单调队列优化DP

主讲老师：党东

一、单调队列

【认识单调队列】

单调队列是一种特殊的**双端队列**，其内部元素具有单调性。常见有最大队列和最小队列两种单调队列，其内部元素分别是单调递减和单调递增的。

单调队列有如下两种操作：

插入：如果新元素从队尾插入后会破坏其单调性，则删除队尾元素，直到插入后不再破坏单调性为止，再将其插入单调队列。

获取最大（最小）值：访问队首元素。

元素进队列的过程，以单调递增队列为例：

对于一个元素a，如果a>队尾元素，那么直接将a扔进队列。

如果a < 队尾元素，则将队尾元素出队列，直到满足 a>队尾元素为止。

【单调队列优化动态规划的方法】

■ 单调队列的性质

一般地，利用单调队列优化动态规划时，每个元素一般存储的是两个值：

①、它在原数列中的位置（即下标）

②、它在动态规划中的状态值

而单调队列则保证这两个值同时**单调**。

【单调队列优化动态规划的方法】

- **单调队列在动态规划中的维护**
- **【引例】**一个含有n项的数列(n<=2000000)，求出每一项前面的第m(m<=10000)个数到它这个区间内的最小值。

【单调队列优化动态规划的方法】

- **单调队列在动态规划中的维护**
- 【引例】一个含有n项的数列(n<=2000000)，求出每一项前面的第m(m<=10000)个数到它这个区间内的最小值。

- 【分析】这道题目，我们很容易想到线段树、或者ST算法之类的RMQ问题的解法。但庞大的数据范围让这些对数级的算法没有生存的空间。我们先尝试用动态规划的方法。用f(i)代表第i个数对应的答案，a[i]表示第i个数，很容易写出状态转移方程：

$$f(i) = \mathop{Min}_{j=i-m+1}^{i}(a[j])$$

【单调队列优化动态规划的方法】

■ 状态转移方程：

$$f(i)=\mathop{Min}_{j=i-m+1}^{i}(a[j])$$

这个方程，直接求解的复杂度是O(nm)的，甚至比线段树还差。这时候，单调队列就发挥了它的作用：

我们维护这样一个队列：队列中的每个元素有两个域{pos,value}，分别代表它在原队列中的位置和a[i]，我们随时保持这个队列中的元素两个域都单调递增。

那计算f(i)的时候，只要在队首不断删除，直到队首的pos大于等于i-m+1，那此时队首的value必定是f(i)的最佳选择，因为队列是单调的！

我们看看怎样将a[i]插入到队列中供后面决策：首先，要保证pos单调递增，由于我们动态规划的过程总是由小到大，所以肯定在队尾插入。又因为要保证队列的value单调递增，所以将队尾元素不断删除，直到队尾元素小于a[i]。

【单调队列优化动态规划的方法】

■ 单调队列时间效率分析

很明显的一点，由于每个元素最多出队一次、进队一次，所以时间复杂度是O(n)。用单调队列可以完美的解决这一题。

■ 单调队列的总结

对于下面这一类动态规划问题，我们可以运用单调队列来解决：

$$f(x) = \mathop{opt}_{i=t[x]}^{x-1}(a[i])$$

其中f[x]随着x单调不降，而a[i]则是可以根据i在常数时间内确定的唯一常数。

二、常见例题

（一）、定长连续子区间的最值问题

- 大部分单调队列优化的动态规划问题都和定长连续子区间的最值问题有关。
- 下面通过一道例题来引出有关问题并给出解决方法。

【例1】连续子段和问题 --1542

Description

输入一个长度为 n 的整数序列（A1,A2,……,An），从中找出一段连续的长度不超过M的子序列，使得这个序列的和最大

Input

第一行为用空格分开的两个整数n,m。

第二行为n个用空格分开的整数序列,每个数的绝对值都小于1000。

Output

文件输出仅一个整数表示连续的长度不超过M的最大子序列和。

Sample Input

6 4

1 -3 5 1 -2 3

Sample Output

7

【数据范围】100%的数据N,M<=200000

【例1】连续子段和问题 --1542

【复习回顾】

一个简化的问题最大连续子序列和：**输入一个长度为 n 的整数序列（A1,A2,……,An），从中找出一段连续的子序列，使得这个序列的和最大。**和原问题相比只是没有M这个序列长度的限制！

- 很明显，以第i个数结尾的最大连续子序列，可能存在两种选择：

  情形一：只包含Ai

  情形二：包含Ai和以Ai-1结尾的最大连续子序列；
- 状态：设F[i]表示以第i个数结尾的最大连续子序列的和；
- 则状态转移方程：F[i]=max{F[i-1],0}+a[i]
- 边界条件：F[1]=a[1]
- Answer=max{F[i]}(1<=i<=n)
- 该算法的时间复杂度为O(n)。

【例1】连续子段和问题 --1542

巴蜀中學 BASHU SECONDARY SCHOOL

【思路点拨】

本题最大特点在于对于连续子序列而言，有M这个序列长度的限制。

方法1：DP

设f[i]为以Ai结尾长度不超过M的最大子序列和；

$$F(i)=\max\{\sum_{j=i-k+1}^{i}A_j \mid k=1..m\}$$

对于每个f(i)，从1到m枚举k的值，完成Aj的累加和取最大值。该算法的时间复杂度为$O(n^2)$

【例1】连续子段和问题 --1542

方法2：堆优化

- 先来简化方程： 令S[i]= $\sum_{j=1}^{i} A_j$

  F(i)=max{ $\sum_{j=i-k+1}^{i} A_j \mid k=1..m$ }

  =max{S(i)-S(i-k)|k=1..m}

  =S(i)-min{S(i-k)|k=1..m}

- 上述式子的含义为：对于所有1<=k<=m，找出所有的S(i-k)的最小值。

  求F(i-1)时，求min{S(i-m-1),…,S(i-2)}

  求F(i)时，求min{S(i-m),…,S(i-1)}

- 很明显，可以用一个二叉堆来维护S(i-k)，每次在求F(i)之前的操作如下：

  ① 在堆中删除元素S(i-m-1)，插入元素S(i-1)复杂度O(2log2n)

  ② 从堆中取出当前最小值.复杂度O(1)

- 所以计算的总复杂度为O(n$\log_2$n)。

【例1】连续子段和问题 --1542

方法3：单调队列优化

在方法二中，若考虑用队列来维护决策值S(i-k)。每次只需要在队首删掉S(i-m-1)，在队尾添加S(i-1)。但是取最小值操作还是需要O(n)时间复杂度的扫描，这样时间复杂度为$O(n^2)$。能否进一步优化?

（1）考察在添加S(i-1)的时候，设现在队尾的元素是S(k)，由于$k<i-1$，所以S(k)必然比S(i-1)先出队。若此时$S(i-1)<=S(k)$，则S(k)这个决策永远不会在以后用到，可以将S(k)从队尾删除掉(此时队列的尾部形成了一个类似栈的结构)；

（2）同理，若队列中两个元素S(i)和S(j)，若$i<j$且$S(i)>=S(j)$，则我们可以删掉S(i)（因为S(i)永远不会被用到）。此时的队列中的元素构成了一个单调递增的序列，即：$S1<S2<S3<\ldots<Sk$

【例1】连续子段和问题 --1542

巴蜀中学 BASHU SECONDARY SCHOOL

方法3：单调队列优化

我们来整理在求F(i)的时候，用队列维护S(i-k)所需要的操作：

① 若当前队首元素S(x)，有x<i-m，则S(x)出队；直到队首元素S(x)有x>=i-m为止。

② 若当前队尾元素S(k)>=S(i),则S(k)出队；直到S(k)<S(i)为止。

③ 在队尾插入S(i)

④ 取出队列中的最小值，即队首元素。

由于每一个元素S(i)只进队一次、出队一次，所以队列维护的时间复杂度是O(n)。而每次求f(i)的时候取最小值操作的复杂度是O(1)，所以这一步的总复杂度也是O(n)。

综上所述，该算法的时间复杂度是O(n)。

【例1】连续子段和问题 --1542

```
int s[200005],num[200005],q[200005],i,n,m,head,tail,x,ans=-0x7fffffff; //单调队列存储下标维护前缀和
//num记录队列中元素的编号;q记录队列中元素的值
int main()
{   scanf("%d%d",&n,&m);
    for(i=1;i<=n;i++){scanf("%d",&x);s[i]=s[i-1]+x;}
    tail=1;head=1;q[tail]=0;num[tail]=0;//初始化队列
    for(i=1;i<=n;i++)
    {   while(num[head]<i-m&&head<=tail)head++; //维护队头，队列中的队头元素必须在区间内(i-m+1~i)
        ans=max(ans,s[i]-q[head]);
        //队尾元素的值大于当前待入队前缀和s[i],s[i]替代它后可以产生更大的差值(连续字段和);
        while(q[tail]>=s[i]&&head<=tail)tail--;//维护队尾，类似与栈,从队尾出队
        q[++tail]=s[i]; //入队
        num[tail]=i; //记录元素对应的下标
   }
   printf("%d\n",ans);     return 0;
}
```

【例2】滑动窗口 --1536

【问题描述】给你一个长度为N的数组，一个长为K的滑动窗体从最左端移至最右端，你只能看到窗口中的K个数，每次窗体向右移动一位，如下图：

窗体位置	最小值	最大值
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

你的任务是找出窗体在各个位置时的最大值和最小值。

【文件输入】

第1行：两个整数N和K； 第2行：N个整数，表示数组的N个元素(<=2*10^9)；

【文件输出】

第1行为滑动窗口从左向右移动到每个位置时的最小值，每个数之间用一个空格分开；

第2行为滑动窗口从左向右移动到每个位置时的最大值，每个数之间用一个空格分开。

【例2】滑动窗口 --1536

【样例输入】

8 3

1 3 -1 -3 5 3 6 7

【样例输出】

-1 -3 -3 -3 3 3

3 3 5 5 6 7

【数据范围】

对于20%的数据，K<=N<=1000；

对于50%的数据，K<=N<=100000；

对于100%的数据，K<=N<=1000000；

【例2】滑动窗口 --1536

【思路点拨】

方法1：朴素算法

朴素方法为直接扫描，先枚举起始元素ax，然后求区间[ax,ax+k-1]的最大值和最小值。时间复杂度为O(nk)。

方法2：线段树or RMQ

熟悉数据结构的读者可能看到这个算法还有优化的余地，这是个RMQ问题，使用线段树来求解最大（最小）值，可以将这个算法优化到O(nlog2n)，然而实际上这是个不变的静态序列，我们可以用实际效果更好的离线算法—ST算法，虽然时间复杂度不变，但实际效果却很优秀。

【例2】滑动窗口 --1536

方法3：单调队列

其实，ST算法与线段树都是求解任意长度区间最值的通用算法，但我们似乎都忽略了一个很重要的信息，即所有的区间都是等长且连续的，那么对于“相邻”两个区间(L,r)与(L+1,r+1)有一些极优美的性质：

aL,aL+1,aL+2….ar-1,ar,ar+1

以最大值为例：我们注意到，在区间(L,r)中

max(aL,aL+1,aL+2….ar-1,ar)=max(aL,max(aL+1,aL+2….ar-1,ar))

max(aL+1,aL+2….ar-1,ar,ar+1)=max(max(aL+1,aL+2….ar-1,ar), ar+1)

两个方程有相同的部分max(aL+1,aL+2….ar-1,ar)，经验告诉我们，区间(L,r)中最大值落在(L+1,r)区间的概率很大。那么，在求(L+1,r+1)的最值时，我们完全没有必要再扫描一次。只有当上一次的最值落在了aL上时才需要重新扫描，这样，算法得到了极大的优化。

【例2】滑动窗口 --1536

方法3：单调队列

继续思考这样的一个问题，以最大值为例，对任意L<=i<=j<=r，如果ai<aj，那么在区间向右移动的过程中，最大值永远也不会落在ai上，因为ai比aj先失效，能用ai一定能用aj，此时，我们便不再需要ai了，这个性质很明显与单调队列的性质重合了。

当我们将区间从(L,r)移动到(L+1,r+1)时，将ar+1插入单调队列中，若队首元素不在(L,r)区间当中，则删除它。

这样处理后的队首元素便是(L+1,r+1)区间内的最大值。

时间复杂度为O(n)。

【例2】滑动窗口 --1536

```
int n,k,a[1000001];
struct xx{int sta,pos;}q[1000001];
void work(bool xx)
{  int L=1,R=0,i;
   for(i=1;i<k;i++)
   {  if(xx)while(a[i]<q[R].sta&&R>0)R--;//队尾
         else while(a[i]>q[R].sta&&R>0)R--;
      q[++R].sta=a[i];q[R].pos=i;//入队
   }
   for(i=k;i<=n;i++)
   {  if(xx)while(a[i]<q[R].sta&&R>=L)R--;//队尾
         else while(a[i]>q[R].sta&&L<=R)R--;
      q[++R].sta=a[i];q[R].pos=i;
      while(q[L].pos+k<=i&&L<=R)L++;//队头
      printf("%d ",q[L].sta);
   }
   printf("\n");
}
```

```
int main()
{  int i,j;
   scanf("%d%d",&n,&k);
   for(i=1;i<=n;i++)scanf("%d",&a[i]);
   work(1);//最小值
   work(0);//最大值
}
```

【例2】滑动窗口 --1536

【小结】

在本题的解决过程中，我们首先通过对于一个简化的问题解答，得到了该类问题一般性的解决思路，随后得到了一个$O(n^2)$的算法

我们通过简化方程，进一步明确和细化了求解目标，运用合理的数据结构——堆，得到了一个O(nlog2n)的算法。

通过进一步的挖掘求解目标的内涵，寻找内在规律，我们得到只用线性表维护的O(n)的算法。

【例3】绿色通道 --1537

【题目描述】 高二数学《绿色通道》总共有n道题目要写(其实是抄)，编号1..n，抄每道题所花时间不一样，抄第i题要花a[i]分钟。由于lsz还要准备NOIP，显然不能成天写绿色通道。lsz决定只用不超过t分钟时间抄这个，因此必然有空着的题。每道题要么不写，要么抄完，不能写一半。一段连续的空题称为一个空题段，它的长度就是所包含的题目数。这样应付自然会引起马老师的愤怒。马老师发怒的程度(简称发怒度)等于最长的空题段长度。

现在，lsz想知道他在这t分钟内写哪些题，才能够尽量降低马老师的发怒度。由于lsz很聪明，你只要告诉他发怒度的数值就可以了，不需输出方案。

【文件输入】输入文件第一行为两个整数n,t，代表共有n道题目,t分钟时间。以下一行，为n个整数，依次为a[1], a[2],... a[n]，意义如上所述。

【文件输出】输出文件仅一行，一个整数w，为最低的发怒度。

【样例输入】

17 11

6 4 5 2 5 3 4 5 2 3 4 5 2 3 6 3 5

【样例输出】3

【数据规模】 100%数据0<n<=50000，0<a[i]<=3000，0<t<=100000000

【例3】绿色通道 --1537

【样例解释】

分别写第4,6,10,14题，共用时2+3+3+3=11分钟。空题段：1-3(长度为3), 5-5(1), 7-9(3), 11-13(3), 15-17(3)。所以发怒度为3。可以证明，此数据中不存在使得发怒度<=2的作法。

【思路点拨】这题的实质是使空白段得最大值最小。类似的使最大值最小，最小值最大的问题，一般用二分答案解决。

构造函数ok(x)，表示能否在t时间内达到发怒度不超过x的要求。显然，最终答案w满足：当x<w(最终答案)时，ok(x)=false；当x>=w时，ok(x)=true。假设有一个高效的求ok(x)的算法，则可以通过二分x来获得w。下面叙述求ok(x)的算法。

最普通的计算ok(x)的方法是dp。令a[i]表示写第i题所需时间，f[i]表示1~i题符合发怒度不超过x的要求且第i题的状态为“写”的最短时间，则状态转移方程为：

F[i]=min{f[j]}+a[i],i-x-1<=j<i,j>=0,1<=i<=n

计算ok(x)的复杂度为O(xn)，总复杂度为O(n^2logn)。(因为不知道x为多少，最坏情况为n-1)这样并不能通过本题。在DP的时候可以用堆或单调队列优化。

【例3】绿色通道 --1537

【思路点拨】

下面用单调队列优化：

可以看到，枚举f[j]并选出最小值花费了大量时间。实际上存在O(1)时间取最小值的方法：

维护一个队列Q，Q中元素为“有用”的，且“有可能为最优解”的题号。初始只有一个0。

对每一个题号i，执行以下操作：

令Q[L]表示队首的元素。F[i]=f[Q[L]]+a[i]；表示写完j题以后写i题。

从队尾向前检查，若f[Q[r]]>f[i]则弹出，否则停止检查，并插入i。

如果Q[L]<i-x则弹出Q[L]

最后检查f[n+1]是否<=t，如果<=t，则ok(x)=true，否则false。

简单解释：这个队列中元素的值(题号)及相应f值都是递增的。对队首元素的检查则保证队内的元素都在计算下一个f[i]时合法，且队首元素f值为最小。所以各个f[i]值都是正确的。

Ok(x)复杂度O(n)，总复杂度为O(nlogn)。

【例3】绿色通道 --1537

【参考代码】

```
const int maxn=100005,oo=0x7fffffff;
int n,limit,a[maxn],q[maxn],f[maxn];
bool DP(int x)
{  int L=1,r=1,i,ans=oo;
   f[0]=0;q[1]=0;
   for(i=1;i<=n+1;i++)
   {  f[i]=f[q[L]]+a[i] ;
      while(f[i]<=f[q[r]]&&L<=r)r--;
      q[++r]=i;
      while(q[L]<i-x&&L<=r)L++;
   }
   if(f[n+1]<=limit)return 1;//注意此处,相当于设置一个哨兵
   return 0;
}
```

【例3】绿色通道 --1537

【参考代码】

```
const int maxn=100005,oo=0x7fffffff;
int n,limit,a[maxn],q[maxn],f[maxn];
void Solve()
{  int L=0,r=n,mid;
   while(L<r)
   {  mid=(L+r)>>1;
      if(DP(mid))r=mid;else L=mid+1;
   }
   cout<<L<<endl;
}
int main()
{  cin>>n>>limit;
   for(int i=1;i<=n;i++)cin>>a[i];
   Solve();
}
```

【例4】修剪草坪 --// 2940 //luogu:2627

【问题描述】在一年前赢得了小镇的最佳草坪比赛后，FJ变得很懒，再也没有修剪过草坪。现在，新一轮的最佳草坪比赛又开始了，FJ希望能够再次夺冠。

然而，FJ的草坪非常脏乱，因此，FJ只能够让他的奶牛来完成这项工作。FJ有N(1<=N<=100,000)只排成一排的奶牛，编号为1...N。每只奶牛的效率是不同的，奶牛i的效率为E_i(0<=E_i<=1,000,000,000)。

靠近的奶牛们很熟悉，如果FJ安排超过K只连续的奶牛，那么这些奶牛就会罢工去开派对:)。因此，现在FJ需要你的帮助，计算FJ可以得到的最大效率，并且该方案中没有连续的超过K只奶牛。

【输入】

第一行：空格隔开的两个整数N和K

第二到N+1行：第i+1行有一个整数E_i

【输出】

第一行：一个值，表示FJ可以得到的最大的效率值。

【例4】修剪草坪 --// 2940 //luogu:2627

【样例输入】

5 2

1

2

3

4

5

【输入解释】FJ有5只奶牛，他们的效率为1，2，3，4，5。他们希望选取效率总和最大的奶牛，但是他不能选取超过2只连续的奶牛

【样例输出】12

【输出解释】FJ可以选择出了第三只以外的其他奶牛，总的效率为1+2+4+5=12。

【例4】修剪草坪 --// 2940 //luogu:2627

【思路点拨】

- 状态：设f[i][0]表示以i结尾并且i这个数字不选所得的最大效率值；

  f[i][1]表示以i结尾并且i这个数字选所得的最大效率值；

- 转移方程为：$f[i][1]=\max\limits_{i-k<=j<i}\{f[j][0]+sum[i]-sum[j]\}$

  f[i][0]=max{f[i-1][0],f[i-1][1]}

- 可以变形为：$f[i][1]=\max\limits_{i-k<=j<i}\{f[j][0]-sum[j]\}+sum[i]$ ，这样可以用单调队列进行优化。

【例4】修剪草坪 --1538

【参考代码】

```
long long a[100001]={0},sum[100001]={0};
long long f[100001][2]={0},q[100001]={0};
int main()
{  int i,n,k,st,en;
   cin>>n>>k;
   for(i=1;i<=n;i++){scanf("%lld",&a[i]);sum[i]=sum[i-1]+a[i];}
   st=1;en=1;
   for(i=1;i<=n;i++)
   {  f[i][0]=max(f[i-1][0],f[i-1][1]);
      while(q[st]<i-k&&st<=en)st++;
      f[i][1]=f[q[st]][0]-sum[q[st]]+sum[i];
      while(f[i][0]-sum[i]>f[q[en]][0]-sum[q[en]]&&st<=en)en--;
      q[++en]=i;
   }
   cout<<max(f[n][0],f[n][1]);
}
```