2019年春

程序设计实习(II): 算法设计

第十七讲 动态规划1

刘家瑛

liujiaying@pku.edu.cn

课前多叽歪

- 由期中考试引发的各种血案
  - Case1
  - Case2
  - Case3

  …
- 引发的思考
- 如何在期末考试中越战越勇

主要内容

- 为什么？什么是动态规划
- 例题：数字三角形(POJ1163)
- 递归 → 动规的一般转化方法
- 动规解题的一般思路
- 例题：最长上升子序列
- 例题：最长公共子序列 (POJ1458)

问题提出：为什么要用动态规划

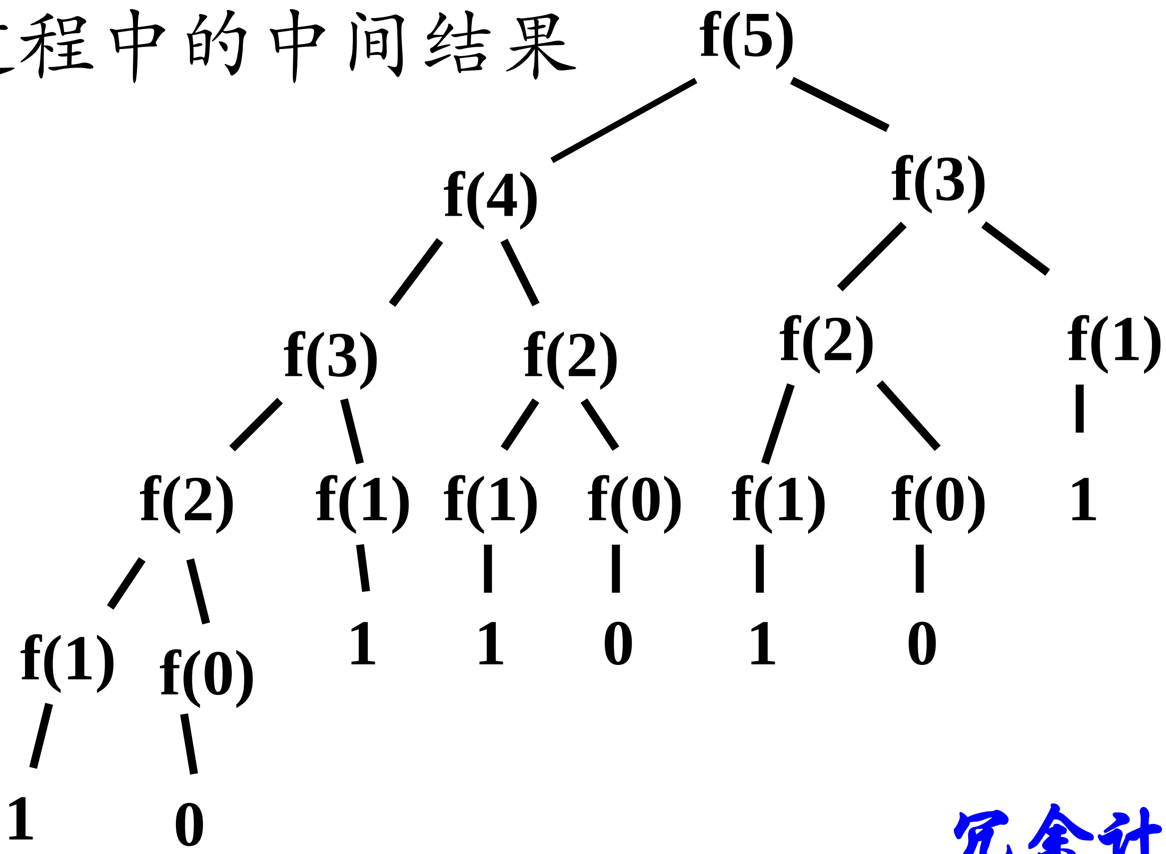
- 树形递归存在冗余计算
- 例1: (POJ 2753) Fibonacci数列

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(1,1)即从最上面一点出发的最长路径

解题思路

- **D(r, j)** -- 第r行第j个数字
- **MaxSum(r, j)** -- 从第r行第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

  * 本题要求**MaxSum(1,1)** (r, j从1开始算)
- 从某个D(r, j)出发，下一步只能走**D(r+1, j)** / **D(r+1, j+1)**，所以对于N行的三角形：

```
if ( r == N ) //最底层
    MaxSum(r, j) = D(N, 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>
#include <algorithm>
#define MAX 101
using namespace std;

int D[MAX][MAX];
int n;
int MaxSum(int i,  int j){
    if(i==n)
        return D[i][j];
    int x = MaxSum(i+1,  j);
    int y = MaxSum(i+1,  j+1);
    return max(x, y)+D[i][j];
}
```

```
int main(){
  int i,  j;
  cin >> n;
  for(i=1; i<=n; i++)
    for(j=1; j<=i; j++)
        cin >> D[i][j];
    cout << MaxSum(1, 1) << endl;
}
```

超时!!!

为什么超时?

- 回答: **重复计算**

$$
\begin{array}{ccccccccc}
 & & & & 7_1 & & & & \\
 & & & 3_1 & & 8_1 & & & \\
 & & 8_1 & & 1_2 & & 0_1 & & \\
 & 2_1 & & 7_3 & & 4_3 & & 4_1 & \\
4_1 & & 5_4 & & 2_6 & & 6_4 & & 5_1
\end{array}
$$

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

改 进

- 如果每算出一个MaxSum(r, j)就保存起来，下次用到其值的时候直接取用

→ 则可免去重复计算

- 那么可以用$O(n^2)$时间完成计算

因为三角形的数字总数是 $n(n+1)/2$

记忆递归型动规程序

```
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];    int n;
int maxSum[MAX][MAX];
int MaxSum(int i,  int j){
    if( maxSum[i][j] != -1 )
        return maxSum[i][j];
    if(i==n) maxSum[i][j] = D[i][j];
    else {
      int x = MaxSum(i+1, j);
      int y = MaxSum(i+1, j+1);
      maxSum[i][j] = max(x, y)+ D[i][j];
    }
    return maxSum[i][j];
}
```

```
int main(){
    int i, j;
    cin >> n;
    for(i=1;i<=n;i++)
    for(j=1;j<=i;j++) {
        cin >> D[i][j];
        maxSum[i][j] = -1;
    }
    cout << MaxSum(1, 1) << endl;
}
```

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

解法一：递归转成递推

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

4	5	2	6	5

解法一：递归转成递推

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

7				
4	5	2	6	5

解法一：递归转成递推

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

7	12			
4	5	2	6	5

解法一：递归转成递推

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

7	12	10		
4	5	2	6	5

解法一：递归转成递推

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

7	12	10	10	
4	5	2	6	5

解法一：递归转成递推

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

20				
7	12	10	10	
4	5	2	6	5

解法一：递归转成递推

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

20	13			
7	12	10	10	
4	5	2	6	5

解法一：递归转成递推

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

20	13	10		
7	12	10	10	
4	5	2	6	5

解法一：递归转成递推

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

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

25

“人人为我”递推型动规程序

```
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];   int n;
int maxSum[MAX][MAX];
int main()  {
    int i,  j;
    cin >> n;
    for( i=1; i<=n; i++)
        for( j=1; j<=i; j++)
            cin >> D[i][j];
    for( int i = 1; i <= n; ++ i )
         maxSum[n][i] = D[n][i];
    for( int i = n-1; i>= 1;  --i )
       for( int j = 1; j <= i; ++j )
           maxSum[i][j] = max(maxSum[i+1][j],  maxSum[i+1][j+1]) + D[i][j]
    cout << maxSum[1][1] << endl;
}
```

解法二

- 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

Vs. 只要**一维数组Sum[100]**，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

空间优化

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

4	5	2	6	5

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

→ 那么只要一维数组maxSum[100]即可
→ 即只要存储一行的MaxSum值就可以

空间优化

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

7	5	2	6	5

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

→ 那么只要一维数组maxSum[100]即可
→ 即只要存储一行的MaxSum值就可以

空间优化

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

7	12	2	6	5

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

→ 那么只要一维数组maxSum[100]即可
→ 即只要存储一行的MaxSum值就可以

空间优化

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

7	12	10	6	5

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

→ 那么只要一维数组maxSum[100]即可
→ 即只要存储一行的MaxSum值就可以

空间优化

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

7	12	10	10	5

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

→那么只要一维数组maxSum[100]即可
→即只要存储一行的MaxSum值就可以

空间优化

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

20	12	10	10	5

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

→ 那么只要一维数组maxSum[100]即可

→ 即只要存储一行的MaxSum值就可以

空间优化

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

20	13	10	10	5

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

→ 那么只要一维数组maxSum[100]即可
→ 即只要存储一行的MaxSum值就可以

空间优化

- 进一步考虑，连maxSum数组都可以不要

→ 直接用**D的第n行替代maxSum**即可

- 节省空间，时间复杂度不变

空间优化

```
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];
int n; int * maxSum;
int main(){
    int i,  j;
    cin >> n;
    for( i=1; i<=n; i++)
        for( j=1; j<=i; j++)
            cin >> D[i][j];
    maxSum = D[n];   //maxSum指向第n行
    for( int i = n-1; i>= 1;  --i )
        for( int j = 1; j <= i; ++j )
            maxSum[j] = max(maxSum[j], maxSum[j+1]) + D[i][j];
    cout << maxSum[1] << endl;
}
```

递归→动规的一般转化方法

- 递归函数有**n个参数**，就定义一个**n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
- 子问题都解决，原问题即解决
- 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**行号和列号**这两个变量构成"状态")

- 如果K个整型变量的取值范围分别是N1, N2, ……Nk, 那么就可以用一个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. 确定一些初始状态（边界状态）的值

- 以"数字三角形"为例，初始状态就是底边数字，值就是底边数字值

动规解题的一般思路

4. 确定状态转移方程

- 定义出什么是"状态"和在该"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" **("人人为我"递推型)**

- 状态的迁移可以用递推公式表示，该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**

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

能用动规解决的问题的特点

- 问题具有**最优子结构**性质
  - 如果问题的最优解所包含的子问题的解也是最优的
  - 就称该问题具有最优子结构性质
- **无后效性**
  - 当前的若干个状态值一旦确定，则此后过程的演变就只和这若干个状态的值有关
  - 和之前是采取哪种手段或经过哪条路径演变到当前的这若干个状态，没有关系

例题：最长上升子序列

- 一个数的序列b_i，当$b_1 < b_2 < \ldots < b_S$的时候，称这个序列是**上升**的。对于给定的一个序列$(a_1, a_2, \ldots, a_N)$，
  可以得到一些上升的子序列$(a_{i_1}, a_{i_2}, \ldots, a_{i_K})$，
  这里$1 <= i_1 < i_2 < \ldots < i_K <= N$
  - 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
    这些子序列中最长的长度是4，即子序列(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

- "求序列的前n个元素的最长上升子序列的长度"是个子问题, 但这样分解子问题, **不具有"无后效性"**
- 假设F(n) = x, 但可能有多个序列满足F(n) = x
  - 有的序列的最后一个元素比 a_n+1小, 则加上a_n+1就能形成更长上升子序列
  - 有的序列最后一个元素不比a_n+1小……以后的事情受如何达到状态n的影响, 不符合"无后效性"

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**

- 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

- 状态定义好后，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，则

MaxLen (1) = 1

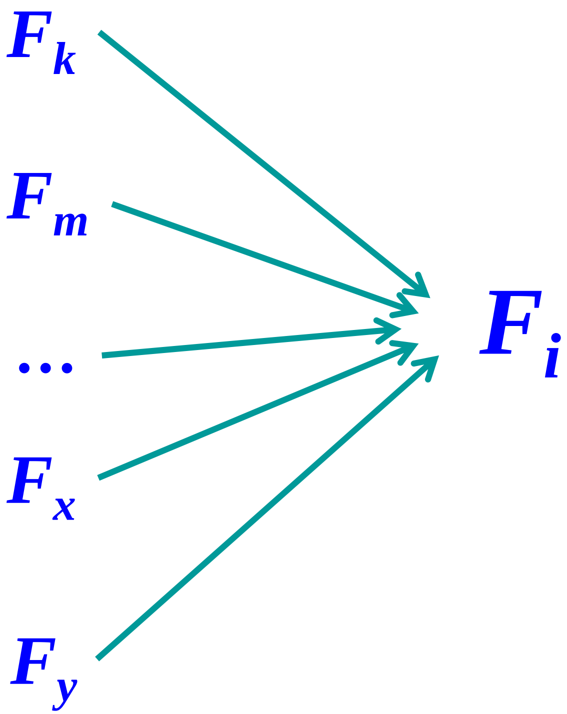
MaxLen (k) = Max { MaxLen (i): 1<i<k 且 $a_i < a_k$ 且 k≠1 } + 1

- 状态转移方程是，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)，…

"人人为我"递推型动归程序

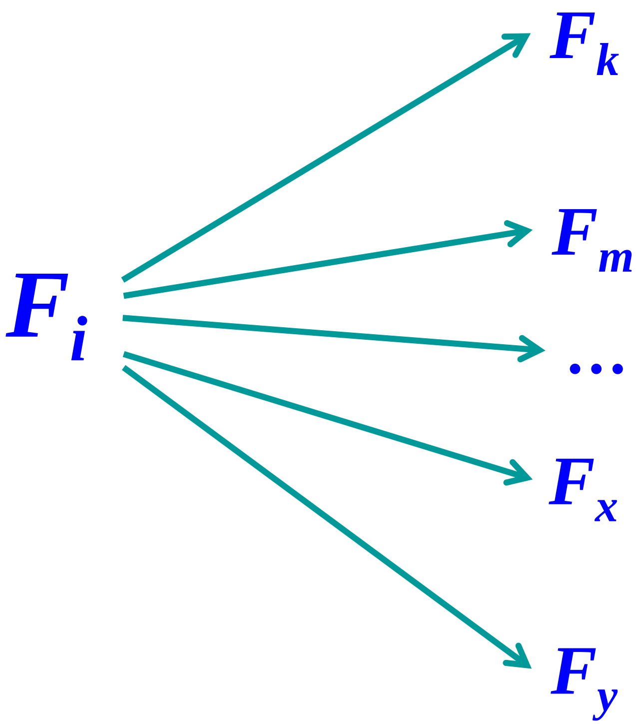
```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
const int  MAXN =1010;
int a[MAXN];   int maxLen[MAXN];
int main() {
    int N;    cin >> N;
    for( int i = 1;i <= N;++i) {
        cin >> a[i];
        maxLen[i] = 1;
    }
    for( int i = 2; i <= N; ++i) { //每次求以第i个数为终点的最长上升子序列的长度
        for( int j = 1; j < i; ++j)  //查看以第j个数为终点的最长上升子序列
            if( a[i] > a[j] )
                maxLen[i] = max(maxLen[i],  maxLen[j]+1);
    }
    cout << * max_element(maxLen+1, maxLen + N + 1 );
    return 0;
  //时间复杂度O(N^2)
```

"人人为我"递推型动归



状态i的值F_i由若干个值已知的状态值F_k, F_m, …, F_y推出,如求和,取最大值 …

"我为人人"递推型动归



状态i的值F_i在被更新(不一定是最终求出)的时候，依据F_i去更新(不一定是最终求出)和状态i相关的其他一些状态的值F_k，F_m，…，F_y

"我为人人"递推型动归

```
#include <cstring>
#include <algorithm>
using namespace std;
const int  MAXN =1010;
int a[MAXN];
int maxLen[MAXN];
int main() {
    int N;     cin >> N;
    for( int i = 1;i <= N;++i) {
        cin >> a[i];
        maxLen[i] = 1;
    }
    for( int i = 1; i <= N; ++i)
        for( int j = i + 1; j <= N; ++j )  //看看能更新哪些状态的值
            if( a[j] > a[i] )
                maxLen[j] = max(maxLen[j], maxLen[i]+1);
    cout << * max_element(maxLen+1, maxLen + N + 1 );
    return 0;
} //时间复杂度O(N^2)
```

人人为我:

```
for( int i = 2; i <= N; ++i)
    for( int j = 1; j < i; ++j)
        if( a[i] > a[j] )
            maxLen[i] =
                max(maxLen[i],  maxLen[j]+1);
```

动归的三种形式

1) 记忆递归型

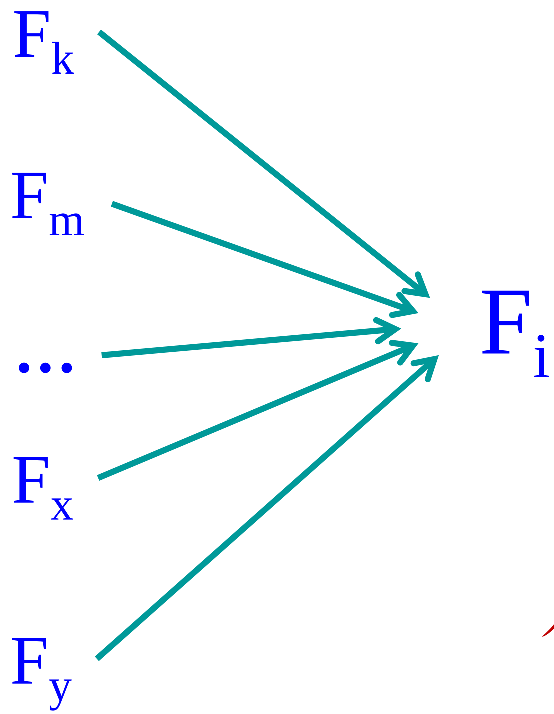
优点： 只经过有用的状态，没有浪费。递推型会查看一些没用的状态，有浪费

缺点： 可能会因递归层数太深导致爆栈，函数调用带来额外时间开销。总体来说，比递推型慢

2) "我为人人"递推型

没有什么明显的优势，有时比较符合思考的习惯。个别特殊题目中会比"人人为我"型节省空间

3) "人人为我"递推型



状态i的值F_i由若干个值已知的状态值F_k, F_m, …, F_y推出，如求和，取最大值……

在选取最优备选状态的值F_m, F_n, …, F_y时，有可能有好的算法或数据结构可以用来显著降低时间复杂度。

例：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

样例输入	样例输出
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[len2-1]，则f(str1, len1, str2, len2) = 1+ f(str1, len1-1, str2, len2-1)
  - **情况二**：str1[len1-1] != str2[len2-1]，则f(str1, len1, str2, len2) = max(f(str1, len1-1, str2, len2), f(str1, len1, str2, len2-1))

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:

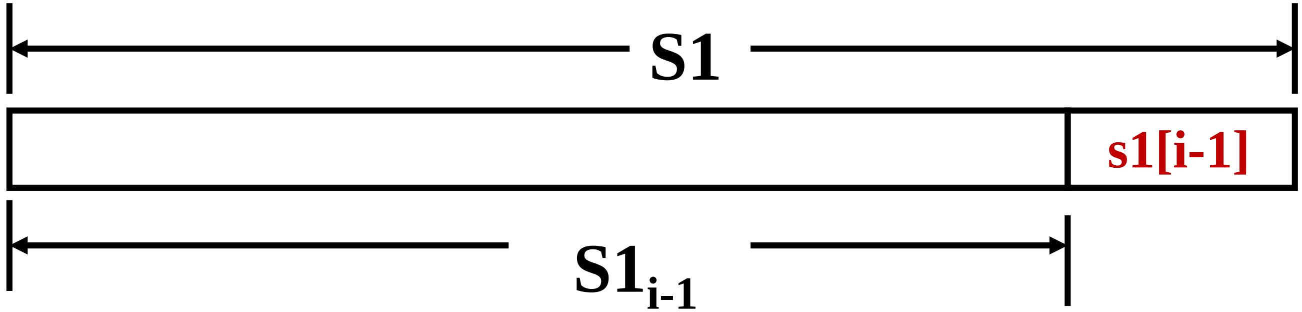
MaxLen(n, 0) = 0 (n=0…len1)

MaxLen(0, n) = 0 (n=0…len2）

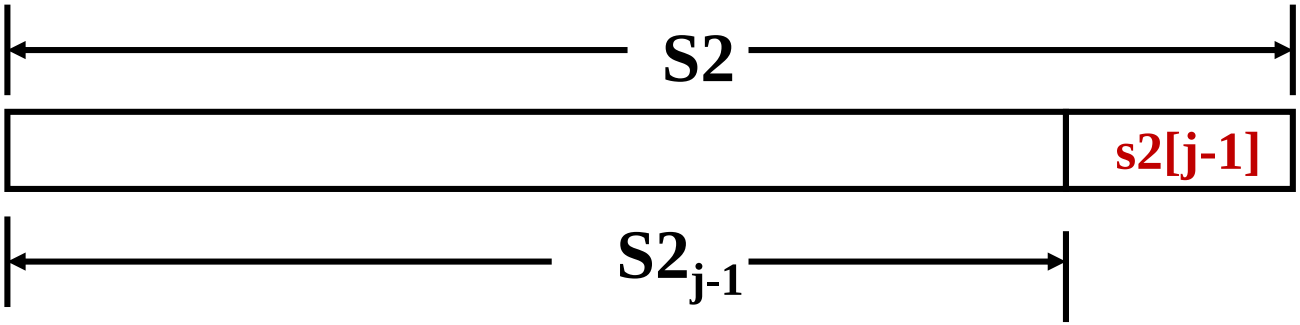
递推公式:

```
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



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 <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

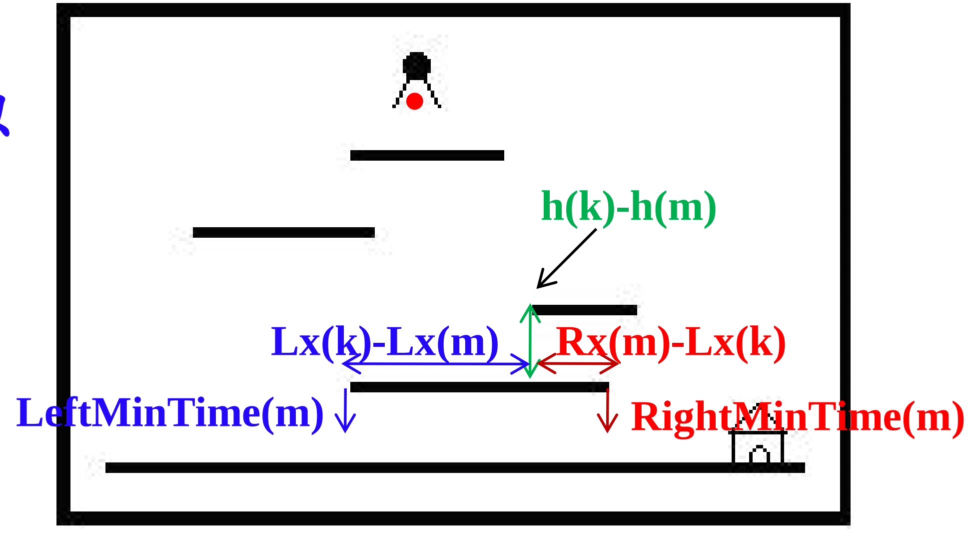
解题思路(2)

- 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
- 则求**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)** --从k号板子跳到m号板子所需要的时间
    Lx(k)-Lx(m) --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) --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

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;
}
```


77

```
    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;
}
```


78

```
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 )  { //板子左端正下方没有别的板子
                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;
}
```


83

```
        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;
}
```