Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Егор Андреевич Горбачев

Выпускная квалификационная работа

Улучшение оценок времени работы алгоритмов Segment Tree Beats

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2018 «Прикладная математика, фундаментальная информатика и программирование»
Профиль «Современное программирование»

Научный руководитель:
д.ф.-м.н., профессор СПбГУ А. С. Куликов

Рецензент:
к.т.н., доцент ИТМО А. С. Станкевич

Санкт-Петербург
2022 г.

Содержание

Введение

Задача поиска значения какой-то функции на отрезке массива является одной из наиболее стандартных и широко изученных проблем в теоретической информатике. Одна из самых известных работ в этой области — статья Фараха-Колтона и Бендера [1], решающая задачу поиска минимума на отрезке за оптимальное время. Кроме того, во многих контекстах бывает необходимо решать так называемые «динамические» версии этих задач, то есть при условии, что элементы массива могут меняться со временем. Широко исследованы такие запросы изменения как прибавление константы ко всем элементам на отрезке и присвоение какой-то константы всем элементам на отрезке. Для решения этих задач применяются такие стандартные структуры данных как дерево отрезков и дерево Фенвика [2]. Однако какие-либо запросы изменения кроме этих двух стандартных были всегда обделены вниманием из-за того, что не удавалось придумать алгоритмов, которые бы работали столь же быстро, как и обычные деревья отрезков.

Большой шаг в эту стороны был сделан Ruyi Ji в 2016 году в его полунаучной статье [3]. К сожалению, в полном виде этот текст доступен только на китайском языке и с достаточно большим количеством неформальностей и допущений. Цель моей работы — воспроизведение результатов Ruyi Ji, их формализация, а также улучшение полученных оценок в уже рассмотренных задачах и применение схожих методов для решения новых задач. Центральной темой моей работы является так называемая задача «$\min =$»: запрос изменения, заменяющий все элементы на отрезке на минимум из текущего значения и константы. Этот запрос, пожалуй, является наиболее естественным запросом изменения кроме стандартных: прибавления и присвоения на отрезке. Кроме того, в работе я не только исследовал теоретическую сложность, но и реализовал и протестировал решения всех исследуемых задач на языке программирования C++. Также, в этой работе я привожу сценарии использования проанализированных мною алгоритмов для решения задач из реальной жизни: к примеру, поддержка актуальной стоимости ценной бумаги на фондовом рынке при появляющихся и пропадающих предложениях.

Постановка задачи

Цель данной работы — анализ и формализация существующих алгоритмов из семейства Segment Tree Beats, улучшение оценок их времени работы, а также применение полученных техник для решения новых задач.

Задачи работы:

1. Изучение существующих наработок по Segment Tree Beats на китайском языке.
2. Воспроизведение, формализация и упрощение уже полученных результатов.
3. Улучшение оценок времени работы уже известных алгоритмов (Extended Ji Driver Segment Tree, GCD Ji Driver Segment Tree и другие).
4. Реализация полученных алгоритмов.

1. Обзор литературы

Впервые идея структуры данных Segment Tree Beats была сформулирована в работе Ruyi Ji [3]. Первое известное мне упоминание на английском языке относится к 2017 году [4]. Позже Ruyi Ji перевел часть своей статьи на английский язык [5] в начале 2018 года. После этого популярность этой структуры за пределами китаеязычного сообщества выросла в разы, и можно сказать, что это стало общеизвестной структурой данных. После чего большое количество статей [6], [7], [8] и видеороликов [9], [10], [11], [12], [13] были записаны на эту тему. Однако никаких новых продвижений за все это время не было сделано. Было представлено несколько новых задач, которые решаются теми же методами [14], [15], [16], [17], [18], однако старые открытые вопросы из оригинальной статьи Ruyi Ji так и остались неотвеченными.

2. Постановка решаемой задачи

Поставим решаемую задачу более формально. В этой работе мы будем решать задачу поиска значения какой-то функции на отрезке массива. Изначально нам дан массив целых чисел длины n. Его можно как-то предобработать, после чего необходимо ответить на q запросов. Запросы могут быть двух типов:

1. **Запрос обновления.** В этом запросе нам обычно даны три целых числа: l, r и x. Необходимо обновить все элементы массива с индексами от l до r[1] константой x. То, как именно нужно обновлять эти элементы, зависит от конкретной задачи. К примеру: прибавить x ко всем элементам на отрезке массива (запрос "$+=$"), присвоить x всем элементам на отрезке массива (запрос "$=$"), заменить все элементы на отрезке массива минимумом (или максимумом) из текущего значения и x (запрос "$\min=$" (или "$\max=$")), заменить все элементы на отрезке массива остатком от деления текущего значения на x (запрос "$\bmod=$"), заменить все элементы на отрезке массива целой частью от деления текущего значения на x (запрос "$/=$"), и так далее.

2. **Запрос поиска.** В этом запросе нам обычно даны два целых числа: l и r. Нам необходимо найти значение какой-то функции, аргументами которой служат все элементы массива с индексами от l до r. К примеру, найти сумму всех элементов на отрезке массива (запрос "$\sum$"), найти минимум (или максимум) из всех элементов на отрезке массива (запрос "$\min$" (или "$\max$")), найти наибольший общий делитель всех элементов на отрезке массива (запрос "$\gcd$"), и так далее.

Подобные задачи можно решать за время $O(q \cdot n)$, просто выполняя действия с массивом, которые от нас просят, непосредственно. Очевидно, мы хотим научиться решать быстрее. Деревья отрезков позволяют решать эти задачи за время $O(n+q\log n)$ ($O(n\log C+q(\log n+\log C))$ для запроса $\gcd$) для

[1]Всегда далее, когда мы говорим про какие-то подотрезки значений массива, мы подразумеваем, что речь идет про полуинтервалы, то есть левая граница берется включительно, а правая не включительно.

запросов обновления $=$ и $+=$, однако они не способны поддерживать другие запросы обновления, потому что когда весь отрезок какой-то вершины дерева отрезков обновляется, мы должны быть в состоянии быстро пересчитать значения необходимой нам функции на этом отрезке. Это легко сделать, к примеру, для функции $\sum$, если мы прибавляем константу на отрезке или присваиваем константу на отрезке, однако это не представляется возможным для таких запросов обновления как $\min =$ и $\bmod =$[2]. Насколько мне известно, без использования представленных в этой работе алгоритмов, лучшие решения для подобных запросов основаны на идее корневой декомпозиции и работают за время порядка $O(\sqrt{n})$ на запрос.

Эта работа направлена на изучение алгоритмов из семейства "Segment Tree Beats", которые в некотором смысле являются обобщением обычных деревьев отрезков. Этот термин впервые был введен Ruyi Ji в 2016 году в его полунаучной статье [3] (в 2018 году в сильно урезанном формате была переведена на английский язык: [5]). В этой статье ему удалось решить задачу с запросами $\min =$ и $\sum$ за время $O((n+q)\log n)$, задачу с запросами $\min =$, $+=$ и $\sum$ за время $O((n+q)\log^2 n)$ и задачу с запросами $\min =$, $+=$, $\sum$ и $\gcd$ за время $O(n \log n \log C + q \log^2 n \log C)$[3]. К сожалению, эта статья доступна только на китайском языке, и в ней часто не достает математической формальности, что привело, в том числе к недостаточно формальным доказательствам (иногда, возможно, даже неверным) и не совсем корректным асимптотикам. В своей работе я полностью формально воссоздал весь прогресс, сделанный Ruyi Ji, улучшил оценки на время работы некоторых уже представленных алгоритмов и применил идеи Segment Tree Beats для решения нескольких новых задач.

[2]Некоторые ограниченные версии задач все еще на самом деле можно решать при помощи обычных деревьев отрезков. К примеру, операцию изменения $\min =$ можно без труда поддерживать, если в запросах поиска нас интересует только нахождение минимума или максимума. Однако нас все таки интересуют более «полные» структуры данных, то есть те, которые могут поддерживать сразу все стандартные запросы поиска для данных запросов обновления.

[3]В оригинальной статье указано время $O(q \log^3 n)$, что верно, если подразумевать $q = n = C$. Здесь я представил более аккуратно уже доказанные асимптотики.

Algorithm 1 Basic segment tree update query with lazy propogation.

```
1:  function Update(v, l, r, ql, qr, newVal)
2:      if qr ≤ l or r ≤ ql then                    ▷ 1. node is outside of the segment
3:          return
4:      else if ql ≤ l and r ≤ qr then              ▷ 2. node is inside the segment
5:          UpdateNode(v, newVal)
6:          SetPush(v, newVal)
7:      else                                        ▷ 3. node intersects the segment
8:          PushDown(v)
9:          mid ⟵ ⌊(r+l)/2⌋
10:         Update(v_l, l, mid, ql, qr, newVal)
11:         Update(v_r, mid, r, ql, qr, newVal)
12:         PullUp(v)
```

3. Пререквизиты

Segment Tree Beats — это структура данных, в основе которой лежит дерево отрезков. Везде далее подразумевается, что читатель знаком с принципом работы стандартных деревьев отрезков, а также деревьев отрезков с массовыми (ленивыми) обновлениями. Несмотря на то, что для лучшего понимания происходящего мы освежим знания о деревьях отрезков в этом разделе, крайне рекомендуется сначала ознакомиться с какой-нибудь статьей по деревьям отрезков перед дальнейшим чтением этой работы: [19], [20].

При работе с обычными деревьями отрезков мы редко задумываемся об их глубинной структуре, потому что это не так важно для решения конкретных задач. Однако в случае с Segment Tree Beats структура дерева отрезков и то, какие именно вершины оно посещает при запросе, будет критически важно, поэтому мы освежим эти знания. Даже если вы хорошо знакомы с деревьями отрезков, не рекомендуется пропускать этот раздел.

В алгоритме 1 представлена стандартная функция для операции обновления на отрезке в дереве отрезков с массовым обновлением. Пускай мы находимся в вершине v, которая отвечает за полуинтервал $[l, r)$ массива, и нас попросили обновить значения массива на полуинтервале $[ql, qr)$ значением $newVal$.

Первое условие (*breakCondition*) проверяет, что если отрезок, за кото-

рый отвечает вершина v, не пересекается с отрезком, на котором мы делаем обновление, то в текущем поддереве ничего менять не надо, и можно просто выйти из рекурсии.

Второе условие (*tagCondition*) проверяет, что если отрезок, за который отвечает вершина v, лежит полностью внутри отрезка, который мы обновляем, то мы обновим значение сразу на данном уровне рекурсии, а также сохраним ленивое обновление, которое в будущем будем проталкивать в детей. После чего мы опять же завершаемся и выходим из рекурсии.

Если же ни первое, ни второе условие не выполнилось, 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 в них старые ленивые обновления, 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 двух вершин могут пересекаться с отрезком запроса, но при этом не лежать в нем полностью (это те вершины, отрезки которых содержат границы отрезка запроса), поэтому только из этих двух вершин мы рекурсивно запустимся на следующий уровень, а значит, на каждом уровне дерева мы посетим не более четырех вершин. Для того, чтобы лучше представить себе устройство посещаемых вершин, стоит рассмотреть листья дерева, соответствующие границам запроса. Тогда мы посетим всех их предков, образуя два вертикальных пути до корня. Кроме того, из каждой вершины на этих двух путях мы запустимся в другого ребенка (который не является продолжением вертикального пути), но сразу же завершимся. Если обстоятельства удачно сложились, может так оказаться, что один (или оба) из вертикальных путей не дойдет до нижнего уровня дерева, а завершится заранее, но это не меняет общую структуру запроса.

В любом случае те вершины, которые лежат на двух вертикальных путях, — это вершины, которые относятся к третьему случаю из алгоритма 1; вершины, располагающиеся между путями, — это вершины, относящиеся ко второму случаю; а оставшиеся вершины относятся к первому случаю. Понимание этой структуры запроса к дереву отрезков будет критически важным

при анализе алгоритмов Segment Tree Beats.

4. Основная идея

Segment Tree Beats основан на следующей идее: пускай запросы изменения таковы, что мы не всегда можем пересчитать значение на отрезке текущей вершины при условии выполнения *tagCondition* сразу. Мы не можем ничего с этим поделать, в таком случае придется рекурсивно запуститься из детей. Однако если мы будем запускаться из детей бесконечно, мы можем потенциально посетить почти все дерево, и асимптотика алгоритма будет квадратичной. Это нас совершенно не устраивает. Мы должны остановить рекурсивные запуски, как только появляется возможность быстро пересчитать значение на отрезке текущей вершины. Для этого нам необходимо усилить условие *breakCondition* и ослабить условие *tagCondition* так, чтобы при условии *breakCondition* все еще ни одно значение на отрезке данной вершины не менялось при текущем запросе изменения, а при условии *tagCondition* мы могли, не запускаясь рекурсивно, пересчитать значение в текущей вершине и оставить ленивое обновление. От того, какие условия *breakCondition* и *tagCondition* мы выберем, будет зависеть асимптотика получившегося алгоритма. Если нам повезет, мы сможем доказать, что в таком случае асимптотика будет не квадратичной.

Сформулированные идеи представлены в алгоритме 2. Как несложно заметить, эта функция отличается от стандартной функции изменения в дереве отрезков лишь добавлением *breakCondition* и *tagCondition*. На протяжении всей оставшейся статьи главный вопрос, который будет перед нами стоять, — какие именно условия *breakCondition* и *tagCondition* нужно выбрать в конкретной задаче, чтобы получить достойную асимптотику. Вся остальная работа будет касаться не придумывания самого алгоритма, а анализа его времени работы.

Замечание 1. Обратите внимание, что запросы поиска в Segment Tree Beats никак не меняются по сравнению с обычным деревом отрезков, потому что мы все еще поддерживаем инвариант, что при спуске от корня до какой-то вершины в ней всегда хранится корректное значение интересующей нас функции.

Дальнейшая структура статьи будет состоять из набора разделов, в

Algorithm 2 Segment tree beats update query.

```
1:  function UpdateSTB(v, l, r, ql, qr, newVal)
2:      if qr ≤ l or r ≤ ql or breakCondition then
3:          return
4:      else if ql ≤ l and r ≤ qr and tagCondition then
5:          UpdateNode(v, newVal)
6:          SetPush(v, newVal)
7:      else
8:          PushDown(v)
9:          mid ⟵ ⌊(r+l)/2⌋
10:         UpdateSTB(v_l, l, mid, ql, qr, newVal)
11:         UpdateSTB(v_r, mid, r, ql, qr, newVal)
12:         PullUp(v)
```

каждом из которых формулируется задача, приводится алгоритм ее решения и доказывается корректность представленного алгоритма. Для каждой из задач также есть реализация алгоритма на языке C++ в github-репозитории проекта: [21].

5. Знакомство с Segment Tree Beats на примере задачи mod $=$, $=$ в точке, $\sum$

5.1. Формулировка

В этой задаче нам дан массив целых неотрицательных чисел A ($0 \leq A_i < C$), а также имеются запросы трех типов:

1. Даны ql, qr, x ($1 \leq x < C$). Необходимо заменить i-й элемент массива A для всех i, таких что $ql \leq i < qr$, на $A_i \bmod x$.
2. Даны qi, y ($0 \leq y < C$). Необходимо заменить элемент массива A на позиции qi на y.
3. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.

Замечание 2. Как говорилось в начале статьи, нас интересуют структуры данных, которые могут поддерживать сразу весь спектр запросов поиска при данных запросах изменения, однако чаще всего мы будем рассматривать только самые интересные. Во всех случаях несложно проверить, что остальные запросы поддерживаются либо точно также, либо даже легче, чем выбранные. В частности, для того, чтобы решить эту задачу, нам все равно придется поддерживать значения минимума и максимума на отрезке.

5.2. Алгоритм

Вторую и третью операции мы будем выполнять, как и в обычном дереве отрезков. 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.

Сначала давайте подумаем, каким должно быть *breakCondition*? 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. Это и будет нашим условием

breakCondition. Для его проверки нам будет нужно поддерживать максимум на отрезке в каждой вершине дерева. Обозначим его за max_v.

Теперь надо придумать такое условие *tagCondition*, 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 и 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 max_v, поэтому операция $\bmod =$ на этом отрезке — это то же самое, что присвоить на этом отрезке всем элементам значение $max_v \bmod x$. Это мы можем сделать без труда, оставив в вершине ленивое обновление и пересчитав значение суммы. Для проверки *tagCondition* нам дополнительно нужно хранить значение минимума на отрезке каждой вершины, которое мы обозначим $\min_v$.

Наконец, последнее, что нам нужно хранить, — это непосредственно сумму на отрезке sum_v, чтобы отвечать на запросы третьего типа.

5.3. Оценка времени работы

Осталось понять, почему при таком ослаблении *tagCondition* асимптотика остается приемлемой. На самом деле асимптотика этого решения — $O((n+q)\log n \log C)$. В отличие от обычных деревьев отрезков, где каждый запрос работает за логарифмическое время, в случае с Segment Tree Beats некоторые отдельные запросы могут действительно работать линейное время. Мы будем оценивать именно суммарное время обработки q запросов, используя амортизационный анализ.

Введем потенциал вершины дерева отрезков $\varphi(v)$, равный $\sum_{l \le i < r} \log(A_i + 1)$[4][5], где l, r — границы полуинтервала исходного массива, за который отвечает данная вершина. Теперь введем потенциал всего дерева Φ, который (как и во всех последующих задачах) будет равен сумме потенциалов всех вершин дерева. В любой момент времени потенциал можно оценить следующим

[4]Прибавление единицы в данном случае не несет никакого сакрального смысла, а лишь позволяет нам избавиться от проблемы со взятием логарифма в том случае, если какие-то элементы массива станут равны нулю.

[5]Это непринципиально в случае асимптотик, но более принципиально в случае потенциалов. Везде в этом тексте под $\log$ подразумевается двоичный логарифм.

образом: $0 \leq \Phi \leq O(n \log n \log C)$, потому что потенциал каждой вершины неотрицателен, а также каждый элемент массива лежит в поддереве у $O(\log n)$ вершин дерева отрезков и дает вклад $\leq \log C$ в потенциал каждой из этих вершин.

Давайте разобьем вершины, которые посетит запрос изменения, на три вида: **обычные**, **дополнительные** и **тупиковые**. Тупиковые вершины — это ты вершины, в которых выполнилось одно из условий *breakCondition* или *tagCondition*, то есть те посещенные вершины, из которых не было сделано рекурсивных вызовов, обычные вершины — это те нетупиковые вершины, которые посетило бы стандартное дерево отрезков на данном запросе (иными словами, вершины, отрезок которых пересекает отрезок запроса, но не лежит в нем полностью), а дополнительные вершины — это все остальные вершины, которые посетил запрос, то есть те необычные вершины, из которых были сделаны рекурсивные вызовы. Тогда заметим, что некоторые верхние вершины дерева будут обычными, потом под обычными будет какое-то количество дополнительных, и из дополнительных и обычных иногда будут торчать тупиковые. Из тупиковых уже рекурсивных вызовов нет.

Время работы запроса определяется как количество посещенных вершин дерева, умноженное на время обработки каждой вершины. В большинстве случаев время обработки одной вершины — это $O(1)$, так что чаще всего мы будем отождествлять время работы запроса с количеством посещенных им вершин.

Лемма 1. *Тупиковые вершины составляют максимум две трети от посещаемых при запросе вершин, а следовательно, если мы не будем их учитывать, это не скажется на асимптотике рассматриваемых алгоритмов.*

Доказательство. Родитель любой тупиковой вершины — это либо обычная, либо дополнительная вершина. При этом у каждой обычной и дополнительной вершины не более двух детей, поэтому посещенных тупиковых вершин может быть максимум в два раза больше, чем обычных и дополнительных. Следовательно, тупиковые вершины составляют максимум две трети всех посещенных вершин. Если мы не будем их учитывать при дальнейшем анализе, то количество посещенных вершин (а значит, и время работы алгоритма)

сократится максимум в три раза, что не повлияет на асимптотику.

Иными словами, если мы посещаем тупиковую вершину, то мы делаем рекурсивный вызов и сразу завершаемся. В каком-то смысле можно считать, что этого рекурсивного вызова не было вовсе. □

Обычные же вершины — это вершины, которые были бы посещены обычным деревом отрезков, поэтому их $O(\log n)$ на каждом запросе. Они дают суммарный вклад $O(q \log n)$ во время работы алгоритма.

Таким образом, ключевой фактор, за которым нам нужно следить для анализа времени работы алгоритма, — это количество посещенных дополнительных вершин. Чуть позже мы докажем, что при посещении дополнительной вершины потенциал в этой вершине (а следовательно, и суммарный потенциал Φ) уменьшается хотя бы на 1, тогда суммарное количество посещенных дополнительных вершин можно оценить как $(\Phi_0 - \Phi_q) + \Phi_+$ (где Φ_0 — это начальный потенциал, Φ_q — это потенциал после выполнения всех запросов, а Φ_+ — это суммарное значение, на которое потенциал Φ увеличился за все время[6]). Тогда в общем случае асимптотика алгоритма составляет $O(q \log n + (\Phi_0 - \Phi_q) + \Phi_+)$ при условии, что каждая вершина обрабатывается за константное время. Во всех задачах мы будем гарантировать, что общий потенциал Φ в любой момент времени неотрицателен, а значит, неотрицательно и значение Φ_q. Тем самым это значение можно сократить, упростив асимптотику до $O(q \log n + \Phi_0 + \Phi_+)$. Остается лишь понять, какое наибольшее значение может иметь начальный потенциал, а также насколько за время всех запросов потенциал может увеличиться.

Как мы видели ранее, в данной конкретной задаче Φ всегда неотрицателен и не больше $O(n \log n \log C)$, тем самым $\Phi_0 \le O(n \log n \log C)$.

Теперь необходимо понять, чему равен Φ_+. Операция первого типа может только уменьшать числа в массиве (а значит, и потенциалы вершин), а операция третьего типа вовсе не меняет элементов массива (как и любой запрос поиска). Поэтому увеличения потенциала могут происходить только во время операций второго типа. В этом случае был изменен один элемент массива. Он был не меньше нуля, а стал $< C$, поэтому потенциал мог уве-

[6]Более формально: Φ_+ — это величина, которая изначально равна нулю, а при каждом изменении потенциала какой-то вершины со старого значения $\varphi(v)$ на новое значение $\varphi'(v)$, Φ_+ увеличивается на $\max(0, \varphi'(v) - \varphi(v))$

личиться максимум на $\log((C-1)+1) - \log(0+1) = \log C$ для каждой вершины, в поддереве которой есть этот элемент, а таких вершин $O(\log n)$ (это предки листа, отвечающего за этот элемент). Всего запросов было q, поэтому суммарно потенциал увеличится максимум на $O(q \log n \log C)$.

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 $\bmod x$, и при этом на этом отрезке есть хотя бы одно число, которое не меньше x, потому что не выполнилось условие *breakCondition*. Воспользуемся следующим известным фактом:

Лемма 2. *Если $k \geq x$, то $k \bmod x \leq \frac{k-1}{2}$.*

Доказательство. Во-первых, заметим, что условие $x \leq \frac{k-1}{2}$ для целых x и k равносильно тому, что $x < \frac{k}{2}$.

Во-вторых, разберем два случая:

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

2. $k < 2x$. В этом случае $k \bmod x = k - x < \frac{k}{2}$. Первое неравенство верно, потому что $k \bmod x = k - xc$ для какой-то неотрицательной константы c, которая в точности равна единице в данном случае, потому что $x \leq k < 2x$ по условию. Второе же неравенство верно в силу того, что $k < 2x$.

□

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

Подытожим. Мы выяснили, что в общем случае асимптотика алгоритма Segment Tree Beats равна $O(q \log n + \Phi_0 + \Phi_+)$ при условии, что при посещении дополнительной вершины потенциал дерева уменьшается на единицу (или любую другую константу). Мы доказали этот факт для представленной задачи, а также выяснили, что $\Phi_0 = O(n \log n \log C)$ и $\Phi_+ = O(q \log n \log C)$. Следовательно, представленный нами алгоритм работает за время $O(q \log n + n \log n \log C + q \log n \log C) = O((n+q) \log n \log C)$, что и требовалось доказать.

Замечание 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. Поменяются лишь потенциал и способ доказательства асимптотики.

Кроме того, можно заметить, что мы нигде не пользовались условием *tagCondition* в доказательстве. На самом деле оно не нужно для этой задачи, но необходимо для случая, когда присвоение происходит на отрезке.

6. Ji Driver Segment Tree ($\min =$, $\sum$)

6.1. Формулировка

В этой задаче нам дан массив целых чисел A, а также имеются запросы двух типов:

1. Даны ql, qr, x. Необходимо заменить i-й элемент массива A для всех i, таких что $ql \le i < qr$, на $\min(A_i, x)$.

2. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.

6.2. Решение

Эта задача является самой стандартной задачей на Segment Tree Beats, и именно она положила начало этой структуре данных. В силу своей важности, структура данных в этой задаче даже носит имя своего автора — Ji Driver Segment Tree.

Решив предыдущую задачу, мы уже выстроили общую схему решения задач на Segment Tree Beats. Остается лишь попытаться применить ее.

Каким должно быть *breakCondition*? 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. То есть, другими словами, если максимум не больше x. Для того чтобы это проверять, мы будем хранить значение максимума на отрезке max_v в каждой вершине дерева отрезков.

А каким же должно быть *tagCondition*? В каком случае мы можем быстро обновить значения в текущей вершине? Здесь уже нет какого-то очевидного ответа, но мы выберем подходящий. Давайте кроме максимума на отрезке в каждой вершине хранить также количество максимумов $cntMax_v$, то есть количество чисел на данном отрезке, которые равны максимуму, а также будем еще хранить второй максимум $secondMax_v$, то есть максимальное значение на этом отрезке, которое *строго* меньше max_v (если все числа на отрезке равны max_v, то $secondMax_v = -\infty$).

Как все эти значения помогут нам придумать правильное условие *tagCondition*? Так как условие *breakCondition* уже не выполнилось, то $x < max_v$, значит, все значения равные максимуму, точно уменьшатся и станут равны x. Однако если $x > secondMax_v$, то никакие другие элементы на этом отрезке не поменяются, потому что они все и так уже строго меньше x. В таком случае мы легко сможем пересчитать сумму на отрезке, ведь мы знаем, что ровно $cntMax_v$ значений, равных max_v, станут теперь равны x, поэтому сумма на отрезке уменьшится ровно на $cntMax_v \cdot (max_v - x)$, а значение max_v теперь станет равно x. Больше ничего на этом отрезке не поменяется, и мы можем завершаться.

Тем самым, условие *tagCondition* в данном случае выглядит как $x > secondMax_v$[7]. Стоит обратить особое внимание на то, что в этом неравенстве знак именно строгий. Если бы знак был нестрогий, то все еще было бы верно, что только значения, равные max_v поменяются, однако теперь максимум и второй максимум «склеились» бы вместе. А это бы означало, что, во-первых, нам нужно пересчитать значение $cntMax_v$, а во-вторых, найти новое значение $secondMax_v$, что не представляется возможным. Именно по этой причине так важно, чтобы знак в неравенстве был строгим.

В отличие от предыдущей задачи, в которой при условии *tagCondition* запрос обновления превращался просто в запрос присвоения на отрезке, в данной задаче этого не происходит. Нам все еще необходимо выполнить операцию $\min =$ для всех детей вершины, в которой выполнилось условие *tagCondition*, однако мы пользуемся тем, что наши условия таковы, что если в вершине выполнилось *tagCondition*, то и для всех потомков выполнится либо *tagCondition*, либо *breakCondition*, так что проталкивать эту информацию в детей не составит труда. Для того чтобы поддерживать отложенную операцию $\min =$, нам нужно уметь ее проталкивать, а также объединять две такие операции в одну.

Для того чтобы протолкнуть операцию, нужно выполнить все те же самые действия, которые мы выполняли для вершины с *breakCondition* или *tagCondition*, потому что если для вершины v выполнено $x > secondMax_v$,

[7]Для большего понимания можно считать, что условие *tagCondition* на самом деле — это $max_v > x > secondMax_v$, но так как *breakCondition* уже не выполнилось, то первое неравенство можно опустить

то и для всех потомков вершины v это выполнено, поэтому для каждой такой вершины либо ничего не поменялось на отрезке, либо же поменялся только максимум. Такое обновление не составит труда проталкивать за константное время.

Объединение двух операций $\min =$ в одну и вовсе не составит труда. Ведь если мы сначала применили операцию $\min = x$ на отрезке, а потом операцию $\min = y$ на отрезке, то в итоге это эквивалентно выполнению всего одной операции $\min =$ с числом $\min(x, y)$.

Таким образом, все описанные операции, которые нам нужно выполнять, несложно поддержать, и на этом описание алгоритма завершается.

Замечание 4 (Заметка по реализации)**.** На самом деле, можно заметить, что проталкиваемое отложенное значение можно не хранить вовсе, потому что если оно присутствует, то оно обязано совпадать со значением max_v. Действительно, если в вершине выполнилось условие *tagCondition*, то теперь значение max_v в точности равно x. Если же на самом деле в вершине v мы не храним никакого отложенного значения, то операция $\min =$ с максимумом на отрезке не поменяет вовсе ничего, потому что все элементы в детях не больше этого максимума, так что проталкивание максимума не испортит ничего, что не стоило менять.

6.3. Доказательство

Докажем, что асимптотика представленного алгоритма составляет $O((n+q)\log n)$.

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

Количество различных чисел на отрезке не больше, чем общее количество чисел на отрезке. Каждое число в массиве принадлежит $O(\log n)$ отрезкам вершин дерева, поэтому суммарный потенциал Φ в любой момент времени не меньше нуля и не больше $O(n \log n)$. А следовательно, $\Phi_0 = O(n \log n)$.

Теперь оценим значение Φ_+. Во время запросов поиска элементы массива не меняются, поэтому потенциал не увеличивается. В запросах $\min =$ потенциал может измениться только для обычных и дополнительных вершин, потому что для непосещенных вершин значения на отрезке не поменяются, а для тупиковых только максимум может немного уменьшиться при условии *tagCondition*, однако это не повлияет на количество различных элементов на отрезке (как уже упоминалось, для потомков тупиковых вершин тоже выполняются либо *tagCondition*, либо *breakCondition*, поэтому для них происходит то же самое). Для дополнительных же вершин количество различных чисел может только уменьшаться за счет того, что числа «склеиваются», но не может увеличиваться, потому что при операции $\min =$ на всем отрезке вершины, равные значения не могли стать различными.

Таким образом, потенциал может увеличиваться только для обычных вершин дерева отрезков. Это частая ситуация, которая не раз еще встретится нам в других задачах. На сколько может увеличиться потенциал для обычной вершины? На самом деле, операция $\min = x$ устроена очень простым образом. Она либо не меняет значение, либо заменяет его на x, поэтому единственное новое значение на отрезке, которое может появиться, — это в точности x, а значит, потенциал в каждой обычной вершине может возрасти максимум на 1. Так как обычных вершин в каждом запросе $O(\log n)$ штук, то мы можем с уверенностью сказать, что $\Phi_+ \leq O(q \log n)$.

Нам осталось лишь понять, что посещение дополнительной вершины уменьшает потенциал хотя бы на 1. Это верно, потому что если вершина является дополнительной, то ни *breakCondition*, ни *tagCondition* не выполнились, а значит, $x \leq secondMax_v$. Это означает, что все значения, которые раньше равнялись max_v и $secondMax_v$, теперь будут равны x, то есть в некотором смысле мы «склеили» два (или даже больше) различных значения в одно[8]. Таким образом, количество различных чисел на отрезке этой вершины уменьшилось хотя бы на 1, а значит, уменьшился и суммарный потенциал дерева. Что и требовалось доказать.

Из всего выше сказанного следует, что асимптотика алгоритма получается равной $O(q \log n + \Phi_0 + \Phi_+) = O(q \log n + n \log n + q \log n) =$

[8]Если $secondMax_v = -\infty$, то *tagCondition* обязано выполниться.

$O((n+q)\log n)$, что завершает наше доказательство.

Замечание 5. По аналогии с этой задачей можно поддерживать также операцию $max =$ и даже их совмещение. Достаточно хранить значения max_v, $cntMax_v$, $secondMax_v$, min_v, $cntMin_v$, $secondMin_v$ и sum_v. Несложно убедиться, что эти операции никак друг другу не мешают, и представленный потенциал работает для них обеих. Оценка времени работы от этого не изменится.

Стоит лишь быть аккуратным с комбинацией отложенных операций $\min =$ и $\max =$, потому что их нельзя объединить в одну отложенную операцию. Если число x, с которым делается операция $\min =$, больше числа y, с которым делается операция $\max =$, то несложно проверить, что от перемены порядка выполнения этих операций результат не меняется, поэтому набор отложенных операций вида $\min =$ и $\max =$ можно переупорядочить так, чтобы сначала шли все операции вида $\min =$, а потом все операции $\max =$. Как мы уже знаем, набор операций $\min =$ можно объединить в одну операцию $\min =$, поэтому любой набор операций обновления можно сократить до всего одной операции $\min =$ и одной операции $\max =$. Если же в какой-то момент значение x не больше числа y, то комбинация этих операций эквивалентна присваиванию константы y (или x, если порядок операций был обратный) на отрезке. Любая последующая операция будет либо не менять элементы на отрезке, либо тоже будет эквивалентна операции присвоения на отрезке. Можно считать, что в любой вершине мы храним либо комбинацию отложенной операции $\min =$ и $\max =$, либо отложенную операцию присвоения на отрезке.

Замечание 6. Так как мы все равно уже храним отложенную операцию присвоения на отрезке, почему бы не добавить ее в список наших запросов. Для этого у нас есть уже все необходимое: мы уже научились хранить отложенную операцию присвоения. Если на отрезке не было отложенной операции присвоения, а были какие-то другие отложенные операции, то это никак нам не мешает, потому что операция присвоения затирает все предыдущие изменения и оставляет только операцию присвоения.

Остается лишь проверить, что операция присвоения на отрезке увели-

чивает потенциал дерева максимум на $O(\log n)$. Это действительно так, потому что для вершин, для которых все значения на их отрезках обновились, новый потенциал равен единице (все числа на отрезке равны), а значит, он мог только уменьшиться от старого значения. Увеличение могло произойти только для вершин, которые посетило дерево отрезков в операции присвоения на отрезке[9], при этом если мы делаем присвоение константы x на отрезке, то точно так же, как и с операцией $\min =$, каждый элемент либо не меняется, либо заменяется на x, поэтому для любой вершины количество различных значений могло увеличиться максимум на 1, а следовательно, так как запрос присвоения посещает $O(\log n)$ вершин, то потенциал дерева действительно возрастает на $O(\log n)$, что вписывается в асимптотику алгоритма.

Тем самым, мы уже можем поддерживать очень широкий спектр операций: $\sum$, $\min$, $\max$, $\min =$, $\max =$, $=$. Единственная стандартная операция, которой нам не хватает, — это операция $+ =$. В следующем разделе мы зададимся вопросом, что будет не так с нашим доказательством при добавлении этой операции, и как мы можем это исправить.

[9]Операция присвоения на отрезке является стандартной операцией дерева отрезков, поэтому она посещает только обычные и тупиковые вершины.

7. Extended Ji Driver Segment Tree ($\min =$, $+ =$, $\sum$)

7.1. Формулировка

В этой задаче нам дан массив целых чисел A, а также имеются запросы трех типов:

1. Даны ql, qr, x. Необходимо заменить i-й элемент массива A для всех i, таких что $ql \le i < qr$, на $\min(A_i, x)$.
2. Даны ql, qr, y. Необходимо прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число y.
3. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.

Замечание 7. Как и в прошлой задаче, мы можем без труда добавить все остальные запросы, но главный фокус для нас сейчас — это как скомбинировать запрос $\min =$ с запросом $+ =$.

7.2. 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.

7.3. Доказательство

Почему же старое доказательство не работает? Чем операция $+ =$ отличается от $\min =$ и $=$? Для понимания рассмотрим пример массива для четного числа n:

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

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

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

В этом случае потенциал корня стал равен n, а потенциалы других вершин не изменились: для них, как раньше, так и теперь, все числа на отрезках различны. То есть, мы за одну операцию увеличили потенциал на $\frac{n}{2}$. Это слишком много, ведь тогда асимптотика получится $O(nq)$.

Мы показали, что потенциал может увеличиваться на $\Theta(n)$ за одну операцию $+=$. Докажем, что это наибольшая величина, на которую он может увеличиться, то есть при операции $+=$ потенциал увеличивается на $O(n)$. Это никак нам не поможет для нового потенциала, но понадобится чуть позже, когда мы будем пытаться дальше улучшать асимптотику.

Теорема 1. *Операция $+=$ увеличивает потенциал дерева на $O(n)$.*

Доказательство. Заметим, что если никакие числа на отрезке вершины не поменялись, то не поменялся и потенциал этой вершины. При этом, если все числа на отрезке вершины поменялись, то количество различных тоже не изменилось, ведь операция $+=$ оставляет равные элементы равными. Так что увеличение потенциала могло происходить только для обычных вершин. При этом, как говорилось в пререквизитах, на каждом уровне дерева отрезков может быть максимум две обычных вершины. Вершина с k-го уровня дерева отрезков содержит на своем отрезке $O\left(\frac{n}{2^k}\right)$ чисел, поэтому максимум на такую величину может увеличиться ее потенциал. Суммарно потенциал может увеличиться максимум на $O\left(2 \cdot \left(\sum_{k=0}^{+\infty} \frac{n}{2^k}\right)\right) = O(2 \cdot 2n) = O(n)$. Что и требовалось доказать. □

Нам нужен другой потенциал.

Замечание 8. Доказывать мы, однако, будем уже не $O((n+q)\log n)$, а $O(n \log n + q \log^2 n)$. При этом не известно ни одного конкретного примера, на котором бы этот алгоритм работал дольше $O((n+q)\log n)$, и именно это и является основным направлением исследования в моей работе: насколько эту асимптотику можно еще улучшить?

Скажем, что вершина дерева отрезков является помеченной, если максимумы в ее левом и правом поддеревьях не совпадают. Обозначим потенциалом $\varphi(v)$ вершины v количество помеченных вершин в поддереве вершины v. Суммарный потенциал дерева Φ как всегда равен сумме потенциалов всех вершин дерева. Можно думать про потенциал дерева немного с другого угла: это сумма глубин всех помеченных вершин, потому что глубина вершины равна количеству вершин, в поддереве которых она встречается.

Очевидно, что в любой момент времени $0 \leq \Phi \leq O(n \log n)$, потому что глубина каждой из $O(n)$ вершин дерева, которые потенциально могут быть помечены, не больше $O(\log n)$. В частности, $\Phi_0 \leq O(n \log n)$.

В какой ситуации потенциал Φ мог увеличиться? Если какая-то вершина раньше не была помеченной, а теперь стала, и тем самым увеличила потенциалы всех своих предков. Метка в вершине могла появиться, если раньше максимумы в ее детях совпадали, а после запроса стали различаться. Если на отрезке вершины никакие значения не поменялись, то и максимумы остались равны. Если на отрезке вершины все значения поменялись, то максимумы опять же остались равны, потому что наши операции, во-первых, оставляют относительный нестрогий порядок на элементах, то есть максимумами не могли стать какие-то другие значения, а во-вторых, меняют одинаковые элементы одинаково, так что в таком случае максимумы тоже остались бы равны. Значит, метка могла появиться только в том случае, если какие-то, но не все, элементы на отрезке данной вершины поменялись. Это в точности значит, что эта вершина является обычной вершиной. Обычных вершин в каждом запросе $O(\log n)$ штук, и каждая из них, если обрела метку, увеличила на 1 потенциал $O(\log n)$ своих предков. Тем самым, итоговый потенциал за один запрос может увеличиться на $O(\log^2 n)$, а значит, $\Phi_+ = O(q \log^2 n)$.

Осталось лишь доказать, что при посещении дополнительной вершины потенциал этой вершины уменьшается хотя бы на 1. Это в точности эквивалентно следующей ключевой теореме:

Теорема 2. *В поддереве любой дополнительной вершины v есть помеченная вершина u, которая после применения операции перестанет быть помеченной.*

Доказательство. Если вершина v дополнительная, то для нее $secondMax_v \geq x$, потому что не выполнились *breakCondition* и *tagCondition*. 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 — $secondMax_v$. И в таком случае сейчас эти числа различаются, а после применения операции оба будут равны x, поэтому из вершины u пропадет пометка.

Почему же такая вершина u существует? Назовем вершину t v-подобной, если $max_t = max_v$ и $secondMax_t = secondMax_v$. В частности, v является v-подобной. Тогда возьмем самую глубокую v-подобную вершину в поддереве вершины v (если таких несколько, возьмем любую) и назовем ее u. Почему она подойдет? Для вершины u верно, что $max_u = max_v$ и $secondMax_u = secondMax_v$. При этом для детей вершины u[10] это уже не верно, потому что u является самой глубокой v-подобной вершиной. max_u равен максимуму из максимума в ее сыновьях u_l и u_r. Пускай Н.У.О. $max_{u_l} = max_u = max_v$. В таком случае число $secondMax_u$ не может встречаться на отрезке вершины u_l, потому что иначе вершина u_l тоже была бы v-подобной. Значит, все вхождения числа $secondMax_u$ на отрезке вершины u принадлежат правому поддереву (и там это значение является максимумом). Из этого, в частности, следует, что $max_{u_r} \neq max_u$, потому что иначе на отрезке u_r встречались бы и max_u и $secondMax_u$, а значит, она была бы v-подобной. Следовательно, все вхождения max_u относятся к левому поддереву, а все вхождения $secondMax_u$ относятся к правому поддереву. Из этого, в частности, следует, что до применения операции в вершине u была метка, потому что эти числа не равны. Однако в силу того, что $secondMax_u = secondMax_v \geq x$, как было замечено в начале доказательства, после применения операции max_{u_l} и max_{u_r} будут оба равны x, а значит, метка из вершины u пропадет. Что и требовалось доказать. □

Просуммируем доказанное. Мы выяснили, что $\Phi_0 = O(n \log n)$, $\Phi_+ = O(q \log^2 n)$, и при посещении дополнительной вершины потенциал уменьшается на 1, следовательно, асимптотика алгоритма равна $O(q \log n + \Phi_0 + \Phi_+) = O(q \log n + n \log n + q \log^2 n) = O(n \log n + q \log^2 n)$, что в точности совпадает

[10]Вершина u не может быть листом, потому что в таком случае $secondMax_v = secondMax_u = -\infty$, но $secondMax_v$ не может быть равно $-\infty$, ведь в таком случае бы в вершине v выполнилось условие *tagCondition*.

с тем, что мы хотели доказать.

Замечание 9. В отличие от предыдущей задачи, в данной задаче потенциал для запроса $\max =$ будет отличаться (все симметрично), поэтому, чтобы проделать такое же доказательство для полного набора запросов, необходимо ввести второй потенциал и следить за тем, как они оба меняются. Итоговый потенциал можно считать суммой этих двух потенциалов.

7.4. Улучшенная оценка

Описанная оценка с другим (неформальным) доказательством и чуть менее аккуратной асимптотикой $O((n+q)\log^2 n)$ была представлена в статье Ruyi Ji. Эта оценка не соответствует оптимальной «нижней» известной оценке, которая на данный момент составляет тривиальные $O((n+q)\log n)$, не отличающиеся от обычной задачи Ji Driver Segment Tree. Иными словами, есть зазор между наихудшим известным входом для данного алгоритма и верхней оценкой на его время работы. Этот вопрос широко обсуждался, но за 6 лет с появления Segment Tree Beats так и не был решен. В этом разделе я представлю улучшение представленной ранее оценки на время, которое сокращает этот зазор.

Для этого улучшения нам понадобится более пристально посмотреть на то, какие запросы у нас есть, и придумать более аккуратный потенциал. Мы будем отдельно следить за тем, сколько запросов $\min =(q_1)$, $+=(q_2)$ и $\sum(q_3)$ у нас есть ($q_1+q_2+q_3=q$). В предыдущем доказательстве запросы изменения работали за амортизированное время $O(\log^2 n)$, потому что именно на столько увеличивали потенциал, а запрос поиска суммы работал за $O(\log n)$. Итоговая асимптотика тогда получалась бы $O(n\log n+q_1\log^2 n+q_2\log^2 n+q_3\log n)$. Мы улучшим эту оценку, предъявив анализ, из которого следует, что тот же самый алгоритм работает на самом деле за $O(n\log n+q_1\log n+q_2\log^2 n+q_3\log n)$, то есть квадратичный логарифм остается только перед запросами $+=$. В частности, из этого следует, что если $q_2=O\left(\frac{q}{\log n}\right)$, мы получаем оценку $O((n+q)\log n)$ на время работы, что совпадает с нижней оценкой, однако ранее такая асимптотика могла быть получена только для $q_2=O(\log n)$, что следует из теоремы 1 (мы не можем использовать улучшенный потенциал для

подобных оценок, потому что в нем $O(\log^2 n)$ занимают оба вида запросов обновления). Из всего выше сказанного я считаю, что представленный далее результат является существенным продвижением относительно известных ранее оценок и потенциально может привести к преодолению зазора между нижними и верхними оценками для данной задачи.

Для того чтобы доказать подправленную оценку, нам нужно будет подправить старый потенциал. Мы все еще будем использовать пометки в вершинах, однако изучим их более подробно.

Как уже говорилось ранее, вершина v является помеченной, если максимумы в ее детях max_{v_l} и max_{v_r} не совпадают. Скажем, что у меток есть цвета, и если вершина v помечена, то цвет ее метки равен $\min(max_{v_l}, max_{v_r})$[11], то есть меньшему из двух максимумов в детях. Скажем, что метка является **жирной**, если в ее поддереве нет других меток такого же цвета как она сама. Потенциалом вершины теперь будет количество жирных меток в ее поддереве, а суммарным потенциалом дерева будет сумма потенциалов всех вершин, что эквивалентно сумме глубин всех жирных меток.

Лемма 3. *Определение жирной метки эквивалентно тому, что меньший из двух максимумов в детях не встречается на отрезке другого ребенка.*

Доказательство. Пусть Н.У.О. $max_{v_l} < max_{v_r}$.

Действительно, если max_{v_l} не встречается на отрезке правого сына, то ниже v метки с таким цветом быть не может, потому что в поддереве v_r такого числа нет вовсе, а в поддереве v_l оно всегда является максимумом.

И наоборот, если max_{v_l} встречается на отрезке правого сына, то посмотрим на лист, соответствующий одному из таких значений в правом поддереве. В этом листе максимум на отрезке равен max_{v_l}, а в v_r максимум равен max_{v_r}. Будем идти вертикально вверх по пути от листа до v_r и найдем первый момент, когда максимумом в поддереве перестало быть число max_{v_l}. Именно у этой вершины и будет метка цвета max_{v_l}, потому что максимум в поддереве, откуда мы пришли, равен max_{v_l}, а в другом он строго больше, потому что max_{v_l} перестало быть максимумом. □

[11] Обратите внимание, что это число может как совпадать, так и отличаться от $secondMax_v$.

Докажем, что аккуратный анализ такого потенциала даст нам улучшенную оценку. Так же, как и для предыдущего потенциала, очевидно, что $0 \leq \Phi \leq O(n \log n)$ в любой момент, поэтому $\Phi_0 = O(n \log n)$. Более аккуратного анализа заслуживает Φ_+.

В случае запроса поиска суммы, конечно же, никакие элементы массива, а значит, и потенциалы не меняются. В случае запроса прибавления на отрезке все тоже остается, как и при старом потенциале. Если на отрезке вершины не поменялись никакие элементы, то не могла появиться и жирная метка. Если в поддереве какой-то вершины ко всем элементам прибавили константу, то хоть цвета меток в ее поддереве и поменялись, все еще равные цвета остались равными, а различные различными, так что жирные метки остались жирными, а не жирные остались не жирными. Поэтому единственные вершины, в которых могли появиться жирные метки (из-за того, что раньше там не было метки, либо же из-за того, что метка была, но ранее была не жирной), — это обычные вершины. Таких вершин $O(\log n)$, и каждая из них даст вклад в $O(\log n)$ потенциалов, поэтому операция $+=$ увеличивает потенциал дерева на $O(\log^2 n)$.

Теперь необходимо понять, почему при операции $\min =$ потенциал может увеличиться только на $O(\log n)$. Если на отрезке вершины не поменялись никакие числа, то с ее меткой тоже ничего не могло произойти. Если на отрезке вершины ко всем элементам применили операцию $\min =$, то в этой вершине не могло появиться жирной метки. Действительно, как мы уже знаем, в таком случае новая метка появиться не могла, а если метка уже была, но не была жирной, то в ее поддереве есть другая метка такого же цвета, с которой в таком случае тоже ничего не произойдет. Следовательно, новые жирные метки могут появляться только в обычных вершинах. Это уже дает нам оценку $O(\log^2 n)$ на увеличение потенциала за операцию, но нам этого недостаточно. Изучим обычные вершины подробнее.

Поймем, что единственные по-настоящему новые метки при операции $\min = x$ могут появиться только с цветом x. Так как жирной может быть из них только та, у которой в поддереве больше нет меток такого цвета, то на каждом из двух вертикальных путей из обычных вершин может появиться максимум одна новая жирная метка цвета x, что увеличивает наш потенциал

на $O(2 \log n)$.

Остается лишь понять, что жирные метки других цветов — это не новые метки, а «всплывшие» старые. Пусть на какой-то обычной вершине v появилась жирная метка цвета y. Есть три варианта, которые особо друг от друга не отличаются:

1. Раньше в этой вершине уже была метка y, но она не была жирной, потому что ниже нее была другая вершина u с жирной меткой такого же цвета.

2. Раньше в этой вершине была метка другого цвета t, которая либо просто пропала, либо всплыла куда-то выше.

3. Раньше в этой вершине не было метки вовсе.

Во всех этих ситуациях после применения операции меньшим из двух максимумов в поддеревьях стало число y, не равное x. Так как операция $\min = x$ каждое число либо оставляет неизменным, либо заменяет на x, то присутствие числа y на отрезке после применения операции свидетельствует о том, что это число было на отрезке и до применения операции. Пусть Н.У.О. $y = max_{v_l}$, а $max_{v_r} = z > y$. Тогда число z по аналогичным причинам либо присутствовало на отрезке раньше, либо $z = x$ (а в таком случае раньше это значение было еще больше). В любом случае, очевидно, что до применения операции max_v было больше y, но при этом y присутствовало на отрезке вершины v. Тогда посмотрим на произвольный лист в поддереве v, соответствующий значению y. В этом листе значение максимума равно y, а в вершине v не равно. Тогда мы аналогично доказательству леммы 3 можем пойти вверх по пути от этого листа до вершины v, пока не встретим первую вершину, для которой y больше не является максимумом. Тогда в этой вершине есть метка цвета y. Она не обязана быть жирной, но если она не жирная, то есть метка цвета y еще ниже. В любом случае мы показали, что до применения операции в поддереве вершины v была метка цвета y. Если после применения операции в вершине v появилась **жирная** метка цвета y, это означает, что все метки цвета y ниже пропали, а значит, мы можем считать, что все кроме одной просто исчезли, а одна жирная (пускай, она была в вершине u) из них «всплыла»

до вершины v, уменьшив тем самым наш суммарный потенциал на $h_u - h_v$[12]. Так как v (а значит, и все ее предки) является обычной вершиной, то на пути от u до корня не более $h_u - h_v$ дополнительных вершин, которые должны были уменьшать потенциал дерева на 1, так что наше «всплытие» вместо полного уничтожения жирной метки не мешает потенциалу уменьшаться на столько, насколько ему нужно.

Тем самым мы показали, что новые метки точно не могли появляться нигде кроме обычных вершин, а в обычных могло разве что появиться две новые жирные метки цвета x, а остальные являются просто всплывшими старыми метками. Таким образом, при операции $\min =$ потенциал дерева может увеличиться максимум на $O(\log n)$.

Для завершения доказательства остается лишь заметить, что при посещении дополнительной вершины потенциал уменьшается на 1. Для этого нам нужно лишь обобщить теорему 2 на случай нового потенциала.

Теорема 3. *В поддереве любой дополнительной вершины v есть вершина u с **жирной** меткой, которая после применения операции перестанет быть помеченной*[13].

Доказательство. Доказательство ничем не отличается от доказательства теоремы 2. Давайте заметим, что вершина u, которую мы нашли в доказательстве теоремы 2, не просто потеряла метку, а потеряла жирную метку. Это следует из того, что цвет вершины u — это $secondMax_v$, и в ее поддереве не может быть ни одной другой вершины такого цвета, ведь в одном поддереве значения $secondMax_v$ нет вовсе, а в другом оно является максимумом, поэтому не может являться меньшим из двух максимумов ни для какой вершины. □

Подведем итог того, что мы доказали. Начальный потенциал дерева Φ_0 не превосходит $O(n \log n)$. Операция $+ =$ увеличивает потенциал на $O(\log^2 n)$, а операция $\min =$ увеличивает потенциал на $O(\log n)$, следова-

[12] h_v — расстояние от вершины до корня дерева.

[13] Как нетрудно заметить, мы доказываем не просто, что в поддереве какая-то метка перестала быть жирной, а то, что этой метки не стало вовсе. К сожалению, это замечание никак не помогает нам в оценке асимптотики алгоритма.

тельно, $\Phi_+ = O(q_1 \log n + q_2 \log^2 n)$. При этом при посещении дополнительной вершины потенциал уменьшается на единицу. Из всего этого следует, что асимптотика алгоритма составляет $O(q \log n + \Phi_0 + \Phi_+) = O(q \log n + n \log n + q_1 \log n + q_2 \log^2 n) = O(n \log n + q_1 \log n + q_2 \log^2 n + q_3 \log n)$. Что и требовалось доказать.

8. GCD Ji Driver Segment Tree ($\min =$, $+ =$, $\gcd$)

8.1. Формулировка

В этой задаче нам дан массив неотрицательных целых чисел A ($0 \leq A_i < C$), а также имеются запросы трех типов:

1. Даны ql, qr, x. Необходимо заменить i-й элемент массива A для всех i, таких что $ql \leq i < qr$, на $\min(A_i, x)$.

2. Даны ql, qr, y ($0 \leq y < C$). Необходимо 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 $[ql, qr)$.

Замечание 10. Очевидно, что запросы поиска суммы, минимума и максимума мы все так же можем параллельно поддерживать, как и в предыдущей задаче.

Замечание 11. Так как у нас есть запросы прибавления на отрезке, числа в массиве могут потенциально стать больше C. В действительности они могут достигнуть значения $O(qC)$. При анализе асимптотики мы будем использовать логарифм максимального значения в массиве, записывая его как $O(\log C)$, что может показаться не совсем корректным, ведь на самом деле оно должно быть $O(\log(qC))$. Но так как это значение равно $O(\log q + \log C)$, это эквивалентно $O(\log C)$ при условии, что $q < poly(C)$, которое мы безусловно подразумеваем. Если же не делать этого допущения, то очевидным образом во всех утверждениях ниже в асимптотике $O(\log C)$ нужно заменить на $O(\log q + \log C)$, что ничего кардинально не изменит.

Если же считать, что элементы массивов полиномиальны относительно размера массива, то $\log C = O(\log n)$, поэтому везде в асимптотиках можно заменить $\log C$ на $\log n$.

8.2. Решение + доказательство упрощенной версии задачи

Давайте сначала поймем, как решать эту задачу, если запросов $\min =$ 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. Однако это можно сделать даже без применения Segment Tree Beats, и сейчас мы разберемся, как.

Теорема 4. *Пускай дано дерево (необязательно бинарное), в вершинах которого написаны числа a_1, a_2, ..., a_n. Запишем на ребре дерева разность чисел в концах этого ребра. Тогда НОД чисел всех попарных разностей $a_i - a_j$ равен НОДу всех чисел, записанных на $n-1$ ребре дерева.*

Доказательство. Общеизвестно, что $\gcd(a,b) = \gcd(a,b,a-b) = \gcd(a,b,a+b)$. Возьмем НОД дерева и постепенно такими операциями превратим в НОД всех разностей. Докажем, что можно получить произвольную разность $a_v - a_u$. Действительно, в дереве есть путь $a_u \to a_{i_1} \to a_{i_2} \to \cdots \to a_{i_k} \to a_v$. На ребрах этого пути написаны числа $a_{i_1} - a_u$, $a_{i_2} - a_{i_1}$, ..., $a_v - a_{i_k}$ (возможно, с противоположными знаками, но это не принципиально). Если просуммировать эти числа, мы получим $a_v - a_u$. Тем самым, линейными комбинациями с целыми коэффициентами наших изначальных чисел мы можем получить все попарные разности, поэтому их НОДы совпадают. □

Следствие 1. *НОД всех чисел, записанных на ребрах дерева, а также одного **любого** числа, записанного в вершине, равен НОДу чисел a_1, a_2, ..., a_n.*

Доказательство. НОД ребер дерева равен НОДу попарных разностей. Пусть Н.У.О. нам еще дали число a_1. Тогда сложив a_1 с $a_v - a_1$, мы можем получить любое изначальное число a_v. С другой стороны, вычитая изначально данные числа, мы можем получить любую попарную разность. Тем самым, оба НОДа равны НОДу всех изначальных чисел и попарных разностей, а значит, они равны друг другу. □

Что говорит нам эта теорема? То, что для вычисления НОДа на отрезке нам достаточно хранить НОД какого-то остовного дерева на графе разностей элементов этого отрезка, а также отдельно один **любой** элемент на отрезке. Взяв НОД этих двух величин, мы сможем получить НОД всех чисел на отрезке.

Как мы будем это поддерживать? На отрезке вершины v мы будем хранить величину $diffGCD_v$, равную НОДу всех попарных разностей элементов

на отрезке, а также значение $anyValue_v$, равное произвольному элементу на отрезке.

Нам нужно научиться делать две вещи. Во-первых, нужно научиться пересчитывать эти величины, когда ко всем элементам на отрезке прибавляется константа. Но ведь если ко всем числам прибавили константу x, попарные разности никак не меняются, а значит, не меняется и $diffGCD_v$. К $anyValue_v$ же просто прибавится x. Во-вторых, нужно уметь пересчитывать $diffGCD$ и $anyValue$ на отрезке через значения на двух подотрезках детей. В качестве $anyValue$ можно взять любое из двух $anyValue$ детей. Для того чтобы пересчитать $diffGCD$, нужно взять два дерева, построенные в детях, и провести между ними одно произвольное ребро (к примеру, между значениями $anyValue$ детей), чтобы тем самым построить дерево на всех элементах на отрезке. То есть, $diffGCD_v = \gcd(diffGCD_{v_l}, diffGCD_{v_r}, anyValue_{v_l} - anyValue_{v_r})$. Для того чтобы в конце ответить на запрос поиска НОДа на отрезке, достаточно посчитать НОД $diffGCD$ и $anyValue$.

Такое решение, очевидно, работает за $O(n \log C + q \log n \log C)$, потому что на обычную асимптотику дерева отрезков накладывается вычисление $\gcd$ в каждой вершине, которое работает за $O(\log C)$. Однако эту асимптотику можно уменьшить до $O(n \log C + q(\log n + \log C))$, используя следующие леммы.

Лемма 4. *Вычисление НОДа двух целых положительных чисел x и y при помощи алгоритма Евклида работает за $O\left(\log\left(\frac{\min(x,y)}{\gcd(x,y)}\right) + 1\right)$.*

Доказательство. Известным фактом является то, что вычисление НОДа чисел x и y при помощи алгоритма Евклида работает за $O(\log(\min(x, y)) + 1)$ (после первого же раза, когда бóльшее число будет взято по модулю меньшего, оба числа станут не больше $\min(x, y)$, после чего алгоритм будет работать за логарифмическое время). Однако эту асимптотику можно уточнить. Пусть $\gcd(x, y) = g$. В таком случае процесс вычисления НОда чисел x и y будет в точности повторять процесс вычисления НОДа чисел $\frac{x}{g}$ и $\frac{y}{g}$ с тем различием, что в первом случае все числа будут в g раз больше. Из этого следует, что асимптотика вычисления НОДа обеих пар равна $O\left(\log\left(\min\left(\frac{x}{g}, \frac{y}{g}\right)\right) + 1\right)$, что и требовалось доказать. □

Лемма 5. *Даны n целых чисел a_1, a_2, ..., a_n ($0 \le a_i < C$). Тогда последовательное вычисление НОДа этих n чисел (вычислить НОД первых двух чисел, потом вычислить НОД результата с третьим числом, и так далее) работает за $O(n + \log C)$.*

Доказательство. Немного преобразовав формулу из предыдущей леммы (логарифм отношения равен разности логарифмов), мы получим такую асимптотику: $O(\log(\min(x, y)) - \log(\gcd(x, y)) + 1)$. Посмотрим на то, как будет работать наш алгоритм последовательного вычисления НОДа. Обозначим число, которое мы получили на i-й итерации, за b_i, то есть $b_i = \gcd(a_1, a_2, \ldots, a_i) = \gcd(b_{i-1}, a_i)$. На $i + 1$-й итерации мы запустим алгоритм Евклида от чисел b_i и a_{i+1}. Итоговая асимптотика алгоритма равна $O((\log(\min(b_1, a_2)) - \log(\gcd(b_1, a_2)) + 1) + (\log(\min(b_2, a_3)) - \log(\gcd(b_2, a_3)) + 1) + \ldots + +(\log(\min(b_{n-1}, a_n)) - \log(\gcd(b_{n-1}, a_n)) + 1)) = O((\log(\min(b_1, a_2)) - \log(b_2)) + (\log(\min(b_2, a_3)) - \log(b_3)) + \ldots + (\log(\min(b_{n-1}, a_n)) - \log(b_n)) + n) \le O((\log(b_1) - \log(b_2)) + (\log(b_2) - \log(b_3)) + \ldots + (\log(b_{n-1}) - \log(b_n)) + n) = O(\log(b_1) - \log(b_n) + n) = O(\log C + n)$. Что и требовалось доказать. □

Как мы говорили в пререквизитах, посещенные вершины в обычном дереве отрезков — это два вертикальных пути, от которых торчат тупиковые вершины. Поэтому если мы посмотрим на то, как работает вычисление НОДа в запросе поиска и пересчет НОДов в запросе обновления на отрезке, то мы поймем, что на самом деле это два последовательных вычисления НОДа по этим двум вертикальным путям, а потом по одному пути, когда эти пути сливаются в один. Тем самым, так как длина пути равна $O(\log n)$, асимптотика каждого запроса равна в реальности $O(2 \cdot (\log n + \log C))$, поэтому асимптотика алгоритма составляет $O(n \log C + q(\log n + \log C))$[14].

8.3. Решение + доказательство полной версии задачи

Итак, мы научились поддерживать операции $+ =$ и $\gcd$ на отрезке. Давайте добавим к этому еще запрос $\min =$.

[14] С множителем при n для построения дерева ничего не меняется, потому что даже для вычисления НОДа на предпоследнем уровне дерева нужно $\frac{n}{2}$ раз запустить независимые вычисления НОДов пар элементов массива.

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 полностью отдельно. Тогда при условии *tagCondition* (которое, как и *breakCondition*, абсолютно такое же, как и в Ji Driver Segment Tree) мы сможем легко пересчитать значения. Нам нужно будет изменить только максимум, а $diffGCD$ никак не поменяется, потому что мы изменяем значения только максимумов, которые не входят в $diffGCD$.

Как же нам тогда находить НОД на отрезке, зная эти значения? Все числа, не равные максимуму, уже объединены в остовное дерево внутри $diffGCD$. Остается только присоединить максимумы (или один из них, потому что они все равно равны друг другу). В этом нам как раз может помочь $secondMax$. Поэтому НОД на отрезке можно вычислить по формуле $\gcd(diffGCD, max - secondMax, max)$. При пересчете значения в вершине через значения в детях нужно быть аккуратным, потому что меньший из двух максимумов в поддеревьях (если они не равны) перестанет быть максимумом во всем поддереве, и его нужно добавить в остовное дерево, но нетрудно проверить, что все это можно пересчитать.

В оригинальной статье асимптотика этого алгоритма указана как $O(q \log^3 n)$. По всему видимому, там подразумевается, что $C = q = n$, так что в реальности такая оценка будет выглядеть как $O(n \log n \log C + q \log^2 n \log C)$, то есть асимптотика обычного Extended Ji Driver Segment Tree, умноженная на $\log C$ за подсчет НОДа на каждой итерации.

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 это не следует очевидным образом из той улучшенной оценки, которую мы получили для задачи без операции $\min =$? Дело в том, что там мы явно пользовались структурой запроса к обычному дереву отрезков: запрос состоит из двух вертикальных путей, и условие на то, что НОД вычисляется за $O(n + \log C)$, работает только в том случае, если вычисление происходит последовательно. В случае же с Segment Tree Beats мы посещаем большое количество дополнительных вершин, которые совсем не обязаны образовывать какие-то красивые пути. Но если внимательнее посмотреть на ситуацию, все таки можно обна-

ружить те самые нужные нам пути.

Давайте вспомним, как мы оценивали время работы в задаче с запросами $\min =$ и $+ =$. Мы говорили, что вершина является помеченной, если максимумы в ее детях отличаются. Мы поняли, что при запросе изменения вершина может становиться помеченной только если это обычная вершина, а значит, за один запрос появляется максимум $O(\log n)$ новых меток. Дальше мы вводили потенциал, равный сумме глубин этих меток, но сейчас нас будет интересовать именно их количество. Изначально их максимум $O(n)$, а затем за все время добавится максимум $O(q \log n)$.

Кроме того вспомним, что в теореме 2 мы доказали, что у любой дополнительной вершины в поддереве есть метка, которая пропадет на этой итерации. На это можно посмотреть с другой стороны: Пусть v_1, v_2, …, v_k — это все метки, которые пропали на текущей итерации. Тогда это значит, что все посещенные дополнительные вершины — это их предки. Тогда все дополнительные посещенные вершины принадлежат k вертикальным путям от этих вершин до корня. Кроме того есть еще два вертикальных пути на обычных вершинах, так что всего все посещенные вершины образуют $k + 2$ вертикальных пути, из которых торчат тупиковые вершины. Суммарно таких меток за все время могло удалиться максимум $O(n + q \log n)$ штук, комбинируя с $O(q)$ путями из обычных вершин мы получаем, что за время выполнения алгоритма мы считали НОД на $O(n + q \log n)$ путях длины $O(\log n)$ каждый, что дает нам оценку $O((n + q \log n)(\log n + \log C))$. Что и требовалось доказать.

Замечание 12. Абсолютно без изменений мы можем проделать те же утверждения с нашим «улучшенным» потенциалом и получить лучшую оценку $O((n + q_1 + q_2 \log n + q_3)(\log n + \log C))$, где q_1 — количество запросов $\min =$, q_2 — количество запросов $+ =$, и q_3 — количество запросов поиска НОДа.

В частности, мы получаем, что если запросов $+ =$ не больше $O(\frac{q}{\log n})$, то алгоритм работает за $O((n+q)(\log n+\log C))$. В частности, такая асимптотика оказывается верной для задачи с запросами $\min =$ и $\gcd$, что совершенно не следует из доказательства времени работы алгоритма Ji Driver Segment Tree, потому что оно не дает нам понять, что на самом деле посещаемые вершины образуют небольшое количество вертикальных путей.

9. $\mathrm{mod}\ =$, $=$ на отрезке, $\sum$

9.1. Формулировка

Вернемся к самой первой задаче, которую мы обсудили, и докажем, что то же самое решение работает для более общей ее версии, когда операция присвоения производится не в точке, а на отрезке.

В этой задаче нам дан массив неотрицательных целых чисел A ($0 \le A_i < C$), а также имеются запросы трех типов:

1. Даны ql, qr, x ($1 \le x < C$). Необходимо заменить i-й элемент массива A для всех i, таких что $ql \le i < qr$, на $A_i \bmod x$.

2. Даны ql, qr, y ($0 \le y < C$). Необходимо 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 $[ql, qr)$.

9.2. Решение

Решение ничем не отличается от решения простой версии задачи, описанного в разделе 5. Операция присвоения на отрезке добавляется очевидным стандартным образом.

9.3. Доказательство

В данном случае 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 *tagCondition*, потому что $max = min$.

Отрезок, за который отвечает любая вершина, разбивается на подотрезки (возможно, длины 1) с одинаковыми значениями. У любой непомеченной вершины таких подотрезков 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 как обычно брать логарифмы элементов, увеличенных на 1. К примеру, если отрезок вершины — это $[3, 3, 2, 2, 2, 5, 2, 1, 1]$, то потенциал этой вершины равен $\log(3+1) + \log(2+1) + \log(5+1) + \log(2+1) + \log(1+1)$.

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 $\Phi_0 = O(n \log n \log C)$ (при этом, конечно, потенциал в любой момент времени неотрицателен).

При посещении тупиковых вершин либо потенциал не меняется, если выполнилось *breakCondition*, либо потенциал как был нулем, так и останется, если выполнилось *tagCondition*, потому что такие вершины помечены, как и все остальные вершины в их поддереве.

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, то есть не увеличивается.

При посещении же обычных вершин в обоих запросах изменения потенциалы могут увеличиваться, но не очень сильно. Могло появиться не более трех[15] новых отрезков элементов на границах отрезка запроса, каждый из

[15]Конечно, двух, но если все числа были равны, то вершина была помечена, и мы считали, что там ноль отрезков, а после присвоения где-то в середине отрезков стало три.

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 *breakCondition*, ни *tagCondition*, так что вершина не помечена, и $max \geq x$, поэтому на отрезке хотя бы одно значение уменьшится хотя бы в два раза после взятия по модулю, а значит, уменьшится и весь его подотрезок равных значений, поэтому потенциал текущей вершины уменьшится хотя бы на 1. Что и требовалось доказать.

Таким образом, асимптотику получившегося алгоритма можно оценить как $O((n + q) \log n \log C)$.

10. $\sqrt{}=, +=, =, \sum$

10.1. Формулировка

В этой задаче нам дан массив неотрицательных целых чисел A ($0 \le A_i < C$), а также имеются запросы четырех типов:

1. Даны ql, qr. Необходимо заменить i-й элемент массива A для всех i, таких что $ql \le i < qr$, на $\lfloor \sqrt{A_i} \rfloor$. [16]
2. Даны ql, qr, x ($0 \le x < C$). Необходимо прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число x.
3. Даны ql, qr, y ($0 \le y < C$). Необходимо присвоить всем элементам массива A на полуинтервале $[ql, qr)$ значение y.
4. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.

Замечание 13. Аналогично ситуации с $\gcd$, так как у нас есть операция прибавления на отрезке, числа потенциально могут возрасти до $O(qC)$. Мы подразумеваем, что $q < poly(C)$, чтобы об этом не беспокоиться, но безусловно если об этом беспокоиться, ничего кардинально не изменится.

10.2. 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. В каждой вершине мы храним два отложенных значения: что нужно прибавить на отрезке и что нужно присвоить на отрезке. При этом оба эти значения не могут одновременно храниться в вершине, потому что любая комбинация операций присвоения и прибавления легко может быть превращена в одну операцию присвоения.

В данной задаче никакого особенного условия *breakCondition* не будет. Мы завершаемся лишь в том случае, если отрезки не пересекаются, как и в обычном дереве отрезков.

[16] $\lfloor\ \rfloor$ — округление вниз до ближайшего целого числа.

А каким же должно быть *tagCondition*? Первая идея, которая приходит в голову — это условие $\lfloor\sqrt{\max}\rfloor = \left\lfloor\sqrt{\min}\right\rfloor$, потому что в этом случае корни из всех чисел на отрезке равны, и нужно просто произвести присвоение этому корню на всем 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 *tagCondition* не выполнится, так что каждая операция $\sqrt{}=$ будет выполняться за $O(n)$ времени, что означает, что итоговая асимптотика будет равна $O(q \cdot n)$. Поэтому это *tagCondition* нам не подходит.

На самом деле, правильный *tagCondition* выглядит следующим образом: $max_v - min_v \leq 1$. С первого взгляда может показаться, что это условие только слабее. Раньше мы проверяли, что всего лишь корни совпадают, а теперь мы проверяем, что либо максимум равен минимуму, либо они отличаются на 1. Но здесь и кроется принципиальная разница. Единственный плохой случай в прошлом *tagCondition* — это когда max_v — это квадрат натурального числа, а min_v на 1 меньше. Тогда при взятии корня разница между ними останется равной единице, и операцией $+=$ можно будет вернуть массив в исходное положение. Если же разница была больше единицы, то она обязательно уменьшится, как мы поймем чуть позже, что и послужит основой нашего потенциала.

Как же нам обновить значение в вершине, когда выполнилось *tagCondition*? 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 *tagCondition*, который мы обсуждали ранее. Остается второй 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 $k - k^2$.

Таким образом мы без труда сможем поддерживать нужные значения при данном *tagCondition*, и остается лишь доказать, что такой *tagCondition* даст нам приемлемую асимпотику.

10.3. Доказательство

Давайте докажем, что с таким *tagCondition* время работы алгоритма будет составлять $O((n + q \log n) \log C)$.

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

Заметим, что потенциал любой вершины неотрицателен и не превышает $O(\log C)$, так что в любой момент времени верно $0 \leq \Phi \leq O(n \log C)$. В частности, $\Phi_0 = O(n \log C)$.

В каких ситуациях потенциал может увеличиваться? В запросах поиска 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. Для запросов же третьего типа в таком случае в принципе потенциал станет равен нулю, потому что все числа станут равны, а значит, он может только уменьшиться. А вершин, которые пересекаются с отрезком запроса, но не лежат в нем полностью, $O(\log n)$ штук, так что потенциал может увеличиться максимум на $O(q \log n \log C)$ для запросов прибавления и присвоения на отрезке.

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 запро-

сом, 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 $\log(1.5)$ [17], тогда итоговая асимптотика будет равна $O(n \log C + q \log n \log C)$. В этом нам поможет следующий факт:

Лемма 6. *Если a и b — неотрицательные целые числа, и $a \geq b + 2$, то* $a - b \geq 1.5 \cdot \left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor\right) + 0.5$.

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

1. $n \leq 1$. То есть $\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor \leq 1$, так что неравенство, которое нам надо доказать, превращается в $a - b \geq 1.5 \cdot 1 + 0.5 = 2$. Это верно из-за условия на то, что $a \geq b + 2$.

2. $n \geq 2$. В этом случае мы знаем, что $a \geq (n+m)^2$, и $b \leq m^2+2m$, поэтому $a-b \geq (n+m)^2-(m^2+2m) = n^2+2nm-2m = n^2+2m\cdot(n-1) \geq n^2$. Нам нужно доказать, что это не меньше, чем $1.5 \cdot n + 0.5$. Это легко проверить, потому что $n^2 \geq 2n = 1.5n + 0.5n \geq 1.5n + 0.5$. Первое неравенство верно из-за того, что $n \geq 2$, а второе из-за того, что $n \geq 1$. Что и требовалось доказать.

□

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

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

И возьмем логарифмы от обеих частей:

[17] Всегда раньше потенциал уменьшался на единицу, но мы в самом начале статьи делали акцент на том, что на самом деле неважно, на сколько именно уменьшается потенциал. Важно лишь, что это какая-то константа, которая не влияет на асимптотику.

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

То есть мы доказали, что потенциал уменьшился хотя бы на $\log(1.5) \approx 0.585$, и асимптотика решения доказана, ведь $\Phi_0 = O(n \log C)$, $\Phi_+ = O(q \log n \log C)$, и при посещении дополнительной вершины потенциал уменьшается на константу, а значит, асимптотика алгоритма равна $O(q \log n + \Phi_0 + \Phi_+) = O(q \log n + n \log C + q \log n \log C) = O(n \log C + q \log n \log C)$.

10.4. Улучшенная оценка

Гибкость нашего подхода к Segment Tree Beats позволит нам проанализировать асимптотику более аккуратно. Действительно, странно, что мы считаем, что логарифм разности при взятии корня уменьшается на константу. Если бы мы брали просто логарифм корня, а не разности, то он бы уменьшился в два раза. Давайте попытаемся как-то это применить.

Введем потенциал вершины равным $\varphi(v) = \log(\log(max_v - min_v + 1) + 1)$. Очевидным образом аналогично предыдущему потенциалу $\Phi_0 \leq O(n \log\log C)$ и $\Phi_+ \leq O(q \log n \log\log C)$, поэтому если мы докажем, что при посещении дополнительной вершины новый потенциал тоже уменьшается на константу, мы покажем, что алгоритм на самом деле работает за асимптотику $O((n + q \log n) \log\log C)$. Почему же это так? Докажем, что внутренний логарифм в потенциале уменьшается в константу раз при посещении дополнительной вершины. Для этого сформулируем лемму.

Лемма 7. *Если a и b — неотрицательные целые числа, такие что $a \geq b + 3$ и $b \geq 4$, то $a - b + 1 \geq \left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor + 1\right)^2$.*

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

1. $n \leq 1$. То есть $\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor \leq 1$, так что неравенство, которое нам надо доказать, превращается в $a - b + 1 \geq 4$. Это верно из-за условия на то, что $a \geq b + 3$.

2. $n \geq 2$. В этом случае мы знаем, что $a \geq (n+m)^2$, и $b \leq m^2 + 2m$, поэтому $a - b + 1 \geq (n+m)^2 - (m^2 + 2m) + 1 = n^2 + 2nm - 2m + 1$. Нам нужно доказать, что это не меньше, чем $(n+1)^2 = n^2 + 2n + 1$.

$$n^2 + 2nm - 2m + 1 \stackrel{?}{\geq} n^2 + 2n + 1$$

$$2nm - 2m \stackrel{?}{\geq} 2n$$

$$nm - n - m \stackrel{?}{\geq} 0$$

$$(n-1) \cdot (m-1) \stackrel{?}{\geq} 1$$

Что верно, потому что $n \geq 2$ и $m \geq 2$ ($b \geq 4$). Что и требовалось доказать.

□

Если взять логарифмы от обеих частей полученного неравенства, мы получим, что если $a \geq b + 3$ и $b \geq 4$, наш старый потенциал вершины $\log(max_v - min_v + 1)$ уменьшится хотя бы в два раза. Если же $a = b + 2$, то $a - b + 1 = 3$, а $\lfloor \sqrt{a} \rfloor - \left\lfloor \sqrt{b} \right\rfloor + 1$ не превышает 2 (не бывает чисел с разностью 2, такие что разность целых частей их корней больше единицы), поэтому старый потенциал уменьшается хотя бы в $\log(3)$ раз. Если $a - b \leq 1$, то выполнилось *tagCondition*, и этот вариант не может присутствовать в дополнительной вершине. Поэтому нам остается лишь рассмотреть ситуацию, когда $b \leq 3$.

Если $1 \leq b \leq 3$, $\lfloor \sqrt{a} \rfloor - \left\lfloor \sqrt{b} \right\rfloor + 1 = \lfloor \sqrt{a} \rfloor \leq \sqrt{a}$. При этом $a - b + 1 \geq a - 2$. Нетрудно проверить, что $a - 2 \geq \sqrt{a}^{1.5}$ при $a \geq 6$ (в точке 6 левая часть больше, а далее производная левой части всегда больше производной правой части).

Если $b = 0$, то $\lfloor \sqrt{a} \rfloor - \left\lfloor \sqrt{b} \right\rfloor + 1 = \lfloor \sqrt{a} \rfloor + 1 \leq \sqrt{a} + 1$. При этом $a - b + 1 = a + 1$. Нетрудно проверить, что $a + 1 \geq (\sqrt{a} + 1)^{1.5}$ при $a \geq 5$ (в

точке 5 левая часть больше, а далее производная левой части всегда больше производной правой части).

Таким образом, единственные случаи, которые нам остались, — это когда $b \le 3$ и $a \le 5$. Их можно перебрать вручную, либо написать программу и выяснить, что $\log(a-b+1) \ge \log_3 5 \cdot \log\left(\lfloor\sqrt{a}\rfloor - \left\lfloor\sqrt{b}\right\rfloor + 1\right)$, и равенство достигается при $b = 0$ и $a = 4$.

Мы поняли, что при разных ситуациях логарифм уменьшается хотя бы в 2, $\log(3) \approx 1.585$, 1.5 и $\log_3 5 \approx 1.465$ раз. В любом случае логарифм уменьшается хотя бы в $\alpha = \log_3 5$ раз.

Обозначим $\log(a-b+1) \ge \log 3$ за x. Тогда раньше потенциал вершины был равен $\log(x+1)$, а стал не больше $\log\left(\frac{x}{\alpha}+1\right)$, то есть он уменьшился на $\log(x+1) - \log\left(\frac{x}{\alpha}+1\right) = \log\left(\frac{\alpha x+\alpha}{x+\alpha}\right) = \log\left(\alpha - \frac{\alpha^2-\alpha}{x+\alpha}\right)$. Числитель и знаменатель дроби положительны, поэтому чем x больше, тем значение логарифма больше. Так что минимум достигается при минимальном значении $x = \log 3$. В этом случае потенциал уменьшается на $\log\left(\alpha - \frac{\alpha^2-\alpha}{\log 3+\alpha}\right) \approx 0.3122$. Тем самым мы доказали, что хотя бы на такую величину уменьшается потенциал при посещении дополнительной вершины, что завершает наше доказательство.

11. $/=$, $+=$, $=$, $\sum$

11.1. Формулировка

В этой задаче нам дан массив неотрицательных целых чисел A ($0 \leq A_i < C$), а также имеются запросы четырех типов:

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

2. Даны ql, qr, y ($0 \leq y < C$). Необходимо прибавить ко всем элементам массива A на полуинтервале $[ql, qr)$ число y.

3. Даны ql, qr, z ($0 \leq z < C$). Необходимо присвоить всем элементам массива A на полуинтервале $[ql, qr)$ значение z.

4. Даны ql, qr. Необходимо вернуть сумму элементов массива A на полуинтервале $[ql, qr)$.

Замечание 14. Аналогично ситуациям с $\gcd$ и $\sqrt{}=$, так как у нас есть операция прибавления на отрезке, числа потенциально могут возрасти до $O(qC)$. Мы подразумеваем, что $q < poly(C)$, чтобы об этом не беспокоиться, но безусловно если об этом беспокоиться, ничего кардинально не изменится.

11.2. Решение

Как бы это ни было удивительно, но решение абсолютно идентично решению предыдущей задачи. *breakCondition* нам не понадобится и останется стандартным из обычного дерева отрезков, а *tagCondition* — это условие $max_v - min_v \leq 1$. 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 *tagCondition* выполнилось, мы делаем точно такие же изменения, как и для операции взятия квадратного корня. Если $\left\lfloor \frac{\max_v}{x} \right\rfloor = \left\lfloor \frac{\min_v}{x} \right\rfloor$, то нужно всем числам на отрезке присвоить $\left\lfloor \frac{\max_v}{x} \right\rfloor$, а если $\left\lfloor \frac{\max_v}{x} \right\rfloor \neq \left\lfloor \frac{\min_v}{x} \right\rfloor$, то тогда $\max_v = kx$ для некоторого k и $\min_v = kx - 1$. При этом

$\left\lfloor \frac{\max_v}{x} \right\rfloor = k$, и $\left\lfloor \frac{\min_v}{x} \right\rfloor = k - 1$, так что надо просто ко всем числам на отрезке прибавить $k - kx$.

11.3. 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 $O((n + q \log n) \log C)$. 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.

Лемма 8. *Если a, b и x — неотрицательные целые числа, при этом $a \geq b+2$ и $x \geq 2$, то $a - b \geq \frac{4}{3} \cdot \left(\left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor \right) + \frac{1}{3}$.*

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

1. $n \leq 1$. То есть $\left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor \leq 1$, так что неравенство, которое нам надо доказать, превращается в $a - b \geq \frac{4}{3} \cdot 1 + \frac{1}{3} = \frac{5}{3}$. Это верно из-за условия на то, что $a \geq b + 2$.

2. $n \geq 2$. В этом случае мы знаем, что $a \geq (n + m) \cdot x$ и $b \leq mx + x - 1$, поэтому $a - b \geq (n + m) \cdot x - (mx + x - 1) = nx - x + 1$. 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:

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

$n \geq 2$ и $x - \frac{4}{3} > 0$, поэтому заменим в левой части:

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

Если раскрыть левую часть, получится:

$$2x - 2 \geq x$$

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

□

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

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

И возьмем логарифмы от обеих частей:

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

То есть мы доказали, что потенциал уменьшился хотя бы на $\log(\frac{4}{3}) \approx 0.415$, и асимптотика решения доказана.

11.4. Улучшенная оценка

Попытаемся получше оценить асимптотику в частном случае. Пусть числа, на которые мы делим, имеют ограничения снизу, то есть x из запроса первого типа всегда не меньше какой-то константы $k \leq C$. Докажем, что в этом случае асимптотика алгоритма принимает вид $O((n + q \log n) \log_k C)$, что весьма логично, ведь если бы мы смотрели не на логарифмы разности в

потенциале, а просто на логарифмы чисел, то они бы уменьшались на $\log(k)$ при делении.

Для любого константного значения k эта асимптотика равна уже доказанной, поэтому для простоты давайте считать, что $k \geq 4$.

Также давайте немного уточним наш потенциал. Кроме логарифмов разностей будем еще следить за пометками. Скажем, что вершина v помечена, если $max_v > min_v + 1$, то есть если бы в ней не выполнилось *tagCondition* при посещении. Тогда несложно понять, что изначально помеченных вершин не больше $O(n)$, и новая пометка может появиться только если разность между минимумом и максимумом в вершине возросла, то есть увеличился потенциал этой вершины, а это может произойти только в обычных вершинах, поэтому за все время появилось максимум $O(q \log n)$ меток, поэтому за все время могло пропасть максимум $O(n + q \log n)$ меток. Тогда докажем, что каждый раз, когда мы посещаем дополнительную вершину, либо из нее пропадает метка, либо ее потенциал уменьшается хотя бы на $\log(\sqrt{x}) = \frac{1}{2}\log x \geq \frac{1}{2}\log k$. Тогда первый вариант случается максимум $O(n + q \log n)$ раз, а второй тогда аналогично доказательству обычной версии задачи дает вклад во время работы $O(n \log C / \log k + q \log n \log C / \log k) = O(n \log_k C + q \log n \log_k C)$, что доминирует $O(n + q \log n)$ и будет являться нашей асимптотикой. Тогда остается лишь доказать, что если из вершины не пропала метка, то потенциал сильно уменьшился. В этом нам поможет следующая лемма.

Лемма 9. *Если a, b и x — неотрицательные целые числа, при этом $\left\lfloor \frac{a}{x} \right\rfloor \geq \left\lfloor \frac{b}{x} \right\rfloor + 2$ и $x \geq 4$, то $a - b + 1 \geq \left(\left\lfloor \frac{a}{x} \right\rfloor - \left\lfloor \frac{b}{x} \right\rfloor + 1\right) \cdot \sqrt{x}$, что как раз таки эквивалентно тому, что либо разность между $\left\lfloor \frac{a}{x} \right\rfloor$ и $\left\lfloor \frac{b}{x} \right\rfloor$ маленькая, и тогда пропадает метка, либо она большая, и тогда потенциал уменьшается на $\log(\sqrt{x})$.*

Доказательство. Обозначим $\left\lfloor \frac{b}{x} \right\rfloor = m$ и $\left\lfloor \frac{a}{x} \right\rfloor = n + m$. При этом $n \geq 2$ по условию. Тогда $b = mx + l$, где $0 \leq l < x$ и $a = (n + m) \cdot x + k$, где $0 \leq k < x$. Мы знаем, что $a \geq (n + m) \cdot x$ и $b \leq mx + x - 1$, поэтому $a - b + 1 \geq (n + m) \cdot x - (mx + x - 1) + 1 = nx - x + 2$. Мы хотим доказать, что это не меньше $(n + 1)\sqrt{x}$.

$$nx - x + 2 \geq (n+1)\sqrt{x}$$

Что эквивалентно

$$n(x - \sqrt{x}) + 2 \geq x + \sqrt{x}$$

$n \geq 2$ и $x - \sqrt{x} > 0$, поэтому заменив n на 2 мы сделаем неравенство только сильнее.

$$2 \cdot (x - \sqrt{x}) + 2 \geq x + \sqrt{x}$$

$$2 + x \geq 3\sqrt{x}$$

Это верно для $x = 4$, а потом производная левой части всегда больше производной правой части, поэтому тем самым мы доказали утверждение.

□

12. Сценарии применения в реальной жизни

Основная цель данной работы — теоретический анализ алгоритмов, однако для обоснования ценности полученных результатов в данном разделе мы обсудим практические применения представленных алгоритмов. Как уже было неоднократно указано, в репозитории проекта [21] можно найти реализации алгоритмов, представленных в работе, а также тестовые примеры, на которых можно убедиться в эффективности обсужденных решений не только в теории, но и на практике. Остается лишь понять, в каких именно жизненных ситуациях могут возникать решенные задачи.

Начнем с немного философской стороны. Какие математические операции являются наиболее стандартными? Наверное, спросив разных людей, мы получим разные ответы, но наверняка ответ будет чем-то похож на следующий список: равенство, сложение, вычитание, умножение, деление, минимум, максимум, возведение в степень, взятие остатка по модулю, НОД и т.д. Многие из этих операций можно воспринимать как операции обновления: присвоить, прибавить, вычесть и т.д. Но как обсуждалось в начале статьи, лишь малая часть из этих операций обновления доступна обычному дереву отрезков. Как можно убедиться, в данной работе мы посвятили внимание многим стандартным операциям, а те, которые мы не обсудили, часто тоже несложно решаются при помощи Segment Tree Beats. Таким образом, решение представленных задач кажется автору очень естественным направлением исследования.

Теперь перейдем к практическим применениям алгоритмов. Представим, что мы — фонд высокочастотной торговли. Перед нами стоит задача анализа цены какого-то актива во времени. Пускай, мы иногда получаем инсайдерскую информацию о том, что в какой-то период времени в будущем какой-то игрок на рынке будет предоставлять этот актив на продажу за определенную стоимость. Также иногда нам нужно узнавать, какова оптимальная цена на данный актив в произвольный данный момент времени и т.п. В таком случае в каждый момент времени оптимальная цена — это минимум всех представленных цен, поэтому когда на отрезке $[l, r)$ появляется предложение с ценой x, это эквивалентно операции $\min = x$ на этом отрезке. В силу

своих особенностей, индустрия высокочастотного трейдинга особенно чувствительна к скорости работы алгоритмов, поэтому безусловно улучшения с $O(\sqrt{n})$ на запрос до $O(\log n)$ на запрос являются существенными. Кроме того, возможно, даже анализ степени логарифма (как в задаче Extended Ji Driver Segment Tree) может играть малую роль, которая в итоге может привести к большим финансовым успехам.

Заключение

В данной работе были исследованы алгоритмы из семейства Segment Tree Beats, получены улучшенные оценки на время работы для задач Extended Ji Driver Segment Tree, GCD Ji Driver Segment Tree, а также задач с запросами взятия квадратного корня и деления нацело на отрезке. А именно:

- Для задач семейства Ji Driver Segment Tree были более точно сформулированы асимптотики.
- Для задачи Extended Ji Driver Segment Tree асимптотика была улучшена с $O(n \log n + q \log^2 n)$ до $O(n \log n + q_1 \log n + q_2 \log^2 n + q_3 \log n)$.
- Для задачи с запросами $\min =$ и $\gcd$ асимптотика была улучшена с $O((n+q) \log n \log C)$ до $O((n+q)(\log n + \log C))$.
- Для задачи GCD Ji Driver Segment Tree асимптотика была улучшена с $O(n \log n \log C + q \log^2 n \log C)$ до $O((n + q \log n)(\log n + \log C))$ и $O((n + q_1 + q_2 \log n + q_3)(\log n + \log C))$, если быть еще точнее.
- Для задачи с запросами $\sqrt{} =$, $+ =$, $=$ и $\sum$ асимптотика была улучшена с $O((n + q \log n) \log C)$ до $O((n + q \log n) \log \log C)$.
- Для задачи с запросами $/ =$, $+ =$, $=$ и $\sum$ при условии, что деление производится на числа, не меньшие k, асимптотика была улучшена с $O((n + q \log n) \log C)$ до $O((n + q \log n) \log_k C)$.

Кроме того, были написаны реализации представленных алгоритмов на языке C++, а также разработаны сценарии применения предложенных алгоритмов на практике.

Продолжением исследования может быть дальнейшее изучение задачи Extended Ji Driver Segment Tree в надежде приравнять теоретическую верхнюю и нижнюю оценки на время работы (что, как следствие, может приравнять эти оценки и для задачи GCD Ji Driver Segment Tree), а также анализ новых задач, для которых может быть применима парадигма Segment Tree Beats.

Кроме того хочется прокомментировать несколько подходов к дальнейшему изучению задачи Extended Ji Driver Segment Tree:

- Одним из возможных направлений в исследовании может быть изучение вопроса о времени работы при условии, что входные данные сгенерированы случайно (в соответствии с каким-то распределением). Это потенциально может помочь для доказательства логарифмического времени работы.

- В общем случае можно считать, что все запросы выполняются не на отрезках массива, а на префиксах. Действительно, сумма на отрезке — это разность сумм префиксов, прибавление на отрезке можно выразить через прибавление на префиксе правой границы и вычитание на префиксе левой границы, а операцию $\min =$ на отрезке можно выразить как вычитание большой константы на префиксе левой границы, применение операции $\min =$ на префиксе правой границы и обратное прибавление большой константы на префиксе левой границы. Возможно, в силу большей простоты операции прибавления на префиксе, это может как-то помочь в анализе алгоритма.

- Хранить кроме первого и второго максимума на отрезке еще и третий, четвертый и так далее кажется бесполезным. Безусловно, хранить их можно, однако это не поможет нам ослабить условие *tagCondition*. Все еще если $x \leq secondMax$, мы не можем пересчитать значения в данной вершине, ведь в этом случае второй максимум стал первым, третий стал вторым, и так далее. Тогда если мы храним k максимумов, то k-й максимум стал $k-1$-м, а $k+1$-й стал k-м, но $k+1$-й максимум мы не храним и не имеем возможности легко его найти. Аналогичная проблема возникает при попытке пересчитать количества максимумов.

Список литературы

[1] M. Bender and M. Farach-Colton, The LCA Problem Revisited, In Proceedings of LATIN 2000, LNCS 1776, 88–94, 2000.

[2] B.Ya Ryabko; A fast on-line adaptive code. IEEE Trans.on Inform.Theory,v.28, n 1, Jul 1992 pp. 1400 - 1404.

[3] Ruyi Ji. Segment Tree Beats. URL: http://www.doc88.com/p-6744902151779.html (дата обр. 26.04.2022).

[4] Комментарий от Xin Huang. URL: https://codeforces.com/blog/entry/54750?#comment-387839 (дата обр. 20.05.2022).

[5] Ruyi Ji. A simple introduction to "Segment tree beats". URL: https://codeforces.com/blog/entry/57319 (дата обр. 26.04.2022).

[6] Rajarshi Basu. Segment Tree Beats - An introduction. URL: http://codingwithrajarshi.blogspot.com/2018/03/segment-tree-beats-introduction.html (дата обр. 02.05.2022).

[7] Егор Горбачев. Segment Tree Beats. URL: https://peltorator.ru/posts/segment_tree_beats/ (дата обр. 02.05.2022).

[8] Benjamin Qi, Dustin Miao. Segment Tree Beats. URL: https://usaco.guide/adv/segtree-beats?lang=cpp (дата обр. 02.05.2022).

[9] Kamil Debowski. Segment tree beats | IOI preparation #6. URL: https://youtu.be/wFqKgrW1IMQ (дата обр. 02.05.2022).

[10] David Harmeyer. AlgorithmsThread 4: Segment Tree Beats. URL: https://youtu.be/_TGNYtkWAsQ (дата обр. 02.05.2022).

[11] Егор Горбачев. Segment Tree Beats: Segment Tree On Steroids. Part 1. URL: https://youtu.be/NwkO73jGSPA (дата обр. 02.05.2022).

[12] Егор Горбачев. Segment Tree Beats: Дерево Отрезков На Стероидах. Часть 1. URL: https://youtu.be/58csqxAD8vM (дата обр. 02.05.2022).

[13] Ista. Segment Tree Beats | Advanced | RMQ & LCA. URL: https://youtu.be/UJyBHCXa-1g (дата обр. 02.05.2022).

[14] Codeforces Round #250 (Div. 1). D. Ребенок и последовательность. URL: https://codeforces.com/problemset/problem/438/D (дата обр. 02.05.2022).

[15] HackerRank. Box Operations. URL: https://www.hackerrank.com/challenges/box-operations/problem (дата обр. 02.05.2022).

[16] CSAcademy. And or Max. URL: https://csacademy.com/contest/round-70/task/and-or-max (дата обр. 02.05.2022).

[17] Codeforces Round #743 (Div. 1). Телебашни. URL: https://codeforces.com/contest/1572/problem/F (дата обр. 02.05.2022).

[18] Manthan, Codefest 17. Нагайна. URL: https://codeforces.com/contest/855/problem/F (дата обр. 02.05.2022).

[19] Максим Иванов. Дерево отрезков. URL: https://e-maxx.ru/algo/segment_tree (дата обр. 26.04.2022).

[20] Максим Ахмедов. Задача RMQ – 2. Дерево отрезков. URL: https://habr.com/ru/post/115026/ (дата обр. 26.04.2022).

[21] Репозиторий с реализациями алгоритмов. URL: https://github.com/Peltorator/segment-tree-beats (дата обр. 26.04.2022).