МФТИ ФПМИ

Алгоритмы и структуры данных

Русскоязычные иностранцы

Кулапин Артур

Осень 2021-Лето 2022

Оглавление

Глава 1

Семестр 1.

Модуль 1. Основы основ.

Лекция 1.

В течение всего курса мы будем изучать алгоритмы и структуры данных. При этом вам предстоит писать очень много программ, но зададимся вопросом, как отличить хорошую программу от плохой? Есть множество критериев, которые зависят от поставленной перед вами задачи. В нашем случае большинство задач будут посвящены применению различных алгоритмов для решения небольших задач, поэтому мы будем пользоваться двумя базовыми характеристиками нашего кода, чтобы оценить его качество, а именно время исполнения или *временная сложность* и потребляемые ею ресурсы, в нашем случае мы будем рассматривать затрачиваемую память, то есть *пространственную сложность*.

Сложность программы

Для оценки парамаетров выше нам необходимо договориться о *модели вычислений* и о том, в чем мы оцениваем. Сначала обсудим единицы измерения.

Def. Пусть имеются две функции $f(n)$ и $g(n)$, при этом $f, g : \mathbb{N} \to \mathbb{N}$, тогда считается, что $f(n) = \mathcal{O}(g(n))$, если $\exists C > 0\ \exists N_0 :\ \forall n > N_0\ f(n) \leq C \cdot g(n)$.

Поясним определение выше. Говорят, что *функция f является O-большим от функции g*, если она растет не быстрее, чем функция g, домноженная на константу (сколь угодно большую, заметьте). Также можно считать, что *функция f ограничена функцией g, домноженной на константу.* Для понимания предлагается рассмотреть несколько примеров.

Примеры.

1. Пусть $f(n) = n$, а $g(n) = n^2$, тогда очевидно, что $f(n) = \mathcal{O}(g(n))$. Например, пусть $C = 1$, а

$N_0 = 2$, тогда $n = f(n) < C \cdot g(n) = g(n) = n^2$, что верно для $n \geq 2$.

2. Пусть $f(n) = P_k$, а $g(n) = P_{k+\alpha}$, где P_r — многочлен степени r, $k \in \mathbb{N}$, $\alpha \in \mathbb{N}$. Тогда также нетрудно показать, что $f(n) = \mathcal{O}(g(n))$. Для скептически настроенных читателей проведем подробный анализ. Введем две функции $f_1(n) = a_f n^k$ и $g_1(n) = a_g n^{k+\alpha}$. Тогда из курса математического анализа известно, что $f_1 \sim f$ и $g_1 \sim g$, то есть можно перейти только к асимптотическому сравнению мономов старших степеней. Более того, мы имеем право опустить константы, так как они кроются в определении $\mathcal{O}$-большого. То есть задача сведена к доказательству соотношения $n^k = \mathcal{O}\left(n^{k+\alpha}\right)$. Далее рассуждение аналогично первому примеру.

Больше примеров можно найти в книге Кормена на страницах 67-84, ссылка на нее есть в соответствующем разделе данной книги.

Def. Пусть имеются две функции $f(n)$ и $g(n)$, при этом $f, g : \mathbb{N} \to \mathbb{N}$, тогда считается, что $f(n) = \Omega(g(n))$, если $\exists C > 0 \; \exists N_0 : \; \forall n > N_0 \; f(n) \geq C \cdot g(n)$.

То есть f в данном случае ограничена снизу функцией g.

Def. Пусть имеются две функции $f(n)$ и $g(n)$, при этом $f, g : \mathbb{N} \to \mathbb{N}$, тогда считается, что $f(n) = \Theta(g(n))$, если $\exists C_1, C_2 > 0 \; \exists N_0 : \; \forall n > N_0 \; C_1 \cdot g(n) \leq f(n) \geq C_2 \cdot g(n)$.

Упражнение. Докажите утверждения ниже

- $f(n) = \Theta(g(n))$ тогда и только тогда, когда $f(n) = \mathcal{O}(g(n))$ и $f(n) = \Omega(g(n))$.
- Если $f(n) = \mathcal{O}(g(n))$ и $g(n) = \mathcal{O}(h(n))$, то $f(n) = \mathcal{O}(h(n))$.

Note. В частности из второго утверждения можно заключить, что оценка в виде $\mathcal{O}$-большого не является точной.

Далее мы будем стремиться получить хотя бы верхнюю оценку на время работы алгоритма. При этом мы будем стараться получать как можно более строгую оценку в силу замечания выше.

Сразу обозначим множество классов, которые мы будем использовать для анализа

- Степенной логарифмический класс $\mathcal{O}(\log^k n)$.
- Полиномиальный класс $\mathcal{O}(n^k)$.
- Полилогарифмический класс $\mathcal{O}(n^k \cdot \log^r n)$.
- Экспоненциальный класс $\mathcal{O}\left(a^{f(n)}\right)$.

Теперь мы готовы перейти к оценке параметров. Здесь и далее мы будем все оценивать в асимптотических обозначениях, не стремясь найти N_0 и C из определений выше. В ходе анализа алгоритмов мы будем пользоваться следующими допущениями

1. Память вычислителя безгранична.
2. Доступ к произвольному блоку памяти происходит за $\mathcal{O}(1)$ времени.

Рекуррентные соотношения

Теорема (Мастер-теорема) (б/д). Пусть зафиксированы константы $a \geq 1$, $b > 1$, $f(n)$ — некоторая функция, а $T(n)$ определена рекуррентно

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Тогда верны следующие утерждения

1. Если $f(n) = \mathcal{O}\left(n^{\log_b a - \varepsilon}\right)$ для некоторой константы $\varepsilon > 0$, то $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. Если $f(n) = \mathcal{O}\left(n^{\log_b a}\right)$, то $T(n) = \Theta\left(n^{\log_b a} \cdot \log n\right)$.
3. Если $f(n) = \mathcal{O}\left(n^{\log_b a + \varepsilon}\right)$ для некоторой константы $\varepsilon > 0$ и

$$\exists C < 1, N_0 \in \mathbb{N}\ \forall n > N_0\ af\left(\frac{n}{b}\right) \leq Cf(n),$$

то $T(n) = \Theta(f(n))$.

Данной теоремой мы будем пользоваться как тяжелой артиллерией. В общем случае мы будем действовать методом подстановки или анализом дерева рекурсии. Приведем пример для первого метода, второй будет рассмотрен позднее.

Пример: Пусть $T(n) = 2T\left(\frac{n}{2}\right) + n$. Очевидно, можно получить ответ из основной теоремы, однако проведем анализ подробнее.

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4T\left(\frac{n}{4}\right) + 2n = \ldots = \log n \cdot T(1) + n\log n = \Theta(n\log n)$$

Теперь мы готовы перейти к анализу алгоритмов.

Бинарный поиск

Пусть имеется монотонный предикат $P(n)$, то есть

$$\exists N_0 : \begin{cases} P(n) = 0, & n < N_0 \\ P(n) = 1, & n \geq N_0 \end{cases}$$

И перед нами стоит задача отыскать это N_0. Например, задача проверки наличия элемента X в отсортированном массиве. В данном случае предикат будет звучать как $P(i) = 1 \iff a[i] \geq X$.

Note. По теореме о промежуточных значениях данная задача разрешима, причем единственным образом.

Можно конечно пройтись по массиву, проверяя, является ли текущий элемент искомым, однако это долго, мы нигде не пользуемся свойством монотонности построенного предиката. Рассмотрим более оптимальный алгоритм, общий для данного класса задач.

1. Вычислим $P(i)$, где i — середина массива.
2. Если $P(i) = 0$, то $N_0 > i$, иначе $N_0 \leq i$. Таким образом, можно перейти лишь к половине массива и повторить шаги заново для нового массива.

Проведем анализ такого алгоритма преполагая, что время вычисления предиката P равно $\mathcal{O}(f(n))$ и $\forall r \leq n \ f(r) \leq f(n)$. Тогда нетрудно построить рекурренту

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(f(n)) = T\left(\frac{n}{4}\right) + \Theta(f(n)) + \mathcal{O}\left(f\left(\frac{n}{2}\right)\right) \leq$$
$$\leq T\left(\frac{n}{4}\right) + 2\mathcal{O}(f(n)) \leq \ldots \leq \log n \cdot \mathcal{O}(f(n)) = \mathcal{O}(f(n) \cdot \log n)$$

В случае задачи выше $f(n) = \mathcal{O}(1)$, то есть итоговое время составит $\mathcal{O}(\log n)$, что заметно лучше наивного прохода, работающего за $\mathcal{O}(n)$.

Упражнение. Какие условия надо поставить на $f(n)$ в анализе алгоритма выше, чтобы решение рекурренты имело вид $\Theta(f(n) \cdot \log n)$?

Лекция 2.

Списки

Список — последовательный набор узлов. Чего мы хотим от списка? Мы хотим поддержку следующих операций:

Операция	Время	Примечание
Вставка в начало	$\mathcal{O}(1)$	
Удаление из начала	$\mathcal{O}(1)$	
Вставка в произвольное место	$\mathcal{O}(1)$	Если известно место
Удаление из произвольного места	$\mathcal{O}(1)$	Если известно место
Поиск	$\mathcal{O}(N)$	
Обращение по индексу	$\mathcal{O}(N)$	

Начнем с односвязных списков. Наш список будет хранить цепочку из узлов, где каждый указывает на следующего за ним, а последний указывает в никуда, что является индикатором конца. Для удобства будем хранить еще и размер списка.

Удовлетворяет ли такая простенькая структура нашим требованиям на асимптотику? Очевидно да, так как поиск и обращение по индексу требуют линейного прохода, но при этом вставка или удаление элемента это всего лишь создание узла и переприсвоение указателей.

Теперь немного о двусвязных списках. Это простое улучшение односвязного списка, позволяющее сильно увеличить его функционал. В односвязном списке мы могли указывать только на узел впереди нас. А давайте теперь будем указывать еще на узел позади нас. То есть в узле хранить два указателя, а в списке хранить указатель на голову и на хвост. Таким образом, получаем «двусторонний» односвязный список.

Благодаря такому нехитрому апгрейду получаем возможность работы с обоими концами списка, не теряя в асимптотике.

Стек

Начнем с самого простого линейного контейнера — стек. Это структура данных, с которой можно проводить следующие операции:

- Вставка в начало за $\mathcal{O}(1)$.
- Удаление из начала за $\mathcal{O}(1)$.
- Узнать размер за $\mathcal{O}(1)$.

Замечание. Данная структура удовлетворяет парадигме LIFO (last in, first out), а именно тот, кто попал позднее, будет удален ранее.

Нетрудно заметить, что мы уже где-то видели все эти операции за такое время. Верно, в списках. Причем нам достаточно односвязного списка, так как работаем мы только с одним концом. Таким образом, можно реализовать стек на односвязном списке.

Обсудим еще один вариант реализации стека. Например, воспользуемся самым обычным массивом. Он тоже умеет делать все операции выше за это время (и даже больше). Казалось бы, немного бесполезная структура данных. В реализации некоторых вещей он используется (например, стек вызова функций или же стековая память, которая сожержит локальные переменные).

Задумаемся совсем об отвлеченном — о префиксных минимумах. К чему это? К тому, что их можно считать напрямую, а можно с помощью стека, поддерживая в узле минимум в стеке на текущий момент. Это нетрудно сделать, так как при получении нового элемента массива, минимум на префиксе это минимум из элемента и значения в вершине стека. Все еще не очень понятно, зачем считать так префиксные минимумы... Но уже близко катарсис.

Очередь

В этот раз наш контейнер будет удовлетворять парадигме FIFO (first in, first out), а именно тот, кто попал ранее, будет удален ранее. Почти как в настоящих очередях, если бы они были идеальны. То есть набор операция таков:

- Вставка в начало за $\mathcal{O}(1)$.
- Удаление из конца за $\mathcal{O}(1)$.
- Узнать размер за $\mathcal{O}(1)$.

Перед вами двусвязный список в гриме, не узнали? Очередь на двусвязном списке это конечно прикольно, но давайте рассмотрим еще один вариант. А именно, воспользуемся стеками. Для реализации очереди нам понадобятся два стека и немного смекалочки.

Понятно, как добавлять элементы, но как их извлекать в нужном порядке? Напишем код удаления, а читателю предложено разобраться, что в нем происходит. У нас будут два стека: `stack_in` и `stack_out`, вставка будет происходить в первый, а удаление из второго. Единственное, нам надо будет в случае удаления из пустого выходного стека переложить все элементы из входного стека в выходной (получим развернутый входной стек). Теперь во втором стеке лежат элементы, которые мы будем извлекать (да еще и в нужном порядке лежат). Казалось бы, операция линейная по времени, но заметим, что мы будем вызывать Reverse только когда второй стек опустел, а значит достаточно редко.

Ради чего вот это все? Используем идею очереди на двух стеках и стека с поддержкой минимума, получаем очередь с поддержкой минимума. На самом деле это уже мощная структура, так как вместо минимума можно считать почти любую операцию на отрезке (например, НОД). Главное, чтобы операция была ассоциативна, то есть $f(a, f(b, c)) = f(f(a, b), c)$. Запрос минимума в очереди сводится к запросу минимума из двух минимумов в стеке.

Пример: Классическая задача, которая требует такой структуры. Пусть имеется массив целых чисел длины N. Требуется найти минимум на всех подотрезках длины K за $\mathcal{O}(N)$ времени и $\mathcal{O}(K)$ доппамяти.

Решение. Сложим первые K элементов в очередь с минимумом. Выведем минимум, теперь будем идти очередью как окном, а именно добавлять следующий элемент, извлекать предыдущий, вывести минимум.

Note. Данный подход можно обобщить с массивов на двумерные таблички и произвольные размерности.

Упражнение. Подумайте, как такую задачу можно решить с помощью префиксных/суффиксных минимумов. Возможно ли это? Приведите пример, что это не удастся сделать так просто. А если операция обратима, например, сумма, то можно обойтись префиксными и суффиксными суммами?

Утверждение. Для использования подхода префиксных сумм необходимо, чтобы операция была обратима и ассоциативна, а для такого способа с очередью необходимо, чтобы операция была только ассоциативна.

Дек

Дек иногда называют двусторонним стеком или двусторонней очередью. Почему? Рассмотрим набор операций, определяющих дек как структуру данных:

- Вставка в начало или в конец за $\mathcal{O}(1)$.
- Удаление из начала или из конца за $\mathcal{O}(1)$.
- Узнать размер за $\mathcal{O}(1)$.

По сути перед нами двусвязный список.

Note. Контейнер `std::deque` реализован далеко не на двусвязном списке. Его реализация значительно сложнее, зато позволяет обращаться к элементу по индексу за константное время и не инвалидировать ссылки и указатели на элементы (что это такое, будет позднее в курсе лекция по C++).

Амортизационный анализ

В ходе амортизационного анализа время, необходимое для выполнения последовательности операций усредняется по всем выполняемым операциям. Нам он пригодится, чтобы показать, что даже если одна из редко выполняемых операций достаточно сложна по времени, то в среднем ее стоимость будет невелика. Далее в этой книге мы будем применять либо метод бухгалтерского учета или метод монеток, либо оценивать стоимость операции по определению ниже.

Def. Пусть имеется последовательность операций, каждая из которых выполняется за время t_i, тогда *амортизированная стоимость* операции $a_i = \frac{1}{n}\sum\limits_{i=1}^{n} t_i$.

Note. Далее будем записывать амортизированную стоимость операции как $\mathcal{O}^*(f(n))$

В рамках данного метода мы определим амортизированную стоимость каждой операции. Для каждого вида операции она будет своей. Легкие операции будут переоценены, и мы сможем этим оплатить недооцененные тяжелые операции. Согласен, что ничего непонятно, поэтому обсудим на примере.

Динамически расширяющийся массив

Как известно, обычные массивы заданы своим размером, а что делать, если мы хотим структуру данных со следующим набором операций?

- Добавить элемент в конец (push) за $\mathcal{O}^*(1)$.
- Удалить элемент из конца (pop) за $\mathcal{O}(1)$.
- Обращение по индексу (get) за $\mathcal{O}(1)$.

Обычный массив нам не подойдет, так как ограничен, а стек не умеет быстро по индексу обращаться. Казалось бы, можно использовать массив и в случае добавления элемента его расширять, но как именно? Если увеличивать на 1 размер, то каждое добавление будет требовать линейное копирование, таким образом, выигрыша по сравнению со стеком нет. Но давайте увеличивать каждый раз, когда некуда добавить элемент, в два раза, что тогда? Ответим на этот вопрос с помощью метода монеток.

Пусть каждый push, не требующий перевыделения памяти стоит 3 монетки. Одна монетка на запись элемента и две монетки в резерве (будем их класть на элементы массива с номерами i и $i - \frac{n}{2}$). Задумаемся, сколько монеток на каждом элементе будет лежать перед тем, как сделать реаллокацию и копирование. Ответ очевиден, на каждом элементе будет лежать по одной монетке, то есть мы их можем потратить как раз на запис в новый массив (выделение памяти мгновенное). Таким образом, амотризированная стоимость push равна трем монеткам, что эквивалентно трем разам по $\mathcal{O}(1)$. То есть амортизированная сложность push равна $\mathcal{O}(1)$.

Note. На лекции было доказательство утверждения, что `push` работает за $\mathcal{O}^*(1)$, по определению амортизированной стоимости. На экзамене разрешено применять любое.

Модуль 2. Сортировки.

Лекция 3.

Теорема о сортировках

Def. Сортировкой, основанной на сравнениях называют сортировку, работающую, в следующем преположении — объекты можно только сравнивать.

Лемма. $\log N! = \Theta(N \log N)$

Доказательство:

$$\log N! = \log \prod_{k=1}^{N} k = \sum_{k=1}^{N} \log k \leq \sum_{k=1}^{N} \log N = N \log N$$

$$\log N! = \log \prod_{k=1}^{N} k = \sum_{k=1}^{N} \log k \geq \frac{N}{2} \log \frac{N}{2} = \Omega(N \log N)$$

Неравенство во второй строке останется простеньким упражнением на индукцию для читателя.

Note. Можете еще использовать выпуклость логарифма.

■

Докажем самую важную теорему этого модуля.

Теорема. Сортировка, основанная на сравнениях, работает за $\Omega(N \log N)$.

Доказательство: Задача сортировки равносильна задаче поиска единственной корректной перестановки среди их множества. Единственное, что мы можем делать — брать два элемента в произвольной перестановке и понимать, надо ли их переставлять. При этом количество перестановок, где надо элементы менять местами, и где не надо, одинаково и равно $\frac{N!}{2}$. Таким образом, каждое сравнение равносильно уменьшению множества рассматриваемых перестановок вдвое, откуда искомое время составит $\Omega(\log N!) = \Omega(N \log N)$ по лемме выше.

■

Рассмотрим несколько важных характеристик сортировок с точки зрения которых мы будем их рассматривать:

1. Стабильность говорит о том, что элементы, одинаковые для сравнения, не поменяют своего расположения относительно друг друга
2. Величина доппамяти (стековой и динамической), среднее и худшее время работы.

Бинарная пирамида

Зададимся новой цель, а именно созданием структуры данных с такими возможностями

Операция	Время
Добавление элемента	$\mathcal{O}(\log N)$
Удаление минимума	$\mathcal{O}(\log N)$
Чтение минимума	$\mathcal{O}(1)$

Все это время мы хранили либо списки, либо массивы, но в этот раз мы будем хранить в виде подвешенного полного бинарного дерева нашу структуру.

Def. Дерево называется *подвешенным*, если есть выделенная вершина, называемая *корнем* дерева.

Def. Дерево называется *бинарным*, если у каждой вершины ствень не более трех, то есть один родитель (его нет у корня) и не более двух детей (у листьев их может не быть вообще).

Def. Бинарное дерево называется *полным*, если расстояния от всех листьев до корня отличаются максимум на единицу.

Можно реализовать бинарную пирамиду на полном бинарном дереве, при этом будем стремиться поддерживать *свойство пирамиды*, а именно то, что все сыновья вершины строго больше ее самой. Тогда очевидно, что в корне будет лежать минимум, и его чтение будет делаться мгновенно.

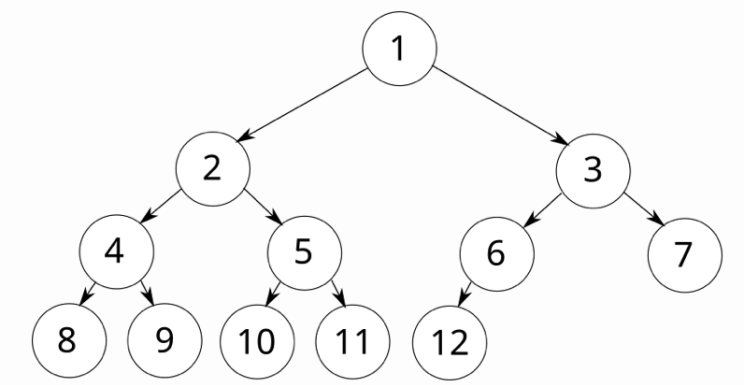
Теперь разберемся со вставкой и удалением. Определим две вспомогательные операции, а именно просеивание вниз *SiftDown* и просеивание вверх *SiftUp*. В чем их смысл? Ну он достаточно прост, просеивание вниз старается утопить элемент как можно ниже, меняя его местами с меньшим из сыновей, не наруша свойство пирамиды. Аналогично устроено просеивание вверх.

Как устроена вставка? Давайте хранить помимо указателя на корень еще и указатель на самый «правый» узел, у которого меньше двух сыновей. Тогда подвесим новый элемент к этому узлу и потом просеем его вверх. Если же таких узлов нет, то у нас нижний уровень занят, поэтому самый левый лист станет новым узлом родителем.

Как устроено удаление? Так как мы храним указатель на корень и на крайний узел, поменяем местами элементы в них и затем просеим вниз элемент в корне, а лист с минимумом просто удалим.

Сложность операций составляет $\mathcal{O}(h)$, где h это высота пирамиды, то есть ее дерева, а у полного бинарного дерева высота не превосходит $\log_2 N + 1$, откуда получаем требуемую сложность.

Теперь задумаемся о нескольких моментах. Первый из них состоит в том, что полное бинарное дерево на самом деле можно хранить как массив, укладывая уровни по очереди друг за другом. Например, пусть имеется дерево Рассмотрим элементы по уровням



Уровень								
0	1							
1	2	3						
2	4	5	6	7				
3	8	9	10	11	12			

Тогда итоговый массив имеет вид $[1, 2, \ldots, 11, 12]$. Можно также получить индексы левого и правого сына. В 0-индексации это $2i + 1$ и $2i + 2$ соответственно. Таким образом, мы реализовали пирамиду на массиве.

Второй момент заключается в решении такой задачи. Имеется массив элементов, надо построить на нем структуру пирамиды на массиве (то есть переставить элементы так, чтобы получилась пирамида). Давайте будем просеивать вниз элементы, начиная с $\frac{N}{2}$-го и до нулевого. Тогда элементы в конце массива поднимутся при просеивании вниз и все будет корректно. Но какая тогда сложность будет? Очевидно, верна оценка $\mathcal{O}(N \log N)$, но давайте исследуем точнее.

Элементов на $(h-1)$-м уровне всего не более $\frac{N}{4}$, а высота их просеивания (глубина, на которую они могут опуститься) равна единице. Аналогично для $(h-2)$-го уровня только на два вниз и таких

элементов не более $\frac{N}{8}$. Тогда итоговая сложность составит

$$\sum_{k=1}^{h} \frac{N}{2^{k+1}} \cdot k \cdot \mathcal{O}(1) = \mathcal{O}(N) \cdot \sum_{k=1}^{h} \frac{k}{2^k} \leq \mathcal{O}(N) \cdot \sum_{k=1}^{h} \frac{2^{\frac{k}{2}}}{2^k} = \mathcal{O}(N) \cdot \sum_{k=1}^{h} \frac{1}{2^{\frac{k}{2}}}$$

Докажем, что полученный ряд сойдется.

$$\sum_{k=1}^{h} \frac{1}{2^{\frac{k}{2}}} = \sum_{k=1}^{h} \frac{1}{2^{\frac{2k}{2}}} + \sum_{k=1}^{h} \frac{1}{2^{\frac{2k+1}{2}}} = \left(1 + \frac{1}{\sqrt{2}}\right) \sum_{k=1}^{h} \frac{1}{2^{\frac{2k}{2}}} = \left(1 + \frac{1}{\sqrt{2}}\right) \sum_{k=1}^{h} \frac{1}{2^k} \leq \left(1 + \frac{1}{\sqrt{2}}\right) \sum_{k=1}^{\infty} \frac{1}{2^k} = 1 + \frac{1}{\sqrt{2}}$$

Таким образом, на самом деле верна оценка $\mathcal{O}(N)$, она даже точна, так как минимум $\frac{N}{2}$ операций придется сделать.

Note. Переход к неравенству в первой строке неочевиден, так как не для всех натуральных k верно, что $k \leq 2^{\frac{k}{2}}$. Упражнением читателю останется доказать, что это верно для $k \geq 4$, что завершает доказательство для самого педантичного читателя.

Пирамидальная сортировка

Как же отсортировать массив? Построим на нем пирамиду и N раз извлечем минимум. Итоговое время $\mathcal{O}(N \log N)$.

Сортировка слиянием

Ну тут алгоритм достаточно простой. Будем пользоваться методом разделяй и властвуй, а именно:

1. Разбей задачу на K подзадач попроще (разбить массив на две примерно равные половины)
2. Реши все подзадачи, применив рекурсивно шаги 1 и 3, пока не дошел до базы, в которой все тривиально (разбивай массивы и дальше пополам, пока не дойдем до массива из двух элементов, там все тривиально)
3. Объедини решения подзадач (слей два отсортированных массива в один большой)

В данной ситуации все более чем тривиально, кроме третьего шага. Как сливать два отсортированных массива в один большой?

Заведем два указателя на начало каждой из половин. Если первый больше второго, то пишем его в итоговый массив и сдвигаем первый (иначе второй). Как только один из указателей дошел до конца, продвигаем другой в конец, выписывая оставшиеся элементы. Это работает за $\mathcal{O}(N)$ времени и доппамяти.

Какое же общее время работы? Рассмотрим дерево рекурсии. В нем всего $\log N$ уровней, при этом на каждом выполняется $\mathcal{O}(N_1) + \ldots + \mathcal{O}(N_k)$ операций на слияние k пар кусков, при этом на каждом

уровне требуется линейная доппамять. Таким образом, общее время работы равно $\mathcal{O}(N \log N)$, а доппамять линейна.

Внимание, материал далее не будет приступствовать в экзаменационной программе. Рекомендуется читать на свой страх и риск.

Exponential Search + Optimal Merge (*)

Задумаемся о слиянии двух отсортированных массивов в один. В случае выше массивы примерно равной длины и понятно, что $\mathcal{O}(N)$ нас более чем устроит. Но пусть длина массива a равна M, а у массива b длина N соответственно.

Пусть $M = 1$, тогда крайне бессмысленно пытаться делать классический вариант слияния, так как достаточно одного бинпоиска и нам хватит $\mathcal{O}(\log N)$ времени (в общем случае будет $\mathcal{O}(M \log N)$. Аналогично для $M << N$, но возникает вопрос, а как определить отношение «много больше»? И еще более трудный вопрос, а какой асимпотически оптимальный по времени алгоритм слияния? Оказывается, что ответ даже не $\mathcal{O}(\min(M + N), (M \log N))$, давайте разбираться.

Заметим, что в общем случае бинарный поиск работает за $\Theta(\log N)$, а давайте будем мыслить не в терминах длины массива, а в терминах позиции, куда вставлять элемент. *Задумайтесь, идея непростая.*

Рассмотрим следующий алгоритм. Пусть k — позиция, куда попадет ответ, а p — номер итерации (изначально ноль).

1. Заведем указатель на нулевой элемент массива, сравним с искомым. Если все ок, то победа, иначе идем дальше.
2. Сдвинем указатель на 2^p, проверим, правда ли, что уже перескочили? Если да, то достаточно запустить обычный бинарный поиск на отрезке $[0, 2^p]$, иначе повтори этот шаг, увеличив номер итерации на единичку.

Алгоритм интересный, но какая асиматотика? Очевидно, что если позиция ответа будет k, то число итераций $\log_2 k - 1 \leq P \leq \log_2 k + 1$, откуда на поиск крайнего положения тратится $\Theta(\log k)$ времени. Теперь заметим, что бинпоиск работает за логарифм от длины отрезка, то есть за $\Theta(\log 2^P) = \Theta(\log k)$.

Вау, получили алгоритм, который вырождается в бинпоиск в одном экстремальном случае и в константу в другом. Теперь давайте искать место вставки не бинарным, а галлопирующим поиском. Пусть k_i — место, куда вставили $a[i]$, тогда можно оценить время работы алгоритма следующим образом:

$$\mathcal{O}\left(\sum_{i=1}^{M} \log k_i\right) = \mathcal{O}\left(M \cdot \frac{\sum\limits_{i=1}^{M} \log k_i}{M}\right) = \mathcal{O}\left(M \log \frac{\sum\limits_{i=1}^{M} k_i}{M}\right) = \mathcal{O}\left(M \log \frac{N}{M}\right)$$

Второй переход верен в силу выпуклости вверх логарифма и неравенства Йенсена.

Теорема (б/д). Асимптотически быстрее слить два отсортированных массива нельзя.

Лекция 4.

Число инверсий в массиве

Обсудим поиск числа инверсий в массиве с помощью сортировки слиянием.

Когда мы сливаем обе части, мы сравниваем элементы одной (первой, левой) части с элементами другой (правой, второй) части соответственно. И если элемент левой части больше элемента правой части соответственно, то значит это и есть инверсия.

И так же все оставшиеся элементы левой части тоже будут больше, т.к. левая и правая часть отсортированы. Поэтому количество инверсий нужно увеличить на количество оставшихся элементов + 1 (текущий элемент).

Таким образом, нам нужно пробрасывать каждый раз глобальный счетчик инверсий в каждое слияние, а затем для каждой инверсии надо прибавить к глобальному счетчику разность длины левой части и индекса элемента в левой части.

Это работает, так как пусть мы сравниваем в сортировке слиянием $l[i]$ и $r[j]$, тогда если $r[j] < l[i]$, то $r[j] < l[i] < l[i+1] < \ldots < l[N]$, то есть число инверсий это $N - i$ для $r[j]$. Тогда итоговое число инверсий равно сумме по j таких значений.

Быстрая сортировка

Алгоритм быстрой сортировки достаточно прост:

1. *Как-то* (позднее обговорим как) выбрать опорный элемент (pivot).
2. Поставить pivot на свое место в отсортированном массиве (то есть все, что меньше, перекинуть влево, а что больше — вправо)
3. Вызвать рекурсивно от правой и левой половин.

Использует быстрая сортировка доппамять? В такой реализации использует, так как нам надо хранить стек рекурсии. Какое время работы? В худшем случае под любую стратегию выбора опроного элемента можно построить контрданные так, чтобы работало все квадратичное время с линейной доппамятью. Если же тест случайный, то в среднем время составит $\mathcal{O}(N \log N)$, а доппамять логарифмична. Предлагаю рассмотреть код рекурсивной реализации с встроенным Partition.

```
void QuickSort(std::vector<int>& array, int l, int h) {
  int i = l;
```

```
  int j = h;
  int pivot = array[(i + j) / 2];

  while (i <= j) {
    while (array[i] < pivot) {
      ++i;
    }
    while (array[j] > pivot) {
      --j;
    }
    if (i <= j) {
      std::swap(array[i], array[j]);
      ++i;
      --j;
    }
  }
  if (j > l) {
    QuickSort(array, l, j);
  }
  if (i < h) {
    QuickSort(array, i, h);
  }
}
```

Здесь опорный элемент выбирается крайне примитивно, просто середина массива.

Оптимизации быстрой сортировки

Быстрая сортировка предполагает несколько оптимизаций, вот некоторые из них:

1. На высоких глубинах рекурсии, когда куски маленькие, сортировать нерекурсивными сортировками. Отлично подходит казалось бы квадратичная сортировка вставками, так как массив почти отсортирован.

2. В случае большого числа равных элементов делать «толстое» разбиение на три части: строго меньшие, равные и строго большие.

Упражнение. Пусть зафиксированы два натуральных числа p и q. Пусть стратегия выбора пивота такова, что массив делится в соотношении $p : q$ каждый раз. Тогда время работы быстрой сортировки составит $\mathcal{O}(N \log N)$.

Теорема (б/д). Среднее время работы быстрой сортировки составляет $\mathcal{O}(N \log N)$, если опорный элемент выбирается равновероятно.

Поиск порядковой статистики

Def. k-й порядковой статистикой массива $a[]$ называют элемент, который после сортировки будет стоять на k-м месте.

Алгоритм поиска достаточно прост, будем использовать все то же разбиение, а именно:

1. Выбрать опорный элемент
2. Провести разбиение
3. Если индекс опорного элемента i окажется больше данного k, то нам надо искать слева k-ю порядковую, в случае равенства завершаемся, в противном случае надо искать $(i - k - 1)$-ю порядковую

Упражнение. Пусть зафиксированы два натуральных числа p и q. Пусть стратегия выбора пивота такова, что массив делится в соотношении $p : q$ каждый раз. Тогда поиск порядковой статистики может быть проведен за линейное время.

Упражнение. Покажите, что в среднем время работы такого алгоритма поиска k-й порядковой статистики равно $\mathcal{O}(N)$.

Медиана медиан

Данная оптимизация выбора опорного элемента позволит нам в худшем случае сортировать массив быстрой сортировкой за $\mathcal{O}(N \log N)$ времени. Как же она работает?

1. Разобьем массив на пятерки элементов.
2. В каждой пятерке выберем медиану (вторую порядковую статистику) руками. Получили массив медиан.
3. Для массива медиан рекурсивно строим массив его медиан, пока не сойдемся к одному элементу. Его назовем медианой медиан.

Утверждение. Медиана медиан гарантированно делит массив в соотношении не хуже чем $3 : 7$.

Доказательство: Сначала определим нижнюю границу для количества элементов, превышающих по величине опорный элемент x. В общем случае как минимум половина медиан, найденных на втором шаге, больше или равны медианы медиан x. Таким образом, как минимум $\frac{N}{10}$ групп содержат по 3 элемента, превышающих величину x, за исключением группы, в которой меньше 5 элементов и

ещё одной группы, содержащей сам элемент x. Таким образом получаем, что количество элементов больших x не менее $\frac{3n}{10}$.

■

Утверждение. Алгоритм нахождения k-й порядковой статистики с использованием медианы медиан работает за линейное время.

Доказательство: У нас есть три составляющих работы алгоритма на каждом шаге:

1. Время на разделение массива на пятерки и сортировка каждой из них: cN
2. Время на поиск медианы медиан $T\left(\frac{N}{5}\right)$
3. Время на поиск k-й порядковой не превзойдет времени его поиска в большей доле, то есть $T\left(\frac{7N}{10}\right)$

Таким образом,

$$T(N) = T\left(\frac{N}{5}\right) + T\left(\frac{7N}{10}\right) + cN$$

Покажем по индукции, что $T(N) \leq 10cN$. Подставим это соотношение и получим, что

$$T(N) = T\left(\frac{N}{5}\right) + T\left(\frac{7N}{10}\right) + cN \leq \frac{10cN}{5} + \frac{7 \cdot 10cN}{10} + cN = 10cN$$

■

Мы закончили обсуждать сортировки, основанные на сравнениях. Приведем краткое саммери

Сортировка	Стабильность	Среднее время	Худшее время	Стек. память	Дин. память
Пирамидальная	Нет	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$
Быстрая	Нет	$\mathcal{O}(N \log N)$	$\mathcal{O}(N^2)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$
Быстрая + ММ	Нет	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$
Слиянием	Да	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(N)$

Цифровая сортировка

Пусть известно, что множество объектов, массив из которых нам придется сортировать, невелико по размеру. Тогда сортировка массива может произойти достаточно тривиально, а именно посчитать, сколько раз какой элемент встречается в массиве и потом вывести по порядку нужное число раз. За сколько это работает? Пусть множество сортируемых элементов имеет размер K, тогда время равно $\mathcal{O}(N + K)$ и доппамять $\mathcal{O}(K)$. Очевидно, она ни разу не стабильна. Теперь напишем стабильную.

Пусть имеется массив A — исходный и массив B — куда будем писать ответ. Также заведем массив P размера N.

1. Пройдем по массиву A и запишем в $P[i]$ число объектов с ключом i.

2. Посчитаем префиксные суммы массива A, тогда мы знаем, начиная с какого индекса массива B надо писать структуру с ключом i.

3. Идем по массиву A и пишем в нужное место, поддерживая, куда писать следующий элемент с ключом k.

Данная сортировка стабильна по очевидным причинам.

Теперь научимся быстро сортировать массив чисел. Как мы знаем, *unsigned int* хранятся в виде строки из четырех байтов. Тогда давайте будем сортировать побайтово данные числа как двоичные строки, сначала сортируем по убыванию последнего байта, потом стабильно по убванию второго байта и так далее. После такого получим, что массив чисел отсортирован. Работает это за $\mathcal{O}(Nk)$ времени, где k это число байтов или 4, то есть за $\mathcal{O}(N)$.

Модуль 3. Деревья поиска.

Лекция 5.

Наивное дерево поиска

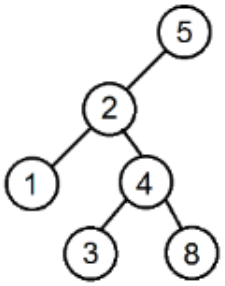
Def. *Деревом поиска* называют дерево, в котором в поддереве левее все элементы не больше элемента в данном узле, а в поддереве правее — строго больше.

Note. В данной книге будут рассматриваться только бинарные деревья поиска, в которых все элементы различны. Поддержка одинаковых элементов будет обсуждена на семинарски занятиях.

Задумаемся, каких операций мы хотим от дерева поиска и за какое время мы хотим их выполнять (h — высота дерева).

Операция	Время
Вставка элемента	$\mathcal{O}(h)$
Поиск элемента	$\mathcal{O}(h)$
Удаление элемента	$\mathcal{O}(h)$

Давайте посмотрим на пример такого дерева поиска



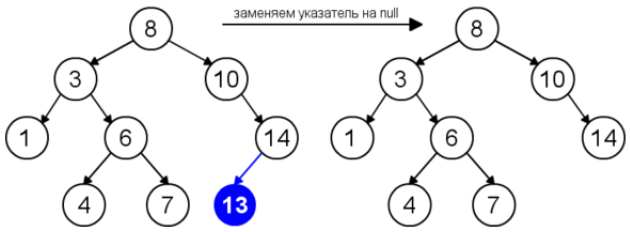
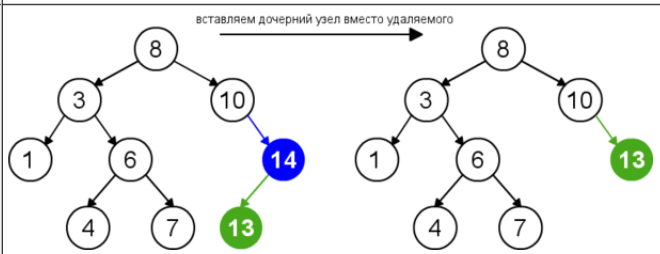
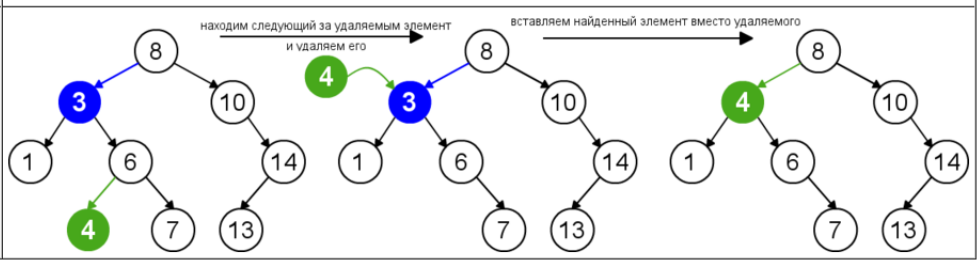
Как осуществлять поиск в таком дереве? Изначально мы стоим в корне, далее смотрим, искомый элемент больше или меньше элемента в корне. Если меньше, то перейдем к левому сыну, иначе к правому, и рекурсивно запустимся от него. Тогда, если мы в какой-то момент найдем элемент, то поиск завершен, иначе мы попытаемся рано или поздно пойти в узел, которого нет, что равносильно отсутствию элемента. Очевидно, что это удовлетворяет нашим требованиям на дерево.

Теперь обсудим вставку. Она выполняется абсолютно аналогично поиску, только вместо того, чтобы завершиться, когда идем в отсутствующий узел, мы его создаем в этом месте. Очевидно, это не нарушит инвариант дерева поиска и удовлетворяет нашим требованиям на асимптотику по времени.

Осталось обсудить удаление. Тут есть три случая:

1. Удаляемый узел не имеет детей, тогда все просто, удаляем его, ничего не сломав.
2. Удаляемый узел имеет ровно одного ребенка. Тогда подвешиваем сына удаляемого узла к родителю удаляемого узла и все.
3. Удаляемый узел имеет ровно двух детей. Тогда нам надо поставить на его место кого-то, чтобы не сломать дерево. Предлагается выбрать наименьший из элементов, больших данного, тогда это позволит нам не сломать инвариант дерева. Но как такой найти? Достаточно просто, один раз вправо и до упора влево. Заметим, что у такого узла не более одного ребенка (может быть правый есть), поэтому нам достаточно поменять местами значения в найденном узле и в удаляемом, а потом запустить удаление от найденного с помененным значением.

Рассмотрим красивую картинку, которую можно найти тут

Случай	Иллюстрация
Удаление листа	
Удаление узла с одним дочерним узлом	
Удаление узла с двумя дочерними узлами	

Теоретически все обсудили, давайте кодом что-нибудь напишем.

```
struct Node {
  int value = 0;
  Node* left = nullptr;
  Node* right = nullptr;
};

class NaiveBST {
 public:
  bool Exists(const int value) const {
    return ShiftDown(root_, value)->value == value;
  }
 private:
  static Node* ShiftDown(Node* cur_node, const int value) {
    if (value > cur_node.value && cur_node->right != nullptr) {
      ShiftDown(cur_node->right, value);
    } else if (value < cur_node.value && cur_node->left != nullptr) {
      ShiftDown(cur_node->left, value);
    }
    return cur_node;
  }
  Node* root_;
};
```

Как вы можете заметить, в реализации рекурсивный спуск вынесен отдельно, так как его придется использовать много раз. Более того, он сделан статическим, так как не требует никакой информации о структуре дерева. Ему достаточно дать узел, а далее он найдет в его поддереве нужный нам элемент.

Казалось бы, все хорошо, но можно построить такую последовательность вставок, что наивное дерево поиска выродится в бамбук, и, тем самым, высота будет составлять $\mathcal{O}(N)$ или же ноль преимуществ по сравнению с массивом.

Теорема (б/д). Если выбирать ключи равновероятно из какого-то множества и вставлять их алгоритмом выше, то высота дерева поиска составит $\mathcal{O}(\sqrt{N})$.

Задумаемся о том, что если бы дерево поиска было бы близко к полному бинарному, то его высота была бы порядка $\mathcal{O}(\log N)$. Именно эта идея и реализована в дереве ниже.

AVL-дерево

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

Теорема. Высота AVL-дерева равна $\mathcal{O}(\log N)$.

Доказательство: Рассмотрим такую величину как $S(h)$ — минимальная число вершин в AVL-дереве высоты h. Заметим, что верно следующее соотношение:

$$S(h+2) = S(h+1) + S(h) + 1$$

Оно верно, так как очевидно, что $S(h)$ растет монотонно вместе с h, а значит нам надо выбрать поддеревья разных глубин для минимизации числа вершин. Тогда у одного глубина составит $h+1$, а у другого, по опоределению, будет высота h. Единица отвечает за узел, являющийся корнем. Можно просто решить эту рекурренту, однако выведем соотношение проще.

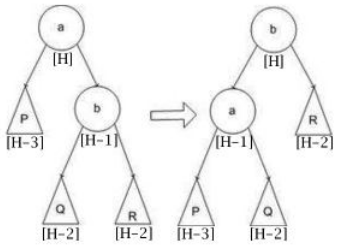
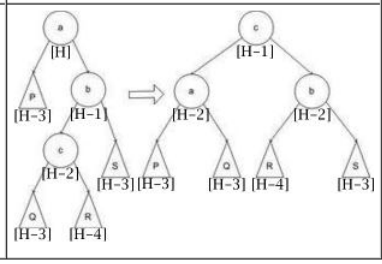
Докажем по индукции, что данную рекурренту можно выразить как $S(h) = F_{h+2} - 1$, где F_n — n-е число Фибоначчи. Для $h = 1$ база верна. Допустим, что $S(n) = F_{n+2} - 1$ для всех $n \leq h$, тогда

$$S(h+1) = S(h) + S(h-1) + 1 = F_{h+2} - 1 + F_{h+1} - 1 + 1 = F_{h+2} + F_{h+1} - 1 = F_{h+3} - 1$$

Таким образом, $S(h) = F_{h+2} - 1$. А про числа Фибоначчи известно многое. Например, что они растут экспоненциально быстро, что завершает доказательство.

■

Балансировка

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 $diff[c] = 0$

Пояснения к иллюстрации. $diff[v] = h(v.left) - h(v.right)$, где $h(v)$ — высота поддерева с корнем в вершине v.

Def. *Балансировкой* AVL-дерева будем называть процесс, который из дерева поиска, в узле которого разность высот поддеревьев равна двум, переподвешивает детей и внуков так, чтобы выполнялось свойство AVL-дерева.

Для балансировки AVL-дерева используются четыре операции, а именно четыре вида поворотов. На изображении выше (можно найти тут) даны два вида поворотов. Симметрично определяются два правых поворота.

Пусть есть AVL-дерево и мы в него вставили элемент как в наивное дерево поиска. Тогда соотношение высот могло поменяться только у тех узлов, которые лежат на пути от нового узла к корню. Поэтому давайте запустимся от вставленного узла и пойдем вверх, балансируя узлы на пути, применяя один из четырех видов поворота. Так как балансировка, как мы видим, не нарушает соотношение высот в поддеревьях на два уровня ниже, таким процессом мы не испортим уже сбаланисированное. А так как высота порядка логарифма, балансировка тоже произойдет за логарифм.

С удалением тоже все просто. Удаляем прям как из наивного дерева, только балансировку запускаем от самого нижнего узла, который подвергся модификации.

Лекция 6.

Декартово дерево поиска

Def. *Декартовым деревом* называют бинарное дерево, содержащее в себе пары (x_i, y_i), при этом данное дерево является деревом поиска по *ключам* (то есть значений x_i) и бинарной пирамидой по *приоритетам* (то есть по значениям y_i).

У декартова дерева есть две фундаментальные операции: split (разрезать декартово дерево на два декартовых) и merge (слить два декартовых дерева). Рассмотрим каждую из них подробнее.

Split

Операция Split принимает на вход декартово дерево T и ключ k, а возвращает пару декартовых деревьев T_1 и T_2 таких, что в T_1 все ключи не больше k, а в T_2 все ключи строго больше. Если $k > x\_max$ или же $k < x\_min$, то одно из деревьев окажется пустым, что не критично. Рассмотрим устройство данной операции. Пусть ключ в корне окажется меньше, чем ключ, по которому разрезаем, тогда:

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

Выполняется данная операция, очевидно, за $\mathcal{O}(h)$.

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 совпадет с девым поддеревом T_1, а правое 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_2.

Данная операция также выполняется за $\mathcal{O}(h)$.

Вставка и удаление

Обсудим вставку элемента.

1. $Split(T, value)$ получаем T_1, T_2.
2. Смотрим, совпадает ли самый правый элемент с k. Если да, то $Merge(T_1, T_2)$. Иначе пункт 3.
3. $T_3 = Tree(value)$. $Merge(Merge(T_1, T_3), T_2)$.

Как можно заметить, вставка полностью выражается через разрезания и слияния в нужных местах. Удаление абсолютно аналогично. Разрезаем по элементу, удаляем самую правую вершину (как в AVL-дереве), сливаем два дерева.

Таким образом, удаление и вставка работают за константное число операций сложности $\mathcal{O}(h)$ или же в общем за $\mathcal{O}(h)$.

Глубина декартова дерева

Теорема (б/д). В декартовом дереве из N узлов, приоритеты у которого являются случайными величинами с равномерным распределением, средняя глубина вершины $\mathcal{O}(\log N)$.

Построение

Пусть $x_1 < \ldots < x_N$. Построим декартово дерево быстрее, чем N вставок.

Будем строить дерево слева направо, то есть начиная с (x_1, y_1) до (x_n, y_n), при этом помнить последний добавленный элемент (x_k, y_k). Он будет самым правым, так как у него будет максимальный ключ, а по ключам декартово дерево представляет собой двоичное дерево поиска. При добавлении

(x_{k+1}, y_{k+1}), пытаемся сделать его правым сыном (x_k, y_k), это следует сделать если $y_k < y_{k+1}$, иначе делаем шаг к предку последнего элемента и смотрим его значение y. Поднимаемся до тех пор, пока приоритет в рассматриваемом элементе больше приоритета в добавляемом, после чего делаем (x_{k+1}, y_{k+1}) его правым сыном, а предыдущего правого сына делаем левым сыном (x_{k+1}, y_{k+1}).

Заметим, что каждую вершину мы посетим максимум дважды: при непосредственном добавлении и, поднимаясь вверх (ведь после этого вершина будет лежать в чьём-то левом поддереве, а мы поднимаемся только по правому). Из этого следует, что построение происходит за линейное время.

Splay-дерево

Смысл данного дерева в том, что доступ к данным, которые использовались недавно, будет осуществлен значительно быстрее.

Splay

Операция `Splay(x)` призвана путем комбинации поворотов различного типа сделать x корнем дерева. Рассимотрим типы поворотов:

- `zig` — применяется один раз, когда предок x корень. Нужен только если глубина x была изначально нечетной. Для этого совершается классический поворот вокруг ребра до родителя.
- `zig-zig` — применяется в ситуациях, когда прапредок, предок и x «лежат на одной прямой», тогда делается сначала поворот относительно ребра прапредок-предок, далее относительно предок-вершина.
- `zig-zag` — применяется в ситуациях, когда прапредок, предок и x «образуют угол», тогда делается сначала поворот относительно ребра предок-вершина, далее относительно прапредок-вершина.

То есть каждый раз x поднимается на 2 вверх и последний раз на 1 с помощью `zig`. Сам по себе `Splay(x)` является всего лишь последовательностью вызовов различных поворотов.

Теперь рассмотрим классические операции:

- `Find(x)` — самый обычный поиск в дереве, только потом вызываем `Splay(x)`.
- `Merge(Tl, Tr)` — вызовем `Splay(Tl.max)` (от самого правого элемента левого дерева), далее заметим, что тогда у него не будет правого сына, кем и станет второе дерево.
- `Split(x)` — вызовем `Splay(x)` и вернем двух детей корня.
- `Insert(x)` — вызовем `Split(x)` и подвесим два дерева как детей x, который станет корнем.

- `Erase(x)` — вызовем `Splay(x)`, далее вызовем `Merge` от двух его детей.

Теорема (б/д). Время операции `Splay` составляет $\mathcal{O}^*(\log N)$.

Модуль 4. Обработка запросов на отрезке.

Лекция 7.

RSQ и RMQ

В данном разделе мы будем изучать две классические задачи и две их постановки.

Задачи в большинстве своем будут делиться на

- `RMQ` или range min query — запрос минимума на подотрезке
- `RSQ` или range sum query — запрос суммы на подотрезке

И варианты задачи:

- `dynamic` — существуют запросы изменения элементов массива
- `static` — отсутствуют запросы изменения элементов массива

Задачу `static RSQ` мы уже умеем решать, достаточно просто вспомнить идею префиксных сумм, а вот с остальными мы еще не сталкивались.

Более того, у нас в алгоритмах будут естественным образом появляться две стадии: предподсчет и ответ на запрос.

Разреженная таблица

Разреженная таблица — двумерная структура данных $ST[i][j]$, построенная на бинарной операции F, для которой выполнено следующее:

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

Иначе говоря, в этой таблице хранятся результаты функции F на всех отрезках, длины которых равны степеням двойки. Объём памяти, занимаемый таблицей, равен $O(N \log N)$, и заполненными являются только те элементы, для которых $i + 2^j \leqslant N$.

Простой метод построения таблицы заключён в следующем рекуррентном соотношении:

$$ST[i][j] = \begin{cases} F(ST[i][j-1], ST[i+2^{j-1}][j-1]), & \text{если } j > 0; \\ A[i], & \text{если } j = 0; \end{cases}$$

Заметим, что для корректности данного соотношения необходимо, чтобы операция F удовлетворяла следующим свойствам:

- Коммутативность $F(a, b) = F(b, a)$
- Ассоциативность $F(a, F(b, c)) = F(F(a, b), c)$
- Идемпотентность $F(a, a) = a$

Заметим, что задачу RSQ нельзя решать с помощью этой структуры, так как сумма неидемпотентна.

Выполним сначала предподсчет, суть которого в вычислении массива $fl\_log[j] = \lfloor \log_2 j \rfloor$. Таким образом, построение и предподсчет занимают $O(N \log N)$ времени.

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

$F(A[l], A[l+1], \ldots, A[r]) = F(ST[l][j], ST[r-2^j+1][j])$, где $j = fl\_log[r-l+1]$

Таким образом, ответ на запрос дается за $O(1)$.

Дерево отрезков

Дерево отрезков способно за $\mathcal{O}(\log N)$ получать на подотрезке результат любой операции, которая ассоциативна, коммутативна и имеет нейтральный элемент. В частности, задача RSQ, как правило, решается с использованием этой структуры данных.

Построение

Опишем нерекурсивное 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 заведем массив длины $2N-1$ и будем его заполнять таким образом, что последние N элементов будут элементами исходного массива, а первые $N-1$ элементов заполним следующим образом: $t[i] = F(t[2i+1], t[2i+2])$ (заполнение от элемента с номером $N-2$ и до 0 в цикле)

Заполним массив дерева отрезов для операции минимум и массива:

a[ ]	—	—	—	—	—	7	1	8	2	3	4
t[ ]	1	2	1	2	3	7	1	8	2	3	4

Обработка операции сверху

Внимание, тут будет дерево отрезков храниться не как массив результатов, а как настоящее дерево из узлов с указателями на детей. Приведем сразу псевдокод, а потом поясним его:

```
int query(int node, int a, int b) {
    l = tree[node].left
    r = tree[node].right
    if (intersection([l, r), [a, b)) is empty)
        return neutral
    if ([l, r) is subset [a, b))
        return tree[node].res
    return f(query(node * 2 + 1, a, b), query(node * 2 + 2, a, b))
}
```

Теперь поясним, что тут происходит. Возможны три ситуации:

- Полуинтервал, за который отвечает вершина, не пересекается с искомым, значит вернем нейтральный элемент
- Полуинтервал, за который отвечает вершина, входит целиком в искомый, значит вернем значение в этой вершине
- Иначе вернем результат операции на детях

Заметим, что по сути мы разбиваем исходный интервал на дизъюнктное объединение интервалов таких, что глубина вершин, отвечающих за каждый «подынтервал» минимальна. То есть спускаемся от корня вниз, сливаем результаты в поддеревьях, пока не поднимемся до корня.

Обновление элемента

Рассмотрим на примере, какие индексы надо изменить:

i[]	0	1	2	3	4	5	6	7	8	9	10
a[]	—	—	—	—	—	7	1	8	2	3	4
t[]	1	2	1	2	3	7	1	8	2	3	4
a'[]	—	—	—	—	—	7	1	8	**6**	3	4
t'[]	**1**	**1**	1	**6**	3	7	1	8	**6**	3	4

Тогда, зная индекс листа i, который надо изменить, заметим, что индекс элемента, который придется заменить: $\lfloor \frac{i-1}{2} \rfloor$. Тогда можно просто написать функцию, аналогичную подъему по дереву во время построения снизу, только она будет обновлять значения от листьев до корней, пересчитывая, какой элемент надо изменить, а потом высчитывать его новое значение (заметим, что все значения глубины ниже уже известны).

Массовое (групповое) обновление

Договоримся, что будут две функции: *op* — функция, по которой строили дерево отрезков, и функция *ch*, по ней будут обновлять **отрезки** элементов исходного массива. Также договоримся, что помимо ограничений на операцию *op* (ассоциативность, коммутативность и существование нейтрального), на операцию *ch* также наложим ограничения:

1. Существование нейтрального элемента
2. $ch(a, ch(b, c)) = ch(ch(a, b), c)$
3. $ch(op(a, b), c) = op(ch(a, c), ch(b, c))$

В каждой вершине, помимо непосредственно результата выполнения операции *ch*, будем хранить несогласованность — величину, с которой нужно выполнить операцию *op* для всех элементов текущего отрезка. Тем самым мы сможем обрабатывать запрос массового обновления на любом подотрезке эффективно, вместо того чтобы изменять все $\mathcal{O}(N)$ значений. Как известно из определения несогласованных поддеревьев, в текущий момент времени не в каждой вершине дерева хранится истинное значение, однако когда мы обращаемся к текущему элементу мы работаем с верными данными. Это обеспечивается «проталкиванием» несогласованности детям (процедура *push*) при каждом обращений к текущей вершине. При этом после обращения к вершине необходимо пересчитать значение по операции *ch*, так как значение в детях могло измениться.

Опишем процедуры для осуществления задуманного:

- Рассмотрим проталкивание, цель которого сделать несогласованность текущей вершины нейтральной

```
void push(int node) {
    tree[2 * node + 1].d = ch(tree[2 * node + 1].d, tree[node].d);
    tree[2 * node + 2].d = ch(tree[2 * node + 2].d, tree[node].d;
    tree[node].d = ch_neutral
}
```

- Процедура обновления на отрезке. Данная процедура выполняет разбиение текущего отрезка на подотрезки и обновление в них несогласованности. Очень важно выполнить push как только идет рекурсивный вызов от детей, чтобы избежать некорректной обработки в детях. И так как значение в детях могло измениться, то необходимо выполнить обновление ответа по операции *op* на текущем отрезке.

```
void update(int node, int a, int b, T val) {
    l = tree[node].left;
    r = tree[node].right;
```

```
    if ([l, r) intersect [a, b) is empty)
        return;
    if ([l, r) is subset [a, b)
        tree[node].d = ch(tree[node].d, val);
        return;
    (1)
    push(node);
    (2)
    update(2 * node + 1, a, b, val);
    update(2 * node + 2, a, b, val);
    (3)
    tree[node].ans = op(ch(tree[2 * node + 1].ans, tree[2 * node + 1].d),
                        ch(tree[2 * node + 2].ans, tree[2 * node + 2].d));
}
```

Прокомментируем данный код:

1. Разбили искомый отрезок на дизъюнктное объединение интервалов
2. Протолкнули несогласованность в детей
3. Заметим, что на данный момент в узле несогласованность нейтральна(!), а значит нам нужно получить истинный ответ в этом узле, который получается путем ответов на запрос для детей (заметим, что тут мы гарантируем тот факт, что при проталкивании мы протолкнули определенность так, чтобы измененные несогласованности давали верный ответ (верно из свойств операции ch)

- Получение запроса. Осталось рассмотреть последнюю (и самую нужную операцию) — получение ответа на запрос. Заметим, что в ней логика та же, что и в обновлении, только возвращаются другие значения (в данном случае возвращаются ответы из поддеревьев), а в обновлении выполняли проталкивание несогласованности так, чтобы она появилась как можно выше в поддеревьях, в которых был запрос на изменение.

```
int query(int node, int a, int b) {
    l = tree[node].left;
    r = tree[node].right;
    if ([l, r) intersect [a, b) is empty)
        return op_neutral;
    if ([l, r) is subset [a, b)
        return ch(tree[node].d, tree[node].ans);
    push(node);
    ans = op(query(node * 2 + 1, a, b), query(node * 2 + 2, a, b));
```

```
        tree[node].ans = op(ch(tree[2 * node + 1].ans, tree[2 * node + 1].d),
                            ch(tree[2 * node + 2].ans, tree[2 * node + 2].d));
        return ans;
    }
```

Лекция 8.

Дерево Фенвика

Познакомимся с еще одной структурой, позволяющую решать задачу `dynamic RSQ`, а именно, с деревом Фенвика. Наложим требования на операцию:

- Ассоциативная
- Существует нейтральный элемент
- Обратимая

В чем смысл такой структуры, если есть дерево отрезков, которое накладывает меньшие требования на операцию? Ну ответ прост:

1. «Быстрее пишется»
2. Константа в $\mathcal{O}$-большом меньше
3. По памяти весит меньше

Для того, чтобы обсуждать дерево Фенвика, нам необходимо ввести две функции:

$$f(x) = x \ \& \ (x+1)$$
$$g(x) = x \mid (x+1)$$

Введем такую величину как $S(i) = op(a_{f(i)}, \ldots, a_i)$, где $a = [a_0, \ldots, a_{n-1}]$ — данный массив.

В силу обратимости операции $op(l, r) = op^{-1}(op(0, r), op(0, l-1))$, где $op(l, r) = op(a_l, \ldots, a_r)$, тогда осталось научиться считать на операцию на префиксе. Поймем, как это свести к $S(i)$. Осознаем следующую формулу:

$$op(0, k) = op(S(k), S(f(k)-1), S(f(f(k)-1)-1), \ldots)$$

Как это можно понимать? $S(k)$ это результат на подотрезке $[f(k), k]$, тогда следующий подотрезое это. $[f(f(k)-1), f(k)-1]$ и так далее.

Теперь в чем глубинный смысл $f(i)$? Рассмотрим двоичную запись i и посмотрим на его младший бит. Если он равен нулю, то $f(i) = i$. Иначе двоичное представление числа i оканчивается на группу из одной или нескольких единиц. Заменим все единицы из этой группы на нули, и присвоим полученное число значению функции $f(i)$, то есть $f(i)$ зануляет все младшие единички. Ну понятное дело, что $op(0, k)$ считается по сути за число единичек в числе k, то есть за $\mathcal{O}(\log N)$.

Теперь заметим, что изменение в точке i затронет только те $S(j)$, для которых $f(j) \le i \le j$, как находить такие числа быстро?

Утверждение. Все такие числа j получаются из i последовательными заменами самого правого (самого младшего) нуля в двоичном представлении. А такой операцией как раз является $g(i)$:)

Доказательство:

$\Longrightarrow$ Очевидно, что $f(g(g(i))) \le f(g(i)) \le i \le g(i) \le g(g(i))$, так как $f(g(i))$ либо уберет только одну единичку, то есть i оканчивалось на 0_2, либо же $f(g(i))$ занулит весь блок из единиц в конце, а в i хотя бы одна такая была (хотя бы последняя). Вторая часть неравенства доказывается из того, что $g(i) > i$. Более того, значит рано или поздно дойдем до N, причем за $\mathcal{O}(\log N)$ шагов.

Рассмотрим $i = 10$.

$$i = 001010_2 = 10$$
$$j = g(i) = 001011_2 = 11$$
$$f(j) = 001000_2 = 8$$

На второй итерации:

$$i = 001010_2 = 10$$
$$j = g(g(i)) = g(j) = 001111_2 = 15$$
$$f(j) = 000000_2 = 0$$

$\Longrightarrow$ Рассмотрим произвольное число k, которое не будет получено из i алгоритмом выше, покажем, что для него неверно, что $f(k) \le i \le k$.

Рассмотрим общий префикс двоичной записи i и k, если он занимает всю длину записи, то $i = k$ и такое число мы бы рассмотрели, противоречие. Если различный суффикс единичной длины? То есть $i' = 0$, а $k' = 1$, тогда такое k как раз могло быть полученое как $g(i)$.

Теперь имеется различный суффикс длины большей единицы и k не может быть получено из i такими операциями, причем $i < k$. Значит $k' = 1\ldots_2$ и $i' = 0\ldots_2$ (иначе $k' \le i'$, что неверно). Так как k не может быть получено из i, значит есть такое место, что $i' = 0\ldots1\ldots_2$ и $k' = 1\ldots0\ldots_2$, а значит расстановка такая:

$$i' = 0\ldots1\ldots_2$$
$$k' = 1\ldots0\ldots_2$$

Тогда $f(k')$ не сможет никак занулить самую левую единицу, а значит $f(k') > i' \Longrightarrow f(k) > i$, что завершает доказательство.

■

Ну и все, получаем, что операции обновления и запроса на отрезке работают за $\mathcal{O}(\log N)$, причем быстрее, как правило, так как пропорциональны числу нулей или единиц в двоичной записи чисел.

Все еще в голове сидит вопрос, зачем мы так запаривались, если все было интуитивно?

Немного кода

```
class FenwickTree {
public:
  int GetSum(int l, int r) { return GetPref(r) - GetPref(l); }
  void Update(int idx, val) { UpdateDelta(idx, val - array_[i]); }

private:
  int GetPref(int idx) {
    int ans = 0;
    for (i = idx; i >= 0; i = f(i) - 1) { ans += tree_[i]; }
    return ans;
  }
  void UpdateDelta(int idx, int delta) {
    for (int j = idx, j < n; j = g(j)) { tree_[j] += val; }
  }

  std::vector<int> tree_;
  std::vector<int> array_;
}
```

Повышаем размерность

Пусть у нас решили спрашивать многомерные запросы на многомерных массивах и обновления в точке, тогда заведем следующую штуку:

$$S(i,j) = \sum_{u=f(i)}^{i} \sum_{v=f(j)}^{j} a_{ij}$$

Тогда очевидно, что сумма на прямоугольнике может выражаться как сумма через «префиксные многоугольники» по формуле включений-исключений, а ответ на «префиксном многоугольнике» будет просто суммой по всем (i,j), которые пересчитываются по формуле выше, а именно:

```
int GetPref(int x, int y) {
    int ans = 0;
    for (i = x; i >= 0; i = f(i) - 1) {
      for (int j = y; j >= 0; j = f(j) - 1) {
        ans += tree_[i][j];
      }
    }
    return ans;
  }
```

Абсолютно аалогично дописывается обновление. Ну и все, победа.

Лекция 9.

Декартово дерево по неявному ключу

Время пройти самую шикарную структуру данных для работы с отрезками.

Вспомним, что если обойти дерево поиска в порядке обхода, то получим, что у нас отсортированный массив ключей. Давайте считать, что ключи это числа от 0 до $N-1$, тогда порядок обхода дерева по сути задает массив, в котором можно хранить значения. Казалось бы, крайне странная идея, зачем нам еще более медленный массив?

Давайте переделаем операцию `Split`, раз у нас ключи стали «индексами» в массиве, то мы можем понять, как отрезать ровно k элементов от массива. Реализация будет полностью аналогичной реализации `Split` в обычном декартовом дереве. Таким образом, можно отрезать от массива часть за логарифм! Но не очень понятно, что делать с индексами в таком случае, ведь они собьются:(

Казалось бы, придется пройти за линейное время по оставшемуся дереву и обновить все индексы. Решение — неявный ключ! Давайте хранить в узле не индекс элемента в массиве, а число элементов в поддереве. ЧТО?!?!?!?!?!? Давайте осознаем, что этой информации более чем достаточно, чтобы получить k-й элемент в массиве, так как это будет «k-я порядковая статистика» в новом дереве. Тогда понятно, как делать запросы взятия по индексу и отрезания края массива, но какой-то странный набор операций.

Мы еще ни разу не сказали о `Merge`, а что он делает? Сливает два дерева за логарифм, если ключи одного дерева не больше ключей в другом и смотрит **только** на приоритеты. Но стоп, у нас же нет ключей, есть что-то странное в виде размеров поддеревьев. То есть операция `Merge` обманута самым наглым образом, она ждет два набора отсортированных ключей, при этом на них выполнено еще соотношение, но мы ей даем дерево **совсем без ключей**. Отработает ли она корректно? Да! По сути `Merge` будет конкатенировать два массива за логарифм!

Подведем краткое резюме, что мы умеем?

Операция	Время
Отрезание конца массива	$\mathcal{O}(\log N)$
Конкатенация массивов	$\mathcal{O}(\log N)$
Получение по индексу	$\mathcal{O}(\log N)$

А теперь время ~~непростительных~~ великих колдунств. Давайте вспомним, как в декартовом дереве работают `Split` и `Merge`, они просто нарезают дерево, а потом сливают. В нашем случае есть взятие по индексу, а значит мы можем делать следующие операции:

Операция	Время
Отрезание конца массива	$\mathcal{O}(\log N)$
Конкатенация массивов	$\mathcal{O}(\log N)$
Получение по индексу	$\mathcal{O}(\log N)$
Удаление по индексу	$\mathcal{O}(\log N)$
Вставка по индексу	$\mathcal{O}(\log N)$
Перестановка подотрезка	$\mathcal{O}(\log N)$

Не круто ли это? Получили самый настоящий *быстрый массив*.

Продолжим дальше совершенствовать нашу структуру. Давайте осознаем, что каждый узел в поддереве содержит подотрезок исходного массива, кто из пройденных нами структур данных имеет схожее строение? Не поверите, но дерево отрезков! Например, давайте хранить в узле не только элемент массива, но и сумму элементов в поддереве, то есть *сумму на подотрезке*, тогда можно получать сумму на произвольном подотрезке по аналогии с деревом отрезков, даже код не изменится почти! Только надо понимать, сколько элементов левее нас, чтобы получать корректно границы подотрезка, за который отвечает вершина, но это снова выводится из размеров поддеревьев!!! Значит мы можем еще брать сумму на подотрезке за логарифм! Более того, любой запрос на ДО можно так интерпретировать! Получаем быстрый массив со всеми возможностями дерева отрезков, победа ли это? Почти.

Вспомним, что дерево отрезков позволяло групповые обновления, верно ли это для нашей новой структуры? Оказывается, да, проталкивание совсем не изменится! А значит мы можем еще групповым образом обновлять элементы на отрезке. Более того, мы не просим никаких дополнительных свойств на операции, то есть мы делаем все с той же асимпотиткой, только получили возможность издеваться над исходным массивом!

Резюме. Мы получили массив, который умеет делать все то же, что и дерево отрезков, только в массиве можно переставлять элементы местами согласно правилам выше!

Модуль 5. Геометрия и хеши

Лекция 9 (часть 2).

Прямые, отрезки. Пересечения

Прямые

Две прямые могут быть либо параллельными, либо пересекаться. Как это определить? Пусть различные точки X_1, X_2 лежат на первой прямой, а Y_1, Y_2 на второй. Тогда параллельность прямых равносильна параллельности $\overrightarrow{X_1X_2}$ и $\overrightarrow{Y_1Y_2}$, а она в свою очередь равносильна тому, что они либо сонаправлены, либо противонаправлены.

В любом из двух случаев для параллельности синус угла между векторами должен быть нулевым. Таким образом, прямые пересекаются тогда и только тогда, когда между ними угол с отличным от нуля синусом (или векторным произведением).

Если вдруг нам надо найти точку пересечения, то для оптимальности написания кода предлагается рассмотреть два случая:

1. Хотя бы одна прямая вертикальна — тут все просто, ее вид $x = A$, подставляем во вторую
2. Обе прямые не вертикальные, тогда приводим их к виду $y = k_i x + b_i$ и решаем систему в таком виде.

Точка может лежать на прямой, а может и не лежать. Пусть B, C — две различные точки на прямой, а точка A — исследуемая точка. Тогда то, что точка лежит на прямой, равносильно тому, что площадь $\triangle ABC$ равна нулю (то есть снова векторное произведение).

Отрезки

В первую очередь проверяем, лежат ли концы отрезка на прямой. Далее начинается часть интереснее. Отрезок пересекает прямую тогда и только тогда, когда его концы находятся по разные стороны от прямой.

Пускай прямая проходит через точки A и B, а концы отрезка назовем C и D. Тут придется поднапрячься и вспомнить определение синуса через единичную окружность. Рассмотрим вектора $b = \overrightarrow{AB}$, $c = \overrightarrow{AC}$ и $d = \overrightarrow{AD}$. Также введем для формальности следующую конструкцию: пусть вектор b направлен вдоль оси OX, а точка A это начало координат (расположение отрезка и прямой не зависит от сдвига и поворота). Тогда мы можем ввести ось OY привычным нам образом.

Теперь пересечение отрезка и прямой возможно тогда и только тогда, когда один из векторов c и d направлен в нижнюю полуплоскость относительно OX, а второй — в верхнюю. Но как проверять,

куда направлен вектор? Используя определение синуса, получаем, что направленность «вверх» равносильна тому, что он больше нуля, и наоборот.

То есть нам нужно, чтобы одно из векторных произведений было отрицательным, а второе положительным, что равносильно отрицательности их произведения.

Разобьем проверку того, что точка лежит на отрезке, на два шага:

1. Проверим, что точка лежит на прямой, содержащей отрезок
2. Пусть точка X, а отрезок AB, тогда нам необходимо и достаточно, чтобы вектора $\overrightarrow{AX}$ и $\overrightarrow{XB}$ были сонаправлены. Заметим, что для сонаправленности нам необходимо проверить не только равенство синусу нулю, но и еще то, что косиунус равен единице (или хотя бы скалярное произведение положительно).

В данном конспекте нет цели сделать что-то прям сверхбыстро, наша цель — достичь понимания. Обычно это самое неприятное место, но мы обойдем его оригинальным способом.

1. Проверяем, что прямые, содержащие отрезки, пересекаются. Находим точку пересечения (как сказано выше).
2. Проверяем, что точка пересечения принадлежит обоим отрезкам

Второй вариант есть в книге Кормена через лес if-ов.

Многоугольники. Проверка на выпуклость. Принадлежность точки

Выпуклость

Многоугольник выпуклый, если повороты все время выполняются в одну сторону. Ничего не понятно, давайте рассмотрим подробнее. Зафиксируем ногоугольник $P_1 \dots P_n$. Рассмотрим то, куда поворачивает вектор $\overrightarrow{P_2P_3}$ относительно вектора $\overrightarrow{P_1P_2}$. Пускай в верхнюю полуплоскость (мы уже выше вводили их), тогда далее все $\overrightarrow{P_iP_{i+1}}$ должны поворачивать в верхнюю полуплоскость относительно $\overrightarrow{P_{i-1}P_i}$.

Площадь многоугольника

Рассмотрим два метода:

1. Через ориентированные площади. Рассмотрим произвольную точку плоскости. A Например, одну из вершин многоугольника. Тогда площадь многоугольника $P_1 \dots P_n$ определяется как сумма *ориентированных* площадей треугольников AP_iP_{i+1}.
2. Через трапеции. Площадь многоугольника равна модулю суммы *ориентированных* площадей трапеций, порождаемых сторонами многоугольника. То есть $\frac{(P_{i+1}.x-P_i.x)(P_{i+1}.y+P_i.y)}{2}$.

Принадлежность точки. Общий случай

Пустим из точки луч вдоль оси OX и посчитаем, сколько раз луч пересекает рёбра многоугольника. Для этого достаточно пройтись в цикле по рёбрам многоугольника и определить, пересекает ли луч каждое ребро. Если число пересечений нечётно, то объявляется, что точка лежит внутри многоугольника, если чётно — то снаружи.

Единственная проблема, если мы случайно попали в вершину многоугольника, тогда предлагается пустить случайный луч с целочисленным угловым коэффициентом и для него все посчитать. Утверждается, что вы должны быть крайне невезучим человеком, чтобы попасть еще раз в вершину многоугольника.

Принадлежность точки. Выпуклый многоугольник

Выберем самую нижнюю (если таких несколько, то самую левую среди них) вершину, теперь будем считать, что с нее начинается обход. Рассмотрим *полярные* углы, то есть углы между $\overrightarrow{P_1P_i}$ и между положительным направлением оси OX. Они строго возрастают, а их косинусы строго убывают (попробуйте сами показать это). Сделаем предподсчет массива косинусов полярных углов, тогда этот массив будет отсортирован по убыванию (можно развернуть его для упрощения жизни).

Теперь пусть спрашивают, лежит ли точка внутри многоугольника. Считаем косинус полярного угла, бинпоиском находим, между какими двумя он окажется. Теперь получаем, что у нас есть треугольник $\triangle P_1P_iP_{i+1}$ и надо проверить, лежит ли точка внутри. Тут уже можете проверять как угодно, можно хоть пользоваться алгоритмом выше, все равно число сторон константа.

Лекция 10.

Выпуклая оболочка на плоскости

Def. Множество S выпукло, если $\forall x, y \in S$ отрезок от x до y также лежит в S.

Def. Выпуклой оболочкой множества точек называют минимальное по мере множество S такое, что S выпукло и при этом оно содержит себе все точки исходного множества.

Далее мы будем рассматривать выпуклые оболочки на плоскости, тогда можно сказать, что нас интересует выпуклый многоугольник минимальной площади, внутри которого (или на границе) находятся все точки из множества. Но как искать такой многоугольник?

Утверждение. Многоугольник выпуклый, если для каждой прямой, проходящей через стороны многоугольника, верно, что весь многоугольник лежит в одной полуплоскости относительно нее.

Алгоритм Джарвиса

1. Найдем самую нижнюю точку P_0, она, очевидно, будет в выпуклой оболочке, так как иначе

наша построенная оболочка не будет ее содержать.

2. Рассмотрим луч, параллельный оси OX из точки P_0, найдем вершину с минимальным полярным углом относительно данного луча, P_1, тогда относительно прямой P_0P_1 все точки будут лежать в верхней полуплоскости.

3. Рассмотрим продолжение луча P_0P_1 за точку P_1, относительно этого луча теперь найдем точку с минимальным полярным углом P_2.

4. Повторяем шаг 3, пока не окажется, что $P_h = P_0$.

Пусть h — число вершин в выпуклой оболочке, тогда время работы алгоритма составит $\Theta(nh) = \mathcal{O}(n^2)$.

Алгоритм Грехема

1. Отсортируем точки по полярному углу, обозначим их $P_0, \ldots, P_{n-1}$

2. Сложим в стек S точки P_0 и P_1, они обязательно войдут в выпуклую оболочку.

3. Рассмотрим точку P_i. Если угол $PrevTop(S), Top(S), P_i$ стал более чем 180 градусов, то делаем $Pop(S)$.

4. Повторяем шаг 3 для всех точек.

5. Получим, что в стеке лежит выпуклая оболочка.

Время работы составляет $\mathcal{O}(n \log n)$ на сортировку и $\mathcal{O}(n)$ на проход стеком, так как каждая точка будет добавлена в стек один раз и удалена не более одного раза.

Сумма Минковского

Def. Рассмотрим два множества $U, V \subset \mathbb{R}^n$, тогда суммой Минковского называют множество

$$U \oplus V = \{u + v \mid u \in U, v \in V\}$$

Теорема. Сумма Минковского двух выпуклых многоугольников S, T — выпуклый многоугольник из $|S| + |T|$ вершин.

Доказательство: Рассмотрим вектор $d \in \mathbb{R}^2$. Рассмотрим семейство прямых $\mathcal{P}$, для которых d является нормалью. Тогда крайней точкой многоугольника в направлении d назовем такую вершину A, что если через нее провести прямую из $\mathcal{P}$, то весь многоугольник окажется в одной полуплоскости.

Заметим, что если рассмотреть крайние вершины S и T в направлении d, то их сумма окажется тоже крайней вершиной $S \oplus T$ в данном направлении. Тогда можно заметить, что если рассмотреть

d такой, что он ортогонален одной из сторон S, тогда данная сторона будет состоять из крайних точек в направлении d, а значит и данная сторона будет крайней в $S \oplus T$.

Из наблюдения выше рассмотрим вектора, ортогональные каждой из $|S| + |T|$ сторон, тогда они будут крайними в $S \oplus T$, откуда следует, что в $S \oplus T$ будет не больше $|S| + |T|$ вершин. Более того, выпуклость вытекает из построения того, что стороны были крайними.

■

Из доказательства теоремы вытекает алгоритм построения:

1. Для S найдем крайнюю вершину в каком-либо направлении, например, в вертикальном. То есть самую нижнюю точку. Сделаем циклический сдвиг вектора точек так, чтобы она шла первой. Аналогично для T. Это займет $\mathcal{O}(|S| + |T|)$ времени.
2. Заметим, что тогда вектора S_iS_{i+1} и T_jT_{j+1} отсортированы по полярному углу в рамках своих многоугольников.
3. Сольем эти два массива, отложенных от точки $S_0 + T_0$, получим последовательные вершины $S \oplus T$

Корректность алгоритма вытекает из того, что мы будто поворачиваем вектор d от направления вниз против часовой стрелки и откладываем последовательно крайние стороны.

Лекция 11.

Идея хеширования

Глобальной задачей этой части программы будет построение структуры данных, которая умеет быстро проверять, имеется ли элемент в множестве. Более того, мы будем хотеть, чтобы эта проверка несильно зависила от размера объекта. Тогда естественно возникает вопрос, например, как сделать такую структуру быстрой со строками?

Осознаем, что мы умеем быстро сравнивать только целые числа, отсюда и вытекает идея хеширования. Более того, эти числа должны быть несильно большими. Поэтому введем следующее определение.

Def. Пусть U — множество рассматриваемых объектов (например, все строки), тогда $h : U \to \{0, 1, \ldots, k-1\}$ называется хеш-функцией.

Def. Элементы $x, y \in U$, где $x \neq y$ образуют *коллизию*, если $h(x) = h(y)$.

Допустим мы ввели как-то хеш-функцию, не накладывая ограничений, тогда первый вариант такой структуры данных: завести массив размера k и говорить, что элемент v есть в множестве, если по индексу $h(v)$ уже занята ячейка.

Def. Схема, введеная выше, называется *прямой адресацией (direct addressing)*.

У данной схемы есть множество проблем. Одна из них — то, что мы требуем $\mathcal{O}(k)$ дополнительной памяти, что является непозволительной роскошью. Вторая — огромное множество коллизий, с которыми мы не умеем пока справляться.

Решение первой проблемы заключается в том, что мы заводим массив определенного небольшого размера m и кладем элемент по индексу $h(x)\%m$. Давайте обсудим вторую.

Хеш-таблица на цепочках

Достаточно понятно, что если у нас коллизия, то нам нужно хранить все данные для данного хеша. Давайте хранить по индексу не элемент, а цепочку из данных (например, `std::list`, не вектор, так как придется еще удалять). Тогда необходимо отметить, что длинные цепочки это плохо, так как в них поиск осуществляется уже за линейное время от длины цепочки.

Def. *Бакетом (bucket)* называют нечто, хранящее все элементы, образующие коллизию.

Хочется как-то минимизировать максимальную длину цепочки, то есть добиться того, чтобы хеш-функция примерно равномерно раскидывала ключи по бакетам. Понятно, что максимальную длину оценивать крайне трудно, так что будем оценивать среднюю длину цепочки. То есть нам нужны ваероятности, но мы нигде из не вводили. Давайте построим модель.

Пусть $\mathcal{H} = \{h : U \to \{0, 1, \dots, k-1\}\}$ — наше множество хеш-функций и мы хотим уметь выбирать случайную (sic!) хеш-функцию из него.

Note. Далее для анализа нам важно, что $U \subset \mathbb{N}$ и $|U| < \infty$, то есть $U = \{0, 1, \dots, n-1\}$.

В таких предположениях на самом деле можно сказать, что хеш-функция — отображение $\{0, 1, \dots, n-1\} \to \{0, 1, \dots, k-1\}$, то есть мы можем задать наше отображение таблицей соответствия ключ-образ. Тогда мы можем определить случайную хеш-функцию как то, что образ для каждого ключа это случайный элемент из множества $\{0, 1, \dots, k-1\}$. Такие выборы независимы в совокупности и при этом они из равномерного распределения на образах.

Def. Модель, описанная выше, называется *простое равномерное хеширование (simple uniform hashing)*.

Теперь все оценки предполагают усреднение по всем возможным хеш-функциям.

Рассмотрим величину L_q — длина цепочки, отвечающая ключу q. Пусть таблица построена для различных ключей $k_1, \dots, k_n$, тогда

$$L_q = \sum_i I(h(q) = h(k_i))$$

Посчитаем матожидание длины цепочки.

$$\mathbb{E}L_q = \mathbb{E}\sum_i I(h(q) = h(k_i)) = \sum_i P(h(q) = h(k_i))$$

Посчитаем вероятность выше.

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

Тогда, с учетом того, что все ключи k_i выше различны, получаем, что

$$\mathbb{E}L_q = \mathbb{E}\sum_i I(h(q) = h(k_i)) = \sum_i P(h(q) = h(k_i)) \leq 1 + \frac{n-1}{k} \leq 1 + \frac{n}{k}$$

Def. *Коэффициентом загруженности (load factor)* называют величину $\alpha = \frac{n}{k}$.

Note. Из оценок выше мы для хеш-таблицы заведомо задаем какую-то константу C такую, что всегда $\alpha < C$. Тогда в среднем сложность операций выше составит $\mathcal{O}(1)$.

Можно было бы сказать, что мы достигли победы и все построили, но давайте задумаемся о том, а сколько памяти мы потребляем, раз со временем все хорошо. Понятно, что памяти $\mathcal{O}(k)$. Но есть нюанс, мы должны полностью хранить хеш-функцию, то есть $\mathcal{O}(|U|)$, что крайне много. Более того, возникает вопрос, а зачем тогда брать остаток по модулю k (размер внешнего массива), раз мы его не используем никак:) Тогда уж пользуйтесь прямой адресацией.

Упражнение. Подумайте, чем плох подкод генрировать в режиме онлайн случайное значение для пришедшего ключа? Тогда не придется хранить весь массив. Осознайте, что, решая данную задачу, вы вернулись к задаче этой лекции.

Осознайте, что мы пришли к проблеме, что simple uniform hashing это всего лишь модель, в которой можно строить оценки, но нельзя ее никак воплотить на практике! Почему? В связи с тем, что память, потребляемая в данной модели на хранение случайной хеш-функции составляет $\mathcal{O}(\log k^{|U|}) = \mathcal{O}(|U| \log k)$, откуда вытекает проблема.

Универсальное семейство хеш-функций

Осознаем, что если задана хеш-функция, удовлетворяющая свойствам выше, то мы умеем строить хеш-таблицу, осталось научиться строить функцию. Новый план, призванный избавиться от старой проблемы — рассмотреть параметризуемое подмножество всех возможных хеш-функций. Но не произвольное. Заметим, что нас устроет следующее соотношение:

$$\forall x \neq y \; P_{h \in \mathcal{H}}(h(x) = h(y)) \leq \frac{\lambda}{k}$$

Def. Семейство $\mathcal{H}$ хеш-функций называют λ-универсальным, если верно соотношение сверху.

Def. Семейство $\mathcal{H}$ хеш-функций называют универсальным, если верно соотношение сверху для $\lambda = 1$.

Осталось построить универсальное семейство хеш-функций, тогда мы придем к логическому завершению.

Пусть $U = \mathbb{Z}_p$, где p — простое. Тогда пусть $a, b \in \mathbb{Z}_p$, где $a \neq 0$

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod k$$

Заметим, что тогда $|\mathcal{H}| = p(p-1)$. Осталось проверить определение. Наш план: зафиксируем $x \neq y$ и посчитаем, сколько хеш-функций даст коллизию.

Теорема. Семейство выше универсально.

Доказательство: Для начала разберемся с тем, что может ли быть коллизия при вычислении по модулю p. Пусть она произошла, тогда

$$ax + b \equiv ay + b \bmod p \Longrightarrow ax \equiv ay \bmod p \Longrightarrow x \equiv y \bmod p \Longrightarrow x = y$$

Откуда на первом этапе нет коллизий ни для каких $x \neq y$! А значит все коллизии возникают при взятии по модулю k. Как посчитать тогда число коллизий? Если зафиксировать x, то заметим, что таких y, дающих коллизию будет $\left\lceil \frac{p}{k} \right\rceil - 1$

Тогда число коллизий не будет превосходить $p \cdot \left(\left\lceil \frac{p}{k} \right\rceil - 1\right)$. Собственно оценим вероятность того, что у нас есть коллизии:

$$\forall x \neq y \; P_{h \in \mathcal{H}}(h(x) = h(y)) \leq \frac{p \cdot \left(\left\lceil \frac{p}{k} \right\rceil - 1\right)}{p(p-1)} \leq \frac{\frac{p+k-1}{k} - 1}{p-1} = \frac{1}{k}$$

■

Таким образом, мы построили хеш-таблицу со следующими операциями:

Операция	Время
Вставка ключа	$\mathcal{O}(1)$ в среднем
Поиск по ключу	$\mathcal{O}(1)$ в среднем
Удаление по ключу	$\mathcal{O}(1)$ в среднем

Лекция 12.

FKS-hashing

Def. Хеш-таблица называется статической, если она построена на каком-то множестве ключей, а затем есть только запросы поиска по ключу, то есть множество не изменяется.

Наша цель — построить такую статическую хеш-таблицу, что в ней нет коллизий. Допустим, нам так повезло, что вероятность того, что есть хотя бы одна коллизия меньше $\frac{1}{2}$, тогда мы можем генрировать случайную хеш-функцию из такого семейства и за разумное время мы сможем такую хеш-функцию угадать.

Давайте сразу представим аналогию. Пусть у нас есть честная монетка, которая выпадает решкой с вероятностью $\frac{1}{2}$, тогда как найти матожидание первой решки? Данное распределение называется геометрическим и обозначается $\mathrm{Geom}(p)$, тогда, очевидно, его плотность имеет вид:

$$P(\xi = k) = (1-p)^{k-1}p$$

В нашем случае $p = \frac{1}{2}$, воспользуемся этим:

$$\mathbb{E}\xi = \sum_{k=1}^{\infty} \frac{k}{2^k} = 2$$

То есть в среднем достаточно две итерации перебрать, если верно соотношение на вероятность отсутствия коллизий. Пусть H — число коллизий, тогда

$$H = \sum_{1 \le i < j \le n} I(h(k_i) = h(k_j))$$

Воспользуемся λ-универсальным семейством хеш-функций:

$$\mathbb{E}H = \sum_{1 \le i < j \le n} \mathbb{E}I(h(k_i) = h(k_j)) = \sum_{1 \le i < j \le n} P(h(k_i) = h(k_j)) = \frac{n(n-1)\lambda}{2k} \le \frac{1}{2} \Longrightarrow k \sim n^2$$

То есть при квадратичном росте числа бакетов можно утверждать, что в среднем не больше половины коллизии. Тогда воспользуемся неравенством Маркова:

$$P(H = 0) = 1 - P(H \ge 1) \ge 1 - \frac{\mathbb{E}H}{1} = \frac{1}{2}$$

Вау, круто, мы умеем строить таблицу квадратичного размера без коллизий, но есть нюанс, она квадратичная по памяти, что непростительно много. То есть наш наивный подбор надо улучшать.

Сделаем двухуровневую хеш-таблицу, где внешняя хеш-таблица будто бы таблица с цепочками, а каждый бакет будет являться наивной хеш-таблицей без коллизий, как описано выше. Тогда размер равен сумме длин цепочек. Пусть длина i-й цепочки равна L_i, тогда надо найти среднюю длину сумм квадратов.

$$\mathbb{E}\sum_{i=0}^{k-1} L_i^2 = \mathbb{E}\sum_{i=0}^{k-1}(L_i^2 - L_i) + \mathbb{E}\sum_{i=0}^{k-1} L_i = \mathbb{E}\left(2 \cdot \sum_{i=0}^{k-1} \frac{L_i(L_i-1)}{2}\right) + n =$$

$$= 2\mathbb{E}H + n \le \frac{n(n-1)\lambda}{k} + n = \{k = \lambda n\} = 2n - 1 = \mathcal{O}(n)$$

Тогда

$$P\left(\sum_{i=0}^{k-1} L_i^2 \ge 4n\right) \le \frac{2n-1}{4n} \le \frac{1}{2}$$

Откуда нам хватит в среднем две итерации на перебор хеш-функции первого уровня.

Итоговый алгоритм построения:

1. Выбираю случайно из универсального семейства хеш-функцию первого уровня для хеш-таблицы с размером равным числу элементов в множестве
2. Считаю сумму длин квадратов цепочек. Если она превосходит $4n$, то расстраиваюсь и возвращаюсь к первому шагу. Иначе иду далее.
3. В каждом бакете перебираю хеш-функцию второго уровня из универсального семейства (быть может с другим p), при этом $k_i = L_i^2$

4. Если коллизии в данном бакете есть, то перегенерирую для него хеш-функцию и возвращаюсь к данному шагу. Иначе с этим бакетом покончено.

В среднем построение первого уровня работает за $\mathcal{O}(n)$ и в среднем построение всех бакетов второго уровня работает за $\mathcal{O}(n)$, итого в среднем $\mathcal{O}(n)$ на построение хеш-таблицы без коллизий.

Как устроен запрос?

1. Посчитать хеш-функцию первого уровня, перейти в нужный бакет
2. Посчитать хеш-функцию второго уровня, перейти в нужный элемент
3. Сравнить ключ в нем с запросом. Если ключи совпали, то победа, иначе расстроиться и сказать, что нет такого значения

Это работает все за чистые $\mathcal{O}(1)$, так как каждый шаг потребляет константное время в худшем случае.

Модуль 6. Динамическое программирование

Лекция 12 (часть 2).

Идея ДП

Def. Динамическое программирование — подход к решению задач, где рассматривается сведение подзадачи к меньшей с оптимизированным перебором возможных вариантов.

Пять шагов для решения задач на ДП:

1. Что хранит в себе состояние динамики
2. Какая база
3. Как пересчитывать
4. В каком порядке
5. Где ответ

Кузнечик

Разберем на примере задачи про кузнечика. Есть массив целых чисел $a = [a_1, \ldots, a_n]$. Кузнечик стоит в нулевой клетке и может прыгать а одну или две клетки вперед. Нужно набрать максмальную сумму по пройденным кузнечиком клеткам, чтобы он допрыгал до a_n.

1. $dp[i]$ — ответ для первых i клеток
2. $dp[0] = a_0$, $dp[1] = a_1 + \max(0, a_0)$. Оба этих значения естественны и для них мы гарантированно знаем ответ
3. $dp[i] = a_i + \max(dp[i-1], dp[i-2])$. Формула получается очевидным образом из задачи, ибо прийти в i-ю клетку можно только двумя способами
4. Пересчет идет в порядке роста i, так как мы смотрим только на предыдущие значения
5. Ответ лежит в $dp[n]$

Теперь уже код не представляет из себя сложности написать, все очевидно и решение задачи понятно.

Упражнение. Рассмотрим задачу про черепашку. Есть таблица $N \times M$, черепашка находится в левом верхнем углу, она хочет добраться в правый нижний. Ей важно, чтобы сумма чисел на пути была наибольшей, при этом ходить она может ровно на один вправо или вниз. Распишите все пять шагов динамики по примеру выше, где состояние динамики — координаты черепашки.

НВП

Дана последовательность чисел $a_i, \ldots, a_n$. Подпоследовательностью назовем такой набор $a_{i_1}, \ldots, a_{i_k}$, где $i_1 < i_2 < \ldots < i_k$.

Задача. Найти по $a_1, \ldots, a_n$ наибольшую возрастающую подпоследовательность.

Решение. Рассмотрим $dp[i]$ — длина НВП, заканчивающейся в a_i. Тогда $dp[1] = 1$, что очевидно. Разберемся с пересчетом.

$$dp[i] = \max(1, 1 + \max_{j<i,\ a_j<a_i} dp[j])$$

Внешний максимум с единицей берется из того, что a_i может быть меньше всех элементов среди a_j, где $j < i$. Далее второй максимум считается по всем тем элементам, к которым можно дописать в конец a_i и при этом не сломать возрастание подпоследовательности.

Пересчет опять же в порядке возрастания i, при этом ответ лежит в $\max\limits_i dp[i]$, так как ответ может заканчиваться на любое из a_i.

Теперь научимся решать данную задачу быстрее, чем за $\mathcal{O}(n^2)$. Для этого рассмотрим следующий алгоритм:

1. Отсортируем (a_i, i) по неубывванию первой компоненты, в случае равенства — по убыванию второй.
2. Создадим массив dp, который изначально имеет вид $[0, 0, \ldots, 0]$. Он будет иметь тот же смысл, что и в квадратичном решении. Построим на нем дерево отрезков на максимум.

3. Теперь будем идти по массиву из первого шага и считать ответ:
   - Узнаем максимум на $[0, idx]$, где idx — вторая компонента текущей пары.
   - Ставим вместо $dp[idx]$ полученное на прошлом шаге значение, увеличенное на единицу.

Сложность: $\mathcal{O}(n \log n) + n\mathcal{O}(\log n) = \mathcal{O}(n \log n)$. Почему это верно? К моменту рассмотрения a_i все меньшие a_j уже рассмотрены и для них корректно подсчитано dp, тогда запрос на префиксе позволяет получить верный ответ. Так как нам нужно строгое возрастание, то сортировка по индексам идет справа налево, чтобы одинаковые элементы не вносили вклад.

Упражнение. Как найти похожим образом наибольшую *невозрастающую* подпоследовательность?

Лекция 13.

НОП

Задача. Даны две последовательности чисел $a_1, \ldots, a_n$ и $b_1, \ldots, b_m$. Хотим найти такую последовательность $c_1, \ldots, c_k$, что она является подпоследовательностью обеих и наидлиннейшая при этом.

Решение. Пусть $dp[i][j]$ — ответ, если рассматривать последоватеьности $a_1, \ldots, a_i$ и $b_1, \ldots, b_j$. Тогда база очевидна: $dp[:][0] = 0$ и $dp[0][:] = 0$, так как общая последовательность с пустой может быть только пустой.

Теперь пересчет:

$$dp[i][j] = \max \begin{cases} dp[i-1][j-1] + 1, & \text{если } a_i == b_j \\ \max(dp[i-1][j], dp[i][j-1]), & \text{иначе} \end{cases}$$

Почему это верно? Рассмотрим верхнюю ветку. Если числа совпадают, то оно дополняет ответ для двух префиксов. В другой ветке как раз рассматривается, что в случае неравенства можно будто откусить от одной из последовательностей крайний элемент и ответ на изменится.

Порядок прост, два вложенных `for` по i и по j. Ответ лежит в $dp[n][m]$.

Рюкзак

Приведем линейно-алгебраическую постановку задачи. Даны два вектора $w = (w_0, \ldots, w_{n-1})$ и $c = (c_0, \ldots, c_{n-1})$, оба вектора из $\mathbb{N}^n$. Нужно построить битовый вектор $b \in \{0, 1\}^n$ такой, что

$$\begin{cases} (w, b) \le W \\ (c, b) \to \max \end{cases}$$

Приведем классическую постановку задачи. Есть n предметов, каждый из них имеет вес w_i и стоимость c_i. Надо взять какие-то предметы в рюкзак так, чтобы их суммарный вес не превосходил W, а стоимость максимальна.

Решение за $\mathcal{O}(nW)$). Пусть $dp[i][w]$ — ответ, если разрешено брать только первые i предметов, а ограничение по весу в рюкзаке w. Тогда какая база? $dp[0][:] = 0$, $dp[:][0] = 0$. Переход:

$$dp[i][w] = \min \begin{cases} dp[i-1][w], & i\text{-й предмет не берем} \\ dp[i-1][w-w_i] + c_i, & i\text{-й предмет берем} \end{cases}$$

Пересчет делается двумя вложенными `for` по числу предметов, а внутри по вместимости рюкзака. Итоговое время решения: $\mathcal{O}(nW)$.

Матричное ДП

Однородные линейные рекурренты

Рассмотрим числа Фибоначчи $F_n = F_{n-1}+F_{n-2}$. В этот раз хотим быстро найти n-е число. Рассмотрим соотношение:

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = \ldots \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \cdot \begin{pmatrix} F_{n-k} \\ F_{n-k-1} \end{pmatrix} = \ldots = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

С помощью бинарного возведения в степень возведем матрицу в степень за $\mathcal{O}(2^3 \log n) = \mathcal{O}(\log n)$, далее перемножим и получим ответ.

Упражнение. Нужно быстро получить числа Фибоначчи $F_n, F_{n-1}, \ldots, F_{n-k+1}$. Найдите способ это сделать за $\mathcal{O}(k^3 \log n)$.

Неоднородные линейные рекурренты

Def. *Неоднородной линейной рекуррентой порядка k с полиномиальной неоднородностью порядка m* назовем рекурренту вида $a_n = \lambda_0 a_{n-1} + \ldots + \lambda_{k-1} a_{n-k} + \gamma_0 n^{m-1} + \ldots + \gamma_{m-2} n + \gamma_{m-1}$.

Как их решать? Нужно воспользоваться утверждением ниже.

Теорема (б/д). Последовательность $\{n^0, n, n^2, \ldots\}$ является базисом в пространстве многочленов.

Следствие. Последовательность $\{(n-1)^0, n-1, (n-1)^2, \ldots\}$ является базисом в пространстве многочленов.

Доказательство: Применяем бином Ньютона к каждому моному и получаем разложение по стандартному базису мономов.

Утверждение. $n^k = C_k^0 (n-1)^k + C_k^1 (n-1)^{k-1} + \ldots + C_k^{k-1}(n-1) + C_k^k (n-1)^0$.

Доказательство: Индукция по степени монома.

Рассмотрим рекурренту $a_n = \lambda_0 a_{n-1} + \ldots + \lambda_{k-1} a_{n-k} + \gamma_0 n^{m-1} + \ldots + \gamma_{m-2} n + \gamma_{m-1}$. Распишем ее в

матричном виде, используя утверждение о разложении выше.

$$\begin{pmatrix} a_n \\ a_{n-1} \\ a_{n-2} \\ \vdots \\ a_{n-k+1} \\ n^{m-1} \\ n^{m-2} \\ \vdots \\ n^1 \\ n^0 \end{pmatrix} = \begin{pmatrix} \lambda_0 & \lambda_1 & \lambda_2 & \dots & \lambda_{k-1} & \gamma'_0 & \gamma'_1 & \dots & \gamma'_{m-2} & \gamma'_{m-1} \\ 1 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & C^0_{m-1} & C^1_{m-1} & \dots & C^{m-2}_{m-1} & C^{m-1}_{m-1} \\ 0 & 0 & \dots & 0 & 0 & 0 & C^0_{m-2} & \dots & C^{m-3}_{m-2} & C^{m-2}_{m-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & C^0_1 & C^0_1 \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & C^0_0 \end{pmatrix} \cdot \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ a_{n-3} \\ \vdots \\ a_{n-k} \\ (n-1)^{m-1} \\ (n-1)^{m-2} \\ \vdots \\ (n-1)^1 \\ (n-1)^0 \end{pmatrix}$$

Нужно возвести эту матрицу $(m+k)\times(m+k)$ в степень n, откуда общая сложность: $\mathcal{O}((m+k)^3 \log n)$.

Как эта матрица устроена? В первой строке вектор лямбд, а далее идут γ'_i, которые получаются путем разложения $\gamma_0 n^{m-1} + \ldots + \gamma_{m-2} n + \gamma_{m-1}$ по базису $\{(n-1)^{m-1}, \ldots, (n-1)^1, (n-1)^0\}$. Далее идет единичная матрица порядка $k-1$ и все остальные столбцы нулевые. Далее идет нулевая матрица размеров $m \times k$ и последний блок имеет вид верхнетреугольной матрицы порядка m, заполненная треугольником Паскаля.

Лекция 14.

ДП по подотрезкам

Большинство задач на ДП по подотрезкам относятся к одному из двух видов.

1. Где легко пересчитать ответ на отрезке $[l, r]$, зная ответы на $[l+1, r]$ и $[l, r-1]$. Обычно тут задачи решаются за $\mathcal{O}(N^2)$, так как схема решения выглядит
   - Задать базу для отрезков длины 1 (может быть 2).
   - Получить формулу пересчета ответа на отрезке $[l, r]$, зная ответы для отрезков меньшей длины.
   - Далее просто перебираем отрезки двумя вложенными циклами, наружний по длине подотрезка и внутренний по индексу начала.
   - Выписать ответ.
2. Где задачу можно свести к задаче на двух кусках массива, тогда мы можем рекурсивно сводить задачу к задаче до отрезков длины один, а потом сливать ответы. Обычно тут задачи решаются за $\mathcal{O}(N^3)$, так как схема решения выглядит
   - Задать базу для отрезков длины 1 (может быть 2).

- Получить формулу пересчета ответа на отрезке $[l, r]$, зная ответы для отрезков меньшей длины. Обычно она имеет вид $f(l, r) = \min\limits_{m \in (l,r)} g(f(l, m), f(m + 1, r))$, то есть f — искомая величина на подотрезке, а g — функция «сливания» ответов на подотрезке.
- Далее просто перебираем отрезки двумя вложенными циклами, наружний по длине подотрезка и внутренний по индексу начала. Третий вложенный цикл перебирает по m лучшее разбиение отрезка.
- Выписать ответ.

Подпалиндромы

Есть строка S. Надо найти длину максимальной подпоследовательности палиндрома за $\mathcal{O}(|S|^2)$.

Решение. Пусть $f(l, r)$ — ответ для строки $S[l : r + 1]$, тогда сразу определим базу: $dp[l][l] = 1$ и $dp[l][r] = 0$, если $r < l$. Сразу определимся, где будет лежать ответ: $f(0, |S| - 1) = dp[0][|S| - 1]$.

Теперь научимся пересчитывать ответ. У нас могут быть две ситуации

- Если $S[l] = S[r]$, то тогда нам выгодно оба эти символа докинуть в ответ и $f(l, r) = f(l + 1, r - 1) + 2$. Оптимальнее быть не может, так как мы верим, что $f(l+1, r-1)$ само по себе оптимально посчитано.
- Если $S[l] \neq S[r]$, то тогда нам нет смысла учитывать данную букву, так что $f(l, r) = \max\{f(l + 1, r), f(l, r - 1)\}$.

Ну и все, получили формулу перехода, итоговый алгоритм выглядит примерно так:

```
for (int len = 2; len < S.size(); ++len) {
  for (int l = 0; l < S.size() - len; ++l) {
    int r = l + len;
    if (S[l] == S[r]) {
      dp[l][r] = dp[l + 1][r - 1] + 2;
    } else {
      dp[l][r] = max(dp[l][r - 1], dp[l + 1][r]);
    }
  }
}
std::cout << dp[0][S.size() - 1];
```

Число подпалиндромов

Найти в строке число подпоследовательностей палиндромов.

Решение. Пусть $dp[l][r]$ равно числу подпоследоватльностей палиндромов на отрезке $s[l : r + 1]$. Выпишем базу: $dp[l][l] = 1$ и $dp[l][l + 1] = 2 + I(S[l] == S[l + 1])$, то есть при равенстве это три, а при неравенстве — два. Тогда подумаем, как устроены переходы. Воспользуемся формулой включений-исключений

$$dp[l][r] = dp[l + 1][r] + dp[l][r - 1] - dp[l + 1][r - 1]$$

Казалось бы, все супер, это работает в общем случае, но на самом деле нет, если $S[l] == S[r]$, то у нас еще появятся палиндромы вида $S[l] + P + S[r]$, где P — палиндром их $S[l : r + 1]$. А значит в случае равенства надо прибавить еще $dp[l + 1][r - 1]$ и еще добавить единичку, так как появляется подпоследовательность палиндром $S[l] + S[r]$.

Итоговый ответ лежит в $dp[0][S.size() - 1]$, не забывайте считать по модулю **все** вычисления.

Правильная скобочная подпоследовательность

Есть строка S из разных типов скобок. Надо найти длину максимальной правильной скобочной подпоследовательности за $\mathcal{O}(|S|^3)$.

Решение. Пусть $f(l, r)$ — минимальное число символов, которые надо удалить из $S[l : r + 1]$, чтобы получилась ПСП, тогда сразу определим базу: $dp[l][l] = 1$ и $dp[l][r] = 0$, если $r < l$. Сразу определимся, где будет лежать ответ: $f(0, |S| - 1) = dp[0][|S| - 1]$.

Теперь научимся пересчитывать ответ. Посмотрим, что может произойти с $S[l]$

- Скобка будет удалена, тогда $f(l, r) = f(l + 1, r) + 1$.
- Скобка не будет удалена, тогда найдется парная ей скобка на позиции $m \in [l + 1, r - 1]$. Тогда итоговая ПСП на $S[l : r + 1]$ будет устроена как $(ans[l + 1 : m])ans[m + 1 : r + 1]$, то есть $f(l, r) = f(l + 1, m - 1) + f(m + 1, r)$. Тогда $f(l, r) = \min\limits_{m \in (l,r)} f(l + 1, m - 1) + f(m + 1, r)$.

Так как реализованы могут быть оба сценария, надо выбрать минимум из двух и записать его в $dp[l][r]$.

Расстановка знаков в выражении

Пускай у нас есть массив неотрицательных $a[i]$ длины N. Мы можем между числами из массива ставить знаки сложения и умножения, а также скобочки. Нужно найти максимально возможное значение после расстановки знаков за $\mathcal{O}(N^3)$.

Решение. Давайте подумаем, что если у нас есть отрезок $a[l : r + 1]$, то можно его разбить на два с помощью скобок и затем поставить между скобками знак нужной операции. То есть решение задачи абсолютно аналогично прошлой задаче, только перебираем еще знак операции.

ДП по подмножествам

Утверждение. Все подмножества множества $\{A_0, \ldots, A_{n-1}\}$ можно закодировать числом, меньшим 2^n.

Доказательство: Рассмотрим следующее кодирование. Если элемент A_i в подмножестве, то вместо i-го бита будем ставить единичку, иначе ноль. Тогда кодирование биективное от нуля до $2^n - 1$.

■

Упражнение. Пусть A, B — подмножества $\{C_0, \ldots, C_{n-1}\}$. Докажите, что операции над множествами выразимы следующим образом:

- $A \cup B = A|B$
- $A \cap B = A\&B$
- $\overline{A} = \sim A$

Задача коммивояжера

Задача. Пусть нам дан полный взвешенный граф (веса неотрицательны) на N вершинах. Нужно найти гамильтонов путь минимальной стоимости.

Решение. Рассмотрим динамику $dp[mask][v]$ — минимальный вес обхода вершин из $mask$, при этом v — последняя вершина. Тогда база: $dp[1 << v][v] = 0$, остальное равно бесконечности.

Переход: $dp[mask \mid (1 << u)][u] = \min\left\{dp[mask|(1 << u)][u], \min\limits_{u \notin mask}\{dp[mask][v] + w(v,u)\}\right\}$.

Ответ: $\min\limits_{v \in [0,N-1]} dp[2^N - 1][v]$.

Казалось бы, победа. Но нет, есть проблема. В каком порядке пересчитывать ДП? В порядке вложенности масок? Это трудно. Давайте пересчитывать в порядке увеличения $mask$, тогда все вложенные маски уже были рассмотрены, а значит такой порядок корректен.

Время работы: $\mathcal{O}(2^N N^2)$.

Глава 2

Семестр 2.

Модуль 1. Обходы графов и их производные.

Лекция 1.

В прошлом семестре мы с вами в качеств бонуса научились обходить деревья. На этой же лекци наша цель — научиться обходить графы. Здесь и далее V — множество вершин, а E — множество ребер.

Хранение графа

Есть три способа хранения графа, однако нам инетересны будут только два:

- Список смежности.

  Изначально заводится массив массивов *adjacency_list* размера $|V|$, где в *adjacency_list[i]* хранится массив вершин, с которыми данная смежна.

- Матрица смежности.

  Изначально заводится массив массивов *adjacency_matrix* размера $|V| \times |V|$, где в *adjacency_matrix[i][j]* хранится единичка, если есть ребро из i в j и ноль — иначе.

Операция	Список смежности	Матрица смежности
Создание	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\| + \|E\|)$
Узнать, смежны ли вершины u и v	$\mathcal{O}(\deg u)$	$\mathcal{O}(1)$
Удалить ребро из вершины u	$\mathcal{O}(\deg u)$	$\mathcal{O}(1)$
Получить список смежных вершин	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|)$
Потребляемая память	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|^2)$

Как видите, у матрицы смежности множество преимуществ, однако есть проблема — потребляемая память. Она достигает своего предельного значения, которое может быть в списке смежности только на почти полных графах.

BFS

Def. *BFS* или *breadth first search*, или *обход в ширину* — такой обход, в ходе которого вершины посещаются обходом в порядке увеличения расстояния от стартовой вершины.

Заведем булевский массив *used* размера $|V|$, который будет обозначать, посещали ли мы i-ю вершину.

Algorithm. BFS

1. Заведем очередь вершин, в нее положим стартовую вершину s.
2. Извлечем вершину s, пометим $used[s] = true$. Добавим все вершины, смежные с ней.
3. Пока очередь не пуста, повторяй шаг 2 с одним условием: если извлеченная вершина уже посещена, то не добавляем ее в очередь.

Утверждение. В очереди BFS могут находиться в один момент только вершины, до которых реберное расстояние от стартовой отличается на единицу. Более того, в начале очереди лежат k вершин с расстоянием l от стартовой, а далее лежат вершины с расстоянием $l + 1$ (таких может и не быть).

Доказательство: Докажем индукцией по числу посещенных вершин. База: посещено не более одной вершины.

- Если посещено ноль вершин, то очередь пуста и все верно.
- Посещена только стартовая вершина. Тогда в очереди хранятся вершины, смежные со стартовой — корректно.

Теперь переход. Пусть в какой-то момент времени мы посетили m вершин, посещаем следующую. Известно, что за мгновение до посещения в начале очереди лежат k вершин с расстоянием l от стартовой, а далее лежат вершины с расстоянием $l+1$. Тогда мы раскрываем вершину с расстоянием l от стартовой, а значит в конец очереди будут добавлены еще непосещенные вершины с расстоянием $l + 1$ от стартовой (l до раскрываемой вершины и еще ребро).

■

Утверждение. В ходе BFS из вершины s будет посещена вся компоненты связности с вершиной s.

Следствие. По определению массива *used*, в вершинах компоненты связности будут стоять пометки *true*. Тогда, если не весь массив *used* помечен в *true*, то граф не связен.

Algorithm. Обход всего графа.

1. Запустить BFS из нулевой вершины.
2. Найти первый *false* в массиве *used*. Если такого нет, то завершить алгоритм, иначе запустить BFS от первой *false* вершины.

Утверждение. Время выполнения обхода всего графа BFS-ом составит $\mathcal{O}(|V| + |E|)$.

Доказательство: Каждая вершина будет посещена не более одного раза, откуда в асимптотику входит слагаемое $|V|$. При этом и каждое ребро не будет просмотрено более одного раза, откуда асимптотика $\mathcal{O}(|V| + |E|)$ в силу того, что «просмотр» вершины или ребра равносилен добавлению вершины в очередь, то есть операции сложностью $\mathcal{O}(1)$.

■

Упражнение. Модифицировать алгоритм выше, чтобы явно выделять компоненты связности, потратив на это $\mathcal{O}(1)$ дополнительной памяти.

DFS

Def. *DFS* или *depth first search*, или *обход в глубину* — такой обход, в ходе которого вершины посещаются как только обнаружены.

Заведем булевский массив *used* размера $|V|$, который будет обозначать, посещали ли мы i-ю вершину.

Опишем рекурсивную реализацию DFS, модифицировать его под нерекурсивную реализацию можно будет заменой одного слова в описании BFS.

Algorithm. DFS

1. Пометим *used[s] = true*.
2. Для каждой смежной вершины с данной запустись рекурсивно от нее, если вершина еще не *used*.

Утверждение. Время выполнения обхода всего графа DFS-ом составит $\mathcal{O}(|V| + |E|)$.

Цвета вершин

Def. *Цветом* вершины в ходе DFS назовем одно из трех состояний:

- *Белый* цвет — вершина еще не была посещена.
- *Серый* цвет — вершина была посещена, но рекурсия еще не вышла из нее.
- *Черный* цвет — рекурсия вышла из вершины.

Def. *Временем входа* или $t\_in[v]$ назовем момент времени, когда вершина v была впервые посещена (покрашена в серый).

Def. *Временем выхода* или $t\_out[v]$ назовем момент времени, когда вершина v была покрашена в черный.

Note. Время увеличивается каждый раз, когда мы посещаем какую-то вершину или же перекрашиваем вершину в новый цвет.

Лемма о белых путях

Лемма (о белых путях). Рассмотрим момент, когда вершина v была покрашена в серый. Тогда все вершины, достижимые по белым путям из v покрасятся в черный к моменту выхода из v.

Доказательство: Индукция по убыванию времени входа в вершину. База: вершина с максимальным t_in. Тогда из v нет белых путей, так как иначе вершина, достижимая по нему, имела большее время входа.

Переход. Рассмотрим какой-то белый путь из вершины v, тогда хотим показать, что к моменту $t\_out[v]$ весь путь станет черным. Предположим противное. Тогда рассмотрим среди не черных самую первую на пути u (выше нее по пути все черные). Предок u на пути черный, а значит u не может быть белой (так как все ребра перебрали из предка). Значит u серая. Но чтобы покрасить v в черный, ему надо было покрасить и u в черный (по стеку рекурсии). Противоречие.

Подробнее: $t\_in[v] < t\_in[u] < t\_out[v] < t\_out[u]$. Второе неравенство следует из того, что u еще не обработана.

■

Лекция 2.

Следствие. Алгоритм обхода всего графа аналогичен вариации с BFS, так как для него верно, что посещается вся компонента связности (так как изначально все пути белые).

Следствие. В графе есть цикл, достижимый из s тогда и только тогда, когда DFS нашел ребро в серую вершину.

Доказательство:

$\Longleftarrow$ Стек рекурсии — путь из серых вершин, так как черные там быть не могут (они удаляются из стека), а белые при попадании красятся в черный. При этом, если найдено ребро в серую, то это ребро куда-то в вершину выше по стеку рекурсии, вот и цикл (стек рекурсии хранит его).

$\Longrightarrow$ Пусть C — цикл, достижимый из s. Пусть v — первая посещенная вершина C, тогда весь цикл белый. Откуда к моменту $t\_out[v]$ весь цикл станет черным, в частности будет посещена

вершина u, предшествующая v по циклу, до $t\_out[v]$, а значит v была серой на тот момент, вот и нашли ребро в серую вершину.

■

Note. У нас нет цели найти все циклы, а лишь проверить, есть ли он (и найти какой-то).

Note. Если хранить предка вершины в порядке DFS, то можно отыскать сам цикл.

Топологическая сортировка

Def. Рассмотрим ориентированный граф. Присвоим каждой вершине номер. Перестановка σ называется *топологической сортировкой* графа, если $\forall (u, v) \in E$ выполняется, что $\sigma(u) < \sigma(v)$.

То есть мы хотим сделать так, чтобы ребра шли только из вершин с меньшим номером в вершины с большим.

Утверждение. Топологическая сортировка существует тогда и только тогда, когда граф ацикличен.

Доказательство: Так как порядок «меньше» ацикличен, если граф не ацикличен, то топологической сортировки не существует. В качестве доказательства в другую сторону приведем алгоритм поиска этой перестановки.

Algorithm. Топологическая сортировка.

1. Запустим DFS на всем графе.
2. В момент выхода DFS из вершины будем добавлять ее в конец массива.
3. Развернем полученный массив. Он содержит ответ.

Лемма. Алгоритм выше корректен.

Доказательство: Заметим, что при обходе в глубину время выхода из какой-либо вершины v всегда больше, чем время выхода из всех вершин, достижимых из неё (так как они были посещены либо до вызова DFS(v), либо во время него). Таким образом, искомая топологическая сортировка — это сортировка в порядке убывания времён выхода.

Так как в ходе алгоритма мы как раз добавляем вершины в порядке увеличения времени выхода, остается лишь его развернуть, чтобы получить корректную сортировку.

■

Таким образом, для любого ациклического графа будет найдена топологическая сортировка.

■

Утверждение. Алгоритм занимает по времени $\mathcal{O}(|V| + |E|)$.

Доказательство: Алгоритм сделал запуск DFS на всем графе и добавил все $|V|$ вершин в массив, что также линейно от $|V|$ и $|E|$.

■

Компоненты сильной связности

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

Note. Очевидно, данное отношение является отношением эквивалентности.

Def. *Компоненты сильной связности* — классы эквивалентности по отношению сильной связности.

Def. *Графом конденсации* называют граф, где все компоненты сильной связности сжаты до одной вершины, а ребра между ними получаются как ребра между компонентами.

Утверждение. Граф конденсации ацикличен.

Доказательство: Предположим противное. Рассмотрим какой-то цикл и две вершины-конмпоненты на нем. Очевидно, по циклу они достижимы друг из друга, а значит можно весь цикл сжать в компоненту сильной связности, то есть цикл должен быть сжат в одну вершину.

■

Лемма. Пусть C и C' — две различные вершины в графе конденсации, при этом между ними есть ребро (C, C'), тогда $t\_out[C] > t\_out[C']$.

Доказательство: Рассмотрим два случая:

- $t\_in[C] < t\_in[C']$, то есть до C добрались раньше, тогда DFS посетит сначала какую-то вершину из C, затем как-то по ней походит и по ребру (C, C') попадет C'. Выйдет алгоритм из C' не раньше, чем посетит всю C' по лемме о белых путях. Только затем алгоритм вернется обратно в C и закончит там свою работу.
- $t\_in[C'] < t\_in[C]$, то есть до C' добрались раньше, тогда DFS посетит всю C', но не сможет добраться до C (в силу ацикличности графа конденсации и наличия ребра (C, C')). Тогда к моменту выхода из C' компонента C даже не будет посещена, а значит и выйдет из нее алгоритм когда-нибудь потом.

Algorithm (Косарайю). Поиск компонент сильной связности.

1. Запустить DFS на графе, получить вершины в порядке увеличения времени выхода (почти топологическая сортировка).
2. Построить транспонированный (развернуть все ребра в обратную сторону) граф.
3. Запустить на транспонированном графе DFS в порядке уменьшения времены выхода в исходном графе. Каждая найденная компонента является компонентой сильной связности.

Утверждение. Алгоритм Косарайю корректен.

Доказательство: То есть нам надо доказать, что на шаге 3 каждый запуск посетит одну компненту сильной связности и только ее. Заметим, что в транспонированном графе компоненты все те же, то есть мы изменили лишь связи между компонентами.

Рассмотрим первый вызов DFS на третьем шаге. Так как это вершина с максимальным временем выхода, то нет ребер в ее компоненту сильной связности (по лемме выше). При этом в транспонированном графе получаем, что из рассматриваемой компоненты нет ребер в другие, а значит DFS посетит только саму компоненту. Ну это именно то, что нам требовалось показать, так как с помощью массива *used* мы отделили всю компоненту.

Рассуждая по индукции, получим, что мы генерируем компоненты сильной связности, при этом в порядке топологической сортировки, так как мы все еще идем по убыванию времени выхода.

■

Note. Для построения графа конденсации достаточно пройтись по ребрам исходного графа и строить новые ребра в графе конденсации, если концы из разных компонент.

Утверждение. Построение графа конденсации составляет $\mathcal{O}(|V| + |E|)$.

Доказательство: Алгоритм Косарайю является запуском двух DFS и построением нового графа из исходного, что легко делается итерированием вдоль ребер, так что эта часть работает за $\mathcal{O}(|V|+|E|)$. Наконец, построение самого графа конденсации работает за $\mathcal{O}(|E|)$.

■

Эйлеровость

Def. Граф *эйлеров*, если есть цикл, посещающий все ребра по одному разу (быть может посещая некоторые вершины более одного раза или же не посещая их).

Def. Граф *полуэйлеров*, если есть путь, посещающий все ребра по одному разу (быть может посещая некоторые вершины более одного раза или же не посещая их).

Теорема (б/д). Связный граф эйлеров тогда и только тогда, когда в нем все вершины имеют четную степень.

Теорема (б/д). Связный граф полуэйлеров, если все вершины кроме двух (или кроме нуля) имеют нечетную степень.

Note. Доказательство в курсе дискретного анализа.

Algorithm. Поиск эйлерова цикла.

Допустим, ответ существует (проверяем критерий).Будем делать DFS по ребрам, то есть запускать обычный DFS, только посещать не вершины, а ребра в массиве *used*. Тогда алгоритм имеет такой

вид:

- Переберем все ребра из вершины v, в ходе перебора «удаляем» ребро из графа и рекурсивно вызываем от второго конца ребра.
- На моменте выхода из рекурсии пишем вершину в итоговый цикл.

Утверждение. Алгоритм выше корректен.

Доказательство: В курсе дискретного анализа.

Лекция 3.

Дерево обхода DFS

Def. *Деревом обхода DFS* называют граф, состоящий из вершин, посещаемых в ходе обхода DFS и следующих ребер:

- *Древесное* ребро — ребро, по которому DFS переходит напрямую (переходы в белые вершины из серых);
- *Обратное ребро* — ребро, которое DFS просматривает, но не идет по нему (переходы в серые вершины).

Реберная двусвязность

Def. Две вершины *реберно двусвязны*, если между ними есть два реберно непересекающихся пути.

Note. Несложно заметить, что отношение реберной двусвязности является отношением эквивалентности.

Def. *Компонентой реберной двусвязности* называют класс эквивалентности по отношению выше.

Def. *Мост* — ребро, при удалении которого увеличивается число компонент связности.

Упражнение. Покажите, что в графе конденсации по отношению реберной двусвязности мосты будут ребрами.

Algorithm. Поиск мостов.

Введем функцию $t\_up(v)$, определяемую следующим образом:

$$t\_up(v) = \min \begin{cases} t\_in(v), \\ t\_in(u), \quad \text{определение } u \text{ ниже} \end{cases}$$

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

Утверждение. Древесное ребро (t, v) является мостом если и только если $t\_up(v) \geq t\_in(v)$.

Доказательство: Очевидно, $t\_up(v) \leq t\_in(v)$ по определелению. Тогда осталось рассмотреть случай равенства $t\_up(v) = t\_in(v)$. Это равносильно тому, что не найдется вершины u, в которую по обратному ребру можно прыгнуть из поддерева v (так как иначе это было бы $t\_in(u) < t\_in(v)$), что то же самое, что ребро является мостом.

■

Осталось разобраться с тем, как насчитывать $t\_up(v)$. В ходе DFS будем поддерживать еще родителя вершины в лереве обхода (откуда пришли). Изначально проставляем $t\_up(v) = t\_in(v)$. Если мы идем в *used* вершину, то как раз обновляем значение $t\_up(v)$. Иначе запускаем рекурсивно и после запуска проверяем, оказалось ли ребро мостом. Пробросить обновленное значение выше можно как раз во время выхода из рекурсии.

Note. Тогда можно за линейное время отыскать все мосты.

Вершинная двусвязность

Def. Две вершины *вершинно двусвязны*, если между ними есть два вершинно непересекающихся пути.

Def. *Точка сочленения* — вершина, при удалении которого увеличивается число компонент связности.

Algorithm. Поиск точек сочленения.

Утверждение. Рассмотрим древесное ребро (v, to) и v не является корнем дерева обхода. Тогда $t\_up(to) \geq t\_in(v)$ равносильно тому, что v — точка сочленения.

Доказательство: Выполнение этого неравенства означает, что, пытаясь выпрыгнуть из поддерева to, мы не можем прыгнуть выше v, то есть путь из u в потомка v обязательно пройдет через нее.

В обратную сторону. Так как v не корень, у нее есть родитель p. Надо показать, что какие-нибудь из компонент (наддерево и поддеревья) при удалении v окажутся несвязанными.

Пусть для всех детей v верно, что $t\_up(to) < t\_in(v)$, тогда из каждого ребенка можно прыгнуть в наддерево, откуда v — не точка сочленения. Противоречие.

■

Утверждение. Если же v — корень дерева обхода, то v — точка сочленения тогда и только тогда, когда у нее есть хотя бы два ребенка в дереве обхода.

Доказательство: Это то же самое, что есть два древесных ребра из v. Докажем в обратную сторону. Так как, обойдя одно поддерево, все вершины в нем станут черными, значит есть второе поддерево, где все вершины белые. Но в черные вершины ребер быть не может, а значит нет ребер между этими поддеревьями, а единственный путь обязательно проходит через корень.

В прямую сторону. Предположим противное. То есть у v всего один ребенок в дереве обхода. Но тогда удаление v не приводит к увеличению числа компонент связности.

■

Note. Теперь можно за линейное время отыскать все точки сочленения.

Производные от BFS

0-1 BFS

Постановка задачи: дан граф, веса ребер которого принимают два значения $\{0, 1\}$. Нужно найти кратчайшие расстояния от стартовой вершины до всех остальных.

Algorithm. 0-1 BFS

1. Создадим дек на вершинах. Добавим в него стартовую вершину.
2. Рассмотрим всех соседей и, если ребро до вершины имеет нлевой вес, то добавляем вершину в начало дека, иначе в конец.
3. Далее аналогично BFS, только с замечанием в шагах 1-2.

Корректность алгоритма вытекает из утверждения про BFS (про устройство вершин в очереди).

1-K BFS

Постановка задачи: дан граф, веса ребер которого принимают значения $\{1, \ldots, K\}$. Нужно найти кратчайшие расстояния от стартовой вершины до всех остальных.

Algorithm. 1-k BFS

1. Заведем $(|V|-1)\cdot K$ очередей — массив очередей вершин at_dist, где в $at\_dist[d]$ лежат вершины на расстоянии d или дубликаты вершин, которые мы рассмотрели ранее.
2. База: $at\_dist[0] = \{s\}$. Рассматривая вершину на уровне d и ее соседа на ребре веса w, вершину надо добавить в $at\_dist[d + w]$. Корректность доказывается по индукции.
3. В порядке возрастания d пометить вершины расстояниями до них.

Note. Нам достаточно использовать только $K + 1$ очередей, так как мы можем только на K вперед пройти. Поэтому надо обновлять расстояния до вершин и писать в $at\_dist[(d + w)\%(K + 1)]$.

Таким образом, срелаксировать вершину (улучшить ответ до нее) и добавить в новую очередь можно только K раз, а просматривать рёбра, исходящие из вершины, мы будем только когда обнаружим эту вершину в самый первый раз. Откуда время выполнения занимает $\mathcal{O}(k|V| + |E|)$.

0-K BFS

Постановка задачи: дан граф, веса ребер которого принимают значения $\{0, \dots, K\}$. Нужно найти кратчайшие расстояния от стартовой вершины до всех остальных.

Algorithm. 0-K BFS

Используем 1-K BFS, только если вес ребра 0 до вершины, то добавляем в ту же самую очередь, а иначе как в 1-K BFS.

Модуль 2. Кратчайшие пути. Миностовы. LCA.

Лекция 4.

В данном модуле нам предстоит решить несколько разнообразных задач на графах, имеющих огромное число приложений как непосредственное решение задачи, так и как вспомогательное приложение.

Взвешенные графы

Def. *Взвешенным графом* будем называть тройку $G = (V, E, w)$, где V и E уже привычные нам составляющие, а вот $w : E \to K \subset \mathbb{R}$ — весовая функция.

Def. *Весом пути* $p = v_1 \dots v_k$ будем называть величину $w(p) = \sum\limits_{i=1}^{k-1} w(v_i, v_{i+1})$.

Def. *Кратчайшим путем* от вершины s до вершины t, $dist(s, t)$ будем называть путь $sv_1 \dots v_k t$ такой, что его вес минимален среди всех возможных путей.

Алгоритм Дейкстры

Задача. Дан взвешенный граф такой, что $w : E \to \mathbb{R}^+$ и зафиксирована $s \in V$. Нужно найти $\forall v \in V$ $dist(s, v)$.

Алгоритм

Опишем алгоритм Дейкстры, решающий данную задачу. Инициируем множество $S = \{s\}$ — множество вершин, для которых кратчайшее расстояние вычислено корректно на текущий момент времени, также будет массив $d[]$ текущих оценок на вес кратчайшего пути до вершин.

Очевидно, $d[s] = 0$, для $v \in V \setminus S$ $d[v] = \infty$. Также надо проставить соседям s $d[t] = w(s, t)$. Далее повторяем следующий алгоритм:

1. Рассмотрим все вершины v такие, что $v \notin S$, выберем среди них такую, что $d[v]$ минимально.

2. Добавим v в множество S, присвоим $dist(s, v) = d[v]$ (докажем ниже).
3. Рассмотрим ребра вида (v, t), запишем $d[t] = \min(dist(s, v) + w(v, t), d[t])$, то есть проведем *релаксацию*.
4. Пока $S \neq V$, то повтори шаги выше.

Корректность

Утверждение. Алгоритм Дейкстры корректен.

Доказательство: Нам по сути надо показать, что на момент добавления v в S верно, что $dist(s, v) = d[v]$. Докажем по индукции. База очевидна, рассмотрим переход.

Допустим, что это не так, то есть $d[v] > dist(s, v)$. Пусть u — такая вершина, что после ее релаксации было достигнуто текущее значение $d[v]$. Так как из u была релаксация, значит $u \in S$. По предположению индукции верно, что $d[u] = dist(s, u)$, а значит на текущий момент $d[v] = dist(s, u)+w(u, v)$. Если данный путь не кратчайший, то есть другой. Данный путь может быть устроен двумя образами:

- Путь кроме вершины v состоит из вершин из множества S, но мы выбрали вершину v таким образом, что она заканчивает путь из вершин из S, то есть такой путь мы бы как раз рассмотрели;
- Путь начался в S, далее попал в $V \setminus S$ и далее пришел в v. Пускай ребро (u, t) — ребро такое, что $u \in S$, а $t \notin S$. Если таких ребер несколько, то выберем последнее на пути из них. Так как $u \in S$, получаем, что $d[t] = dist(s, t)$, при этом известно, что $d[t] \geq d[v]$, так как иначе мы бы рассмотрели t, а не v. Рассмотрим суффикс пути от t до v. Известно, что $w(t \dots v) \geq 0$, откуда

$$dist(s, v) = dist(s, t) + w(t \dots v) \geq dist(s, t) = d[t] \geq d[v] > dist(s, v)$$

  Первое равенство следует из принципа *подпуть кратчайшего пути кратчайший*, далее неравенство из-за того, что веса неотрицательны, равенство по тому же принципу, далее нестрогое неравенство по предположению индукции, а строгое неравенство по предположению противного.

Получаем противоречие, из которого следует, что переход корректен.

■

Время работы

Рассмотрим, какие операции происходят в ходе алгоритма:

- Уменьшение оценки d для вершины. Каждое ребро уменьшает не более одного раза, значит таких операций $\mathcal{O}(|E|)$.

- Получение вершины с минимальной оценкой d не из S. Каждая вершина извлекается не более одного раза, а значит таких операций $\mathcal{O}(|V|)$.

Таким образом, приходим к следующим оценкам:

Контейнер для вершин	Время релаксации	Время извлечения	Итого
Массив	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|V\|^2)$
Приоритетная очередь/std::set	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(\|E\| \log \|V\|)$
Фибоначчиева куча	$\mathcal{O}^*(1)$	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(\|E\| + \|V\| \log \|V\|)$

Отдельно отметим, что в реальных кейсах данные недостаточно велики, чтобы продемонстрировать величие фибоначчиевой кучи, поэтому применимы на практике только первые два варианта.

Алгоритм Форда-Беллмана

Задача. Дан взвешенный граф такой, что $w : E \to \mathbb{R}$ и зафиксирована $s \in V$. Нужно найти $\forall v \in V$ $dist(s, v)$.

Note. Допустим для начала, что нет циклов отрицательного веса, с ними разберемся отдельно.

Алгоритм

Опишем алгоритм Форда-Беллмана, решающий данную задачу. Введем динамику $dp[v][k]$, равную минимальному весу пути длины k (из k ребер) из s до v.

База: $dp[s][0] = 0$, для остальных $dp[v][k] = \infty$.

Далее напишем код, осуществляющий переход и насчет динамики (в угоду красоте std включен).

```
vector<int> FordBellman(const Graph& g, int s) {
  int num_vertices = g.NumVertices();
  vector<vector<int>> dp(num_vertices, vector<int>(num_vertices, kInf));
  dp[s][0] = 0;
  for (int k = 1; i < num_vertices - 1; ++k) {
    for (const auto& [u, v] : g.GetEdges()) {
      if (dp[u][k-1] < kInf) {
        dp[v][k] = min(dp[v][k], dp[u][k-1] + w(u, v));
      }
    }
  }
  // dist[:] = dp[:][num_vertices-1]
  // return dist
}
```

Очевидно, что в данном случае у нас используется слишком много памяти, так как пересчет k-го слоя требует только $(k-1)$-го слоя, а значит достаточно хранить только два из них, чтобы снизить память до $\mathcal{O}(|V|)$.

Упражнение. Покажите, что достаточно хранить один слой и все пересчитывать прямо на нем.

Корректность

Утверждение. Алгоритм Форда-Беллмана корректен.

Доказательство: Из формулировки динамики вытекает очевидность того, что она корректна. ■

Время работы

Тут все тоже просто, два вложенных цикла дают как раз асимптотику $\mathcal{O}(|V||E|)$.

Алгоритм Флойда-Уоршелла

Задача. Дан взвешенный граф такой, что $w : E \to \mathbb{R}$. Нужно найти $\forall u, v \in V \ dist(u, v)$.

Note. Допустим для начала, что нет циклов отрицательного веса, с ними разберемся отдельно.

Алгоритм

Опишем алгоритм Флойда-Уоршелла, решающий данную задачу. Введем динамику $dp[u][v][k]$, равную минимальному весу пути из u в v, если путь состоит из вершин с номерами меньшими k.

База: $dp[u][u][0] = 0$, для остальных $dp[u][v][k] = \infty$.

Далее напишем код, осуществляющий переход и насчет динамики (в угоду красоте std включен).

```
vector<vector<int>> FloydWarshall(const Graph& g, int s) {
  int num_vertices = g.NumVertices();
  // three axes with num_vertices in each one, values are kInf by default
  vector<vector<vector<int>>> dp;
  for (u in g.GetVertices()) {
    dp[u][u][0] = 0;
  }
  for (int k = 1; i < num_vertices; ++k) {
    for (u in g.GetVertices()) {
      for (v in g.GetVertices()) {
        dp[u][v][k] = min(dp[u][v][k], dp[u][k][k-1] + dp[k][v][k-1])
      }
    }
  }
```

```
  // dist[:][:] = dp[:][:][num_vertices-1]
  // return dist
}
```

Очевидно, что в данном случае у нас используется слишком много памяти, так как пересчет k-го слоя требует только $(k-1)$-го слоя, а значит достаточно хранить только два из них, чтобы снизить память до $\mathcal{O}(|V|^2)$.

Упражнение. Покажите, что достаточно хранить один слой и все пересчитывать прямо на нем.

Корректность

Утверждение. Алгоритм Флойда-Уоршелла корректен.

Доказательство: Из формулировки динамики вытекает очевидность того, что она корректна. ■

Время работы

Тут все тоже просто, три вложенных цикла дают как раз асимптотику $\mathcal{O}(|V|^3)$.

Отрицательные циклы

Идея работы с ними будет разобрана на примере алгоритма Форда-Беллмана, адаптировать идеи под Флойда-Уоршелла предлагается читателю.

Допустим, из s вершина v достижима по пути, проходящему вдоль отрицательного цикла. Тогда, очевидно, алгоритм Форда-Беллмана сможет бесконечно релаксировать расстояние до v, а значит кратчайшее расстояние не определено...

Давайте адаптируем алгоритм Форда-Беллмана так, чтобы он хранил для каждой вершины еще предка, из которого она релаксировалась (для этого надо в момент пересчета динамики вставить простой if).

Утверждение. На $|V|$-й итерации найдется вершина v, до которой расстояние уменьшилось по сравнению с $(|V|-1)$-й итерацией тогда и только тогда, когда в графе есть цикл отрицательного веса, достижимый из s.

Доказательство: Очевидно, простой кратчайший путь не может быть длиннее $k-1$ ребер, значит, если произошла релаксация, то существует не простой путь, который имеет вес строго меньший, чем оптимальный простой, а значит есть цикл отрицательного веса.

В обратную сторону. Рассмотрим цикл отрицательного веса $C = v_1 \dots, v_k$. Так как длина цикла не превосходит $|V|$, на $|V|$-й итерации гарантируется, что хотя бы одна вершина цикла будет рассмотрена второй раз, при этом она будет рассмотрена по пути вдоль цикла отрицательного веса, а значит произойдет релаксация.

Достижимость из s гарантирует необходима, чтобы алгоритм мог пройти вдоль цикла.

■

Лекция 5.

Остовы

Def. *Остовом* графа $G = (V, E)$ будем называть граф $H = (V, E')$, где $E' \subseteq E$.

Def. *Остовным деревом* графа $G = (V, E)$ будем называть остов, образующий дерево.

Def. *Минимальным остовным деревом* графа $G = (V, E, w)$ будем называть такое остовное дерево $H = (V, E', w)$, что $\sum\limits_{e \in E'} w(e) \to \min$

Лемма о безопасном ребре

Def. $\langle S, T \rangle$ — разрез, если $S \cup T = V, S \cap T = \varnothing$

Def. (u, v) пересекает разрез $\langle S, T \rangle$, если u и v в разных частях разреза.

Def. Пусть $G' = (V, E')$ — подграф некоторого минимального остовного дерева G. Ребро $(u, v) \notin G'$ называется *безопасным*, если при добавлении его в G', $G' \cup \{(u, v)\}$ также является подграфом некоторого минимального остовного дерева графа G.

Лемма. 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, $\langle S, T \rangle$ — разрез G, такой, что ни одно ребро из E' не пересекает разрез, а (u, v) — ребро минимального веса среди всех ребер, пересекающих разрез $\langle S, T \rangle$. 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 путь в T_{min} от вершины u до вершины v. Так как эти вершины принадлежат разным долям разреза, то хотя бы одно ребро пути пересекает разрез, назовем его e'. По условию леммы $w(e) \leqslant w(e')$. Заменим ребро e' в T_{min} на ребро e. Полученное дерево также является минимальным остовным деревом графа G, поскольку все вершины G по-прежнему связаны и вес дерева не увеличился. Следовательно $E' \cup \{e\}$ можно дополнить до минимального остовного дерева в графе G, то есть ребро e — безопасное.

■

Алгоритм Прима

Задача. Найти минимальный остов.

Алгоритм

Опишем алгоритм Прима, решающий данную задачу. Инициируем множество $S = \{s\}$ — множество вершин, на которых уже построен миностов.

1. Рассмотрим разрез $\langle S, T\rangle$. Найдем безопасное для него ребро $e = (u, v)$, где $u \in S$.
2. Добавим e в миностов и v в S.
3. Добавим в множество ребер, пересекающих разрез ребра, выходящие из v и идущие не в S.
4. Пока $S \neq V$, повтори шаги выше.

Корректность

Утверждение. Алгоритм Прима корректен.

Доказательство: Непосредственно по лемме о безопасном ребре. ■

Время работы

Рассмотрим, какие операции происходят в ходе алгоритма:

- Добавление ребер в множество пересекающих разрез $|E|$ раз
- Удаление ребра минимального веса через разрез $|V| - 1$ раз

Добавление можно реализовать через изначальное проставление всем ребрам будто они не пересекают разрез (оценку в ∞), а далее, при пересечении разреза, ставить им реальный вес.

Таким образом, приходим к следующим оценкам:

Контейнер для ребер	Время добавления	Время удаления	Итого
Массив	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|V\|^2)$
Приоритетная очередь/std::set	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(\|E\| \log \|V\|)$
Фибоначчиева куча	$\mathcal{O}^*(1)$	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(\|E\| + \|V\| \log \|V\|)$

Отдельно отметим, что в реальных кейсах данные недостаточно велики, чтобы продемонстрировать величие фибоначчиевой кучи, поэтому применимы на практике только первые два варианта.

СНМ

Хотим структуру данных, которая будет работать с множеством непересекающихся множеств. В частности, она должна 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

Давайте хранить каждое множество как подвешенное дерево, тогда его представителем будет корень дерева. Как тогда исполнять запросы выше?

Допустим у нас будет процедура $FindSet$, которая будет возвращать корень дерева (или представителя множества). Тогда объединение это подвешивание одного корня к другому, а проверка $AreSame$ сводится к проверке представителей на равенство. Тогда нам надо научиться оценивать сложность $FindSet$.

Нам понадобятся две эвристики:

- Ранговая (весовая) эвристика. При объединении двух множеств подвешивать то, чей ранг (вес) меньше к тому, чей ранг (вес) больше. Ранг — глубина дерева, а вес — его размер.
- Сжатие путей. Давайте при запросе $FindSet$ подвешивать все вершины сразу к корню в ходе подъема по пути.

Утверждение. При использовании одной лишь эвристики время на запрос составит $\mathcal{O}(\log N)$.

Упражнение. Докажите для весовой эвристики утверждение выше.

Def. Введем функцию Аккермана на паре целых чисел следующим образом:

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

Нам важно лишь одно соотношение: $A(4,4) = 2^{2^{2^{2^{65536}}}} - 3$.

Def. Введем обратную функцию Аккермана $\alpha(N) = \min\{k \mid A(k,k) \geq N\}$.

Note. Формально для всех реальных случаев $\alpha(N) \leq 4$, но это не значит, что ее можно считать константой!!!

Теорема (Тарьян) (б/д). При использовании обеих эвристик время на запрос составит $\mathcal{O}(\alpha(N))$.

Напишем код, сочетающий обе эвристики (ранговую и сжатие путей):

```
class DSU {
 public:
  DSU(int n) {
    ancestors_ = std::vector<int>(n, 0);
    for (int i = 0; i < n; ++i) {
      ancestors_[i] = i;
    }
```

```
    ranks_ = std::vector<int>(n, 0);
  }

  bool AreSame(int u, int v) {
    return FindSet(u) == FindSet(v);
  }

  void Unite(int u, int v) {
    u = FindSet(u);
    v = FindSet(v);
    if (u != v) {
      if (ranks_[u] < ranks_[v]) {
        std::swap(u, v);
      }
      ancestors_[v] = u;
      if (ranks_[v] == ranks_[u]) {
        ++ranks_[u];
      }
    }
  }

 private:
  int FindSet(int elem) {
    if (elem == ancestors_[elem]) {
      return elem;
    }
    return ancestors_[elem] = FindSet(ancestors_[elem]);
  }

  std::vector<int> ancestors_;
  std::vector<int> ranks_;
};
```

Алгоритм Крускала

Задача. Найти миностов.

Алгоритм

Опишем алгоритм Крускала, решающий данную задачу.

- Отсортируем ребра по весу.
- Итерируясь по ребрам, проверяем, приводит ли его добавление к циклу. Если не приводит, то берем, иначе — пропускаем.

Корректность

Утверждение. Алгоритм Крускала корректен.

Доказательство: Обобщив лемму о безопасном ребре на подграфы, корректность очевидна. ■

Время работы

Тут все тоже просто $\mathcal{O}(|E|\log|E|)$ на сортировку, а далее надо понять, как проверять, создает ли ребро цикл. Создадим СНМ на вершинах, добавление ребра будет равносильно объединению, а вот проверка на наличие цикла равносильна проверке на то, лежат ли вершины в одной компоненте связности (то есть в одном множестве). А значит на этот шаг $\mathcal{O}(|E|\alpha(|V|))$, итого $\mathcal{O}(|E|\log|E|)$.

Лекция 6.

LCA

Пусть дано дерево T, подвешенное за вершину r. Тогда назовем наименьшим общим предком двух вершин u, v $LCA(u, v)$ такую вершину X, что она лежит на путях $u \to r$ и $v \to r$, при этом такая вершина глубже всех подходящих.

Algorithm. Наивное решение.

Решим задачу максимально просто, а именно для каждой вершины за линейное время найдем ее глубину, поднимемся до уровня той, что выше, потом одновременный подъем, пока не добрались до общего предка. Сложность линейная на запрос.

Метод двоичных подъемов

Сделаем предподсчет двумерной матрицы $dp[v][i]$ — номер вершины, в которую мы придём если пройдём из вершины v вверх по подвешенному дереву 2^i шагов, причём если мы пришли в корень, то мы там и останемся. Для этого сначала обойдем дерево в глубину, и для каждой вершины запишем номер её родителя $p[v]$ и глубину вершины в подвешенном дереве $d[v]$. Если v — корень, то $p[v] = v$. Тогда для функции 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 в силу определения $i \leq \log_2 n$, так как иначе мы остаемся в корне. Таким образом, матрица $dp[][]$ занимает $O(N \log N)$ памяти, ее построение занимает $O(N \log N)$ времени.

Теперь рассмотрим, как будет происходить ответ на запрос. Пусть $c = LCA(v, u)$, 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. Как это быстро сделать? Разложим разность глубин по степеням двойки и заметим, что в силу определения dp, проходя последовательно вверх на эти степени, мы поднимемся на тот же уровень, что и вершина u. Этот шаг занимает $\mathcal{O}(\log N)$ времени.

Теперь будем считать, что $d[u] = d[v], u \neq v$. Заведем счетчик $k = \log_2 N$ и будем уменьшать его, поднимаясь обеими вершинами, пока они не сравняются. Заметим, что данный метод уравняет высоты вершин (то есть найдет самого глубокого предка) за $\mathcal{O}(\log N)$ времени.

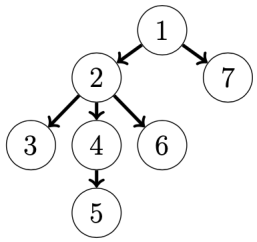
Таким образом, данный метод имеет предподсчет за $\mathcal{O}(N \log N)$ и ответ на запрос за $\mathcal{O}(\log N)$ времени. При этом памяти тратится $\mathcal{O}(N \log N)$.

Сведение LCA к RMQ

Заметим, что если w — LCA вершин u и v, то DFS из корня посетит сначала w, потом только зайдет в вершину u, затем *снова* посетит вершину w, аналогично с v и только потом выйдет из вершины w. Поэтому нам достаточно знать две вещи для каждой вершины: времена посещения и высоты. Для этого просто запустим DFS, который насчитает нам два массива:

- Массив $Order$, где $Order[i]$ — номер вершины, который был посещен в момент времени i
- Массив высот h, где $h[i]$ — высота вершины $Order[i]$
- Массив $First$, где $First[i]$ — момент времени, когда вершина i была посещена впервые

Тогда высоту $LCA(u, v)$ можно найти как минимум на подотрезке $[First[u], First[v]]$ в массиве h. Чтобы узнать саму вершину нам надо поддержать еще позицию минимума (индекс), тогда мы узнаем время, когда мы заходили в LCA, а значит в массиве $Order$ по этому индексу лежит ответ.



Эйлеров обход 1-го типа (обычный)	(1,2) (2,3) (3,2) (2,4) (4,5) (5,4) (4,2) (2,6) (6,2) (2,1) (1,7) (7,1)
Эйлеров обход второго типа (±1)	Обход: 1 2 3 2 4 5 4 2 6 2 1 7 1 Высоты: 0 1 2 1 2 3 2 1 2 1 0 1 0
Обход третьего типа	Обход: 1 2 3 4 5 6 7

- Массив $H = [0, 1, 2, 1, 2, 3, 2, 1, 2, 1, 0, 1, 0]$
- Массив $First = [0, 1, 2, 4, 5, 8, 11]$
- Массив $Order = [1, 2, 3, 2, 4, 5, 4, 2, 6, 2, 1, 7, 1]$

Сведение RMQ к LCA

То есть умеем решать задачу LCA, давайте научимся решать задачу RMQ. Построим на массиве декартово дерево по неявному ключу, где приоритетом будет значение элемента. Вспоминая, что по приоритетам декартово дерево это куча, минимум на подотрезке $[l, r]$ как раз соответствует LCA двух вершин l и r.

Таким образом, научились сводить задачи RMQ и LCA друг к другу и решать LCA за $\mathcal{O}(N \log N)$ на предподсчет и константу на ответ. Долго, не находите?

Алгоритм Фараха-Колтона-Бендера

Задача. Пусть в массиве разница между соседними элементами равна ± 1. Решите задачу RMQ на нем с линейным предподсчетом и константным ответом на запрос.

Разделим массив на блоки длины k, в каждом за линию найдем минимум (и его позицию). На массиве минимумов построим классическую разреженную таблицу за $\mathcal{O}(k \log k)$. Тогда ответ на подотрезке сводится к

- Ответу на префиксе в виде суффикса блока
- Ответу на суффиксе в виде префикса блока
- Ответу на подотрезке из блоков

На третье научились отвечать за $\mathcal{O}(k \log k)$ предподсчета и константное время, осталось разобраться с первыми двумя.

Назовем блок нормализованным, если у него первый элемент равен нулю. Давайте каждый блок нормализуем и запомним значение первого элемента (то, что вычли для нормализации).

Для каждого возможного нормализованного блока найдем минимумы (позиции) на всех префиксах и суффиксах за линейное время и запомним их. Пусть всего нормализованных блоков длины k всего $B(k)$, тогда нам нужно для каждого из $B(k)$ блоков запомнить k минимумов на суффиксах и префиксах. Но этого мало, вдруг запрос внутри одного блока? Значит надо на всех подотрезках помнить или $k^2 \cdot B(k)$ памяти и времени

Тогда ответ на запрос займет $\mathcal{O}(1)$, так как ответ на подотрезок первого и второго типа получается как просмотр соответствующей позиции в соответствующем блоке. Итого научились за $\mathcal{O}(1)$ отвечать. А что по предподсчету?

Утверждение. $B(k) = 2^{k-1}$.

Доказательство: Так как разница между соседними элементами ± 1, пусть на месте i стоит единичка, если $a[i] - a[i-1] = 1$ и ноль иначе. Тогда получили биекцию между нормализованными блоками и двоичными числами длины $k - 1$.

■

Тогда итоговый предподсчет (пусть k — длина блока, тогда их число составит N/k):

$$\mathcal{O}\left(\frac{N}{k}\log\frac{N}{k} + k^2 \cdot 2^{k-1}\right) = \mathcal{O}\left(\frac{N}{k}\log N - \frac{N}{k}\log k + k^2 \cdot 2^{k-1}\right) =$$
$$= \left\{k = \frac{1}{2}\log_2 N\right\} = \mathcal{O}\left(\frac{2N}{\log_2 N}\log N + \frac{1}{4}\log^2 N 2^{\frac{1}{2}\log_2 N}\right) = \mathcal{O}\left(N + \sqrt{N}\log^2 N\right) = \mathcal{O}(N)$$

Продвинутое RMQ

Собственно давайте решим RMQ за линейный предподсчет и константным временем на ответ.

Предподсчет:

1. Сведем RMQ к задаче LCA за линию путем построения декартача
2. Эйлеровыми обходами сведем задачу LCA к RMQ на массиве высот. Заметим, что в нем как раз соседние элементы отличаются на ± 1
3. Применим алгоритм Фараха-Колтона-Бендера к массиву высот

Ответ на запрос:

1. Находим в массиве высот соответствующие вершины
2. Находим позицию минимума через Фараха-Колтона-Бендера
3. Находим вершинку на этой позиции
4. Берем значение в этой вершинке

Модуль 3. Паросочетания. Потоки.

Лекция 7.

Паросочетания

Def. *Паросочетанием* в неориентированном графе $G = (V, E)$ называют множество ребер $M \subseteq E$ такое, что не найдется двух ребер из M с общей вершиной.

Def. Паросочетание M называется *максимальным*, если не существует паросочетания M' такого, что $|M| < |M'|$.

Def. Вершина называется *насыщенной паросочетанием* M, если она является концом какого-то ребра из M.

Note. Определения выше справедливы для произвольных графов, однако далее речь будет идти только о *двудольных* графах.

Def. *Увеличивающая цепь* относительно паросочетания M — путь $p = (v_1, \dots, v_{2k})$ такой, что $(v_1, v_2) \notin M$, $(v_2, v_3) \in M$, $(v_3, v_2) \notin M$, $\dots$, $(v_{2k-1}, v_{2k}) \notin M$, при этом вершины v_1 и v_{2k} не насыщены M.

Теорема (Бёрдж). Паросочетание M максимально тогда и только тогда, когда относительно M нет увеличивающих цепей.

Доказательство:

$\Longrightarrow$ Докажем методом от противного. Рассмотрим паросочетание M и увеличивающую цепь относительно него. Проведем *чередование* вдоль нее, то есть все ребра из паросочетания на этом пути удалим из него, а ребра из пути, отсутствовавшие в M, добавим. Полученное множество все еще будет паросочетанием, так как крайние вершины пути не были насыщенны M, а остальные вершины как были насыщенными, так ими и остались. Таким образом, построили M' такое, что $|M'| > |M|$.

$\Longleftarrow$ Пусть относительно M нет увеличивающей цепи, а M' — максимальное паросочетание такое, что $|M'| > |M|$. Рассмотрим $Q = M \triangle M'$ — симметрическую разность двух паросочетаний. В Q степень каждой вершины не превосходит двух, так как она могла быть насыщенна не более чем каждым из паросочетаний.

Лемма. Если в графе степень каждой вершины не превосходит двух, то его ребра разбиваются на непересекающиеся пути и циклы.

Доказательство: Изолированные вершины никак не влияют на множество ребер, поэтому удалим из графа, это не изменит структуру множества ребер.

Пусть нашлась вершина v_1 степени один, тогда рассмотрим смежное ей ребро $e = (v_1, v_2)$. Так как степень v_2 не превосходит двух, после удаления e степень вершины равна либо нулю, либо единице. В первом случае получаем, что удалено ребро, не пересекающееся по вершинам больше ни с чем. Иначе степень v_2 равна единице, тогда применим к ней ту же операцию отрезания ребра. По индукции за конечное число шагов придем к вершине v_k, степень которой станет нулевой после удаления ребра. Тогда построили путь $p = (v_1, \dots, v_k)$ такой, что он не пересекается с другими ребрами по вершинам, а значит можем удалить из графа его вершины.

После конечного числа итераций алгоритма выше получим, что в графе могли остаться только степени вершины два. Рассмотрим какую-нибудь из них и запустим схожую процедуру. Тогда рано или поздно мы придем в ту же вершину, причем не могли пойти никуда иначе, так как степени вершин будут равны единице на каждой итерации. За конечное число шагов вернемся в ту же вершину, иначе мы бы пришли в вершину, изначальная степень которой равна единице, чего быть не может. Значит нашли изолированный цикл, который тоже можно удалить.

Данный алгоритм доказывает требуемое.

■

Значит Q является объединением непересекающихся циклов и путей.

1. Рассмотрим циклы. Пусть есть цикл нечетной длины $C = (v_1, \ldots, v_{2k}, v_1)$. В данном цикле не могут идти два ребра подряд из одного паросочетания, а значит есть вершина степени два, у которой оба ребра из одного и того же паросочетания. Значит есть только циклы четной длины, причем в них поровну ребер из M и M'.
2. Рассмотрим пути. Пусть есть путь нечетной длины $p = (v_1, \ldots, v_{2k})$. Рассмотрим два случая:
   - $(v_1, v_2) \in M$, тогда относительно M' есть увелицивающая цепь, что по доказанной необходимости влечет немаксимальность M'.
   - $(v_1, v_2) \in M'$, тогда относительно M есть увелицивающая цепь, но M определялось как паросочетание без них.

   Откуда все пути четной длины и в них поровну ребер из M и M'.

Получаем, что $|Q \cap M| = |Q \cap M'|$.

Очевидно, что $M = (M \cap Q) \sqcup (M \cap Q^C)$, при этом нетрудно показать, что $M \cap Q^C = M \cap M'$, то есть $M = (M \cap Q) \sqcup (M \cap M')$. Аналогично $M' = (M' \cap Q) \sqcup (M \cap M')$. Откуда

$$|M| = |M \cap Q| + |M \cap M'|$$
$$|M'| = |M' \cap Q| + |M \cap M'|$$
$$|M \cap Q| = |M' \cap Q|$$

Получаем, что $|M'| = |M|$.

■

Алгоритм Куна

Algorithm. Поиск максимального паросочетания в двудольном графе.

Утверждение. Если из вершины v не была найдена увеличивающая цепь, то и далее из нее не найдется увеличивающая цепь.

Доказательство: Если оставить только просмотренные к моменту просмотра вершины v вершины левой доли, то на тот момент найденное паросочетание максимально по предположению индукции. Обозначим на этот момент покрытые вершины левой доли за S.

Пусть G_A — граф, который остается, если оставить в левой доли только $A \subseteq L$. В G_s есть паросочетание размера $|S|$ (максимального размера), при этом в $G_{S \cup \{v\}}$ нет паросочетания размера $|S| + 1$, так как из v не найдена увеличивающая цепь по предположению.

Заметим, что вершины, когда-то насыщенные паросочетанием, не могут перестать быть насыщенными, так как чередование вдоль них не меняет насыщенности. Пусть далее алгоритм в какой-то момент времени пытается из v найти поиск увеличивающей цепи. Если получилось, то покрыто S и еще v, значит есть в $G_{S\cup\{v\}}$ паросочетание размера $|S|+1$, что неверно.

■

Утверждение выше доказывает, что достаточно только один раз из каждой вершины попытаться найти увеличивающую цепь. Таким образом, алгоритм крайне прост.

Попытаться для текущей вершины левой доли найти увеличивающую цепь. Если получилось, то выполняем чередование вдоль нее, иначе пропускаем.

Поиск увеличивающей цепи осуществляется с помощью специального обхода в глубину. Изначально обход в глубину стоит в текущей ненасыщенной вершине v левой доли. Просматриваем все рёбра из этой вершины, пусть текущее ребро (v, to). Если вершина to ещё не насыщена паросочетанием, то, значит, мы смогли найти увеличивающую цепь: она состоит из единственного ребра (v, to); в таком случае просто включаем это ребро в паросочетание и прекращаем поиск увеличивающей цепи из вершины v. Иначе, если to уже насыщена каким-то ребром (to, p), то попытаемся пройти вдоль этого ребра: тем самым мы попробуем найти увеличивающую цепь, проходящую через рёбра (v, to), (to, p). Для этого просто перейдём в нашем обходе в вершину p — теперь мы уже пробуем найти увеличивающую цепь из этой вершины.

Таким образом, алгоритм Куна это n запусков DFS или его сложность: $\mathcal{O}(|V|(|V|+|E|))$.

Вершинное покрытие и независимое множество

Def. Множество $I \subseteq V$ неориентированного графа называется *независимым множеством*, если $\forall v \in V\ \forall u \in V\ (u, v) \notin E$.

Def. Множество $C \subseteq V$ неориентированного графа называется *вершинным покрытием*, если $\forall e = (u, v) \in E$ верно, что $u \in C$ или $v \in C$.

Note. В произвольном графе задачи поиска минимального вершинного покрытия или максимального независимого множества являются NP-полными.

Note. Далее независимое множество будет обозначаться за `IS`, а вершинное покрытие — `VC`.

Упражнение. Докажите, что дополнение к `VC` является `IS`. Выведите из этого, что дополнение к минимальному `VC` является максимальным `IS`.

Algorithm. Поиск минимального `VC` и максимального `IS` в двудольном графе.

Пусть нам дано максимальное паросочетание `MM`, тогда найдем искомые величины.

0. Ориентируем ребра в G так, что из L в R будут вести ребра не из `MM` и наоборот.

1. Вызовем DFS от каждой ненасыщенной вершины левой доли. Посещенные вершины левой доли обозначим за L^+, а непосещенные за L^-, аналогично введем R^+ и R^-.

2. Заметим, что не может быть ребер из R^+ в L^- и из L^+ в R^-.

3. Докажем, что нет ребер из R^- в L^+. Предположим противное, то есть существует $e = (u, v)$ такое, что $u \in R^-$ и $v \in L^+$. Значит $e \in M$, откуда v насыщенна и DFS из нее запустился рекурсивно, а не сразу (v не является корнем своего дерева DFS). Поэтому обход как-то пришел в v, очевидно он пришел из u, так как других входящих ребер (по тому, как ребра ориентировали) в v нет. Но тогда u была посещена и должна быть в R^+.

   Таким образом, между L^+ и R^- нет ребер или $L^+ \sqcup R^-$ — `IS`, откуда $L^- \sqcup R^+$ — `VC`. Докажем, что данные $L^- \sqcup R^+$ — минимальное `VC`.

   (a) Очевидно, что для произвольного вершинного покрытия его размер не меьше размера максимального паросочетания, так как надо покрыть хотя бы его ребра по одной вершины на каждое.

   (b) Покажем, что $|L^- \sqcup R^+| \leq |MM|$.

   - Очевидно, в L^- только насыщенные вершины (из ненасыщенной запускался DFS, то есть она сразу в L^+).
   - В R^+ только насыщенные вершины.
     Допустим это не так, тогда пусть v_1 — ненасыщенная вершина из R^+. Мы в нее как-то пришли, так как она посещена, причем посетили из вершины u_1 левой доли, тогда перед нами ребро не из паросочетания. Если вершина u_1 насыщена паросочетанием, то по ребру из паросочетания пришли в u_1 из $v_2 \in R^+$. Если вспомнить о дереве DFS, то мы по сути по нему поднимаемся до корня r, но он ненасыщенный и из левой доли. Но тогда найденный путь является увеличивающей цепью, откуда MM не максимально.
   - Так как нет ребер из R^+ в L^-, нет такого ребра из паросочетания, что его концы лежат в L^- и в R^+ одновременно.
   - Тогда каждое ребро паросочетания «забирает» не более одной вершины из $L^- \sqcup R^+$, откуда получаем, что $|L^- \sqcup R^+| \leq |MM|$.

   Получаем, что произвольное `VC` не меньше $|MM|$, а алгоритм построил `VC`, размер которого не больше $|MM|$. Отсюда следует минимальность построенного `VC`.

Лекция 8.

Потоки в сетях

Def. *Транспортной сетью* назовем кортеж $\mathcal{N} = (G, c, s, t)$, где $G = (V, E)$ — ориентированный граф, в котором нет такой пары вершин u, v, что $(v, u) \in E$ и $(u, v) \in E$; $c : E \to \mathbb{R}^+$ — *вместимость* или *capacity* и $s, t \in V$ — исток и сток соответственно.

Note. Для удобства, если $(u, v) \notin E$ полагают, что $c(u, v) = 0$.

Def. *Поток* в транспортной сети — функция $f : V \times V \to \mathbb{R}^+$ такая, что:

1. $\forall u, v \in V \ 0 \leq f(u, v) \leq c(u, v)$ — свойство ограниченности потока;
2. $\forall u \in V \setminus \{s, t\} \ \sum\limits_{v \in V} f(u, v) = \sum\limits_{v \in V} f(v, u)$ — свойство сохранения потока (сколько втекло, столько и вытекло).

Note. Если нет ребра, то через него нет потока, так как пропускная способность нулевая.

Def. Пусть в сети $\mathcal{N}$ зафиксирован поток f, тогда *величиной потока* называют

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Def. Задача о *максимальном потоке* — задача поиска в сети такого потока f, что $|f|$ максимален.

Иногда в задачах возникает такая пара вершин u, v, что $(v, u) \in E$ и $(u, v) \in E$ (такие ребра называются антипараллельными). Тогда вместо ребра (u, v) вводят фиктивную вершину w и ребра (u, w) и (w, v), при этом $c(u, w) = c(w, v) = c(u, v)$.

Упражнение. Покажите, что любому потоку в сети до модификации выше соответствует поток той же величины, проходящий по тем же ребрам (кроме новых и удаленных). Покажите это же в обратную сторону.

Следствие. Теперь задачу о максимальном потоке можно решать на произвольном орграфе в качестве основы сети.

Иногда в задачах возникает необходимость ввести несколько истоков $s_1, \ldots, s_k$ и несколько стоков $t_1, \ldots, t_l$. Тогда введем фиктивный супериcток s и ребра (s, s_i) такие, что $c(s, s_i) = \infty$ и фиктивный суперсток t и ребра (t_i, t) такие, что $c(t_i, t) = \infty$.

Данная модификация сети выполняет сведение задачи с множеством стоков и истоков к классической сети.

Остаточная сеть

Интуитивно, остаточная сеть — сеть, показывающая, сколько еще потока можно пропустить вдоль ребер в сети.

Def. *Остаточная сеть* для сети $\mathcal{N} = (G = (V, E), c, s, t)$ и потока f в $\mathcal{N}$ — сеть $\mathcal{N}_f = (G' = (V, E_f), c_f, s, t)$ такая, что

$$\forall u \in V, \ \forall v \in V \ c_f(u, v) = \begin{cases} c(u, v) - f(u, v), & (u, v) \in E \\ f(v, u), & (v, u) \in E \\ 0, & \text{иначе} \end{cases}$$

где $E_f = \{(u,v) \mid u,v \in V,\ c_f(u,v) \geq 0\}$.

Поясним ветки определения c_f: пусть через ребро (u,v) пропущен положительный поток такой, что $c(u,v) - f(u,v) = q > 0$. Тогда вместо одного ребра появятся два ребра: (u,v) с $c_f(u,v) = q$ и (v,u) с $c(v,u) = f(u,v)$. При этом, если $c(u,v) = f(u,v)$, то будет только одно ребро (v,u) такое, что $c_f(v,u) = f(u,v)$.

Обратные ребра как раз позволяют уменьшить поток через ребро, если его надо будет перераспределить. Поэтому в определении сети не допускают антипараллельных ребер, чтобы не было путаницы при возникновении обратных ребер.

Def. *Дополняющим потоком* для сети $\mathcal{N}$ и потока f определим поток f' в остаточной сети $\mathcal{N}_f$.

Def. Сложение потоков $f + f'$ определяется следующим образом:

$$\forall u \in V,\ \forall v \in V\ (f+f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u), & (u,v) \in E \\ 0, & \text{иначе} \end{cases}$$

Утверждение. Сложение потока f с дополняющим потоком f' является потоком, при этом $|f+f'| = |f| + |f'|$.

Доказательство:

1. Докажем, что $f + f'$ — это поток. Для этого надо проверить два свойства.

   - Ограничение пропускной способности. Сначала неотрицательность.

   $$(f+f')(u,v) = f(u,v) + f'(u,v) - f'(v,u) \geq f(u,v) + f'(u,v) - c_f(v,u) = \\ = f(u,v) + f'(u,v) - f(u,v) = f'(u,v) \geq 0$$

   Теперь ограничение пропускной способности.

   $$(f+f')(u,v) = f(u,v) + f'(u,v) - f'(v,u) \leq f(u,v) + f'(u,v) \leq \\ \leq f(u,v) + c_f(u,v) = f(u,v) + c(u,v) - f(u,v) = c(u,v)$$

   - Теперь надо проверить свойство сохранения потока.

   $$\forall u \in V \setminus \{s,t\}\ \sum_{v \in V}(f+f')(u,v) = \sum_{v \in V}((f(u,v) + f'(u,v) - f'(v,u)) = \\ = \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) - \sum_{v \in V} f'(v,u) = \{\text{для } f \text{ и } f' \text{ верно сохранение потока}\} = \\ = \sum_{v \in V} f(v,u) + \sum_{v \in V} f'(v,u) - \sum_{v \in V} f'(u,v) = \sum_{v \in V}((f(v,u) + f'(v,u) - f'(u,v)) = \sum_{v \in V}(f+f')(v,u)$$

   Проверили оба свойства, значит $f + f'$ — поток.

2. Докажем, что $|f + f'| = |f| + |f'|$. В $\mathcal{N}$ антипараллельные ребра запрещены, тогда пусть $V_1 = \{v \mid (s, v) \in E\}$ и $V_2 = \{v \mid (v, s) \in E\}$, при этом $V_1 \cap V_2 = \varnothing$ и $V_1 \cup V_2 \subseteq V$. Тогда, с учетом того, что сумма потоков равна нулю не через ребра:

$$|f + f'| = \sum_{v \in V}(f + f')(s, v) - \sum_{v \in V}(f + f')(v, s) = \sum_{v \in V_1}(f + f')(s, v) - \sum_{v \in V_2}(f + f')(v, s) =$$
$$= \sum_{v \in V_1}((f(s, v) + f'(s, v) - f'(s, u)) - \sum_{v \in V_2}((f(v, s) + f'(v, s) - f'(v, s)) =$$
$$= \sum_{v \in V_1} f(s, v) - \sum_{v \in V_2} f(v, s) + \sum_{v \in V_1 \sqcup V_2} f'(s, v) - \sum_{v \in V_1 \sqcup V_2} f'(v, s) =$$

$= \{$во всех суммах распространяем суммирование на V, так как остальное равно нулю$\} =$

$$= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{v \in V} f'(s, v) - \sum_{v \in V} f'(v, s) = |f| + |f'|$$

■

Note. Из утверждения выше следует, что если удалось найти поток в остаточной сети, то можно дополнить поток до большей величины. Далее мы применим данное рассуждение, а пока еще немного теории.

Разрезы в сетях

Def. *Разрезом* в сети $\mathcal{N} = (G = (V, E), c, s, t)$ назовем пару (S, T) такую, что:

1. $S, T \subset V$
2. $s \in S$, $t \in T$
3. $S \cap T = \varnothing$ и $S \cup T = V$

Def. *Потоком через разрез* (S, T) назовем

$$f(S, T) = \sum_{u \in S}\sum_{v \in T} f(u, v) - \sum_{u \in S}\sum_{v \in T} f(v, u)$$

Def. *Вместимостью* или *пропускной способностью* разреза назовем

$$c(S, T) = \sum_{u \in S}\sum_{v \in T} c(u, v)$$

Def. *Минимальным разрезом* назовем разрез с минимальной пропускной способностью.

Утверждение. Пусть в сети $\mathcal{N}$ задан поток f, тогда $\forall (S, T)$ — разрез $f(S, T) = |f|$.

Доказательство: По определению потока $\forall u \in V \setminus \{s, t\}$

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0$$

Тогда, используя определение величины потока и ~~переливание из пустого в порожнее~~ коммутативность в конечных рядах

$$|f| = \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) = \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) + \sum_{u\in S\setminus s}\left(\sum_{v\in V} f(u,v) - \sum_{v\in V} f(v,u)\right) =$$
$$= \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) + \sum_{u\in S\setminus s}\sum_{v\in V} f(u,v) - \sum_{u\in S\setminus s}\sum_{v\in V} f(v,u) =$$
$$= \sum_{v\in V}\left(f(s,v) + \sum_{u\in S\setminus s} f(u,v)\right) - \sum_{v\in V}\left(f(v,s) + \sum_{u\in S\setminus s} f(v,u)\right) =$$
$$= \sum_{v\in V}\sum_{u\in S} f(u,v) - \sum_{v\in V}\sum_{u\in S} f(v,u) = \{S \sqcup T = V\} =$$
$$= \sum_{v\in S}\sum_{u\in S} f(u,v) + \sum_{v\in T}\sum_{u\in S} f(u,v) - \sum_{v\in S}\sum_{u\in S} f(u,v) + \sum_{v\in T}\sum_{u\in S} f(v,u) =$$
$$= \left(\sum_{v\in S}\sum_{u\in S} f(u,v) - \sum_{v\in S}\sum_{u\in S} f(u,v)\right) + \sum_{v\in T}\sum_{u\in S} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u) = 0 + f(S,T) = f(S,T)$$

■

Следствие. Величина любого потока ограничена не превосходит пропускной способности произвольного разреза.

Доказательство:

$$|f| = f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \le \sum_{u\in S}\sum_{v\in T} f(u,v) \le \sum_{u\in S}\sum_{v\in T} c(u,v) = c(S,T)$$

■

Теорема Форда-Фалкерсона

Теорема. Пусть задана сеть $\mathcal{N}$ и поток f в ней, тогда следующие утверждения эквивалентны.

1. Поток f в сети максимален.
2. В $\mathcal{N}_f$ нет пути из истока в сток.
3. Существует разрез (S,T) такой, что $c(S,T) = |f|$

Доказательство:

$1 \Longrightarrow 2$. Предпологаем противное, тогда в $\mathcal{N}_f$ есть путь из истока в сток, пустим вдоль него поток f', $|f'| > 0$. Тогда, как доказано ранее, $f + f'$ — поток и $|f + f'| = |f| + |f'| > |f|$. Противоречие с максимальностью f.

$3 \Longrightarrow 1$. Доказано, что для любого потока и разреза $|f| \leq c(S,T)$, но построен поток такой, что $|f| = c(S,T)$, откуда следует максимальность f.

$2 \Longrightarrow 1$. Пусть в $\mathcal{N}_f$ нет пути из s в t, тогда определим разрез (S,T) как:

$$S = \{v \mid \text{есть путь в } \mathcal{N}_f \text{ из } s \text{ в } v\}$$
$$T = V \setminus S$$

Нетрудно проверить по определению, что это действительно разрез. Рассмотрим пару вершин $u \in S$ и $v \in T$.

- Если $(u,v) \in E$, то $f(u,v) = c(u,v)$, так как иначе $c_f(u,v) > 0$ и есть в $\mathcal{N}_f$ ребро (u,v), а значит t достижима из s, что неверно.
- Если $(v,u) \in E$, то $f(v,u) = 0$, так как иначе $c_f(u,v) = f(v,u) > 0$ и есть в $\mathcal{N}_f$ ребро (u,v), а значит t достижима из s, что неверно.

Откуда получаем, что

$$f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) = \sum_{u\in S}\sum_{v\in T} c(u,v) - \sum_{u\in S}\sum_{v\in T} 0 = \sum_{u\in S}\sum_{v\in T} c(u,v) = c(S,T)$$

■

Лекция 9.

Алгоритм Форда-Фалкерсона

Алгоритм Форда-Фалкерсона является непосредстрвенным применением теоремы Форда-Фалкерсона.

1. Построим сеть $\mathcal{N}$, определим поток $f = 0$. Строим $\mathcal{N}_f$.
2. Пока существует путь p из s в t в $\mathcal{N}_f$, пропустим поток f' вдоль него.
3. Складываем поток $f = f + f'$
4. Перестраиваем вдоль p остаточную суть $\mathcal{N}_f$

Корректность вытекает из теоремы Форда-Фалкерсона. Временная сложность: в худшем случае будет $\mathcal{O}(|f|)$ итераций, а каждая из них — поиск пути в остаточной сети, то есть запуск какого-то обхода за $\mathcal{O}(|V| + |E|)$. Итого: $\mathcal{O}(|E||f|)$.

Утверждение. Если $c : E \to \mathbb{Z}^+$, то алгоритм сходится за указанное выше время. Иначе существует сеть, в которой алгоритм не сходится за конечное время и даже не сходится к верному ответу.

Алгоритм Эдмондса-Карпа

Алгоритм Эдмондса-Карпа это алгоритм Форда-Фалкерсона, только выбирается кратчайший (по числу ребер) путь из s в t в остаточной сети с помощью BFS.

Теорема (б/д). Время работы алгоритма Эдмондса-Карпа составляет $\mathcal{O}(|V||E|^2)$.

Алгоритм Диница

Def. *Слоистой сетью* $\mathcal{N}_l$ для сети $\mathcal{N}$ назовем $\mathcal{N}_l = (G_l = (V, E_l), c_l, s, t)$, где

$$E_l = \{e = (u, v) \in E \mid d[v] - d[u] = 1\}$$
$$c_l(u, v) = c(u, v) \cdot I\{(u, v) \in E_l\}$$

$d[v]$ — расстояние в ребрах от s до v.

Def. *Блокирующий поток* — такой поток f, что не найдется пути, вдоль которого поток из s в t можно увеличить.

Note. Заметьте, тут не идет речь про остаточную сеть. Говорится только о ребрах в исходной сети без возникающих обратных.

Algorithm. Схема алгоритма Диница

1. Построим сеть $\mathcal{N}$, определим поток $f = 0$. Строим $\mathcal{N}_f$.
2. Строим слоистую сеть $\mathcal{N}_{f_l}$ из остаточной сети.
3. Если в $\mathcal{N}_{f_l}$ t недостижима из s, то алгоритм окончен. Иначе найдем блокирующий поток f' в $\mathcal{N}_{f_l}$.
4. $f = f + f'$, обновляем остаточную сеть вдоль пропущенного потока.

Утверждение. Алгоритм Диница корректен

Доказательство: По теореме Форда-Фалкерсона.

■

Очевидно, шаги 1, 2 и 4 делаются за $\mathcal{O}(|V| + |E|)$. Осталось узнать, как искать блокирующий поток и сколько итераций будет, пока не произойдет выход из алгоритма.

Утверждение. Расстояние между истоком и стоком строго увеличивается после каждой фазы алгоритма, т.е. $d'[t] > d[t]$, где $d'[t]$ — значение, полученное на следующей фазе алгоритма.

Доказательство: Проведём доказательство от противного. Пусть длина кратчайшего пути из истока в сток останется неизменной после очередной фазы алгоритма. Слоистая сеть строится по остаточной. Из предположения следует, что в остаточной сети будут содержаться только рёбра остаточной

сети перед выполнением данной фазы, либо обратные к ним. Из этого получаем, что нашёлся путь из s в t, который не содержит насыщенных рёбер и имеет ту же длину, что и кратчайший путь. Но этот путь должен был быть «заблокирован» блокирующим потоком, чего не произошло. Получили противоречие. Значит длина изменилась.

■

Следствие. Число итераций в алгоритме Диница составляет $|V| - 1$.

Доказательство: На каждой итерации реберное расстояние увеличивается, а оно не может превосходить $|V| - 1$.

■

Удаляющий обход

Идея заключается в том, чтобы по одному находить пути из истока s в сток t, пока это возможно. Обход в глубину найдёт все пути из s в t, если из s достижима t, а пропускная способность каждого ребра $c(u, v) > 0$ поэтому, насыщая рёбра, мы хотя бы единожды достигнем стока t, следовательно блокирующий поток всегда найдётся.

Ускорим данный алгоритм. Будем удалять в процессе обхода в глубину из графа все рёбра, вдоль которых не получится дойти до стока t. Это очень легко реализовать: достаточно удалять ребро после того, как мы просмотрели его в обходе в глубину (кроме того случая, когда мы прошли вдоль ребра и нашли путь до стока). С точки зрения реализации, надо просто поддерживать в списке смежности каждой вершины указатель на первое не удалённое ребро, и увеличивать этот указатель в цикле внутри обхода в глубину.

Если обход в глубину достигает стока, насыщается как минимум одно ребро, иначе как минимум один указатель продвигается вперед. Значит один запуск обхода в глубину работает за $\mathcal{O}(|V| + K)$, где K — число продвижения указателей. Ввиду того, что всего запусков обхода в глубину в рамках поиска одного блокирующего потока будет $\mathcal{O}(P)$, где P — число рёбер, насыщенных этим блокирующим потоком, то весь алгоритм поиска блокирующего потока отработает за $\mathcal{O}(P|V|+\sum\limits_i K_i)$, что, учитывая, что все указатели в сумме прошли расстояние $\mathcal{O}(|E|)$, дает асимптотику $\mathcal{O}(P|V|+|E|)$. В худшем случае, когда блокирующий поток насыщает все рёбра, асимптотика получается $\mathcal{O}(|V||E|)$.

Таким образом, научились искать блокирующий поток за $\mathcal{O}(|V||E|)$, а значит алгоритм Диница с удаляющим обходом отработает за $\mathcal{O}(|V|^2|E|)$, что «на одну степень $|V|$ быстрее».

Теоремы Карзанова

Существует две теоремы Карзанова о числе итераций алгоритма Диница, однако нас будет интересовать только первая.

Def. Пусть $\mathcal{N} = (G, c, s, t)$ такова, что $c : E \to \mathbb{Z}^+$, тогда *потенциал вершины* $v \in V$ равен $\varphi(v) = \min(c_+(v), c_-(v))$, где $c_+(v) = \sum\limits_{u \in V} c(u, v)$ и $c_-(v) = \sum\limits_{u \in V} c(v, u)$.

Def. *Потенциалом сети* $\mathcal{N}$ с целочисленными емкостями назовем $\varphi(\mathcal{N}) = \sum\limits_{v \in V \setminus \{s,t\}} \varphi(v)$.

Теорема (Карзанов) (б/д). Число итераций в алгоритме Диница в целочисленной сети $\mathcal{N}$ равно $\mathcal{O}(\sqrt{\varphi(\mathcal{N})})$.

Алгоритм Хорпкрофта-Карпа

Хотим искать максимальное паросочетание в двудольном графе быстрее, чем алгоритмом Куна. Построим классическую сеть для сведения задачи поиска максимального паросочетания в двудольном графе к задаче о максимальном потоке.

Посчитаем ее потенциал. Очевидно, что $\forall v \in L \sqcup R \ \varphi(v) \leq 1$, откуда $\varphi(\mathcal{N}) \leq |V|$. По теореме Карзанова число итераций составит $\mathcal{O}(\sqrt{|V|})$, а каждая итерация за линейное от размеров графа время. Откуда итоговое время работы: $\mathcal{O}(|E|\sqrt{|V|})$.

Модуль 4. Строковые алгоритмы.

Лекция 10.

Мы приступаем к строковым алгоритмам. В данном модуле у нас будет одна базовая задача: дан текст T и паттерн P, необходимо найти вхождения паттерна P в текст T. В рамках этой лекции будут рассмотрены три подхода к решению этой задачи, а далее будет рассмотрены вариации этой задачи и как их можно решать.

Алгоритм Рабина-Карпа

Обсудим полиномиальную хеш-функцию в отношении строк. Для данного шаблона $p[1 : m]$ такой хеш определён следующим образом:

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

где q — некоторое простое число, а x — число от 0 до $q - 1$. Начитаем массив $p_h[]$ хешей префиксов, тогда хеш подстроки имеет вид:

$$hash(p[l : r]) = (p_h[r] - p_h[l - 1] \cdot x^{r-l}) \bmod q$$

Algorithm. Алгоритм Рабина-Карпа поиска паттерна в тексте.

1. Посчитаем полиномиальный хеш паттерна за $\mathcal{O}(|P|)$.
2. Посчитаем массив префиксных хешей текста T за $\mathcal{O}(|T|)$.
3. Переберем в тексте все подстроки размера $|P|$ (их $\mathcal{O}(|T|)$) и для каждой сравним хеш с хешом P. Если не совпали, то точно в данном месте вхождения нет, иначе проверяем в лоб.

Теорема (б/д). Полиномиальная хеш-функция выше гарантирует, что вероятность коллизии не превосходит $\frac{|P|-1}{q}$.

Следствие. Путем выбора простого $q > |P|^2$ получаем, что вероятность коллизии меньше $\frac{1}{|P|}$, а значит в среднем время работы составит $\mathcal{O}\left(|P| + |T|(1 + |P|\frac{1}{|P|})\right) = \mathcal{O}(|P| + |T|)$.

Префикс-функция

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

Note. Пустая строка всегда является супрефиксом любой строки (кроме пустой).

Def. *Префикс-функцией* от строки S называют такой массив $\pi(S)$ длины $|S|$, что $\pi(S)[i]$ равно длине максимального *несобственного (то есть не равного всей строке)* супрефикса строки $S[:i]$.

Note. В данном разделе подразумевается, что $S[:i]$ включает в себя $S[i]$.

Algorithm. Линейный алгоритм построения.

Заметим, что $\pi[i+1] \leqslant \pi[i] + 1$. Чтобы показать это, рассмотрим суффикс, оканчивающийся на позиции $i+1$ и имеющий длину $\pi[i+1]$. Удалив из него последний символ, мы получим суффикс, оканчивающийся на позиции i и имеющий длину $\pi[i+1] - 1$, следовательно неравенство $\pi[i+1] > \pi[i] + 1$ неверно.

Теперь осознаем, что у нас возможны два случая:

- Если $S[i+1] = S[\pi[i]]$, то $\pi[i+1] = \pi[i] + 1$ — тривиально по определению.
- Если $S[i+1] \neq S[\pi[i]]$. Как быстро найти $\pi[i+1]$? Заметим, что мы ищем максимальный по длине супрефикс, заканчивающийся в $(i+1)$-й позиции. Такой супрефикс состоит из какого-то супрефикса строки $S[:i]$ и символа $S[i+1]$. А теперь остается осознать, что все супрефиксы имеют длины $\pi[i], \pi[\pi[i]], \ldots$. Почему это верно? Рассмотрим $S[:i]$.

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

  А из этого следует, что мы перебираем все супрефиксы в порядке вложенности, то есть $\pi[i+1] = \pi[j] + 1$, где j — такой индекс, что $S[\pi[j]] = S[i+1]$ и при этом j — первый индекс из последовательности $\pi[i], \pi[\pi[i]], \ldots$

Объединяя оба случая, получаем, что $\pi[i+1] = \pi[j] + 1$, где j — такой индекс, что $S[\pi[j]] = S[i+1]$ и при этом j — первый индекс из последовательности $i, \pi[i], \pi[\pi[i]], \ldots$

Псевдокод алгоритма:

```
int[] prefixFunction(string s):
    p[0] = 0
    for i = 1 to s.length - 1
            k = p[i - 1]
            while k > 0 and s[i] != s[k]
                    k = p[k - 1]
            if s[i] == s[k]
                    ++k
            p[i] = k
    return p
```

Осталось разобраться со временем работы. Очевидно, оно определяется количеством итераций внутреннего `while`. Теперь стоит отметить, что k увеличивается на каждом шаге не более чем на единицу, значит максимально возможное значение $k = |S|-1$. Поскольку внутри цикла `while` значение k лишь уменьшается, получается, что k не может суммарно уменьшиться больше, чем $|S| - 1$ раз. Значит цикл `while` в итоге выполнится не более $|S|$ раз, что дает итоговую оценку времени алгоритма $\mathcal{O}(|S|)$.

Алгоритм Кнута-Морриса-Пратта

Задача. Дана цепочка T и образец P. Требуется найти все позиции, начиная с которых P входит в T.

Algorithm. Построим строку $S = P\#T$, где $\#$ — любой символ, не входящий в алфавит P и T. 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 допишите в ответ $i - |P|$.

Корректность и линейность времени напрямую вытекают из определения префикс-функции и алгоритма выше.

Лекция 11.

В данном разделе будет фигурировать *размер алфавита* $|\Sigma|$. Полагаем, что он конечен.

Зет-функция

Def. Зет-функция от строки S и индекса i — длина наиболее длинного префикса начинающегося с позиции i суффикса строки S, который одновременно является и префиксом всей строки S. Обычно

считается, что зет-функция в нулевой позифии равна нулю.

Пример: $zet\_func(\texttt{abcdabscabcdabia}) = [0, 0, 0, 0, 2, 0, 0, 0, 6, 0, 0, 0, 2, 0, 0, 1]$

~~*Данный конспект не стремится дискредетировать чьи-либо действия, поэтому в нем не будет неоптимального или наивного алгоритма построения зет-функции*~~

Algorithm. Линейный алгоритм

Зет-блоком назовем подстроку с началом в позиции i и длиной $zet[i]$. Для работы алгоритма заведём две переменные: $left$ и $right$ — начало и конец Зет-блока строки S с максимальной позицией конца $right$ (среди всех таких Зет-блоков, если их несколько, выбирается наибольший). Изначально $left = 0$ и $right = 0$.

Пусть нам известны значения Зет-функции от 0 до $i - 1$. Найдём $zet[i]$. Рассмотрим два случая:

1. Пусть $i > right$, тогда просто пробегаемся по строке S и сравниваем символы на позициях $S[i+j]$ и $S[j]$. Пусть j первая позиция в строке S для которой не выполняется равенство $S[i+j] = S[j]$, тогда j это и Зет-функция для позиции i. Тогда $left = i$, $right = i + j - 1$. В данном случае будет определено корректное значение $zet[i]$ в силу того, что оно определяется наивно, путем сравнения с начальными символами строки.

2. Пусть $i \leq right$, тогда сравним $zet[i-left]+i$ и $right$. Если $right$ меньше, то надо просто наивно пробежаться по строке начиная с позиции $right$ и вычислить значение $zet[i]$. Корректность в таком случае также гарантирована. Иначе мы уже знаем верное значение $zet[i]$, так как оно равно значению $zet[i - left]$.

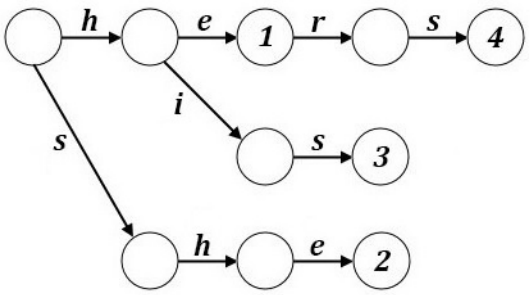
Заметим, что данный алгоритм обращается к каждому символу строки не более двух раз, поэтому время работы: $O(|S|)$. Далее будет приведен псевдокод алгоритма выше:

```
int[] ZetFunction(s : string):
    int[] zet_func = int[n]
    int left = 0, right = 0
    for i = 1 to n - 1
        zet_func = max(0, min(right - i, zet_func[i - left]))
        while i + zet_func[i] < n and s[zet_func[i]] == s[i + zet_func[i]]
            ++zet_func[i]
        if i + zet_func[i] > right
            left = i, right = i + zet_func[i]
    return zet_func
```

Бор

Def. Бор — структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной.

Пример: Бор для набора образцов he,she,his,hers



Отметим, что внутри узла у нас есть переходы по буквам, но как их хранить? Составим табличку:

Контейнер	Время построения	Время поиска	Память
vector	$\mathcal{O}\left(\sum\limits_{w\in dict}\|w\|\right)$	$\mathcal{O}\left(\|S\|\right)$	$\mathcal{O}\left(\|\Sigma\|\sum\limits_{w\in dict}\|w\|\right)$
map	$\mathcal{O}\left(\log\|\Sigma\|\sum\limits_{w\in dict}\|w\|\right)$	$\mathcal{O}\left(\|S\|\cdot\log\|\Sigma\|\right)$	$\mathcal{O}\left(\sum\limits_{w\in dict}\|w\|\right)$
хеш-таблица	$\mathcal{O}\left(\sum\limits_{w\in dict}\|w\|\right)$	$\mathcal{O}\left(\|S\|\right)$	$\mathcal{O}\left(\sum\limits_{w\in dict}\|w\|\right)$

Казалось бы, хеш-таблица наше все, но константа велика.

Алгоритм Ахо-Корасик

Note. $[u]$ — слово, которое составляет путь от корня до вершины u в боре.

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

Note. Обозначать суффиксную ссылку вершины u будем за $link(u)$.

Note. Суффиксная ссылка для корня в общем случае не определена. Давайте доопределим ее вершиной `NIL`.

Алгоритм Ахо-Корасик строит суффиксные ссылки в боре.

Def. Введем функцию $to(u, c)$, равную вершине, куда из вершины u можно перейти по букве c.

Определяется она следующим образом:

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

Note. Заметим такое соотношение: $link(vertex([u] + c)) = vertex([to(link(u), c)])$.

То есть по сути перед нами автомат Ахо-Корасик. А если приглядеться, то перед нами по сути автомат префикс-функции. Действительно, если бы у нас была одна строка, то суффиксные ссылки как раз вели бы в позиции префикс-функции. То есть мы построили *обобщенную префикс-функцию* для набора строк.

За сколько строится данный автомат?

1. Строим бор на данном наборе слов.
2. С помощью BFS из корня насчитаем функции $link$ и $to(\cdot, c)$. Почему это работает? Из соотношений выше напрямую следует, что данные функции можно насчитать через вершины выше, чем текущая. А по предположению индукции мы верили, что для всех выше все посчитано корректно. База индукции — корень. Для него ручками посчитаем все по определению.

Так как пересчет $link$ работает за $\mathcal{O}(1)$ и пересчет всех $to(\cdot, c)$ для вершины работает за $\mathcal{O}(|\Sigma|)$, получаем, что построение занимает $\mathcal{O}\left(|\Sigma| \sum\limits_{w \in dict} |w|\right)$ (в наивном варианте с хранением в векторе).

Вопрос: за сколько работает все это на других контейнерах?

Задача. Пусть есть набор паттернов $P_1, \ldots, P_k$ и текст T, в который мы хотим найти вхождения всех паттернов. Как это можно сделать быстрее, чем наивный КМП, который отработает за $\mathcal{O}\left(k|T| + \sum\limits_{i=1}^{k} |P_i|\right)$?

Решение. Построим автомат Ахо-Корасик на паттернах за $\mathcal{O}\left(|\Sigma| \sum\limits_{i=1}^{k} |P_i|\right)$. Теперь будем идти по функции to текстом T из корня за $\mathcal{O}(|T|)$. Для каждой вершины бора будем запоминать, сколько раз мы в ней были c_i. Посчитаем для каждой терминальной вершины сумму c_i для вершин, из которых в неё можно дойти по суффиксным ссылкам. Сделаем это просто динамикой по дереву обратных суффиксных ссылок. Это число и будет количеством вхождений каждого паттерна. Итого: $\mathcal{O}\left(|T| + |\Sigma| \sum\limits_{i=1}^{k} |P_i|\right)$.

Note. Данный алгоритм реализован в утилите grep, то есть алфавит маленький, а паттернов крайне много может быть.

Лекция 12.

Суффиксный автомат. Определения

Note. В данном разделе считается, что читатель знаком с теорией конечных автоматов.

Def. *Правый контекст* слова x для языка L называют можество $R_L(x) = \{w \mid xw \in L\}$.

Def. Два слова *эквивалентны* относительно языка L, то есть $x \sim_L y$, если $R_L(x) = R_L(y)$.

Теорема (Майхилл, Нероуд) (б/д). Если в языке L конечное число классов эквивалентности, равное k, то

1. Язык автоматный
2. В любом ДКА, распознающем данный язык, хотя бы k состояний
3. Существует ДКА, распознающий L, в котором ровно k состояний, где каждое состояние отвечает за соответствующий класс эквивалентности

Def. *Суффиксным автоматом* для строки S назовем минимальный по числу состояний ДКА, распознающий язык суффиксов L.

Note. Далее будем считать, что речь идет только о подстроках S, а вместо языка L будем обозначать эквивалентность по языку S, порожденному суффиксами строки S.

Def. *longest(v)* — самая длинная строка, которую можно прочесть, дойдя до состояния v в суффиксном автомате.

Def. *len(v)* — длина *longest(v)*.

Def. *link(v)* — такая вершина, что в ней лежит самый длинный суффикс *longest(v)*, не лежащий в v.

Суффиксный автомат. Критерий longest

Утверждение. Пусть $u \sim_S v$, тогда либо u — суффикс v, либо v — суффикс u.

Доказательство: Существует хотя бы одно слово w такое, что либо $uw \in L_S$ и является суффиксом S, либо $vw \in L_S$ и является суффиксом S, то тогда u и v имеют общую позицию последнего вхождения их символа, и одно слово является суффиксом другого.

■

Утверждение. Пусть v — вершина автомата. Тогда в v лежит *longest(v)*, и несколько её самых длинных суффиксов.

Доказательство: В C лежат только суффиксы *longest(v)* по предудущему утверждению. Теперь поймем, почему там лежит непрерывный отрезок суффиксов. Пусть $u = longest(v)$, а $w \in v$, тогда, в

силу того, что вершины соответствуют классам эквивалентности, $R_L(u) = R_L(w)$. Пусть x — какой-то суффикс u, при этом $|x| > |w|$. Тогда заметим, что $R_L(x) \subseteq R_L(w)$, так как x — суффикс w. Аналогично $R_L(u) \subseteq R_L(x)$, при этом $R_L(u) = R_L(w)$. Значит $R_L(u) = R_L(x) = R_L(w)$.

■

Теорема (Критерий *longest*). Пусть u — подстрока S. Тогда $u = longest([u]_S)$ тогда и только тогда, когда либо u — префикс S, либо существуют различные a и b, что au, bu — подстроки S.

Доказательство:

$\Longrightarrow$ Пусть u — не $longest([u]_S)$, тогда $\gamma u = longest([u]_S)$, $|\gamma| \geq 1$. Откуда u не является префиксом, при этом любое вхождение u начинается с γ, так как $R_S(u) = R_S(\gamma u)$.

$\Longleftarrow$ Пусть $u = longest([u])$, u — не префикс S. Рассмотрим все символы перед вхождениями u. Нужно доказать, что среди них есть хотя бы 2 различных. Если это неверно, то они равны d, тогда $u \sim_S du$, и $|du| > |longest([du])|$. Противоречие.

■

Следствие. Что было *longest*, то навсегда им и будет.

Суффиксный автомат. Устройство переходов в одну вершину

Утверждение. Пусть в автомате есть ребра $u_1 \to v, u_2 \to v, \ldots, u_k \to v$, при этом $len(u_i) > len(u_{i+1})$. Тогда

1. Все переходы идут по одной букве c
2. $link(u_i) = u_{i+1}$

Доказательство:

1. Рассмотрим u_i, из которой возможно перейти в v по букве c, но единственный способ получить одно из слов, которым соответствует состояние v — сделать конкатенацию u_k и c. То же и для других классов эквивалентности.
2. Так как в $[v]_S$ лежат ее самые длинные суффиксы, то откусывание последнего символа даст множество суффиксов $longest(u_1)$, которое, в силу убывания длин, как раз по определению суффиксной ссылки, дает нам второе утверждение: $link(u_i) = u_{i+1}$.

■

Суффиксный автомат. Новые состояния при дописывании символа

Пусть к строке S дописали символ c. Нужно исследовать, какие новые состояния могут появиться в автомате.

Утверждение. При дописывании символа c к строке S образуется не более двух новых состояний:

1. Состояние, у которого $longest(v) = Sc$, которое обязательно появится.
2. Состояние $[T]$, где $T = longest([T]_{Sc})$ — самый длинный суффикс Sc, который является подстрокой S, которое необязательно появится.

Доказательство:

1. Так как Sc — префикс Sc, значит, по критерию *longest*, $Sc = longest([Sc]_{Sc})$.
2. Если T не подстрока S, то она является суффиксом Sc, не являющейся подстрокой S, значит она ранее не могла быть рассмотрена и обязательно попадет в $[Sc]_{Sc}$. Значит T — суффикс Sc, который является подстрокой S. Если T при этом не наибольший такой суффикс, то они являются суффиксами наибольшей такой строки и попадут в $[longest([T]_{Sc})]_{Sc}$, а значит T должна быть наибольшим таким суффиксом.

   При этом, по критерию *longest*, найдется такое y, что yT — не подстрока S. Иначе нового состояния не возникнет в автомате.

Следствие. $link([Sc]_{Sc}) = [T]_{Sc}$, где T из утверждения выше.

Утверждение. При $|S| \geq 2$ число состояний в автомате не превосходит $2|S| - 1$.

Доказательство: Индукция по длине строки. База. При $|S| = 2$ число состояний равно трем, что равно $2|S| - 1 = 1 \cdot 2 - 1 = 3$.

Переход. Пусть число состояний в автомате для S равно k, тогда для Sc число состояний не превосходит $k + 2$. При этом $k \leq 2|S| - 1$, откуда $k + 2 \leq 2|S| - 1 + 2 = 2(|S| + 1) - 1$.

■

Теорема (б/д). Начиная с некоторого n, в суффиксном автомате для строки длины n число ребер не превосходит $3n - 4$.

Лекция 13.

Суффиксный автомат. Линейный алгоритм

Алгоритм будет итеративным: перестраивание автомата при дописывании по одному символу. Потенциальные изменения:

1. Ребра, ведущие в $[Sc]_{Sc}$. (Ребер из нее быть не может по соображениям длины пути и строки Sc).

2. При возникновении нового состояния T, ребра, ведущие в/из него.

Тогда итоговый алгоритм:

1. Очевидно, что должно появиться ребро $([S]_{Sc}, c) \to [Sc]_{Sc}$. По утверждению об устройстве ребер в одну вершину, все такие ребра ведутся из вершин, образующих *суффиксный путь* (путь вдоль суффиксных ссылок). Таким образом, нужно провести ребро $([S]_{Sc}, c) \to [Sc]_{Sc}$ и, прыгая по суффиксным ссылкам, проводить ребра, пока не доберемся до вершины, из которой есть переход по c. Допустим, что пришли в q_0, из которого нет ребра по букве c, тогда не было буквы c в S и при этом $link([Sc]_{Sc}) = q_0$.

Теперь допустим, что пришли в вершину p такую, что $(p, c) \to q \in \Delta$. Тогда есть два варианта: надо ли разделять вершину или нет. Тогда, из обозначений выше, $T = longest(p) + c$. Вершину q надо будет расщепить, если $T \neq longest(q)$. Таким образом, критерий расщепления: $len(q) > len(p) + 1$.

2. Допустим, что не надо расщеплять, то есть $len(q) = len(p) + 1$. Тогда достаточно провести $link([Sc]_{Sc}) = q$. На этом обработка окончена.

3. Допустим, что надо расщеплять, то есть $len(q) > len(p)+1$. Тогда создадим вершину clone, при этом в q будут лежать суффиксы $longest(q)$, которые длиннее T, а $longest(clone) = T$. Рассмотрим суффиксный путь. Для вершин, соответствующим более длинным суффиксам строки S, чем p, ничего не изменится, исходящие рёбра, ведущие в q, сохранятся. А вот для остальных надо будет перенаправить ребра в $clone$, то есть делаем снова прыжки из p по суффиксным ссылкам и, пока ребра шли в q, отправляем их в $clone$.

   Заметим, что раз T — подстрока S и входила в q, то $R_{Sc}(clone) = R_{Sc}(q) \sqcup \{\varepsilon\}$, так как множество вхождений T в Sc совпадает с множеством вхождений T в S, кроме того, что T еще и суффикс Sc. А значит переходы из $clone$ совпадут с переходами из q. При этом $link([Sc]_{Sc}) = clone$, $link(clone) = link(q)$, $link(q) = clone$, $len(clone) = len(p) + 1$.

Утверждение. После построения автомата терминальными будут те вершины, которые лежат на суффиксном пути из $[S]_S$.

Доказательство: Следует напрямую из определения суффиксной ссылки. ■

Суффиксный автомат. Оценка асимптотики

Рассмотрим первые два случая. Так как в них никакие старые ребра не меняются, а просто добавляются новые ребра, при этом, как известно, их линейное количество, а значит суммарно все итерации алгоритма 1 и 2 типов отработают за $\mathcal{O}(|S|)$.

Осталось разобраться с третьим случаем. Рассмотрим длину суффиксного пути из вершины, соответствующей всей строке. Он имеет вид: $[u_1, \ldots, u_l, \ldots, v_1, \ldots, v_k, w_1, \ldots]$, где $(u_i, c) \to [Sc]_{Sc} \in \Delta$, переходы из v_i вели в q, а стали вести в *clone*, а суммарная длина суффиксного пути равна m, тогда длина суффиксного пути после перестроения стала $m + 1 - (k - 1) = m + 2 - k$, так как все v_i схлопнулись в *clone*, то есть уменьшилось на $k - 1$ и, при этом, добавилась в конце одна вершина.

Таким образом, длина пути увеличивается не более чем на два (*clone* и $[Sc]_{Sc}$) и при этом каждый раз уменьшается хотя бы на единицу, а значит получаем линейную асимптотику.

Лекция 14.

FFT

Задача. Перемножить два многочлена

$$P(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{p-1}$$
$$Q(x) = b_0 + b_1 x + \ldots + b_{m-1} x^{q-1}$$

за $\mathcal{O}((n+m)\log(n+m))$.

Решение. Пусть $n = 2^{\lceil \log_2(p+q+1) \rceil}$, дополним P и Q нулями до такой длины. Пусть $\omega = e^{\frac{2i\pi}{n}}$ Какой у нас будет план?

1. Найти $P(\omega^0), \ldots, P(\omega^{n-1})$ и $Q(\omega^0), \ldots, Q(\omega^{n-1})$ (то есть вычислить *дискретные преобразования Фурье*) за $\mathcal{O}(n \log n)$
2. Найти $R(\omega^i) = P(\omega^i) \cdot Q(\omega^i)$. Делается в лоб за $\mathcal{O}(n)$
3. Найти $c_0, \ldots, c_n$ (коэффициенты $R(x)$) через $R(\omega^i)$ (то есть вычислить *обратные дискретные преобразования Фурье*) за $\mathcal{O}(n \log n)$

1. Рассмотрим первый шаг нашего алогритма. Пусть

$$P(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{p-1}$$
$$P_0(x) = a_0 + a_2 x + a_4 x^2 + \ldots$$
$$P_1(x) = a_1 + a_3 x + a_5 x^2 + \ldots$$

Тогда $P(x) = P_0(x^2) + xP_1(x^2)$. То есть мы свели задачу к тому, что надо вычислить значения двух многочленов в $n/2$ точках $(\omega^0, \omega^2, \ldots, \omega^{2(n-1)} = \omega^{-2} = \omega^{n-2})$, так как степени многочленов в два раза меньше. Откуда получаем, что время работы составит:

$$T(n) = 2T(n/2) + O(n) \Longrightarrow T(n) = \mathcal{O}(n \log n)$$

2. Тут в лоб все считаем за $\mathcal{O}(n)$

3. Обратимся к матричной постановке задачи вычисления задачи значения многочлена в точках:

$$\begin{pmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \dots & \omega^{0\cdot(n-1)} \\ \omega^{1\cdot 0} & \omega^{1\cdot 1} & \dots & \omega^{1\cdot(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^{(n-1)\cdot 0} & \omega^{(n-1)\cdot 1} & \dots & \omega^{(n-1)\cdot(n-1)} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(\omega^0) \\ P(\omega^1) \\ \vdots \\ P(\omega^{n-1}) \end{pmatrix}$$

Обозначим через $W = (\omega^{ij})_{i,j=0}^{n-1}$. Тогда задача восстановления коэффициентов равносильна тому, что необходимо умножить W^{-1} на вектор значений. Найдем W^{-1}. Докажем, что $W^{-1} = \frac{1}{n}V$, где $V = (\omega^{-ij})_{i,j=0}^{n-1}$.

$$(WV)_{ij} = \sum_{k=0}^{n-1} W_{ik}V_{kj} = \sum_{k=0}^{n-1} \omega^{k(i-j)} = \begin{cases} \sum\limits_{k=0}^{n-1} \omega^0 = n, & i = j \\ \frac{\omega^{i-j}(1-\omega^{n(i-j)})}{1-\omega^{i-j}} = \frac{\omega^{i-j}(1-1)}{1-\omega^{i-j}} = 0, & i \neq j \end{cases}$$

Откуда следует, что $W^{-1} = \frac{1}{n}V$.

Заметим, что на первом шаге был приведен алгоритм умножения такой специфической матрицы на вектор за $\mathcal{O}(n \log n)$, тогда данный шаг это реализация первого шага, только вместо точек $(\omega^0, \dots, \omega^{n-1})$ надо взять точки $(\omega^{-0}, \dots, \omega^{-(n-1)})$. Время такого шага будет тоже $\mathcal{O}(n \log n)$.

Note. Отметим, что, в силу закольцованности корней из единицы, $\omega^{-k} = \omega^{n-k}$, то есть не надо напрямую вычислять данные числа, достаточно реверснуть часть массива ω^k.

Таким образом, научились перемножать два многочлена за заявленное время.

Свертка последовательностей

Def. Пусть даны две последовательности $a = [a_0, \dots, a_{n-1}]$ и $b = [b_0, \dots, b_{m-1}]$, $n \geq m$. Назовем *сверткой* $a * b = c$, где $c = [c_0, \dots, c_{n-m+2}]$ — последовательность, полученная по следующей формуле:

$$(a * b)_i = c_i = \sum_{k=0}^{m-1} a_{i+k}b_k$$

Note. Перед нами скалярное произведение вектора b и всех подотрезков a как со скользящим окном. Можно увидеть непосредственную аналогию со сверткой из гармонического анализа, только перед нами свертка *дискретных функций* или *дискретизированных сигналов*.

Рассмотрим произведение

$$P(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$$
$$Q(x) = b_0 + b_1x + \dots + b_{m-1}x^{m-1}$$

Обозначим его за $R(x) = P(x)Q(x) = R_0 + R_1 x + \ldots + R_l x^l$

$$
\begin{aligned}
R_0 &= a_0 b_0 \\
R_1 &= a_0 b_1 + a_1 b_0 \\
R_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
&\vdots \\
R_{m-1} &= a_0 b_{m-1} + a_1 b_{m-2} + \ldots + a_{m-2} b_1 + a_{m-1} b_0 \\
R_m &= a_1 b_{m-1} + a_2 b_{m-2} + \ldots + a_{m-1} b_1 + a_m b_0 \\
&\vdots
\end{aligned}
$$

Заметим, что R_{m-1+k} почти совпадает с c_k, но $R_{m-1+k} = (a * b^R)_k$, откуда, для вычисления свертки, необходимо сначала развернуть b^R, а затем перемножение многочленов даст $R_{m-1+k} = (a * (b^R)^R)_k = (a * b)_k = c_k$. То есть научиличь вычислять свртку за $\mathcal{O}((n+m)\log(n+m))$.

Note. Казалось бы, как так все органично получается? Но тут стоит отметить, что из курса гармонического анализа известно, что $\widehat{f * g} = \widehat{f} \cdot \widehat{g}$, именно поэтому на самом деле все сходится.

Note. Применение данного алгоритма можно найти в сверточных нейронных слоях в нейросетях для задач компьютерного зрения и анализа сигналов.

Применение FFT к задаче неточного вхождения

Def. *Расстоянием Хэмминга* для двух строк равной длины $\rho_H(S,T) = \sum\limits_{i=0}^{|S|-1} I(S_i \neq T_i)$.

Задача. Необходимо в тексте S найти все вхождения паттерна P с точностью до k символов. То есть найти все подстроки T в S такие, что $\rho_H(T,P) \leq k$.

Решение. Для каждого символа алфавита $\sigma \in \Sigma$ построим вектора v_S^σ, которые определяются как $(v_S^\sigma)_i = I(S_i = \sigma)$. Аналогично построим вектора v_T^σ.

Посчитаем $v_S^\sigma * v_T^\sigma$ для всех $\sigma \in \Sigma$. $(v_S^\sigma * v_T^\sigma)_i$ соответствует числу таких позиций, что ровно в $(v_S^\sigma * v_T^\sigma)_i$ позиций строки $S[i : i+|T|]$ и T совпадают по букве σ. Просуммируем по всем $\sigma \in \Sigma$ вектора $(v_S^\sigma * v_T^\sigma)_i$, получим вектор u, тогда u_i соответствует тому, что $S[i : i+|T|]$ и T совпадают в u_i позициях.

Теперь построим последний вектор $w = (|T|)_{i=0}^{|S|-|T|+2} - u$, он как раз и будет соответствовать расстоянию Хэмминга между T и всеми подстроками S длины $|T|$. Тогда осталось вывести те позиции, где $w_i \leq k$, это и будет ответом.

Сложность алгоритма: $\mathcal{O}(|\Sigma|(|S|+|T|)\log(|S|+|T|))$.

Note. После бинаризации алфавита сложность: $\mathcal{O}(\log|\Sigma|(|S|+|T|)\log(\log|\Sigma|(|S|+|T|)))$