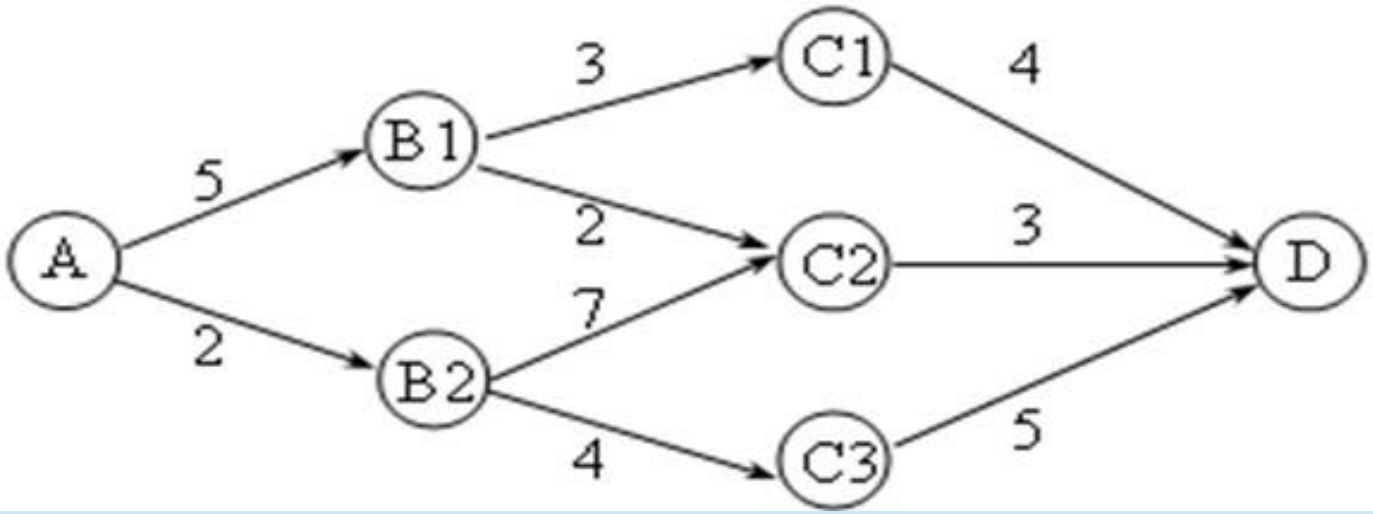
动态规划初步

主讲老师：党东

【引例】最短路径问题

【题目简述】给出你一个地图，地图中每个顶点代表一个城市，两个城市间的连线代表道路，连线上的数值代表道路的长度。现在，想从城市A到达城市D，怎样走路程最短，最短路程的长度是多少?



首先，我们看这里共有4条路径： A→B1→C1→D，A→B1→C2→D，A→B2→C2→D，A→B2→C3→D ，它们的长度分别为：5+3+4=12，5+2+3=10，2+7+3=12，2+4+5=11。

【引例】最短路径问题

【方法1】如果采用枚举算法，分别枚举出4条路径，然后比较每条路经的长度，得出最优解10，路径为A→B1→C2→D。

【方法2】如果我们采用分层递推的思想来做这个题目，会出现什么结果呢?

设F(i)表示从点A到达点i的最短距离，则有

F(A)=0

F(B1)=min{F(A)+5}=5

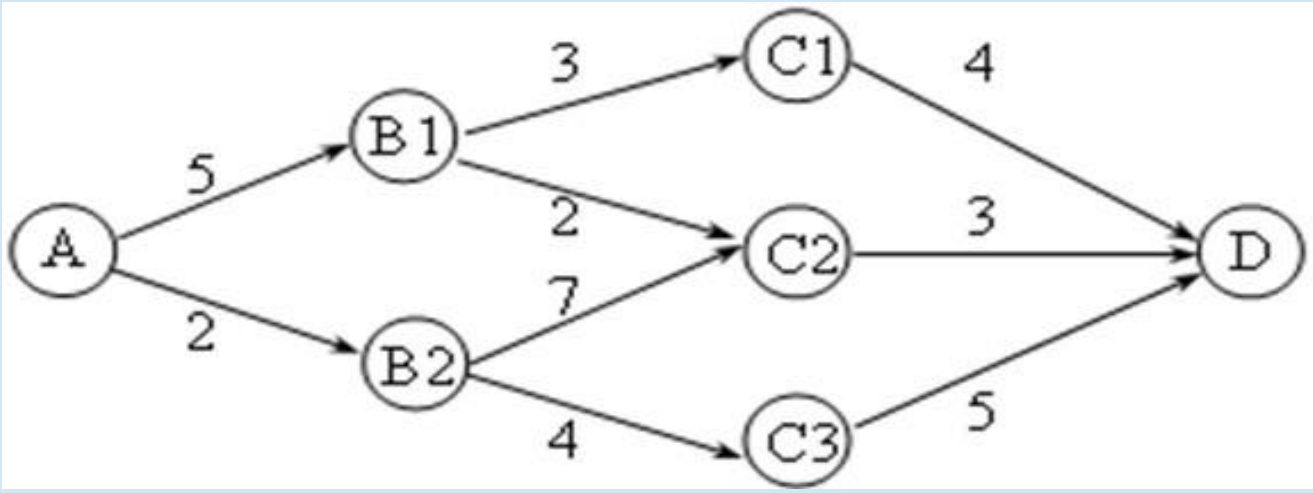
F(B2)=min{F(A)+2}=2

F(C1)=min{F(B1)+3}=8

F(C2)=min{F(B1)+2,F(B2)+7}=7

F(C3)=min{F(B2)+4}=6

F(D)=min{F(C1)+4,F(C2)+3,F(C3)+5}=10



所以从A到D的最短距离是10，路径为A→B1→C2→D。

【引例】最短路径问题

比较上述两种方法：

方法1采用了枚举法对于边A→B1,A→B2,C2→D都计算了2次。

方法2采用逐层递推的方法，每条边只计算了1次。

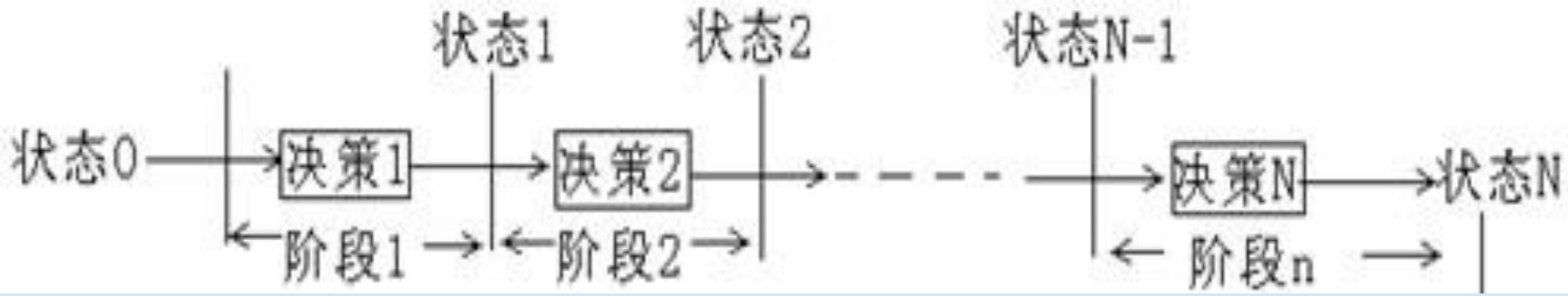
因此，方法2比方法1优秀。

那么造成两种方法效率不同的根本原因是什么呢？从引例中可以看出，方法2采用的是逐层递推的方法从根本上消除了枚举法对路径中部分重复路径的冗余运算。

由上例可以看出，整个问题分成了A、B、C、D四个阶段来做，每个阶段的数值计算只会跟上一个阶段的数值相关，这样一直递推下去直到目标。

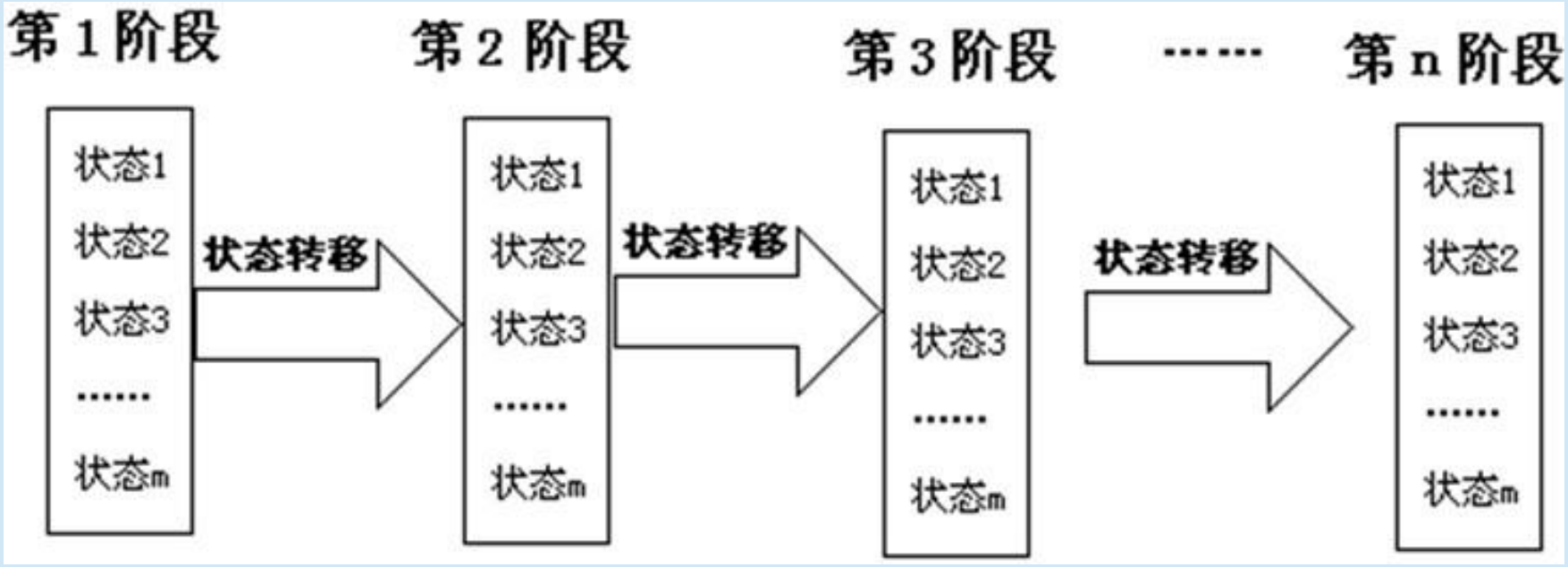
【引例】最短路径问题

即由**初始状态**开始，通过对中间阶段**决策的选择**，达到**结束状态**。这些决策形成了一个决策序列，同时确定了完成整个过程的一条最优的活动路线。这种把一个问题看作是一个前后关联且具有链状结构的多阶段过程，这过程称为多阶段决策过程，这种问题称为多阶段决策问题，如下图：



【引例】最短路径问题

在多阶段决策的问题中，各个阶段采取的决策，一般来说是与时间或空间有关的，决策依赖于当前状态，又随即引起状态的转移，一个决策序列就是在变化的状态中产生出来，故有“动态”的含义，我们称这种**解决多阶段决策最优化的过程为动态规划**。



动态规划的示意图

【例1】数字三角形 --1346

【问题描述】

如下图所示，它是一个数字三角形。数字三角形中的数字为不超过100的正整数。先规定从最顶层走到最底层，每一步可沿左斜线向下或右斜线向下走。假设三角形的行数<=100，编程求解从最顶层走到最底层的一条路径，使得沿该路径所经过的数字总和最大，输出最大值。

【文件输入】

输入文件第一行为一个整数n(1<=n<=100)，表示数字三角形的行数，接下来的n行，分别是从最顶层到最底层的每一层中的数字。

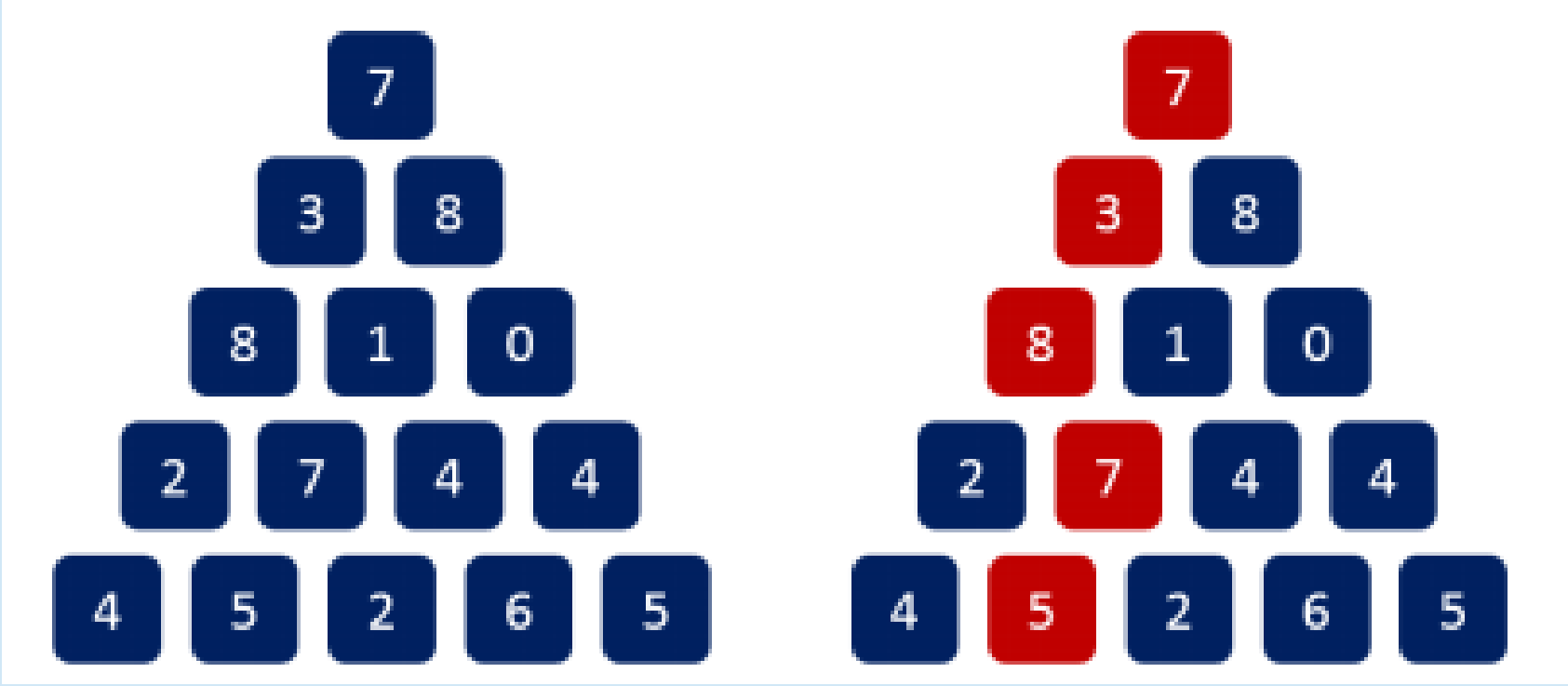
【文件输出】

输出仅有一行为一个整数，表示要求的最大总和。

【例1】数字三角形 --1346

【题目简述】

写一个程序来查找从最高点到底部任意处结束的路径，使路径经过数字的和最大。每一步可以走到左下方的点也可以到达右下方的点。



【例1】数字三角形 --1346

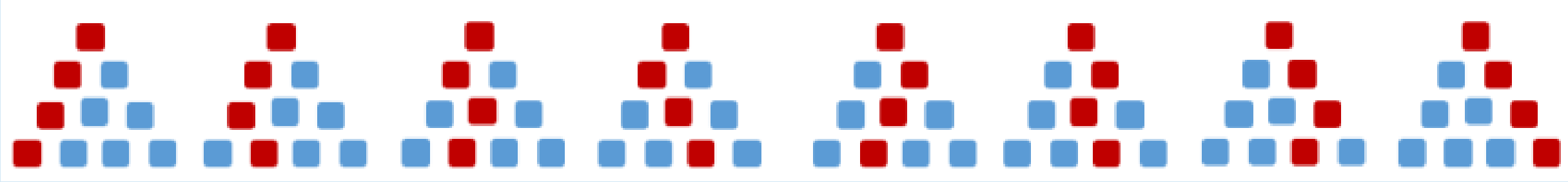
【题目分析】

我们现在这里讨论搜索如何实现

状态：目前在第x行第y列

行动：向左走，向右走

一个底边为4的三角形共有八种状态：



【例1】数字三角形 --1346

【方法1：暴力搜索】

```
void dfs(int x,int y,int val)
{
    val+=a[x][y];//加上权值
    if(x==n-1)
    {
        if(val>ans) ans=val;//更新更大的ans
        return;
    }
    dfs(x+1,y,val);//往左边走
    dfs(x+1,y+1,val);//往右边走
}
```

【例1】数字三角形 --1346

【暴力搜索分析】

考虑时空效率，DFS确实很暴力啊，有没有什么优化呢?

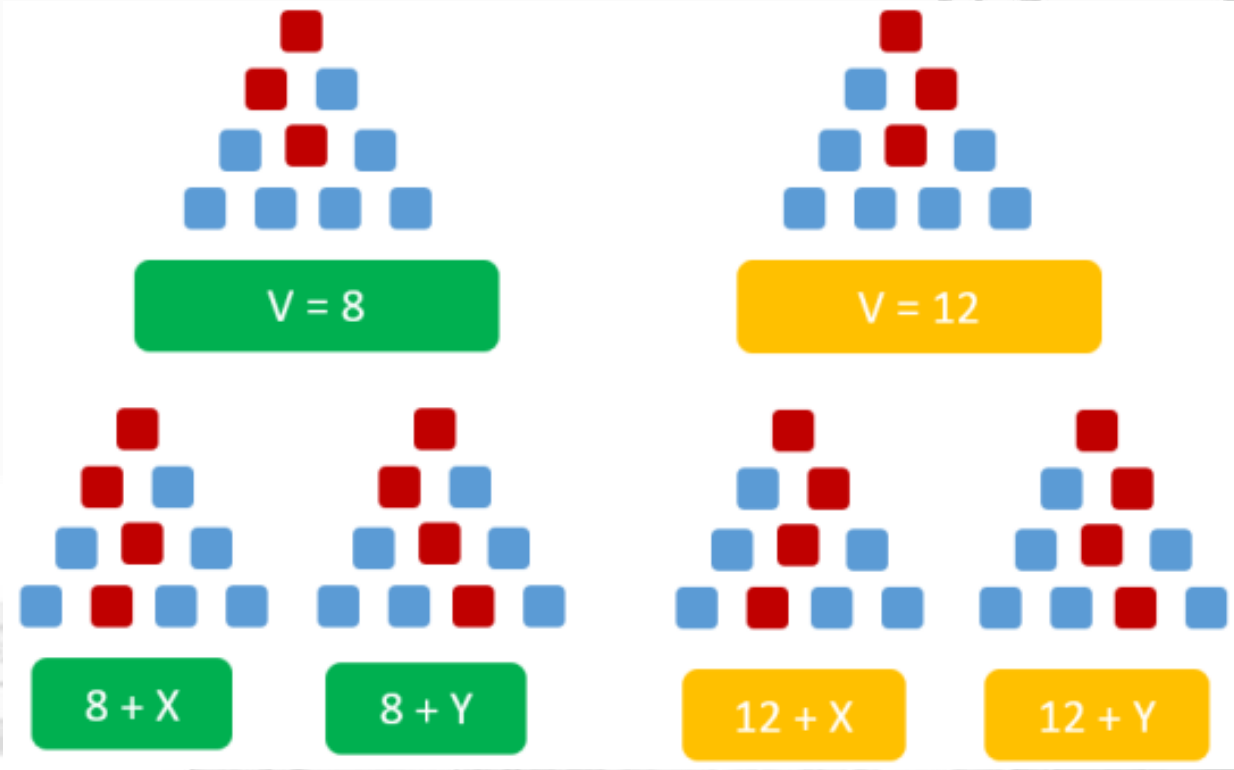
我们引入**“冗余搜索”**这个概念：无用的，不会改变答案的搜索

【例1】数字三角形 --1346

【暴力搜索分析】

观察两个例子。

- 用两种方式都能到达第 3 行第 2 列，只是路径不同，同时走到这个点两条路权值和不一样，其中一个总和为 8，一个总和 12。
- 那么可以观察可得，总和为 8 的搜索是冗余的(不会改变答案)，即使不继续搜索， 答案也不会改变。
- 因为 12 往下搜索，无论往左往右，都会比 8 对应的路径大。

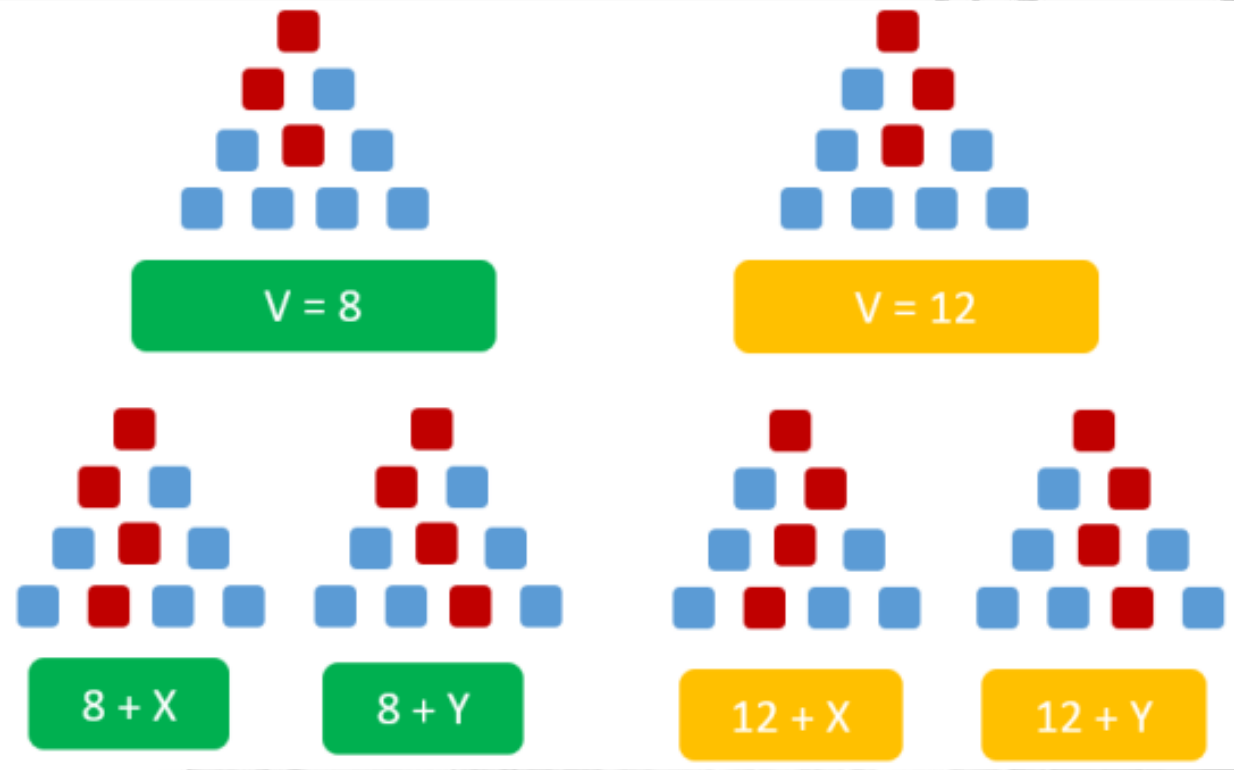


【例1】数字三角形 --1346

【暴力搜索分析】

观察两个例子。

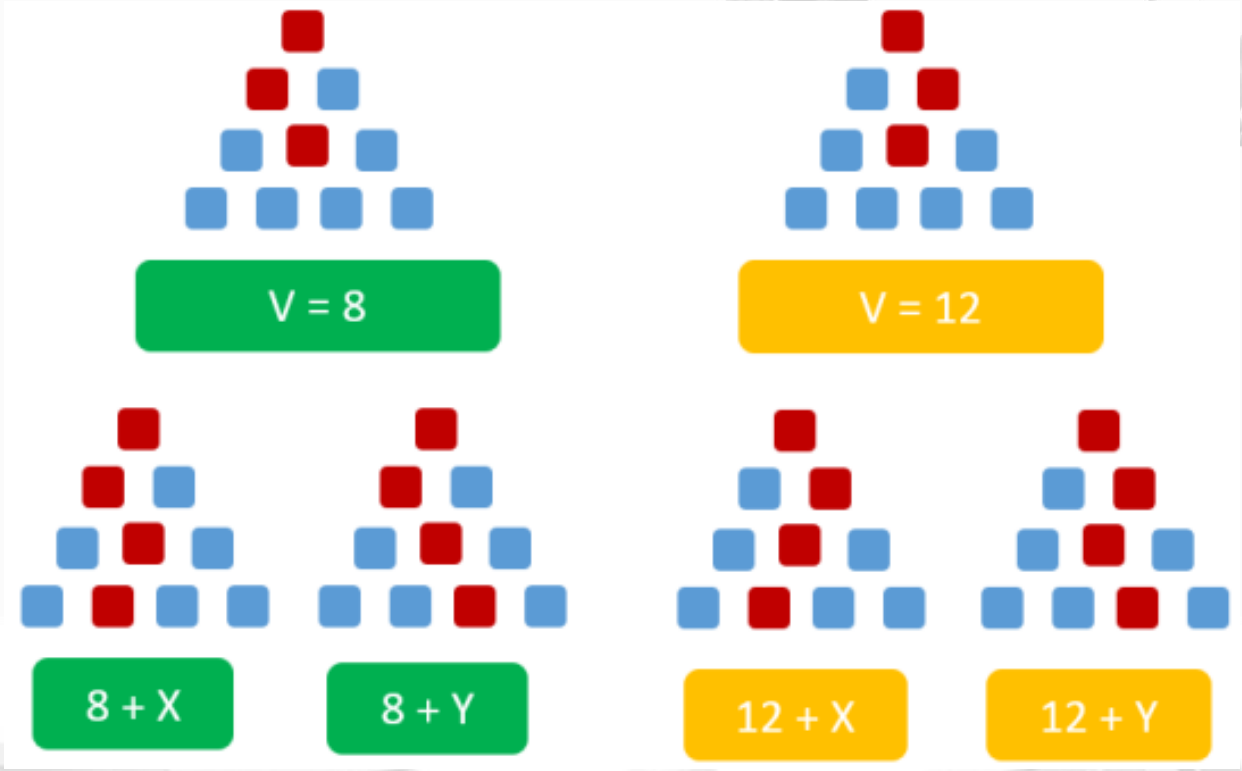
- 可见，**冗余**就是剪枝的“**枝**”，那么如何利用冗余搜索，来优化程序呢?



【例1】数字三角形 --1346

【暴力搜索分析】

- 可见，**冗余**就是剪枝的“**枝**”，那么如何利用冗余搜索，来优化程序呢?
- 我们可以对于每一个位置记录一个值F，代表搜索到此位置时，最大的路径和是多少 ，这样如果搜到某一个位置时候，路径和不大于记录值F，说明这个搜索是冗余搜索，直接退出，如果大于，就需要更新F值并且继续搜索。



- **我们就把这种搜索叫做记忆化搜索，根据之前的“记忆”来优化搜索；**
- **在这道题中，每个位置的“记忆”就是最大的路径和**

【例1】数字三角形 --1346

【方法2：记忆化搜索】

```
void dfs(int x,int y,int val)
{
    val+=a[x][y]; // 记忆化过程
    if(val<=f[x][y]) return;//发现冗余搜索，退出
    f[x][y]=val;//f[x][y]记录这个点当前最大权值
    if(x==n-1)//如果搜到了最后一个点，ans更新保存最大值，退出即可
    {
        if(val>ans) ans=val;
        return;
    }
    dfs(x+1,y,val);//继续搜索
    dfs(x+1,y+1,val);
}
```

【动态规划初步】

回到正题：动态规划！记忆化搜索是DP的基础。

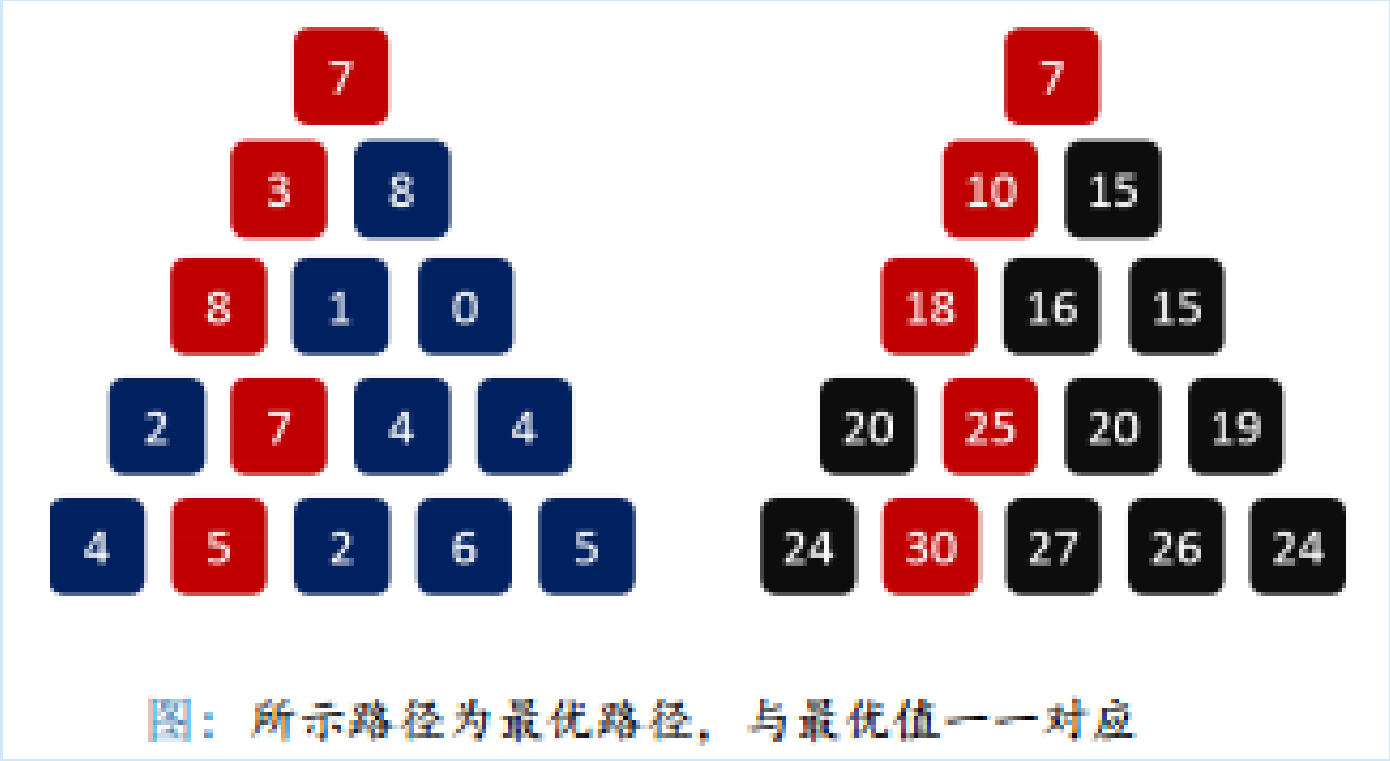
回到数字金字塔这个问题来，下图的黑色三角形是我们记忆化搜索的路径。
想一想，是不是可以不通过记忆化搜索就能得到这个黑色三角形？

图：右侧三角形：每个位置的路径和最大值

【动态规划初步】

最优性：设走到某一个位置的时候，它达到了路径最大值，那么在这之前，它走的每一步都是最大值。

-考虑这条最优的路径：每一步均达到了最大值

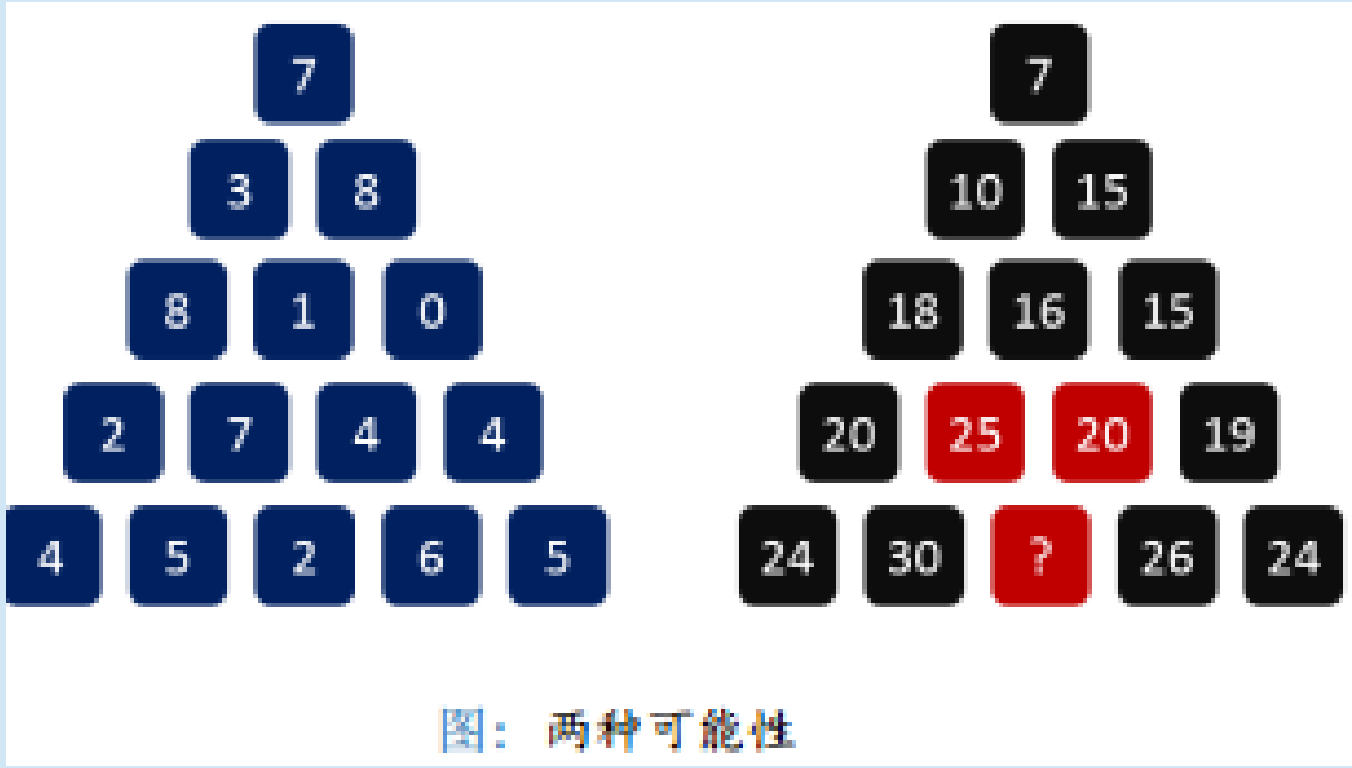


图：所示路径为最优路径，与最优值一一对应

【动态规划初步】

最优性的好处：要达到一个位置的最优值，它的前一步也一定是最优的。

-考虑图中位置，如果它要到达最优值，有两个选择，从左上方或者右上方的最优值得到：



图：两种可能性

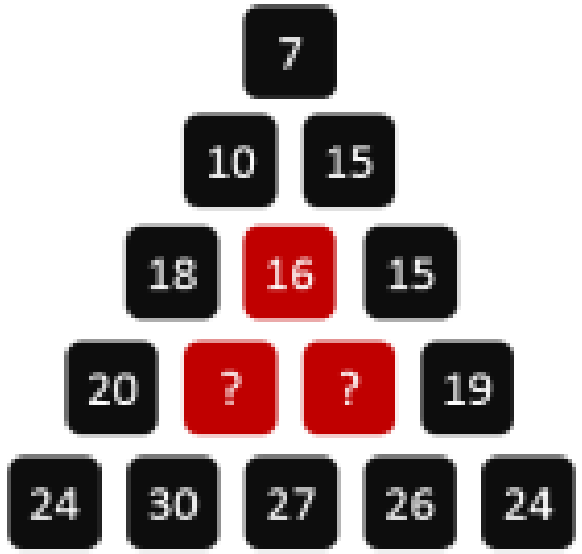
【动态规划初步】

定义动态规划（DP）：只记录状态的最优值，并用最优值来推导出其他的最优值。

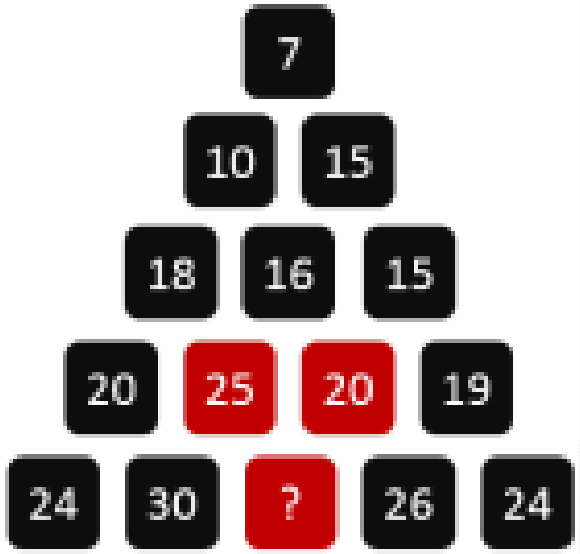
记录 F[i][j] 为第 i 行第 j 列的路径最大值，有两种方法可以推导：（两个分支两种状态，选取最大）

- **顺推：用 F[i][j] 来计算 F[i+1][j],F[i+1][j+1]**
- **逆推：用 F[i-1][j],F[i-1][j-1] 来计算 F[i][j]**

这两种思考方法也是动态规划中最基本的两种方法，解决绝大部分DP我们都可以采用这样的方法。



顺推



逆推

【动态规划初步】

【数字金字塔-顺推】

```
f[0][0]=a[0][0];
for(int i=0;i<n-1;++i)
for(int j=0;j<=i;++j)//f数组为最优值路径（黑色金字塔，a为源数据数组（紫色金字塔）
{
    //分别用最优值来更新左下方和右下方
    f[i+1][j]=max(f[i+1][j],f[i][j]+a[i+1][j]);//和当前的f[i+1][j]比较
    f[i+1][j+1]=max(f[i+1][j+1],f[i][j]+a[i+1][j+1]);//和当前的f[i+1][j+1]比较
}
```

【动态规划初步】

【数字金字塔-逆推】

```
f[0][0]=a[0][0];
for(int i=0;i<n;++i)//单独处理
{
    f[i][0]=f[i-1][0]+a[i][0];//最左的位置没有左上方
    f[i][i]=f[i-1][i-1]+a[i][i];//最右的位置没有右上方
    for(int j=1;j<i;++j)//在左上方和右上方取较大的
    f[i][j]=max(f[i-1][j-1],f[i-1][j])+a[i][j];
}
//答案可能是最后一行的任意一列
ans=0;
for(int i=0;i<n;++i)   ans=max(ans,f[n-1][i]);
```

【动态规划初步】

【状态转移方程】

***转移方程：最优值之间的推导公式。**

顺推：

F[i+1][j] = MAX (F[i][j] + a[i+1][j]);

F[i+1][j+1] = MAX (F[i][j] + a[i+1][j+1]);

逆推：

F[i][j] = MAX (F[i-1][j], F[i-1][j-1]) + a[i][j]; (注意！逆推时要注意边界情况！)

顺推和逆推本质上是一样的（复杂度一致）；顺推和搜索的顺序类似；而逆推则是将顺序反过来；顺推考虑的是“**我这个状态的下一步去哪里**”，逆推的考虑的是“**从什么状态可以到达我这里**”。同时在转移的过程中我们要时刻注意边界情况。

【动态规划初步】

【改变搜索顺序-逆推/路径自底向上】//改变顺序：记录从底部向上走的路径最优值

```
for(int i=0;i<n;++i)      f[n-1][i]=a[n-1][i];//备份底部自己这一行

//逆推过程：可以从左下方或右下方走过来；没有边界情况
for(int i=n-2;i>=0;--i)
    for(int j=0;j<=i;++j)
        f[i][j]=max(f[i+1][j+1],f[i+1][j])+a[i][j];//当前[i][j]左下方和右下方取较大加上当前的

ans=f[0][0]; //答案则是顶端
```

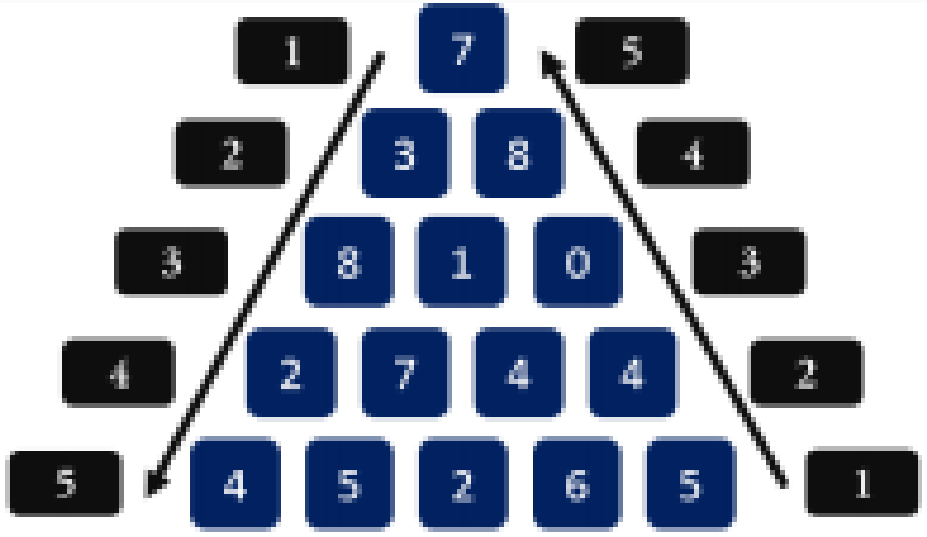
/和之前的逆推区别：这样较自顶向下不需要判断边界，更加简单。

【动态规划初步】

【多阶段决策】转移顺序：最优值之间的推导顺序

小问题：在数字金字塔中，为什么能够使用动态规划呢？

答：因为有明确的顺序： 自上而下 。也就是说，**能划分成不同的阶段，这个阶段是逐步进行的**，这和搜索顺序也是类似的，所以，只要划分好阶段， 从前往后推，与从后往前推都是可以的



【动态规划初步】

【状态设计-重点】

状态设计：记录 F[i][j] 为第 i 行第 j 列的路径最大值，有两种方法可以推导：

顺推：“我这个状态的下一步去哪里”

逆推：“从什么状态可以到达我这里”

【例2】最长上升子序列 --1355

Description

一个数的序列bi，当b1< b2 < ... < bS的时候，我们称这个序列是上升的。对于给定的一个序列(a1,a2,...,aN)，我们可以得到一些上升的子序列(ai1, ai2, ..., aiK)，这里1<=i1 你的任务，就是对于给定的序列，求出最长上升子序列的长度。

Input

输入的第一行是序列的长度N(1<=N<=1000)。第二行给出序列中的N个整数，这些整数的取值范围都在0到10000。

Output

最长上升子序列的长度。

Sample Input

6

1 6 2 5 4 7

Sample Output

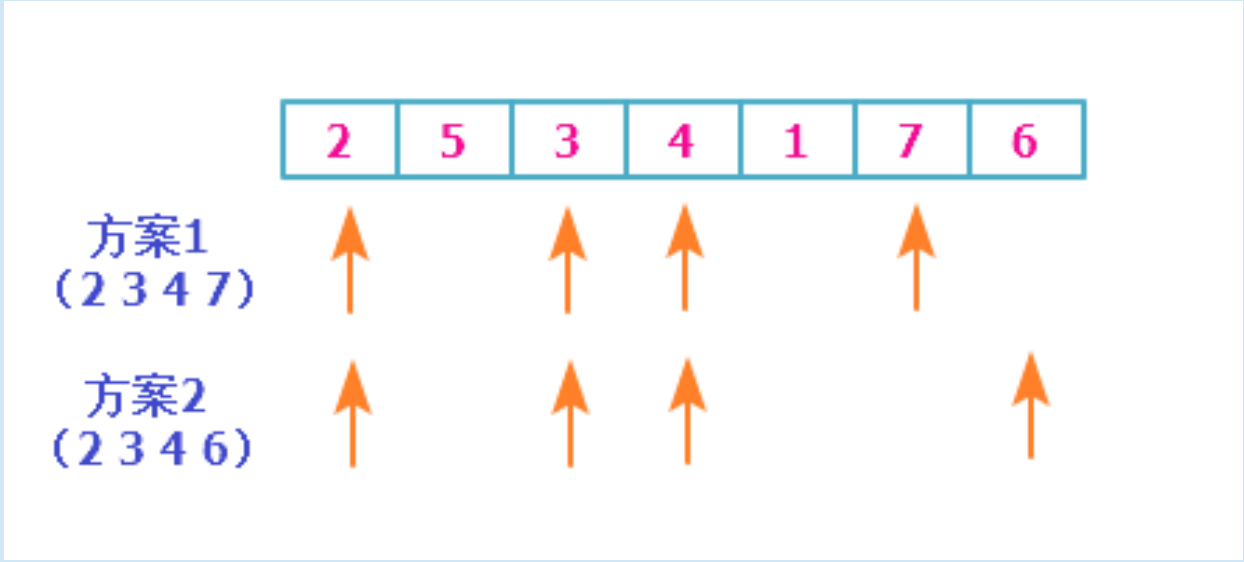
4

【例2】最长上升子序列 --1355

【题目分析】

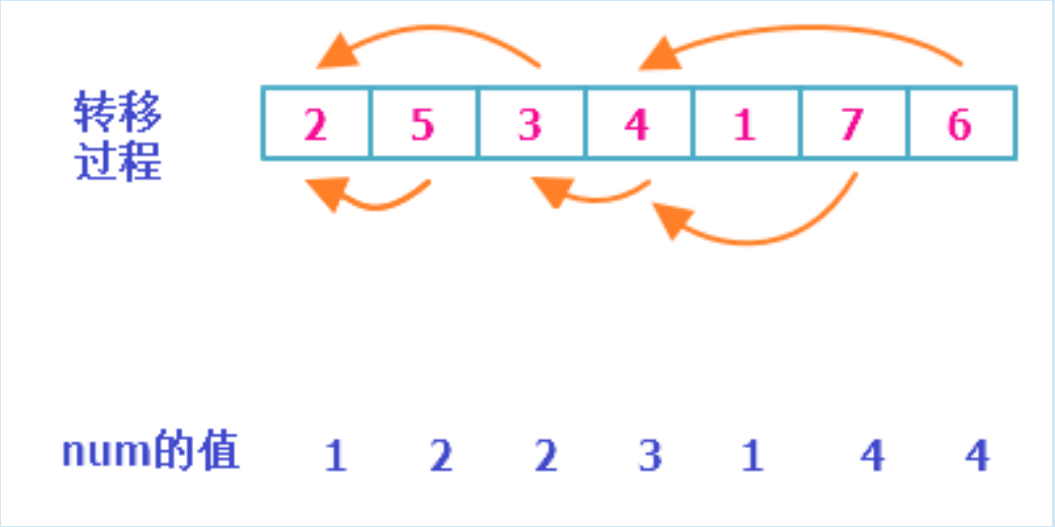
什么是最长上升子序列?

就是给你一个序列，请你在其中求出一段不断严格上升的部分，它不一定要连续。



【例2】最长上升子序列 --1355

```
int dp[1005],m[1005];
int main(){
        int n,dmax=-1;
        cin>>n;
        for(int i=1;i<=n;i++) cin>>m[i];
        for(int i=1;i<=n;i++)  //以i结尾的最大长度序列
        {
                dp[i]=1;
                for(int j=1;j<i;j++)
                {
                        if(m[j]<m[i])
                                dp[i]=max(dp[i],dp[j]+1);
                }
                dmax=max(dmax,dp[i]);
        }
        cout<<dmax;
    return 0;
}
```



动态规划的基本概念和原理

【基本概念】动态规划的基本概念

（1）状态与状态特征

我们把每个子问题的中间解称为一个状态，例如引例中的F[A]、F[B1]、F[C3]等就是不同的状态。通常一个阶段包含若干个不同的状态。状态的特征是指影响问题目标的一个或几个因素，我们根据这些因素来区分不同的状态。引例中的状态特征就是顶点的序号，根据序号我们才把F[A]、F[B1]和F[C3]这些状态区分开。

（2）决策与策略

在递推过程中，我们要做出一些取舍（选择），以决定如何从已知的状态推出未知的状态，对于特定状态的选择就被叫做决策。如引例中，在决定F[D]的值时，取F(C2)+3而舍F(C1)+4和F(C3)+5，就是一个明智的决策。

由一连串的决策所构成的序列称为策略，能够得到满足要求的解的策略就是最优策略。如引例中的最优策略就是选择A→B1→C2→D作为最短路径上的点。

【基本概念】动态规划的基本概念

（3）规划方向与阶段

因为所有状态在整个解题过程中的地位和作用并不完全相同，所以每个问题的解决必须依照一定的次序。如引例中F[B1]与F[B2]的地位是相同的，所以哪个先求解都没有关系；而F[B1]与F[C1]处在不同的地位，因此F[B1]必须在F[C1]之前求解。

我们把这种解题的次序称为规划方向，把地位相同的状态称为一个阶段。可以看出引例中的规划方向是从前到后的：由A→B→C→D；阶段A由F[A]独立构成，阶段B由F[B1]和F[B2]构成，阶段C由F[C1]、F[C2]和F[C3]构成，阶段D由F[D]构成。

（4）边界

在实践中，问题一般包含多个阶段才有讨论的意义，问题的最初阶段，我们称作为边界。边界必须在递推之前根据题意人为给定，它是递推的基础。这就像是在运用数学归纳法证明命题的正确性，首先证明命题n=n0时是成立的，接下来才能证明n>n0时命题也是成立的，因为前者是后者的基础。在引例中，阶段A就是边界，即F(A)=0。

【基本概念】动态规划的基本概念

（5）状态转移方程

我们知道，除边界外的任一阶段都得由其前面的阶段递推得到，这递推的过程就表现出了阶段的动态演变。这种由已知状态求得未知状态的过程，我们称之为状态转移，状态转移的规则用数学语言来描述，就称为状态转移方程。

状态转移方程的形式多样，如引例中的形式为：设$F_k(i)$—k阶段状态i的最优权值，即初始状态至状态i的最优代价。

$$F_{k+1}(i) = \min\{ F_k(j) + a(i,j) \}$$

$F_{k+1}(i)$：第k+1阶段状态　$F_k(j)$：第k阶段状态　$a(i,j)$：决策

（6）表上操作

动态规划一个显著的特点就是“表上操作”。也就是说，动态规划把所有的状态都保存在了一张表格之中，这表格称为状态变量，如引例中的F。状态变量可以是一维、二维，也可以是三维、四维等，这就要由阶段的划分和状态的特征来决定。

【基本原理】动态规划的基本原理

运用动态规划解题往往编程简单，但效率很高，如何使用动态规划，它的核心实质上是对问题进行状态的设计和阶段的划分。那么阶段的划分和状态描述能否符合动态规划的解题原理，是解决问题的根本所在。

1.无后效性

对于每个阶段的状态而言，如果确定了某一阶段的状态后，则在这一阶段以后过程的发展不再受这阶段以前各段状态的影响，如引例中，要求F[D]的最优值，只会跟F[Ci]的最优值有关，而跟F[Bi]的各个状态无关。换句话说，每个状态都是“**过去历史的一个完整总结**”，这就是无后效性。

2.最优性原理

对于每个阶段的决策而言，无论初始状态及初始决策如何，对于先前决策所形成的状态而言，其以后的所有决策应构成最优策略。这就是最优化原理。简言之，就是“**最优策略的子策略也是最优策略**”。如引例中的A点到D点的最短路径F[D]的问题，必须先求得A点至阶段3中的C1,C2,C3三点的最短路径F[C1]、F[C2]和F[C3]，这3个为子问题，而且也必须是最优的。以此类推，直至求出A至第2阶段中的两个点的最短路径F[B1],F[B2]。

【动态规划的解题步骤】

动态规划的解题步骤

由动态规划的概念及相关术语可知，对于动态规划类的题目，可以分为以下解题步骤：

①判断问题是否具有最优子结构性质，若不具备，则不能用动态规划；

②把问题分成若干个子问题（分阶段）；

③建立状态转移方程（递推公式）；

④找出边界条件；

⑤设定初始值；

⑥递推求解；

【动态规划的解题步骤】

按照阶段、状态和决策的层次关系，我们给出程序流程的一般形式：

```
void dp()
{  int i,j,k……;
   所有状态f的最小费用初始化：f[j]=0 (j=j0) or ∞(j!=j0)
   for(i=阶段最小值;i<=阶段最大值;i++)  //顺推每一个阶段
      for(j=状态最小值;j<=状态最大值;j++)//枚举阶段i的每一个状态
         for(k=决策最小值;k<=决策最大值;k++)
            //枚举阶段i中状态j可选择的每一种决策
           f[ji']=min{f[ji-1']+a[ji-1'][ ki-1']| ji-1'通过决策ki-1'可达ji'}
}
```

编程实现方式：

①递推

②记忆化搜索（一般在状态的拓朴顺序不很明确时使用）

【记忆化搜索与动态规划】

【记忆化搜索与动态规划】

- 记忆化搜索实际上是一种递归形式的动态规划，所以它无论从时间上还是从空间上来看都与后面的动态规划算法差不多。
- 记忆化搜索的特点是可以使动态规划算法看起来比较直观，它能让我们比较清楚的看出动态规划是如何具体用空间换取时间的。它的最大优点是，只要发现了搜索的重复计算之处，用它可以很快的优化算法。
- 在竞赛中，我们可以先编一个搜索程序，如果这个题目可以用动态规划解决，一般都可以采用记忆化搜索，这样我们可以在事先有一个搜索保本的情况下考虑动态规划，最后即便是没有想到怎样用记忆化搜索做，至少也会有一个搜索保本，比较划算。
- 记忆化搜索也有缺点，当动态规划空间不足需要用滚动数组时，它显然是无法用程序实现的，这就要我们在审题时能够准确估出该问题用动态规划大概所需空间，提前判断它是否能用记忆化搜索做。