2016 级算法第二次上机解题报告

15081070 张雨任

一、总结

本次上机主要考查知识点为快排、二分查找、优先队列、数组线性操作等。

关于题目，A 题考察循环的剪枝和数组的灵活使用，B 题考查贪心算法，即局部最优解的选择，C 题考察数组和映射的灵活使用，D 题考察二分，E 题没有太好的思路，我觉得可以考虑使用循环链表来模拟，F 题考察前缀数组和逆向思维，G 题考察快排和分治的思想，也可以使用模拟的方法。关于数组和映射，我觉得思路要灵活一些，数组的下标不一定是编号，映射关系也不一定是编号到结果，也可以作结果到编号的映射。二分法我经常想不到，以后只要是遇到查找的题目，都可以先考虑考虑二分法。贪心的题目有的可以看作是模拟，然后每一步都去选择当前情况的最优解。对于分治法，要理解这个问题的本质，分割成合理可求的子问题，然后再合并。循环的剪枝一般要和排序相关，至少序列中要有一定的规律性，才能够做到不一一遍历而预测某些情况的解，从而做到减少时间复杂度。

二、解题报告

A. 画个圈圈诅咒你

思路分析

本题考察数组的灵活使用和循环的剪枝。

输入圆的圆心坐标和半径，求解不重叠的圆的个数。这里半径和圆心坐标对于计算不重叠没用，而一个圆的左右边界才有用，因为查看两个圆是否完全分开，其实是看前一个圆圆 A 的右边界是否能够小于后一个圆圆 B 的左边界。

算法分析

首先在输入圆的圆心和半径的时候，计算左边界 low 和右边界 high 并存放在数组中。然后对两个数组进行排序，排序对于后面的剪枝很重要。接下来，进入二重 for 循环，分别遍历两个圆的关系，这里就要进行剪枝了。

关于剪枝：要考虑对于前一个圆圆 A 的 high（右边界）小于后一个圆圆 B 的 low（左边

界）时，可以说明 A 和 B 是不相交的，因此对于两个有序数组，先前的遍历一直是 high>low，在某一次迭代不满足的时候，就会说明这两个圆不相交了，那么对于 low 数组的剩余未遍历的圆，因为 low 是升序的，因此就更大于 high 了，因此 low 中剩余也都是不相交的圆，所以计数器累计总共圆的个数减去恰好打破 high>low（相交关系）的 low 的下标值，就是当前 high 代表的圆所拥有的不相交（完全分离）的圆的个数，因此累加后 break。

在 break 跳出之前的操作很关键，下一次 j 的起始位置应为此次跳出时 j 的值。要明确 high 是升序的，对于之前和左边界较小的圆相交的圆，和左边界较大的圆有两种关系：相交和不相交，对于相交的情况，即为左边界较小的圆都和这个圆相交了，就更不用说左边界稍大的圆了，而不相交的情况属于这个圆在当前圆的左边，已经计算过，因此不用考虑。所以下一次 j 的起始位置应为此次跳出时 j 的值。最后输出计数器累加的值即为所求。

时间复杂度：二重循环，看起来时间复杂度为 O(n^2)。仔细分析可发现，内层循环 j 是连续的，即 j 的取值从 0 到 n-1，无论外层迭代几次，j 都是不重复的，在外层循环结束时，内层循环仅迭代 n 次，而内层循环达到 n 之后，外层循环每次遍历仅仅作一个 j 和 n 的比较，因此是线性的时间消耗。综上所述，时间复杂度应为 O(n)。

拓展思考

关于内含问题的思考：

此题不包括内含情况，若包括内含情况则会稍微复杂一些。那么分开的圆圈有两种分开的模式：外离和内含。可以考虑使用结构体来保存圆的信息，其中存储圆心、半径、左端点、右端点。接下来按照圆心从小到大排序。那么外离情况直接使用上面的思路即可，即后圆的左端点大于前圆的右端点；内含的情况则有两种，后圆内含了前圆、前圆内含了后圆。后圆内含前圆要求后圆的左端点大于前圆的左端点。内含很好证明：后圆的圆心坐标大于前圆，而后圆的左端点又大于前圆的左端点，显然后圆的半径大于前圆的半径，那么画出图来即可保证后圆内含了前圆。前圆内含后圆则要求后圆的右边界小于前圆的右边界，此时后圆的半径必小于前圆的半径。

参考代码

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

int n,start,x,r;
```

```
ll cnt;

int low[500004];
int high[500004];

int main() {
    while(~scanf("%d",&n)) {
        for(int i=0;i<n;i++) {
            scanf("%d%d",&x,&r);
            low[i]=x-r;
            high[i]=x+r;
        }
        sort(low,low+n);
        sort(high,high+n);
        cnt=0;
        start=0;
        for(int i=0;i<n;i++) {
            for(int j=start;j<n;j++) {
                if(high[i]<low[j]) {
                    cnt+=(n-j);
                    start=j;
                    break;
                }
            }
        }
        printf("%lld\n",cnt);
    }
}
```

B. Bamboo 的 OS 实验

思路分析

本题可以从广义上的贪心算法来考虑。

当间隔时间为 n 时，为了保证两个相同的值之间间隔为 n，可以每次放置 n+1 个不同元素，这样可以保证相同元素之间间隔为 n。然后每次选取剩余个数最多的 n+1 个元素放置，防止最后剩下过多某相同的元素，而不得不放置过多的“思考人生”时间，造成浪费。

算法分析

首先，输入指令的时候通过累加记录每种指令的个数，因此声明一个数组 insToNum，代表从指令序号到该指令个数的映射。结束输入时得到一个从指令序号到对应指令个数的映射。该数组下标范围为 [1,30]，代表指令编号，对应的值的范围为 (0,1e5)，代表对应的指令个数。

由于设置的时间间隔为 n，因此每次要在 n+1 个连续的位置放置不同的指令编号，以保证在 n 间隔内不出现两个相同编号的指令，为了保证元素数量的平衡，防止最后剩下过多某相同的元素，而不得不放置过多的“思考人生”时间造成浪费，必须选择当前 insToNum 数组前 n+1 个最大的放置在当前 n+1 个空位中，然后把这些值减 1，如果是 0 就不减了，并且当作放置了一次“思考人生”时间。这样每次增加的总长度为 n+1，但还要考虑特殊情况。最后一轮，如果 insToNum 数组中最大的都为 0 的话，说明此时已经结束，数组中的所有元素都是 0 了，但是此处不能直接加 n+1，因为此时很可能本身可放置的指令就已经不足 n+1 个，因此要在遍历的时候记录可以放置（也就是非零）的指令的个数，最后把累加的结果加在答案上即可。

此题贪心在于每次都要选择数组中指令剩余个数最多的 n+1 个，相当于选择当前状况下的局部最优解。时间复杂度为 O(n^2)。

参考代码

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

int data[100005];//index:[0,1e5) val:[1,30]
int insToNum[35];//index:[1,30] val:(0,1e5)
```

```
int x,n,ans,cnt,ins;

bool cmp(int a,int b) {
    return a>b;
}

int main() {
    while(~scanf("%d",&x)) {
        memset(insToNum,0,sizeof(insToNum));
        for(int i=0;i<x;i++) {
            scanf("%d",&ins);
            insToNum[ins]++;
        }
        scanf("%d",&n);
        ans=0;
        sort(insToNum+1,insToNum+31,cmp);
        while(1) {
            cnt=0;
            if(insToNum[1]==0) break;
            for(int i=1;i<=n+1;i++) {
                if(insToNum[i]!=0) {
                    insToNum[i]--;
                    cnt++;
                }
            }
            sort(insToNum+1,insToNum+31,cmp);
            if(insToNum[1]!=0) ans+=(n+1);
            else ans+=cnt;
        }
        printf("%d\n",ans);
    }
}
```

C. AlvinZH 的儿时梦想——坦克篇

思路分析

本题考察数组和映射的灵活使用。

给出的数据是每块墙的长度，这些数据的特点是每行数据的个数是相同的，每行的数据累加的和也是相同的。这些数据无法直接使用，但是转换成坐标就可以体现出相互的关系。大体思路是通过墙的长度来确定建筑之间的“空隙”的位置，然后就可以基于“空隙”的位置来计算从这个方位穿过走过的“空隙”的数量，从而再用总长度减去“空隙”的数量即可得到穿过建筑物的个数。

算法分析

首先读入数据，通过累加的方式来计算每个空隙的位置（坐标），比如某行第一个建筑物的长度为 2，那么一定存在横坐标为 2 的一条空隙，第二个建筑物长度为 4，那么第二个空隙的坐标为 2+4=6。这样，就可以得到所有空隙所对应的横坐标。由于要计算相同横坐标的空隙的个数，这样就形成了一个从间隙横坐标到间隙个数的映射，这里不能使用数组，因为数组的元素个数不能开到 int 那么大。这里通过累计 sum，sum 代表当前行的横坐标，时时累加更新映射关系，累加到当前 sum，就在 sum（当前缝隙的横坐标）到“间隙”个数的映射上自增 1，代表该横坐标多出一个缝隙。由于要计算穿过建筑物最少的个数，因此可以转化为计算穿过间隙最多的个数，因为每一列间隙加上墙的个数是常量 m。所以每次计算了一次 sum 到间隙个数的映射，就更新一次最大值，保证 ans 中时刻存储着映射关系的最大值（即缝隙的最大值）。当结束之后，输出 m-ans 即可（m 为行数，即每一列间隙加上建筑个数的和）。

坑点：更新间隙坐标的时候不能累加最后一个建筑，累加的话就会造成误把最右边当作间隙，但题目中说过这条路不能走，因此累加的时候不可以累加最后一个建筑的长度。即保证每个空隙的横坐标都是合法的。到边界做检查处理，但是代码没有解决 m=1 的情况（即每行一个建筑物），所以 1ms 的样例没过。比较好的处理方式是每列的最后一个数据正常读入，但是不累加到 sum 中。map 操作的时间复杂度是 O(lgN)，因此此题的时间复杂度 O(n*m*lg(m*n))。

关于数组和映射的一点想法，数组其实也属于一种映射，映射关系既可以从 a 到 b，也可以从 b 到 a，解题的时候不能死守着某一种想法，可以有下标到值的映射，也可以有值到下标的映射。甚至很大的 int 或者非 int 类型也可以通过 map 进行映射。

参考代码

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

int m,n,flag,sum,ans;

int main() {
    while(~scanf("%d%d",&m,&n)) {
        map<int,int> check;
        ans=0;
        for(int i=1;i<=m;i++) {
            sum=0;
            for(int j=1;j<=n;j++) {
                scanf("%d",&flag);
                if(j==n) break;
                sum+=flag;
                check[sum]++;
                ans=(ans>check[sum]?ans:check[sum]);
            }
        }
        printf("%d\n",m-ans);
    }
}
```

D. Bamboo 的饼干

思路分析

本题考查二分查找。

基本题意是给定两行数，求两行数各取一个值相加得到 t 的数对。最基本的想法是暴力 O(n^2)。这里还可以使用二分查找可以降低时间复杂度，做法是：按住数组 r1 的一个值 r1[i]，在有序数组 r2 中二分查找 t-r1[i]，如果找到，即是所求，没找到则更换 r1[i]。

算法分析

首先，对两个无序数组排序，由于结果要求按照 r1 从小到大的顺序输出数对，因此线性遍历排序好的数组 r1，对于每个 r1[i]，求得 t-r1[i]作为 key，在另一个有序数组 r2 中二分查找，找到则输出（由于题目要求按 r1 升序输出，此处 r1 为升序，因此可以输出，没有顺序问题），并置标记 found 为 1。如果结束遍历 r1，found 仍为 0，则说明没有找到匹配的数对，输出“OTZ”。

其次，由于不能输出重复数对，因此要设立查重机制，方法有很多。图方便可以直接使用 map 映射，一开始都设为 0，每输出一个数对就置为 1，线性遍历 r1 数组时，先检查当前 r1[i] 经过 map 映射得到的结果是多少，是 1 则说明输出过数对 (r1[i], t-r1[i])，则作 continue 处理。当然，也可以设立一个值 tmp 来保存上次找到符合数对时的 r[i]值，每次线性遍历 r1 的时候提前检查，比 map 更节省时间。

线性遍历 r1 的时间复杂度为 O(n)，二分查找的时间复杂度为 O(lgn)，因此此题的时间复杂度为 O(nlgn)。

拓展思考

二分法的应用：

① 计算一个数的平方根，仅返回整数。计算 x 的平方根实际上是寻找满足等式 mid*mid <= x <= (mid+1)*(mid+1) 的 mid，因此可以使用二分查找

② 计算单调区间的极值。二分法的根本在于使用了区间单调的性质，因此遇到了单调序列或区间内单调的题可以来考虑使用二分法来解决。重点在于区间的单调性。

③ 有关查找或匹配的题可以考虑二分法。

参考代码

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

const int maxn=1e9;
int n,t,cnt;
int r1[100005];
int r2[100005];

int main() {
    while(~scanf("%d",&n)) {
        for(int i=0;i<n;i++) {
            scanf("%d",&r1[i]);
        }
        for(int i=0;i<n;i++) {
            scanf("%d",&r2[i]);
        }
        map<int,bool> check;
        sort(r1,r1+n);
        sort(r2,r2+n);
        scanf("%d",&t);
        bool found=false;
        for(int i=0;i<n;i++) {
            int x=t-r1[i];
            if(check[r1[i]]) continue;
            int l=0,r=n-1;
            while(l<=r) {
                int mid=(l+r)/2;
                if(r2[mid]<x) {
                    l=mid+1;
                }
                else if(r2[mid]>x) {
                    r=mid-1;
                }
                else {
                    found=true;
```

```
                printf("%d %d\n",r1[i],x);
                check[r1[i]]=1;
                break;
            }
        }
    }
    if(!found) printf("OTZ\n");
    printf("\n");
  }
}
```

F. ModricWang's Number Theory II

思路分析

本题考查前缀数组的应用，是一道非常灵活的题。

此题看似要求一组数的 gcd，但是如果每组数列都求 gcd 时间可能会爆炸，而且还有那么多可能的变换方式，所以不如反过来想，遍历所有可能的 gcd 的情况，在该 gcd 下求得最优的数列变换方式，在该 gcd 下，数列的每个元素必须为该 gcd 的倍数，然后通过分析每个数应该变到该 gcd 的多少倍，来计算最优解。这里涉及到了区间问题，一般来讲，区间问题有时可以很巧妙的使用前缀数组来解决。

算法分析

首先录入数列中的数，在录入的时候进行累加，使用两个数组，num 和 sum，num [i]用来记录等于 i 的数的出现次数，sum[i] 用来记录等于 i 的数的和，相当于 i*num[i]。接下来来处理前缀数组，遍历一遍两个数组，通过累加来改变 num 和 sum 的定义，num[i] 定义为小于等于 i 的数的出现次数，sum[i] 定义为小于等于 i 的数的加和。前缀数组的优点在于可以很好的计算区间问题，用这里的 num 数组举例：num[i] 定义为小于等于 i 的数的出现次数，num[j] 定义为小于等于 j 的数的出现次数，那么要计算小于等于 i，大于 j 的数的出现次数就可以用 num[i]-num[j]来表示，非常方便，同理这里 sum 也可以通过这样的方式来计算数列中某区间的值的加和。再比如求序列的连续子序列的异或和也可以使用前缀数组来解决。当然前缀数组不一定能够解决所有区间问题，还是要具体情况具体分析。

接下来来遍历 gcd，每个数都有可能是 gcd，所以 gcd 的取值从 1 到 1e6（左右均可取到），然后来看对于数列中在区间 [t*gcd+1, (t+1)*gcd] 的某个数，应该选择删除，还是选择自增到右边界。

关于删除和自增到右边界的选择，这里涉及到 x 和 y，x 为删掉一个数的代价，y 为给一个数自增 1 的代价。这里来对这两个操作进行分析，首先，删掉一个数是最“省心”的，给一个数自增不仅变化很有限，而且可能还需要付出一定的代价。其次，这里在选择操作的时候，应该这样想：对于当前的某数 z，处于区间 [t*gcd+1, (t+1)*gcd] 中，z 面临着选择删除还是自增到右边界 (t+1)*gcd 的问题，假设 z 通过自增加到右边界需要 lmt 次自增操作，总共的消耗为 lmt*y，而 z 也可以选择一次删除操作，消耗掉 x，两个操作的效果都是使得这个数列减少一个非此 gcd 倍数的数，因此就要看 lmt*y 和 x 的大小关系，哪个小就选哪个，因此可以算出 lmt=x/y。

lmt 的作用：可以划分区间，在一个区间中，距离右边界为 lmt 的都可以选择自增，因为代价要比删除小，因此变量 bnd 代表在本区间中的划分位置的值，在区间 [t*gcd+1, bnd-1] 的值应选择删除，因为自增的代价太大了，而在区间 [bnd, (t+1)*gcd] 的值应选择自增到右边界，因为对他们来说删除的代价太大了。但是要考虑 lmt 大于区间长度的情况（每个区间的长度都是 gcd），lmt 较大说明删除的代价比自增的代价多了很多，此时的 bnd 即为区间的左边界，即此区间所有在数列中的值都应该选择自增，因为删除的代价过大。

而在每个小区间内，要计算该小区间的代价和，根据上面的分析，先明确：数列中位于 [t*gcd+1, bnd-1] 的数要做删除操作，数列中位于 [bnd, (t+1)*gcd] 的数要做自增操作。这里需要使用前缀数组的知识，先计算删除的代价和，这个比较简单，直接用 x 乘以需要做删除操作的元素个数，根据前缀数组的知识，区间中要做删除操作的元素个数等于 num[bnd-1]-num[t*gcd+1] ；然后计算自增的代价和，首先明确所有要自增的值都要自增到 (t+1)*gcd 以保证成为 gcd 的倍数，对于数 p，需要自增 (t+1)*gcd-p 次，消耗 ((t+1)*gcd-p)*y，这里计算所有要自增操作的数的消耗，可以直接计算这些数的和，然后再用目标和减去他们的和即可。假设有 k 个数要自增，那么目标和为 k*(t+1)*gcd，要自增的数的和可以使用前缀数组来计算，sum[i] 定义为小于等于 i 的数的和，那么区间 [bnd, (t+1)*gcd] 的和就可以表示为 sum[(t+1)*gcd]-sum[bnd]，然后再用 k*(t+1)*gcd 减去 sum[(t+1)*gcd]-sum[bnd]即得到数列中所有在该区间的数需要自增的总操作次数，然后再乘以 y 即可得到消耗。最后把删除的消耗加上自增的消耗，即得到输入数列处于该区间中的数为了满足条件而做出的最小消耗。

以上是一个区间的情况，对于 gcd 来说，要把所有区间的情况都计算出来，即计算 [t*gcd+1, (t+1)*gcd] （t 取所有自然数）的总消耗，就得到了当前 gcd 的总最小消耗，然后遍历所有可能的 gcd，记录这些 gcd 中消耗最小的消耗值即可。

这种做法比较暴力，思路也比较易懂，时间复杂度 O(n^2)。

参考代码

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

const int maxn=3e6+7;
int n;//[1,5e5]
ll x;//[1,1e9]
ll y;//[1,1e9]
ll a;//[1,1e6]
```

```
ll num[maxn];
ll sum[maxn];
ll tmp;
ll lmt;
ll bnd;
ll ans=1e25;

int main() {
    scanf("%lld",&n);
    memset(num,0,sizeof(num));
    memset(sum,0,sizeof(sum));
    scanf("%lld%lld",&x,&y);
    for(int i=0;i<n;i++) {
        scanf("%lld",&a);
        num[a]++;
        sum[a]+=a;
    }
    for(int i=1;i<=2e6;i++) {
        num[i]+=num[i-1];
        sum[i]+=sum[i-1];
    }
    lmt=x/y;
    for(int i=2;i<=1e6;i++) {
        tmp=0;
        for(int j=i;j<2e6;j+=i) {
            if(i-1<lmt) bnd=j-i+1;
            else bnd=j-lmt;
            tmp+=((num[bnd-1]-num[j-i])*x+
                 ((num[j]-num[bnd-1])*j-(sum[j]-sum[bnd-1]))*y);
        }
        if(ans>tmp) ans=tmp;
    }
    printf("%lld\n",ans);

}
```

G. ModricWang's Real QuickSort

思路分析

本题考查分治思想和快速排序的应用、也可以使用纯模拟的方法解决。

思路一（分治）：

通过分治法进行快排，输出对应趟数对应部分的序列。划分的时候要注意题中的操作，注意小于还是小于等于之类的细节。对于趟数的计算，可以使用 cnt 来进行标记，cnt 达到一定的值即可确认当前是第几趟第几部分了。

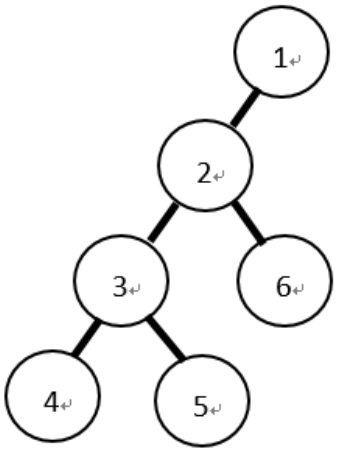
思路二（模拟）：

也可以使用非递归的方法解决，直接模拟划分的过程。做两次划分即可得到第二次递归的数组，输出对应的子序列即可。首先划分整个数组，得到中轴数的下标 x，然后再划分 0 到 x-1 的子数组，得到 y，到 x-1 即为所求。

算法分析

思路一（分治）：

首先看一下输出，要输出第二趟的第二部分，这个很容易出错，容易输出其他趟数其他部分的元素，或者上下界多一少一，可以使用计数器来进行数组部分的标记，每次递归前进行累加，代表现在在操作第几部分的元素，递归树（部分）如图（结点的数字为标记）：



只要大于 3 就判断：如果到了 4 和 5，就返回，这样才能够保证计数器为 6 的时候输出的是第二趟第二部分的元素。因此输出第二趟第二部分即等价于输出计数器为 6 时的 i 和 j 之

间的数。

这次的题和上次有不同之处。按照题中划分的操作步骤严格模拟整个划分的过程（否则虽然排序没问题，但是具体每次递归的元素可能与题意不同，导致输出不同），循环中要维护“pivot 左边的数都比 pivot 严格小，pivot 右边的数都比 pivot 严格大”，因此每次都要把 pivot 左边比 pivot 大的数和 pivot 右边比 pivot 小的数交换，具体步骤分析见代码注释。除了要注意的数组下标上下界的多一少一，还要额外要注意 mid（分隔元素 pivot 的下标）的取值，应为 (low+high)/2+1，因为数列元素个数为偶数个时选择后一个作为分隔元素。注意题干中的小于和小于等于。快排的时间复杂度为 O(nlgn)。

思路二（模拟）：

首先划分整个数组，得到中轴数的下标 x，代表连个数组中，后一个数组的起始位置为 x，则上一个数组（即第一趟第一部分）结束位置为 x-1，因此再划分 0 到 x-1 的子数组，得到 y，同理，y 为第二趟第二部分的起始位置，因此从 y 到 x-1 即为所求。此处不再赘述划分的注意事项和解释。

参考代码一（分治）

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

int n,cnt;
int data[1000007];

void partition(int val[],int low,int high,int &cnt) {
    if(cnt>3) {
        if(cnt==6) {
            for(int i=low;i<=high;i++) printf("%d ",val[i]);
            printf("\n");
            return;
        }
        else return;
    }
    else {
        if(low<high) {
            int i=low;//对应步骤 1
            int j=high;//对应步骤 1
            int pivot=val[(i+j)/2+1];//对应步骤 1
```

```
            while(i<=j) {//对应步骤 3，每次比较 i 和 j 的大小
                while(i<j && val[i]<pivot) i++;//对应步骤 4
                while(i<j && val[j]>pivot) j--;//对应步骤 5
                if(i<=j) swap(val[i],val[j]);
                //对应步骤 6，交换两指针指向的值，并循环
            }//对应步骤 7，退出
            partition(val,low,i-1,++cnt);//分治
            partition(val,i,high,++cnt);//分治
        }
    }
}

int main() {
    while(~scanf("%d",&n)) {
        for(int i=0;i<n;i++) scanf("%d",&data[i]);
        cnt=1;
        partition(data,0,n-1,cnt);

    }

}
```

参考代码二（模拟）

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

int n;
int arr[1000005];

int partition1(int l,int r) {
    int i=l;
    int j=r;
    int mid=arr[(i+j)/2+1];
    while(i<=j) {
        while(i<j && arr[i]<mid) i++;
        while(i<j && arr[j]>mid) j--;
        if(i<=j) {
            swap(arr[i],arr[j]);
            i++;
            j--;
```

```
        }
    }
    return i;
}

int main() {
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&arr[i]);
    int x=partition1(0,n-1);
    int y=partition1(0,x-1);
    for(int i=y;i<=x-1;i++) {
        printf("%d ",arr[i]);
    }
    printf("\n");
}
```