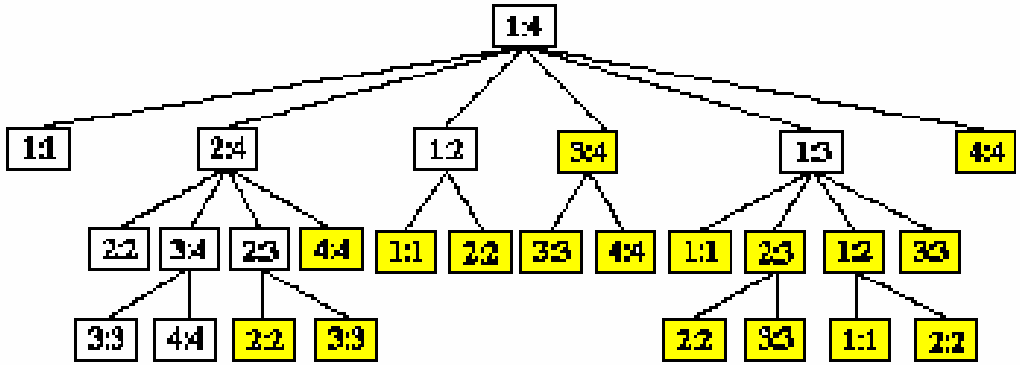
比较下面两个算法段

```
void RecurMatrixChain(int i, int j)
{
        if (i == j) return 0;
        int u = RecurMatrixChain(i, i) + RecurMatrixChain(i+1, j) + p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k<j; k++)
        {
            int t = RecurMatrixChain(i, k) + RecurMatrixChain(k+1, j) + p[i-1]*p[k]*p[j];
            if (t < u)
            {
                u=t; s[i][j] = k;
            }
            return u;
        }
}
```

动态规划算法的基本要素

二、重叠子问题-自底向上的解决方法

- 递归算法求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。这种性质称为子问题的重叠性质。
- 动态规划算法，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。
- 通常不同的子问题个数随问题的大小呈多项式增长。因此用动态规划算法只需要多项式时间，从而获得较高的解题效率。



动态规划算法的基本要素

三、备忘录方法

•备忘录方法的控制结构与直接递归方法的控制结构相同，区别在于备忘录方法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。

```
int LookupChain(int i，int j)
{
    if (m[i][j] > 0) return m[i][j];
    if (i == j) return 0;
    int u = LookupChain(i，i) + LookupChain(i+1，j) + p[i-1]*p[i]*p[j];
    s[i][j] = i;
    for (int k = i+1; k < j; k++) {
      int t = LookupChain(i，k) + LookupChain(k+1，j) + p[i-1]*p[k]*p[j];
      if (t < u) { u = t; s[i][j] = k;}
      }
    m[i][j] = u;
    return u;
}
```

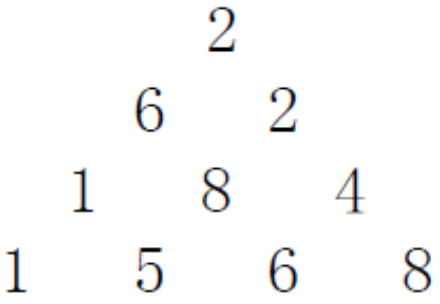
动态规划法：自底向上
备忘录方法：自顶向下

练习一下动态规划的思维方式

- 重点：
- 学习动态规划的状态划分和表示方法

实例一、数字三角形问题

给定一个具有N层的数字三角形，从顶至底有多条路径，每一步可沿左斜线向下或沿右斜线向下，路径所经过的数字之和为路径得分，请求出最小路径得分。

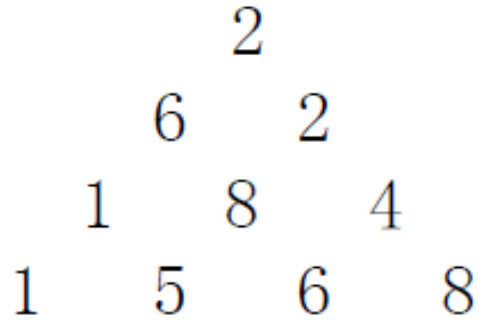


数字三角形

实例一、数字三角形问题

2．解题思路

这道题可以用动态规划成功地解决，但是，如果对问题的最优结构刻画得不恰当(即状态表示不合适)，则无法使用动态规划。



数字三角形

状态表示法一：

用一元组D(X)描述问题，D(X)表示从顶层到达第X层的最小路径得分。因此，此问题就是求出D(N)(若需要，还应求出最优路径)。这是一种很自然的想法和表示方法。遗憾的是，这种描述方式并不能满足最优子结构性质。因为D(X)的最优解(即最优路径)可能不包含子问题例如D(X-1)的最优解。如下图所示：

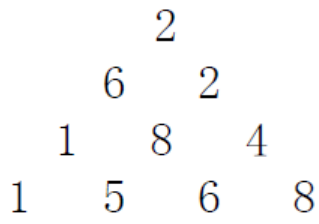
2

6 2

1 8 4

1 5 6 8

数字三角形



数字三角形

显然，D(4)＝2+6+1+1＝10，其最优解(路径)为2-6-1-1

而D(3)＝2+2+4＝8，最优解(路径)为2-2-4。

故D(4)的最优解不包含子问题D(3)的最优解。

由于不满足最优子结构性质，
因而无法建立子问题最优值之间的递归关系，
也即无法使用动态规划。

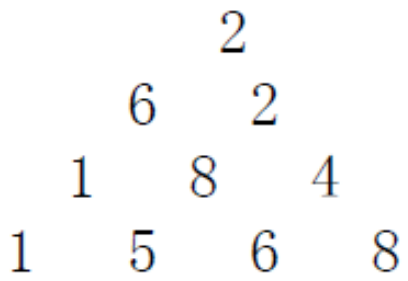
状态表示法二：

用二元组D(X,y)描述问题，D(X,y)表示从顶层到达第X层第y个位置的最小路径得分。

最优子结构性质：

容易看出，D(X,y)的最优路径 Path(X,y)
一定包含 子问题D(X-1,y)或D(X-1,y-1)的最优路径。

否则，取D(X-1,y)和D(X-1,y-1)的最优路径中得分小的那条路径加上第X层第y个位置构成的路径得分必然小于Path(X,y)的得分，这与Path(X,y)的最优性是矛盾的。



数字三角形

- 如上图所示，D(4,2)的最优路径为2-6-1-5，它包含D(3,1)最优路径2-6-1。因此，用二元组D(X,y)描述的计算D(X,y)的问题具有最优子结构性质。

- 递归关系：

$$\begin{cases} D(X,y)=\min\{D(X-1,y),\ D(X-1,y-1\}+a(X,y) \\ D(1,1)=a(1,1) \end{cases}$$

  - 其中，a(X,y)为第X层第y个位置的数值。

- 原问题的最小路径得分可以通过比较D(N,i)获得，其中i＝1，2，...，N。

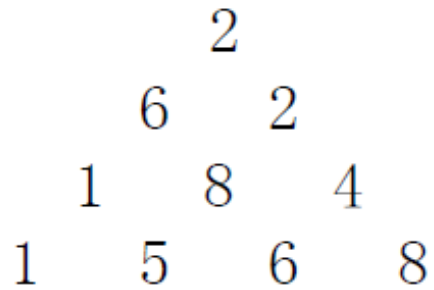
- 在上述递归关系中，求D(X,y)的时候，先计算D(X-1,y)和D(X-1,y-1)，下一步求D(X，y+1)时需要D(X-1，y+1)和D(X-1，y)，但其中D(X-1，y)在前面已经计算过了。于是，子问题重叠性质成立。

- 因此，采用状态表示法二描述的问题具备了用动态规划求解的基本要素，可以用动态规划进行求解。

状态表示法三：

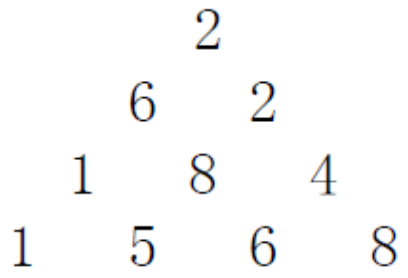
采用状态表示法二的方法是从顶层开始，逐步向下至底层来求出原问题的解。事实上，还可以从相反的方向考虑。仍用二元组D(X,y)描述问题，D(X,y)表示从第X层第y个位置到达底层的最小路径得分。原问题的最小路径得分即为D(1,1)。

最优子结构性质：显然，D(X,y)的最优路径Path(X,y)一定包含子问题D(X+1,y)或D(X+1,y+1)的最优路径，否则，取D(X+1,y)和D(X+1,y+1)的最优路径中得分小的那条路径加上第X层第y个位置构成的路径得分必然小于Path(X,y)的得分，这与Path(X,y)的最优性矛盾。



数字三角形

- 如上图所示，D(1,1)的最优路径为2-6-1-1，它包含D(2,1)的最优路径6-1-1。因此，这种状态表示描述的计算D(X，y)的问题同样具有最优子结构性质。



数字三角形

- 递归关系：

$$\begin{cases} D(X,y) = \min\{D(X+1,y), D(X+1,y+1)\} + a(X,y) \\ D(N,k) = a(N,k),\ k=1,\ \ldots,\ N \end{cases}$$

其中，a(X，y)为第X层第y个位置的数值。

- D(X,y)表示从第X层第y个位置到达底层的最小路径得分。原问题的最小路径得分即为D(1,1)。

问题：给定两个序列

X={ x1, x2, ……,xm}

Y={ y1, y2, ……,yn}

找出X和Y的一个最长公共子序列。

最长公共子序列

公共子序列

例如，

序列 X={A，B，C，B，D，A，B}

序列Z={B，C，D，B}

序列Z是序列X的子序列

相应的递增下标序列为{2，3，5，7}。

若给定序列$X=\{x_1,x_2,\ldots,x_m\}$，

则另一序列$Z=\{z_1,z_2,\ldots,z_k\}$，是X的子序列

是指 存在一个严格递增下标序列$\{i_1,i_2,\ldots,i_k\}$

使得对于所有$j=1,2,\ldots,k$有：$z_j=x_{i_j}$。

最长公共子序列

- X = { A, B, C, B, D, A, B }
- Y = { B, D, C, A, B, A }
- 其中，序列 { B, C, A }是长度=3的公共子序列；{ B, C, B, A }是长度=4的最长公共子序列

最长公共子序列

- 给定2个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的**公共子序列**。

- 给定2个序列$X=\{x_1, x_2, \ldots, x_m\}$和$Y=\{y_1, y_2, \ldots, y_n\}$，找出X和Y的最长公共子序列。

又见穷举

- 对X的所有子序列，检查它是否也是Y的子序列，从而确定它是否为X和Y的公共子序列，并且在检查过程中记录最长的公共子序列。

1.最长公共子序列的结构

设序列$X=\{x_1,x_2,\ldots,x_m\}$和$Y=\{y_1,y_2,\ldots,y_n\}$的最长公共子序列为$Z=\{z_1,z_2,\ldots,z_k\}$，则

(1)若$x_m=y_n$，则$z_k=x_m=y_n$，且Z_{k-1}是x_{m-1}和y_{n-1}的最长公共子序列。

(2)若$x_m\neq y_n$且$z_k\neq x_m$，则Z是x_{m-1}和Y的最长公共子序列。

(3)若$x_m\neq y_n$且$z_k\neq y_n$，则Z是X和y_{n-1}的最长公共子序列。

由此可见，2个序列的最长公共子序列包含了这2个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有**最优子结构性质**。

2.子问题的递归结构

由最长公共子序列问题的最优子结构性质建立子问题最优值的递归关系。用c[i][j]记录序列和的最长公共子序列的长度。其中，$X_i=\{x_1,x_2,\ldots,x_i\}$； $Y_j=\{y_1,y_2,\ldots,y_j\}$。

当i=0或j=0时， 空序列是X_i和Y_j的最长公共子序列。故此时c[i][j]=0。

其它情况下， 由最优子结构性质可建立递归关系如下：

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

3. 计算最优值

X(i)={x1,x2,…,xi}, 1≤i ≤ m
Y(j)={y1,y2,…,yj}, 1≤j≤ n；

对X(m)和Y(n)，总共有**θ(mn)**个不同的子问题，用动态规划算法自底向上地计算最优值，以提高算法效率。

void **LCSLength**(int m，int n，char *x，char *y，int **c，int **b)

数组x*和y*：存储2个输入序列X(m)、Y(n)
输出数组c：c[i][j]存储序列X[i]、Y[j]的最长公共子序列的长度
输出数组b： b[i][j] 记录c[i][j]的值是由哪个子问题得到的(3种情况之一)，构造最长公共子序列时用到

3. 计算最优值

当在b[i][j]遇到：

1时，最长公共子序列由Xi-1,Yj-1的最长公共子序列+1得到；

2时，最长公共子序列=X[i-1],Y[j]的最长公共子序列；

3时，最长公共子序列=X[i],Y[j-1]的最长公共子序列；

```
void LCSLength(int m，int n，char *x，char *y，int **c，int **b)
{
      int i，j;
      for (i = 1; i <= m; i++) c[i][0] = 0;   /*初始化，Y[j]为空时
      for (i = 1; i <= n; i++) c[0][i] = 0;   /*初始化，X[i]为空时

      for (i = 1; i <= m; i++)          /*两重循环，自下而上，计算子问题{X(i), Y(j)}
        for (j = 1; j <= n; j++) {
          if (x[i]==y[j]) {                    /*情况1
              c[i][j]=c[i-1][j-1]+1;
              b[i][j]=1; }
          else if (c[i-1][j]>=c[i][j-1]) {      /*情况2
              c[i][j]=c[i-1][j]; b[i][j]=2; }
          else {
              c[i][j]=c[i][j-1];               /*情况3
              b[i][j]=3; }
          }
}
```

4. 构造最长公共子序列

```
void LCS(int i，int j，char *x，int **b)
{
    if (i ==0 || j==0) return;
    if (b[i][j]== 1){ LCS(i-1，j-1，x，b); cout<<x[i]; }
    else if (b[i][j]== 2) LCS(i-1，j，x，b);
    else LCS(i，j-1，x，b);
}
```

算法的改进

- 在算法LCSLength和LCS中，可进一步将数组b省去。事实上，数组元素c[i][j]的值仅由c[i-1][j-1]，c[i-1][j]和c[i][j-1]这3个数组元素的值所确定。对于给定的数组元素c[i][j]，可以不借助于数组b而仅借助于c本身在O(1)时间内确定c[i][j]的值是由c[i-1][j-1]，c[i-1][j]和c[i][j-1]中哪一个值所确定的。
- 如果只需要计算最长公共子序列的长度，则算法的空间需求可大大减少。事实上，在计算c[i][j]时，只用到数组c的第i行和第i-1行。因此，用2行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至O(min(m,n))。

扩展：利用最长公共子序列求解最长上升/递减子序列问题

最长上升/递减子序列问题描述：

- 给定由n个整数a1，a2，…，an构成的序列，在这个序列中随意删除一些元素后可得到一个子序列ai，aj，ak，…，am，其中1≤i≤m≤n，则称序列ai，aj，ak，…，am为原序列的一个上升/递减子序列，长度最长的上升/递减子序列即为原序列的最长上升/递减子序列。（子序列中的相邻元素不能相等）
- 例如：序列{1，6，2，3，7，5}，可以选出上升子序列{1，2，3，5}，也可以选出{1，6，7}.

解题思路

- 最长递减子序列问题可以用动态规划方法直接求解，思路是先求出以原序列中的每一个元素为尾元素的最长递减子序列的长度，然后从中选择长度最大的即为整个序列的最长递减子序列。

- 该问题也可以利用最长公共子序列问题来求解，该方法与直接动态规划方法求解时间复杂度相当。

利用最长公共子序列求解最长递减子序列

- 由于最长递减子序列问题是求解一个给定序列的最长递减子序列，因此要用最长公共子序列问题来求解最长递减子序列问题，
- 首先要构造出第二个序列，即让原序列和哪个序列求最长公共子序列才能得到原序列的最长递减子序列
- 将原序列降序排序，得到降序序列
- 再让原序列和降序序列求一下最长公共子序列，则这个最长公共子序列即为原序列的一个最长递减子序列。

利用最长公共子序列求解最长递减子序列

```
void LIS（int a[]，int x[]，int m）
//该算法求解x[1：m]这个长度为m的整型数组的最长递减子序列长度
{
for（int i=1;i<=m;i++） //输入数组x
      scanf（“%d”，&x[i]）;
      for（int i=1;i<=m;i++） //数组a为数组x的副本
      a[i]=x[i];
      sort（a，1，m）; //对数组a进行递减归并排序
       LCSLength（x，a，m，m）; //求数组x与数组a的最长公共子序列长度
}
```

扩展：利用最长公共子序列求解回文词的构造问题

- 回文词的构造问题描述
- 对于任意给定的一个字符串（由大写字母、小写字母和数字字符构成），都可以通过插入若干个字符的方法将其变成一个回文串。
- 例如：给定一个字符串Ab3bd，在插入两个字符后就可以变成一个回文串，如可以在A的左边添上一个d，再在最后一个d之前添上一个A，这样改造后的字符串为dAb3bAd，它是一个回文串。
- 那么对于任意给定的一个字符串，最少要插入几个字符才能将其变成回文串呢？

解题思路

- 首先将给定的字符串X翻转得到它的逆串Y，然后求X与Y的最长公共子序列，那么X的字符个数n减去最长公共子序列的长度即为将X变成回文串时最少需要添加的字符个数。

利用最长公共子序列求解回文词的构造问题

```
int huiwen（int n，char x[]，char y[]） //该算法求解x[1：n]
        这个长度为n的字符串变为回文串时最少要插入的字符个数
{
        for（int i=1;i<=n;i++）    //向字符数组x中输入一个字符串
              scanf（”%c”，&x[i]）;
        for（int i=1;i<=n;i++）    //求字符串x的逆串，保存在字符数组y中
              y[i]=x[n-i+1];
         LCSLength（x，y，n，n）;  //求原串x与逆串y的最长公共子序列长度
        return n-c[n][n]; //返回将串x变为回文串最少要插入的字符个数，c为全局数组
}
```

最大子段和

问题描述：最大子段和求解

给定由n个整数(可能为负整数)组成的序列$a_1,a_2,\ldots,a_n$,求该序列形如 $\sum_{k=i}^{j} a_k$ 的子段和的最大值。当所有整数均为负整数时定义其最大子段和为 0 。依此定义，所求的最优值为:

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

例如当$(a_1, a_2, a_3, a_4, a_5, a_6)=(-2, 11, -4, 13, -5, -2)$时，最大子段和为 $\sum_{k=2}^{4} a_k = 20$ 。

此题的意义

- 程序员面试题精选100题解析之一
- 这题曾被很多公司当做面试题，微软亚洲研究院曾在2006年的笔试题中也出过这道题。
- 这是一道典型的动态规划(Dynamic Programming,后面简述为DP)算法题目，最优解法的算法复杂度为O(N)。
- 那么如果我们没有接触过DP，是不是就无法想出最优的解法？答案当然是否定的，下面我就假设你从来没接触过DP，来分析这道题目。

简单解法

- 枚举法，$O(n^3)$，算法见第一章PPT.
- 枚举法+前缀求和，$O(n^2)$，算法见第一章PPT.
- 注：前缀求和，即sum[i]=sum[i-1]+a[i]

解法3：分治法的解决

如果将所给的序列a[1:n]分为长度相等的2段a[1:n/2]和a[n/2+1:n]，分别求出这2段的最大子段和，则a[1:n]的最大子段和有3种情况：

(1) a[1:n]的最大子段和与a[1:n/2]最大子段和相同；

(2) a[1:n]的最大子段和与a[n/2+1:n]最大子段和相同；

(3) a[1:n]的最大子段和为 $\sum_{k=i}^{j} a_k$ ，且$1 \leqslant i \leqslant n/2$, $n/2+1 \leqslant j \leqslant n$。

请想想，此时的情形与分治法的那个例子相似？

Center = (left + right)/2

$$S1 = \max_{1 \le i \le n/2} \sum_{k=i}^{n/2} a[k]$$

```
int s1 = 0;
int lefts = 0;
for (int i = center; i >= left; i--)
{
        lefts+= a[i];
        if(lefts >s1) s1 = lefts;
}
```

$$S2 = \max_{\frac{n}{2}+1 \le i \le n} \sum_{k=\frac{n}{2}+1}^{i} a[k]$$

```
int s2 = 0;
int rights = 0;
for (int i = center +1; i <= right; i++)
{
        rights += a[i];
        if(rights > s2) s2 = rights;
}
```

```
sum = s1 + s2;
```

分治法求解代码

```
int MaxSubSum(int *a, int left, int right)
{
    int sum = 0;
    if (left = right) sum = a[left] > 0 ? a[left]: 0;
    else{
        int center = (left + right )/2;
        int leftsum = MaxSubSum(a, left, center);
        int rightsum = MaxSubSum(a, center+1, right);

        int s1 = 0;
        int lefts = 0;
        for (int i = center; i >= left; i--)
        {
                lefts+= a[i];
                if(lefts >s1) s1 = lefts;
        }

        int s2 = 0;
        int rights = 0;
        for (int i = center +1; i <= right; i++)
        {
                rights += a[i];
                if(rights > s2) s2 = rights;
        }
        sum = s1 + s2;
        if( sum < leftssum) sum = leftsum;
        if( sum < rightsum) sum = rightsum;
    }
    return sum;
}
```

复杂度分析:

$T(n)=O(n\log n)$

从例子中观察规律

从分治法的合并过程中，找出一般规律。

解法4：妙不可言的方法

- 上述解法都是最基本的，下面这种解法的复杂度为O(N)。
- 我们可以想到，当枚举起始点i与结束点j的时候，这里的复杂度就是O(N^2)。那我们可不可以只枚举起始点i，或者只枚举结束点j而最终得到最大子数组的和呢？

- 假如现在我们只枚举数组的结束点j。

假如现在我们只枚举数组的结束点j

1,-5, 3, 4,-2,-3, 10

以下标 0 为结束的数组：1

以下标 1 为结束的数组：1,-5

以下标 2 为结束的数组：1,-5,3

以下标 3 为结束的数组：1,-5,3,4

以下标 4 为结束的数组：1,-5,3,4,-2

以下标 5 为结束的数组：1,-5,3,4,-2,-3

以下标 6 为结束的数组：1,-5,3,4,-2,-3,10

观察枚举的过程：

- 数组b[j]：表示以下标j结束的最大子数组的和的值。
- 以下标0为结束的数组的最大子数组和就是该独立的元素1。即b[0]=1。
- 思考？以下标1为结束的数组：1,-5的最大子数组和是什么？（注意：这里指的最大子数组必须以下标1(或者说j)结束）

以下标 0 为结束的数组：1

以下标 1 为结束的数组：1,-5

以下标 2 为结束的数组：1,-5,3

以下标 3 为结束的数组：1,-5,3,4

以下标 4 为结束的数组：1,-5,3,4,-2

以下标 5 为结束的数组：1,-5,3,4,-2,-3

以下标 6 为结束的数组：1,-5,3,4,-2,-3,10

观察以上枚举的过程：

- 思考？
  - 以下标1为结束的数组：1,-5的最大子数组和是什么？（注意：这里指的最大子数组必须以下标1(或者说j)结束）
- 答案很明确：
  - 如果以下标0结束的最大子数组的和为正数，那么b[1]就是b[0]+a[1]；如果以下标0结束的最大子数组的和为负数，那么b[1]就是a[1]。所以b[1]=b[0]+a[1]=-4。
  - 以下标2为结束的数组：
  - 1,-5,3的最大子数组和是什么？

以下标 0 为结束的数组：1

以下标 1 为结束的数组：1,-5

以下标 2 为结束的数组：1,-5,3

以下标 3 为结束的数组：1,-5,3,4

以下标 4 为结束的数组：1,-5,3,4,-2

以下标 5 为结束的数组：1,-5,3,4,-2,-3

以下标 6 为结束的数组：1,-5,3,4,-2,-3,10

观察以上枚举的过程：

- 思考？
  - 以下标2为结束的数组：1,-5,3的最大子数组和是什么？
- 答案很明确：
  - 如果b[1]>0，则b[2]=b[1]+a[2]。
    否则b[2]=a[2]。
    所以b[2]=3。

以下标 0 为结束的数组：1
以下标 1 为结束的数组：1,-5
以下标 2 为结束的数组：1,-5,3
以下标 3 为结束的数组：1,-5,3,4
以下标 4 为结束的数组：1,-5,3,4,-2
以下标 5 为结束的数组：1,-5,3,4,-2,-3
以下标 6 为结束的数组：1,-5,3,4,-2,-3,10

动态规划的求解?

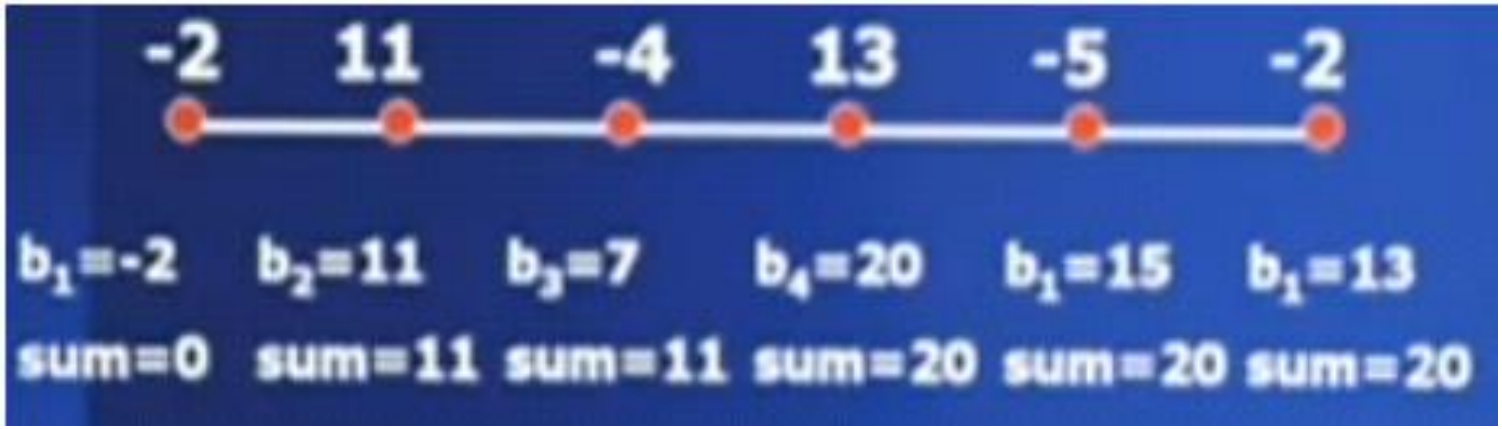
记 $b[j]=\max\limits_{1\le i\le j}\{\sum\limits_{k=i}^{j}a[k]\}$，1≤ j ≤n，则所求的最大子段和为

$$\max_{1\le i\le j\le n}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}\max_{1\le i\le j}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}b[j]$$

当b[j-1]>0时b[j]=b[j-1]+a[j]，否则b[j]=a[j]。由此可得计算b[j]的动态规划递归式

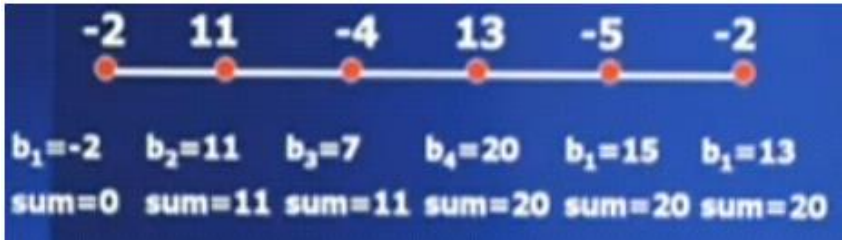
b[j]=max{b[j-1]+a[j]，a[j]}，　1≤ j ≤n

晕？ 看个例子吧！


-2 11 -4 13 -5 -2
b1=-2 b2=11 b3=7 b4=20 b1=15 b1=13
sum=0 sum=11 sum=11 sum=20 sum=20 sum=20

算法描述和分析

```
int maxSum(int n, int * a)
{
      int sum=0, b=0;
      for (int i=1;i<=n;i++) {
        if (b>0) b+=a[i];
        else b=a[i];
        if (b>sum) sum=b;
      }
      return sum;
}
```



算法显然需要O(n)计算时间

妙不可言!

- 令人赞叹的算法复杂性!
  - 从代码中也可以看到，只有一层循环i，用来枚举结束的下标。
- 上面的代码中，也对空间复杂度做了小小的优化，b[]这个数组没有必要开，原因是b[j]只和b[j-1]有关系，所以用一个变量b即可。从代码中也可以看到，只有一层循环i，用来枚举结束的下标。

还有新思路吗？

换个例子试试?

1,-5, 3,4,-2,-3,10

那么最大子数组和即为 12．由 3,4,-2,-3,10 组成.

我们考虑以下标 i 为结尾的数组：

以下标 0 为结束的数组： 1

以下标 1 为结束的数组： 1,-5

以下标 2 为结束的数组： 1,-5,3

以下标 3 为结束的数组： 1,-5,3,4

以下标 4 为结束的数组： 1,-5,3,4,-2

以下标 5 为结束的数组： 1,-5,3,4,-2,-3

以下标 6 为结束的数组： 1,-5,3,4,-2,-3,10

利用: $b[i] - minSum$

算法描述

```
int getmax3_N(int a[],int n)
{
    int sum=0,sumMin=0,Max=a[0];
    for(int i=0;i<n;i++)
    {
        if(sumMin>sum)
            sumMin=sum;
        sum=sum+a[i];
            if(Max<sum-sumMin)
        Max=sum-sumMin;
    }
    return Max;
}
```

扩展1：如何求最大子数组的位置?

- 即如何求最大子数组的起点下标、结束点下标
- 求最大子数组的位置时，可能会出现一个数组中有许多和是一样的最大子数组，这里我们可以限定，优先考虑位置最靠前（起点靠前）的最大子数组，当最大子数组的起点也相同时，我们选定结束点靠前的。

例如：求2, 3, -1, 1, -5, 4, 1的最大子数组的和，并求该子数组的位置：

- 结果为：5 0 1。
- 这里第一个整数5代表最大子数组的和，0代表最大子数组的起点下标，1代表最大子数组的结束点下标。虽然5 5 6也对，但是子数组的起点靠后，5 0 3也对，但是子数组的结束点靠后。

起始点与结束点下标如何来记录?

- 由于我要求起始点下标、结束点下标都靠前的子数组，所以我们在动态规划的时候最好从后向前递推，这样dp[i]表示的值就是以下标i为开始的最大子数组的值，那么当dp[i]与dp[j]相同时我们选取i, j中较小的下标作为起点。当出现2,3,-1,1这样的情况时，我们需要选择3的下标作为结束点，考虑下面的代码：

```
if(dp[i+1]>0)
{
    dp[i]=dp[i+1]+a[i];
    tmp[i]=tmp[i+1];
}
else
{
    dp[i]=a[i];
    tmp[i]=i;
}
```

- 这里tmp变量记录子数组的结束点，当dp[i+1]==0时，我们要选取i作为此时的子数组结束点。如上面的例子2,3,-1,1；当我们计算dp[1]时，它的结束点就是1，而不是dp[2]的结束点3。
- 最终最大的dp[i]代表最大子数组的和，最小的i代表子数组的起始点，tmp[i]代表子数组的结束点。

代码优化

```
void getMax(int a[],int n) {
    int dp, tmp;
    dp=Max=a[n-1];
    tmp=n-1;
    B=n-1;
    E=n-1;
    for(int i=n-2;i>=0;i--) {
        if(dp>0)
            dp=dp+a[i];
        else {
            dp=a[i];
            tmp=i;
        }
        if(Max<=dp) {
            B=i;
            E=tmp;
            Max=dp;
        }
    } }
```

int B,E,Max;//B为子数组的起始点下标,E为子数组的结束点下标
//Max为最大子数组的和。
//getMax求数组a(元素个数为n,以下标0作为起始点)的最大子数组,外加起点下标与
//结束点下标。

由于dp[i]只与dp[i+1]有关系，tmp[i]只与tmp[i+1]有关系，所以我们可以省略数组，改为变量dp, tmp。

更上一层楼

1.将问题拓展到两段和，一个数组的两段最大子数组的和。例如1,-2,2,3,-4结果就是6。第一段就是1，第二段就是2,3。

2.将问题拓展到一个循环数组，就是一个圈，下标为n-1的元素后面的元素下标为0，求这样的数组的最大子数组的和。

3.加入限制条件m，m代表子数组的最大长度，即求长度不超过m的最大子数组的和。

4.将问题拓展到二维，求一个矩阵的最大子矩阵的和

最大子矩阵推广

(1) 最大子矩阵和问题

给定一个m行n列的整数矩阵a，试求矩阵a的一个子矩阵，使其各元素之和为最大。

记 $s(i1,i2,j1,j2)=\sum_{i=i1}^{i2}\sum_{j=j1}^{j2}a[i][j]$

最大子矩阵和问题的最优值为

$$\max_{\substack{1\le i1\le i2\le m\\ 1\le j1\le j2\le n}} s(i1,i2,j1,j2)$$

时间复杂性：

由于解最大子段和问题的动态规划算法需要时间$O(n)$，故算法的双重for循环需要计算时间$O(m^2n)$。

（2）最大m子段和问题

问题描述：给定由n个整数（可能为负整数）组成的序列$a_1, a_2, \ldots, a_n$，以及一个正整数m，要求确定序列的m个不相交子段，使这m个子段的总和达到最大。

解题思路：设b(i, j)表示数组a的前j项中i个子段和的最大值，且第i个子段含a[j]($1 \le i \le m, i \le j \le n$)。则所求的最优值显然为

$$\max_{m \le j \le n} b(m, j)$$

与最大子段和问题类似，计算b(i, j)的递归式为

$$b(i,j) = \max\{b(i,j-1) + a[j], \max_{i-1 \le t < j} b(i-1,t) + a[j]\} \quad (1 \le i \le m, i \le j \le n)$$

初始时，b(0, j)=0，($1 \le j \le n$); b(i, 0)=0，($1 \le i \le m$)。

求解：一个数组的两段最大子数组的和

- 例如：
  - 1,-2,2,3,-4结果就是6。第一段就是1，第二段就是2,3。
- 已知整数序列(数组) A={a1, a2,..., an}, 定义函数d(A) 如：

$$d(A) = \max_{1 \le s_1 \le t_1 < s_2 \le t_2 \le n} \left\{ \sum_{i=s_1}^{t_1} a_i + \sum_{j=s_2}^{t_2} a_j \right\}$$

算法分析：

1	-1	2	2	3	-3	4	-4	5	-5

设：
dp1[i]代表以下标i结束的最大子数组的值，
max1[i]代表dp1[0]~dp1[i]最大的那个值；
设
dp2[i]代表以下标i开始的最大子数组的值，
max2[i]代表dp1[i]~dp1[n-1]中最大的那个值。

例如，max1[6]就是dp[6]即{2,2,3,-3,4}；max2[7]就是dp[8]即{5}，
所以上面那个例子求出的解就是max1[6]+max2[7]。而为何是max1[6]+max2[7]，是因为在max1[i]+max2[i+1](i=0~n-2)中，它最大。我们只需要枚举这两段的分界点i，0<=i<=n-2，就可以了。

最后得出方法：

- 利用标准动态规划求最大子子段和的方法，求出dp1[]、dp2[]、max1[]、max2[]；
- 枚举出max1[i]+max2[i+1](i=0~n-2)中，最大的那个数，就是结果。

```
int max1[MAXN],max2[MAXN];
int getMax(int a[],int n)
{
    int Max,dp;
```

```
max1[0]=dp=a[0];
for(int i=1;i<n;i++)
{
    if(dp<0)
        dp=a[i];
    else
        dp=dp+a[i];
    if(max1[i-1]<dp)
        max1[i]=dp;
    else
        max1[i]=max1[i-1];
}
```

```
max2[n-1]=dp=a[n-1];
for(int i=n-2;i>=0;i--)
{
    if(dp<0)
        dp=a[i];
    else
        dp=dp+a[i];
    if(max2[i+1]<dp)
        max2[i]=dp;
    else
        max2[i]=max2[i+1];
}
```

```
    Max=max1[0]+max2[1];
    for(int i=1;i<n-1;i++)
        if(Max<max1[i]+max2[i+1])
            Max=max1[i]+max2[i+1];
    return Max;
}
```