《完美的队列》命题报告

福建师范大学附属中学 林旭恒

摘 要

本文介绍了作者在此次集训队互测中命制的一道传统数据结构题。

本题题意简单，正解使用到的算法及代码实现都不复杂，却十分考验选手对问题性质的分析能力。本文将介绍其解法和命题背景。

1 试题

1.1 问题描述

有 n 个队列，编号为1到 n 。其中第 i 个队列的容量限制为 a_i ，一开始所有队列均为空。

给出 m 次操作，每次操作用 $l\ r\ x$ 来描述，表示往编号在 $[l,r]$ 之间的队列末尾插入元素 x 。若有队列在插入操作后其中元素个数超过其容量限制，删除该队列最前端的元素。

你需要计算出每次操作结束后所有队列中不同的元素种数。

1.2 输入格式

第一行两个正整数 n,m ，分别表示队列个数和操作个数。

第二行 n 个正整数，第 i 个表示 a_i 。

接下来 m 行，每行三个正整数 $l\ r\ x$ ，其中第 i 行表示第 i 次操作。

1.3 输出格式

输出共 m 行，每行一个非负整数，表示第 i 次操作结束后所有队列中的不同的元素种数。

1.4 数据范围与约定

对于所有数据， $n, m, a_i, x \le 10^5$ ， $l \le r$ 。

共20个测试点，每个测试点5分，其中第 k 个测试点满足 $n, m, a_i, x \le 5000k$ 。

特别地，以下几个测试点满足一些特殊性质：

测试点5： $a_i = 1$ 。

测试点7： $a_i = 2$ 。

测试点9： $a_i = 10$ 。

测试点11： $a_i \le 10$ 。

测试点13,15： $\sum a_i \le 10^6$ 。

对于每个测试点，你需要通过满足该点数据范围及性质的所有数据才能获得该点的分数。

2 朴素算法

2.1 模拟

直接使用数组或者C++提供的std::queue来模拟每个操作。

用一个数组记录每种权值的出现次数，在插入和删除的时候维护这个数组，并在权值的存在性被改变时更新答案。

时间复杂度 $O(nm)$ ，空间复杂度 $O(\sum a_i)$ 。

根据实现的不同，期望得分5~15分。

2.2 初步分析

直接模拟的做法忽视了太多可以用到的性质，现在我们对问题进行初步的分析。

问题并没有要求我们在线处理每次操作，所以我们可以考虑离线处理这个问题。先考虑每个操作对答案的影响，每个操作对答案存在影响的时间显然只在这个操作执行之后，该操作插入的所有元素都被删除之前。我们把一个操作插入的所有元素都被删除的时间称为这个操作的被删除时间。

如果我们能求出每个操作的被删除时间，那么不难计算出我们需要的答案。一种方法是先将每个操作的存在时间抽象成时间轴上的区间，那么对于插入元素相同的操作，这种元素对答案有贡献的时间为这些时间区间的并。我们对每种元素求出区间的并之后利用差分数组即可快速计算答案。另一种方法是将问题转化成维护一个可重集合，原问题的每个操作转化为在执行操作时往集合中加入一个该操作插入的元素，在该操作被删除时删除集合中的一个该元素，那么原问题的答案与这个集合中不同的元素个数相等，用模拟做法中提到的方法不难维护不同的元素个数。第二种方法在实现上会比第一种略微简单一些。

那么接下来我们的问题完全转化为求每个操作的被删除时间。

2.3 基于初步分析的朴素算法

容易想到，一个操作插入的所有元素都被删除的时间就是该操作插入的每个元素被删除的时间（若始终未被删除，视作正无穷）中的最大值。我们考虑分别计算每个元素的出队时间。不难发现，在第 i 个队列中，第 j 次插入的元素会在第 $j+a_i$ 次插入时被删除。我们可以对每个队列分别求出哪些操作往这个队列中加入了元素，然后暴力更新其中每个操作的被删除时间。

时间复杂度 $O(nm)$，空间复杂度 $O(n+m)$。

期望得分20~30分。

3 在线算法

这个部分与正解的思路关联不大，其中的算法以特殊数据的形式作为部分分来考察选手在对正解没有特别清晰的思路时能否通过其他方式来获得更高的分数。以下通过分别介绍各个特殊测试点的解法来介绍此部分的算法。

3.1 测试点5：$a_i = 1$

每个队列的容量限制恰好为1。问题相当于要求维护一个序列，每次操作将序列的一个区间中所有元素修改为 x，并且计算序列中不同的元素种数。

如何维护这个序列呢？传统的维护区间修改操作往往要用到线段树，但是这个问题中要求的不同的元素种数难以在线段树上直接维护。

注意到一个事实：由于我们的修改操作是将整段区间全部修改为 x，我们维护的序列中往往会出现大段的重复元素。我们能否针对这个性质设计一个算法来维护这个序列呢？

答案是肯定的。我们可以将相邻的相同元素合并成一整段，在修改操作时，不难根据之前每一段元素与修改区间的位置关系来维护这些元素段，答案也可以通过维护每种元素出现了多少段来计算。显然，在修改区间均匀随机的情况下，不同的元素段始终不会太多，这个做法的效率十分可观。

然而，在构造数据中，不同的元素段数很容易就能达到 $O(m)$ 的级别（我们不妨假设 m 与 n 同级），暴力维护元素段的时间复杂度将不能接受。

我们可以用数据结构来优化这个维护的过程。首先，只有与修改区间相交的元素段才会被修改，如果我们能快速找到这些元素段，就能进一步优化我们的算法。

不难想到，我们可以用平衡树（或是C++提供的std::set）按序列中的位置顺序来维护这些元素段，那么每次我们可以用 $O(\log m)$ 的时间快速找到一个与修改区间相交的元素段，

并处理修改操作带来的影响。

然而，与一个修改区间相交的元素段的数量仍然可能达到 $O(m)$，似乎这个优化本质上并没有降低我们的时间复杂度。但我们注意到，一个元素段如果被修改区间包含，这个元素段会被直接删除，又由于我们维护的元素段之间互不相交，与一个修改区间相交但不被包含的元素段最多只有2个。那么除了这至多2个元素段，其他与修改区间相交的元素段都被我们删除。而每次操作新增的元素段也至多只有2个（修改区间被某个元素段包含时，元素段被分成两段，又新增一段，相较原来多2段，否则被修改的元素段要么被缩短要么被删除，只新增一段），故所有操作能产生的元素段只有 $O(m)$ 个，那么删除元素段的次数也不会超过 $O(m)$ 次。于是我们证明了这个算法的时间复杂度为 $O(m\log m)$，足以通过该测试点的数据。

实现时，我们只需要维护这些元素段的左端点。为了方便实现，我们可以假设开始时存在一个覆盖整个序列的空元素段，并且只将同一个操作修改出的相同元素合并成一段，显然这样并不影响复杂度。那么在修改时，修改区间的左端点以及右端点的下一个位置必然成为修改后某个元素段的左端点，直接加入平衡树即可。而原来在这两个端点之间的元素段端点可以直接删除。

直接维护左端点的做法省去了许多不必要的特判，思路也更为清晰。而从这个做法的过程来看，每次只加入2个新端点和删除旧端点，总修改次数为 $O(m)$ 的结论也更加明显。

3.2 测试点7：$a_i = 2$

当容量限制增加到2时，问题不再是简单的维护一个序列。但我们仍然可以将其转化成一个序列问题：我们需要维护两个序列，分别表示每个队列的两个元素，每次操作我们要将第一个序列的 $[l, r]$ 区间修改为 x，而每有一个元素被修改，第二个序列上的对应位置就要被替换为这个元素。

仍然考虑使用测试点5的算法，维护第一个序列的元素段，当第一个序列有元素段被修改时，被缩短或是删除的部分就要到第二个序列对应位置上进行区间修改。由于我们在维护第一个序列的元素段时，对元素段的总修改次数为 $O(m)$，那么我们对第二个序列进行的区间修改次数也为 $O(m)$，使用一样的算法即可维护。

这个算法的总时间复杂度同为 $O(m\log m)$，可以通过该测试点。

3.3 测试点9：$a_i = 10$

现在我们考虑容量限制均为 K 的情况。不难想到，我们可以用与容量限制均为2时类似的方法。我们维护 K 个序列，其中第 i 个序列表示每个队列中的第 i 个元素，每一个序列的元素段在被修改的同时对下一个序列进行区间修改。如果一个序列的修改次数为 $O(m)$，那么下一个序列的修改次数也为 $O(m)$。于是我们似乎理所当然地得到了一个时间复杂度为 $O(Km\log m)$ 的算法。

但是注意到一点，在计算维护元素段的复杂度时，我们得出的 $O(m)$ 次修改实际上忽略了其中的常数因子。而这个常数因子很可能会在我们将修改由一个序列转到下一个序列时不断扩大，甚至以指数级增长。例如，是否有可能第一个序列需要 $2m$ 次，第二个需要 $4m$ 次，第三个需要 $8m$ 次，第 K 个需要 $2^K m$ 次呢？

我们不妨换一个角度分析这个算法的复杂度。我们考虑所有元素段端点。事实上，在每次操作中，所有序列中新增的端点只会在操作的左右端点（前后，下略）处出现，也就是最多增加 $O(K)$ 个。但这并不意味着我们的总时间复杂度为 $O(Km\log m)$，因为我们并没有在所有修改时都至少删去一个端点。

那么如何计算复杂度呢？我们可以这么考虑：对于每个位置，定义其深度为其作为元素段端点出现的第一个序列的编号。那么每次操作时，操作的左右端点的深度会被更新为1，而其他所有被我们影响到的端点位置，深度都会被加1，如果超过 K 则被完全删除。也就是说，每次操作加入的所有新端点至多只会被我们处理 $O(K)$ 次，那么这个算法真正的时间复杂度为 $O(K^2 m\log m)$。

实际上，这个算法也可以进一步优化。注意到每次操作我们做的修改实际上等同于先在修改区间的左右端点处插入元素段端点，然后将修改区间中在第 K 个序列的元素段删除，在其他序列的元素段移动到下一个序列。我们可以用支持区间插入删除的平衡树（如splay）来维护元素段端点，这样就可以快速地将元素段移动到下一个序列。

优化后，每个端点只会在被从第 K 个序列中删除时被我们处理一次，每次将各个序列的元素段移动到下一序列的时间复杂度是 $O(K\log m)$，所以总时间复杂度为 $O(Km\log m)$。

3.4 测试点11：$a_i \le 10$

当容量限制不一定相等时，问题变得更加复杂。一种想法是对每种容量限制都用测试点9的优化后的算法分开维护，这样做的时间复杂度为 $O((\max a_i)^2 m\log m)$。

实际上我们有一个更简单的做法。考虑将所有队列一起维护，维护 $\max a_i$ 个序列，此时序列可能因为某个队列的容量不足而被“拦腰截断”，但是如果我们也把对这些序列的修改截断来强行维护，复杂度显然是错误的。

我们换一个思路，把序列中被截断的位置补上，在被补上的位置的元素没有实际意义，只是用来辅助我们维护序列。而这些序列中某个元素段有至少一个元素真实存在于队列中当且仅当这个元素段对应的所有队列中容量限制 a_i 的最大值大等于这个元素段所在序列的编号。预处理之后用RMQ不难 $O(1)$ 判断一个元素段是否真实存在。我们先不考虑用高级平衡树来优化。根据之前的复杂度分析，这个做法的时间复杂度也为 $O((\max a_i)^2 m\log m)$。

如果用支持区间插入删除的平衡树优化，相较测试点9的做法，此时元素段可能在移动到下一个序列的时候就失去对答案的贡献。我们可以对每个仍然真实存在的元素段用RMQ求出其移动到哪一个序列时会失去贡献，然后在平衡树上维护这个值的最小值。那么我们每次移动元素段时不难在平衡树上找到失去贡献的颜色段并处理对答案的影响。

可以证明，优化后的时间复杂度为 $O(\max a_i m\log m)$。

3.5 测试点13,15：$\sum a_i \le 10^6$

在 a_i 的总和有限制时，不容易出现太多容量较大的队列。

我们可以按 a_i 分类， $a_i \le K$ 的队列用之前的方法一起维护， $a_i > K$ 的队列直接模拟。总时间复杂度为 $O(Km\log m + \frac{\sum a_i}{K}m)$ 。选取合适的 K 值，我们得到一个时间复杂度为 $O(m\sqrt{\sum a_i \log m})$ 的做法，配合一些常数优化可以通过该部分数据。

4 二分法

下面介绍此题的一类离线解法，该类解法通过二分或一些其他类似的方法找到每个操作的被删除时间来计算答案，能获得可观的分数。

4.1 判断合法性

之前我们已经提到，如果能求出每个操作的被删除时间，就可以得到答案。现在我们的问题是如何快速求出每个操作的被删除时间。

我们很自然地想到对每一个操作通过二分法来求其被删除时间，那么我们就将问题转化为了一个判定性问题：第 x 个操作插入的元素在第 mid 个操作结束后是否被完全删除？

首先，我们知道，在第 i 个队列中，第 j 次操作插入的元素会在第 $j+a_i$ 次，即其往后第 a_i 次操作时被删除。也就是说，只有第 x 次操作之后的操作才会对它有影响。我们用 b_i 表示第 x 次操作在第 i 个队列插入的元素还需要多少次在 i 号队列的插入才会被删除。那么在第 x 个操作执行时，其影响的区间 $[l,r]$ 中所有 b_i 都会被初始化为 a_i 。而之后的每个操作，都会使这个操作对应的区间中所有 b_i 减1，一旦有 b_i 被减为 0 ，说明第 x 次操作在这个队列里插入的元素被删除了。当所有 b_i 都被减为0时，第 x 次操作插入的所有元素都被删除。

为了方便，我们允许 b_i 小于0，那么每次操作相当于区间减1，当 x 号操作插入的区间 $[l,r]$ 中 b_i 的最大值小等于0 时， x 号操作插入的元素均被删除。

区间减和区间最大值可以用线段树维护。于是我们得到一个判定方法：以 a_i 为初始值建线段树，将时间在 $(x,mid]$ 内的每个操作对应的 $[l,r]$ 区间减1，再判断 x 号操作插入的区间 $[l,r]$ 的最大值是否小等于0。

如果对于每次判定都从头建线段树并执行这些操作来判断，总时间复杂度为 $O(n^2\log^2 n)$（我们认为 n 与 m 同级，不作区分），甚至不如朴素做法。

但我们注意到，如果我们已经得到了时间在 $[l,r]$ 内的每个操作的区间都减1后的线段树，再用 $O(\log n)$ 的时间进行一次区间加1或减1操作就可以得到时间在 $[l-1,r]$ ， $[l+1,r]$ ， $[l,r-1]$ ， $[l,r+1]$ 四者之一的线段树。

显然，我们可以利用这个性质设计一些算法来优化时间复杂度。

4.2 整体二分

我们设计一个类似分治的算法。用 $solve(S,l,r)$ 表示我们在时间 $[l,r]$ 中寻找操作集合 S 中每个操作的被删除时间。我们先取 $[l,r]$ 的中点 mid，然后依次判断 S 中的每个操作在第 mid 个操作后是否被完全删除，并依此将 S 分成两部分，分别递归到 $solve(S_1,l,mid)$ 和 $solve(S_2,mid+1,r)$。

为了优化我们的判断时间，我们只维护一棵线段树，并记录其对应的时间区间（即对哪些操作进行了区间减1）的左右端点，在需要查询时，用每次 $O(\log n)$ 时间的代价将左端点或右端点往相邻位置移一步，一直移动到我们需要的位置。那么我们的复杂度就只跟我们的左右端点进行了多少次移动有关。特别地，第一次查询相当进行了 $O(n)$ 次端点的移动，以下分析中略。

先考虑右端点，我们可以这么考虑：在 $solve(S,l,r)$ 时，右端点在 mid，调用 $solve(S_1,l,mid)$ 时，我们将右端点从 mid 移动到 $[l,mid]$ 的中点，结束调用时移动回来；调用 $solve(S_2,mid+1,r)$ 时，我们将右端点从 mid 移动到 $[mid+1,r]$ 的中点，同样在结束调用时移动回来。事实上我们不难发现，右端点的实际移动次数还要更少一些，但这么分析已经足够了。这样右端点的移动次数与 $O(r-l)$ 同级，故右端点的总移动次数不会超过 $O(n\log n)$。

然而，对于左端点，我们无法保证其总移动次数在可以接受的范围内。由于所有操作的被删除时间分布并没有特别的性质，每个 $solve$ 函数的调用中，左端点的移动次数都可能达到 $O(n)$ 次，极端情况下，左端点的总移动次数可以达到 $O(n^2)$ 次，难以接受。

我们注意到，左端点的移动只会在 S 集合中编号最小到编号最大的操作的范围内进行。我们可以将操作按照编号分成 K 个块，每个块调用一遍 $solve(S_i,1,m)$，其中 S_i 表示这个块内所有操作构成的集合，那么我们每次判断时进行的左端点移动都不会超过 $O(\frac{n}{K})$ 次。而每个操作都会被递归并判断 $O(\log n)$ 次，故左端点的总移动次数不会超过 $O(\frac{n^2\log n}{K})$ 次。又因为我们总共调用了 K 次 $solve(S_i,1,m)$，右端点的总移动次数为 $O(Kn\log n)$。再算上线段树的复杂度，这个做法的总时间复杂度为 $O((\frac{n^2}{K}+Kn)\log^2 n)$。

K 取 $O(\sqrt{n})$ 时，时间复杂度为 $O(n\sqrt{n}\log^2 n)$。由于实际中这个算法的时间复杂度是不满的，配合一些优化可以得到相当可观的分数。

4.3 类莫队算法

我们回到直接二分的做法。

注意到我们进行左右端点移动的操作恰好与莫队算法类似。常规的莫队算法是将询问按左端点分块，每一块按右端点的顺序依次处理，左端点只在块内移动，右端点不断向右移动，以此保证复杂度。而我们可以设计一个类似的算法来满足我们二分时判断合法性的需求。

我们按判断时需要查询的时间区间（即之前提到的 $(x,mid]$）的右端点分块，然后为每个块建一棵线段树，并记录其对应的左右端点。查询时我们找到右端点所在的块，将对

应的线段树左右端点移动到查询的位置上。如果我们按时间顺序依次对每个操作进行二分，不难发现，所有线段树的左端点只会向右移动，如果共 K 个块，左端点的总移动次数就是 $O(Kn)$ 的。而右端点只在块内移动，又因为我们二分时总共要进行 $O(n\log n)$ 次判断，故右端点的总移动次数为 $O(\frac{n^2\log n}{K})$ 。算上线段树的复杂度，总时间复杂度为 $O((\frac{n^2\log n}{K}+Kn)\log n)$ 。

K 取 $O(\sqrt{n\log n})$ 时，时间复杂度为 $O(n\sqrt{n}\log^{1.5}n)$ 。

4.4 类莫队算法的改进

在之前的算法中，我们将所有线段树左端点不断向右移的过程中实际上得到了以每个操作为左端点，右端点在每个块内的信息，而二分的做法并没有完全利用到这些信息，我们不妨改进一下这个做法。

我们在时间轴上均匀设置 K 个关键点，相当于之前的块。然后分别以每个关键点为右端点，计算每个左端点的信息。在计算的同时，我们判断每个操作到每个关键点时是否被删除，以此确定每个操作的被删除时间在哪两个关键点之间。这部分的时间复杂度为 $O(Kn\log n)$ 。

最后，我们对每对相邻关键点，将被删除时间在其之间的操作按顺序通过依次判断每个时间点的方式找到被删除时间。这样对于每对相邻关键点，左端点不断向右移，总移动次数为 $O(Kn)$ 。对于每个操作，其在对应的相邻关键点之间暴力判断所有右端点，故右端点的总移动次数为 $O(\frac{n^2}{K})$ 。算上线段树的复杂度，总时间复杂度为 $O((\frac{n^2}{K}+Kn)\log n)$ 。

K 取 $O(\sqrt{n})$ 时，这个算法的时间复杂度为 $O(n\sqrt{n}\log n)$ 。

5 序列分块

此部分为本题的标准解法，相较之前的算法更好地利用了本题的性质。

5.1 单调性

我们注意到一个显然的事实：在每个队列中，每个元素的被删除时间是随插入时间单调递增的。可是对于所有操作，它们的被删除时间却并不以操作时间单调递增。其原因是不同的操作插入的元素可能在不同的队列中，而不同的队列之间是没有关联的。

我们从中产生一个想法：如果有若干个操作插入的队列区间完全相同，它们之间的被删除时间是否按时间顺序单调递增呢？

答案是显然的。在每个队列中，元素的被删除时间随时间递增，那么插入的队列完全相同的操作，它们的被删除时间，也就是插入的每个元素被删除时间的最大值，自然也随时间递增。

我们可以设计这样一个算法来计算插入队列完全相同的操作的被删除时间：先用与上一部分相同的方式，对 a_i 建线段树，然后以第一个要求的操作为左端点，右端点不断向右推，直到我们要求的操作的插入区间中最大值小等于0，我们就求出了第一个要求的操作的被删除时间。接下来我们按时间顺序求每一个要求的操作的被删除时间，由于它们的被删除时间单调递增，每次我们只需要将左端点移到对应位置，右端点继续往右推即可。因为我们的左右端点都是不断往右推的，总移动次数为 $O(n)$，所以求一次插入队列为某一区间的所有操作的被删除时间的时间复杂度为 $O(n\log n)$。

可是两个不同的操作插入的区间很可能并不相同，如何才能利用这个性质呢？对于每个操作，我们只关心其插入的每个元素被删除时间中的最大值，我们可以将每个操作通过将插入区间划分成若干个子区间的方式分成若干个子操作，然后分别求出这些子操作的被删除时间，再取其中的最大值即可。

我们想到可以将所有队列按编号分块，那么如果每一块的大小设为 K，每个操作就可以被分成 $O(\frac{n}{K})$ 个整块操作和 $O(1)$ 个非整块操作。由于总共只有 $O(\frac{n}{K})$ 个不同的块，我们可以对每个块用前面提到的算法求出所有这个块的整块操作的被删除时间，这部分的总时间复杂度为 $O(\frac{n^2\log n}{K})$。

而对于非整块操作，总共只有 $O(n)$ 个，也就是最多只插入 $O(nK)$ 个元素，我们考虑直接求出其中每个元素的被删除时间。由于第 i 个队列中第 j 个插入的元素会在第 $j+a_i$ 次插入时被删除，我们可以先求出一个操作在第 i 个队列中是第几个操作，再求出其在第 i 个队列中往后第 a_i 个操作是哪个操作。

我们可以按队列的编号顺序依次求出每个队列对应的操作集合，对于每个操作，在其插入区间的左端点处加入集合，右端点处删出集合。要支持查询信息，只需要按操作时间维护一个树状数组，支持前缀求和以及在树状数组上二分即可。总时间复杂度为 $O(nK\log n)$。

故 K 取 $O(\sqrt{n})$ 时，我们得到了一个 $O(n\sqrt{n}\log n)$ 的做法。

5.2 对整块操作的优化

在对每个块计算这个块的整块操作的被删除时间时，我们注意到，实际用到的 a_i 只有块内的部分。

我们可以只对块内的 a_i 建线段树，所有操作对块外的影响直接无视。此时我们发现，绝大多数的区间加1和减1操作，都是直接对整块进行的。事实上，在分块时，每个操作对每个块的影响也被分成了 $O(\frac{n}{K})$ 个整块和 $O(1)$ 个不完整块。也就是说，我们总共要进行 $O(\frac{n^2}{K})$ 次整块加减和 $O(n)$ 次区间加减。直接打标记就可以 $O(1)$ 完成一次整块加减，时间复杂度降为 $O(\frac{n^2}{K}+n\log n)$。事实上，我们可以直接用数组模拟代替线段树来完成区间加减和维护最值，整块加减仍然使用标记，复杂度虽然变为 $O(\frac{n^2}{K}+nK)$，但是更易实现并且不会影响总复杂度。

对非整块操作，我们用与之前相同的做法。适当选取 K 值，时间复杂度降为 $O(n\sqrt{n\log n})$，可以通过全部数据。

5.3 对非整块操作的优化

在之前通过维护树状数组来计算非整块操作的被删除时间的做法中，并没有利用到同一队列中元素被删除时间单调递增的性质。我们能否对非整块操作也设计一个与整块类似的算法呢？

我们可以将每个块分开考虑。对于一个块，我们将这个块的整块操作集合称为 A，这个块内的非整块操作集合称为 B。我们需要对 B 中每个操作求出其在该块中插入的每个元素的被删除时间。对于块中某个队列 i，我们将 B 中对这个队列有影响的操作集合称为 B_i。对于 B_i 中的每个操作 x，我们求出 B_i 中最早的操作 y，满足在 y 执行前，x 在 i 中插入的元素已被删除。根据同一队列中元素被删除时间单调递增的性质，y 显然也单调不降，那么我们仍然可以在 B_i 中用左右端点不断向右推的方法求出，问题是如何判断 x 插入的元素是否已被删除，即计算两个操作之间有多少个对 i 有影响的操作。由于对 i 有影响的只有 B_i 和 A 中的操作，我们可以对每个块预处理出 B 中每个操作的前一个 A 中操作在 A 中是第几个，即可 $O(1)$ 计算。在求出 y 后，通过一些简单的计算，我们不难得出操作 x 在 i 中插入的元素的实际被删除时间是 y 往前第几个对 i 有影响的操作，通过预处理的信息可以 $O(1)$ 求出这个操作。那么除了预处理的时间，我们只用 $O(|B_i|)$ 的时间就能求出 B_i 中所有操作在 i 中的被删除时间。

不难证明，用这个算法优化后，总时间复杂度为 $O(\frac{n^2}{K}+nK)$。K 取 $O(\sqrt{n})$ 时，我们得到了一个时间复杂度为 $O(n\sqrt{n})$ 的优秀算法。

6 命题背景和得分情况

本题最初的原型是这样一道题目：要求维护一个序列，支持区间修改为 x 和查询区间中不同权值种数。原题利用了本文在线算法部分提到的维护元素段的方法，再将每个元素分别以其下标和前一个相同权值的下标抽象成二维平面上的点，将询问转化为矩形求和解决。在偶然中，我产生了通过计算每个修改的被删除时间来得出答案的想法，并专门为这个想法设计了这道题目。在此之后，我不断改进自己的算法，最终将此题放在集训队互测与大家分享。

近来，随着广大OI选手对数据结构的研究不断深入，许多命题人常用复杂的模型和高级的数据结构及算法来提升题目的难度，并形成了许多常见的“套路”。这不仅使得做数据结构题往往需要掌握许多难度较大的算法和大量零碎的知识点，还需要具备很强的代码能力，与竞赛锻炼人思维的初衷相违背。

相较之下，本题的标准解法仅使用到了分块以及two pointer（指左右端点不断向右推的

做法）的知识点。最终标算的一个实现中，只用到了最简单的数组，没有用到包括线段树在内的任何较高级的数据结构，且代码长度仅1.4kb。

在此基础上，本题仍然具有一定的思维难度和区分度。在命题时，我预测候选队中会有1~3人通过这道题目，一半左右的人能通过各种做法拿到较高的分数。而在互测中，候选队中无人通过该题，3人拿到了60以上的分数，略微少于预期，说明大家对这种类型的题目还不够擅长。此外，在LOJ的同步赛中，来自北京大学的吉如一学长现场用 $O(n\sqrt{n})$ 的做法通过了该题。

7 总结

本题考察了选手对问题性质的分析，以及针对性质设计算法的能力。特殊的部分分设计方式能根据选手不同的算法与实现，让选手获得不同的分数。而要拿到高分，不仅需要对性质的分析，还需要对分块算法和基础数据结构的充分理解与应用。

希望本题能对大家有所启发，加强对基础数据结构的灵活运用。

致谢

感谢中国计算机学会提供交流和学习的平台。

感谢张瑞喆教练的指导。

感谢周成老师对我的指导和帮助。

感谢黄哲威学长在信息竞赛路上对我的启发与帮助。

感谢所有在赛前赛后与我交流讨论本题的同学们，其中特别感谢吉如一学长分享他的做法。

感谢胡学浚同学和陈喆桓同学为本文审稿。

感谢各位抽出宝贵时间阅读这篇文章。

感谢所有帮助过我的人。

参考文献

[1] 徐明宽，《非常规大小分块算法初探》，2017年国家集训队论文。