2019年春

程序设计实习(II): 算法设计

第十八讲 动态规划2

刘家瑛

liujiaying@pku.edu.cn

课前多叽歪

- "习题课 讲解"报名
  - 时间: 下周三3-4节 习题课
  - 每个人6-7min
  - 本周日下午18点前交PPT
- 下周日上机时间会进行模考
- 期末考试 6月21日下午14点
  - 具体考试机房, 6月20日见教学网/邮件/微信群通知
  - 具体考试机位, 考试当天见机房门口/微信群

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**
- 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**

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：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，使得所形成的算术表达式的**值最小**

5

最佳加法表达式

- 添完加号后, 表达式的最后一定是个数字串
- 从这里入手, 不难发现:
  - 前一状态: 在**前i个字符**中插入**(m-1)**个加号
    (这里的 **i是当作决策在枚举**)
  - 然后 **i+1到最后一位** 一定是整个没有被分割的数字串
  - 第m个加号就添在i与i+1个数字之间

  →这样就构造出了整个数字串的最优解
- 而至于前i个字符中插入 (m-1)个加号

  →这又回到了原问题的形式, 也就是回到了以前状态

  →所以状态转移方程就能很快的构造出来了

6

最佳加法表达式

- 例如:

  数字串79846, 若需要加入两个加号,

  则最佳方案为79+8+46, 算术表达式的值为133

- 算法实现分析:
  - V[m][n]: 在n个数字中插入m个加号能达到的最小值

最佳加法表达式

- 算法实现分析:
  - **V[m][n]:** 在**n个数字**中插入**m个加号**能达到的最小值
  - 动规的递推方程:

if m = 0

V(m, n) = n个数字构成的整数

else if n < (m + 1) //加号多于数字的个数

V(m, n) = ∞

else

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

- Num(k, j)表示从第k个数字到第j个数字所组成的整数
- 数字编号从1开始算

```
#include <iostream>
#include <algorithm>
#include <string>
#include <stdlib.h>
using namespace std;
#define MAXN  15
#define MAXM 15
#define INFINITE 999999999
//不考虑大整数加法
int anMinValue[MAXM][MAXN]; //anMinValue[i][j]表示把i个加号
                            //放到j个数字前面所能的到的最小值
                            //题目要求 "anMinValue[m][n-1]"
int main(){
    int m;
    string s;
    cin >> s >> m;
    int n = s.length();
    int i;
    for( i = 0; i < n ; i ++ )
        anMinValue[0][i]= atoi( s.substr(0, i+1).c_str() );
   // c_str()函数表示将str转换成 char*格式; atoi()表示将字符转换为整数
```

```
for(  i = 1; i <= m; i ++ )
  for( int j = 0; j < n; j ++ )  { //把i个加号放第j个数字前面
      if( j < i )
         anMinValue[m][j] = INFINITE;
      else {
          int nMin = INFINITE;
          for( int k = 0; k <= j - 1; k ++ ) { //把i个加号里的最右边加号
                                               //放在第k个字符后面
              int nVal = 0;
              for( int u = k+1; u <= j ; u ++)
                  nVal = nVal * 10 + s.c_str()[u]-'0';
              nMin = min(nMin,  anMinValue[i-1][k] + nVal);
          }
          anMinValue[i][j] = nMin;
      }
  }
 cout << anMinValue[m][n-1];
 return 0;
}
```


10

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)

- 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

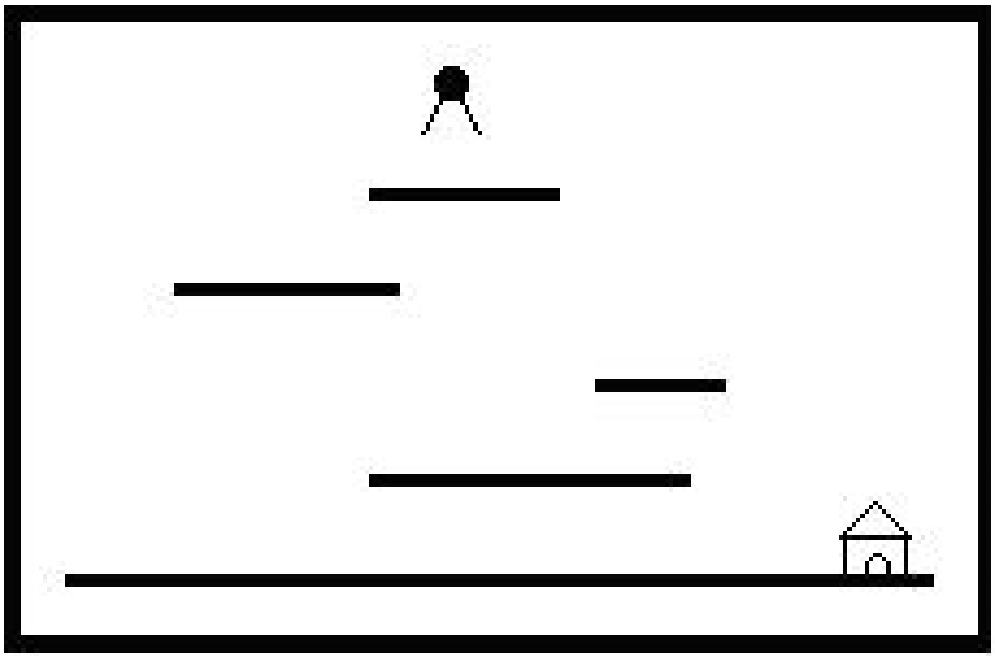
动规的要诀

枚举 --------->搜索--------->动态规划

(系统化)　　(记忆化)

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
- **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

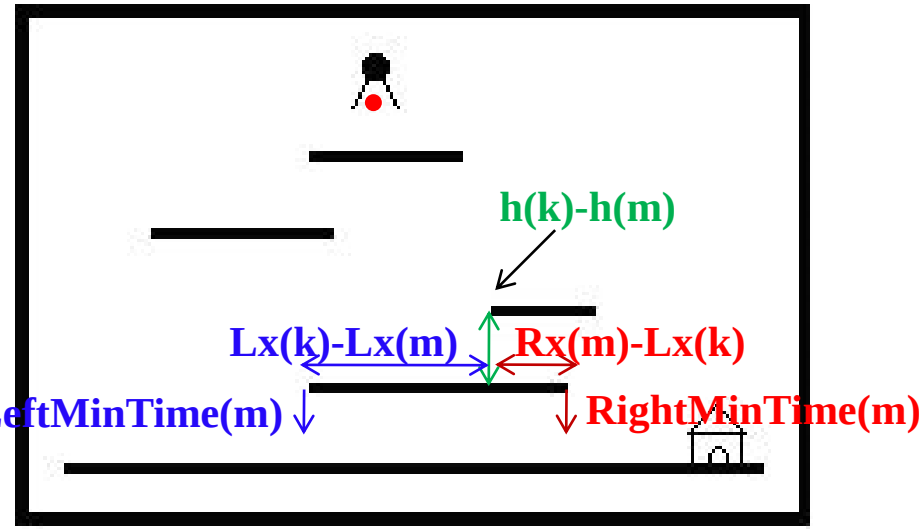
    Lx(k)-Lx(m) --从m号板子的落脚点走到m号板子左端点的时间

    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）

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$O(n^2)$

21

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;
    }
};
```


22

```
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;
```


23

```
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;
}
```


24

```
    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;
}
```


25

```
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;
}
```


26

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;
    }
};
```


27

```
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);
```


28

```
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;
        }
```


29

```
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;
}
```


30

```
        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;
}
```