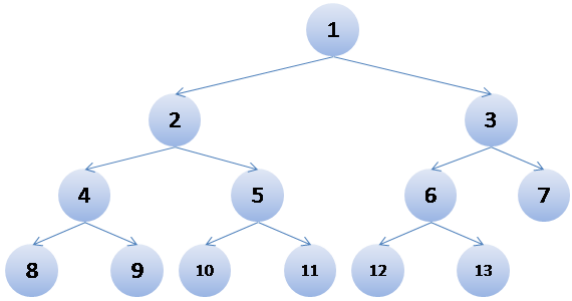
优先队列

优先队列是一个以集合为基础的抽象数据结构，队列中的每个元素都有一个优先级，譬如元素值愈小，优先级愈高。优先队列不同于先前讲的“先进先出”的队列，因为其服务不是按照排队的先后顺序进行的，而是按照每个对象的优先顺序进行的，即“优先级高的先出”。

优先队列是通过二叉堆来进行维护的，我们先来了解关于二叉堆的知识。

二叉堆（heap）是一棵满足下列性质的完全二叉树：
（1）如果某节点有孩子，且根节点的值都小于孩子节点的值，我们称之为小根堆，如下图。
（2）如果某节点有孩子，且根节点的值都大于孩子节点的值，我们称之为大根堆。



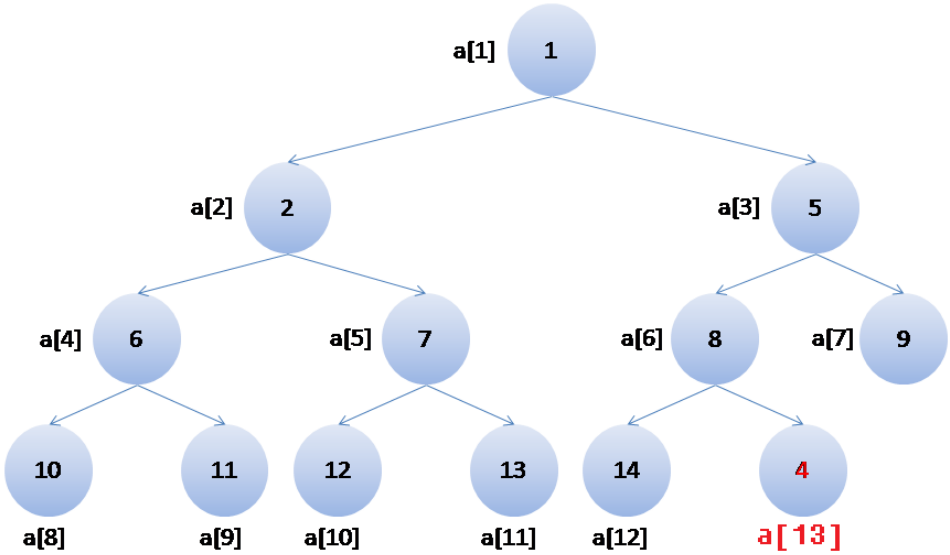
显然，小根堆的根节点heap[1]是最小值，大根堆的根节点heap[1]是最大值。

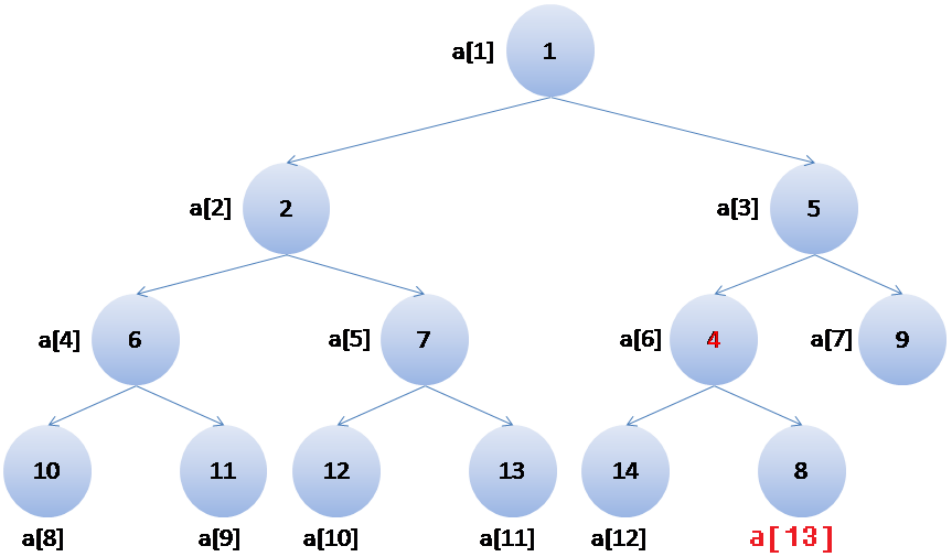
二叉堆的树结构同其他完全二叉树一样，所有叶子在同一层或者两个连续层上，最后一层的节点占据尽量左的位置。因此二叉堆的元素可保存在一维数组heap[1..n]中，其中heap[i]节点的父节点是heap[i div 2]，左子节点是heap[2 * i]，右子节点是heap[2 * i+1]，若2*i>n，则节点i为叶节点。

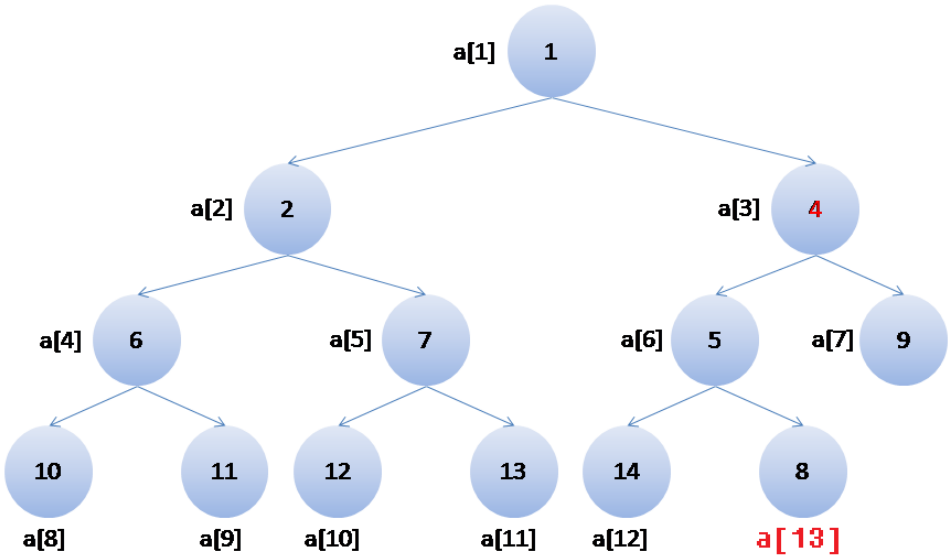
在插入或删除操作前，完全二叉树保持着堆性质。但插入或删除一个节点后，堆的性质就被破坏了，需要通过调整来恢复堆性质，这就是堆的动态维护。下面以小跟堆为例，了解插入和删除节点的维护过程。

【插入节点】

先将插入元素x添加到堆尾，再向上调整：比较当前节点p和父节点值，若父节点的值大于等于p节点的值，则将父节点与p交换，继续向上调整，直至父节点值比p小为止。显然，此时的完全二叉树恢复了小根堆性质。时间复杂度为O(logn)。


a[1] 1
a[2] 2
a[3] 5
a[4] 6
a[5] 7
a[6] 8
a[7] 9
10
11
12
13
14
4
a[8]
a[9]
a[10]
a[11]
a[12]
a[13]


a[1] 1
a[2] 2
a[3] 5
a[4] 6
a[5] 7
a[6] 4
a[7] 9
10
11
12
13
14
8
a[8]
a[9]
a[10]
a[11]
a[12]
a[13]


a[1] 1
a[2] 2
a[3] 4
a[4] 6
a[5] 7
a[6] 5
a[7] 9
10
11
12
13
14
8
a[8]
a[9]
a[10]
a[11]
a[12]
a[13]

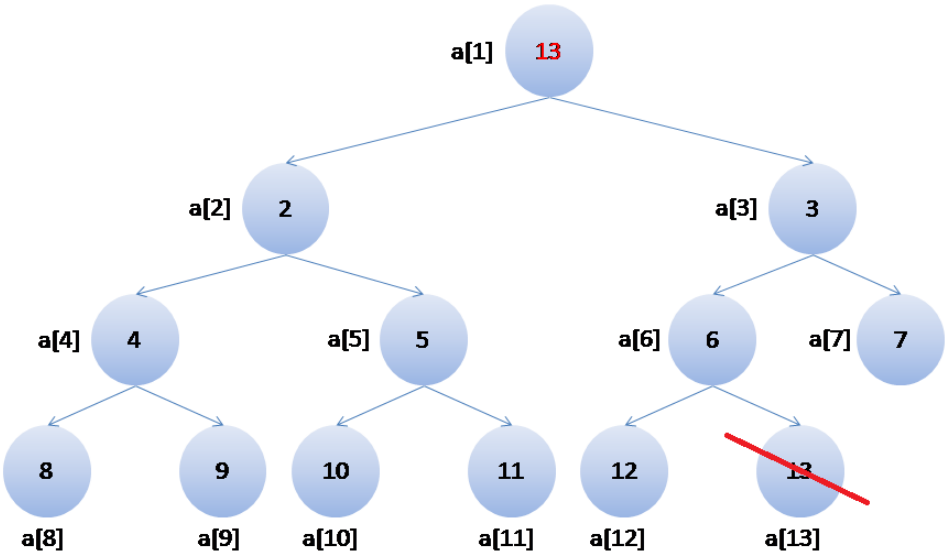
代码实现如下：

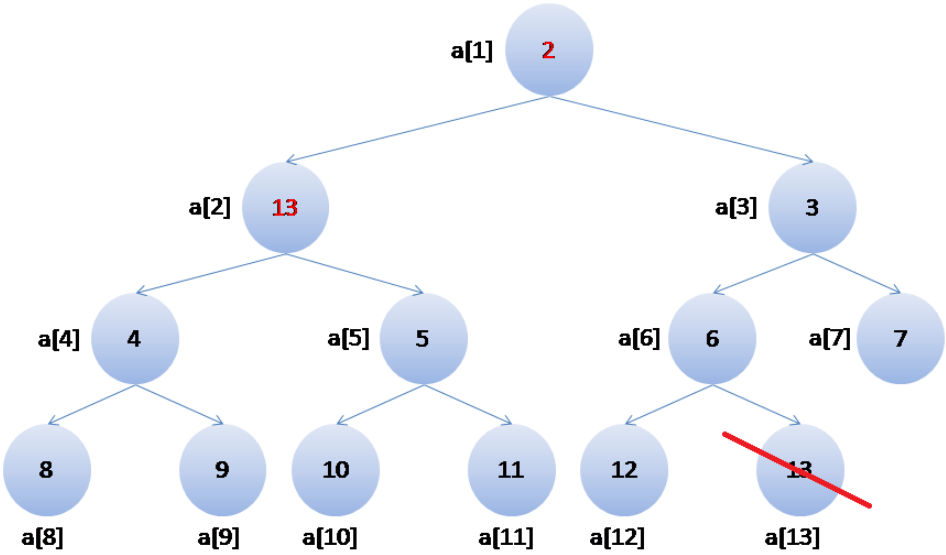
```
void push(int x){
    heap[++len]=x;
    int p=len;
    while(p>0&&heap[p/2]>heap[p]){
        int t=heap[p];
        heap[p]=heap[p/2];
        heap[p/2]=t;
        p=p/2;
    }
}
```

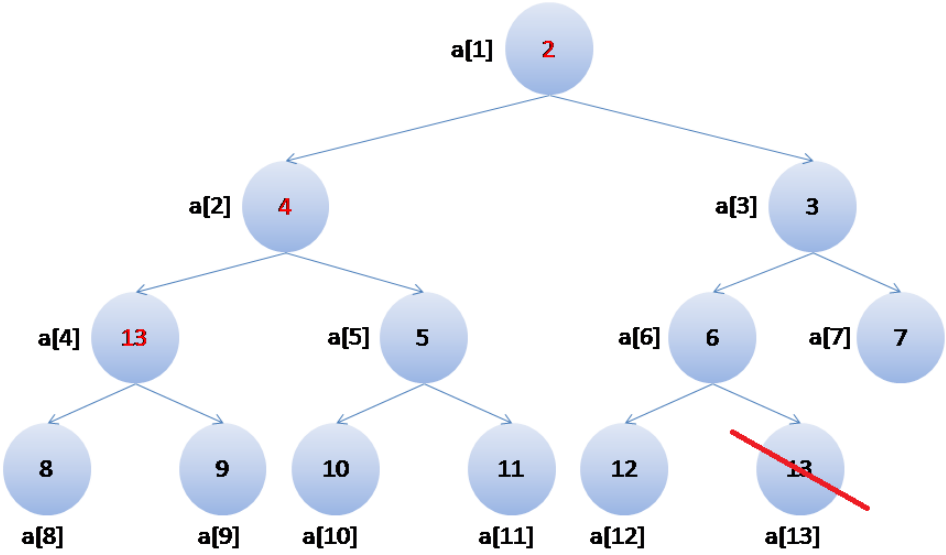
【删除根节点】

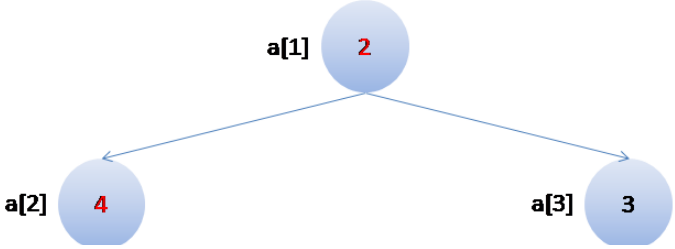
在堆中删除根heap[1]，即删除最小值元素。可先用heap[n]覆盖heap[1]，堆长n减1。但此时由于根的调整，破坏了小根堆性质，因此需要向下调整根，调整过程如下：

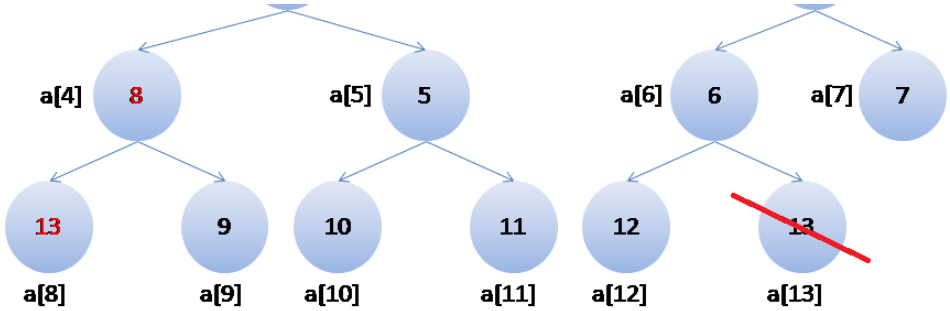
（1）找出当前节点s的左右子节点中权值较小者，记其位置为j。
（2）将heap[j]与heap[s]比较：若heap[s]的权值小于等于heap[j]的权值，则由于完全二叉树恢复了小根堆性质而结束调整。否则交换heap[s]、heap[j]的值，s赋值为j，继续从s出发向下调整。


a[1] 13
a[2] 2
a[3] 3
a[4] 4
a[5] 5
a[6] 6
a[7] 7
8
9
10
11
12
13
a[8]
a[9]
a[10]
a[11]
a[12]
a[13]


a[1] 2
a[2] 13
a[3] 3
a[4] 4
a[5] 5
a[6] 6
a[7] 7
8
9
10
11
12
13
a[8]
a[9]
a[10]
a[11]
a[12]
a[13]


a[1] 2
a[2] 4
a[3] 3
a[4] 13
a[5] 5
a[6] 6
a[7] 7
8
9
10
11
12
13
a[8]
a[9]
a[10]
a[11]
a[12]
a[13]


a[1] 2
a[2] 4
a[3] 3



代码实现如下：

```
void pop(){
    heap[1]=heap[len--];
    int p=1,t=-1;
    while(1){
        if(p*2<=len&&heap[p*2]<heap[p])t=p*2;
        if(p*2+1<=len&&heap[p*2+1]<heap[t])t=p*2+1;
        //定位出左右儿子（若存在）中较小一个
        if(t==-1)break;
        if(t!=-1){
            swap(heap[p],heap[t]);
            p=t;
            t=-1;
        }
    }
}
```

通过以上认识，可见用二叉堆维护优先队列既合适又效率高。

Priority_queue 的应用

Priority_queue是STL所提供的一个非常有效的容器。作为队列的一个延伸，优先队列包含在头文件中。

定义方式为：

```
priority_queue<Type, Container, Functional>
```

其中Type 为数据类型， Container 为保存数据的容器，Functional 为元素比较方式。
Container 必须是用数组实现的容器，比如 vector、 deque 但不能用 list。
STL里面默认用的是 vector，比较的方式默认用 operator<。 所以如果把后面俩个参数缺省，优先队列就是大根堆，即队头元素为最大值。如果要用到小根堆，则一般要把模板的三个参数都带进去。
STL里面定义了一个仿函数 greater<>，对于基本类型可以用这个仿函数声明小根堆。

例：

```
Priotity_queue<int> bq;     //声明大根堆
Priotity_queue<int,vector<int>,greater<int> > sq;//声明小根堆
```

Priority_queue的常用操作 ：

函数名	功能
top()	返回队列头部数据
push(elem)	在队列尾部增加elem数据
pop()	队列头部数据出队
empty()	判断队列是否为空
size()	返回队列中数据的个数

【例题】输入n个数字，使用优先队列由小到大输出。

```
#include <iostream>
#include <queue>
using namespace std;
int n,x;
priority_queue<int, vector<int>, greater<int> > q;//定义小根堆
int main(){
    cin>>n;
    for( int i= 0; i<n; ++i ){
        cin>>x;
        q.push(x);//插入元素
    }
    while( !q.empty() ){
        cout << q.top() << endl;//输出队头值
        q.pop();//弹出队头
    }
}
```

但是在优先队列的应用中，用于处理结构体数据的情况更为常见。那么，该如何处理排序的规则呢？

【例题】输入n个元素，每个元素有{x，y}两个参数，要求按x的值由小到大排序后输出。若x值相等，则y值小的优先输出。

方法一：重载运算符operator<

```
#include <iostream>
#include <queue>
using namespace std;
int n,x,y;
struct Node{
    int x,y;
}a[1001];
priority_queue<Node> q;
bool operator<(Node a,Node b){
    if(a.x==b.x) return a.y>b.y;
    return a.x>b.x;
}
int main(){
    cin>>n;
    for(int i=0;i<n;++i){
        cin>>x>>y;
        q.push((Node){x,y});
    }
    while( !q.empty() ){
```

```
        cout<<q.top().x<<' '<<q.top().y<<endl;
        q.pop();
    }
}
```

其中：

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

这个部分就是重载运算符的操作，注意这里使用的是大于号，似乎跟题意的要求相悖，但该代码却能按照题意要求正确输出。因此，要注意在优先队列中关于排序优先级的定义与我们实际要求的理解是相反的，这样才能正确处理优先级。另，如此书写代码较冗长，另一种写法更为多见：

```
#include <iostream>
#include <queue>
using namespace std;
int n,x,y;
struct Node{
    int x,y;
    bool operator<(Node k)const{
        if(x==k.x) return y>k.y;
        return x>k.x;
    }
}a[1001];
priority_queue<Node> q;
int main(){
    cin>>n;
    for(int i=0;i<n;++i){
        cin>>x>>y;
        q.push((Node){x,y});
    }
    while( !q.empty() ){
        cout<<q.top().x<<' '<<q.top().y<<endl;
        q.pop();
    }
}

```

即，在结构体的定义中直接重载运算符。这种写法也可用于一般sort排序函数中，但是在sort中排序的规则是与题意理解一致的。要注意区分，正确书写。

方法二：自定义仿函数

即另外声明优先队列中的优先级别定义，类似于在sort函数中声明cmp函数的定义。

```
#include <iostream>
#include <queue>
using namespace std;
int n,x,y;
struct Node{
    int x,y;
}a[1001];
struct cmp{
```

```
    bool operator()(Node a,Node b){
        if(a.x==b.x)return a.y>b.y;
        return a.x>b.x; }
};
priority_queue<Node,vector<Node>,cmp> q;
int main(){
    cin>>n;
    for(int i=0;i<n;++i){
        cin>>x>>y;
        q.push((Node){x,y});
    }
    while( !q.empty() ){
        cout<<q.top().x<<' '<<q.top().y<<endl;
        q.pop();
    }
}

```

A.懒羊羊吃草

【题目描述】

众所周知，懒羊羊是所有小羊里最贪吃的一只。然而，鲜为人知的是，懒羊羊也有存储粮食的习惯。而更让大家吃惊的事实是，我们的懒羊羊做事很有条理，每当他存储一份粮食时，他会专门拿出一个筐来存放。因此，他的仓库里有很多很多筐的青草。而我们的懒羊羊又是一个经常馋嘴的小羊，每当他想吃草时，就会从仓库里找出数量最少的一筐草，把它吃掉。可是懒羊羊因为草吃得太多了导致大脑运转缓慢，所以他不得不向你请求支援，帮他找出他应该吃数量为多少的青草。

【输入】

第一行为一个正整数n，表示懒羊羊一共进行了n次操作($2 <= n <= 1000000$)
第二行至第n+1行每行表示一个懒羊羊的操作，当这行形式为 单独一个字符'q' 时，表示懒羊羊肚子饿了，要吃掉仓库里当前数量最少的那份青草；当这行形式为一个字符'i' 和一个整数k时，表示懒羊羊将一份数量为k(1<=k<=maxlongint=2147483647，maxlongint是pascal语言中最大的长整型数，长整型数即longint)的青草存入了仓库，'i'和k之间用空格隔开。
输入数据保证每次询问时仓库里都有草可吃且所有操作中懒羊羊至少会吃一次草。

【输出】

每当输入为'q' 时， 输出懒羊羊当前吃掉的那份青草的数量是多少。

样例输入	样例输出
5 i 5 i 2 q i 9 q	2 5

【提示】

[样例解释]
共有5次操作，分别为懒羊羊存入数量为5的青草，存入数量为2的青草，吃掉当前数量最少的青草(2)，存入数量为9的青草，吃掉当前数量最少的青草(5)。

[数据规模]
30%数据满足$1 <= p <= 3000$；
60%数据满足$1 <= p <= 40000$；
100%数据满足$1 <= p <= 1000000$。

【分析】

本题为优先队列的基础应用，使用的是小根堆。注意需采用标准化输入输出。

【代码】

```
#include<cstdio>
#include<queue>
using namespace std;
int n,x;
char ch[2];
priority_queue<int,vector<int>,greater<int> > q;
int main(){
    scanf("%d",&n);
    while(n--){
        scanf("%s",&ch);
        if(ch[0]=='i'){
            scanf("%d",&x);
            q.push(x);
        }else{
            printf("%d\n",q.top());
            q.pop();
        }
    }
}
```

B.合并果子

【题目描述】

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。
每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过n-1次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。
因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。
例如有3种果子，数目依次为1，2，9。可以先将1、2堆合并，新堆数目为3，耗费体力为3。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为12，耗费体力为12。所以多多总共耗费体力=3+12=15。可以证明15为最小的体力耗费值。

【输入】

输入文件fruit.in包括两行，第一行是一个整数n(1<=n<=10000)，表示果子的种类数。第二行包含n个整数，用空格分隔，第i个整数ai(1<=ai<=20000)是第i种果子的数目。

【输出】

输出文件fruit.out包括一行，这一行只包含一个整数，也就是最小的体力耗费值。输入数据保证这个值小于2^31。

样例输入	样例输出
3 1 2 9	15

【提示】

[数据规模]
对于30%的数据，保证有$n <= 1000$;
对于50%的数据，保证有$n <= 5000$;
对于全部的数据，保证有$n <= 10000$。

【分析】

在若干堆果子的合并中，几乎所有的堆都需要重复搬动，因此解题的思路为先搬动重量小的去合并，即重量大的尽量少搬动。具体实现方法为每次挑出所有堆中最小的两堆，合并后成为一个新的元素代替这两者参与到下次决策的判断，直至只剩下一堆。

【代码】

```
#include<iostream>
#include<queue>
using namespace std;
long long x,n,ans;
priority_queue<long long,vector<long long>,greater<long long> > q;
int main(){
    cin>>n;
    while(n--){
        cin>>x;
        q.push(x);
    }
    while(q.size()>1){
        int a=q.top();
        q.pop();
        int b=q.top();
        q.pop();
        ans+=a+b;
        q.push(a+b);
    }
    cout<<ans;
}
```

C.Argus

【题目描述】

一个数据流是一个实时，连续，有序的条目序列。一些实例包括传感器数据，互联网贸易，金融报价，网上拍卖，交易日志，Web使用日志和电话呼叫记录。同样，在数据流上的查询每隔一定时间就要连续运行，在产生新的数据的时候就要产生新的结果。例如，一家工厂的仓库的温度检测系统可以执行如下的查询：
查询- 1：“每五分钟，检索在过去5分钟内的最高温度。”
查询- 2：“返回在过去10分钟各个楼层的平均气温。”
我们开发了一个名为Argus的数据流管理系统，以处理在数据流上的查询。用户可以在Argus上登记查询。 Argus将在不断变化的数据上持续执行查询，并以要求的频率向相应的用户返回结果。
对Argus，我们以下述的指令来登记查询：
Register Q_num Period

Q_num (0 < Q_num <= 3000)是查询的ID编号，Period (0 < Period <= 3000)是两个连续的查询结果返回之间的时间间隔。在登记了Period秒之后，首次返回结果，此后，每隔Period秒返回结果。
在Argus上登记了几个不同的查询，所有的查询都有不同的Q_num。你的任务是给出前K个返回结果的查询。如果两个或多个查询同时返回结果，则将他们按照Q_num的升序进行排列。

【输入】

输入的第一部分是在Argus上登记的指令，一条指令占一行，假定指令的编号不超过1000，并且所有的指令同时开始执行。这一部分的结束用"#"表示。
第二部分是你的工作，只有一行，给出一个正整数K ($K <= 10000$)。

【输出】

输出前K个返回结果的查询的Q_num，每个数字一行。

样例输入	样例输出
Register 2004 200 Register 2005 300 # 5	2004 2005 2004 2004 2005

【分析】

本题可通过优先队列的维护，通过堆排序快速找出当前队列中最早发生的指令编号。但要注意的是，每个编号的查询有多次产生的时刻，即有多次按照时间间隔重复入队的情况。那么就在每次队头弹出的时候，计算出它的下次发生时间重新入队参与堆排序。

【代码】

```
#include<iostream>
#include<queue>
#include<string>
using namespace std;
string s;
struct node{
    int id,step,timen;
    bool operator<(node h) const{
        if(timen==h.timen) return id>h.id;
        else return timen>h.timen;
    }
};
priority_queue<node> q;
int x,y,n;
int main(){
    while(cin>>s){
        if(s=="#") break;
        cin>>x>>y;
        q.push((node){x,y,y});
    }
    cin>>n;
    while(n--){
        node f=q.top();
        q.pop();
        cout<<f.id<<endl;
        q.push((node){f.id,f.step,f.timen+f.step});
```

```
    }
}
```

D.猴子除草

【题目描述】

猴子们有一个很大的花园，花园可以分成n块不同的区域（编号为 1到 n）。春暖花开，同时也是杂草开始生长的季节，编号为i的区域中有 ai 个单位面积的杂草。为了除去花园中的杂草，猴子们请了 n 个除草员分别给 n 块区域除草，每个除草员每个单位时间内可以除去 1个单位面积的杂草，某 杂草除完后该除草员立即停止工作。
为了提高除草的效率，希望尽快完成除草任务，猴子们买了一台除草机帮助除草员进行除草，但这台除草机在一个单位时间内仅能供一个除草员使用，使用除草机后除草员的工作效率提高 m，即该除草员每个单位时间可以除去 m 个单位面积的杂草（当然不使用后仍然恢复到原来的工作效率）。
现在，请你编程帮猴子们决定，怎样分配这台除草机的使用对象才能使完成所有除草任务的时间最短？

【输入】

输入共2行。
第1行两个整数n和m，表示花园中共有n块待除草的不同区域和除草员使用除草机时的工作效率为m。
第2行n个整数，第 i个整数a 表示第i块区域中有 a 个单位面积的杂草。

样例输入	样例输出
3 4 2 3 5	2

【提示】

[样例解释]
输入中有3块需要除草的区域，每块区域杂草的面积依次为 2，3，5个单位面积。如果使用除草机在单位时间内能除 4个单位面积的草。
完成所有除草任务最少需要 2个单位时间。方案可能不唯一，比如：
方案一：在第一个单位时间内，除草机给 2号区域的除草员使用，第一个单位时间结束时各区域依次还有1，0，4个单位面积的杂草。在第二个单位时间内，除草机给 3 号区域的除草员使用，第二个单位时间结束时各区域依次还有 0，0，0个单位面积的杂草，即经过2个单位时间后所有的杂草被除尽。
方案二：在第一个单位时间内，除草机给 3号区域的除草员使用，第一个单位时间结束时各区域依次还有1，2，1个单位面积的杂草。在第二个单位时间内，除草机给 2 号区域的除草员使用，第二个单位时间结束时各区域依次还有 0，0，0个单位面积的杂草，即经过2个单位时间后所有的杂草被除尽。

[数据范围]
30%的数据，$1 \leq n \leq 1000$，$0 \leq ai \leq 10000$。
100%的数据，$1 \leq n \leq 10^5$，$0 \leq ai \leq 10^9$ ，$1 \leq m \leq 10^9$。

【分析】

通过对题意的分析可得，使用除草机后单位时间内的效率将增加m-1。为保证这增量能物尽其用，应给至少剩余m杂草量的人使用，若所有区域剩余都不足m，则给剩余最多的人使用。因此，本题的解决重点就是每次使用都给剩余杂草量最多的区域来使用。我们可以用大根堆在每个时间点找出余量最多者，减去m-1的差值后重新入队。当时间的累计数与队头值相等，则说明在该时间内所有草量均能清除完毕。

【代码】

```
#include<iostream>
#include<queue>
```

```cpp
using namespace std;
int n,m,x,k;
priority_queue<int> q;
int main(){
    cin>>n>>m;
    while(n--){
        cin>>x;
        q.push(x);
    }
    while(k<q.top()){
        k++;//累计时间
        int t=q.top();
        q.pop();
        q.push(t-(m-1));//减去单位时间内的增量，而非总除草量
    }
    cout<<k;
}
```

E.航空管制

【题目描述】

为了迎接* * *的召开， * * *的机场进行了航空管制。所有的飞机必须在第k分钟之后才能起飞。已知为了保证飞行的安全每两架飞机的起飞时间之间至少间隔1分钟。现在机场一共有n架飞机需要起飞，第i架飞机的预定起飞时间是第i分钟,第i架飞机每晚于预定起飞时间1分钟机场都会有Ci的损失。飞机不能够提前起飞。问如何安排起飞顺序能够使机场的损失最小，输出这个最小的损失。

【输入】

第一行两个整数n，k分别表示机场的飞机数量和最早的起飞时间。
第二行n个整数，第i个整数Ci表示第i架飞机的损失。
输入数据保证 $n <= 100000k,Ci <= 100000$ 。

【输出】

一个整数表示最小损失

样例输入	样例输出
5 3 4 2 3 5 1	24

【提示】

[样例解释]
1号飞机到5号飞机的起飞时间分别是 3 6 5 4 7 这样就能够得到样例的解答，而且没有更小的损失。

[子任务]
对于30%的数据$n, k <= 10$

【分析】

从题意可了解到，在每一个时刻只能有一架飞机起飞，且需在最早起飞时刻之后，其他飞机必须等待，在等待过程中会产生消耗。因此必须使得单位时间内损耗大的飞机先起飞以使得总损失最小。

【代码】

```cpp
#include<cstdio>
```

```cpp
#include<queue>
using namespace std;
long long n,m,c,k;
long long ans;
struct pl{
    long long ci,ti;
    bool operator<(pl b) const{
        return ci<b.ci;
    }
};
priority_queue<pl> q;
int main(){
    scanf("%lld%lld",&n,&m);
    for(int i=1;i<=n;i++){
        scanf("%lld",&c);
        q.push((pl){c,i});
        if(i>=m){//能起飞则尽快安排当前损耗最大的飞机起飞，及时止损
            ans+=(i-q.top().ti)*q.top().ci;
            q.pop();
        }
    }
    k=n+1;
    while(!q.empty()){
        if(k>=m){//存在起飞时间在所有航班预计时间之后
            pl t=q.top();
            ans+=(k-t.ti)*t.ci;
            q.pop();
        }
        k++;
    }
    printf("%lld",ans);
}
```

F.中位数之和

【题目描述】

flower有N-1个朋友，他们要一起玩一个游戏：首先确定三个非负整数a,b,c,然后每个人依次在纸上写一个数，设第i个人写下的数字为f[i],flower先写下数字f[1]=1，对于第i个写数字的人(i>1)有：f[i]=(a*m[i-1]*+*bi*+c)mod1,000,000,007；其中m[i-1]为前i-1个写下的数字的中位数，如果i-1为偶数，那么取靠前的那个数。
flower想要知道，所有人写下的数字的和。

【输入】

输入仅一行,包含四个非负整数 a,b,c,N；意义如上；

【输出】

输出只有一行一个整数，表示数字和。

样例输入	样例输出
3 1 2 6	103

【提示】

对于 30%的数据： $N \leq 1,000$；
对于 100%的数据： $N \leq 200,000$；$a,b,c \leq 1,000,000,007$。

【分析】

求中位数，即有序数列中间位置的数。可将该有序数列一分为二，视第一段的末尾为中间数。这里需要用一个大根堆用于存储前半段数，一个小根堆用于存储后半段数。序列的元素位置调整，以及中位数的确定，关系就密切的两个数就是这两个堆的堆顶。
具体做法为首先将f[1]=1放入大跟堆，后续每次生成的数则放入小根堆。其中需要注意的是，mod后的数字若小于大根堆顶，就与大根堆顶交换存储位置。如果本次操作为第奇数次操作，还需将小根堆顶放入大根堆，确保左右两堆的量均衡。

【代码】

```
#include<iostream>
#include<queue>
using namespace std;
priority_queue<int> dq;
priority_queue<int,vector<int>,greater<int> > xq;
long long a,b,c,n,x,ans;
int main(){
    cin>>a>>b>>c>>n;
    dq.push(1);
    ans=1;
    for(int i=2;i<=n;i++){
        x=(a*dq.top()+b*i+c)%1000000007;
        ans+=x;
        if(x<dq.top()){
            xq.push(dq.top());
            dq.pop();
            dq.push(x);
        }else{
            xq.push(x);
        }
        if(i&1){
            dq.push(xq.top());
            xq.pop();
        }
    }
    cout<<ans;
}
```