МФТИ ФПМИ

Алгоритмы и структуры данных

Годовой курс

Кулапин Артур

Осень 2022-Лето 2023

Оглавление

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 $\Theta(f(n) \cdot \log n)$?

Префиксные суммы

Рассмотрим такую задачу. Дан массив $a_1, \ldots, a_N$. К нему поступает Q запросов вида сумма на подотрезке. Наивный алгоритм работает за $\mathcal{O}(NQ)$, давайте придумаем что-нибудь интеллектуальнее.

Заметим, что $\sum\limits_{i=l}^{r} a_i = \sum\limits_{i=1}^{r} a_i - \sum\limits_{i=1}^{l-1} a_i$, то есть свели задачу к запросам на префиксах. Всего префиксов N, откуда можно за $\mathcal{O}(N)$ найти все их значения. Пусть $pref$ — массив, для которого $pref[k] = \sum\limits_{i=1}^{k} a_i$, тогда формула расчета $pref$ такова: $pref[0] = 0$, $pref[i] = a_i + pref[i-1]$.

Теперь на каждый запрос можно ответить за $\mathcal{O}(1)$, что дает асимптотику $\mathcal{O}(N + Q)$.

Как видите, мы пришли к первому алгоритму, который использует некоторый предподсчет, а далее быстро отвечает на запросы за счет особой организации данных, построенной в ходе предподсчета.

Note. Необходимо отметить, что далеко не любую операцию можно таким подходом считать. Наша операция должна быть *обратимой*.

Def. Функция $f(x, y)$ *обратима*, если $\forall x, z\ \exists! y:\ f(x, y) = z$.

Пример: Пример необратимой операции — минимум, так как уравнение $\min(x, 3) = 3$ имеет далеко не единственное решение.

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 $\mathcal{O}(1)$.

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.

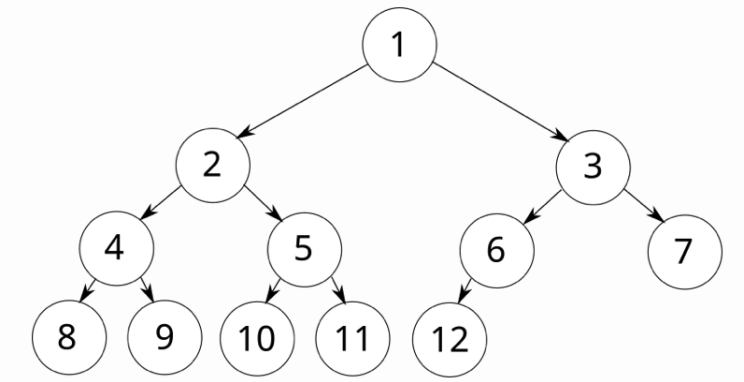
Теперь разберемся со вставкой и удалением. Определим две вспомогательные операции, а именно просеивание вниз *SiftDown* и просеивание вверх *SiftUp*. В чем их смысл? Ну он достаточно прост, просеивание вниз старается утопить элемент как можно ниже, меняя его местами с меньшим из сыновей, не наруша свойство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

Exponential Search (*)

Данный материал не попадет в экзаменационную программу

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.

Merge lower bound

Теорема. Даны два отсортированных массива A и B длины, соответственно, N и $M \geq 2N$. Любой алгоритм их слияния работает за $\Omega(N \log \frac{M}{N})$.

Доказательство: Допустим, наш алгоритм это решающее дерево, где в каждом узле спрашивается, а верно ли, что $A[i] < B[j]$. Если да, то элемент $A[i]$ должен получить в итоговом массиве меньший индекс, чем элемент $B[j]$. Тогда в листьях этого дерева получим все возможные варианты итогового массива.

Их C_{M+N}^N, так как надо найти место для N элементов среди $M + N$ позиций. При этом наше дерево полное бинарное, так как иначе будет отсутствовать какой-то лист, то есть какой-то результат слияния, а значит наша модель не полна. Таким образом, высота такого дерева равна $\log_2 C_{M+N}^N$, осталось оценить эту высоту, для этого будем пользоваться приближением Стирлинга.

$$\log_2 C_{M+N}^N \sim \log_2 \frac{\sqrt{2\pi(M+N)}\left(\frac{M+N}{e}\right)^{M+N}}{\sqrt{2\pi N}\left(\frac{N}{e}\right)^N \cdot \sqrt{2\pi M}\left(\frac{M}{e}\right)^M} = \frac{1}{2}\log_2 \frac{M+N}{2\pi MN} + \log_2 \frac{(M+N)^{M+N}}{M^M N^N} =$$

$$= \frac{1}{2}\log_2 \frac{M+N}{2\pi MN} + \log_2 \left(\frac{M+N}{M}\right)^M + \log_2 \left(\frac{M+N}{N}\right)^N =$$

$$= \frac{1}{2}\log_2 \frac{M+N}{2\pi MN} + \log_2 \left(1 + \frac{N}{M}\right)^M + \log_2 \left(1 + \frac{M}{N}\right)^N =$$

$$= \frac{1}{2}\log_2 \frac{M+N}{2\pi MN} + M \log_2 \left(1 + \frac{N}{M}\right) + N \log_2 \left(1 + \frac{M}{N}\right)$$

Теперь обратимся к факту, что $M \geq 2N$, значит $\frac{N}{M} \leq \frac{1}{2}$, то есть $M \log_2 \left(1 + \frac{N}{M}\right) = \Theta(M)$, при этом $\frac{M+N}{MN} = \frac{1}{N} + \frac{1}{M}$, откуда получаем, что первое слагаемое просто стремится к нулю, значит верно следующее рассуждение:

$$\log_2 C_{M+N}^N \sim \frac{1}{2} \log_2 \frac{M+N}{2\pi MN} + M \log_2 \left(1 + \frac{N}{M}\right) + N \log_2 \left(1 + \frac{M}{N}\right) =$$
$$= \Theta(M) + N \log_2 \left(1 + \frac{M}{N}\right) = \Omega(N) + N \log_2 \left(1 + \frac{M}{N}\right) = \Omega\left(N \log \frac{M}{N}\right)$$

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.

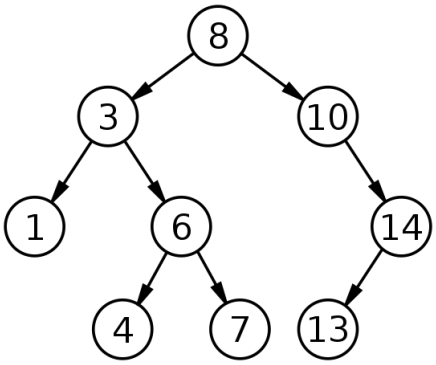
Note. В данной книге будут рассматриваться только бинарные деревья поиска, в которых все элементы различны. Поддержка одинаковых элементов будет обсуждена на семинарски занятиях.

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:

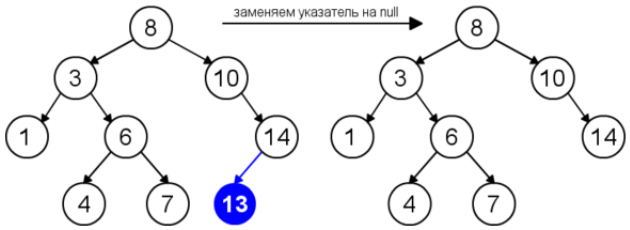
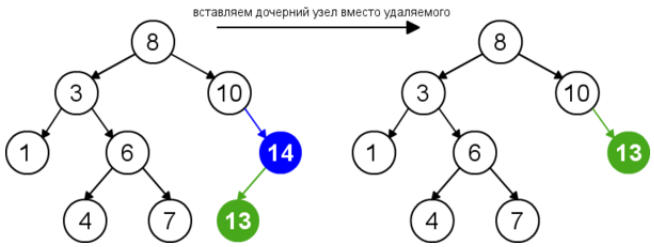
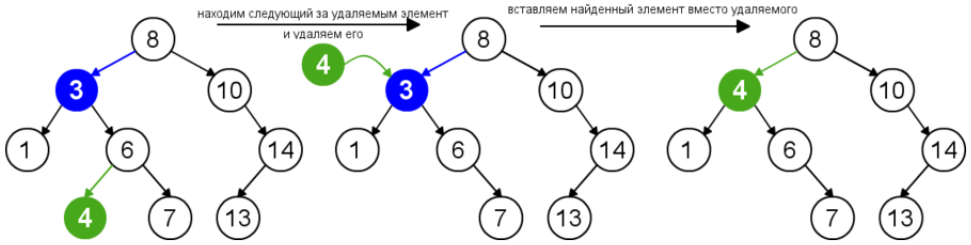
1. Удаляемый узел не имеет детей, тогда все просто, удаляем его, ничего не сломав.
2. Удаляемый узел имеет ровно одного ребенка. Тогда подвешиваем сына удаляемого узла к родителю удаляемого узла и все.
3. Удаляемый узел имеет ровно двух детей. Тогда нам надо поставить на его место кого-то, чтобы не сломать 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

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 {
```

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

```
    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 алго-

ритмом 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.

■

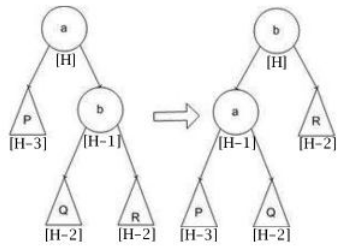
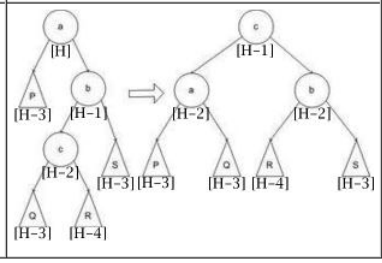
Балансировка

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

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 $diff[c] = 0$

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 сбалансированное. 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 $\mathcal{O}^*(\log N)$.

Лекция 7.

B-дерево

Прилетело НЛО и украло записи (когда-нибудь вернет).

Определение

Мотивация

Поиск

Вставка

Удаление

Модуль 4. Обработка запросов на отрезке.

Лекция 8.

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:

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

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 9.

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 $f(g(g(i))) \le f(g(i)) \le i < g(i) < g(g(i))$, 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 $f(k) \leq i \leq k$.

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 $k' \leq i'$, 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:

$$i' = 0\ldots x\ldots_2$$
$$k' = 1\ldots0\ldots_2$$

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_[idx]);
    array_[idx] = val;
  }

 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] += delta; }
  }

  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:

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

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 (int 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 аналогично дописывается обновление. Ну и все, победа.

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. Хеш-таблицы. ДП.

Лекция 10.

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 $h : U \to \{0, 1, \dots, k-1\}$ 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 11.

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.

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.

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

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 12.

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: $\mathcal{O}(2^N N^2)$.

Модуль 6. Геометрия

Геометрические примитивы

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.

Лекция 13.

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.

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.

Лекция 14.

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. Дествительно, рассмотрим две точки, суммой которых она является. Рассмотри проекции их радиус-векторов на d. Тогда при взятии точек с наибольшими проекциями на d мы должны были получить крайнюю точку.

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.

Поиск ближайших двух точек

`Данный материал недоступен.`