Imprecise Permutation Sort

Идея: Михаил Дворкин
Разработка: Михаил Дворкин

В этой задаче, хочется надеяться, есть несколько разных подходов, позволяющих уложиться в ограничение в 300 000 запросов, основанных на разных алгоритмах сортировки с надстройками для учёта странного компаратора. Опишем решение основанное на быстрой сортировке Хоара с элементами сортировки вставками, которое укладывается в ограничение «с запасом».

Но сперва исследуем, насколько вообще разрешима задача — можно ли отличить предложенными неточными сравнениями любое число из отрезка $[1, n]$ от другого. Несложная алгебра показывает, что с помощью какого-то меньшего числа b можно отличить числа a и $a + 1$ всегда кроме случая, когда a заканчиввается на 99. Говоря «можно отличить» мы имеем в виду, что $b =_{imprecise} a$ и при этом $b <_{imprecise} a + 1$. А пары неотличимые благодаря меньшим числам, к счастью, можно отличить благодаря какому-то большему числу c, то есть $c >_{imprecise} a$ и при этом $c =_{imprecise} a + 1$. Единственная фундаментальная проблема — если это большее число c не влезает в отрезок $[1, n]$.

Так при $n = 16\,384$ невозможно отличить числа 16 299 и 16 300. А при $n = 16\,300$ невозможно отличить не только их, но и 16 199 от 16 200. Впрочем, при любом n из предлоенного диапазона таких пар будет не более чем две, поэтому можно решить задачу, расставив эти пары как угодно, а затем за 2^2 операций обмена, перебирая все возможные расположения этих пар, добиться отсортированности массива.

Теперь заметим, что запросы на обмен местами (swap) тратят имеющийся ресурс в 300 000 запросов, поэтому минимизируем их количество. Для этого решим задачу сортировки индексов, не делая ни единого обмена, получим полное понимание (кроме одной—двух неотличимых пар), какой элемент где находится, и расположим их на свои места простым линейным алгоритмом за 16 383 обменов или меньше — например, будем жадно ставить любой элемент, лежащий не на своём месте, на своё место, пока такой вообще есть.

Затем заметим, что жюри может подготовить какой-нибудь неприятный для нашего решения тест. Чтобы лишить их такой возможности, сгенерируем произвольную перестановку π размера n и будем в запросах на сравнение каждый раз вместо пары (i, j) спрашивать о паре $(\pi(i), \pi(j))$. Тогда можно считать, что жюри даёт нам работать со случайной перестановкой и в ней нет никаких заготовленных «спецэффектов».

Эти идеи стоит применить в любом решении. Теперь опишем подход QuickSort+InsertionSort.

Для начала применим общую идею быстрой сортировки Хоара (принцип «разделяй и властвуй»), но получим как результат не отсортированный массив, увы, а набор групп элементов, таких что любой элемент более левой группы меньше любого элемента более правой.

Выберем произвольный элемент-разделитель (благодаря идее перестановки π можно выбрать, например, первый). Разделим за линейное время все элементы на меньшие, «равные» и большие его согласно неточному компаратору. Меньшие элементы обработаем этим же алгоритмом рекурсивно, большие — тоже, и положим полученные группы в ответ в естественном порядке: полученные рекурсивно группы с меньшими числами, группа чисел «равных» разделителю, и полученные рекурсивно группы с большими числами.

Кстати, первые 98 групп будут точно состоять из одного числа каждая, что помогает думать над этим решением, а также отлаживать его.

Теперь применим идею сортировки вставками. Инвариант будет такой: числа из первых нескольких групп уже идеально отсортированы кроме, возможно, одной или двух пар чисел вида $100k - 1$ и $100k$ (то есть неотличимых меньшими числами пар), которые могут быть перепутаны между собой. Причём речь идёт только о старших одной-двух парах, то есть тех, для которых различающее их большее число просто ещё не было добавлено в отсортированную область.

Научимся одну следующую группу вставлять в отсортированную область. Для этого к каждому элементу новой группы применим двоичный поиск, ищущий границу между числами меньшими его

и числами «равными» ему. Переформулируем искомую величину так: сколько чисел в отсортированной области «равны» данному числу. И если найденная двоичным поиском граница оказалась вблизи неразличимых пар, то аккуратно уточним искомую величину, сделав ещё несколько сравнений.

Теперь все числа добавляемой группы разбились на две подгруппы: те, у которых хотя бы одно число в отсортированной области оказалось «равно» ему, и те, у которых таковых не нашлось. Первую группу уже можно добавлять к отсортированной области, а именно в порядке уменьшения количества чисел «равных» данному. А со второй группой следует повторить описанную процедуру ещё раз после добавления первой группы, ведь теперь отсортированная область немного выросла, и данных стало больше.

Кроме того, если после добавления очередной группы, то есть после нарастания отсортированной области, какую-то пару вида $100k - 1$ и $100k$ стало возможно отличить (добавилось различающее их большее число), лучше отличить их сейчас.

Итого, после добавления последней группы мы имеем отсортированный массив кроме одной-двух неразличимых пар. Как было сказано выше, теперь за не более чем $n - 1$ обмен расположим числа в массиве a в соответствии с нашим пониманием порядка и за не более чем 2^2 обмена перебором найдём расположение чисел в неразличимых парах.

Добавим также, что как и в «обычных» задачах о сортировке, QuickSort работает лучше, если элемент-разделитель выбирать не просто случайно, а как медиану из случайной выборки некоторого нечётного размера. В данной задаче это действительно помогает уменьшить число сравнений. А искать медиану небольшой нечётной выборки можно либо отсортировав эти числа по критерию «скольких чисел в этой выборке данное число меньше», либо ещё эффективнее: добавив числа из выборки в сбалансированное дерево поиска, используя неточный компаратор, но заменяя ответ «равно» на случайный ответ меньше/больше.

Также, в зависимости от реализации, может помочь уменьшить количество сравнений запоминание полученных ответов от компаратора: если про пару (i, j) уже спрашивали, то незачем неидеально написанному коду повторно спрашивать про (i, j) или про (j, i).

Программа с описанными здесь идеями делает на 1000 тестах от 272 185 до 287 010 запросов, в среднем 276 530 со стандартным отклонением 1425. Если вызывает опасения «хвост» этого распределения, то можно делать большего размера нечётные выборки для поиска медианного элемента-разделителя в быстрой сортировке, это немного стабильно увеличит число запросов, но уменьшит вероятность серии неудачных разделителей и превышения ограничения в 300 000 запросов.