Испытание силомера

Автор задачи и разработчик: Рябчун Владимир

В **первой подзадаче** для решения было достаточно реализовать ответ на каждый запрос за линейное время. Сложность решения будет $\mathcal{O}(qn)$.

Для решения **второй подзадачи** необходимо было заметить, что $\texttt{popcount}(2^i) = 1$, поэтому запрос третьего типа на самом деле ничем не отличается от запроса второго типа, где необходимо присвоить 1 на отрезке. Таким образом в этой подзадаче требовалось написать дерево отрезков с присвоением на отрезке и суммой. Асимптотика решения будет $\mathcal{O}(q \log n)$

В **третьей подзадаче** напрямую такой подход применить было невозможно. Однако можно было поддерживать в вершинах дерева отрезков информацию о количестве нулевых элементов на отрезке. Тогда, зная длину отрезка и количество нулей на нем, можно легко пересчитывать суммы после применения операции `popcount`. Сложность решения такая же — $\mathcal{O}(q \log n)$.

Для решения **четвертой подзадачи** заметим, что числа 1 и 0 не меняются от операции `popcount`. При этом легко видеть, что $\texttt{popcount}(x) \leqslant \log_2 x$. Таким образом, можно применить не более $\mathcal{O}(\log^* x)$ операций, пока x не станет равным единице. Это позволяет решить задачу таким образом: будем поддерживать множество позиций в массиве, где находятся числа, большие 1. Тогда для ответа на запрос достаточно найти в этом множестве первый элемент, больший или равный l, и применить операцию над настоящим массивом до нужным индексам. Любое число будет рассмотрено в этой реализации не более $\log^* n$ раз, после чего оно станет равным 1. Для ответа на запросы суммы можно поддерживать дерево отрезков или дерево Фенвика, поскольку запросы изменения здесь будут в точке.

В **пятой подзадаче** нужно было добавлять в это множество один элемент, если его значение становилось больше единицы. Асимптотика решения в обоих случаях была $\mathcal{O}(q \log n \log^* n)$.

В **шестой подзадаче** надо было воспользоваться тем, что значения элементы массива не очень большие. Можно было хранить в каждой вершине дерева отрезков количество каждого числа. Тогда пересчет при применении каждой операции очевиден — перенести количество из каждой ячейки массива в другую. Сложность времени решения равна $\mathcal{O}(q \log n \log 20)$

Седьмая подзадача подразумевала решение при помощи не очень быстрого дерева отрезков или корневой декомпозиции. Нужно в каждом блоке поддерживать сумму, отложенное присвоение, количество отложенных `popcount` и количество чисел с каждым количеством бит. Все остальное делается как в обычной корневой декомпозиции. Для крайних блоков, которые попали в запрос не полностью, применяются все отложенные операции. Если нужно сделать отложенное присвоение, то оно затирает все другие отложенные операции. Если нужно сделать отложенный `popcount` и в блоке уже есть отложенное присвоение, то достаточно его изменить, иначе просто увеличить счетчик запросов третьего типа. Поддерживать сумму при применении `popcount` к блоку можно благодаря поддерживаемой информации, сколько на блоке чисел каждой битности. Итоговая асимптотика будет $\mathcal{O}(q\sqrt{n} \log 10^9)$.

В **восьмой подзадаче** нужно было сначала выполнить запросы модификации, а потом отвечать на запросы суммы. Когда все запросы изменения выполнены, достаточно посчитать префиксные суммы на массиве и отвечать на запросы первого типа за $\mathcal{O}(1)$. Для обработки запросов модификации можно было применить scanline, поддерживая `ordered_set` актуальных запросов обоих типов по времени, или два дерева отрезков.

Во втором способе необходимо поддерживать одно дерево отрезков с количеством применений `popcount` к элементу массива, и второе — с последним значением, присвоенном элементу. При запросе второго типа необходимо сделать присвоение на отрезке во втором дереве, а в первом на том же отрезке записать 0. При запросе третьего типа нужно увеличить все числа на отрезке в первом дереве на 1. В каждый момент времени мы знаем для точки, какое было ее последнее присвоение и сколько раз после этого был применён `popcount`. Эта часть решения работает за $\mathcal{O}(q \log n)$

После обработки всех изменений надо обойти дерево и применить все операции. Это можно выполнить за $\mathcal{O}(n \log^* n)$. Итоговая асимптотика будет $\mathcal{O}(q \log n + n \log^* n)$.

Полное решение задачи было теоретически возможно при помощи аккуратно написанного дерева отрезков из шестой подзадачи, но авторское решение использовало другие идеи. Будем поддерживать в дереве отрезков сумму, минимум и максимум в поддереве. Зная эту информацию, легко определить, состоит ли отрезок из одинаковых чисел. Будем выполнять запрос третьего типа таким образом: в дереве отрезков будем спускаться до тех пор, пока не окажемся в вершине, где максимум не больше единицы или пока минимум не равен максимуму. В первом случае это означает, что отрезок просто не поменяется, а во втором — что отрезок состоит из равных чисел, поэтому можно просто сделать соответствующее присвоение на этом отрезке. Запрос второго типа реализуется как в стандартном дереве отрезков.

Оценим время работы данного алгоритма при помощи метода потенциалов. Мысленно у каждого непрерывного отрезка равных элементов число $\log^* n$. Потенциал $\Phi = (\text{количество непрерывных отрезков равных чисел}) \cdot \log n \cdot (\sum \text{числа в уме})$.

Пусть в операции третьего типа мы посетили x отрезков, тогда реальное время работы будет $x \log n$. Будем при каждом заходе в непрерывный отрезок вычитать из числа, которое держим в уме, 1. Ясно, что это можно сделать не более $\log^* n$ раз, после чего отрезок будет состоять только из единиц и в него мы больше не будем заходить. Тогда изменение потенциала $\Delta\Phi = -x \log n$ и амортизированное время работы данного запроса $\mathcal{O}(1)$.

Запрос второго типа может увеличить количество непрерывных отрезков не более, чем на 1. Запишем в этом большов отрезке в уме число $\log^* n$. Реальное время работы будет $\mathcal{O}(\log n)$, а изменение потенциала $\Delta\Phi \leqslant \log n \log^* n$. Амортизированное время работы будет соответственно $\mathcal{O}(\log n \log^* n)$.

Запрос суммы не влияет на потенциал и тратит $\mathcal{O}(\log n)$ времени.

Итоговое время работы будет $\mathcal{O}(q \log n \log^* n)$ и потребление памяти будет $\mathcal{O}(n)$.