

2015 级算法第二次上机解题报告（助教版）

目 录

一、引言	1
二、解题报告	1
A 数论の初见.....	1
B 模式寻数.....	5
C jhljx 水水的补习班	6
D jhljx 水水的最短路径	9
E 我有特殊的快排技巧	12
F 四合归零	15

2015 级算法第二次上机解题报告（助教版）

马国瑞

一、引言

较上次上机相比，本次上机在 D 题没有给出解题提示的情况下难度较第一次上机有所上升，但因为本次上机的 A、B 两道简单题较第一次上机相比难度有较大幅度的降低，因此本次上机平均分较第一次上机高一些（第一次上机平均分 2.84，本次上机平均分 3.04），0 分人数也有大幅度减少，但依旧有 5 位同学没有获得上机得分。在此，助教希望上机不理想的同学课下多加练习，不懂多向同学和助教提问，争取达到熟练掌握，同时鼓励大家积极但必须要**认真**撰写解题报告。

本篇助教版解题报告和上一篇有所不同，上一篇主要由我来完成，这一篇大部分内容会包含大家上交的解题报告的内容。在此向所有提交解题报告的同学们的学习积极性提出表扬。

不得不提，本次上机大家提交的解题报告总体质量较上次上机相比有所提高，但还有相当多的同学仅仅对题目的解题思路进行说明，尽管它很详细，并没有对算法分析的内容加以体现，包括使用某种优化的算法相比没有使用该优化的算法相比有什么优点、主方法求解递归式得到算法时间复杂度的过程等等。希望之后大家提交的每一份解题报告都要对第二次上机讲解 PPT 第三页内容提到的解题报告要素有所体现。

此外，欢迎大家对这份解题报告内容中出现的问题批评指正。

二、解题报告

A 数论の初见

思路分析

本题若不使用打素数表的方式做，需要使用针对 n 次查询花费时间为 $O(n \lg \lg n)$ 的埃氏筛法及针对 n 次查询花费时间为 $O(n)$ 的欧拉筛法；若使用事先处理数据得到 $1 \sim 1000000$ 之间任何一个数比它小的素数个数，数据预处理时间复杂度最大 $O(n^{3/2})$ ，单次查询时间复杂度为 $O(1)$ 可 AC。

但这道题很值得思考：如何才能使用尽量少的时间来处理针对大量的数的素数筛选问题。

这里介绍两种素数筛法供大家参考。

方法一 Eratosthenes 筛选法

事实上，这种筛选法在 2015 级 C++ 期末上机模拟赛考查过 (<https://biancheng.love/problem/226/index>)。

这里先给出如下数学定理：

- **整数的唯一分解定理的引理：**任何大于 1 的正整数 n 都可以写成素数之积
- 由上述引理推出：若一个数不是素数，则必存在一个小于它的素数为其的因数

Eratosthenes 筛选法就是基于上述两条定理而得出的算法，其正确性不言而喻。其基本思想如下：要得到自然数 n 以内的全部素数，必须把不大于 $n^{1/2}$ 的所有素数的倍数剔除，剩下的就是素数。

具体操作如下：给出要筛数值的范围 n ，找出 n 以内的素数。先用 2 去筛，即把 2 留下，把 2 的倍数剔除掉；再用 2 的下一个没被筛掉的数，也就是 3 筛，把 3 留下，把 3 的倍数剔除掉；接下去用下一个质数 5 筛，把 5 留下，把 5 的倍数剔除掉；不断重复下去.....

另外，这种素数筛选法可以在 $[2, n^{1/2}]$ 内从小到大枚举基数 i ，去掉 $[i^2, n]$ 范围内所有的 i 的倍数，枚举 i 直到超过 $n^{1/2}$ 为止，剩下的所有数都是素数。


至于为什么在选定基数 i 后，不用考虑 (i, i^2) 范围内的数，是因为此范围内的数一定有比 i 小的因子，肯定在之前已经被删除出集合了。事实上，每个数在它的最小的素因子被选定为基数时，就一定会被删除出集合。

对于求 n 以内的素数个数，这种筛法优化后的时间复杂度可达到 $O(n \lg \lg n)$ 。

方法二 欧拉筛法

Eratosthenes 筛选法有一种缺点，就是像类似于 210 这种数会同时被 2、3、5、7 这四个质数同时成倍数的筛除，会造成重复操作带来的时间浪费。在数据规模达到 1,000,000,000 以及更大的时候就会发现 Eratosthenes 筛选法很费时，就是这个原因。因此，为了解决这个问题，就有了欧拉筛法的提出。

算法的代码表示如下：

```
for (int i = 2; i <= N; i++) {
    if (!check[i]) prime[tot++] = i;
    for (int j = 0; j < tot; j++) {
        if (i * prime[j] > N) break;
        check[i * prime[j]] = true;
         if (i % prime[j] == 0) break;
    }
}
```

时间复杂度证明如下：

设合数 n 最小的质因数为 p ，它的另一个大于 p 的质因数为 p' ，令 $n = pm = p'm'$ 。观察上面的程序片段，可以发现 j 循环到质因数 p 时合数 n 第一次被标记（若循环到 p 之前已经跳出循环，说明 n 有更小的质因数），若也被 p' 标记，则是在这之前（因为 $m' < m$ ），考虑 i 循环到 m' ，注意到 $n = pm = p'm'$ 且 p, p' 为不同的质数，因此 $p|m'$ ，所以当 j 循环到质数 p 后结束，不会循环到 p' ，这就说明不会被 p' 筛去。

因此，欧拉筛法完全避免了重复操作，对于求 n 以内的素数，时间复杂度为 $O(n)$ 。

参考代码

//此代码为算法练习赛 J 题 Magry's Prime 的参考代码

//对本题同样适用

```
#include<iostream>
#include<cstdio>
#include<cstring>
#define N 1000000
using namespace std;
int prime[3100010]; //prime 数组存储素数表，从小到大排序
bool flag[50000010]; //flag 数组用于筛法筛选素数
int p;                //p 为全局变量，在数据预处理后的值是 N 以内所有的素数个数
void f()               //f 函数为埃氏筛法
{
    p=0;
    memset(flag,true,sizeof(flag));
    flag[0]=false;
    flag[1]=false;
    for(int i=2;i<=N;i++){
        if(flag[i]){
            prime[p]=i;
            ++p;
            for(int j=2*i;j<=N;j+=i)    //其实 j 的初值可以为 i*i
                flag[j]=false;
        }
    }
}

void g()               //g 函数为欧拉筛法
{
    p=0;
    memset(flag,true,sizeof(flag));
    flag[0]=false;
    flag[1]=false;
    flag[2]=true;
    for(int i=2;i<=N;i++){
        if(flag[i]){
            prime[p]=i;
            ++p;
        }

        for(int j=0;((j<p)&&(i*prime[j]<=N));j++){
```

```

        flag[i*prime[j]]=false;
        if(i%prime[j]==0) break;    //关键在这句保证合数不被重复筛除
    }
}
int main()
{
    //f();
    g();          //此处仅使用欧拉筛法进行数据预处理
    int n;
    while(cin>>n){
        int ans=0;
        //对于本题可通过  $O(1)$  的时间复杂度查询
        //而对于练习赛 J 题，由于内存限制，单次查询时间复杂度最小值能达到  $O(\lg n)$ 
        //此代码仅仅展示  $O(n)$  的线性查找
        //对于二分查找，可自己思考练习
        for(int i=0;i<p&&prime[i]<=n;i++){
            if(prime[i]<=n)
                ans++;
        }
        cout<<ans<<endl;
    }
}

```

参考资料

- 1.《数论与应用》 纪建 清华大学出版社
- 2.15211088 王意如 算法第二次上机解题报告 A 题
3. <http://wenku.baidu.com/view/2d706761aa00b52acec7ca63.html>

问题思考

Miller Rabin 素数判定法对于解决这类问题可能会有帮助，同时素数的判定算法可能会对大质因数分解问题的解决有一定帮助（Pollard-rho 算法）。因时间仓促，解题报告作者并没有时间思考这个问题，于是把这个问题留给大家思考一下啦！

B 模式寻数

思路分析

本题是本次上机最简单的一题，只需注意两个数组各取一个数得到所有结果的最小值相当于两个数组分别取最小值相加得到的结果即可。

取一个数组最小值的最优方法是边输入边比较，分别找出两个数组的最小值，时间复杂度为 $O(n)$ 。伪代码如下：

```
mina=a[0],minb=b[0];
for i=1 to n
    if(a[i]<mina) mina=a[i];
    if(b[i]<minb) minb=b[i];
return mina+minb;
```

笔者认为，这样的思想其实就体现了贪心思想。大家之后会学到贪心算法，和贪心相关的题目的难度既可以相当低，也可以相当高，并且是《算法导论》中提到的三种高级设计和分析技术之一，大家可以通过书上和课上的例子理解一下。

参考代码

```
#include<iostream>
using namespace std;
#define INF 0x7FFFFFFF
int a[2][4000];      //用于存储 A、B 两个数组
int main()
{
    int n;
    while(cin>>n){
        //设定最小值的初值，对于 int 范围的数需要设定为 INT_MAX
        int mina[2]={INF,INF};
        for(int i=0;i<2;i++){
            for(int j=0;j<n;j++){
                int inp2=scanf("%d",&a[i][j]);
                if(a[i][j]<mina[i]) mina[i]=a[i][j];
            }
        }
        cout<<mina[0]+mina[1]<<endl;
    }
}
```

C jhljx 水水的补习班

思路分析

Author: 15211103 乐昀

本题考察了 STL 中优先队列 `priority_queue` 使用。这种队列它可以按照自定义的一种方式（数据的优先级）来对队列中的数据进行动态的排序。每次的 `push` 和 `pop` 操作，队列都会动态的调整，以达到我们预期的方式来存储。例如：我们常用的操作就是对数据排序，优先队列默认的是数据大的优先级高所以我们无论按照什么顺序 `push` 数据，最终在队列里总是 `top` 出最大的元素。

首先，创建一个 `student` 结构体，包含 `student_number`, `sex`, `scores` 三个 `int` 类型变量。

输入：用一个 `student` 类型的数组储存输入的数据。

创建优先队列 `priority_queue`。当操作为 `Add` 时，将数组中的数据入队。当操作为 `Delete` 时，执行出队操作。由于优先队列中，每操作一次，队列都会动态的调整，使优先级最大的元素排在队首。因此当操作为 `Query` 时，直接输出队首元素即可。

那么，如何设置队列的优先级呢？

标准库默认使用元素类型的 `<` 操作符来确定它们之间的优先级关系。因此只要重载 `<` “运算符”，就能按照自定义的方式设置优先队列中元素的优先级。

值得注意的是，重载 `>` 号会编译出错，因为标准库默认使用元素类型的 `<` 操作符来确定它们之间的优先级关系。而且自定义类型的 `<` 操作符与 `>` 操作符并无直接联系。

因而，使用以下代码，设置队列的优先级顺序：（在结构体内设置，无须声明为友元函数）

```
friend bool operator<(student s1, student s2){
    if(s1.score==s2.score){
        if(s1.sex==s2.sex)
            return s1.student_number>s2.student_number;
        else
            return s1.sex>s2.sex;
    }
    else
        return s1.score<s2.score;
}
```

算法分析

优先队列算法有如下四个操作：

1. `Insert(S, x)` - 插入，即 `S.push(x)`;

2. Meximum(S) - 查询优先级最大值, 即 S.top();
3. Extract-Max(S) - 去掉 S 中优先级最大元素, 即 S.pop();
4. Increase-Key(S,x,k) - 将元素 x 的关键字值增加到 k, $k \geq x$

一次操作 1 与 4 的时间复杂度均为 $O(\lg n)$, 一次操作 2 与 3 的时间复杂度均为 $O(1)$.

对于本题, 会执行上述操作的 1、2、3, 对运行时间主要的影响因素为插入操作。对于 n 次插入操作, 时间复杂度为 $O(n \lg n)$; 对于 n 次查询删除操作, 时间复杂度为 $O(n)$, 因此对于每组数据时间复杂度为 $O(n \lg n)$.

参考代码

```
#include<cstdio>
#include<queue>
#include<string>
#include<iostream>
using namespace std;
struct Student
{
    int s;
    int g;
    int sc;
    //这里 s 代表学号, g 代表性别, sc 代表成绩
    bool operator < (const Student &a) const
    {
        if(a.sc==sc&&g==g)
            return a.s<s;
        else if(a.sc==sc)
            return a.g<g;
        else
            return a.sc>sc;
    }
};

priority_queue<Student> q;
Student stu[100010];
int main()
{
    int n,m;
    while(~scanf("%d%d",&n,&m)){
```



```

while(!q.empty())    //这里注意若 q 为全局变量需要对每组数据进行 q 的清空
    q.pop();

for(int i=0;i<n;i++){
    scanf("%d%d%d",&stu[i].s,&stu[i].g,&stu[i].sc);
}

for(int i=0;i<m;i++){
    char token[10];
    scanf(" %s",token); //无空格及换行符的字符串的输入姿势
    int k;
    if(token[0]=='A'){
        scanf("%d",&k);
        q.push(stu[k-1]);    //插入
    }
    else if(token[0]=='D')
        q.pop();            //删除
    else if(token[0]=='Q'){    //查询
        printf("%d %d %d\n",q.top().s,q.top().g,q.top().sc);
    }
}
}
}

```

参考资料

1. 《算法导论》中文第三版，机械工业出版社

问题思考

这个问题使用非 C++ STL 的小根堆如何实现？

Dijkstra 水水的 shortest path

思路分析

本题至少可由如下两种方法可做：

方法一 优先队列优化的 Dijkstra 算法

Author: 15211115 陈瀚清（原文有改动）

最短路径算法通常采用 Dijkstra 算法。对 Dijkstra 算法的步骤介绍如下：

对于图 $G=\{V, E\}$

1. 初始时令 $S=\{V_0\}, T=V-S=\{\text{其余顶点}\}$ ， T 中顶点对应的距离值

若存在 $\langle V_0, V_i \rangle$ ， $d(V_0, V_i)$ 为 $\langle V_0, V_i \rangle$ 弧上的权值

若不存在 $\langle V_0, V_i \rangle$ ， $d(V_0, V_i)$ 为 ∞

2. 从 T 中选取一个与 S 中顶点有关联边且权值最小的顶点 W ，加入到 S 中

3. 对其余 T 中顶点的距离值进行修改：若加进 W 作中间顶点，从 V_0 到 V_i 的距离值缩短，则修改此距离值；

4. 重复上述步骤 2、3，直到 S 中包含所有顶点，即 $W=V_i$ 为止

本题由于数据量相当大，不可能采取之前的邻接矩阵的方式储存图，因此需要一个高效的容器能够动态改变大小，无需提前声明大小，这就是 `vector` 容器。由于输入的数据有起点、终点、长度三个数据，所以 `vector` 内需要存放的不仅仅是一个 `int` 类型，而是结构体类型，我们可以将起点当作容器数组的下标，终点和长度作为数组元素存放入 `vector`。

（补充说明：其实可以认为是邻接表，既可用链表表示也可用 `vector` 表示）

接下来说明堆优化，由于 Dijkstra 算法中，每次选取一个结点时，需要更新其他结点到走过结点的最短距离，并返回一个最短距离的结点（距离最短优先），这就可以利用已经声明的结构体，将终点当成下一个走的结点，长度作为路径长度，声明一个优先队列，代替以前用来存当前最短路径的数组。

方法二 队列优化的 Bellman-Ford 算法

Author: 15211103 乐昀

1、用数组 `dist` 记录每个结点的最短路径估计值，用一个结构体 `Edge` 的数组来存储图 G 。我们采取的方法是动态逼近法：设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点 u ，并且用 u 点当前的最短路径估计值对离开 u 点所指向的结点 v 进行松弛操作，如果 v 点的最短路径估计值有所调整，且 v 点不在当前的队列中，就将 v 点放入队尾。这样不断从队列中取出结点来进行松弛操作，

直至队列空为止

2、建立一个队列：

(1) 将起始点放入队列中。

(2) 建立一个表格记录起始点到所有点的最短路径（该表格的初始值要赋为极大值，该点到他本身的路径赋为 0。

(3) 然后执行松弛操作，用队列里有的点作为起始点去刷新到所有点的最短路，如果刷新成功且被刷新点不在队列中则把该点加入到队列最后。重复执行直到队列为空。

具体代码如下：

```
while(!que.empty()){
    int u=que.front();
    que.pop();
    vis[u]=false;
    for(int i=head[u]; i!=-1; i=nnext[i]){
        int v=E[i].v;
        if(dist[v]>dist[u]+E[i].cost){
            dist[v]=dist[u]+E[i].cost;
            if(!vis[v]){
                vis[v]=true;
                que.push(v);
            }
        }
    }
}
```

3、判断有无负环：

如果某个点进入队列的次数超过 N 次则存在负环（SPFA 无法处理带负环的图）

4、如果遍历完全图之后，两点之间的距离仍然为 INF，说明这两点之间不联通，输出-1.

算法分析

对于方法一，如仅用 Dijkstra 算法，while 循环选取 $n-1$ 个结点需要 $O(n)$ ，找到去各个点路径中的最短路径需要一个 for 循环即 $O(n)$ ，更新各个点的路径也需要 $O(n)$ ，所以复杂度为 $O(n^2)$ ；采用堆优化的 Dijkstra 算法，while 循环时间复杂度不变（本题可能由于队列里有多余的数据有所改变），找到各个点路径中的最短路径需要 $O(1)$ 的时间，更新各个点的路径需要 $O(\lg n)$ 的时间，所以时间复杂度共 $O(n \lg n)$ 。

对于方法二，每实施一次松弛操作，最短路径树上就会有一层顶点达到其最短距离，此后这层顶点的最短距离值就会一直保持不变，不再受后续松弛操作的影响。（但是，每次还要判断松弛，这里浪费了大量

的时间，这就是 Bellman-Ford 算法效率底下的原因，也正是 SPFA 优化的所在）。

因代码篇幅过大，故本题参考代码（两种方法）参见给出的两个 `cpp` 文件。

参考资料

1. <http://www.cnblogs.com/hxsyl/p/3270401.html>

E 我有特殊的快排技巧

思路分析

其实这道题也可以视作求无序序列的第 k 小数问题，可以用类似于快速排序的算法求解。

快速排序的“三步走”思想如下：

- 划分问题：各个元素重排后分成左右两部分，使得左边任意元素都不大于右边的任意元素（或者相反）
- 递归求解：把左右两部分分别排序
- 合并问题：此时数组已经完全有序，不用合并

对于快速排序过程如何运行，我们举下述简单的排序实例进行简要说明：

初始数组为-----> $S: 6, 10, 13, 5, 8, 3, 2, 11$

将第一个元素赋值给 v -----> $v = 6$;

以 v 为标准将 S 进行拆分---> $[2, 5, 3], [6], [8, 13, 10, 11]$ <-----将得到的数组命名为 S_1, S_2 ;

同样对子数组 S_1 进行拆分-> $[], [2], [5, 3]$ <-----拆分之后，第一个子数组为空。将得到的数组命名为 S_{12} ;

对子数组 S_2 进行拆分-----> $[], [8], [13, 10, 11]$ <-----将得到的数组命名为 S_{22} ;

此时的数组 S 为-----> $2, 5, 3, 6, 8, 13, 10, 11$

对子数组 S_{12} 进行拆分----> $[3], [5], []$;

对子数组 S_{22} 进行拆分----> $[10, 11], [13], []$ <-----将得到的数组命名为 S_{221}

此时的数组 S 为-----> $2, 3, 5, 6, 8, 10, 11, 13$

对子数组 S_{221} 进行拆分---> $[], [11], [13]$

最后得到的数组为-----> $2, 3, 5, 6, 8, 10, 11, 13$;

然而，对于整个数组进行快速排序是完全没有必要的。

我们假设求的是第 k 小的数，数组 $A[p \cdots r]$ 被划分成 $A[p \cdots q]$ 和 $A[q+1 \cdots r]$ 两部分，则可以根据左边的元素个数 $q-p+1$ 和 k 的大小关系只在左边和右边递归求解（这里需要注意递归右边的时候找的是第 $k-(q-p+1)$ 小的元素）。

算法分析

在期望意义下，算法的时间复杂度可由下述递归式由主方法求解：

$$T(n) = T(n/2) + O(n)$$

符合《算法导论》中文第三版 54 页提到的条件 3，由此解得 $T(n) = O(n)$

参考代码一（非 STL 版）

```
#include<iostream>
using namespace std;
int a[100007];
//划分问题
//当然也可以使用书上提到的随机化算法
int divide(int l, int r)
{
    int x=a[r];        //选择最右元素作为主元
    int i=l-1;
    /*
    循环体内的循环不变量性质如下：
    每轮迭代开始时，对于任意数组下标 k：
    1. 若  $l \leq k \leq i$ ，则  $a[k] \leq x$ 
    2. 若  $i+1 \leq k \leq j-1$ ，则  $a[k] > x$ 
    3. 若  $k=r$ ，则  $a[k]=x$ 
    */
    for(int j=l;j<r;j++){
        if(a[j]<=x){
            i++;
            //exchange a[i]&a[j]
            swap(a[i],a[j]);
        }
    }
    swap(a[i+1],a[r]);
    return i+1;
}
//快速排序
int Quick_Sort(int head, int tail, int i )
{
    if(head==tail) return a[head];
    else if(head<tail){
        int q = divide(head, tail);
        int k = q-head+1;
        if(i==k) return a[q];
        //下述四行既可以在期望意义下完成一半的快速排序过程，避免不必要操作
        else if(i<k)
            return Quick_Sort(head,q-1,i);
        else
            return Quick_Sort(q+1,tail,i-k);
    }
}
```

```

}
int main()
{
#ifdef ONLINE_JUDGE
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
#endif
    int n;
    while(cin>>n){
        for(int i=0;i<n;i++) cin>>a[i];
        int mid = (n%2==0)? n/2 : n/2+1 ;
        int ans=Quick_Sort(0,n-1,mid);
        cout<<ans<<endl;
    }
}

```

参考代码二（STL 版）

```

#include<cstdio>
#include<algorithm>
#include<iostream>
using namespace std;
int a[100010];
int main()
{
    int k,n;
    while(~scanf("%d",&n)){
        for(int i=0;i<n;i++){
            scanf("%d",&a[i]);
        }
        int mid=(n-1)/2;
        nth_element(a,a+mid,a+n); //这个算法库函数实现思想和上述相同
        printf("%d\n",a[mid]);
    }
}

```

F 四合归零

思路分析

对于本题，想解出来很简单，想使用高效的算法解出来并不是一件特别容易的事（虽然也不是特别难）。

对于本题，我们可以想到如下解题思路：

1.直接枚举计算查找——四重 for 循环，时间复杂度 $O(n^4)$

2.对 D 数组排序，枚举 $a+b+c$ 并对排好序的 D 数组二分查找 $-(a+b+c)$

——这一方法运行时间 $T(n)=O(n\lg n)+O(n^3\lg n)=O(n^3\lg n)$

3.枚举 $a+b$ ，排序，枚举 $c+d$ 并在有序数组中对 $-(c+d)$ 二分查找

我们可以将方法 2 进行推广，推广成这样的算法。这种算法两个步骤的时间复杂度均为 $O(n^2\lg n)$ ，总的时间复杂度也是 $O(n^2\lg n)$ 。

然而，对于本题时间复杂度的要求以及给出的各数组各元素的取值范围，我们还可以这样做：

枚举 $a+b$ ，对所有 $a+b$ 的值进行计数，再枚举 $c+d$ ，查找到 $-(c+d)$ 对应的 $a+b$ 的值的数目并累加。

至于如何计数，由于数组取负值会造成数组下标越界，因此根据数据范围， $a+b$ 的取值范围为 $[-1000000, 1000000]$ 。我们可以开辟一个长度为 2000000 的 int 类型数组 c（该数组需要在每组数据操作前初始化为 0），用于存储 $a+b+1000000$ 出现的次数，再枚举 $c+d$ ，查找到 $c[1000000-c-d]$ 的值并累加即可。

这种算法利用的是计数排序的思想，对于数据范围一定且比较小的大量数而言，这种排序所需要的时间是线性的，也是最优的。由于需要枚举 $a+b$ 和 $c+d$ ，因此该算法的时间复杂度为 $O(n^2)$ 。

参考代码

```
#include<cstdio>
#include<cstring>
#define MID 1000000 //此为偏置值
long long bufa[2000010];
int main()
{
    int a[4010],c[4010];
    int n;
    while(~scanf("%d",&n)){
        long long ans=0;
        memset(bufa,0,sizeof(bufa));
        for(int i=0;i<n;i++){
            scanf("%d",&a[i]);
            int b;
```



```

for(int i=0;i<n;i++){
    scanf("%d",&b);
    for(int j=0;j<n;j++){
        int temp_1=b+a[j]+MID;
        bufa[temp_1]++;          //对 a+b+1000000 进行计数
    }
}
for(int i=0;i<n;i++)
    scanf("%d",&c[i]);
int d;
for(int i=0;i<n;i++){
    scanf("%d",&d);
    for(int j=0;j<n;j++){
        int temp_2=MID-d-c[j];
        ans+=bufa[temp_2];      //查找 1000000-c-d 的值并累加
    }
}
printf("%lld\n",ans);
}
}

```