# 6. Segment Tree Beats

Segment Tree Beats (STB) — это структура данных[1], которая была разработана Ruyi jiry_2 Ji в 2016 году. Это очень мощный инструмент, идея которого состоит в том, что мы ослабляем условия выхода из рекурсии в дереве отрезков, в результате чего кажется, что алгоритм начинает работать за квадратичное время, но при помощи амортизационного анализа можно доказать, что на самом деле время работы сильно меньше ($O(n \log n)$, $O(n \log^2 n)$ и т.д.). Также эта структура данных позволяет работать с «исторической информацией» массива. Частным случаем Segment Tree Beats является структура Ji Driver Segment Tree, которая позволяет поддерживать операции вида «заменить все числа на отрезке массива $A$ на $\max(A_i, x)$», а также узнавать сумму на отрезке.

На английском языке на эту тему есть по большому счету только одна статья. На русском же языке, насколько мне известно, материалов на эту тему нет в принципе. Статья, которую вы сейчас читаете, не только полностью покрывает англоязычный текст, но и затрагивает большое количество тем, которые в ней не упоминались, поэтому, пожалуй, является самым полным материалом про Segment Tree Beats не на китайском языке на данный момент. Я попытался собрать все возможные идеи, которые есть на эту тему, а также дополнить несколькими своими.

На данный момент в русском языке нет какой-либо используемой альтернативы английскому названию, но если вы предпочитаете локализацию, то есть вариант «Анимешное Дерево Отрезков»[2].

В этой статье мы часто будем говорить про асимптотику. Всегда подразумевается, что $n$ — это размер массива, а $q$ — суммарное количество запросов.

[1] Назвать это структурой данных можно весьма условно. Это скорее набор идей, которые наслаиваются на дерево отрезков.

[2] codeforces.com/blog/entry/90460?locale=ru#comment-789207

## 6.1 Общая идея

Давайте посмотрим, как выглядит стандартная функция изменения в дереве отрезков с массовым обновлением и проталкиванием:

```
1  void update(int node, int l, int r, int ql, int qr, int
       newval) {
2      if (qr <= l || r <= ql) { // node is outside of the
           segment
3          return;
4      }
5      if (ql <= l && r <= qr) { // node is inside the segment
6          update_node(node, newval);
7          set_push(node, newval);
8          return;
9      }
10     // node intersects the segment
11     push_down(node);
12     int mid = (r + l) / 2;
13     update(2 * node, l, mid, ql, qr, newval);
14     update(2 * node + 1, mid, r, ql, qr, newval);
15     pull_up(node);
16 }
```

Пускай мы находимся в вершине `node`, которая отвечает за полуинтервал $[l, r)$ массива, и нас попросили обновить значения массива на полуинтервале $[ql, qr)$ значением `newval`.

Первое условие (`break_condition`) проверяет, что если отрезок, за который отвечает вершина `node`, не пересекается с отрезком, на котором мы делаем обновление, то в текущем поддереве ничего менять не надо, и можно просто вернуться назад.

Второе условие (`tag_condition`) проверяет, что если отрезок, за который отвечает вершина `node`, лежит полностью внутри отрезка, который мы обновляем, то мы обновим значение прямо здесь, а также сохраним `push`, который в будущем будем проталкивать в детей. После чего мы опять же завершаемся и возвращаемся назад.

Если же ни первое, ни второе условие не выполнились, то это значит, что отрезки запроса и текущей вершины пересекаются, но при этом текущая вершина не лежит полностью внутри запроса. В таком случае мы рекурсивно запускаемся из детей, не забыв предварительно протолкнуть информацию о старых обновлениях, а после завершения работы в детях восстанавливаем значение в текущей вершине через значения детей.

Этот код будет работать за $O(\log n)$, потому что на каждом уровне дерева отрезков не больше, чем две вершины могут пересекаться с отрезком запроса, но при этом не лежать в нем полностью, поэтому только из этих двух вершин мы рекурсивно запустимся на следующий уровень, а значит, на каждом уровне дерева мы посетим не более четырех вершин.

Segment Tree Beats основан на следующей идее: пускай запросы изменения таковы, что мы не всегда можем пересчитать значение на отрезке при условии выполнения `tag_condition`. Тогда давайте усилим условие `break_condition` и ослабим условие `tag_condition`, чтобы теперь мы могли уже пересчитать значение в вершине, не запускаясь рекурсивно, но при этом асимптотика не стала квадратичной.

То есть, теперь функция изменения будет выглядеть следующим образом:

```
void update(int node, int l, int r, int ql, int qr, int
    newval) {
    if (break_condition(node, ql, qr, newVal)) {
        return;
    }
    if (tag_condition(node, ql, qr, newval)) {
        update_node(node, newval);
        set_push(node, newval);
        return;
    }
    push_down(node);
    int mid = (r + l) / 2;
    update(2 * node, l, mid, ql, qr, newval);
    update(2 * node + 1, mid, r, ql, qr, newval);
    pull_up(node);
}
```

Иными словами, все, что нам нужно сделать — это придумать наиболее сильное условие `break_condition`, при котором в текущем поддереве запрос изменения точно ничего не изменит, а также наиболее сильное условие `tag_condition`, при котором можно будет обновлять значение в текущей вершине, не запускаясь рекурсивно из детей.

При этом заметьте, что `break_condition` и `tag_condition` из обычного дерева отрезков никуда не деваются. Скорее всего, если отрезки запроса и текущей вершины не пересекаются, то в этой вершине точно ничего не надо менять. С другой стороны, если текущая вершина не лежит полностью внутри отрезка запроса, то вряд ли можно пересчитать значение в ней, не запустившись рекурсивно в детей, так что эти условия будут выглядеть примерно следующим образом:

```
break_condition = qr <= l || r <= ql || ???
tag_condition = ql <= l && r <= qr && ???
```

В этой статье мы будем пытаться придумать, чем нужно заменить каждый из `???` в разных задачах.

Задачи в основном будут описываться запросами, которые в них нужно выполнять. К примеру, запрос `+=` означает, что необходимо уметь прибавлять какое-то значение на отрезке. Запрос `=` означает, что необходимо уметь присваивать какое-то значение на отрезке. Эти запросы являются весьма стандартными для дерева отрезков. Однако кроме них будут рассмотрены и более сложные: `max=`, `min=`, `%=`, `/=` и так далее. Они означают, что в результате запроса нужно заменить все элементы на отрезке на результат выполнения соответствующей функции от текущего значения и `newval`. К примеру, операция `max=` заменят все элементы на отрезке массива $A$ по правилу $A_i \to \max(A_i, newval)$.

Кроме того, не менее важно, какие операции типа `get` есть в задаче. К примеру, мы будем рассматривать следующие операции: $\sum$ — сумма на отрезке, $\max$ — максимум на отрезке, $\min$ — минимум на отрезке, $gcd$ — НОД на отрезке и т.д.

> **Замечание 6.1.1** Обратите внимание, что запрос типа `get`, то есть запрос получения какой-либо функции на отрезке, не меняется, потому что мы все еще поддерживаем корректную информацию о подотрезке вершины, когда мы в нее спустились от корня.

## 6.2 %=, = в точке, $\sum$

### 6.2.1 Формулировка

В этой задаче у нас есть массив неотрицательных целых чисел $A$ $(0 \leqslant A_i < C)$, а также имеются запросы трех типов:

1. Даны $ql, qr, x$ $(1 \leqslant x < C)$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $A_i \bmod x$.
2. Даны $qi, y$ $(0 \leqslant y < C)$. Нужно заменить элемент массива $A$ на позиции $qi$ на $y$.
3. Даны $ql, qr$. Необходимо вернуть сумму элементов массива $A$ на полуинтервале $[ql, qr)$.

Эту задачу можно найти здесь.

Также можно без проблем поддерживать и другие `get`-запросы, такие как max или min.

### 6.2.2 Решение

Вторую и третью операции мы будем выполнять как обычно. Осталось понять, в какой момент мы можем остановиться в первом запросе, чтобы обновить значение в текущей вершине, а также в будущем иметь возможность эффективно проталкивать это изменение в детей.

Давайте подумаем, каким должно быть `break_condition`? При каком условии ни одно число в данном поддереве не поменяется? В том случае, если все числа на подотрезке текущей вершины меньше, чем $x$. Иными словами, если максимум на этом отрезке меньше $x$. Поэтому `break_condition` в данном случае будет выглядеть следующим образом: `qr <= l || r <= ql || maxvalue[node] < x`.

Теперь надо придумать такое условие `tag_condition`, при котором нам не придется идти в детей. Здесь уже может быть несколько вариантов. К примеру, если целые части от деления всех чисел на отрезке на $x$ совпадают. Однако нам хватит более простого условия: если все числа на отрезке равны, то есть, иными словами, максимум на отрезке равен минимуму. В этом случае все числа на отрезке равны `maxvalue[node]`, поэтому операция `%=` на этом отрезке — это то же самое, что присвоить на этом отрезке всем элементам `maxvalue[node]` $\bmod x$. Это мы можем сделать так же, как и с запросом второго типа, не заходя в детей.

### 6.2.3 Доказательство

Осталось понять, почему при таком ослаблении `tag_condition` асимптотика остается приемлимой. На самом деле асимптотика этого решения — $O((n+q)\log n \log C)$.

Введем потенциал вершины дерева отрезков $\varphi(node)$, равный $\sum_{l \leqslant i < r} \log(A_i + 1)$, где $l, r$ — границы полуинтервала исходного массива, за который отвечает данная вершина. Прибавление единицы, конечно, ни на что особо влиять не будет, но оно необходимо, потому что $A_i$ могут быть равны нулю. Теперь введем потенциал $\Phi$, который будет равен сумме потенциалов всех вершин дерева. В любой момент времени потенциал можно оценить следующим образом: $0 \leqslant \Phi \leqslant O(n \log n \log C)$, потому что каждый элемент массива лежит в поддереве у $\log n$ вершин дерева отрезков и дает вклад $O(\log C)$.

И пускай суммарно за все время этот потенциал увеличился на $\Phi_+$. Давайте разобьем вершины, которые посетит запрос изменения, на три вида: **обычные**, **дополнительные** и **тупиковые**. Обычные вершины — это те вершины, которые посетило бы стандартное дерево отрезков, тупиковые вершины — это те необычные вершины, в которых выполнилось одно из условий `break_condition` или `tag_condition`, а дополнительные вершины — это все остальные вершины, которые посетил запрос, то есть те не обычные вершины, из которых мы вызвались рекурсивно. Тогда заметим,

что некоторые верхние вершины дерева будут обычными, потом под обычными будет какое-то количество дополнительных, и из дополнительных иногда будут торчать тупиковые. Из тупиковых уже рекурсивных вызовов нет. При этом заметим, что отец любой тупиковой вершины — это либо обычная вершина, либо дополнительная. Кроме того, у каждой вершины максимум 2 сына, так что тупиковых вершин максимум в два раза больше, чем обычных и дополнительных, поэтому их посещение не влияет на асимптотику, и можно следить только за дополнительными вершинами. Это можно было понять немного иначе: если мы делаем рекурсивный вызов и сразу завершаемся, то в каком-то смысле можно считать, что этот рекурсивный вызов не был сделан вовсе.

Теперь, когда мы оставили только обычные и дополнительные вершины, докажем, что при посещении дополнительной вершины потенциал в этой вершине (а следовательно и суммарный потенциал $\Phi$) уменьшается хотя бы на 1, тогда суммарное количество посещенных дополнительных вершин можно оценить как $O(n\log n\log C)+\Phi_+$, а обычных вершин мы на каждом запросе посещаем $O(\log n)$ штук. При этом в каждой вершине дерева мы делаем константное количество операций, так что асимптотика алгоритма будет равна $O(n\log n\log C+\Phi_+ + q\log n)$. Остается только показать, что $\Phi_+ \leqslant O(q\log n\log C)$, а также то, что потенциал уменьшается при посещении дополнительной вершины.

Данное рассуждение может показаться сложным и запутанным, но на самом деле в будущем все рассуждения об амортизированном времени работы будут очень похожи на это. Мы вводим какой-то потенциал. Понимаем, что он всегда находится в пределах от 0 до $\max\Phi$, смотрим, на сколько он может увеличиваться, а также, на сколько он уменьшается при посещении дополнительных вершин. Из этого делается вывод о времени работы алгоритма.

**Замечание 6.2.1** Это не совсем обычный способ измерения амортизированного времени работы. Обычно вводят $a_i = t_i - \Delta\Phi_i$, после чего получается, что время работы можно оценить как $\Delta\Phi+\sum a_i$. Однако в данном контексте такие рассуждения, пожалуй, менее удобны для понимания. Мы воспринимаем потенциал как кучку камней, которая может иметь ограниченный размер, за все время в нее положат какое-то конкретное ограниченное количество камней, а за каждую необычную операцию мы будем забирать из этой кучки камень.

Итак, давайте поймем, чему равен $\Phi_+$. Операция первого типа может только уменьшать числа в массиве, а операция третьего типа вовсе не меняет элементов массива. Поэтому увеличения потенциала могут происходить только во время операций второго типа. Мы изменили один элемент массива. Он был $\geqslant 0$, а стал $< C$, поэтому потенциал мог увеличиться максимум на $\log((C-1)+1)-\log(0+1)=\log C$ для каждой вершины, в поддереве которой есть этот элемент, а таких вершин $O(\log n)$. Всего запросов было $q$, поэтому суммарно потенциал увеличится максимум на $O(q\log n\log C)$. Что и требовалась показать.

Теперь покажем, почему при посещении дополнительной вершины ее потенциал уменьшается как минимум на 1. Если мы посещаем дополнительную вершину, это значит, что ко всем элементам на ее подотрезке нужно применить операцию %=, и при этом на этом отрезке есть хотя бы одно число, которое не меньше $x$. Воспользуемся следующим известным фактом:

**Теорема 6.2.2** Если $k \geqslant x$, то $k \bmod x \leqslant \frac{k-1}{2}$.

*Доказательство.* Во-первых, заметим, что условие $l \leqslant \frac{k-1}{2}$ для целых $l$ и $k$ равносильно

тому, что $l < \frac{k}{2}$.

Во-вторых, разберем два случая:

1. $k \geqslant 2x$. В этом случае $k \bmod x < x \leqslant \frac{k}{2}$. Первое неравенство верно просто потому, что остаток от деления всегда меньше модуля, а второе верно из-за того, что $k \geqslant 2x$.
2. $k < 2x$. В этом случае $k \bmod x = k - x < \frac{k}{2}$ в силу того, что $k < 2x$.

■

То есть, если мы посетили дополнительную вершину, то какое-то число на этом отрезке уменьшится больше, чем в два раза. Тогда раньше это число давало вклад $\log(k+1)$ в потенциал, а теперь $\leqslant \log(\frac{k-1}{2}+1) = \log(\frac{k+1}{2}) = \log(k+1)-1$. Таким образом, мы доказали, что потенциал этой вершины уменьшился хотя бы на один. Что и требовалось.

> **Замечание 6.2.3** В одной из последующих секций мы докажем, что абсолютно такое же решение будет работать за такую же асимптотику даже если присвоение происходит на отрезке.

## 6.3 `min=`, $\sum$, max (Ji Driver Segment Tree)

### 6.3.1 Формулировка

В этой задаче у нас есть массив целых чисел $A$, а также имеются запросы трех типов:

1. Даны $ql, qr, x$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны $ql, qr$. Необходимо вернуть сумму элементов массива $A$ на полуинтервале $[ql, qr)$.
3. Даны $ql, qr$. Необходимо вернуть максимум в массиве $A$ на полуинтервале $[ql, qr)$.

Эта задача доступна здесь или здесь. Это китайские сайты, поэтому там не так просто зарегистрироваться.

Также есть усложненная версия этой задачи «Нагайна».

### 6.3.2 Решение

Это самая стандартная задача на Segment Tree Beats. Собственно, статья на английском появилась после того, как участники из Китая массово решали задачу «Нагайна» при помощи Ji Driver Segment Tree, хотя изначально эта задача была на корневую декомпозицию.

В этой задаче мы будем хранить в каждой вершине следующие значения: `sum` — сумма на отрезке; `max` — максимум на отрезке; `cnt_max` — количество элементов на этом отрезке, которые равны максимуму; `second_max` — наибольший элемент на отрезке, который строго меньше, чем `max`. Обратите внимание на то, что это строгий второй максимум, то есть если на отрезке есть числа $0, 1, 2, 3, 3$, то `second_max` $= 2$.

Каким должно быть `break_condition`? В каком случае операция `min=` не поменяет ничего на текущем отрезке? В том случае, если все элементы на этом отрезке уже не больше, чем $x$, и ничего менять не надо. То есть, другими словами, если максимум не больше $x$:

```
break_condition = qr <= l || r <= ql || max <= x
```

А каким же должно быть `tag_condition`? В каком случае мы можем быстро обновить значения в текущей вершине? В том случае, если меняются только максимумы, то есть `second_max < x`.

```
tag_condition = ql <= l && r <= qr && second_max < x
```

В таком случае мы знаем, что `cnt_max` максимумов заменятся на `x`, и в этом случае легко можно пересчитать все значения в вершине, и вниз мы будем проталкивать как раз эту информацию: с каким числом нужно произвести операцию `min=`. Если нужно протолкнуть две операции `min=`, то достаточно проталкивать всего одну такую операцию с меньшим из параметров.

На самом деле можно заметить, что проталкиваемое значение можно не хранить вовсе, потому что оно всегда совпадает с `max` в этой вершине (либо проталкивать ничего не надо, но если мы протолкнем максимум, ничего не изменится). То есть мы в каком-то смысле проталкиваем в детей наш максимум на отрезке. Так писать код становится сильно приятнее.

### 6.3.3 Доказательство

Докажем, что это решение работает за $O((n+q)\log n)$.

Опять же воспользуемся методом потенциалов. Определим потенциал вершины $\varphi(node)$ как количество различных чисел на отрезке, за который отвечает эта вершина. Общий потенциал $\Phi$ определяется опять же как сумма потенциалов по всем вершинам дерева.

Количество различных чисел на отрезке не больше, чем длина этого отрезка, так что сумма потенциалов на одном уровне дерева отрезков не больше $n$. Таким образом, в любой момент времени потенциал ограничен следующим образом: $0 \leqslant \Phi \leqslant O(n\log n)$.

Чему равно $\Phi_+$? Во время `get`-запросов элементы массива не меняются, поэтому в них потенциал не увеличивается. В запросах первого типа потенциал может измениться только для обычных и дополнительных вершин, потому что для непосещенных вершин значения на отрезке не поменяются, а для тупиковых никакие значения не склеятся. При этом для дополнительных вершин количество различных чисел только уменьшается за счет того, что числа склеиваются, но увеличиться точно не может. Поэтому потенциал увеличивается только для обычных вершин, при этом единственное новое значение, которое может появиться, — это `x`, потому что каждое число либо не меняется, либо заменяется на `x`. Таким образом, потенциал мог увеличиться только у $O(\log n)$ обычных вершин, и для каждой такой вершины он увеличился максимум на 1. Поэтому за все время $\Phi_+ \leqslant O(q\log n)$.

Теперь поймем, что посещение дополнительных вершин уменьшает потенциал этой вершины хотя бы на 1. Это следует из того, что для дополнительных вершин $x \leqslant$ `second_max`, поэтому после выполнения этого запроса и `max`, и `second_max` заменятся на $x$, то есть два максимума как бы склеятся между собой, так что количество различных чисел в этой вершине точно уменьшится хотя бы на 1.

Таким образом, асимптотика алгоритма получается равной $O(n\log n + q\log n + q\log n) = O((n+q)\log n)$ (первое слагаемое от глобального изменения потенциала от начала до конца, второе слагаемое от $\Phi_+$, а третье слагаемое от посещения обычных вершин). Что и требовалось доказать.

> **Замечание 6.3.1** По аналогии с этой задачей можно поддерживать также операцию `max=` и даже их совмещение. Достаточно хранить 2 максимума, 2 минимума и их количества. Оценка времени работы от этого не изменится.
>
> Кроме того, можно добавить операцию присвоения на отрезке, потому что она так же, как и `min=` не сильно увеличивает потенциал.

## 6.4 min=, max=, +=, ∑, max, min

### 6.4.1 Формулировка

В этой задаче у нас есть массив целых чисел $A$, а также имеются запросы шести типов:

1. Даны $ql, qr, x$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны $ql, qr, y$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $\max(A_i, y)$.
3. Даны $ql, qr, z$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $z$.
4. Даны $ql, qr, t$. Нужно прибавить ко всем элементам массива $A$ на полуинтервале $[ql, qr)$ число $t$.
5. Даны $ql, qr$. Необходимо вернуть сумму элементов массива $A$ на полуинтервале $[ql, qr)$.
6. Даны $ql, qr$. Необходимо вернуть максимум элементов массива $A$ на полуинтервале $[ql, qr)$.
7. Даны $ql, qr$. Необходимо вернуть минимум элементов массива $A$ на полуинтервале $[ql, qr)$.

### 6.4.2 Решение

Решение никак не меняется. Оно такое же, как и раньше (только надо добавить стандартную операцию `+=`), однако теперь старое доказательство перестает работать. Нам надо будет придумать новое.

### 6.4.3 Доказательство

Почему же старое доказательство не работает?

По факту, новая операция здесь всего одна: `+=`, потому что по замечанию 6.3.1 остальные запросы просто встраиваются в Ji Driver Segment Tree. Однако с операцией `+=` потенциал из прошлой задачи не пройдет. Давайте посмотрим на пример массива:

$$1, 2, 3, 4, \ldots, \frac{n}{2} - 1, \frac{n}{2}, 1, 2, 3, 4, \ldots, \frac{n}{2} - 1, \frac{n}{2}$$

Корень дерева отвечает за весь массив, поэтому его потенциал равен $\frac{n}{2}$. Однако если мы прибавим $\frac{n}{2}$ ко второй половине массива, то массив станет выглядеть так:

$$1, 2, 3, 4, \ldots, n-1, n$$

И в этом случае потенциал корня стал равен $n$, а потенциалы других вершин не изменились. То есть, мы за одну операцию увеличили потенциал на $\frac{n}{2}$. Это слишком много. Нам нужен другой потенциал.

Доказывать мы, однако, будем уже не $O((n+q)\log n)$, а $O(n \log n + q \log^2 n)$. Кроме того, вроде бы теста, на котором это работало бы за $\log^2 n$ на запрос неизвестно, так что, возможно, это настоящий $\log n$, но этим мы, конечно, пользоваться не будем.

В этот раз потенциалом дерева $\Phi_1$ будет количество его вершин, для которых максимумы в левом и правом поддеревьях не совпадают. Скажем, что такие вершины являются помеченными. Аналогично, $\Phi_2$ — это количество вершин, для которых минимумы в левом и правом поддеревьях не совпадают. Тогда очевидно, что в любой момент времени $0 \leqslant \Phi_1, \Phi_2 \leqslant O(n)$. В каких случаях потенциал $\Phi_1$ мог увеличиваться? Если для какой-то вершины раньше максимумы в детях совпадали, а потом стали различаться. Это могло произойти только в том случае, если какие-то элементы на отрезке данной вершины поменялись, но не все. То есть текущая вершина пересекается с запросом, но не лежит в нем полностью. Это обязательно обычная вершина, таких

вершин $O(\log n)$ штук, так что за один запрос $\Phi_1$ могло увеличиться максимум на $O(\log n)$. Аналогично для $\Phi_2$. Так что $\Phi_{1_+}, \Phi_{2_+} \leqslant O(q \log n)$.

Теперь поймем, как уменьшаются потенциалы при операциях `min=` и `max=`. Докажем, что если мы посетили $m$ дополнительных вершин в операции `min=`, то $\Phi_1$ уменьшится хотя бы на $\frac{m}{\log n}$ (иными словами, если потенциал уменьшится на $k$, то мы посетим не больше $k \log n$ дополнительных вершин). Аналогично, для операции `max=`, уменьшаться будет $\Phi_2$. Тогда итоговая асимптотика будет равна $O(n \log n + q \log^2 n)$.

Почему потенциал $\Phi_1$ будет уменьшаться? Давайте докажем следующее утверждение:

**Теорема 6.4.1** В поддереве любой дополнительной вершины $v$ есть помеченная вершина $u$, которая после применения операции перестанет быть помеченной.

*Доказательство.* Если вершина $v$ дополнительная, то для этой вершины `max[v]` $> x$ и `second_max[v]` $\geqslant x$, потому что не выполнились `break_condition` и `tag_condition`. Тогда докажем, что в поддереве вершины $v$ есть такая вершина $u$, что для нее максимум в одном из детей равен `max[v]`, а в другом `second_max[v]`. И тогда сейчас эти числа различаются, а после применения операции оба будут равны $x$, поэтому из вершины $u$ пропадет пометка.

Почему же такая вершина $u$ существует? Посмотрим на самую глубокую вершину $t$ в поддереве $v$, для которой `max[t] = max[v]` и `second_max[t] = second_max[v]`. Такая вершина точно есть, потому что как минимум подходит сама вершина $v$. Для вершины $t$ это условие выполнено, а для ее детей — нет, потому что вершина $t$ — это самая глубокая такая вершина. Тогда заметим, что вершина $t$ как раз таки подходит на роль вершины $u$. Если максимум в поддереве $t$ равен `max[v]`, то и у одного из сыновей максимум равен тому же самому числу. При этом второй максимум в этом сыне не равен `second_max[v]`, так как тогда этот сын был бы более глубокой подходящей вершиной. То есть в этом сыне нет значений, равных `second_max[v]`. Тогда все эти значения находятся в другом сыне. При этом в другом сыне не может быть значений, равных `max[v]`, потому что тогда этот другой сын был бы более глубокой вершиной, чем $t$, которая нам подходит. Так что максимум в другом сыне — это `second_max[v]`. Поэтому как раз таки вершина $t$ подходит на роль вершины $u$, и в ней была метка, а после применения операции эта метка пропадет. ■

Пускай после применения операции пропало $k$ меток. Тогда все посещенные дополнительные вершины — это предки этих $k$ вершин, потому что по теореме у любой дополнительной вершины есть потомок, в котором пропала метка. У каждой из этих $k$ вершин есть $\log n$ предков, так что суммарно у них не более $k \log n$ предков, значит, мы посетим не больше, чем столько вершин. Что и требовалось доказать.

## 6.5 min=, +=, *gcd*

### 6.5.1 Формулировка

В этой задаче у нас есть массив неотрицательных целых чисел $A$ $(0 \leqslant A_i < C)$, а также имеются запросы трех типов:

1. Даны $ql, qr, x$ $(0 \leqslant x)$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны $ql, qr, y$ $(0 \leqslant y < C)$. Нужно прибавить ко всем элементам массива $A$ на полуинтервале $[ql, qr)$ число $x$.
3. Даны $ql, qr$. Необходимо вернуть наибольший общий делитель ($gcd$, НОД) элементов массива $A$ на полуинтервале $[ql, qr)$.

### 6.5.2 Решение + доказательство упрощенной версии задачи

Давайте сначала поймем, как решать эту задачу, если запросов первого типа нет. В отличие от других задач, это уже не так очевидно. Если ко всем числам на отрезке прибавили $x$, то не совсем понятно, как изменился НОД чисел на этом отрезке.

Как известно, $gcd(a,b) = gcd(a-b,b)$ (это факт, на котором основан алгоритм Евклида). То есть мы можем заменить одно из чисел на их разность. Но давайте вместо того, чтобы заменять, просто его добавим. Хуже не будет:

$$gcd(a,b) = gcd(a,b,a-b)$$

Этот факт можно обобщить для большего количества членов:

$$gcd(a,b,c) = gcd(a,b,c,a-b,b-c,c-a)$$

И так далее. То есть НОД чисел $a_1, a_2, \ldots, a_k$ равен НОДу этих чисел и всех их возможных $\frac{k\cdot(k-1)}{2}$ попарных разностей.

С другой стороны, давайте заметим, что из этих членов можно оставить только несколько. Нужно оставить всего $k$ из них так, чтобы их линейными комбинациями можно было получить $a_1, a_2, \ldots, a_k$. И если эти числа можно получить линейными комбинациями, то НОД такого подмножества будет равен $gcd(a_1, a_2, \ldots, a_k)$.

Давайте представим все числа в виде графа. Ребро между двумя вершинами будет соответствовать их разности. Тогда на самом деле нам достаточно взять какое-то остовное дерево этого графа, а также одно любое из чисел $a_1, a_2, \ldots, a_k$. Почему? Пускай мы взяли число $a_i$, а также какое-то остовное дерево. Как получить линейную комбинацию, равную $a_j$? Так как мы взяли остовное дерево, то между $a_i$ и $a_j$ в этом дереве есть путь (все, что нам надо от дерева — это связность) $b_1, b_2, \ldots, b_l$, где $b_1 = a_i$ и $b_l = a_j$. Тогда в нашем множестве есть числа $b_1 = a_i$, $b_2 - b_1$, $b_3 - b_2$, $\ldots$, $b_l - b_{l-1}$. Их сумма как раз равна $b_l = a_j$. Что и требовалось доказать. Кроме того, если бы в нашем множестве не было числа $a_i$, а были бы только разности из какого-то остовного дерева, то исходные числа мы бы получить не смогли таким образом, но разность любых двух — без проблем. Нужно опять же просуммировать все разности на каком-то пути.

Для чего мы это доказывали? Давайте хранить на отрезке отдельно какое-то любое число с этого отрезка (`any_value`), а также НОД всех попарных разностей (`diff_gcd`), который, как мы уже выяснили, равен НОДу какого-то остовного дерева. И тогда НОД на отрезке будет вычисляться просто как $gcd$(`any_value`,`diff_gcd`). Однако заметим, что если мы поддерживаем такие значения в вершине, то прибавление числа — это уже не проблема. Если ко всем числам на отрезке прибавили константу $y$, то их попарные разности не поменялись, а значит, не поменялось и `diff_gcd`. А к `any_value`, так же, как и ко всем остальным значениям на отрезке, просто прибавится эта самая константа $y$.

Осталось только понять, как пересчитывать значения `any_value` и `diff_gcd` в вершине через значения в детях. `any_value` пересчитать очень легко. Можно просто взять любое из `any_value` для левого и правого поддеревьев. А как пересчитать `diff_gcd`? В `diff_gcd` левого поддерева хранится НОД какого-то остовного дерева левого подотрезка, а в `diff_gcd` правого поддерева хранится НОД какого-то остовного дерева правого подотрезка. Поэтому все, что нам нужно, — это соединить эти два остовных дерева, то есть добавить какое-то ребро. Но у нас как раз хранятся `any_value` для обоих поддеревьев. Возьмем их разность. То есть

```
diff_gcd[v] = gcd(diff_gcd[left_child], diff_gcd[right_child],
any_value[left_child] - any_value[right_child])
```

И тогда такое решение работает за $O(n\log C + q(\log n + \log C))$. Во втором слагаемом логарифмы складываются, а не перемножаются по той же причине, по которой так происходит в дереве отрезков с поиском *gcd* на отрезке и присваиванием на отрезке. Потому что последовательное вычисление НОД у $k$ чисел, не больших $C$ по модулю, работает за $O(k + \log C)$.

**Упражнение 6.1** Докажите, что последовательное вычисление НОД $k$ чисел, не больших $C$ по модулю, работает за $O(k + \log C)$. ■

### 6.5.3 Решение + доказательство полной версии задачи

Итак, мы научились поддерживать операции `+=` и *gcd* на отрезке. Давайте добавим к этому еще `min=`.

Давайте немного изменим концепцию. Будем поддерживать на отрезке не НОД всех попарных разностей, а НОД всех попарных разностей чисел, не равных максимуму на отрезке. А максимумы (как и в Ji Driver Segment Tree) мы будем обрабатывать отдельно. Тогда при условии `tag_condition` (которое, как и `break_condition`, абсолютно такое же, как и в Ji Driver Segment Tree) мы сможем легко пересчитать значения. Нам нужно будет изменить только максимум, а `diff_gcd` никак не поменяется, потому что мы изменяем значения только максимумов, а они не входят в `diff_gcd`.

Как же нам тогда находить НОД на отрезке, зная эти значения? Все числа, не равные максимуму, уже объединены в остовное дерево внутри `diff_gcd`. Остается только присоединить максимумы (или один из них, потому что они все равно равны друг другу). В этом нам как раз может помочь `second_max`. Поэтому НОД на отрезке можно вычислить по такой формуле:

```
gcd(diff_gcd, max - second_max, max)
```

В оригинальной статье асимптотика этого алгоритма указана как $O(q\log^3 n)$. Там подразумевается, что $C = q = n$, так что в реальности такая оценка будет выглядеть как $O(n\log n\log C + q\log^2 n\log C)$.

Однако давайте улучшим эту оценку. Давайте докажем, что асимптотика на самом деле $O(n(\log n + \log C) + q\log n(\log n + \log C))$.

Почему мы не можем сказать так же, как и в облегченной версии задачи, что логарифмы складываются, а не перемножаются? Дело в том, что НОД $k$ чисел, не больших $C$ по модулю, вычисляется за $O(k + \log C)$ только в том случае, если мы вычисляем НОД последовательно. То есть сначала берем НОД двух чисел, потом берем НОД этого НОДа и следующего числа, и так далее. Именно так работает обычное дерево отрезков: в нем мы спускаемся вниз по левой и правой границам отрезка, то есть по бамбукам, в которых будет последовательно вычисляться НОД. Однако в Segment Tree Beats мы посещаем большое количество дополнительных вершин, которые вовсе не образуют бамбуки, так что логарифмы будут перемножаться. Но давайте копнем глубже.

Давайте вспомним, как мы оценивали асимптотику в задаче `min=`, `+=`. Мы ввели потенциал, равный количеству помеченных вершин в дереве, то есть тех вершин, у которых максимум в левом поддереве не совпадает с максимумом в правом поддереве. Мы доказали, что суммарно за все время он может увеличиться максимум на $O(n + q\log n)$, а также на каждые $O(\log n)$ посещенных дополнительных вершин этот потенциал уменьшается на 1, поэтому асимптотика будет $O(n\log n + q\log^2 n)$. Там мы показали, что в поддереве любой дополнительной вершины есть метка, которая удалится, поэтому если удалилось $k$ меток, то было посещено не более $k\log n$ вершин. Однако нас интересуют не все вершины, а те, в которых происходит разветвление. Ведь в

бамбуке, как мы уже поняли, НОД вычисляется быстро. Но ведь все дополнительные вершины как раз таки лежат на $k$ путях до корня от удаленных меток. И на каждом таком пути НОД будет вычисляться за $O(\log n + \log C)$, а так как всего за все время было удалено максимум $O(n + q\log n)$ меток, то асимптотика алгоритма будет равна $O(n(\log n + \log C) + q\log n(\log n + \log C))$. Что и требовалось доказать.

## 6.6 %=, = на отрезке, $\sum$

### 6.6.1 Формулировка

Простую версию этой задачи мы уже рассматривали ранее. В этой задаче у нас есть массив неотрицательных целых чисел $A$ $(0 \leqslant A_i < C)$, а также имеются запросы трех типов:

1. Даны $ql, qr, x$ $(1 \leqslant x < C)$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $A_i \bmod x$.
2. Даны $ql, qr, y$ $(0 \leqslant y < C)$. Нужно заменить все элементы массива $A$ на **полуинтервале** от $ql$ до $qr$ на $y$.
3. Даны $ql, qr$. Необходимо вернуть сумму элементов массива $A$ на полуинтервале $[ql, qr)$.

### 6.6.2 Решение

Решение ничем не отличается от решения простой версии задачи, описанного ранее [3].

### 6.6.3 Доказательство

Спасибо Антону Степанову за доказательство

Однако теперь старый потенциал, равный сумме логарифмов элементов на отрезке, больше не работает, потому что при присвоении на отрезке он может очень сильно увеличиваться. Придумаем новый потенциал и докажем, что время работы останется все тем же самым: $O((n+q)\log n \log C)$.

Для начала скажем, что вершины, на отрезке которых все числа равны, являются помеченными, а потенциал этих вершин равен нулю. В любом случае, такая вершина не может быть дополнительной, так как в ней выполнится `tag_condition`, потому что $max = min$.

Отрезок, за который отвечает любая вершина разбивается на подотрезки с одинаковыми значениями. У любой непомеченной вершины таких отрезков не меньше двух. Если раньше мы считали в потенциале сумму логарифмов всех элементов на отрезке, то теперь подотрезок одинаковых значений в потенциале будет считаться за одно значение, то есть потенциал вершины — это сумма логарифмов элементов из подотрезков равных значений, на которые разбивается отрезок текущей вершины. Разумеется, чтобы у нас не было логарифма нуля, мы будем брать логарифмы элементов, увеличенных на 1.

Как уже было сказано ранее, потенциалы помеченных вершин при этом считаются равными нулю. Потенциал всего дерева как всегда равен сумме потенциалов всех вершин.

Наш новый потенциал не больше старого, так что очевидно, что изначально $\Phi \leqslant n\log n \log C$.

При посещении тупиковых вершин либо потенциал не меняется, если выполнилось `break_condition`, либо потенциал как был нулем, так и останется, если выполнилось `tag_condition`, потому что такие вершины помечены.

[3] 6.2

При посещении дополнительных вершин в запросе первого типа не появляется никаких новых отрезков значений. Только старые значения уменьшаются и, возможно, некоторые отрезки склеиваются, так что потенциал может только уменьшаться.

Вершины, которые отвечают за отрезки, лежащие полностью внутри отрезка изменения запроса второго типа, становятся помеченными, поэтому потенциал в них становится равен нулю.

При посещении же обычных вершин потенциалы могут увеличиваться, но не очень сильно, при запросах обоих типов. Могло появиться не более трех новых отрезков элементов, каждый из которых даст вклад $O(\log C)$ в потенциал. А если учесть, что обычных вершин в каждом запросе $O(\log n)$, можно понять, что потенциал за все время увеличится максимум на $O(q \log n \log C)$.

Остается лишь показать, что при посещении дополнительной вершины потенциал уменьшается хотя бы на 1. Действительно, не выполнилось ни `break_condition`, ни `tag_condition`, так что вершина не помечена и $max \geqslant x$, поэтому на отрезке $max$ значение уменьшится хотя бы в два раза после взятия по модулю, поэтому потенциал текущей вершины уменьшится хотя бы на 1. Что и требовалось доказать.

Таким образом, асимптотика получившегося алгоритма — $O((n+q)\log n \log C)$.

## 6.7 $\sqrt{=}$, +=, $\sum$, max, min

### 6.7.1 Формулировка

В этой задаче у нас есть массив неотрицательных целых чисел $A$ $(0 \leqslant A_i < C)$, а также имеются запросы пяти типов:

1. Даны $ql, qr$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $\lfloor \sqrt{A_i} \rfloor$.
2. Даны $ql, qr, x$ $(0 \leqslant x < C)$. Нужно прибавить ко всем элементам массива $A$ на полуинтервале $[ql, qr)$ число $x$.
3. Даны $ql, qr$. Необходимо вернуть сумму элементов массива $A$ на полуинтервале $[ql, qr)$.
4. Даны $ql, qr$. Необходимо вернуть максимум элементов массива $A$ на полуинтервале $[ql, qr)$.
5. Даны $ql, qr$. Необходимо вернуть минимум элементов массива $A$ на полуинтервале $[ql, qr)$.

Эта задача доступна здесь или здесь. Это китайские сайты, поэтому там не так просто зарегистрироваться. На этих сайтах нет двух последних операций, но они поддерживаются абсолютно очевидно и все равно необходимы во время решения, так что это ничего не меняет.

### 6.7.2 Решение

Мы будем поддерживать в каждой вершине сумму, максимум и минимум. Все запросы кроме первого типа выполняются так же, как и в обычном дереве отрезков.

Кроме того, для первой операции нам понадобится еще уметь присваивать на отрезке, поэтому в каждой вершине хранится 2 `push`-а: что надо присвоить на отрезке и что надо прибавить на отрезке. При этом только один из них может быть в вершине, потому что комбинация присвоения и прибавления — это то же самое, что просто присвоение.

`break_condition` в этом случае является стандартным:

```
break_condition = qr <= l || r <= ql.
```

А каким же должно быть `tag_condition`? Первая идея, которая приходит в голову — это

`tag_condition = ql <= l && r <= qr &&` $\lfloor\sqrt{\max}\rfloor = \left\lfloor\sqrt{\min}\right\rfloor$,

потому что в этом случае корни из всех чисел на отрезке равны, и нужно просто произвести присвоение этому корню на этом отрезке. **Однако это тот случай, когда очевидный вариант не работает!** Представим себе ситуацию, в которой изначально массив имеет вид $1,2,1,2,1,2,\ldots,1,2$. После чего к нему $\frac{q}{2}$ раз применяют две операции: $+=2$ на всем массиве и $\sqrt{}=$ на всем массиве. После первой операции массив превращается в $3,4,3,4,3,4,\ldots,3,4$. А после второй он возвращается в исходное состояние. Однако заметим, что целая часть от корня из 3 — это 1, а целая часть от корня из 4 — это 2, поэтому ни в одной вершине кроме листьев `tag_condition` не выполнится, так что каждая операция $\sqrt{}=$ будет выполняться за $O(n)$, и итоговая асимптотика будет $O(q\cdot n)$. Поэтому это `tag_condition` нам не подходит.

На самом деле, правильный `tag_condition` выглядит следующим образом:

`tag_condition = ql <= l && r <= qr && max[v] - min[v] <= 1`.

Казалось бы, условие стало только слабее. Раньше мы проверяли, что корни совпадают, а теперь мы проверяем, что либо максимум равен минимуму, либо они отличаются на 1. Но здесь кроется принципиальная разница. Единственный плохой случай — это когда max — это квадрат, а минимум на 1 меньше. Тогда при взятии корня разница между ними останется равной 1, и операцией `+=` можно будет вернуть массив в исходное положение.

Как же нам обновить значение в вершине, когда выполнилось `tag_condition`? Максимум и минимум, очевидно, заменяются на свои корни. А как поменяется сумма? Если целые части корней из минимума и максимума равны, то после применения операции на отрезке все числа будут равны, поэтому нужно просто присвоить $\lfloor\sqrt{\max}\rfloor$ на отрезке. Это частный случай неправильного `tag_condition`, который мы обсуждали ранее.

Остается один случай: если целые части корней из минимума и максимума не совпадают. При этом максимум и минимум отличаются в точности на 1. Тогда и корни тоже будут отличаться в точности на 1. Максимум был равен $k^2$, а минимум $k^2-1$. При этом $k^2$ заменился на $k$, а $k^2-1$ заменился на $k-1$. Поэтому нужно просто ко всем числам на отрезке прибавить $k-k^2$.

### 6.7.3 Доказательство

Давайте докажем, что с таким `tag_condition` асимптотика будет $O(n\log C+q\log n\log C)$.

Давайте введем потенциал вершины $\varphi(v)=\log(\texttt{max[v] - min[v]}+1)$. Прибавление единицы опять же нужно только для того, чтобы не брать логарифм нуля в случае, когда максимум равен минимуму. Общим потенциалом $\Phi$ будет сумма потенциалов всех вершин дерева.

Заметим, что потенциал любой вершины неотрицателен и не превышает $\log C$, так что в любой момент времени верно $0\leqslant\Phi\leqslant O(n\log C)$.

При этом как может увеличиваться этот потенциал? В запросах типа `get` элементы массива не меняются, так что потенциал тоже не меняется. Для запросов второго типа потенциал вершины мог поменяться только в том случае, если данная вершина пересекается с отрезком запроса, но не лежит в нем полностью, потому что если она лежит в нем полностью, то и к минимуму, и к максимуму прибавится $x$, так что разность не поменяется. А вершин, которые пересекаются с отрезком запроса, но не лежат в нем полностью, $O(\log n)$ штук, так что потенциал может увеличиться максимум на $O(q\log n\log C)$ за все время.

Аналогично, операция $\sqrt{}=$ не может увеличить потенциал, если вершина полностью лежит в отрезке запроса, потому что разность корней не больше разности исходных чисел (это легко проверить, но дальше мы докажем даже более сильное условие). А вершин, которые пересекаются с запросом, но не лежат в нем полностью, опять же $O(\log n)$ штук, так что опять же увеличение за все время — это $O(q \log n \log C)$.

Осталось понять, что при посещении дополнительной вершины потенциал уменьшается хотя бы на $\log(1.5)$ (не пугайтесь этого числа, на самом деле неважно, чему оно равно, главное, что это положительная константа), тогда итоговая асимптотика будет равна $O(n \log C + q \log n \log C)$. В этом нам поможет следующий факт:

**Теорема 6.7.1** Если $a$ и $b$ — неотрицательные целые числа, и $a \geqslant b+2$, то $a-b \geqslant 1.5 \cdot \left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor\right) + 0.5$.

*Доказательство.* Обозначим $\left\lfloor\sqrt{b}\right\rfloor = m$ и $\lfloor\sqrt{a}\rfloor = n+m$. При этом $n \geqslant 0$, потому что $a > b$. Тогда $b = m^2 + l$, где $0 \leqslant l \leqslant 2 \cdot m$ и $a = (n+m)^2 + k$, где $0 \leqslant k \leqslant 2 \cdot (n+m)$. Разберем два случая:

1. $n \leqslant 1$. То есть $\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor \leqslant 1$, так что неравенство, которое нам надо доказать, превращается в $a-b \geqslant 1.5 \cdot 1 + 0.5 = 2$. Это верно из-за условия на то, что $a \geqslant b+2$.
2. $n \geqslant 2$. В этом случае мы знаем, что $a \geqslant (n+m)^2$ и $b \leqslant m^2 + 2m$, поэтому $a-b \geqslant (n+m)^2 - (m^2+2m) = n^2 + 2nm - 2m = n^2 + 2m \cdot (n-1) \geqslant n^2$. Последнее неравенство верно, потому что все множители во втором слагаемом неотрицательны. Нам нужно доказать, что это не меньше, чем $1.5 \cdot n + 0.5$. Это легко проверить, потому что $n^2 \geqslant 2n = 1.5n + 0.5n \geqslant 1.5n + 0.5$. Первое неравенство верно из-за того, что $n \geqslant 2$, а второе из-за того, что $n \geqslant 1$. Что и требовалось доказать.

■

Давайте немного преобразуем получившееся неравенство. Прибавим к обеим частям 1:

$$a - b + 1 \geqslant 1.5 \left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor + 1\right)$$

И возьмем логарифмы от обеих частей:

$$\log(a-b+1) \geqslant \log\left(1.5\left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor + 1\right)\right) = \log(1.5) + \log\left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor + 1\right).$$

То есть мы доказали, что потенциал уменьшился хотя бы на $\log(1.5)$, и асимптотика решения доказана.

**Замечание 6.7.2** Обратите внимание, что операция присвоения на отрезке будет менять потенциалы так же несильно, как и операция прибавления на отрезке, так что ее мы тоже можем поддерживать. Кроме того, мы и так присваиваем на отрезке в одном из случаев, когда берем корень, поэтому особо ничего нового к решению добавлять не придется.

## 6.8 /=, +=, ∑, max, min

### 6.8.1 Формулировка

В этой задаче у нас есть массив неотрицательных целых чисел $A$ ($0 \leqslant A_i < C$), а также имеются запросы пяти типов:

1. Даны $ql, qr, x$ ($x \geqslant 1$). Нужно поделить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $x$ нацело.

2. Даны $ql, qr, y$ $(0 \leqslant y < C)$. Нужно прибавить ко всем элементам массива $A$ на полуинтервале $[ql, qr)$ число $y$.
3. Даны $ql, qr$. Необходимо вернуть сумму элементов массива $A$ на полуинтервале $[ql, qr)$.
4. Даны $ql, qr$. Необходимо вернуть максимум элементов массива $A$ на полуинтервале $[ql, qr)$.
5. Даны $ql, qr$. Необходимо вернуть минимум элементов массива $A$ на полуинтервале $[ql, qr)$.

### 6.8.2 Решение

Как бы это ни было удивительно, но решение абсолютно идентично решению предыдущей задачи.

```
break_condition = qr <= l || r <= ql
tag_condition = ql <= l && r <= qr && max[v] - min[v] <= 1
```

Есть только одно новое условие: если мы пытаемся поделить на отрезке на единицу, то мы просто проигнорируем этот запрос, потому что деление на 1 не меняет элементов.

И в случае, если `tag_condition` выполнилось, мы делаем точно такие же изменения, как и для корня. Если $\left\lfloor \frac{\max}{x} \right\rfloor = \left\lfloor \frac{\min}{x} \right\rfloor$, то нужно всем числам на отрезке присвоить $\left\lfloor \frac{\max}{x} \right\rfloor$, а если $\left\lfloor \frac{\max}{x} \right\rfloor \neq \left\lfloor \frac{\min}{x} \right\rfloor$, то тогда $\max = kx$ для некоторого $k$ и $\min = kx - 1$. При этом $\left\lfloor \frac{\max}{x} \right\rfloor = k$ и $\left\lfloor \frac{\min}{x} \right\rfloor = k - 1$, так что надо просто ко всем числам на отрезке прибавить $k - kx$.

### 6.8.3 Доказательство

Почему же такое решение будет работать?

Проведем абсолютно такое же доказательство с таким же потенциалом. Для всех операций кроме первой ничего не поменялось. Для первой операции опять же потенциал вершины не мог увеличиться, если вершина полностью лежит в отрезке запроса, потому что после деления разность максимума и минимума не могла увеличиться (это несложно проверить, но далее мы докажем более сильное условие).

Так что все, что нам надо доказать, — это то, что при посещении дополнительной вершины потенциал уменьшится хотя бы на $\log(\frac{4}{3})$ (как и раньше, просто положительная константа), и тогда асимптотика будет равна $O(n \log C + q \log n \log C)$. В этот момент нам как раз пригодится то, что мы игнорируем запросы, в которых $x = 1$, потому что в этом случае никакие потенциалы не меняются, и мы бы просто делали лишние действия.

**Теорема 6.8.1** Если $a$, $b$ и $x$ — неотрицательные целые числа, при этом $a \geqslant b + 2$ и $x \geqslant 2$, то $a - b \geqslant \frac{4}{3} \cdot \left( \left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor \right) + \frac{1}{3}$.

*Доказательство.* Обозначим $\left\lfloor \frac{b}{x} \right\rfloor = m$ и $\left\lfloor \frac{a}{x} \right\rfloor = n + m$. При этом $n \geqslant 0$, потому что $a > b$. Тогда $b = mx + l$, где $0 \leqslant l < x$ и $a = (n + m) \cdot x + k$, где $0 \leqslant k < x$. Разберем два случая:

1. $n \leqslant 1$. То есть $\left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor \leqslant 1$, так что неравенство, которое нам надо доказать, превращается в $a - b \geqslant \frac{4}{3} \cdot 1 + \frac{1}{3} = \frac{5}{3}$. Это верно из-за условия на то, что $a \geqslant b + 2$.
2. $n \geqslant 2$. В этом случае мы знаем, что $a \geqslant (n + m) \cdot x$ и $b \leqslant mx + x - 1$, поэтому $a - b \geqslant (n + m) \cdot x - (mx + x - 1) = nx - x + 1$. И мы хотим доказать, что это не меньше, чем $\frac{4}{3}n + \frac{1}{3}$. Перенесем все из правой части в левую, а $x$ перенесем в правую:

$$n \cdot \left( x - \frac{4}{3} \right) + \frac{2}{3} \geqslant x$$

$n \geqslant 2$, поэтому заменим в левой части:

$$2 \cdot \left(x - \frac{4}{3}\right) + \frac{2}{3} \geqslant x$$

Если раскрыть левую часть, получится:

$$2x - 2 \geqslant x$$

Это верно в силу того, что $x \geqslant 2$. Что и требовалось доказать.

Давайте немного преобразуем получившееся неравенство. Прибавим к обеим частям 1:

$$a - b + 1 \geqslant \frac{4}{3}\left(\left\lfloor\frac{a}{x}\right\rfloor - \left\lfloor\frac{b}{x}\right\rfloor + 1\right)$$

И возьмем логарифмы от обеих частей:
$\log(a-b+1) \geqslant \log\left(\frac{4}{3}\left(\left\lfloor\frac{a}{x}\right\rfloor - \left\lfloor\frac{b}{x}\right\rfloor + 1\right)\right) = \log\left(\frac{4}{3}\right) + \log\left(\left\lfloor\frac{a}{x}\right\rfloor - \left\lfloor\frac{b}{x}\right\rfloor + 1\right)$.
То есть мы доказали, что потенциал уменьшился хотя бы на $\log(\frac{4}{3})$, и асимптотика решения доказана.

■

## 6.9 &=, |=, max

### 6.9.1 Формулировка

В этой задаче у нас есть массив целых чисел $A$ $(0 \leqslant A_i < C)$, а также имеются запросы трех типов:

1. Даны $ql, qr, x$ $(0 \leqslant x < C)$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $A_i \& x$ (побитовое «и»).
2. Даны $ql, qr, x$ $(0 \leqslant y < C)$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $A_i | x$ (побитовое «или»).
3. Даны $ql, qr$. Необходимо вернуть максимум элементов массива $A$ на полуинтервале $[ql, qr)$.

В данном случае стоит считать, что $C = 2^k$ для некоторого натурального $k$, то есть мы работаем с $k$-битными числами.

Эта задача доступна здесь.

### 6.9.2 Решение

Давайте заметим, что & — это то же самое, что побитовый минимум, а | — побитовый максимум, так что эта задача — это в каком-то смысле вариация Ji Driver Segment Tree для битовых операций.

Давайте хранить в каждой вершине дерева максимум на соответствующем подотрезке, побитовое «или», побитовое «и», а также ленивые обновления типа `pushAnd` и `pushOr`, которые означают, что ко всем числам на отрезке нужно применить операцию «и», а также «или». Можно заметить, что даже если к нам приходят много операций первых двух типов, мы все равно можем скомбинировать их в два таких значения, которые нужно проталкивать.

Теперь осталось понять, как это все считать, и какие будут `break_condition` и `tag_condition`. В данном случае нестандартным будет только `tag_condition`. Почему нам вообще не подходит обычное дерево отрезков? Потому что если ко всем числам на отрезке применить запрос одного из первых двух типов, то их относительных порядок может сильно поменяться, а значит, поменяется и максимум. Но в каком

случае относительный порядок не поменяется? Посмотрим для примера на первую операцию. Какими свойствами должны обладать числа $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_k$, чтобы при этом выполнялось $a_1 \& x \leqslant a_2 \& x \leqslant \ldots \leqslant a_k \& x$? Те биты, которые в $x$ установлены в единицу, не поменяются в числах. Поменяются только те биты, которые у $a_i$ установлены в 1, а у $x$ — в 0. И если у какого-то $a_i$ была установлена 1, а у другого 0, то их относительный порядок мог поменяться. Но если на всех позициях, в которых у числа $x$ стоит ноль, у всех $a_i$ стоят одинаковые биты, то относительный порядок никак не изменится. Аналогичное утверждение можно сформулировать для операции «или»: если на всех позициях, в которых у числа $y$ стоит единица, у всех $a_i$ выставлены одинаковые биты, то их относительный порядок не изменится.

А проверить, что на этих позициях у всех чисел стоят одинаковые биты, очень легко. Достаточно посмотреть на побитовое «и» и побитовое «или» всех чисел на отрезке. Тогда если какой-то бит у них совпадает, это как раз равносильно тому, что у всех чисел на отрезке этот бит совпадает. Так что `tag_condition` для первой и второй операции соответственно будут выглядеть следующим образом:

```
tag_condition_and = ql <= l && r <= qr && ((and[v] ^ or[v]) & ~x) == 0
tag_condition_or = ql <= l && r <= qr && ((and[v] ^ or[v]) & y) == 0
```

**Упражнение 6.2** Хорошим упражнением будет понять, почему эта битовая магия соответствует именно тем условиям, которые мы описали ранее. ■

### 6.9.3 Доказательство

Почему же такое решение будет работать быстро?

Давайте введем потенциал $\varphi(v)$ вершины $v$ дерева отрезков, который будет равен количеству битов, в которых не все числа на соответствующем отрезке совпадают, то есть, иными словами, `__builtin_popcount(and[v] ^ or[v])`. Потенциал всего дерева $\Phi$, как обычно, будет равен сумме потенциалов всех вершин.

Заметим, что $0 \leqslant \varphi(v) \leqslant \log C$, потому что у нас есть всего $\log C$ битов, так что отличающихся не больше. Поэтому в любой момент времени $0 \leqslant \Phi \leqslant n \log C$.

В каких случаях потенциал может увеличиваться? Операция третьего типа не меняет элементов массива, так что потенциал тоже не меняется. Если у всех чисел на каком-то отрезке какой-то бит был равен, и мы применили ко всем этим числам операцию одного из первых двух типов, то этот бит не мог перестать быть равен. Так что увеличения потенциала могли происходить только в вершинах, подотрезок которых пересекается, но не лежит полностью в отрезке запроса, то есть в обычных вершинах. При этом для каждой такой вершины потенциал мог увеличиться максимум на $\log C$, а для каждого запроса таких вершин $O(\log n)$, так что можно сказать, что $\Phi_+ \leqslant O(q \log n \log C)$.

Теперь поймем, почему при посещении дополнительных вершин потенциал уменьшается. Если мы посетили какую-то дополнительную вершину, то существует такой бит, что на данный момент не у всех чисел он одинаковый, и при этом у $x$ он выставлен в 0 (либо у $y$ в 1 для запросов второго типа), то есть после применения текущего запроса у всех чисел этот бит будет равен нулю (или единице для запросов второго типа). Так что потенциал текущей вершины уменьшится как минимум на 1. Что и требовалось доказать.

Поэтому асимптотика получившегося алгоритма — $O(n \log C + q \log n \log C)$.

## 6.10 `min=`, `+=`, max over $\sum$ для нескольких массивов параллельно

### 6.10.1 Формулировка

В этой задаче у нас есть сразу **два массива** целых чисел $A$ и $B$, а также имеются запросы пяти типов:

1. Даны $ql, qr, x$. Нужно заменить все элементы массива $a$ на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Аналогичный запрос для массива $B$.
3. Даны $ql, qr, y$. Нужно прибавить ко всем элементам массива $A$ на полуинтервале $[ql, qr)$ число $y$.
4. Аналогичный запрос для массива $B$.
5. Даны $ql, qr$. Необходимо вернуть $\max_{ql \leqslant i < qr} A_i + B_i$.

Последний запрос как раз называется max over $\sum$, то есть максимум от поэлементной суммы двух массивов.

Также в конце решения мы покажем, как это обобщается на случай бóльшего количества массивов.

### 6.10.2 Решение + доказательство

Если бы у нас не было операции `min=`, то эта задача была бы весьма простой. Мы бы пользовались обычным деревом отрезков, и если мы на отрезке прибавляем число в одном из массивов, то к максимуму поэлементной суммы массивов тоже прибавится это самое число. Иными словами, можно считать, что у нас есть всего один массив $C$, определенный по правилу $C_i = A_i + B_i$, и операции производятся с ним.

Однако, если у нас есть операция `min=`, то нам важны значения каждого из массивов по отдельности.

Мы воспользуемся стандартной техникой из Ji Driver Segment Tree, то есть будем хранить максимум, их количество и второй максимум. Тогда при условии `tag_condition` мы меняем в массиве какой-то разрозненный набор позиций, на которых стоят максимумы. Как в таком случае пересчитать максимум поэлементной суммы массивов? Давайте разделим позиции на подотрезке, которому соответсвует текущая вершина, на 4 типа:

1. В обоих массивах на этой позиции стоят максимумы на отрезке
2. В первом массиве на этой позиции стоит максимум на отрезке, а во втором — нет
3. Во втором массиве на этой позиции стоит максимум на отрезке, а в первом — нет
4. В обоих массивах на этой позиции стоят не максимумы на отрезке

И после такого разделения выясняется, что хранить максимум сумм для каждого из четырех типов по отдельности оказывается очень просто. Если выполнилось `tag_condition`, то нужно поменять значения, соответствующие тем типам, в которых в этом массиве стоит максимум. При этом значение максимума изменилось с `max` на $x$, поэтому из соответствующих значений как раз надо отнять `max`$-x$. Ответом на запрос тогда будет просто максимум из четырех отдельных максимумов для каждого из типов.

Осталось научиться пересчитывать значения в вершине через значения в детях. Но это тоже делается очень просто. Мы сначала пересчитываем максимумы в обоих массивах в текущей вершине, после чего для каждого типа каждого из детей понимаем, стоят ли там максимумы или нет. Если там был не максимум для сына, то это будет не максимум и для отца; а если это был максимум для сына, то он мог как остаться максимумом, так и перестать им быть.

Решение ничем не отличается от задачи `max=`, `+=`, мы просто поддерживаем в вершине четыре дополнительных значения. Поэтому асимптотика получается $O(n \log n + q \log^2 n)$.

**Замечание 6.10.1** Если подумать, можно заметить, что эта идея легко обобщается на большее количество массивов. Если у нас $k$ массивов, то мы будем поддерживать $2^k$ дополнительных величин в каждой вершине, потому что в каждом массиве на каждой позиции может стоять либо максимум, либо не максимум (2 варианта), и можно брать всевозможные комбинации максимумов и не максимумов для разных массивов. Получается всего $2^k$ различных комбинаций. И асимптотика тогда возрастет до $O((n\log n + q\log^2 n)\cdot 2^k)$.

## 6.11 Историческая информация: количество изменений элемента

### 6.11.1 Формулировка

В этой задаче у нас есть массив целых чисел $A$, а также имеются запросы четырех типов:

1. Даны $ql, qr, x$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны $ql, qr, y$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $\max(A_i, y)$.
3. Даны $ql, qr, z$. Нужно прибавить ко всем элементам массива $A$ на полуинтервале $[ql, qr)$ число $z$.
4. Даны $ql, qr$. Необходимо вернуть сумму элементов массива $S$ на полуинтервале $[ql, qr)$, где $S_i$ — это количество раз, когда $i$-й элемент массива $A$ менялся.

### 6.11.2 Решение + доказательство

Эта задача — первый базовый пример того, что такое «историческая информация». Обычно мы делаем запросы к элементам массива на данный момент, и мы не заботимся о том, как текущий массив был получен. Историческая информация же хранит в себе знания о том, как наш массив менялся с течением времени, а не только его финальный вид.

Однако нет ничего сложного в том, чтобы поддерживать эту информацию. Если бы у нас была только операция `+=`, то каждый раз, когда массив $A$ меняют на полуинтервале $[ql, qr)$, к массиву $S$ надо было бы на этом полуинтервале просто прибавить 1, потому что при каждой операции меняются абсолютно все элементы на подотрезке. Однако когда появляется операция `min=`, это уже неверно. Но все еще мы можем понять, сколько элементов поменялось. Мы останавливаем рекурсию в тот момент, когда выполнилось `tag_condition`. Это значит, что меняются только максимумы. Поэтому изменится ровно `cnt_max` элементов. На асимптотику алгоритма это никак влиять не будет, она все еще будет равна $O(n\log n + q\log^2 n)$. Стоит помнить о том, что если мы храним отложенную информацию о двух операциях `min=`, то элементы ниже поменялись дважды.

Далее мы рассмотрим более сложные версии исторической информации.

## 6.12 Историческая информация: исторический максимум

### 6.12.1 Формулировка

В этой задаче у нас есть массив целых чисел $A$, а также имеются запросы трех типов:

1. Даны $ql, qr, x$. Нужно заменить все элементы массива $A$ на полуинтервале $[ql, qr)$ на $\min(A_i, x)$.
2. Даны $ql, qr, y$. Нужно прибавить ко всем элементам массива $A$ на полуинтервале $[ql, qr)$ число $y$.

3. Даны $ql, qr$. Необходимо вернуть сумму элементов массива $M$ на полуинтервале $[ql, qr)$, где $M_i$ — это исторический максимум $A_i$, то есть самое большое значение, которое хранилось в $A_i$ за все время.

Эта задача доступна здесь в немного ином виде. Там вместо максимумов минимумы, и третий запрос берет не сумму элементов массива $M$, а максимум.

Также похожая задача есть здесь.

### 6.12.2 Решение + доказательство

Первые две операции мы поддерживаем как обычно. И асимптотика получившегося алгоритма будет опять же $O(n \log n + q \log^2 n)$. Остается понять, как поддерживать сумму на отрезке массива $M$.

Давайте введем дополнительный массив $D$, элементы которого будут определяться по формуле $D_i = M_i - A_i$. А массив $M$ мы на самом деле не будем нигде поддерживать. Действительно, заметим, что

$$\sum_{ql \leqslant i < qr} M_i = \sum_{ql \leqslant i < qr} ((M_i - A_i) + A_i) = \sum_{ql \leqslant i < qr} (D_i + A_i) = \sum_{ql \leqslant i < qr} D_i + \sum_{ql \leqslant i < qr} A_i$$

Поэтому нам нужно уметь отдельно поддерживать сумму на отрезках массивов $A$ и $D$, и с их помощью мы сможем находить сумму на отрезке массива $M$. В этом и заключается основная идея исторического максимума. Но почему пересчитывать массив $D$ проще, чем массив $M$?

Давайте сначала рассмотрим, что произойдет с массивом $D$ после операции второго типа. К элементам массива $A$ прибавится $x$, поэтому из $D$ нужно вычесть $x$. Однако если $A_i$ стало новым историческим максимумом, то $M_i$ тоже изменится, и станет равно $A_i$. Это происходит ровно в тот момент, когда $M_i - A_i$ становится отрицательным, и в этот момент нужно заменить $D_i$ на ноль. То есть на самом деле $D_i$ заменяется на $\max(D_i - x, 0)$. А эту операцию можно разделить на две: сначала вычесть на отрезке, а потом применить `max=` на отрезке с нулем. Это все мы умеем делать.

Теперь разберемся, что происходит с массивом $D$ при операции `min=`. В этой операции элементы могут только убывать, поэтому $M_i$ точно не поменяются. А $A_i$ поменяются весьма понятным образом. Когда мы пришли в вершину, в которой выполнилось `tag_condition`, $D_i$ поменяется точно так же, как и $A_i$, а далее это изменение надо будет просто протолкнуть в детей.

## 6.13 Историческая информация: историческая сумма

### 6.13.1 Формулировка

В этой задаче у нас есть массив целых чисел $A$, а также имеются запросы двух типов:

1. Даны $ql, qr, x$. Нужно прибавить ко всем элементам массива $A$ на полуинтервале $[ql, qr)$ число $x$.
2. Даны $ql, qr$. Необходимо вернуть сумму элементов массива $S$ на полуинтервале $[ql, qr)$, где $S_i$ — сумма элементов, стоящих на позиции $i$ в массиве $A$ за все время. То есть после каждой операции к $S_i$ прибавляется $A_i$ для всех позиций $i$.

### 6.13.2 Решение + доказательство

Эта задача похожа на предыдущую, но на этот раз вместо исторического максимума мы поддерживаем историческую сумму. В этой задаче нам хватит на самом деле даже обычного дерева отрезков.

Введем дополнительный массив $D$, элементы которого будут определяться по формуле $D_i = S_i - ind \cdot A_i$, где $ind$ — это количество операций, которые уже были произведены с массивом на данный момент (после каждой операции $ind$ увеличивается на 1).

Опять же, сам массив $S$ мы нигде поддерживать не будем, потому что

$$\sum_{ql \leqslant i < r} S_i = \sum_{ql \leqslant i < r} (D_i + ind \cdot A_i) = \sum_{ql \leqslant i < r} D_i + ind \cdot \sum_{ql \leqslant i < r} A_i$$

Так что поддерживая сумму на отрезке в массивах $D$ и $A$, мы сможем вычислять сумму на отрезке в массиве $S$. Посмотрим, как меняются элементы массива $D$ при операции первого типа. Пускай до этой операции у нас были массивы $S$, $A$ и $D$, а после применения операции они превратились в массивы $S'$, $A'$ и $D'$. Мы хотим научиться вычислять $D'$. Мы знаем, что $D_i = S_i - ind \cdot A_i$ и $D'_i = S'_i - (ind + 1) \cdot A'_i$, потому что после применения операции $ind$ увеличился на 1. Давайте раскроем последнюю формулу:

$$D'_i = S'_i - (ind + 1) \cdot A'_i = (S_i + A'_i) - (ind + 1) \cdot A'_i =$$

$$= S_i - ind \cdot A'_i = S_i - ind \cdot (A_i + x) = S_i - ind \cdot A_i - ind \cdot x = D_i - ind \cdot x$$

Так что у массиву $D$ тоже просто прибавляется константа $(-ind \cdot x)$ на отрезке. Для элементов, которые не меняются, к $S_i$ прибавится $A_i$ и $ind$ увеличится на 1, поэтому $D_i$ никак не поменяются.

## 6.14 Задачи для практики

- Первая задача из этой главы. $\%=$ на отрезке, присвоение в точке и поиск суммы на отрезке.
- Ji Driver Segment Tree ($\min=$ на отрезке и поиск суммы на отрезке) можно решить здесь и здесь.
- Кроме того есть не совсем очевидное применение Ji Driver Segment Tree в задаче «Нагайна».
- Задачу с операцией $\sqrt{}=$ на отрезке можно решить здесь или здесь.
- Задачу с операцией $/=$ на отрезке можно решить здесь.
- Задача с операциями $\&=$ и $|=$ на отрезке и поиском максимума на отрезке доступна здесь.
- Задача на сумму исторических максимумов доступна здесь. Также есть похожая задача здесь.
- Кроме того можете прорешать специально подготовленный контест на codeforces. Если у вас нет доступа к соревнованию, нужно сначала вступить в группу.