搜索练习讲评

[NOIP2014 普及组] 珠心算测验

题目背景

NOIP2014 普及 T1

题目描述

珠心算是一种通过在脑中模拟算盘变化来完成快速运算的一种计算技术。珠心算训练，既能够开发智力，又能够为日常生活带来很多便利，因而在很多学校得到普及。

某学校的珠心算老师采用一种快速考察珠心算加法能力的测验方法。他随机生成一个正整数集合，集合中的数各不相同，然后要求学生回答：其中有多少个数，恰好等于集合中另外两个（不同的）数之和?

最近老师出了一些测验题，请你帮忙求出答案。

输入格式

共两行，第一行包含一个整数 n，表示测试题中给出的正整数个数。

第二行有 n 个正整数，每两个正整数之间用一个空格隔开，表示测试题中给出的正整数。

输出格式

一个整数，表示测验题答案。

样例 #1

样例输入 #1

```
4
1 2 3 4
```

样例输出 #1

```
2
```

提示

【样例说明】

由 $1+2=3, 1+3=4$，故满足测试要求的答案为 2。

注意，加数和被加数必须是集合中的两个不同的数。

【数据说明】

对于 100% 的数据，$3 \leq n \leq 100$，测验题给出的正整数大小不超过 $10,000$。

题目思路

题目的不仅仅认定两个加数正序、逆序重复，如 $1+4$ 和 $4+1$ 重复，还判断只要和相同也算重复，如 $1+4$ 和 $2+3$ 的情况。

这道题并非求方案数，而是求有多少个数能被其他两个不同的数相加，所以某一个数一旦被其他两个不同的数相加而得，就完成了这个数的判断。

参考代码

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 110;
int a[N];  // 用来存储输入的 n 个数
int main()
{
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
    {  // 输入 n 个整数，存储到数组 a 中
        cin >> a[i];
    }
    sort(a + 1, a + n + 1);  // 将数组 a 升序排列
    int cnt = 0;  // 用来记录有多少个数，恰好等于集合中另外两个（不同的）数之和
    for (int i = n; i >= 3; i--)
    {  // 枚举第 n~3 个数
        int l = 0;  // 代表左边的数(其实是下标)
        int r = i - 1; // 代表右边的数(其实是下标)
        while (l < r)
        {  // 当 l < r 成立时
            if (a[l] + a[r] < a[i])
            {  // 若左边的数加右边的数加起来小于当前的数
                l++;  // 往右找下一个数
            } else if (a[l] + a[r] > a[i]) {  // 若左边的数加右边的数加起来大于当前的数
                r--;  // 往左找下一个数
        } else if (a[l] != a[r])
            {  // 若左边的数加右边的数加起来等于当前的数
                cnt++;  // 计数器加 1
                break;  // 退出本层循环(while 循环)，避免重复计算
            }
        }
    }
    cout << cnt;  // 输出计数器

    return 0;
}

```

导弹拦截

题目描述

经过 11 年的韬光养晦，某国研发出了一种新的导弹拦截系统，凡是与它的距离不超过其工作半径的导弹都能够被它成功拦截。当工作半径为 0 时，则能够拦截与它位置恰好相同的导弹。但该导弹拦截系统也存在这样的缺陷：每套系统每天只能设定一次工作半径。而当天的使用代价，就是所有系统工作半径的平方和。

某天，雷达捕捉到敌国的导弹来袭。由于该系统尚处于试验阶段，所以只有两套系统投入工作。如果现在的要求是拦截所有的导弹，请计算这一天的最小使用代价。

输入格式

第一行包含 4 个整数 x_1, y_1, x_2, y_2，每两个整数之间用一个空格隔开，表示这两套导弹拦截系统的坐标分别为 $(x_1, y_1), (x_2, y_2)$。第二行包含 1 个整数 N，表示有 N 颗导弹。接下来 N 行，每行两个整数 x, y，中间用一个空格隔开，表示一颗导弹的坐标 (x, y)。不同导弹的坐标可能相同。

输出格式

一个整数，即当天的最小使用代价。

样例 #1

样例输入 #1

```
0 0 10 0
2
-3 3
10 0
```

样例输出 #1

```
18
```

样例 #2

样例输入 #2

```
0 0 6 0
5
-4 -2
-2 3
4 0
6 -2
9 1
```

样例输出 #2

```
30
```

提示

两个点 $(x_1, y_1), (x_2, y_2)$ 之间距离的平方是 $(x_1 - x_2)^2 + (y_1 - y_2)^2$。

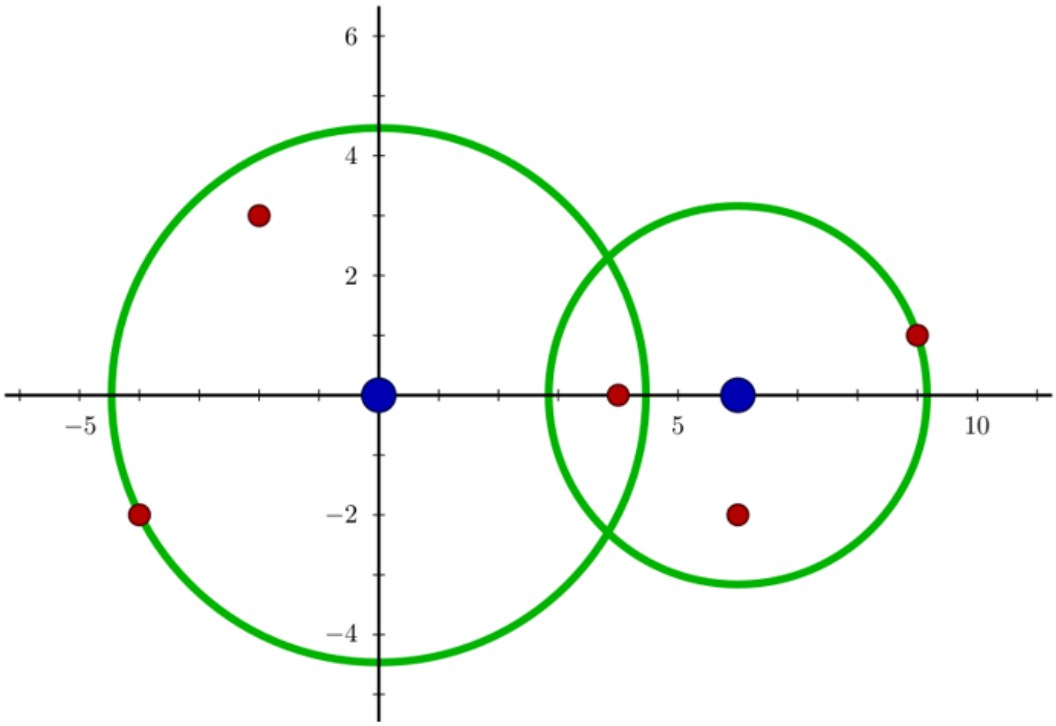
两套系统工作半径 r_1, r_2 的平方和，是指 r_1, r_2 分别取平方后再求和，即 $r_1^2 + r_2^2$。

样例 1 说明

样例 1 中要拦截所有导弹，在满足最小使用代价的前提下，两套系统工作半径的平方分别为 18 和 0。

样例 2 说明

样例 2 中的导弹拦截系统和导弹所在的位置如下图所示。要拦截所有导弹，在满足最小使用代价的前提下，两套系统工作半径的平方分别为 20 和 10。



【数据范围】。

- 对于 10% 的数据，$N = 1$。
- 对于 20% 的数据，$1 \le N \le 2$。
- 对于 40% 的数据，$1 \le N \le 100$。
- 对于 70% 的数据，$1 \le N \le 1000$。
- 对于 100% 的数据，$1 \le N \le 10^5$，且所有坐标分量的绝对值都不超过 1000。

解题说明

经典题，每个系统的**半径**即为**该系统拦截的导弹中距离系统坐标最远的导弹的距离**，所以不妨先算出每个导弹距离1号系统的距离，然后以距1号系统的距离进行**升序排序。**然后从距离1号系统最远的那枚导弹开始，计算出**比它距离1号系统的距离远的所有导弹中距离2号系统的距离最远**的那个距离（即以它前一枚导弹为1号系统所拦截的最远的导弹时，2号系统的半径），预处理结束后直接进行枚举答案，枚举1号系统所拦截的最远的导弹，然后输出最小的答案即可。 为了方便代码中所有距离直接用它的平方进行比较

参考代码

```
#include<iostream>
#include<cstring>
#include<algorithm>
#include<cmath>
using namespace std;
int x11,y11,x22,y22,n,minn=4000010;
struct DI
{
  int d1,d2,i;   //d1、d2分别为于1、2号系统的距离，i为该导弹的初始编号
}di[1000010];    //导弹的信息
bool cmp(DI a,DI b)
{
  return a.d1<b.d1;
}
int main()
{
  cin>>x11>>y11>>x22>>y22>>n;
  int x[n+5],y[n+5];//存储每枚导弹的坐标
  memset(di,0,sizeof(di));//初始化
  for(int i=1;i<=n;i++)
  {
    cin>>x[i]>>y[i];
    di[i].d1=pow(x[i]-x11,2)+pow(y[i]-y11,2); //该导弹与1号系统的距离
    di[i].i=i; //存储该枚导弹的初始编号
  }
  sort(di+1,di+n+1,cmp);
  for(int i=n;i>0;i--)
  {
    int a;
    a=pow(x[di[i].i]-x22,2)+pow(y[di[i].i]-y22,2); //该导弹与2号系统的距离
    di[i].d2=max(a,di[i+1].d2);//比较，若比后面所有导弹的距离都大，则替换
  }
  for(int i=0;i<=n;i++)
  {
    int a;
    a=di[i].d1+di[i+1].d2; //计算以该枚导弹为1号系统拦截的最远导弹所需的代价
    minn=min(a,minn);
  } //枚举答案，取最小值
  cout<<minn;//输出
  return 0;
}
```

Knight Moves

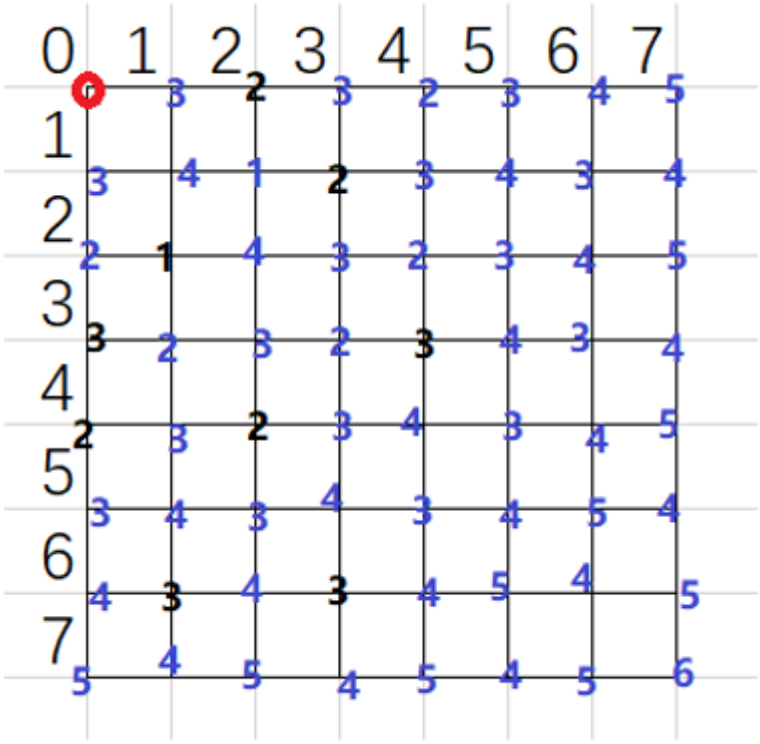
给定一个 $L \times L (4 \leq L \leq 300)$ 的棋盘，求从起点到终点最小需要移动的步数。

从起始点 0 开始，将一步就能走到的未访问过的点标1；

从已标1的点向外拓展一步，将未访问过的标2

……

直到目标点被标记为 k 。



参考代码

```
#include<bits/stdc++.h>
using namespace std;
struct st
{
    int x,y,step;
 } f[90005];
 const int dx[8]={-2,-1,1,2,2,1,-1,-2};
 const int dy[8]={1,2,2,1,-1,-2,-2,-1};
 bool map[305][305]; // 标记走过
 int sx,sy,ex,ey,n,l; // s起点，e终点

 bool onboard(int x,int y)
 {
    return ((x>=0)&&(x<l)&&(y>=0)&&(y<l)&&map[x][y]==0);
  }
```

```
16
17  bool isans(int x,int y)
18  {
19    return(x==ex && y==ey); // 终点判断
20  }
21
22  void bfs()
23  {
24    int x,y,head,tail;
25    memset(map,0,sizeof(map));
26    head=tail=1; // 起点
27    f[head].x=sx;
28    f[head].y=sy; // 标记
29    f[head].step=0;
30    map[sx,sy]=1;
31    while(head<=tail)
32    {
33        for (int i=0;i<8;i++) //8个方向
34        {
35            int tx=f[head].x+dx[i];
36            int ty=f[head].y+dy[i];
37            if (onboard(tx,ty))
38            {
39                // tx,ty 进队且标记
40                tail++;
41                f[tail].x=tx;
42                f[tail].y=ty.
43                f[tail].step=f[head].step+1;
44                map[f[tail].x] [f[tail].y]=1;
45                // 输出答案
46                if (isans(f[tail.x],f[tail].y))
47                {
48                    printf("%d\n",f[tail].step);
49                    return ;
50                }
51
52             }
53        }
54        head++;
55   }
```

奶酪

题目背景

NOIP2017 提高组 D2T1

题目描述

现有一块大奶酪，它的高度为 h，它的长度和宽度我们可以认为是无限大的，奶酪中间有许多半径相同的球形空洞。我们可以在这块奶酪中建立空间坐标系，在坐标系中，奶酪的下表面为 $z = 0$，奶酪的上表面为 $z = h$。

现在，奶酪的下表面有一只小老鼠 Jerry，它知道奶酪中所有空洞的球心所在的坐标。如果两个空洞相切或是相交，则 Jerry 可以从其中一个空洞跑到另一个空洞，特别地，如果一个空洞与下表面相切或是相交，Jerry 则可以从奶酪下表面跑进空洞；如果一个空洞与上表面相切或是相交，Jerry 则可以从空洞跑到奶酪上表面。

位于奶酪下表面的 Jerry 想知道，在不破坏奶酪的情况下，能否利用已有的空洞跑 到奶酪的上表面去?

空间内两点 $P_1(x_1, y_1, z_1)$、$P2(x_2, y_2, z_2)$ 的距离公式如下：

$$\text{dist}(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

输入格式

每个输入文件包含多组数据。

第一行，包含一个正整数 T，代表该输入文件中所含的数据组数。

接下来是 T 组数据，每组数据的格式如下： 第一行包含三个正整数 n, h, r，两个数之间以一个空格分开，分别代表奶酪中空洞的数量，奶酪的高度和空洞的半径。

接下来的 n 行，每行包含三个整数 x, y, z，两个数之间以一个空格分开，表示空洞球心坐标为 (x, y, z)。

输出格式

T 行，分别对应 T 组数据的答案，如果在第 i 组数据中，Jerry 能从下表面跑到上表面，则输出 `Yes`，如果不能，则输出 `No`。

样例 #1

样例输入 #1

```
3
2 4 1
0 0 1
0 0 3
2 5 1
0 0 1
0 0 4
2 5 2
0 0 2
2 0 4
```

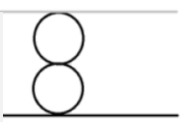
样例输出 #1

```
Yes
No
Yes
```

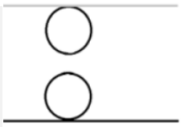
提示

【输入输出样例 1 说明】

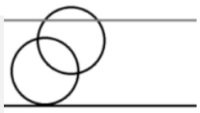
【输入】
2 4 1
0 0 1
0 0 3
【输出】
Yes

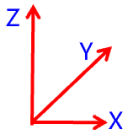


【输入】
2 5 1
0 0 1
0 0 4
【输出】
No



【输入】
2 5 2
0 0 2
2 0 4
【输出】
Yes





第一组数据,由奶酪的剖面图可见:

第一个空洞在 $(0,0,0)$ 与下表面相切;

第二个空洞在 $(0,0,4)$ 与上表面相切;

两个空洞在 $(0,0,2)$ 相切。

输出 `Yes`。

第二组数据,由奶酪的剖面图可见:

两个空洞既不相交也不相切。

输出 `No`。

第三组数据,由奶酪的剖面图可见:

两个空洞相交，且与上下表面相切或相交。

输出 `Yes`。

【数据规模与约定】

对于 20% 的数据，$n=1$，$1\le h$，$r\le 10^4$，坐标的绝对值不超过 10^4。

对于 40% 的数据，$1\le n\le 8$，$1\le h$，$r\le 10^4$，坐标的绝对值不超过 10^4。

对于 80% 的数据，$1\le n\le 10^3$，$1\le h,r\le 10^4$，坐标的绝对值不超过 10^4。

对于 100% 的数据，$1\le n\le 1\times 10^3$，$1\le h,r\le 10^9$，$T\le 20$，坐标的绝对值不超过 10^9。

题目分析

题目给出了球心坐标与半径，并且给出了奶酪高度，询问我们是否能从奶酪底部到奶酪顶部。

我们可以分出以下几种情况:

1. 当没有空洞接触下表面时，输出“No”;
2. 当一个空洞同时接触上表面与下表面，输出“Yes”;
3. 利用深搜进行遍历，寻找是否有路径可以连通上下表面。

我们需要开三个数组，来存放**每个点的坐标**，再利用题目已经给出的公式**求出两个球心间的距离**，如果这个距离**小于两倍的r**，说明这两个空洞是相通的，此时我们就可以将该空洞**标记**为已走过，然后继续寻找下一个空洞，如果最后能够走到上表面，我们就可以将$flag$赋值为1，代表我们有方案，然后输出$\backslash\backslash Yes\prime\prime$，否则，如果$flag=0$，我们就输出$\backslash\backslash No\prime\prime$。

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;

double cd(double x,double y,double z,double x1,double y1,double z1)
{
    // 计算两个球心间的长度
    return sqrt((x-x1)*(x-x1)+(y-y1)*(y-y1)+(z-z1)*(z-z1));
}

double x[1010],y[1010],z[1010],r,h;
int n,used[1010],t,flag=0;

void dfs(double x1,double y1,double z1)
{
    // 如果一个空洞能直接连通上下表面
    if(z1+r>=h)
    {
        flag=1;
        return ;
     }
    if (flag) return ;
    for (int i=1;i<=n;i++)
    {
        double c=cd(x1,y1,z1,x[i],y[i],z[i]);
        if (c<=2*r && used[i]==0)
        {
            // 如果两个空洞连通且没走过
            used[i]=1;  //标记
            dfs(x[i],y[i],z[i]); // 寻找下一个
        }
    }
}

int main()
{
    cin>>t;
    while(t--)
    {
        flag=0;
        cin>>n>>h>>r;
        memset(used,0,sizeof(used));
        for (int i=1;i<=n;i++)
            cin>>x[i]>>y[i]>>z[i];
        for (int i=1;i<=n;i++)
        {
            if (z[i]<=r)
            {
                // 遍历每个接触下表面的空洞
                used[i]=1;
```

```
                dfs(x[i],y[i],z[i]);
                if(flag)
                {
                    cout<<"Yes"<<endl;
                    break;
                 }
            }
        }
        if(flag==0) cout<<"No"<<endl;
    }
    return 0;
}
```