Структура данных Дерево отрезков и её применение в задачах

Девятериков Иван

Содержание

1. Дерево отрезков **1**
1.1. Дерево отрезков . . . 1
1.2. Структура . . . 2
1.3. Реализация . . . 3
1.4. Отрезок с максимальной суммой . . . 6
1.5. Максимальная последовательность подряд идущих 0 . . . 6
1.6. Возрастающая на 1 последовательность . . . 6
1.7. Ближайший меньший на отрезке . . . 7
1.8. К-я единица . . . 7
1.9. Посчитать количество и сумму делителей у произведения . . . 8
1.10. Ксюша и битовые операции . . . 8
1.11. Сбалансированный плейлист . . . 10

2. Дерево отрезков с массовыми операциями **11**
2.1. Прибавление на отрезке и значение в точке . . . 11
2.2. Прибавление и минимум на отрезке . . . 12
2.3. Проталкивание . . . 14
2.4. Девочка и максимальная сумма . . . 15
2.5. Сортировка подстроки . . . 15
2.6. XOR на отрезке . . . 16
2.7. Занятия физкультуры . . . 17
2.8. Количество троек инверсии . . . 17
2.9. Копирование данных . . . 17
2.10. НВП . . . 18
2.11. Количество различных НВП . . . 18
2.12. Точка, покрытая максимальным числом прямоугольников . . . 19
2.13. Площадь объединения прямоугольников . . . 19
2.14. Количество точек в прямоугольнике . . . 21

3. Количество различных чисел на отрезке **23**
3.1. оффлайн запросы без изменения за $\log_2(n)$ . . . 23
3.2. онлайн запросы без изменений за $\log_2(n)$ с помощью ПЕРС ДО . . . 24
3.3. мердж-сорт три с изменениями . . . 25

4. Дебаг и стресс-тестирование **26**
4.1. Дебаг . . . 26
4.2. Стресс-тест . . . 26

5. Ссылки **28**

Лекция #1: Дерево отрезков

14 января 2022 г.

1.1. Дерево отрезков

Дерево отрезков (ДО или segment tree) — это **самая мощная структура** данных в спортивном программирование.

Условие

Задана последовательность $[a_1, a_2, \dots, a_n]$. Будем называть отрезок элементов заданной последовательности $[a_l, a_{l+1}, \dots, a_r]$ корректным, если он представляет собой корректную последовательность: a_l является минимальным числом на этом отрезке, а a_r — максимальным.
Например, $[2, 3, 1, 1, 5, 1]$ можно разбить на три корректных отрезка: $[2, 3]$ и $[1, 1, 5]$ и $[1]$.

Определите минимальное количество корректных непересекающихся отрезков, на которое можно разбить заданную последовательность.

Тут работает жадный алгоритм. Пусть i начало текущего отрезка. Найдём $i < j$, такую что $a_i > a_j$. Логично, что отрезки вида $[i, \dots, j]$ и более не подходят, так как a_i уже не является минимумом, а отрезки меньше $[i, j-1]$ как раз подходят. Значит, достаточно на отрезке $[i, j-1]$ найти индекс с максимальным значением, если таких индексов несколько взять максимально правый.
Данный алгоритм работает за $O(n^2)$.

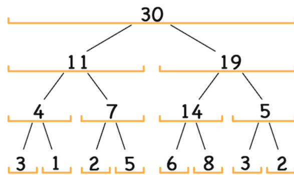
Давайте поймём из чего состоит такая временная сложность, во-первых нахождения ближайшего j меньшего a_i, во-вторых нахождения максимально значения на отрезке $[i, j-1]$.
Нахождения ближайшего меньшего можно сделать для каждого i изначально всего за $O(n)$ линейно с помощью стека.
А как брать максимум? Вот тут можно использовать ДО.... Итоговая асимптотика $O(n \log n)$.

Дерево отрезков умеет такие операции:
`get(l, r)` Получить значение f на множестве элементов $[a_l \dots a_r]$ за $O(\log_2(n))$.
`update(pos, val)` Обновить значение на позиции $a_{pos} = x$ за $O(\log_2(n))$.
`update(l, r, val)` Обновить множество элементов $[a_l \dots a_r]$ за $\approx O(\log_2(n))$.
`build(a)` Построить дерево на множестве элементов $[a_0 \dots a_{n-1}]$ за $O(2 \times n)$.

ДО занимает $\approx 2 \times n$ памяти.

Функция f может быть любой: почти всегда это **арифметические** операции над числами $(min/max/sum)$, но иногда могут быть более сложные функции. Главное, что мы можем скомбинировать ($combine$) результат для [TODO] от двух частей.
Сначала мы будем рассматривать арифметические операции, но потом будут красивые и не очевидные трюки.

1.2. Структура



Структура представляет собой бинарное дерево.
Вершины-листья - элементы исходного массива.

Другие вершины дерева имеют не более двух детей и содержат результат операции от своих них [разделяйка].
Корень $f(a[0, \dots, n))$, левая $f(a\left[0, \dots, \frac{n}{2}\right))$, правая $f(a\left[\frac{n}{2}, \dots, n-1\right))$ и тд.

Теорема 1.2.1. ДО хранит в памяти $4n$ вершин (при достраивании до $n = 2^k$).

Доказательство. Понять это можно следующим образом: первый уровень дерева отрезков содержит одну вершину (корень), второй уровень — две вершины, на третьем уровне будет четыре вершины, и так далее, пока число вершин не достигнет n. Таким образом, число вершин в худшем случае оценивается суммой $1 + 2 + 4 + \dots + 2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2 n \rceil + 1} < 4n$. ■

Замечание 1.2.2. При $n \neq 2^k$, не все уровни дерева отрезков будут полностью заполнены. Например, при $n = 3$ левый сын корня есть отрезок $[0 \dots 1]$, имеющий двух потомков, в то время как правый сын корня — отрезок $[2 \dots 2]$, являющийся листом.

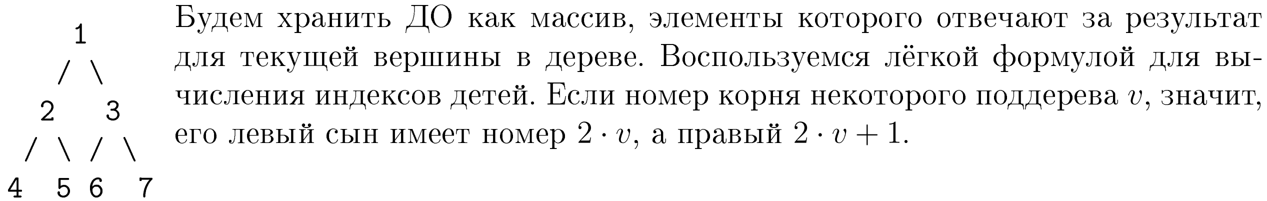
Теорема 1.2.3. Высота ДО $\log n$

Доказательство. Если на одном слое есть k вершин, то на следующем будет $2k$ вершин. Это следует из количества слагаемых в предыдущем доказательстве. ■

Следствие 1.2.4. Время одной операции в ДО – $\log n$.

1.3. Реализация

ДО можно писать как снизу, так и сверху. Эти подходы лишь различимы по написанию кода, но ДО сверху легче писать и понять. Пока мы будем рассматривать только ДО, написанное **сверху вниз**. Оно представляет собой спуск из вершины в левого и правого сыновей, используя рекурсию. Все запросы будут логично запускаться из корня дерева (самой верхней вершины).



Будем хранить ДО как массив, элементы которого отвечают за результат для текущей вершины в дереве. Воспользуемся лёгкой формулой для вычисления индексов детей. Если номер корня некоторого поддерева v, значит, его левый сын имеет номер $2 \cdot v$, а правый $2 \cdot v + 1$.

Такой приём помогает быстрее писать, а также занимает всего $4 \cdot n$ памяти (если n не степень двойки). Это можно заметить, если выписать все номера вершин, они будут образовывать последовательность целых чисел с шагом 1.
Приёмы, чтобы проще и удобнее писать:

1. Определиться с нейтральным элементом — такой элемент, который не надо рассматривать. Он будет находиться на позициях, в которых нет данных. В ДО на сумму для удобства можно использовать 0, как нейтральный элемент.
2. Написать **всего один** раз функцию `combine(a, b)`, которая будет принимать два аргумента (результат левого и правого сына) и возвращать результат, который будет записан в корне. И просто вызывать её, когда надо.
3. Границы запроса использовать не как отрезок, а как полуинтервал $[l, r)$.
4. Писать всё в одном стиле с одинаковыми переменными, потому что все функции очень похожи.

• **Введём обозначения** .

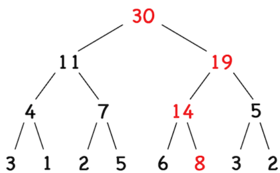
v — корень некоторого поддерева.
Вершина v отвечает за полуинтервал $[tl, tr)$.
Запросы к ДО будут $[ql, qr)$.
Тогда отрезок, за который отвечает левый сын, будет $[tl, (tl+tr)/2))$, а правый — $[(tl+tr)/2, tr)$.
Условие, при котором мы будем находиться в листе, когда полуинтервал будет отвечать всего за 1 элемент, $tl + 1 = tr$.

• Построение

Построение — самая лёгкая функция в ДО.
Мы спускаемся от корня, идём в сыновья, пока не дойдём до листьев. В лист мы записываем результат, который должен храниться, допустим, это значение в массиве.
Получается, что мы заходим в каждую вершину единожды, а возвращаясь, записываем результат в вершине, равный применению функции `combine` от результатов его сыновей. Логично, что в такой рекурсии перед присвоением ответа в вершину, значение в детях будет уже посчитано.

```
void build(int v, int tl, int tr) {
	if (tl + 1 == tr) {
		t[v] = a[tl];
	} else {
		int tm = (tl + tr) / 2;
		build(2 * v, tl, tm);
		build(2 * v + 1, tm);
		t[v] = combine(t[2 * v], t[2 * v + 1]);
	}
}
```

• Модификация



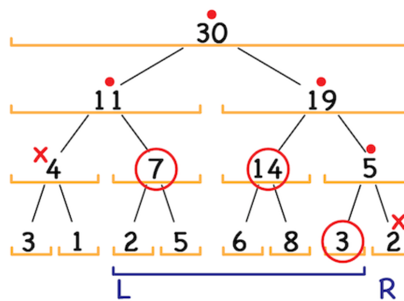
Когда изменяется элемент массива, нужно изменить соответствующее число в листе ДО, и далее пересчитать значения, которые от этого изменятся.
Надо пересчитать все вершины, которые содержат данный отрезок с этим элементом, заново. Такие вершины лежат по одной на каждом уровне от листа до корня. Значит, их ровно $\log n$.
Находясь в вершине, нам надо спуститься в того сына, который отвечает за отрезок, в котором произойдёт изменение.

```
void upd(int v, int tl, int tr, int pos, int val) {
	if (tl + 1 == tr) {
		t[v] = val;
		return ;
	}
	int tm = (tl + tr) / 2;
	if (pos < tm)
	    upd(2 * v, tl, tm, pos, val);
	else
	    upd(2 * v + 1, tm, tr, pos, val);
	t[v] = combine(t[2 * v], t[2 * v + 1]);
}
```

• Запрос получения результата

Давайте попробуем собрать результат на отрезке $[l \dots r)$ из уже посчитанных результатов. Для этого запустим рекурсивный обход дерева отрезков. При этом будем обрывать рекурсию в двух ситуациях.

1. Отрезок, соответствующий текущему узлу не пересекается с отрезком $[l \dots r)$. В этом случае все элементы в этом поддереве находятся вне области, в которой нам нужно посчитать сумму, поэтому глубже можно не идти.
2. Отрезок, соответствующий текущему узлу, целиком вложен в отрезок $[l \dots r)$. В этом случае все элементы в этом поддереве находятся в области, в которой нам нужно посчитать сумму, поэтому нам нужно добавить к ответу результат этого поддерева, который записан в текущем узле.



$\times$ - рекурсия оборвалась $\circ$ - число добавилось к ответу, и рекурсия оборвалась

Чтобы разобраться, почему это работает за $\mathcal{O}(\log n)$, нужно оценить количество «интересных» отрезков — тех, которые порождают новые вызовы рекурсии. Это будут только те, которые только частично содержат границу запросов — остальные сразу завершатся. Обе границы отрезка содержатся в $\log n$ отрезках, а значит, и итоговая асимптотика будет такая же.

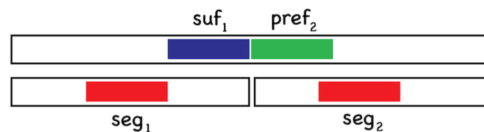
```
int get(int v, int tl, int tr, int ql, int qr) {
        if (qr <= tl || tr <= ql) {
                ruturn neitral_element;
        }
        if (ql <= tl && tr <= qr) {
                return t[v];
        }
        int tm = (tl + tr) / 2;
        return combine(get(2 * v, tl, tm, ql, qr),
                       get(2 * v + 1, tm, tr, ql, qr));
}
```

1.4. Отрезок с максимальной суммой Задача

Рассмотрим отрезок x, который разделяется на две половинки. Мы хотим для отрезка x найти значение seg — сумму на подотрезке с максимальной суммой. Заметим, что зная только $seg1$ и $seg2$ (ответы для половинок), мы не можем получить seg, потому что ответ для x может пересекать оба отрезка.

Но в случае пересечения, отрезок ответа состоит из суффикса левой половины и префикса правой половины.

Давайте на каждом отрезке будем хранить ещё значения $pref$ и suf — префикс и суффикс с максимальной суммой. Тогда можно посчитать seg следующим образом: $seg = max(seg1, seg2, suf1 + pref2)$.



Пересчёт. Рассмотрим $pref$, suf (будет считаться аналогично).
Для максимальный префикса возможны 2 случая:
1. $left_{pref} < len(left_{node})$. Ничего не меняется.
2. $left_{pref} = len(left_{node})$. Тогда префикс можно дополнить префиксом правой половины.
Для удобства (чтобы не писать проверки на размеры частей) можно ввести sum, равное обычной сумме на отрезке.
Тогда $pref = max(pref1, sum1 + pref2)$, аналогично, $suf = max(suf2, sum2 + suf1)$.
Если написать `combine` аккуратно, то можно будет отвечать на запросы вида $max_segment(l, r)$. Асимптотика $\mathcal{O}(\log n)$ на запрос.

1.5. Максимальная последовательность подряд идущих 0 Задача

Эта задача похожа на предыдущую. Только тут будем хранить не отрезки для максимальной суммы, а максимальные длины последовательностей из 0. Асимптотика $\mathcal{O}(\log n)$ на запрос.

1.6. Возрастающая на 1 последовательность Задача

Для удобства сделаем $a_i = a_{i-1} - 1$.
Данная задача свелась к задаче нахождения максимальной последовательности подряд идущих 0. Это мы уже умеем делать.
С запросами изменения на отрезке просто. Когда к отрезку добавляется некоторый x, это означает, что ответ для отрезка не поменялся. Но поменяться должны элементы на границах, потому что $a_l = a_{l-1} - 1$ и $a_{r+1} = a_r - 1$. Тогда надо просто изменить значение в позициях l и $r + 1$. Асимптотика $\mathcal{O}(\log n)$ на запрос.

1.7. Ближайший меньший на отрезке

Решение #1 За $\log^2 n$
Будем изменять размер отрезка, который будем рассматривать (с помощью простейшего бинпоиска).
Рассмотрим полуинтервалы $[l, m)$ и $[r, m)$, возьмем минимум в них с помощью ДО.
Если в первом полуинтервале есть число $< x$, то мы сдвигаем правую границу, иначе — левую.
Если ни в первом, ни во втором нет чисел $< x$, тогда ответ не существует.

Решение #2 За $\log n$ спуск по дереву отрезков.
Обычное ДО на минимум.
Давайте смотреть на вершины слева направо и делать вот что: если интервал, за который в ДО отвечает левый сын, пересекается с интервалом, на котором мы ищем ответ и минимум в левом поддереве $< x$, то пойдем в левого сына. Если ответ не был найден в левом сыне, то повторим то же самое с правым.
Если левый и правый сын нам не подходят, то перестанем дальше спускаться в эти поддеревья.

Теорема 1.7.1. Данная рекурсивная функция займет у нас $\mathcal{O}(\log n)$ времени.

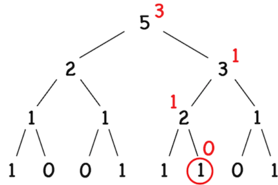
Доказательство. Пусть мы посетили хотя бы 3 листа. Лист, выходящий за пределы слева и выходящий за пределы справа, и ещё какой-то лист между ними. Если лист с ответом между левой и правой границей, то мы не могли зайти в лист, который за границей справа, потому что попросту обходим слева направо, значит, он совпадает с каким-то из прошлых двух, чего быть не может, ибо мы посещаем каждую вершину по одному разу.
Из того, что мы посетим не более двух листов, очевидно, следует, что асимптотика такого решения $\mathcal{O}(\log n)$. ■

1.8. К-я единица Задача

Будем в ДО на сумму хранить только 1, если на данной позиции находится 1, и 0 иначе.

Нахождение k-ой единицы равносильно нахождению самого левого префикса с суммой k. Сумму единиц берём по отрезкам, которые внутри запроса.
Если $left_{sum} >= k$, значит, k-ая единица находится в левом поддереве, иначе, нам нужно запустить поиск единицы под номером $k - left_{sum}$ в правом поддереве.
Это называется спуск по ДО.

Если нет запросов на изменение, для получения количества единиц на отрезке можно использовать префиксные суммы. Асимптотика будет $O(\log n)$ на запрос.
При изменениях элементов префиксные суммы нам уже не помогут. Можно будет использовать ДО, чтобы получать количество 1 на отрезке (нужные нам для спуска). Тогда асимптотика $O(q \times log^2 n)$.



Пример запроса `find(k=3, [1, 7))`. Мы начинаем в корне, отрезок $[0, 8)$. Смотрим на левый подотрезок $[0, 4)$, на нём сумма $sum[1, 4) < k$.
Поэтому мы спускаемся в правый подотрезок $[4, 8)$ и ищем на нём $k - sum[1, 4) =$ 2-ю единицу. В левом подотрезке $[4, 6)$ сумма $sum[4, 6) \geqslant k$, значит, наша единица лежит в подотрезке $[4, 6)$. И наконец, в левом подотрезке $[4, 5)$ сумма $1 < 2$, значит, наша единица находится в правом подотрезке $[5, 6)$. Ответ 5.

1.9. Посчитать количество и сумму делителей у произведения

Пусть мы разложим произведение на простые множители $n = p_1^{k_1} \cdot \ldots \cdot p_i^{k_i}$
Количество (τ) и сумма (σ) делителей — **мультипликативные функции**.
Мультипликативные функции $f(ab) = f(a) \cdot f(b)$ (если a и b **взаимно простые**).
Пусть p — простое число.
$\tau(p) = 2 \Rightarrow \tau(p^k) = k + 1$
$\tau(n) = (k_1 + 1)(k_2 + 1) \ldots (k_i + 1)$
$\sigma(p^k) = p^0 + p^1 + \ldots + p^k = \frac{p^(k+1) - 1}{p - 1}$ (геометрическая прогрессия)
$\sigma(n) = \sigma(p_1^{k_1}) \cdot \ldots \cdot \sigma(p_i^{k_i})$

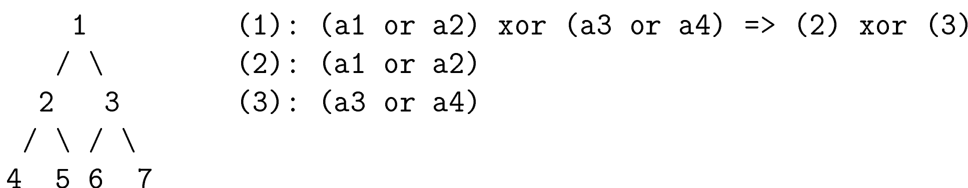
В вершине будем хранить разложение на простые множители с их степенями (факторизация). (`vector<pair<int, int»`)
Функция `combine(a, b)` сольёт два массива в 1 (для одинаковых простых делителей увеличим их степени).

- **Асимптотика**

У числа A не больше $log(A)$ простых делителей. Следователь $tau(n)$ и $sigma(n)$ за $\mathcal{O}(\log(A))$ для произвольного A. НО я ПОКА НЕ ДУМАЛ ОБ ЭТОМ ТАК КАК ЭТО ЗАШЛО СРАЗУ. ПРОСТЫХ ЧИСЕЛ ОТ [1; 1e6] = 78498.
cnt - количество различных простых в вершине. Но я даже фиг знает как это доказывать.
`combine(a, b)` за $\mathcal{O}(cnt)$
`update` будет за $\mathcal{O}(\sqrt{x} + \log n \cdot cnt)$ (можно раскладывать число x используя решето всего за $\log(x)$)
`get` будет за $\mathcal{O}(\log n \cdot cnt)$

1.10. Ксюша и битовые операции

Задача AC
Рассмотрим получение ответа. Из условия понятно, что она уменьшает длину массива в 2 раза на каждом шаге, только чередуя операцию (побитовое OR/XOR).
Запишем в дереве на каждом уровне (уровень — все вершины, которые находятся на одной глубине от корня) результаты операций от двух детей (от соседних следующих элементов), применяя нужную операцию.



```
    1         (1): (a1 or a2) xor (a3 or a4) => (2) xor (3)
   / \        (2): (a1 or a2)
  2   3       (3): (a3 or a4)
 / \ / \
4  5 6  7
```

Эта структура похожа на ДО. Получается, что при в реализации функций `build` и `update` надо дополнительно поддерживать глубину.
Асимптотика: каждый запрос обновления будет выполняться за $\mathcal{O}(n)$ (массив a размера 2^n). Памяти при этом требуется $2n$.

1.11. Сбалансированный плейлист

Задача
Чтобы сделать циклический плейлист линейным, достаточно повторить его три раза.
Для решение данной задачи можно воспользоваться разными подходами.
Решение #1 За $O(n \log^2 n)$ или за $\mathcal{O}(n \log n)$, используя разряженную таблицу.
Это идея состоит из 2 частей.
1. Используя бинпоиск, будем перебирать границу, когда мы ещё будем слушать.
2. Проверку отрезка напишем с помощью ДО.
   Надо проверить, что $a_{pos} > 2 \times min(pos, right)$. Здесь $right$ — правая граница, которую мы рассматриваем, а pos — первый трек.

Решение состоит из запросов min, бинпоиска, перебора n позиции.
Асимптотика: $O(n \log^2 n)$.

Можно сделать ещё быстрее min за $\mathcal{O}(1)$, если использовать разряженную таблицу.
Асимптотика: $\mathcal{O}(n \log n)$. Um_nik duality

Решение #2 За $O(n \log n)$ или за $\mathcal{O}(n)$ zemen

Утверждение 1.11.1. Граница, при которой мы закончим слушать, будет не уменьшаться, при сдвиге влево позиции, с которой мы начнём слушать.

Доказательство. Рассмотрим (правильную) последовательность $a_1, a_2, \ldots, a_x, last$, она обрывается, когда $x = max(a) < \frac{last}{2}$. Заметим, что все значения элементов a находятся в диапазоне $\left[\lceil\frac{x}{2}\rceil, x\right]$. Любые числа из a дают хорошую последовательность. ■

Это очень похоже на метод двух указателей с поддержкой максимума $a[i \ldots j]$. Легко написать ДО или использовать метод скользящего окна.

Решение #3 За $O(n \log n)$ 300iq
Можно решить, сделав поиск ближайшего числа $< \frac{x}{2}$, где x — крутость текущего трека. Это можно сделать с помощью спуска по ДО. Это разбиралось ранее.
Также можно комбинировать эти идеи.
multiset RomaWhite
map mnbvmar
st + 2pointer rng_58

Лекция #2: Дерево отрезков с массовыми операциями

14 января 2022 г.

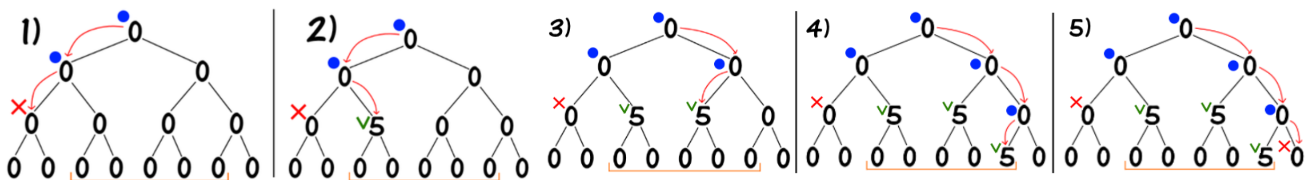
2.1. Прибавление на отрезке и значение в точке

1. `add(l, r, x)` — прибавить ко всем a_i $(l \leqslant i \leqslant r)$ значение x
2. `get(pos)` — получить значение a_i

• add

Чтобы прибавить к элементам на отрезке $[l \ldots r)$ значение x, разобьём исходный отрезок на несколько отрезков, каждый из которых покрывается каким-то узлом дерева. Разбиение мы делаем спуском, как до этого.
Но! Когда мы находимся в вершине ДО, которая отвечает за отрезок, который внутри запроса обновления, достаточно прибавить к значению в текущем узле x.
Время работы операции `add` $O(\log n)$ аналогично сумме на отрезке.

Пример вызова: `add(3, 8, 5)`



× - рекурсия оборвалась ✓- число прибавилось к вершине, и рекурсия оборвалась

• get

На значение a_i повлияли только вершины, отрезки которых содержат i.
Тогда все изменения, которые могли происходить с данным элементом, находятся на пути от листа до корня.
Тогда надо просто применить функцию (тут $+$) на пути. Асимптотика $\mathcal{O}(\log n)$ из-за высоты дерева.

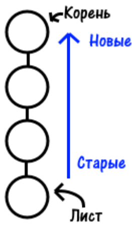
• Где это можно использовать?

Пусть $\otimes$ — произвольная операция.
Рассмотрим, что будет происходить со значением $a[i]$ при запросах, которые его изменяли. x — первый запрос, y — второй.

Спускаясь по дереву от корня до листа, мы пересчитываем значение не в том порядке, как мы их должны были применить, следуя запросам.
$(a[i] \otimes x) \otimes y = a[i] \otimes (x \otimes y)$ — это ассоциативность.
Также значение $a[i]$ не должно зависеть от порядка запросов над ним.

$a[i] \otimes x \otimes y = a[i] \otimes y \otimes x$ — это коммутативность.

- **Некоммутативные операции**

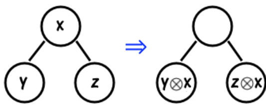


Научимся обрабатывать ассоциативные, некоммутативные операции.

Будем поддерживать инвариант, что операции на пути от любой вершины дерева до корня отсортированы по времени применения запросов от самых новых до более старых.

Реализовать это можно с помощью техники проталкивания `push`.

Если в вершине записано значение x, а в детях y и z, то после проталкивания значение в вершинах будет таким:



1. в левом $y \otimes x$
2. в правом $z \otimes x$
3. в текущей вершине исчезнет $x = -1$ (-1 — пометим, что ничего не надо).

   Мы применили запрос к детям, поэтому текущее значение больше не нужно.

Таким образом, мы сохранили инвариант и освободили текущую вершину.

Чтобы применить операцию к некоторой вершине, нужно освободить все вершины на пути от неё до корня, то есть необходимо пройти от корня до вершины сверху вниз и выполнить проталкивания.
Проталкивание работает за $\mathcal{O}(1)$, поэтому мы обрабатываем запросы, как раньше за $\mathcal{O}(\log n)$.

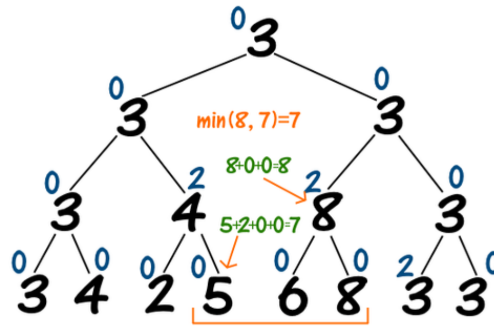
2.2. Прибавление и минимум на отрезке

1. `add(l, r, x)` — прибавить ко всем a_i ($l \leqslant i \leqslant r$) значение x
2. `min(l, r)` — найти минимум из всех a_i ($l \leqslant i \leqslant r$)

Давайте использовать два наших подхода.
Будем в каждой вершине ДО хранить минимум на отрезке и величину, прибавленную на отрезке.
Настоящее значение в вершине — это минимум в вершине плюс сумма прибавлений от предка этой вершины до корня дерева (изменения только в этих узлах ведёт к изменению текущего минимума).
$v_{val} = v_{val} + sum($на пути от предка этой вершины до корня дерева $)$.

При запросе `min(l, r)`, мы будем рекурсивно обходить дерево и поддерживать сумму на пути от корня до предка текущей вершины.

Пример: вызова `min(4, 7)` после вызова `add(3, 8, 2)`



Асимптотика $\mathcal{O}(\log n)$.

• Где это можно использовать?

Пусть у нас есть два вида запросов:

1. `modify(l, r, x)` — применить к каждому a_i $(l \leqslant i \leqslant r)$ $a_i = a_i \otimes x$
2. `calc(l, r)` — вычислить функцию на отрезке $[l, r]$ $a_l \odot a_{l+1} \odot \ldots \odot a_r$

$\otimes$ и $\odot$ — это любые операции, которые обладают свойствами ниже.

Для вычисления функции на отрезке ($\odot$) надо комбинировать её для детей.
Для изменения на отрезке надо комбинировать старую информацию в узле с новой ($\otimes$).
Следовательно необходимо, чтобы $\otimes$ и $\odot$ были ассоциативными.

Необходимо, чтобы $\otimes$ была коммутативной. Для проталкивания.

Мы только что использовали факт, что $min(left + x, right + x) = min(left, right + x)$
Давайте его обобщим:
Мы строим ДО на операции $\odot$.
Тогда результат за который отвечает вершина v будет $v_{val} \otimes x$).
Но мы знаем что $v_{val} = left \odot right$. Получаем.
Необходимо, $(left \odot right) \otimes x = (left \otimes x) \odot (right \otimes x)$

Примеры операций, подходящих под условия выше.

`modify`	`calc`
$*$	$+$
$+$	min, max
&	|

Если операция изменения будет сложение, а на заброс сумма. Надо будет сумму на отрезке домножить на его длину.

2.3. Проталкивание

Это симбиоз всех этих мини-подходов.
Если операция в запросе `modify` не коммутативна, то воспользуемся следующей техникой.
Будем сохранять порядок операций, проталкивая старые операции в глубь дерева.
При любом входе в вершину будем проталкивать изменение в детей, а при выходе из вершины будем пересчитывать значение от детей.
Пример: прибавление и сумма на отрезке.

```
void push(int v, int tl, int tr) {
    t[v] += lazy[v];
    if (tl + 1 != tr) {
        lazy[2 * v] += lazy[v];
        lazy[2 * v + 1] += lazy[v];
    }
    lazy[v] = 0;
}

void upd(int v, int tl, int tr, int ql, int qr) {
    push(v, tl, tr);
    if (qr <= tl || tr <= ql)  {
        return ;
    }
    if (ql <= tl && tr <= qr) {
        lazy[v]++;
        push(v, tl, tr);
        return ;
    }
    int tm = (tl + tr) / 2;
    upd(2 * v, tl, tm, ql, qr);
    upd(2 * v + 1, tm, tr, ql, qr);
    t[v] = t[2 * v] + t[2 * v + 1];
}
```

2.4. Девочка и максимальная сумма

Задача AC

Хотя эту задачу можно решить, используя только сортировку. Но мы поступим по-другому. Эти идеи помогут в следующей задаче.
Максимальный ответ можно получить, если мы элементы с большим значением поставим на позиции, в которых происходит больше запросов.
Нам осталось посчитать для каждого индекса количество запросов, которые будут содержать данный индекс. Это можно сделать с помощью прибавления на отрезке (ДО массовое).
После запросов нам надо получить количество в каждом индексе. Можно поступить различными способами. Предлагаю написать вам второй способом.

Способы:

1. Для каждой позиции найдём количество, используя функцию *get*.
   Асимптотика $\mathcal{O}(m \log n + n \log n))$.
2. В детях мы содержим не совсем правильную информацию. Она как бы правильная, если учесть массив массовых операций. А что нам мешает последовательно от корня сделать `push` от всех вершины? Очень будет похоже на `build`, только с `push`.

```
void apply(int v, int tl, int tr) {
        push(v, tl, tr);
        if (tl + 1 == tr) {
                return ;
        }
        int tm = (tl + tr) / 2;
        apply(2 * v, tl, tm);
        apply(2 * v + 1, tm, tr);
        t[v] = t[2 * v] + t[2 * v + 1];
}
```

   После этого все вершины будут хранить правильные ответы. Осталось их получить.
   Можно заметить (очень полезный приём), что ответ для i-го индекса находится уже на последнем уровне дерева. И мы просто можем получить индекс вершины. Это просто количество вершин на всех уровнях выше плюс i. Тогда просто можно сделать цикл.
   Асимптотика $\mathcal{O}(m \log n + 4 \cdot n)$.

2.5. Сортировка подстроки

sp1: Попробуйте решить за $\mathcal{O}(r - l)$ на запрос.
sp2: Сортировка подсчётом. Всего букв 26.
sp3: $\ldots abacb \cdots \to \ldots aabbc \ldots$. В части, которую мы отсортировали, для каждой буквы будет последовательный отрезок.
sp4: Для каждой буквы, своё ДО на массовые операции. (26 до).
sp5: Для вывода исходной строки, сделаем *apply*, как в предыдущей задаче.
sum: $\mathcal{O}(26 \times q \cdot \log n + 4 \cdot n)$.
Если интересно то вот using merging segment tree to solve problems about sorted list.

2.6. XOR на отрезке

Задача

Битовая операция *xor* **ассоциативна**.

Для чисел в десятичной системе счисления *xor* применяется как последовательный *xor* соответствующих разрядов в двоичной системе счисления. Давайте так и сделаем. Запишем все числа в двоичной системе счисления друг под другом. Недостающие разряды дополним 0. Давайте посмотрим на такую таблицу.

```
 100 4
1010 10
  11 3
1101 13
 111 7
abcd x
```

Пусть *abcd* равняется представлению x в двоичной системе счисления.

Тогда *xor* между битом на некотором разряде со всеми другими битами чисел на отрезке на этом разряде можно сделать ДО (с массовыми) для каждого бита.

Значит, нам надо поддерживать ДО для каждого разряда, а также для этого переписать функции.

```
1 void XOR(int ql, int qr, int x) {
2         bitset<MAXBIT> ar = x;
3         for (int DO = 0; DO < MAXBIT; DO++) {
4                 upd(DO, 1, 0, MAXN, ql, qr, ar[DO]);
5         }
6 }
```

Это легко сделать, если просто добавить вторую размерность: сделать массив дерева вида $t[MAXNBIT][MAXN]$, а в функцию дополнительно передавать разряд.

Сумму можно получить, последовательно складывая все разряды, домножив их на $2^{\text{в текущем разряде - 1}}$

Количество ДО $MAXBIT = \lceil \log_2 n \rceil = 20$. Асимптотика $\mathcal{O}(20 \cdot \log n)$.

2.7. Занятия физкультуры

Задача AC
Данную задачу нельзя решить, используя обычное ДО, потому что $n \leqslant 10^9$.
День — это отрезок длины 1.
Заметим, что отрезок длины 1 можно объединить ещё с некоторыми точками рядом, если все текущие сжатые отрезки можно будет сопоставить запросам.
Сопоставление отрезка запросу — это такой непрерывный отрезок в сжатых отрезках, который явно отвечает за этот вопрос.

Пример отрезков запросов:
$[1,5], [5,80], [80,150], [100,120]$

$$\left[1,\dots 4, \left[5\right], 6\dots 79, \left[80\right], 81,\dots,99, \left[100,\dots 120\right], \dots 150\right]$$

Пример сжатия:
$\underset{i=1,len=4}{[1,\dots 4]}, \underset{i=2,len=1}{[5,5]}, \underset{i=3,len=74}{[6,79]}, \underset{i=4,len=1}{[80,80]}, \underset{i=5,len=19}{[81,99]}, \underset{i=6,len=21}{[100,120]}, \underset{i=7,len=30}{[121,150]}$

Сопоставление запросам:
$[1,5] = [1,4] \cup [5,5]$
$[5,80] = [5,5] \cup [6,79] \cup [80,80]$
$[80,150] = [80,80] \cup [81,99] \cup [100,120] \cup [121,150]$
$[100,120] = [100,120]$

Чтобы обрабатывать общие границы, мы будем добавлять точки $l-1, l, r$.
Тогда мы можем сделать ДО по массиву длин отрезков.
Чтобы понять границы сжатого запроса, достаточно сделать бинпоиск по массиву сжатых границ запросов.
Максимальное количество границ запросов $3 \cdot q$ (это когда все тройки границ для всех элементов различные).
В ДО на сумму надо просто включать/выключать отрезок.
Тогда асимптотика:
1. Сортировка тройки границ $\mathcal{O}((3q) \cdot \log 3q)$.
2. Два бинпоиска для получения каждой границы в ДО. $\mathcal{O}(2q \cdot \log 3q)$.
3. Изменение при обработке запроса в ДО. Для каждого запроса $\mathcal{O}(\log 3q)$.
4. Ответ на каждый запрос находится в корне, заранее сделаем от него `push`.

Итоговая асимптотика $\mathcal{O}((3q) \cdot \log 3q + 2q \cdot \log 3q + q \cdot \log 3q) = \mathcal{O}((3q) \cdot \log 3q) = \mathcal{O}(q \cdot \log q)$.
Также можно решить задачу с помощью динамического ДО.

2.8. Количество троек инверсии

Задача AC
sp1: Для каждого j посчитаем количество i. $(a_i > a_j)$.
sp2: Для каждого k почитаем сумму по всем ответам для j. $(a_j > a_k)$.
sp3: Два ДО. Первое для количеств инверсий $(a_i > a_j)$. Второе для суммы ответов из первого ДО. Также надо сжать массив.

2.9. Копирование данных

Задача AC
sp1: Что делать с отрезком $[y \dots y+k]$ массива b?
sp2: Какую информацию нам надо знать для ответа на запрос типа 2?
sp3: Присваиваем $[y \dots y+k]$ значение номера запроса.
sp4: Для запроса типа 2, мы будем спускаться в лист. Какую информацию мы должны вернуть.
sp5: Возвратим максимальный номер запроса. Задача решается без $push$.

2.10. НВП

НВП - такая последовательность индексов $i_1 \dots i_k$, что:
$i_1 < i_2 < \dots < i_k$ и $a[i_1] < a[i_2] < \dots < a[i_k]$

Будем использовать ДП. $dp[x]$ — max длина НВП, оканчивающейся в элементе x.

$$dp[a[i]] = \max\left(1, \max_{\substack{j=0\dots i-1 \\ a[j]<a[i]}} (dp[j]+1)\right)$$

$dp = dp[0], dp[1], \dots dp[a[i]-1], dp[a[i]+1] \dots dp[max_element]$.
Следовательно все значение $dp[j]$, которые на текущем шаге надо рассмотреть находятся левее $a[i]$. Значит $dp[a[i]] = max(dp[0 \dots a[i]-1])$.

Это работает корректно, потому что на текущем шаге рассматриваем все значения которые встречались раннее.

В ДО нам нужно получать максимум на отрезке и изменять значение в точке $a[i]$.
Асимптотика $\mathcal{O}(n\log(max_elemet))$.
Если сжать значения, асимптотика будет $\mathcal{O}(n\log n)$.

2.11. Количество различных НВП

В ДО надо хранить пару значений (длина НВП, количество НВП).
Также надо переписать `combine`.
1. Если длины НВП двух частей одинаковые, возвращает пару $(len, cnt_a + cnt_b)$.
2. Иначе возвращает элемент с большой длиной НВП.

Асимптотика $\mathcal{O}(n\log n)$.

2.12. Точка, покрытая максимальным числом прямоугольников

Задача
https://pastebin.com/0psZgeC1
Сожмём координаты x.
Сделем ДО, которое будет отвечать за координаты y.
Сделаем сканлайн, при открытие будем помечать $(+1)$ отрезок $[y_{down}, y_{max}]$. При закрытие -1 на отрезке.
Нахождение количества это максимум на отрезке.
Асимптотика $\mathcal{O}(n \log n)$.

2.13. Площадь объединения прямоугольников

На самом деле $S_{\text{объединения}} = L \times L - S_{\text{вне объединения}}$.
Будем делать, как предыдущую задачу.
В ДО будем хранить сколько y координат не заняты.
Тогда каждый раз если мы рассматриваем $[x_1, x_2]$ будем добавлять к ответу $(x_2 - x1) \times (L - CNT_0)$.
Чтобы поддерживать количество 0 в ДО с массвовыми операциями вида ± 1. Надо поддерживать в вершине количество и сам минимум в поддереве.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const int MAXN = 2e5 + 7;

int a[MAXN];
int p[4 * MAXN];

pair<int, int> T[4 * MAXN];

inline pair<int, int> merge(pair<int, int> l, pair<int, int> r) {
    if (l.first == r.first) {
        return {l.first, l.second + r.second};
    } else {
        return (l.first < r.first ? l : r);
    }
}

void push(int v, int tl, int tr) {
    if (p[v] == 0) return;
    if (tl + 1 != tr) {
        p[v << 1] += p[v];
        p[v << 1 | 1] += p[v];
    }
```

```
25     T[v].first += p[v];
26     p[v] = 0;
27 }
28
29 void build(int v, int tl, int tr) {
30     if (tl + 1 == tr) {
31         T[v] = {0, a[tl]};
32     } else {
33         int tm = (tl + tr) >> 1;
34         build(v << 1, tl, tm);
35         build(v << 1 | 1, tm, tr);
36         T[v] = merge(T[v << 1], T[v << 1 | 1]);
37     }
38 }
39
40 void upd(int v, int tl, int tr, int ql, int qr, int val) {
41     push(v, tl, tr);
42     if (qr <= tl || tr <= ql) return;
43     if (ql <= tl && tr <= qr) {
44         p[v] += val;
45         push(v, tl, tr);
46         return;
47     }
48     int tm = (tl + tr) >> 1;
49     upd(v << 1, tl, tm, ql, qr, val);
50     upd(v << 1 | 1, tm, tr, ql, qr, val);
51     T[v] = merge(T[v << 1], T[v << 1 | 1]);
52 }
53
54 void solve() {
55     int n;
56     cin >> n;
57     if (n == 0) {
58         cout << 0;
59         return ;
60     }
61
62     fill(begin(a), end(a), 1);
63     vector<int> x1(n), x2(n), y1(n), y2(n);
64     vector<int> y;
65     vector<pair<int, int>> line;
66     for (int i = 0; i < n; i++) {
67         cin >> x1[i] >> y1[i] >> x2[i] >> y2[i];
68         if (x1[i] > x2[i]) swap(x1[i], x2[i]);
69         if (y1[i] > y2[i]) swap(y1[i], y2[i]);
70         line.push_back({x1[i], (i + 1)});
```

```
71         line.push_back({x2[i], -(i + 1)});
72         y.push_back(y1[i]);
73         y.push_back(y2[i]);
74     }
75     sort(y.begin(), y.end());
76     y.resize(unique(y.begin(), y.end()) - y.begin());
77     for (int i = 0; i + 1 < y.size(); i++) {
78         a[i] = y[i + 1] - y[i];
79     }
80     sort(line.begin(), line.end());
81     build(1, 0, MAXN);
82     int SUMA = 0;
83     for (int i = 0; i < MAXN; i++) SUMA += a[i];
84     ll S = 1LL * SUMA * (line.back().first - line.front().first);
85 
86     for (int i = 0; i < 2 * n; i++) {
87         int ind = abs(line[i].second) - 1;
88         int l = lower_bound(y.begin(), y.end(), y1[ind]) - y.begin();
89         int r = lower_bound(y.begin(), y.end(), y2[ind]) - y.begin();
90         int val = (line[i].second > 0 ? 1 : -1);
91         upd(1, 0, MAXN, l, r, val);
92         if (i + 1 < 2 * n && line[i].first != line[i + 1].first) {
93             push(1, 0, MAXN);
94             S -= 1LL * T[1].second * (line[i + 1].first - line[i].first);
95         }
96     }
97     cout << S << '\n';
98 }
```

Асимптотика $\mathcal{O}(n \log n)$.

2.14. Количество точек в прямоугольнике

РОИ2015 day1 c

Даны n прямоугольников и m точек. Для каждого прямоугольника требуется определить, сколько из данных m точек лежат внутри каждого из n прямоугольника.

Сожмём все координаты x и y.

Применим идею на подобие преффиксных сумм.

Решение #1 За $\log n$

Количество точек в прямоугольнике $\begin{bmatrix} (x1, y2) & (x2, y2) \\ \dots & \dots \\ (x1, y1) & (x2, y1) \end{bmatrix}$

Это количество точек с координатами $y1 \leqslant y \leqslant y2$ на префиксе по $x2$ минус количество точек не включая границу $x1$.

Сканлайн. 3 типа — открытие/закрытие прямоугольника, точка.

Будем поддерживать количество точек с y координатой в ДО.

Ответ на запрос это просто $get(0, x2) - get(0, x1 - 1)$.
$get(0, x2)$ мы посчитаем в момент закрытия прямоугольника, а $get(0, x1 - 1)$ - посчитали при открытие.
Асимптотика $\mathcal{O}(n \log n)$.

Решение #2 За $\log n$
Запрос на прямоугольники распишем на 3 запроса на префиксных прямоугольниках. Тогда нам надо находить кол-во точек в прямоугольники $(0, 0) - (x, y)$. [я пока писал, что можно и на два там просто запрос не на префиксе будет а на отрезке).
У нас два типа точек. Первый тип поставить точку на (x, y), а второй точки на прямоугольнике.
Сделаем как-бы сканлайн. Для каждой координаты y отсортируем по x и реальные точки, и точки запросов. При равенстве x-ов сначала будет идти запрос на добавление точки.
Заведём просто ДО на сумму для x координат. Где $a[i]$ - количество точек с координатой $x = i$.
Переберём y по возрастанию, для y переберём все события. Если текущее событие добвить точку сделаем upd в ДО. При запросе, просто возьмём сумму на префиксе.
Это работает верно, так как точки с координатой x мы не посчитаем при запросе на префиксе. А точки с большим y мы ещё не рассмотрел.
Решение #3 За $\log^2 n$ с изменениями
Простое merge sort tree. Для изменения в вершине декартач или gnu-set.

Лекция #3: Количество различных чисел на отрезке

14 января 2022 г.

3.1. оффлайн запросы без изменения за $\log_2(n)$

Задача у нас оффлайн, поэтому будем делать что-то на подобие сканлайна.
Будем считать массив $next$, передвигая правую границу. $next[x]$ - следующая позиция x.
Запрос сводится к нахождению количества элементов массива $next[i] \neq -1$, таких что $l \leqslant i \leqslant r$.
Это можно сделать сделать ДО или gnu_pnds_treeset.
Это в два раза быстрее чем алгоритм МО. $O(n \log_2(n))$ против $O(n\sqrt{n})$

```
#include <bits/stdc++.h>

using namespace std;

const long long MAXN = (1 << 19);
int t[2 * MAXN];

int get(int v, int tl, int tr, int ql, int qr) {
    if (qr <= tl || tr <= ql) return 0;
    if (ql <= tl && tr <= qr) return t[v];
    int tm = (tl + tr) / 2;
    return get(2 * v, tl, tm, ql, qr) + get(2 * v + 1, tm, tr, ql, qr);
}

void upd(int v, int tl, int tr, int pos, int val) {
    if (tl + 1 == tr) {
        t[v] = val;
        return ;
    }
    int tm = (tl + tr) / 2;
    if (pos < tm)
        upd(2 * v, tl, tm, pos, val);
    else
        upd(2 * v + 1, tm, tr, pos, val);
    t[v] = t[2 * v] + t[2 * v + 1];
}

struct Q {
    int l, r, i;
};

void solve() {
    int n;
    cin >> n;
```

```
    vector<int> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];

    auto b = a;
    sort(b.begin(), b.end());
    for (int i = 0; i < n; i++) a[i] = lower_bound(b.begin(), b.end(), a[i]) - b.begin();

    vector<int> nxt(n, -1);
    int q;
    cin >> q;
    vector<Q> qq(q);
    for (int i = 0; i < q; i++) {
        cin >> qq[i].l >> qq[i].r;
        qq[i].i = i;
        qq[i].l--, qq[i].r--;
    }
    sort(qq.begin(), qq.end(), [](Q a, Q b) {
        return a.r < b.r;
    });

    int qq_ind = 0;
    vector<int> ans(q);
    for (int r = 0; r < n; r++) {
        int x = a[r];
        if (nxt[x] != -1) {
            upd(1, 0, MAXN, nxt[x], 0);
        }
        nxt[x] = r;
        upd(1, 0, MAXN, nxt[x], 1);
        while (qq_ind < q && qq[qq_ind].r == r) {
            ans[qq[qq_ind].i] = get(1, 0, MAXN, qq[qq_ind].l, qq[qq_ind].r + 1);
            qq_ind++;
        }
    }
    for (int i = 0; i < q; i++) cout << ans[i] << "\n";
}
```

3.2. онлайн запросы без изменений за $\log_2(n)$ с помощью ПЕРС ДО

7sec/512mb. TODO сделать интерактивку.
$n, q \leqslant 300\ 000$
sp1: Пусть запросы с фиксированной правой границей. $(l_1, r), (l_2, r), \dots, (l_x, r)$.
sp2: Задача на Персистентное Дерево отрезков. Как удобно хранить информацию, используя персистентность?
sp3: Давайте сведём исходный запрос, к запросу количество единичек.

sp4: Хранить будем самое правое вхождение **для нашей фиксированной границы** $[x, y, x, z] \to [0, 1, 1, 1]$.
sp5: Запросы одного типа (без изменения).
sp6: Давайте как бы сохраним до, для каждой правой границы своей версией.
sum: $\mathcal{O}(n \log n + q \log n)$
https://pastebin.com/Agy5yBww

3.3. мердж-сорт три с изменениями

с изменениями

Лекция #4: Дебаг и стресс-тестирование

14 января 2022 г.

4.1. Дебаг

1. Сделать размер массива маленьким. Оптимально 16 элементов.
2. Посмотреть, что происходит с запросами и деревом. Попутно нарисовав его на листочке.
3. Посмотреть тест (или его большую часть).
4. Написать стресс-тест.

4.2. Стресс-тест

Стресс-тестирование - это метод, с помощью которого мы можем запустить наше решение (которое, не правильное) на случайных тестах и сопоставить его результат с вывод решения, которое является решением грубой силы (медленное но точно правильное) или принятым решением кого-то другого.
Правильность медленного решения можно проверить, отослав код и оно получает вердикт TL.

- **Требования** :

1. Решение, которое мы хотим протестировать.
2. Решение методом грубой силы, которое дает правильное решение.
3. Генератор для генерации тестовых примеров в соответствии с задачей.

- **Принцип работы**

1. Генерировать случайный тест. Лучше его записать в файл.
2. Запустить решение, которое даёт правильный ответ но медленно.
3. Запустить решение, которое неправильное.
4. Сравнить результаты вывода двух решений.

- **Написание отдельного файла для стресс-тестирования**

1. Пишите генератор теста и проверку ответов двух решений в одном файле.
2. Лучше использовать *python*. Получается очень кратко и быстро писать.
3. Используйте *seed* для генератора. Чтобы при перезапуске стресс-теста проверять предыдущие тесты.

Есть другой подход в котором мы делаем тоже самое просто внутри основной программы.

Пример стресс-теста в одном отдельном от решения файле на *python*:

WA.cpp

```cpp
#include <iostream>
int main() {
  int t;
  long long a, b;
  std::cin >> t;
  while (t--) {
    std::cin >> a >> b;
    std::cout << (a + b) % 228 << '\n';
  }
  return 0;
}
```

OK.cpp

```cpp
#include <iostream>
int main() {
  int t;
  long long a, b;
  std::cin >> t;
  while (t--) {
    std::cin >> a >> b;
    std::cout << (a + b) << '\n';
  }
  return 0;
}
```

Код стресса

```
test_case 1 WA
Input:
2
314 192
137 71

Incorrect Output:
50
208

Expected:
506
208

test_case 2 OK
```

Пример вывода

Лекция #5: Ссылки

- Segment Tree Problems
- Persistent segment tree Problems
- много задач по темам + комментарии
- codeforces - площадка, где регулярно проводятся соревнования, система с более 6700 официальных задач
- polygon - сервис для подготовки задач по программированию.
- бакалаврская диссертация «Compressed segment trees and merging sets in $O(N \log U)$» (Lucian Bicsi, Бухарестский университет)
- Книга «Конспект лекций по алгоритмам» (Сергей Копелиович, НИУ ВШЭ)
- Учебный курс (Павел Маврин, Университет ИТМО)
- A simple introduction to "Segment tree beats" (Ruyi Ji, Пекинский университет)
- Efficient and easy segment trees (Александр Бачериков)
- Algorithm Gym :: Everything About Segment Trees (AmirMohammad Dehghan, Массачусетский технологический институт)
- Matrix Exponentiation tutorial (Kamil Debowski, Варшавский университет)
- Mo's Algorithm on Trees (Animesh Fatehpuria, Технологический институт Джорджии)
- Tutorial on Permutation Tree (Ashley Khoo)
- Searching Binary Indexed Tree in $O(\log(N))$ using Binary Lifting (Siddharth Nayya, Делийский технологический университет)
- e-maxx.ru/algo/segment_tree
- много структур