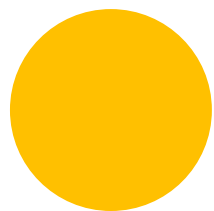
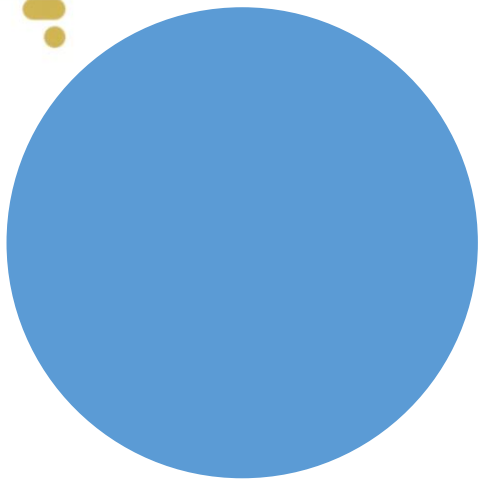


马上开始

35tang-C++竞赛系列三阶课程





《 35tang-C++竞赛系列三阶课程 》



C++提供的的队列数据结构

定义 `queue<int> q;`

队列不空 `while(!q.empty())`

新元素入队 `q.push(m);`

取队首元素 `q.front();`

队首元素出队 `q.pop();`

注意：先进先出，`push`是插入队列的尾部，`front`是取到队首元素，`pop`是把队首的元素从队列删除

一头进一头出



数组实现的队列last(队尾)进，first(队头)出

定义 `int que[100], first = 1, last = 0;`

队列不空 `while(first <= last)`

新元素入队 `que[++last] = i;`

等价于

`last++; que[last] = i;`

取队首元素并出队 `int cur = que[first++];`

等价于

`int cur = que[first]; first++`

右图演示依次入队100, 200, 300, 然后依次删除的数组变化

注意 `first > last` 就是空的

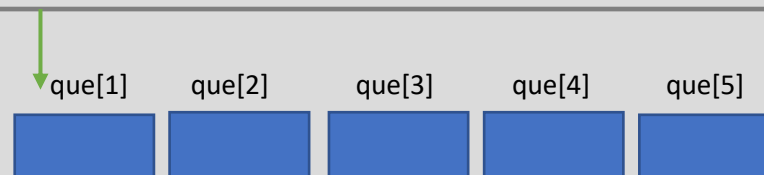
first



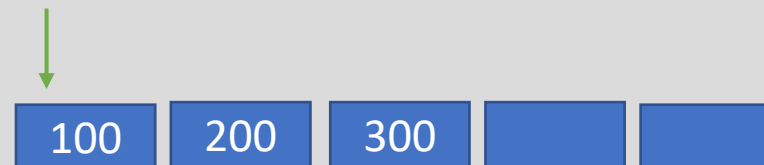
last



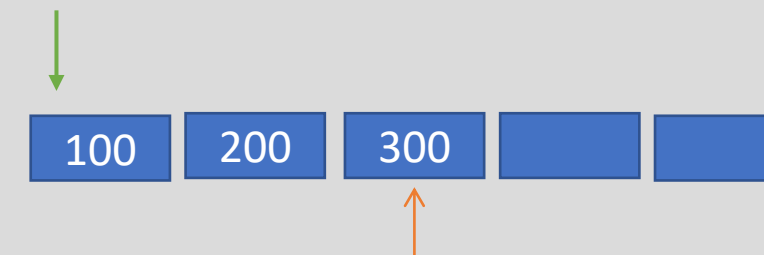
初始状态 `first > last`



入队



出队



pair

#include <utility>

- pair 是一种模版类型。每个pair 可以存储两个值。这两个值类型无限制。也可以将自己写的struct的对象放进去。不过用的最多的是2个整数。

```
pair<string,int> p;
```

```
pair<int ,int > p1;
```

```
pair<double,int> p2;
```

- 赋值和初始化:

```
pair<int ,int >p (5,6);
```

```
P={10,12}; //这种用法不建议使用, C++11
```

```
p=make_pair(40,50);
```

- 如何取得pair的值呢
 - 每个pair 都有两个属性值 first 和second

```
cout<<p.first<<p.second;
```

- 和queue一起使用, 比struct更方便, 尤其是队列里面需要的状态是2个元素组成的时候。比如迷宫, 二维矩阵, BFS搜索的时候需要保存当前状态是当前走到哪一个位置 (row, col) 这样的2个整数, 用pair最方便
- pair数组存放坐标轴的点 (x,y)

```
pair<int ,int > points[100];
```

```
points[i].first就是第i个点的x坐标, points[i].second就是y
```

```
points[i].first=10; points[i].second=10 //直接赋值
```



奶牛吃草

奶牛沿着直线从最左面的位置 n ($n \geq 0$) 出发，往右行走，在位置 k ($n < k < 10000$) 有草，奶牛一次可以从 x 位置移动到 $x-1, x+1$ ，也可以使用魔法，跳跃当 $x*2$ 位置。问最少多少次可以到达 k 位置？

注意：走动的过程中不能走到小于0的位置。

示例输入：

5 17

示例输出

4

最短最少一般要用BFS

- 为什么搜索?因为要枚举各种可能, 循环没办法写出这样的枚举。而这些枚举是一步一步的, 根据前一步可以枚举出后面的几个可能 (相邻节点)。
- DFS不好做, 需要把所有的路线都枚举了才知道哪一个最短。
- 用BFS, BFS队列放的东西就是每一个状态, 也就是DFS的参数, 就是我们根据什么才能决定是否找到了答案, 根据什么才能决定下一步怎么走。显然: 就是当前到的位置以及用了几步 (step), 下一个状态就是step+1, 位置-1, +1或者*2
- 以前我们会设置一个struct包含2个整数: 当前位置, 当前step, 可以用pair

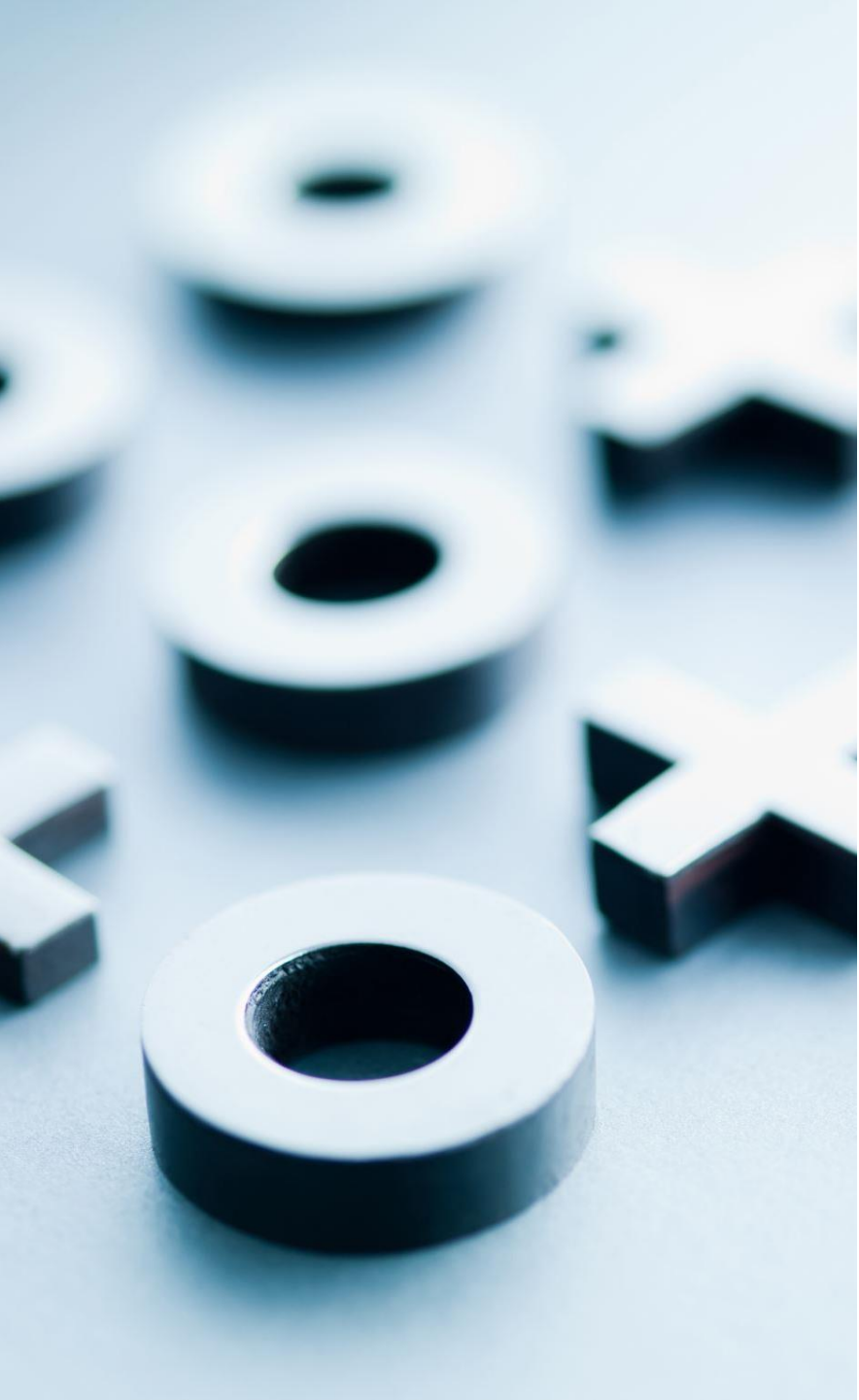
```
queue <pair<int,int> > myqueue;  
myqueue.push(make_pair(0,0));  
myqueue.push(make_pair(now+1,step+1));
```
- 寻找最短, 队列的性质或者说BFS的性质, 同一个位置肯定第一次访问可能是最短, 可以用visited数组来避免一个位置重复访问。



写

对么?

```
int n,k;
queue <pair<int,int> > myqueue;
cin>>n>>k;
myqueue.push(make_pair(n,0));
while (!myqueue.empty()) {
    int now=myqueue.front().first;
    int step=myqueue.front().second;
    myqueue.pop();
    if (now==k){//到了
        cout<<step<<endl;break;
    }
    //下面把3个合法相邻点入队列
    if (now+1<=k && !visited[now+1]) {
        myqueue.push(make_pair(now+1,step+1));
        visited[now+1]=true;
    }
    if (now-1>=0 && !visited[now-1]) {
        myqueue.push(make_pair(now-1,step+1));
        visited[now-1]=true;
    }
    if (now*2<=k && !visited[now*2]) {
        myqueue.push(make_pair(now*2,step+1));
        visited[now*2]=true;
    }
}
```



细节决定成败

- 注意下面不对:

```
if (now*2<=k && !visited[now*2]) {
```

- 因为有-1，所以最大2倍走到的位置可能 $k+1$ ，然后再-1走回来。所以修改成:

```
if (now*2<=k+1 && !visited[now*2]) {
```

例子：

定义以 a 为基的集合 B 满足如下条件：

- a 是集合 B 的基，且 a 是 B 的第一个元素；
- 如果 x 在集合 B 中，则 $2x+1$ 和 $3x+1$ 也都在集合 B 中；
- 没有其他元素在集合 B 中了。

如果将集合 B 中元素按照升序排列，第 N 个元素会是多少？

输入格式：

输入包括两个数字，集合的基 $a(1 \leq a \leq 30)$ 以及所求元素序号 $N(1 \leq n \leq 1000000)$ 。

输出格式：

输出集合 B 的第 N 个元素值。

输入样例

1 100

输出样例

418

- 样例说明：第一个元素是1，那么3和4都在，然后3在所以7和10也在，4在，所以9和13也在，以此类推

算法：模拟一下

先尝试小的，比如试一下a为1，N为5.

如果每一次用当前生成的元素 $*2+1$ ， $*3+1$ 生成后2个元素

$1 \rightarrow 1 \ 3 \ 4 \rightarrow 1 \ 3 \ 4 \ 7 \ 10 \rightarrow 1 \ 3 \ 4 \ 7 \ 10 \ 9 \ 13$

这时候的第5个元素显然不对，不是10，而应该是9，也就是说后生成的数有可能比前一个小。

改一下：每一次都先排序，然后用当前最小元素 $*2+1$ ， $*3+1$ 生成后一个元素

$1 \rightarrow 1 \ 3 \ 4 \rightarrow 1 \ 3 \ 4 \ 7 \ 10 \rightarrow 1 \ 3 \ 4 \ 7 \ 9 \ 10 \ 13$

为了避免后面出现比前面还小的元素，我们每生成一次数都排序，每一次都取当前最小的，那么第n个最小的就是答案了。或者优化一下，每一次排序取最小生成后面的数，然后把这个数删除，第n次最小的就是答案。

数据结构

- 用数组保存生成的数显然不行，因为每一次排序时间复杂度太高。
- 用set，set排序而且可以去重复，一种变相的队列，set还知道总的元素数目，set的时间复杂度每次访问是 $O(\log N)$ ，现在N是10的6次方。那么循环从1到N，每一次取当前最小数，删除，并且把这个最小数的 $*2+1$ ， $*3+1$ 的结果放到set。那么N-1次操作后set里面最小的`.begin()`就是答案。
- 用优先队列是类似的。优先队列每一次最小的在`.top()`，时间复杂度和set基本一致，但是不能去重复，如果要用最好手工去重复。
- 总的时间复杂度 $O(N \cdot \log N)$ ，大约是10的7次方，有点危险。
- 最后结果可能很大，用long long而不是int



写！至少一半分数有了

```
set <long long > myset;
cin>>a>>n;
myset.insert(a);
for (int i=1;i<n;i++)
{
    //取当前最小的
    long long t=*myset.begin();
    //算出后两个入set，并且删除计算过的最小的
    myset.erase(t);
    myset.insert(t*2+1);
    myset.insert(t*3+1);
}
cout<<*myset.begin()<<endl;
```

优化数据结构

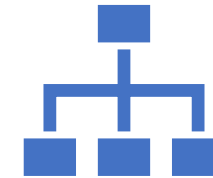
- 仔细分析，就差一点。我们用set排序就是为了避免重复，有没有别的办法？如果一直 $*2+1$ ，肯定不重复而且递增；如果一直 $*3+1$ ，也不会重复而且递增。关键是两个一起有重复，比如i的 $*2+1$ 可能和j的 $*3+1$ 重复。可以用两个队列，一个记录 $*2+1$ ，一个记录 $*3+1$ ，这样每次找两个队列最小的去处理（也就是队头，因为每一个队列是从小到大插入的），如果两个队头相同，扔掉一个。
- 所以用不着set或者优先队列。用queue，无论是系统的还是我们自己实现的，都可以，这样时间函数就是 $O(N)$ 了，因为set和优先队列插入是 $O(\log N)$ ，队列插入是 $O(1)$

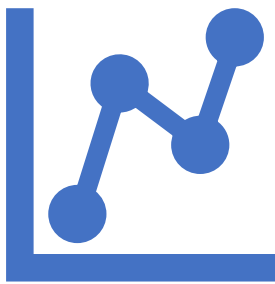


手工模拟一下

注意：我们的程序是在翻译我们的算法，如果我们的算法手工无法得到正确答案，就不要开始写。或者说写程序的第一步永远是手工验证你的算法。


```
queue <long long> q2,q3;
cin>>a>>n;
long long tmp=a;
q2.push(tmp*2+1);    q3.push(tmp*3+1);    int i=1;
while (i<n) { //取出*2队列和*3队列头上小的一个，分别处理后入两个队列
    if (q2.front()>q3.front()) {
        tmp=q3.front();
        q3.pop();
        q2.push(tmp*2+1);
        q3.push(tmp*3+1);
        i++;
    }
    else if (q2.front()<q3.front()) {
        tmp=q2.front();
        q2.pop();
        q2.push(tmp*2+1);
        q3.push(tmp*3+1);
        i++;
    }
    else q2.pop();
}
cout<< tmp<<endl;
```





2019 NOIP 公交换乘

一起看题目，并且分析输入数据
大家先自己想想，怎么做？

分析

- 坐地铁要花钱，但是可以得到等额优惠券，优惠券可以用来免费做公共车，没有优惠券可用就花钱，但是优惠券是有45分钟时间限制的。给定n条乘坐地铁，公交车的时间（t）和票价（price），求最少用的钱。
- $n \leq 10^5$; $t \leq 10^9$; $price < 1000$
- 模拟的思路：是不是每一次坐公交车都要去检查一下前面有没有没有用的优惠券（而且可能的话，按照时间排在最前面的可用的先用掉：先进先用）？而优惠券超过45分钟无效？做一次地铁就要花钱并且增加一个优惠券？
- 优惠券放在什么数据结构？按照时间顺序选择，并且优惠券金额要大于票价，数组？如果数组只是加入新的不删除，最后这个数组很大，每一次查询太消耗时间了，所以是不是可以队列？按照时间顺序入队列并且删除用过的和过期的(这样队列最多也就是45个数据)。从前往后选择能用的，但是用过的不一定是队头，也就是删除的不一定是队头，所以不能直接用系统队列（因为删除一个优惠券就意味着需要重新排序）。



队列的两种方式

- 用数组，设置头尾两个变量分别表述当前队列的头和尾在数组的哪个位置（数组下标），出队就是头++，进入队列就是尾++。优点：好控制。
- 用系统提供的queue：缺点：只能先进先出，删除最好是从队头删除，如果删除的不是队头元素，出去了再进去顺序就乱了。

解决办法：模拟循环可以搞定

首先需要有一个累加器，不停的累加花的钱

循环处理每一个坐车记录，根据公交还是地铁做不同处理

优惠券是不是要存起来？怎么存？

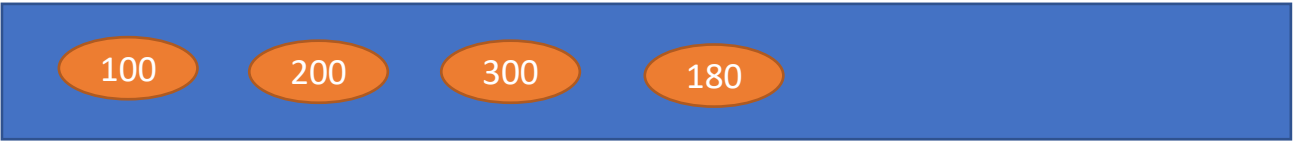
- 这里最关键的是数据量太大，而且 t 更大，所以我们不能对时间做计数。而且不能记录在数组每次去数组里面查（太大了），我们维护一个45分钟有效期优惠券队列，每一次遇到公交和地铁做不同的处理，做地铁就加优惠券，坐公交就找从前往后找能用的优惠券。每一次还要把过期的删除。由于每一个时间都不同，这里的队列理论上会很小的。
- 这个队列用数组手工实现比较好，因为**按照时间顺序选择，并且优惠券金额要大于票价**，从前往后。而最前面的优惠券不一定能使用（金额不够），先进先用但是不是先进先出。也就是说队头取出来的不一定出队还要按照时间顺序入队，所以用系统队列会比较麻烦一些，如果要用就用优先队列（带时间排序的），删除一个优惠券就意味着需要重新排序。

手工队列，数组，根据时间控制头尾，但是如果用过的，由于不是队头，是否可以设置为0，防止下次使用。

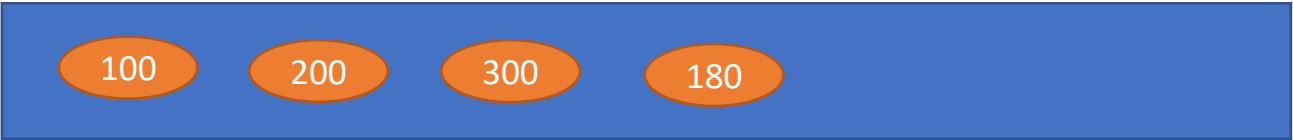
优惠券按照时间顺序



需要做地铁，票价180元
 $r+=180$



需要做公交，票价180元



每次读到新的坐车记录
检查队列是否有过期的



一起写

```
for (int i=0;i<n;i++){//依次读入坐车记录，逐条处理
```

```
    fin>>bus>>price>>time;
```

```
    while(time-times[head]>45 && (head<=last))
```

```
        head++; //每次维护一次队列，通过增加head，把离当前时间超过45分钟的请出队列
```

```
    if (bus==0) { //如果坐地铁，r就是总价格加上票价，同时通过增加last，把优惠券加到队列
```

```
        last++;
```

```
        prices[last]=price;
```

```
        times[last]=time;
```

```
        r+=price;
```

```
    }
```

```
    else if(head>last) r+=price; //做公交，如果head>last说明队列空，也就是没有优惠券，花钱
```

```
    else{ //做公交，有优惠券，使用，然后把优惠券价格修改为0，这里不能请出队列，因为可以使用的优惠券不一定是队头的
```

```
        bool find=false;
```

```
        int from=head;
```

```
        //和普通队列不同，不能head++因为取出来不能出队，前面的优惠券有可能还没有使用过，所以用form来++遍历队列
```

```
        while (from<=last) {
```

```
            if (prices[from]>=price ) {find=true; break;} //找到可用的优惠券，退出循环
```

```
            from++;
```

```
        }
```

```
        if (find) prices[from]=0; else r+=price;
```

```
    }
```

```
}
```

//prices[]和times[]2个队列维护优惠券信息，一个是优惠券价格，一个是优惠券时间，通过head和last来同时维护2个队列的头尾。可以换成一个结构体队列

新数据结构：map类型

map类型存放的是键值对，就是说根据key的值找value.

比如：

字符串"abc"配对数字100

字符串"ac"配对数字120

字符串"ad"配对数字99

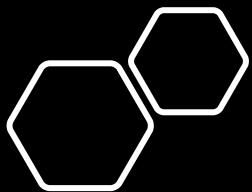
..

给出了很多个这样的键值对，也就是说字符串是key，后面一个整数是value。现在如果给出一个字符串，怎么才能最快的查找到这个字符串对应的整数呢？用map。

和set类似，内部也是用二叉树(二叉查找树,红黑树)实现的，所以查找效率也是 $\log n$.

也是排序的，缺省从小到大

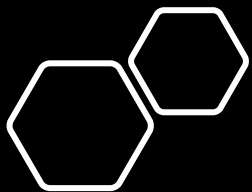




和set类似的 iterator访问

```
#include <iostream>
#include <map>
using namespace std;
int main ()
{
    map<char,int> mymap;
    map<char,int>::iterator it;
    mymap['a']=50;
    mymap['b']=100;
    mymap['c']=150;
    mymap['d']=200;

    it = mymap.find('b');
    if (it != mymap.end())    mymap.erase (it);
    // print content:
    cout << "elements in mymap:" << '\n';
    cout << "a => " << mymap.find('a')->second << '\n';
    cout << "c => " << mymap.find('c')->second << '\n';
    cout << "d => " << mymap.find('d')->second << '\n';
    return 0;
}
```



遍历类似set;

但是类似数组的
[]访问非常方便

```
#include <iostream>
#include <map>
using namespace std;
int main ()
{
    map<char,int> mymap;
    //类似数组的赋值
    mymap['b'] = 100;
    mymap['a'] = 200;
    mymap['c'] = 300;
    //类似数组的访问
    cout << mymap['c']; //找到就是value
    cout << mymap['d']; //找不到就是0
    // 遍历
    for (map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
        cout << it->first << " => " << it->second << "\n";
    return 0;
}
```

Map--flag计数法的替代，都可以用[]下标形式访问。

例如：统计给出的每一个整数出现了多少次。如果给出整数1 10 10

flag计数：整数和整数的对应关系。

用flag计数需要一个数组，这个数组的长度至少需要时出现的整数的最大值那么大。

int flag[11]; 当访问任意元素的时候，比如flag[10]时间复杂度是O(1)

flag[0]	flag[1]	flag[2]	flag[3]	flag[4]	flag[5]	flag[6]	flag[7]	flag[8]	flag[9]	flag[10]
0	1	0	0	0	0	0	0	0	0	2

map：任意两个数据类型的对应关系。也可以表示整数和整数的对应关系。

用map有一个整数对就添加一个元素，并不像数组提前开出很大的空间。

map<int,int> mym; 当访问任意元素的时候，比如mym[10]时间复杂度是O(logn)

mym[1]	mym[10]
1	2

flag计数法 的替代

- 整数和整数的对应关系，但是很大
 - 给出很多学生的编号（整数）和成绩（比如有10的5次方），然后输入若干学生的编号去查找他们的成绩（最多可能也有10的5次方）。怎么查？

如果给出的学生的编号不大，比如最大编号不超过10的4次方，显然可以用flag计数的方法，最简单，时间复杂度 $O(1)$ 。数组大小差不多40k。

但是，如果编号最大会有10的9次方呢？数组大小？

用map把编号和学生成绩map起来，这样查询的时候就是 $O(\log n)$ 的时间复杂度了。

还有什么时 候用map而 不是flag计 数数组

- 不是整数和整数的对应关系。
 - 如果：给出很多学生姓名和成绩（比如有10的5次方），然后输入若干学生的姓名去查找他们的成绩（最多可能也有10的5次方）。怎么查？

用map把成绩和学生姓名map起来，这样查询的时候就是 $O(\log n)$ 的时间函数了。

选择合适的数据结构

- 队列，set，优先队列，map，数组，二维数组...
- 合适的数据结构能够优化我们的算法
- **map, set, multimap, and multiset (后面两个容许重复)**
- **map支持下标[]操作，set不支持下标操作。**
- 上述四种容器采用红黑树实现，红黑树是平衡二叉树的一种。不同操作的时间复杂度近似为：
 - 插入: $O(\log N)$
 - 查看: $O(\log N)$
 - 删除: $O(\log N)$
- 优先队列（堆，类似树结构）：
 - 入队出队是 $O(\log N)$ ，删除 $O(N)$
- queue和stack，数组等都可以基于线性表实现
 - Push和pop基本上都是 $O(1)$
- unordered_set 和unordered_map基本上都是 $O(1)$ 基于 hash，不能重复，c++11才能支持
- map是键值对，两个元素的对应关系；其他set，优先队列，queue，stack，数组等等基本上都是单个元素按照一定顺序（从小到大，还是先入先出，先入后出）和规则（是否可以重复）存起来，访问的方式也各不相同。

例题dupcnt.cpp

给出 n 个整数 ($1 \leq n \leq 2 \cdot 10^5$)，其中每一个整数大于0并且小于 10^9 ，找出其中出现次数最多的数和他出现的次数，如果多个数出现的次数一样，输出最大的一个。

输入格式：

第一行一个整数 n ，后面一行空格分隔的 n 个整数。

输出格式：

一行，两个整数，第一个是出现次数最多的整数，第二个是出现的次数，空格分隔。

示例输入：

```
10
10 9 8 1 9 9 2 3 2 2
```

示例输出

```
9 3
```


方法一： 数组实现

- 一组数字一般情况下总是可以用数组实现的。用数组实现首先要想，要不要排序？这里要排序，只有排序了，一样的数字才会连在一起，比如排序后
- 1 2 2 2 3 8 9 9 9 10
- 这个时候我们可以遍历数组，根据某一个元素和他前面或者后面是否相等来判断当前数字是否是连续出现的，如果是连续出现，计数器++，换了新的数字，计数器重新初始化计数。那么，这个计数器就是当前数字重复重现的次数，找最大，打擂就好了。
- 遍历数组时间复杂度 $O(n)$ ，排序时间复杂度 $O(n*\log n)$ 。所以总的时间复杂度 $O(n*\log n)$

```

#include <bits/stdc++.h>
using namespace std;
#define maxn 200001
int a[maxn]; // 这里用全局遍历应为静态数组可能比较大，放在main函数可能内存不够
int main()
{
    int n;
    int ansCnt=INT_MIN; // =0 也可以，存放最终结果，打擂台用
    int ansNum; // 记录最大出现次数的整数
    cin>>n;
    for (int i=0;i<n;i++) cin>>a[i]; // 读入

    sort(a,a+n); // 从小到大排序

    int cnt=1; // 当前相同元素出现的次数，初始化为1，表示a[0]出现了一次
    for (int i=1;i<n;i++) // 从1开始，因为需要和前一个元素相比较
    {
        if(a[i]==a[i-1]) cnt++; // 和前一个相同，连续次数++
        else { // 新的元素，前面的元素连续次数结束了
            if(cnt>=ansCnt)
            { // 打擂台，注意一定要>=，因为我们找的是最大的
                ansCnt=cnt;
                ansNum=a[i-1];
            }
            cnt=1; // 新的元素初始化个数为1
        }
    }
    // 每一次找的是前一个连续出现的数，最后一个数没有后面的了，这里手工处理
    if(cnt>=ansCnt)
    { // 打擂台，注意一定要>=，因为我们找的是最大的
        ansCnt=cnt;
        ansNum=a[n-1];
    }

    cout<<ansNum<<" "<<ansCnt<<endl;
    return 0;
}

```

方法二： vector实现

- 其实和静态数组实现算法是一样的，时间复杂度也一样。
- 只不过vector的用法和静态数组不同
- `vector<int> a(n);` 可以申明一个vector数组，并且同时分配n个元素空间，这样就可以类似静态数组一样，直接访问`a[]`了。
- 如果是这样申明`vector<int> a;` 那么a刚开始是空的，不能直接用`a[]`访问，而是用`push_back`方法插入每一个元素到a

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    int ansCnt=INT_MIN;//=0 也可以，存放最终结果，打擂台用
    int ansNum;//记录最大出现次数的整数
    cin>>n;
    vector<int> a(n);// 初始化a数组n个元素
    for (int i=0;i<n;i++) cin>>a[i]; //读入

    sort(a.begin(),a.end());//从小到大排序

    int cnt=1;//当前相同元素出现的次数，初始化为1，表示a[0]出现了一次
    for (int i=1;i<n;i++) //从1开始，因为需要和前一个元素相比较
    {
        if(a[i]==a[i-1]) cnt++; //和前一个相同，连续次数++
        else { //新的元素，前面的元素连续次数结束了
            if(cnt>=ansCnt)
            { //打擂台，注意一定要>=,因为我们找的是最大的
                ansCnt=cnt;
                ansNum=a[i-1];
            }
            cnt=1;//新的元素初始化个数为1
        }
    }
    //每一次找的是前一个连续出现的数，最后一个数没有后面的了，这里手工处理
    if(cnt>=ansCnt)
    { //打擂台，注意一定要>=,因为我们找的是最大的
        ansCnt=cnt;
        ansNum=a[n-1];
    }

    cout<<ansNum<<" "<<ansCnt<<endl;
    return 0;
}
```

方法三：二分法

- 数字序列，排序。一旦想到了这些，我们就可以尝试看看二分法是否可以使用？
- 系统有二分查找函数upper_bound：查找第一个大于某个元素的位置。
- 所以，先排序，然后遍历数组，对每一个元素，用二分查找去看看他出现的最后的位置，减去当前位置，就是出现了多少次。打擂。
- `int cnt=upper_bound(a.begin(),a.end(),a[i])-a.begin()-i;`
 - 注意，upper_bound返回的是位置，不是数组下标，所以减去begin()位置，得到数组下标。
- 优化一些的算法，可以考虑相同元素只有最后一个元素才去二分查找。
- 时间复杂度n个元素，每一个做一次二分查找， $O(n*\log n)$

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    int ansCnt=INT_MIN; //=0 也可以, 存放最终结果, 打擂用
    int ansNum; //记录最大出现次数的整数
    cin>>n;
    vector<int> a(n); // 初始化a数组n个元素, 也可以不初始化, 直接读入n个元素, 每一个元素push_back
    for (int i=0; i<n; i++) cin>>a[i]; //读入

    sort(a.begin(), a.end()); //从小到大排序

    for (int i=0; i<n; i++)
    { //对每一个元素二分查找出现的次数并且打擂
        int cnt=upper_bound(a.begin(), a.end(), a[i]) - a.begin() - i;
        if(cnt>=ansCnt)
        { //打擂, 注意一定要>=, 因为我们找的是最大的
            ansCnt=cnt;
            ansNum=a[i];
        }
    }

    cout<<ansNum<<" "<<ansCnt<<endl;
    return 0;
}
```

方法四：flag计数的方法

- 当我们发现涉及到出现次数的问题，肯定很容易联想到计数排序的方法，就是flag计数，通过数组下标和出现的数对应，数组元素的值和出现次数对应，通过这种映射关系，打表，轻松的把每一个数出现的次数记录在flag数组中。然后去遍历这个flag数组，找里面出现次数最多的最后一个数就可以了。
- 注意：计数排序的缺陷是无论一个数是否出现，计数数组里面都会留一个位置给这个数，所以空间浪费，要多少空间呢？由于数组下标和整数对应，所以出现的整数的组大值就是你数组的元素个数了。这个题目给定的整数最大可能是10的9次方，实际上用这个方法不能解决所有的问题，而且，如果出现的整数是负的呢？如何和数组下标对应？
- 为了和map做比较，还是写出来，得部分分数。时间复杂度是 $O(\max(m,n))$ ， m 是出现得最大整数。

```
#include <bits/stdc++.h>
#define maxm 100000
//这里最大整数为10的9次方太大了，设置10的5次方得部分分数
using namespace std;
int main()
{
    int n;
    int ansCnt=INT_MIN;//=0 也可以，存放最终结果，打擂用
    int ansNum;//记录最大出现次数的整数
    cin>>n;
    vector<int> flag(maxm,0);//初始化flag数组100000个元素，都是0
    // 用静态数组也是可以的，
    int a;
    for (int i=0;i<n;i++) {
        cin>>a;
        flag[a]++;
    }

    for (int i=0;i<maxm;i++)
    {//遍历flag数组，对出现次数打擂
        if(flag[i]>=ansCnt)//注意：flag[i]就是i出现得次数
        {//打擂，注意一定要>=,因为我们找的是最大的
            ansCnt=flag[i];
            ansNum=i;
        }
    }

    cout<<ansNum<<" "<<ansCnt<<endl;
    return 0;
}
```


方法五： map

- 当我们发现涉及到出现次数的问题，肯定很容易联想到计数排序的方法，就是flag计数。如果出现整数太大，flag数组放不下，肯定可以用map替代。如前面所述，map并不存放所有小于最大整数的数，只把出现的整数和他的次数，这种映射关系保存下来。用法和flag数组也非常类似。
- 如果用unordered_map时间复杂度和flag数组一样，可以做导 $O(n)$ ，但是只有c++11支持。用map，每一次访问map时间复杂度 $O(\log n)$ ，所以总的时间复杂度是 $O(n * \log n)$
- 如果重复数字比较多，其实这是最快的。

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    int ansCnt=INT_MIN; // =0 也可以, 存放最终结果, 打擂台
    int ansNum; // 记录最大出现次数的整数
    cin>>n;
    map<int,int> flagm; // map的第一个元素就是出现的整数, 第二个就是个数, 也可以通过[]访问
    int a;
    for (int i=0;i<n;i++) {
        cin>>a;
        flagm[a]++; // flagm[a]的初始值可以认为是0
    }

    // 遍历和数组不同
    for (map<int,int>::iterator it=flagm.begin(); it!=flagm.end(); it++)
    { // 遍历map, 对出现次数打擂台
        if ((*it).second>=ansCnt) // 注意: flag[i]就是i出现的次数
        { // 打擂台, 注意一定要>=, 因为我们找的是最大的
            ansCnt=(*it).second;
            ansNum=(*it).first;
        }
    }

    cout<<ansNum<<" "<<ansCnt<<endl;
    return 0;
}
```

可能的数据结构

- 静态数组： $O(n \cdot \log n)$
- Vector（和静态数组算法类似） $O(n \cdot \log n)$
- 二分法 $O(n \cdot \log n)$
- flag计数的方式（还是数组和vector） $O(\max(n, m))$
- map $O(n \cdot \log n)$
- set 由于set不能保存重复数字，经常用来去重复，所以，这里用set保存这些整数不方便。如果题目变成输出去掉重复之后的数，或者去判断一个数是否出现过，就可以用set了（把数字插入set，查找有没有出现过用find函数，查找的时间复杂度 $O(\log n)$ ）。当然，可以用multiset,这样就可以把重复的数也放在multiset，插入后自动排序，然后遍历multiset。可以看出，这个时候我们用multiset没有必要，和数组相比较，无非是插入后排序，还是每插入一个就排序，最后都是处理一个排好序的序列。时间复杂度也一样 $O(n \cdot \log n)$
- 优先队列：和multiset一样的道理，和数组排序算法其实差不多。时间复杂度 $O(n \cdot \log n)$
- Queue或者stack？没有必要，这里我们没有找到先进先出或者先进后出的需要。而queue和stack是不会帮你排序的。

- map是键值对，可以用来统计出现了多少次，非常类似flag计数数组。如果只是判断一个数是否出现过，也可以用set来做。
- 比如前面搜索用到的visited数组就完全可以用map来取代。当然，flag计数更快，是 $O(1)$ 时间复杂度，而map是 $O(\log N)$ ，所以能用flag计数还是用flag计数好。当然unordered_map效率和flag计数几乎一样，可惜只有C++11可以使用。
- 优先队列和set非常类似排序后的数组，不同的在于插入的同时实现了排序，非常适合插入的同时进行处理。注意：set去重复。

课后思考：明明的随机数

<https://www.luogu.com.cn/problem/P1059>

- 给出若干整数，求其中去掉重复后还剩下多少数，并且按照从小到大的顺序输出这些数。
- 思考：
 - 题目给定这些数的范围在1到1000，所以用flag计数是最方便的。
 - 如果题目给定这些数的范围在 10^{-9} 到 10^9 之间呢？想一想，你可以用哪些数据结构，哪些方法来做？哪个你最喜欢？

下节课预习

- 理解题目，尝试思考，不需要做
- 还记得斐波拉契数列用数组是怎么求出第 n 项的值？



作业1

奶牛吃草

cowgrass.cpp

一条直线从左到右表示位置(从0开始)，奶牛沿着直线从开始位置 p 往右行走，位置 q ($q < 1000$) 有草，奶牛一次操作可以朝右面移动1个位置，或者2个位置，也可以使用魔法，跳跃当前位置的两倍（比如从位置5跳到位置10）。问最少多少次操作可以到达 q 位置？

输入格式：两个整数 p 和 q ($p < q < 1000$)，分别表示开始位置和有草的位置。

输出格式：一个整数，最少多少次操作可以从 p 到达 q 。

示例输入1:

10 90

示例输出1:

5

示例1数据说明：奶牛从位置10出发，第一次+1，走到位置11，然后*2，走到位置22，然后*2，走到位置44，然后+1走到位置45，然后*2，走到位置90。一共5次操作，这是最小的操作次数。

示例输入2:

101 1000

示例输出2:

15

作业2

奶牛吃草2

cowgrass1.cpp

一条直线，从0开始，奶牛沿着直线从最位置p往右行走，位置q ($q < 10^9$) 有草，奶牛一次可以朝右移动1步，也可以使用魔法，跳跃当前位置的两倍（比如从位置5跳到位置10）。问最少多少次可以到达q位置？

示例输入：

10 90

示例输出

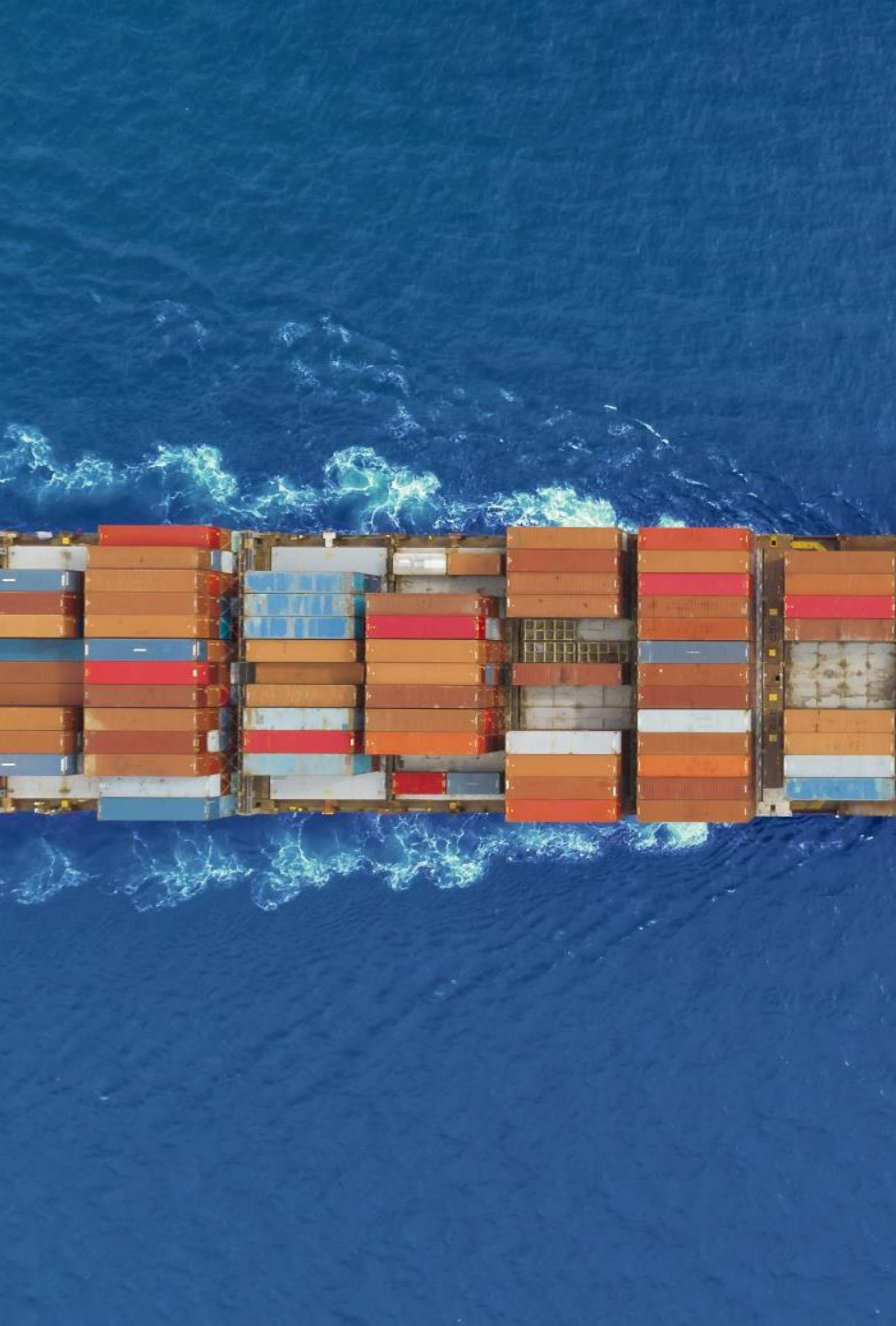
5

示例输入：

101 1000

示例输出

27



超级挑战作业3

- NOIP 2016 第三题 海港
- 题目大意：多个船分别在某个到达（时间小于 10^9 次方），有小于 10^5 左右的数字表示不同国家，问每艘船到达的时候24小时之内到达的所有船里面的人的不同国家数目

作业3提示

- 如果没有24小时这个限制，维护一个国家的计数，遍历每一个船去统计这个船上所有的人的国家。现在多了个24小时，就意味着我们不能只处理当前的船，还要把24小时内的船（或者船上的人）都保存下来（队列），随着新的船的到达时间的变化，这个队列不断地变化，到一个新的船就要把24小时内所有的船上的人都统计一下。
- 把所有人的国家和到达时间(也就是他所在的船的到达时间)存起来, 用一个队列。可以用静态数组实现队列（也可以用系统queue，因为我们是按照时间顺序处理的，也就是先来的人一定先超过24小时先出队），通过这个队列来计算国家的情况。同时根据新的船的时间删除过期的人，从而保证队列长度比较小。
- 维护一个国家的flag（或者map），对应国家的人出队列的时候就把flag--，入队列的时候就flag++，但是统计当前国家数目的时候不能每次遍历国家flag数组累加非零的（太慢了）。可以这么做：维护一个当前国家数目的flag计数数组，比如xflag[i]表示国家i的当前人数，随着时间的推移维护这个数组。当来了一个新船的时候把这个船所有人对应的国家的xflag++，并先看flag是否原来为零，是的话国家数+1；来新船了同时检查队列中已经前面来的人，超过24小时，就减掉一个船的所有人相对应的xflag标记，看flag是否变成0，是的话国家数-1；有点像前面“离散化”的时候根据时间点寻找变化，维护一个当前国家数的变量，在每艘船来的时候进行修改，而不是每一次从头计算。

由易到难，思维体系训练
实战结合，创新协作培养
兴趣导向，未来职业引领

<https://www.35tang.com>



扫码关注公众号

<https://www.三五堂.com>



添加辅导老师