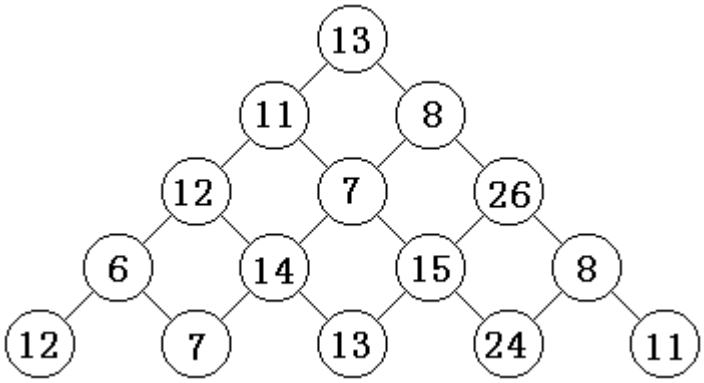
数字金字塔

描述 观察下面的数字金字塔。写一个程序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。每一步可以从当前点走到左下方的点也可以到达右下方的点。



在上面的样例中,从13到8到26到15到24的路径产生了最大的和86。

输入描述 第一个行包含R(1≤ R≤1000)，表示行的数目。后面每行为这个数字金字塔特定行包含的整数。

所有的被供应的整数是非负的且不大于100。

输出描述 单独的一行，包含那个可能得到的最大的和。

用例输入 1

```
5
13
11 8
12 7  26
6  14 15 8
12 7  13 24 11
```

用例输出 1

```
86
```

方法一. 搜索

问题要求的是从最高点按照规则走到最低点的路径的最大的权值和, 路径起点终点固 定,走法规则明确, 可以考虑用搜索来解决。

定义递归函数 `void Dfs(int  x, int y , int sum)`, 其中 `x,y` 表示当前已从 $(1,1)$ 走到 (x ,y), 目前已走路径上的权值和为 sum。

当 $x = N$ 时,到达递归出口, 如果 sum 比 ans 大, 则把 ans 更新为 sum;

否则向下一行两个位置行走, 即递归执行 `dfs(x + 1 , y , sum + a[x + 1][y])` 和 `dfs(x + 1, y + 1 , sum + a[x + 1][y + 1])`。

```
#include <iostream>
using namespace std;
```

```
int const N = 1005;
int dp[N][N] ;
int a[N][N];
int n,ans;
void dfs(int x,int y,int sum){//(x,y)表示已经从(1,1)走到了(x,y) 目前路径上权值和是sum
    if(x == n){//到达递归出口 更新ans
        ans = max(ans,sum);
        return ;
    }
    dfs(x+1,y,sum + a[x+1][y]);//向下
    dfs(x+1,y+1,sum + a[x+1][y+1]);//向右下
}
int main(){
    cin >> n;
    for(int i = 1 ; i <= n; i++){
        for(int j = 1; j <= i; j++){
            cin >> a[i][j];
        }
    }
    ans = 0;
    dfs(1,1,a[1][1]);
    cout << ans << endl;
    return 0;
}

```

该方法实际上是把所有路径都走了一遍, 由于每一条路径都是由 $N-1$ 步组成, 每一步 有"左"、"右"两种选择, 因此路径总数为 2^{N-1}, 所以该方法的时间复杂度为 $\mathrm{O}\left(2^{\mathrm{N}-1}\right)$, 超时。

方法二: 记忆化搜索

方法一之所以会超时, 是因为进行了重复搜索, 如样例中从 $(1,1)$ 到 $(3,2)$ 有"左右"和 "右左"两种不同的路径, 也就是说搜索过程中两次到达 $(3,2)$ 这个位置, 那么从 $(3,2)$ 走到终 点的每一条路径就被搜索了两次, 我们完全可以在第一次搜索 $(3,2)$ 到终点的路径时就记录 下 $(3,2)$ 到终点的最大权值和, 下次再次来到 $(3,2)$ 时就可以直接调用这个权值避免重复搜 索。我们把这种方法称为记忆化搜索。

记忆化搜索需要对方法一中的搜索进行改装。由于需要记录从一个点开始到终点的路 径的最大权值和, 因此我们重新定义递归函数 dfs。

定义 $dfs(x,y)$ 表示从 (x,y) 出发到终点的路径的最大权值和, 答案就是 $dfs(1,1)$ 。计 算 `dfs (x, y)` 时考虑第一步是向左还是向右,我们把所有路径分成两大类:

(1)第一步向左: 那么从 (x,y) 出发到终点的这类路径就被分成两个部分, 先从 (x,y) 到 $(x+1,y)$ 再从 $(x+1,y)$ 到终点, 第一部分固定权值就是 $a[x][y]$, 要使得这种情况的路径权 值和最大, 那么第二部分从 $(x+1,y)$ 到终点的路径的权值和也要最大, 这一部分与前面的 $dfs(x,y)$ 的定义十分相似, 仅仅是参数不同, 因此这一部分可以表示成 $dfs(x+1,y)$ 。综上, 第一步向左的路径最大权值和为 $a[x][y]+dfs(x+1,y)$;

(2)第一步向右: 这类路径要求先从 (x,y) 到 $(x+1,y+1)$ 再从 $(x+1,y+1)$ 到终点, 分析方法与上面一样, 这类路径最大权值和为 $a[x][y]+dfs(x+1,y+1)$;

为了避免重复搜索, 我们开设全局数组 $f[x][y]$ 记录从 (x,y) 出发到终点路径的最大权值和, 一开始全部初始化为 -1 表示未被计算过。在计算 $dfs(x,y)$ 时, 首先查询 $f[x][y]$, 如果 $f[x][y]$ 不等于 -1 , 说明之前已经$dfs(x,y)$被计算过, 直接返回 $f[x][y]$ 即可, 否则计 算出$dfs(x,y)$ 的值并存储在$f[x][y]$ 中。

```cpp
//记忆化搜索 记录从一个点开始到终点的路径的最大权值和
//时间复杂度是复杂度O(n^2)
#include <iostream>
#include <cstring>
using namespace std;
int const N = 1005;
int f[N][N] ;
int a[N][N];
int n,ans;
int dfs(int x,int y){//表示从(x,y)出发到终点的路径的最大权值和。
    if(f[x][y] == -1){//如果尚未计算过
        if(x == n) f[x][y] = a[x][y];
        else f[x][y] = a[x][y] + max(dfs(x+1,y),dfs(x+1,y+1));
    }
    return f[x][y] ;
}
int main(){
    cin >> n;
    for(int i = 1 ; i <= n; i++){
        for(int j = 1; j <= i; j++){
            cin >> a[i][j];
            f[i][j] = -1;
        }
    }
    dfs(1,1);
    cout << f[1][1] << endl;
    return 0;
}
```

由于 $F[x][y]$ 对于每个合法的 (x,y) 都只计算过一次, 而且计算是在O(1)内完成的, 因 此时间复杂度为 $O(n^2)$本题可以通过。

方法三: 动态规划(顺推法)

方法二通过分析搜索的状态重复调用自然过渡到记忆化搜索,而记忆化搜索本质上已 经是动态规划了。

下面我们完全从动态规划的算法出发换一个角度给大家展示一下动态规划的解题过程, 并提供动态规划的迭代实现法。

(1) 确定状态:

题目要求从 $(1,1)$ 出发到最底层路径最大权值和, 路径中是各个点串联而成, 路径起点固定, 终点和中间点相对不固定。因此定义 $dp[x][y]$ 表示从 $(1,1)$ 出发到达 (x,y) 的路径最大权值和。最终答案 $ans = max\{F[N][1], F[N][2], \ldots, F[N][N]\}$ 。

(2) 确定状态转移方程和边界条件:

不去考虑 $(1,1)$ 到 (x,y) 的每一步是如何走的, 只考息最后一步是如何走, 根据最后一步是向左还是向右分成以下两种情况:

- 向左: 最后一步是从 $(x-1,y)$ 走到 (x,y), 此类路径被分割成两部分,第一部分是从 $(1,1)$ 走到 $(x-1,y)$, 第二部分是从 $(x-1,y)$ 走到 (x,y), 要计算此类路径的最大权值和, 必须用到第一部分的最大权值和, 此部分问题的性质与 $dp[x][y]$ 的定义一样, 就是 $dp[x-1][y]$, 第二部分就是 $a[x][y]$, 两部分相加即得到此类路径的最大权值和为 $dp[x-1][y] + a[x][y]$;

- 向右: 最后一步是从 $(x-1,y-1)$ 走到 (x,y), 此类路径被分割成两部分, 第一部分是 从 $(1,1)$ 走到 $(x-1,y-1)$, 第二部分是从 $(x-1,y-1)$ 走到 (x,y), 分析方法如上。此类路径 的最大权值和为 $dp[x-1][y-1]+a[x][y]$

  $dp[x][y]$ 的计算需要求出上面两种情况的最大值。综上, 得到状态转移方程如下:

  $dp[x][y]=max\{dp[x-1][y-1],dp[x-1][y]\}+a[x,y]$

  与递归关系式还需要递归终止条件一样, 这里我们需要对边界进行处理, 以防无限递归 下去。观察发现计算 $dp[x][y]$ 时需要用到 $dp[x-1][y-1]$ 和 $dp[x-1,y]$, 是上一行的元素, 随着递推的深人, 最终都要用到第一行的元素 $dp[x-1][y-1]$, $dp[x-1][y-1]$的计算不能再使用状态转 移方程来求, 而是应该直接赋予一个特值 $a[1][1]$ 。这就是边界条件。

  综上得:

  状态转移方程: $dp[x][y]=\max\{dp[x-1][y-1],dp[x-1][y]\}+a[x,y]$

  边界条件: `dp[1][1]=a[1][1]`

  现在让我们来分析一下该动态规划的正确性,分析该解法是否满足使用动态规划的两 个前提:

- 最优化原理 (最优子结构性质): 这个在分析状态转移方程时已经分析得比较透彻, 明 显是符合最优化原理的;
- 无后效性: 状态转移方程中, 我们只关心 $dp[x-1][y-1]$ 与 $dp[x-1][y]$ 的值, 计算 $dp[x-1][y-1]$ 时可能有多种不同的决策对应着最优值, 选哪种决策对计算 $dp[x][y]$ 的决策 没有影响. $dp[x-1][y]$ 也是一样。这就是无后效性。

(3) 程序实现 :

由于状态转移方程就是递归关系式, 边界条件就是递归终止条件, 所以可以用递归来完 成, 递归存在重复调用, 利用记忆化可以解决重复调用的问题, 方法二已经讲过。记忆化实 现比较简单, 而且不会计算无用状态, 但递归也会受到"栈的大小"和"递推 + 回归执行方式" 的约束,另外记忆化实现调用状态的顺序是按照实际需求而展开, 没有大局规划, 不利于进一步优化。

这里介绍一种选代法。与分析边界条件方法相似, 计算 $dp[x][y]$ 用到状态 $dp[x-1][y-1]$ 与 $dp[x-1][y]$, 这些元素在 $F[x][y]$ 的上一行, 也就是说要计算第 x 行的状态的值, 必 须要先把第 $x-1$ 行元素的值计算出来, 因此我们可以先把第一行元素 $F[1][1]$ 赋为`a[1][1]`。再从第二行开始按照行递增的顺序计算出每一行的有效状态即可。时间复杂度为$O(N^2)$。

```cpp
//动态规划，顺推
//时间复杂度是复杂度O(n^2)
#include <iostream>
#include <cstring>
using namespace std;
int const N = 1005;
int dp[N][N] ;//从(1,1)出发到达(x,y)的路径最大权值和
int a[N][N];
int n,ans;

int main(){
    cin >> n;
    for(int i = 1 ; i <= n; i++){
        for(int j = 1; j <= i; j++){
            cin >> a[i][j];
        }
    }
    dp[1][1] = a[1][1];
    for(int i = 2 ; i <= n; i++){
        for(int j = 1; j <= i; j++){
```

```
            dp[i][j] = max(dp[i-1][j],dp[i-1][j-1]) + a[i][j];
        }
    }
    for(int j = 1; j <= n; j++){//最后一行取最大值才是答案
        ans = max(ans,dp[n][j]);
    }

    cout << ans << endl;
    return 0;
}
```

方法四: 动态规划(逆推法)

【算法分析】

(1) 贪心法往往得不到最优解: 本题若采用贪心法,则: $13-11-12-14-13$,其和为 63 ; 但存在另一条路: $13-8-26-15-24$, 其和为 86 。

贪心法问题所在: 眼光短浅。

(2)动态规划求解 : 动态规划求解问题的过程归纳为: 自顶向下分析,自底向上计算。 其基本方法是:

划分阶段: 按三角形的行划分阶段,若有 n 行, 则有 $n-1$ 个阶段。

A. 从根结点 13 出发, 选取它的两个方向中的一条支路, 当到倒数第二层时, 每个结点其 后继仅有两个结点, 可以直接比较, 选择最大值为前进方向, 从而求得从根结点开始到底端 的最大路径。

B. 自底向上计算: (给出递推式和终止条件)

(1)从底层开始, 本身数即为最大数;

(2)倒数第二层的计算, 取决于底层的数据: $12+6=18, 13+14=27, 24+15=39, 24+8=32$;

(3)倒数第三层的计算,取决于底二层计算的数据: $27+12=39, 39+7=46, 39+26=65$;

(4)倒数第四层的计算, 取决于底三层计算的数据 : $46+11=57, 65+8=73$;

(5)最后的路径: $13-8-26-15-24$ 。

C. 数据结构及算法设计

(1)图形转化: 直角三角形, 便于搜索: 向下、向右

(2)定义三个数组，$a[x][y]$ 表示x行y列的结点本身数据，`dp[x][y]`表示从底层走到(x,y)能获得的最大值，`dir[x][y]`表示上一步走到(x,y)的方向: 一 0 向下, 1 向右;

(3)算法:
数组初始化, 输人每个结点值及初始的最大路径、前进方向为 0 ;

从倒数第二层开始向上一层求最大路径, 共循环 $n-1$ 次;

从顶向下输出路径: 究竟向下还是向右取决于方向数组的值。

```
//动态规划，倒推 记录最大值的路径
//时间复杂度是复杂度O(n^2)
#include <iostream>
#include <cstring>
using namespace std;
int const N = 1005;
```

```cpp
int dp[N][N] ;//从底层出发到达(x,y)的路径最大权值和
bool dir[N][N];//表示前进方向，0表示向下，1表示向右下
int a[N][N];
int n,ans;

int main(){
    cin >> n;
    for(int i = 1 ; i <= n; i++){
        for(int j = 1; j <= i; j++){
            cin >> a[i][j];
        }
    }
    for(int j = 1; j <= n; j++){//最后一行
        dp[n][j] = a[n][j];
    }
    for(int i = n - 1 ; i >= 1; i--){//从最后一层向上倒推
        for(int j = 1; j <= i; j++){
            if(dp[i+1][j+1] > dp[i+1][j]) {
                dp[i][j] = dp[i+1][j+1] + a[i][j];
                dir[i][j] = 1;
            }else{
                dp[i][j] = dp[i+1][j] + a[i][j];
                dir[i][j] = 0;
            }
        }
    }

    cout << dp[1][1] << endl;

    /*输出路径
    int j = 1;
    for(int i = 1; i < n; i++){
        cout << a[i][j] << "->" ;
        j = j + dir[i][j];
    }
    cout << a[n][j] << endl;
    */
    return 0;
}
```