PROJECT 1000

Anton Layk, Cesh4, Mellstroy, ChickenBurger

January 2026

Содержание

1 Введение

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

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

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 $(0 \le n \le 17)$. *Поступают* q *запросы 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 считать 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

8.1 Мотивация

Давайте поставим какую-то базовую задачу на *бинарные подъемы в дереве*. Например, найти LCA двух вершин.

Задача 8.1. *Дано дерево из* n $(1 \le n \le 10^5)$ *вершин и* q $(1 \le q \le 10^5)$ *запросов вида:*

- v u *- найти LCA (наименьший общий предок)* v *и* u*.*

Хочется научиться решать это за $O(n + q \log n)$ *и* $O(n)$ *памяти.*

8.2 Алгоритм

В обычной версии мы для каждой вершины храним $\log n$ прыжков из нее. Тогда давайте хранить только 2 значения: прыжок в предка и какой-то еще. Пусть p_v и $junp_v$ - это родитель вершины v и прыжок из нее соответсвенно. Надо понять, как считать $jump_v$, чтобы подниматься до любого предка за $O(\log n)$.

Допустим, сейчас мы хотим посчитать $jump_v$ для какой-то вершины v. Посмотрим на p_v, $jump_{p_v}$ и $jump_{jump_{p_v}}$ (ее предка, прыжок из предка и прыжок из прыжка предка). Тогда если расстояние между p_v и $jump_{p_v}$ равно расстоянию между $jump_{p_v}$ и $jump_{jump_{p_v}}$, то $jump_v = jump_{jump_{p_v}}$, а иначе $jump_v = par_v$. Если вы запутались, то словами это можно описать так: *если длины двух предыдущих прыжков из* v *равны, то приравняем* $jump_v$ *самой высокой вершине этих прыжков, а иначе приравняем родителю* v*.*

Чтобы с помощью таких прыжков узнать $\mathrm{LCA}(v,\ u)$, посмотрим на прыжок из v, если $jump_v$ является предком u, то $v = p_v$, а иначе $v = jump_v$. Утверждается, что это будет работать за $O(\log n)$. С реализацией можете ознакомиться тут.

8.3 Доказательство

Далее будет удобно рассматривать дерево не полностью, а как ***бамбук****, потому что прыжки зависят только от глубины вершины, но не от структуры дерева.*

Для начала давайте заметим, что длина любого прыжка (по вершинам) является **степенью двойки**. Это легко понять, потому что прыжок является либо комбинацией двух других, длина которых также степень двойки, либо идет в предка, в этом случае его длина 2.

При подъеме, описанном в алгоритме, длина прыжков сначала будет увеличиваться, а потом постепенно уменьшаться (когда прыжок станет слишком

длинным). Давайте рассмотрим *две части* этого пути (длина прыжков увеличивается, длина прыжков уменьшается).

Теорема 8.1. *Не существует пересекающихся прыжков (но есть вложенные).*

Доказательство 8.1. Докажем по индукции. Пусть для предков v это верно, тогда докажем для v. Рассмотрим, когда $jump_v$ является комбинацией двух других прыжков, а так как для них уже известно, что они не пересекаются, то и новый прыжок точно не будет пересекаться. В случае, когда прыжок из v ведет в родителя, пересечений и так не будет.

Теорема 8.2. *Не может быть трех и более равных по длине прыжков подряд.*

Доказательство 8.2. Пусть прыжки из v, $jump_v$ и $jump_{jump_v}$ равны по длине, тогда прыжок из сына $jump_v$ будет вести в $jump_{jump_{jump_v}}$ и пересекаться с прыжком из v, что противоречит предыдущей теореме.

Теорема 8.3. *Прыжок из v по длине не больше, чем прыжок из $jump_v$ (исключение: $jump_v$ - корень).*

Доказательство 8.3. Докажем по индукции. Пусть для всех предко v это верно, докажем для v. Если прыжок из v ведет в родителя, то он и так самый маленький по размеру. А если прыжок ведет не в родителя, значит он состоит из двух прыжков в два раза меньших по длине. Так как утверждение верно для предков, то длина прыжка из $jump_v$ будет не меньше, чем длина прыжка из v деленная на 2. Но ровно в два раза меньше она не может быть из предыдущей теоремы, следовательно она не меньше, чем просто длины прыжка из v.

Теперь, что мы знаем:

1. Длины прыжков являются степенью двойки.
2. Длины прыжков не убывают.
3. Не существует трех и более одинаковых по длине прыжков подряд.

Из этого можно сделать вывод: на возрастающей части пути будет сделано не более $2 \log n$ прыжков.

В случае с убывающей частью, давайте посмотрим на прыжок из текущей вершины, если он слишком длинный, значит нужная нам вершина находится между v и $jump_v$. Тогда при переходе в родителя v длина прыжка уменьшается в 2 раза, следовательно получится своего рода *спуск*. Всего в таком спуске будет сделано не более $2 \log n$ прыжков.

Получается, что суммарно подъемы на обоих путях работают за $O(\log n)$.

8.4 Задачи

Задача 8.2. *Минимум на пути. Дано дерево из n вершин, в котором на каждом ребре записано число, и q запросов вида:*

- *v u - найти минимум на пути между v и u.*

Асимптотика: $O(n + q \log n)$.

Решение 8.1. Давайте помимо массива $jump$ хранить массив min, где min_v - минимум на пути от v до $jump_v$. Считать его можно аналогично массиву $jump$: если расстояния между двумя предыдущими прыжками равны, то $min_v = \min(min_{p_v},\ min_{jump_{p_v}},\ a_v)$, иначе $min_v = \min(a_{p_v},\ a_v)$. Теперь можно отвечать на запросы, поднимаясь от вершины до $LCA(v,\ u)$. Реализация: тут.

Задача 8.3. *К-й предок. Дано дерево из n вершин, массив $a_1,\ a_2,\ ..,\ a_n$ и q запросов вида:*

- *v k - найти k-ого родителя вершины v.*

Асимптотика: $O(n + q \log n)$.

Решение 8.2. В этой задаче можно подниматься аналогично задаче про LCA. Если прыжок из вершины v длинее k, то $v = p_v$, иначе $v = jump_v$. Вот реализация.

9 Kruskal Reconstruction Tree

9.1 Введение

Предположим, что нам дан взвешенный неориентированный граф с n вершинами и m ребрами. Пусть ребра будут отсортированы в порядке возрастания веса. Тогда эта структура данных умеет:

1. Находить ребро максимального веса на пути от v до u.
2. Поддерживать информацию о вершинах, достижимых из v используя первые k ребер.

9.2 Построение Kruskal Tree

Дерево краскала это подвешенное дерево, имеющее $n + m$ вершин. Изначально у нас будет n листьев, отвечающих за наши вершины. Обрабатываем ребра в необходимом порядке. Пусть сейчас мы хотим соединить v и u. Тогда создадим новую вершину, значение которой равно весу ребра, которое сейчас обрабатываем. И соединим с корнями деревьев в которых сейчас лежат v и u. Это можно реализовать с помощью СНМ. Для большей понятности можно ознакомиться с базовой реализацией.

9.3 Разбор задач

Структура простая, однако позволяет делать очень много интересных вещей. Давайте разберем некоторые задачи на нее.

Задача 9.1. *Дан неориентированный граф, имеющий n вершин и m ребер. У каждой вершины есть значение на ней. Надо обрабатывать 2 типа запросов:*

1. *Дана вершина v. Среди всех вершин, достижимых из нее найти с наибольшим значением, вывести и заменить значение на 0.*
2. *Удалить ребро e.*

Решение 9.1. Будем обрабатывать запросы в offline. Пройдем по ним с обратного конца и будем постепенно конструировать дерево Краскала. Для второго типа запросов просто будем добавлять ребро e в граф (так как мы идем с конца). Для запросов первого типа сохраним номер вершины в дереве, которая сейчас является корнем v. Теперь пройдемся в прямом порядке по запросам. Запросы второго типа будем просто пропускать. Запросы первого типа теперь превращаются в запросы максимума в поддереве и изменения. Это достаточно известная задача, решаемая деревом отрезков.

Задача 9.2. *Дан неориентированный взвешенный граф с n вершинами и m ребрами. Дано q запросов. Каждый из них задается двумя вершинами v и u. Предположим, что там стоят люди, они хотят поменяться местами не оказавшись в одной вершине одновременно. Стоимостью этого запрроса назовем максимальное ребро, по которому прошли люди. Надо выбрать такой путь, чтобы минимизировать стоимость. Отвечать на запросы надо в онлайн.*

Решение 9.2. Для начала давайте заметим, что 2 человека могут поменяться местами только если они в одной компоненте и в ней есть цикл, либо вершина с 3 и более соседями. Назовем компоненту подходящей, если она удовлетворяет требованиям выше. Будем строить дерево Краскала отсортировав ребра по возрастанию. Если после добавления ребра компонента становится подходящей, то пометим вершину. Теперь ответ на запрос это просто наименьший помеченный общий предок v и u.

9.4 Заключение

Задачи на эту технику встречаются часто по сравнению с другими из этого сборника:

1. IOI 2018
2. APIO 2020
3. Atcoder E

10 Multipoint Evaluation

11 Mo‘s Algorithm Extended

12 MinPlus Convolution

13 Or/And/Xor Convolution

14 Slope Trick

15 Farach-Colton and Bender