单调队列

单调队列：单调递增或递减的队列，属于优先队列的一种，限制较多，各种神奇算法的基础，队列中元素之间的关系具有单调性，而且，队首和队尾都可以进行出队操作，只有队尾可以进行入队操作。

理论基础：

不妨用一个问题来说明单调队列的作用和操作：

不断地向缓存数组里读入元素，也不时地去掉最老的元素，不定期的询问当前缓存数组里的最小的元素。

最直接的方法：普通队列实现缓存数组。

进队出队都是O(1)，一次查询需要[遍历](http://baike.baidu.com/item/%E9%81%8D%E5%8E%86/9796023)当前队列的所有元素，故O(n)。

用堆实现缓存数组

堆顶始终是最小元素，故查询是O(1)。

而进队出队，都要调整堆，是O(log(n))。

RMQ的方法

[RMQ](http://baike.baidu.com/item/RMQ/1797559)即Range Maximum(Minimum) Query，用来求某个区间内的最大值或最小值。使用[线段树](http://baike.baidu.com/item/%E7%BA%BF%E6%AE%B5%E6%A0%91/10983506)或稀疏表是O(log(n))级的。对于这类问题这两种方法也搞得定，但是没有单调队列快。

单调队列的舞台

由于单调队列的队头每次一定最小值，故查询为O(1)。

进队出队稍微复杂点：

进队时，将进队的元素为e，从队尾往前扫描，直到找到一个不大于e的元素d，将e放在d之后，舍弃e之后的所有元素；如果没有找到这样一个d，则将e放在队头(此时队列里只有这一个元素)。

出队时，将出队的元素为e，从队头向后扫描，直到找到一个元素f比e后进队，舍弃f之前所有的。(实际操作中，由于是按序逐个出队，所以每次只需要出队只需要比较队头)。

每个元素最多进队一次，出队一次，摊排分析下来仍然是 O(1)。

上面的话可能还是没能讲出**单调队列的核心**：队列并不实际存在的，实际存在的是具有[单调性](http://baike.baidu.com/item/%E5%8D%95%E8%B0%83%E6%80%A7/6194133)的子序列。对这个子序列按心中的队列进行操作，譬如在进队时丢弃的元素，虽然它不存在于这个子序列里，但是还是认为他存在于队列里。

另外，进队的顺序和出队的顺序并不一定相同，因为这个队列本身是隐含存在的，可以在进队时看成一个队列，出队时看成另一个队列，只要出队的元素在队列中就行。可以想象成一个队列只有头和身，另一个队列只有身和尾，而这身是共用的。

在OI赛场上，大多数题目为单调队列力所不能及的，取而代之的是单调队列基础上改进的斜率优化，单调栈等，因为其限制条件，故潜力不大。但需要掌握，因为有许多算法建立在其基础上。

例如斜率优化即为f[i] = min/max{f[j] + g[i] \* g[j]}，和单调队列尤为相似。

单调栈即为单调队列的“栈”版。

这两种复杂度也是O(n)的。

实例一、

我们从最简单的问题开始：

给定一个长度为N的整数数列a(i),i=0,1,...,N-1和窗长度k.

要求：

f(i) = max{a(i-k+1),a(i-k+2),..., a(i)},i = 0,1,...,N-1

问题的另一种描述就是用一个长度为k的窗在整数数列上移动，求窗里面所包含的数的最大值。

解法一：

很直观的一种解法，那就是从数列的开头，将窗放上去，然后找到这最开始的k个数的最大值，然后窗最后移一个单元，继续找到k个数中的最大值。

这种方法每求一个f(i),都要进行k-1次的比较，复杂度为O(N\*k)。

那么有没有更快一点的算法呢？

解法二：

我们知道，上一种算法有一个地方是重复比较了，就是在找当前的f(i)的时候，i的前面k-1个数其它在算f(i-1)的时候我们就比较过了。那么我们能不能保存上一次的结果呢？当然主要是i的前k-1个数中的最大值了。答案是可以，这就要用到单调递减队列。

单调递减队列是这么一个队列，它的头元素一直是队列当中的最大值，而且队列中的值是按照递减的顺序排列的。我们可以从队列的末尾插入一个元素，可以从队列的两端删除元素。

1.首先看插入元素：为了保证队列的递减性，我们在插入元素v的时候，要将队尾的元素和v比较，如果队尾的元素不大于v,则删除队尾的元素，然后继续将新的队尾的元素与v比较，直到队尾的元素大于v,这个时候我们才将v插入到队尾。

2.队尾的删除刚刚已经说了，那么队首的元素什么时候删除呢？由于我们只需要保存i的前k-1个元素中的最大值，所以当队首的元素的索引或下标小于i-k+1的时候，就说明队首的元素对于求f(i)已经没有意义了，因为它已经不在窗里面了。所以当index[队首元素]<i-k+1时，将队首元素删除。

从上面的介绍当中，我们知道，单调队列与队列唯一的不同就在于它不仅要保存元素的值，而且要保存元素的索引（当然在实际应用中我们可以只需要保存索引，而通过索引间接找到当前索引的值）。

举个简单的例子。

假设数列为：8，7，12，5，16，9，17，2，4，6.N=10,k=3.

那么我们构造一个长度为3的单调递减队列：

首先，那8和它的索引0放入队列中，我们用（8，0）表示，每一步插入元素时队列中的元素如下：

0：插入8，队列为：（8，0）

1：插入7，队列为：（8，0），（7，1）

2：插入12，队列为：（12，2）

3：插入5，队列为：（12，2），（5，3）

4：插入16，队列为：（16，4）

5：插入9，队列为：（16，4），（9，5）

。。。。依此类推

那么f(i)就是第i步时队列当中的首元素：8，8，12，12，16，16，。。

实战：

Poj 2823

给定一个数列，从左至右输出每个长度为m的数列段内的最小数和最大数。数列长度：N <=10^6 ，m<=N

显然，如果暴力时间复杂度为O(N\*m) 不超时就怪了。

我们知道，上一种算法有一个地方是重复比较了，就是在找当前的f(i)的时候，i的前面m-1个数其它在算f(i-1)的时候我们就比较过了。

当你一个个往下找时，每一次都是少一个然后多一个，如果少的不是最大值，然后再问新加进来的，看起来很省时间对吧，那么如果少了的是最大值呢？第二个最大值是什么？？

那么我们能不能保存上一次的结果呢？当然主要是i的前k-1个数中的最大值了。答案是可以，这就要用到单调队列。

对于单调队列，我们这样子来定义：

1、维护区间最值

2、去除冗杂状态 如上题，区间中的两个元素a[i],a[j]（假设现在再求最大值） 若 j>i且a[j]>=a[i] ，a[j]比a[i]还大而且还在后面（目前a[j]留在队列肯定比a[i]有用，因为你是往后推， 认真想！ 重点）

3、保持队列单调，最大值是单调递减序列，最小值反之

4、最优选择在队首

大致过程：

1、维护队首（对于上题就是如果你已经是当前的m个之前那你就可以被删了,head++）

2、在队尾插入（每插入一个就要从队尾开始往前去除冗杂状态）

当然还有不少变形，不过大致就是这样，理解好遇到变通你也不会怕了。

举个简单的例子。

数列为：6 4 10 10 8 6 4 2 12 14

N=10,K=3;

那么我们构造一个长度为3的单调递减队列：

首先，那6和它的位置0放入队列中，我们用(6,0)表示，每一步插入元素时队列中的元素如下

插入6：(6,0);

插入4：(6,0),(4,1);

插入10：(10,2);

插入第二个10，保留后面那个：(10,3);

插入8：(10,3),(8,4);

插入6：(10,3),(8,4),(6,5);

插入4，之前的10已经超出范围所以排掉：(8,4),(6,5),(4,6);

插入2，同理：(6,5),(4,6),(2,7);

插入12：(12,8);

插入14：(14,9);

那么f(i)就是第i步时队列当中的首元素：6,6,10,10,10,10,8,6,12,14

同理，最小值也可以用单调队列来做。

如果你看懂了，那你就会发现，单调队列的时间复杂度是O（N），因为每个数只会进队和出队一次，所以这个算法的效率还是很高的。

注意：建议直接用数组模拟单调队列，因为系统自带容器不方便而且不易调试，同时，每个数只会进去一次，所以，数组绝对不会爆，空间也是S（N），优于堆或线段树等数据结构。

算法实现非常精妙，仔细体会！！

代码：

#include<bits/stdc++.h>

using namespace std;

struct node{

int x,y;

}v[1000000];//x表示值，y表示下标

int a[1000],n,m,mx[100000],mn[100000];

void getmin()//单点递增队列

{

int i,head=1,tail=0; //默认起始位置为1 因为插入是v[++tail]故初始化为0

for(i=1;i<m;i++)

{

while(head<=tail && v[tail].x>=a[i]) tail--;

v[++tail].x=a[i],v[tail].y=i;// 根据题目 前m-1个先直接进入队列

}

for(;i<=n;i++)

{

while(head<=tail && v[tail].x>=a[i]) tail--;

v[++tail].x=a[i],v[tail].y=i;

while(v[head].y<i-m+1) head++;//要把已经超出范围的从head开始排出,然后每个队首则是目前m个数的最小值

mn[i-m+1]=v[head].x;

}

}

void getmax()

{

int i,h,t;

h=1;//头

t=0;//尾

for(i=1;i<m;i++)//单调递减队列初始化，首先把m-1个数值入队

{

while( t>=h && v[t].x<=a[i])t--;

v[++t].x=a[i],v[t].y=i;

}

for(;i<=n;i++)//单调递减队列，计算m到n的结果,此时可以开始着手找每个m区间的最大值了

{

while( t>=h && v[t].x<=a[i])t--;

v[++t].x=a[i],v[t].y=i;

//处理队列头部的删除，当头部已经不在m大小窗口内，头指针h++

while(v[h].y<i-m+1)h++;

mx[i-m+1]=v[h].x;//最大值就是每次单调队列头

}

}

int main()

{

int i,j;

scanf("%d%d",&n,&m);

for(int i=1;i<=n;i++)

scanf("%d",&a[i]);

getmax();

getmin();

for(int i=1;i<=n-m+1;i++)

if(i==1)printf("%d",mn[i]);

else printf(" %d",mn[i]);

printf("\n");

for(int i=1;i<=n-m+1;i++)

if(i==1)printf("%d",mx[i]);

else printf(" %d",mx[i]);

printf("\n");

return 0;

}