#### 数字三角形

【题目描述】数字三角形（tower）

图2.2是一个层数为n（n≤1 000）的数字三角形，现有一只蚂蚁从顶层开始向下走，每走下一级时，可向左下方向或右下方向走。求走到底层后它所经过数字的总和的最大值。

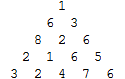


图2.2

【输入格式】

第一个整数为n，以下n行为各层的数字。

【输出格式】

一个整数，即最大值，保证不超过整型的最大范围。

【输入样例】

5

1

6 3

8 2 6

2 1 6 5

3 2 4 7 6

【输出样例】

23

【样例说明】

最大值＝1＋3＋6＋6＋7＝23

##### 记忆化递归搜索算法

如果用递归的方法，可以这样认为，要知道某一点所能达到的最大值，必须知道其左下或右下两点所能达到的最大值。依次推至最底层，获得最底层的值，再依次回算至顶点（1，1）。

三角形的数据存储一般使用二维数组，如图2.3所示。

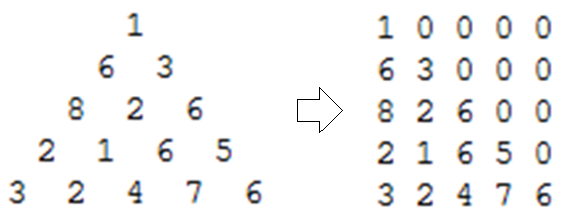


图2.3

|  |  |
| --- | --- |
|  | 为了提高程序的运行效率，可以把运行计算出来的值保存起来，下次用的时候直接拿来用，速度会快上很多。这就是所谓的记忆化搜索。 |

参考程序如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | //数字三角形 — 记忆化递归搜索  #include <bits/stdc++.h>  using namespace std;  const int MAXN=100;  int a[MAXN][MAXN],N,used[MAXN][MAXN];  int Fun(int i,int j)  {  if (i==N-1)  return a[i][j];  if (used[i][j]) //如果已有值，直接返回值  return used[i][j];  used[i][j]=max(Fun(i+1,j),Fun(i+1,j+1))+a[i][j];//求出的值保存在数组  return used[i][j];  }  int main()  {  scanf("%d",&N);  for (int i=0; i<N; i++)  for (int j=0; j<=i; j++)  scanf("%d",&a[i][j]);  printf("%d\n",Fun(0,0));  return 0;  } |

##### 动态规划算法

我们试着自顶向下的分析：从顶点13出发，是向顶点11方向走还是向顶点8方向走呢？这要比较从最底层到顶点11所可能获得的最大值和从最底层到顶点8所可能获得的最大值究竟哪个值更大，于是问题变成了分别从顶点11出发和从顶点8出发查找路径的子问题。

同理：查找从顶点11出发的最大路径值问题转换为从顶点12出发和从顶点7出发的查找路径最大值的子问题……

当走到倒数第二层，如顶点6，由于下面仅有2个顶点。此时可以直接选择一个大的数作为前进的方向。

如图2.4所示，方框里的数字是由下向上走到该顶点时能取得的最大值。

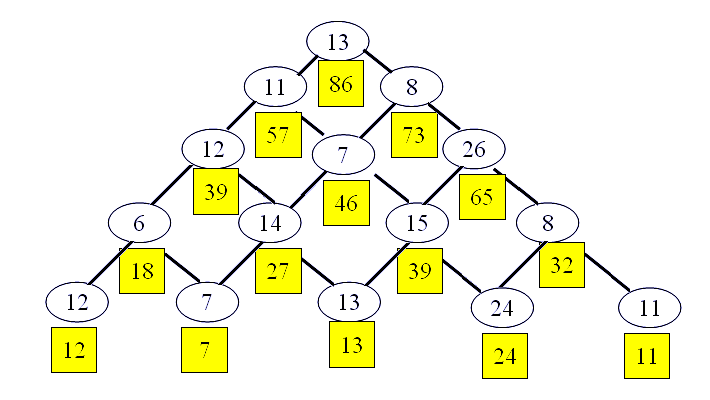


图2.4

设F[i][j]表示从底部走到第i行第j个数字时最大的和是多少。则有动态转移方程：

F[i][j]＝max{ F[i－1][j－1]，F[i－1][j] }＋Num[i][j]，其中Num[i][j]为第i行第j个数字的值。

参考代码如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | //数字三角形 - 动规  #include <bits/stdc++.h>  using namespace std;  int F[1001][1001];  int main()  {  int n;  scanf("%d",&n);  for(int i=1; i<=n; i++)  for(int j=1; j<=i; j++)  scanf("%d",&F[i][j]);  for(int i=n-1; i>=1; i--) //动规  for(int j=1; j<=i; j++)  F[i][j]+=max(F[i+1][j],F[i+1][j+1]);  printf("%d\n",F[1][1]);  return 0;  } |

##### 非完美算法

实际比赛中，如果对该类题无从下手，除了暴力搜索法外，还可以考虑以下两种方法：

（1）贪心算法，对于某些求最小值的问题，可以先用贪心算法得到较优解。然后开始搜索，搜索时只要计算结果比已求得的最小值大就剪枝。搜索到顶点时更新最小值。

（2）随机化算法＋卡时技术，即从最顶层开始，让向左走和向右走的概率相等。当然可以加入贪心思想，即让较大值方向的概率大一些。将这个过程重复若干次，总有可能得到正确答案。那么重复多少次合适呢？理论上是越多越好，但重复过多会造成程序超时，所以应在循环中时时统计程序运行时间，一旦程序运行即将超过时限时，即强制结束循环输出当前的最优结果。

随机化算法参考代码如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | //数字三角形 — 非完美算法  #include <bits/stdc++.h>  using namespace std;  #define maxn 100  int a[maxn][maxn],n;  int main()  {  srand(time(0));  clock\_t Time;  Time=clock(); //获取程序开始运行的时间  scanf("%d",&n);  for (int i=0; i<n; i++)  for (int j=0; j<=i; j++)  scanf("%d",&a[i][j]);  int result=0,t=0;  while(1)  {  if(t++%10==0) //此处调节获取时间的频率  if(clock()-Time>950) //如累计运行时间已接近1秒  break;  int j=0,v=a[0][j];  for (int k=1; k<n; k++)  {  int m1=a[k][j];  int m2=a[k][j+1];  int p=rand()%10;  if(m1>m2)  {  if (p>=4) //大值优先  v+=a[k][j];  else  v+=a[k][++j];  }  else  {  if (p<4)  v+=a[k][j];  else  v+=a[k][++j];  }  }  if (v>result)  result=v;  }  printf("%d\n",result);  return 0;  } |