Разбор задачи «Порядок реставрации»

Упорядочим числа, соответствующие позициям в списке реставрации, по неубыванию (выполним их сортировку) и будем сопоставлять полученную последовательность чисел с исходной. Если на какой-то позиции в обоих списках оказались одинаковые числа, то можно считать, что упорядочение было выполнено таким образом, чтобы это число осталось на своей позиции. Это как раз и позволит найти максимально возможное улучшение настроения министра.

Чтобы решить подзадачу 1, достаточно было воспользоваться алгоритмом сортировки с асимптотикой $O(n^2)$ (например, хорошо известной пузырьковой сортировкой). Подзадача 2 требовала использования более эффективного алгоритма сортировки с асимптотикой $O(n \cdot logn)$; также можно было применить сортировку подсчетом.

Остановимся на решении с помощью сортировки подсчетом чуть подробнее. Поскольку известно, что все числа, встречающиеся во входных данных, не меньше 1 и не больше n, создадим вспомогательный массив $counts[]$. Элемент $counts[i]$ этого массива будет равен количеству чисел, равных i, во входных данных. Заполнить массив $counts[]$ легко —для этого достаточно один раз пройти по массиву $r[]$, каждый раз увеличивая на единицу элемент $counts[r[j]]$.

Теперь, когда массив $counts[]$ заполнен, можно получить отсортированный массив следующим образом. Пойдем по массиву $counts[]$ и запишем в новый массив $sorted[]$ элемент i $counts[i]$ раз подряд. Запись в новый массив будем вести слева направо, в итоге получим упорядоченные по неубыванию числа. После этого останется посчитать количество одинаковых чисел на одинаковых позициях в массивах $r[]$ и $sorted[]$.

Разбор задачи «Звукопоглощение»

В данной задаче требуется найти любой отрезок длины k такой, что:

- минимальное число, не лежащее на этом отрезке, принимает наибольшее значение (например, среди чисел 1, 4, 2, 6 таким числом будет 3);
- он не выходит за границы массива (если вы получали WA 1.8 и 1.12, то выводили слишком большое значение в качестве стартовой позиции отрезка, из-за чего конец отрезка выходил за границу массива).

Также гарантируется, что каждое число в массиве встречается ровно 1 раз — т.е. у вас перестановка длины n.

Для решения подзадачи #1 можно было написать любое решение со сложностью $O(n^2 \cdot k)$ или быстрее. К примеру, можно было зафиксировать старт отрезка ($n - k + 1$ возможных вариантов), для каждого отрезка вы однозначно знаете его конец; теперь вы перебираете числа в порядке от 1 до n (n возможных итераций) и для каждого числа находите, лежит ли он в зафиксированном отрезке (проверить надо k позиций).

Для прохождения тестов подзадачи #2 требовалось заметить, что вам нет смысла искать позицию каждого числа циклом. Используя факт, что каждый элемент в массиве уникален и принадлежит отрезку $1 \dots n$, можно завести массив pos, и для каждого значения элемента сохранить его индекс в массиве. Это делается одним проходом по массиву: $pos[a[i]] = i$.

После этого ваша проверка вхождения элемента в фиксированный отрезок будет использовать константное число операций: $start \leqslant pos[value] < start + k$ (помните, что среди реальных языков программирования только Python поддерживает двойные неравенства). Это решение за $O(n \cdot k)$.

Третья группа тестов отличалась от первых двух ограничениями на 2 порядка выше, что должно было намекнуть на существование быстрого решения с асимптотикой $O(k)$. Пусть мы нашли оптимальный отрезок и минимальное число не в отрезке равно x, а начинается отрезок с позиции s. В таком случае для всех чисел v от 1 до $x-1$ выполнено условие $s \leqslant pos[v] < s+k$, а для x верно $pos[x] < s$ или $s + k \leqslant pos[x]$.

Можно задаться вопросом — а что нам помешало немного подвинуть старт s так, чтобы и число x вошло в отрезок? Ведь в таком случае мы бы увеличили ответ минимум на 1. Единственной возможной причиной может являться то, что при сдвиге мы бы потеряли из нашего отрезка какое-то число $y < x$, которое в нашем оптимальном ответе на нем находилось.

Давайте рассмотрим задачу с другой стороны. Обозначим через $left$ величину $min(pos[1], pos[2], \ldots, pos[x-1])$, а через $right$, соответственно, величину $max(pos[1], pos[2], \ldots, pos[x-1])$. В таком случае условием существования какого-либо отрезка длины k, включающего все эти числа, будет $right - left < k$. В самом деле, если такое неравенство выполняется, то мы можем выбрать в качестве величины $s = min(n-k+1, left)$ (в 1-индексации). Для такого числа s будет выполняться $s \leqslant left \leqslant right < s + k \leqslant n$ — то, что нам надо!

Итоговым решением для подзадачи #3 будет следующий алгоритм: выполним предпросчет всех позиций значений в массив pos (смотрите решение подзадачи #2). После этого будем пытаться «добавить» в ответ по очереди числа от 1 до n, обновляя величины $left$ и $right$. Если в какой-то момент после добавления $right - left \geqslant k$, то последнее добавленное число не может находиться на отрезке длины k с одним из ранее добавленных, а это значит, что и дальше не получится расширяться. В этот момент надо отменить последнее добавление и завершить поиск ответа.

Псевдокод выглядит следующим образом:

```
left = n + 1, right = -1
v = 0
while v < n {
nextL = min(left, pos[v + 1])
nextR = max(right, pos[v + 1])
if nextR - nextL < k: v = v + 1
else: break
}
```

Разбор задачи «Плохая погода»

Ограничения в первой подзадаче позволяют просто перебрать все возможные варианты и выбрать наилучший из них.

Решить эту задачу на полный балл можно бинарным поиском по ответу. Будем выбирать некоторое количество бригад и проверять, получится ли успеть выполнить реставрацию в отведенные сроки.

Проверка выполняется аккуратной симуляцией процесса, в которой может помочь, например, такая структура данных, как очередь с приоритетами (priority_queue в C++, PriorityQueue в Java, в Python потребуется использовать несколько функций из модуля heapq). Такая структура умеет возвращать минимальное содержащееся в ней значение.

Выбрав некое количество бригад k для проверки, следует поместить в очередь длительности реставрации первых k зданий. Затем будем извлекать наименьший элемент из очереди, добавлять его значение к длительности реставрации первого еще не отреставрированного здания и помещать результат в очередь.

Когда неотреставрированных зданий не останется, извлечем из очереди все оставшиеся элементы. Самый большой из них и будет сроком завершения реставрации. Если этот срок не превосходит требуемого, то такое количество бригад может быть ответом — разумеется, если проверку не пройдет меньшее количество.

Бинарный поиск в границах от 1 бригады до n бригад выполняется стандартным образом. Однако доказательство того, что бинарный поиск можно применять в этой задаче, не вполне тривиально. Нам необходимо убедиться, что увеличение числа бригад приводит к уменьшению времени реставрации.

Пусть мы выполнили подсчет для k бригад, и знаем, что время начала реставрации здания $\#i$ было равным $t_k[i]$. Наше предположение состоит в том, что в случае $k+1$ бригады будет выполняться $t_k[i] \geqslant t_{k+1}[i]$.

Для всех i от 1 до $k+1$ это выполняется автоматически (первые k и в том, и в другом случае имеют время начала 0, в случае k бригад реставрация $k+1$-го здания начнется по окончании наименее продолжительной реставрации среди первых k зданий).

Предположим, что наше предположение выполняется вплоть до какого-то значения i, и попробуем осуществить индуктивный переход к $i+1$.

По мере формирования очереди с приоритетами мы помещали в нее сроки завершения реставрации всех i зданий. В каждый момент там находится k элементов (мы сейчас не обсуждаем окончание процесса, когда реставрация всех зданий начата), и нам нужно получить наименьший из этих элементов. Иными словами, если бы мы рассмотрели $t_k[i]$ как массив из i элементов и упорядочили его по невозрастанию, то нам потребовался бы k-й от начала (т.е. от наибольшего) элемент.

Такой элемент называют k-й порядковой статистикой массива. Мы можем записать это так: $t_k[i+1] = kth_k(t_k[1..i])$, полагая, что kth_k — это функция, возвращающая k-й по величине, если считать от наибольшего, элемент. Аналогично можно записать для $k+1$ бригад: $t_{k+1}[i+1] = kth_{k+1}(t_{k+1}[1..i])$.

Для этой функции верно следующее:

- если для любого i верно, что $a[i] \geqslant b[i]$, то $kth_k(a) \geqslant kth_k(b)$;
- для любого a верно, что $kth_k(a) \geqslant kth_{k+1}(a)$.

Поскольку $t_k[1..i] \geqslant t_{k+1}[1..i]$ по индуктивному предположению, то

$$t_k[i+1] = kth_k(t_k[1..i]) \geqslant kth_{k+1}(t_k[1..i]) \geqslant kth_{k+1}(t_{k+1}[1..i])\,.$$

Разбор задачи «Цветовой баланс»

Подзадача 1 имеет очень маленькие ограничения, что позволяет выполнить полный перебор вариантов.

На полный балл задача решается методом динамического программирования.

Опишем состояние $dp[i][j][k]$, где i — номер рассматриваемого изображения, j — значение разности между количеством изображений, подготовленных первым художником, и количеством изображений, подготовленных вторым художником, а k — номер художника, который будет готовить изображение i. Значением, хранящимся в этом элементе, будет как раз абсолютная величина максимального расхождения цветовых схем, достигнутая к этому моменту. Технически также удобно сохранять путь, которым мы приходим в это состояние, а именно — значения j' и k' того элемента, из которого мы осуществляем переход.

Размерности j и k будут маленькими. Действительно, художников у нас всего двое, а возможных разностей между количеством изображений, подготовленных одним художником, и количеством изображений, подготовленных другим художником, будет 5 (-2, -1, 0, 1, 2).

Динамику удобно строить вперед, генерируя из каждого состояния с некоторым j два возможных новых с $j+1$ и $j-1$ и отсекая те, которые запрещены условием. Если одно и то же состояние может получиться из разных предшественников, нужно выбрать то, в котором значение, хранящееся в элементе, будет меньше.

В последнем поколении состояний нужно будет выбрать состояние с наименьшим расхождением, а затем восстановить путь, по которому мы в него пришли.