# 程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

# 动态规划

## 几个例题

# 例一、最长公共子序列（POJ1458)

给出两个字符串，求出这样的一个最长的公共子序列的长度：子序列中的每个字符都能在两个原串中找到，而且每个字符的先后顺序和原串中的先后顺序一致。

# 最长公共子序列

## Sample Input

```
abcfbc abfcab
programming    contest
abcd    mnp
```

## Sample Output

```
4
2
0
```

# 最长公共子序列

输入两个串s1,s2,

设MaxLen(i,j)表示:

s1的左边i个字符形成的子串，与s2左边的j个字符形成的子串的最长公共子序列的长度(i,j从0开始算）

MaxLen(i,j)就是本题的“状态”

假定 len1 = strlen(s1),len2 = strlen(s2）

那么题目就是要求 MaxLen(len1,len2)

# 最长公共子序列

显然：

MaxLen(n,0) = 0 （n= 0…len1）

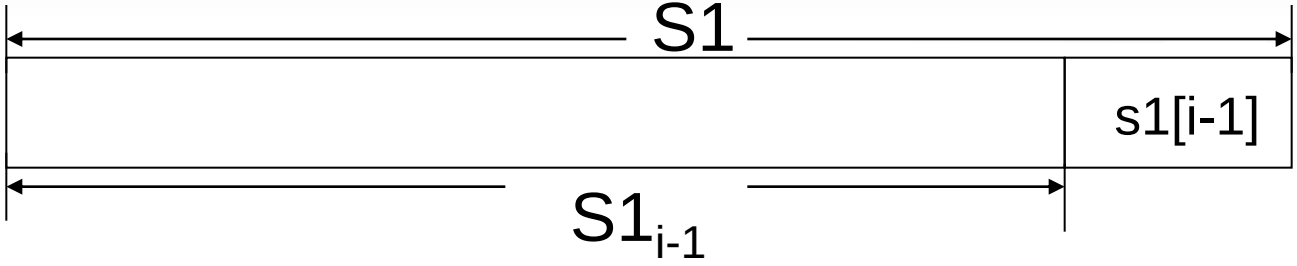
MaxLen(0,n) = 0 （n=0…len2）

递推公式：

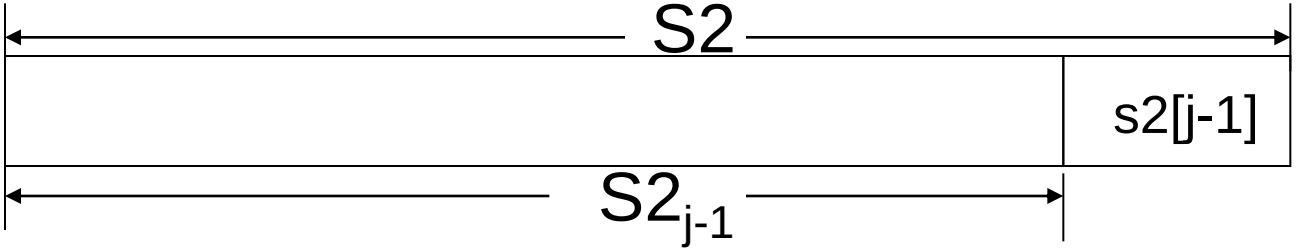
```
if ( s1[i-1] == s2[j-1] ) //s1的最左边字符是s1[0]
    MaxLen(i,j) = MaxLen(i-1,j-1) + 1;
else
    MaxLen(i,j) = Max(MaxLen(i,j-1),MaxLen(i-1,j) );
```

时间复杂度O(mn) m,n是两个字串长度

S1长度为 i




S2长度为 j




S1[i-1]!= s2[j-1]时，MaxLen(S1,S2)不会比MaxLen(S1,$S2_{j-1}$)

和MaxLen($S1_{i-1}$,S2)两者之中任何一个小，也不会比两者都大。

```
#include <iostream>
#include <cstring>
using namespace std;
char sz1[1000];
char sz2[1000];
int maxLen[1000][1000];
int main() {
        while( cin >> sz1 >> sz2 ) {
                int length1 = strlen( sz1);
                int length2 = strlen( sz2);
                int nTmp;
                int i,j;
                for( i = 0;i <= length1; i ++ )
                        maxLen[i][0] = 0;
                for( j = 0;j <= length2; j ++ )
                        maxLen[0][j] = 0;
```

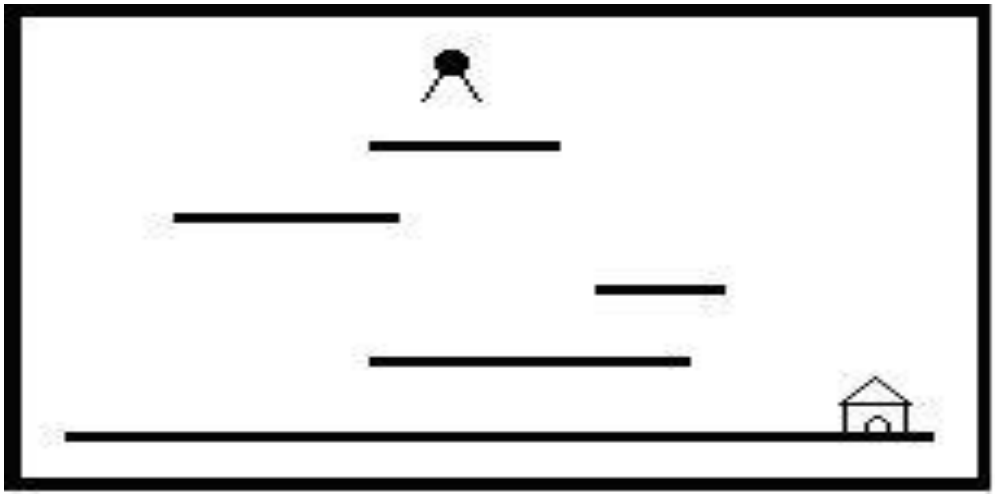
```
            for( i = 1;i <= length1;i ++ )  {
                    for( j = 1; j <= length2; j ++ ) {
                            if( sz1[i-1] == sz2[j-1] )
                                maxLen[i][j] =  maxLen[i-1][j-1] + 1;
                            else
                                maxLen[i][j] = max(maxLen[i][j-1],maxLen[i-1][j]);
                    }
            }
            cout <<  maxLen[length1][length2] << endl;
    }
    return 0;
}
```

# 活学活用

- 掌握递归和动态规划的思想，解决问题时灵活应用

# 例二、 Help Jimmy(POJ1661)

"Help Jimmy" 是在下图所示的场景上完成的游戏：



# Help Jimmy(POJ1661)

场景中包括多个长度和高度各不相同的平台。

地面是最低的平台，高度为零，长度无限。

Jimmy老鼠在时刻0从高于所有平台的某处开始下落，它的下落速度始终为1米/秒。当Jimmy落到某个平台上时，游戏者选择让它向左还是向右跑，它跑动的速度也是1米/秒。当Jimmy跑到平台的边缘时，开始继续下落。Jimmy每次下落的高度不能超过MAX米，不然就会摔死，游戏也会结束。

设计一个程序，计算Jimmy到地面时可能的最早时间。

# Help Jimmy(POJ1661)

**输入数据**

第一行是测试数据的组数t（0 <= t <= 20）。每组测试数据的第一行是四个整数N，X，Y，MAX，用空格分隔。N是平台的数目（不包括地面），X和Y是Jimmy开始下落的位置的横竖坐标，MAX是一次下落的最大高度。接下来的N行每行描述一个平台，包括三个整数，X1[i]，X2[i]和H[i]。H[i]表示平台的高度，X1[i]和X2[i]表示平台左右端点的横坐标。1 <= N <= 1000，-20000 <= X，X1[i]，X2[i] <= 20000，0 < H[i] < Y <= 20000（i = 1..N）。所有坐标的单位都是米。

Jimmy的大小和平台的厚度均忽略不计。如果Jimmy恰好落在某个平台的边缘，被视为落在平台上。所有的平台均不重叠或相连。测试数据保Jimmy一定能安全到达地面。

# Help Jimmy(POJ1661)

**输出要求**

对输入的每组测试数据，输出一个整数，Jimmy到地面时可能的最早时间。

**输入样例**

```
1
3 8 17 20
0 10 8
0 10 13
4 14 3
```

**输出样例**

```
23
```

# 解题思路

Jimmy跳到一块板上后，可以有两种选择，向左走，或向右走。
走到左端和走到右端所需的时间，是很容易算的。
如果我们能知道，以左端为起点到达地面的最短时间，和以右端为起点到达地面的最短时间，那么向左走还是向右走，就很容选择了。
因此，整个问题就被分解成两个子问题，即Jimmy所在位置下方第一块板左端为起点到地面的最短时间，和右端为起点到地面的最短时间。
这两个子问题在形式上和原问题是完全一致的。将板子从上到下从1开始进行无重复的编号(越高的板子编号越小，高度相同的几块板子，哪块编号在前无所谓)，那么，和上面两个子问题相关的变量就只有板子的编号。

不妨认为Jimmy开始的位置是一个编号为0，长度为0的板子，假设LeftMinTime(k)表示从k号板子左端到地面的最短时间，RightMinTime(k)表示从k号板子右端到地面的最短时间，那么，求板子k左端点到地面的最短时间的方法如下：

```
if ( 板子k左端正下方没有别的板子) {
    if( 板子k的高度 h(k) 大于Max)
        LeftMinTime(k) = ∞;
    else
        LeftMinTime(k) = h(k);
}
else if( 板子k左端正下方的板子编号是m )
    LeftMinTime(k) = h(k)-h(m) +
      Min( LeftMinTime(m) + Lx(k)-Lx(m),
          RightMinTime(m) + Rx(m)-Lx(k));
}
```

上面，h(i)就代表i号板子的高度，Lx(i)就代表i号板子左端点的横坐标，Rx(i)就代表i号板子右端点的横坐标。那么 h(k)-h(m)　当然就是从k号板子跳到m号板子所需要的时间，Lx(k)-Lx(m) 就是从m号板子的落脚点走到m号板子左端点的时间，Rx(m)-Lx(k)就是从m号板子的落脚点走到右端点所需的时间。

求RightMinTime(k)的过程类似。

不妨认为Jimmy开始的位置是一个编号为0，长度为0的板子，那么整个问题就是要求LeftMinTime(0)。

输入数据中，板子并没有按高度排序，所以程序中一定要首先将板子排序。

时间复杂度：

一共 n个板子，每个左右两端的最小时间各算一次 O(n)

找出板子一段到地面之间有那块板子，需要遍历板子 O(n)

总的时间复杂度$O(n^2)$

记忆递归的程序：

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
using namespace std;
#define MAX_N 1000
#define INFINITE 1000000
int t,n,x,y,maxHeight;
struct Platform{
    int Lx, Rx, h;
    bool operator < (const Platform & p2) const {
        return h > p2.h;
    }
};
```

```
Platform platForms[MAX_N + 10];
int leftMinTime[MAX_N + 10];
int rightMinTime[MAX_N + 10];
int L[MAX_N + 10];
int MinTime( int l, bool bLeft )
{
    int y = platForms[l].h;
    int x;
    if( bLeft )
        x = platForms[l].Lx;
    else
        x = platForms[l].Rx;
    int i;
```

```
for(  i = l + 1;i <= n;i ++ ) {
        if( platForms[i].Lx <= x && platForms[i].Rx >= x)
                break;
}
if( i <= n ) {
        if( y - platForms[i].h > maxHeight )
                return INFINITE;
}
else {
        if( y > maxHeight )
                return INFINITE;
        else
                return y;
}
```

```
    int nLeftTime = y - platForms[i].h + x - platForms[i].Lx;
    int nRightTime = y - platForms[i].h  +  platForms[i].Rx - x;
    if( leftMinTime[i] == -1 )
            leftMinTime[i] = MinTime(i,true);
    if( L[i] == -1 )
            L[i] = MinTime(i,false);
    nLeftTime += leftMinTime[i];
    nRightTime += L[i];
    if( nLeftTime < nRightTime )
            return nLeftTime;
    return nRightTime;
}
```

```
int main()  {
    scanf("%d",&t);
    for( int i = 0;i < t; i ++ ) {
            memset(leftMinTime,-1,sizeof(leftMinTime));
            memset(L,-1,sizeof(rightMinTime));
            scanf("%d%d%d%d",&n, &x, &y, &maxHeight);
            platForms[0].Lx = x;            platForms[0].Rx = x;
            platForms[0].h = y;
            for( int j = 1; j <= n; j ++ )
                scanf("%d%d%d",&platForms[j].Lx,& platForms[j].Rx, & platForms[j].h);
            sort(platForms,platForms+n+1);
            printf("%d\n", MinTime(0,true));
    }
    return 0;
}
```

递推的程序：

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
using namespace std;
#define MAX_N 1000
#define INFINITE 1000000
int t,n,x,y,maxHeight;
struct Platform{
    int Lx, Rx, h;
    bool operator < (const Platform & p2) const {
        return h > p2.h;
    }
};
```

```
Platform platforms[MAX_N + 10];
int leftMinTime[MAX_N + 10]; //各板子从左走最短时间
int rightMinTime[MAX_N + 10]; //各板子从右走最短时间
int main()
{
    scanf("%d",&t);
    while( t--) {
        scanf("%d%d%d%d",&n, &x, &y, &maxHeight);
        platforms[0].Lx = x;  platforms[0].Rx = x;        platforms[0].h = y;
        for( int j = 1; j <= n; j ++ )
            scanf("%d%d%d",&platforms[j].Lx,& platforms[j].Rx,
                & platforms[j].h);
        sort(platforms,platforms+n+1);
```

```
for( int i =  n ; i >= 0; -- i ) {
        int j;
        for(  j = i + 1; j <= n ; ++ j ) { //找i的左端的下面那块板子
                if( platforms[i].Lx <= platforms[j].Rx
                        && platforms[i].Lx >= platforms[j].Lx)
                        break;
        }
        if( j >n )  { //找不到
                if( platforms[i].h > maxHeight )
                        leftMinTime[i] = INFINITE;
                else
                        leftMinTime[i] = platforms[i].h;
        }
```

```
else {
        int y = platforms[i].h - platforms[j].h;
        if( y > maxHeight)
                leftMinTime[i] = INFINITE;
        else
                leftMinTime[i] = y +
            min(leftMinTime[j]+platforms[i].Lx-platforms[j].Lx,
                rightMinTime[j]+platforms[j].Rx-platforms[i].Lx);
}
for(  j = i + 1; j <= n ; ++ j ) { //找i的右端的下面那块板子
        if( platforms[i].Rx <= platforms[j].Rx
                && platforms[i].Rx >= platforms[j].Lx)
                break;
}
```

```
                if( j > n ) {
                        if( platforms[i].h > maxHeight )
                                rightMinTime[i] = INFINITE;
                        else        rightMinTime[i] = platforms[i].h;
                }
                else {
                        int y = platforms[i].h - platforms[j].h;
                        if( y > maxHeight)  rightMinTime[i] = INFINITE;
                        else
                            rightMinTime[i] = y +
                            min(leftMinTime[j]+platforms[i].Rx-platforms[j].Lx,
                                rightMinTime[j]+platforms[j].Rx-platforms[i].Rx);
                }
        }
        printf("%d\n", min(leftMinTime[0],rightMinTime[0]));
}
return 0;
}
```

# 例三、最佳加法表达式

有一个由1..9组成的数字串.问如果将m个加号插入到这个数字串中,在各种可能形成的表达式中，值最小的那个表达式的值是多少

# 解题思路

假定数字串长度是n，添完加号后,表达式的最后一个加号添加在第 i 个数字后面，那么整个表达式的最小值，就等于在前 i 个数字中插入 m – 1 个加号所能形成的最小值，加上第 i + 1到第 n 个数字所组成的数的值（i从1开始算）。

# 解题思路

设V(m,n)表示在n个数字中插入m个加号所能形成的表达式最小值，那么：

if m = 0

V(m,n) = n个数字构成的整数

else if n < m + 1

V(m,n) = ∞

else

V(m,n) = Min{ V(m-1,i) + Num(i+1,n) } ( i = m … n-1)

Num(i,j)表示从第i个数字到第j个数字所组成的数。数字编号从1开始算。此操作复杂度是O(j-i+1)

总时间复杂度：$O(mn^2)$.

# 例四、滑雪(百练1088)

Michael喜欢滑雪百这并不奇怪， 因为滑雪的确很刺激。

可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，

你不得不再次走上坡或者等待升降机来载你。

Michael想知道载一个区域中最长的滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子

1 2 3 4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为24-17-16-1。当然25-24-23-...-3-2-1更长。事实上，这是最长的一条。输入输入的第一行表示区域的行数R和列数C(1 <= R,C <= 100)。下面是R行，每行有C个整数，代表高度h，0<=h<=10000。输出输出最长区域的长度。

输入

输入的第一行表示区域的行数R和列数C
(1 <= R,C <= 100)。下面是R行，每行有C个整数，
代表高度h，0<=h<=10000。

输出

输出最长区域的长度。

样例输入

```
5 5
1  2  3  4  5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

样例输出

```
25
```

# 解题思路

L(i,j)表示从点(i,j)出发的最长滑行长度。

一个点(i,j)，如果周围没有比它低的点，L(i,j) = 1

否则

递推公式：L(i,j) 等于(i,j)周围四个点中，比(i,j)低，且L值最大的那个点的L值，再加1

复杂度：$O(n^2)$

# 解题思路

解法1）“人人为我”式递推

L(i,j)表示从点(i,j)出发的最长滑行长度。

一个点(i,j), 如果周围没有比它低的点，L(i,j) = 1

将所有点按高度从小到大排序。每个点的L值都初始化为1

从小到大遍历所有的点。经过一个点(i,j)时，用递推公式求L(i,j)

# 解题思路

解法2） “我为人人”式递推

L(i,j)表示从点(i,j)出发的最长滑行长度。
一个点(i,j), 如果周围没有比它低的点，L(i,j) = 1

将所有点按高度从小到大排序。每个点的L值都初始化为1

从小到大遍历所有的点。经过一个点(i,j)时，要更新他周围的，比它高的点的L值。例如：

```
if  H(i+1,j) > H(i,j)  // H代表高度
        L(i+1,j) = max(L(i+1,j),L(i,j)+1)
```