单调队列

引入

在学习单调队列前，让我们先来看一道例题。

> **例题**
>
> Sliding Window
>
> 本题大意是给出一个长度为 n 的数组，编程输出每 k 个连续的数中的最大值和最小值。

最暴力的想法很简单，对于每一段 $i \sim i+k-1$ 的序列，逐个比较来找出最大值（和最小值），时间复杂度约为 $O(n \times k)$。

很显然，这其中进行了大量重复工作，除了开头 $k-1$ 个和结尾 $k-1$ 个数之外，每个数都进行了 k 次比较，而题中 100% 的数据为 $n \leq 1000000$，当 k 稍大的情况下，显然会 TLE。

这时所用到的就是单调队列了。

定义

顾名思义，单调队列的重点分为「单调」和「队列」。

「单调」指的是元素的「规律」——递增（或递减）。

「队列」指的是元素只能从队头和队尾进行操作。

Ps. 单调队列中的 "队列" 与正常的队列有一定的区别，稍后会提到

例题分析

解释

有了上面「单调队列」的概念，很容易想到用单调队列进行优化。

要求的是每连续的 k 个数中的最大（最小）值，很明显，当一个数进入所要 "寻找" 最大值的范围中时，若这个数比其前面（先进队）的数要大，显然，前面的数会比这个数先出队且不再可能是最大值。

也就是说——当满足以上条件时，可将前面的数 "弹出"，再将该数真正 push 进队尾。

这就相当于维护了一个递减的队列，符合单调队列的定义，减少了重复的比较次数，不仅如此，由于维护出的队伍是查询范围内的且是递减的，队头必定是该查询区域内的最大值，因此输出时只需输出队头即可。

显而易见的是，在这样的算法中，每个数只要进队与出队各一次，因此时间复杂度被降到了 $O(n)$。

而由于查询区间长度是固定的，超出查询空间的值再大也不能输出，因此还需要 site 数组记录第 i 个队中的数在原数组中的位置，以弹出越界的队头。

过程

> 例如我们构造一个单调递增的队列会如下：

原序列为：

```
1 3 -1 -3 5 3 6 7
```

因为我们始终要维护队列保证其 **递增** 的特点，所以会有如下的事情发生：（假设 $k=3$）

操作	队列状态
1 入队	`{1}`
3 比 1 大，3 入队	`{1 3}`
-1 比队列中所有元素小，所以清空队列 -1 入队	`{-1}`
-3 比队列中所有元素小，所以清空队列 -3 入队	`{-3}`
5 比 -3 大，直接入队	`{-3 5}`
3 比 5 小，5 出队，3 入队	`{-3 3}`
-3 已经在窗体外，所以 -3 出队；6 比 3 大，6 入队	`{3 6}`
7 比 6 大，7 入队	`{3 6 7}`

例题参考代码

```cpp
#include <cstdlib>
#include <cstring>
#include <iostream>
constexpr int MAXN = 1000100;
using namespace std;
int q[MAXN], a[MAXN];
int n, k;

void getmin() {  // 得到这个队列里的最小值，直接找到最后的就行了
  int head = 0, tail = -1;
  for (int i = 1; i < k; i++) {
    while (head <= tail && a[q[tail]] >= a[i]) tail--;
    q[++tail] = i;
  }
  for (int i = k; i <= n; i++) {
    while (head <= tail && a[q[tail]] >= a[i]) tail--;
    q[++tail] = i;
    while (q[head] <= i - k) head++;
    cout << a[q[head]] << ' ';
  }
}

void getmax() {  // 和上面同理
  int head = 0, tail = -1;
  for (int i = 1; i < k; i++) {
    while (head <= tail && a[q[tail]] <= a[i]) tail--;
    q[++tail] = i;
  }
  for (int i = k; i <= n; i++) {
    while (head <= tail && a[q[tail]] <= a[i]) tail--;
    q[++tail] = i;
    while (q[head] <= i - k) head++;
    cout << a[q[head]] << ' ';
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> k;
  for (int i = 1; i <= n; i++) cin >> a[i];
  getmin();
  cout << '\n';
  getmax();
  cout << '\n';
  return 0;
}
```

Ps. 此处的 "队列" 跟普通队列的一大不同就在于可以从队尾进行操作，STL 中有类似的数据结构 deque。

例题 2 Luogu P2698 Flowerpot S

给出 N 滴水的坐标，y 表示水滴的高度，x 表示它下落到 x 轴的位置。每滴水以每秒 1 个单位长度的速度下落。你需要把花盆放在 x 轴上的某个位置，使得从被花盆接着的第 1 滴水开始，到被花盆接着的最后 1 滴水结束，之间的时间差至少为 D。 我们认为，只要水滴落到 x 轴上，与花盆的边沿对齐，就认为被接住。给出 N 滴水的坐标和 D 的大小，请算出最小的花盆的宽度 W。
$1 \le N \le 100000, 1 \le D \le 1000000, 0 \le x, y \le 10^6$

将所有水滴按照 x 坐标排序之后，题意可以转化为求一个 x 坐标差最小的区间使得这个区间内 y 坐标的最大值和最小值之差至少为 D。我们发现这道题和上一道例题有相似之处，就是都与一个区间内的最大值最小值有关，但是这道题区间的大小不确定，而且区间大小本身还是我们要求的答案。

我们依然可以使用一个递增，一个递减两个单调队列在 R 不断后移时维护 $[L, R]$ 内的最大值和最小值，不过此时我们发现，如果 L 固定，那么 $[L, R]$ 内的最大值只会越来越大，最小值只会越来越小，所以设 $f(R) = \max[L, R] - \min[L, R]$，则 $f(R)$ 是个关于 R 的递增函数，故 $f(R) \ge D \implies f(r) \ge D, R < r \le N$。这说明对于每个固定的 L，向右第一个满足条件的 R 就是最优答案。 所以我们整体求解的过程就是，先固定 L，从前往后移动 R，使用两个单调队列维护 $[L, R]$ 的最值。当找到了第一个满足条件的 R，就更新答案并将 L 也向后移动。随着 L 向后移动，两个单调队列都需及时弹出队头。这样，直到 R 移到最后，每个元素依然是各进出队列一次，保证了 $O(n)$ 的时间复杂度。

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N = 100005;
using ll = long long;
int mxq[N], mnq[N];
int D, ans, n, hx, rx, hn, rn;

struct la {
  int x, y;

  bool operator<(const la &y) const { return x < y.x; }
} a[N];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> D;
  for (int i = 1; i <= n; ++i) cin >> a[i].x >> a[i].y;
  sort(a + 1, a + n + 1);
  hx = hn = 1;
  ans = 2e9;
  int L = 1;
  for (int i = 1; i <= n; ++i) {
    while (hx <= rx && a[mxq[rx]].y < a[i].y) rx--;
    mxq[++rx] = i;
    while (hn <= rn && a[mnq[rn]].y > a[i].y) rn--;
    mnq[++rn] = i;
    while (L <= i && a[mxq[hx]].y - a[mnq[hn]].y >= D) {
      ans = min(ans, a[i].x - a[L].x);
      L++;
      while (hx <= rx && mxq[hx] < L) hx++;
      while (hn <= rn && mnq[hn] < L) hn++;
    }
  }
  if (ans < 2e9)
    cout << ans << '\n';
  else
    cout << "-1\n";
  return 0;
}
```

sundyloveme, untitledunrevised