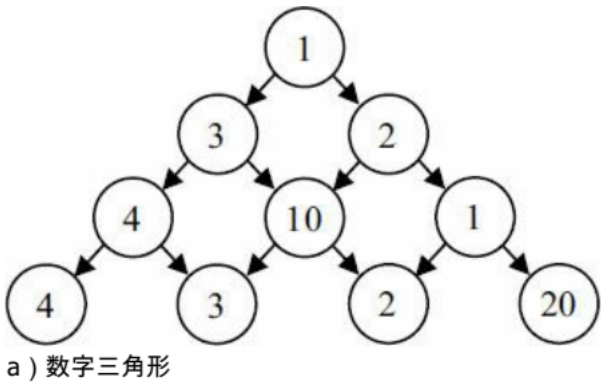
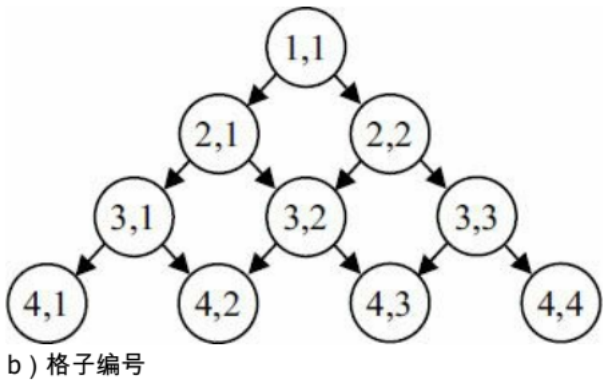
最简单的动归（数字三角形）

数字三角形问题。 有一个由非负整数组成的三角形， 第一行只有一个数， 除了最下行之外每个数的左下方和右下方各有一个数.



（a）数字三角形



（b）格子编号

从第一行的数开始， 每次可以往左下或右下走一格， 直到走到最下行， 把沿途经过的数全部加起来。 如何走才能使得这个和尽量大?

使用二维数组存储这个树如下：

0	1	2	3	4
1	1	0	0	0
2	3	2	0	0
3	4	10	1	0
4	4	3	2	20

如果用回溯法求出所有可能的路线， 就可以从中选出最优路线。 但和往常一样， 回溯法的效率太低： 一个n层数字三角形的完整路线有2^{n-1}条， 当n很大时回溯法的速度将让人无法忍受。

为了得到高效的算法， 需要用抽象的方法思考问题： 把当前的位置（i， j）看成一个状态(还记得吗？)，然后定义状态（i， j）的指标函数 d(i， j) 为从格子（i， j）出发时能得到的最大和(包括格子（i， j）本身的值)。 在这个状态定义下， 原问题的解是 d(0， 1) 。

下面看看不同状态之间是如何转移的。 从格子（i, j）出发有两种决策。 如果往左走， 则走到（i+1 ， j）后需要求**从**（i+1 ， j）**出发后能得到的最大和**这一问题， 即 d(i+1, j) 。 类似地， 往右走之后需要求解 d(i+1 ， j+1) 。 由于可以在这两个决策中自由选择， 所以应选择 d(i+1, j) 和 d(i+1, j+1) 中较大的一个。 换句话说， 得到了所谓的状态转移方程：

$$d(i,j) = a(i,j) + max\{d(i+1,j),\ d(i+1,j+1)\}$$

如果往左走， 那么最好情况等于（i， j）格子里的值 a(i， j) 与**从**（i+1 ， j）**出发的最大总和**之和， 此时需注意这里的**最大**二字。 如果连**从**（i+1, j）**出发走到底部**这部分的和都不是最大的， 加上 a(i， j) 之后肯定也不是最大的。 这个性质称为最优子结构（ optimal substructure） ，也可以描述成**全局最优解包含局部最优解。**

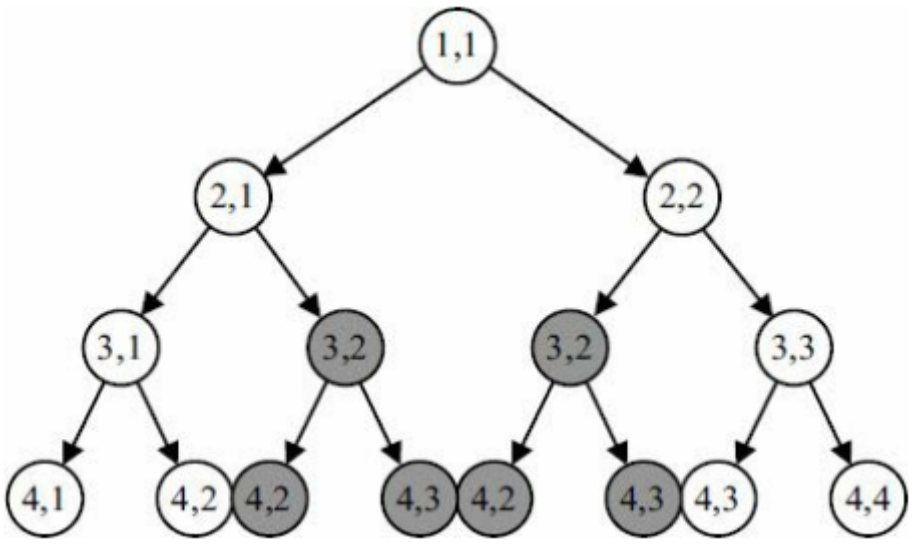
In [1]:
```
#include <string.h>
#include <stdio.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int dfstircount=0;   //记录基本计算次数
    int imax(int a,int b){
        return a>=b?a:b;
    }
    int tri[4][5]={0,1,0,0,0,0,3,2,0,0,0,4,10,1,0,0,4,3,2,20};
    int dfstir(int i, int j, int n){
        dfstircount++;
        return tri[i][j] + (i == n ? 0 : imax(dfstir(i+1,j,n),dfstir(i+1,j+1,n)));
    }
}
#endif
```

Out[1]:

In [2]:
```
#include <iostream>
using namespace std;
cout<<dfstir(0,1,4)<<" "<<dfstircount<<endl;
```

```
24 31
```

Out[2]: (std::basic_ostream<char, std::char_traits<char> >::__ostream_type &) @0x7fd370e88e60



黑色的状态被计算了多次！如何解决？改正重复计算是我们优化算法的一个常用的方法。

In [3]:
```
#include <string.h>
#include <stdio.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int dfstircount_1=0;   //记录基本计算次数
    int imax_1(int a,int b){
        return a>=b?a:b;
    }
    int tri_1[4][5]={0,1,0,0,0,0,3,2,0,0,0,4,10,1,0,0,4,3,2,20};
    int tri_1vis[4][5]={0};   //最直接的优化是记忆数组，将计算过的节点数值存起来，
    int dfstir_1(int i, int j, int n){
        if(tri_1vis[i][j]!=0) return tri_1vis[i][j];  //如果已经算过，直接返回
        dfstircount_1++;
        return tri_1vis[i][j]=tri_1[i][j] + (i == n ? 0 : imax_1(dfstir_1(i+1,j,n),dfstir_1(i+1,j+1,n)));  //将新算的节点值记住
    }
}
#endif
```

Out[3]:

In [4]:
```
#include <iostream>
using namespace std;
cout<<dfstir_1(0,1,4)<<" "<<dfstircount_1<<endl;
```

```
24 17
```

Out[4]: (std::basic_ostream<char, std::char_traits<char> >::__ostream_type &) @0x7f3a83c7ae60

加入记忆数组，计算次数明显减少了。这还是树比较小的情况，如果树很复杂，那么优化效果更明显。

这有一个很牛的名字叫记忆化搜索。

通过观察发现，每一层的每个节点和只与下面的两个节点和有关。所以完全可以从下向上一层一层的递推计算。

In [1]:
```
#include <string.h>
#include <stdio.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int tircount=0;   //记录基本计算次数
    int imax_2(int a,int b){
        return a>=b?a:b;
    }
    int tri_2[4][5]={0,1,0,0,0,0,3,2,0,0,0,4,10,1,0,0,4,3,2,20};
    int tri_2vis[4][5]={0};

    int tritui(){
        int i_2, j_2, n_2=3;
        for(j_2=1; j_2<=n_2+1;j_2++){   //将最下面一行数据存入vis数组
            tircount++;
            tri_2vis[n_2][j_2] = tri_2[n_2][j_2];
            //printf("%d ",tri_2vis[n_2][j_2]);
        }
        for(i_2 = n_2-1; i_2>=0; i_2--)
            for(j_2= 1; j_2<= i_2+1; j_2++){
                tircount++;
                tri_2vis[i_2][j_2] = tri_2[i_2][j_2]+imax_2(tri_2vis[i_2+1][j_2],tri_2vis[i_2+1][j_2+1]);
                //printf("%d ",tri_2vis[i_2][j_2]);
            }
        return tri_2vis[0][1];
    }
}
#endif
```

Out[1]:

In [2]:
```
#include <iostream>
using namespace std;
cout<<tritui()<<" "<<tircount<<endl;
```

```
24 10
```

Out[2]: (std::basic_ostream<char, std::char_traits<char> >::__ostream_type &) @0x7fcb2faa8e60

这个由下向上递推的方法是计算次数最少的。当数据量很大的时候，最爽。

我们也可以采用数学归纳法的思想从新思考这个问题：

1. 最下面一行的数据为归纳基础，每一个节点的和就是其本身。
2. 当知道从下向上算任意一层$k+1$的所有节点的和。
3. 推出k层所有节点的和。

思考解决问题的方法是将实际问题通过某种方式划分为自然数n上的一个递进过程，当n足够大或小的时候，问题可解（归纳基础）；然后研究n沿着某一个方向。逐步探索问题的解！

In [ ]: