程序设计实习(II): 算法设计

第二十讲 动态规划

上一节课回顾

- 影响搜索效率的因素
  - 搜索对象(枚举什么)
  - 搜索顺序(先枚举什么，后枚举什么)
    - BFS：广度优先
    - DFS：深度优先
    - A*：估价函数最小优先
  - 剪枝(及早判断出不符合要求的情况)

3

问题提出：为什么要用动态规划

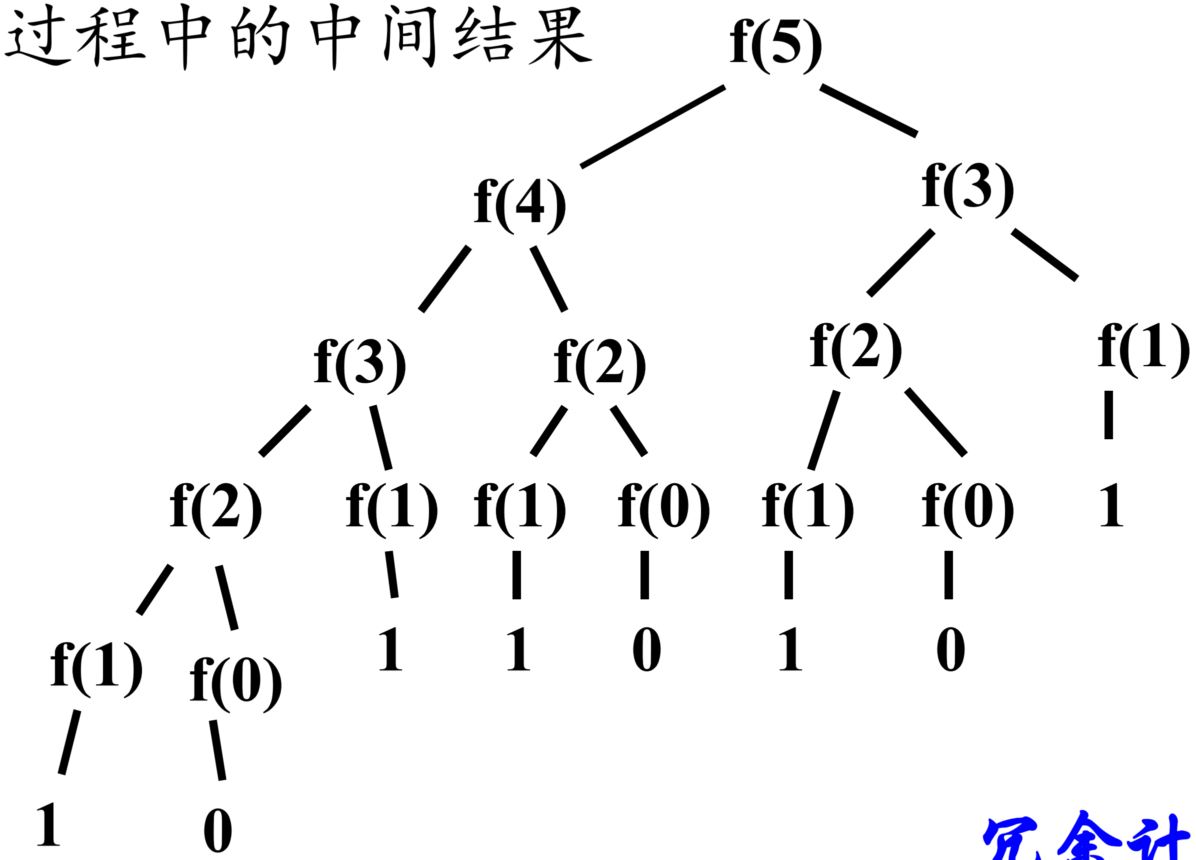
- BFS和DFS：树形递归存在冗余计算
- 例1：POJ 2753 Fibonacci数列

求 Fibonacci数列的第n项

```
int f(int n){
    if(n==0 || n==1) return n;
    return f(n-1)+f(n-2);
}
```

树形递归存在冗余计算

为了除去冗余，需要从已知条件开始计算，并记录计算过程中的中间结果



冗余计算!

树形递归存在冗余计算

去除冗余:

```
int f[n+1];
f[1]=f[2]=1;
int i;
for(i=3;i<=n;i++)
    f[i] = f[i-1]+f[i-2];
cout << f[n] <<endl;
```

用空间换时间 → 动态规划

什么是动态规划?

- 动态规划是求解包含重叠子问题的最优化方法
  - 基本思想: 将原问题分解为相似的子问题
  - 在求解的过程中通过保存子问题的解求出原问题的解（注意:不是简单分而治之）
  - 只能应用于有最优子结构的问题（即局部最优解能决定全局最优解，或问题能分解成子问题来求解）
- 动态规划=记忆化搜索

动态规划是如何工作的

- 例2 POJ 1163 数字三角形

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

- 在上面的数字三角形中寻找一条从顶部到底边的路径，使得路径上所经过的数字之和最大
- 路径上的每一步都只能往左下或右下走。只需要求出这个最大和即可，不必给出具体路径
  - 三角形的行数大于1小于等于100
  - 数字为 0 - 99

POJ 1163 数字三角形问题

输入格式:

5 //三角形行数,下面是三角形

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

要求输出最大和

解题思路

- 算法一:递归
  - 设f(i, j) 为三角形上从点(i, j)出发向下走的最长路径，则
    f(i, j) = max(f(i+1, j), f(i+1, j+1))+D(i, j)
  - 要输出的就是f(0,0)即从最上面一点出发的最长路径

解题思路

- D(r, j)表示第r行第j个数字，
- MaxSum(r, j) 代表从第r行的第j个数字到底边的各条路径中，数字之和最大的那条路径的数字之和
  则本题是要求MaxSum(0,0)
  (假设行编号和一行内数字编号都从0开始)
- 从某个D(r, j)出发，显然下一步只能走D(r+1,j)或者D(r+1, j+1)，所以对于N行的三角形：

```
if ( r == N-1 ) //最底层
    MaxSum(r,j) = D(N-1,j)
else
    MaxSum(r,j) = D(r, j)+Max(MaxSum(r+1,j), MaxSum(r+1,j+1));
```

数字三角形的递归程序

```
#include <iostream>  using namespace std;
#define MAX 101
int n, triangle[MAX][MAX];
int longestPath(int i, int j);
void main(){
    int i, j;
    cin >> n;
    for(i=0; i<n; i++)
        for(j=0; j<=i; j++)
            cin >> triangle[i][j];
    cout << longestPath(0,0) << endl;
}
int longestPath(int i, int j){
    if(i==n) return 0;
    int x = longestPath(i+1, j);
    int y = longestPath(i+1, j+1);
    if(x<y) x=y;
    return x+triangle[i][j];
}
```

超时！！

为什么超时?

- 回答: 重复计算

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

如果采用递归的方法，深度遍历每条路径，存在大量重复计算，则时间复杂度为2^n，对于 $n = 100$，肯定超时

解题思路

- 一种可能的改进思想: 从下往上计算，对于每一点，只需要保留**从下面来的路径中最大的路径的和**即可
  - 因为在它上面的点只关心到达它的最大路径和，不关心它从那条路径上来的
- 问题: 有几种解法?
  - 从使用不同的存储开销角度分析

解法1

- 如果每算出一个MaxSum(r,j)就保存起来，则可以用$o(n^2)$时间完成计算

  因为三角形的数字总数是 n(n+1)/2

- 此时需要的存储空间是:
  - **int D[100][100];** //用于存储三角形中的数字
  - **int MaxSum[100][100];** //用于存储每个MaxSum(r,j)

7				
3	8			
8	1	0		
2	7	4	4	
4	5	2	6	5

MaxSum

30				
23	21			
20	13	10		
7	12	10	10	
4	5	2	6	5

```c
#define MAX_NUM 100
int D[MAX_NUM + 10][MAX_NUM + 10];
int N;
int MaxSum[MAX_NUM + 10][MAX_NUM + 10];
int main() {
        int i, j;
        scanf("%d", & N);
        for( i = 1; i <= N; i ++ )
            for( j = 1; j <= i; j ++ )
                scanf("%d", &D[i][j]);
        for(  j = 1; j <= N; j ++ )
            MaxSum[N][j] = D[N][j];
        for( i = N; i > 1; i -- )
            for( j = 1; j < i ; j ++ ) {
                if( MaxSum[i][j] > MaxSum[i][j+1] )
                    MaxSum[i-1][j] = MaxSum[i][j] + D[i-1][j];
                else
                    MaxSum[i-1][j] = MaxSum[i][j+1] + D[i-1][j];
        }
        printf("%d",  MaxSum[1][1]);
        return 0;
}
```

解法2

- 没必要用二维Sum数组存储每一个MaxSum(r,j), 只要从底层一行行向上递推

  那么只要一维数组Sum[100]即可, 即只要**存储一行**的MaxSum值就可以

- 比解法一改进之处在于节省空间，时间复杂度不变

```c
#define MAX_NUM 100
int D[MAX_NUM + 10][MAX_NUM + 10];
int N;
int aMaxSum[MAX_NUM + 10];
int main() {
        int i, j;
        scanf("%d", & N);
        for( i = 1; i <= N; i ++ )
            for( j = 1; j <= i; j ++ )
                scanf("%d", &D[i][j]);
        for(  j = 1; j <= N; j ++ )
            aMaxSum[j] = D[N][j];
        for( i = N; i > 1; i -- )
            for( j = 1; j < i; j ++ ) {
                if( aMaxSum[j] > aMaxSum[j+1] )
                    aMaxSum[j] = aMaxSum[j] + D[i-1][j];
                else
                    aMaxSum[j] = aMaxSum[j+1] + D[i-1][j];
        }
        printf("%d", aMaxSum[1]);
        return 0;
}
```

解法3

- 能否不用二维数组D[100][100]存储数字呢?
  - 存储开销： n×sizeof(int)
- 思路：倒过来考虑。前面的思路是从底往上最终算出MaxSum(0,0)。如果**从顶往下**算，最终算出每一个MaxSum(N-1,j)，然后再取最大的MaxSum(N-1,j)，不就是答案吗？此时递推公式为：

```
 if( r == 0 )
   MaxSum(r,j) = D[0][0];
 else
   MaxSum(r,j) = Max(MaxSum(r-1,j-1), MaxSum(r-1,j))+ D[r][j];
```

解法3

由于两重循环形式为：

```
for( r = 0 ; r < N ; r ++)
    for( j = 0 ; j < r+1 ; j ++ ) {
        ...........
}
```

r, j不断递增，所以实际上不需要D[100][100]数组，

每次从文件里读取下一个数字即可

而每一行的每个MaxSum(r,j)仍然都需要存储起来.

这样，只需要一个Sum[100]数组

(注：只需设一个临时存储变量，在计算MaxSum(r,j)后写入数组时，暂存MaxSum(r-1,j))

解法3

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

nTmp

Sum							...

i=0	0				
	7				
i=1	7	0			
	10	15			
i=2	10	15	0		
	18	16	15		
i=3	18	16	15	0	
	20	25	20	19	
i=4	20	25	20	19	0
	24	30	27	26	24

```
#include <stdafx.h>
#define MAX_NUM 100
int N;
int Sum[MAX_NUM+1];
int main() {
        int i, j;
        int nInput = 0, nTmp = 0;
        memset( Sum, 0, sizeof(Sum));
        scanf("%d", & N);
        scanf("%d", &Sum[0]);
        for( i = 1; i < N; i ++ ) {
            nTmp = 0;
            for( j = 0; j < i+1; j ++ ) {
              scanf("%d", &nInput);
              if (Sum[j]> nTmp)
                    { nTmp = Sum[j];  Sum[j] += nInput; }
              else { nInput += nTmp; nTmp = Sum[j]; Sum[j] = nInput; }
            }
        }
        nTmpSum = 0;
        for( i = 0; i < N; i ++ ) {
          if (Sum[i]> nTmpSum)
                nTmpSum = Sum[i];
        }
        printf("%d", nTmpSum);   return 0;
}
```

递归到动规的一般转化方法

- 原来递归函数有几个参数，就定义一个几维的数组
- 数组的下标是递归函数参数的取值范围
- 数组元素的值是递归函数的返回值
- 这样就可以从边界开始，逐步填充数组，相当于计算**递归函数值的逆过程**

动规解题的一般思路

1. 将原问题分解为子问题

- 把原问题分解为若干个子问题，子问题经常和原问题形式相似，有时甚至完全一样，只不过规模变小了
- 子问题的解一旦求出就会被保存，所以每个子问题只需求解一次

动规解题的一般思路

2. 确定状态

- 将和子问题相关的各个变量的一组取值，称之为一个状态
- 一个状态对应于一个或多个子问题
- 所谓某个状态下的“值”，就是这个状态所对应的子问题的解
- 所有状态的集合构成问题的“**状态空间**”

“状态空间”的大小，与用动态规划解决问题的**时间复杂度**直接相关

- 在数字三角形的例子里，一共有N×(N+1)/2个数字，所以这个问题的状态空间里一共就有N×(N+1)/2个状态
- 在该问题里每个状态只需要经过一次，且在每个状态上作计算所花的时间都是和N无关的常数

动规解题的一般思路

2. 确定状态

- 用动态规划解题，经常碰到的情况是:
- K个整型变量能构成一个状态（如数字三角形中的行号和列号这两个变量构成“状态”）
- 如果这K个整型变量的取值范围分别是N1, N2, ……Nk，那么，我们就可以用一个K维的数组array[N1] [N2]……[Nk]来存储各个状态的“值”
- 这个“值”未必就是一个整数或浮点数，可能是需要一个结构才能表示的，那么array就可以是一个结构数组。
- 一个“状态”下的“值”通常会是一个或多个子问题的解

动规解题的一般思路

3. 确定状态转移方程

- 定义出什么是“状态”，以及在该“状态”下的“值”后，就要找出不同的状态之间如何迁移

→即如何从一个或多个“值”已知的“状态”，求出另一个“状态”的“值”

- 状态的迁移可以用递推公式表示，此递推公式也可被称作“状态转移方程”
- 数字三角形的状态转移方程

$$anMaxSum[r][j] = \begin{cases} D[r][j], & r = N \\ Max\left(anMaxSum[r+1][j].anMaxSum[r+1][j+1]\right) + D[r][j], & otherwise \end{cases}$$

例题：最长上升子序列

- 一个数的序列b_i，当$b_1 < b_2 < ... < b_S$的时候，我们称这个序列是上升的。对于给定的一个序列$(a_1, a_2, ..., a_N)$，我们可以得到一些上升的子序列$(a_{i_1}, a_{i_2}, ..., a_{i_K})$，这里$1 <= i_1 < i_2 < ... < i_K <= N$
  - 比如，对于序列(1, 7, 3, 5, 9, 4, 8)，有它的一些上升子序列，如(1, 7), (3, 4, 8)等等。这些子序列中最长的长度是4，比如子序列(1, 3, 5, 8)或(1, 3, 4, 8)
- 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

例题:最长上升子序列

- **输入要求**
  - 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。
- **输出要求**
  - 最长上升子序列的长度。
- **输入样例**
  - 7
  - 1 7 3 5 9 4 8
- **输出样例**
  - 4

解题思路(1):找子问题

- 经过分析，发现 求以a_k（k=1, 2, 3…N）为终点的最长上升子序列的长度是个好的子问题
  - 这里把一个上升子序列中最右边的那个数，称为该子序列的“终点”
  - 虽然这个子问题和原问题形式上并不完全一样，但是只要这N个子问题都解决了，那么这N个子问题的解中，最大的那个就是整个问题的解

解题思路(2): 确定状态

- 上面所述的子问题只和一个变量相关，就是数字的位置。因此序列中数的位置k就是“状态”
  - 状态 k 对应的“值”，就是以a_k做为“终点”的最长上升子序列的长度
  - 这个问题的状态一共有N个

解题思路(3):找出状态转移方程

- 状态定义出来后，转移方程就不难想了。假定MaxLen (k)表示以a_k做为“终点”的最长上升子序列的长度，那么:
  - **MaxLen (1) = 1**
  - **MaxLen (k) = Max { MaxLen (i)： 1<i < k 且 $a_i < a_k$ 且 k≠1 } + 1**
- 这个状态转移方程的意思就是，MaxLen(k)的值，就是在a_k左边，“终点”数值小于a_k，且长度最大的那个上升子序列的长度再加1。因为a_k左边任何“终点”小于a_k的子序列，加上a_k后就能形成一个更长的上升子序列
- 实际实现的时候，可以不必编写递归函数，因为从 MaxLen(1)就能推算出MaxLen(2)，有了MaxLen(1)和MaxLen(2)就能推算出MaxLen(3)……

```
#include <stdio.h>
#include <memory.h>
#define MAX_N 1000
int b[MAX_N + 10];
int aMaxLen[MAX_N + 10];
main(){
        int N;
        scanf("%d", & N);
        for( int i = 1;i <= N;i ++ )
                scanf("%d", & b[i]);
        aMaxLen[1] = 1;
        for( i = 2; i <= N; i ++ ) {
                //每次求以第i个数为终点的最长上升子序列的长度
                int nTmp = 0;  //记录满足条件的，第i个数左边的上升子
                                //序列的最大长度
```

```
        for( int j = 1; j < i; j ++ ) { //察看以第j个数为终点的最
                                        //长上升子序列
            if( b[i] > b[j] ) {
                if( nTmp < aMaxLen[j] )
                    nTmp = aMaxLen[j];
            }
        }
        aMaxLen[i]  = nTmp + 1;
    }
    int nMax = -1;
    for( i = 1; i <= N; i ++ )
        if( nMax < aMaxLen[i])
            nMax = aMaxLen[i];
    printf("%d\n", nMax);
}
```

aMaxLen[i]的值，就是在b[i]左边，“终点”数值小于b[i]、且长度最大的那个上升子序列的长度再加1

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
  - 最长的公共子序列：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	Sample Output
abcfbc abfcab	4
programming contest	2
abcd mnp	0

算法1: 递归

- 设两个字符串分别是
  char str1[MAXL]; 长度是len1
  char str2[MAXL]; 长度是len2
- 设f(str1,len1,str2,len2)为str1和str2的最大公共子串的长度，则可以对两个字符串的最后一个字符的情况进行枚举：
  - 情况一： str1[len1-1] == str2[len1-1]，则f(str1,len1,str2,len2) = 1+ f(str1,len1-1,str2,len2-1)
  - 情况二： str1[len1-1] != str2[len1-1]，则f(str1,len1,str2,len2) = max(f(str1,len1-1,str2,len2), f(str1,len1,str2,len2-1))
  - 程序如下：

算法1:递归

```
#include <iostream>  using namespace std;
#include <string.h>
#define MAX 1000
char str1[MAX], str2[MAX];
int commonstr(int,int);
void main(){
    while(cin>> str1 >> str2){
        intlen1=strlen(str1);
        intlen2=strlen(str2);
        cout<<commonstr(len1-1,len2-1);
        cout<<endl;
    }
}
```

算法1:递归

```
int commonstr(int len1,int len2){
    if(len1==-1 || len2==-1) return 0;
    if(str1[len1]==str2[len2])
        return 1+commonstr(len1-1,len2-1);
    else{
        int tmp1=commonstr(len1-1,len2);
        int tmp2=commonstr(len1,len2-1);
        if(tmp1>tmp2)
            return tmp1;
        else
            return tmp2;
    }
}
```

超时！

算法2:动态规划

- 输入两个子串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
  - MaxLen(i, j) 就是本题的“状态”，定义一数组
- 假定 len1 = strlen(s1), len2 = strlen(s2）
  - 那么题目就是要求MaxLen(len1,len2)

算法2:动态规划

显然:

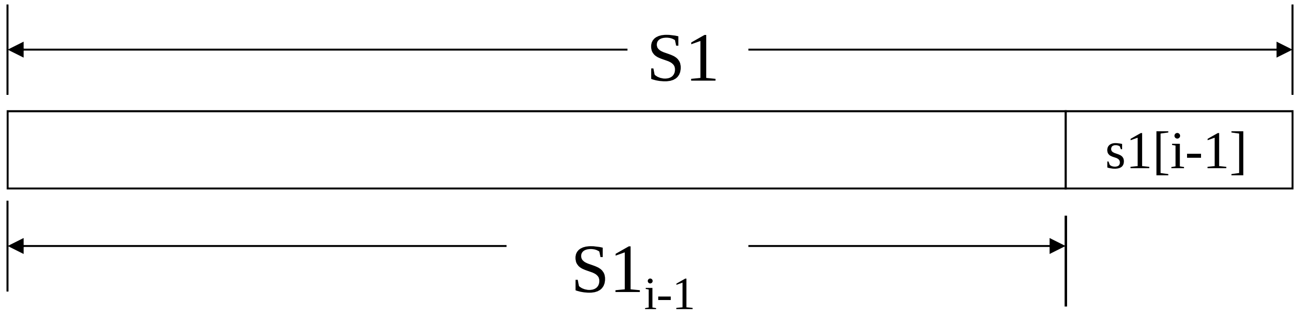
MaxLen(n,0) = 0 (n=0…len1)

MaxLen(0,n) = 0 (n=0…len2)

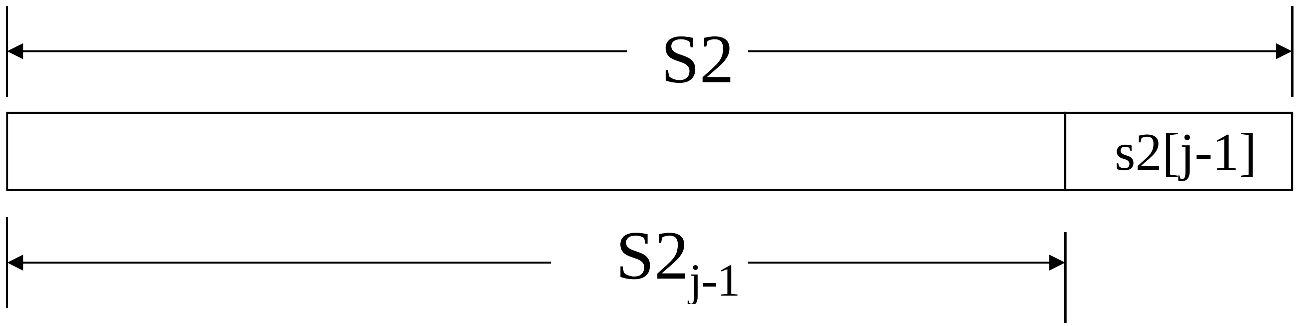
递推公式:

```
if ( s1[i-1] == s2[j-1] )
    MaxLen(i,j) = MaxLen(i-1,j-1) + 1;
else
    MaxLen(i,j) = Max(MaxLen(i,j-1), MaxLen(i-1,j));
```

S1长度为 i



S2长度为 j



MaxLen(S1,S2)不会比MaxLen(S1,$S2_{j-1}$) 和

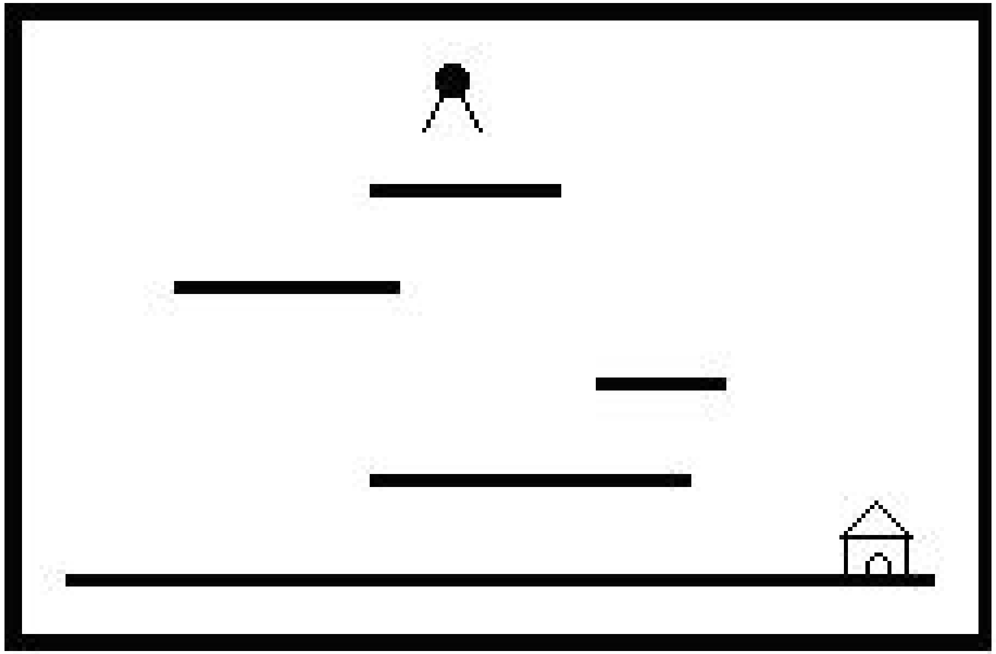
MaxLen($S1_{i-1}$,S2)都大，也不会比两者中任何一个小

```
#include <iostream>
#include <string.h>
using namespace std;
char sz1[1000];
char sz2[1000];
int anMaxLen[1000][1000];
int main(){
        while( cin >> sz1 >> sz2 ) {
                int nLength1 = strlen( sz1);
                int nLength2 = strlen( sz2);
                int nTmp;
                int i, j;
                for( i = 0; i <= nLength1; i ++ )
                    anMaxLen[i][0] = 0;
                for( j = 0; j <= nLength2; j ++ )
                    anMaxLen[0][j] = 0;
```

```
        for( i = 1;i <= nLength1;i ++ )  {
            for( j = 1; j <= nLength2; j ++ ) {
                if( sz1[i-1] == sz2[j-1] )
                    anMaxLen[i][j] = anMaxLen[i-1][j-1] + 1;
                else {
                    int nLen1 = anMaxLen[i][j-1];
                    int nLen2 = anMaxLen[i-1][j];
                    if( nLen1 > nLen2 )
                        anMaxLen[i][j] = nLen1;
                    else
                        anMaxLen[i][j] = nLen2;
                }
            }
        }
        cout <<  anMaxLen[nLength1][nLength2] << endl;
    }
}
```

例题: POJ 1661 Help Jimmy

- "Help Jimmy" 是在下图所示的场景上完成的游戏:



例题: POJ 1661 Help Jimmy

- 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
- 老鼠Jimmy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

- 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

- **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(1)

- 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，是很容易计算
- 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
- 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

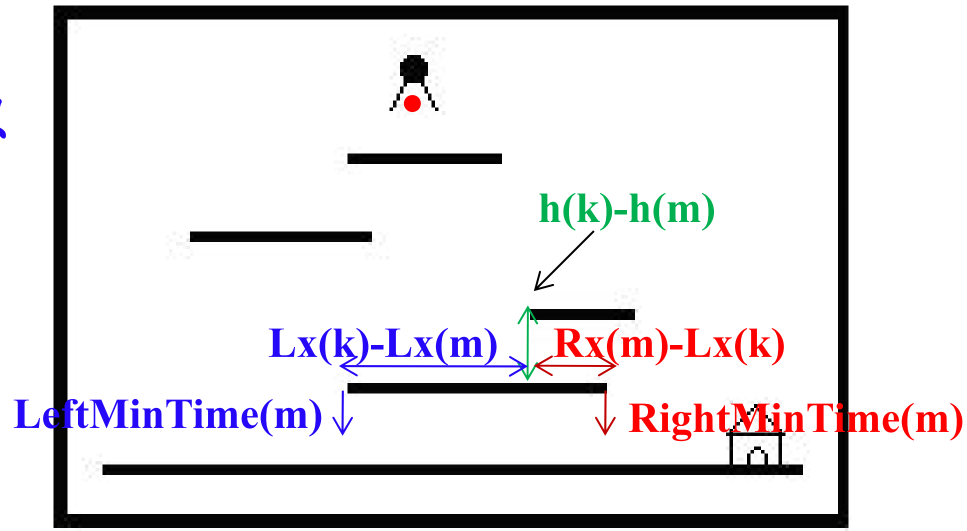
解题思路(2)

- 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：
  - 令: h(i)代表i号板子的高度, Lx(i)代表i号板子左端点的横坐标, Rx(i)代表i号板子右端点的横坐标
  - 则： 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LeftMinTime(k)的过程

```
if ( 板子k左端正下方没有别的板子) {
        if( 板子k的高度 h(k)>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) );
}
```

求RightMinTime(k)的过程类似



实现考虑

- 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
- LeftMinTime(k)和RightMinTime(k)可以用同一个过程来实现（用一个布尔变量来区分）
- 具体实现参考教材P231

Help Jimmy的非递归解法

```
#include<iostream>
using namespace std;
const int inf=0x7ffffff;
struct Board{
    int high;
    int left;
    int right;
};
Board board[1005];
int cmp(const void *a,const void *b) { // 按高度从大到小排序
    return (*(Board *)b).high-(*(Board *)a).high;
}
int min(int a, int b) {return a > b? b:a;}
int main(){
    int i, j, t, N, X, Y, MAX, ans;
    int f[1005][2];
    bool LEFT, RIGHT;
    cin>>t;
```

```
while(t--){
    cin>>N>>X>>Y>>MAX;
    ans=inf;
    board[0].left=board[0].right=X;
    board[0].high=Y;
    f[0][0]=f[0][1]=0;
    for(i=1;i<=N;i++){
        cin>>board[i].left>>board[i].right>>board[i].high;
        f[i][0]=f[i][1]=inf;
    }
    qsort(board,N+1,sizeof(board[0]), cmp);
    for(i=0;i<=N;i++){
         if(f[i][0]==f[i][1]&&f[i][0]==inf)
                continue;
         LEFT=RIGHT=0;
   for(j=i+1;j<=N&&board[i].high-board[j].high<=MAX
                &&!(LEFT&&RIGHT);j++){ //在可以掉下的范围内进行
   if(board[i].left>=board[j].left&&board[i].left<=board[j].right&&!LEFT) {
                          //如果左边可以掉下
             f[j][0]=min(f[j][0],f[i][0]+board[i].left-board[j].left);
             f[j][1]=min(f[j][1],f[i][0]+board[j].right-board[i].left);
             LEFT=1;
    }
```

```
    if(board[i].right>=board[j].left&&board[i].right<=board[j].right&&!RIGHT){
    //如果右边可以掉下
          f[j][0]=min(f[j][0], f[i][1]+board[i].right-board[j].left);
          f[j][1]=min(f[j][1], f[i][1]+board[j].right-board[i].right);
          RIGHT=1;
    }
  }
  if(board[i].high<=MAX&&!LEFT)
        ans=min(ans, f[i][0]);   //如果可以直接从左边掉在地上
  if(board[i].high<=MAX&&!RIGHT)
        ans=min(ans, f[i][1]);   //如果可以直接从右边掉在地上
  }
  cout<<ans+Y<<endl;
 }
 return 0;
}
//类似于数字三角形问题，可能还会有其他解法
```

总结：动规的要诀

- 用动态规划解题，关键是要找出“状态”，和在“状态”间进行转移的办法（即状态转移方程）

- 我们一般在动规的时候所用到的一些数组，也就是用来存储每个状态的最优值的

小 结

枚举 -------->搜索-------->动态规划

(系统化)　　(记忆化)

搜索的实现

- 方式1:递归-剪枝
  - 整个搜索过程中，最终状态始终不变
  - 不要考虑明显不能到达最终状态的路径
- 方式2:动态规划
  - 目的
    - 在搜索过程中，把计算的结果保留下来
    - 后面的搜索过程中，努力使用前面搜索过程的计算机结果，避免重复的计算
  - 方法
    - 把最终目标分解成一些相对简单的目标
    - 先实现这些相对简单的目标，在此基础上实现最终的目标
- 具体使用哪种方式？依情况而定
  - 没有什么重复的计算可以使用：递归，为保持简洁
  - 重复的计算占的比重很大：动态规划，为提高效率