2016 级算法第三次上机解题报告

15081070 张雨任

一、总结

本次上机主要考查知识点为流水线问题、背包、计算几何、数组的线性操作等算法知识。

A、B 流水线问题是比较好考虑的初级动态规划问题，C、D 是有难度的，主要是转化成背包问题的过程不太好想，还好助教给了提示，E 题重点在于只有正四边形才能保证每个点都是整点，F 题考察动态规划，最长非增子序列问题，重点在于状态转移方程，G 题很不好想，是一道高维 dp，模型建立是难点。

二、解题报告

A. Bamboo 的小吃街

思路分析

本题考察流水线问题，基础的动态规划问题。两条小吃街可看作两条流水线。

算法分析

用 $f_1[j]$ 表示通过左边第 j 个店铺时的最短时间， $f_2[j]$ 表示通过右边第 j 个店铺时的最短时间。两边分别到第 j 个店铺的最快路线必定都是经过左边或右边的第 j-1 个店铺。

左边第 j 个店铺可以从左边第 j-1 个店铺到达，也可以从右边第 j-1 个店铺到达，取两者时间消耗最小的。同理，右边第 j 个店铺可以从左边第 j-1 个店铺到达，也可以从右边第 j-1 个店铺到达，取两者时间消耗最小的。

因此，此步的最优解 ＝min（本流水线上一步的最优解+本流水线此步所需时间，另一流水线上一步的最优解+本流水线此步所需时间+切换流水线的时间）

所以可写出状态转移方程：

$$f_1[j]=\begin{cases}e_1+a_{1,1} & \text{如果 } j=1\\ \min(f_1[j-1]+a_{1,j},f_2[j-1]+t_{2,j-1}+a_{1,j}) & \text{如果 } j\geqslant 2\end{cases}$$

$$f_2[j]=\begin{cases}e_2+a_{2,1} & \text{如果 } j=1\\ \min(f_2[j-1]+a_{2,j},f_1[j-1]+t_{1,j-1}+a_{2,j}) & \text{如果 } j\geqslant 2\end{cases}$$

其中 $a_{1,j}$ 代表左边第 j 家要停留的时间，$a_{2,j}$ 代表右边第 j 家要停留的时间，$t_{1,j-1}$ 代表从左边第 j-1 家移动到右边第 j 家需要的时间，$t_{2,j-1}$ 代表从右边第 j-1 家移动到左边第 j 家需要的时间，$e_1=e_2=0$（进入第一个店铺前没有时间消耗）。最后取 $f_1[n]$和 $f_2[n]$较小者即可。时间复杂度为 O(n)。

参考代码

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define ll long long
using namespace std;

int n;
int p1[507];
int p2[507];
int t[3][507];
int a1[507];
int a2[507];
int ans1,ans2;

int main() {
    while(~scanf("%d",&n)) {
        for(int i=1;i<=n;i++) {
            scanf("%d",&p1[i]);
        }
        for(int i=1;i<=n;i++) {
            scanf("%d",&p2[i]);
        }
        for(int i=1;i<=n-1;i++) {
            scanf("%d",&t[1][i]);
        }
        for(int i=1;i<=n-1;i++) {
```

```
            scanf("%d",&t[2][i]);
        }
        memset(a1,0,sizeof(a1));
        memset(a2,0,sizeof(a2));
        a1[1]=p1[1];
        a2[1]=p2[1];
        for(int i=2;i<=n;i++) {
            a1[i]=min(a1[i-1]+p1[i],a2[i-1]+t[2][i-1]+p1[i]);
            a2[i]=min(a2[i-1]+p2[i],a1[i-1]+t[1][i-1]+p2[i]);
        }
        printf("%d\n",min(a1[n],a2[n]));
    }
}
```

B. Bamboo 和巧克力工厂

思路分析

本题考察流水线问题，虽然比 A 题多了一个流水线，但原理相同。

算法分析

从前一机器手到当前机器手，需要消耗的时间为当前机器手消耗的时间加上流水线之间转移所消耗的时间（无需转移则为 0）。第 1 条流水线可从第 1 条流水线直接到达，也可以从第 2 条或第 3 条流水线转移过去，应选取三个中最小的。因此可写出第一条流水线的转移方程：

a1[i]=min{a1[i-1]+p[1][i], a2[i-1]+p[1][i]+t[2][1], a3[i-1]+p[1][i]+t[3][1]}

从流水线 1 到流水线 1，在当前机械臂之前所花费的最短时间基础上，加上当前机械臂花费的时间。从流水线 2 到流水线 1，在当前机械臂之前所花费的最短时间和当前机械臂花费时间的基础上，加上从 2 转移到 1 的时间。从流水线 3 到 1，在当前机械臂之前所花费的最短时间和当前机械臂花费时间的基础上，加上从 3 转移到 1 的时间。

同理可得第二条流水线和第三条流水线的转移方程，

a2[i]=min{a2[i-1]+p[2][i], a1[i-1]+p[2][i]+t[1][2], a3[i-1]+p[2][i]+t[3][2]}

a3[i]=min{a3[i-1]+p[3][i], a1[i-1]+p[3][i]+t[1][3], a2[i-1]+p[3][i]+t[2][3]}

最后，取 a1[m]、a2[m]、a3[m]最小值即可。

时间复杂度为 O(n)。

参考代码

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define ll long long
using namespace std;

int m;
int p[4][507];
```

```
int t[4][4];
int a1[507];
int a2[507];
int a3[507];

int main() {
    while(~scanf("%d",&m)) {
        for(int i=1;i<=3;i++) {
            for(int j=1;j<=m;j++) {
                scanf("%d",&p[i][j]);
            }
        }
        for(int i=1;i<=3;i++) {
            for(int j=1;j<=3;j++) {
                scanf("%d",&t[i][j]);
            }
        }
        memset(a1,0,sizeof(a1));
        memset(a2,0,sizeof(a2));
        memset(a3,0,sizeof(a3));
        a1[1]=p[1][1];
        a2[1]=p[2][1];
        a3[1]=p[3][1];
        for(int i=2;i<=m;i++) {
            a1[i]=min(min(a1[i-1]+p[1][i],a2[i-1]+p[1][i]+t[2][1]),
                  a3[i-1]+p[1][i]+t[3][1]);
            a2[i]=min(min(a2[i-1]+p[2][i],a1[i-1]+p[2][i]+t[1][2]),
                  a3[i-1]+p[2][i]+t[3][2]);
            a3[i]=min(min(a3[i-1]+p[3][i],a1[i-1]+p[3][i]+t[1][3]),
                  a2[i-1]+p[3][i]+t[2][3]);
        }
        int ans=min(a1[m],min(a2[m],a3[m]));
        printf("%d\n",ans);
    }
}
```

C. AlvinZH 的奇幻猜想——三次方

思路分析

本题考察完全背包问题。

背包问题的重点在于首先要分析出问题的分类，属于 01 背包、完全背包还是多重背包等，确定是某种背包问题之后，要从问题中抽象出 cost 和 value，确定选取某件物品（或事件）的代价和收益。

由于题目中每种数字可以选多个，因此是完全背包。

算法分析

从背包的角度入手，此题中要求正整数个数的最小值，因此可以确定放入每个数的收益 value 都为 1，相当于每加入一个数就多了一个数，收益增加 1，并且此题要求收益的最小值，因此改背包问题中 max 函数为 min 函数即可。总的背包容量是目标和 n，每选一个数 i，都要从 n 中消耗掉 i^3 的代价，因此 cost 为 i^3。

对于正整数 i，可以有两种操作，选择和不选择。不选择则保持 f[j]不变，放入背包则从总数 n 中消耗掉 i^3 的花费，获得 1 单位的收益，取两者较小的。由于每个数都可以重复，属于完全背包，因此内层循环从小到大。

因此可以写出状态转移方程：

$$f[j]=\min(f[j], f[j-i^3]+1)$$

此题还要额外注意初始化问题，由于求 f[n]的最小值，因此初始化时应把 f[i]都设置为最大值，正整数 i 最多可以分成 i 个 1^3 相加，因此初始状态 f[i]=i。

时间复杂度为 $O(n^{4/3})$。

Tips

一个降低运行时间的技巧。由于 f[i]=x 是一一映射关系，不受输入的 n 影响，因此可在输入前把 n 置为可能的最大输入，通过上述操作完成一次背包过程。这样可避免每次输入 n 都完成一次背包过程，减少大量重复运算。此优化可把运行时间从 667ms 降到 274ms。

最后判断是否只能由 n 个 1 组成时，显然 f[n]=n 的是全 1 的情况，另一种思路是当且仅当 $n<2^3$ 的时候，只能由全 1 组成。大于等于 8 的至少可以加入 2，就是非全 1 情况，因此只有 n<8 的时候才可能会 Oh NO!

参考代码一（完全背包）

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define ll long long
using namespace std;

int n;
int a[1000007];

int main() {
    while(~scanf("%d",&n)) {
        for(int i=0;i<=n;i++) a[i]=i;
        for(int i=2;i<=107;i++) {
            for(int j=i*i*i;j<=n;j++) {
                if(a[j]>a[j-i*i*i]+1) {
                    a[j]=a[j-i*i*i]+1;
                }
            }
        }
        if(a[n]==n) printf("Oh NO!\n");
        else printf("%d\n",a[n]);
    }
}
```

参考代码二（输入前完成背包过程）

```
#include <cstdio>
#include <algorithm>
using namespace std;

const int maxn=1000000;
int n;
int a[1000007];

int main() {
    for(int i=0;i<=maxn;i++) a[i]=i;
    for(int i=1;i<=107;i++) {
```

```
        for(int j=i*i*i;j<=maxn;j++) {
            if(a[j]>a[j-i*i*i]+1) {
                a[j]=a[j-i*i*i]+1;
            }
        }
    }
    while(~scanf("%d",&n)) {
        if(a[n]==n) printf("Oh NO!\n");
        else printf("%d\n",a[n]);
    }
}
```

D. 双十一的抉择

思路分析

本题考察 01 背包的应用。

要想把总糖数为 sum 的糖分的尽量接近，相当于两者都要尽量接近 sum/2，因此问题等价于把容量为 sum/2 的背包装的尽量满，能够装多少块糖的问题。

首先，每包糖都是唯一的，只有两种选择，选或不选，因此是 01 背包问题。然后要确定 cost 和 value，最终完成背包过程。

算法分析

本题的 cost 和 value 都是每袋糖的数量。每放入一袋内含 data[i]块糖的糖袋，总空间为 sum/2 的背包的空间就减少 data[i]，消耗掉 data[i]的空间，同时得到的收益，也就是当前拥有的糖的块数多出了 data[i]。对于当前的这袋糖 data[i]，我们可以把它放入背包，也可以不放入背包，维持原状，选两者较大的。因此转移方程可写成：

$$f[j]=max(f[j], f[j-data[i]]+data[i])$$

01 背包要保证面对第 i 袋糖时，从未装入过该糖，因此要保证面对第 i 袋糖时，还未更新过 f[j]。因此内层循环从大到小遍历，防止一个物品被放入多次。

最终即可算出背包容量为 sum/2 时可以放入的糖的块数的最大值 m。因此，一人得到 m，另一人得到 sum-m，因此结果为 sum-2*m，为 0 时输出 GF&SI。

参考代码

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define ll long long
using namespace std;

int n;
int data[1007];
int sum,v;
int a[50007];
```

```
int main() {
    while(~scanf("%d",&n)) {
        sum=0;
        for(int i=1;i<=n;i++) {
            scanf("%d",&data[i]);
            sum+=data[i];
        }
        v=sum/2;
        memset(a,0,sizeof(a));
        for(int i=1;i<=n;i++) {
            for(int j=v;j>=data[i];j--) {
                if(a[j]<a[j-data[i]]+data[i]) {
                    a[j]=a[j-data[i]]+data[i];
                }
            }
        }
        if(sum-2*a[v]==0) printf("GF&SI\n");
        else printf("%d\n",sum-2*a[v]);
    }
}
```

E. ModricWang's Polygons

思路分析

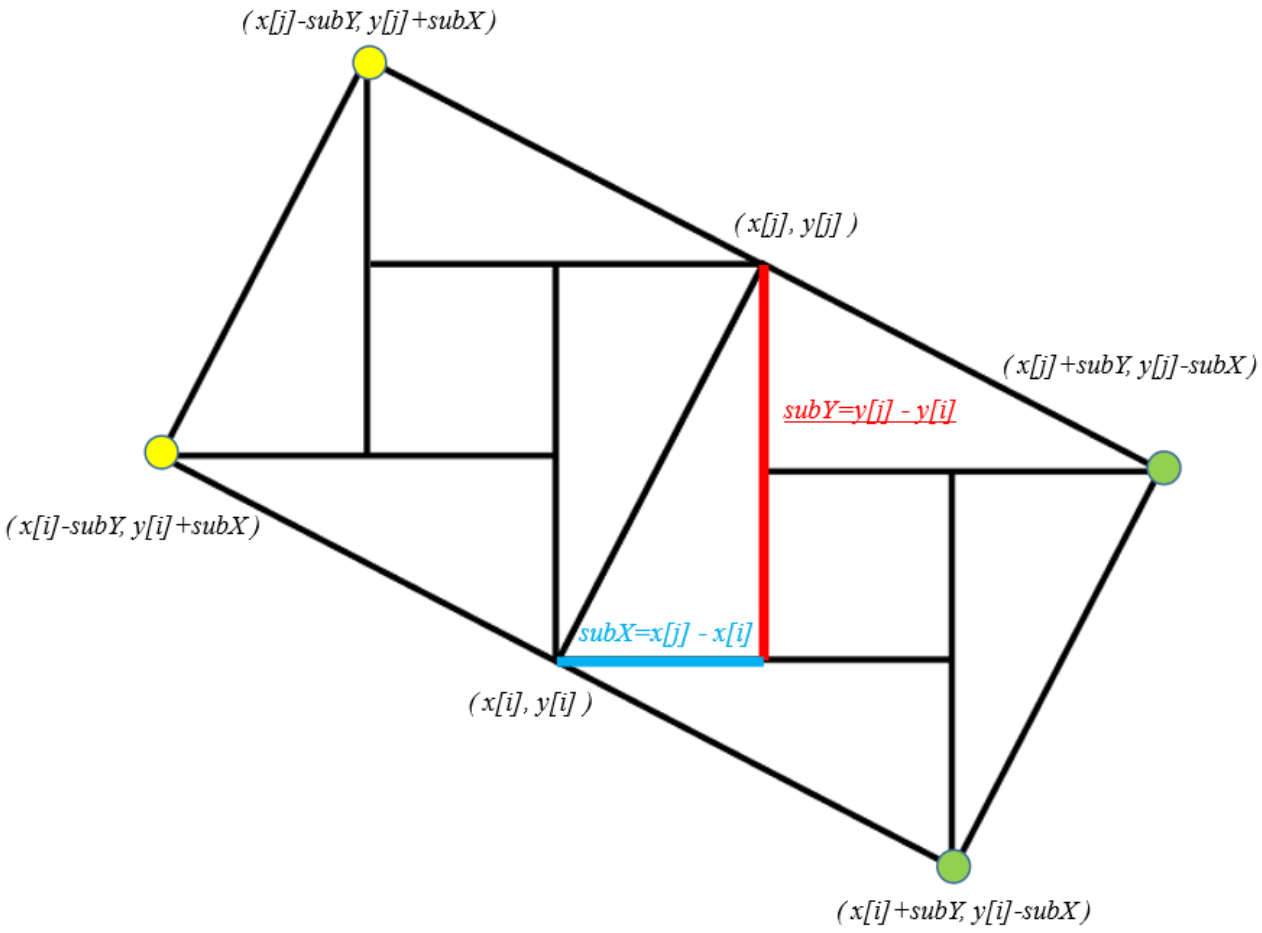
本题考查计算几何问题。本题的核心在于正多边形中只有正四边形才能保证每个点都是整点（没找到相关证明），因此问题转化为计算给定点集中有多少组点可以组成正四边形。

此题可采用映射的方式记录点集，当然也可以采用二分等其他搜索方法。

算法分析

思路一（二维数组映射，取边）：

点的坐标范围为 [-100, 100]，不方便用二维矩阵记录点是否属于点集，可以采用先前讲过的偏移量来解决，横纵坐标都加 100，使坐标范围为 [0, 200]，exist[x][y]=1 表示 (x, y) 属于给定点集。二重循环遍历所有的点对，查看是否有和此点对组成正方形的点对，有则计数器加一。做图分析思路更清晰：



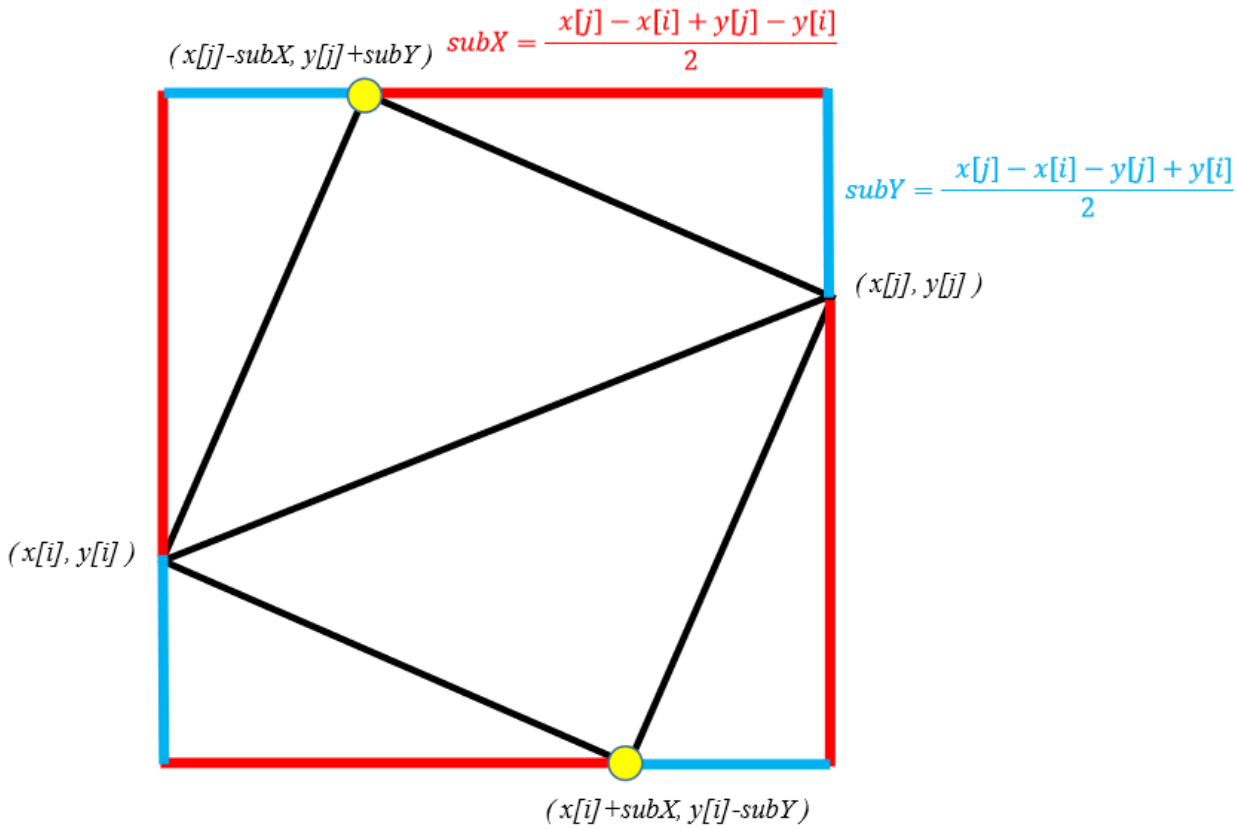
若图中 2 个黄点的横纵坐标在 0 到 200 之间，并且属于点集，则计算器自增。同理，若图中 2 个绿点的横纵坐标在 0 到 200 之间，并且属于点集，则计算器自增。

由于每两个点都会统计出对应的正方形，因此正方形的每个边都会作为基准边来统计一遍，每个正方形统计四次，所以最终答案除以 4 即可。

查找的时间复杂度为 O(1)，算法总时间复杂度 $O(n^2)$。

思路二（二维数组映射，取对角线）：

两个点组成的对角线也可以确定一个正方形。根据下图列式计算即可。



每个正方形有两个对角线，每个正方行计算两次，因此结果除以 2 即是所求。

查找的时间复杂度为 O(1)，算法总时间复杂度 $O(n^2)$。

思路三（二分查找）：

查找或搜索题也可以考虑二分查找，但二分的时间复杂度高于数组映射的方法。首先排序，以横坐标 x 为主关键字升序，横坐标 x 相等时以纵坐标 y 为关键字升序。二分时按照该排序规则查找即可。浮点数比大小要注意精度（不过这里应该不用，因为小数也都是 x.5 的

情况）。

查找的时间复杂度为 O(lgn)，总时间复杂度为 $O(n^2lgn)$。

参考代码一（二维数组映射，取边）

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#include <map>
#define ll long long
using namespace std;

int x[505];
int y[505];
bool exist[205][205];
int n,res,subX,subY;

int main() {
    while(~scanf("%d",&n)) {
        res=0;
        memset(exist,0,sizeof(exist));
        for(int i=1;i<=n;i++) {
            scanf("%d%d",&x[i],&y[i]);
            x[i]+=100;
            y[i]+=100;
            exist[x[i]][y[i]]=1;
        }
        for(int i=1;i<=n;i++) {
            for(int j=i+1;j<=n;j++) {
                subX=x[j]-x[i];
                subY=y[j]-y[i];
                if(x[i]+subY>=0 && x[i]+subY<=200 &&
                   y[i]-subX>=0 && y[i]-subX<=200 &&
                   x[j]+subY>=0 && x[j]+subY<=200 &&
                   y[j]-subX>=0 && y[j]-subX<=200) {
                    if(exist[x[i]+subY][y[i]-subX] &&
                       exist[x[j]+subY][y[j]-subX]) res++;
                }
                if(x[i]-subY>=0 && x[i]-subY<=200 &&
```

```
                y[i]+subX>=0 && y[i]+subX<=200 &&
                x[j]-subY>=0 && x[j]-subY<=200 &&
                y[j]+subX>=0 && y[j]+subX<=200) {
                 if(exist[x[i]-subY][y[i]+subX] &&
                    exist[x[j]-subY][y[j]+subX]) res++;
            }
        }
    }
    printf("%d\n",res/4);
  }
}
```

参考代码二（二维数组映射，取对角线）

```
#include <cstdio>
#include <cstring>
using namespace std;

int n,res;
int x[507];
int y[507];
bool exist[207][207];
double subX,subY;

int main() {
  while(~scanf("%d",&n)) {
    res=0;
    memset(exist,0,sizeof(exist));
    for(int i=0;i<n;i++) {
      scanf("%d%d",&x[i],&y[i]);
      x[i]+=100;
      y[i]+=100;
      exist[x[i]][y[i]]=1;
    }
    for(int i=0;i<n;i++) {
      for(int j=i+1;j<n;j++) {
        subX=(x[j]-x[i]+y[j]-y[i])/2.0;
        subY=(x[j]-x[i]-y[j]+y[i])/2.0;
        if((x[j]-subX)-(int)(x[j]-subX)==0 &&
          (y[j]+subY)-(int)(y[j]+subY)==0 &&
          (x[i]+subX)-(int)(x[i]+subX)==0 &&
```

```
                (y[i]-subY)-(int)(y[i]-subY)==0) {
                 if(x[j]-subX>=0 &&
                    x[j]-subX<=200 &&
                    y[j]+subY>=0 &&
                    y[j]+subY<=200 &&
                    x[i]+subX>=0 &&
                    x[i]+subX<=200 &&
                    y[i]-subY>=0 &&
                    y[i]-subY<=200) {
                     if(exist[(int)(x[j]-subX)][(int)(y[j]+subY)]
                     && exist[(int)(x[i]+subX)][(int)(y[i]-subY)]){
                         res++;
                     }
                 }
             }
         }
     }
     printf("%d\n",res/2);
 }
}
```

参考代码三（二分查找）

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const double eps=1e-7;
int n,res;
struct Node {
    int x;
    int y;
}s[507];
double subX,subY;

bool cmp(Node a,Node b) {
    if(a.x==b.x) return a.y<b.y;
    return a.x<b.x;
}
```

```
bool bSearch(double x,double y) {
    int l=0;
    int r=n-1;
    while(l<=r) {
        int mid=(l+r)/2;
        if(s[mid].x-x<eps && s[mid].x-x>-eps &&
           s[mid].y-y<eps && s[mid].y-y>-eps) {
            return true;
        }
        else if(s[mid].x-x>eps ||
                (s[mid].x-x<eps &&
                 s[mid].x-x>-eps &&
                 s[mid].y-y>eps)) {
            r=mid-1;
        }
        else l=mid+1;
    }
    return false;
}

int main() {
    while(~scanf("%d",&n)) {
        res=0;
        for(int i=0;i<n;i++) {
            scanf("%d%d",&s[i].x,&s[i].y);
        }
        sort(s,s+n,cmp);
        for(int i=0;i<n;i++) {
            for(int j=i+1;j<n;j++) {
                subX=(s[j].x-s[i].x+s[j].y-s[i].y)/2.0;
                subY=(s[j].x-s[i].x-s[j].y+s[i].y)/2.0;
                if(bSearch(s[j].x-subX,s[j].y+subY)) {
                    if(bSearch(s[i].x+subX,s[i].y-subY)) {
                        res++;
                    }
                }
            }
        }
        printf("%d\n",res/2);
    }
```

```
}
```

F. ModricWang 的导弹防御系统

思路分析

本题考查最长非增子序列问题，一个经典的 dp 问题，要点在于分析最优子结构，求出状态转移方程。

算法分析

给定一组数 data[1…n]，求该序列的最长非增子序列的长度。定义 a[i]表示前 i 个数的最长非增子序列的长度，那么 a[n]表示题中给定序列的最长非增子序列的长度，因此 a[i]可从任意 a[j]（1<=j<i）转移过来（只要满足非增的条件即可，即 data[i]<=data[j]），取他们之中最大者即可。

例如，求 a[6]时，按照如下规则求得（从蓝色序列的最长非增子序列长度推得当前长度）：

① 计算从 a[1]转移到 a[6]的情况，若满足 data[1]>=data[6]，最长非增子序列长度增加 1，data[6]加入序列中，则 a[6]=a[1]+1

② 计算从 a[2]转移到 a[6]的情况，若满足 data[2]>=data[6]，最长非增子序列长度增加 1，data[6]加入序列中，则 a[6]=a[2]+1

③ 计算从 a[3]转移到 a[6]的情况，若满足 data[3]>=data[6]，最长非增子序列长度增加 1，data[6]加入序列中，则 a[6]=a[3]+1

④ 计算从 a[4]转移到 a[6]的情况，若满足 data[4]>=data[6]，最长非增子序列长度增加 1，data[6]加入序列中，则 a[6]=a[4]+1

⑤ 计算从 a[5]转移到 a[6]的情况，若满足 data[5]>=data[6]，最长非增子序列长度增加 1，data[6]加入序列中，则 a[6]=a[5]+1

取以上 5 种情况的最大值（最长长度）即可算出 a[6]。

按照以上步骤，动态规划算出 a[1…n]的值，最后 a[n]即为所求。

参考代码

```
#include <cstdio>
#include <algorithm>
using namespace std;

int data[20];
int a[20];
int n,ans;

int main() {
    scanf("%d",&n);
    ans=0;
    for(int i=1;i<=n;i++) {
        scanf("%d",&data[i]);
    }
    for(int i=1;i<=n;i++) {
        a[i]=1;
        for(int j=1;j<i;j++) {
            if(data[j]>=data[i]) {
                a[i]=max(a[i],a[j]+1);
            }
        }
    }
    printf("%d\n",a[n]);
}
```

G. Winter is coming

思路分析

本题考查高维动态规划，难点在于模型的建立。

设 0 代表北境，1 代表野人。

本题的模型为 dp[0 的人数][1 的人数][队尾的人数][队尾的人种]。

（队尾的定义：01010101**00000**，黄色部分为队尾，队尾为队伍中末尾处同种人种的最长子串）

算法分析

要明确这道题的做法是向队伍后面添加人，可以添加 0，也可以添加 1。假设当前队伍的人数为 i+j 个人（i 个 0，j 个 1），现在要在队尾添加一个人，使队伍成为 i+j+1 人的队伍。

因此当前队伍有两种状态：

① 队伍状态为 a[i][j][k][0]，即 i 个 0、j 个 1、队尾有连续的 k 个 0，该状态可以有 2 种选择：一是在队尾继续放 0，得到 a[i+1][j][k+1][0]；二是在队尾放 1，打破连续 k 个 0 的队尾，得到队尾为 1 个 1 的情况，得到 a[i][j+1][1][1]。

② 队伍状态为 a[i][j][k][1]，即 i 个 0、j 个 1、队尾有连续的 k 个 1，该状态可以有 2 种选择：一是在队尾继续放 1，得到 a[i][j+1][k+1][1]；二是在队尾放 0，打破连续 k 个 1 的队尾，得到队尾为 1 个 0 的情况，得到 a[i+1][j][1][0]。

由以上分析不难写出状态转移方程。

循环中有以下几点要额外注意：

① 边界情况。a[..][..][0][..]的情况不存在，因为队尾至少有一人，因此 a[..][..][0][..]=0

② 初始状态。队伍中仅有一个人。该人为 0，则 a[1][0][1][0]=1。该人为 1，则 a[0][1][1][1]=1。还要注意在循环中不要破坏这个初始化条件。代码写好之后发现初始化条件等价于 a[0][0][0][0]=1，但原因尚不明确。

最终。计算所有 i=n, j=m 的情况，因此遍历所有合法的“队尾人数”和“队尾人种”，作累加即可。

时间复杂度 *O(n*m*max(x,y))*

参考代码

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define ll long long
using namespace std;

const int MOD=1000007;
int a[207][207][12][3];
int n,m,x,y,ans;

int main() {
    while(~scanf("%d%d%d%d",&n,&m,&x,&y)) {
        memset(a,0,sizeof(a));
        a[1][0][1][0]=1;
        a[0][1][1][1]=1;
        for(int i=0;i<=n;i++) {
            for(int j=0;j<=m;j++) {
                for(int k=0;k<=x;k++) {
                    if(i==0 && j+1==1) break;
                    a[i][j+1][1][1]+=a[i][j][k][0];
                    a[i][j+1][1][1]%=MOD;
                }
                for(int k=0;k<x;k++) {
                    if(i+1==1 && j==0 && k+1==1) break;
                    a[i+1][j][k+1][0]+=a[i][j][k][0]%MOD;
                    a[i+1][j][k+1][0]%=MOD;
                }
                for(int k=0;k<=y;k++) {
                    if(i+1==1 && j==0) break;
                    a[i+1][j][1][0]+=a[i][j][k][1];
                    a[i+1][j][1][0]%=MOD;
                }
                for(int k=0;k<y;k++) {
                    if(i==0 && j+1==1 && k+1==1) break;
                    a[i][j+1][k+1][1]+=a[i][j][k][1];
                    a[i][j+1][k+1][1]%=MOD;
                }
```

```
            }
        }
        ans=0;
        for(int i=0;i<=x;i++) {
            ans+=a[n][m][i][0];
        }
        for(int i=0;i<=y;i++) {
            ans+=a[n][m][i][1];
        }
        printf("%d\n",ans%MOD);
    }
}
```