Содержание

АТП Гос (ИВТ: Формалки + Оси)

Disclaymer: доверять этому конспекту или нет выбирайте сами

1. Алгоритмы и структуры данных

1.1. Динамический массив. Амортизационный анализ. Учётная оценка времени добавления элемента в динамический массив.

Определение 1.1.1: Пусть f, g – произвольные функции, тогда
$$f(x) = O(g(x)) \Leftrightarrow \exists M > 0 : \exists x_0 : \forall x \geq x_0 : |f(x)| \leq M|g(x)|$$

Определение 1.1.2: Пусть время n последовательных операций над некоторой структурой данных составит $t_1, ..., t_n$, тогда говорят, что **амортизированная стоимость** или **учётное время** операции составляет $t^* = \frac{1}{n}\sum_{i=1}^{n} t_i$.

Замечание 1.1.1: Далее, амортизированная сложность будет обозначаться со звездочкой, например, $O^*(1)$

Замечание 1.1.2 (Динамический массив): Интерфейс динамического массива:
- Обращение по индексу за $O(1)$
- Добавить элемент в конец за $O^*(1)$

Замечание 1.1.3 (Объяснение амортизационной сложности добавления): Очевидно, что при заполнении массива нам нужно делать реаллокацию. (Которая условно бесплатная, но копирование всех элементов на новое место – линейно).

Но, оказывается, если при каждой реаллокации мы будем увеличивать массив в два раза, то средняя стоимость добавления элемента в массив будет константной

Для доказательства будем использовать метод монеток.
- Пусть мы только что совершили реаллокацию – у нас есть $\frac{n}{2}$ свободного места и столько же уже добавленных элементов
- Пусть добавляем один из $\frac{n}{2}$ новых элементов – пусть его индекс $i \geq \frac{n}{2}$. Тогда, потратив монетку на добавление без реалокации, заложим по одной монетке на будущее копирование i-го и $i - \frac{n}{2}$-элемента.
- Таким образом, к заполнению буфера из n элементов, мы заложим по монетке за каждый элемент и после новой реалокации сможем условно бесплатно скопировать их на новое место, потратив монетки.

1.2. Связные списки. Стек, очередь, дека и их реализации

Замечание 1.2.1 (Список): Мы хотим от списка следующее:

Операция	**Время**
Вставка в известное место	O(1)
Удаление из известного места	O(1)
Поиск	O(N)
Обращение по индексу	O(N)

Наш список будет хранить цепочку из узлов, где каждый указывает на следующего за ним, а последний указывает в никуда.

Поиск и обращение по индексу требуют линейного прохода, но при этом вставка или удаление элемента – это создание узла и переприсвоение указателей.

Существует также двусвязный список – хранит в себе два указателя на узел после и позади нас. Благодаря дополнительному указателю, получаем возможность работы с обоими концами списка, не теряя в асимптотике.

Замечание 1.2.2 (Стек): Мы хотим от стека следующее:

Операция	**Время**
Вставка в начало	O(1)
Удаление из начала	O(1)
Узнать размер	O(1)

Стек можно реализовать на односвязном списке, однако для быстрого получения размера следует завести счётчик, изменяемый при вставке/удалении.

Замечание 1.2.3 (Очередь): Мы хотим от очереди следующее:

Операция	**Время**
Вставка в начало	O(1)
Удаление из конца	O(1)
Узнать размер	O(1)

Очередь тривиально реализуется на двухсвязном списке, однако есть способ реализовать её на двух стеках.

У нас будут два стека: входной и выходной. Вставка будет происходить в перывй, а удаление из второго. В случае удаление из пустого выходного стека требуется переложить все элементы из выходного стека в выходной (получим развёрнутый выходной стек).

Удаление из такой очереди уже будет $O^*(1)$, так как на каждый элемент хватит по 3 монетки – на его добавление, переброс в выходной стек и удаление.

Замечание 1.2.4 (Дека): Деку иногда называют двусторонним стеком или двусторонней очередью:

Операция	**Время**
Вставка в начало или в конец	O(1)
Удаление из начала или из конца	O(1)
Узнать размер	O(1)

Тривиально реализуется на двухсвязном списке.

1.3. Быстрая сортировка. Поиск порядковой статистики методом «Разделяй и властвуй»

Замечание 1.3.1 (QuickSort): Алгоритм быстрой сортировки:
- Случайно выбираем опорный элемент (pivot)
- Всё, что меньше опорного элемента перекинуть влево, а что больше – вправо
- Вызвать рекурсивно от правой и левой половины.

Минусы:
- Использует доппамять, так как нужно хранить стек рекурсии
- Под любую стратегию выбора опорного элемента можно построить контрданные так, чтобы работало квадратичное время с линейной доппамятью.

Теорема 1.3.1: Среднее время работы быстрой сортировки составляет $O(N \log N)$, если опорный элемент выбирается равновероятно.

Определение 1.3.1: **k-й порядковой статистикой** массива называют элемент, который после сортировки будет стоять на k-м месте.

Замечание 1.3.2 (QuickSelect): Алгоритм быстрого поиска:
- Выбрать опорный элемент
- Провести разбиение
- Если индекс опорного элемента окажется больше, чем k – то ищем слева k -ю статистику, иначе, справа – $k - i - 1$-ю.

Теорема 1.3.2: Среднее время работы алгоритма выше составляет $O(N)$, если опорный элемент выбирается равновероятно.

1.4. Сортировка слиянием. Поразрядные сортировки.

Замечание 1.4.1 (MergeSort): Воспользуемся стратегией разделяй и властвуй:
- Разбиить массив на две примерно равные половины
- Разбиваем массивы дальше пополам, пока не дойдём до массива из двух элементов, там всё тривиально
- Сольём два отсортированных массива в один большой

Два отсортированных массива сливаются методом двух указателей – $O(N)$ времени и доппамяти.

Для подсчёта общего времени работы рассмотрим дерево рекурсии. В нём $\log N$ уровней, при этом на каждом уровне после слияния отсортированная часть массива будет увеличиваться примерно в два раза.

Значит нам потребуется $\log N$ слияний – итоговое время $O(N \log N)$. А доппамять линейна.

Определение 1.4.1: Сортировка называется **стабильной**, если равные элементы относительно компаратора не поменяют своего взаимного расположения после сортировки.

Замечание 1.4.2 (Поразрядная сортировка): Рассмотрим сортировку подсчётом, считать будем в массиве P:

- Пройдёмся по исходному массиву и запишем в $P[i]$ число объектов с ключом (по чему сортируем) i
- Посчитаем префиксные суммы массива P, тогда мы знаем, начиная с какого индекса в сортированном массиве надо писать структуру с ключом i.
- Идём по изначальному массиву слева направо и вписываем каждый элемент в сортированный массив, согласно префиксным суммам.

Данная сортировка, очевидно, стабильна, однако при большом количестве разных чисел, расходы на массив подсчёта станут огромными.

Тогда научимся сортировать числа из большого диапазона! Рассмотрим массив произвольных uint32_t.

Давайте сортировать данные числа как двоичные строки, сначала сортируем по убыванию последнего байта, потом стабильно по убыванию второго байта etc. Работает это за $O(Nk)$ времени, где k - это число байтов, то есть константа. Массив подсчёта, очевидно, будет небольшим - 256.

1.5. Двойная куча и сортировка кучей. Слияние k отсортированных массивов с помощью кучи.

Определение 1.5.1: Дерево называется **подвешенным**, если у него есть выделенная вершина, называемая **корнем** дерева.

Определение 1.5.2: Дерево называется **бинарным**, если у каждой вершины степень не более трёх, то есть один родитель и не более двух детей.

Определение 1.5.3: Бинарное дерево называется **полным**, если все его уровни полностью заполнены.

Замечание 1.5.1 (Бинарная пирамида): Хотим такую структуру:

Операция	**Время**
Добавление элемента	$O(\log N)$
Удаление минимума	$O(\log N)$
Чтение минимума	$O(1)$

Бинарную пирамиду будем реализовывать на полном бинарном дереве, при этом будем поддерживать **свойство пирамиды** – все сыновья вершины строго больше её самой.

Тогда, очевидно, в корне всегда будет минимум, который и будем брать за константу.

Определим две вспомогательные операции – просеивание вниз и просеивание вверх. Просеивание вниз старается «утопить» элемент как можно ниже, меняя его местами с меньшим из сыновыей, не нарушая свойство пирамиды.

Для вставки будем хранить помимо указателя на корень ещё и указатель на самый правый узел, у которого меньше двух сыновый. Тогда подвесим новый элемент к этому узлу и потом просеим его вверх.

Если же таких узлов нет, то весь нижний уровень занят, а значит подвесим новый элемент к самому левому узлу и также просеим вверх.

Для удаления поменяем местами корень и крайний узел, а затем просеим вниз элемент в корне, а лист с минимумом просто удалим.

Замечание 1.5.2 (HeapSort): Для сортировки массива построим на нём пирамиду и N раз извлечём минимум. Итоговое время $O(N \log N)$.

Замечание 1.5.3 (Слияние k отсортированных массивов): Обобщение метода двух указателей:

- Из каждого из k массивов берём по первому элементу, добавляем в кучу
- Берём из кучи минимальный, он будет следующим элементов слитого массива, добавляем в кучу следующий элемент из исходного массива только что взятого элемента.
- Повторяем до опустошения всех массивов

Очевидно, сложность $O(Nk \log k)$

1.6. Хэш-таблица, полиномиальная хэш-функция

Определение 1.6.1: Пусть U – множество рассматриваемых объектов, тогда $h : U \to \{0, 1, ..., k-1\}$ называется **хэш-функцией**

Определение 1.6.2: Элементы $x, y \in U, x \neq y$ образуют **коллизию**, если $h(x) = h(y)$

Определение 1.6.3: Заведём массив размера k и скажем, что элемент v есть в множестве, если по индексу $h(v)$ уже занята ячейка.

Такая схема называется **прямой адресацией**.

Определение 1.6.4: **Бакетом** называют нечто, хранящее все элементы, образующие коллизию.

Замечание 1.6.1 (Хэш-таблица на цепочках): На практике, элементы, образующие коллизию, добавляются в бакет, например, список.

Давайте посчитаем средний размер бакета:

Считаем, что хэш-функция отображает из $\{0, 1, ..., n-1\}$ в $\{0, 1, ..., k-1\}$, а вероятность коллизиции равномерная:

$$P(\{h(x) = h(y)\}) = \begin{cases} 1, x = y \\ \frac{1}{k}, x \neq y \end{cases}$$

Тогда для произвольного бакета, соответствующего некоторому элементу $q \in \{0, ..., n\}$, матожидание его размера:

$$\mathbb{E}L_q = \mathbb{E}\sum_{i=1}^{n} \mathbb{I}(\{h(q) = h(i)\}) =$$

$$P(\{h(q) = h(q)\}) + \sum_{i=1, i\neq q}^{n} P(\{h(q) = h(i)\}) = 1 + \frac{n-1}{k} \leq 1 + \frac{n}{k}$$

Определение 1.6.5: **Коэффициентом загруженности** называют величину $\alpha = \frac{n}{k}$.

Заметим, что мы знаем эту величину уже при определении хэш-функции, а значит стоимость всех операций заведомо ограничивается некоторой константой, зависимой от α.

Замечание 1.6.2 (Полиномиальное хеширование): Приведём пример некоторой хэш-функции в отношении строк. Для строки p размера m такой хэш определён следующий образом:

$$h(p[1:m]) = \left(\sum_{i=1}^{m} p[i]x^{i-1}\right) \bmod q$$

где q – некоторое простое число, а x – число от 0 до $q-1$.

1.7. Динамическое программирование: общая идея, линейная динамика, матричная, динамика на отрезках

Определение 1.7.1: **Динамическое программирование** – способ решения алгоритмических задач, который основывается на переходе от частного к общему.

Замечание 1.7.1 (Нахождение наибольшей возрастающей подпоследовательности): Построим массив d, где $d[i]$ – это длина наибольшей возрастающей подпоследовательности, оканчивающейся в элементе с индексом i.

Пусть искомая последовательность – a.

Массив будет заполняться следующим образом:

$$d[i] = \begin{cases} 1, i{=}0 \\ 1 + \max_{j=0\ldots i-1; a[j]<a[i]} d[j] \end{cases}$$

Причём в случае максимума по пустому множеству будет возвращаться 0.

Данная задачка является примером линейного ДП с квадратичной сложностью.

Замечание 1.7.2 (Нахождение наибольшей общей подпоследовательности): Построим матрицу d, где $d[i, j]$ – это длина наибольшей общей подпоследовательности, оканчивающаяся в первой последовательности на i-м индексе, а во второй – на j-м.

Пусть искомые последовательность – s_1, s_2.

Тогда заполним матрицу:

$$d[n_1, n_2] = \begin{cases} 0, n_1{=}0 \vee n_2{=}0 \\ d[n_1{-}1, n_2{-}1]{+}1, s_1[n_1{-}1]{=}s_2[n_2{-}1] \\ \max(d[n_1{-}1, n_2], d[n_1, n_2{-}1]), \text{ else} \end{cases}$$

Данная задачка является примером матричного ДП с квадратичной сложностью.

Замечание 1.7.3 (ДП на подотрезках): В рассмотренных задачах у нас была «индукция» по индексам массива или матрицы, однако часто можно встретить «индукцию» по длинам подотрезков.

Например, пусть есть строка S и надо найти длину максимальной подпоследовательности палиндрома.

Пусть $f(l, r)$ – ответ для подстроки $S[l : r + 1]$, тогда сразу определим базу $f(l, l) = 1$ и $f(l, r) = 0, r < l$.

Сразу определимся, что ответом будет $f(0, |S| - 1)$.

Теперь научимся пересчитывать ответ. У нас могут быть две ситуации:

- Если $S[l] = S[r]$, то нам выгодно добавить эти символы к итоговому палиндрому, а значит $f(l, r) = f(l + 1, r - 1) + 2$.
- Если же $S[l] \neq S[r]$, то просто прокинем максимум из подстрок $f(l, r) = \max(f(l + 1, r), f(l, r - 1))$.

1.8. RMQ. Sparse table. Дерево отрезков.

Замечание 1.8.1 (Разреженная таблица): Разреженная таблица – двумерная структура данных $ST[i][j]$, построенная на бинарной, коммутативной, ассоциативной и идемпотентной операции F, для которой

$$ST[i][j] = F(...F(F(A[i], A[i+1]), ...), ..., A[i+2^j-1]); j \in [0...\log N]$$

Объём памяти, занимаемый таблицей, равен $O(N \log N)$, и заполненными являются только те элементы, для которых $i + 2^j \leq N$.

Теперь заметим, что для отрезка $[l, r]$ верно, что

$$F(A[l], ..., A[r]) = F(ST[l][j], ST[r-2^j+1][j]), j = \lfloor \log_2(r-l+1) \rfloor$$

Определение 1.8.1: **RMQ или range min query** – запрос минимума на подотрезке. Решается с помощью Sparse Table с $F = \min$

Замечание 1.8.2 (Дерево отрезков): Дерево отрезков способно за $O(\log n)$ получать на подотрезке результат любой операции, которая ассоциативна, коммутативна и имеет нейтральный элемент.

Опишем нерекурсивной построение дерева на массиве длины N. Для удобства будем считать, что $\log_2 N \in \mathbb{N}$, иначе дозаполним до степени двойки нейтральными элементами.

Теперь заведём массив длины $2N-1$ и будем его заполнять таким образом, что последние N элементов будут элементами исходного массива, а первые $N-1$ элементов заполним, как $t[i] = F(t[2i+1], t[2i+2])$.

Для обновления листа $i \geq \frac{N}{2}$ обновим его самого, а потом будем подниматься по дереву (На каждой итерации делаем $i' = \lfloor \frac{i}{2} \rfloor$) и пересчитаем все значения, зависимые от изменённого.

1.9. LCA. Двоичные подъёмы и сведение к RMQ

Определение 1.9.1: Пусть дано дерево T, подвешенную за вершину r.

Тогда назовём **наименьшим общим предком (LCA)** двух вершин u, v такую вершину $X = \text{LCA}(u, v)$, то она лежит на путях $u \to r, v \to r$, при этом такая вершина глубже всех подходящих.

Замечание 1.9.1 (Метод двоичных подъёмов): Сделаем предпосчёт двумерной матрицы $dp[v][v]$ – номер вершины, в которую мы придём, если пройдём из вершины v вверх по дереву 2^i шагов.

Пусть номер родители вершины v — $p[v]$, её глубина – $d[v]$, причём если v – корень, то $p[v] = v$.

Тогда для dp есть рекуррентная формула:

$$dp[v][i] = \begin{cases} p[v], i=0 \\ dp[dp[v][i-1]][i-1], i>0 \end{cases}$$

Заметим, что $i \leq \log_2 n$, так как иначе мы остаёмся в корне. Значит матрица занимает $O(N \log N)$ памяти.

Пусть $c = \text{LCA}(v, u), u \neq v$, тогда по определению $c \leq \min(d[v], d[u])$. Пусть $d[u] < d[v]$, тогда нам надо подняться вверх на $d[v] - d[u]$ шагов, для этого разобьём данную разность на степени двойки и поднимемся за $O(\log N)$.

Теперь будем считать, что $d[u] = d[v], u \neq v$. Остаётся тривиальный параллельный подъём до общего предка не более чем на $\log_2 N$ вверх.

Замечание 1.9.2 (Сведение LCA к RMQ): Заметим, что если w – LCA вершин u и v, то DFS из корня посетит сначала w, потом БОО зайдёт в u, затем снова посетит w и аналогично с v.

Поэтому нам достаточно знать для каждой вершины время её посещения обходом в глубину и её высоту:

- Массив Order, где $\text{Order}[i]$ – номер вершины, посещённой в момент времени i
- Массив высот h, где $h[i]$ – высота вершины $\text{Order}[i]$
- Массив First, где $\text{First}[v]$ – момент времени, когда вершина v была посещена впервые

Тогда для $\text{LCA}(u, v)$ нужно найти $\text{Order}[\min(h[\text{First}[u] : \text{First}[v]])]$, задачу поиска минимума на подотрезке как раз поможет решить RMQ.

1.10. Двоичное дерево поиска. Обходы в глубину и в ширину. Поиск ключа, наивные вставка и удаление ключа. АВЛ-дерево.

Определение 1.10.1: **Деревом поиска** называют дерево, в котором в поддереве левее все элементы не больше элемента в данном узле, а в поддереве правее – строго больше.

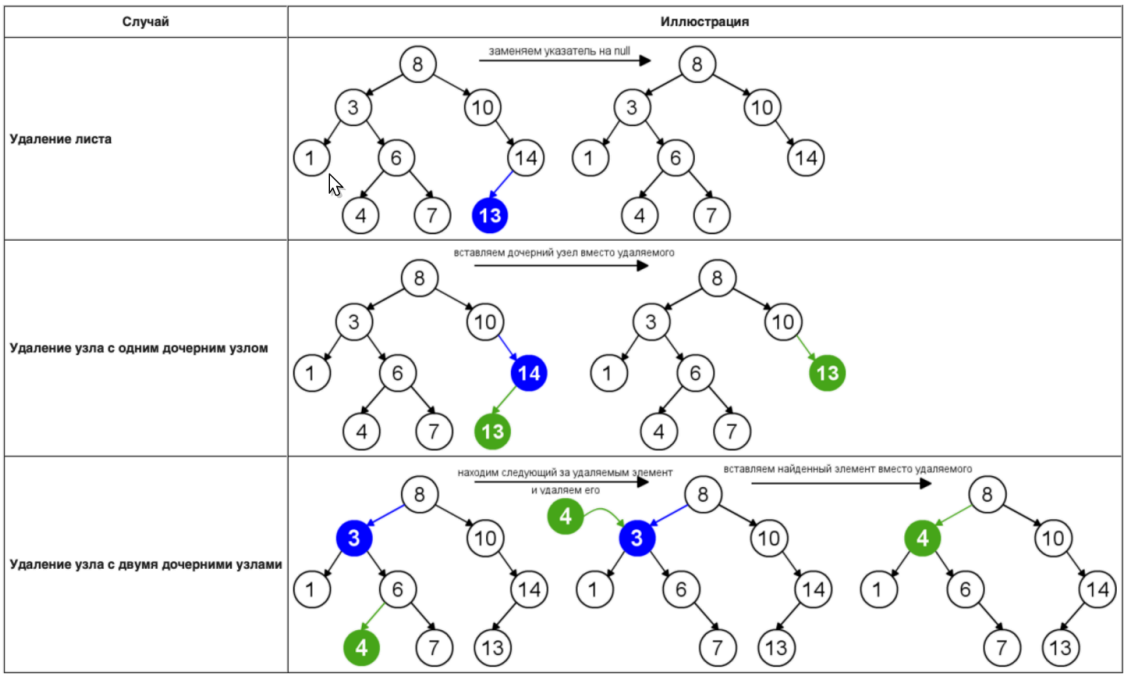
Пусть h – высота дерева, тогда мы хотим

Операция	Время
Вставка элемента	$O(h)$
Поиск элемента	$O(h)$
Удаление элемента	$O(h)$

Поиск в таком дереве тривиальный – начинаем с корня, если искомый ключ меньше – идём влево, иначе – вправо. Если элемент есть – мы его найдём, иначе – попытаемся зайти в несуществующий узел.

Для вставки мы пойдём аналогично поиску, но когда дойдём до несуществующего узла – то добавим его на это место.

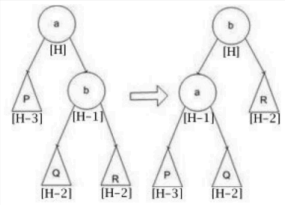
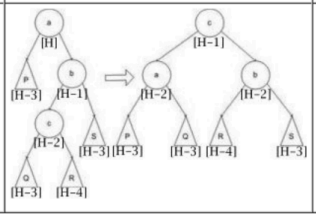
Иллюстрация удаления:



Определение 1.10.2: Дерево поиска является **AVL-деревом**, если для каждой вершины высота её правого и левого поддеревьев различаются не более, чем на единицу.

Теорема 1.10.1: Высота AVL-дерева равна $O(\log N)$.

Определение 1.10.3: **Балансировкой** AVL-дерева будем называть процесс, который из дерева поиска, в узле которого разных высот поддедревьев равна двум, переподвешивает детей и внуков так, чтобы выполнялось свойство AVL-дерева.

Тип вращения	Иллюстрация	Когда используется	Расстановка балансов
Малое левое вращение		$diff[a] = -2$ и $diff[b] = -1$ или $diff[a] = -2$ и $diff[b] = 0$.	$diff[a] = 0$ и $diff[b] = 0$ $diff[a] = -1$ и $diff[b] = 1$
Большое левое вращение		$diff[a] = -2, diff[b] = 1$ и $diff[c] = 1$ или $diff[a] = -2, diff[b] = 1$ и $diff[c] = -1$ или $diff[a] = -2, diff[b] = 1$ и $diff[c] = 0$.	$diff[a] = 0, diff[b] = -1$ и $diff[c] = 0$ $diff[a] = 1, diff[b] = 0$ и $diff[c] = 0$ $diff[a] = 0, diff[b] = 0$ и $diff[c] = 0$

Симметрично определяются два правых поворота.

После каждой операции изменения дерева, запускаем балансировку от нового узла до корня, причём баланс уже сбалансированных поддеревьев никак не изменится.

1.11. Декартово дерево. Декартово дерево по неявному ключу.

Определение 1.11.1: **Декартовым деревом** называют бинарное дерево, содержащее в себе пары (x_i, y_i), при этом данное дерево является деревом поиска по **ключам** и бинарной пирамидой по **приоритетам**.

У декартова дерева есть две фундаментальные операции: split и merge.

Замечание 1.11.1 (Split): Операция Split принимает на вход декартово дерево T и ключ k, а возвращает пару декартовых деревьев T_1, T_2 таких, что в T_1 все ключи не больше k, а в T_2 все ключи строго больше.

Пусть ключ в корне окажется меньше, чем ключ, по которому разрезаем, тогда:

- Левое поддерево T_1 совпадает с левым поддеревом T. Для нахождения правого поддерева T_1 рекурсивно по тому же ключу разрежем правого сына T на T_L, T_R. Тогда правым поддеревом T_1 будет T_L.
- T_2 совпадает с T_R.

Выполняется данная операция, очевидно, за $O(h)$.

Замечание 1.11.2 (Merge): Данная операция принимает на вход два декартовых дерева (T_1, T_2) таких, что все ключи в T_1 меньше ключей в T_2. Рассмотрим два случая:

- Приоритет корня левого поддерева больше приоритета корня правого. Тогда верно, что левое поддерево итогового дерева T совпадает с левым поддеревом T_1, а правой будет результатом слияния правого поддерева T_1 и T_2.
- Приоритет корня левого поддерева меньше приоритета корня правого. Тогда верно, что правое поддерево итогового дерева T совпадет с правым поддеревом T_2, а левое будет результатом слияния T_1 и левого поддерева T.

Данная операция также выполняется за $O(h)$.

Замечание 1.11.3 (Вставка и удаление): Обсудим вставку элемента с ключом k:

- $\text{Split}(T, k) =: (T_1, T_2)$
- Смотрим, совпадает ли наибольший элемент в T_1 с k. Если да, то мёрджим T_1, T_2, иначе идём дальше
- Создаём T_3 дерево-синглетон над вставляемым элементом и делаем $\text{Merge}(\text{Merge}(T_1, T_3), T_2)$

Теорема 1.11.1 (Глубина декартова дерева): В декартовом дереве из N узлов, приоритеты у которого являются случайными величинами с равномерным распределением, средняя глубина вершины $O(\log n)$.

Замечание 1.11.4 (Декартово дерево по неявному ключу): Хотим получить «быстрый» динамический массив:

Операция	**Время**
Отрезание конца массива	$O(\log n)$
Конкатенация массивов	$O(\log n)$
Получение по индексу	$O(\log n)$
Удаление по индексу	$O(\log n)$
Вставка по индексу	$O(\log n)$
Перестановка подотрезка	$O(\log n)$

Для этого будем использовать декартово дерево, но ключом будет не индекс элемента в массиве, а число элементов в его поддеревьях, что позволит нам за константу пересчитывать эту характеристику при удалениях или вставках.

1.12. Минимальное остовное дерево: алгоритмы Прима и Крускала

Определение 1.12.1: $T \subset G$ – **остовное дерево**, если T содержит все вершины G и является деревом.

Определение 1.12.2: **Минимальным** остовным деревом во взвешенном неориентированном графе называется остовное дерево минимального веса.

Определение 1.12.3: (S, T) – **разрез** графа (V, E), если $S \cup T = V \wedge V \cap T = \emptyset$.

Определение 1.12.4: Ребро (u, v) **пересекает разрез** (S, T), если u и v – в разных частях разреза.

Определение 1.12.5: Пусть $G' = (V, E')$ – подграф некоторого минимального остовного дерева графа G.

Ребро $(u, v) \notin G'$ называется безопасным, если при добавлении его в G', $G' \cup \{(u, v)\}$ также является подграфом минимального остовного дерева графа G.

Теорема 1.12.1: Рассмотрим связный неориентированный взвешенный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$.

Пусть $G' = (V, E')$ – подграф некоторого минимального остовного дерева G, (S, T) – разрез G, такой, что ни одно ребро из E' не пересекает разрез, а (u, v) – ребро минимального веса среди всех рёбер, пересекающих разрез (S, T).

Тогда ребро $e = (u, v)$ является безопасным для G'.

Доказательство: Достроим E' до некоторого минимального остовного дерева, обозначим его $T_{\min}$. Если ребро $e \in T_{\min}$, то лемма доказана, поэтому рассмотрим случай, когда ребро $e \notin T_{\min}$.

Рассмотрим пусть в $T_{\min}$ от вершины u до вершины v. Так как эти вершины принадлежат разным долям разреза, то хотя бы одно ребро на пути пересекает разрез, назовём его e'.

По условию леммы, $w(e) \leq w(e')$.

Заменим ребро e' в $T_{\min}$ на ребро e. Полученное дерево также является минимальным остовным деревом графа G, поскольку все вершины по-прежнему связаны и вес дерева не увеличился.

Следовательно $E' \cup \{e\}$ можно дополнить до минимального остовного дерева $\Rightarrow$ ребро e – безопасное. $\square$

Замечание 1.12.1 (Алгоритм Прима): По сути является реализацией теоремы выше – выбираем произвольную вершину и обходим весь граф по минимальным рёбрам, которые сейчас доступны.

Из теоремы следует, что в конце получим минимальное остовное дерево.

Сложность $O((V + E) \log V)$:

- В куче находится не более V вершин.
- Извлечём из кучи V вершин.
- Добавим в неё E вершин.

Замечание 1.12.2 (DSU): Хотим структуру данных, которая будет работать с множеством непересекающихся множеств, она будет обрабатывать два типа запросов:

- Unite(a, b) – объединить два множества, где находятся a и b
- AreSame(a, b) – узнать, лежат ли a и b в одном множестве

Храним каждое множество, как подвешенное дерево, тогда его представителем (тем, что сравнивается при AreSame, назовём функцию поиска представителя FindSet) будет корень дерева. Нам понадобятся две эвристики:

- Ранговая. При объединении двух множеств подвешивать то, чей ранг меньше к тому, чей ранг больше. Ранг – глубина дерева.
- Сжатие путей. При запросе FindSet будем подвешивать все вершины сразу к корню в ходе подъёма по пути.

Определение 1.12.6: **Функцией Аккермана** на паре целых чисел определим

$$A(m,n) = \begin{cases} n+1, m=0 \\ A(m-1,1), n=0 \\ A(m-1,A(m,n-1)) \end{cases}$$

Нам важно лишь знать, что $A(4,4) = 2^{2^{2^{65536}}} - 3$

Определение 1.12.7: Введём **обратную функцию Аккермана**:

$$\alpha(N) = \min\{k \mid A(k,k) \geq N\}$$

Теорема 1.12.2 (Тарьян): При использовании обеих эвристик, время на запрос DSU составляет $O(\alpha(N))$

Замечание 1.12.3 (Алгоритм Крускала):

- Отсортируем рёбра графа по весу
- Инициализируем DSU на $|V|$ множествах, каждое из которых отвечает за компоненту связности, которой принадлежит вершина
- Берём минимальное ребро, из тех, вершины которого находятся в разных компонентах связности
- Берём рёбра, пока не останется 1 компонента связности

Очевидно, сложность $O(E \log E)$

1.13. Обход графа в глубину, ширину

Это база, это знать надо

1.14. Поиск кратчайших путей в графе.

Замечание 1.14.1 (Алгоритм Дейкстры): Ищет кратчайшие пути в графе от заданной вершины s до всех, если веса рёбер неотрицательны.

- Инициализируем множество $S = \{s\}$ – множество вершин, для которых кратчайшее расстояние вычислено корректно на текущий момент времени, также будет массив d текущих оценок на весь кратчайшего пути до вершин.
- Очевидно, $d[s] = 0$, а для любой другой вершины – бексконечность
- Рассмотрим все вершины v такие, что $v \notin S$, выберем среди них такую, что $d[v]$ минимально
- Добавим v в множество S, присвоим $\text{dist}(s,v) = d[v]$
- Рассмотрим рёбра вида (v,t), запишем $d[t] = \min(\text{dist}(s,v) + w(v,t), d[t])$
- Пока $S \neq V$, то повторим шаги выше

Замечание 1.14.2 (Алгоритм Форда-Беллмана): Ищет кратчайшие пути из k рёбер в графе от заданной вершины s до всех, если веса рёбер неотрицательны.

- Инициализируем матрицу dp[v][k] текущих оценок на весь кратчайшего пути до вершины v с количеством рёбер k.
- Очевидно, $\mathrm{dp}[s][0] = 0$, а иначе – бесконечность
- Далее будем увеличивать $k = 1...|V| - 1$ в предположении, что для предыдущего k все значения корректны
- Далее перебираем все рёбра $e = (v, u)$ и пытаемся отрелаксировать текущее значение $\mathrm{dp}[u][k] = \min(\mathrm{dp}[u][k], \mathrm{dp}[v][k-1] + w(e))$

Замечание 1.14.3 (Алгоритм Флойда-Уоршелла): Дан взвешенный граф без циклов отрицательного веса, нужно найти расстояние от всех вершин до всех

- Инициализируем трёхмерную dp[u][v][k] текущих оценок на вес кратчайшего пути до вершины v из u, если путь состоит из вершин с номерами, меньшими k.
- Очевидно, $\mathrm{dp}[u][u][0] = 0$, а иначе – бесконечность
- Далее будем увеличивать $k = 1...|V| - 1$ в предположении, что для предыдущего k все значения корректны
- Далее перебираем все пары вершин u, v и пытаемся отрелаксировать ДП следующим образом $\mathrm{dp}[u][v][k] = \min(\mathrm{dp}[u][v][k], \mathrm{dp}[u][k][k-1] + \mathrm{dp}[k][v][k-1])$.

1.15. Поиск сильно-связных компонент в графе

Определение 1.15.1: Две вершины $u, v \in V$ **сильно связаны** в орграфе G, если есть путь как из u в v, так и наоборот.

Определение 1.15.2: **Компоненты сильной связности** – классы эквивалентности по отношению сильной связности.

Определение 1.15.3: Графом конденсации называют граф, где все компоненты сильной связности сжаты до одной вершины, а рёбра между ними получаются, как рёбра между компонентами.

Лемма 1.15.1: Пусть C и C' – две различные вершины в графе конденсации, при этом между ними есть ребро (C, C'), тогда $t_{\mathrm{out}[C]} > t_{\mathrm{out}[C']}$

Доказательство: Рассмотрим два случая:
- Зашли DFS'ом в C раньше, чем в C', значит выйдет алгоритм из C не раньше, чем посетит всю C'. Только затем алгоритм вернётся обратно в C и допроходит его.
- Если же зашли в C' раньше, чем в C, то значит мы зайдём в C только после выхода из C' (иначе существовало бы ребро (C', C)), а значит и выйдем из C точно позднее C'.

□

Замечание 1.15.1 (Алгоритм Касарайю): Ищет компоненты сильной связности
- Запускаем DFS на графе, получаем вершины в порядке увеличения времени выхода
- Строим транспонированный граф
- Запускаем на транспонированном графе DFS в порядке уменьшения времени выхода в исходном графе. Каждая найденная компонента является компонентой сильной связности.

Доказательство: Рассмотрим первый вызов DFS на третьем шаге, так как это вершина с максимальным временем выхода, то нет ребёр в её компоненту связности (по лемме выше).

При этом в транспонированном графе получаем, что из рассматриваемой компоненты нет рёбер в другие, а значит DFS посетит только саму компоненту. □

1.16. Мосты и точки сочленения в графе

Определение 1.16.1: **Мост** – ребро, при удалении которого увеличивается число компонент связности.

Замечание 1.16.1 (Поиск мостов): Введём функцию $t_{\text{up}}(v)$, определяемую следующим образом:

$$t_{\text{up}}(v) = \min \begin{cases} t_{\text{in}}(v) \\ t_{\text{in}}(u), \text{ где u} \end{cases}$$

u – предок v и при этом u достижима по обратному ребру из w – вершины поддерева v.

Лемма 1.16.1: Ребро (t, v) является мостом тогда и только тогда, когда $t_{\text{up}}(v) = t_{\text{in}}(v)$

Доказательство: Условие равносильно тому, что не найдётся вершины u, в которую по обратному ребру можно прыгнуть из поддерева v, что то же самое, что суть моста. □

Определение 1.16.2: **Точка сочленения** – вершина, при удалений которой увеличивается число компонент связности.

Замечание 1.16.2 (Поиск точек сочленения): Рассмотрим ребро (v, w), где v не является корнем дерева обхода. Тогда $t_{\text{up}}(w) \geq t_{\text{in}}(v)$ равносильно тому, что v – точка сочленения.

Доказательство: Выполнение этого неравенства означает, что, пытаясь выпрыгнуть из поддерева w, мы не можем прыгнуть выше v, то есть путь из u в потомках v обязательно пройдёт через неё.

В обратную сторону. Пусть для всех детей v верно, что $t_{\text{up}}(w) < t_{\text{in}}(v)$, тогда из каждого ребёнка можно прыгнуть в наддерево, откуда v – не точка сочленения. □

1.17. Нахождение подстроки в строке

Определение 1.17.1: Строка T называется **супрефиксом** строки S, если она является одновременно и префиксом, и суффиксом строки S.

Определение 1.17.2: **Префикс-функцией** от строки S называют такой массив $\pi(S)$ длины $|S|$, что $\pi(S)[i]$ равно длине максимального несобственного супрефикса строки $S[:i]$.

Замечание 1.17.1 (Линейный алгоритм построения): Заметим, что $\pi[i+1] \leq \pi[i] + 1$. Это верно, так как, добавив 1 символ к максимальному суффиксу, мы увеличим длину максимального супрефикса не более, чем на 1.

Теперь осознаем, что

$$\pi[i+1] = \max_{s,\, s \text{ супрефикс } S[:i],\, S[|s|]=S[i+1]}(|s|+1)$$

то есть мы пытаемся «расширить» подходящий супрефикс подстроки $S[:i]$, а «расширить» его можно, только если после его префикса идёт символ, равный $S[i+1]$.

Осталось осознать, что все супрефиксы имеют длины $\pi[i], \pi[\pi[i]], \ldots$. Рассмотрим $S[:i]$:

$$\begin{cases} S[:\pi[i]] = S[i-\pi[i]:i] \\ S[:\pi[\pi[i]]] = S[\pi[i]-\pi[\pi[i]]:\pi[i]] \end{cases} \Rightarrow S[:\pi[\pi[i]]] = S[i-\pi[\pi[i]] : i]$$

Итого, получаем, что $\pi[i+1] = \pi[j] + 1$, где j – первый подходящий индекс из последовательности $i, \pi[i], \pi[\pi[i]], \ldots$, такой, что $S[\pi[j]] = S[i+1]$.

Замечание 1.17.2 (Алгоритм Кнута-Морриса-Пратта): Дана цепочка T и образец P. Требуется найти все позиции, начиная с которых P входит в T.

Построим цепочку $S = P\#T$, где $\#$ – любой символ не входящий в алфавит искомых строк.

Вычислим от этой строки префикс-функцию π, тогда, если в какой-то момент $\pi[i] = |P|$, значит мы нашли конец вхождения P в T, то есть допишем в ответ $i - |P|$.

1.18. Стандартные контейнеры

Замечание 1.18.1 (vector): std::vector – динамический массив.

Основные операции:

Операция	**Время**
Получение элемента по индексу	$O(1)$
Вставка в конец	$O^*(1)$
Вставка в начало/середину	$O(N)$
Удаление из конца	$O(1)$
Удаление из начала/середины	$O(N)$

Замечание 1.18.2 (deque): std::deque – «двухсторонний» динамический массив, основан на кольцевом буффере на динамическом массиве.

Основные операции:

Операция	**Время**
Получение элемента по индексу	$O(1)$
Вставка в начало/конец	$O^*(1)$
Вставка в середину	$O(N)$
Удаление из начала/конца	$O(1)$
Удаление из середины	$O(N)$

Замечание 1.18.3 (priority_queue): std::priority_queue – бинарная пирамида, по умолчанию является адаптером над std::vector.

Основные операции:

Операция	**Время**
Вставка	$O(\log N)$
Удаление минимума	$O(\log N)$
Получение минимума	$O(1)$

Замечание 1.18.4 (map): std::map – ассоциативный контейнер, является красно-чёрным деревом пар ключ-значение, где сравнения все по ключам.

Основные операции:

Операция	Время
Вставка пары ключ-значение	$O(\log N)$
Удаление по ключу	$O(\log N)$
Поиск по ключу	$O(\log N)$

Замечание 1.18.5 (Итераторы): Итератор – объект, созданный поверх некоторого контейнера, в зависимости от предоставляемых возможностей, делится на категории:

<table>
<tr><th colspan="5">Iterator category</th><th>Defined operations</th></tr>
<tr><td rowspan="5">LegacyContiguousIterator</td><td rowspan="4">LegacyRandomAccessIterator</td><td rowspan="3">LegacyBidirectionalIterator</td><td rowspan="2">LegacyForwardIterator</td><td>LegacyInputIterator</td><td>• read
• increment (without multiple passes)</td></tr>
<tr><td></td><td>• increment (with multiple passes)</td></tr>
<tr><td colspan="2"></td><td>• decrement</td></tr>
<tr><td colspan="3"></td><td>• random access</td></tr>
<tr><td colspan="4"></td><td>• contiguous storage</td></tr>
<tr><td colspan="6">Iterators that fall into one of the above categories and also meet the requirements of LegacyOutputIterator are called mutable iterators.</td></tr>
<tr><td colspan="5">LegacyOutputIterator</td><td>• write
• increment (without multiple passes)</td></tr>
</table>

Определение 1.18.1: **Компаратор** - функциональный объект, проверяющий, принадлежат ли два объекта одного типа к требуему отношению «меньше».

1.19. Бор. Алгоритм Ахо-Корасик.

Определение 1.19.1: **Бор** – структура данных для хранения набора строка, представляющая из себя подвешенное дерево с символами на рёбрах.

Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной.

Обозначим за $[u]$ – слово, которое составляет путь от корня до вершины u в боре.

Определение 1.19.2: **Суффиксной ссылкой** вершины u называют такую вершину v, то $[v]$ является максимальным по длине суффиксом $[u]$, который можно прочесть, идя по бору из корня до, быть может, не терминальной вершины.

Обозначение $v := \text{link}(u)$.

Определение 1.19.3: Введём функцию $to(u, c)$, равную вершине, куда из вершины u можно перейти по букве c:

$$to(u,c) = \begin{cases} \text{vertex}([u]+c), \text{ если из } u \text{ есть переход по } c \\ to(\text{link}(u),c), \text{ если из } u \text{ нет перехода по } c \end{cases}$$

Заметим соотношение $\text{link}(\text{vertex}([u] + c)) = \text{vertex}([to(\text{link}(u), c)])$

Замечание 1.19.1 (Алгоритм Ахо-Корасик): Строит суффиксные ссылки в боре:

- Строим бор на данном наборе слов
- С помощью BFS из корня вычисляем функции link и $to(\cdot, c)$. Из соотношения выше следует, что данные функции можно вычислить через вершины выше, чем текущая.

1.20. Вычисление выпуклой оболочки в 2D

Определение 1.20.1: **Выпуклая оболочка** для набора точек на плоскости – многоугольник минимальной площади, внутри которого (или на его границе) лежат все точки из набора.

Замечание 1.20.1 (Алгоритм Джарвиса):

- В качестве начальной берётся самая левая нижняя точка P_1, она точно является вершиной выпуклой оболочки.
- Следущей точкой P_2 берём такую точку, которая имеет наименьший положительный полярный угол относительно точки P_1 как начала координат. После этого для каждой точки P_i против часовой стрелки за $O(n)$ ищется такая точка P_{i+1} среди оставшихся точек, в которой будет образовываться наибольший угол между прямыми $P_{i-1}P_i$ и P_iP_{i+1}. Она и будет следующей вершиной выпуклой оболочки.
- Нахождение вершин выпуклой оболочки продолжается до тех пор, пока $P_{i+1} \neq P_1$

Работает за $O(N^2)$.

Замечание 1.20.2 (Алгоритм Грэхэма):
- В качестве начальной берётся самая левая (затем самая нижняя) точка P_0, она точно является вершиной выпуклой оболочки.
- Проводим сортировку точек по углу между векторами $\overrightarrow{P_0P_i}, \overrightarrow{OX}$. Получим набор $P_1, ..., P_n$
- Заведём стек точек, добавим в него точки P_0, P_1.
- Будем строить оболочку против часовой стрелки. Тогда будем добавлять в стек точки, пока поворот от вектора $\overrightarrow{P_{s-1}P_s}$ до $\overrightarrow{P_sP_k}$ будет левым, где P_{s-1}, P_s – крайние две точки в стеке, а P_k – точка из иходного набора, которая ранее не рассматривалась.
- Если получилось, что рассматриваемый поворот правый, то удалим из стека P_s и проверим тип поворота на новых двух крайних точках стека
- Пройдя таким образом по всем точкам из набора, в стеке останутся лишь точки, образующие выпуклую оболочку.

Сложность $O(N \log N)$

1.21. Планиметрия

Какая-то база из школы и алгема

1.22. Проверка принадлежности точки многоугольнику

Замечание 1.22.1 (Для выпуклого многоугольника):
- Выберем самую левую (а затем нижнюю) точку P_0
- Отсортируем все остальные точки по полярному углу относительно точки P_0
- При запросе на проверку P, посчитаем её полярный угол относительно P_0 и найдём две ближайшие к ней точки бинарным поиском (если таких нет, что ответ точно нет)
- Пусть эти точки (L, R), тогда нам остаётся проверить, что P лежит внутри треугольника P_0LR, а для этого достаточно проверить, что тройка вершин (L, P, R) лежит по часовой стрелке (синус угла LPR должен быть отрицательным).