2024年春

程序设计实习(II)：算法设计

第十九讲 动态规划补充

贾川民
北京大学

课前提醒

- QT大作业时间线
  - 6月1日提交初版，进行路演筛选
  - 6月5日课上Project路演（参加队伍有加分）
  - 6月30日（毕业年级6月22日）最终项目提交
  - 下周一公布路演名单

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**

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**

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

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
- 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

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( intj = 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;
}
```

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

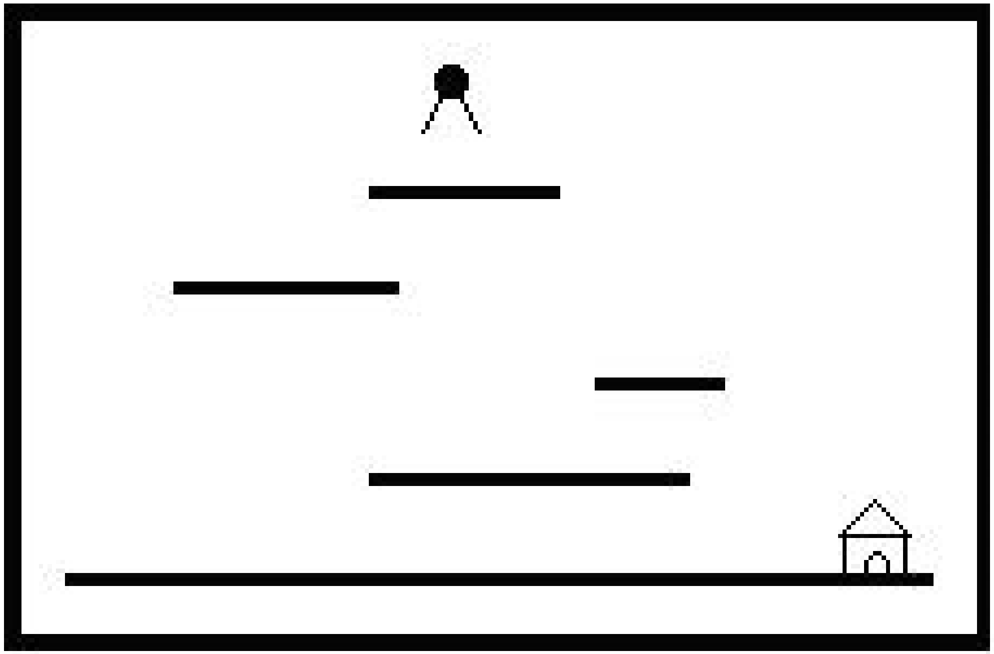
动规的要诀

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，就很容易选择
- 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

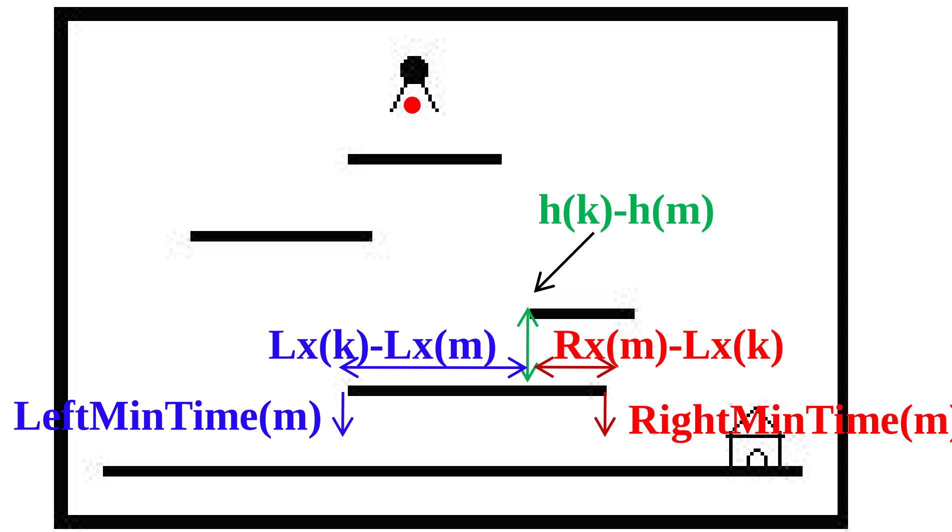
解题思路(2)

- 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 $O(n)$
- 找出板子一端到地面之间有哪块板子，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 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 ++ ) {  //找到正下方的板子
        if( platForms[i].Lx <= x && platForms[i].Rx >= x)
                break;
}
if( i <= n ) {  //找到了板子
        if( y - platForms[i].h > maxHeight )
                return INFINITE;
}
else {          //没找到板子
        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( rightMinTime[i] == -1 )
            rightMinTime[i] = MinTime(i,  false);
    nLeftTime += leftMinTime[i];
    nRightTime += rightMinTime[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(rightMinTime, -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 ) { //从下往上枚举
        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;
 }
```

```
        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;
}
```

滑雪 (百练1088)

Michael喜欢滑雪，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想知道在一个区域中最长的滑坡，其中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

滑雪 (百练1088)

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

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

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),

例如:

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

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)

```
#include <iostream>
#include <algorithm>
using namespace std;
struct Point{
	int r,c;
	int h;
	bool operator <( const Point & p) const {
		return h < p.h;
	}
} points[10100]; //用一维数组存储所有的点，以便排序处理
int field[110][110];
int L[110][110]; //L[i][j]是从(i,j)出发的最长滑行长度
int R,C;
```

```
int main()
{
        cin >> R >> C;
        for( int i = 0;i  < R; ++i )
                for( int j = 0; j < C; ++j ) {
                        //数据初始化
                        cin >> field[i][j];
                        points[i * C + j].h = field[i][j];
                        points[i * C + j].r = i;
                        points[i * C + j].c = j;
                        L[i][j] = 1;
                }
        sort(points,points+R*C);
```

```
    for( int i = 1; i < R * C; ++i ) {
            //每次循环,要求points[i]的L值
            int r = points[i].r;
            int c = points[i].c;
            if( r > 0 && field[r-1][c] < field[r][c] )
                    L[r][c] = max(L[r][c],L[r-1][c]+1);
            if( c > 0 && field[r][c-1] < field[r][c] )
                    L[r][c] = max(L[r][c],L[r][c-1]+1);
            if( r < R -1 && field[r+1][c] < field[r][c] )
                    L[r][c] = max(L[r][c],L[r+1][c]+1);
            if( c < C-1 && field[r][c+1] < field[r][c] )
                    L[r][c] = max(L[r][c],L[r][c+1]+1);
    }
    int maxLen = 0;
    for( int i = 0;i  < R; ++i )
            for( int j = 0; j < C; ++j )
                    maxLen = max(maxLen,L[i][j]);
    cout << maxLen <<endl;
}
```

“人人为我”式递推

“我为人人”式递推

```
    for( int i = 0; i < R * C; ++i ) {
        //每次循环开始时，points[i]的L值是已经最终求出了的
        int r = points[i].r;
        int c = points[i].c;
        if( r > 0 && field[r-1][c] > field[r][c] )
            L[r-1][c] = max(L[r-1][c],L[r][c]+1);
        if( c > 0 && field[r][c-1] > field[r][c] )
            L[r][c-1] = max(L[r][c-1],L[r][c]+1);
        if( r < R -1 && field[r+1][c] > field[r][c] )
            L[r+1][c] = max(L[r+1][c],L[r][c]+1);
        if( c < C-1 && field[r][c+1] > field[r][c] )
            L[r][c+1] = max(L[r][c+1],L[r][c]+1);
    }
    int maxLen = 0;
    for( int i = 0;i  < R; ++i )
        for( int j = 0; j < C; ++j )
            maxLen = max(maxLen,L[i][j]);
    cout << maxLen <<endl;
}
```

状态压缩动态规划

- 有时状态相当复杂，看上去需要很多空间，比如一个数组才能表示一个状态，那么就需要对状态进行某种编码，进行压缩表示
- 例如，状态和某个集合有关，集合里可以有一些元素，没有另一些元素，那么就可以用一个整数表示该集合，每个元素对应于一个bit，有该元素，则该bit就是1

TSP问题

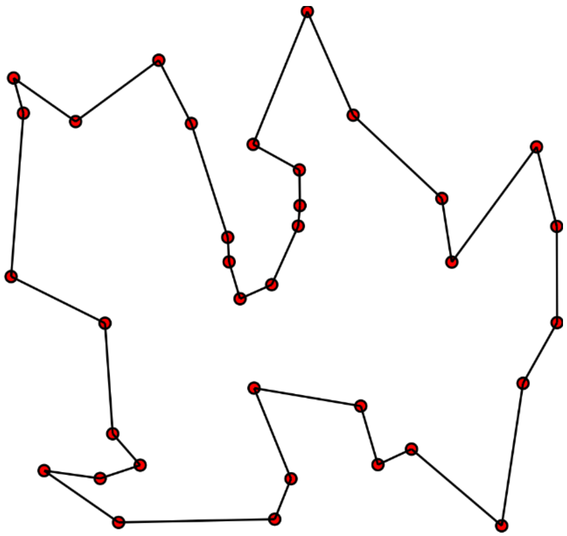
旅行商问题（最短路径问题）

Travelling Salesman Problem, TSP

N个城市，编号1到N，(N<=16)

任意两个城市间都有路，A → B 和 B → A 的路可能不一样长

已知所有路的长度，问**经过每个城市**恰好一次的**最短路径的长度**



TSP问题

- 用 dp[s][j] 表示经过集合s中的每个点恰好一次，且最后走的点是j (j ∈s)的最佳路径的长度

- 最终就是要求：

  min(dp[all][j]) (0 <= j < N)

  all是所有点的集合

TSP问题

状态方程:

$$dp[s][j] = \min\{ dp[s'][k] + w[k][j] \}$$

$j \in s$, $s' = s-j$, $k \in s'$, 枚举每个k, w[k][j]是k到j的边权值

边界条件: $dp[i][i] = 0$

问题:

如何表示点集s?

由于只有16个点, 可以用一个short变量表示点集

每个点对应一个bit, 例如:

$$5 = 000000000000010\mathbf{1}_2$$

5代表的点集是{0, 2}

全部n个点的点集, 对应的整数是: (1 << n) – 1

最终要求: min(dp[(1<<n)-1][j]) (0 <= j < n)

问题:

如何进行集合操作?

位运算

例: 从集合i中去掉点j, 得到新集合s':

$$s' = s \& (\sim(1 << j))$$

或

$$s' = s - (1 << j)$$

问题：

最终时间复杂度：

- 状态数目: dp[s][j]   s: 0 – 2^n-1, j: 0 – (n-1)
- 状态转移: $O(n)$
- 总时间: $O(n^2 2^n)$
- 硬枚举: $O(n!)$

海贼王之伟大航路（百练4124）

“我要成为海贼王的男人！”，路飞一边喊着这样的口号，一边和他的伙伴们一起踏上了伟大航路的艰险历程

路飞他们伟大航路行程的起点是罗格镇，终点是拉夫德鲁（那里藏匿着“唯一的大秘宝”—— ONE PIECE）

而航程中间，则是各式各样的岛屿

海贼王之伟大航路（百练4124）

因为伟大航路上的气候十分异常，所以来往任意两个岛屿之间的时间差别很大，从A岛到B岛可能需要1天，而从B岛到A岛则可能需要1年
当然，任意两个岛之间的航行时间虽然差别很大，但都是已知的

现在假设路飞一行从罗格镇(起点)出发，遍历伟大航路中间所有的岛屿（但是已经经过的岛屿不能再次经过)，最后到达拉夫德鲁(终点)
假设他们在岛上不作任何的停留，请问他们最少需要花费多少时间才能到达终点?

海贼王之伟大航路（百练4124）

输入数据

包含多行，第一行包含一个整数N($2 < N \le 16$)，代表有N个岛屿（包含起点和终点）

其中起点的编号为1，终点的编号为N

之后的N行每一行包含N个整数，其中第i ($1 \le i \le N$)行的第j ($1 \le j \le N$)个整数代表从第i个岛屿出发到第j个岛屿需要的时间t ($0 < t < 10000$)，第i行第i个整数为0

输出数据

一个整数，代表路飞一行从起点遍历所有中间岛屿（不重复）之后到达终点所需要的最少的时间

海贼王之伟大航路（百练4124）

样例输入：

4			
0	10	20	999
5	0	90	30
99	50	0	10
999	1	2	0

样例输出：

100

海贼王之伟大航路（百练4124）

这个问题的解可以直接推导出TSP (旅行商问题) 的解，而后者被证明是NP-Hard的，不能够在多项式时间内解决，所以这题最基本的就是 搜索+剪枝了

最先想到的当然是直接DFS，结果……

Time Limit Exceeded!

```
#define INF 1<<30
#define MAXN 16
int map[MAXN][MAXN];
int n;
int mindist = INF;
unsigned int dp[14][1<<14];

inline bool vis(int city, int state)
//vis函数:判断某个状态是否已经到达过
{
    if ( state&(1<<(city-1)) )
    {
        return true;
    }
    return false;
}
```

```
int dfs(int dist, int num, int crt, int state)
{
    if ( dist >= mindist )
    {
        return 0;  //剪枝,如果当前总路径已经超过已找到的最优路径,
                     则继续走下去只会更长,放弃后续搜索
    }
    if ( num == n-1 ) //Successfully getting one answer
    {
        dist += map[crt][n-1];
        if ( dist < mindist )
        {
            mindist=dist;  //更新最短路径长度
        }
        return 0;
    }
```

```
    for (int i = 1; i <n-1; i++)
    {
        if (vis(i,state)==0)
        {
            int newState=state|(1<<(i-1));
            if ( crt == 0 )  //当前位于起点岛屿
            {
                dp[i-1][newState]=map[crt][i];
                dfs(dist+map[crt][i],num+1,i,newState);
            }
            else if(dp[i-1][newState]>dp[crt-1][state]+map[crt][i])
//找到更优解,更新动态规划数组
            {
                dp[i-1][newState]=dp[crt-1][state]+map[crt][i];
                dfs(dist+map[crt][i],num+1,i,newState);
            }
        }
    }
    return 0;
}
```

```
int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            scanf("%d", &map[i][j]);
        }
    }
    memset(dp, -1, sizeof(dp));
    dfs(0, 1, 0, 0);
    printf("%d\n", mindist);
    return 0;
}
```

大炮阵地 (POJ1185)

- 司令部的将军们打算在N*M的网格地图上部署他们的大炮
- 一个N*M的地图由N行M列组成，地图的每一格可能是山地（用“H”表示），也可能是平原（用“P”表示），如图所示
- 在每一格平原地形上最多可以布置一门大炮 (山地上不能够部署大炮)

P	P	H	P	H	H	P	P
P	H	P	H	P	H	P	P
P	P	P	H	H	H	P	H
H	P	H	P	P	P	P	H
H	P	P	P	P	H	P	H
H	P	P	H	P	H	H	P
H	H	H	P	P	P	P	H

大炮阵地 (POJ1185)

- 如果在地图中的灰色所标识的平原上部署一门大炮，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格
- 图上其它白色网格均攻击不到，从图上可见大炮的攻击范围不受地形的影响
- 现在，将军们规划如何部署大炮，在防止误伤的前提下 (保证任何两门大炮之间不能互相攻击，即任何一门大炮都不在其他支大炮的攻击范围内)，在整个地图区域内最多能够摆放多少大炮
- 数据范围: 1<=n<=100, 1<=m<=10

P	P	H	P	H	H	P	P
P	H	P	H	P	H	P	P
P	P	P	H	H	H	P	H
H	P	H	P	P	P	P	H
H	P	P	P	P	H	P	H
H	P	P	H	P	H	H	P
H	H	H	P	P	P	P	H

大炮阵地 (POJ1185)

思 路:

如果用 dp[i]表示前i行所能放的最多大炮数目，

能否形成递推关系？

显然不能，因为不满足无后效性

大炮阵地 (POJ1185)

- 思路: 如果用 **dp[i]**表示前i行所能放的最多大炮数目，
  能否形成递推关系? 显然不能，因为不满足无后效性

因为dp[i]在第i行的某种放置方法会依赖于dp[i-1]中第i-1行的放置方法的约束，但是dp[i-1]中并没有这部分的信息

- 按照加限制条件加维度的思想，**加个限制条件**:

  dp[i][j]表示第i行的大炮布局为j的前提下，前i行所能放的最多大炮数目

  布局为j体现了状态压缩: j是个10位二进制数，表示一行大炮的一种布局

  有大炮的位置，对应位为1; 没有大炮的位置，对应位为0

大炮阵地 (POJ1185)

- 思路: 如果用 dp[i]表示前i行所能放的最多大炮数目，能否形成递推关系? 显然不能, 因为不满足无后效性
- 按照加限制条件加维度的思想, 加个限制条件:

dp[i][j]表示第i行的大炮布局为j的前提下，前i行所能放的最多大炮数目，布局为j体现了状态压缩, j是个10位二进制数，表示一行大炮的一种布局

最多10列, 因此每一行最多有2^10=1024种放法

因仅从 dp[i-1][k] (k = 0,…,1023) 无法推出dp[i][j], 达成 dp[i-1][k]可能有多种方案, 有的方案允许第i行布局为j, 有的方案不允许第i行布局为j, 然而却没有信息可以用来进行分辨

大炮阵地 (POJ1185)

- 再加限制条件，再加一维 (多加的状态变量用于补充必要的信息)：

dp[i][j][k]表示第i行布局为j，第i-1行布局为k时，前i行的最多大炮数目

1) j, k这两种布局必须相容，否则 dp[i][j][k] = 0

2) dp[i][j][k] = max{dp[i-1][k][m], m = 0,...,1023} + Num(j),

Num(j)为布局j中大炮的数目，j和m必须相容，k和m必须相容

此时满足无后效性

dp[i][j][k]	i	j	
	i-1	i-1	dp[i-1][k][m]
	i-2	m	

大炮阵地 (POJ1185)

- 再加限制条件, 再加一维 (多加的状态变量用于补充必要的信息):

dp[i][j][k]表示第i行布局为j, 第i-1行布局为k时, 前i行的最多大炮数目

1) j, k这两种布局必须相容, 否则 dp[i][j][k] = 0

2) dp[i][j][k] = max{dp[i-1][k][m], m = 0,...,1023} + Num(j), Num(j)为布局j中大炮的数目, j和m必须相容, k和m必须相容

此时满足无后效性

3) 初始条件:

dp[0][j][0] = Num(j)

dp[1][i][j] = max{dp[0][j][0]} + Num(i)

大炮阵地 (POJ1185)

- 问题：dp数组为:

  int dp[100][1024][1024], 太大

  时间复杂度和空间复杂度都太高

大炮阵地 (POJ1185)

- 问题：dp数组为：

  int dp[100][1024][1024], 太大, 时间复杂度和空间复杂度都太高

解决:

每一行里最多能放4个大炮, 就算全是平地, 能放大炮的方案数目也不超过60 (用一遍dfs可以全部求出）

大炮阵地 (POJ1185)

- 问题：dp数组为：

  int dp[100][1024][1024], 太大,

  时间复杂度和空间复杂度都太高

解决：

每一行里最多能放4个大炮。就算全是平地，能放大炮的方案数目也不超过60 (用一遍dfs可以全部求出）

算出一行在全平地情况下所有大炮的排列方案，存入数组 state[70]

int dp[100][70][70] 足矣

大炮阵地 (POJ1185)

- 问题：dp数组为：

  int dp[100][1024][1024]，太大，时间复杂度和空间复杂度都太高

解决：

  每一行里最多能放4个大炮，就算全是平地，能放大炮的方案数目也不超过60 (用一遍dfs可以全部求出)

  算出一行在全平地情况下所有大炮的排列方案，存入数组state[70]

  int dp[100][70][70] 足矣

  dp[i][j][k]表示第i行布局为state[j]，第i-1行布局为state[k]时，前i行的最多大炮数目

例题: 课程大作业

小明是北京大学信息科学技术学院三年级本科生。他喜欢参加各式各样的校园社团。这个学期就要结束了, 每个课程大作业的截止时间也快到了, 可是小明还没有开始做

每一门课程都有一个课程大作业, 每个课程大作业都有截止时间。如果提交时间超过截止时间X天, 那么他将会被扣掉X分。对于每个大作业, 小明要花费一天或者若干天来完成。他不能同时做多个大作业, 只有他完成了当前的项目, 才可以开始一个新的项目

小明希望你可以帮助他规划出一个最好的办法 (完成大作业的顺序) 来减少扣分

例题: 课程大作业

输入

输入包含若干测试样例

输入的第一行是一个正整数T, 代表测试样例数目

对于每组测试样例, 第一行为正整数N（1 <= N <= 15）代表课程数目

接下来N行, 每行包含一个字符串S（不多于50个字符）代表课程名称和两个整数D（代表大作业截止时间）和C（完成该大作业需要的时间）

注意所有的课程在输入中出现的顺序按照字典序排列

例题：课程大作业

输出

对于每组测试样例，请输出最小的扣分以及相应的课程完成的顺序

如果最优方案有多个，请**输出字典序**靠前的方案

例题: 课程大作业

样例输入	样例输出
2	2
3	Computer
Computer 3 3	Math
English 20 1	English
Math 3 2	3
3	Computer
Computer 3 3	English
English 6 3	Math
Math 6 3	

例题: 课程大作业

解题思路:

- dp[s] 表示已经完成的作业集合为s 时, 所能达到的最少扣分

- **状态方程:**

$$dp[s] = \min\{ dp[s'] + c(j) \} \quad (j \in s,\ \ s' = s - j)$$

c(j)表示, 完成s'后, 再完成j所造成的扣分

例题: 课程大作业

解题思路:

由于要记录完成作业的过程, 所以 在每个状态, 不但要记录到达该状态的最小扣分, 还要记录当初是从哪个状态转移到目前这个状态的（即计算出dp[s]时, dp[s]里面应该要记录计算时选出的最优的那个s'）, 这样从终态出发, 就能往回依次找到状态转移的路径（作业完成的顺序）

例题: 课程大作业

dp数组可以如下定义:

```
struct Node {   pre即上一个状态对应的结构体在dp数组中的下标
        int pre;      //上一个状态 (比当前状态完成的作业少了1个)
        int minScore;   //到达当前状态的最低扣分
        int last;     //当前状态下, 最后完成的作业的编号
        int finishDay;   //作业last完成的时间
} dp[ (1 << 16) + 10] ;
```

则dp[i]代表状态i的情况, i的上一个状态就是 dp[i].pre

例题: 课程大作业

- 边界条件:

  dp[0].minScore = 0

- 递推顺序:

  dp[0]->dp[1]->dp[2] ..... ->dp[1<<m - 1] (共m个大作业)

例题: 课程大作业

- **字典序问题：**

按如下公式计算 dp[s].minScore时:

dp[s] = min{ dp[s'] + c(j) }   (j ∈s, s’ = s – j)

如果发现有一个新的 s’, 导致dp[s’] + c(j) (先完成s’, 再完成作业j) 和当前 dp[s] 相等, 则由s出发, dp[s].last -> dp[s.pre].last -> dp[dp[s.pre].pre].last -> ….. 就是当前作业完成顺序的逆

j->dp[s’].last -> dp[[dp[s’].pre]].last -> … 就是另一条同样优的作业完成顺序的逆

比较这两个顺序的字典序; 如果从j出发的更小, 则更新 dp[s].last 为j, dp[s].pre为 s’, 相应的finishDay也更新

Thanks！