Тинькофф A'. Корневая оптимизация.

kik0s, isaf27

7 сентября 2019

Основное равенство, которое нам понадобится:
$$n\sqrt{n} = \frac{n^2}{\sqrt{n}}$$

1 Корневая как структура данных

1.1 Basics

Если разбить массив на блоки длной $\sqrt{n}$, Таких блоков будет не более, чем $\sqrt{n}$. Тогда любой подотрезок массива представим из $\sqrt{n}$ блоков и $\sqrt{n}$ отдельных элементов по краям отрезка. Посчитав для блока некие величины (за $O(\sqrt{n})$ для каждого блока, получим возможность отвечать на запросы на массиве за $O(\sqrt{n})$. Для запросов обновления надо будет пересчитывать блок за $O(\sqrt{n})$.

1.2 Массовые обновления

Аналогично другим структурам данных, в корневой можно делать массоввые обновления. Для этого достаточно поддерживать push-величины, с помощью которых обновлять элементы внутри блока, когда понадобится обратиться к конкретному элементу.

1.3 Split-rebuild

Пусть мы не хотим отвечать на запросы для единичных элементов, а хотим всегда работать с блоками. Тогда откажемся от константного ограничения на размер блока, и если границы запроса попадают в середину блока, то разобьем его на две штуки. За каждый запрос мы создадим не более двух новых блоков. Раз в $\sqrt{n}$ запросов будем заново строить нашу структуру данных, как будто заново решаем задачу. Если оставить ограничение $\sqrt{n}$ на максимальный размер блока, то количество блоков все еще $O(\sqrt{n})$, а их размеры $O(\sqrt{n})$. Количество суммарных перестраиваний $O(\sqrt{n})$, каждое из которых работает за $O(rebuild)$.

1.4 Split-merge

иногда бывает так, что rebuild стоит $O(n \log n)$, потому что нам понадобится пересортировать блоки. Вспомним, что мы умеем сливать два отсортированных блока за линейное время. Тогда будем поддерживать блоки так, чтобы у нас не существовало блоков с размером больше $2\sqrt{n}$ (с такими мы сделаем split за линейное время), и пар соседних блоков размером меньше $\frac{\sqrt{n}}{2}$ каждый (с такими мы сделаем merge за линейное время). Теперь у нас add/erase/update работают за $\sqrt{n}$, а get — за $O(\sqrt{n} \log n)$.

1.5 Играем с константами

Если переписать все приведенные выше вычисления, обозначив размер блока за K, то получим, что в случае, когда у нас **один** из двух типов операций работает за $O(K)$, а другой — за $O(\frac{n \log n}{K})$, то оптимально будет сделать $K = O(\sqrt{n \log n})$, а не $O(\sqrt{n})$. Таким образом, мы добьемся времени работы $O(n\sqrt{n \log n})$.

2 Корневая в задачах на строки

Данные идеи очень полезны в задачах, где есть ограничение на суммарный размер строк. Обозначим это ограничение за $\sum |S|$.

2.1 Разбиение на тяжелые и легкие строки

Назовем строку S *длинной*, если $|S| \geq \sqrt{\sum |S|}$. Все оставшиеся строки назовем *легкими*.
Утверждение. Существует не более $2 \cdot \sqrt{\sum |S|}$ тяжелых строк.
Доказательство. Пусть существует более $2 \cdot \sqrt{\sum |S|}$ тяжелых строк. Тогда их суммарная длина больше, чем $\frac{2 \cdot \sqrt{\sum |S|} \cdot \sqrt{\sum |S|}}{2} = \sum |S|$, чего не может быть.

2.2 Количество разных длин

Утверждение. Количество различных длин строк не более, чем $O(\sqrt{\sum |S|})$. **Доказательство.** Пусть оно равно x. Тогда минимальная возможная сумма — это сумма чисел от 1 до x. $\sum_1^x = \frac{x(x+1)}{2}$. Так как это число должно быть меньше, чем $\sum |S|$, то $x = O(\sqrt{\sum |S|})$

3 Корневая на графах

3.1 Тяжелые и легкие вершины

Назовем *тяжелой* вершину, имеющую более $\sqrt{E}$ соседей. Все оставшиеся вершины назовем *легкими*.
Утверждение. В графе не более $2 \cdot \sqrt{E}$ тяжелых вершин.
Доказательство. Пусть в графе более $2 \cdot \sqrt{E}$ тяжелых вершин. Тогда число ребер в графе больше, чем $\frac{2 \cdot \sqrt{E} \cdot \sqrt{E}}{2} = E$, чего не может быть.

3.2 Нахождение количества треугольников в графе за $O(E\sqrt{E})$

Мысленно разобьем все вершины графа на легкие и тяжелые. Заметим, что треугольников, образованных только тяжелыми вершинами, всего $O(E\sqrt{E})$, как $C^3_{\sqrt{E}}$. Теперь рассмотрим треугольники, которые содержат в себе легкие вершины. В таком треугольнике точно будут два ребра, инцидентных легкой вершине. Сколько таких пар может быть? Всего таких ребер $O(E)$, при этом для каждого ребра парными могут быть только $O(\sqrt{E})$ ребер, в силу степени легкой вершины. Таким образом, *число треугольников в графе равно* $O(E\sqrt{E})$.
Каким алгоритмом их искать? Можно явно провести процесс, описанный выше, но это не самое приятное в реализации решение этой задачи. Можно отсортировать вершины по возрастанию степеней, после чего поставить на всех соседей пометки, после чего запустить поиск путей длины 2 и фиксировать треугольник при нахождении пометки.
Аналогичным способом можно показать, что время работы такого алгоритма тоже будет равно $O(E\sqrt{E})$.

4 Алгоритм Мо

4.1 Алгоритм Мо

Пусть есть задача на массиве, которую мы решаем без запросов обновления, в offline. Запросы могут быть любой степени ужасности, если мы умеем на них отвечать структурой данных, которая хранит информацию об отрезке $[l, r]$ и поддерживает следующие операции:

1. add_left
2. add_right
3. del_left
4. del_right
5. query

, которые меняют границы отрезка и внутреннюю структуру данных, а также позволяют сделать запрос. Все запросы мы пересортируем. Теперь они будут отсортированы по паре $(\lfloor\sqrt{l}\rfloor, r)$. Отвечать на запрос мы будем таким образом — с помощщью реализованных операций подстроим необходимые нам границы отрезка, после чего вызовем query.
Оценим время выполнения. Заметим, что левые границы разбиваются по блокам, а внутри запросов одного блока правые границы отсортированы. При переходе от запроса к запросу внутри блока, левая граница делает $O(\sqrt{n})$ сдвигов. Правая же граница для запросов одного блока суммарно сделает $O(n)$ сдвигов. Так как число блоков $O(\sqrt{n})$, то и суммарное время работы $O(\sqrt{n})$.

4.2 Алгоритм Мо на дереве (эйлеровом обходе)

В случае, если в задаче надо считать операции на дереве, то можно выписать эйлеров обход дерева (одну вершину выписываем дважды — в моменты вхожа и выхода). Теперь задача сводится к задаче на массиве с одним исключением — мы должны добавлять в структуру элементы, которые содержатся на отрезке **ровно** 1 раз.

4.3 Алгоритм Мо + корневая как структура для Мо

Часто бывает так, что в качестве структуры данных для Мо хочется использовать дерево отрезков, делая $O(\log n)$ операций и на обновление, и на запрос. А можно в качестве структуры данных использовать корневую, тогда мы будем делать $O(1)$ на обновление и $O(\sqrt{n})$ на запрос. Такой вариант может оказаться быстрее, тк в Мо количество обновлений $O(n\sqrt{n})$, а количество запросов $O(n)$.

4.4 3D Мо

Оказывается, если очень захотеть, Мо можно применять и в задачах, где надо обрабатывать запросы обновления. Введем третий параметр get-запроса — t, который будет равен числу update-запросов до текущего get. Снова пересортируем get-запросы, но в этот раз в порядке $(\frac{t}{n^{\frac{2}{3}}}, \frac{l}{n^{\frac{2}{3}}}, r)$. После чего оставим такой же алгоритм Мо, как и раньше, только в трех измерениях. Время работы нового алгоритма будет $O(n^{\frac{5}{3}})$, что доказывается аналогично обычному Мо.

5 Корневая декомпозиция по запросам

5.1 Rebuild

Некоторые запросы обновления можно легко обрабатывать группой, сделав scanline, но тяжело обрабатывать по одному. В таких случаях можно откладывать их выполнение, и запоминать их в какой-нибудь список. При размере списка $O(\sqrt{n})$ можно обработать все запросы этого списка группой. При выполнении get-запросов достаточно смотреть на все запросы внутри списка и смотреть, как они влияют на ответ.

6 Решение задачи о рюкзаке за $O(S\sqrt{S})$

Если у нас есть n предметов с весами $w_1, w_2, \ldots, w_n$, таких что $w_1 + w_2 + \ldots + w_n = S$, то мы можем решить задачу о рюкзаке за время $O(Sn)$ стандартной динамикой. Как решать быстрее? Попытаемся сделать так, чтобы n было $O(\sqrt{S})$. Заметим, что количество различных весов среди $w_1, w_2, \ldots, w_n$ будет $O(\sqrt{S})$, потому что если среди них k различных чисел, то $S = w_1 + w_2 + \ldots + w_n \geq 1 + 2 + \ldots + k = \frac{k(k+1)}{2}$, значит $k \leq \sqrt{2S}$.
Рассмотрим некоторый вес x, который встречается в наборе весов. Обозначим за c его количество вхождений. Рассмотрим такое максимальное натуральное t, что $2^t - 1 \leq c$. Тогда в наборе весов c вхождений веса x заменим на веса $x, 2x, \ldots, 2^{t-1}x, (c + 1 - 2^t)x$. Легко видеть, что все суммы $0, x, \ldots, cx$ можно по-прежнему набрать и только их.
Уже сейчас легко видеть, что новое количество весов будет $O(\sqrt{S}\log S)$, потому что для каждого веса мы оставили $\leq \log S$ весов, а различных весов $O(\sqrt{S})$. Для нужной нам оценки посмотрим, для каких x мы могли сделать замену на $\geq p$ весов (при $p \geq 2$)? Мы всегда добавляем $t + 1$ число, то есть $p - 1 \leq t$, то есть $2^{p-1} \leq 2^t \leq \ +1$. Значит количество вхождений каждого из таких x будет $\geq 2^{p-1} - 1$, значит их сумма $\leq \frac{S}{2^{p-1}-1} \leq \frac{4S}{2^p}$, значит количество $\leq \sqrt{8S}\frac{1}{\sqrt{2^p}}$ (потому что они все различны, как мы помним). Значит в сумме мы добавим $\leq \sqrt{8S}(\sum\limits_{p=2}^{\log S} \frac{1}{\sqrt{2^p}})$ чисел, что является $O(\sqrt{S})$.

Bonus: если далее делать рюкзак с помощью *bitset*, то получаем решение за $O(\frac{S\sqrt{S}}{64})$.

7 Корневая для задачи dynamic records.

В случае задачи dynamic records можно разбить массив на блоки размером $\sqrt{n}$ элементов в каждом блоке сохранить список рекордов для **этого** блока. Для update-запросов надо перестраивать блок. Для ответа на get-запрос надо идти по блокам слева направо, поддерживая текущий префиксный максимум. Для каждого нового блока мы должны найти бинпоиском в списке рекордов первый рекорд больше префиксного максимума — это актуальный рекорд. После чего мы прибавим к ответу количество элементов на этом суффиксе и обновим префиксный максимум. С учетом подбора константы решение работает за $O(n\sqrt{n\log n})$.