PROJECT 1000

Anton Layk, Cesh4, Mellstroy, hUeSoSiK

January 2026

Содержание

1 Введение

TODO

2 Xor Basis

2.1 Мотивация

Как и с любым алгоритмом/техникой, для начала надо разобраться, какую задачу мы хотим научиться решать.

Задача 2.1. *Дан массив целых чисел a размера n,* $1 \le a_i < 2^{30}$*. Надо отвечать на запросы вида: Дано число x, проверить, существует ли такой подмассив, что xor его элементов == x?*

Для дальнейшей простоты введем некоторые определения:

Определение 2.1. $\mathbb{Z}_2^d$ — множество векторов в d-мерном пространстве, где $x_1, x_2, \ldots, x_d \in \{0, 1\}$.

Определение 2.2. Векторное пространство — просто множество векторов.

Определение 2.3. Множество векторов называется базисом векторного пространства, если каждый элемент этого пространства может быть **единственным** образом представлен линейной комбинацией базиса.

Определение 2.4. Множество векторов называется независимым, если никакой из них не может быть представлен линейной комбинацией оставшихся.

2.2 Алгоритм

Заметим, что числа представляются как вектора $\mathbb{Z}_2^d$*. Для этого надо взять их битовое представление. Тогда xor — это сложение координат по модулю 2.*

Будем поддерживать базис векторов, которые уже добавлены. Пусть мы хотим добавить вектор $\vec{v}$.

1. Если $\vec{v}$ выражается через текущий базис — ничего не делаем.
2. Иначе надо добавить $\vec{v}$ в базис.

Теперь остается только научиться проверять, можно ли составить $\vec{v}$ текущим базисом?

Определение 2.5. Пусть $f(\vec{v})$ — максимальная позиция бита единицы в бинарном представлении вектора.

Будем хранить базис так, что все значения f различны.

1. Если $f(\vec{v}) > f(\vec{b_1})$, то $\vec{v}$ нельзя выразить.
2. Если $f(\vec{v}) = f(\vec{b_1})$, заменяем $\vec{v} := \vec{v} - \vec{b_1}$ и продолжаем.

Итоговая асимптотика:

$$O(n \cdot d) = O(n \log A)$$

2.3 Реализация

```
void insertVector(int mask) {
    for (int i = LOG_A - 1; i >= 0; i--) {
        if ((mask & (1 << i)) == 0) continue;

        if (!basis[i]) {
            basis[i] = mask;
            return;
        }

        mask ^= basis[i];
    }
}
```

2.4 Заключение

Ссылки для практики:

1. Codeforces div2 C
2. Codeforces div2 F
3. Educational Codeforces G

3 Persistent Centroid Decomposition

3.1 Мотивация

Эта статья является дополнением к базовой версии Centroid Decomposition. Поэтому, если вы не знаете базовую версию, рекомендую для начала ознакомиться с ней.

Для начала, давайте поставим задачу, которую мы хотим научиться решать.

Задача 3.1. *Дано дерево из n вершин, перестановка целых чисел* $a_1, a_2, ...a_n$ *длины* n $(1 \le n \le 10^5)$ *и* q $(1 \le q \le 10^5)$ *запросов вида:*

- l r v - *посчитать* $\sum_{i=l}^{r} dist(a_i, v)$

На запросы необходимо отвечать в ***online***

3.2 Идея решения

Каждый запрос вида $(l,\ r,\ v)$ можно разбить на два запроса: $(1,\ r,\ v)$ - $(1,\ l-1,\ v)$. Поэтому для решения задачи достаточно научиться отвечать на запрос *на префиксе*. Чтобы отвечать на такой запрос, можно рассматривать его как сумму расстояний до всех активированных вершин, если активированы все вершины на префиксе.

Мы уже можем решать эту задачу в **offline**, используя ***Центроидную декомпозицию***. Для решения задачи в **online**, центроидное дерево необходимо сделать **персистентным**.

3.3 Алгоритм

У каждой вершины центроидного дерева **не более** $\boldsymbol{log\ n}$ **предков**, следовательно, при активации какой то вершины, можно просто *подниматься по ее родителям и копировать их*, после чего уже обновить значения в них.

Проблема здесь заключается в том, что наше дерево не бинарное (у вершины может быть больше 2 детей). Из-за этого мы не сможем быстро копировать вершины. Давайте исправим это, **добавив дополнительные** ***фиктивные*** **вершины в дерево**.

3.4 Бинаризация дерева

Определение 3.1. Будем считать вершину *бинарной*, если у нее не более 3 соседей.

Определение 3.2. Будем считать дерево *бинарным*, если все его вершины *бинарные*

Рассмотрим вершину v исходного дерева, у которой более 3 соседей. Пусть $b_1, b_2, .., b_k$ - список детей этой вершины. Для того, чтобы сделать v бинарной, необходимо добавить новую вершину u в дерево, добавить ребра $v - u$, $u - b_2$, $u - b_3$, .., $u - b_k$ и удалить ребра $v - b_2$, $v - b_3$, .. $v - b_k$. Проделаем эту операцию рекурсивно для всех v. Можно показать, что суммарно в дерево добавится не более n новых вершин.

На получившемся бинарном дереве ***запустим центроидную декомпозицию***. Теперь, когда у каждой вершины не более 3 детей, *их можно копировать за* $O(1)$.

3.5 Заключение

Проблема с бинаризацией дерева решена, следовательно, теперь мы *можем за* $O(log\ n)$ *скопировать предков активируемой вершины и обновить в них значения.* После чего отвечать на запросы, обращаясь к нужным версиям **центроидного** дерева.

Задачи для практики:

1. Div (1 + 2) G

4 1D1D Optimization

4.1 Мотивация

Определение 4.1. $w(j, i)$ - функция, удовлетворяющая условию четырехугольника: $w(a, c) + w(b, d) \leq w(a, d) + w(b, c)$, где $a \leq b \leq c \leq d$.

Хотим научиться считать динамику вида: $dp_i = \min\limits_{j \leq i} dp_j + w(j, i)$ за асимптотику быстрее, чем $O(n^2)$. Будем считать, что $dp_0 = 0$.
Конкретную задачу разберем позже.

4.2 Идея решения

Определение 4.2. $k(j, i) = \arg\min\limits_{x \leq j} dp_x + w(x, i)$. $(j < i)$

Другими словами, $k(j, i)$ - точка оптимума, если мы прорелексировали dp_i через j первых значений.

Заметим, что если мы насчитали, $k(i-1, i)$, то мы можем легко узнать значение $dp_i = dp_{k(i-1,i)} + w_{k(i-1,i)}$ Также заметим, что если мы знаем $k(j, i)$, то можно насчитать $k(j+1, i)$:

$$k(j+1, i) = \begin{cases} \text{j + 1}, & \text{если } j+1 \text{ - точка разреза} \\ \text{k(j, i)}, & \text{иначе} \end{cases}$$

То есть, чтобы обновить значение, надо проверить, что:
$$\boldsymbol{dp_{j+1} + w(j+1, i) < dp_{k(j,i)} + w(k(j, i), i)}$$

Теорема 4.1. *$k_j(i) = k(j, i)$ - неубывающая функция для $i = j, j+1, ..n$.*

Доказательство 4.1. Пойдем от обратного. Пусть есть пары индексов (j_1, i_1) и (j_2, i_2) такие, что $j_1 = k_j(i_1)$ и $j_2 = k_j(i_2)$, причем $j_2 \leq j_1 \leq j < i_1 < i_2$. Тогда выполняются неравентсва

1. $dp_{j_1} + w(j_1, i_1) < dp_{j_2} + w(j_2, i_1)$
2. $dp_{j_2} + w(j_2, i_2) < dp_{j_1} + w(j_1, i_2)$
3. $w(j_2, i_1) + w(j_1, i_2) \leq w(j_2, i_2) + w(j_1, i_1)$

Заметим, что $dp_{j_2} + w(j_2, i_2) \overset{(3)}{\geq} dp_{j_2} + w(j_2, i_1) + w(j_1, i_2) - w(j_1, i_1) \overset{(1)}{>} dp_{j_1} + w(j_1, i_1) + w(j_1, i_2) - w(j_1, i_1) = dp_{j_1} + w(j_1, i_2)$. **Получили противоречие с неравенством (2)**.

4.3 Алгоритм

Теперь перейдем к реализации. Для текущего j будем поддерживать значения $k_j(i)$ в виде дека отрезков разных значений (надо хранить начало отрезка и значение k_j на нем). **Тогда чтобы посчитать значение dp_{j+1} надо взять $k(j, j+1)$ из первого отрезка дека**. Далее надо обновиться на значения $k_{j+1}(i)$. **Удалим первый элемент и обновим дек соответствующе**. **Теперь надо найти суффикс на котором $k_{j+1}(i) = j + 1$. Это можно сделать бинпоиском.** Для некоторых задач мы можем найти начало отрезка без бинпоиска. Надо уметь считать функцию $D(j, i)$.

Определение 4.3. $D(j, i)$ - минимальное значение x такое, что $dp_j + w(j, x) \geq dp_i + w(i, x)$

Каждое значение отрезка добавляется и удаляется максимум один раз, значит асимптотика $O(n)$, если мы можем считать $D(j, i)$ и $O(n \log n)$ иначе.

4.4 Пример

Рассмотрим следующую задачу. Она сводится к подсчету динамики вида $dp_i = \min\limits_{j<i} dp_j + a_i * b_j$. Где массив a отсортирован по возрастанию, а b по убыванию. Пусть $w(j, i) = b_j * a_i$. Тогда для функции выполняется неравенство четырехугольника и мы можем решить задачу 1D1D оптимизацией. Реализация за $O(n \log n)$. Так же отмечу, что эту задачу можно решить за $O(n)$, однако это остается упражнением читателю.

5 Queue Undo (Euphoria Trick)

5.1 Мотивация

Мы знаем такую структуру данных как СНМ с откатами. Она умеет добавлять ребра в конец и откатывать последние добавления. В каком то смысле это работает по принципу stack'а (добавить в конец и удалить из конца). Однако, что если нам надо откатывать первые добавления (как в queue)? Алгоритм, описанный далее позволит сделать это. Так же сразу определимся с задачей, которую хотим решить.

Задача 5.1. *Дан список из m ребер. Посчитайте количество подотрезков* $[l, r]$ $(1 \le l \le r \le m)$ *таких, что граф на n вершинах, содержащий только ребра из этого подотрезка, является связаным.*

5.2 Идея решения

Давайте подумаем, как это вообще можно делать. Будем хранить снм с откатами и порядок, в котором в нем объединялись компоненты. Пусть теперь нам пришел запрос объединения двух компонент, назовем тип этого добавления A. До первой операции *pop* (откат начала) все хорошо, будем просто добавлять ребра в снм и сохранять порядок. Теперь рассмотрим, что делать при первом откате. Давайте воспользуемся классической идеей: **откатим все наши запросы и добавим их в обратном порядке, но теперь поменяем их тип на B**. После этого мы можем просто откатить последнее изменение B. Но теперь, при добавлении нового запроса, у нас будет тип A в конце, в то время, как откатить надо последний B. Назовем это проблемой stack'а.

5.3 Решение проблемы stack'а

Итак, теперь у нас есть порядок в котором добавлены элементы в наш СНМ. Вновь рассмотрим запрос *pop*, однако теперь в массиве порядка добавлений могут быть как и A, так и B. **Найдем первый суффикс, в котором количество B будет больше, чем A, либо возьмем всю строку, если такого суффикса нет. Откатим весь этот суффикс и добавим эти ребра в другом, более выгодном для нас порядке.** Какой же порядок будет для нас лучше всех? Очевидно, что сначала должны идти все A из этого суффикса, а затем B (так как теперь мы вновь можем легко откатывать). В случае, когда остались одни A, просто провернем трюк с разворотом всего массива из начала (и изменения типов соответственно). **Утверждается, что асимптотика данного решения амартизированно $O(n \log n)$.**

Доказательство 5.1. Для оценки асимптотики на самом деле нам лишь надо оценить $\sum_{i=1}^{2n} o_i$, где o_i - длинна суффикса, который мы переупорядочивали на шаге i, если это был шаг переупорядочивания и 0 ина-

че. Заметим, что если $o_i \geq k$, $o_j \geq k$ и $j \geq i$, то $j - i \geq k$ (1). Тогда: $\sum_{i=1}^{2n} o_i = \sum_{i=1}^{2n} \sum_{j=1}^{o_i} 1 = \sum_{j=0}^{\infty} \sum_{i=1}^{2n} [o_i \geq j] \overset{(1)}{\leq} \sum_{i=1}^{2n} \frac{2n}{i} = O(n \log n)$

5.4 Решение задачи и заключение

Теперь, когда мы научились делать откаты по принципу очереди, можно вернуться к задаче из начала. Давайте для каждого i насчитаем rgt_i - минимальную правую границу такую, что граф на $[i, rgt_i]$ - связанный. Очевидно, что этот массив будет возрастающий. Значит воспользуемся методом двух указателей: пока граф несвязанный, двигаем правую границу и добавляем соответствующее ребро, иначе левую и делаем откат. Теперь ответ на задачу это просто $\sum_{i=1}^{n} n - rgt_i$.
Реализацию можно найти тут.

6 Heavy-Light Decomposition $O(log\ n)$

6.1 Мотивация

Эта статья является дополнением к базовой версии Heavy-Light Deomposition. Поэтому, если вы не знаете базовую версию, рекомендую для начала ознакомиться с ней.

Давайте поставим задачу, которую хотим научиться решать.

Задача 6.1. *Дано дерево из n вершин и массив целых чисел* $a_1,\ a_2,\ ..,\ a_n$ $(1 \le a_i \le 10^9)$*, где* a_v *- число, записанное на вершине* v*. Поступают запросы двух типов:*

1. $v\ u$ *- Найти максимальное число на кратчайшем пути от* v *до* u*.*
2. $v\ x$ *- Присвоить* a_v *значение* x*.*

Хочется научиться отвечать на запросы за $\boldsymbol{O(\log\ n)}$.

6.2 Алгоритм

Проблема базовой версии заключается в том, что тяжелые пути мы обрабатываем за $\boldsymbol{O(\log\ n)}$. Здесь $\boldsymbol{\log\ n}$ появляется из-за обращения к структуре (например дереву отрезков). В этом случае дерево отрезков никак не пользуется размерами поддеревьев вершин. Тогда давайте модифицируем его.

Далее будет идти речь о каком-то тяжелом пути размера $\boldsymbol{k}$. Обозначим за $\boldsymbol{s_1,\ s_2,\ ..,\ s_k}$ размеры поддеревьев вершин без учета тяжелых сыновей. То есть $\boldsymbol{s_v}$ - размер поддерева вершины $\boldsymbol{v}$ минус размер поддерева ее *тяжелого* сына. Пусть $\boldsymbol{S = \sum_{i=1}^{k} s_i}$, то есть сумма всех $\boldsymbol{s_i}$. Обычно, в дереве отрезков, мы делим длину текущего отрезка пополам. Теперь, давайте делить отрезок пополам относительно $\boldsymbol{S}$. Для отрезка $\boldsymbol{1,\ 2,\ ..,\ k}$ его серединой будет такое минимальное $\boldsymbol{j}$, что $\boldsymbol{\sum_{i=1}^{j} s_i \ge \frac{S}{2}}$. Иными словами, мы хотим, чтобы сумма $\boldsymbol{s_i}$ в левой части была близка к $\boldsymbol{\frac{S}{2}}$ (тогдя для правой части будет выполнятся то же самое). Тогда высота нашего ДО будет $\boldsymbol{\log\ S \le \log\ n}$ (так как $\boldsymbol{S \le n}$).

Заметим, что запрос на *вертикальном пути* (то есть от вершины $\boldsymbol{v}$ до вершины $\boldsymbol{u}$ из поддерева $\boldsymbol{v}$) разбивается на *запросы к одному отрезку и префиксам тяжелых путей*. Этот один запрос на отрезке тяжелого пути мы можем обработать отдельно за $\boldsymbol{O(\log\ S)}$. Теперь в случае с префиксами, давайте просто делать спуск по нашему ДО. Тогда **суммарно всех операций спуска будет не более** $\boldsymbol{O(\log S)}$.

Доказательство 6.1. Почему же так? Давайте рассмотрим переход с одного тяжелого пути на другой. Пусть S_1 и S_2 - суммы s_i на первом и втором тяжелых путях соответственно. Тогда при спуске по ДО на первом пути, S_1 *будет уменьшаться в 2 раза*, и когда мы спустимся до листа i, значение s_i сохранится для следующего тяжелого пути. Иными словами: $s_i \geq S_2$. Поэтому суммарно спуск сделает $O(\log\ S)$ операций.

6.3 Заключение

Теперь мы научились отвечать на запрос суммарно за $O(\log\ n)$. К сожалению, в реализации придется строить дерево отрезков на каждом тяжелом пути отдельно, из-за чего на практике это работает не сильно быстрее базового $O(\log^2 n)$ *(но все равно быстрее)*.

6.4 Реализация

С реализацией можете ознакомиться по ссылке.

7 Xor Segment Tree

7.1 Мотивация

Давайте поставим задачу, похожую на ту, что мы уже умеем решать деревом отрезков.

Задача 7.1. *Дан массив* $a_0, a_1, .., a_{2^n-1}$ *размера* 2^n $(0 \leq n \leq 17)$. *Поступают* q *запрсы двух типов:*

1. $l\ r\ k\ x$ - *Заменить элементы* $a_{l \oplus k}, a_{(l+1) \oplus k}, .., a_{r \oplus k}$ *на число* x.
2. $l\ r\ k$ - *Вывести сумму элементов* $a_{l \oplus k}, a_{(l+1) \oplus k}, .., a_{r \oplus k}$.

$\oplus$ - *операция побитового исключающего ИЛИ (XOR)*

7.2 Алгоритм

Далее будет очень важно, что размер массива является ***степенью двойки****, если это не так, дополните его фиктивными элементами.*

Пусть нам дан запрос с числом k. Давайте посмотрим на битовую запись этого числа. Теперь, можно заметить, что бит i в этом представлении будет соответствовать уровню $i + 1$ дерева отрезков (если счить снизу). Это значит, что если мы сейчас находимся в какой то вершине v на уровне $i+1$ дерева отрезков, то ее левый сын отвечает за отрезок, где бит i выключен, а правый за тот, где включен. Получается, что если на позиции i битовой записи числа k стоит 1, то на уровне $i + 1$ ДО необходимо поменять для каждой вершины левого и правого сыновей местами.

7.3 Заключение

Получается, что единственное изменение в базовом ДО будет то, что на уровне $i + 1$ мы смотрим на бит i в битовой записи числа k из запроса, если он равен 1, то надо поменять сыновей местами, и затем запускаться как обычно, а если бит 0, то ничего менять не надо, и просто запускаемся как в обычном ДО. С реализацией можете ознакомиться по ссылке.

8 Linear Memory Binary Lifting

9