Специальные структуры данных (подборка материалов)

Алексей Толстиков

11 сентября 2019 г.

1 Интервальные структуры данных

1.1 Абстрактная структуда данных

Абстрактные структуры данных предназначены для удобного хранения и доступа к информации. Они предоставляют удобный интерфейс для типичных операций с хранимыми объектами, скрывая детали реализации от пользователя. Конечно, это весьма удобно и позволяет добиться большей модульности программы. Абстрактные структуры данных иногда делят на две части: интерфейс, набор операций над объектами, который называют АТД (абстрактный тип данных) и реализацию.

Ниже, где возможно, собственно АТД будут отделены от их реализации:

1.1.1 Статический массив

При создании статического массива указывается его размер и память сразу выделяется под все данные.

Интерфейс:

- Получить элемент с номером i
- Записать элемент с номером i

1.1.2 Динамический массив

При создании динамического массива указывается его начальный размер, это память сразу выделяется под данные.

Интерфейс:

- Добавить элемент в конец массива
- Удалить элемент в конце массив
- Узнать размер массива
- Получить элемент с номером i
- Записать элемент с номером i

1.1.3 Список, стек, очередь, дек

Классические структуры данных, интерфейсы этих структур данных.

Придумать метод релизации интерфейса очереди, имея только возможность создавать сущности с интерфейсом стека.

1.2 Оценка трудоемкости операций

Время работы алгоритма на входных данных X — это число элементарных операций, выполняемых алгоритмом при обработке X.

1.2.1 Трудоемкость в худшем случае

Пусть имеется некоторый алгоритм A, тогда $T_A(n)$ — максимальное время работы алгоритма A на входе размера n. $T_A(n)$ — трудоемкость алгоритма A.

Определите трудоемкость (предложите алгоритм):

- Подсчета числа делителей числа
- Сортировки массива произвольных целых чисел
- Сортировки массива возрастов пользователей социальной сети
- Поиска элемента в отсортированном массиве
- Слияние двух отсортированных массивов
- Слияние двух отсортированных массивов, расположенных в последовательной памяти, не используя дополнительной памяти
- Поиска k элементов $a_1, \ldots, a_k$ $(a_i \leq a_{i+1})$ в отсортированном массиве B.

1.2.2 Амортизационный анализ

Амортизационный анализ (англ. amortized analysis) — метод подсчета времени, требуемого для выполнения последовательности операций над структурой данных. При этом время усредняется по всем выполняемым операциям, и анализируется средняя производительность операций в худшем случае.

Такой анализ чаще всего используется, чтобы показать, что даже если некоторые из операций последовательности являются дорогостоящими, то при усреднении по всем операциям средняя их стоимость будет небольшой за счёт низкой частоты встречаемости. Оценка, даваемая амортизационным анализом, не является вероятностной: это оценка среднего времени выполнения операций для худшего случая.

Средняя амортизационная стоимость операций — величина a, находящаяся по формуле: $a = \frac{\sum_i^n t_i}{n}$, где t_1,t_2, $\ldots$,t_n — время выполнения операций 1, 2, $\ldots$, n, совершённых над структурой данных.

Амортизационный анализ использует следующие методы:

1. Метод усреднения (метод группового анализа).
2. Метод потенциалов.
3. Метод предоплаты (метод бухгалтерского учета).

В методе усреднения амортизационная стоимость операций определяется напрямую по формуле, указанной выше: суммарная стоимость всех операций алгоритма делится на их количество.

Введём для каждого состояния структуры данных величину Φ — потенциал. Изначально потенциал равен Φ_0, а после выполнения i-й операции — Φ_i. Стоимость i-й операции обозначим $a_i = t_i + \Phi_i - \Phi_{i-1}$. Пусть n — количество операций, m — размер структуры данных. Тогда средняя амортизационная стоимость операций $a = O(f(n, m))$, если выполнены два условия:

1. Для любого i: $a_i = O(f(n, m))$.
2. Для любого i: $\Phi_i = O(n \cdot f(n, m)$.

Представим, что использование определенного количества времени равносильно использованию определенного количества монет (плата за выполнение каждой операции). В методе предоплаты каждому типу операций присваивается своя учётная стоимость. Эта стоимость может быть больше фактической, в таком случае лишние монеты используются как резерв для выполнения других операций в будущем, а может быть меньше, тогда гарантируется, что текущего накопленного резерва достаточно для выполнения операции. Для доказательства оценки средней амортизационной стоимости $O(f(n,m))$ нужно построить учётные стоимости так, что для каждой операции она будет составлять $O(f(n,m))$. Тогда для последовательности из n операций суммарно будет затрачено $n \cdot O(f(n,m))$ монет, следовательно, средняя амортизационная стоимость операций будет $a = \frac{\sum_i^n t_i}{n} = \frac{n \cdot O(f(n,m))}{n} = O(f(n,m))$.

Определите амортизационную стоимость (предложите алгоритм):

- Стек с multipop.
- Двоичный счётчик.
- КМП алгоритм поиска вхождения шаблона в текст.
- Построение всех перестановок.
- Рекурсивный обход дерева.
- Рекурсивный обход бинарного поискового дерева с выводом ключей k ($x \leq k \leq y$).

1.2.3 *Стек, очередь, дек с поиском минимального элемента

Необходимо добавить операцию поиска минимального элемента в классический интерфейс.

1.2.4 Оценка времени работы на случайных входных данных

Среднее время работы — усредненное время работы алгоритма по всем входным данным размерности n. Т.е. в среднем случае мы рассматриваем различные входные данные как равновероятные.

Среднее время работы (предложите алгоритм):

- Поиск в перестановке.
- Поиск в отсортиванном массиве.
- Поиск вхождения шаблона в строку.
- Сортировка перестановки.
- Сортировка результатов централизованного тестирования.
- Проверка числа от 1 до n на простоту.

1.2.5 Примеры структур данных с балансировкой времени работы между различными операциями

AddElementh + **ExtractMin** для алгоритма Дейкстры: в зависимости от плотности графа можно использовать кучу или тривиальную реализацию на массиве.

1.2.6 Примеры структур данных с балансировкой между объемом используемой памяти и временем работы

Очень часто перед нами стоит вопрос, нужно ли стараться сжать данные (упаковать), чтобы потом распаковывать на каждый запрос.

1.2.7 *Стек, очередь, дек с поиском минимального элемента

Как изменятся стандартные операции. Какой амортизированной оценки удалось достичь?

2 Дерево отрезков

Дерево отрезков (англ. Segment tree) — это структура данных, которая позволяет за асимптотику $O(logN)$ реализовать любые операции, определяемые на множестве, на котором данная операция ассоциативна, и существует нейтральный элемент относительно этой операции, то есть на моноиде. Например, суммирование на множестве натуральных чисел, поиск минимума на любом числовом множестве, перемножение матриц на множестве матриц размера $N \times N$, объединение множеств, поиск наибольшего общего делителя на множестве целых чисел и многочленов.

При этом дополнительно возможно изменение элементов массива: как изменение значения одного элемента, так и изменение элементов на целом подотрезке массива, например разрешается присвоить всем элементам $a[l..r]$ какое-либо значение, либо прибавить ко всем элементам массива какое-либо число. Структура занимает $O(n)$ памяти, а ее построение требует $O(n)$ времени.

2.1 Построение дерева

Пусть исходный массив a состоит из n элементов. Для удобства построения увеличим длину массива a так, чтобы она равнялась ближайшей степени двойки, т.е. 2^k, где $2^k \geq n$. Это сделано, для того чтобы не допустить обращение к несуществующим элементам массива при дальнейшем процессе построения. Пустые элементы необходимо заполнить нейтральными элементами моноида. Тогда для хранения дерева отрезков понадобится массив t из 2^{k+1} элементов, поскольку в худшем случае количество вершин в дереве можно оценить суммой $n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 < 2n$, где $n = 2^k$. Таким образом, структура занимает линейную память.

Процесс построения дерева заключается в заполнении массива t. Заполним этот массив таким образом, чтобы i-й элемент являлся бы результатом некоторой бинарной операции (для каждой конкретной задачи своей) от элементов с номерами $2i+1$ и $2i+2$, то есть родитель являлся результатом бинарной операции от своих сыновей (обозначим в коде эту операцию как `OP`). Один из вариантов — делать рекурсивно. Пусть у нас имеются исходный массив a, а также переменные tl и tr, обозначающие границы текущего полуинтервала. Запускаем процедуру построения от корня дерева отрезков ($i = 0$, $tl = 0$, $tr = n$), а сама процедура построения, если её вызвали не от листа, вызывает себя от каждого из двух сыновей и суммирует вычисленные значения, а если её вызвали от листа — то просто записывает в себя значение этого элемента массива (Для этого у нас есть исходный массив a). Асимптотика построения дерева отрезков составит, таким образом, $O(n)$.

Выделяют два основных способа построения дерева отрезков: построение снизу и построение сверху. При построении снизу алгоритм поднимается от листьев к корню (Просто начинаем заполнять элементы массива t от большего индекса к меньшему, таким образом при заполнении элемента i его дети $2i + 1$ и $2i + 2$ уже будут заполнены, и мы с легкостью посчитаем бинарную операцию от них), а при построении сверху спускается от корня к листьям. Особенные изменения появляются в реализации запросов к таким деревьям отрезков.

2.2 Реализация запроса в дереве отрезков сверху

Данная операция позволяет выполнять запросы на дереве отрезков, причем алгоритм запускается от корня и рекурсивно идет сверху вниз.

Алгоритм

Замечание. Используем в алгоритме не отрезки, а полуинтервалы (левая граница включительно, а правая — нет).

Пусть есть уже построенное дерево отрезков и идет запрос на полуинтервале $[a \ldots b)$.

В качестве параметров рекурсий передаем следующие переменные:

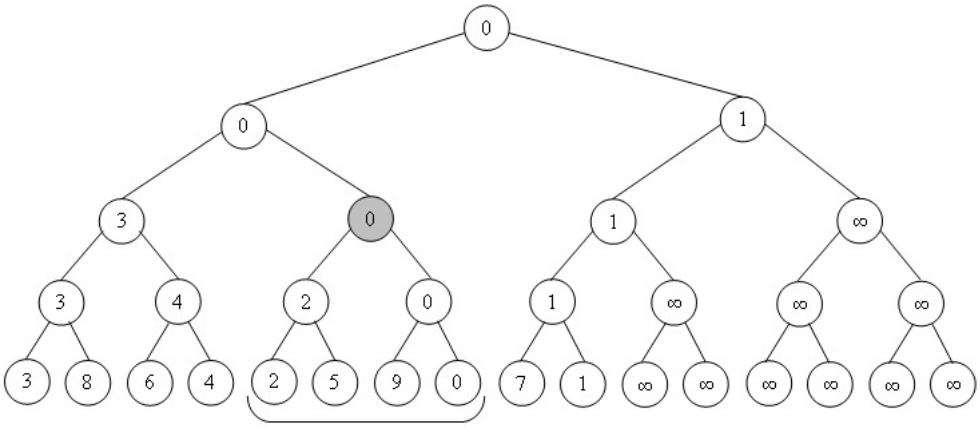

Рис. 1: Дерево отрезков

- *node* — номер (в массиве с деревом отрезков) текущей вершины дерева.
- a, b — левая и правая границы запрашиваемого полуинтервала.

Пусть l, r — это левая и правая границы полуинтервала, за которые "отвечает"наша вершина. Запустим рекурсивную процедуру от всего полуинтервала (то есть от корневой вершины).

Для текущего состояния проверяем следующие условия :

- Если текущий полуинтервал не пересекается с искомым, то возвращаем нейтральный элемент. *Например*: текущий $[1 \ldots 3)$, а искомый $[3 \ldots 5)$.
- Если текущий полуинтервал лежит внутри запрашиваемого полуинтервала, то возвращаем значение в текущей вершине. *Например*: текущий $[2 \ldots 3)$, а искомый $[2 \ldots 4)$.
- Иначе переходим к рекурсивным вызовам функций от детей вершины. При этом возвращаем значение на текущем полуинтервале, как функцию (соответствующую типу нашего запроса) от результатов выполнения на детях.

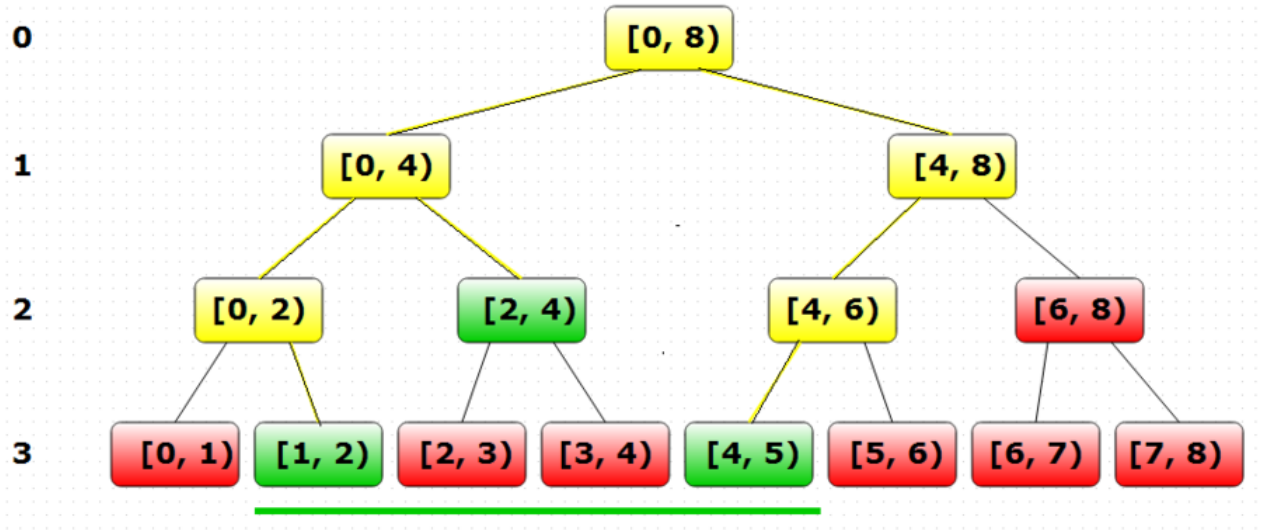

Рис. 2: Дерево отрезков. Интервальный запрос

Так как на каждом уровне дерева рекурсия может дойти до не более, чем двух вершин (иначе бы нашлось две рядом стоящие вершины одного уровня, объединение которых дало отрезок, за который отвечает вершина предыдущего уровня), а всего уровней $\log n$, то операция выполняется за $O(\log n)$.

2.3 Примеры использования дерева отрезков

2.3.1 Поиск минимума/максимума

Немного изменим условие задачи: будем производить запрос минимума/максимума на отрезке.

Тогда в дереве отрезков надо изменить способ вычисления пометки во внутренних вершинах: использовать минимум/максимум из пометок сынове. По сути пометка вершины дерева будет равна минимальному/максимальному элемент на всем отрезке, за который отвечает вершины дерева.

2.3.2 Поиск минимума/максимума и количества раз, которое он встречается

Задача аналогична предыдущей, только теперь помимо минимума/максимума требуется также возвращать количество его вхождений. Эта задача встаёт естественным образом, например, при решении с помощью дерева отрезков такой задачи: найти количество наидлиннейших возрастающих подпоследовательностей в заданном массиве.

Для решения этой задачи в каждой вершине дерева отрезков будем хранить пару чисел: кроме минимума/максимума количество его вхождений на соответствующем отрезке. Тогда при построении дерева мы должны просто по двум таким парам, полученным от сыновей текущей вершины, получать пару для текущей вершины.

Объединение двух таких пар в одну стоит выделить в отдельную функцию, поскольку эту операцию надо будет производить и в запросе модификации, и в запросе поиска минимума/максимума.

2.3.3 Поиск наибольшего общего делителя / наименьшего общего кратного

Т.е. мы хотим научиться искать НОД/НОК всех чисел в заданном отрезке массива.

Это довольно интересное обобщение дерева отрезков получается абсолютно таким же путём, как и деревья отрезков для суммы/минимума/максимума: достаточно просто хранить в каждой вершине дерева НОД/НОК всех чисел в соответствующем отрезке массива.

Обратим внимание, что в каждой вершине при обновлении информации нужно будет выполнить более сложную операцию – вычисление наибольшего общего делителя пометок сыновей.

2.3.4 Подсчёт количества нулей, поиск k-го нуля

Мы хотим научиться отвечать на запрос количества нулей в заданном отрезке массива, а также на запрос нахождения k-го нулевого элемента.

Снова немного изменим данные, хранящиеся в дереве отрезков: будем хранить теперь в пометке вершины количество нулей, встречающихся в соответствующих отрезках массива. Понятно, как поддерживать и использовать эти данные в функциях построения дерева, обновления элемента и запроса — тем самым мы решили задачу о количестве нулей в заданном отрезке массива.

Теперь научимся решать задачу о поиске позиции k-го вхождения нуля в массиве. Для этого будем спускаться по дереву отрезков, начиная с корня, и переходя каждый раз в левого или правого сына в зависимости от того, в каком из отрезков находится искомый k-ый ноль. В самом деле, чтобы понять, в какого сына нам надо переходить, достаточно посмотреть на значение, записанное в левом сыне: если оно больше либо равно k, то переходить надо в левого сына (потому что в его отрезке есть как минимум k нулей), а иначе — переходить в правого сына.

При реализации можно отсечь случай, когда k-го нуля не существует, ещё при входе в функцию, вернув в качестве ответа, например, -1.

2.3.5 Поиск префикса массива с заданной суммой

Мы хотим по данному значению x быстро найти такое i, что сумма первых i элементов массива a больше либо равна x (считая, что массив a содержит только неотрицательные числа).

Эту задачу можно решать бинарным поиском, вычисляя каждый раз внутри него сумму на том или ином префиксе массива, но это приведёт к решению за время $O(\log^2 n)$.

Вместо этого можно воспользоваться той же самой идеей, что и в предыдущем пункте, и искать искомую позицию одним спуском по дереву: переходя каждый раз в левого или правого сына в зависимости от величины суммы в левом сыне. Тогда ответ на запрос поиска будет представлять собой один такой спуск по дереву, а, следовательно, будет выполняться за $O(\log n)$.

2.3.6 Поиск подотрезка с максимальной суммой

По-прежнему на вход даётся массив a, и поступают запросы (l, r), которые означают: найти такой подотрезок $a[l' \dots r']$, что $l \le l', r' \le r$, и сумма этого отрезка $a[l' \dots r']$ максимальна. Запросы модификации отдельных элементов массива допускаются. Элементы массива могут быть отрицательными (и, например, если все числа отрицательны, то оптимальным подотрезком будет пустой — на нём сумма равна нулю).

Это весьма нетривиальное обобщение дерева отрезков получается следующим образом. Будем хранить в каждой вершине дерева отрезков четыре величины: сумму на этом отрезке, максимальную сумму среди всех префиксов этого отрезка, максимальную сумму среди всех суффиксов, а также максимальную сумму подотрезка на нём. Иными словами, для каждого отрезка дерева отрезков ответ на нём уже предпосчитан, а также дополнительно ответ посчитан среди всех отрезков, упирающихся в левую границу отрезка, а также среди всех отрезков, упирающихся в правую границу.

Подойдём к этому с рекурсивной точки зрения: пусть для текущей вершины все четыре значения в левом сыне и в правом сыне уже подсчитаны, посчитаем их теперь для самой вершины. Заметим, что ответ в самой вершине равен:

- либо ответу в левом сыне, что означает, что лучший подотрезок в текущей вершине целиком помещается в отрезок левого сына,
- либо ответу в правом сыне, что означает, что лучший подотрезок в текущей вершине целиком помещается в отрезок правого сына,
- либо сумме максимального суффикса в левом сыне и максимального префикса в правом сыне, что означает, что лучший подотрезок лежит своим началом в левом сыне, а концом — в правом.

Значит, ответ в текущей вершине равен максимуму из этих трёх величин. Пересчитывать же максимальную сумму на префиксах и суффиксах совсем несложно. Необходимо аккуратно рассмотреть случаи, когда левый/правый сын целиком входит в префикс или суффикс.

Осталось разобраться с ответом на запрос. Для этого мы так же, как и раньше, спускаемся по дереву, разбивая тем самым отрезок запроса $[l \dots r]$ на несколько подотрезков, совпадающих с отрезками дерева отрезков, и объединяем ответы в них в единый ответ на всю задачу. Тогда понятно, что работа ничем не отличается от работы обычного дерева отрезков, только надо вместо простого суммирования/минимума/максимума значений использовать функцию комбинирования информации, описанной выше.

2.4 Сохранение всего подмассива в каждой вершине дерева отрезков

Это отдельный подраздел, стоящий особняком от остальных, поскольку в каждой вершине дерева отрезков мы будем хранить не какую-то сжатую информацию об этом подотрезке (сумму, минимум, максимум и т.п.), а все элементы массива, лежащие в этом подотрезке. Таким образом, корень дерева отрезков будет хранить все элементы массива, левый сын корня — первую половину массива, правый сын корня — вторую половину, и так далее.

Самый простой вариант применения этой техники — когда в каждой вершине дерева отрезков хранится отсортированный список всех чисел, встречающихся в соответствующем отрезке. В более сложных вариантах хранятся не списки, а какие-либо структуры данных, построенные над этими списками (set, map и т.д.). Но все эти методы объединяет то, что в каждой вершине дерева отрезков хранится некая структура данных, имеющая в памяти размер порядка длины соответствующего отрезка.

Первый естественный вопрос, встающий при рассмотрении деревьев отрезков этого класса — это объём потребляемой памяти. Утверждается, что если в каждой вершине дерева отрезков хранится список всех встречающихся на этом отрезке чисел, либо любая другая структура данных размера того же порядка, то в сумме всё дерево отрезков будет занимать $O(n \log n)$ ячеек памяти. Потому что каждое число $a[i]$ попадает в $O(\log n)$ отрезков дерева отрезков (хотя бы потому, что высота дерева отрезков есть $O(\log n)$), а на каждом уровне дерева число встречается ровно один раз.

Итак, несмотря на кажущуюся расточительность такого дерева отрезков, он потребляет памяти не сильно больше обычного дерева отрезков.

Рассмотрим несколько типичных применений такой структуры данных.

2.4.1 Поиск наименьшего числа, больше либо равного заданного, в указанном отрезке. Запросов модификации нет

Требуется отвечать на запросы следующего вида: (l, r, x), что означает найти минимальное число в отрезке $a[l \dots r]$, которое больше либо равно x.

Построим дерево отрезков, в котором в каждой вершине будем хранить отсортированный список всех чисел, встречающихся на соответствующем отрезке. Например, корень будет содержать массив a в отсортированном виде. Для построения такого дерева подойдём к задаче, как обычно, с точки зрения рекурсии: пусть для левого и правого сыновей текущей вершины эти списки уже построены, и нам требуется построить этот список для текущей вершины. При такой постановке вопроса становится почти очевидно, что это можно сделать за линейное время: нам просто надо объединить два отсортированных списка в один, что делается одним проходом по ним с двумя указателями (операция слияния из сортировки слиянием).

Построенное таким образом дерево отрезков будет занимать $O(n \log n)$ памяти. А благодаря такой реализации время его построения также есть величина $O(n \log n)$ — ведь каждый список строится за линейное относительно его размера время. (Кстати говоря, здесь прослеживается очевидная аналогия с алгоритмом сортировки слиянием: только здесь мы сохраняем информацию со всех этапов работы алгоритма, а не только итог.)

Теперь рассмотрим ответ на запрос. Будем спускаться по дереву, как это делает стандартный ответ на запрос в дереве отрезков, разбивая наш отрезок $a[l \dots r]$ на несколько подотрезков (порядка $O(\log n)$ штук). Понятно, что ответ на всю задачу равен минимуму среди ответов на каждом из этих подотрезков. Поймём теперь, как отвечать на запрос на одном таком подотрезке, совпадающем с некоторой вершиной дерева.

Итак, мы пришли в какую-то вершину дерева отрезков и хотим посчитать ответ на ней, т.е. найти минимальное число, больше либо равное данного x. Для этого нам всего лишь надо выполнить бинарный поиск по списку, посчитанному в этой вершине дерева, и вернуть первое число из этого списка, больше либо равное x.

Таким образом, ответ на запрос в одном подотрезке происходит за $O(\log n)$, а весь запрос обрабатывается за время $O(\log^2 n)$.

2.4.2 Поиск наименьшего числа, больше либо равного заданного, в указанном отрезке. Допускаются запросы модификации

Задача аналогична предыдущей, только теперь разрешены запросы модификации: обработать присвоение $a[i] = y$.

Решение также аналогично решению предыдущей задачи, только вместо простых списков в каждой вершине дерева отрезков мы будем хранить сбалансированный список, который позволяет быстро искать требуемое число, удалять его, а также вставлять новое число. Учитывая, что вообще говоря число во входном массиве могут повторяться, оптимальным выбором является структура данных multiset.

Построение такого дерева отрезков происходит примерно так же, как и в предыдущей задаче, только теперь надо объединять не отсортированные списки, а multiset, что приведёт к тому, что асимптотика построения ухудшится до $O(n \log^2 n)$.

Ответ на запрос поиска вообще практически эквивалентен приведённому случаю,, только теперь lower_bound надо нужно искать в multiset (вершинах из рабиения отрезка на элементарные).

Наконец, запрос модификации. Для его обработки мы должны спуститься по дереву, внеся изменения во все $O(\log n)$ списков, содержащих затрагиваемый элемент. Мы просто удаляем старое значение этого элемента (не забыв, что нам не надо удалить вместе с ним все повторы этого числа) и вставляем его новое значение.

Обработка этого запроса происходит также за время $O(\log^2 n)$.

2.5 Обновление на отрезке

Выше рассматривались только задачи, когда запрос модификации затрагивает единственный элемент массива. На самом деле, дерево отрезков позволяет делать запросы, которые применяются к целым отрезкам подряд идущих элементов, причём выполнять эти запросы за то же время $O(\log n)$.

2.5.1 Прибавление на отрезке

Начнём рассмотрение деревьев отрезков такого рода с самого простого случая: запрос модификации представляет собой прибавление ко всем числам на некотором подотрезке $a[l \dots r]$ некоторого числа x. Запрос чтения — по-прежнему считывание значения некоторого числа $a[i]$.

Чтобы делать запрос прибавления эффективно, будем хранить в каждой вершине дерева отрезков, сколько надо прибавить ко всем числам этого отрезка целиком. Например, если приходит запрос "прибавить ко всему массиву $a[0 \dots n-1]$ число 2 то мы поставим в корне дерева число 2. Тем самым мы сможем обрабатывать запрос прибавления на любом подотрезке эффективно, вместо того чтобы изменять все $O(n)$ значений.

Если теперь приходит запрос чтения значения того или иного числа, то нам достаточно спуститься по дереву, просуммировав все встреченные по пути значения, записанные в вершинах дерева.

2.5.2 Присвоение на отрезке

Пусть теперь запрос модификации представляет собой присвоение всем элементам некоторого отрезка $a[l \dots r]$ некоторого значения p. В качестве второго запроса будем рассматривать считывание значения массива $a[i]$.

Чтобы делать модификацию на целом отрезке, придётся в каждой вершине дерева отрезков хранить, покрашен ли этот отрезок целиком в какое-либо число или нет (и если покрашен, то хранить само это число). Это позволит нам делать "запаздывающее"обновление дерева отрезков: при запросе модификации мы, вместо того чтобы менять значения во множестве вершин дерева отрезков, поменяем только некоторые из них, оставив флаги "покрашен"для других отрезков, что означает, что весь этот отрезок вместе со своими подотрезками должен быть покрашен в этот цвет.

Итак, после выполнения запроса модификации дерево отрезков становится, вообще говоря, неактуальным — в нём остались недовыполненными некоторые модификации.

Например, если пришёл запрос модификации "присвоить всему массиву $a[0 \dots n-1]$ какое-то число то в дереве отрезков мы сделаем единственное изменение — пометим корень дерева, что он покрашен целиком в это число. Остальные же вершины дерева останутся неизменёнными, хотя на самом деле всё дерево должно быть покрашено в одно и то же число.

Предположим теперь, что в том же дереве отрезков пришёл второй запрос модификации — покрасить первую половину массива $a[0 \dots n/2]$ в какое-либо другое число. Чтобы обработать такой запрос, мы должны покрасить целиком левого сына корня в этот новый цвет, однако перед тем как сделать это, мы должны разобраться с корнем дерева. Тонкость здесь в том, что в дереве должно сохраниться, что правая половина покрашена в старый цвет, а в данный момент в дереве никакой информации для правой половины не сохранено.

Выход таков: произвести проталкивание информации из корня, т.е. если корень дерева был покрашен в какое-либо число, то покрасить в это число его правого и левого сына, а из корня эту

отметку убрать. После этого мы можем спокойно красить левого сына корня, не теряя никакой нужной информации.

Обобщая, получаем: при любых запросах с таким деревом (запрос модификации или чтения) во время спуска по дереву мы всегда должны делать проталкивание информации из текущей вершины в обоих её сыновей. Можно понимать это так, что при спуске по дереву мы применяем запаздывающие модификации, но ровно настолько, насколько это необходимо (чтобы не ухудшить асимптотику с $O(\log n)$).

При реализации это означает, что нам надо сделать функцию, которой будет передаваться вершина дерева отрезков, и она будет производить проталкивание информации из этой вершины в обоих её сыновей. Вызывать эту функцию следует в самом начале функций обработки запросов (но не вызывать её из листьев, ведь из листа проталкивать информацию не надо, да и некуда).

Для получения значения элемента $a[i]$ будет искать самую высокую "покрашеную"вершину на пути к элементу, или возьмем само значение элемента, если покрашеных вершин нет.

2.5.3 Прибавление на отрезке, запрос максимума

Запросом модификации снова будет запрос прибавления ко всем числам некоторого подотрезка одного и того же числа, а запросом чтения будет нахождение максимума в некотором подотрезке.

Тогда в каждой вершине дерева отрезков надо будет дополнительно хранить максимум на всём этом подотрезке. Но тонкость здесь заключается в том, как надо пересчитывать эти значения.

Например, пусть произошёл запрос "прибавить ко всей первой половине, т.е. $a[0 \dots n/2]$, число 2". Тогда в дереве это отразится записью числа 2 в левого сына корня. Как теперь посчитать новое значение максимума в левом сыне и в корне? Здесь становится важно не запутаться — какой максимум хранится в вершине дерева: максимум без учёта прибавления на всей этой вершине, или же с учётом его. Выбрать можно любой из этих подходов, но главное — последовательно использовать его везде. Например, при первом подходе максимум в корне будет получаться как максимум из двух чисел: максимум в левом сыне плюс прибавление в левом сыне, и максимум в правом сыне плюс прибавление в нём. При втором же подходе максимум в корне будет получаться как прибавление в корне плюс максимум из максимумов в левом и правом сыновьях, но прибавление на отрезке кроме изменения поментки добавленной суммы будет изменять и максимум на отрезке.

2.6 Задания по теме

2.6.1 Теоретическое задание

В решении этой задачи необходимо использовать Дерево отрезков БЕЗ ИНТЕРВАЛЬНОЙ МОДИФИКАЦИИ.

Опишите алгоритм решения следующей задачи. Перед вами полоска длины N, разбитая на N одинаковых квадратиков, пронумерованных слева направо числами от 0 до $N-1$. За одну операцию изменения (метод, предоставляемый интерфейсом вашей структуры данных) вы должны перекрасить все клетки с номерами от L до R включительно: белые должны стать черными, а черные - белыми. В запросах поиска необходимо определить цвет клетки в позиции X. Изначально задается раскраска полоски в черный и белый цвет.

Интерфейс:

- Init(vector colors);
- ChangeColor(L, R);
- GetColor(X).

Время работы метода Init должно быть линейным, а методов ChangeColor и GetColor должно быть $O(\log N)$.

2.6.2 Практическое задание

Задан массив A из n положительных чисел $a_i (1 \leq a_i \leq 10^9)$. Необходимо уметь выполнять 4 вида запросов:

- 1 p v — присвоить p-му элементу массива значение v.
- 2 l r $(l \leq r)$ — прибавить единицу ко всем числам на отрезке $[l..r]$.
- 3 l r $(l \leq r)$ — найти сумму четных чисел на отрезке $[l..r]$.
- 4 l r $(l \leq r)$ — найти сумму нечетных чисел на отрезке $[l..r]$.

3 Meet-in-the-middle

Meet-in-the-middle разбивает задачу пополам и решает всю задачу через частичный расчет половинок. Он работает следующим образом: переберем все возможные значения x и запишем пару значений $(x, f(x))$ в множество. Затем будем перебирать всевозможные значения y, для каждого из них будем вычислять $g(y)$, которое мы будем искать в нашем множестве. Если в качестве множества использовать отсортированный массив, а в качестве функции поиска — бинарный поиск, то время работы нашего алгоритма составляет $O(XlogX)$ на сортировку, и $O(YlogX)$ на двоичный поиск, что дает в сумме $O((X+Y)logX)$.

3.1 Задача о нахождении четырех чисел с суммой равной нулю

Дан массив целых чисел A. Требуется найти любые 4 числа, сумма которых равна 0 (одинаковые элементы могут быть использованы несколько раз).

Например: $A = (2, 3, 1, 0, -4, -1)$. Решением данной задачи является, например, четверка чисел $3 + 1 + 0 - 4 = 0$ или $0 + 0 + 0 + 0 = 0$.

Наивный алгоритм заключается в переборе всевозможных комбинаций чисел. Это решение работает за $O(N^4)$. Теперь, с помощью Meet-in-the-middle мы можем сократить время работы до $O(N^2logN)$.

Для этого заметим, что сумму $a + b + c + d = 0$ можно записать как $a + b = -(c + d)$. Мы будем хранить все N^2 пар сумм $a + b$ в массиве sum, который мы отсортируем. Далее перебираем все N^2 пар сумм $c + d$ и проверяем бинарным поиском, есть ли сумма $-(c + d)$ в массиве sum. Кроме этого можно отсортировать оба списка $a + b$ и $-(c + d)$, а замет двигаться по ним параллельно.

Итоговое время работы $O(N^2logN)$.

Если вместо отсортированного массива использовать хэш-таблицу, то задачу можно будет решить за время $O(N^2)$.

3.2 Задача о рюкзаке

Классической задачей является задача о наиболее эффективной упаковке рюкзака. Каждый предмет характеризуется весом $(w_i \leq 10^9)$ и ценностью $(cost_i \leq 10^9)$. В рюкзак, ограниченный по весу, необходимо набрать вещей с максимальной суммарной стоимостью. Для ее решения изначальное множество вещей N разбивается на два равных(или примерно равных) подмножества, для которых за приемлемое время можно перебрать все варианты и подсчитать суммарный вес и стоимость, а затем для каждого из них найти группу вещей из первого подмножества с максимальной стоимостью, укладывающуюся в ограничение по весу рюкзака. Сложность алгоритма $O(2^{\frac{N}{2}} \times N)$. Память $O(2^{\frac{N}{2}})$.

Разделим наше множество на две части. Подсчитаем все подмножества из первой части и будем хранить их в массиве $first$. Отсортируем массив $first$ по весу. Далее пройдемся по этому массиву и оставим только те подмножества, для которых не существует другого подмножества с меньшим весом и большей стоимостью. Очевидно, что подмножества, для которых существует другое, более легкое и одновременно более ценное подмножество, можно удалять. Таким образом в массиве $first$ мы имеем

подмножества, отсортированные не только по весу, но и по стоимости. Тогда начнем перебирать все возможные комбинации вещей из второй половины и находить бинарным поиском удовлетворяющие нам подмножества из первой половины, хранящиеся в массиве $first$.

Итоговое время работы $O(2^{\frac{N}{2}} \times (N + \log 2^{\frac{N}{2}})) = O(2^{\frac{N}{2}} \times N)$.

3.3 Задача о нахождении кратчайшего расстояния между двумя вершинами в графе

Еще одна задача, решаемая Meet-in-the-middle — это нахождение кратчайшего расстояния между двумя вершинами, зная начальное состояние, конечное состояние и то, что длина оптимального пути не превышает N. Стандартным подходом для решения данной задачи, является применение алгоритма обхода в ширину. Пусть из каждого состояния у нас есть K переходов, тогда бы мы сгенерировали K^N состояний. Асимптотика данного решения составила бы $O(K^N)$. Meet-in-the-middle помогает снизить асимптотику до $O(K^{\frac{N}{2}})$.

Алгоритм решения:

- Сгенерируем BFS-ом все состояния, доступные из начала и конца за $\frac{N}{2}$ или меньше ходов.
- Найдем состояния, которые достижимы из начала и из конца.
- Найдем среди них наилучшее по сумме длин путей.

Таким образом, BFS-ом из двух концов, мы сгенерируем максимум $O(K^{\frac{N}{2}})$ состояний.

3.4 Дискретное логарифмирование

Необходимо добавить материал! Алгоритм Гельфонда—Шенкса

3.5 Задания по теме

3.5.1 Теоретическое задание

Предложите алгоритм, который по заданному числу N находит количество четверок целых положительных чисел (x, y, z, t) - решений уравнения:

$$x^2 + y^2 + z^2 + t^2 = N.$$

Время работы алгоритма: $O(N)$ или $O(NlogN)$.

3.5.2 практическое задание

Дано $n(n \leq 40)$ предметов, каждый из них характеризуется весом и ценностью.

Необходимо собрать рюкзак весом не больше W такой, что суммарная ценность всех вещей в рюкзаке будет максимально возможной. Другими словами, необходимо выбрать подмножество предметов с ограничением на суммарную массу с максимальной ценностью.

4 Битовое представление множества

Во многих случаях оказывается очень удобным представить множество с помощь битовой маски. Такое представление позволяет выполнять логические опирации в векторном стиле, т.к. машинное слово, как правило, содержит 32 или 64 бита и позволяет за одну инструкцию процесса выполнить попарных битовые операции “И”, “ИЛИ”, “логическое НЕ”, “исключающее ИЛИ”. Кроме этого в некоторых случаях удобным может оказаться и использование сдвига множества на некоторое число.

Далее рассмотри несколько примеров, когда битовое представление может быть очень удобным и эффективным.

4.1 Задача о коммивояжере

Задача о коммивояжере (англ. Travelling salesman problem, TSP) – задача, в которой коммивояжер должен посетить N городов, побывав в каждом из них ровно по одному разу и завершив путешествие в том городе, с которого он начал. В какой последовательности ему нужно обходить города, чтобы общая длина его пути была наименьшей?

Можно решить задачу перебором всевозможных перестановок. Для этого нужно сгенерировать все $N!$ всевозможных перестановок вершин исходного графа, подсчитать для каждой перестановки длину маршрута и выбрать минимальный из них. Но тогда задача оказывается неосуществимой даже для достаточно небольших N. Сложность алгоритма $O(N! \times N)$.

Задача о коммивояжере представляет собой поиск кратчайшего гамильтонова цикла в графе. Зафиксируем начальную вершину s и будем искать гамильтонов цикл наименьшей стоимости — путь от s до s, проходящий по всем вершинам (кроме первоначальной) один раз. Т.к. искомый цикл проходит через каждую вершину, то выбор s не имеет значения. Поэтому будем считать $s = 0$.

Подмножества вершин будем кодировать битовыми векторами, обозначим $mask_i$ значение i-ого бита в векторе $mask$.

Обозначим $d[i][mask]$ как наименьшую стоимость пути из вершины i в вершину 0, проходящую (не считая вершины i) единожды по всем тем и только тем вершинам j, для которых $mask_j = 1$ (т.е. $d[i][mask]$ уже найденный оптимальный путь от i-ой вершины до 0-ой, проходящий через те вершины, где $mask_j = 1$. Если $mask_j = 0$,то эти вершины еще не посещены).

Алгоритм поиска цикла будет выглядеть следующим образом:

- Начальное состояние — когда находимся в 0-й вершине, ни одна вершина не посещена, а пройденный путь равен 0 (т.е. $i = 0$ и $mask = 0$).
- Для остальных состояний ($i \neq 0$ или $mask \neq 0$) перебираем все возможные переходы в i-ую вершину из любой посещенной ранее и выбираем минимальный результат.
- Если возможные переходы отсутствуют, решения для данной подзадачи не существует (обозначим ответ для такой подзадачи как ∞).

Стоимостью минимального гамильтонова цикла в исходном графе будет значение $d[0][2^n - 1]$ – стоимость пути из 0-й вершины в 0-ю, при необходимости посетить все вершины. Данное решение требует $O(2^n \times n)$ памяти и $O(2^n \times n^2)$ времени.

Для того, чтобы восстановить сам путь, воспользуемся соотношением $d[i][mask] = w(i, j) + d[j][mask - 2^j]$, которое выполняется для всех ребер, входящих в минимальный цикл. Начнем с состояния $i = 0$, $mask = 2^n - 1$, найдем вершину j, для которой выполняется указанное соотношение, добавим j в ответ, пересчитаем текущее состояние как $i = j$, $mask = mask - 2^j$. Процесс заканчивается в состоянии $i = 0$, $mask = 0$.

4.2 Класс bitset

Тип объекта, который хранит последовательность, состоящую из фиксированного числа битов, предоставляющих компактный способ хранения флагов для набора элементов или условий. Класс bitset поддерживает операции над объектами типа bitset, которые содержат коллекцию битов и обеспечивают доступ в константном времени к каждому биту.

4.2.1 Транзитивное замыкание

Алгоритм Флойда-Уоршала можно значительно ускорить, если строки обрабатывать в векторном стиле.

4.2.2 Решение систем линейных уравнение по модулю 2

Алгоритм Гаусса можно значительно ускорить, если строки обрабатывать в векторном стиле.

Аналогичный трюк можно провернуть, если нужно найти ранг двоичной матрицы.

4.2.3 Задача о рюкзаке

Если внимательно посмотреть на решение задачи о рюкзаке, то получается, что новая строка матрицы возможных весов (которые можно набрать, используя какие-то из k первых предметов) является результатом "логического ИЛИ" предыдущей строки и ее же, но сдвинутой на w_k единиц вправо.

Список литературы

- Томас Кормен. Алгоритмы. Построение и анализ. - Санкт-Петербург, 2005.
- https://ru.wikipedia.org/wiki/Структура_данных
- http://algolist.manual.ru/ds/basic/
- https://neerc.ifmo.ru/wiki/index.php?title=Амортизационный_анализ
- http://neerc.ifmo.ru/wiki/index.php?title=Дерево_отрезков._Построение
- http://www.e-maxx-ru.1gb.ru/algo/segment_tree
- http://neerc.ifmo.ru/wiki/index.php?title=Реализация_запроса_в_дереве_отрезков_сверху
- http://neerc.ifmo.ru/wiki/index.php?title=Задача_о_рюкзаке
- http://neerc.ifmo.ru/wiki/index.php?title=Meet-in-the-middle