单调队列

单调队列并非一种特定的数据结构，而是一种解题思想。借助这个思想，使队列中的元素保持单调递增或单调递减，它的优点在于可以很方便地求出某元素的左边（或右边）第一个比它小或大的元素，总时间复杂度O(N)。

下面以在单向队列中维护单调性，模拟进队出队过程。

例：在序列a（1， 7， 5， 9， 4， 8）求每一项左边第一个比它大的值。

（1）初始时，队列为空，即a[1]的左边不存在比它大的值。
1入队.....队内元素（1）

（2）7待入队，队尾元素1小于7，1出队，队列为空，即a[2]的左边不存在比它大的值。
7入队.....队内元素（7）

（3）5待入队，队尾元素7大于5，即a[3]的左边第一个比它大的元素为7。
5入队.....队内元素（7，5）

（4）9待入队，队尾元素5小于9，弹出。新的队尾7小于9，继续弹出，队列为空，即a[4]的左边不存在比它大的值。
9入队.....队内元素（9）

（5）4待入队，队尾元素9大于4，即a[5]的左边第一个比它大的元素为9。
4入队.....队内元素（9，4）

（5）8待入队，队尾元素4小于8，弹出。新的队尾9大于8，即a[6]的左边第一个比它大的元素为9。
8入队.....队内元素（9，8）

在这类操作中，我们都是在队尾进行插入和删除，因此可用栈来解决这类问题，称单调栈。

在实际应用中，还常有通过在双端队列中通过双端的删除来维护单调性。在这类问题的处理规则中，除了要考虑值的大小，也需将元素的距离考虑在内。

例：在序列中找出每三个数的最大值。

该问题为滑动窗口问题，求区间最值。使用RMQ算法时间复杂度为O（$nlogn+(n-k+1)$）。利用单调递减队列维护，可将时间复杂度优化为O（n）。

下面以序列a（1，5，7，3，2，6，4）模拟求解过程。

（1）初始时队列为空，a[1]直接入队。队列中的元素除了记录值，还需记录下标，以进行距离判断。

值	**1**								
下标	1								

（2）a[2]待入队，5大于队尾元素的值1，弹出，队为空，a[2]入队。

值	**5**								
下标	2								

（3）a[3]待入队，7大于队尾元素的值5，弹出，队为空，a[3]入队。

值	**7**								
下标	3								

此时，队头的值7为a[1]--a[3]这组中的最大值。

（4）a[4]待入队，3小于队尾元素的值1，符合单调递减性质，a[4]入队。

值	**7**	**3**							
下标	3	4							

此时，队头的值7为a[2]--a[4]这组中的最大值。

（5）a[5]待入队，2小于队尾元素的值3，符合单调递减性质，a[5]入队。

值	**7**	**3**	**2**						
下标	3	4	5						

此时，队头的值7为a[3]--a[5]这组中的最大值。

（6）a[6]待入队，6大于队尾的值2，弹出产生新的队尾，6依旧大于队尾的值3，继续弹出产生新的队尾。6小于队尾值7，a[6]入队。

值	**7**	**6**							
下标	3	6							

但此时，队头元素为a[3]，与a[6]的距离超过3，队头需出队。

值	**6**								
下标	6								

则队头的值6为a[4]--a[6]这组的最大值。

（7）a[7]待入队，4小于队尾元素的值6，符合单调递减性质，a[7]入队。

值	**6**	**4**							
下标	6	7							

队头的值6为a[5]--a[7]这组的最大值。

维护好队列的单调性，取出最值的时间复杂度仅为O(1)。序列中每个元素遍历判断后入队，整个过程的时间复杂度为O(n)，这为该思路的优势，在动态规划算法中也常利用这个思想来进一步优化。

A. 懒惰的牛

【题目描述】

在炎热的夏天，奶牛Bessie懒得动弹。她想在草场中选择一个可以走很短的距离就吃到尽可能多青草的位置呆着。
Bessie的可视作一维数轴的草场中有N块草地(1 <= N <= 100,000)。第i块草地上长有g_i个单位的青草(1 <= g_i <= 10,000)，位于草场上的点x_i(0 <= x_i <= 1,000,000)，每块草场的位置均不同。Bessie想选择一个点作为她的初始位置（可以和草地的位置相同），从该点出发走不超过K(1 <= K <= 2,000,000)步可以吃到总量最多的青草。请帮助Bessie确定如果她选择最佳的初始位置，她最多可以吃到多少青草。

【输入】

第1行：整数N和K。
第2..1+N行：第i+1行用两个整数g_i和x_i描述第i块草地。

【输出】

第1行：以Bessie的最佳位置为中心K距离之内的青草总量。

样例输入	样例输出
4 3 4 7 10 15 2 2 5 1	11

【提示】

Bessie应该将初始位置设在x=4处，这样x=1，x=2和x=7处的青草她都能吃到。

【分析】

该问题要求出全局中区间和的最优值，区间大小为$2*k$。队列中通过队头与队尾的距离维护区间的长度在$2*k$之内，并定义变量sum动态维护区间和的值实现求解过程。

【代码】

```
#include<iostream>
#include<queue>
#include<algorithm>
using namespace std;
int n,k,maxn,sum;
struct point{
    int x,g;
}a[100001];
queue<point> q;
bool cmd(point f,point t){
    if(f.x!=t.x) return f.x<t.x;
}
int main(){
    cin>>n>>k;
    for(int i=1;i<=n;i++){
        cin>>a[i].g>>a[i].x;
    }
    sort(a+1,a+n+1,cmd);
    for(int i=n;i>=1;i--){
        q.push(a[i]);
        sum+=a[i].g;//动态累计区间总量
        int t=q.front().x;//队头坐标
        int w=q.back().x;//队尾坐标
```

```
        while(t-w>2*k){//维护区间范围长度
            sum-=q.front().g;
            q.pop();
            t=q.front().x;
        }
        if(sum>maxn) maxn=sum;
    }
    cout<<maxn;
}
```

B.滑动的窗户

【题目描述】

在一个包含n个元素的数组上，有一个长度为k的窗户在从左向右滑动。窗户每滑动到一个位置，我们都可以看到k个元素在窗户中。如下的例子所示，假设数组为 [1 3 -1 -3 5 3 6 7]，而k等于3:

窗户位置	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

对于窗户滑动过的每个位置，请给出窗户内k个元素的最小值和最大值。

【输入】

输入的第一行包括两个整数n，k，n表示数组的长度，k表示窗户的长度。
接下来一行包括n个整数，表示这个n个元素的数组。

【输出】

输出包含两行，每行包括n-k+1个整数，第一行表示窗户从左到右滑动过程中的最小值，第二行表示窗户从左到右滑动过程中的最大值。

样例输入	样例输出
8 3 1 3 -1 -3 5 3 6 7	-1 -3 -3 -3 3 3 3 3 5 5 6 7

【提示】

对于100%的数据，3<=n<=1000000，1<=k<=n，数组中的每个元素均在int范围内

【分析】模板题，分两步分别求长度为k的区间的最大值与最小值。

【代码】

```
#include<cstdio>
#include<deque>
```

```cpp
using namespace std;
int n,K;
struct node{
    int k,s;
}a[1000001];
deque<node> q;
int main(){
    scanf("%d%d",&n,&K);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i].s);
        a[i].k=i;
    }
    for(int i=1;i<=n;i++){
        while(!q.empty()&&a[i].k-q.front().k>=K) q.pop_front();
        while(!q.empty()&&a[i].s<=q.back().s) q.pop_back();
        q.push_back(a[i]);
        if(i>=K) printf("%d ",q.front().s);
    }
    while(!q.empty()) q.pop_back();//注意求最大值前需清空队列
    printf("\n");
    for(int i=1;i<=n;i++){
        while(!q.empty()&&a[i].k-q.front().k>=K) q.pop_front();
        while(!q.empty()&&a[i].s>=q.back().s) q.pop_back();
        q.push_back(a[i]);
        if(i>=K) printf("%d ",q.front().s);
    }
}
```

当队列需要重复使用时，队列的初始化会耗费一定时间。同样在单调队列的思路下，我们可以使用数组的方式来书写代码，设置变量f和r分别指向队头和队尾来操作。

```cpp
#include<cstdio>
using namespace std;
struct node{
    int v,p;
}q[10000001];
int n,k,a[10000001],f,r;
int main(){
    scanf("%d %d",&n,&k);
    for(int i=1;i<=n;i++)scanf("%d",&a[i]);
    for(int i=1;i<=n;i++){
        while(f<=r&&i-q[f].p>=k)f++;//队头出队
        while(f<=r&&q[r].v>=a[i])r--;//队尾出队
        q[++r]=(node){a[i],i};//元素入队
        if(i>=k)printf("%d ",q[f].v);
    }
    printf("\n");
    f=r=0;
    for(int i=1;i<=n;i++){
        while(f<=r&&i-q[f].p>=k)f++;
        while(f<=r&&q[r].v<=a[i])r--;
        q[++r]=(node){a[i],i};
        if(i>=k)printf("%d ",q[f].v);
    }
    printf("\n");
}
```

C. 最大子序和

【题目描述】

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。例如：
序列 1，-3，5，1，-2，3
当M=2或3时，S=5+1=6；当M=4时，S=5+1+(-2)+3=7。

【输入】

输入文件第一行一个整数n表示序列的长度，第二行n个整数，代表序列的元素。第三行一个整数表示M。

【输出】

一个整数，即子序列的最大和。保证结果不超过longint范围。

样例输入	样例输出
6 1 -3 5 1 -2 3 3	6

【提示】

数据范围:
50%的数据$N, M <= 1000$
100%的数据$N, M <= 20000$

【分析】

本题可先用前缀和处理数据，通过队列中队尾元素的值a[r]减去队头元素的值a[f]，可求得f+1至r之间的区间和。队列处理过程用维护单调递增性，使得队头元素值越小，则总和能取得更大的值。。

【代码】

```
#include<cstdio>
#include<deque>
using namespace std;
int n,a[20001],m,sum,maxn=-0x7fffff;
struct node{
    int x,p;
};
deque<node> q;
int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
        a[i]+=a[i-1];
    }
    scanf("%d",&m);
    q.push_back((node){0,0});//需考虑第一个元素也包含在区域内的情况
    for(int i=1;i<=n;i++){
        while(!q.empty()&&i-q.front().p>=m) q.pop_front();//长度限定
        while(!q.empty()&&a[i]<=q.back().x) q.pop_back();//维护队列递增状态
        q.push_back((node){a[i],i});
        maxn=max(maxn,a[i]-q.front().x);
```

```
    }
    printf("%d",maxn);
}
```

D.修剪草坪

【题目描述】

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 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。
靠近的奶牛们很熟悉，因此，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:)。
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 1 行：空格隔开的两个整数 N 和 K；
第 2 到 N+1 行：第 i+1 行有一个整数 E_i，表示第 i 只奶牛的效率；

【输出】

输出共一行一行一个整数，表示 FJ 可以得到的最大的效率值。

样例输入	样例输出
5 2 1 2 3 4 5	12

【提示】

[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；
[输出解释]FJ 可以选择除了第三只以外的其他奶牛，总的效率为 1+2+4+5=12。
[数据规模]
对于 40%的数据： $1 \leq N \leq 100$； 对于 60%的数据： $1 \leq N \leq 2,000$；
对于 100%的数据：$1 \leq N \leq 100,000$；$1 \leq K \leq N$；$0 \leq E_i \leq 1,000,000,000$；

【分析】本题有一定难度，非求一个区间的最值，而是求多个连续区间和的最值。因此可在序列中挑出不参与方案的一个子序列，该子序列的和为损失值，将所有奶牛的效率总和减去该损失值进行求解。可见损失值越小，全局效率值越大。应用动态规划算法，f[i]表示第i只奶牛不选择，且前i只的损失和最小。在这个过程中，处理第i只奶牛时，则需要遍历第1至第i-1只奶牛的情况，将状态进行转移后取最小值。我们可以利用单调队列的思想对动规进行优化，队列维护单调递增的性质，使得队头元素的损失值为最小，若其位置与当前队尾的距离不超过k+1的距离，则损失值可直接累加到队尾元素项，时间复杂度可优化为O(n)。

【代码】

```
#include<iostream>
#include<deque>
using namespace std;
int n,k;
```

```cpp
long long f[100001],ans,minn=0x7fffffffffffffff;
deque<int> q;
int main(){
    cin>>n>>k;
    q.push_back(0);//第0只不选，即第i只之前全选
    for(int i=1;i<=n;i++){
        cin>>f[i];//边界
        ans+=f[i];
        while(!q.empty()&&i-q.front()>k+1) q.pop_front();
        if(!q.empty()) f[i]=f[i]+f[q.front()];//损失值叠加
        while(!q.empty()&&f[i]<=f[q.back()]) q.pop_back();//维护递增
        q.push_back(i);
        if(i>=n-k) minn=min(minn,f[i]);
    }
    cout<<ans-minn;
}
```

E.跳棋

【题目描述】

小明迷恋上了一个新的跳棋游戏，游戏规则如下：棋盘是一排从0开始，顺序编号的格子，游戏开始时你位于0号格子，你每次只能往编号大的格子跳，而且你每次至少需要跳过L个格子，至多只能跳过R个格子。每个格子都有一个给定的伤害值，显然你希望得到的伤害值越少越好。
你能告诉小明他当他跳到最后一个格子时受到的累积伤害值最小为多少吗？
如果无论如何小明都无法跳到最后一个格子，这个时候你需要输出"-1"。
注：从i号格子跳过x个格子表示从i号格子跳到第i+x+1号格子。

【输入】

输入文件jump.in第一行有三个整数n、L和R，n表示格子的编号从0到n。L和R表示最少需要跳过的格子数和最多能够跳过的格子数。
第二行有n个正整数，两个数字间用空格隔开，表示每个格子的伤害值。

【输出】

输出文件jump.out仅有一个整数，表示受到的最小伤害值，保证结果小于maxlongint。

样例输入	样例输出
10 2 6 1 3 5 7 9 2 4 6 8 10	12

【提示】

0	1	3	5	7	9	2	4	6	8	10
★						★				★

[数据规模]
50%的数据，$1 <= n <= 1000$
65%的数据，$1 <= n <= 10000$
100%的数据，$1 <= n <= 1000000$，$1 <= L <= R <= n$
其中有15%的数据，$1 <= n <= 1000000$，$1 <= L <= R <= 10$

【分析】

本题与上一题类似，均是利用单调队列对动态规划进行优化。f[i]表示到达第i个格子的最小损耗值，队列依据每个格子的最小损耗值维护单调递增性，使队头的值最小，将其状态转移到f[i]，即可得到第i个格子的最有策略。但需要注意的是，队列中存储的必须是可到达第i点的起跳点，即若队头的坐标无法跳至第i个格子，则需要弹出，不得进行状态的转移。

【代码】

```
#include<iostream>
#include<cstring>
#include<queue>
using namespace std;
struct node{
    int id,v;
};
deque<node> q;
int n,l,r,a[1000001],f[1000001];
int main(){
    cin>>n>>l>>r;
    for(int i=1;i<=n;i++)cin>>a[i];
    memset(f,127,sizeof(f));
    int inf=f[0];
    f[0]=0;
    for(int i=l+1;i<=n;i++){
        while(!q.empty()&&q.front().id<i-r-1)q.pop_front();
        int v=f[i-l-1];
        while(!q.empty()&&q.back().v>=v)q.pop_back();
        q.push_back((node){i-l-1,v});
        f[i]=q.front().v+a[i];
    }
    if(f[n]<inf)cout<<f[n];
    else cout<<-1;
}
```

F.跳房子

【题目描述】

跳房子，也叫跳飞机，是一种世界性的儿童游戏，也是中国民间传统的体育游戏之一。
跳房子的游戏规则如下：
在地面上确定一个起点，然后在起点右侧画n 个格子，这些格子都在同一条直线上。每个格子内有一个数字（整数），表示到达这个格子能得到的分数。玩家第一次从起点开始向右跳，跳到起点右侧的一个格子内。第二次再从当前位置继续向右跳，依此类推。规则规定：
玩家每次都必须跳到当前位置右侧的一个格子内。玩家可以在任意时刻结束游戏，获得的分数为曾经到达过的格子中的数字之和。
现在小R 研发了一款弹跳机器人来参加这个游戏。但是这个机器人有一个非常严重的缺陷，它每次向右弹跳的距离只能为固定的d。小R 希望改进他的机器人，如果他花g 个金 币改进他的机器人，那么他的机器人灵活性就能增加g，但是需要注意的是，每次弹跳的距 离至少为1。具体而言：
当g < d时，他的机器人每次可以选择向右弹跳的距离为d-g, d-g+1,d-g+2，...，d+g-2，d+g-1，d+g；
否则（当g ≥ d时），他的机器人每次可以选择向右弹跳的距离为1，2，3，...，d+g-2，d+g-1，d+g。
现在小R 希望获得至少k 分，请问他至少要花多少金币来改造他的机器人。

【输入】

第一行三个正整数n，d，k，分别表示格子的数目，改进前机器人弹跳的固定距离，以及希望至少获得的分数。相邻两个数之间用一个空格隔开。
接下来n 行，每行两个正整数xi ,si，分别表示起点到第i个格子的距离以及第i个格子的分数。两个数之间用一个空格隔开。保证xi按递增顺序输入。

【输出】

共一行，一个整数，表示至少要花多少金币来改造他的机器人。若无论如何他都无法获得至少k 分，输出-1。

样例输入	样例输出
7 4 10 2 6 5 -3 10 3 11 -3 13 1 17 6 20 2	2
7 4 20 2 6 5 -3 10 3 11 -3 13 1 17 6 20 2	-1

【提示】

[输入输出样例1 说明]
花费 2 个金币改进后，小R 的机器人依次选择的向右弹跳的距离分别为2，3，5，3，4， 3，先后到达的位置分别为2，5，10，13，17，20，对应1, 2, 3, 5, 6, 7 这6 个格子。这些格 子中的数字之和15 即为小R 获得的分数。
[输入输出样例2 说明]
由于样例中7 个格子组合的最大可能数字之和只有18 ，无论如何都无法获得20 分。
[数据规模与约定]
本题共10 组测试数据，每组数据10 分。
对于全部的数据满足$1 \leq n \leq 500000, 1 \leq d \leq 2000, 1 \leq xi, k \leq 10^9$,$|si| < 10^5$。
对于第1，2 组测试数据，$n \leq 10$；
对于第3，4，5 组测试数据，$n \leq 500$
对于第6，7，8 组测试数据，$d = 1$

【分析】

本题首先需要利用二分算法假设可行性解m，再通过动规思想求到达每个格子的的最大分值f[i]，通过判断是否有某个点能达到k来决断m是否可行，通过不断的缩小范围，求最大值最小。

【代码】

本题在判断可行性解部分如直接用动规来书写，并不难理解。

```
#include<cstdio>
#include<deque>
```

```cpp
#include<cstring>
using namespace std;
long long n,k,d,f[500001];
struct point{
    int x,s;
}a[500001];
bool check(int mid){
    long long x=max(d-mid,(long long)1);
    long long y=d+mid;
    memset(f,-128,sizeof(f));
    f[0]=0;
    for(int i=1;i<=n;i++){//遍历目的地
        for(int j=i-1;j>=0;j--){//往前遍历可能的起跳点
            if(a[i].x-a[j].x>y) break;
            if(a[i].x-a[j].x<x) continue;
            f[i]=max(f[i],f[j]+a[i].s);//求最有策略
            if(f[i]>=k) return true;
        }
    }
    return false;
}
int main(){
    scanf("%lld %lld %lld",&n,&d,&k);
    for(long long i=1;i<=n;i++) {
        scanf("%lld %lld",&a[i].x,&a[i].s);
    }
    long long ub=a[n].x+1,lb=0,mid;
    while(ub-lb>1) {
        mid=(ub+lb)>>1;
        if(check(mid))ub=mid;
        else lb=mid;
    }
    if(ub==a[n].x+1)printf("-1\n");
    else printf("%lld",ub);
}
```

但是，在动规求解的过程中，时间复杂度是接近O(n^2)的，虽然循环的中间添加了特判使之可以提前退出。但由于本题数据量较大，假设最大的金币花费使之能一步跳跃到终点的情况，时间复杂度会达O($lb(xi)*n^2$)。以上代码因超时只能拿80分。在本题动规求解的过程，我们可使用单调队列进行优化，维护队列的单调递减，使之将可行的最优起跳点的状态进行转移。

```cpp
#include<cstdio>
#include<deque>
#include<cstring>
using namespace std;
long long n,d,k,head,f[500005];
struct point{
    long long x,s;
}a[500001];
bool check(long long m){
    deque<long long>q;
    head=0;
    for(long long i=1;i<=n;i++) {//枚举每个目的地
        long long t=max(d-m,(long long)1);
        while(head<i&&a[head].x<=a[i].x-t) {
            while(q.size()&&f[q.back()]<=f[head]){//维护递减
                q.pop_back();
```

```
            }
            q.push_back(head++);
            //通过head变量逐个判断起跳点，避免在决策不同的目的地时重复枚举
        }
        while(q.size()&&a[q.front()].x<a[i].x-(d+m)){
            q.pop_front();
        }
        if(q.empty())f[i]=-9999999999999;//队为空，说明无法到达，赋予极小值。
        else f[i]=f[q.front()]+a[i].s;
        if(f[i]>=k)return true;
    }
    return false;
}
int main(){
    scanf("%d %d %d",&n,&d,&k);
    for(long long i=1;i<=n;i++) {
        scanf("%lld %lld",&a[i].x,&a[i].s);
    }
    long long ub=a[n].x+1,lb=0,mid;
    while(ub-lb>1) {
        mid=(ub+lb)>>1;
        if(check(mid))ub=mid;
        else lb=mid;
    }
    if(ub==a[n].x+1)printf("-1\n");
    else printf("%lld",ub);
}
```