第四章 动态规划

第四章 动态规划

- 动态规划概述
- 求解整数拆分问题
- 求解最大连续子序列和问题
- 求解最长递增子序列问题
- 合唱队形问题
- 求解0/1背包问题
- 求解资源分配问题
- 求解会议安排问题

动态规划概述

从求解斐波那契数列看动态规划法

求解斐波那契数列的递归算法

```
int count=1;             //累计调用的步骤
int Fib(int n)           //算法
{  printf("(%d)求解Fib(%d)\n", count++, n);
   if (n==1 || n==2)
   {  printf("   计算出Fib(%d)=%d\n", n, 1);
      return 1;
   }
   else
   {  int x=Fib(n-1);
      int y=Fib(n-2);
      printf("   计算出Fib(%d)=Fib(%d)+Fib(%d)=%d\n",
                       n, n-1, n-2, x+y);
      return x+y;
   }
}
```

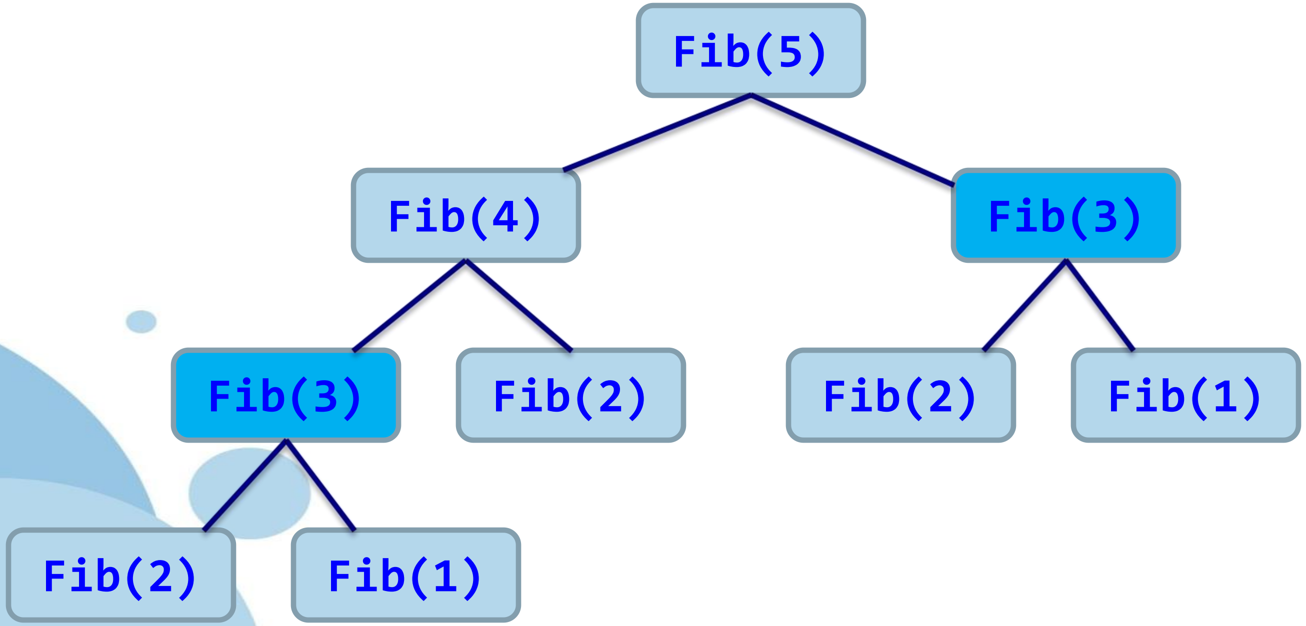
Fib1(5)时的输出结果：

```
(1) 求解Fib(5)
(2) 求解Fib(4)
(3) 求解Fib(3)
(4) 求解Fib(2)
   计算出Fib(2)=1
(5) 求解Fib(1)
   计算出Fib(1)=1
   计算出Fib(3)=Fib(2)+Fib(1)=2
(6) 求解Fib(2)
   计算出Fib(2)=1
   计算出Fib(4)=Fib(3)+Fib(2)=3
(7) 求解Fib(3)
(8) 求解Fib(2)
   计算出Fib(2)=1
(9) 求解Fib(1)
   计算出Fib(1)=1
   计算出Fib(3)=Fib(2)+Fib(1)=2
   计算出Fib(5)=Fib(4)+Fib(3)=5
```

从中看出如下几点：

（1）递归调用Fib(5)采用自顶向下的执行过程，从调用Fib(5)开始到计算出Fib(5)结束。

（2）计算过程中存在大量的重复计算，例如求Fib(5)的过程如图8.1所示，存在两次重复计算Fib(3)值的情况。



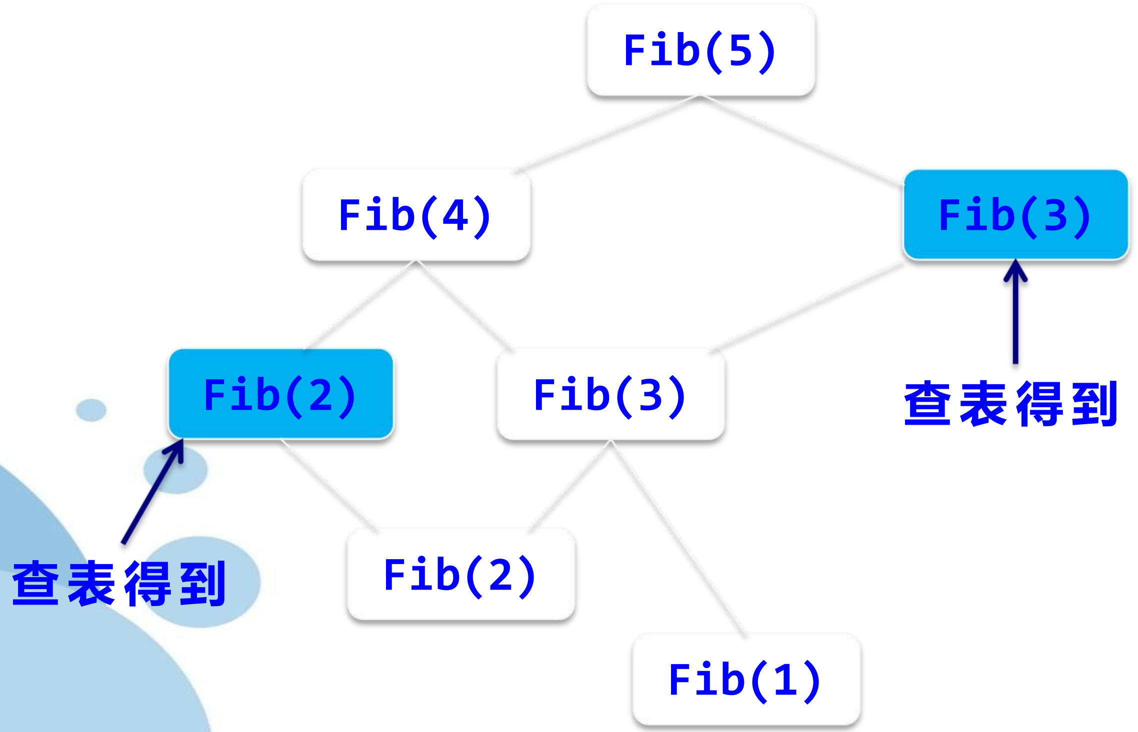
为此避免重复设计，设计一个dp数组，dp[i]存放Fib(i)的值，首先设置dp[1]和dp[2]均为1，再让i从3到n循环以计算dp[3]到dp[n]的值，最后返回dp[n]即Fib1(n)。对应的算法1如下：

```
int dp[MAX];                        //所有元素初始化为0
int count=1;                        //累计调用的步骤
int Fib1(int n)                     //算法1
{  dp[1]=dp[2]=1;
   printf("(%d)计算出Fib(1)=1\n", count++);
   printf("(%d)计算出Fib(2)=1\n", count++);
   for (int i=3;i<=n;i++)
   {  dp[i]=dp[i-1]+dp[i-2];
      printf("(%d)计算出Fib(%d)=%d\n", count++, i, dp[i]);
   }
   return dp[n];}
```

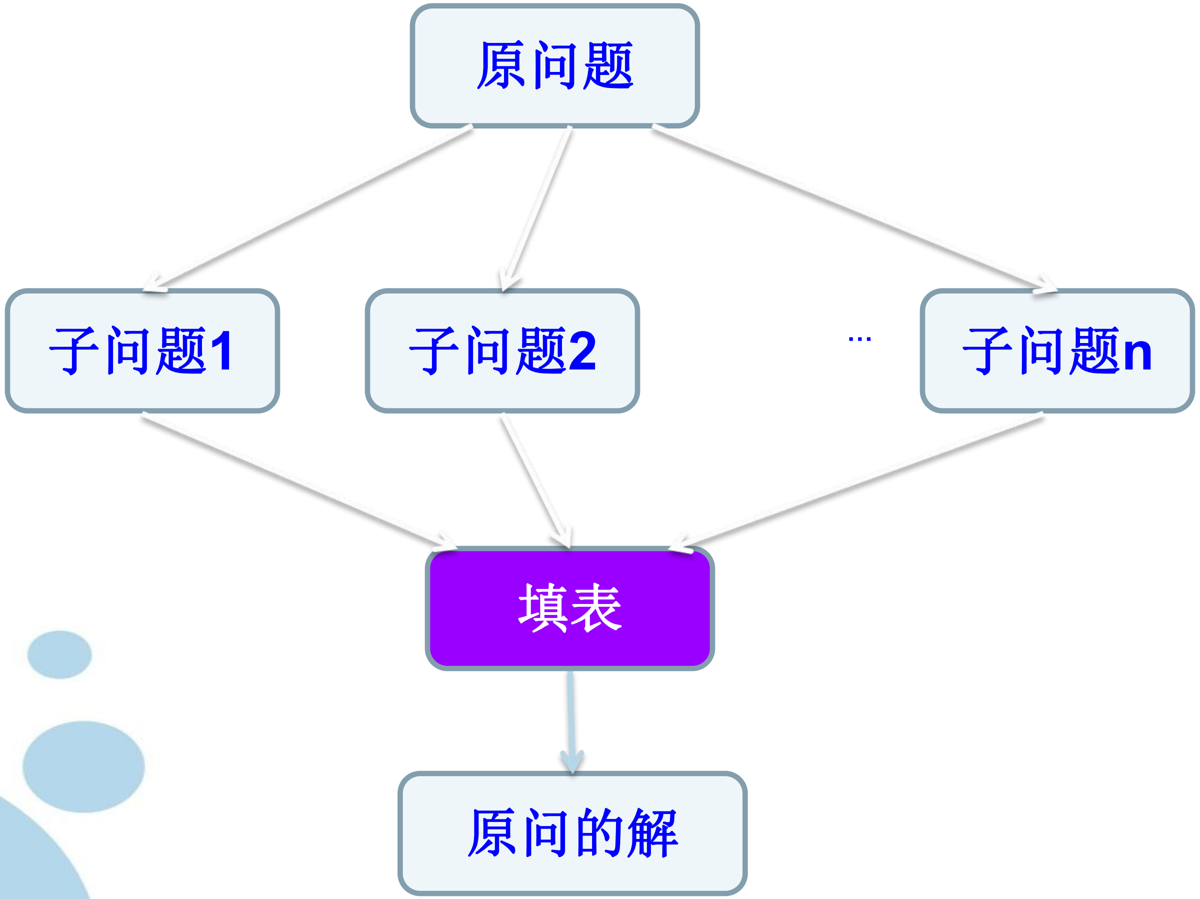
执行Fib1(5)时的输出结果如下：

（1）计算出Fib1(1)=1
（2）计算出Fib1(2)=1
（3）计算出Fib1(3)=2
（4）计算出Fib1(4)=3
（5）计算出Fib1(5)=5

其执行过程改变为自底向上，即先求出子问题解，将计算结果存放在一张表中，而且相同的子问题只计算一次，在后面需要时只有简单查表，以避免大量的重复计算。



上述求斐波那契数列的算法1属于动态规划法，其中数组dp（表）称为动态规划数组。动态规划法也称为记录结果再利用的方法，其基本求解过程如下图所示。



动态规划的原理

动态规划是一种解决多阶段决策问题的优化方法，把多阶段过程转化为一系列单阶段问题，利用各阶段之间的关系，逐个求解。

动态规划的基本思想

- 多阶段最优化决策解决问题的过程称为动态规划。动态规划通常是用递归程序实现的，递推关系是实现由分解后的子问题向最终问题求解转化的纽带。
- **动态规划的基本思想**
  - **指导思想：**

    动态规划是建立在**最优原则**的基础上 ，在每一决策步上列出各种可能局部解，按某些条件舍弃肯定不能得到最优解的局部解，这是一个寻找最优判断序列的过程，即不论初始策略如何，下一次决策必须相对前一次决策产生的新状态构成最优序列。这样，在每一步都经过筛选，以每一步的最优性来保证全局的最优性。
  - **基本思想：**

    记录子问题并不断填表。即将待求解的问题分成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。适合动态规划算法求解的问题，经分解后不是互相独立的，即它们可以在多项式时间内被求解出来

动态规划求解的基本步骤

能采用动态规划求解的问题的一般要具有3个性质：

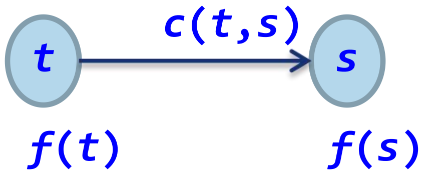
- 最优性原理（最优子结构）：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优性原理。

$$f(A)=0$$

$$f(s)=\underset{\text{存在}<t,s>\text{的有向边}}{\mathrm{MIN}}\{f(t)+c(t,s)\}$$

f(s)表示初始状态A到状态s的最短路径长度

子问题的解也是最优的



动态规划求解的三要素

1、最优子结构

证明最优子结构性质的方法：反证法。

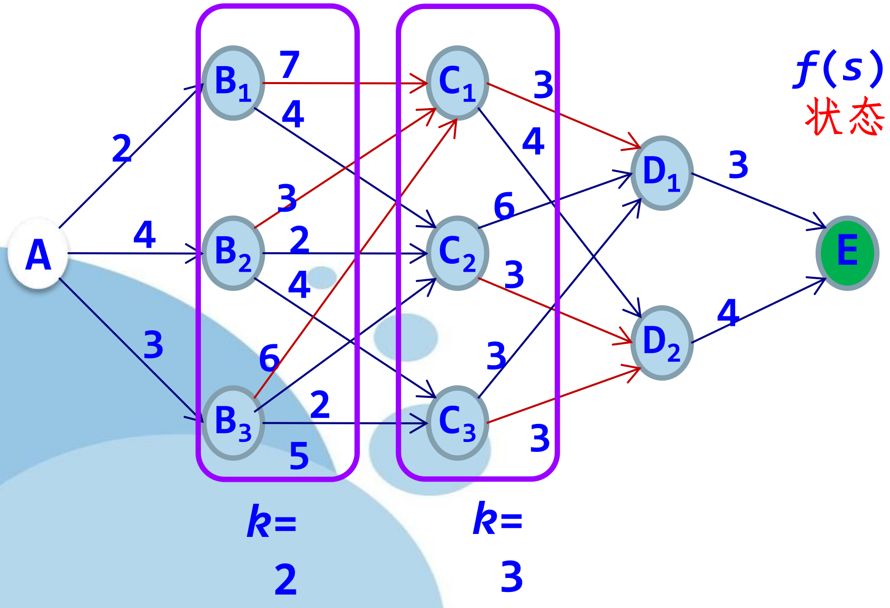
•首先假设由问题的最优解导出的子问题的解不是最优的；

•然后再设法证明在这个假设下可以构造出比原问题最优解更好的解，导致矛盾。

•说明假设不成立，从而证明该问题具有最优子结构性质。

动态规划求解的三要素

- 2、无后效性：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。
- 动态规划问题进行抽象：
  - 顶点表示状态，边表示状态之间的关系，由无后效性可知，状态图必将是一个有向无环图。



f(*s*)表示初始状态A到状态*s*的最短路径长度

$$f(C_1)=\mathrm{MIN}\begin{pmatrix} f(B_1)+c(B_1,C_1)=9 \\ f(B_2)+c(B_2,C_1)=7 \\ f(B_3)+c(B_3,C_1)=9 \end{pmatrix}=7,\ \mathrm{pre}(C_1)=B_2$$

$$f(C_2)=\mathrm{MIN}\begin{pmatrix} f(B_1)+c(B_1,C_2)=6 \\ f(B_2)+c(B_2,C_2)=6 \\ f(B_3)+c(B_3,C_2)=5 \end{pmatrix}=5,\ \mathrm{pre}(C_2)=B_3$$

$$f(C_3)=\mathrm{MIN}\begin{pmatrix} f(B_1)+c(B_1,C_3)=\infty \\ f(B_2)+c(B_2,C_3)=8 \\ f(B_3)+c(B_3,C_3)=8 \end{pmatrix}=8,\ \mathrm{pre}(C_3)=B_2$$

动态规划求解的三要素

- 3、有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）。

求斐波那契数列

$f(1)=1$

$f(2)=1$

$f(n)=f(n-1)+f(n-2)\quad n>2$

动态规划求解的基本步骤

实际应用中简化的步骤：

① 分析最优解的性质，并刻画其结构特征。

② 递归的定义最优解。

③ 以自底向上或自顶向下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，构造问题的最优解。

动态规划与其他方法的比较

动态规划的基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。

在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

动态规划与其他方法的比较

动态规划方法又和贪心法有些相似，在动态规划中，可将一个问题的解决方案视为一系列决策的结果。

不同的是，在贪心法中，每采用一次贪心准则便做出一个不可回溯的决策，还要考察每个最优决策序列中是否包含一个最优子序列。

动态规划法求解组合数

分别用递归和动态规划算法解。

递归：重复求解子问题，如算法3.1所示。

【算法3.1 用递归求解组合问题】

```
/*功能：求解。
  输入：正整数n,m
  输出：输出的结果
*/
   int ComB(int n,int m)
   {  if  (m==0 || n==m)     return(1);
     else
      return(ComB(n-1,m-1)) +return(ComB(n-1, m));
   }
```

动态规划法求解组合数

动态规划算法求解

(2)动态规划：记录子问题，如算法3.2所示。

步骤1：分析最优子结构

计算组合数C_n^m ，可将原问题分解为求解两个子问题C_{n-1}^{m-1} C_{n-1}^m ，要采用自底向上的方法，先计算出C_i^1(i=1...n)，然后用公式 $C_{i+1}^2=C_i^1+C_i^2$（i=1...n，初值 $C_1^2=0$） ......，依次计算出其他各项值并填表，当子问题被求解，原问题得解。由于采用了填表技术，子问题被求解后就不用重复计算，达到子问题最优求解方式，具有最优子结构的性质。

步骤2：建立递归关系

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

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

动态规划法求解组合数

动态规划算法解。

步骤3：计算最优值

如求解 C_5^3 （n=5，m=3），通过动态规划算法来记录 C_i^j （其中 C_i^j $i=1..5, j=1..3$ ），

结果如表3-1所示，如计算表中第四行第二列的值 C_4^2 ，可以用第三行第一列的 C_3^1 与第三行第二列的 C_3^2 求和得到，即 $C_4^2=C_3^1+C_3^2$ =6

表3-2组合数计算动态规划表

	i=1	i=2	i=3
j=1	1	0	0
j=2	2	1	0
j=3	3	3	1
j=4	4	6	4
j=5	5	10	10

动态规划法求解组合数

动态规划算法解组合数。

步骤4：算法描述及分析

【算法3.2 用动态规划求解组合问题】

```
/*功能：求解。
  输入：正整数n,m
  输出：输出的结果
*/
int ComB(int n, int m )
{  int C[n+1][m+1],i,j;    /*为更加简洁，本例数组下标从1开始*/
   for (j=1;j<=m;j++)  C[1][j]=0;
     for (i=1;i<=n; i++)  C[i][1]=i;
    for (i=2;i<=n;i++)
      for (j=2 ;j<=m; j++)
          if(i<j)   C[i][j]=0;
            else   C[i][j]=C[i-1][j-1]+C[i-1][j];
         return(C[n][m]);
 }
```

动态规划法求解整数拆分问题

【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。

动态规划法求解整数拆分问题

【问题求解】设n=5，k=5，对应的拆分方案有：

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

为了防止重复计数，让拆分数保持从大到小排序。正整数5的拆分数为7。

动态规划法求解整数拆分问题

采用动态规划求解整数拆分问题。设$f(n, k)$为n的k拆分的拆分方案个数：

（1）当$n=1$，$k=1$时，显然$f(n, k)=1$。

（2）当$n<k$时，有$f(n, k)=f(n, n)$。

（3）当$n=k$时，其拆分方案有将正整数n无序拆分成最大数为$n-1$的拆分方案，以及将n拆分成1个n（$n=n$）的拆分方案，后者仅仅一种，所以有$f(n, n)=f(n, n-1)+1$。

动态规划法求解整数拆分问题

（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$f(n-k, k)$。

$$n = k + \{x_1, x_2, \ldots, x_i\}$$

$$\Downarrow$$

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

动态规划法求解整数拆分问题

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$f(n, k-1)$。

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

$$\Downarrow$$

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

因此，$f(n, k) = f(n-k, k) + f(n, k-1)$

动态规划法求解整数拆分问题

状态转移方程：

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

动态规划法求解整数拆分问题

显然，求$f(n, k)$满足动态规划问题的最优性原理、无后效性和有重叠子问题性质。所以特别适合采用动态规划法求解。设置动态规划数组dp，

用dp[n][k]存放$f(n, k)$。

dp[n][k] ⇔ $f(n, k)$

↑

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

动态规划法求解整数拆分问题

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

```
int dp[MAXN][MAXN];                    //动态规划数组
void 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];
      }
}
```

动态规划法求解整数拆分问题

Split()算法计算dp[5][5]的过程：
Split()算法中按行优先计算dp[i][j]，其中dp[1][*]和dp[*][1]为边界，值均为1.

(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

n \ k	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

动态规划法求解整数拆分问题

实际上，该问题本身是递归的，可以直接采用递归算法实现！

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

```
int fun(int n, int k)                          //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,k)+fun(n,k-1);
}
```

动态规划法求解整数拆分问题

但由于子问题重叠，存在重复的计算！

可以采用这样的方法避免重复计算：设置数组dp，用dp[n][k]存放$f(n，k)$，首先初始化dp的所有元素为特殊值0，当dp[n][k]不为0时表示对应的子问题已经求解，直接返回结果。

递归+dp[n][k]的备忘录法

动态规划法求解整数拆分问题----备忘录方法

采用自顶向下（备忘录方法）的动态规划法

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

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]; }
}
```

动态规划法求解整数拆分问题----备忘录方法

- 这种方法是一种递归算法，其执行过程也是自顶向下的，但当某个子问题解求出后，将其结果存放在一张表（dp）中，而且相同的子问题只计算一次，在后面需要时只有简单查表，以避免大量的重复计算。这种方法称之为备忘录方法（memorization method）。
- 备忘录方法是动态规划方法的变形，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的。

动态规划法求解最大连续子序列和问题

【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。

动态规划法求解最大连续子序列和问题

【问题求解】对于含有n个整数的序列a，设

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

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：

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

动态规划法求解最大连续子序列和问题

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

$$\underbrace{\overbrace{a1\quad a2\quad \dots\quad aj\text{-}1}^{bj\text{-}1}\quad aj}_{bj}\quad \dots\quad an$$

发现：

bj-1>0时bj=bj+aj
bj-1≤0时放弃前面选取的元素，从aj开始重新选取，bj=aj。

动态规划法求解最大连续子序列和问题

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	边界条件
dp[j]=MAX{dp[j-1] +a_j，a_j}	$1\leq j\leq n$

构造问题的解

序列a的最大连续子序列和等于：dp[j]（$1\leqslant j\leqslant n$）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。**

动态规划法求解最大连续子序列和问题

dp[0]=0　　　　边界条件

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

```
//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
{   边界条件   dp[0]=0;
    其他dp[j]
       for (int j=1;j<=n;j++)
           dp[j]=max(dp[j-1]+a[j], a[j]);
}
```

动态规划法求解最大连续子序列和示例

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

动态规划法求解最大连续子序列和问题

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

动态规划法求解最大连续子序列和问题

```
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");
}
```

【算法分析】maxSubSum()的时间复杂度为O(n) 。