ЦРИТО
Центр Развития
ИТ-Образования

ОЛИМПИАДНЫЕ
ШКОЛЫ МФТИ

МФТИ

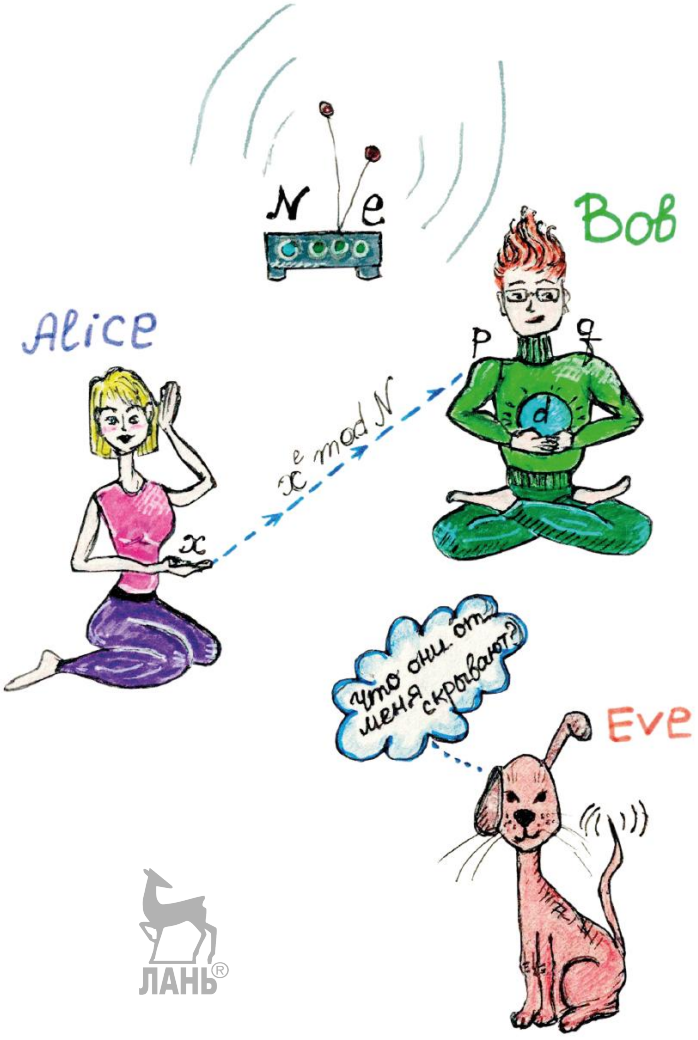
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ»
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЦЕНТР РАЗВИТИЯ ИТ-ОБРАЗОВАНИЯ

ЛАНЬ®

АЛГОРИТМЫ И МОДЕЛИ ВЫЧИСЛЕНИЯ



ЛАНЬ®

Дмитрий Голубенко, Алексей Крошнин, Эдуард Горбунов


ЛАНЬ®


ЛАНЬ®

Федеральное государственное автономное
образовательное
учреждение высшего образования
«Московский физико-технический институт
(государственный университет)»
Центр развития ИТ-образования

Алгоритмы и модели вычисления

Дмитрий Голубенко
Алексей Крошнин
Эдуард Горбунов

Москва, 2019

Оглавление

Предисловие

Эта книжка написана по мотивам материалов одноименного курса кафедры моу на Физтехе, записанных Димой Голубенко, Лешей Крошниным и Эдом Горбуновым. Курс вел Сергей Тарасов, и он же придумал концепцию, сильно отличающую «Алгоритмы и модели вычисления» от других аналогичных курсов. Стандартный курс по алгоритмам — обзор основных алгоритмов(быстрая сортировка, поиск медианы, обходы графов) и структур данных, необходимых каждому программисту. Нетрудно видеть, что почти все эти основные алгоритмы возникли во время решения некоторых математических задач. Читая «Алгоритмы и модели вычисления», Сергей в первую очередь стремился продемонстрировать связь алгоритмов и математики, какие алгоритмы и как позволяют решать математические задачи (из самых разных областей, будь то теория чисел или топология) и какая математика лежит в основе тех или иных алгоритмов. Теория алгоритмов возникла естественным образом из вычислительных задач в разных областях математики и по сей день остается живой областью, в которой некоторые тривиально сформулированные вопросы до сих пор открыты. В этой книжке мы расскажем математическим языком об основных алгоритмах сортировки, алгебры, теории чисел и теории графов.

Многие школьники изучают алгоритмы, готовясь к олимпиадам по программированию. Возможно, что эта книга поможет математикам-олимпиадникам, не занимавшимся алгоритмами, заинтересоваться олимпиадным программированием, а может быть — и теорией алгоритмов.

Авторы благодарят Сергея Тарасова, собравшего необычную комнаду семиинаристов и придумавшего оригинальную концепцию курса, а также сотрудников Центра Развития ИТ-Образования и лично Татьяну Бабичеву, чьи доброта, отзывчивость и усилия сделали существование этой книжки возможным.


ЛАНЬ®


ЛАНЬ®

Часть 1

Введение

Со школы каждому знакомо интуитивное определение алгоритма — «набор инструкций, описывающих порядок действий исполнителя для достижения некоторого результата». Считается, что алгоритм принимает на вход некоторый *конструктивный объект*, выполняет определенную последовательность действий и возвращает (если останавливается) новый конструктивный объект. Конструктивный объект — тот, для которого существует конструктивный способ задать его, например, натуральное число, матрица или граф.

Такое понятие алгоритма довольно расплывчато и не позволяет конкретно ответить на такие вопросы:

- *любая ли задача разрешима алгоритмически?*
- *если некоторая задача не разрешается никаким алгоритмом, как это доказать?* Иными словами, как показать, что любой алгоритм разрешает задачу, отличную от данной?
- *как вычислить время, совершенных алгоритмом, или требуемую память?* Нельзя вычислить время работы алгоритма, просто рассмотрев некоторую его реализацию (например, на языке C), и сложив количества выполений каждой операции. Команды не унитарны ни по времени, ни по ресурсоемкости, время выполнения одной и той же команды может быть разным. Лучшее, что можно сделать в данном случае — оценить (если возможно) время работы команды (в худшем случае) сверху некоторой константой. Но тут важно знать, какие операции мы считаем элементарными и у каких операций время можно оценить сверху константой (в частности, сложение чисел нельзя a priori считать операцией, выполянемой за не более чем постоянное время — если мы не считаем, например, что сами слагаемые ограничены сверху).
- *как показать, что данная задача не может быть решена точно за относительно малое время?* Само по себе существование некоторого алгоритма не гарантирует, что нельзя построить более быстрый алгоритм, разрешающий ту же задачу.

Четкие ответы на эти вопросы требуют наличия логического понятия алгоритма. В этой книге мы дадим точное понятие алгоритма в разделе, посвященном теории сложности. Для практических нужд обычно достаточно интуитивного определения алгоритма; каждый раз мы будем оговаривать, что является единицей памяти и какие операции *элементарны*, то есть выполняются за не более чем постоянное время.

Большинство алгоритмов, с которыми мы будем иметь дело, являются *детерминированными*, то есть шаги, которые они делают, а так же результат, который они выдают, зависят только от входа алгоритма. Однако в ряде случаев можно использовать *вероятностные* процедуры, вместо детерминированных. Делается это прежде всего для того, чтобы уменьшить время работы алгоритма. Конечно, за это приходится платить тем, что

решение задачи получается неточным или правильный ответ даётся с некоторой вероятностью. Часто бывает, что приходится делать больше итераций, но стоимость одной итерации становится гораздо «дешевле». Стоит отметить, что вероятностные алгоритмы важны не только с теоретической точки зрения, но и с практической, так как некоторые вероятностные процедуры очень хорошо себя зарекомендовали на практике. Например, некоторые люди, которые занимаются глубоким обучением (Deep Learning), под словами «градиентный спуск» (что бы это не значило) часто понимают его стохастическую версию, ввиду её высокой практической ценности. Так что полезно буквально с первых шагов привыкнуть к языку теории вероятностей, а при разработке и анализе алгоритмов к случайности следует относиться как к потенциально важному ресурсу. Но осознать, в чем, собственно, этот ресурс заключается, можно только решая задачи.

Как оценивать время вероятностных процедур? Существует 2 основных подхода.

(1) Говорят, что алгоритм на входе размера n работает *в худшем* случае время $T(n)$, если на любом входе размера n он работает время $\leqslant T(n)$, причём равенство достигается.
(2) Говорят, что алгоритм на входе размера n работает *в среднем* время $T(n)$, если математическое ожидание времени его работы на входе размера n равняется $T(n)$.

Рассмотрим следующую задачу. На вход подаётся массив из $2N$ букв, причём половину из них составляют буквы a, а другую половину — буквы b. Требуется найти номер любой ячейки, в которой лежит буква a. Рассмотрим 2 идеологии вероятностного решения такой задачи.

Алгоритмом типа Лас-Вегас называется вероятностный алгоритм, который всегда на выходе даёт корректный результат. Рассмотрим для примера следующий алгоритм.

Require: Массив букв A размера N
1: **repeat**
2: Случайно равновероятно выбрать индекс $i \in \{1, 2, \ldots, N\}$
3: **until** $A[i] \neq a$
4: **return** i

Время работы в худшем случае этого алгоритма равно бесконечности. Действительно, есть ненулевая вероятность, что на каждом шаге алгоритм будет выбирать ячейку, в которой лежит буква b, а значит, мы не можем ограничить количество итераций. Однако алгоритм останавливается за конечное число шагов с вероятностью 1. Более того, среднее число выполненных итераций равно

$$A \stackrel{\text{def}}{=} \sum_{i=1}^{\infty} \frac{i}{2^i} = \sum_{i=1}^{\infty} \frac{i+1}{2^i} - \sum_{i=1}^{\infty} \frac{1}{2^i} = 2\left(\sum_{i=1}^{\infty} \frac{i}{2^i} - \frac{1}{2}\right) - 1 = 2\left(A - \frac{1}{2}\right) - 1 = 2A - 2 \Rightarrow A = 2,$$

то есть алгоритм *в среднем делает 2 итерации.*

Алгоритмом типа Монте-Карло называется вероятностный алгоритм, который может давать на выходе неправильный результат с некоторой (как правило маленькой) вероятностью. Рассмотрим для примера следующий алгоритм.

Require: Массив букв A размера N; натуральное число k
1: $i := 0$
2: **repeat**
3: Случайно равновероятно выбрать индекс $i \in \{1, 2, \ldots, N\}$
4: **until** $k = i$ или $A[i] \neq a$
5: **return** i

Время работы в худшем случае этого алгоритма равно k. Вероятность получить правильный ответ равна $1 - \frac{1}{2^k}$. Более того, среднее число выполненных итераций равно

$$B \stackrel{\text{def}}{=} \sum_{i=1}^{k} \frac{i}{2^i} = \sum_{i=1}^{k} \frac{i+1}{2^i} - \sum_{i=1}^{k} \frac{1}{2^i} = 2\left(\sum_{i=1}^{k} \frac{i}{2^i} - \frac{1}{2} - \frac{k}{2^k}\right) - 1 = 2\left(B - \frac{1}{2} + \frac{k+1}{2^{k+1}}\right) - 1,$$

следовательно,

$$B = 2 - \frac{k+1}{2^{k+1}},$$

то есть алгоритм *в среднем делает меньше 2-х итераций.*

Псевдослучайные числа. Отдельного внимания заслуживает вопрос эмуляции генератора случайных чисел на практике. Компьютер — детерминированная арифметическая машина, и любой алгоритм на нем даст некоторую определенную, не случайную, последовательность чисел. Лучшее, что можно сделать — сгенерировать последовательность чисел, которая внешне «похожа» на случайную. Отдельная проблема — дать формальное определение псевдослучайной числовой последовательности: нужно определиться с тем, какое мерило случайности мы используем.

Определение 1.1. Пусть $(x_n)_{n\in\mathbb{N}}$ — последовательность остатков по модулю m. Будем говорить, что $(x_n)_{n\in\mathbb{N}}$ *k-распределена*, если для любой последовательности $a_0, \ldots a_{k-1} \in \mathbb{Z}/m\mathbb{Z}$ и для любого $n \in \mathbb{N}$ выполнено

$$\mathbb{P}(x_n = a_0, x_{n+1} = a_1 \ldots x_{n+k-1} = a_{k-1}) = \frac{1}{m^k}$$

Последовательность $(x_n)_{n\in\mathbb{N}}$ называется *∞-распределенной*, если она k-распределена для любого натурального k.

Это интуитивное понятие псевдослучайной последовательности не совсем точно, подробности можно посмотреть в секции 3.5 книги [31].

Простейший способ генерировать псевдослучайные числа — использовать линейные реккуренты, то есть условия вида

$$f_n = a_1 f_{n-1} + \ldots + a_k f_{n-k} \bmod m$$

При некоторых значениях $a_1, \dots a_k$ и m полученные f_n действительно будут «случайными». Функция `rand` в C — реализация именно такого генератора случайных чисел.

Пример 1.2. Рассмотрим $f_n = F_n \bmod m$ — числа Фибоначчи по модулю некоторого натурального m. Нетрудно заметить, что эта последовательность периодична: поскольку $f_n = (f_{n-1} + f_{n-2}) \bmod m$, то различных f_n не больше, чем возможных пар остатков по модулю m, то есть не более, чем m^2. На самом деле верна более точная оценка *периода Пизано* $\pi(m)$ — периода последовательности $F_n \bmod m$: можно убедиться в том, что $\pi(n) \leqslant 6n$.

Для $m = 2$, например, последовательность $(f_n)_{n\in\mathbb{N}}$ есть просто $0, 1, 1, 0, 1, 1, \dots$, период имеет длину 3. Для $m = 11$ получим

$$0, 1, 1, 2, 3, 5, 8, 2, 10, 1, 0, 1, 1, \dots,$$

период имеет длину 10, а при $m = 10$ последовательность имеет вид

$$0, 1, 1, 2, 3, 5, 8, 3, 1, 4, 5, 9, 4, 3, 7, 0, 7, 7, 4, 1, 5, 6, 1, 7, 8, 5, 3, 8, 1, 9, 0, 9, 9, 8, 7, 5, 2, 7, 9, 6, 5, 1, 6,$$
$$7, 3, 0, 3, 3, 6, 9, 5, 4, 9, 3, 2, 5, 7, 2, 9, 1, 0, 1, \dots$$

то есть период имеет длину 60.

Еще одно важное требование к генератору псевдослучайных чисел — криптоустойчивость, то есть устойчивость к предсказанию механизма этого генератора. Генератор псевдослучайных чисел, генерирующий последовательность по формуле $f_n = af_{n-1} + b \bmod m$ для взаимнопростых a и m, не является криптоустойчивым: поскольку

$$f_{n+1} - f_n = a(f_n - f_{n-1}) \bmod m, \quad f_{n+2} - f_{n+1} = a(f_{n+1} - f_n) \bmod m,$$

то

$$m | (f_{n+1} - f_n)^2 - (f_n - f_{n-1})(f_{n+2} - f_{n+1})$$

Далее можно построить алгоритм, угадывающий a, b и m. По формуле выше можно угадать m, а затем подобрать некоторым образом a и b, а затем далее запустить генератор. Вполне возможно, что сами a, b и m не будут угаданы, но полученный генератор будет угадывать ту же самую последовательность. Бояр (см. [5]) построила полиномиальный алгоритм, позволяющий таким образом взламывать генератор.

В наши дни наиболее популярным генератором псевдослучайных чисел является *твистер Мерсенна*.

Короче говоря, как говорил Ричард Ковейю (Richard Coveyu): «Генерирование псевдослучайных чисел слишком важная вещь, чтобы предоставить ее случаю». Подробнее о псевдослучайных числах можно почитать в [31] и [12].

Асимптотические оценки. Метод Акра-Баззи

При теоретическом анализе сложности алгоритма, как правило, нас не интересуют конкретные константы в функциях, а только их асимптотики при длине входа $|x| \to \infty$.

- Функция натурального аргумента $g(n) = O\big(f(n)\big)$, если существует такая константа $C > 0$ и число n_0, что для всех $n \geq n_0$ выполняется $|g(n)| \leq C|f(n)|$.
- $g(n) = o\big(f(n)\big)$, если $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.
- $g(n) = \Omega\big(f(n)\big)$, если $f(n) = O\big(g(n)\big)$.
- $g(n) = \omega\big(f(n)\big)$, если $f(n) = o\big(g(n)\big)$.
- $g(n) = \Theta\big(f(n)\big)$, если $g(n) = O\big(f(n)\big)$ и $f(n) = O\big(g(n)\big)$.
- $g(n) = \widetilde{O}\big(f(n)\big)$, если $f(n) = O\big(g(n)\log^k g(n)\big)$ для некоторого $k \in \mathbb{N}$.

В примере выше мы столкнулись с реккурентным неравеством. Забегая слегка вперед, отметим, что нам и впоследствии попадутся реккурентные уравнения и неравенства. Как правило, реккурентные уравнения будут появляться при оценке сложности процедур, в ходе которых они сами вызываются реккурентно (как, например, стандартная процедура вычисления факториала). Поэтому сейчас мы разберем некоторые способы решения реккурентных уравнений, включая теорему Акра-Баззи.

Самый простой способ — построить *дерево рекурсии*. Рассмотрим дерево, каждая вершина которого соответствует вызову экземпляра функции. Внутренние вершины помечены числом операций, которые потребовались для обработки результатов следующих рекурсивно вызванных функций (в нашем случае — для слияния двух массивов), а листы — числом операций на минимальной подзадаче, для которой уже не использовалась рекурсия. Кроме того, каждой вершине соответствует уровень, или глубина, n — размер задачи, для которой была вызвана эта функция. Очевидно, время решения задачи (в худшем случае), зависит только от ее размера, поэтому все вершины на одном уровне эквивалентны. Суммарное время работы программы складывается из времени работы в каждой вершине.

Пример 1.3. Рассмотрим реккуренту

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn,\ n \geq 2, \tag{1}$$

$$T(1) = C. \tag{2}$$

ПИКЧА Будем для удобства считать, что $n = 2^k$ — как мы увидим позже, это не помешает правильно оценить сложность. Нас интересует асимптотическая оценка $T(n)$. Можно циклически подставлять рекуррентное соотношение, пока не дойдем до нижнего уровня

$(n = 1)$:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn = 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn = 4T\left(\frac{n}{4}\right) + 2cn = \dots \tag{3}$$
$$\dots = 2^k T\left(\frac{n}{2^k}\right) + kcn = 2^k T(1) + cnk = Cn + cn\log n = \Theta(n\log n).$$

Покажем, что это верно для любых n, не только степеней двойки: очевидно, $T(n)$ неубывающая функция, так что

$$T(n) = O\left(2^{\lceil\log n\rceil}\lceil\log n\rceil\right) = O\left(2^{\log n+1}(\log n + 1)\right) = O(n\log n), \tag{4}$$

$$T(n) = \Omega\left(2^{\lfloor\log n\rfloor}\lfloor\log n\rfloor\right) = \Omega\left(2^{\log n-1}(\log n - 1)\right) = \Omega(n\log n). \tag{5}$$

Таким образом, $T(n) = \Theta(n\log n)$ для всех n.

При анализе сложности алгоритмов нам будут встречаться рекурренты вида

$$T(n) = \sum_{i=1}^{k} a_i T(b_i n) + f(n).$$

Оказывается, существует общий метод, позволяющий во многих случаях найти аналитическое выражение для асимптотики $T(n)$.

Теорема 1.4 (Akra–Bazzi). *Пусть дано следующее рекуррентное уравнение:*

$$T(x) = \begin{cases} \Theta(1), & 1 \le x \le x_0, \\ \sum_{i=1}^{k} a_i T\big(b_i x + h_i(x)\big) + g(x), & x > x_0, \end{cases}$$

где $x \in [1, +\infty)$; $a_i > 0$, $0 < b_i < 1$ — константы;

$$h_i(x) = O\left(\frac{x}{\log^2 x}\right);$$

$g(\cdot)$ — функция полиномиального роста, т.е. для любого $x \ge 1$, $1 \le i \le k$ и $u \in [b_i x, x]$ выполняется

$$c_1 g(x) \le g(u) \le c_2 g(x),$$

где $c_1, c_2 > 0$ — некоторые константы; x_0 — некоторое достаточно большое число.

Пусть $p \in \mathbb{Q}$ — решение уравнения $\sum_{i=1}^{k} a_i b_i^p = 1$. Тогда $T(x)$ имеет следующую асимптотику:

$$T(x) = \Theta\left(x^p\left(1 + \int_1^x \frac{g(u)}{u^{p+1}}du\right)\right).$$

Идея доказательства. Угадав асимптотику, можно ее проверить по индукции, используя все условия теоремы. Но как угадать вид асимптотики? Для этого рассмотрим упрощенное уравнение:

$$T(x) = \sum_{i=1}^{k} a_i T(b_i x) + g(x),\ x \geq 1.$$

Зафиксируем x. Сначала оценим "плотность числа вершин" $f(t)$ в дереве рекурсии на уровне $\frac{x}{e^t}$ (в логарифмическом масштабе), т.е. между уровнями $\frac{x}{e^{t+\delta}}$ и $\frac{x}{e^{t-\delta}}$ находится примерно $2\delta e^{f(t)}$ вершин. Предположим, что эта функция гладкая. Тогда для любого $0 < t < \ln x$ получаем

$$e^{f(t)} \approx \sum_{i=1}^{k} a_i \exp\big(f(t + \ln b_i)\big) \approx \sum_{i=1}^{k} a_i \exp\big(f(t) + f'(t) \ln b_i)\big) = e^{f(t)} \left(\sum_{i=1}^{k} a_i b_i^{f'(t)} \right).$$

Следовательно, $\sum_{i=1}^{k} a_i b_i^{f'(t)} \equiv 1$, поэтому $f'(t) \equiv p$ и $f(t) = pt$.

Таким образом, $e^{f(\ln x)} = e^{p \ln x} = x^p$ — число минимальных подзадач. Вспомогательные вычисления занимают примерно

$$\int_0^{\ln x} e^{f(t)} g\left(\frac{x}{e^t}\right) dt = \int_1^x \left(\frac{x}{u}\right)^p g(u) \frac{du}{u} = x^p \int_1^x \frac{g(u)}{u^{p+1}} du.$$

В итоге получаем то, что нужно:

$$T(x) = \Theta\left(x^p + x^p \int_1^x \frac{g(u)}{u^{p+1}} du \right).$$

□

Добавочные члены $h_i(\cdot)$ в условии теоремы позволяют учитывать, например, округление чисел, прибавление константы и так далее. Заметим, что если заменить все равенства на ограничения сверху, то есть рассматривать рекурренту $T(x) \leq \sum_{i=1}^{k} a_i T\big(b_i x + h_i(x)\big) + O\big(g(x)\big)$, то

$$T(x) = O\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du \right) \right).$$

Пример 1.5. Пусть дано рекуррентное уравнение

$$T(n) = 2T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + T\left(\left\lceil \frac{n}{3} \right\rceil\right) + \frac{1}{3} T\left(\left\lfloor \frac{4n}{5} \right\rfloor\right) + \Theta\left(n \log^2 n\right).$$

(Откуда мог взяться множитель $\frac{1}{3}$? Например, мы рассматривали *вероятностный* алгоритм, и с вероятность $\frac{2}{3}$ соответствующей подзадачи не возникает.) Т.к.

$$2\frac{1}{5} + \frac{1}{3} + \frac{1}{3}\frac{4}{5} = 1,$$

то $p = 1$. Тогда

$$\int_1^n \frac{u \ln^2 u}{u^2} du = \int_1^n \ln^2 u d \ln u = \frac{\ln^3 n}{3} = \Theta\left(\log^3 n\right).$$

Следовательно, $T(n) = \Theta\left(n + n \log^3 n\right) = \Theta\left(n \log^3 n\right)$.

Амортизационный анализ. Напоследок рассмотрим еще одно средство анализа алгоритмов, производящих последовательность однотипных операций — *амортизационный анализ*. При амортизационном анализе каждой операции присваивается некоторая учётная стоимость (amortized cost), которая может быть больше или меньше реальной длительности операции. При этом должно выполняться следующее условие: для любой последовательности операций фактическая суммарная длительность всех операций (предполагается, что до выполнения операций структура данных находится в начальном состоянии — например, стек пуст) не превосходит суммы их учётных стоимостей. Если это условие выполнено, то говорят, что учётные стоимости присвоены корректно. Оценка, даваемая амортизационным анализом, не является вероятностной: это оценка среднего времени выполнения одной операции для худшего случая.

Рассмотрим следующий пример амортизационного анализа. Рассматривается структура данных, состоящая из

- числа $n \in \mathbb{N}$ в двоичной записи ($\lceil \log_2 n \rceil$ битов),
- операции $Add : \mathbb{N} \to \mathbb{N}$, $Add(n) = n + 1$.

При инициализации $n = 1$, всякий вызов Add увеличивает хранимое число на 1. Так как

$$\underbrace{1 \ldots 1}_{k \text{ нулей}} + 1 = 1 \underbrace{0 \ldots 0}_{k \text{ нулей}},$$

то Add совершает в худшем случае $\Theta(\log n)$ побитовых операций.

Утверждение 1.6. *Рассмотрим процедуру, совершающую большое число N запросов к процедуре Add. Суммарное время работы всех выполнений процедуры оценивается как $\Theta(N)$ (а не $\Theta(N \log N)$).*

Доказательство. Нужно честно просуммировать сложность всех применений Add и оценить ее как $\Theta(N)$. Для начала заметим, что все числа от 1 до N можно разбить в дизъюнктное объединение множеств A_k чисел, оканчивающихся на ровно k единиц (возможно, ноль); иными словами,

$$[1; N] = \bigsqcup_{k=0}^{N} A_k, \quad A_k = \{1 \ldots 0 \underbrace{1 \ldots 1}_{k \text{ единиц}}\}.$$

Теперь заметим, что для любого $x \in A_k$ сложность операции *Add* равна $k+1$ битовых операций:

$$\underbrace{1\ldots1}_{k \text{ нулей}} + 1 = 1\,\underbrace{0\ldots0}_{k \text{ нулей}},$$

тогда для получения N надо заменить последние $k+1$ цифру на противоположную, 0 на 1 и наоборот. Осталось только честно просуммировать сложности каждой операции *Add*, примененной N раз, то есть $T(N)$. Очевидно, $T(N) \leqslant T(2^{\lceil \log_2 N \rceil})$, а $T(2^n)$ можно честно вычислить. Заметим, что

$$|A_k| = \begin{cases} 2^{n-k-1}, & k < n \\ 1, & k = n \end{cases},$$

ведь при $k \neq 0$ все числа A_k можно представлять как конкатенацию последовательности 0 и 1 длины $n-k$ и слова $01\ldots1$ [в старших разрядах могут стоять нули — у нас получится просто число, меньше некоторой степени], а слов, заканчивающихся на 0, ровно половина среди положительных не больших 2^n чисел. Тогда

$$T(2^n) = \sum_{k=1}^{n}(k+1)|A_k| = n + \sum_{k=1}^{n-1}(k+1)2^{n-k-1} = n + 2^{n-1}\sum_{k=1}^{n-1}\frac{k+1}{2^k} \leqslant$$

$$\leqslant n + 2^{n-1}\sum_{k=1}^{\infty}\frac{k+1}{2^k} = n + 2^{n-1}\left(\left(\frac{1}{1-z}\right)' - 1 - z\right)\Big|_{z=\frac{1}{2}} = n + \frac{5}{2}\cdot 2^{n-1},$$

и, таким образом

$$T(N) \leqslant T(2^{\lceil \log_2 N \rceil}) \leqslant \log_2(N) + 3N.$$

Аналогичную оценку снизу сделать совсем просто: достаточно лишь сказать, что $T(N) \geqslant T(2^{\lfloor \log_2 N \rfloor})$ и оценить $T(2^{\lfloor \log_2 N \rfloor})$ снизу как $\Theta(N)$, что совершенно очевидно. Тогда $T(N) = \Theta(N)$. □

В прошлой задаче мы могли сказать, что среднее, или *амортизированное* время работы операции *Add* есть $\Theta(1)$. Теперь рассмотрим еще одну задачу, решающуюся с помощью амортизационного анализа. Мы собираемся получить нижнюю оценку на минимальное число $T_{\min}$ попарных сравнений, которое необходимо сделать для нахождения минимального элемента n-элементного массива.

Утверждение 1.7. *Число шагов любого устойчивого (не зависящего от значений элементов входного массива) алгоритма нахождения минимального элемента $A[1\,..\,n]$ не меньше $n-1$.*

Доказательство. Запишем шаги алгоритма в формате конфигураций (a, b, c), где

- a элементов пока не сравнивались,

- b элементов никогда не были больше в сравнениях,
- c элементов хотя бы раз оказались больше.

Начальная конфигурация суть $Init = (n, 0, 0)$. Введем «потенциальную функцию» на конфигурациях: $f[(a, b, c)] = a + b$. Покажем, что при любом сравнении потенциал $f(\cdot)$ может уменьшиться не больше, чем на единицу, отсюда вытекает, что число шагов любого такого алгоритма не меньше $n - 1$. Алгоритм находит минимальный элемент массива, базируясь на результатах попарных сравнений элементов. Продемонстрируем, как могут меняться параметры a, b, c и значение функции f. Мы оценим трудоемкость алгоритма, просто поделив «разность потенциалов» между начальной и конечной конфигурациями на максимальное изменения потенциала за один шаг алгоритма. Для доказательства утверждения задачи нужно аккуратно разобрать шесть случаев [в зависимости от того, как сравниваемые на данном шаге x и y участвовали в предыдущих сравнениях].

- Если оба элемента не участвовали в сравнении, то либо один из них в предстоящем сравнении станет больше, а другой — меньше, тогда значение f уменьшится на один — a уменьшилось на два, а b увеличилось на один, либо оба элемента будут равны, то есть b увеличилось на два, а a уменьшилось на два, тогда f не изменилась;
- Если x не участвовал в сравнении, а y хотя бы однажды оказался больше, то либо x окажется меньше, что не изменит потенциал f [a уменьшилось на 1, c увеличилось на 1], либо y, что уменьшит потенциал на 1 [a уменьшилось на 1, b не изменилось];
- Если x не участвовал в сравнении, а y во всех сравнениях был не больше, то либо x окажется меньше, что уменьшит значение f на один [a уменьшилось на 1, c не изменилось — выкинули y из множества «всегда меньших элементов» и добавили x], либо y, что уменьшит потенциал на 1 [a уменьшилось на 1, b не изменилось];
- Если x был во всех предыдущих сравнениях не больше, а y уже участвовал в сравнениях, то x будет либо больше [и тогда b уменьшится на 1], либо будет меньше или равен [тогда b не изменится], значение f уменьшается не более чем на один;
- Если оба элемента были больше хотя бы в одном сравнениии, то независимо от итога сравнения a и b не поменяются, ведь в сравнении не участвуют «всегда меньшие» или неиспользованные элементы и новых таких появится, значение f не изменилось.

Все случаи разобраны, осталось только вывести, что сравнений должно быть хотя бы $n - 1$. Это тривиально: $a + b$ уменьшается с каждым сравнением не более чем на единицу, а алгоритм сможет выдать верный ответ, только лишь просмотрев все элементы [в противном случае можно построить пару массивов, на которых алгоритм делает сравнения тех же (по номеру) пар элементов и выдает в одном случае верный, а в другом

— неверный ответ; подробнее об этом — в пункте **(b)** этой же задачи]. Тогда a должно стать равным 0, а b — не большим 1 [все элементы массива просмотрены, единственный минимальный из них найден либо не существует]. На последнем шаге $f[(0,b,c)] \leqslant 1$, и алгоритм должен проделать хотя бы $n-1$ сравнение. □

Линейные рекурренты

Определение 1.8. Будем говорить, что последовательность $(X_n)_{n\in\mathbb{N}}$ задана *линейным рекуррентным уравнением*

$$X_n = a_1 X_{n-1} + a_2 X_{n-2} + \cdots + a_k X_{n-k},$$

если даны $X_0, \dots X_{k-1}$, а формула выше верна для всех $n \geqslant k$. *Характеристическим уравнением* такой линейной реккуренты будем называть уравнение

$$\lambda^k - a_1\lambda^{k-1} - \cdots - a_k = 0,$$

где λ — переменная.

Пусть $\lambda_1, \dots, \lambda_k$ — корни характеристического уравнения. Оказывается, они тесно связаны с пространством решений линейной рекурренты, а именно: если они все различны, то общее решение рекурренты имеет вид

$$X_n = C_1\lambda_1^n + C_2\lambda_2^n + \cdots + C_k\lambda_k^n.$$

Константы $C_1, \dots, C_n$ находятся из начальных условий $X_0, \dots, X_{k-1}$.

Если же какой-то корень λ_j имеет кратность $s_j > 1$, то в общее решение входит слагаемое вида

$$C_{j,0}\lambda_j^n + C_{j,1}n\lambda_j^n + \cdots + C_{j,s_j-1}n^{s_j-1}\lambda_j^n.$$

Несложно проверить (например, по индукции), что такие последовательности действительно удовлетворяют данному рекуррентному уравнению. Например, можно ввести «повышающий оператор» $\square X_n := X_{n+1}$ (он действует, вообще говоря, на всю последовательность $\{X_n\}_{n\in\mathbb{N}}$). Тогда рекуррентное уравнение записывается в виде

$$\square^k X_n - a_1\square^{k-1}X_n - \cdots - a_k X_n = \prod_{i=1}^{k}(\square - \lambda_i)X_n = 0.$$

Если λ_j имеет кратность s_j, то по индукции устанавливается, что для $0 \leqslant l \leqslant s_j - 1$ выполняется

$$(6)\quad (\square-\lambda_j)^{l+1}n^l\lambda_j^n = (\square-\lambda_j)^l\big((n+1)^l\lambda_j^{n+1} - n^l\lambda_j^{n+1}\big) =$$
$$= (\square-\lambda_j)^l\sum_{i=0}^{l-1}C_l^i n^i\lambda_j^{n+1} = \lambda_j\sum_{i=0}^{l-1}C_l^i(\square-\lambda_j)^{l-i-1}\big((\square-\lambda_j)^{i+1}n^i\lambda_j^n\big) = 0.$$

Почему любое решение имеет такой вид? Заметим, что пространство решений k-мерное: они полностью определяются начальными условиями, т.е. k числами $X_0, \ldots, X_{k-1}$. Можно рассмотреть последовательности X_n^j, удовлетворяющие рекуррентному уравнению и $X_i^j = \delta_{ij}$, $0 \leqslant i < k$. Они образуют базис в пространстве решений. С другой стороны, решения вида $n^l \lambda_j^n$ линейно независимы, т.к. они имеют разный порядок роста. Сумма кратностей всех корней равна k, поэтому эти решения тоже образуют базис, т.е. любое решение можно представить в виде

$$X_n = \sum_j \sum_{l=0}^{s_j - 1} C_{j,l} n^l \lambda_j^n,$$

что и требовалось.

Пример 1.9. Рассмотрим числа Фибоначчи: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$. Характеристическое уравнение для этой рекурренты:

$$\lambda^2 - \lambda - 1 = 0.$$

Его корни $\lambda_1 = \frac{1-\sqrt{5}}{2}$, $\lambda_2 = \frac{1+\sqrt{5}}{2}$. Следовательно, общее решение имеет вид

$$F_n = C_1 \left(\frac{1-\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1+\sqrt{5}}{2}\right)^n.$$

Найдем константы C_1 и C_2:

$$F_0 = C_1 + C_2 = 0, \tag{7}$$

$$F_1 = C_1 \frac{1-\sqrt{5}}{2} + C_2 \frac{1+\sqrt{5}}{2} = 1. \tag{8}$$

Следовательно, $C_2 = -C_1 = \frac{1}{\sqrt{5}}$. Таким образом, числа Фибоначчи равны

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right).$$

До этого мы рассматривали однородное линейное рекуррентное уравнение. Теперь рассмотрим *неоднородное*:

$$X_n = a_1 X_{n-1} + a_2 X_{n-2} + \cdots + a_k X_{n-k} + f(n).$$

Его общее решение имеет вид

$$X_n = X_n^{\text{о.о.}} + X_n^{\text{ч.н.}},$$

где $X_n^{\text{о.о.}}$ — общее решение соответствующего однородного уравнения, а $X_n^{\text{ч.н.}}$ — некоторое частное решение неоднородного.

Пример 1.10. Рассмотрим числа Фибоначчи, но с дополнительным слагаемым: $\tilde{F}_0 = 0$, $\tilde{F}_1 = 1$, $\tilde{F}_n = \tilde{F}_{n-1} + \tilde{F}_{n-2} + n$, $n \geq 2$. Общее решение однородного уравнения, как мы знаем, имеет вид

$$F_n = C_1 \left(\frac{1-\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1+\sqrt{5}}{2}\right)^n.$$

Частное решение неоднородного можно угадать. Т.к. $f(n)$ линейна, то решение будем искать тоже линейное: $F_n = an + b$. Тогда

$$an + b = a(n-1) + b + a(n-2) + b + n = (2a+1)n + 2b - 3a.$$

Следовательно, $a = -1$, $b = 3a = -3$, и решение имеет вид

$$\tilde{F}_n = C_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1-\sqrt{5}}{2}\right)^n - an - 3.$$

Константы C_1 и C_2 находятся из системы уравнений:

$$\tilde{F}_0 = C_1 + C_2 - 3 = 0, \tag{9}$$

$$\tilde{F}_1 = C_1 \frac{1-\sqrt{5}}{2} + C_2 \frac{1+\sqrt{5}}{2} - 4 = 1. \tag{10}$$

Вероятность: введение

На всякий случай здесь приведено краткое напоминание дискретной версии теории вероятности. Для более подробного ознакомления советуем книги [38, 16, 22], а также задачник [14].

Определение 1.11. *Вероятностным пространством* называется множество Ω, элементы которого называются возможными или элементарными исходами $\omega \in \Omega$. На вероятностном пространстве задана функция $\mathbb{P} : \Omega \to [0,1]$, называемая *вероятностным распределением*, для которой выполнено равенство $\sum\limits_{\omega\in\Omega} \mathbb{P}(\omega) = 1$. Число $\mathbb{P}(\omega)$ понимается как *вероятность* ω.

Определение 1.12. *Событием* называется произвольное подмножество Ω. Соответственно, вероятность события $A \subseteq \Omega$ определяется формулой $\mathbb{P}(A) \stackrel{def}{=} \sum_{\omega\in A} \mathbb{P}(\omega)$.

В качестве простейшего примера можно рассмотреть классическую, или наивную, теорию вероятности, когда все исходы полагаются равновероятными: например, при бросании симметричной монетки (то есть вероятность любого исхода есть $\frac{1}{2}$) или игральной кости (выпадает число от 1 до 6, вероятность каждого исхода есть $\frac{1}{6}$).

Определение 1.13. События A и B называют *независимыми*, если

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B).$$

События $A_1, \ldots, A_n$ называются *попарно независимыми*, если любые два из них являются независимыми. События $A_1, \ldots, A_n$ называются *независимыми в совокупности*, если для любого набора из этих событий $A_{i_1}, \ldots, A_{i_k}$ выполняется

$$\mathbb{P}(A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}) = \mathbb{P}(A_{i_1})\mathbb{P}(A_{i_2}) \ldots \mathbb{P}(A_{i_k}).$$

Определение 1.14. *Условная вероятность* $\mathbb{P}(A \mid B)$ события A при условии B определяется из формулы

$$\mathbb{P}(A \cap B) = \mathbb{P}(A \mid B) \cdot \mathbb{P}(B).$$

В частности, если $\mathbb{P}(B) > 0$, то справедливо равенство

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}.$$

Теорема 1.15 (Bayes). *Если* $\mathbb{P}(A) > 0$ *и* $\mathbb{P}(B) > 0$, *то*

$$\mathbb{P}(A \mid B) = \mathbb{P}(A) \cdot \frac{\mathbb{P}(B \mid A)}{\mathbb{P}(B)}.$$

Лемма 1.16. *Если события $B_1, \ldots, B_n$ образуют разбиение вероятностного пространства Ω, то есть $\Omega = B_1 \sqcup \cdots \sqcup B_n$ и пересечения попарно пусты: $B_i \cap B_j = \emptyset$, $i \neq j$, то для любого события A выполнена формула полной вероятности:*

$$\mathbb{P}(A) = \sum_{i=1}^{n} \mathbb{P}(A \mid B_i) \cdot \mathbb{P}(B_i).$$

Определение 1.17. *Случайная величина* $\xi : \Omega \to \mathbb{R}$ — это числовая функция на вероятностном пространстве. Иначе говоря, случайная величина — это обычная числовая функция, принимающая некоторое значение с некоторой вероятностью.

Определение 1.18. *Математическим ожиданием* случайной величины ξ называется ее средневзвешенное значение

$$\mathbb{E}\xi \stackrel{def}{=} \sum_{\omega \in \Omega} \xi(\omega) \cdot \mathbb{P}(\omega).$$

Математическое ожидание линейной комбинации случайных величин равно соответствующей линейной комбинации их матожиданий:

$$\mathbb{E}(a\xi + b\psi) = a\mathbb{E}(\xi) + b\mathbb{E}(\psi).$$

Пример 1.19. Рассмотрим *модель Эрдёша-Реньи* случайного графа на n вершинах — то есть случайный граф, в котором ребра между каждой парой вершин независимо генерируются с вероятностью p. Найдем математическое ожидание числа простых циклов длины r в случайном графе $G = (V, E)$ на n вершинах. Воспользуемся линейностью матожидания. Пусть $S \subset V$, $|S| = r$ — подмножество вершин, определим

$$\mathbb{I}(S, G) = \begin{cases} 1, & \text{через вершины } S \text{ проходит простой цикл} \\ 0, & \text{иначе} \end{cases}$$

На вероятностном пространстве графов на n вершинах случайные величины $\mathbb{I}(S, G)$ не обязательно независимы. Однако если $N_r(G)$ — число простых циклов длины r, то

$$N_r(G) = \sum_{S \subset V(G)} \mathbb{I}(S, G);$$

через каждое множество вершин мощности r проходит не более одного простого цикла, в противном случае один из циклов не будет простым. Действительно, пусть в первом цикле вершины занумерованы в некотором порядке $(1,\ 2\ \ldots r)$, другой цикл содержит некоторое ребро (ij) для некоторого $j \neq i+1$, тогда цикл $(i,\ j,\ j+1,\ \ldots,\ i-1)$ имеет длину не более $r-1$.

Тогда из только что доказанной формулы следует

$$\mathbb{E}N_r(G) = \sum_{S \subset V(G)} \mathbb{E}\mathbb{I}(S, G) = \binom{n}{r} \mathbb{E}_{Cyc},$$

где $\mathbb{E}_{Cyc}$ — матожидание одного простого цикла длины r (нетрудно видеть, что все слагаемые $\mathbb{EI}(S,G)$ равны). Нетрудно также видеть, что $\mathbb{E}_{Cyc}$ есть просто вероятность возникновения простого цикла длины r. А именно, должны возникнуть r ребер, $\frac{r(r-3)}{2}$ диагонали должны отсутствовать, а самих различных циклов всего $\frac{(r-1)!}{2}$ (они должны быть инварианты относительно замены ориентации на обратную и циклического сдвига), тогда получаем ответ $\mathbb{E}_{Cyc} = \frac{(r-1)!}{2} p^r (1-p)^{\frac{r(r-3)}{2}}$ и

$$\mathbb{E}N_r(G) = \frac{n!}{2r(n-r)!} p^r (1-p)^{\frac{r(r-3)}{2}}$$

Лемма 1.20 (Вероятностный метод)**.** *Пусть для случайной величины ξ выполнено $\mathbb{E}\xi = A$, тогда существует исход ω, что $\xi(\omega) \geqslant A$ (равно как существует исход, при котором выполнено противоположное неравенство).*

Доказательство. Допустим обратное, то есть $\xi(\omega) < A$ для всех $\omega \in \Omega$, тогда

$$\mathbb{E}[\xi] = \sum_{\omega\in\Omega} \xi(\omega)\cdot\mathbb{P}(\omega) < A\sum_{\omega\in\Omega}\mathbb{P}(\omega) = A.$$

□

Вероятностный метод удобно применять в комбинаторных задачах.

Утверждение 1.21. *Для любого графа G существует такая раскраска вершин в черный и белый цвет, что не менее половины ребер окажутся разноцветными.*

Доказательство. Пусть в данном графе $G = (V, E)$ всего n вершин. Зададим на множестве Ω раскрасок графа G в два цвета равномерное распределение, то есть вероятность любой раскраски ω графа G будет одинаковой и равной $\frac{1}{2^n}$. Для каждого ребра (ij) зададим случайную величину

$$\mathbb{I}_{(ij)}(\omega) = \begin{cases} 1, & i \text{ и } j \text{ раскрашены в разные цвета} \\ 0, & \text{иначе} \end{cases}$$

Тогда если $\xi(\omega)$ — число разноцветных ребер в раскраске ω, то

$$\xi(\omega) = \sum_{(ij)\in E} \mathbb{I}_{(ij)}(\omega).$$

Теперь заметим, что если зафиксировать цвета двух смежных вершин, то все остальные вершины можно раскрасить 2^{n-2} способами. Таким образом получаем, что

$$\mathbb{E}[\mathbb{I}_{(ij)}] = \underbrace{0\cdot\frac{2^{n-2}}{2^n}}_{\text{ЧЧ}} + \underbrace{0\cdot\frac{2^{n-2}}{2^n}}_{\text{БЧ}} + \underbrace{0\cdot\frac{2^{n-2}}{2^n}}_{\text{ЧБ}} + \underbrace{0\cdot\frac{2^{n-2}}{2^n}}_{\text{ББ}} = \frac{1}{2}.$$

Отсюда получаем, что

$$\mathbb{E}[\xi] = \mathbb{E}\left[\sum_{(ij)\in E} \mathbb{I}_{(ij)}\right] = \sum_{(ij)\in E} \mathbb{E}[\mathbb{I}_{(ij)}] = |E| \cdot \frac{1}{2}.$$

Воспользуемся вероятностным методом и получим, что существует раскраска такая, что в ней не менее $\frac{|E|}{2}$ ребер разноцветные, что и требовалось доказать. □

Определение 1.22. Две случайные величины ξ и ψ *независимы*, если для любых чисел c и d события $\xi^{-1}(c) = \{\omega \in \Omega : \xi(\omega) = c\}$ и $\psi^{-1}(d)$ независимы.

Лемма 1.23. *Математическое ожидание произведения независимых случайных величин равно произведению их математический ожиданий:*

$$\mathbb{E}(\xi \cdot \psi) = \mathbb{E}\xi \cdot \mathbb{E}\psi.$$

Пример 1.24. Приведя явный контрпример, покажем, что из попарной независимости случайных величин не следует их независимость в совокупности. Пусть симметричная монета бросается 4 раза подряд, исходы $x_i \in \{0, 1\}$ для $i \in [1; 4]$; говорим, что $x_i = 0$, если выпал орел, $x_i = 1$, если выпала решка. Последовательность исходов будем обозначать соответствующим словом из нулей и единиц. Рассмотрим события

$$A = \{\sum_{i=1}^{4} x_i \vdots 2\},$$
$$B = \{x_1 = 1 \ \wedge x_2 = 1\},$$
$$C = \{x_3 = 1 \ \wedge x_4 = 1\}.$$

Заметим, что

$$\mathbb{P}(A) = \frac{\binom{4}{0} + \binom{4}{2} + \binom{4}{4}}{2^4} = \frac{1}{2},$$
$$\mathbb{P}(B) = \mathbb{P}(C) = \frac{2^2}{2^4} = \frac{1}{4}.$$

Далее,

$$\mathbb{P}(B \cap C) = \mathbb{P}(\{1111\}) = \frac{1}{16} = \frac{1}{4} \cdot \frac{1}{4} = \mathbb{P}(B)\mathbb{P}(C),$$
$$\mathbb{P}(A \cap B) = \mathbb{P}(\{1100, 1111\}) = \frac{2}{16} = \frac{1}{2} \cdot \frac{1}{4} = \mathbb{P}(A)\mathbb{P}(B),$$

и аналогично получим, что A и C независимы. Однако же

$$\mathbb{P}(A \cap B \cap C) = \mathbb{P}(\{1111\}) = \frac{1}{16} \neq \frac{1}{32} = \mathbb{P}(A)\mathbb{P}(B)\mathbb{P}(C)$$

Определение 1.25. *Дисперсией* $\mathbb{D}$ случайной величины называется средний квадрат ее отклонения от математического ожидания:

$$\mathbb{D}\xi = \mathbb{E}(\xi - \mathbb{E}\xi)^2.$$

Теорема 1.26 (Markov). *Для неотрицательной случайной величины* ξ *имеет место*

$$\mathbb{P}(\xi \geqslant a) \leqslant \frac{\mathbb{E}\xi}{a}.$$

Теорема 1.27 (Chebysheff). *Для неотрицательной случайной величины* ξ *имеет место*

$$\mathbb{P}(\mid \xi - \mathbb{E}\xi \mid \geqslant a) \leqslant \frac{\mathbb{D}\xi}{a^2}.$$

Теорема 1.28 (Chernoff). *Обозначим через* ξ_n *случайную величину, равную количеству выпавших орлов после* n *подбрасываний симметричной монеты, а через* $x_n = \frac{\xi_n}{n}$ *— частоту выпавших орлов. При больших* n *частота с очень большой вероятностью оказывается близкой к* $\frac{1}{2}$. *Имеет место следующая оценка:*

$$\mathbb{P}\left(\mid \xi_n - \frac{n}{2} \mid > \epsilon n\right) = \mathbb{P}(\mid x_n - \frac{1}{2} \mid > \epsilon) < 2\exp(-2\epsilon^2 n).$$

Имеют место еще несколько неравенств подобного рода, если монетка несимметричная. Пусть орел выпадает в вероятностью p. *Тогда*

$$\mathbb{P}(x_n < (1-\delta)p) < \exp\left(-\frac{\delta^2 p}{2}\right), \qquad 0 \leqslant \delta \leqslant 1,$$

$$\mathbb{P}(x_n > (1+\delta)p) < \exp\left(-\frac{\delta^2 p}{3}\right), \qquad 0 \leqslant \delta \leqslant 1,$$

$$\mathbb{P}(x_n > (1+\delta)p) < \exp\left(-\frac{\delta p}{3}\right), \qquad \delta > 1.$$


ЛАНЬ®


ЛАНЬ®

Часть 2

Сортировки и медианы

Сортировки

Перед тем, как рассматривать конкретные примеры алгоритмов сортировки и оценивать их время работы, зафиксируем постановку задачи сортировки, а также договоримся о том, как мы будем оценивать время работы алгоритма сортировки.

Задача сортировки формулируется следующим образом. На вход подаётся N элементов $a_1, a_2, \ldots, a_N$ (например, это может быть массив чисел, картинок, имён и так далее). Каждый элемент a_i состоит из двух частей: некоторой информации и ключа k_i (например, в случае массива чисел ключ просто совпадает с k_i; в случае массива картинок информацией может быть сама картинка, а ключом — некоторый id этой картинки или имя файла; когда мы имеем дело со списком имён, ключами опять-таки могут быть сами имена). На множестве ключей введено отношение порядка «$<$» так, что для любых трёх значений ключей x, y, z выполняются два свойства:

(1) закон трихотомии: либо $x < y$, либо $x > y$, либо $x = y$;
(2) закон транзитивности: если $x < y$ и $y < z$, то $x < z$.

Грубо говоря, мы можем сравнивать элементы массива при помощи сравнения ключей.

Задача сортировки состоит в том, чтобы на выходе получить масиив $a_{i_1}, a_{i_2}, \ldots, a_{i_N}$, у которого $k_{i_1} \leqslant k_{i_2} \leqslant \ldots \leqslant k_{i_N}$. При этом в абстрактной задаче сортировки сами ключи могут быть любой природы. Единственное, на что мы можем рассчитывать, так это на то, что мы можем сравнивать ключи. Поэтому принято оценивать временную сложность алгоритма сортировки числом попарных сравнений (считается, что время работы в таком случае пропорционально числу сравнений).

Сортировка слиянием. Неформальное описание следующее: делим массив на две почти равные части, каждую из них сортируем, вызывая процедуру рекурсивно для каждой из частей, и соединяем отсортированные части за линейное время при помощи процедуры Merge, которую мы опишем далее. Данная идея записывается формально в виде следующей процедуры, которая принимает на вход часть массива $A[p\,..\,r]$ и сортирует его.

```
1:  procedure Merge(A, p, q, r)
2:      i ← 0
3:      j ← 0
4:      while p + i < q и q + j < r do
5:          if A[p + i] < A[q + j] then
6:              B[i + j] ← A[p + i]
7:              i ← i + 1
8:          else
9:              B[i + j] ← A[q + j]
10:             j ← j + 1
11:         end if
```

```
    end while
    while p + i < q do
        B[i + j] ← A[p + i]
        i ← i + 1
    end while
    while q + j < r do
        B[i + j] ← A[q + j]
        j ← j + 1
    end while
    for k ← 0 to i + j do
        A[p + k] ← B[k]
    end for
end procedure
```

На каждом шаге этой процедуры из каждого из двух массивов $A[p\,..\,q]$ и $A[q\,..\,r]$ берется минимальный невставленный элемент, x и y соответственно, в результирующий массив вставляется минимальный из элементов x и y. Нетрудно видеть, что если $A[p\,..\,q]$ и $A[q\,..\,r]$ упорядочены, то и $A[p\,..\,r]$ также упорядочен.

Теперь опишем сам алгоритм.

```
procedure MERGESORT(A, p, r)
    if p < r then
        MERGESORT(A, p, ⌊(p+r)/2⌋)
        MERGESORT(A, ⌊(p+r)/2⌋ + 1, r)
        MERGE(A, p, ⌊(p+r)/2⌋, r)
    end if
end procedure
```

Корректность сортировки слиянием легко установить индукцией по длине массива. База очевидно верна — сортировка массива длины 1 его же и возвращает. Если MERGESORT корректно сортирует все массивы длины меньше n, то работая на массиве длины n MERGESORT корректно сортирует $A[p\,..\,\lfloor\frac{p+r}{2}\rfloor]$ и $A[\lfloor\frac{p+r}{2}\rfloor + 1\,..\,r]$ и корректно их сливает.

Нетрудно показать, что процедура MERGE работает за время $\Theta(n)$ для массива размера n. Если обозначить через $T(n)$ временную сложность работы алгоритма MERGESORT на массиве длины n, то можно получить следующее рекуррентное соотношение:

$$T(n) = T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + T\left(\left\lceil\frac{n}{2}\right\rceil\right) + O(n).$$

По теореме Аккра-Баззи $a_1 = a_2 = 1$, $b_1 = b_2 = \frac{1}{2}$, следовательно, $p = 1$.

$$\int_1^n \frac{u}{u^2}du = \ln n.$$

Следовательно, $T(n) = \Theta(n + n\log n) = \Theta(n\log n)$.

Быстрая сортировка: детерминированная и вероятностная версии. Рассмотрим теперь вероятностный алгоритм сортировки массива под названием *быстрая сортировка* (Quicksort). Как мы увидим далее, этот алгоритм в худшем случае работает как сортировка пузырьком то есть $\Theta(n^2)$, однако математическое ожидание времени работы этого алгоритма равняется $\Theta(n \log n)$, причём множитель при $n \log n$ достаточно мал. Кроме того, быстрая сортировка требует только $O(1)$ дополнительной памяти. Поэтому этот алгоритм любят использовать на практике.

Однако начнём мы для простоты с детерминированного аналога процедуры Quicksort (и именно его будем называть Quicksort; вероятностный аналог мы будем называть Randomized-Quicksort). Быстрая сортировка основана на методе «разделяй и властвуй», а именно, сортировка участка $A[p\,..\,r]$ происходит следующим образом.

(1) Элементы массива A переставляются так, чтобы любой из элементов участка $A[p\,..\,q]$ был не больше любого из элементов участка $A[q+1\,..\,r]$, где $p \leqslant q < r$. Эту операцию мы будем называть Partition. Мы с ней уже сталкивались, когда изучали поиск медианы за линейное время на первом семинаре.

(2) Процедура сортировки вызывается рекурсивно для массивов $A[p\,..\,q]$ и $A[q+1\,..\,r]$.

В результате такой процедуры массив $A[p\,..\,r]$ будет отсортирован.

```
1: procedure Quicksort(A, p, r)
2:     if p < r then
3:         q ← Partition(A, p, r)
4:         Quicksort(A, p, q)
5:         Quicksort(A, q + 1, r)
6:     end if
7: end procedure
```

Чтобы отсортировать весь массив необходимо выполнить процедуру Quicksort$(A, 1, |A|)$.

Основной шаг данного алгоритма состоит в разбиении массива на две части. Рассмотрим следующую процедуру.

```
1: procedure Partition(A, p, r)
2:     x ← A[p]
3:     i ← p − 1
4:     j ← r + 1
5:     while True do
6:         repeat j ← j − 1
7:         until A[j] ⩽ x
8:         repeat i ← i + 1
9:         until A[i] ⩾ x
10:        if i < j then
```

```
11:             A[i] ↔ A[j]
12:         else
13:             return j
14:         end if
15:     end while
16: end procedure
```

В качестве граничного элемента берётся $x = A[p]$. Далее массив проходится с двух концов на встречу друг другу. Если при проходе слева направо встречается элемент не меньший x, то мы останавливаемся на нём. Аналогично, если при проходе справа налево встречается элемент не больший x, то мы останавливаемся на нём. Если $i < j$, то, элементы $A[i]$ и $A[j]$ нужно поменять местами (устранить коллизию). Когда все элементы будут обработаны, то индекс j будет границей левой части массива. Так как каждый элемент мы сравниваем с x не более двух раз и сравниваем числа i и j тоже не более n раз, то время работы процедуры Partition равняется $\Theta(n)$.

Время работы процедуры Quicksort зависит от того, насколько хорошо мы разделяем массив на две части. Нетрудно показать, что если разделение происходит примерно на равные части, то рекуррентное соотношение на время работы быстрой сортировки будет иметь вид

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),$$

из которого следует, что $T(n) = O(n \log n)$. Это соответствует наилучшему разбиению. Наихудший случай наступает при условии, что при каждом вызове процедуры Partition массив делится на две части размеров 1 и $n - 1$ соответственно. В таком случае рекуррентное соотношение получается следующим:

$$T(n) = T(n-1) + \Theta(n),$$

откуда

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2).$$

Более того, заметим, что если на каждом шаге массив делится на части размера αn и $(1-\alpha)n$, где $\alpha \in (0,1)$, то рекуррентное соотношение будет следующим:

$$T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n),$$

откуда получаем, что $T(n) = \Theta(n \log n)$ (эту оценку можно показать, например, по индукции).

Мы видим, что процедура Partition на некоторых входах будет работать долго, а на некоторых — быстро. Если же мы хотим, чтобы время работы процедуры не зависело от того, как расположены элементы во входе, можно рассмотреть вероятностный аналог этой процедуры. Например, можно делать следующее: выбирать каждый раз барьерный

элемент случайным образом при помощи некоторой функции Random. На самом деле это эквивалентно тому, что просто сгенерировать некоторую случайную перестановку индексов (это можно сделать за $\Theta(n)$), а затем использовать её в процессе работы. В результате получим следующие процедуры.

```
1: procedure Randomized-Partition(A, p, r)
2:     i ← Random(p, r)
3:     A[p] ↔ A[i]
4:     return Partition(A, p, r)
5: end procedure
6: procedure Randomized-Quicksort(A, p, r)
7:     if p < r then
8:         q ← Randomized-Partition(A, p, r)
9:         Randomized-Quicksort(A, p, q)
10:        Randomized-Quicksort(A, q + 1, r)
11:    end if
12: end procedure
```

В худшем случае эта процедура по прежнему работает $\Theta(n^2)$. Интуитивно понятно, что с ненулевой вероятностью мы на каждой итерации будем делить массив наиболее несбалансированно. На самом деле это можно даже строго обосновать (посмотреть, как это делается можно в книге Кормена).

Анализ среднего времени работы мы проведём в случае, когда все элементы массива различны. Тогда заметим, что значение q, которое вернёт процедура Partition зависит только от того, сколько в массиве элементов не больших $x = A[p]$ (число таких элементов будем называть рангом и обозначать через $\mathrm{rank}(x)$). Если ранг больше единицы, то левая часть разбиения будет содержать $\mathrm{rank}(x) - 1$ элементов. Если же $\mathrm{rank}(x) = 1$, то левая часть будет содержать один элемент. Так как в качестве барьерного может быть выбран любой элемент массива, то значение $\mathrm{rank}(x)$ принимает каждое из значений $1, 2, \ldots, n$ с одинаковой вероятностью $\frac{1}{n}$. Отсюда следует, что левая часть будет содержать $2, 3, \ldots, n-1$ элементов с вероятностью $\frac{1}{n}$, а с вероятностью $\frac{2}{n}$ — один элемент. Пусть $T(n)$ — это математическое ожидание времени работы процедуры Randomized-Quicksort, то есть $T(n) = \mathbb{E}[T(A)]$, где $T(A)$ — время работы Randomized-Quicksort на массиве A (это случайная величина). Если расписать $T(n)$ по формуле полного математического ожидания, то мы получим

$$T(n) = \frac{1}{n}\left(2T(1) + 2T(n-1) + \sum_{q=2}^{n-1}\left(T(q) + T(n-q)\right)\right) + \Theta(n).$$

Поскольку $T(1) = \Theta(1)$ и $T(n-1) = O(n^2)$, то

$$\frac{1}{n}\left(T(1) + T(n-1)\right) = \frac{1}{n}\left(\Theta(1) + O(n^2)\right) = O(n),$$

поэтому можно записать рекуррентное соотношение в следующем виде:

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1}\left(T(q) + T(n-q)\right) + \Theta(n).$$

Так как каждое слагаемое в скобке встречается ровно два раза, то

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n).$$

Покажем индукцией по n, что $T(n) \leqslant an\log n + b$, где $a > 0$ и $b > 0$ — некоторые константы, которые будут подобраны далее. При $n = 1$ утверждение верно, если взять достаточно большое b. Теперь докажем шаг индукции. При $n > 1$ имеем

$$\begin{aligned} T(n) &= \tfrac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \\ &\leqslant \tfrac{2}{n}\sum_{k=1}^{n-1}(ak\log k + b) + \Theta(n) \\ &= \tfrac{2a}{n}\sum_{k=1}^{n-1} k\log k + \tfrac{2b}{n}(n-1) + \Theta(n). \end{aligned}$$

Ниже мы покажем, что $\sum\limits_{k=1}^{n-1} k\log k \leqslant \frac{1}{2}n^2\log n - \frac{1}{8}n^2$. Используя это, получаем

$$\begin{aligned} T(n) &\leqslant \tfrac{2a}{n}\left(\tfrac{1}{2}n^2\log n - \tfrac{1}{8}n^2\right) + \tfrac{2b}{n}(n-1) + \Theta(n) \\ &\leqslant an\log n - \tfrac{a}{4}n + 2b + \Theta(n) \\ &= an\log n + b + \left(\Theta(n) + b - \tfrac{a}{4}n\right) \leqslant an\log n + b, \end{aligned}$$

где последнее неравенство верно, если взять a достаточно большим. Отсюда получаем, что $T(n) = O(n\log n)$.

Осталось показать, что $\sum\limits_{k=1}^{n-1} k\log k \leqslant \frac{1}{2}n^2\log n - \frac{1}{8}n^2$. Так как при $k < \left\lceil\frac{n}{2}\right\rceil$ имеем $\log k \leqslant \log n - 1$. Поэтому

$$\begin{aligned} \sum_{k=1}^{n-1} k\log k &\leqslant (\log n - 1)\sum_{k=1}^{\left\lceil\frac{n}{2}\right\rceil - 1} k + \log n \sum_{k=\left\lceil\frac{n}{2}\right\rceil}^{n-1} k = \log n\sum_{k=1}^{n-1} k - \sum_{k=1}^{\left\lceil\frac{n}{2}\right\rceil - 1} k \\ &\leqslant \tfrac{1}{2}n(n-1)\log n - \tfrac{1}{2}\left(\tfrac{n}{2} - 1\right)\tfrac{n}{2} \\ &\leqslant \tfrac{1}{2}n^2\log n - \tfrac{1}{8}n^2. \end{aligned}$$

Сортировка пузырьком. Рассмотрим алгоритм сортировки пузырьком (Bubblesort) и проведем его анализ. Сразу договоримся, что массив элементов и массив ключей мы различать не будем, и под сравнением элементов мы будем понимать сравнение соответствующих ключей. Итак, пусть нам дан массив $A[1\,..\,N]$, который мы хотим отсортировать. Опишем процедуру сортировки пузырьком.

```
1: procedure Bubblesort(A)
2:     for i ← 1 to N − 1 do
3:         for j ← N to i + 1 do
4:             if A[j] < A[j − 1] then
5:                 A[j] ↔ A[j − 1]
6:             end if
7:         end for
8:     end for
9: end procedure
```

На каждой итерации внутреннего цикла (строки 4-5) происходит ровно одно сравнение. Итоговое число сравнений равняется $(N-1)+(N-2)+\ldots 1 = \frac{N(N-1)}{2} = O(N^2)$. Таким образом, временная сложность сортировки пузырьком есть $O(N^2)$. Отметим, что можно немного видоизменить процедуру, а именно, завести некоторую переменную, которая будет обнуляться перед выполнением внутреннего цикла и ей будет присваиваться значение 1, если хотя бы одно сравнение в строке 4 было верно (такая переменная называется флагом). Кроме того, нужно перед очередным выполнением внутреннего цикла проверять, равен ли флаг единице. Если вдруг он не равен единице, то это означает, что массив уже полностью отсортирован и можно завершить процедуру. Если рассматривать видоизменённую процедуру сортировки пузырьком, то её время работы в худшем случае будет равно $O(N^2)$, а время работы в лучшем случае будет равно $O(N)$ (в том случае, когда исходный массив уже отсортирован в нужно порядке).

Кроме того, отметим, что алгоритм сортировки пузырьком требует дополнительной памяти размера $O(1)$ (размер одного элемента).

Сортировка с помощью кучи. Итак, мы убедились, что сортировка пузырьком работает за время $O(n^2)$, но при этом использует всего лишь $O(1)$ дополнительной памяти. Сортировка слиянием работает почти на порядок быстрее, а именно за время $O(n \log n)$, однако использует $O(n)$ дополнительной памяти, что на порядок больше, чем у сортировки пузырьком.

Оказывается, что можно предложить алгоритм сортировки, который использует структуру данных, называемую *двоичной кучей*, и который работает за время $O(n \log n)$ (как сортировка слиянием), используя при этом $O(1)$ дополнительной памяти (как сортировка пузырьком). Этот способ называется *сортировкой с помощью кучи* (Heapsort).

Двоичная куча — это массив с определёнными свойствами упорядоченности, которые мы опишем далее. Чтобы описать эти свойства нам будет удобно представлять массив в виде двоичного дерева (то есть дерева, у каждой вершины которого существует не более двух потомков). Во-первых, каждая вершина дерева соответствует элементу массива. Во-вторых, если вершина имеет индекс i, то её родитель имеет индекс $\lfloor \frac{i}{2} \rfloor$, если он определён (отсюда, например, следует, что вершина с индексом 1 является корнем), а её дети — индексы $2i$ и $2i+1$ (что следует из первой части утверждения). В дальнейшем нам будет удобно считать, что размер кучи может быть меньше длины массива, поэтому вместе с массивом A мы будем хранить его длину $len[A]$ и размер кучи $heapsize[A]$. Движение по дереву осуществляется следующими процедурами.

```
1: procedure PARENT(i)
2:     return ⌊i/2⌋
3: end procedure
4: procedure LEFT(i)
5:     return 2i
6: end procedure
7: procedure RIGHT(i)
8:     return 2i + 1
9: end procedure
```

Наконец, должно выполняться *основное свойство кучи*: для каждой вершины i, кроме корневой,

$$A[\text{PARENT}(i)] \geqslant A[i].$$

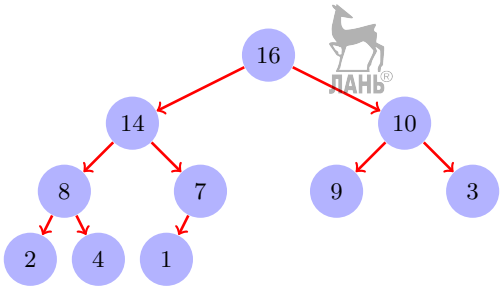


Рис. 1. Двоичная куча [16, 14, 10, 8, 7, 9, 3, 2, 4, 1], представленная в виде дерева.

Высотой вершины дерева будем называть высоту поддерева, корнем которого является эта вершина (то есть число рёбер в самом длинном пути с началом в этой вершине

вниз по дереву к листу). Заметим, что все уровни дерева, составляющего кучу, заполнены полностью, кроме, быть может, последнего уровня. Поэтому высота этого дерева равна $\Theta(\log n)$, где n — число элементов в куче. Как увидим далее, время работы основных операций над кучей пропорционально высоте дерева и, следовательно, составляет $O(\log n)$.

Одной из таких операций является процедура HEAPIFY. Она позволяет сохранять основное свойство кучи. На вход эта процедура получает массив A и индекс i. Предполагается, что поддеревья с корнями LEFT(i) и RIGHT(i) обладают основным свойством кучи. Идея проста: если основное свойство не выполнено для вершины i, то её следует поменять с наибольшим из её детей и так далее, пока элемент $A[i]$ не «погрузится» до нужного места.

```
 1: procedure HEAPIFY(A, i)
 2:     l ← LEFT(i)
 3:     r ← RIGHT(i)
 4:     if l ⩽ heapsize[A] и A[l] > A[i] then
 5:         largest ← l
 6:     else
 7:         largest ← i
 8:     end if
 9:     if r ⩽ heapsize[A] и A[r] > A[largest] then
10:         largest ← r
11:     end if
12:     if largest ≠ i then
13:         A[i] ↔ A[largest]
14:         HEAPIFY(A, largest)
15:     end if
16: end procedure
```

В строках 4-10 в переменную largest помещается индекс наибольшего из элементов $A[i]$, $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$. Если largest $= i$, то элемент уже «погрузился» до нужного места и работа процедуры закончена. Иначе процедура меняет местами $A[i]$ и $A[\text{largest}]$, что обеспечивает выполнение основного свойства кучи в вершине i, но, возможно, основное свойство могло нарушится в вершине largest, поэтому нужно вызвать процедуру HEAPIFY(A, largest), чтобы восстановить основное свойство кучи в этой вершине.

Оценим время работы процедуры HEAPIFY. На каждом шаге требуется совершить $\Theta(1)$ сравнений. Кроме того производится рекурсивный вызов. Если поддерево с корнем i содержит n элементов, то поддеревья LEFT(i) и RIGHT(i) содержат не более $\frac{2n}{3}$ элементов (в силу того, что все уровни кучи, кроме, быть может, последнего, заполнены полностью, мы получаем, что наихудший случай — когда последний уровень для данного поддерева заполнен наполовину). Пусть $T(n)$ — время работы процедуры HEAPIFY на поддереве

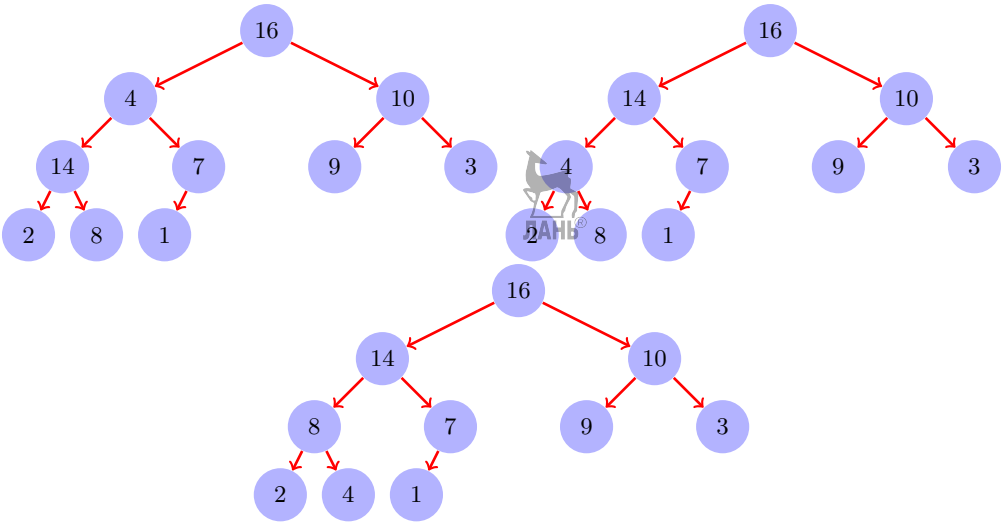


Рис. 2. Работа процедуры Heapify$(A, 2)$ при $heapsize[A] = 10$.

размера n. Тогда

$$T(n) \leqslant T\left(\frac{2n}{3}\right) + \Theta(1).$$

Отсюда получаем, что $T(n) = O(\log n)$. Но это можно понять и из более простых соображений: на каждой итерации мы спускаемся по дерево на уровень вниз (либо завершаем работу). Поэтому итераций будет совершено не более $\log n$, то есть сравнений будет $O(\log n)$.

Теперь покажем, что из массива $A[1\,..\,n]$ можно за время $O(\log n)$ сделать двоичную кучу. Эту задачу решает процедура Build-Heap. Основная идея этой процедуры — вызывать процедуру Heapify в правильном порядке. Поскольку вершины $\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n$ являются листьями, то поддеревья в этих вершинах по очевидным причинам удовлетворяют основному свойству кучи. Теперь для каждой из оставшихся вершин в порядке убывания индексов мы будем применять процедуру Heapify. Порядок обработки вершин гарантирует, что каждый раз условия вызова процедуры Heapify будут выполнены.

1: **procedure** Build-Heap(A)
2: $\quad heapsize[A] \leftarrow \mathrm{length}[A]$
3: $\quad$ **for** $i \leftarrow \left\lfloor \frac{len[A]}{2} \right\rfloor$ *downto* 1 **do**

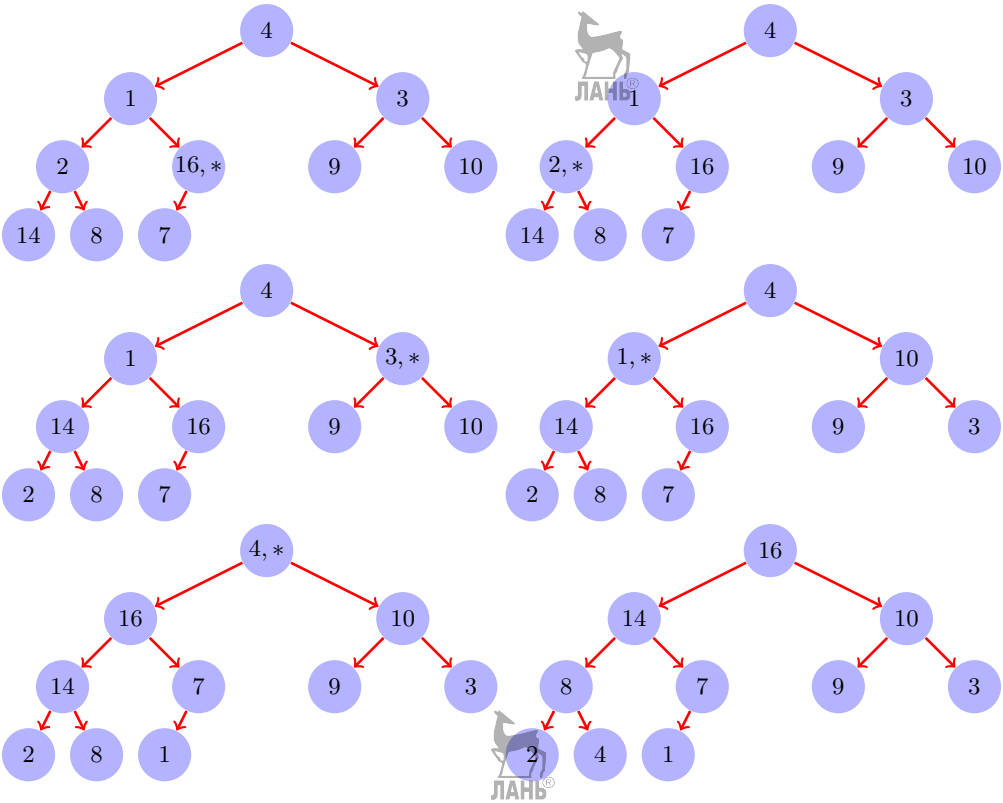


Рис. 3. Работа процедуры Build-Heap(A) для $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$. На рисунке показано состояние данных перед каждым вызовом процедуры Heapify в строке 3. Знаком $*$ помечалась та вершина, для которой вызывалась процедура Heapify. Последняя картинка — результат работы процедуры Build-Heap.

4: Heapify(A, i)
5: **end for**
6: **end procedure**

Оценим временную сложность работы процедуры Build-Heap. Во-первых, можно было бы сказать, что время работы равно $O(n \log n)$, поскольку время работы Heapify равно

$O(\log n)$ и она вызывается не более n раз. Но такая оценка является на завышенной, множитель $\log n$ можно убрать.

Как мы отметили ранее, работа процедуры Heapify пропорциональна высоте дерева, для которого она вызывается. Кроме того, число вершин высоты h в двоичной куче из n элементов не превосходит $\lceil \frac{n}{2^{h+1}} \rceil$, а высота всей кучи не превосходит $\lfloor \log n \rfloor$. Поэтому время работы процедуры Build-Heap не превышает

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O(n),$$

где последняя формула следует из

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} < \sum_{h=0}^{\infty} \frac{h}{2^h} = 2.$$

Равенство $\sum\limits_{h=0}^{\infty} \frac{h}{2^h} = 2$ было доказано на втором семинаре (см. пример вероятностного алгоритма типа «Монте-Карло»).

Теперь мы можем описать сортировку при помощи кучи (Heapsort). Она состоит из двух частей. Сначала вызывается процедура Build-Heap, чтобы сделать из массива кучу. В таком случае максимальный элемент массива будет находится в корне дерева (то есть первым). Соответственно, алгоритм меняет местами элементы $A[1]$ и $A[n]$, уменьшает размер кучи на единицу и восстанавливает основное свойство кучи в корневой вершине (так как поддеревья с корнями Left(i) и Right(i) не утратили основного свойства кучи, то это можно сделать с помощью процедуры Heapify). После этого в корне будет находиться максимальный из оставшихся элементов. Так делается до тех пор, пока в куче не останется один элемент.

```
1: procedure Heapsort(A)
2:     Build-Heap(A)
3:     for i ← len[A] downto 2 do
4:         A[1] ↔ A[i]
5:         heapsize[A] ← heapsize[A] − 1
6:         Heapify(A, 1)
7:     end for
8: end procedure
```

Время работы процедуры Heapsort равняется $O(n \log n)$: на выполнение процедуры Build-Heap требуется время $O(n)$, каждая итерация цикла требует времени $O(\log n)$, а итераций всего $n - 1$. При этом алгоритм требует лишь $O(1)$ дополнительной памяти.

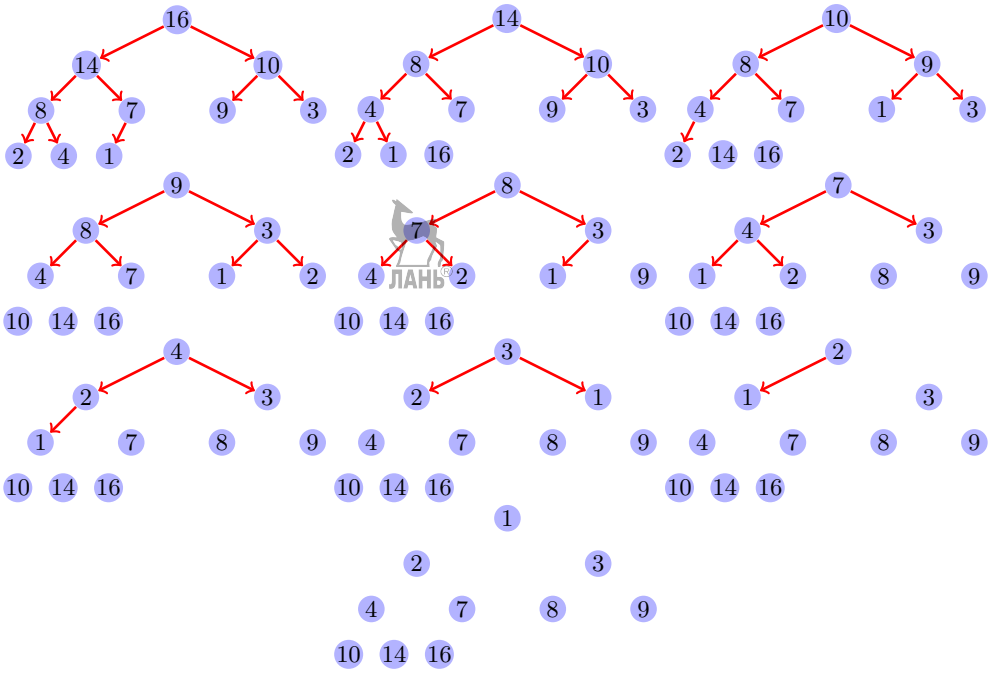


Рис. 4. Работа процедуры HEAPSORT(A) для $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$. На рисунке показано состояние данных перед каждым вызовом процедуры HEAPIFY в строке 6.

Нижние оценки для сортировки. Итак, мы до этого рассмотрели 3 алгоритма сортировки с временной сложностью $O(n \log n)$ и один алгоритм с временной сложностью $\Theta(n^2)$. На фоне времени $O(n^2)$ время $O(n \log n)$ выглядит достаточно симпатично. Но оптимальное ли (асимптотически) это время для сортировки, основанной на сравнениях? Оказывается, что да и доказать это можно с помощью так называемых *разрешающих деревьев*.

Любой детерминированный алгоритм сортировки попарными сравнениями можно представить в виде двоичного дерева. Действительно, пусть мы сортируем n элементов $a_1, a_2, \ldots, a_n$. Каждая внутренняя вершина (то есть не листовая вершина) соответствует операции сравнения двух элементов a_i и a_j и снабжена пометкой $a_i ? a_j$. Каждый лист

разрешающего дерева снабжён пометкой $(\pi(1), \pi(2), \ldots, \pi(n))$, где π — перестановка n элементов. Рёбра дерева снабжены пометками «$\leqslant$» и «$>$».

Каждому алгоритму сортировки соответствует своё разрешающее дерево (и наоборот). Чтобы по дереву получить алгоритм сортировки, нужно идти по дереву от корня к листьям и сравнивать элементы, соответствующие текущей вершине: если в текущей вершине написано $a_i ? a_j$, то нужно сравнить a_i и a_j, и если $a_i \leqslant a_j$, то нужно пойти налево, а в противном случае — направо. Если пришли в лист, в котором записана перестановка π, то нужно вернуть результат $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$. Время работы алгоритма сортировки равно длине соответствующего пути в разрешающем дереве.

Теорема 2.1. *Устойчивый алгоритм сортировки массива длины n требует $\Omega(n \log n)$ сравнений элементов входного массива.*

Доказательство. Каждая из $n!$ перестановок должна появиться хотя бы в одном листе дерева (правильный алгоритм сортировки должен предусматривать все возможные порядки). Нетрудно видеть, что $n! \geqslant \left(\frac{n}{e}\right)^n$. Двоичное дерево высоты h имеет не более 2^h листьев, поэтому высота разрешающего дерева сортировки должна быть не меньше

$$\log_2(n!) \geqslant n \log_2 n - n \log_2 e = \Omega(n \log n).$$

□

Поиск k-ой статистики

Теперь перейдем к другой задаче. Пусть, опять же, дан набор чисел $a[0], \ldots, a[n-1]$. Рассмотрим задачу поиска *k-й порядковой статистики*, т.е. такого элемента $a_{(k)}$, который в отсортированном массиве стоит на k-м месте. Например, $a_{(n/2)}$, т.е. средний элемент, — это *медиана* массива. Предложим линейный алгоритм поиска медианы. Допустим, что мы выбрали некоторый элемент x в качестве опорного, и разбили массив на две части b и c так, что $b[i] < x$, $c[j] > x$. Если $len(b) < k$, то $x \leq a_{(k)}$, и k-я порядковая статистика не может находится в массиве b. Аналогично, если $len(c) \leq n - k$, то она не может находится в массиве c. В принципе, возможны и оба случая, когда искомой величиной оказывается x. Если мы можем выбрать x так, чтобы размер массива b и c был не слишком большим, скажем, меньше чем αn, $\alpha < 1$, то алгоритм будет работать быстро. Один из способов это сделать следующий: разобьем исходный массив на пятерки, и в каждой из них найдем медиану — это займет $O(n)$ времени; в полученном массиве из медиан m размера $\frac{n}{5}$ рекурсивно найдем медиану x, и используем ее в качестве опорного элемента — разбиение исходного массива a на b и c также занимает $O(n)$ времени. Как несложно видеть, в этом случае $len(b), len(c) \leq \frac{7}{10}n$ (с точностью до константы). Затем выбираем массив b или c, в котором находится искомая порядковая статистика (например, если $len(b) < k$, то она точно не может находиться в массиве b), и рекурсивно запускаем поиск (если это массив c, то нужно соответственно изменить номер k).

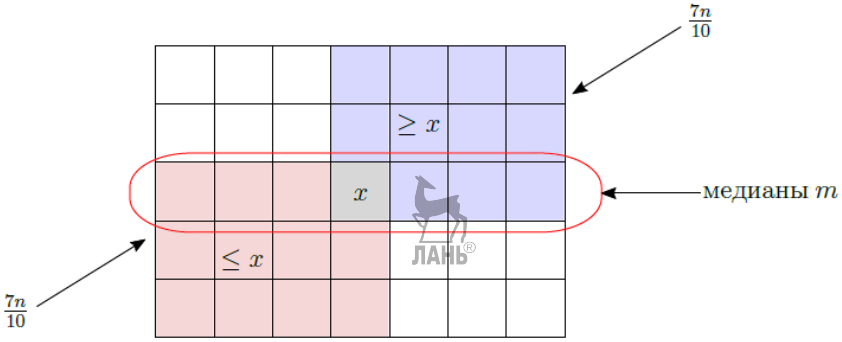


Рис. 5. Разбиение массива

Таким образом, получаем следующее рекуррентное соотношение:

$$T(n) = T\left(\frac{n}{5} + A\right) + T\left(\frac{7n}{10} + A\right) + O(n).$$

Воспользуемся теоремой Аккра-Баззи; здесь $a_1 = a_2 = 1$, $b_1 = \frac{1}{5}$, $b_2 = \frac{7}{10}$, $p \approx 0.84 < 1$. Поэтому

$$\int_1^n \frac{u}{u^{p+1}} du = \frac{1 - n^{1-p}}{p-1} = \Theta\left(n^{1-p}\right).$$

Отсюда $T(n) = \Theta\left(n^p + n\right) = \Theta(n)$.

Часть 3

Алгебра и теория чисел

В этом разделе мы будем изучать алгоритмы в алгебре и теории чисел. Сначала напомним необходимые нам определения и теоремы.

Определение 3.1. *Группа* — пара $(G, \cdot)$, где

- M — множество,
- $\cdot$ — операция умножения, удовлетворяющая следующим аксиомам:
  - $\forall a, b, c \in M\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$ (ассоциативность);
  - $\exists e \in M\ e \cdot a = a \cdot e = a$ (наличие нейтрального элемента);
  - $\forall a \in M\ \exists b \in M$ такой, что $ab = ba = e$.

Часто в обозначении моноида мы будем опускать операцию умножения и писать G вместо $(G, \cdot)$. Также мы будем опускать $\cdot$, записывая произведение элементов: под ab будем понимать $a \cdot b$.

Определение 3.2. Группы $\mathbb{Z}/n\mathbb{Z}$ и $(\mathbb{Z}/n\mathbb{Z})^\times$. Будем обозначать через $(\mathbb{Z}/n\mathbb{Z}, +_n)$ — аддитивную (по сложению) группу вычетов по модулю n, а через $((\mathbb{Z}/n\mathbb{Z})^\times, \cdot_n)$ — мультипликативную (по умножению) группу вычетов по модулю n (далле будем опускать индекс n при обозначении групповой операции, а сами группы обозначать $\mathbb{Z}/n\mathbb{Z}$ и $(\mathbb{Z}/n\mathbb{Z})^\times$ соответственно).

Отметим, что аддитивная группа вычетов содержит вычеты $[0]_n, [1]_n, \ldots, [n-1]_n$ (то есть все вычеты), а мультипликативная группа состоит только из вычетов, взаимно простых с n. Количество элементов в группе $((\mathbb{Z}/n\mathbb{Z})^\times, \cdot_n)$ определяется *функцией Эйлера*, которая обозначается через $\varphi(n)$ (то есть $\varphi(n)$ — это количество чисел, которые меньше n и взаимно просты с n). Можно доказать, что

$$\varphi(n) = n\left(1 - \frac{1}{p_1}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_s}\right),$$

где $p_1, p_2, \ldots, p_s$ — список всех (различных) простых делителей числа n. Это делается в три шага:

1) показать, что для простых n выполняется $\varphi(n) = n - 1$;
2) показать, что для всех n вида $n = p^k$, где p — простое число, выполняется $\varphi(n) = p^k - p^{k-1}$;
3) показать, что для взаимно простых чисел m и n выполняется $\varphi(mn) = \varphi(m)\varphi(n)$.

Группу G можно задавать с помощью таблицы *таблицы умножения*, то есть напрямую задав произведение на всех парах элементов $x, y \in G$.

Определение 3.3. *Образующие (порождающие)* элементы группы G — подмножество $X \subset G$ элементов моноида такое, что любой элемент $m \in G$ может быть представлен как произведение элементов X, то есть $m = x_1 \ldots x_k$ для некоторых $x_1, \ldots x_k \in X$.

Группу G можно задать с помощью множества образующих $X \subset M$ и *соотношений*, то есть множества пар элементов $(x_1 \dots x_m, y_1 \dots y_n)$, где $x_1, \dots x_m, y_1, \dots y_n \in X$ и $x_1 \dots x_m = y_1 \dots y_n$. Например, $\mathbb{Z}/n\mathbb{Z}$ порождается одним элементом 1 с соотношением $\underbrace{1 + \dots + 1}_{n} = 0$.

Определение 3.4. Группы, порождённые одним элементом, называют *циклическими*.

Пусть $x \in G$, где G — конечная группа. Будем применять групповую операцию $\cdot$ к элементам e и x, затем к x и x, потом к x и x и так далее. В результате мы получим последовательность

$$e,\ x,\ x \cdot x,\ x \cdot x \cdot x, \dots\ .$$

Для $k \in \mathbb{N}$ введём обозначения: $x^k = \underbrace{x \cdot x \cdot \ldots \cdot x}_{k \text{ штук}}$ и $x^{(-k)} = \left(x^{-1}\right)^k = \left(x^k\right)^{-1}$. Кроме того, будем считать, что $x^0 = e$. Тогда указанная выше последовательность может быть записана следующим образом:

$$x^0,\ x^1,\ x^2,\ x^3, \dots\ .$$

Так как группа G конечна, то найдутся такие i и j, что $x^i = x^j$, что эквивалентно тому, что найдётся такое целое неотрицательное число m, что $x^m = x^0 = e$. Понятно, что если $x^m = e$, то и $x^{2m} = e, x^{3m} = e$ и так далее. Нас же будет интересовать наименьшее целое неотрицательное число m, что $x^{(m)} = e$. Это число называют *порядком* элемента x и обозначают $m = \mathrm{ord}x$. Более того, элементы $x^0,\ x^1,\ x^2,\ x^3 \dots, x^{m-1}$ образуют подгруппу группы G. Такую подгруппу называют *подгруппой, порождённой элементом* x и обозначают $\langle x \rangle$. Элемент x называют *образующим* подгруппы $\langle x \rangle$. Из всего написано ранее следуют три простых факта.

Теорема 3.5. *Пусть $(G, \cdot)$ — конечная группа. Если $x \in G$, то $|\langle x \rangle| = \mathrm{ord}x$.*

Утверждение 3.6. *Для любого $x \in G$, где G — конечная группа, последовательность $x^{(0)}, x^{(1)}, x^{(2)}, \dots$ имеет период $m = \mathrm{ord}x$, то есть $x^{(i)} = x^{(j)}$ тогда и только тогда, когда $i \equiv j \pmod m$.*

Следующий факт получается из теоремы Лагранжа и теоремы 3.5.

Утверждение 3.7. *Пусть G — конечная группа. Тогда для любого $x \in G$ выполняется $x^{(|G|)} = e$.*

Определение 3.8. Отображение моноидов $h : M_1 \to M_2$ называется *морфизмом*, если для любых $x, y \in M$ верно

$$h(xy) = h(x)h(y). \tag{11}$$

Определение 3.9. Пусть G — группа. Множество $G' \subset G$ будем *подгруппой* G, если для любых $x, y \in G'$ верно $xy \in G'$. Подгруппа G' группы G называется собственной, если она не совпадает со всей группой G.

Определение 3.10. *Порядок* конечной группы G — количество элементов в группе G. Обозначается через $|G|$.

Теорема 3.11 (теорема Лагранжа). *Если $(G', \cdot)$ является подгруппой группы $(G, \cdot)$, то $|G'|$ является делителем числа $|G|$.*

Доказательство. См. [20]. □

Отметим важное следствие теоремы Лагранжа.

Утверждение 3.12. *Если G' является собственной подгруппой конечной группы G, то $|G'| < \frac{|G|}{2}$.*

Оказывается, что если применить следствие 3.7 для группы $(\mathbb{Z}/n\mathbb{Z})^\times$, то получим следующий факт, известный многим со школьных олимпиад.

Теорема 3.13 (теорема Эйлера). *Если $n > 1$ — целое число, то для любого целого числа x, взаимно простого с n, выполнено*

$$x^{\varphi(n)} \equiv 1 \pmod{n}.$$

Если n — просто число, то получаем ещё один известный многим со школы факт.

Теорема 3.14 (малая теорема Ферма). *Если $p > 1$ — простое число, то для любого целого числа x, взаимно простого с n, выполнено*

$$x^{p-1} \equiv 1 \pmod{p}.$$

Теперь предположим, что в группе $(\mathbb{Z}/n\mathbb{Z})^\times$ существует некоторый элемент g, такой что $\mathrm{ord} g = |(\mathbb{Z}/n\mathbb{Z})^\times| = \varphi(n)$. Это означает, что всю группу $(\mathbb{Z}/n\mathbb{Z})^\times$ можно получить, возводя элемент g в степени $0, 1, 2, \ldots, \varphi(n) - 1$, то есть все элементы $(\mathbb{Z}/n\mathbb{Z})^\times$ являются степенями g. Если такой элемент g существует, то его называют *первообразным корнем по модулю n*. Первообразный корень существует не всегда

Теорема 3.15. *В группе $(\mathbb{Z}/n\mathbb{Z})^\times$ существует первообразный корень тогда и только тогда, когда n равно 2, 4, имеет вид p^k или $2p^k$, где $p > 2$ — простое число, а k — натуральное число.*

Если g — первообразный корень в группе $(\mathbb{Z}/n\mathbb{Z})^\times$, то для всякого $a \in (\mathbb{Z}/n\mathbb{Z})^\times$ существует x, для которого $g^x \equiv a \pmod{n}$. Такое x называют *дискретным логарифмом* или *индексом* элемента $a \in (\mathbb{Z}/n\mathbb{Z})^\times$ по основанию g. Из теоремы 3.5 вытекает следующий факт.

Теорема 3.16. *Пусть g — первообразный корень по модулю n. Тогда сравнение $g^x \equiv g^y \pmod{n}$ равносильно сравнению $x \equiv y \pmod{\varphi(n)}$.*

Из этой теоремы видно, что понятие индекса определено с точностью до слагаемого, кратного $\varphi(n)$.

Определение 3.17. *Кольцом* называют такое множество R, на котором заданы две бинарные операции: $+$ (сложение) и $\times$ (умножение), для которых выполняются следующие свойства для любых $a, b, c \in R$:

1) множество R с операцией $+$ образует *коммутативную* (то есть $\forall a, b \in R \hookrightarrow a + b = b + a$) группу;
2) ассоциативность умножения: $(a \times b) \times c = a \times (b \times c)$;
3) дистрибутивность: $a \times (b + c) = a \times b + a \times c$ и $(b + c) \times a = b \times a + c \times a$.

Кольцо с единицей — это кольцо, в котором есть нейтральный элемент по умножению, обозначаемый обычно через 1: $\forall a \in R \hookleftarrow 1 \times a = a \times 1 = a$. Коммутативное кольцо — это кольцо, у которого операция умножения является коммутативной: $\forall a, b \in R \hookrightarrow a \times b = b \times a$.

Определение 3.18. Множество $\mathbb{Z}/n\mathbb{Z}$, операция сложения $+$ и операция умножения $\cdot$ образуют коммутативное кольцо с единицей. Грубо говоря, при работе с вычетами по некоторому модулю n можно их складывать и перемножать как обычные числа, причём промежуточные результаты можно заменять на остатки по модулю n.

Определение 3.19. Пусть R — кольцо. Элемент кольца $a \in R$ называется *делителем нуля*, если $ab = 0$ для некоторого $b \in R$.

Например, 2 — делитель нуля в $\mathbb{Z}/6\mathbb{Z}$, так как $2 \cdot 3 = 0$ в $\mathbb{Z}/6\mathbb{Z}$.

Определение 3.20. *Идеал* I коммутативного кольца R — подмножество $I \subset R$, являющееся абелевой группой по сложению, для которого верно

$$\forall i \in I, r \in R \quad ir \in I$$

Пример 3.21. Все элементы, делящиеся на 2, образуют идеал в $\mathbb{Z}$.

В дальнейшем под $(f_1, \dots f_k) \subset R$ будем понимать идеал, порожденный элементами $f_1, \dots f_k \in R$:

$$(f_1, \dots f_k) = \{f_1 r_1 + \dots f_k r_k \mid r_1, \dots r_k \in R\}.$$

Например, $(2) \subset \mathbb{Z}$ — идеал, порожденный 2.

Определение 3.22. *Полем* называют коммутативное кольцо R, в котором определено деление $a/b = ab^{-1}$ и отсутствуют делители нуля.

Теорема 3.23. *Любое конечное поле имеет мощность p^k, где $k \geqslant 1$, а p — простое. Всякое конечное поле мощности p^k единственно с точностью до изоморфизма.*

Доказательство. См. [20]. □

Поле мощности p^k мы будем обозначать $\mathbb{F}_{p^k}$. Нетрудно заметить, что если $p(x)$ — неприводимый над $\mathbb{F}_p$ многочлен степени k, то $\mathbb{F}[x]/(p(x))$ является полем, изоморфным $\mathbb{F}_{p^k}$.

Решение линейных диофантовых уравнений. *Деление* по модулю n — это решение линейного сравнения $ax \equiv b \pmod{n}$ относительно x. Такие сравнения ещё называют *линейными диофантовыми уравнениями*. Оказывается, что такие сравнения можно решать при помощи расширенного алгоритма Евклида, о котором мы говорили ранее. Для начала докажем несколько вспомогательных утверждений.

Теорема 3.24. *Для любых положительных целых чисел чисел a, n и d, таких что $d = (a, n)$, выполняется*

$$\langle a \rangle = \langle d \rangle = \Big\{0, d, 2d, \ldots, \Big(\frac{n}{d} - 1\Big)\, d\Big\}$$

и

$$|\langle a \rangle| = \frac{n}{d}.$$

Доказательство. В алгоритм Extended-Euclid(a, n) вернёт тройку (d, x', y'), для которой $d = (a, n)$ и $ax' + ny' = d$. Тогда $ax' \equiv d \pmod{n}$ и поэтому $d \in \langle a \rangle$. С другой стороны, d — это делитель a, а значит, $a \in \langle d \rangle$. Следовательно, $\langle a \rangle = \langle d \rangle = \big\{0, d, 2d, \ldots, \big(\frac{n}{d} - 1\big)\, d\big\}$ и $|\langle a \rangle| = \frac{n}{d}$. □

Утверждение 3.25. *Уравнение $ax \equiv b \pmod{n}$ разрешимо относительно x тогда и только тогда (a, n) является делителем числа b.*

Доказательство. Сравнение $ax \equiv b \pmod{n}$ имеет решение тогда и только тогда, когда $b \in \langle a \rangle = \langle d \rangle = \big\{0, d, 2d, \ldots, \big(\frac{n}{d} - 1\big)\, d\big\} \iff d$ является делителем числа b, где $d = (a, n)$. □

Иными словами, «делить число b на число a по модулю n» можно тогда и только тогда, когда (a, n) — это делитель числа b. В частности, если $(a, n) = 1$, то делить на a можно всегда. Но это и понятно: если $(a, n) = 1$, то $a \in (\mathbb{Z}/n\mathbb{Z})^\times \implies \exists a^{-1} \mod n$. Тем не менее «разделить» на a можно даже и в случае, когда a и n не взаимно просты. Например, сравнение $2x \equiv 4 \pmod{6}$ имеет решения $x = 2$ и $x = 5$. Заметим, что в первом случае решение (в $\mathbb{Z}/n\mathbb{Z}$) единственно, а во втором — решений два. Следующий факт даёт ответ на вопрос о числе решений в общем случае.

Утверждение 3.26. *Уравнение* $ax \equiv b \pmod{n}$ *имеет* $d = (a, n)$ *различных решений в* $\mathbb{Z}/n\mathbb{Z}$ *или не имеет их вовсе.*

Доказательство. Здесь полезно смотреть на $\mathbb{Z}/n\mathbb{Z}$ как на группу. Вспомним, как мы определяли $\langle a \rangle$: мы рассматривали последовательность $0, a, 2a, 3a, \ldots$. Мы доказали, что эта последовательность имеет период $|\langle a \rangle| = \frac{n}{d}$. Так как $b \in \langle a \rangle$, то вычет b встретится ровно один раз среди первых $\frac{n}{d}$ членов последовательности $0, a, 2a, 3a, \ldots$. В силу периодичности последовательности, элемент b встретится ещё ровно один раз среди вторых $\frac{n}{d}$ элементов указанной последовательности и так далее. Тогда всего он встретится d раз. Каждому вхождению b в последовательность $0, a, 2a, 3a, \ldots, (n-1)a$ соответствует свой вычет x, а значит, всего таких значений x ровно d штук. □

Следующие две теоремы проливают свет на то, как находить решение линейного диофантового уравнения при помощи расширенного алгоритма Евклида.

Теорема 3.27. *Пусть* $d = (a, n) = ax' + by'$, *где* x' *и* y' *— целые числа (например, они могут быть получены процедурой Extended-Euclid*(a, n)*). Если* $d \mid b$ *(*d *делит* b*), то число* $x_0 = x' \cdot \frac{b}{d} \mod n$ *является решением уравнения* $ax \equiv b \pmod{n}$.

Доказательство. Из $d = ax' + by'$ следует, что $ax' \equiv d \pmod{n}$, поэтому $ax_0 \equiv ax' \cdot \frac{b}{d} \equiv d \cdot \frac{b}{d} \equiv b \pmod{n}$. □

Теорема 3.28. *Пусть уравнение* $ax \equiv b \pmod{n}$ *разрешимо и* x_0 *является его решением. Тогда уравнение имеет* $d = (a, n)$ *решений в* $\mathbb{Z}/n\mathbb{Z}$, *задаваемых формулой* $x_i = x_0 + \frac{i \cdot n}{d}$, *где* $i = 0, 1, 2, \ldots, d-1$.

Доказательство. Из доказательства следствия 3.26 мы знаем, что решения сравнения $ax \equiv b \pmod{n}$ соответствуют d числам из последовательности $0, a, 2a, \ldots, (n-1)a$, причём эти числа расположены с периодом $\frac{n}{d}$. Но это как раз и означает, что все числа вида $x_0 + i \cdot \frac{n}{d}$, где $i = 0, 1, \ldots, d-1$, являются решениями сравнения $ax \equiv b \pmod{n}$. Кроме того, их ровно d штук, причём они попарно различны по модулю n в силу того, что $x_0 + (d-1) \cdot \frac{n}{d} - x_0 = (d-1) \cdot \frac{n}{d} < n$. Значит, это в точности все решения сравнения. □

Следующая процедура по целым числам a, b и $n > 0$ даёт все решения уравнения $ax \equiv b \pmod{n}$.

```
1: procedure Modular-Linear-Equation-Solver(a, b, n)
2:     (d, x', y') ← Extended-Euclid(a, n)
3:     if d | b then
4:         x_0 ← x' · b/d  mod n
5:         for i ← 0 to d − 1 do
6:             print (x_0 + i · n/d)  mod n
7:         end for
```

```
8:     else
9:         print «нет решений»
10:    end if
11: end procedure
```

Процедура Modular-Linear-Equation-Solver(a, b, n) выполнит $O(\log n)$ арифметических операций в строке 2 и $O((a, n))$ операций в остальных строках, то есть всего $O(\log n + (a, n))$. Но если нам достаточно найти все решения, а не печатать их на экран, то число арифметических операций будет $O(\log n)$, а битовых операций будет $O(\log^3 n)$, т.к. процедура Extended-Euclid(a, n) делает $O(\log n)$ делений целых чисел длины $O(\log n)$ с остатком, что делается за $O(\log^2 n)$.

Отметим ещё два важных следствия.

Утверждение 3.29. *Пусть $n > 1$. Если $(a, n) = 1$, то уравнение $ax \equiv b \pmod n$ имеет единственное решение в $\mathbb{Z}/n\mathbb{Z}$. В частности, если $b = 1$ и $(a, n) = 1$, то уравнение $ax \equiv 1 \pmod n$ имеет единственное решение в $\mathbb{Z}/n\mathbb{Z}$. При $(a, n) > 1$ уравнение $ax \equiv 1 \pmod n$ решения не имеет.*

Решение систем линейных сравнений. Китайская теорема об остатках. Развивая тему предыдущего раздела и намерено отходя немного в сторону от модулярной арифметики, рассмотрим систему линейных сравнений вида:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\dots \\ x &\equiv a_k \pmod{n_k}, \end{aligned}$$

где $n_1, n_2, \dots, n_k$ — попарно взаимно простые числа. Наша цель — найти такой вычет a по модулю $n = n_1 n_2 \cdot \ldots \cdot n_k$, что a — это решение указанной линейной системы сравнений, то есть $x \equiv a \pmod n$. Китайская теорема об остатках утверждает, что $\mathbb{Z}/n\mathbb{Z}$ устроено как произведение колец вычетов $\mathbb{Z}/n_1\mathbb{Z} \times \mathbb{Z}/n_2\mathbb{Z} \times \ldots \times \mathbb{Z}/n_k\mathbb{Z}$ (с покомпонентным сложением и умножением). Это соответствие полезно с алгоритмической точки зрения, так как бывает проще выполнить операции во всех множествах $\mathbb{Z}/n_i\mathbb{Z}$, чем в кольце $\mathbb{Z}/n\mathbb{Z}$.

Теорема 3.30 (Китайская теорема об остатках)**.** *Пусть $n = n_1 n_2 \cdot \ldots \cdot n_k$, причём числа $n_1, n_2, \dots, n_k$ попарно взаимно просты. Рассмотрим соответствие*

$$a \leftrightarrow (a_1, a_2, \dots, a_k),$$

где $a \in \mathbb{Z}/n\mathbb{Z}, a_i \in \mathbb{Z}/n_i\mathbb{Z}$ и $a_i \equiv a \pmod{n_i}$ при $i = 1, 2, \dots, k$. Тогда данное соответствие является взаимно однозначным между $\mathbb{Z}/n\mathbb{Z}$ и $\mathbb{Z}/n_1\mathbb{Z} \times \mathbb{Z}/n_2\mathbb{Z} \times \ldots \times \mathbb{Z}/n_k\mathbb{Z}$. При этом операциям сложения, вычитания и умножения в $\mathbb{Z}/n\mathbb{Z}$ соответствуют покомпонентные операции над k-элементными кортежами: если

$$a \leftrightarrow (a_1, a_2, \dots, a_k)$$

и

$$b \leftrightarrow (b_1, b_2, \ldots, b_k),$$

то

$$\begin{aligned} (a+b) \mod n &\leftrightarrow ((a_1+b_1) \mod n_1, \ldots, (a_k+b_k) \mod n_k), \\ (ab) \mod n &\leftrightarrow ((a_1 b_1) \mod n_1, \ldots, (a_k b_k) \mod n_k). \end{aligned}$$

Доказательство. Начнём с того, что отметим, что отображение из $\mathbb{Z}/n\mathbb{Z}$ в декартово произведение задано корректно: если два числа сравнимы по мудлю n, то их разность кратна n, и потому эти числа дают одинаковые остатки при делении на любое из n_i, так как n_i — делитель числа n.

Построим обратное отображение. Положим $m_i = \frac{n}{n_i}$ для всех n_i. Отсюда следует, что $m_i \equiv 0 \pmod{n_j}$ при $i \neq j$. Положим

$$c_i = m_i \left(m_i^{-1} \mod n_i\right)$$

для всех $i = 1, 2, \ldots, k$. Тогда $c_i \equiv 1 \pmod{n_i}$ и $c_i \equiv 0 \pmod{n_j}$ при $j \neq i$, и числу c_i поэтому соответствует набор с одной единицей на i-м месте:

$$c_i \leftrightarrow (0, 0, \ldots, 0, \underset{i}{1}, 0, \ldots, 0).$$

Тем самым, положив

$$a \equiv (a_1 c_1 + a_2 c_2 + \ldots + a_k c_k) \pmod{n},$$

мы получим число соответствующее набору $(a_1, a_2, \ldots, a_k)$. Следовательно, для каждого набора можно найти соответствующий элемент. Далее заметим, что если a и a' дают одинаковые остатки при делении на все n_i, то $a - a'$ делится на все n_i, а значит, и на n в силу взаимной простоты чисел n_i. Следовательно, отображение взаимно однозначно. □

Утверждение 3.31. *Если $n_1, n_2, \ldots, n_k$ попарно взаимно просты и $n = n_1 n_2 \ldots n_k$, то система сравнений*

$$x \equiv a_i \pmod{n_i}$$

относительно x (где $i = 1, \ldots, k$) имеет единственное решение по модулю n.

Утверждение 3.32. *Если $n_1, n_2, \ldots, n_k$ попарно взаимно просты, $n = n_1 n_2 \ldots n_k$ и x и a — целые числа, то свойство*

$$x \equiv a \pmod{n}$$

равносильно выполнению сравнений

$$x \equiv a \pmod{n_i}$$

при всех $i = 1, \ldots, k$.

Пример 3.33. Рассмотрим систему сравнений

$$\begin{aligned} a &\equiv 2 \pmod 5 \\ a &\equiv 3 \pmod{13}. \end{aligned}$$

Здесь $a_1 = 2, a_2 = 3, n_1 = m_2 = 5, n_2 = m_1 = 13, n = 65$, если сохранять обозначения китайской теоремы об остатках. Чтобы восстановить a по модулю $n = 65$, нам нужно найти числа c_i. Так как $13^{-1} \equiv 2 \pmod 5$ и $5^{-1} \equiv 8 \pmod{13}$ (как мы выяснили в предыдущем разделе, при помощи расширенного алгоритма Евклида мы можем находить обратный элемент (если он существует) за $O(\log^3 n)$ битовых операций), мы получаем

$$\begin{aligned} c_1 &= 13 \cdot (2 \pmod 5) = 26 \\ c_2 &= 5 \cdot (8 \pmod{13}) = 40, \end{aligned}$$

откуда

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65} \\ &\equiv 52 + 120 \pmod{65} \\ &\equiv 42 \pmod{65}. \end{aligned}$$

Хорошим введением в теорию чисел считается учебник [24], также рекомендуем читать [9].

Теперь рассмотрим теоретико-числовые алгоритмы. Начнем с тестов натурального числа на простоту. Нетрудно видеть, что алгоритм, проверяющий простоту n перебором всевозможных кандидатов в делители среди элементов отрезка $[1; \sqrt{n}]$, не полиномиален. Мы сперва проверим, что язык простых чисел лежит как в классе $\mathbf{co-NP}$, как и в классе $\mathbf{NP}$. Затем мы рассмотрим вероятностные алгоритмы проверки простоты, после чего докажем, что существует полиномиальный алгоритм, проверяющий простоту числа.

Теорема 3.34 (асимптотический закон распределения простых чисел)**.** *Пусть $\pi(n)$ — это количество простых чисел, не превосходящих n. Тогда для $n \geqslant 20$ имеем*

$$n\left(\log n + \log\log n - \frac{3}{2}\right) < p_n < n\left(\log n + \log\log n - \frac{1}{2}\right).$$

В частности,

$$\lim_{n\to\infty} \pi(n) \cdot \frac{\ln n}{n} = 1,$$

то есть простых чисел на отрезке $[1, n]$ асимптотически столько: $\frac{n}{\ln n}$.

Это утверждение мы оставим без доказательства, его можно прочитать в книге [23].

Полиномиальные арифметические алгоритмы

Договоримся, что в данном разделе мы будем использовать битовую сложность (то есть размер входа для нас — количество битов, использованных для записи чисел, число операций теоретико-числового алгоритма мы будем измерить в битовых операциях).

Число по умолчанию представлено в двоичной системе, длина числа n равна $\lfloor \log_2 n \rfloor + 1$. Полиномиальный алгоритм на числах должен работать за $O(\log^d n)$ для некоторого $d \in \mathbb{N}$; в частности, переборный алгоритм проверки числа n на простоту (проверяющий делимость n на числа из интервала $[1; \sqrt{n}]$) не полиномиален.

Сложение и умножение можно реализовать в столбик. Проверим, что эти реализации полиномиальны. Сложение в столбик двух чисел, не превышающих n, использует не более $2(\log_2 n + 1)$ операций сложения — не более 2 в каждом разряде. Умножение же в столбик двух чисел, не превышающих n, есть сложение не более $\log_2 n + 1$ двоичных чисел, каждое из которых не длинее $\log_2 n^2 + 1 = 2\log_2 n + 1$, то есть имеет сложность $O(\log^2 n)$. Сравнение двух двоичных чисел может быть проделано за $O(\log n)$ побитовых сравнений.

Деление с остатком двух целых чисел, реализованное в столбик, также полиномиально. Если $a = bq + r$, где $|r| < |b|$, то длина q не превышает $\log a - \log b$, следовательно, при делении a на b с остатком будет произведено не более $\log a - \log b$ вычитаний (возможно, что придется добавить цифру в конец, а не проделать вычитание) чисел длины $\log b$, то есть в итоге сложность деления с остатком есть $O(\log a \log b)$ — слагаемое $\log^2 b$ не превышает $\log a \log b$.

В дальнейшем при рассмотрении числовых алгоритмов будем по умолчанию считать, что длины входных чисел не ограничены.

Полиномиальность алгоритма Евклида

Теперь проверим полиномиальность алгоритма Евклида.

```
1: procedure EUCLID(a, b)
2:     if b = 0 then
3:         return a
4:     else
5:         return EUCLID(b, a mod b)
6:     end if
7: end procedure
```

Нам достаточно показать, что алгоритм Евклида совершается за полиномиальное число арифметических операций над a и b. Поскольку все умножения и деления с остатком делаются за не более чем $O(\log a \log b)$, а все сложения и вычитания делаются за не более чем $O(\log a)$, то из полиномиальности числа арифметических операций следует полиномиальность самого алгоритма.

Теорема 3.35. *Алгоритм Евклида вычисляет наибольший общий делитель a и b за $O(\log a + \log b)$ арифметических операций.*

Доказательство. Очевидно, что алгоритм Евклида остановится на любом входе. После каждого деления с остатком пара (x, y) заменяется на пару $(y, x \mod y)$, при этом

$$x = yq + (x \mod y) \geqslant y + (x \mod y) > 2(x \mod y)$$

Как следствие, $xy \geqslant 2y(x \mod y)$, то есть произведение элементов пары каждый раз уменьшается хотя бы вдвое. Как следствие, требуется $O(\log ab) = O(\log a + \log b)$ арифметических операций. □

Пусть F_k — k-е число Фибоначчи. Тогда справедлива следующая лемма.

Лемма 3.36. *Пусть $a > b \geqslant 0$. Если процедура Euclid(a, b) во время работы вызывает себя k раз $(k \geqslant 1)$, то $a \geqslant F_{k+2}$ и $b \geqslant F_{k+1}$.*

Доказательство. Доказательство проведём индукцией по k. Рассмотрим $k = 1$. Если происходит хотя бы один рекурсивный вызов, то $b > 0$, а значит, $b \geqslant 1 = F_2 \Longrightarrow a \geqslant 2 = F_3$, тем самым доказана база индукции. Пусть утверждение выполнено для $k - 1$ вызовов, покажем, что таким образом утверждение верно для k вызовов. На первом шаге процедура Euclid(a, b) вызывает процедуру Euclid$(b, a \mod b)$, внутри которой происходит $k-1$ рекурсивный вызов. Применим предположение индукции для Euclid$(b, a \mod b)$: $b \geqslant F_{k+1}$ и $(a \mod b) \geqslant F_k$. Из неравенства $a > b > 0$ следует, что $\left\lfloor \frac{a}{b} \right\rfloor \geqslant 1$ и $b + (a \mod b) = b\left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b\right) \leqslant a$, так что $a \geqslant b + (a \mod b) \geqslant F_{k+1} + F_k = F_{k+2}$, что и требовалось доказать. □

Из доказанной леммы получаем следующую теорему.

Теорема 3.37 (Ламе). *Пусть k — целое положительное число. Если $a > b \geqslant 0$ и $b < F_{k+1}$, то процедура Euclid(a, b) выполняет менее k рекурсивных вызовов.*

Поскольку F_k примерно равно $\frac{\varphi^k}{\sqrt{5}}$, где $\varphi = \frac{1+\sqrt{5}}{2}$ — «золотое сечение», то число рекурсивных вызовов для процедуры Euclid(a, b) составляет $O(\log b)$ при $a > b \geqslant 0$. Если процедура Euclid применяется к двум m-битовым числам, то ей приходится выполнять $O(m)$ арифметических операций, а значит, $O(m^3)$ битовых операций. На самом деле эта оценка достаточно грубая. Можно показать, что число битовых операций равняется $O(m^2)$.

Лемма 3.38. *Если d делит оба числа a и b, а также $d = ax + by$ для некоторых целых чисел x и y, то $d = \gcd ab$*

Доказательство. Так как d — общий делитель чисел a и b, то он не превосходит наибольшего общего делителя по определению: $d \leqslant \gcd ab$. С другой стороны, $\gcd ab$ делит a и b, а значит, делит и $ax + by = d$, то есть $\gcd ab \leqslant d$. Из этих двух неравенств получаем, что $d = \gcd ab$. □

Оказывается, что можно немного видоизменить процедуру Euclid(a, b) так, чтобы она возвращала и коэффициенты x и y указанной линейной комбинации.

```
1: procedure Extended-Euclid(a, b)
2:     if b = 0 then
3:         return (1, 0, a)
4:     end if
5:     (x', y', d) ← Extended-Euclid(b, a mod b)
6:     return (y', x' − ⌊a/b⌋ y', d)
7: end procedure
```

Лемма 3.39. *Для произвольных неотрицательных чисел a и b ($a \geqslant b$) расширенный алгоритм Евклида возвращает целые числа x, y, d, для которых* $\gcd ab = d = ax + by$.

Доказательство. Во-первых, если выкинуть из алгоритма x и y, то получим обычный алгоритм Евклида, поэтому он действительно вычисляет $d = \gcd ab$. Оставшееся утверждение докажем инукцией по b. База для $b = 0$ очевидна. Доказывая шаг индукции, воспользуемся предположением индукции для рекурсивного вызова Extended-Euclid($b, a \bmod b$) (это корректно, ведь $a \bmod b < b$):

$$\gcd ba \mod b = bx' + (a \mod b)y'.$$

Перепишем $(a \bmod b)$ как $\left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b\right)$ и получим:

$$\begin{aligned} d &= \gcd ab = \gcd ba \mod b = bx' + (a \mod b)y' \\ &= bx' + \left(a - \left\lfloor \tfrac{a}{b} \right\rfloor \cdot b\right) y' = ay' + b\left(x' - \left\lfloor \tfrac{a}{b} \right\rfloor \cdot y'\right). \end{aligned}$$

Итак, $d = ax + by$ при $x = x'$ и $y = x' - \left\lfloor \frac{a}{b} \right\rfloor \cdot y'$, что и требовалось доказать. □

При помощи расширенного алгоритма Евклида можно решать линейные диофантовы уравнения. Следующая процедура по целым числам a, b и $n > 0$ даёт все решения уравнения $ax \equiv b \pmod n$.


```
1: procedure Modular-Linear-Equation-Solver(a, b, n)
2:     (d, x', y') ← Extended-Euclid(a, n)
3:     if d | b then
4:         x_0 ← x' · b/d mod n
5:         for i ← 0 to d − 1 do
6:             print (x_0 + i · n/d) mod n
7:         end for
8:     else
9:         print «нет решений»
10:    end if
11: end procedure
```

Процедура Modular-Linear-Equation-Solver(a, b, n) выполнит $O(\log n)$ арифметических операций в строке 2 и $O(\gcd an)$ операций в остальных строках, то есть всего $O(\log n + \gcd an)$. Но если нам достаточно найти все решения, а не печатать их на экран, то число

арифметических операций будет $O(\log n)$, а битовых операций будет $O(\log^3 n)$, т.к. процедура Extended-Euclid(a, n) делает $O(\log n)$ делений целых чисел длины $O(\log n)$ с остатком, что делается за $O(\log^2 n)$.

Отметим ещё два важных следствия.

Вывод 3.40. Пусть $n > 1$. Если $\gcd an = 1$, то уравнение $ax \equiv b \pmod n$ имеет единственное решение в $\mathbb{Z}_n$. В частности, если $b = 1$ и $\gcd an = 1$, то уравнение $ax \equiv 1 \pmod n$ имеет единственное решение в $\mathbb{Z}_n$. При $\gcd an > 1$ уравнение $ax \equiv 1 \pmod n$ решения не имеет.

Итак, деление по модулю n и вычисление обратного по модулю n имеют сложность $O(\log^3 n)$.

Напоследок рассмотрим *бинарный алгоритм Евклида*. Идея его в следующем: поиск наибольшего общего делителя x и y можно свести к поиску НОДа их максимальных нечетных делителей $\widetilde{x}$ и $\widetilde{y}$, вынеся общую степень двойки, а $\widetilde{x}$ и $\widetilde{y}$ можно заменить на $\min(\widetilde{x}, \widetilde{y})$ и $|\widetilde{x} - \widetilde{y}|$; дальше с полученной парой проделываются те же операции, пока не получится пара вида $(x, 0)$ или $(0, y)$.

Имплементация следующая:

procedure Binary-Euclid(x, y)
  $d_x = \max\{d \in \mathbb{N} | x \vdots 2^d\}$
  $d_y = \max\{d \in \mathbb{N} | y \vdots 2^d\}$
  $a = x2^{-d_x}$, $b = y2^{-d_y}$
  if $b < a$ **then**
    поменять местами a и b
  end if
  while $b > 1$ **do**
    $a = a - b$
    $b = \min(a, b)$
    while $a \vdots 2$ **do**
      $a /= 2$
    end while
    while $b \vdots 2$ **do**
      $b /= 2$
    end while
  end while
  return $a2^{\min(d_x, d_y)}$
end procedure

Продемонстрируем работу этого алгоритма на одном примере. Пусть $x = 36$, $y = 90$; тогда $d_x = 2$, $d_y = 1$, и мы переходим к паре $(a, b) = (9, 45)$. Затем мы заменяем два числа на их разность и минимальное, то есть после этого шага $(a, b) = (9, 36)$. Но это еще не все: 9 не является четным, а 36 — является, поэтому мы делим его на 2, пока не получим нечетное число. Итого мы получили $(a, b) = (9, 9)$. Следующим же шагом мы получим $(a, b) = (9, 0)$; финальный результат есть $9 \cdot 2^{\min(2,1)} = 18$. Конечно же, $\gcd(36, 90) = 18\,\gcd(2, 5) = 18$.

Докажем корректность этого алгоритма. z делится на максимально возможную степень двойки, делящую и x, и y. Таким образом, у чисел a и b НОД нечетен и, следовательно, $\gcd(a, b) = \gcd(\frac{a}{2}, b)$ [если a четный] либо $\gcd(a, b) = \gcd(a, \frac{b}{2})$ [если b четный]. Замена a, b на $\min(a, b), |a - b|$ сохраняет НОД: если $a, b \vdots d$, то $\min(a, b) \vdots d$ и $|a - b| \vdots d$, обратное также верно [по аналогичным причинам]. Последняя полученная пара есть пара вида $(a, 0)$, ее НОД равен a, таким образом, $a2^{\min(d_x, d_y)} = \gcd(x, y)$.

Основной пафос подобной модификации алгоритма Евклида в том, что делить на два можно исключительно быстро: надо лишь убрать один ноль в конце двоичной записи числа. Пусть n — максимальная длина одного из двух чисел, тогда на каждом шагу хотя бы одно из чисел уменьшается хотя бы на 1, нам придется сделать $O(n)$ итераций большого цикла в алгоритме. Каждая итерация есть вычисление $a - b$, минимума из двух чисел a и b и возможное деление на степень двойки; вычитание в столбик имеет сложность $O(n)$ битовых операций, вычисление минимума требует сравнения двух чисел длины не длиннее n, которое не сложнее $O(n)$, а деление на степень двойки можно осуществить, найдя последний бит, не равный 0, что тоже можно осуществить за $O(n)$ битовых операций. В конце нужно будет дописать сокращенное в начале работы алгоритма количество нулей, их же не более n, ведь $2^{\min(d_x, d_y)} \leqslant \max(x, y)$. Итого сложность бинарного алгоритма Евклида можно ограничить $O(\max(\log x, \log y)^2)$.

Быстрое умножение чисел и матриц

Пусть даны два больших числа x, y в бинарной записи, длины n. Наивный метод умножения, «в столбик», требует, как мы видели $\Theta\left(n^2\right)$ операций. Однако существуют и более быстрые алгоритмы перемножения двух чисел, например, описанный ниже *алгоритм Карацубы*. Представим x и y в виде $x = a2^k + b$, $y = c2^k + d$, где $b, d < 2^k$. По сути, мы разбили бинарную запись каждого числа на две части: длины k и $n - k$. Тогда

$$xy = (a2^k + b)(c2^k + d) = ac2^{2k} + (ad + bc)2^k + bd = ac2^{2k} + \big((a + b)(c + d) - ac - bd\big)2^k + bd.$$

Так как сложение и вычитание чисел осуществляется за линейное по длине время, а умножение на степень двойки в бинарной записи заключается просто в дописывании нужного числа нулей, то основная задача сводится к умножению трех пар чисел: ac, bd, $(a+b)(c+d)$. Взяв $k = \lfloor \frac{n}{2} \rfloor$, для сложности алгоритма получаем следующее рекуррентное

уравнение:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lceil \frac{n}{2} \right\rceil + 1\right) + O(n), n \geq 2, \tag{12}$$

$$T(1) = C. \tag{13}$$

По теореме Акра–Баззи $T(n) = \Theta\big(n^{\log_2 3} + n^{\log_2 3}\Theta(1)\big) = \Theta\big(n^{\log_2 3}\big)$. Поэтому этот алгоритм действительно асимптотически быстрее, чем наивное умножение.

Оказывается, подобные идеи можно использовать и для быстрого умножения матриц. Рассмотрим две матрицы A, B размера $n \times n$, и $C := AB$. Так как

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j},$$

то обычное умножение требует $\Theta\left(n^3\right)$ операций (умножений и сложений). Быстрое умножение матриц с помощью *алгоритма Штрассена* заключается в следующем. Пусть $n = 2^s$. Представим матрицы в блочном виде:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix},\ B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix},\ C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix},$$

где $A_{i,j}, B_{i,j}, C_{i,j} \in \mathbb{Q}^{n/2 \times n/2}$. Тогда $C_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j}$. Определим матрицы

$$\begin{aligned} P_1 &:= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}), \\ P_2 &:= (A_{2,1} + A_{2,2})B_{1,1}, \\ P_3 &:= A_{1,1}(B_{1,2} - B_{2,2}), \\ P_4 &:= A_{2,2}(B_{2,1} - B_{1,1}), \\ P_5 &:= (A_{1,1} + A_{1,2})B_{2,2}, \\ P_6 &:= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}), \\ P_7 &:= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}). \end{aligned}$$

Непосредственно можно проверить, что

$$\begin{aligned} C_{1,1} &= P_1 + P_4 - P_5 + P_7, \\ C_{1,2} &= P_3 + P_5, \\ C_{2,1} &= P_2 + P_4, \\ C_{2,2} &= P_1 - P_2 + P_3 + P_6. \end{aligned}$$

Таким образом, для перемножения двух матриц размера $n \times n$ требуется 7 умножений и 18 сложений матриц размера $\frac{n}{2} \times \frac{n}{2}$. Сложение выполняется за $O\left(n^2\right)$, поэтому получается следующая рекуррента:

$$T(n) = 7T\left(\frac{n}{2}\right) + O\left(n^2\right).$$

Ее решение: $T(n) = \Theta\left(n^{\log_2 7}\right)$. $\log_2 7 \approx 2.807 < 3$, поэтому алгоритм Штрассена асимптотически быстрее обычного умножения.

Быстрое возведение в степень

Пусть нужно вычислить a^k. Можно просто n раз перемножить a, но можно сделать это и быстрее. Рассмотрим n в двоичном представлении:

$$n = \sum_{i=0}^{k} x_i 2^i = x_0 + 2\big(x_1 + 2(\dots)\big),$$

где $k = \lfloor \log n \rfloor$. Так как $(a^s)^2 = a^{2s}$, то естественным образом возникает идея следующего алгоритма: пусть $b_0 = a^{x_k} = a$, $b_j = a^{x_{k-j}}(b_{j-1})^2$, $1 \leq j \leq k$. По индукции несложно показать, что

$$b_j = a^{\sum_{i=k-j}^{k} x_i 2^{i+j-k}}.$$

Следовательно, $b_k = a^n$, и для его вычисления требуется не больше $2\lfloor \log n \rfloor$ умножений (сюда входят и возведения в квадрат).

В работе теоретико-числовых алгоритмов часто нужно уметь возводить некоторое число в степень по некоторому модулю n. Причём хочется делать это за полиномиальное время. На первом семинаре мы касались алгоритма быстрого возведения в степень. Напомню основные моменты. Пусть мы хотим вычислить a^m. Для этого представим число m в двоичной системе счисления: $m = (\overline{m_k m_{k-1} \dots m_0})_2 = m_k \cdot 2^k + m_{k-1} \cdot 2^{k-1} + \dots + m_1 \cdot 2 + m_0$, где $m_i \in \{0, 1\}$. Тогда

$$a^m = a^{((\dots((m_k \cdot 2 + m_{k-1}) \cdot 2 + m_{k-2}) \cdot 2 \dots) \cdot 2 + m_1) \cdot 2 + m_0} = ((\dots(((a^{m_k})^2 \cdot a^{m_{k-1}})^2 \dots)^2 \cdot a^{m_1})^2 \cdot a^{m_0}.$$

Тогда последовательность действий следующая: изначально у нас текущее число равно a (что соответствует числу $a^{m_k} = a$) и $i = k$; затем мы возводим текущее число в квадрат и если $m_{i-1} = 1$, то домножаем результат на a, а если $m_i = 0$, то оставляем как есть; уменьшаем счётчик: $i := i - 1$. При таком подходе происходит $\lfloor \log_2 m \rfloor + \eta(m) - 1$ умножений при вычислении a^m, где $\eta(m)$ — количество единиц в двоичной записи m. Поскольку $\eta(m) \leqslant \log_2 m + 1$, то сложность быстрого возведения есть $O(\log m)$.

Быстрое возведение в степень по модулю n основано на той же самой идее, но теперь после каждого умножения нужно ещё делить результат на n, чтобы найти остаток по модулю n. Алгоритм приведён ниже.

1: **procedure** MODULAR-EXPONENTIATION(a, m, n)
2: $c \leftarrow 0$
3: $d \leftarrow 1$
4: пусть $(\overline{m_k m_{k-1} \dots m_0})_2$ — двоичная запись числа m
5: **for** $i \leftarrow k$ *downto* 0 **do**
6: $c \leftarrow 2c$

```
        d ← (d · d) mod n
        if b_i = 1 then
            c ← c + 1
            c ← (d · a) mod n
        end if
    end for
    return d
end procedure
```

Если a, m и n имеют двоичную запись длины не более β, то число арифметических операций есть $O(\beta)$, а число битовых — $O(\beta^3)$.

Нетрудно видеть, что быстрое возведение в степень не является оптимальным: возвести число в степень 54 можно, произведя 7 умножений:

$$a^2 = a \cdot a, \quad a^4 = a^2 \cdot a^2, \quad a^8 = a^4 \cdot a^4, \quad a^{16} = a^8 \cdot a^8$$
$$a^{18} = a^{16} \cdot a^2, \quad a^{36} = a^{18} \cdot a^{18}, \quad a^{54} = a^{36} \cdot a^{18}$$

Естественный вопрос о минимальном количестве умножений, необходимых для возведения в степень n, приводит к изучению аддитивных цепочек.

Определение 3.41. *Аддитивная цепочка* числа n — последовательность $1 = a_0, a_1, \ldots a_k$ такая, что любое a_i, кроме нулевого, выражается как сумма двух предыдущих элементов (возможно, одинаковых) последовательности:

$$\forall i \in [1; k]\ \exists j, l < i \quad a_i = a_j + a_l$$

По аддитивной цепочке можно построить последовательность $(i_1, j_1), \ldots (i_k, j_k)$, в которой $j_r \leqslant i_r < r$. Число k называется *длиной аддитивной цепочки*.

Определение 3.42. Определим $l(n)$ как длину кратчайшей аддитивной цепочки числа n.

Из корректности алгоритма быстрого умножения следует, что $l(n) \leqslant \lfloor \log_2 n \rfloor + \eta(n)$ и $l(n) = O(\log n)$. Нетрудно заметить также, что $l(n) \geqslant \lfloor \log_2 n \rfloor$. Точная формула $l(n)$ остается неизвестной, равно истинность утверждения $l(2^n - 1) = l(n) + n - 1$ (гипотеза Брауэра-Шольца). Известно, однако, что $l(n)$ не монотонная функция, например, неверно, что $l(2n) \geqslant l(n)$ для всех n: для $n = 375494703$ имеем $34 = l(2n) < l(n) = 35$ (см. [8]). Достаточно точную оценку сверху можно получить с помощью следующего утверждения:

Теорема 3.43 (Brauer, 1939)**.** *Для любого натурального $k > 1$ выполнено следующее неравенство:*

$$l(n) \leqslant \left(1 + \frac{1}{k}\right) \log_2 n + 2^k$$

Доказательство. По заданному k мы явно построим аддитивную цепочку числа n и покажем, что ее длина ограничена правой частью неравенства в формулировке теоремы. Пусть $m = 2^k$, а

$$n = \sum_{i=0}^{\log_{2^k} m+1} d_{t-i} m^i$$

— m-ичное представление числа n. Рассмотрим последовательность

$$1, \dots m-1,$$
$$d_0, 2d_0, \dots md_0, md_0 + d_1, 2(md_0 + d_1), \dots m^2 d_0 + md_1, m^2 d_0 + md_1 + d_2, \dots$$

Мы сначала получаем всевозможные значения d_i, а затем, их используя, «выращиваем» число в m-ичной записи, начиная с младшего разряда. Формально говоря, пусть (a_l) и (b_l) — последовательности, заданные формулами

$$a_i = i,\ i \in [1; m], \qquad b_{(k+1)i+j} = 2^j \left(m^i d_0 + \dots + d_i m\right),\ i \in [0; t], j \in [1; k+1]$$

Нетрудно видеть, что последовательность $a_1, \dots a_m, b_1, \dots$ является аддитивной цепочкой: так как все d_i лежат в $[1; m-1]$, то либо $b_i = 2b_{i-1}$, либо $b_i = b_{i-1} + a_j$ для некоторого j, а $b_1 = 2a_j$. Очевидно также, что последний элемент последовательности равен $2^{tk} d_0 + \dots + d_t = n$, а длина всей последовательности равна сумме длин последовательнсотей (a_i) и (b_i), то есть не превышает

$$m - 1 + (k+1)t = 2^k - 1 + (k+1)(\lfloor \log_{2^k} n \rfloor + 1) = 2^k + (k+1)\lfloor \log_{2^k} n \rfloor \leqslant 2^k + \frac{k+1}{k} \lfloor \log_2 n \rfloor$$

□

Если в теореме выше принять $k = \log\log n + 1$, то можно получить ([6])

$$l(n) \leqslant \log_2 n \left(1 + \frac{1}{\log\log n} + \frac{2\log 2}{(\log n)^{1-\log 2}}\right)$$

Полиномиальность алгоритма Гаусса

Рассмотрим следующий *алгоритм Гаусса*, позволяющий привести прямоугольную матрицу к ступенчатому виду.

```
1: procedure GAUSS((A_ij) ∈ Mat_{n×m}(ℚ))
2:     for t = 0; t ⩽ m; t+ = 1 do k = min{i ∈ [t + 1; n] | A_it ≠ 0}
3:         for i = 0; i ⩽ n; i+ = 1 do поменять местами A_ti и A_ki
4:         end for
5:         for i = t + 1; i ⩽ m; i+ = 1 do
6:             for j = 1; j ⩽ n; j+ = 1 do A_ij − = A_tj · A_it / A_tt
7:             end for
8:         end for
```

9: **end for**
10: **return** (A_{ij})
11: **end procedure**

Пока предполагаем, что рациональные дроби не сокращаются.

Вспомним, что арифметические операции на дробях определены так:

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+bc}{bd},\quad \frac{a}{b}\cdot\frac{c}{d}=\frac{ac}{bd}$$

Мы не производим сокращения, а все $|A_{ij}|\leqslant h$, поэтому после вычитания первой строчки из остальных числители и знаменатели элементов во всех строчках, начиная со второй, будут ограничиваться по модулю h^2. Следующая итерация, в ходе которой вторая [после соответствующей перестановки] строчка вычитается из всех последующих, увеличит максимально возможную степень на 2. Вообще из вышеприведенной формулы очевидно, что если $|a|,|b|,|c|,|d|\leqslant x$, то числители и знаменатели суммы и произведения дробей ограничены сверху x^2. Это означает, что каждая следующая итерация алгоритма Гаусса удваивает степень h, которой ограничиваются числители и знаменатели дробей. Ясно, что ненулевых строчек будет получаться не более $\mathrm{rk}A$, остальные строчки будут линейными комбинациями первых $\mathrm{rk}A$ строк и занулятся при преобразованиях строчек. Соответственно, числители и знаменатели дробей ограничены $h^{2^{\mathrm{rk}A}}$; изначально $\mathrm{rk}A\leqslant\min(m,n)$ и максимально возможное ограничение есть $h^{2^{\min(m,n)}}$. В частности, для квадратных матриц получается ограничение h^{2^n}.

Итак, выше получено слишком быстрорастущее ограничение, в частности, не являющееся полиномиальным по n. Поэтому для больших n и приходится использовать алгоритм Евклида для сокращения дробей: после сразу же вычисления p_{ij}, q_{ij} мы вычисляем $\gcd(p_{ij},q_{ij})$ и делим на них.

Сначала докажем, что все числители и знаменатели будут ограничены по модулю некоторым полиномом от m,n,h. Пусть ведущие элементы находятся на главной диагонали [в противном случае переставим соотвествующим образом строки]. Пусть $(A_{ij^{(k)}})$ — матрица, полученная из A на k-ой итерации, $D^{(k)}=(A_{ij})_{i,j\in[1;k]}$ — подматрица A [оригинальной], состоящая из пересечений первых k строк и k столбцов, $D_{ij}^{(k)}$ (для $i,j\in[k{+}1;n]$) — матрица, составленная из пересечений строк с номерами $1,2,\ldots,k,i$ и столбцов с номерами $1,2,\ldots,k,j$, тогда

$$A_{ij}^{(k)}=\frac{\det D_{ij}^{(k)}}{\det D^{(k)}}$$

Действительно, после применения алгоритма Гаусса к $D_{ij}^{(k)}$ получим на диагонали элементы $A_{11}^{(n)},\ldots,A_{kk}^{(n)},A_{ij^{(k)}}$: диагональные элементы $A_{ii}^{(n)}$ будут получены на i-ой итерации и не будут впоследствии, в последней же строчке единственным ненулевым элементом будет $A_{ij^{(k)}}$; заметим, что, применяя алгоритм Гаусса к $D^{(k)}$, получим на диагонали

$A_{11}^{(n)}, \ldots, A_{kk}^{(n)}$, а детерминант треугольной матрицы равен произведению диагональных элементов.

В доказанной выше формуле $D_{ij}^{(k)}$ и $D^{(k)}$ — подматрицы оригинальной матрицы, поэтому их детерминанты ограничены сверху $k!h^k$, для записи числителей и знаменателей потребуется не более

$$\log(k!h^k) \leqslant \log(N!h^N) = \log(N!) + N\log(h) \leqslant N(\log(N) + \log(h))$$

символов, где $N = \max(m, n)$, что есть полином от $m, n, \log(h)$. Соответственно, если $\widetilde{A}_{ij} = \frac{p_{ij}}{q_{ij}}$, то $p_{ij} \leqslant \det D_{ij}^{(k)}$, а $q_{ij} \leqslant \det D^{(k)}$ — отношение детерминантов может быть как несократимой, так и сократимой дробью.

Теперь осталось оценить сложность, мы предъявим O-асимптотику сложности.

Сразу замеитим, что мы считаем $A_{ij} = \frac{p_{ij}}{q_{ij}}$, а раз на каждой итерации происходит

$$\frac{p_{ij}}{q_{ij}} \mapsto \frac{p_{ij}}{q_{ij}} - \frac{p_{tj}}{q_{tj}} \cdot \frac{p_{it}}{q_{it}} \frac{q_{tt}}{p_{tt}},$$

то

$$\begin{cases} p_{ij} \mapsto p_{ij}q_{tj}q_{it}p_{tt} - q_{ij}p_{tj}p_{it}p_{tt} \\ q_{ij} \mapsto q_{ij}q_{tj}q_{it}p_{tt} \end{cases}$$

Из доказанного выше следует, что все числители и знаменатели можно ограничить сверху $N(\log(N) + \log(h))$ символов, где $N = \max(m, n)$: после арифметических операций, преобразующих A_{ij}, числители и знаменатели приводятся. Поэтому произведение четырех чисел длины не более $N(\log(N)+\log(h))$ имеет сложность $O(N^2\log^2(Nh))$ битовых операций: перемножаются две пары чисел длины $N\log(Nh)$, а затем — два полученных числа, имеющие длину $2N\log(Nh)$ [квадрат числа длины $N\log(Nh)$ имеет длину $2N\log(Nh)$]. Сложение двух чисел длины $O(N\log(Nh))$ имеет сложность $N\log(Nh)$. Соответственно, вычисление неприведенных числителей и знаменателей имеет сложность $O(N^2\log^2(Nh))$. Сложность приведения оценим грубо: алгоритм Евклида для чисел длины $N\log(Nh)$ выполняется за $O(N\log(Nh))$ арифметических операций, а каждая операция, будь то сложение, вычитание или деление с остатком, не сложнее $O(N^2\log^2(Nh))$ [квадратичное время занимает деление с остатком], поэтому приведение можно сделать за $O(N^3\log^3(Nh))$ битовых операций.

Такие операции проводятся с A_{ij} для $i \in [t+1; m]$ и $j \in [1; n]$, то есть над $(m-t)n$ элементами на t-ой итерации, а t пробегает от 0 до $\mathrm{rk}A$, то есть таких операций не более $n\sum_{t\in[0;m]}(m-t) \leqslant \frac{1}{2}nm^2$. Таким образом, арифметика в алгоритме Гаусса имеет битовую сложность $O(nm^2N^3\log^3(Nh))$ операций.

Последний шаг — оценка вклада выбора нужной строчки и перестановка строчек; строчку можно выбрать за $O(m)$ битовых операций [надо просмотреть $m-t$ элементов и выбрать ненулевой либо заявить, что таких не бывает], а переставить строчки можно за $O(nN\log(Nh))$ битовых операций [ведь поменять местами A_{ti} и A_{ki} можно за

$O(N\log(Nh))$ битовых операций, поменяв биты местами]. Таким образом, битовая сложность алгоритма Гаусса равна

$$O\Big(nm^2\max(m,n)^3\big(\log h+\log\max(m,n)\big)^3\Big)$$

Заметим, что арифметических операций с элементами матрицы всего $O(nm^2)$.

Нетрудно видеть, что система линейных уравнений $Ax=b$ над $\mathbb{Q}$ также может быть решена за полиномиальное время. Применяя метод Гаусса последовательно к строчкам и стообам матрицы A, мы можем получить, что

$$\underbrace{\begin{bmatrix}1&&&&\\ *&1&&0&\\ *&*&1&&\\ *&*&*&1&\\ *&*&*&*&1\end{bmatrix}}_{\Delta_A}A\underbrace{\begin{bmatrix}1&*&*&*&*\\ &1&*&*&*\\ &&1&*&*\\ &0&&1&*\\ &&&&1\end{bmatrix}}_{\nabla_A}=\begin{bmatrix}d_1&&&&\\ &d_2&&0&\\ &&\ddots&&\\ &0&&\ddots&\\ &&&&d_n\end{bmatrix},$$

поэтому система уравнений $Ax=b$ эквивалентна системе линейных уравнений

$$\operatorname{diag}(d_1,\dots d_n)(\nabla_A)^{-1}y=\Delta_A^{-1}b$$

Если $d_i\neq 0$, то $((\nabla_A)^{-1}y)_i=\frac{(\Delta_A^{-1}b)_i}{d_i}$. В противном случае получим, что

- если $(\Delta_A^{-1}b)_i\neq 0$, то решения системы нет;
- если $(\Delta_A^{-1}b)_i=0$, то i-ый столбец матрицы $\nabla_A x$ является одним из базисных векторов пространства решений системы $Ax=0$, поэтому для частного решения $Ax=b$ берем $((\nabla_A)^{-1}x)_i=0$, а i-ый столбец матрицы $\nabla_A x$ добавляем к множеству X.

В итоге имеем получим частное решение и базис пространства решений системы $Ax=0$, что дает полное описание решений системы.

Простейшие криптографические протоколы

Полиномиального алгоритма факторизации натурального числа пока не существует, и на этой «сложности» факторизации основаны простейшие крипотграфические схемы.

Постановка вопроса такова: две стороны, условно Алиса и Боб, должны договориться о протоколе обмена информацией (закодированной целым числом), который был бы как можно более криптоустойчив, то есть чтобы дешифровка сообщений сторонним участником, условно Евой, или имитация ей одной из сторон (то есть Ева может переписываться с Алисой «от имени Боба») была как можно более тяжелой.

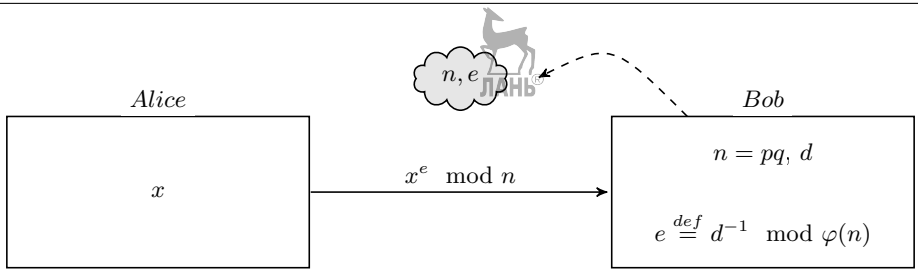
В симметрической криптосистеме Алиса и Боб используют один и тот же ключ для шифрования и дешифрования сообщения, передавать этот ключ они могут лишь по открытому каналу. Диффи и Хеллман предложили достаточно крипотустойчивую симметрическую схему.

(1) Алиса и Боб выбирают простое число p и первообразный корень g по этому модулю. Числа p и g публикуются в открытый доступ. Сообщения, которыми Алиса и Боб будут обмениваться в обе стороны, будут иметь вид $g^{mn} \mod p$.
(2) Алиса секретно выбирает число n и *по открытому каналу* посылает Бобу $g^n \mod p$.
(3) Боб секретно выбирает число m и по открытому каналу посылает Алисе $g^m \mod p$.
(4) Боб дешифрует сообщение Алисы $s = (g^n)^m \mod p$, Алиса дешифрует сообщение Боба $s = (g^m)^n \mod p$.

Поскольку g^m и g^n посылаются по открытому каналу, они видны всем. Чтобы вычислить сообщение Алисы и Боба, Ева должна по $g^m \mod p$ и $g^n \mod p$, не зная m и n, вычислить $g^{mn} \mod p$. Вычисление дискретного логарифма пока нельзя осуществить полиномиальной процедурой. Формально, впрочем, вычисление $g^{mn} \mod p$ по $g^m \mod p$ и $g^n \mod p$ — не вычисление дискретного логарифма, эквивалентность этих процедур пока не установлена.

Если Ева может не только подслушивать, но и выступать активным агентом, то она может, например, перехватывать и подменять сообщения — например, послать Бобу от имени Алисы некоторое $g^t \mod p$ и получить секретный ключ $g^{tm} \mod p$ для дешифровки сообщений Боба. Возникает проблема идентификации участников коммуникации. Она может быть решена асимметричной криптографической схемой, например, RSA.

(1) Боб выбирает два простых числа $p \neq q$ и вычисляет $n = pq$. Сообщения, которые Боб будет принимать, будут остатками по модулю n.
(2) Он же выбирает *секретный ключ* – d (он известен только ему).
(3) Боб вычисляет *открытый ключ* $e = d^{-1} \mod (p-1)(q-1)$.
(4) В открытый доступ публикуются числа $\{e, n\}$.
(5) Если Алиса хочет послать секретное сообщение x Бобу, то она проводит шифровку $x \to e(x) = x^e \mod n$ и посылает $e(x)$ по открытому каналу.
(6) Боб дешифрует сообщение с помощью секретного ключа $d(e(x)) \to (e(x))^d \mod n = x \mod n$.



На картинке открытые n, e — открытая информация Боба: каждый адресат генерирует свой модуль и свой открытый ключ. Таким образом, в схеме RSA «нехороший человек» Ева не сможет перехватывать и подменять сообщения Бобу, поскольку для «прохождения авторизации» ей нужно найти делители n. Криптоустойчивость схемы основана на предполагаемой сложности задачи факторизации: для вычисления $\varphi(n)$ как правило нужно разложение n на множители [пока не явится иного способа], а полиномиального алгоритма факторизации пока не существует.

Задача 1. **(a)** Пусть открытый ключ Боба $(25, 2021)$. Он хочет послать сообщение (число) за своей подписью. В какую степень он должен его возвести?

(b) Докажите или опровергните, что кодирование в системе RSA $M \to M^e \mod n$ биективно отображает отрезок $\{0, \ldots, n-1\}$ в себя.

Решение. **(a)** Необходимо вычислить d. Само число придется разлагать на множители каким-нибудь дурацким способом, например, перебирая делители, не превышающие $\lceil\sqrt{2021}\rceil = 44$. Выйдет, что $2021 = 43 \cdot 47$. Осталось решить только $25d = 42 \cdot 46$; надо будет расписать расширенный алгоритм Евклида для 25 и $42 \cdot 46 = 1932$:

$$\begin{aligned} 1932 &= 25 \cdot 77 + 7 \\ 25 &= 7 \cdot 3 + 4 \\ 7 &= 4 \cdot 1 + 3 \\ 4 &= 3 \cdot 1 + 1 \\ 3 &= 1 \cdot 3, \end{aligned}$$

тогда

$$1 = 4 - 3 \cdot 1 = 4 - (7 - 4 \cdot 1) \cdot 1 = 4 \cdot 2 - 7 \cdot 1 =$$
$$= (25 - 7 \cdot 3) \cdot 2 - 7 \cdot 1 = 25 \cdot 2 - 7 \cdot 7 = 25 \cdot 2 - (1932 - 25 \cdot 77) \cdot 7 = 25 \cdot (2 + 7 \cdot 77) - 1932 \cdot 7,$$

и $d = 2 + 7 \cdot 77 = 541$.

(b) Здесь важно, что в системе RSA $(e, \varphi(n)) = 1$, в противном случае контрпример строится тривиально. Разумеется, отображение $M \to M^e \mod n$ является

биекцией отрезока $\{0,\ldots,n-1\}$ в себя: отображение $M \to M^d \mod n$ является обратным для данного отображения,

- для $x \in \{0,\ldots,n-1\}$ взаимно простых с n все верно сразу, так как $x^{de} = x \mod n$ в силу теоремы Эйлера;
- для $x = pk \in \{0,\ldots,n-1\}$ все получается в силу того, что $de = 1 \mod (p-1)$, и $x^{de} = x \mod n$ в силу малой теоремы Ферма;
- случай $x = qk \in \{0,\ldots,n-1\}$ разбирается аналогично.

□

Дискретное преобразование Фурье

Нетрудно видеть, что обычное перемножение многочленов $a_0 + \ldots + a_n x^n$ и $b_0 + \ldots + b_n x^n$ потребует выполнения $\Theta(n^2)$ арифметических операций. Действительно, поскольку

$$(a_0 + \ldots + a_n x^n)(b_0 + \ldots + b_n x^n) = \sum_{k=0}^{2n} x^k \sum_{i=0}^{k} a_i b_{k-i},$$

а вычисление $\sum_{i=0}^{k} a_i b_{k-i}$ требует $2k-1$ арифметических операций, то и сама сложность обычного перемножения многочленов квадратична по n. Между тем, если известны значения многочленов $a_0 + \ldots + a_n x^n$ и $b_0 + \ldots + b_n x^n$ в $2n+2$ точках $\xi_1, \ldots x_{2n+2}$, то значения $(a_0 + \ldots + a_n x^n)(b_0 + \ldots + b_n x^n)$ в точках $\xi_1, \ldots x_{2n+2}$ можно вычислить за $O(n)$. Если удастся найти «хороший» набор точек $\{\xi_i\}$ и найти субквадратичную (то есть имеющую сложность $o(n^2)$) процедуру вычисления набора значений многочлена в этих точках (а также восстановления многочлена по набору значений), то можно построить процедуру, перемножающую многочлены степени не более n за субквадратичное время. Мы покажем, что в качестве набора чисел можно взять $e^{2\pi p/q}$, а соответствующая процедура перехода к набору значений имеет сложность $O(n \log n)$.

Определение 3.44. *Дискретное преобразование Фурье* — это отображение $F_n : \mathbb{C}^n \to \mathbb{C}^n$, ставящее в соответствие вектору $a = (a_0, a_1, \ldots, a_{n-1})^\top$ вектор $y = (y_0, y_1, \ldots, y_{n-1})^\top$, где $y_j = f(\omega_n^j)$, $j = 0, 1, \ldots, n-1$ и

$$f(x) = \sum_{j=0}^{n-1} a_j x^j.$$

Иными словами, мы поставили в соответствие вектору коэффициентов многочлена f вектор значений полинома f в корнях степени n из единицы. Вектор y называется дискретным преобразованием Фурье вектора a и обозначается $y = DFT_n(a)$.

Преобразование Фурье является черезвычайно важным инструментом в гармоническом анализе, при исследовании свойств функций и других преобразований, а также уравнений в частных производных. Оказывается, его можно определить не только над

функциями, но и над дискретными объектами, т.е. векторами. Дискретное преобразование Фурье — естественный аналог непрерывного преобразования. На самом деле, в качестве множителей, на которые умножаются элементы a_j, можно выбрать не только $e^{\frac{2\pi i}{n}jk}$, но и числа вида ω_n^{jk}, где ω_n — корень n-й степени из 1. Как несложно видеть, в поле комплексных чисел $\mathbb{C}$ все возможные ω_n имеют вид $e^{\frac{2\pi i}{n}s}$, где s взаимно просто с n. Если не оговорено обратное, мы будем использовать именно $\omega_n = e^{\frac{2\pi i}{n}}$. Заметим также, что если $m \mid n$, то ω_n^m является корнем из единицы степени $\frac{n}{m}$.

Приведем некоторые свойства ДПФ.

(1) Непосредственно из определения следует, что F_n — линейное преобразование с матрицей

$$\Xi := \left(\omega_n^{kj}\right)_{k,j=0}^{n-1}.$$

(2) F_n обратимо, и обратное преобразование F_n^{-1} является линейным с матрицей

$$\Xi^{-1} = \frac{1}{n}\left(\omega_n^{-kj}\right)_{k,j=0}^{n-1},$$

т.к. все столбцы (и строки) матрицы Ξ ортогональны:

$$\sum_j \omega_n^{kj}(\omega_n^{sj})^* = \sum_{j=0}^{n-1}(\omega_n^{k-s})^j = \begin{cases} n, & k = s, \\ \frac{(\omega_n^{k-s})^n - 1}{\omega_n^{k-s} - 1} = 0, & k \neq s. \end{cases}$$

(3) Для $a \in \mathbb{C}^n$ определим полином $P_a(z) := \sum_{j=0}^{n-1} a_j z^j$, тогда его значения в корнях из единицы даются ДПФ: $F_n[a]_k = P_a(\omega_n^k)$.

Отметим, что преобразование Фурье можно определить в произвольном гильбертовом пространстве как переход к координатам в некотором ортогональном базисе (сравните со свойствами 1) и 2).

Из пункта 3) можно получить еще важное свойство. Рассмотрим два многочлена P_a и P_b степени $n_a - 1$ и $n_b - 1$, соответственно. Их произведение имеет вид

$$(P_a P_b)(z) = \sum_{j=0}^{\deg P_a + \deg P_b} \sum_{\ell=0}^{j} a_\ell b_{j-\ell} z^j =: \sum_j c_j z^j = P_c(z).$$

Вектор $c \in \mathbb{C}^{n_a+n_b-1}$ называется *линейной сверткой* векторов a и b: $c = a \otimes b$. Свертка возникает в анализе сигналов и изображений при использовании *линейных фильтров*: если u — входной дискретизованный сигнал, f — вектор, описывающий импульсную характеристику фильтра (это может быть сглаживающий фильтр, фильтр низких или высоких частот, дискретная производная и т.д.), то $u \otimes f$ — результат применения фильтра f к сигналу u. Так как ДПФ от вектора есть значения соответствующего полинома в корнях из единицы, то свертке векторов должно соответствовать произведение их ДПФ. Например,

пусть $a, b \in \mathbb{C}^n$. Тогда

$$P_{a\otimes b}(\omega_n^k) = P_a(\omega_n^k)P_b(\omega_n^k) = F_n[a]_k F_n[b]_k.$$

Казалось бы, тогда обратное преобразование от поэлементного произведения $F_n[a]F_n[b]$ должно дать свертку $a\otimes b$. Однако надо учитывать, что степень произведения полиномов возрастает, а $\omega_n^n = 1$, поэтому

$$P_{a\otimes b}(\omega_n^k) = \sum_{j=0}^{2n-2} \omega_n^{kj} \sum_{\ell=0}^{j} a_\ell b_{j-\ell} = \sum_{j=0}^{2n-2} \omega_n^{kj \bmod n} \sum_{\ell=0}^{j} a_\ell b_{j-\ell} = \sum_{j=0}^{n-1}\sum_{\ell=0}^{n-1} a_\ell b_{j-\ell \bmod n} \omega_n^{kj} = P_d(\omega_n^k),$$

где $d \in \mathbb{C}^n$ называется *циклической сверткой* a и b. Таким образом, *ДПФ от циклической свертки двух векторов равняется поэлементному произведению их ДПФ*. В частности, чтобы можно было восстановить коэффициенты произведения полиномов $P(z)Q(z)$ с помощью ДПФ в $\mathbb{C}^N$, необходимо выбрать $N \geq \deg P + \deg Q + 1$.

Наконец, ДПФ можно определить не только над комплексными числами, но и над произвольным полем. Так, пусть $\mathcal{F}$ — некоторое поле, а $\xi \in \mathcal{F}$ имеет порядок n. Тогда прямое преобразование задается формулой

$$F_n[a]_k := \sum_{j=0}^{n-1} \xi^{kj} a_j,$$

а обратное

$$F_n^{-1}[a]_k := \frac{1}{n} \sum_{j=0}^{n-1} \xi^{-kj} a_j.$$

Можно, конечно, определить ДПФ и над кольцом, но тогда оно не обязательно будет обратимо.

Быстрое преобразование Фурье. Теперь построим алгоритм, осуществляющий дискретное преобразование Фурье за субквадратичное по n время. Будем считать, что входной вектор есть $(f_0, \dots f_{n-1})$, где n — это степень двойки. Идея быстрого преобразования Фурье (БПФ) заключается в следующем:

$$(14)\quad F_n[a]_k := \sum_{j=0}^{n-1} \omega_n^{jk} a_j = \sum_{j=0}^{n/2-1} \omega_n^{2jk} a_{2j} + \sum_{j=0}^{n/2-1} \omega_n^{(2j+1)k} a_{2j+1} =$$
$$= \sum_{j=0}^{n/2-1} \omega_{n/2}^{jk} a_j^0 + \omega_n^k \sum_{j=0}^{n/2-1} \omega_{n/2}^{jk} a_j^1 = F_{n/2}\left[a^0\right]_{k \bmod n/2} + \omega_n^k F_{n/2}\left[a^1\right]_{k \bmod n/2},$$

где вектор a^0 состоит из четных компонент a, а вектор a^1 — из нечетных. Отсюда видно, что можно рекурсивно вычислить $F_{n/2}\left[a^0\right]$ и $F_{n/2}\left[a^1\right]$, а затем найти $F_n[a]$ по формулам

$$F_n[a]_k = F_{n/2}\left[a^0\right]_k + \omega_n^k F_{n/2}\left[a^1\right]_k, \quad F_n[a]_{k+n/2} = F_{n/2}\left[a^0\right]_k - \omega_n^k F_{n/2}\left[a^1\right]_k, \quad 0 \leq k < \frac{n}{2}.$$

Таким образом мы получаем алгоритм, который будем называть *быстрое преобразование Фурье*:

```
1: procedure FFT(f_0, ... f_{n-1})
2:     if n = 1 then
3:         return f_0
4:     end if
5:     (x_0, ..., x_{n/2-1}) = FFT(f_0, f_2, ... f_{n-2})
6:     (y_0, ..., y_{n/2-1}) = FFT(f_1, f_3, ... f_{n-1})
7:     for i = 1; i ⩽ n; i+ = 1 do
8:         r_j = x_j + exp(2πj/n) y_j
9:         r_{j+n/2} = x_j − exp(2πj/n) y_j
10:    end for
11:    return (r_0, ... r_{n-1})
12: end procedure
```

Нетрудно индукцией по K показать, что $\mathrm{FFT}(f_0, \dots f_{2^K-1})$ вычисляет вектор $(r_0, \dots r_{2^k-1})$, где

$$r_j = \sum_{k=0}^{2^K-1} a_j \exp\left(\frac{2\pi i k}{2^K}\right)$$

База очевидно верна, так как $\mathrm{FFT}(f_0) = f_0$. Теперь проверим шаг индукции. Поскольку по предположению $x_j = \sum_{k=0}^{\frac{n}{2}-1} f_{2k}\omega_{n/2}^{kj}, y_j = \sum_{k=0}^{\frac{n}{2}-1} f_{2k+1}\omega_{n/2}^{kj}$, то

$$r_j = \sum_{k=0}^{\frac{n}{2}-1} f_{2k}\omega_{n/2}^{kj} + \omega_n^j \sum_{k=0}^{\frac{n}{2}-1} f_{2k+1}\omega_{n/2}^{kj} = \sum_{k=0}^{\frac{n}{2}-1} f_{2k}\omega_n^{2kj} + \omega_n^j \sum_{k=0}^{\frac{n}{2}-1} f_{2k+1}\omega_n^{2kj} = \sum_{k=0}^{n-1} f_k \omega_n^{kj}.$$

Итак, быстрое преобразование Фурье работает корректно. Работая на входном массиве длины n, FFT вызывает себя дважды и работает с массивами длины $\frac{n}{2}$, а затем амальгамирует результат за $O(n)$. Время работы такого алгоритма удовлетворяет рекуррентной оценке

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(Mn),$$

где M — оценка времени выполнения одной арифметической операции над комплексными числами (либо в поле $\mathcal{F}$ для более общего случая). По теореме Аккра-Баззи получаем, что

$$T(n) = \Theta(Mn \log n),$$

что гораздо лучше времени работы наивного алгоритма: $\Theta(Mn^2)$.

Пример 3.45. Рассмотрим многочлены $P(z) = 3z + 2$ и $Q(z) = z^2 + 1$. Требуется найти их произведение с помощью БПФ. Т.к. $\deg P + \deg Q = 3$, то можно выбрать $n = 4$. Тогда вектор коэффициентов P равен $a = (2, 3, 0, 0)$, и коэффициенты Q — $b = (1, 0, 1, 0)$. $\omega_4 = i$, $\omega_2 = -1$. Применим алгоритм БПФ:

$$F_2\left[a^0\right] = F_2[(2,0)] = (2+0, 2-0) = (2,0),\ F_2\left[a^1\right] = F_2[(3,0)] = (3,0), \tag{15}$$

$$F_4[a] = (2 + 3i^0, 0 + 0i, 2 - 3i^0, 0 - 0i) = (5, 0, -1, 0), \tag{16}$$

$$F_2\left[b^0\right] = F_2[(1,1)] = (1+1, 1-1) = (2,0),\ F_2\left[b^1\right] = F_2[(0,0)] = (0,0), \tag{17}$$

$$F_4[b] = (2, 0, 2, 0). \tag{18}$$

Осталось научиться переходить от значений многочлена в комплексных корнях из единицы к его коэффициентам, то есть посчитать обратное ДПФ. Обратное преобразование вычисляется так же, только ω_n нужно заменить на ω_n^{-1}, и разделить итоговый результат на n. Дискретное преобразование Фурье можно записать в матричном виде:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \dots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \dots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix},$$

то есть $y = V_n a$, где элемент матрицы V_n с индексами (k, j) равен ω_n^{kj} (при $j, k = 0, 1, \dots, n-1$). Обратная операция $a = DFT_n^{-1}(y)$ состоит в умножении матрицы V_n^{-1} (обратной к V_n) слева на y.

Теорема 3.46. *Элемент с индексами (j, k) матрицы V_n^{-1} равен $\frac{\omega_n^{-kj}}{n}$.*

Доказательство. Покажем, что составленная из таких элементов матрица V_n^{-1} удовлетворяет требованию $V_n^{-1}V_n = I_n$, где I_n — единичная матрица размера $n \times n$. По определению, (j, j')-й элемент произведения $V_n^{-1}V_n$ есть

$$\left[V_n^{-1}V_n\right]_{jj'} = \sum_{k=0}^{n-1} \left(\frac{\omega_n^{-kj}}{n}\right)(\omega_n^{kj'}) = \frac{1}{n}\sum_{k=0}^{n-1} \omega_n^{k(j'-j)}.$$

По лемме о сложении последняя сумма равна 1, если $j = j'$, и равна 0 при $j \neq j'$. Действительно, $-(n-1) \leqslant j' - j \leqslant n - 1$, так что $j - j'$ не делится на n при $j \neq j'$. $\square$

Зная обратную матрицу V_n^{-1}, мы можем найти $a = DFT_n^{-1}(y)$ по формуле

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj}$$

для $j = 0, 1, \ldots, n-1$. Мы видим, что для вычисления обратного преобразования Фурье можно применить тот же алгоритм БПФ, поменяв в нём a и y местами, заменив ω_n на ω_n^{-1} и разделив каждый элемент результата на n. Отсюда следует, что DFT_n^{-1} также можно вычислить за $O(n \log n)$.

Быстрое перемножение многочленов

Теперь мы можем описать последовательность действий для перемножения многочленов, заданных своими коэффициентами. Для этого мы сначала перейдём к значениям в комплексных корнях из единицы за время $\Theta(n \log n)$; затем перемножим значения, получив значения многочлена-произведения в комплексных корнях из единицы за время $\Theta(n)$; после этого применим к полученным значениям обратное преобразование Фурье за время $\Theta(n \log n)$. Общее время работы составит $\Theta(n \log n)$. Чтобы найти свертку двух векторов $a, b \in \mathbb{C}^n$, нужно дополнить их 0 до $a', b' \in \mathbb{C}^{2^s}$, где $s = \lceil \log n \rceil + 1$, так что $2n \le 2^s < 4n$, и вычислить $F_{2^s}^{-1}\left[F_{2^s}[a']F_{2^s}[b']\right]$.

Пример 3.47. Рассмотрим многочлены $P(z) = 3z + 2$ и $Q(z) = z^2 + 1$. Требуется найти их произведение с помощью БПФ. Т.к. $\deg P + \deg Q = 3$, то можно выбрать $n = 4$. Тогда вектор коэффициентов P равен $a = (2, 3, 0, 0)$, и коэффициенты $Q - b = (1, 0, 1, 0)$. $\omega_4 = i$, $\omega_2 = -1$. Применим алгоритм БПФ:

$$F_2\left[a^0\right] = F_2[(2,0)] = (2+0, 2-0) = (2,0),\ F_2\left[a^1\right] = F_2[(3,0)] = (3,0), \tag{19}$$

$$F_4[a] = (2 + 3i^0, 0 + 0i, 2 - 3i^0, 0 - 0i) = (5, 0, -1, 0), \tag{20}$$

$$F_2\left[b^0\right] = F_2[(1,1)] = (1+1, 1-1) = (2,0),\ F_2\left[b^1\right] = F_2[(0,0)] = (0,0), \tag{21}$$

$$F_4[b] = (2, 0, 2, 0). \tag{22}$$

Теперь найдем обратное преобразование от $F_4[a]F_4[b] = (10, 0, -2, 0)$:

$$2F_2^{-1}[(10,-2)] = (8,12),\ 2F_2^{-1}[(0,0)] = (0,0), \tag{23}$$

$$4F_4^{-1}[(10,0,-2,0)] = (8 + 0i^0, 12 + 0i^{-1}, 8 - 0i^0, 12 - 0i^{-1}) = (8,12,8,12). \tag{24}$$

Таким образом, $F_4^{-1}\left[F_4[a]F_4[b]\right] = (2,3,2,3)$, и $P(z)Q(z) = 2 + 3z + 2z^2 + 3z^3$.

Основанный на БПФ алгоритм быстрого перемножения многочленов имеет забавное применение в задаче поиске подстроки с джокером (символом «?», на месте которого может быть любая буква). Если и строка, и подстрока заранее известны, то существует линейный по сумме длин строк алгоритм Кнута-Морриса-Пратта. Его же, однако, не

получается столь же тривиально обобщить на случай с джокерами, вопрос существования линейного алгоритма в этом случае до сих пор открыт.

Идея БПФ-алгоритма поиска подстрок следующая. Нужно найти подстроку (образец) $p_0, \ldots, p_{m-1}$ в строке (тексте) $t_0, \ldots, t_{n-1}$, здесь p_i, t_j — это символы некоторого алфавита. Говорят, что подстрока входит с i-й позиции, если $p_j = t_{i+j}$, $j = 0, \ldots, m-1$. Если считать буквы алфавита различными целыми числами, то вхождение подстроки с i-й позиции эквивалентно обнулению суммы квадратов: $B_i = \sum_{j=0}^{m-1}(p_j - t_{i+j})^2 = \sum_{j=0}^{m-1}(p_j^2 - 2p_j t_{i+j} + t_{i+j}^2) = 0$, а вычисление массива чисел $\{B_i,\, i = 0, \ldots, n-m\}$ позволяет определить все места вхождения подстроки в текст.

С помощью БПФ можно построить $O(n \log m)$-процедуру. Сумма $S = \sum\limits_{j=0}^{m-1} p_j^2$ присутствует как слагаемое в каждом B_i и вычисляется за $O(n)$ шагов.

Рассмотрим два полинома степени не выше n: $T(x) = t_{n-1}x^{n-1} + \cdots + t_1 x + t_0$, $P(x) = p_0 x^{n-1} + \cdots + p_{m-1}$ Их произведение можно вычислить с помощью БПФ за $O(n \log n)$. Посмотрим на коэффициент их произведения $C(x)$ при x^{m-1+i} $(0 \le i \le n-m)$:

$$c_{m-1+i} = p_0 t_i + p_1 t_{i+1} + p_2 t_{i+2} + \ldots + p_{m-2} t_{m-2+i} + p_{m-1} t_{m-1+i} = \sum_{j=0}^{m-1} p_j t_{j+i}$$

Как видим, это одно из слагаемых для B_i. Таким образом, алгоритм для подсчета всех B_i таков. Сначала вычисляем за $O(n \log n)$ шагов коэффициенты произведения $C(x)$ указаных многочленов. Далее за максимум $O(n)$ шагов вычисляем сумму квадратов S и за $O(n)$ шагов считаем сумму первых m квадратов t_i, т. е. $H = \sum\limits_{j=0}^{m-1} t_j^2$. Далее вычисляем $B_0 = S - 2 \cdot c_{m-1} + H$ и $B_1 = S - 2 \cdot c_m + \sum\limits_{j=0}^{m-1} t_{j+1}^2 = B_1 + 2 \cdot c_{m-1} - 2 \cdot c_m - t_1^2 + t_m^2$.

Для этого нам потребуется $O(1)$ операций. Аналогично получим $B_i = B_{i-1} + 2(c_{m-2+i} - c_{m-1+i}) - t_{i-1}^2 + t_{m-1+i}^2$, т. е. для получения каждого следующего члена требуется $O(1)$ операций. Таким образом, для вычисления всех членов потребуется еще $O(n)$ операций. Таким образом, весь алгоритм асимптотически требует $O(n \log n)$ операций.

Ту же идею для проверки вхождения подстроки можно использовать, если разрешается использовать символ джокера, «?». Тогда в тех же обозначениях нужно вычислить массив $\{A_i,\, i = 0, \ldots, n-m\}$, где $A_i = \sum\limits_{j=0}^{m-1} p_j t_{i+j}(t_{i+j} - p_j)^2 = \sum\limits_{j=0}^{m-1} (p_j^3 t_{i+j} - 2p_j^2 t_{i+j}^2 + p_j t_{i+j}^3)$. При этом символу «?» соответствует число 0, а все остальные символы кодируются ненулевыми числами. Тогда нетрудно заметить, что обнуляться полученная сумма будет тогда и только тогда, когда подстрока входит в слово с данной позиции. Вычисление отдельных

частей суммы можно произвести при помощи БПФ, если правильно подобрать многочлены (по аналогии с предыдущим случаем, то есть так, чтобы коэффициенты произведения полученных многочленов выражались через отдельные части суммы A_i).

Обобщения и некоторые применения БПФ. Теперь покажем, как получить БПФ для произвольного n.

$$F_n[a]_k = \sum_{j=0}^{n-1} \omega_n^{kj} a_j = \sum_{j=0}^{n-1} \omega_n^{k^2/2} \omega_n^{-(k-j)^2/2} \omega_n^{j^2/2} a_j = \omega_n^{k^2/2} \sum_{j=0}^{n-1} \omega_n^{-(k-j)^2/2} \left(\omega_n^{j^2/2} a_j\right),$$

поэтому для вычисления $F_n[a]$ достаточно найти свертку векторов $\left[\omega_n^{-j^2/2}\right]_{j=0}^{n-1}$ и $\left[\omega_n^{j^2/2} a_j\right]_{j=0}^{n-1}$, что можно сделать за $O(Mn \log n)$.

Помимо одномерных сверток, бывает необходимо вычислять также и многомерные — например, для применения фильтров к изображениям. Это, опять же, можно делать с помощью ДПФ, которое в многомерном случае определяется следующим образом: пусть $a \in \mathbb{C}^{n_1 \times n_2 \times \cdots \times n_d}$, тогда

$$F[a]_{k_1,k_2,\ldots,k_d} := \sum_{j_1=0}^{n_1-1} \omega_{n_1}^{j_1 k_1} \cdots \sum_{j_d=0}^{n_d-1} \omega_{n_d}^{j_d k_d} a_{j_1,\ldots,j_d}.$$

Как видно, ДПФ в данном случае получается как последовательное применение одномерного преобразования по всем измерениям (порядок не важен). Но тогда, используя одномерное БПФ, его можно вычислить за

$$\sum_{s=1}^{d} \frac{n}{n_s} O(n_s \log n_s) = \sum_{s=1}^{d} O(n \log n_s) = O\left(n \sum_s \log n_s\right) = O(n \log n)$$

арифметических операций, где $n := \prod_{s=1}^{d} n_s$ — размер входа.

БПФ можно также рассмотреть и над другим кольцом R, которое может не содержать корня из единицы некоторой степени вида 2^k. Тогда покажем, что за время $O(n \log n \log\log n)$ можно в кольце $R[x]/(x^n+1)$ перемножить многочлены $f(x) = \sum_{j=0}^{n} f_j x^j$ и $g(x) = \sum_{j=0}^{n} g_j x^j$ степени не более n. Снова положим $n = 2^k$, введем также $m = 2^{\lfloor \frac{n}{2} \rfloor}$ и

$$\widetilde{f}(x,y) = \sum_{i=0}^{n/m} \sum_{j=0}^{m-1} f_{mi+j} x^j y^i, \quad \widetilde{g}(x,y) = \sum_{i=0}^{n/m} \sum_{j=0}^{m-1} g_{mi+j} x^j y^i,$$

тогда $f(x) = \widetilde{f}(x, x^m)$, $g(x) = \widetilde{g}(x, x^m)$. Если $\widetilde{f}\widetilde{g} = \widetilde{h} + \widetilde{q}(y^{n/m} + 1)$, то

$$fg = \widetilde{h}(x, x^m) + \widetilde{q}(x, x^m)(x^n + 1) = \widetilde{h}(x, x^m) \bmod x^n + 1$$

$\widetilde{h}$ удобно вычислять как $\widetilde{h} \bmod (R[x]/(x^{2m}+1))[y]$: поскольку степень $\widetilde{h}$ как многочлена по x меньше $2m$, то $\widetilde{h} \bmod (R[x]/(x^{2m}+1))[y]$ будет по факту тем же многочленом,

но в кольце $R[x]/(x^{2m}+1)$ есть нужный нам корень из единицы, поэтому мы можем применить БПФ: поскольку $R[x]/(x^{2m}+1$ содержит примитивный корень ν степени $2n/m$, то $\nu^{n/m}=-1$ и

$$(\widetilde{f} \mod y^{n/m}-1)(\widetilde{g} \mod y^{n/m}-1)=\widetilde{h} \mod y^{n/m}-1,$$

и свертку этих многочленов можно вычислить обычным БПФ, но для умножения коэффициентов $\widetilde{f}$ и $\widetilde{g}$ (элементов $R[x]/(x^{2m}+1)$) рекурсивно используется сам алгоритм.

Итак, мы получили следующую процедуру перемножения многочленов $f(x)$ и $g(x)$ степени не более $n=2^k$.

procedure MODULAR-FFT-CONVOLUTION($f(x), g(x)$)
  if $k \leqslant 2$ **then**
    return $f(x)g(x) \bmod x^n+1$
  end if
  $m=2^{\lfloor \frac{k}{2} \rfloor}$
  $\widetilde{f}(x,y)=\sum_{i=0}^{n/m}\sum_{j=0}^{m-1} f_{mi+j}x^j y^i$, $\widetilde{g}(x,y)=\sum_{i=0}^{n/m}\sum_{j=0}^{m-1} g_{mi+j}x^j y^i$
  if $n/m=2m$ **then**
    $\eta \leftarrow x \bmod x^{2m}+1$
  else
    $\eta \leftarrow x^2 \bmod x^{2m}+1$
  end if
  $f^*=\widetilde{f} \mod y^{n/m}-1$, $g^*=\widetilde{g} \mod y^{n/m}-1$
  $h^*=$ FFT-CONVOLUTION-MOD(f^*, g^*)
  взять $\widetilde{h} \in R[x,y]$ такой, что $h^*=\widetilde{h} \bmod x^{2m}+1$
  return $h(x, x^m) \bmod x^n+1$
end procedure

Здесь FFT-CONVOLUTION-MOD — модифицированное умножение многочленов, где для умножения коэффициентов используется рекурсивно MODULAR-FFT. Из рассуждений выше следует корректность MODULAR-FFT-CONVOLUTION. Теперь оценим сложность этого алгоритма. Пусть $T(k)$ — сложность умножения многочленов степени 2^k. При вычислении h^* процедура MODULAR-FFT-CONVOLUTION вычисляется на многочленах степени не более $2^{\lfloor \frac{k}{2} \rfloor+1}$, степень h^* как многочлена от y не более $2^{\lceil \frac{k}{2} \rceil}$. Таким образом, получим реккуренту

$$T(k) \leqslant 2^{\lceil \frac{k}{2} \rceil} T\left(\lfloor \frac{k}{2} \rfloor+1\right)+O\left(2^k\left(\lceil \frac{k}{2} \rceil+1\right)\right),$$

Отсюда можно вывести, что

$$T(k) \leqslant \mathrm{const} \cdot 2^k k \log k + O(2^k k),$$

то есть сложность FFT-CONVOLUTION-MOD есть $O(n \log n \log\log n)$.

Используя конструкцию выше, БПФ можно применить для умножения больших чисел. Рассмотрим два числа в m-арной записи: $x = \sum_{j=0}^{n} a_j m^j = P_a(m)$, $y = P_b(m)$. Тогда, чтобы вычислить произведение xy, достаточно найти коэффициенты произведения полиномов P_a и P_b. Осуществляя БПФ в подходящем кольце $\mathbb{Z}/m\mathbb{Z}$, где m имеет вид $m = 2^s + 1$, можно найти произведение xy за время $O(n \log n \log\log n)$. Этот алгоритм называется *алгоритмом Шенхаге-Штрассена*.

Преобразование Фурье можно также использовать для анализа или решения систем линейных уравнений. Дело в том, что некоторые операторы со сложной структурой имеют относительно простой вид в пространстве преобразования Фурье. Например, рассмотрим СЛАУ $Cx = b$ с *циркулянтной матрицей* C:

$$C = \begin{bmatrix} c_0 & c_{n-1} & c_1 \\ c_1 & c_0 & c_2 \\ \dots & \dots & \dots \\ c_{n-1} & c_{n-2} & c_0 \end{bmatrix},$$

т.е. $C_{i,j} = c_{i-j \bmod n}$. Несложно видеть, что $Cx = c \otimes x$ (циклическая свертка), т.е. циркулянты — это матричное определение циклической свертки. Отсюда получаем, что

$$F_n[Cx] = \Xi C x = F_n[c] F_n[x] = \operatorname{diag}(F_n[c]) \Xi x.$$

Таким образом, в пространстве преобразования Фурье оператор имеет диагональную матрицу:

$$\Xi C \Xi^{-1} = \operatorname{diag}(F_n[c]),$$

и уравнение переписывается как

$$\operatorname{diag}(F_n[c]) F_n[x] = F_n[b].$$

Это позволяет решать СЛАУ с циркулянтными матрицами за $O(n \log n)$, так как

$$x = F_n^{-1} \left[\operatorname{diag}(F_n[c])^{-1} F_n[b] \right].$$

Еще один класс матриц, работу с которыми может упростить преобразование Фурье — это тёплицевы матрицы. Матрица A является *матрицей Тёплица*, если $A_{i+1,j+1} = A_{i,j}$. Покажем, как можно умножать вектор на тёплицеву матрицу за время $O(n \log n)$ вместо $O(n^2)$, которое требуется при обычном умножении. Заметим, что матрица

$$A := \begin{bmatrix} A & A^T \\ A^T & A \end{bmatrix}$$

циркулянтная, поэтому

$$A \begin{bmatrix} 0^n \\ x \end{bmatrix} = \begin{bmatrix} A^T x \\ Ax \end{bmatrix}.$$

А умножение на циркулянтную матрицу дает свертку, которую можно вычислить с помощью БПФ.

Решето Эратосфена

По аналогии с асимптотическим законом распределения простых чисел можно получить, что если p_r — r-ое простое число, то $p_r < 2r \log r$ для всех $r > 2$. Поэтому для нахождения первых k простых чисел обычно используется *решето Эратосфена*: мы записываем список всех целых чисел от 1 $x = 2r \log r$, затем вычеркиваем все четные числа, все кратные 3, все кратные 5, и так для каждого простого меньше, чем $\sqrt{x}$. Оставшиеся целые числа не делятся на простые числа, не превышающие $\sqrt{x}$ и, следовательно, являются простыми.

```
1: procedure Eratosthene(r)
2:     x = ⌈2r log r⌉
3:     p ∈ {0,1}^x = (p_1, p_2, ... p_x) = (0, 1 ... 1)
4:     for i ∈ [1; x] do
5:         if p_i = 1 then
6:             for j ∈ [2; x/p_i] do
7:                 p_j = 0
8:             end for
9:         end if
10:    end for
11: end procedure
```

Для каждого простого $p < \sqrt{x}$ совершается $\lfloor \frac{x}{p} \rfloor$ шагов, суммарное число шагов оценивается сверху (см. уравнение (3.20) в статье [19])

$$x \sum_{p<\sqrt{x},\ p \text{ простое}} \frac{1}{p} < x \left(\log\log\sqrt{x} + B + \frac{1}{\log^2 \sqrt{x}} \right) = x \left(\log \frac{\log x}{2} + B + \frac{4}{\log^2 x} \right),$$

Поскольку каждый шаг может быть проделан за $O(\log x)$ элементарных операций [сложение индексов во время исполнения внутреннего цикла **for**], сложность решета Эратосфена есть $O(x \log x \log\log x)$, а поскольку $\log x = \log 2r + \log\log r = O(\log r)$, то сложность решета Эратосфена можно оценить как $O(r \log^2 r \log\log r)$.

Теперь рассмотрим алгоритмы, проверяющие простоту данного на вход натурального числа n. Без ограничения общности можно считать, что n нечетно.

Вероятностные тесты на простоту

Рассмотрим следующий *тест Ферма*:


```
1: procedure Fermat(N)
2:     a ← random ∈ [2; N − 1]
3:     if (a, N) > 1 then
4:         return composite
5:     else
```

```
        if a^{N-1} = 1 mod N then
            return prime
        else
            return composite
        end if
    end if
end procedure
```

Тест Ферма — это вероятностный алгоритм типа Монте-Карло, то есть работающий, пока не получит противоречие, либо не превысит допустимое число итераций, который на каждой итерации генерирует случайное целое число a из отрезка $[1, N-1]$ (каждый из вычетов может появиться с одной и той же вероятностью). Нетрудно видеть, что если N простое, то тест Ферма обязательно выведет prime. Теперь надо оценить вероятность того, что при составном N тест Ферма даст неверный ответ, то есть prime.

Утверждение 3.48. *Пусть НОД$(a, N) = 1$ и $a^{N-1} \neq 1 \bmod N$. Тогда по крайней мере для половины чисел из промежутка $1 \leqslant b < N$ выполнено $b^{N-1} \neq 1 \bmod N$.*

Доказательство. Условие нужно понимать так, что n — не число Кармайкла, и существует такое число a, что НОД$(a, N) = 1$ и $a^{N-1} \neq 1 \bmod N$. Тогда для $x_1, \ldots x_k$ таких, что $x_i^{N-1} = 1 \bmod N$ имеем

$$(ax_i)^{N-1} = a^{N-1}x_i^{N-1} = a^{N-1} \neq 1 \bmod N,$$

таким образом, хотя бы половина искомых b существует. □

Из утверждения выше следует, что если число n — нечётное составное, то тест Ферма выдаст prime с вероятностью не более $\frac{1}{2}$. Иными словами, на таком входе n не более чем за t итераций тест Ферма определит, что это число составное, с вероятностью большей $1 - \frac{1}{2^t}$, что является очень близким к единице числом даже для $t = 10$ (вероятность получается примерно $99,9\%$).

Утверждение 3.49. *Тест Ферма может быть реализован за полиномиальное по входу число операций.*

Доказательство. В тесте Ферма случайное число находится один раз, покажем, что остальные манипуляции могут быть реализованы за полиномиальное время. Действительно, алгоритм Евклида полиномиален, а вычисление $a^{N-1} \bmod N$ можем реализовать следующим образом:

- возводить в степень по-индийски, то есть вычислять $x, x^2, x^4, \ldots$;
- после каждого возведения в квадрат брать остаток по модулю N;

Отсюда получаем, что тест Ферма может быть реализован за полиномиальное по входу число операций. □

Однако не всё так хорошо для данной вероятностной процедуры.

Определение 3.50. Составное натуральное число n называется *числом Кармайкла*, если для всех натуральных a таких, что $(a, n) = 1$, верно $a^{n-1} = 1 \bmod n$.

Существует альтернативное определение чисел Кармайкла.

Теорема 3.51 (Korselt)**.** *Число n является числом Кармайкла тогда и только тогда, когда оно нечетно, свободно от квадратов и $p - 1 | n - 1$ для любого простого делителя $p | n$.*

Доказательство. Нетрудно убедиться в достаточности условий теоремы: если $n = p_1 \dots p_k$ нечетно и $p_i - 1 | n - 1$, то для любого $a \in \mathbb{Z}/n\mathbb{Z}$ существует такой p_i (простой делитель n), что $(p_i, a) = 1$ — в противном случае все простые делители n делят a, следовательно, $n | a$. Имеем в таком случае

$$a^{\phi(n)} = a^{(p_1-1)\dots(p_i-1)\dots(p_k-1)} = 1^{\frac{\phi(n)}{p_i-1}} = 1 \bmod n$$

.

Теперь проерим необходимость этих условий. Действительно, если кармайклово n четное, то $(-1)^n = -1 \bmod n$, что неверно. Пусть $n = p_1^{k_1} \dots p_m^{k_m}$, и $k_i \geqslant 2$ для некоторого i. Тогда рассмотрим a — порождающий $(\mathbb{Z}/p_i^{k_i}\mathbb{Z})^\times \subset (\mathbb{Z}/n\mathbb{Z})^\times$. Его порядок равен $|(\mathbb{Z}/p_i^{k_i}\mathbb{Z})^\times| = \phi(p_i^{k_i}) = p_i^{k_i-1}(p_i - 1)$, следовательно, $p_i^{k_i-1}(p_i - 1) | n - 1$. Поскольку p_i не делит $p_i - 1$ и $n - 1$, то имеем противоречие. Более того, отсюда видно, что $p_i - 1 | n - 1$. Все эти рассуждения верны для любого p_i, тем самым достаточность установлена. □

Используя критерий Корсельта, можно найти первые несколько чисел Кармайкла.

$$\begin{aligned}
561 &= 3 \cdot 11 \cdot 17 && (2 \mid 560; \quad 10 \mid 560; \quad 16 \mid 560)\\
1105 &= 5 \cdot 13 \cdot 17 && (4 \mid 1104; \quad 12 \mid 1104; \quad 16 \mid 1104)\\
1729 &= 7 \cdot 13 \cdot 19 && (6 \mid 1728; \quad 12 \mid 1728; \quad 18 \mid 1728)\\
2465 &= 5 \cdot 17 \cdot 29 && (4 \mid 2464; \quad 16 \mid 2464; \quad 28 \mid 2464)\\
2821 &= 7 \cdot 13 \cdot 31 && (6 \mid 2820; \quad 12 \mid 2820; \quad 30 \mid 2820)\\
6601 &= 7 \cdot 23 \cdot 41 && (6 \mid 6600; \quad 22 \mid 6600; \quad 40 \mid 6600)\\
8911 &= 7 \cdot 19 \cdot 67 && (6 \mid 8910; \quad 18 \mid 8910; \quad 66 \mid 8910)
\end{aligned}$$

Более того, впоследствии было установлено, что чисел Кармайкла бесконечно много (хотя их и не так много), так что есть интерес в более сильном тесте.

Рассмотрим другую вероятностную процедуру под названием *тест Соловея-Штрассена*

Прежде напомним несколько определений.

Определение 3.52. Целое число a называется *квадратичным вычетом* по модулю m, если разрешимо сравнение $x^2 \equiv a \pmod m$. В противном случае число a называется *квадратичным невычетом.*

Определение 3.53. Пусть p — нечётное простое число и a — целое число. Определим *символ Лежандра* следующим образом:

$$\left(\frac{a}{p}\right) = \begin{cases} 1, & a \text{ — квадратичный вычет по модулю } p \\ -1, & a \text{ — квадратичный невычет по модулю } p \\ 0, & p|a \end{cases}$$

Утверждение 3.54. $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \mod p$

На первый взгляд кажется, что символ Лежандра оптимально вычислять при помощи быстрого возведения в степень по модулю. Такая процедура потребует $O(\log^3 p)$ битовых операций. Однако если ввести в рассмотрение более общую конструкцию, называемую *символом Якоби*, то символ Лежандра можно будет вычислять за $O(\log^2 n)$.

Определение 3.55. Пусть P — нечётное число, большее единицы и $P = p_1 p_2 \cdot p_k$ — его разложение на простые множители (среди $p_1, \ldots, p_k$ могут быть и равные). Тогда для произвольного целого числа a *символ Якоби* определяется равенством

$$\left(\frac{a}{P}\right) = \left(\frac{a}{p_1}\right)\left(\frac{a}{p_2}\right) \cdot \ldots \cdot \left(\frac{a}{p_k}\right),$$

где $\left(\frac{a}{p_i}\right)$ — символы Лежандра. Кроме того, по определению будем считать, что $\left(\frac{a}{1}\right) = 1$ для всех a.

Для любых натуральных p и q меют место следующие утверждения:

(1) *Мультипликативность.* $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right) \cdot \left(\frac{b}{p}\right)$.

(2) *Периодичность.* Если $a \equiv b \pmod p$, то $\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$.

(3) $\left(\frac{1}{p}\right) = 1$.

(4) $\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}}$.

(5) $\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}}$.

Эти утверждения верны и для символа Лежандра, и для символа Якоби.

Теорема 3.56 (Квадратичный закон взаимности)**.** *Для любых нечетных взаимнопростых P и Q верно* $\left(\frac{P}{Q}\right)\left(\frac{Q}{P}\right) = (-1)^{\frac{P-1}{2} \cdot \frac{Q-1}{2}}$.

Пример 3.57. Из квадратичного закона взаимности:

$$\left(\frac{219}{383}\right) = -\left(\frac{383}{219}\right) = -\left(\frac{164}{219}\right) = -\underbrace{\left(\frac{4}{219}\right)}_{1}\left(\frac{41}{219}\right) =$$

$$= -\left(\frac{219}{41}\right) = -\left(\frac{14}{41}\right) - \left(\frac{2}{41}\right)\left(\frac{7}{41}\right) = -\left(\frac{7}{41}\right) = -\left(\frac{41}{7}\right) = -\left(\frac{-1}{7}\right) = 1.$$

Оказывается формулу Эйлера можно превратить в критерий простоты числа, если перейти от символа Лежандра к символу Якоби.

Теорема 3.58. *Нечётное число $n > 1$ является простым тогда и только тогда, когда для всякого $a \in \{1, \ldots, n-1\}$ выполняется* $\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \bmod n$.

Именно это сравнение и проверяет тест Соловея-Штрассена.

```
1: procedure SOLOVAY-STRASSEN(N)
2:     a ← random ∈ [2; N − 1]
3:     if (a, N) > 1 then
4:         return composite
5:     else
6:         if (a/N) = a^((N−1)/2) mod N then
7:             return prime
8:         else
9:             return composite
10:        end if
11:    end if
12: end procedure
```

Теорема 3.59 (Соловей-Штрассен, 1977). *Пусть n — нечётное и составное, тогда*

(1) существует число $a \in \{1, 2, \ldots, n-1\}$, такое что $\gcd an = 1$ *и* $a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod n$;

(2) выполнено неравенство

$$\frac{\left|\left\{a \in \{1, 2, \ldots, n-1\} \mid \gcd an > 1 \text{ или } \left(\gcd an = 1 \text{ и } a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod n\right)\right\}\right|}{n-1} > \frac{1}{2}.$$

Отметим, что тест Соловея-Штрассена на любом нечётном составном числе за не более чем t шагов даст ответ «нет» с вероятностью большей, чем $1 - \frac{1}{2^t}$. При этом исключений в стиле чисел Кармайкла, как было для теста Ферма, для теста Соловея-Штрассена уже нет. Проверка сравнения Эйлера требует $O(\log^3 n)$ операций, как и проверка сравнения Ферма.

Следующий алгоритм будет основываться тоже на некотором утверждении, которое вытекает из сравнения Ферма и ещё некоторых групповых свойств.

Теорема 3.60. *Пусть $n > 2$ — нечётное число и пусть $n = 2^l k$, где $l \geqslant 1$ и k — нечётное число. Тогда число n является простым тогда и только тогда, когда для любого $a \in \{1, 2, \ldots, n-1\}$ выполнено одно из условий:*

(1) $a^k \equiv 1 \pmod{n}$;

(2) существует $0 \leqslant i \leqslant l-1$, такое что $a^{2^i k} \equiv -1 \pmod{n}$.

Теперь рассмотрим вероятностный *тест Миллера-Рабина*. В описании алгоритма $n - 1 = 2^l k$, где $l \geqslant 1$ и k — нечётное число.

```
1: procedure MILLER-RABIN(N)
2:     a ← random ∈ [2; N − 1]
3:     if (a, N) > 1 then
4:         return composite
5:     else
6:         if a^{N−1} ≠ 1 mod N then
7:             return composite
8:         else
9:             if a^{k−1} = 1 mod N then
10:                return prime
11:            else
12:                r ← min{x ∈ [1; l] | a^{2^x k} = 1 mod N}
13:                if a^{2^{r−1} k} ≠ −1 mod N then
14:                    return composite
15:                else
16:                    return prime
17:                end if
18:            end if
19:        end if
20:    end if
21: end procedure
```

Нетрудно видеть, что эта реализация теста Рабина-Миллера полиномиальна. Случайное число находится один раз, алгоритм Евклида полиномиален, а вычисление $a^{N-1} \bmod N$ и $a^{k-1} \bmod N$ можем реализовать следующим образом:

- возводить в степень по-индийски, то есть вычислять $x, x^2, x^4, \ldots$;
- после каждого возведения в квадрат брать остаток по модулю N;

Поиск числа r потребует перебора не более $l \leqslant \log_2 \frac{n-1}{k} \leqslant \log_2 n$ вычислений $a^{2^x k} \bmod N$. Итого несложно видеть, что процедура Miller-Rabin работает за $O(\log^3 n)$. Теперь оценим вероятность ошибки.

Определение 3.61. Пусть $n > 2$ — нечётное число и $a \in \{1, 2, \ldots, n-1\}$. Тогда число a называется *лжесвидетелем Миллера-Рабина*, если последовательность $(a^k, a^{2k}, a^{4k}, \ldots, a^{2^{l-1}k})$ по модулю n равна либо последовательности, у которой на первом месте стоит единица, либо последовательности, у которой на каком-то месте стоит (-1), и *свидетелем Миллера-Рабина* в противном случае.

Теорема 3.62. *Пусть $N > 2$ — нечётное составное число. Если N составное, то вероятность нахождения лжесвидетеля не более $\frac{1}{4}$.*

Доказательство. Пусть $N - 1 = t \cdot 2^h$. Рассмотрим следующие случаи.

- Если N простое, то $(a, N) = 1$ и $a^{N-1} = 1 \bmod N$, а поскольку r — минимальное натуральное число такое, что $a^{2^r k} = 1$, то $(a^{2^{r-1}k})^2 = 1$ и $a^{2^{r-1}k} = -1$. Следовательно, на N тест Рабина-Миллера выдает prime всегда.
- Если $N = p^e$ для некоторого $e > 1$ и простого нечетного p, то множество лжесвидетелей очевидно содержится во множестве $\{a \in [2; N-1] \mid a^{N-1} = 1 \mod N\}$, которое согласно теореме Эйлера имеет мощность $(\phi(N), N-1)$. Отсюда

$$\mathbb{P}(p^e - \text{лжесвидетель}) \leqslant \frac{(\phi(N), N-1)}{N-1} = \frac{(p^{e-1}(p-1), p^e - 1)}{p^e - 1} = \frac{p-1}{p^e - 1} = \frac{1}{1 + \ldots + p^{e-1}} \leqslant$$
$$\leqslant \frac{1}{1 + p^{e-1}} \leqslant \frac{1}{4}.$$

- Пусть $N = p_1^{e_1} \ldots p_m^{e_m}$ для $r \geqslant 2$. Тогда $\phi(p_i^{e_i}) = t_i 2^{l_i}$. Введем $g = \min\{l, l_1, \ldots l_m\}$ и покажем, что для любого лжесвидетеля a имеет место $a^{t2^g} = 1 \bmod N$. По определению l имеем $a^{t2^l} = 1 \bmod N$, так что будем считать $g < l$, то есть $g = l_i$ для некоторого i. Пусть $a^{t2^g} \neq 1 \bmod N$, а $j \in [g+1, l]$ — минимальное, для которого $a^{t2^j} = 1 \bmod N$. Тогда $a^{t2^{j-1}} = -1 \bmod N$. Воспользовавшись КТО, найдем такое a_i, что $a = a_i \bmod p_i^{e_i}$, тогда $a_i^{t2^{j-1}} = -1 \bmod p_i^{e_i}$. Следовательно, порядок a_i^t в группе $(\mathbb{Z}/p_i^{e_i}\mathbb{Z})^\times$ равен 2^j, то есть больше 2^{h_i}, что противоречит тому, что в $(\mathbb{Z}/p_i^{e_i}\mathbb{Z})^\times$ верно $a_i^{t2^{h_i}} = 1$.

  Теперь пусть $\rho_j : (\mathbb{Z}/N\mathbb{Z})^\times \to (\mathbb{Z}/N\mathbb{Z})^\times$, определенный формулой $\rho_j(x) = x^{t2^j}$. Нетрудно видеть, что множество лжесвидетелей содержится в $\rho_{g-1}^{-1}(\{1, -1\})$, тогда мощность множества лжесвидетелей не более $2|\ker \rho_{g-1}|$. Осталось только

оценить мощность ядра. В силу КТО получаем

$$|\ker \rho_j| = \prod_{i=1}^{m} \gcd(t_i 2^{l_i}, t 2^j)$$

Нетрудно видеть, что

$$2^m |\ker \rho_{g-1}| = |\ker \rho_g| \leqslant |\ker \rho_l|$$

Таким образом, всего лжесвидетелей не более чем

$$2|\ker \rho_{g-1}| \leqslant 2^{1-m} |\ker \rho_l|$$

Если делителей хотя бы 3, то лжесвидетелей не более чем $\frac{|\ker \rho_l|}{4}$, то есть не более четверти элементов группы. Если же делителей ровно 2, то N не является числом Кармайкла [проверка этого — задача], следовательно, $|\ker \rho_l| \leqslant \frac{N-1}{2}$, откуда также следует, что лжесвидетелей не более $\frac{N-1}{4}$.

□

Алгоритм AKS

Как мы видели, язык простых чисел лежит в $\mathbf{NP} \cap \mathbf{co}-\mathbf{NP}$. Поскольку не доказано и не опровергнуто утверждение $P = \mathbf{NP} \cap \mathbf{co}-\mathbf{NP}$, из всего вышесказанного не следует полиномиальность языка простых чисел. Полиномиальный алгоритм, детерминированно проверяющий простоту числа, был предъявлен лишь в 2002 году, тремя индусами — Manindra Agrawal, Neeraj Kayal и Nitin Saxena (см. [1], [2]).

Основой вероятностных тестов на проверку простоты была малая теорема Ферма. Существуют, однако, числа Кармайкла — контрпримеры к обратному к МТФ утверждению. Предлагается перейти к кольцу $\mathbb{Z}[x]/(x^r - 1)$ для некоторого удобного r и рассмотреть аналог малой теоремы Ферма, который является критерием и может быть легко использован для построения детерминированного алгоритма.

Сперва мы выберем соответствующее r.

Лемма 3.63. *Существует такое* $r \leqslant \lceil \log^5 n \rceil$, *что порядок* n *в* $\mathbb{Z}/r\mathbb{Z}$ *превышает* $\log^2 n$.

Доказательство. Рассмотрим число $S = n \prod_{j=1}^{\log^2 n} (n^j - 1)$. Покажем, что существует $r \leqslant \lceil \log^5 n \rceil$, на который S не делится. Действительно,

$$S = n \prod_{j=1}^{\log^2 n} (n^j - 1) < n \prod_{j=1}^{\log^2 n} n^j = n^{1+\sum_{j=1}^{\log^2 n} j} \leqslant n^{(\log^2 n)^2} = 2^{\log^5 n}$$

Если x делится на любое число из $[1; \lceil \log^5 n \rceil]$, то x делится на наименьшее общее кратных чисел $1, \ldots \log^5 n$. Однако $\operatorname{lcm}(1, \ldots, m) \geqslant 2^{m-1}$. Таким образом, нужное нам r должно существовать.

Отдельный момент — как доказать, что $\operatorname{lcm}(1,\ldots,n) \geqslant 2^{n-1}$ для лбого натурального n. Это можно сделать по-разному, в работе [1] приведена ссылка на работу Наира [17], который показывает истинность этого утверждения при $n \geqslant 9$. Наир замечает, если

$$I(m,n) = \int_0^1 x^{m-1}(1-x)^{n-m}dx = \sum_{i=0}^{n-m}(-1)^i\binom{n-m}{i}\int_0^1 x^{m+i-1}dx,$$

то $\operatorname{lcm}(1,\ldots,n)I(m,n) \in \mathbb{Z}$ для любого $m \in [0;n]$, а интегированием по частям можно показать, что $I(m,n) = \frac{1}{m\binom{n}{m}}$. Таким образом получаем, что $m\binom{n}{m}|\operatorname{lcm}(1,\ldots,n)$, в частности,

$$n\binom{2n}{n}|\operatorname{lcm}(1,\ldots,2n),$$

$$(2n+1)\binom{2n}{n} = (n+1)\binom{2n+1}{n+1}|\operatorname{lcm}(1,\ldots,2n+1)$$

Таким образом получаем, что $n(2n+1)\binom{2n}{n}|\operatorname{lcm}(1,\ldots,2n+1)$, откуда следует, что

$$\operatorname{lcm}(1,\ldots,2n+1) \geqslant n(2n+1)\binom{2n}{n} \geqslant n\sum_{i=0}^{2n}\binom{2n}{i} = n4^n$$

Таким образом, при $n \geqslant 2$ имеем $\operatorname{lcm}(1,\ldots,2n+1) \geqslant 2^{2n+1}$ и $\operatorname{lcm}(1,\ldots,2n+2) \geqslant \operatorname{lcm}(1,\ldots,2n+1) \geqslant 2^{2n+2}$.

Альтернативное доказательство этого факта также привел Бакир Фархи [10], доказав, что для любого $m \in \mathbb{N}$

$$\operatorname{lcm}\left(\binom{m}{0},\binom{m}{1},\ldots,\binom{m}{m}\right) = \frac{\operatorname{lcm}(1,\ldots,m+1)}{m+1},$$

Из этого утверждения следует, что

$$\operatorname{lcm}(1,\ldots,m) = m\operatorname{lcm}\left(\binom{m-1}{0},\binom{m-1}{1},\ldots,\binom{m-1}{m-1}\right) \geqslant m\max_{i\in[0;m-1]}\binom{m-1}{i} \geqslant$$

$$\geqslant \sum_{i\in[0;m-1]}\binom{m-1}{i} = 2^{m-1}$$

Само доказательство Фархи опирается на *теорему Куммера*: для любых положительных целых $m \geqslant n$ и простого p максимальная степень p, входящая в разложение $\binom{m}{n}$, равна числу переносов, которые совершаются при сложении $(m-n)+n \bmod p$. □

Теперь опишем сам алгоритм и докажем его корректность. Здесь $(n, x^r - 1) \subset \mathbb{Z}[x]$ — идеал, порожденный n и $x^r - 1$. Отметим, что проверка того, является ли число n степенью какого-либо натурального числа, также может быть реализована полиномиально.

```
1:  procedure AKS(N)
2:      if ∃a, b ⩾ 2   a^b = n then
3:          return composite
4:      end if
5:      S ← n ∏_{j=1}^{log^2 n} (n^j − 1)
6:      for t ∈ [1; ⌈log^5 n⌉] do
7:          if S ≠ 0 mod t then
8:              r ← t
9:          end if
10:     end for
11:     for a ∈ [1; r] do
12:         if 1 < gcd(a, n) < n then
13:             return composite
14:         end if
15:     end for
16:     for a ∈ [1; √r log n] do
17:         if (x + a)^n ≠ x^n + a  mod (n, x^r − 1) then
18:             return composite
19:         end if
20:     end for
21:     return prime
22: end procedure
```

Нетрудно видеть, что если n простое, то $(x+a)^n = x^n + a^n = x^n + a$ в $\mathbb{Z}[x]/(n, x^r - 1)$. Теперь покажем, что верно и «обратное» утверждение (есть некоторое условие на n, выполнимость которого можно проверить за полиномиальное по $\log n$ время).

Теорема 3.64 (AKS). *Пусть $n \neq b^c$ ($для\ c \geqslant 2$) — натуральное число, не имеющее простых делителей, не превосходящих r. Если $(x+a)^n = x^n + a \mod (n, x^r - 1)$ для всех $a \in [1; \sqrt{r} \log n]$, то n — простое.*

Доказательство. Пусть $l = \lceil \sqrt{r} \log n \rceil$, а $n > p > r$ — простой делитель числа n. Для всех $0 \leqslant a \leqslant l$ выполнено

$$(x+a)^n = x^n + a \mod (p, x^r - 1), \quad (x+a)^{\frac{n}{p}} = x^{\frac{n}{p}} + a \mod (p, x^r - 1)$$

Первое равенство очевидно, второе можно проверить так: если $(x+a)^{\frac{n}{p}} = x^{\frac{n}{p}} + a + q(x) \mod (p, x^r - 1)$, то $(x+a)^n = x^n + a + [q(x)]^p \mod (p, x^r - 1)$, откуда следует, что $[q(x)]^p = 0 \mod (p, x^r - 1)$, поскольку $x^r - 1$ свободен от квадратов над $\mathbb{Z}/p\mathbb{Z}$ [так как $p > r$,

корней $x^r - 1$, лежащих в $\mathbb{Z}/p\mathbb{Z}$, не больше $p - 1$, а значит, они все различны; корни $x^r - 1$ в соответствующем расширении также различны, и их ровно r], поэтому $q(x) = 0 \mod (p, x^r - 1)$ [так как $q(x)$ делится на все делители $x^r - 1$].

Будем называть число m *интроспективным* для многочлена $f \in \mathbb{Z}[x]$, если

$$[f(x)]^m = f(x^m) \mod (p, x^r - 1).$$

Нетрудно проверить, что если m_1 и m_2 интроспективны для $f(x)$, то $m_1 m_2$ интроспективно для $f(x)$:

$$[f(x)]^{m_1 m_2} = [f(x^{m_1})]^{m_2} = f(x^{m_1 m_2}) \mod (p, x^r - 1).$$

Аналогично если m интроспективно для $f_1(x)$ и $f_2(x)$, то m интроспективно для $f_1(x)f_2(x)$. Число p интроспективно для любого полинома, а $\frac{n}{p}$ интроспективно для многочлена $x+a$, следовательно, любое число из множества $I = \{p^i \left(\frac{n}{p}\right)^j \mid i, j \geqslant 0\}$ интроспективно для любого многочлена из множества $P = \{\prod_{a=0}^{l}(x+a)^{e_a} \mid e_a \geqslant 0\}$.

Теперь пусть $h(x)$ — некоторый неприводимый множитель r-ого циклотомического многочлена, а поле $\mathbb{F} = \mathbb{Z}[x]/(p, h(x))$ — поле. Введем

$$I_r = \left\{ p^i \left(\frac{n}{p}\right)^j \mod r \mid i, j \geqslant 0 \right\}, \quad P_r = \left\{ \prod_{a=0}^{l}(x+a)^{e_a} \mod (r, h(x)) \mid e_a \geqslant 0 \right\}.$$

Нетрудно видеть, что поскольку порядок n в $\mathbb{Z}/r\mathbb{Z}$ не меньше $\log^2 n$, то $|I_r| \geqslant \log^2 n$. Оценим мощность P_r сверху и снизу и получим противоречащие друг другу оценки.

Сначала оценим $|P_r|$ снизу. Покажем, что если $f(x) \neq g(x)$ — многочлены из P степени менее $|I_r|$, то $f(x) - g(x) \neq 0 \mod (p, h(x))$. Допустим, что $f(x) - g(x) = 0 \mod (p, h(x))$. Для любого $m \in I$ и $\widetilde{m} = (m \mod r)$ имеем

$$f(x^{\widetilde{m}}) - g(x^{\widetilde{m}}) = f(x^m) - g(x^m) = [f(x)]^m - [g(x)]^m = 0 \mod (p, h(x))$$

Следовательно, если x — корень $f(x) - g(x)$ над $\mathbb{F}$, то все элементы множества $X = \{x^m \mid m \in I_r\}$ также являются корнями. Поскольку $h(x)$ — неприводимый делитель r-ого циклотомического многочлена, то $h(x)$ не может делить $x^{\widetilde{r}} - 1$ для $\widetilde{r} < r$. Поэтому $x^{m_1} - x^{m_2} \mod (p, h(x))$ тогда $r | m_1 - m_2$, и, таким образом, все элементы X различны. Следовательно, $f(x) - g(x)$ имеет над $\mathbb{F}$ не менее $|I_r|$ корней, откуда следует, что $\deg[f(x) - g(x)] \geqslant |I_f|$, что неверно. Таким образом заключаем, что элементов P_r не меньше, чем число всевозможных остатков многочленов из P степени менее $|I_r|$. Мы можем указать не менее $\binom{|I_r|+l}{|I_r|-1}$ таких многочленов: заметим, что многочлены $x, x+1, \ldots x+l$ лежат в I_r (так как $l = \lceil \sqrt{r} \log n \rceil < r$) и все попарно различны, семейство всевозможных произведений не более $|I_r|$ таких мономов насчитывает ровно $\binom{|I_r|+l}{|I_r|-1}$ (шары и перегородки, а каждому набору корней соответствует ровно один многочлен).

Теперь, пользуясь тем, что $n \neq p^b$, покажем, что $|P_r| \leqslant n^{\sqrt{|I_r|}}$. Рассмотрим

$$I_{\sqrt{|I_r|}} = \{p^i \left(\frac{n}{p}\right)^j \mid 0 \leqslant i, j \leqslant \lfloor\sqrt{|I_r|}\rfloor\}$$

Поскольку $n \neq p^b$, все элементы $I_{\sqrt{|I_r|}}$ различны, и $|I_{\sqrt{|I_r|}}| = (1 + \lfloor\sqrt{|I_r|}\rfloor)^2 > |I_r|$. Таким образом получаем, что существуют такие $m_1, m_2 \in I_{\sqrt{|I_r|}}$, что $x^r - 1 | x^{m_1} - x^{m_2}$. Тогда в любом многочлене можно заменить x^{m_1} на x^{m_2} и получить, что

$$[f(x)]^{m_1} = f(x^{m_1}) = f(x^{m_2}) = [f(x)]^{m_2} \mod (p, h(x))$$

Таким образом, любой элемент P_r является корнем уравнения $y^{m_1} - y^{m_2} = 0$. Корней такого уравнения не больше степени этого многочлена, то есть не больше максимального элемента $I_{\sqrt{|I_r|}}$, который равен

$$\left(\frac{n}{p}\right)^{\lfloor\sqrt{|I_r|}\rfloor} p^{\lfloor\sqrt{|I_r|}\rfloor} \leqslant n^{\sqrt{|I_r|}}.$$

Таким образом мы получили, что

$$\binom{|I_r| + l}{|I_r| - 1} \leqslant |P_r| \leqslant n^{\sqrt{|I_r|}}$$

Теперь покажем, что

$$\binom{|I_r| + l}{|I_r| - 1} > n^{\sqrt{|I_r|}}$$

Действительно, поскольку $|I_r| \geqslant \log^2 n$, то

$$\binom{|I_r| + l}{|I_r| - 1} \geqslant \binom{1 + \lfloor\log(n)\sqrt{|I_r|}\rfloor + l}{\lfloor\log(n)\sqrt{|I_r|}\rfloor} \geqslant \binom{1 + 2\lfloor\log(n)\sqrt{|I_r|}\rfloor}{\lfloor\log(n)\sqrt{|I_r|}\rfloor} > 2^{\lfloor\log(n)\sqrt{|I_r|}\rfloor + 1} \geqslant n^{\sqrt{|I_r|}}$$

Таким образом, $n = p^a$ для некоторого положительного a. Раз $a < 2$, то $a = 1$ и n — простое число. □

Нетрудно видеть, что каждая проверка в алгоритме AKC может быть реализована полиномиальна. Сложность, однако, получается слишком большой: самая грубая оценка дает сложность $\widetilde{O}(n^{\frac{21}{2}})$.

Взятие квадратного корня по модулю

Взятие квадратного корня из a по модулю n — решение уравнения $x^2 = c \mod n$. Следуя книжке Диксона [9], рассмотрим метод Тонелли.

Рассмотрим $x^2 = c \mod p$, где $p = 1 + 2^s\gamma$ — простое (здесь γ — нечетное число), и пусть известен g — квадратичный невычет по модулю p [то есть $g^{2^{s-1}\gamma} = -1$]. Нетрудно видеть, что γ и s вычисляются за $O(\log p)$. Если есть хотя бы одно решение, то $c^{2^{s-1}\gamma} = 1 \mod p$. Подберем значение $\epsilon_0 \in \{0, 1\}$ так, чтобы $c^{2^{s-2}\gamma\epsilon_0} = 1 \mod p$, в таком случае

$$g^{2^{s-1}\gamma\epsilon_0} c^{2^{s-2}\gamma} = 1 \mod p$$

Подберем $\epsilon_1 \in \{0, 1\}$ так, чтобы выполнялось

$$g^{2^{s-1}\gamma\epsilon_1 + 2^{s-2}\gamma\epsilon_0} c^{2^{s-3}\gamma} = 1 \mod p$$

Продолжим подбирать таким же образом $\epsilon_i \in \{0, 1\}$, чтобы

$$g^{2^{s-1}\gamma\epsilon_i + \ldots + 2^{s-1-i}\gamma\epsilon_0} c^{2^{s-2-i}\gamma} = 1 \mod p$$

Обозначим $m = \sum_{j=0}^{s-2} \epsilon_j 2^j$, тогда в итоге получим $g^{2m\gamma} c^\gamma = 1$, откуда получаем решения уравнения:

$$x = \pm g^{m\gamma} c^{\frac{\gamma+1}{2}}$$

Нетрудно убедиться в том, что если заранее известен невычет g, то решения $x^2 = c \mod p$ можно найти за полиномиальное по $\log p$ число операций: найти γ и s можно за $O(\log p)$, при этом $s \leqslant \log p$; все числа $c^\gamma, c^{2\gamma} \ldots c^{2^{s-2}\gamma}$ и $g^\gamma, g^{2\gamma} \ldots g^{2^{s-2}\gamma}$ вычисляются за $O(s \log p) = O(\log^2 p)$ [одно умножение, остальные операции — взятие остатка по модулю p], а дальше, подбирая каждый ϵ_i, нужно перемножать не более $s-2$ чисел; таким образом, суммарное число операций есть $O(\log^3 p)$.

Метод Тонелли элементарно обобщается на уравнения вида $x^2 = c \mod p^k$, где $(p, c) = 1$: если x_0 — решение уравнения $x^2 = c \mod p$, то

$$x_0^{p^{k-1}} c^{\frac{p^k - 2p^{k-1}+1}{2}} \mod p^k$$

является корнем: поскольку $x_0^2 = c + pl \mod p^k$

$$\left(x_0^{p^{k-1}} c^{\frac{p^k-2p^{k-1}+1}{2}}\right)^2 = x_0^{2p^{k-1}} c^{p^k - 2p^{k-1}+1} = (c+pl)^{p^{k-1}} c^{p^{k-1}} \cdot c^{p^k - p^{k-1}} \cdot c \mod p^k$$

По теореме Эйлера $c^{p^k - p^{k-1}} = 1 \mod p^k$, а $(c+pl)^{p^{k-1}} = c^{p^{k-1}} \mod p^k$ по теореме Кумера [раскроем скобки, посмотрим, что все мономы, кроме первого, делятся на p^k], откуда и следует результат.

Чтобы получить решение $x^2 = c \mod p^k$ для любого c, можно воспользоваться *трюком Гензеля*. Допустим, что мы уже нашли такое целое x_{k-1}, что $x_{k-1}^2 = c \mod p^{k-1}$, тогда решение x_k уравнения $x^2 = c \mod p^k$ также удовлетворяет $x^2 = c \mod p^{k-1}$, откуда следует, что $x_k = x_{k-1} + p^{k-1}b$. Таким образом,

$$x_k^2 = (x_{k-1} + p^{k-1}b)^2 = x_{k-1}^2 + 2x_{k-1}p^{k-1}b \mod p^k.$$

Поэтому нам стоит найти b так, чтобы выполнялось

$$2x_{k-1}p^{k-1}b = c - x_{k-1}^2 \mod p^k$$

Это уравнение имеет единственное решение, так как $p \not| 2b$, то $\gcd(2p^{k-1}b, p^k) = p^{k-1}$, а $c - x_{k-1}^2$ делится на p^{k-1}.

Дискретное логарифмирование

Сначала, следуя [20], опишем вероятностный алгоритм поиска порождающего элемента $(\mathbb{Z}/p\mathbb{Z})^\times$. Пусть нам известно разложение $p - 1 = \prod_{i=1}^r p_i^{k_i}$.

```
1: procedure PRIMITIVE-ELEMENT(p)
2:     for i ∈ [1; p] do
3:         while b ≠ 1 do
4:             a ← random ∈ (ℤ/pℤ)^× \ {1}
5:             b ← a^((p−1)/p_i)
6:         end while
7:         q_i ← a^((p−1)/p_i^{k_i})
8:     end for
9:     return ∏_{i=1}^r q_i
10: end procedure
```


Утверждение 3.65. *Алгоритм* PRIMITIVE-ELEMENT *вычисляет примитивный элемент* $(\mathbb{Z}/p\mathbb{Z})^\times$, *среднее время работы равно* $O(\log^4 p)$.

Доказательство. Когда i-ая итерация большого цикла [по $i \in [1; p]$] пройдена, получаем, что раз $a \in (\mathbb{Z}/p\mathbb{Z})^\times \setminus \{1\}$, что

$$q_i^{p_i^{k_i}} = 1, \quad q_i^{p_i^{k_i-1}} \neq 1$$

Отсюда следует, что порядок q_i делит $p_i^{k_i}$, но не равен $p_i^{k_i-1}$ — значит, порядок q_i равен $p_i^{k_i}$. Отсюда очевидно следует, что порядок $\prod_{i=1}^r q_i$ равен наименьшему общему кратному порядков q_i, то есть $p - 1$.

Теперь найдем L_i — среднее число итераций цикла **while** во время i-ой итерации цикла. Поскольку все b, полученные при вычислении $a^{\frac{p-1}{p_i}}$, равновероятны, то вероятность

того, что цикл **while** остановится, равна $\frac{1}{p_i}$. Таким образом, L_i — случайная величина, имеющая геометрическое распределение, и

$$\mathbb{E}[L_i] = \sum_{n\in\mathbb{N}} n\frac{p_i-1}{p_i^n} = \frac{1}{1-\frac{1}{p_i}} \leqslant 2$$

Таким образом, суммарное среднее число итераций цикла **while** есть $\mathbb{E}[L_1+\ldots+L_r] \leqslant 2r$. Каждая итерация цикла вычисляется за $O(\log^3 p)$; поскольку

$$r \leqslant \sum_i k_i \leqslant \sum_i k_i \log p_i = \log p,$$

то среднее время работы равно $O(\log^4 p)$. □

Нетрудно видеть, что полный перебор по всем элементам $e, a, a^2 \ldots$, имеет сложность $O(|\langle a\rangle| \log^2 p)$, где $\langle a\rangle \subset (\mathbb{Z}/n\mathbb{Z})^\times$ — подгруппа, порожденная n. В худшем случае $\langle a\rangle = \Theta(n)$. Существует более эффективный *алгоритм Шенкса*, также известный как *baby step-giant step*, который мы рассмотрим чуть ниже. Считаем, что $(a, n) = 1$.

```
1: procedure Baby-Step-Giant-Step(a, b, n)
2:     m ← ⌈√n⌉
3:     for i ∈ [0, m) do
4:         S[i] ← a^i
5:     end for
6:     y ← b
7:     for i ∈ [0; m) do
8:         if ∃j : a^j = b then
9:             return im + j
10:        else
11:            y ← ya^{-m}
12:        end if
13:    end for
14: end procedure
```

Здесь "baby-step— шаг на $j < m$, а "giant-step— шаг на m. Фактически дискретный логарифм x можно переписать как $mi + j$, где j можно найти поиском по $S[1 \ldots m]$. Поскольку любое число в $[0; n]$ можно переписать как $mi + j$ для некоторых $0 \leqslant i, j < m$, алгоритм очевидно корректен. И baby steps, и giant steps требуют $O(\sqrt{|\langle a\rangle|} \log^2 p)$ битовых операций.

Факторизация целых чисел

Факторизация целого числа — задача, тесно связанная с проверкой простоты. Однако до сих пор нет ни рандомизированного алгоритма, решающего задачу факторизации

за полиномиальное в среднем время. На данный момент самый быстрый рандомизированный алгоритм, совершает в среднем субэкспоненциальное число битовых операций.

Без ограничения общности будем считать n, которое разлагаем на множители, нечетным [найти максимальную степень двойки, делящую n, тривиально можно сделать за полином] и не представимым в виде a^b.

Прежде чем рассмотреть некоторые алгоритмы факторизации целых чисел, разберем один результат, доказанный Neeraj Kayal и Nitin Saxena, соавторами полиномиального алгоритма проверки простоты.

Теорема 3.66 (Kayal, Saxena [2]).
- *Число n может быть разложено на простые множители детерминированным полиномиальным алгоритмом тогда существует полиномиальный алгоритм, находящий некоторый нетривиальный автоморфизм кольца $(\mathbb{Z}/n\mathbb{Z})[x]/(x^2-1)$.*
- *Число n может быть разложено на простые множители в среднем за полиномиальное алгоритмом тогда существует полиномиальный алгоритм, находящий число автоморфизмов кольца $(\mathbb{Z}/n\mathbb{Z})[x]/(x^2)$.*

Доказательство.
- Пусть $\phi : x \mapsto ax + b$ [для некоторых $a, b \in \mathbb{Z}/n\mathbb{Z}$] — автоморфизм $(\mathbb{Z}/n\mathbb{Z})[x]/(x^2-1)$. Покажем, что ϕ — автоморфизм тогда $b = 0$, $a^2 = 1$ в $\mathbb{Z}/n\mathbb{Z}$. Действительно, если $d = \gcd(a, n)$, то

$$\phi\left(\frac{n}{d}x\right) = a\frac{n}{d}x + \frac{n}{d}b = n\frac{a}{d}x + \frac{n}{d}b = \frac{n}{d}b,$$

и ϕ инъективен тогда множество $\{\frac{n}{d}x \mid x \in \mathbb{Z}/n\mathbb{Z}\}$ состоит из одного элемента, то есть 0 [который будет в любом случае], то есть тогда $d = 1$. Более того,

$$\phi(x^2) = (\phi(x))^2 = (ax+b)^2 = a^2x^2 + 2abx + b^2 = a^2 + 2abx + b^2,$$

откуда $a^2 + b^2 = 1 \mod n$ и $ab = 0 \mod n$, что также означает, что $b = 0 \mod n$ (поскольку $\gcd(a, n) = 1$) и $a^2 = 1 \mod n$. Нетрудно видеть, что отображение $\phi : x \mapsto ax$, где $a^2 = 1 \mod n$, является автоморфизмом.

Итак, нетривиальный автоморфизм $(\mathbb{Z}/n\mathbb{Z})[x]/(x^2-1)$ существует тогда есть нетривиальное решение a уравнения $x^2 = 1$, то $\gcd(a+1, n)$ дает нетривиальный делитель n.

Теперь проверим, что если n составное, то существует нетривиальное решение уравнения $x^2 = 1 \mod n$. Это следует из КТО: если $n = n_1n_2$, где $\gcd(n_1, n_2) = 1$, то a, удовлетворяющий $a = 1 \mod n_1$ и $a = -1 \mod n_2$, также удовлетворяет условиям $a^2 = 1 \mod n$ и $a \neq \pm 1 \mod n$.
- Аналогично рассуждениям выше можно показать, что все автоморфизмы кольца $(\mathbb{Z}/n\mathbb{Z})[x]/(x^2)$ имеют вид $x \mapsto ax$, где $(a, n) = 1$. Таких автоморфизмов всего $\phi(n)$, и если $\phi(n)$ известно, то разложение n на простые можно восстановить в

среднем за полиномиальное по $\log n$ время [алгоритм строится аналогично тесту Рабина-Миллера].

□

Метод квадратов Диксона. Начнем с метода Диксона, использующего случайные квадраты по модулю целого числа n. Это был первый метод, работающий за субэкспоненциальное время. Наблюдение, лежащее в основе алгоритма, можно проиллюстрировать на следующем примере.

Пример 3.67. Пусть $n = 2183$. Предположим, что мы уже установили равенства

$$453^2 = 7 \mod n, \quad 1014^2 = 3 \mod n, \quad 209^2 = 21 \mod n.$$

Тогда мы получим $(53 \cdot 1014 \cdot 209)^2 = 21^2 \mod n$, то есть $687^2 = 21^2 \mod n$. Отсюда получаем, что $(687-21)\cdot(687+21)$ делится на n, поскольку $687 \neq \pm 21 \mod n$, то хотя бы один из $\gcd(687-21, n)$ и $\gcd(687+21, n)$ не равен 1 или n. В данном случае мы вообще получаем $\gcd(687-21, n) = 37$, $\gcd(687+21, n) = 59$, оба эти числа простые и $2183 = 37 \cdot 59$.

Здесь мы взяли квадраты, дающие по модулю малые простые [не превосходящие, скажем, некоторого B] и произведения малых простых [в примере нам хватило $3, 7$ и $3 \cdot 7$], и из сравнения $a^2 = b^2 \mod n$ получили нетривиальные делители n.

Далее пусть $B \in \mathbb{R}_+$.

Определение 3.68. Определим *фактор-базу* $P_B = \{p_1, \dots p_t\}$ как множество всех простых чисел, не превосходящих B. Число b будем называть *B-числом*, если

$$b^2 \mod n = p_{i_1} \dots p_{i_k} \text{ для некоторых } p_{i_1}, \dots p_{i_k} \in P_B.$$

Определение 3.69. Натуральное число x называется *y-гладким*, если все его простые делители не превышают y.

Например, степени двойки являются 2-гладкими, числа $1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24$ — 3-гладкие, $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16$ — 5-гладкие.

Оценим плотность y-гладких чисел. Введем

$$\Psi(x, y) = \{m \leqslant x \,|\, m \; is \; y - \text{гладкое}\}, \quad \psi(x, y) = |\Psi(x, y)|.$$

Положим, что m y-гладкое. Тогда $m = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_t^{\alpha_t}$, где $p_1, p_2, \dots, p_t$ — простые числа, не превосходящие y. Сверху функцию $\psi(x, y)$ можно оценить следующим образом:

$$\psi(x, y) = \prod_{i=1}^{t} (1 + \log_{p_i}(x)) \leqslant \prod_{i=1}^{t} (1 + \log_2(x)) = O((\log x)^t) = O((\log x)^{\frac{y}{\log y}})$$

Теперь оценим $\psi(x,y)$ снизу. Максимальная степень y, на которую может делиться x, равна $\log_y(x)$. Нетрудно видеть, что $\sum_{i=1}^{t}\alpha_i \geqslant \log_y(x)+t$. Таким образом,

$$\psi(x,y) \geqslant \binom{\log_y(x)+t}{t} \geqslant \frac{t^{\log_y x}}{(\log_y x)!} = \Omega\left(\frac{t^{\log_y x}}{(\log_y x)^{\log_y x}}\right) = \Omega\left(\frac{x}{(\log_y x)^{2\log_y x}}\right)$$

Теперь опишем сам алгоритм.

(1) Случайно выберем $a \in \mathbb{Z}/n\mathbb{Z}$.
(2) $b = a^2 \mod n$.
(3) Проверить, является ли b B-гладким
(4) Если является [B-гладким], то $b = \prod\limits_{i=1}^{t} p_i^{\alpha_i}$, где $\{p_1, \cdots, p_t\}$ — множество простых, не превышающих B.
(5) Вычислить $t+1$ таких пар $(a_1,b_1),(a_2,b_2),\cdots,(a_{t+1},b_{t+1})$.
(6) $b_j = \prod\limits_{i=1}^{t} p_i^{\alpha_{ij}}$.
(7) Найти β_j такие, что $\sum\limits_{j=1}^{t+1}\beta_j\alpha_{ij}$ четно для каждого i.
(8) $x = \prod\limits_{j=1}^{t+1} a_j^{\beta_j}$ и $y = (\prod\limits_{j=1}^{t+1} b_j^{\beta_j})^{\frac{1}{2}}$.

Число B мы определим позднее. Сначала поймем, почему шаг 7 необходим. Рассмотрим $\prod\limits_{j=1}^{t+1} b_j^{\beta_j}$, где $\beta_j \in \{0,1\}$, для него выполнено

$$\prod_{j=1}^{t+1} b_j^{\beta_j} = \prod_{j=1}^{t+1}\prod_{i=1}^{t} p_i^{\beta_j\alpha_{ij}} = \prod_{i=1}^{t} p_i^{\sum\limits_{j=1}^{t+1}\beta_j\alpha_{ij}}$$

Если $\sum\limits_{j=1}^{t+1}\beta_j\alpha_{ij}$ четна для всех $i \in [1;t]$, то само произведение является квадратом. Поэтому задача поиска β_j таких, что все суммы $\sum\limits_{j=1}^{t+1}\beta_j\alpha_{ij}$ четны, эквивалентна поиску решения системы линейных уравнений $\{\sum\limits_{j=1}^{t+1}\beta_j\alpha_{ij} = 0\}$ над $\mathbb{F}_2$.

Нетрудно также проверить, что x и y удовлетворяют $x^2 = y^2 (\mod n)$:

$$x^2 = \prod_{j=1}^{t+1} a_j^{2\beta_j} = \prod_{j=1}^{t+1} b_j^{\beta_j} \mod n = y^2 \mod n$$

Чтобы оценить вероятность успешного выполнения операций (4)-(8), надо оценить T — число B-гладких чисел в $\mathbb{Z}/n\mathbb{Z}$ вида $a^2 \mod n$. Легко видеть, что

$$T \geqslant \psi(\sqrt{n}, k) \approx (\frac{\frac{1}{2}\ln n}{\ln k})^{\frac{\frac{1}{2}\ln n}{\ln k}}$$

Но нам нужна также нижняя оценка T.

Лемма 3.70. *Предположим, что n не делится ни на одно из простых $p_1 < \ldots < p_t$, а $r \in \mathbb{N}$ таково, что $p_r^{2h} < n$. Обозначим*

$$S = \{b \in \mathbb{N} \mid 1 \leqslant b \leqslant N,\ b^2 \mod N \in \Psi(N, p_t)\}$$

Тогда

$$|S| \geqslant \frac{h^{2r}}{(2r)!}$$

Доказательство. Для начала доказательства положим $n = q_1^{l_1} \ldots q_t^{l_t}$ и определим квадратичный характер χ_i на $(\mathbb{Z}/q_i^{l_i}\mathbb{Z})^\times$ как символ Якоби по модулю $q_i^{l_i}$:

$$\chi_i(a) = \left(\frac{a}{q_i^{l_i}}\right)$$

Отображение $\chi_i : (\mathbb{Z}/q_i^{l_i}\mathbb{Z})^\times \to \mathbb{Z}/2\mathbb{Z}$ является групповым гомоморфизмом. Тогда мы можем построить отображение $\chi : (\mathbb{Z}/n\mathbb{Z})^\times \to G = (\mathbb{Z}/2\mathbb{Z})^t$ таким образом:

$$a \mod n \mapsto \Big(\chi_1(a \mod q_1^{l_1}), \ldots \chi_t(a \mod q_t^{l_t})\Big)$$

Пусть

$$Q = \{a \in \mathbb{N} \mid 1 < a < n,\ \gcd(a, n) = 1,\ \exists b \in \mathbb{N}\ a = b^2 \mod n\}$$

множество (обратимых) квадратов по модулю n. Согласно КТО число является квадратичным вычетом по модулю n тогда и только тогда когда он является квадратичным вычетом по модулю каждого $q_i^{l_i}$, то есть

$$a \in Q \iff \chi(a) = (1, \ldots, 1)$$

Кроме того, обратимый квадрат по модулю $q_i^{l_i}$ имеет ровно два квадратных корня [q_i — нечетное простое], и согласно КТО любой $\in Q$ имеет 2^t квадратных корней [по модулю n].

Теперь, для $x \in \mathbb{R}$ и $s \in \mathbb{N}$ положим

$$T_s(x) = \{a \in \mathbb{N} \mid a \leqslant x,\ \exists e_1, \ldots, e_t \in \mathbb{N}\ a = p_1^{e_1} \ldots p_t^{e_t} \text{ и } e_1 + \ldots + e_t = s\},$$

то есть $T_s(x)$ — набор p_h-гладких целых чисел не больше x с ровно s (не обязательно отличными) простыми делителями. Разобьем $T_r(\sqrt{n})$ на 2^t подмножеств U_g для $g \in G$:

$$U_g = \{a \in T_r(\sqrt{n}) : x \mod n = g\}.$$

Обозначим за V образ отображения умножения:

$$\mu : \bigcup_{g \in G} (U_g \times U_g) \to \mathbb{N}, \quad \mu(b,c) = bc \mod n$$

Поскольку $\chi(bc \mod n) = (1, \ldots, 1)$ для всех $b, c \in U_g$ и $g \in G$, получим $V \subset Q$. Кроме того, $V \subset T_{2r}(n)$, так что $V \subset T_{2r}(n) \cap Q$.

Каждый элемент в $T_{2r}(n) \cap Q$ имеет ровно 2^t квадратных корней [по модулю n] и все эти корни лежат в S, поэтому $|S| \geqslant 2^t |T_{2r}(n) \cap Q|$. Теперь оценим количество элементов $(b,c) \in \bigcup_{g \in G} U_g \times U_g$ склеиваются умножением μ. Поскольку $b, c \leqslant \sqrt{n}$ и $bc = a \mod n$, тогда $bc =$ в $\mathbb{N}$. Таким образом, мы должны разбить $2r$ простых множителей на два подмножества, чтобы получить b и c [как произведения по подмножествам], и существует не более $\binom{2r}{r}$ способов сделать это. Таким образом,

$$|V| \frac{(2r)!}{(r!)^2} \geqslant |\bigcup_{g \in G} (U_g \times U_g)| = \sum_{g \in G} |U_g|^2$$

Таким образом,

$$|S| \geqslant 2^t |T_{2r}(n) \cap Q| \geqslant 2^t |V| \geqslant 2^t \frac{(r!)^2}{(2r)!} \sum_{g \in G} |U_g|^2$$

Теперь применим неравенство Коши-Буняковского:

$$2^t \sum_{g \in G} |U_g|^2 = \left(\sum_{g \in G} 1 \right) \left(\sum_{g \in G} |U_g|^2 \right) \geqslant \left(\sum_{g \in G} 1 \cdot |U_g| \right)^2 = |T_r(\sqrt{n})|^2$$

Поскольку $p_t^r \leqslant \sqrt{n}$, элемент $T_r(\sqrt{n})$ распадается в произведение r простых чисел, не превосходящих p_h, поэтому

$$|T_r(\sqrt{n})| = \binom{t+r-1}{r} \geqslant \frac{t^r}{r!}.$$

Таким образом,

$$|S| \geqslant 2^t \frac{(r!)^2}{(2r)!} \sum_{g \in G} |U_g|^2 \geqslant \frac{(r!)^2}{(2r)!} |T_r(\sqrt{n})|^2 \geqslant \frac{t^{2r}}{(2r)!}$$

□

Таким образом, нам понадобится в среднем не более $\frac{N(2r)!}{t^{2r}}$ попыток для поиска подходящего B-числа. Согласно теореме о распределении простых чисел $t > \frac{B}{\log B}$ [для $B \geqslant 59$]. Зафиксируем $r \in \mathbb{N}$ и положим $B = n^{1/2r}$. Теперь оценим снизу число попыток:

$$\frac{n}{t^{2r}}(2r)! < \frac{n(\log B)^{2r}}{B^{2r}}(2r)^{2r} = (\log n)^{2r}$$

Отсюда получаем, что сложность метода Диксона равна

$$\tilde{O}(B^3 + Bn^{2r+2})$$

Подберем B так, чтобы оба слагаемых имели одинаковый порядок, тогда получим $3n/2r \sim n/2r + 2r\log n$, или же

$$r = \lceil \frac{n}{2\log n} \rceil.$$

Отсюда получаем следующее утверждение.

Теорема 3.71. *Алгоритм Диксона совершает в среднем* $\tilde{O}\left(\exp\left(\sqrt{\frac{9}{2}\log n \log\log n}\right)\right)$ *битовых операций.*

$(p-1)$-метод Полларда. Пусть нам надо разложить число $n = pq$ на простые множители. Пусть $p-1$ — k-гладкое число. Пусть $K = (k!)^{lg\, p}$. Согласно малой теореме Ферма $a^K = 1 \mod p$. Допустим также, что $q-1$ не является k-гладким. Тогда $a^K = 1 \mod q$ для «немногих a»: если $a^K = 1 \mod q$, то $a^{gcd(K,q-1)} = 1 \mod q$, и не более $gcd(K, q-1)$ остатков удовлетворяют $a^{gcd(K,q-1)} = 1 \mod q$ и $gcd(K, q-1) \leqslant \frac{q-1}{2}$.

Таким образом, можно построить следующий алгоритм разложения на простые множители. Пусть n — положительное целое.

```
1: procedure (p − 1)-Pollard(n)
2:     for k ∈ ℕ \ {1} do
3:         взять случайный a ∈ ℤ/nℤ
4:         K ← (k!)^(⌊lg n⌋)
5:         b ← a^K mod n
6:         d ← gcd(b − 1, n)
7:         if 1 < d < n then
8:             return d
9:         else
10:            return failure
11:        end if
12:    end for
13: end procedure
```

Если k выбран корректно, то алгоритм выше возвращает собственный делитель n с вероятностью не менее $\frac{1}{2}$. Поскольку

$$(k!)^{\log n} = ((k-1)!)^{\log n} \cdot k^{\log n},$$

на втором шаге совершается $\tilde{O}(k \log n \cdot \log k)$ битовых операций. На третьем шаге надо совершить $\tilde{O}(k \log^2 n \cdot \log k)$ битовых операций, а на четвертом — $\tilde{O}(log\, n)$ битовых операций. Таким образом, сложность алгоритма выше $\tilde{O}(k^2 log^2 n \cdot log\, k)$.

Факторизация многочленов. Алгоритм Кантора-Цассенхауса

Нам дан полином f над полем $\mathbb{F}_p$ степени n, наша задача — разложить его на множители. Мы хотим найти его разложение на множители в виде

$$f = g_{11} g_{12} \cdots g_{1m_1} g_{21} \cdots g_{dm_d}$$

где каждый многочлен g_{ij} — неприводимый множитель f степени i. Сначала попробуем по данному f получить $\prod_j g_{ij}$ для каждого i, то есть произведения всех неприводимых множителей f степени i. Эту процедуру мы будем называть *distinct degree factorization.* Мы хотим найти полиномиальный алгоритм для поиска данного разложения. Нетрудно убедиться, что размер входа многочлена [заданного набором коэффициентов] есть $n \log p$, так что нам нужен алгоритм сложности $O\left((n \log p)^c\right)$ для c.

Для начала приведем многочлен f к многочлену $\tilde{f}$, свободному от квадратов, не разлагая f на множители. Заметим, что если $f = h_1^{d_1} \dots h_k^{d_k}$ [где h_i — неприводимые делители], то

$$f' = h_1^{d_1 - 1} \dots h_k^{d_k - 1} \left(h_1 \dots h_k \sum_{i \in [1;k]} \frac{d_1}{h_1} \right),$$

откуда

$$\gcd(f, f') = h_1^{d_1 - 1} \dots h_k^{d_k - 1} \gcd \left(h_1 \dots h_k, h_1 \dots h_k \sum_{i \in [1;k]} \frac{d_1}{h_1} \right),$$

и многочлен $h_1 \dots h_k \sum_{i \in [1;k]} \frac{d_1}{h_1}$ взаимнопрост с любым h_i, так как он может быть представлен как $h_i P(x) + d_i h_1 \dots h_{i-1} h_{i+1} \dots h_k$, откуда

$$\gcd(h_i P(x) + d_i h_1 \dots h_{i-1} h_{i+1}, h_i) = \gcd(d_i h_1 \dots h_{i-1} h_{i+1}, h_i) = 1,$$

поскольку все многочлены h_j неприводимы. Поэтому если f не свободен от квадратов, вместо него в дальнейшем можно рассмотреть $\frac{f}{\gcd(f, f')}$.

В случае конечных полей под производной мы будем понимать отображение $x^m \mapsto m x^{m-1}$. Все те же рассуждения имеют место. Однако в случае конечных полей производная от непостоянного многочлена может быть нулевой: производная $f(x) = x^{2p} - 3x^p + 5$ над полем $\mathbb{F}_p$ равна 0. Нетрудно видеть, что над $\mathbb{F}_p$ производная многочлена f равна 0

титтк все его мономы имеют степень кратную p, то есть $f(x) = g(x^p)$ для некоторого многочлена g. Таким образом, если $f' = 0$, то $f(x) = g(x^p)$, и мы переходим к рассмотрению $g(x)$. Если же $g' = 0$, то с ним делаем то же самое. В конце концов получится многочлен, хотя бы один моном которого имеет степень не кратную p, а значит, и производная такого многочлена не равна нулю тождественно.

Для нахождения $\prod_j g_{ij}$ по данному f нам стоит воспользоваться следующей формулой:

$$x^{p^m} - x = \prod_{f \in \mathbb{F}_p[x] \text{ – неприводимый, } \deg f = d, d|m} f(x),$$

В частности, для $m = 1$ получаем, что $x^p - 1$ — произведение всех неприводимых линейных многочленов над $\mathbb{F}_p$, причем каждый содержится с единичной кратностью. Таким образом, $g_1 = gcd(f, x^p - x)$ будет произведением всех неприводимых делителей f первой степени — ведь $x^p - x = \prod_{a \in \mathbb{F}_p}(x - a)$, а НОД двух многочленов будет состоять ровно из неприводимых делителей f первой степени. Теперь пусть $f_2 = \frac{f}{g_1}$ и $g_2 = gcd(f_2, x^{p^2} - x)$ будет в точности произведением неприводимых делителей f степени 2 [$x^{p^2} - x$ раскладывается в произведение всех неприводимых многочленов степени 1 и 2, на произведение многочленов степени 1 мы уже поделили.]

1: **procedure** DistinctDegreeFactorization($f(x) \in \mathbb{F}_p[x]$ степени n.)
2: $f_0 = f$
3: **for** $i \in [1; n]$ **do**
4: Используя быстрое возведение в степень $s_i = x^{p^i} \mod f_{i-1}$
5: $g_i = \gcd(f_{i-1}, s_i - x)$.
6: $f_i = \frac{f_{i-1}}{g_i}$
7: **end for**
8: **return** $\{g_1, g_2, \cdots, g_n\}$.
9: **end procedure**

Алгоритм Distinct Degree Factorization находит разложение $f = g_1 \dots g_n$, в котором g_k — произведение неприводимых делителей степени k. Нетрудно убедиться в том, что этот алгоритм полиномиален.

Теперь мы переходим к разложению многочлена из $\mathbb{F}_p$ на простые множители. Как и в случае с числами, нам достаточно найти хотя бы один нетривиальный делитель.

Согласно рассуждениям выше мы можем считать, что f свободен от квадратов — в противном случае мы приведем его к многочлену, свободному от квадратов. Для свободного от квадратов многочлена можно посчитать DDF и проверить, является ли этот многочлен неприводимым [почему это можно сделать — упражнение для читателя].

С каждым g_i, полученным алгоритмом Distinct Degree Factorization, будем работать отдельно. Так что считаем, что наш $f \in \mathbb{F}_p[x] = g_1 \cdots g_m$ таков, что все g_i неприводимы, отличны и имеют степень d.

Тогда в этих предположениях в силу КТО имеем

$$\mathbb{F}_p[x]/(f(x)) \simeq \mathbb{F}_p[x]/(g_1(x)) \times \cdots \times \mathbb{F}_p[x]/(g_m(x))$$

Поскольку каждый g_i — неприводимый многочлен степени d, то $\mathbb{F}_p[x]/(g_i(x)) \simeq \mathbb{F}_{p^d}$. Поэтому

$$\mathbb{F}_p[x]/(f(x)) \simeq \mathbb{F}_{p^d} \times \cdots \times \mathbb{F}_{p^d}$$

Более того,

$$(\mathbb{F}_p[x]/(f(x)))^\times \simeq \mathbb{F}_{p^d}^\times \times \cdots \mathbb{F}_{p^d}^\times$$

Если g — делитель нуля в $\mathbb{F}_p[x]/(f)$, то его образ при гомоморфзиме из КТО есть $(a_1, a_2, \cdots, a_m)$, где $a_i = 0$. Более того, если этот ненулевой делитель нетривиален, то $a_j \neq 0$ для некоторого $j \neq i$. Иными словами, g имеет нуль по i-ой координате, что значит, что g делится на g_i, таким образом, $g \neq 1$. Более того, g ненулевой по j-координате и, таким образом, g_j не делит g, то есть $g \neq f$. Таким образом, $gcd(g, f)$ не равен ни f, ни 1 и является тогда нетривиальным делителем f.

Таким образом, задача поиска нетривиального делителя сводится к поиску делителей нуля в $\mathbb{F}_p[x]/(f(x))$. Теперь надо прибегнуть к рандомизации. Рассмотрим случайный элемент $\mathbb{F}_p[x]/(f(x))$ — некоторый многочлен $a(x)$ степени менее n. Если нам повезло и $gcd(a, f) \neq 1$, то мы получили нетривиальный делитель f. Тогда будем считать, что a — не делитель нуля в нашем кольце [в частности, $a \in (\mathbb{F}_p[x]/(f)^\times)$.]

Поскольку мы не знаем делителей g_i, мы не знаем явно соответствующего морфизма $\mathbb{F}_p[x]/(f(x)) \simeq \mathbb{F}_{p^d} \times \cdots \times \mathbb{F}_{p^d}$, хотя и знаем о его существовании. Однако если одна из координат a при таком отображении равна -1, то можно рассмотреть образы $a(x) + 1$, и соответствующая координата у $a(x) + 1$ равна 0. Значит, $a(x) + 1$ является делителем нуля. Возможно, однако, что все координаты $a(x) + 1 = 0$ равны -1. Поэтому мы желаем найти элементы, не все координты которых равны -1. Мультипликативная группа $\mathbb{F}_{p^d}^\times$ — абелева группа of order $p^d - 1$, так что для любого ее элемента b верно $b^{p^d-1} = 1$, поэтому нам достаточно возвести a в степень $(p^d - 1)/2 = M$, чтобы получить ± 1.

Таким образом получим следующую процедуру. Возьмем случайный многочлен $a(x)$ степени менее n, проверим $\gcd(a, f)$. Если $\gcd(a, f) \neq 1$, то вычислим $a(x)^{(p^d-1)/2} \bmod f(x)$. Если a отображалось по КТО в $(a_1, a_2, \cdots, a_m)$, то a^M отображается в $(a_1^M, \cdots, a_m^M)$, в данном случае каждый из a_i^M равен либо 1, либо -1.

Утверждение 3.72. *Каждый $a_i^M = 1$ с вероятностью 1/2, и все события $\{a_i^M = 1\}$ независимы.*

Доказательство. Поскольку все g_i различны, в силу КТО события $\{a_i^M = 1\}$ независимы. Чтобы удостовериться в том, что вероятность события $\{b^M = 1\}$ равна $1/2$, рассмотрим отображение

$$\psi : \mathbb{F}_{p^d}^{\times} \longrightarrow \mathbb{Z}/2\mathbb{Z}, \quad b \longrightarrow b^M$$

Нетрудно видеть, что ψ является гомоморфзизмом. Поэтому ядро ψ — подгруппа $\mathbb{F}_{p^d}^{\times}$ элементов b таких, что $b^m = 1$. Классов смежности по этой подгруппе два, второй класс — множество элементов b таких, что $b^M = -1$. Поскольку мощности классов смежности два, то вероятность события $\{b^M = 1\}$ равна $1/2$. □

Поэтому каждая координата a_i равна 1 или -1 с вероятностью $1/2$. Поэтому вероятность того, что все координаты равны либо 1, либо -1, равна $1/2^m$. Поэтому с вероятностью хотя бы $1 - 2^{m-1}$ мы найдем вектор, в котором есть координаты 1 и -1 одновременно.

1: **procedure** CANTOR-ZASSENHAUS($f \in \mathbb{F}_p[x]$ of degree n.)
2: **if** f не свободен от квадратов **then**
3: **return** $\widetilde{f}$, полученный из f приведением квадратов
4: **end if**
5: $\{g_1, g_2, \cdots, g_n\}$ = DISTINCTDEGREEFACTORING(f)
6: **if** $g_n \neq 1$ **then**
7: **return** IRREDUCIBLE
8: **end if**
9: **for** $g_i \neq 1$ **do**
10: EQUALDEGREEFACTORIZE(g_i, d)
11: **end for**
12: **end procedure**

С вероятностью не менее $1 - 2^{m-1}$ получится делитель нуля, а значит, $gcd(a^M + 1, f)$ — нетривиальный делитель f.

1: **procedure** EQUALDEGREEFACTORIZE($f \in \mathbb{F}_p[x]$ of degree n, d — степень всех его неприводимых делителей)
2: взять случайный многочлен $a(x)$ степени меньше n.
3: **if** $gcd(a, f) \neq 1$ **then**
4: **return** $gcd(a, f)$
5: **end if**
6: $M \leftarrow (p^d - 1)/2$.
7: вычислить $a'(x) = a(x)^M + 1$.
8: **if** $a' \neq 0$ and $gcd(a', f) \neq 1$ **then**
9: **return** $gcd(a', f)$
10: **end if**

11: **end procedure**

Алгоритм Берлекемпа

Теперь рассмотрим другой алгоритм факторизации многочлена над конечным полем — алгоритм Берлекемпа. Сначала, как и раньше, мы проверим, является ли входной $f(x) \in \mathbb{F}_p[x]$ свободным от квадратов. Далее будем считать, что

$$f = f_1 f_2 \cdots f_m$$

гле f_i — различные неприводимые делители f. По КТО получаем

$$R = \mathbb{F}_p[x]/(f) = (\mathbb{F}_p[x]/(f_1)) \times (\mathbb{F}_p[x]/(f_2)) \times \cdots \times (\mathbb{F}_p[x]/(f_m))$$

Пусть $d_i = \deg f_i$, а $n = \deg f$. Обозначим за B отображение $T - I$, где I — тожественное отображение. Тогда B отображает любой $a \in R$ в $a^p - a$.

Определение 3.73. *Подалгеброй Берлекемпа* будем называть $\mathcal{B} = \ker(B) = \ker(T - I)$.

Пусть $a \in \mathcal{B}$. По КТО ему соответствует элемент $(a_1, a_2, \cdots, a_m)$ в прямой сумме слагаемых $\mathbb{F}_p[x]/(f_i)$, в данном случае

$$a^p - a \mapsto (a_1^p - a_1, \cdots, a_m^p - a_m).$$

В силу того, что $a \in \mathcal{B}$, $a_i^p - a_i = 0$ для любого i. Заметим, что $a_i^p - a_i$ является элементом $\mathbb{F}_p[x]/(g_i) \simeq \mathbb{F}_{p^{d_i}}$ и следовательно $a_i^p - a_i = 0$ верно титтк $a_i \in \mathbb{F}_p$. Нетрудно видеть, что элементы $\mathbb{F}_{p^d}$, для которых $x^p - x = 0$, образуют в точности $\mathbb{F}_p \subset \mathbb{F}_p[x]/(g_i)$.

Таким образом, каждая координата набора $(a_1, a_2, \cdots, a_m)$ будет элементом поля $\mathbb{F}_p$ и, следовательно,

$$\mathcal{B} \simeq \mathbb{F}_p \times \cdots \times \mathbb{F}_p$$

Таким образом, каждая координата будет элементом $\mathbb{F}_p$ и, следовательно,

$$\mathcal{B} \simeq \mathbb{F}_p \times \cdots \times \mathbb{F}_p$$

Поскольку $\mathcal{B}$ — произведение m копий $\mathbb{F}_p$, $\mathcal{B}$ — m-мерное пространство R над $\mathbb{F}_p$.

Очевидно, что $\{1, x, x^2, \cdots, x^{n-1}\}$ задает базис для R, надо найти базис для $\mathcal{B}$. Пусть оператор T действует на R следующим образом:

$$T(x^i) = \sum_{j=0}^{n-1} \alpha_{ji} x^j$$

Тогда можно рассмотреть матрицу $(\alpha_{ji})_{i,j}$, соответствующую T, и действие T — умножение на эту матрицу.

Тогда матрицу для B можно задать как $\hat{B} = (\alpha_{ji})_{i,j} - I$. Следовательно, ядро этого отображения — множество векторов v таких, что $\hat{B}v = 0$, — может быть найдено с помощью алгоритма Гаусса.

Имея базис для $\{b_1, b_2, \cdots, b_m\}$, мы можем взять случайный элемент $\mathcal{B}$, просто рассмотрев m случайных элементов a_m из $\mathbb{F}_p$ и $\sum a_i b_i$ будет нашим случайным элементом $\mathcal{B}$. Любой элемент $a \in \mathcal{B}$ отправляется в $\mathbb{F}_p \times \cdots \times \mathbb{F}_p$ по КТО. Таким образом, мы можем применить идею алгоритма Кантора-Цассенхауса: $a^{\frac{p-1}{2}}$ соответствует вектору из 1 и -1.

procedure BerlekampFactorization($f \in \mathbb{F}_p[x]$ степени n)
    Сделать f свободным от квадратов.
    $R \leftarrow \mathbb{F}_p[x]/(f)$, рассмотренное как n-мерное линейное пространство над $\mathbb{F}_p$.
    Построить матрицу $\hat{B}$, соответствующую отображению $a \mapsto a^p - a$.
    Используя алгоритм Гаусса, найти базис $\{b_1, b_2, \cdots, b_m\}$ подалгебры Берлекемпа $\mathcal{B}$.
    Взять $\{a_1, \cdots, a_{m-1}\} \in_R \mathbb{F}_p$, $b \leftarrow \sum a_i b_i$.
    if $gcd(b^{\frac{p-1}{2}} + 1, f) \neq 1$ **then**
        return $gcd(b^{\frac{p-1}{2}} + 1, f)$
    end if
end procedure

Теоретико-групповые алгоритмы

Теперь поговорим об алгоритмах в теории групп. Мы будем иметь дело с подгруппами $Sym(\Omega)$ — группы перестановок некоторого конечного множества Ω. Мы будем использовать нотацию Вейландта: пусть $\phi : G \times X \to X$ — действие, тогда будем обозначать $\phi(g)(x)$ как x^g. Будем также называть группу $G \subset Sym(\Omega)$ *транзитивной*, если ее действие на Ω транзитивно.

Вычисление орбиты элемента является одним из основных вопросов в теоретико-групповых алгоритмах.

Нам дана группа G, которая действует на конечном множестве Ω, и, следовательно, G можно рассматривать как подгруппу в $Sym(\Omega)$. Поскольку G может быть очень большой, в качестве ввода также дан небольшой порождающий набор A группы G. Учитывая элемент $\alpha \in \Omega$, мы бы хотели вычислить G-орбиту α, обозначаемую α^G. Напомним, что

$$\alpha^G = \{\beta \in \Omega \mid \exists g \in G, \beta = \alpha^g\}$$

Наивным способом является попытка «добраться» до каждого элемента на орбите, используя действие элементов A на α, а именно:

procedure OrbitComputation(A, α)
    $\Delta = \{\alpha\}$
    while Δ растет **do**
        for $a \in A, \delta \in D \in \Delta$ **do**
            $\Delta = \Delta \cup \{\delta^g\}$
        end for

7: **end while**
8: **end procedure**

Время выполнения по крайней мере квадратично по $|\Omega|$.

Впоследствии мы можем воспользоваться подходом «разделяй и властвуй»: как только мы получим разложение по орбите Ω, мы можем изучить действие группы на каждой орбите по отдельности. Очевидно, что на каждой орбите действие группы является транзитивным. Для изучения транзитивно действующих групп изучим «блоки». Будем предполагать, что действие G транзитивно.

Определение 3.74. Множество $\Delta \subseteq \Omega$ называется *блоком*, если для любого $g \in G$ либо $\Delta^g = \Delta$, либо $\Delta^g \cap \Delta = \varnothing$. Группа G называется *примитивной*, если все ее блоки — либо одноэлементные множества, либо все Ω.

Рассмотрим некоторые примеры.

(1) Нетрудно видеть, что $Sym(\Omega)$ примитивна.
(2) Если $G = A_{|\Omega|}$ — множество чётных перестановок над Ω, у нее достаточно элементов, чтобы все еще оставаться примитивной. Это также можно легко проверить.

Предположим, что G действует на себе, скажем, путем умножения слева, $G \leqslant Sym(G)$. Каждая подгруппа в G является блоком, так как смежные классы либо совпадают, либо непересекающиеся. В действительности любой смежный класс также будет блоком.

Таким образом, у нас есть замощение Ω такими блоками, мы будем называть их *блочной системой*.

Определение 3.75. *Блочная система* — разбиение множества Ω на блоки заданного действия группы G на Ω.

Обратите внимание, что если Δ является блоком, то и Δ^g для каждого $g \in G$ также является блоком. Для этого замощения верна теорема, очень похожая на теорему Лагранжа о подгруппах группы. Обратите внимание, что в таком случае действие должно быть транзитивным, иначе такие блоки Δ^g не покроют все множество Ω.

Теорема 3.76. *Пусть G действует транзитивно на множестве Ω и пусть $\Delta \subseteq \Omega$ будет любым блоком. Тогда $|\Delta|$ делит $|\Omega|$.*

Эта теорема сразу приводит к следующему следствию.

Утверждение 3.77. *Любая группа G, действующая на Ω транзитивно, примитивна, если $|\Omega|$-е число.*

Пример 3.78. Пусть X — граф, набор из некоторых k треугольников. Рассмотрим группу его автоморфизмов, действующих на него. Во-первых, обратите внимание, что это действие транзитивно, и, кроме того, каждый треугольник будет блоком.

Таким образом, группа автоморфизмов примитивна.

Последний пример хорошо мотивирует использовать подход «разделяй и властвуй». Если G транзитивна и примитивна, пусть Δ будет наименьшим блоком. Теперь групповые элементы Ω соответствуют блочной системе, порожденной Δ. Обратите внимание, что G действует на этой системе блоков перестановками. А также пусть система блоков будет новым набором Ω_1, а группа будет проективной версией G и теперь мы можем задать вопрос «Является ли G' примитивной или транзитивной?» относительно меньшего множества Ω.

Проверка, является ли действие транзитивным, может быть выполнена с помощью вычисления орбиты, но нам нужно также проверить, является ли группа примитивной. Заметим, что если Δ_1 и Δ_2 являются G-блоками, то и $\Delta_1 \cap \Delta_2$ также является блоком. Теперь мы можем рассмотреть наименьший блок, содержащий набор элементов Ω.

Лемма 3.79. *Группа $G \leqslant Sym(\Omega)$, действующая транзитивно на Ω, примитивна если и только если G_α-максимальная подгруппа в G.*

Доказательство. Обратите внимание, что α указывать не нужно, поскольку в силу транзитивности действия G G_α и G_β сопряжены друг с другом. Легко проверить, что для любого $g \in G$ такого, что $g\alpha = \beta$, выполняется $gG_\alpha g^{-1} = G_\beta$.

Предположим, что $\{\alpha\} < \Delta < \Omega$ — нетривиальный блок. Пусть $H = \{g \in G \mid \Delta^g = \Delta\}$. Мы покажем, что $G_\alpha < H < G$, доказав тем самым лемму в одну сторону.

Ясно, что поскольку G действует транзитивно и $\Delta < \Omega$, H — собственная подгруппа в G. Также, если $g \in G_\alpha$, то $\alpha \in \Delta \cap \Delta^g \neq \varnothing$. Так как Δ является блоком, то $\Delta = \Delta^g$ и, следовательно, $g \in H$. Так как $\{\alpha\} < \Delta$ существует $\beta \in \Delta$, отличный от α. Пусть g — элемент группы, который переводит α в β. Тогда $\beta \in \Delta \cap \Delta^g$, $g \in H$, но $g \notin G_\alpha$. Таким образом, $G_\alpha < H$.

Теперь докажем теорему в другую сторону. Пусть $G_\alpha < H < G$, покажем тогда, что $\alpha^H = \Delta$ — наш нетривиальный блок. Так как $G_\alpha < H$, $\Delta \neq \{\alpha\}$. Показать $\Delta < \Omega$ немного сложнее. Поскольку G_α является подгруппой в G, G_α и его классы смежности по нему разбивают G:

$$G = \bigcup_{\beta \in \Omega} \bigcup_{g_\beta : \alpha \mapsto \beta} G_\alpha g_\beta$$

И обратите внимание, что если любой $g_\beta \in H$, то весь класс $G_\alpha g_\beta$ содержится в H. Следовательно, так как $\Delta = \Omega$ подразумевает $H = G$, наше предположение $H < G$ влечет $\Delta < \Omega$.

Осталось только показать, что Δ является блоком. Предположим, что $\Delta^g \cap \Delta \neq \varnothing$, то для некоторого $h, h' \in H$, $\alpha^{hg} = \alpha^{h'}$, откуда $h'gh^{-1} \in G_\alpha < H$. Следовательно, $g \in H$ и, таким образом, $\Delta^g = \Delta$. □

Установленная выше лемма является взаимно-однозначным соответствием между подгруппами в G и блоками в Ω.

Можно также думать, что G действует на $\{G_\alpha g \mid \alpha \in \Omega\}$, сопоставив каждой точке $\alpha \in \Omega$ подгруппу G_α и ее смежные классы.

Лемма 3.80. *Пусть $N \lhd G$ — нормальная транзитивная подгруппа в G. Тогда орбиты N формируют блочную систему.*

Доказательство. Мы хотим показать, что α^N — блок. Предположим, $\alpha^{Ng} \cap \alpha^N \neq \varnothing$, тогда $\alpha_{n_1 g} = \alpha_{n_2}$ для некоторых $n_1, n_2 \in N$ и, следовательно, $n_1 g n_2^{-1} \in G_\alpha$. В силу нормальности N вышеприведенные элементы можно выразить как $n_3 g$ для некоторого $n_3 \in N$. Отсюда $n_1 g n_2^{-1} \in Ng$ и, следовательно, $n_2 \in Ng$, тогда Ng и N совпадают, тем самым $\alpha^{Ng} = \alpha^N$. □

Вывод 3.81. Если G примитивна, то все ее нормальные подгруппы транзитивны.

Поиск блочного разложения. Теперь перейдем к такой задаче: по входной группе $\langle A \rangle = G \leqslant Sym(\Omega)$ найти нетривиальную блочную систему или показать ее примитивность.

Заметьте, что если G не является примитивной, для каждого $\alpha \in \Omega$ существует такой $\beta \neq \alpha$, что G имеет нетривиальный блок, содержащий $\{\alpha, \beta\}$. И, следовательно, достаточно решить эффективно следующую задачу.

MINBLOCK: по входным $\{\alpha, \beta\} \subseteq \Omega$ найти минимальный блок, содержащий α и β.

Определить неориентированный граф $X = (V, E)$ такое, что $V = \Omega$ и $E = \{(\alpha, \beta)\}^G = \{(\alpha^g, \beta^g) \mid g \in G\}$.

Утверждение 3.82. *Связная компонента C, содержащая α, является минимальным блоком.*

Доказательство. Обратите внимание, что $G \leqslant Aut(X)$, а также G транзитивна на Ω. Следовательно, связные компоненты должны двигаться в целом и, таким образом, они являются блоками; в частности, C является блоком.

Предположим, что C не является минимальным, пусть $C_1 \subsetneq C$ будет блоком. Поскольку ограничение строгое, существует ребро $(\gamma, \delta) = (\alpha^g, \beta^g)$ такой, что $\gamma \in C_1$ и $\delta \in C \setminus C_1$. Теперь $\gamma \in C_1^g \cap C_1$, но $\delta \in C_1^g \setminus C_1$, что противоречит тому, что C_1 является блоком. Следовательно, C должен быть минимальным блоком, содержащим α и β. □

Теперь можно перебрать все β для данного α, чтобы найти решение нетривиальной задачи о блоках.

Задача принадлежности

Теперь решим задачу принадлежности. По входной группе $\langle A \rangle = G \leqslant Sym(\Omega)$ и $g \in Sym(\Omega)$ надо определить, верно ли утверждение $g \in G$.

Эта проблема явно сводится к проблеме вычисления порядка группы, заданной набором порождающих (для проверки g можно добавить g в набор порождающих и проверить,

изменился ли порядок). Мы попробуем использовать подход «разделяй и властвуй» для решения этой задачи.

Нам пригодится формула $|G| = |G_\alpha||\alpha^G|$. Мы знаем, как эффективно вычислять α^G, и следовательно, можем свести задачу к меньшей подгруппе G_α. Осталось найти небольшой набор порождающих для G_α.

Теорема 3.83 (Schreier). *Пусть $\langle A\rangle = G$ и $H \leqslant G$. Пусть R будет множеством различные представители смежных классов H, то есть*

$$G = \bigsqcup_{r\in R} Hr$$

затем

$$B = \{r_1 a r_2^{-1} \mid a \in A; r_1, r_2 \in R\} \cap H$$

генерирует H

Доказательство. Для любых заданных $r_1 \in R$ и $a \in A$ существует единственный r_2 такой, что $r_1 a r_2^{-1} \in H$ (потому что $Hr_1a = Hr_2$).

$$\begin{aligned} RA &\subseteq BR(r_1 a = (r_1 a r_2^{-1})r_2) \\ &\subseteq \langle B\rangle R \\ \Rightarrow RAA &\subseteq \langle B\rangle RA \\ &\subseteq \langle B\rangle\langle B\rangle R = \langle B\rangle R \\ \Rightarrow \forall t \geq 0, RA^t &\subseteq \langle B\rangle R \\ \Rightarrow G &= \langle B\rangle R \end{aligned}$$

Теперь, так как G можно разбить на смежные классы H и есть различные представители каждого смежного класса в R, если только не получилось, что $\langle B\rangle = H$, $\langle B\rangle R$ не могут покрыть всю группу. Следовательно, $\langle B\rangle = H$. □

Затем можно посмотреть на $H = G_\alpha$, представителей смежных классов имеют вид $g_\beta : \alpha \mapsto \beta$ и могут быть найдены вычислением орбиты. Но это все не дает желаемое решение, так как мощность набора порождающих быстро растет ($|B| = |R||H|$), и сложность в итоге будет экспоненциальной по n. Но существует более оптимальный алгоритм.

Идея состоит в следующем: для любых двух элементов π и ψ в B, если $1^\pi = 1_\psi$ (и π, и ψ отображают 1 в один и тот же элемент), заменим $\{\pi, \psi\}$ на $\{\pi, \pi^{-1}\psi\}$. Таким образом мы получим два элемента, отображающие 1 в разные элементы. Повторяя этот процесс замены, мы можем гарантировать, что элементы B никогда не больше n^2.

Для данной подгруппы G (заданной малым порождающим множеством, скажем, $G = \langle A\rangle$) группы перестановок S_n и перестановки $g \in S_n$, проблема принадлежности состоит в том, чтобы решить, истинно ли утверждение $g \in G$. Если ответ «да», мы также захотели бы представить g как произведение порождающих. Насколько длинным было бы

представление $g = a_{i_1}a_{i_2}\dots a_{i_N}$ как слова от образующих в худшем случае? Мы можем убедиться, что $N \leqslant |G| - 1$: рассмотрим произведения

$$s_1 = a_{i_1}, s_2 = a_{i_1}a_{i_2}, \dots, s_N = a_{i_1}a_{i_2}\dots a_{i_N}.$$

Если $N > |G|-1$, то либо $s_j = e$, в этом случае мы можем написать $g = a_{i_{j+1}}\dots a_{i_N}$, либо $s_j = s_k$ для некоторых j и k, и в этом случае мы можем написать $g = a_{i_1}\dots a_{i_j}a_{i_{k+1}}\dots a_{i_N}$, поэтому в любом случае длину представления можно уменьшить.

Тем не менее N может быть очень большим. Например, пусть $n = p_1 + p_2 + \dots + p_m$, где p_i являются различными простыми числами, тогда рассмотрим циклическую группу, порождаемую

$$a = (1, 2 \dots, p_1)(p_1 + 1 \dots, p_1 + p_2)(\dots) \quad \dots \quad (\dots) = C_1C_2\dots C_m.$$

Здесь C_i — непересекающиеся циклы с длинами p_i, поэтому порядок a равен $M = p_1p_2\dots p_m$. Группа является $G = \langle\{a\}\rangle = \{1, a, a_2, \dots, a_{M-1}\}$, в ней a_{M-1} можно записать только как произведение $M-1$ раз a. Итак, здесь $N = p_1p_2\dots p_m - 1 \geqslant 2^m - 1$, и по теореме о простых числах мы знаем, что $m = \Omega(\frac{p_m}{\ln p_m})$. Далее, поскольку

$$n = p_1 + p_2 + \dots + p_m \leqslant 1 + 2 + 3 + \dots + p_m \leqslant {p_m}^2,$$

получаем $p_m \geqslant \sqrt{n}$, поэтому

$$N \geqslant 2^{\frac{c\sqrt{n}}{\log n}} - 1,$$

то есть не полиномиальна по n.

Таким образом, мы не можем найти за полиномиальное время представление элемента как произведение порождающих, представление должно быть схемой на A. В приведенном выше примере a_{M-1} может быть вычислено за полиномиальное время с помощью быстрого умножения. Наш алгоритм даст аналогичное хорошее представление.

Проверка принадлежности сводится к задаче *вычисления порядка*, как $g \in G \iff |G| = |G \cup g|$. Идея решения последней задачи, как мы видели, состоит в том, чтобы найти башню подгрупп

$$G = G^{(0)} \geqslant G^{(1)} \geqslant \dots \geqslant G^{(n-1)} = \{1\}$$

так, чтобы легко было вычислить индекс $[G^{i-1} : G^i]$ для каждого i. Тогда можно воспользоваться формулой $|G| = \prod_{i=1}^{n-1}[G^{i-1} : G^i]$.

Рассмотрим поточечные стабилизаторы и взглянем на башню подгрупп

$$G^{(i)} = \{g \in G \mid j^g = j \text{ для всех } j, 1 \leqslant j \leqslant i\}$$

Здесь $[G^{i-1} : G^i] \leqslant n - i$ для каждого i (поскольку при фиксированном $i-1$ остается $n - i$ выборов для индекса i), и наш алгоритм будет вычислять его путем нахождения системы представителей смежных классов для него. Алгоритм также даст нам новый порождающий набор для G; его можно использовать, чтобы не пересчитывать набор порождающих при повторных проверках принадлежности.

Как найти представителей смежных классов по $G^{(1)}$ в G? Пусть X_1 — орбита 1 под действием G и для любого $k \in X_1$ пусть g_{1k} — элемент, который отображает на него 1 (то есть $1_{g_{1k}} = k$). Тогда

$$G = \cup_{k \in X_1} G^{(1)} g_{1k}$$

Позже мы увидим, как найти небольшой порождающий для $G^{(1)}$. Предполагая, что мы можем сделать это, мы можем аналогичным образом найти X_2, орбиту 2 под действием $G^{(1)}$, найти g_{2k} такие, что $k = 2^{g_{2k}}$, и аналогично найти представителей смежных классов $G^{(2)}$ в $G^{(1)}$.

В общем, для каждого i от 1 до n мы находим X_i — орбиту i под действием $G^{(i-1)}$, а также для каждого элемента $k \in X_i$ — элемент g_{ik} такой, что $i^{g_{ik}} = k$. Затем $G^{(i-1)} = \cup_{k \in X} G^{(i)} g_{ik}$

Взяв объединение наборов представителей смежных классов, мы получаем порождающий набор для G.

Определение 3.84. Пусть Ω — упорядоченное множество $\{\omega_1, \omega_2, \ldots, \omega_n\}$, а $G \leqslant Sym(\Omega)$. *Сильный порождающий набор* относительно этого упорядочения — объединение множества представителей правых смежных классов для $G^{(i)}$ в $G^{(i-1)}$ для $0 \leqslant i \leqslant n-1$.

Для данного $g \in G$ положим $k_1 = 1^g$. Если $k \notin X_1$, мы немедленно делаем вывод, что $g \notin G$. Иначе пусть $g^{(1)} = g g_{1k}^{-1}$. По построению $g^{(1)} \in G^{(1)}$. Затем, допустив, что $k_2 = 2^{g^{(1)}}$ при условии $k_2 \notin X_2$, мы можем остановиться и выдать $g \notin G$; иначе положим $g^{(2)} = g g_{1k_1}^{-1} g_{2k_2}^{-1}$ и продолжим аналогичным образом. Другими словами, для каждого i от 1 до n мы положим $k_i = i_{G^{(i-1)}}$, прерваться в случае если $k_i \notin X_i$ и установить $g^{(i)} = g^{(i-1)} g_{ik_i}$ в противном случае, тогда $g^{(i)}$ гарантированно находится в $G^{(i)}$. Если в любой момент

$$g^{(r)} = g g_{1k_1}^{-1} g_{2k_2}^{-1} \ldots g_{rk_r}^{-1} = 1,$$

тогда мы имеем представление g как произведение элементов G. Если $g \in G$, то такое представление гарантировано будет найдено за конечное время, ибо если мы не прервались к моменту, когда i принимает значение n, то $g^{(n-1)} \in G^{(n-1)} = \{1\}$.

Лемма Шрайера. Наш алгоритм выше для нахождения сильного порождающего множества зависит от возможности найти порождающий набор для $G^{(i)}$, если известно множество порождающих для $G^{(i-1)}$. Это возможно благодаря лемме Шрайера.

Для каждого r_1 и a существует единственный r_2 такой, что $r_1 a r_2^{-1} = h$, для любого h в H, поэтому мы знаем, что $|B| \leqslant |R||A|$. Однако одного этого недостаточно, чтобы построить полиномиальный алгоритм — размер множества порождающих может увеличиваться на $\Theta(n)$ каждый раз. Чтобы избежать этого, нам нужно убедиться, что мы можем сохранить маленькое порождающее множество.

Теорема 3.85. *Для любой группы $G \leqslant S_n$ мы можем найти порождающий ее набор элементов размером не более n^2.*

Идея состоит в том, чтобы удалить коллизии. Предположим, π и ψ — два элемента порождающего множества, которые отображают 1 в один и тот же элемент, то есть $1^\pi = 1^\psi$. Мы заменим $\{\pi, \psi\}$ на $\{\pi, \psi\pi^{-1}\}$. Это гарантирует, что один из элементов фиксирует 1, и учитывается на последующих уровнях.

Лучший способ смотреть на это — рассмотреть таблицу, в которой строка i представляет $G^{(i)}$. Начнем с нижней строки, представляющей $G^{(0)} = G$. Для каждого элемента $\pi \in B$, если $1\pi = 1$, переместим его в строку $G^{(1)}$, иначе поместим его в столбец 1^π. Всякий раз, когда возникают коллизии, проделаем указанное выше сокращение и отправим элемент, который сохраняет 1, на следующую строчку. После завершения строки 1 больше нет столкновений перейдем к следующей строке и повторимэтот процесс.

Иными словами, мы проделываем следующий алгоритм.

procedure REDUCE(B)
  $B_0 = B$
  $A[][]$, пустой массив $n \times n$
  for $i = 0$to $n - 1$ **do**
    for $\psi \in B_i$ **do**
      $j = i^\psi$
      if $A[i][j] = \varnothing$ **then**
        if $j = i$ **then**
          $B_{i+1} = B_{i+1} \cup \{\psi\}$
        else
          $A[i][j] = \psi$
        end if
      else
        $\pi = A[i][j]$
        $B_{i+1} = B_{i+1} \cup \{\pi \cdot \psi^{-1}\}$
      end if
    end for
  end for
  выбросить все тривиальные элементы из $\cup B_i$
  return $\cup B_i$
end procedure

procedure MEMBERSHIP($g \in Sym_n$, i, A — множество порождающих для $G^{(i)} \leqslant Sym_n$)
  if $g = id$ **then**
    return true
  end if
  $X_i = (i+1)^{Gi}$
  вычислить множество R представителей различных классов смежности $G^{(i+1)}$ в $G^{(i)}$

```
    k = (i+1)^g
    if k ∉ X_i then
        return false
    else
        вычислить множество порождающих B для G^(i+1) по лемме Шрайера
        Reduce(B)
        взять g_ik ∈ R, представитель класса смежности по G^(i)
        g' = g · g_ik^(-1)
        return Membership(g', B, i+1)
    end if
end procedure
```

Фильтр Джеррама

Мы уже установили, что можно построить множество порождающих подгруппы $G \subset Sym_n$ мощности не более n^2. Тем не менее, есть гораздо более сильные результаты.

Теорема 3.86 (Neumann). *При $n > 3$ каждая подгруппа в Sym_n может быть порождена множеством из не более $\lfloor \frac{n}{2} \rfloor$ элементов.*

Оценка $\frac{n}{2}$ в приведенной выше теореме точна оценку. Рассмотрим

$$\Omega = \{a_1, a_2, \cdots, a_m, b_1, \cdots, b_m\}$$

и пусть G — группа, порождаемая транспозициями $\{(a_i, b_i) \mid 1 \leqslant i \leqslant m\}$. Этот порождающий набор имеет размер $\frac{n}{2}$; поскольку G изоморфна $\mathbb{F}_2^m$, это соответствует нижней границе числа прождающих.

Однако для алгоритмического вычисления порождающих нам пригодится немного более слабая оценка.

Теорема 3.87 (Jerram). *Любая подгруппа G из Sym_n имеет порождающий набор размером $(n-1)$. Более того, по данной группе $G = \langle A \rangle$ мы можем вычислить множество порождающих мощности $(n-1)$ за полиномиальное время.*

Доказательство. Определим граф $X_A = \{[n], E_A\}$ следующим образом: для каждого $g \in A$, пусть $i_g \in [n]$ будет наименьшей точкой, сдвинутой на g. Добавим ребро $e^g = \{i^g, i_g^g\}$ до E_A. Теперь у нас есть неориентированный граф на n вершинах. Мы покажем, что можно за полиномиальное число итераций изменить A таким образом, чтобы получить в итоге ациклический граф $X_{A'}$ (и, конечно, $G = \langle A' \rangle$), откуда будет следовать утверждение теоремы.

Для любой перестановки $T \in Sym_n$ определим вес графа X_T (обозначается через $w(T)$) как $\sum_{g \in T} i^g$. Очевидно, что $w(T) \leqslant |T| n$. Алгоритм, который мы предъявим, будет

онлайн-алгоритмом, он работает со множеством A перестановок, для которого X_A является ациклическим и по мере включения нового элемента g он изменяет A для размещения g и поддержания ацикличности соответствующего графа.

Предположим, что у нас уже есть множество S, для которого X_S ацикличен; добавим g. Если даже при добавлении e_g граф остается ациклическим, добавим e_g в граф, а g — к множеству S.

В противном случае в $X_{S\cup\{g\}}$ существует единственный цикл, содержащий ребро e_g. Теперь каждое ребро цикла помечено элементом S (для каждого ребра из-за неопределенности направления это или g_k, или g_k^{-1}, обычно обозначается как g_k^{ϵ}, где ϵ может быть или 1, или -1). Пусть i — наименьшая точка в этом цикле. Так как i является частью цикла, одно из двух ребер, смежных с i в цикле, должно быть помечено как g_0, что $i = i_{g_0}$. Обойдем в этом направлении от i вокруг полного цикла до i; обходу естественно соответствует произведение $g_0 g_1^{\epsilon_1} g_2^{\epsilon_2} \cdots g_k^{\epsilon_k} = h$. Если h не тривиален, заменим g_0 на h во множестве S(если же $h = e$, в котором отклоним его).

Такое преобразование не меняет группу, порожденную имеющимся множеством, но с другой стороны, выбор h требует, чтобы $j^h = h$ для всех $1 \leqslant j \leqslant i$ и, следовательно, становится фиксированным иногда за i. И так как мы выбрали i как наименьший элемент цикла, вес графа увеличивается, когда мы заменим g_0 на h. Размер нашего множества остался прежним, но в силу выбора h вес вырос. Следовательно, за полиномиальное количество шагов мы точно достигнем верхней границы и, следовательно, в конечном итоге получим ациклический граф (некоторые h могут стать тривиальными в процессе).

Повторяя процесс со всеми элементами A, мы получим множество порождающих A', для которых $X_{A'}$ ациклический, и, следовательно, $|A| \leqslant n - 1$. □

В оригинальном алгоритме Reduce, который порождающий набор мощности не более n^2, мы разрешали коллизии, которые в сеттинге Джеррама соответствуют 2-циклам в X_A. Следовательно, фильтр Джеррама также можно считать обощением оригинального Reduce, но рассматривающий весь граф, а не только 2-циклы.

Задача graph-iso и теоретико-групповые алгоритмы

Можно показать, что задача изоморфизма графов является в общем случае **NP**-полной. Вряд ли существует эффективный алгоритм для graph-iso, но некоторые частные случаи этого могут быть решены за полиномиальное время. Мы рассмотрим два таких случая [достаточно широких] — цветные графы с ограниченными цветовыми классами и кубическим графом.

Ограниченная кратность цвета graph-iso: $BCGI_b$. Определим $BCGI_b$ как множество неориентированных раскрашенных графов таких, что в каждый цвет раскрашено не более b вершин. Мы ищем полиномиальный алгоритм для задачи изморфизма в

$BCGI_b$: для данных графов $X_1, X_2 \in BCGI_b$ определить, являются ли они изоморфными. Без ограничения общности можно считать, что X_1 и X_2 связны (в противном случае рассмотрим их дополнения). Рассмотрим граф $(V, E) = X = X_1 \sqcup X_2$, в таком случае проверка изоморфности сводится к вычислению группы автоморфизмов X — если известна группа автоморфизмов графа X, где каждый элемент задан как перестановка вершин графа, то достаточно вычислить орбиту одной вершины графа и проверить, непусто ли ее пересечение с обоими компонентами связности. Дополнительная цветовая структура накладывает ограничение на то, что $Aut(X) \leqslant \bigotimes Sym(C_i)$, где C_i — множество вершин, раскрашенных в цвет i. Идея состоит в том, что мы используем сокращение SET-STAB в этом случае и покажем, что мы можем эффективно вычислить необходимый стабилизатор.

Пусть $E_i = E \cap \binom{C_i}{2}$ для каждого цвета i (набор одноцветных ребер) и $E_{ij} = E \cap (C_i \times C_j)$ для пар цветов $i \neq j$ (разноцветные ребра). Ясно, что любой $\pi \in \bigotimes Sym(C_i)$ является сохраняющим цвет автоморфизмом титтк $E_i^\pi = E_i$ и $E_{ij}^\pi = {}_{ij}$. Таким образом, теперь мы получили его в форму SET-STAB.

Пусть $G = \bigotimes SymC_i = \langle A \rangle$, выбрав транспозиции как порождающие. Теперь определим

$$\Omega = (\cup C_i) \cup \left(\cup 2_{\binom{C_i}{2}}\right) \cup \left(\cup 2_{C_i \times C_j}\right),$$

в котором подмножества E_i и E_{ij} являются точками. Из ограничения на мощности классов C_i получаем

$$|\Omega| \leqslant n + r2^{\binom{b}{2}} + \binom{r}{2} 2^{b^2},$$

где r — количество цветов, и так как b — константа, а $r \leqslant n$, то $|\Omega| = poly(n)$.

G может быть естественным образом расширена для действия на Ω (расширение в наиболее очевидном смысле, если $i \mapsto i'$, $j \mapsto j'$, то $E_{ij} \mapsto E_{i'j'}$). И теперь $Aut(X)$ — подгруппа, которая поточечно стабилизирует кластеры E_i и E_{ij}, это можно сделать с помощью нашего сильного множества порождающих. Нетрудно видеть, что время выполнения равно $poly(|\Omega|)$ и, следовательно, полиномиально по $|X|$.

Кубические графы. Другим ограничением, которое мы можем наложить на графы, является степень каждой вершины; предположим, нам дано, что степень каждой вершины ограничена константой d, можем ли мы решить GRAPH-ISO за полиномиальное время?

Случаи при $d \leqslant 2$ несложны. Первый интересный случай, когда $d = 3$, соответствующая задача называется *изоморфизм кубического графа*. Нам даны два графа X_1 и X_2, степени вершин которых ограничены 3, мы должны проверить их на изоморфность. Во-первых, обратимвнимание, что мы можем предположить, что оба они связны, так как в противном случае мы можем ограничиться связные компоненты и работать со всеми

возможными парами компонент связности первого графа и второго. Проверка изоморфности X_1 и X_2 эквивалентна вычислению группы автоморфизмов $X_1 \sqcup X_2$.

Предположим, что мы уже выделили ребро, и нас интересуют только автоморфизмы, которые сохраняют это ребро. Оказывается, достаточно вычислить именно такие группы.

Зафиксируем некоторое ребро $e_{uv} \in X_1$. Для каждого $e_{pq} \in X_2$ добавим новое ребро, которое «соединяет» e_{uv} и e_{pq} (добавим середины как другие две вершины и соединим их). Таким образом, если изоморфизм поменял местами e_{uv} и e_{pq}, он сохраняет e. А также так как мы делаем это по всем ребрам e_{pq}, достаточно научиться эффективно вычислять группу $Aut_e(X)$ — группу автоморфизмов графа, сохраняющих ребро e.

Алгоритм работает путем построения групп автоморфизмов с помощью аппроксимаций $Aut_e(X_r)$, где каждый X_r является подграфом X. Для каждого r определим X_r как состоящий из всех вершин и ребер, которые появляются в путях длиной не более r, проходящих через . Это слои X относительно расстояния от e.

И ясно, что, поскольку e является выделенным ребром, любой $\pi \in Aut_e(X)$ должен сохранять эти слои.

Теорема 3.88 (Tutte). *Если X — связный кубический граф, а e — любое ребро в X, тогда $Aut_e(X)$ является 2-группой.*

Доказательство. Основная идея заключается в том, что группы автоморфизмов являются последовательными аппроксимацями и каждое разложение проходит посредством 2-группы.

Доказательство проведем индукцией по i. Предположим, что $Aut_e(X_i)$ — 2-группа. Поскольку любой автоморфизм, сохраняющий e, должен сохранять слои, у нас есть естественный гомоморфизм $\phi : Aut_e(X_{i+1}) \longrightarrow Aut_e(X_i)$, который является функцией проекции ($Aut_e(X_{i+1})$ сохраняет слои до X_{i+1}).

Следовательно, $|Aut_e(X_{i+1})| = |\phi(Aut_e(X_{i+1}))| \cdot |\ker\phi|$ и так как $\phi(Aut_e(X_{i+1}))$ является подгруппой в $Aut_e(X_i)$, это 2-группа. Следовательно, остается только проверить, что $\ker\phi$ также является 2-группой. Мы заинтересованы в подсчете $\pi \in Aut_e(X_{i+1})$, который фиксирует X_i поточечно. Если $V = V(X_{i+1}) \setminus V(X_i)$, то любой нетривиальный π должен что-то делать только на V. Однако граф X кубический и, следовательно, любой $u \in V(X_i)$ может быть связан не более чем с 2 вершинами в V и, следовательно, любой автоморфизм X_{i+1}, который фиксирует X_i, может поменять местами двух соседей u. Таким образом, любой $\pi \in \ker\phi$ удовлетворяет условию $\pi_2 = id$ и, следовательно, $\ker\phi$ также является 2-группой. □

Мы хотим вычислить $Aut_e(X)$, и мы сделаем это, используя башню, компоненты которой индуцированы разными слоями X_r. Общая философия такова: если $\phi : G \longrightarrow H$ — гомоморфизм, и у нас есть порождающий набор для $\ker\phi = \{k_1, k_2, \cdots, k_n\}$ и $\phi(H) = \{\phi(g_1), \phi(g_2), \cdots, \phi(g_m)\}$, затем мы можем эффективно найти порождающее множество для G.

Мы будем использовать гомоморфизмы $\phi : Aut_e(X_{r+1}) \longrightarrow Aut_e(X_r)$, чтобы подняться по этажам башни и, наконец, добраться до порождающего множества для $Aut_e(X)$. Нам нужно найти

(1) порождающее множество для $\ker \phi$,
(2) порождающее множество для $\phi(Aut_e(X_{r-1}))$

Алгоритм для поиска порождающего множества для $\ker \phi$ нам дает доказательство доказательства теоремы Татта. Заметим, что каждая вершина $v \in X_r$ смежна с не более чем двумя вершинами $X_{r+1} \setminus X_r$, и автоморфизм может поменять местами эти два соседа v. Это будет невозможно именно тогда, когда окрестности двух вершин не совпадают. Таким образом, если $w_1, w_2 \in X_{r+1} \setminus X_r$ являются соседями $v \in X_r$, транспозиция (w_1, w_2) будет действительным автоморфизмом, фиксирующим X_r титтк $\Gamma(w_1) = \Gamma(w_2)$. И это может быть легко проверено. Таким образом, за линейное время мы можем получить множество порождающих (набор непересекающихся транспозиций) для $\ker \phi$.

Гораздо сложнее найти порождающий набор для $\phi(Aut_e(X_{r+1}))$. Будем в дальнейшем считать $V_r = X_{r+1} \setminus X_r$. Заметим, что каждая вершина $v \in V_r$ связана с 1 или 2 или 3 соседями X_r. Пусть A будет совокупностью всех подмножеств X_r размером 1 или 2 или 3. Тогда у нас есть следующее отображение окрестностей $\Gamma_r : V_r \to A$, которая сопоставляет каждой вершине множество ее соседей в графе X_{r+1}. Обратите внимание, что для каждого автоморфизма $\sigma \in Aut_e(X_{r+1})$ имеет место $\Gamma_r(\sigma(v)) = \sigma(\Gamma_r(v))$, и в ядре получилось бы $\Gamma_r(v) = \sigma(\Gamma_r(v))$. Назовем две вершины $v_1, v_2 \in V_r$ *близнецами*, если $\Gamma_r(v_1) = \Gamma_r(v_2)$.

Следовательно, любой $\sigma \in \phi(Aut_e(X_{r+1}))$ должен стабилизировать набор предков с только одним потомком.

$$A_1 = \{a \in A \mid a = \Gamma_r(v) \text{для некоторого единственного} v \in V_r\}$$

σ также должен стабилизировать множество предков близнецов,

$$A_2 = \{a \in A \mid a = \Gamma_r(v_1) = \Gamma_r(v_2), v_i \neq v_j\}$$

И кроме достижения вершин V_r следующий слой также индуцирует ребра между вершинами X_r, и любой автоморфизм должен сохранять их.

$$A_3 = \{\{w_1, w_2\} \in A \mid (w_1, w_2) \in X_{r+1}\}$$

На самом деле, все этого достаточно, чтобы убедиться в том, что $\sigma \in Aut_e(X_r)$ лежит в $\phi(Aut_e(X_{r+1}))$.

Утверждение 3.89. *Образ $\phi(Aut_e(X_{r+1}))$ — как раз в точности те автоморфизмы $\sigma \in Aut_e(X_r)$, которые стабилизируют множества A_1, A_2 и A_3.*

Доказательство. Нам нужно показать, что если σ стабилизирует три множества, то мы можем продолжить его до автоморфизма X_{r+1}. Продолжение построено следующим образом:

- Для каждого отдельного потомка v, для которого $\Gamma_r(v) \in A_1$, так как $\sigma(\Gamma_r(v)) \in A_1$, отобразим v в единственного потомка $\Sigma(\Gamma_r(v))$.
- Для каждой пары близнецов v_1, v_2, для которых $\Gamma_r(v_1) = \Gamma_r(v_2) \in A_2$, так как $\sigma(\Gamma_r(v)) \in A_2$, сопоставим $\{v_1, v_2\}$ сыновей $\sigma(\Gamma_r(v_1))$ в любом порядке.

Построенное отображение уважает ребра, соединяющие X_r и V_r. И поскольку он также стабилизирует A_3, то σ также уважает ребра между вершинами X_r, которые были дополнительно построены. Следовательно, σ действительно может быть продолжено до автоморфизма X_{r+1}. □

Часть 4

Графы и алгоритмы

Краткое введение в теорию графов — часть матфаковского курса дискретной математики, повторить ее можно по книжкам [**pras**, 35, 4, 15, 26, 29].

Договоримся о способах хранения графов:

- *список ребер*: $|V|$, число вершин, и $|E|$ упорядоченных пар (x_i, y_i), каждая из которых кодирует ребро.
- *списки смежности*: $|V|$, число вершин, и $A_1, \dots A_{|V|}$ — списки смежности, то есть $A_i = j \in [1; |V|] \mid (ij) \in E$.
- *матрица смежности*: $A \in \mathrm{Mat}_{|V|\times|V|}(\mathbb{Q})$

  для неориентированного задается так:

  $$A_{ij} = \begin{cases} 1, & (ij) \in E \\ 0, & (ij) \notin E \end{cases}$$

  Для ориентированного задается так:

  $$A_{ij} = \begin{cases} 1, & (ij) \in E \\ -1, & (ji) \in E \\ 0, & \text{иначе} \end{cases}$$

Матричный вид удобен тем, что если A — матрица смежности неориентированного графа, то $(A^k)_{ij}$ есть число путей длины k из вершины i в вершину j.

Теорема 4.1. *Граф G имеет эйлеров путь тогда и только тогда, когда все вершины, кроме, возможно, не более двух, имеют четную степень.*

Доказательство. См. книгу [35]. □

На основе доказательства этой теоремы можно построить алгоритм, находящий эйлеров цикл в эйлеровом графе за линейное время. На вход мы будем подавать граф в виде списка смежности $\{A_v \mid v \in V\}$. Нам понадобится двусвязный список.

```
1: procedure Euler(G, s)
2:     L ← {s}, K ← ∅
3:     while E ≠ ∅ do
4:         взять первую v из A_s, E ← E \ {sv}, добавить v в L
5:         while v ≠ s do
6:             взять первую u из A_v, E ← E \ {vu}, добавить u в L
7:         end while
8:         for v ∈ L do
9:             K_v = Euler(G, v)
10:            заменить v на K_v в L
11:        end for
12:    end while
13:    return K
14: end procedure
```

Нетрудно видеть, что этот алгоритм корректен: G без полученного после цикла **while**$(v \neq s)$ ... **end while** распадается на конечное число компонент связности, если каждый K_v является эйлеровым циклом в этих компонентах связности, то подстановка K_v на место L дает эйлеров обход большого графа. Нетрудно также видеть, что время работы алгоритма

Depth-first search

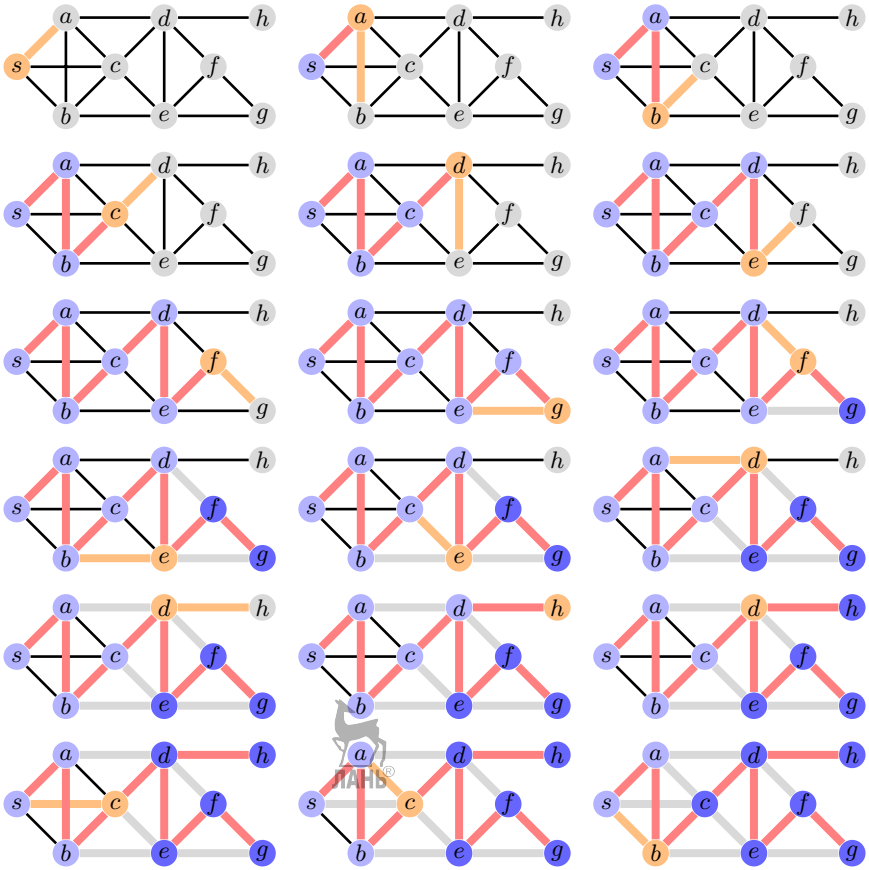
Очень часто в задачах на графы (определение связности, наличия пути, цикла итд.) необходимо «обойти» граф. Один из способов это сделать — обойти его в глубину (depth-first search): из уже достигнутой вершины v_1 пройти в смежную и еще не посещенную вершину v_2, из нее — в другую смежную и еще не посещенную вершину v_3 и так далее; если вдруг все смежные с v_i вершины посещены, вернуться к v_{i-1} и пройти в смежную и еще не посещенную вершину, отличную от v_i.

На вход подается граф в виде списка смежности. В реализации ниже мы строим дерево предшествования p (формально $p : V \to V$ сопоставляет вершине v вершину $p(v)$, из которой v была впервые посещена) и $nr : V \to \mathbb{N}$ — функцию, сопоставляющую вершине v ее порядковый номер при посещении (если $nr(v) = k$, то до v было посещено $k-1$ вершина). В принципе нам необязательно вычислять $p(\cdot)$ и $nr(\cdot)$ — это полезные мелочи, которые пригодятся нам для приложений DFS.

```
1:  procedure DFS(G = (V, E), s)
2:      for v ∈ V do
3:          nr(v) ← 0, p(v) ← 0
4:      end for
5:      for e ∈ E do
6:          u(e) ← false
7:      end for
8:      i ← 1, v ← s, nr(s) ← 1
9:      while v ≠ s или ∃w ∈ A_s, u(sw) = false do
10:         while ∃w ∈ A_v : u(vw) = false do
11:             выбрать w ∈ A_v : u(vw) = false
12:             u(vw) = true
13:             if nr(w) = 0 then
14:                 p(w) ← v, i ← i + 1, nr(w) = i, v ← w
15:             end if
16:         end while
17:         v ← p(v)
18:     end while
19:     return p, nr
20: end procedure
```

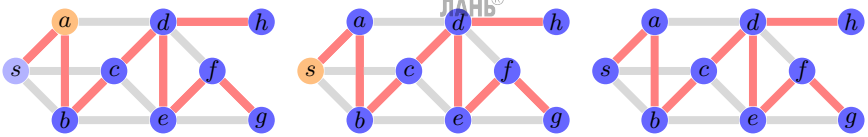
Мы полагаем, что V линейно упорядочено, то есть $V = \{v_1, \dots v_{|V|}\}$, и эта нумерация вершин индуцируется на все списки смежности, то есть для любой вершины v получаем $A_v = \{v_{i_1}, \dots v_{i_{\deg v}}\}$, где $1 \leqslant i_1 < \dots < i_{\deg v} \leqslant |V|$. Вершины в цикле **while**($\exists w \in A_v$: $u(vw) =$ **false**)...**end while** выбираются по порядку.

Пример 4.2. Продемонстрируем работу DFS на графе, изображенном на картинке 1. Здесь порядок вершин такой: $s, a, b, c, d, e, f, g, h$.



[Окончание — на следующей странице]. Рыжим цветом отмечена сканируемая вершина и проверяемое ребро. Посещенные вершины v, то есть те, для которых $nr(v) \neq 0$,

отмечены светло-синим цветом. Ребра e, для которых $u(e) = \mathbf{true}$, выделены жирным, ребра вида $p(v)v$ — красные, остальные посещенные — серые. [Есть продолжение на следующей странице.] На раскадровке опущены кадры с проверкой уже посещенных ребер.



Теорема 4.3. *Каждое ребро в компоненте связности G, содержащей s, используется ровно один раз в каждом направлении при выполнении алгоритма* DFS. *В частности, алгоритм* DFS *имеет сложность $O(|E|)$ для связных графов.*

Доказательство. Можно считать, что G связен. На шаге 11 для перехода из v в w используется ребро $e = vw$ (где изначально $v = s$); если $nr(w) = 0$, то w еще не посещена и v заменяется на w. Если $nr(w) \neq 0$, то w уже посещена и ребро e используется для прохода в обратном направлении для перехода от w к v, и алгоритм приступает (если возможно) к обработке другого ребра, инцидентного с v, которое еще не использовалось. Если такого ребра нет, то есть если все ребра, инцидентные v, использовались по реброней мере один раз – ребро $p(v)v$, которое использовалось для достижения v из $p(v)$, проходится в обратном направлении, и v заменяется на $p(v)$. Таким образом, алгоритм действительно строит обход графа G.

Теперь мы покажем, что ни одно ребро не может быть использовано дважды в одном и том же направлении. Предположим обратное, тогда существует ребро $e = vw$, которое используется дважды в одном и том же направлении. Положим, что e впервые используется для прохода от v к w, тогда в таком направлении пройти можно ровно один раз — когда $u(e) = \mathbf{false}$. Так что дважды e может быть пройдено только во время выполнения шага 16, замены $w \leftarrow v = p(w)$. В таком случае все ребра, инцидентные с v, должны быть пройдены ранее [$u(e) = \mathbf{true}$]. Таким образом, из вершина была посещена хотя бы $degv + 1$ раз [выход совершается по всем ребрам, а по $vp(v)$ — дважды]; поэтому какое-то ребро uv должно использоваться дважды для прохода от u до v, что невозможно, как мы показали чуть выше.

Из предыдущих соображений следует, что алгоритм завершает работу за конечное время. Более того, отсюда следует, что сложность DFS равна $O(|E|)$.

Теперь докажем, что каждое ребро G используется в обоих возможных направлениях. Пусть S — множество вершин v, для которых каждое ребро, инцидентное с v, используется в обоих направлениях. Когда DFS завершается, он должен достигнуть вершины v с $p(v) = 0$, для которого нет инцидентного ребра с v, на котором $u(\cdot)$ принимает значение **false**. Это может произойти только для $v = s$; более того, все ребра, инцидентные с s,

должны были использоваться для перемещения от s к другой вершине. Но тогда все эти ребра также должны были быть использованы для достижения s, поскольку ни один из них не использовался дважды в одном и том же направлении. Таким образом, $s \in S$, и S непусто.

Теперь покажем $S = V$. Предположим обратное. Поскольку G связный [по предположению], существуют ребра, соединяющие S с $V \setminus S$; рассмотрим первое посещенное алгоритмом DFS такое ребро $e = vw$, где $v \in S$ и $w \in V \setminus S$. Каждое ребро, соединяющее S и $V \setminus S$, используется в обоих направлениях. Как мы достигаем вершины w в первый раз, переходя по от v до w, имеем $nr(w) = 0$. Затем положим $v = p(w)$, и $u(e) = \mathbf{true}$. Теперь мы можем использовать e снова только при переходе от w к v. В этот момент все ребра, связанные с w, должны были помечены как пройденные. Поскольку каждое ребро, инцидентное с w, может использоваться не более одного раза в каждом направлении, каждое из этих ребер должно было использоваться в обоих направлениях, так что $w \in S$ противоречие. □

Нетрудно видеть, что если входной граф G связен, полученное в DFS дерево p является остовным для G. Таким образом, с помощью DFS легко проверять связность графа — достаточно проверить, все ли вершины входят в полученное с помощью DFS остовное дерево [по-английски формально не tree и не rooted tree, а arborescence — корневое дерево, в котором нет вершин, входящих в корень]. По умолчанию ориентация на DFS-дереве p мы зададим от предков к потомкам, то есть ребра будут иметь вид $p(v)v$.

Определение 4.4. *DFS-деревом* связного графа $G = (V, E)$ будем называть дерево

$$(V, \{p(v)v \mid v \in V\}),$$

где p — функция, вычисленная DFS.

В примере 4.2 ребра DFS-дерева отмечены красным цветом.

Определение 4.5. Ребра вида $p(v)v$ [ориентированные] мы будем называть *древесными ребрами*, остальные [в том числе и $vp(v)$] мы будем называть *задними ребрами*.

Если же граф G несвязен, то полученное T будет остовным лесом — каждая компонента связности T будет остовным деревом некоторой компоненты связности G.

Теперь рассмотрим версию DFS на ориентированных графах. Теперь ребро vw понимается как ориентированное $v \to w$, от v к w. Сам алгоритм выполняется как и прежде, так что единственное отличие от версии для неориентированных графов можно использовать непройденные ребра только в одном направлении.

Даже если G связен, мы можем не достичь всех вершин G, пример — некоторые вершины могут быть недостижимы из стартовой вершины. Начиная с вершины s, DFS обойдет все вершины, в которые можно попасть по ориентированному пути из s. Так же, как и в случае неориентированного графа, DFS построит остовный лес, и все оценки на сложность остаются такими же.

Поиск точек сочленения

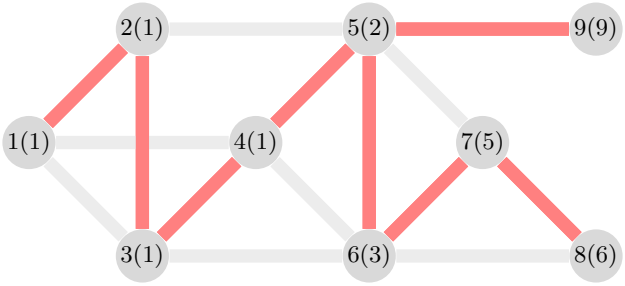
Теперь перейдем к менее очевидным применениям поиска в глубину. В этом разделе считаем графы неориентированными.

Определение 4.6. *Точка сочленения* графа G — вершина v такая, что $G \setminus v$ имеет больше компонент связности, чем G.

Определение 4.7. Связный граф, содержащий точки сочленения, будем называть *разделимым*. Максимальный индуцированный подграф G, не являющийся разделимым, называется *блоком*.

Будем считать граф $G = (V, E)$ связным. Предположим, что мы, вызвав DFS, построили остовное дерево T для G с корнем s; теперь будем использовать функции $nr(\cdot)$ и $p(\cdot)$ для определения точек сочленения G (и, следовательно, блоков). Определим функцию $L : V \to \mathbb{N}$ следующим образом: пусть B_v — множество вершин, в которые можно попасть из v по пути, где все ребра, кроме, возможно, последнего — древесные, тогда $L(v) = \min_{u \in B_v} nr(u)$.

Пример 4.8. Рассмотрим граф из примера 4.2. Вершины помечены номерами, которые им присвоены во время DFS; числа в скобках являются значениями функции L.



Лемма 4.9. *Пусть G — связный граф, s — вершина G, T — остовное дерево G, вычисленное алгоритмом* DFS. *Вершина $u \neq s$ является точкой сочленения тогда и только тогда существует древесное ребро $e = uv$, удовлетворяющее $L(v) \geqslant nr(u)$, где L — функция, определенная выше.*

Доказательство. Сначала предположим, что u — точка сочленения G. Тогда существует разбиение

$$V \setminus u = V_1 \cup \ldots \cup V_k, \quad k \geqslant 2,$$

для которого все пути, соединяющие две вершины в различных V_i, должны проходить через u. Положим, что $s \in V_1$. Пусть e будет первым пройденным DFS древесным ребром $e = uv$, для которого $v \notin V_1$; скажем, $v \in V_2$. Посокольку нет ребер, соединяющих вершину в V_2 с вершиной в $V \setminus (V_2 \cup \{u\})$, а все вершины, достижимые из v по древесным ребрам, также лежат в V_2 (и потому достигаются алгоритмом позже u), мы заключаем, что $L(v) \geqslant nr(u)$.

Теперь пусть $e = uv$ — древесное ребро, для которого $L(v) \geqslant nr(u)$. Введем S — множество вершин на пути из s в u в T (включая s, но не u), и пусть T' — поддерево T с корнем v [то есть T' состоит из потомков v]. Поскольку любая вершина в T' — потомок v, не существует ребра xy, где $x \in T'$, $y \in V \setminus S \cup T' \cup \{u\}$. Более того, нет ребер xy, где $x \in T'$, $y \in S$: такое ребро будет задним, откуда следует, что существует путь, проходящий сначала из v в x по древесным ребрам T', затем по ребр xy, откуда

$$L(v) \leqslant nr(y) < nr(u)$$

Значит, каждый путь, соединяющий вершину в T' с вершиной в S должна содержать u, так что u является точкой сочленения. □

Лемма 4.10. *В условиях прошлой леммы корень* DFS*-дерева s — точка отсечения тогда и только тогда s принадлежит хотя бы на двух древесных ребрах.*

Доказательство. Пусть s — точка сочленения, а

$$V \setminus u = V_1 \cup \ldots \cup V_k, \quad k \geqslant 2,$$

разбиение $V \setminus s$, для которых все пути, соединяющие две вершины в различных компонентах раздел должен пройти через s. Кроме того, пусть $e = sv$ будет первым древесным ребром, пройденным DFS; скажем, $v \in V_1$. Тогда никакая вершина вне V_1 не достижима из v в T, так что s должна быть инцидентна с еще одним древесным ребром.

Наоборот, пусть sv и sw — древесные ребра, и пусть T' — поддерево T с корнем v. Нетрудно видеть, что нет ребер, соединяющих вершину из T' с вершиной из $V \setminus (T' \cup \{s\})$. Поскольку по предположению множество $V \setminus (T' \cup \{s\})$ — оно содержит w — то s — точка сочленения. □

Если мы сможем во время обхода в глубину вычислить $L(\cdot)$, то сможем за $O(|E|)$, пользуясь двумя доказанными только что леммами, определить все точки сочленения. По определению функции $L(\cdot)$ получаем

$$L(v) = \min\{nr(u) \mid u = v \text{ или } vu \text{ — задний ребро в } G\},$$

если v — лист дерева T, и

$$L(v) = \min\left(\{nr(u) \mid u = v \text{ или } vu \text{ — задний ребро в } G\} \cup \{L(u) \mid vu \in T\}\right)$$

в ином случае.

В случае листа мы заменяем $L(v)$ на $\min\{L(v), nr(u)\}$, как только алгоритм использует заднее ребро vu, то есть когда $nr(u) \neq 0$ во время проверки vu. Когда алгоритм возвращается от v к $p(v)$, все задние ребра были проверены, так что $L(v)$ принял нужное значение. В противном случае мы должны заменить $L(v)$ на $\min\{L(v), L(u)\}$, как только древесное ребро vu используется во второй раз, а именно от u до v. Нетрудно убедиться [индукцией по минимальному расстоянию от листьев], что после проверки $L(u)$ принял правильное значение

```
1:  procedure Block-Cut(G = (V, E), s)
2:      for v ∈ V do
3:          nr(v) ← 0, p(v) ← 0
4:      end for
5:      for e ∈ E do
6:          u(e) ← false
7:      end for
8:      i ← 1, v ← s, nr(s) ← 1, C ← ∅, k ← 0, L(s) ← 1
9:      создать стек S, содержащий единственный элемент s
10:     while v ≠ s или ∃w ∈ A_s u(sw) = false do
11:         while ∃w ∈ A_v : u(vw) = false do
12:             выбрать w ∈ A_v : u(vw) = false
13:             u(vw) = true
14:             if nr(w) = 0 then
15:                 p(w) ← v, i ← i + 1, nr(w) = i, L(w) ← i, добавить w в S, v ← w
16:             else
17:                 L(v) ← min{L(v), nr(u)}
18:             end if
19:         end while
20:         if p(v) ≠ s then
21:             if L(v) < nr(p(v)) then
22:                 L(p(v)) ← min{L(p(v)), L(v)}
23:             else
24:                 C ← C ∪ {p(v)}, k ← k + 1
25:                 создать список B_k = S ∪ {v}, опустошить S, добавить p(v) в B_k
26:             end if
27:         else
28:             if ∃w ∈ A_s : u(sw) = false then
29:                 C ← C ∪ {s}
30:             end if
31:             создать список B_k = S ∪ {v}, опустошить S, добавить p(v) в B_k
32:         end if
```

33: $v \leftarrow p(v)$
34: **end while**
35: **return** $B_1, \ldots B_k$, C
36: **end procedure**

Теорема 4.11. *Независимо от выбора $s \in V$ множество C, полученное алгоритмом* Block-Cut, *есть множество разрезов, множества $B_1, \ldots B_k$ — множества вершин блоков. Время работы алгоритма* Block-Cut *составляет $O(|E|)$.*

Доказательство. Как и в оригинальном DFS, каждое ребро используется ровно один раз в каждом направлении, кроме того, для каждого ребра выполняется не более чем постоянное число шагов казнены. Следовательно, сложность равна $O(|E|)$. Соображения выше показывают, что $L(v)$ принимает правильное значение в тот момент, когда алгоритм заканчивает проверку v. Формальное доказательство этого факта может быть дано с использованием индукции на $nr(v)$ (в порядке убывания). Обратите внимание, что ребро vw, выбранное на шаге (12), является задним титтк когда $nr(w) \neq 0$, и что используется древесное ребро $p(v)v$ на этапе (21) для обновления значения $L(p(v))$ после определения $L(v)$ (если это обновление не является избыточным из-за условия $L(v) \geqslant nr(p(v)) \geqslant L(p(v))$).

Осталось показать, что точки сочленения и блоки определены правильно. После того, как алгоритм закончил обработку вершины v, в строчке (20) или (27) проверяется, является ли $p(v)$ точкой сочленения. Сначала предположим, что $p(v) \neq s$. Если условие в (20) выполнено, значение $L(v)$ используется для обновления $L(p(v))$; в противном случае $p(v)$ является точкой сочленения, как мы показали выше. В этом случае $p(v)$ добавляется к множеству точек среза C. Вершины в S до v (включая v) являются потомками v в T, где T — DFS-дерево. Эти вершины — не обязательно все потомки v: возможно, что некоторые потомки были вырезаны ранее; среди них могли быть точки сочленения. Тем не менее, индукцией по минимальному расстоянию до листьев можно показать, что ни один собственный потомок таких точек сочленения [то есть не совпадающий с этими точками] не содержится в S к этому момент. Следовательно, множество B_k на шаге (24) действительно является блоком.

Теперь предположим, что $p(v) = s$. Если условие на этапе (27) выполнено, s является точкой сочленения. Как показано выше, B_k является блоком G. В частности, s добавляется в C титтк не все ребра, инцидентные с s, были обработаны, когда впервые стало выполняться равенство $p(v) = s$.

В обоих случаях v заменяется в этой точке его предшественником $p(v)$. По доказанным выше леммам все точки сочленения были найдены после завершения работы алгоритма (после того, как все ребра были использованы в обоих направлениях). □

Компоненты сильной связности

По аналогии с понятием блоков в неориентированном графе множество вершин можно определить компоненты сильной связности в ориентированных графах. В этой секции графы полагаются ориентированными.

Определение 4.12. Скажем, что ориентированный граф $G = (V, E)$ *сильно связный*, если для любой пары вершин v и u есть ориентированный путь из v в u и из u в v.

Можно построить алгоритм поиска компонент сильной связности, аналогичный Block-Cut. Мы рассмотрим более простой алгоритм Kosaraju-Sharir, запускающий DFS дважды — на G и на орграфе, полученном из G обращением ориентации.

Сначала немного модицируем DFS.

```
1: procedure DFSM(G = (V, E), s)
2:     for v ∈ V do
3:         nr(v) ← 0, p(v) ← 0
4:     end for
5:     for e ∈ E do
6:         u(e) ← false
7:     end for
8:     i ← 1, v ← s, nr(s) ← 1, Nr(s) ← |V|
9:     while v ≠ s или ∃w ∈ A_s u(sw) = false do
10:        while ∃w ∈ A_v : u(vw) = false do
11:            выбрать w ∈ A_v : u(vw) = false, u(vw) = true
12:            if nr(w) = 0 then
13:                p(w) ← v, i ← i + 1, nr(w) = i, v ← w
14:            end if
15:        end while
16:        j ← j + 1, Nr(v) ← j, v ← p(v)
17:    end while
18:    return p, nr, Nr
19: end procedure
```

Здесь $Nr(\cdot)$ — нумерация вершин в порядке, в котором они были полностью просмотрены (то есть если $Nr(v) = i$, то до v ровно $i-1$ вершина была полностью просканирована DFS).

```
1: procedure Kosaraju-Sharir(G = (V, E), s)
2:     p, nr, Nr = DFSM(G = (V, E), s)
3:     H = (V*, E*) — граф, полученный из G обращением ориентации
4:     while V* ≠ ∅ do
5:         взять u ∈ V* такую, что Nr(u) максимально
6:         k ← k + 1
```

```
7:        $\widetilde{nr}, \widetilde{p} = \mathrm{DFS}(H, u)$
8:        $C_k \leftarrow \{v \in V^* \mid \widetilde{nr}(v) \neq 0\}$
9:        удалить из $H$ вершины $C_k$ и смежные с $C_k$ ребра
10:    end while
11:    return $C_1, \ldots$
12: end procedure
```

Пример 4.13. Рассмотрим граф на изображении ниже слева.

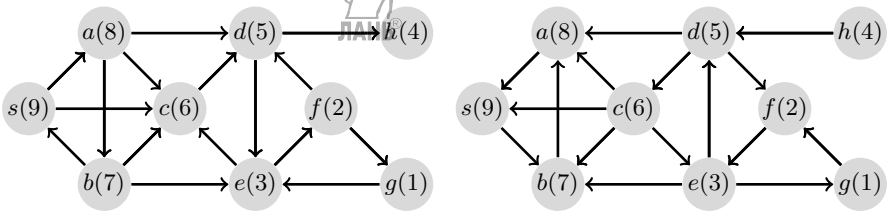


Рис. 6. Слева — значения Nr на оригинальном графе G. Справа — H, полученный из G обращением ребер

[Этот граф без ориентации был в примере 4.2.] Нетрудно убедиться, что DFS посещает вершины здесь в том же порядке, что и в примере 4.2. В скобках указаны значения Nr.

Теперь обратим ориентацию и запустим DFS. Начинаем с вершины s — у нее Nr максимальный. Иллюстрация **??** — на следующей странице.

В итоге мы получаем $C_1 = \{s, a, b\}$, $C_2 = \{c, d, e, f, g\}$, $C_3 = \{h\}$. Нетрудно видеть, что эти множества — компоненты сильной связности графа G.

Теорема 4.14. *Пусть G — орграф с корнем s. Тогда* Kosaraju-Sharir *вычисляет компоненты сильной связности и имеет сложность $O(|E|)$.*

Доказательство. Сначала мы должны показать, что множества вершин $C_1, \ldots, C_k$, которые вычисляются в каждой итерации большого цикла, есть набор компонент сильной связности G. Пусть v и w — две вершины в одной и той же компоненте сильной связности G. Тогда существуют ориентированные пути из v в w и из w в v в G, а следовательно, также и в H. Можно предположить, что v достигается раньше w во время работы DFS на H. Кроме того, пусть T_i — ориентированное дерево, содержащее v, а x — корень T_i. Так как w достижима из v в H и не рассматривалась ранее, w должен также содержаться в T_i: нетрудно видеть, что w достигается во время выполнения DFS с корнем x.

Наоборот, пусть v и w две вершины, содержащиеся в одном и том же ориентированном дереве $_i$ (на C_i). Опять же, пусть x будет корнем T_i; мы можем предположить, что $v \neq$

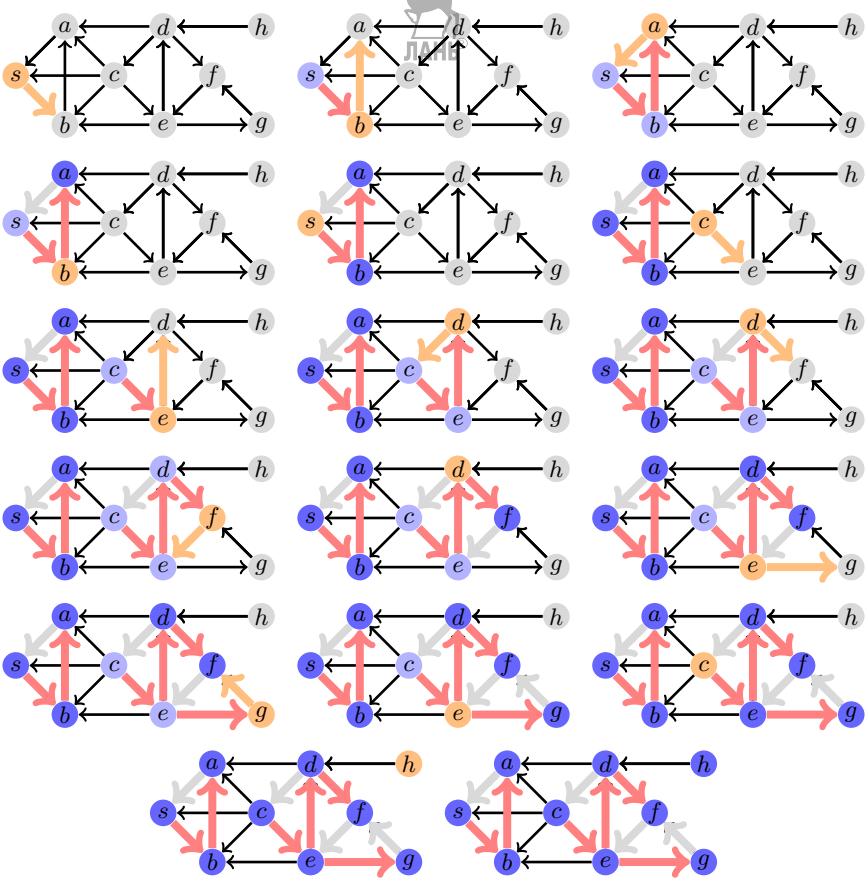


Рис. 7. Второй запуск DFS на графе в алгоритме Kosaraju-Sharir.

. Поскольку v — потомок x в T_i, существует ориентированный путь из x в v в H и, следовательно, и в G. Теперь v еще не был полностью просканирован во время DFS на H с корнем x, так что $Nr(v) < Nr(x)$. Таким образом, v была полностью просмотрена раньше, чем x. Но так как достижима из V в G, v не может быть достигнута раньше, чем x. Отсюда следует, что v должно быть потомком x в связующем дереве T для G. Следовательно, также существует направленный путь от x до v в G, и x и v содержатся

в том же сильном компоненте. Аналогично w должен содержаться в той же компоненте сильной связности.

То, что сложность Kosaraju-Sharir есть $O(|E|)$, очевидно — DFSM работает за $O(|E|)$, а DFS на каждой итерации основного цикла обрабатывает компоненту связности некоторой вершины, то есть за все итерации DFS совершает $O(|E|)$ операций. □

Алгоритм поиска компонент сильной связности имеет применение в частном случае классической задачи разрешения КНФ. Напомним, что 2-SAT — язык 2-КНФ [то есть булевых формул вида $\phi_1 \wedge \ldots \wedge \phi_n$, где $\phi_i = x_{i,1}^{\pm} \vee x_{i,2}^{\pm}$], которые выполнены хотя бы на одном наборе значений переменных.

Теорема 4.15. *Язык 2-SAT принадлежит классу* **P**.

Доказательство. Рассмотрим граф на $2n$ вершинах, соответствующих парам литералов, которые встречаются в формуле. Для любого дизъюнкта вида $a \vee b$, встречающегося в формуле, проведём два ориентированных ребра: $\neg a \to b$ и $\neg b \to a$ (опираемся на тот факт, что $a \vee b$ эквивалентно $(\neg a \to b) \wedge (\neg b \to a)$; см. рисунок 1). Смысл в том, что если b не равняется единице, то a равняется единице (и наоборот); здесь на ребро можно смотреть как на импликацию. Во-первых, в полученном графе существует путь из вершины a в вершину b тогда и только тогда, когда существует путь из $\neg b$ в $\neg a$. Действительно, если есть путь $a \to a_1 \to \ldots \to a_k \to b$, то есть и путь $\neg b \to \neg a_k \to \ldots \to \neg a_1 \to \neg a$, так как построению для любого ребра $a \to b$ есть ребро $\neg b \to \neg a$.

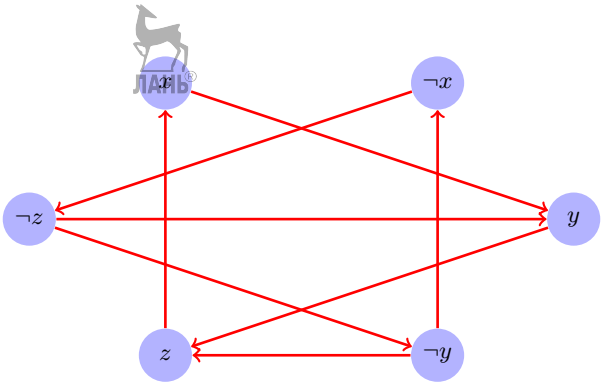


Рис. 8. Граф, построенный по формуле $(\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$

Теперь покажем, что исходная КНФ выполнима тогда и только тогда, когда не существует пути из x в $\neg x$ и не существует пути из $\neg x$ в x одновременно для некоторой переменной x. Пусть в графе есть путь из x в $\neg x$ и путь из $\neg x$ в x. Предположим, что существует выполняющий набор. Тогда, если $x = 1$ в этом наборе, то в пути $x \to x_1 \to \ldots \to x_{l-1} \to x_l \to \neg x$ все литералы должны принемать значение равное единице (в силу того, что ребру $\alpha \to \beta$ соответсвтует импликация $\alpha \to \beta$, которая эквивалентна дизъюнкту $\neg\alpha \vee \beta$, который в свою очередь равен нулю только на наборе $\alpha = 1, \beta = 0$). Но тогда $x = \neg x = 1$, противоречие. Если же $x = 0$, то $\neg x = 1$, и, аналогично рассматривая путь $\neg x \to y_1 \to \ldots \to y_{t-1} \to y_t \to x$, получаем противоречащее равенство $\neg x = x = 1$.

В обратную сторону: пусть для любой переменной x из вершины x нельзя достичь $\neg x$ и из вершины $\neg x$ нельзя достичь x одновременно. Покажем, что тогда существует выполняющий набор. Не умаляя общности, пусть из x не достижима вершина $\neg x$. Тогда не существует такой вершины y, что из x достижима вершина y, из которой достижима $\neg y$. Действительно, если бы такая вершина существовала, то существовал бы и путь из $\neg y$ в $\neg x$, как мы отметили ранее. Но тогда можно было бы из x попасть в вершину y, затем в $\neg y$, а потом в $\neg x$, то есть существовал бы путь из x в $\neg x$. Противоречие. Тогда выполняющий набор можно построить следующим способом: рассмотрим литерал x такой, что из x не достижим $\neg x$. Далее рассмотрим все вершины, достижимые из вершины x (включая x) и присвоим соответствующим литералам значение 1. Конфликтов возникнуть не может, т.к. если из x достижим и y и $\neg y$, то есть путь из x в y, а затем из y в $\neg x$, то есть путь из x в $\neg x$. А таких путей для вершины x нет по предположению. Таким образом мы получим множество литералов A_x, значения которых равны единице и которые достижимы из x. Этому множеству литералов также соответствует множество литералов B_x, состоящее из отрицаний литералов множества A_x. Значит, во множестве B_x все литералы равны нулю, конфликтов быть не может. При этом во множество B_x больше не входит никаких рёбер не из множества B_x, а из множества A_x не выходят рёбра в вершины не из множества A_x. Это означает, что мы частично (возможно и полностью) присвоили значения литералам так, что получился граф, у которого из присвоенных вершин выходит ребро либо в присвоенные вершины, а в неприсвоенные вершины выходит ребро только из нулевых вершин; аналогично, рёбро входит в присвоенную вершину, если ей присвоено значение 1, либо если это ребро проведено из другой присвоенной вершины. Иными словами, мы можем отбросить присвоенные вершины (вместе с рёбрами входящими и исходящими из них) и присваивать оставшимся вершинам значения по аналогии с тем, как мы это сделали для вершины x. При этом конфликтов, связанных с рёбрами, которые мы откинули не будет: мы отбросили рёбра, соответствующие импликациям вида $a \to 1$ и $0 \to b$, которые всегда выполнено, вне зависимости от a и b. Такими дийствиями мы получим набор, который выполняет все импликации, которые соответствуют рёбрам, а значит, и все дизъюнкты в исходной формуле.

Поэтому достаточно проверить, что в графе нет путей между x и $\neg x$, $\neg x$ и x для каждой переменной, встречающейся в формуле. Это можно сделать, найдя компоненты сильной связности. □

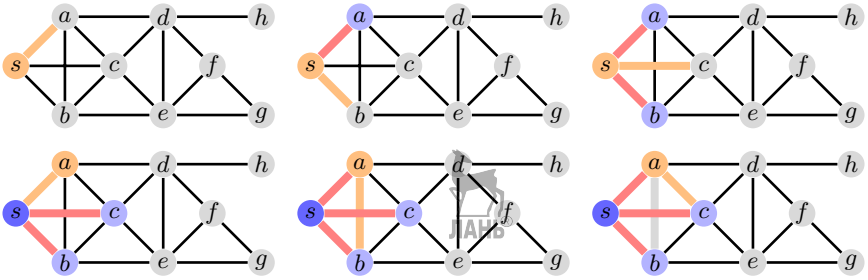
Breadth-first search

Теперь перейдем к изучению алгоритмов поиска кратчайших путей. Начнем с фундаментального алгоритма — поиска в ширину. Начиная с вершины s, алгоритм сканирует сначала 1-окрестность s, затем 2-окрестность итд. Как и DFS, поиск в ширину можно одинаковым образом запускать и на графах [неориентированных], и на орграфах.

Мы будем вычислять $d : V \to \mathbb{N}$ — длину кратчайшего пути от данной вершины до s. Вначале нам нужно создать очередь Q — линейно упорядоченное множество, добавлять элемент в которое можно только в конец, а удалять — только первый [first in — first out].

```
1:  procedure BFS(G = (V, E), s)
2:      Q ← ∅, d(s) ← 0
3:      while Q ≠ ∅ do
4:          извлечь первый элемент v из Q
5:          for w ∈ A_v do
6:              if d(w) не определено then
7:                  d(w) ← d(v) + 1
8:                  добавить w в Q
9:              end if
10:         end for
11:     end while
12:     return d
13: end procedure
```

Пример 4.16. Продемонстрируем работу BFS на графе из примера 4.2. Изначально $Q = \{s\}$.



Здесь показаны два шага — после первого шага $Q = \{a, b, c\}$, поэтому на втором шаге рассматривается вершина a. Тут получаем $Q = \{b, c, d\}$. Продолжая в том же духе, получим остовное дерево с рисунка на следующей странице.

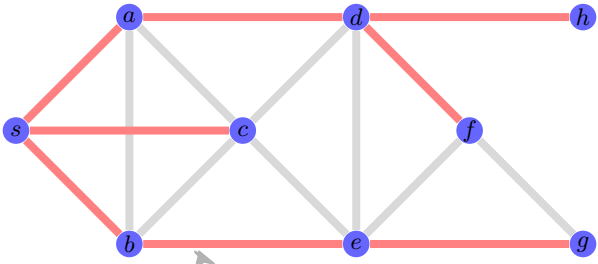


Рис. 9. BFS-дерево.

Теорема 4.17. *Алгоритм* BFS *имеет сложность* $O(|E|)$. *Для каждой вершины* v *вычисленное значение* $d(v)$, *если оно определено, равно длине кратчайшего пути от* s *до* v.

Доказательство. Очевидно, что каждое ребро проверяется не более двух раз BFS (в случае ориентированного только один раз) — каждая вершина, из которой выходит ребро, рассматривается однажды (в момент вычисления d), что дает утверждение о сложности. Более того, $d(s, v)$ не определено титтк v не достижима из s, и, таким образом, $d(v)$ остается неопределенным на протяжении работы алгоритма. Пусть теперь v — вершина, для которой $d(s, v)$ определено. Тогда $d(s, v) \leqslant d(v)$, так как существует путь из s в v длины $d(v)$ — v смежна с некоторой вершиной v_1, для которой $d(v_1) = d(v) - 1$, для v_1 аналогично найдем v_2, продолжая таким образом, найдем путь. Покажем индукцией по $d(s, v)$, что выполняется равенство. Для $d(s, v) = 0$ получаем $s = v$, равенство тривиально. Теперь предположим, что $d(s, v) = n + 1$ и пусть $(s, v_1, ..., v_n, v)$ — кратчайший путь от s до v. Тогда $(s, v_1, ..., v_n)$ — кратчайший путь из s к v_n и, согласно нашему предположению индукции, $d(s, v_n) = n = d(v_n)$. Следовательно, $d(v_n) = d(s, v_n) < d(s, v) = d(v)$, и, таким образом, BFS обрабатывает v_n раньше v. С другой стороны, G содержит ребро $v_n v$, так что BFS достигает v при рассмотрении списка смежности v_n (если не раньше). Это показывает $d(v) \leqslant n + 1$ и, следовательно, $d(v) = n + 1$. □

Нетрудно модифицировать BFS так, чтобы вычислять остовное дерево. По аналогии с DFS мы будем называть такое дерево *BFS-деревом*.

Теперь построим линейный по $|E|$ алгоритм проверки двудольности неориентрованного графа.

```
1: procedure Bipart(G = (V, E), s)
2:     Q ← ∅, d(s) ← 0, bip ← true, S ← ∅
3:     while Q ≠ ∅ и bip = true do
4:         извлечь первый элемент v из Q
5:         for w ∈ A_v do
6:             if d(w) не определено then
7:                 d(w) ← d(v) + 1
8:                 добавить w в Q
9:             else
10:                if d(v) = d(w) then
11:                    bip ← false
12:                end if
13:            end if
14:        end for
15:        if bip = true then
16:            for v ∈ V do
17:                if d(v) = 0 mod 2 then
18:                    S ← S ∪ {v}
19:                end if
20:            end for
21:            T ← V \ S
22:        end if
23:    end while
24:    return S, T, bip
25: end procedure
```

Теорема 4.18. *Граф G является двудольным тогда и только тогда, когда он не содержит циклы нечетной длины.*

Доказательство. Сначала предположим, что G двудольный, и пусть $V = S \cup T$ — соответствующее разбиение его множества вершин. Рассмотрим произвольный замкнутый путь в G, скажем

$$v_1 \text{ —— } v_2 \text{ —— } \cdots \text{ —— } v_n \text{ —— } v_1$$

Можно считать, что $v_1 \in S$. Тогда $v_2 \in T, v_3 \in S, v_4 \in T, \ldots, v_n \in T, v_1 \in S$, любое ребро соединяет одну вершину из S с одной вершиной из T. Следовательно, n должно быть четным.

Теперь предположим, что G не содержит циклов нечетной длины. Можно считать, что G связен. Выберите некоторую вершину x_0. Пусть S будет множество всех вершин x, имеющих четное расстояние $d(x, x_0)$ от x_0, и пусть $T = V \setminus S$. Предположим теперь, что в G

есть ребро xy с $x, y \in S$. Пусть W_x и W_y — кратчайшие пути из x_0 в x и y соответственно. По определению S оба эти пути имеют четную длину. Обозначим последнюю общую вершину W_x и W_y по z (пересекает оба пути, начиная с x_0), а их последние части (ведущие от z к x и y соответственно) — W'_x и W'_y. Тогда легко увидеть, что путь

$$x \overset{W'_x}{---} z \overset{W'_y}{---} y \overset{xy}{---} x$$

образует цикл нечетной длины в G, противоречие. Точно так же G не может содержать ребро xy, где $x, y \in T$. Отсюда $S \cup T$ является разбиением V таким, любое ребро соединяет одну вершину из S с одной вершиной из T. Следовательно, граф G двудольный. □

Утверждение 4.19. Bipart *определяет двудольность входного графа за* $O(|E|)$. *Если граф двудольный,* Bipart *также вычисляет соответствующее разбиение.*

Доказательство. Значение *bip* поменяется на **false** титтк $d(v) = d(w)$ для некоторой пары смежных вершин v и w; согласно теореме 4.18, эквивалентно наличию в графе цикла нечетной длины. Таким образом, алгоритм корректно определяет двудольность графа. Нетрудно также видеть, что все вершины, для которых $d(v)$ четно, образуют одну долю графа. Действительно, если вершины v_1 и v_2, для которых $d(v_1) = 2k_1$ и $d(v_2) = 2k_2$, соединены ребром, то рассуждениями, аналогичными теореме 4.18, получаем цикл нечетной длины, которого нет. Нетрудно также видеть, что сложность данного алгоритма $O(|E|)$. □

Поиск кратчайших путей

BFS позволяет находить наименьшие по количеству рёбер пути из заданной вершины во все вершины, в которые из неё можно попасть. Иными словами, поиск в ширину решает задачу о поиске кратчайших путей из данной вершины во все остальные вершины для графов, у которых все рёбра равноценны (имеют одинаковый вес). Но если граф моделирует, например, карту автомобильных дорог (вершины соответствуют населённым пунктам, рёбра — дорогам), то для нахождения кратчайшего (в житейском смысле) пути между двумя пунктами имеет смысл различать рёбра между собой. Обычно в таких случаях вводят *весовую функцию* $w : E \to \mathbb{R}$, которая каждому ребру графа ставит в соответствие некоторое *вещественное* число. В случае примера с картой автомобильных дорог разумно выбрать весовую функцию как длину соответствующей дороги в некоторых единицах измерения длины — в таком случае весовая функция будет положительной. Однако мы допускаем, что она может быть и отрицательной для некоторых графов. Такая общая постановка рассматривается не случайно: граф может задавать состояния некоторой фирмы; при переходе от одного состояния к другому фирма, скажем, заключает сделку; от сделки она может нести убытки или, наоборот, получать прибыль. В такой

постановке становится понятно, что если мы собираемся ввести весовую функцию, то она должна иметь возможность принимать значения обоих знаков.

Весом пути $p = (v_0, v_1, \ldots, v_k)$ называется следующая сумма:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Весом кратчайшего пути из вершины u в вершину v будем называть следующую величину

$$\delta(u, v) = \begin{cases} \min\{w(p) \mid u \overset{p}{\rightsquigarrow} v\}, & \text{если существует путь из } u \text{ в } v, \\ \infty & \text{иначе.} \end{cases}$$

Важно понимать, что если мы допускаем наличие рёбер отрицательного веса, то задача имеет смысл только в том случае, если нет циклов отрицательного веса (или их нет на любом пути между двумя интересующими нас вершинами). Действительно, по циклу отрицательного веса можно пройти сколько угодно раз, тем самым каждый раз уменьшая вес пути на некоторую положительную константу, то есть вес кратчайшего пути будет неопределён (в таком случае говорят, что он равен $(-\infty)$).

Кроме того, мы будем рассматривать задачи, в которых нам нужно найти какой-то кратчайший путь (кратчайших путей из одной вершины в другую может быть несколько, но для нас они все будут равноценны). Поэтому в рассматриваемых нами процедурах мы будем запоминать некоторого предшественника для каждой вершины (как это мы делали в алгоритмах поиска в ширину и в глубину). В течении работы рассматриваемые нами алгоритмы будут строить так называемые подграфы предшествования. Допустим, мы хотим найти кратчайшие пути из вершины s во все остальные вершины. *Подграфом предшествования* $G_\pi = (V_\pi, E_\pi)$ определяется следующи образом: V_π — это те вершины, у которых предшественник отличен от NIL, и стартовая вершина, то есть

$$G_\pi = \{v \in V \mid \pi[v] \neq \text{NIL}\} \cup \{s\};$$

рёбра G_π — это рёбра из $\pi[v] \neq \text{NIL}$ в v:

$$E_\pi = \{(\pi[v], v) \in E \mid v \in V_\pi \setminus \{s\}\}.$$

Во-первых, зафиксируем простое, но важное свойство: любая часть кратчайшего пути есть кратчайший путь. Формально это можно записать в виде следующей теоремы.

Теорема 4.20. *Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$. Если $p = (v_1, v_2, \ldots, v_k)$ — кратчайший путь из v_1 в v_k и $1 \leqslant i \leqslant j \leqslant k$, то $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ — кратчайший путь из v_i в v_j.*

Из него следуют ещё два полезных для нас факта.

Вывод 4.21. Пусть $G = (V, E)$ — взвешенный ориентированнный граф с весовой функцией $w : E \to \mathbb{R}$. Рассмотрим кратчайший путь p из s в v. Пусть $u \to v$ — последнее ребро в этом пути (p есть $s \overset{p'}{\rightsquigarrow} u \to v$). Тогда $\delta(s, v) = \delta(s, u) + w(u, v)$.

Вывод 4.22. Пусть $G = (V, E)$ — взвешенный ориентированнный граф с весовой функцией $w : E \to \mathbb{R}$; пусть $s \in V$. Тогда для всякого ребра $(u, v) \in E$ имеем $\delta(s, v) \leqslant \delta(s, u) + w(u, v)$.

Теперь обсудим технику *релаксации*. Идея состоит в том, чтобы для каждой вершины $v \in V$ хранить некоторое число $d[v]$, являющееся верхней оценкой веса кратчайшего пути из вершины s в вершину v (*оценка кратчайшего пути*). В начале работы алгоритмов поиска кратчайших путей мы будем вызывать функцию Initialize-Single-Source(G, s), которая для каждой вершины $v \in V$ делает два присваивания: $d[v] \leftarrow \infty, \pi[v] \leftarrow$ NIL; затем для вершины s мы кладём $d[s] \leftarrow 0$. Релаксация ребра (u, v) состоит в следующем: если $d[v] > d[u] + w(u, v)$, то $d[v] \leftarrow d[u] + w(u, v)$, так как мы можем попасть в вершину v через вершину u. Эту незатейливую процедуру мы будем обозначать Relax(u, v, w). Отметим, что сразу после вызова Relax(u, v, w) будет выполнено неравенство $d[v] \leqslant d[u] + w(u, v)$.

Кроме того, если мы сначала вызовем процедуру Initialize-Single-Source(G, s), а затем в некотором порядке будем релаксировать рёбра, то всегда будет выполнено неравенство $d[u] \geqslant \delta(s, u)$ для всех $u \in V$. Если в какой-то момент для некоторой вершины $v \in V$ будет выполнено равенство $d[v] = \delta(s, v)$, то оно останется верным и при последующих релаксациях рёбер. Отсюда следует, что если вершина $v \in V$ недостижима из s, то при произвольной последовательности релаксации рёбер значение $d[v]$ будет оставаться бесконечным.

Наконец, если в какой-то момент будет выполнено $d[u] = \delta(s, u)$ и мы через некоторое количество шагов прорелаксируем ребро (u, v), то в силу Следствия 1 получим, что $d[v] = \delta(s, v)$.

Теперь посмотрим, что происходит с графом предшествования G_π при произвольной последовательности релаксаций рёбер графа G. Будем рассматривать графы без циклов отрицательного веса, достижимых из начальной вершины. Оказывается, что в таком случае граф G_π будет всегда являться деревом с корнем в вершине s. Зафиксируем этот факт в виде следующей теоремы.

Теорема 4.23. *Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией w и стартовой вершиной s, причём в графе G нет циклов отрицательного веса, достижимых из s. Тогда после операции* Initialize-Single-Source(G, s), *за которой следует произвольная последовательность релаксаций рёбер графа G, подграф предшественников G_π является деревом с корнем s.*

Доказательство. В графе G_π после вызова Initialize-Single-Source(G, s) находится только вершина s, то есть в начальный момент этот подграф является деревом с корнем s. Покажем индукцией по количеству релаксаций рёбер, что это свойство сохраняется при произвольной последовательности релаксаций рёбер.

В графе G_π появляются новые вершины только в том случае, если происходит релаксация некоторого ребра (u, v), причём до этого $d[v]$ было бесконечным, а после релаксации стало конечным (это происходит только в том случае, если $d[u]$ на момент релаксации конечно). Тогда $\pi[v]$ становится равным u, то есть к дереву G_π добавляется лист v, то есть G_π остаётся деревом.

Проверим теперь ситуацию, когда происходит релаксация некоторого ребра (u, v), для которого и $d[u]$, и $d[v]$ являются конечными величинами. Это означает, что вершины u и v уже находятся в графе предшествования G_π. В случае $d[v] > d[u]+w(u, v)$ при релаксации ребра (u, v) поддерево с корнем v графа G_π отрезается от $\pi[v]$ и присоединяется к вершине u, то есть $\pi[v] \leftarrow u$. Возникает резонный вопрос: а не появлось ли после такой операции в графе G_π цикла? Если цикл возник после релаксации ребра (u, v), то из рассуждений выше следует, что вершина u обязана быть потомком v. Покажем, что тогда этот цикл обязан иметь отрицательный вес, откуда и получим противоречие.

Итак, перед реклаксацией ребра (u, v) выполнялось неравенство $d[v] > d[u] + w(u, v)$, то есть $d[u]-d[v]+w(u, v) < 0$. Покажем, что путь в графе G_π от вершины v до вершины u не превосходит $d[u] - d[v]$ (отсюда будет следовать, что вместе с ребром (u, v) он образует цикл отрицательного веса). Достаточно это показать для одного ребра, то есть показать, что если в какой-то момент ребро (x, y) принадлежит графу G_π, то в этот момент

$$w(x, y) \leqslant d[y] - d[x].$$

Непосредственно после релаксации (x, y) это неравенство обращается в равенство. Далее могут уменьшаться значения $d[y]$ и $d[x]$. Если уменьшается $d[x]$, то неравенство остаётся верным. Если же уменьшаетс $d[y]$, то это означает, что произошла релаксация некоторого ребра (z, y), а значит, теперь $\pi[y] = z$ и выполняется неравенство $w(z, y) \leqslant d[y] - d[z]$, а ребра (x, y) в графе G_π больше нет. □

Из доказанной теоремы следует, что если в какой-то момент $\forall v \in V \hookrightarrow d[v] = \delta(s, v)$, то граф G_π является *деревом кратчайших путей*.

Алгоритм Дейкстры. Алгоритм Дейкстры позволяет найти кратчайшие пути из вершины s во все достижимые из неё вершины для взвешенного ориентированного графа $G = (V, E)$, в котором *веса всех рёбер неотрицательные*.

Идея алгоритма состоит в следующем: поддерживается множество S, состоящее из вершин, для которых мы уже нашли кратчайшие пути (т.е. $d[u] = \delta(s, u)$). Далее алгоритм добавляет к S вершину $v \in V \setminus S$ с наименьшим $d[v]$, а затем производит релаксацию всех рёбер, выходящих из v, после чего цикл повторяется. Вершины, не лежащие в S, хранятся

в очереди с приоритетами Q (приоритеты определяются значениями d; чем меньше d, тем больше приоритет). Кроме того, считаем, что граф задан списком смежных вершин.

```
1: procedure Dijkstra(G, w, s)
2:     Initialize-Single-Source(G, s)
3:     S ← ∅
4:     Q ← V[G]
5:     while Q ≠ ∅ do
6:         u ← Extract-Min(Q)
7:         S ← S ∪ {u}
8:         for (для) всех вершин v ∈ Adj[u] do
9:             Relax(u, v, w)
10:        end for
11:    end while
12: end procedure
```

В строке u происходит вызов процедуры Extract-Min(Q), которая извлекает из очереди элемент с наименьшиv d. Далее мы обсудим детали реализации очереди с приоритетами, а пока что разберёмся, почему алгоритм Дейкстры корректно вычисляет кратчайшие расстояния.

Теорема 4.24. *Пусть $G = (V, E)$ — взвешенный ориентированный граф с неотрицательной весовой функцией $w : E \to \mathbb{R}$ и исходной вершиной s. Тогда после применения алгоритма Дейкстры к этому графу для всех вершин $u \in V$ будут выполняться равенства $d[u] = \delta(s, u)$.*

Доказательство. Покажем, что после любого числа итераций цикла **while** выполнены следующие свойства:

а) для вершин $v \in S$ значение $d[v]$ равно $\delta(s, v)$, причём существует кратчайший путь из s в v, целиком лежащий во множестве S, кроме последней вершины;

б) для вершин $v \in Q = V \setminus S$ значение $d[v]$ равно наименьшему весу пути из s в v среди тех путей, все вершины которых, кроме последней, лежат в S (если же таких путей нет, то $d[v] = \infty$).

После первой итерации цикла **while** в S лежит только вершина s и проведены релаксации всех рёбер, выходящих из s. Тогда свойства а) и б) будут выполнены. Проверим, что они не нарушатся и на следующих итерациях.

Пусть u — вершина из Q с наименьшим d. Если $d[u] = \infty$, то для всех вершин из Q значение d равно ∞, а значит, из S нельзя попасть во множество $V \setminus S$. Поэтому свойства а) и б) уже выполнены (по предположению индукции).

Если же $d[u]$ конечно, то существует кратчайший путь из s в u, целиком лежащий во множестве S, кроме последней вершины u. Действительно, если есть какой-то другой путь из s в u, не удовлетворяющий описанному свойству, то пусть первая вершина на этом

пути, которая не лежит в S есть $y \neq u$. Но в силу выбора u мы имеем, что $d[y] \geqslant d[u]$. По индуктивному предположению а) вес этого пути не меньше $d[y]$ (веса рёбер неотрицательны). Но тогда и вес всего пути не меньше $d[u]$. Следовательно, условие а) останется верным после добавления u ко множеству S. Осталось проверить условие б). Но оно будет выполнено после релаксации рёбер, исходящих из u, так как все пути, которые мы добавили в рассмотрение, проходят через вершину u (остальные уже были рассмотрены на предыдущих шагах). Поэтому, проведя релаксацию рёбер, выходящих из u, значения d для всех вершин из $V \setminus S'$, где $S' = S \cup \{u\}$, будут равняться весам наименьших путей, которые целиком лежат в S', кроме последней вершины. □

Из Теоремы 2 следует, что алгоритм Дейкстры позволяет построить дерево кратчайших путей с корнем в вершине s.

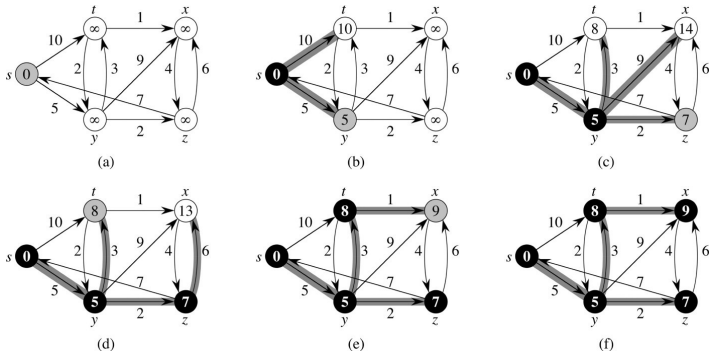


Рис. 10. Пример работы алгоритма Дейкстры.

Оценим время работы алгоритма Дейкстры. Для начала рассмотрим случай, когда очередь с приоритетами задаётся как обычный массив. Тогда стоимость операции Extract-Min есть $O(V)$ (эквивалентно задаче поиска наименьшего элемента в массиве длины V). Так нужно сделать для каждой вершины ровно V раз, а значит, суммарная стоимость всех удалений вершин из очереди равна $O(V^2)$. Кроме того, каждая вершина добавляется ко множеству S ровно один раз, и каждое ребро из $\mathrm{Adj}[v]$ обрабатывается только один раз. Стоимость каждой релаксации есть $O(1)$, поэтому общая стоимость алгоритма есть $O(V^2 + E) = O(V^2)$.

Однако если граф является разреженным, то выгоднее реализовать очередь с приоритетами с помощью двоичной кучи. Как это делать? Во-первых, мы будем поддерживать основное свойство кучи с обратным знаком, то есть будем поддерживать $A[i] \geqslant$

$A[\textsc{Parent}[i]]$ (сравниваются значения d). Процедура Extract-Min в таком случае реализуется простым способом: нужно вернуть корневую вершину двоичной кучи (там будет минимум), а затем поменять в массиве первый элемент с последним, уменьшить размер кучи на единицу и вызвать процедуру $\textsc{Heapify}(A, 1)$. Время работы процедуры Extract-Min составляет $O(\log V)$. Стоимость построения кучи (строка 4) составляет $O(V)$. Присваивание $d[v] \leftarrow d[u] + w(u, v)$ можно так же реализовать за $O(\log V)$ (уменьшаем ключ у вершины v до значения $d[u] + w(u, v)$, а затем «поднимаем» её на нужный уровень вверх: если значение ключа у родителя больше, чем у данной вершины, то меняем их местами, и так далее). Всего таких присваиваний будет произведено не более E штук, так что общая стоимость алгоритма Дейкстры, основанного на очередях с приоритетами, реализованными при помощи двочиных куч, составляет $O((V + E) \log V)$. Если граф связен, то $E \geqslant V - 1$ и предыдущая оценка записывается в виде $O(E \log V)$.

Алгоритм Беллмана-Форда. *Алгоритм Беллмана-Форда* решает задачу о поиске кратчайших путях из одной вершины для случая, когда веса рёбер могут быть отрицательными, но в графе нет циклов отрицательного веса, достижимых из s. В отличие от алгоритма Дейкстры данный алгоритм умеет решать задачу и для графов с отрицательными весами рёбер. Более того, он может определить, есть ли в графе цикл отрицательного веса (если нет, то алгоритм выдаст True, а иначе — False).

```
 1: procedure Bellman-Ford(G, w, s)
 2:     Initialize-Single-Source(G, s)
 3:     for i ← 1 to |V(G)| − 1 do
 4:         for (для) каждого ребра (u, v) ∈ E(G) do
 5:             Relax(u,v,w)
 6:         end for
 7:     end for
 8:     for (для) каждого ребра (u, v) ∈ E(G) do
 9:         if d[v] > d[u] + w(u, v) then
10:             return False
11:         end if
12:     end for
13:     return True
14: end procedure
```

Иными словами, $V - 1$ раз производится релаксация всех рёбер графа G. Как мы увидим далее, после такой операции для всех $v \in V$ должно выполняться равенство $d[v] = \delta(s, v)$, если в графе нет циклов отрицательной длины. Цикл в строках 8-12 проверяет граф G на отсутствие циклов отрицательной длины (см. Следствие 2).

Время работы алгоритма Беллмана-Форда есть $O(VE)$, поскольку в строках 3-7 выполняется $O(VE)$ релаксаций рёбер, время работы релаксации одного ребра равняется $O(1)$, строка 1 выполняется за время $O(V)$, а цикл в строках 8-12 — за время $O(E)$.

Теорема 4.25. *Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$ и исходной вершиной s, не содержащий циклов отрицательного веса, достижимых из s. Тогда после применения алгоритма Беллмана-Форда к этому графу для всех вершин $u \in V$ будут выполняться равенства $d[u] = \delta(s, u)$.*

Доказательство. Рассмотрим произвольный кратчайший путь $s = v_0, v_1, \ldots, v_k = v$. Так как в графе нет циклов отрицательной длины, то можно считать, что в пути p нет циклов (цикл не уменьшает длину пути). Поэтому можно считать, что $k \leqslant V - 1$. Покажем индукцией по i, что после i-й итерации цикла в строках 3-7 будет выполнено равенство $d[v_i] = \delta(s, v_i)$ (если это доказать, то получим, что после $V - 1$ итерации в $d[v_k] = d[v]$ будет записано $\delta(s, v)$, а в силу произвольности выбора v это будет выполнено и для всех вершин в графе G).

При $i = 0$ утверждение верно ($d[s] = \delta(s, s) = 0$). Пусть при $i > 0$ после $(i - 1)$-й итерации выполнялось равенство $d[v_{i-1}] = \delta(s, v_{i-1})$. Тогда из Следствия 1 получаем, что после i-й итерации выполняется $d[v_i] = \delta(s, v_i)$, поскольку на i-й итерации произойдёт релаксация ребра (v_{i-1}, v_i). □

Из Теоремы 2 следует, что алгоритм Беллмана-Форда позволяет построить дерево кратчайших путей с корнем в вершине s.

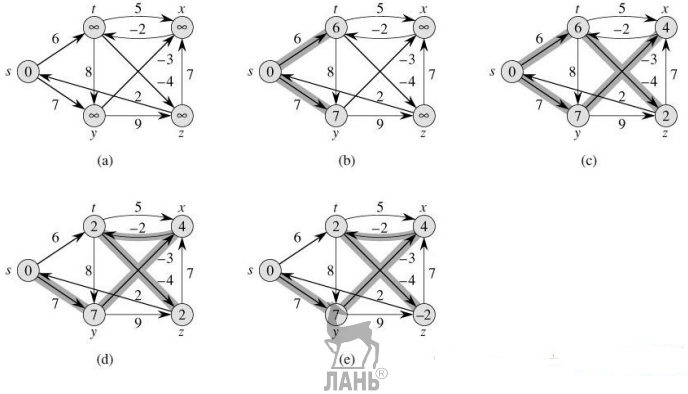


Рис. 11. Пример работы алгоритма Беллмана-Форда.

Алгоритм Флойда-Уоршолла. *Алгоритм Флойда-Уоршолла* решает задачу поиска кратчайших путей между любыми двумя парами вершин ориентированного взвешенного графа, у которого нет циклов отрицательного веса. Мы покажем, что этот метод работает за время $\Theta(V^3)$, то есть для плотных графов (когда $E = O(V^2)$) алгоритмы Флойда-Уоршолла и Беллмана-Форда имеют одинаковую сложность $O(V^3)$.

Перейдём к описанию алгоритма. *Промежуточной* вершиной простого пути $v_1, v_2, \ldots, v_l$ буде называть любую из вершин $v_2, v_3, \ldots, v_{l-1}$ (напомню, что раз мы работаем с графами, в которых нет циклов отрицательного веса, то можем считать, что кратчайший путь между любыми двумя вершинами не содержит циклов, а значит, кратчайший путь состоит не более чем из $V - 1$ ребра).

Будем считать, что вершинами графа G являются числа $1, 2, 3, \ldots, n$. Рассмотрим произвольное $k \leqslant n$. Для данной пары вершин $i, j \in V$ рассмотрим все пути из i в j, у которых все промежуточные вершины лежат во множестве $\{1, 2, \ldots, k\}$. Пусть p — путь минимального веса среди всех таких путей (он будет простым, как мы уже обсудили). Есть две возможности. Если k не является промежуточной в пути p, то все промежуточные вершины принадлежат $\{1, 2, \ldots, k-1\}$, то есть p будет и кратчайшим путём между i и j среди путей, у которых промежуточные вершины лежат во множестве $\{1, 2, \ldots, k-1\}$. Если же k является промежуточной вершиной в пути p, то она разбивает путь p на две части p_1 и p_2: $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ (вершина k встречается лишь однажды, ибо путь p является простым). Но тогда путь p_1 является кратчайшим путём из i в k среди путей с промежуточными вершинами из $\{1, 2, \ldots, k-1\}$, а путь p_2 — кратчайшим из k в j с промежуточными вершинами из множества $\{1, 2, \ldots, k-1\}$.

Зафиксируем эти рассуждения в виде рекуррентной формулы. Пусть $d_{ij}^{(k)}$ — вес кратчайшего пути из i в j с промежуточными вершинами из $\{1, 2, \ldots, k\}$. Тогда из написанного выше следует, что

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{если } k = 0, \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}, & \text{если } k \geqslant 1. \end{cases}$$

Матрица $D^{(n)} = (d_{ij}^{(n)})$ содержит искомое решение, поскольку разрешаются любые промежуточные вершины.

Кроме того, нам хотелось бы уметь находить не только веса кратчайших путей, но и сами пути. Поэтому нужен способ вычисления предшественников. К счастью, это тоже можно задать рекуррентной формулой. Пусть $\pi_{ij}^{(k)}$ — это вершина, предшествующая вершине j на кратчайшем пути из i в j с промежуточными вершинами из множества $\{1, 2, \ldots, k\}$. Тогда

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL}, & \text{если } i = j \text{ или } w_{ij} = \infty, \\ i, & \text{если } i \neq j \text{ и } w_{ij} < \infty, \end{cases}$$

и для $k \geqslant 1$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{если } d_{ij}^{(k-1)} \leqslant d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)}, & \text{если } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Напишем процедуру, которая по матрице весов рёбер W вычисляет матрицу кратчайших расстояний $D^{(n)}$.

```
1:  procedure FLOYD-WARSHALL(W)
2:      n ← rows[W]
3:      Initialize Π^(0)
4:      D^(0) ← W
5:      for k ← 1 to n do
6:          for i ← 1 to n do
7:              for j ← 1 to n do
8:                  if d_ij^(k-1) ⩽ d_ik^(k-1) + d_kj^(k-1) then
9:                      d_ij^(k) ← d_ij^(k-1)
10:                     π_ij^(k) ← π_ij^(k-1)
11:                 else
12:                     d_ij^(k) ← d_ik^(k-1) + d_kj^(k-1)
13:                     π_ij^(k) ← π_kj^(k-1)
14:                 end if
15:             end for
16:         end for
17:     end for
18:     return D^(n)
19: end procedure
```

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Рис. 12. Пример работы алгоритма Флойда-Уоршолла.

Минимальные остовные деревья

Определение 4.26. Пусть (G, w) — взвешенный граф. *Весом* подграфа T будем называть $w(T) = \sum_{e \in T} e \in T$.

Определение 4.27. Остовный лес [дерево] T графа G будем называть *минимальным остовным лесом*, если среди всех остовных лесов [lthtdmtd] он обладает минимальным весом.

Будем считать G неориентированным графом. Если весовая функция w постоянна, каждое остовное дерево минимально, его можно найти со сложностью $O(|E|)$ использованием BFS или DFS. Для случая общего взвешенного графа простой перебор всевозможных деревьев с поиском минимального в силу теоремы Кэли может иметь неполиномиальную сложность [для полного графа, например]. Мы предъявим три непереборных алгоритма.

Но сначала мы характеризуем минимальные остовные деревья. Рассмотрим остовное дерево T и ребро $e \notin T$. Очевидно, что граф $T \cup \{e\}$ содержит единственный цикл, обозначим этот цикл через $C_T(e)$.

Теорема 4.28. *Пусть (G, w) — сеть, где G — связный граф. Остовное дерево T группы G минимально титтк для любого ребра e в $G \setminus T$ и $f \neq e$ в $C_T(e)$ выполняется*

$$w(e) \geqslant w(f) \tag{25}$$

Доказательство. Предположим сначала, что T минимально. Если неравенство 25 теоремы не выполняется, то существует ребро e в $G \setminus T$ и ребро f в $C_T(e)$ с $w(e) < w(f)$. Удаление f разбивает T на две компоненты связности V_1 и V_2, так как f является мостом [$f \neq e$, поэтому лежит в T]. Если мы добавим путь $C_T(e) \setminus \{f\}$ к $T \setminus \{f\}$, компоненты V_1 и V_2 снова связны. Следовательно, $C_T(e) \setminus \{f\}$ должен содержать ребро, соединяющее вершину в V_1 с вершиной в V_2, и может быть только . Таким образом, добавление e к $T \setminus \{f\}$ дает новое дерево T. Поскольку $w(e) < w(f)$, это дерево имеет меньший вес, чем T, что противоречит минимальности T.

Наоборот, предположим, что T удовлетворяет 25. Покажем, что его вес равен весу некоторого минимального дерева $\widehat{T}$; это сделаем индукцией по числу ребер в $\widehat{T} \setminus T$. Случай $k = 0$ тривиален; пусть по предположению индукции любое дерево T_k, удовлетворяющее 25 и такое, что в $T_k \setminus T$ ровно k ребер, является минимальным, покажем, что если $\widehat{T} \setminus T$ содержит $k + 1$ ребро, а условие 25 выполнено для $\widehat{T}$, то и $\widehat{T}$ минимально. Рассмотрим ребро $\widehat{e}$ в $\widehat{T} \setminus$. При удалении $\widehat{e}$ дерево $\widehat{T}$ распадается на две компоненты связности, $\widehat{T}_1$ и $\widehat{T}_2$. Аналогичными рассуждениями получаем, что граф $C_T(\widehat{e}) \setminus \{\widehat{e}\}$ должен содержать ребро e, соединяющее $\widehat{T}_1$ и $\widehat{T}_2$. Обратите внимание, что e не может быть ребром $\widehat{T}$, так как в противном случае $\widehat{T} \setminus \{\widehat{e}\}$ будет связным. Обозначим $\widehat{T}_e = T_1 \cup T_2 \cup \{e\}$.

Минимальное остовное дерево $\widehat{T}$ удовлетворяет 25 и, следовательно, $w(e) \geqslant w(\widehat{e})$. С другой стороны, так как также удовлетворяет условию 25, то $w(\widehat{e}) \geqslant w(e)$ и, следовательно, $w(\widehat{e}) = w(e)$. Следовательно, $w(\widehat{T}_e) = w(\widehat{T})$, так что $\widehat{T}_e$ снова минимальное остовное дерево. У $\widehat{T}'$ на одно общее ребро больше, чем у $\widehat{T}$, по предположению индукции $w(T) = w(\widehat{T}_e)$, таким образом, $w(T) = w(\widehat{T}_e) = w(\widehat{T})$. □

Определение 4.29. Пусть G — граф с множеством вершин V. *Разрез* $S = \{X, Y\}$ — разбиение $V = X \sqcup Y$ на два непустых непересекающихся подмножества.

Пусть S — разрез графа. Обозначим множество всех ребер, инцидентных одной вершине в X и одной вершине в Y как $E(S)$ или $E(X, Y)$. Иногда [перегружая обозначение] под разрезом мы будем понимать $E(X, Y)$ вместо $\{X, Y\}$.

Лемма 4.30. *Пусть G — связный граф, а T — остовное дерево группы G. Для каждого ребра e в T существует единственный разрез $S_T(e)$ графа G такой, что e является единственным общим ребром T и $S_T(e)$.*

Доказательство. Если мы удалим e из T, дерево будет разделено на две компоненты связности, таким образом, мы получаем разрез $S_T(e)$. Очевидно, что $e \in S_T(e)$ и никакое другое ребро дерева T не может лежать в разрезе. Легко видеть, что разрез с искомым свойством единственный — вершины x и y ребра e должны лежать в разных компонентах разреза, все остальные смежные в T вершины должны быть в одной компоненте разреза, таким образом, все вершины поддерева с корнем x и поддерева с корнем y должны образовывать компоненты разреза. □

Теорема 4.31. *Пусть (G, w) — взвешенный граф и G — связный. Остовное дерево T графа G минимально тогда и только тогда, когда для любого ребра e в T и $f \neq e$ в $S_T(e)$ выполняется*

$$w(e) \geqslant w(f)$$

Доказательство. Допустим, что T минимально. Если в T есть ребро e и ребро f в $S_T(e)$ такие, что $w(e) > w(f)$, то удалив e из T и добавив f, мы могли бы построить остовное дерево меньшего веса, чем T [новых циклов не появится по определению $S_T(e)$].

Теперь предположим, что для любого ребра e в $G \setminus T$ и $f \neq e$ в $S_T(e)$ выполняется $w(e) \geqslant w(f)$. Рассмотрим ребро e в $G \setminus T$ и $f \neq e$ ребро в $C_T(e)$. Очевидно e содержится в $S_T(f)$, следовательно, неравенство $w(e) \geqslant w(f)$ влечет 25, откуда по теореме 4.28 следует, что T минимально. □

Алгоритмы Прима, Крускала и Борувки

Теоремы, приведенные выше, лежат в основе самых популярных алгоритмов поиска остовных деревьев. Исторически алгоритм Борувки был предъявлен первым, еще в 1926

году — Борувке была поручена задача нахождения наименее затратного способа проложить электрическую сеть в Южной Моравии. Впоследствии задача поиска минимального остовного дерева появлялась во множестве разных ситуаций, следующая диаграмма приведена в статье [13]:

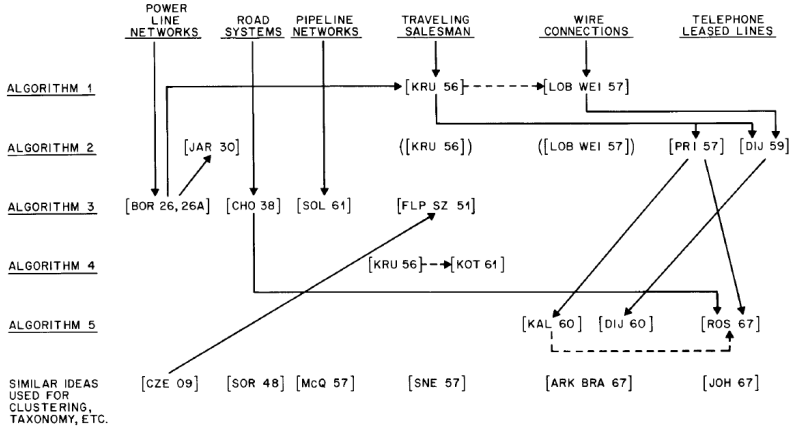


Рис. 13. История появления алгоритмов нахождения минимального остовного дерева. По порядку здесь занумерованы алгоритм Крускала, Прима и Борувки. Последние два алгоритма, 4 и 5 — модификации алгоритма Крускала. Ребро $X \to Y$ интерпретируется как «в работе Y есть ссылка на работу X»

Сперва опишем общий метод, спецификациями которого и являются алгоритмы Прима, Крускала и Борувки.

procedure MINTREE($G = (\{1, \ldots, n\}, E)$, $w : E \to \mathbb{R}$)
  for $i \in [1; n]$ **do**
    $V_i \leftarrow \{i\}$, $T_i \leftarrow \varnothing$
  end for
  for $i \in [1; n-1]$ **do**
    выбрать V_i с помощью $V_i \neq \varnothing$
    выбрать ребро $e = uv$ с $u \in V_i, v \notin V_i$ и $w(e) \leqslant w(\widehat{e})$ для всех ребер $\widehat{e} = \widehat{u}\widehat{v}$ с $\widehat{u} \in V_i$, $\widehat{v} \notin V_i$
    определить индекс j, для которого $v \in V_j$
    $V_i \leftarrow V_i \cup V_j$, $V_j \leftarrow \varnothing$

```
10:         T_i ← T_i ∪ T_j ∪ {e}, T_j ← ∅
11:         if k = n − 1 then
12:             T ← T_i
13:         end if
14:     end forreturn T
15: end procedure
```

Теорема 4.32. *Процедура* Mintree, *описанная выше корректно вычисляет минимальное остовное дерево входного графа.*

Доказательство. Индукцей по $t = |T_1|+\ldots+|T_n|$ докажем следующее утверждение: для любого $t \in \{0,\ldots,n-1\}$ существует минимальное остовное дерево T графа G, содержащее $T_1,\ldots,T_n$. Из истинности этого утверждения для $t = n-1$ следует корректность алгоритма. Ясно, что при $t = 0$, то есть перед выполнением второго цикла, получается остовное дерево. Теперь предположим, что для $t = k-1$, то есть до того, как цикл будет выполнен в k-й раз, получилось остовное дерево. Пусть $= uv$ с $u \in V_i$ — ребро, построенное на k-й итерации. Если содержится в минимальном остовном дереве T, содержащем $T_1,\ldots T_n$ при $t = k-1$, доказывать нечего. Таким образом, мы можем считать, что $e \notin T$. Тогда $T \cup \{e\}$ содержит единственный цикл $C = C_T(e)$; очевидно, C должен содержать другое ребро $f = rs$ с $r \in V_i$ и $s \notin V_i$. По теореме 4.28 $w(e) \geqslant w(f)$. С другой стороны, по выбору на шаге $w(e) \leqslant w(f)$. Следовательно, $w(e) = w(f)$ и $T' = (T \cup \{e\}) \setminus \{f\}$ является минимальным остовным деревом G, удовлетворяющее нашим условиям при $t = k$. □

Сложность указанной прцедуры зависит как от выбора индекса i, так и от деталей реализации. Теперь перейдем к спецификациям этой процедуры.

Начнем с *алгоритма Прима*. Пусть G — связный граф с множеством вершин $V = \{1,\ldots,n\}$ задается списками смежности A_v, и пусть $w : E \to \mathbb{R}$ — весовая функция графа G.

```
1: procedure Prim(G = (V, E), w)
2:     g(1) ← 0, S ← ∅, T ← ∅
3:     for i ∈ [2; n] do
4:         g(i) = ∞
5:     end for
6:     while S ≠ V do
7:         выбрать i ∈ V \ S так, чтобы g(i) было минимальным, S ← S ∪ {i}
8:         if i ≠ 1 then
9:             T ← T ∪ {e(i)}
10:        end if
11:        for j ∈ A_i ∩ (V \ S) do
12:            if g(j) > w(ij) then
13:                g(j) ← w(ij), e(j) ← ij
```

14: **end if**
15: **end for**
16: **end while**
17: **end procedure**

Пример 4.33. Применим алгоритм Прима к взвешенному графу на рисунке **??**, расположенном ниже. Мы помечаем ребра следующим образом: $e_1 = \{1,5\}, e_2 = \{6,8\}, e_3 = \{1,3\}, e_4 = \{4,5\}, e_5 = \{4,8\}, e_6 = \{7,8\}, e_7 = \{6,7\}, e_8 = \{4,7\}, e_9 = \{2,5\}, e_{10} = \{2,4\}, e_{11} = \{2,6\}, e_{12} = \{3,6\}, e_{13} = \{5,6\}, e_{14} = \{3,8\}, {}_15 = \{1,2\}$. Таким образом, ребра упорядочены в соответствии с их весом. Алгоритм Prim работает так, как показано в таблице ниже. Результирующее минимальное остовное дерево обозначено жирными ребрами на рисунке **??**.

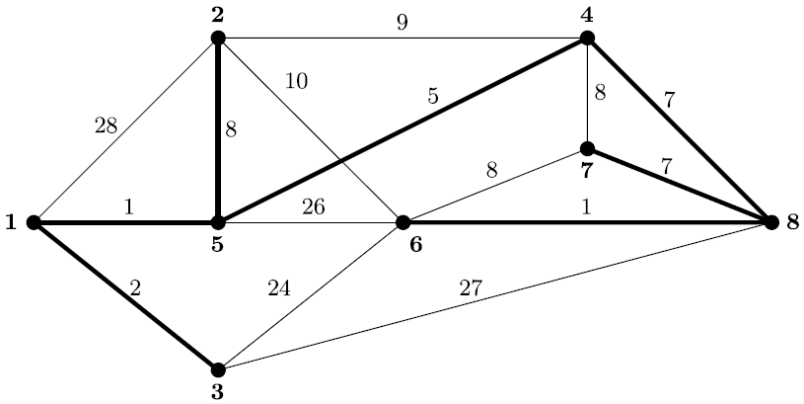


Рис. 14. Пример работы алгоритма Прима.

Итерация 1: $i = 1, S = \{1\}, T = \varnothing, g(2) = 28, e(2) = e_{15}, g(5) = 1, e(5) = e_1, g(3) = 2, (3) = e_3$,
Итерация 2: $i = 5, S = \{1,5\}, T = \{e_1\}, g(2) = 8, e(2) = e_9, g(4) = 5, e(4) = e_4, g(6) = 26, e(6) = e_{13}$,
Итерация 3: $i = 3, S = \{1,5,3\}, T = \{e_1, e_3\}, g(6) = 24, e(6) = e_{12}, g(8) = 27, e(8) = e_{14}$,
Итерация 4: $i = 4, S = \{1,5,3,4\}, T = \{e_1, e_3, e_4\}, g(7) = 8, e(7) = e_8, g(8) = 7, (8) = {}_5$,
Итерация 5: $i = 8, S = \{1,5,3,4,8\}, T = \{e_1, e_3, e_4, e_5\}, g(6) = 1, e(6) = e_2, g(7) = 7, (7) = {}_6$,
Итерация 6: $i = 6, S = \{1,5,3,4,8,6\}, T = \{e_1, e_3, e_4, e_5, e_2\}$,
Итерация 7: $i = 7, S = \{1,5,3,4,8,6,7\}, T = \{e_1, e_3, e_4, e_5, e_2, e_6\}$,

Итерация 8: $i = 2, S = \{1, 5, 3, 4, 8, 6, 7, 2\}, T = \{e_1, e_3, e_4, e_5, e_2, e_6, e_9\}$,

Теорема 4.34. *Алгоритм Прима корректно находит минимальное остовное дерево; сложность алгоритма Прима есть* $O(|V|^2)$.

Доказательство. Легко видеть, что алгоритм Прима является частным случаем алгоритма Mintree: если мы всегда выбираем V_1 на шаге 5 алгоритма Mintree, получаем алгоритма Прима. Функция $g(i)$, введенная здесь, используется для упрощения поиска кратчайшего ребра, выходящего из $V_1 = S$. Следовательно, согласно теореме 4.32. Оценивая сложность, заметим, что цикл **while** выполняется $|V|$ раз, каждая итерация которого проделывается за не более чем $|V| - |S|$ элементарных операций, так что мы получаем сложность $O(|V|^2)$. Поскольку G — простой граф, это также общая сложность: на шаге (11) каждое ребро G рассматривается ровно дважды. □

Теперь перейдем к *алгоритму Крускала*. Сначала дадим несколько общую версию. Пусть $G = (V, E)$ — связный граф с $V = \{1, \ldots, n\}$, $w : E \to \mathbb{R}$ — весовая функция. Ребра G упорядочены в соответствии с их весом, то есть $E = \{e_1, \ldots, e_m\}$ с $w(e_1) \leqslant \ldots \leqslant w(e_m)$.

1: **procedure** Kruskal($G = (V, E)$, w)
2: $T \leftarrow \varnothing$
3: **for** $k \in [1; m]$ **do**
4: **if** e_k не образует цикл вместе с некоторыми ребрами T **then**
5: добавить E_k к T
6: **end if**
7: **end for**
8: **return** T
9: **end procedure**

Алгоритм Крускала является частным случаем Minstree, в котором V_i и e выбраны таким образом, чтобы $w(e)$ было минимальным среди всех еще доступных ребер [которые не соединяют две вершины некоторого V_j и, следовательно, создадут цикл]. Из теоремы 4.32 следует корректность алгоритма.

Чтобы расположить ребра в очереди в соответствии с их весом, мы используем структуру данных *очередь с приоритетами* — множество пар (x, p) [где $p \in \mathbb{R}$ — *приоритет*, x — ключ] с операциями «добавить новый элемент» и «извлечь минимальный элемент» [в нашем случае мы хотим извлекать минимальный элемент]. Эти операции могут быть выполнены за $O(|E| \log |E|)$ шагов.

1: **procedure** Kruskal($G = (V, E)$, w)
2: $T \leftarrow \varnothing$
3: **for** $i \in [1; n]$ **do**
4: $V_i \leftarrow \{i\}$
5: **end for**

```
    поместить E в очередь приоритетов Q с функцией приоритета w
    while Q ≠ ∅ do
        e := DELETEMIN (Q)
        найти конечные вершины u и v из e
        найти компоненты V_u и V_v, содержащие u и v соответственно
        if V_u ≠ V_v then
            MERGE(V_u, V_v)
            T ← T ∪ {e}
        end if
    end while
    return T
end procedure
```

Теорема 4.35. *Алгоритм Крускала определяет со сложностью* $O(|E|\log|E|)$ *минимальное остовное дерево для* (G, w).

Доказательство. Корректность, как мы отметили выше, следует из теоремы 4.32.

Теперь проверим, что сложность действительно $O(|E|\log|E|)$. Легко определить сложность итерации. Нахождение и удаление минимального ребра e в очереди приоритетов требует $O(\log|E|)$ шагов. Последовательное объединение исходных n тривиальных компонентов и поиск компонентов на этапе (7) может быть выполнено с общим усилием $O(n\log n)$ шагов. Поскольку G связан, G имеет по крайней мере $n-1$ ребра, так что общая сложность $O(|E|\log|E|)$.

□

Пример 4.36. Грименим алгоритм Крускала к графу **??**. Ребра $e_{1,\ 2,\ 3,\ 4,\ 5}$ e_6 и e_9 выбираются последовательно, так что мы получаем то же самое остовное дерево, что и с алгоритмом Prim (хотя там ребра были выбраны в другом порядке).

В примере **??** ровно одно остовное дерево. В общем случае алгоритмы Прима и Крускала будут давать разные минимальные остовные деревья [могут выбираться разные минимальные ребра].

Обратимся теперь к алгоритму Борувки. Пусть $G = (V, E)$ связный граф с $V = \{1, \ldots, n\}$, $w : E \to \mathbb{R}$ - весовая функция/

```
1: procedure Boruvka(G = (V, E), w)
2:     for i ∈ [1; n] do
3:         V_i ← {i}
4:     end for
5:     T ← ∅, M ← {V1, ..., Vn}
6:     while |T| < n − 1 do
7:         for U ∈ M do
```

```
        найти ребро e = uv с u ∈ U, v ∉ U и w(e) < w(ê) ∀ê = ûv̂ с û ∈ U, v̂ ∉ U
        найти компонент Û, содержащий v
        T ← T ∪ {e}
    end for
    for U ∈ M do
        MERGE (U, Û)
    end for
  end while
  return T
end procedure
```

Пример 4.37. Применяем алгоритм Борувки к графу на **??**. Когда цикл **while** выполняется впервые, ребра $\{1,2\}$, $\{3,6\}$, $\{4,5\}$, $\{4,7\}$ и $\{7,8\}$ (выделены жирным шрифтом на рисунке ниже) выбраны и вставлены в T. Это оставляет только три компоненты связности, которые объединяются во время второго выполнения цикла **while** путем добавления ребер $\{2,5\}$ и $\{1,3\}$ (показаны жирным штрихом на картинке).

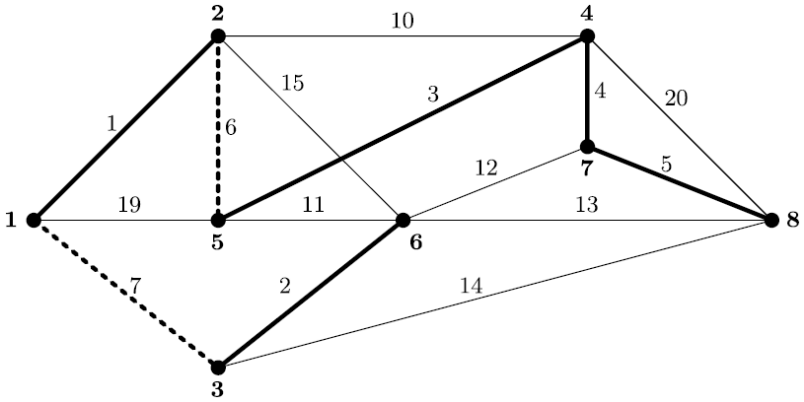


Рис. 15. Пример работы алгоритма Борувки

Теорема 4.38. *Алгоритм Борувки определяет минимальное остовное дерево для* (G, w) *за* $O(|E| \log |V|)$ *шагов.*

Доказательство. Из теоремы 4.32 следует, что алгоритм корректен; теперь оценим сложность. Для каждой вершины v у нас должен изначально быть список E_v ребер, инцидентных с v. Состояние все веса ребер являются четкими гарантиями отсутствия циклов

во время выполнения цикла **while**. Поскольку число компонент связности уменьшается на каждой итерации по меньшей мере вдвое, цикл **while** выполняется не более $\log |V|$ раз. Каждая итерация цикла может быть проделана за $O(|E|)$ шагов: среди ребер E_U, инцидентных U, максимальное находится за $O(|E_U|)$, суммарно за каждый шаг совершается $O(\sum_{U \in M} |E_U|) = O(E)$. Это приводит к сложности $O(|E| \log |V|)$. □

Потоки и сети

Определение 4.39. *Транспортной сетью* (G, c, s, t) будем называть простой [без петель и кратных ребер] ориентированный граф $G = (V, E)$ с выделенными вершинами [*истоком* s и *стоком* t], где каждому ребру $e \in E$ сопоставлена *пропускная способность* $c(e) > 0$.

Можно считать, например, что ребра — это трубопроводы, а $c(e)$ — максимальное количество воды, которое можно пустить по данному трубопроводу. Естественная задача, которая при этом возникает — какое максимальное количество воды можно пустить по данной сети от истока к стоку? Для удобства продолжим функцию $c(\cdot, \cdot)$ на все пары вершин, так что $c(u, v) = 0$, если $(u, v) \notin E$, и $c(u, v) = -c(v, u)$.

Определение 4.40. *Потоком* в сети G называется функция $f \colon V \times V \to \mathbb{R}_+$, удовлетворяющая следующим условиям.

- Ограничения пропускных способностей: $0 \leqslant f(u, v) \leqslant c(u, v)$ для всех $uv \in E$.
- Кососимметричность: $f(u, v) = -f(v, u)$.
- Сохранение потока (закон Кирхгофа): *дивергенция* $\nabla_f(u) := \sum_v f(u, v) - \sum_v f(v, u) = 0$ для всех $u \in V \setminus \{s, t\}$.

Соответственно, *величина потока* f — это количество воды, которое переходит из s в t, и в силу закона сохранение оно равно

$$w(f) = \nabla_f(s) = \sum_v f(s, v) - \sum_v f(v, s).$$

Первый метод, который мы рассмотрим, основан на следующей идее: будем пускать дополнительный поток по какому-нибудь пути из s в t, пока это возможно.

Сначала сделаем несколько технических предположений на сеть G, помогающих при анализе метода, но от которых несложно отказаться при реализации:

- в G нет *антипараллельных* ребер, т.е. если $(u, v) \in E$, то $(v, u) \notin E$; этого всегда можно добиться, разбивая ребра, т.е. если $(u, v) \in E$ и $(v, u) \in E$, то добавляем вспомогательную вершину z и заменям ребро (v, u) на (v, z) и (z, u) с той же пропускной способностью;
- через каждую вершину графа проходит какой-нибудь путь из s в t (иначе через эту вершину нельзя бы было пустить поток); отсюда следует, что $|E| \geq |V| - 1$, т.к. граф связный.

Чтобы формально описать метод Форда–Фалкерсона, введем понятие остаточной сети.

Определение 4.41. *Остаточная сеть* потока f — сеть $G_f = (V, E_f)$ с пропускными способностями

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

и ребрами $E_f := \{(u,v) : c_f(u,v) > 0\}$.

Заметим, что в этой сети уже могут быть антипараллельные ребра. Остаточная сеть показывает, как и следует из названия, сколько дополнительного потока можно еще пустить по исходной сети. Если f' — поток в G_f, то определим следующий поток $f + f'$ в G:

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

Как несложно видеть, это действительно поток, т.е. он удовлетворяет всем ограничениям: ограничение пропускных способностей выполнено в силу определения c_f, а дивергенция

(26)
$$\nabla_{f+f'}(u) = \sum_{(u,v)\in E} \big(f(u,v) + f'(u,v) - f'(v,u)\big) - \sum_{(v,u)\in E} \big(f(v,u) + f'(v,u) - f'(u,v)\big) =$$
$$= \nabla_f(u) + \sum_{(u,v)\in E \vee (v,u)\in E} f'(u,v) - \sum_{(u,v)\in E \vee (v,u)\in E} f'(v,u) = \nabla_f(u) + \nabla_{f'}(u).$$

В частности, величина потока

$$|f + f'| = \nabla_{f+f'}(s) = \nabla_f(s) + \nabla_{f'}(s) = |f| + |f'|.$$

Мы будем пускать дополнительный поток вдоль путей из s в t, поэтому введем еще несколько определений.

Определение 4.42. *Пропускная способность пути* p из s в t в сети G есть $c(p) := \min_{e\in p} c(e)$. Соответственно, максимальный поток, который можно пустить вдоль этого пути, равен

$$f_p(u,v) := \begin{cases} c(p), & (u,v) \in p, \\ 0, & \text{иначе.} \end{cases}$$

Определение 4.43. Путь, ведущий из s в t в остаточной сети G_f, называется *увеличивающим путем*. Иначе говоря, путь W от s до t увеличивающий для f, если $f(e) < c(e)$ выполняется для каждого переднего ребра $e \in W$ и $f(e) > 0$ для каждого обратного ребра $e \in W$.

Определение 4.44. Рассмотрим произвольный (s,t)-*разрез* графа G, т.е. такую пару подмножеств вершин $C=(S,T)$, что $s\in S$, $t\in T$, $T=V\setminus S$. *Пропускной способностью разреза* C назовем

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

Определение 4.45. Соответственно, *минимальным* (s,t)-*разрезом* называется (s,t)-разрез с минимальной пропускной способностью.

Определение 4.46. Если же в сети задан поток f, то *потоком через разрез* C называется величина

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

Ребро e будем *насыщенным*, если $f(e)=c(e)$.

Очевидно, $f(C)\leqslant c(C)$. Кроме того, оказывается, поток через любой (s,t)-разрез одинаков и равен величине потока $|f|$:

$$(27)\quad f(C):=\sum_{u\in S,v\in T}f(u,v)-\sum_{u\in T,v\in S}f(u,v)=$$
$$=\left(\sum_{u\in S,v\in V}f(u,v)-\sum_{u\in S,v\in S}f(u,v)\right)-\left(\sum_{u\in S,v\in V}f(v,u)-\sum_{u\in S,v\in S}f(v,u)\right)=$$
$$=\sum_{u\in S}\sum_{v\in V}(f(u,v)-f(v,u))=\sum_{u\in S}\nabla_f(u)=\nabla_f(s)=|f|.$$

Таким образом, величина максимального потока не превосходит пропускной способности минимального (s,t)-разреза. Теорема Форда–Фалкерсона утверждает, что верно и обратное.

Теорема 4.47 (Ford-Fulkerson)**.** *Величина максимального потока равна пропускной способности минимального* (s,t)*-разреза.*

Нам нужны дополнительные определения.

Определение 4.48. Пусть $(v_0,v_1\dots v_n)$ — последовательность вершин неориентированного пути в ориентированном графе. *Переднее ребро* — ребро вида v_iv_{i+1}, *обратное ребро* — ребро вида v_iv_{i-1}.

Теорема 4.49 (об увеличивающем пути)**.** *Поток* f *в сети* $N=(G,c,s,t)$ *максимален тогда и только тогда для* f *нет увеличивающих путей.*

Доказательство. Пусть поток f будет максимален. допустим, что существует увеличивающий путь W. Пусть

$$d = \min_{e\ —\ \text{переднее}} (c(e) - f(e))$$

Тогда по определению увеличивающего пути $d > 0$. Теперь мы определим отображение $f' : E \to \mathbb{R}_+$:

$$f'(e) = \begin{cases} f(e) + d, & e \text{ — передное ребро в } W, \\ f(e) - d, & e \text{ — обратное ребро в } W, \\ f(e), & \text{в противном случае.} \end{cases}$$

Легко проверить, что f' является потоком в N со значением $w(f') = w(f) + d > w(f)$, что противоречит максимальности f.

Наоборот, допустим, что в N нет увеличивающих путей для f. Пусть S — множество вершин v, для которых существует увеличивающий путь из s в v (включая саму s) и положим $T = V \setminus S$. По условию (S, T) — разрез N. Отметим, что каждое ребро $e = uv$ с $e^- = u \in S$ и $e^+ = v \in T$ должно быть насыщенным — в противном случае оно может быть добавлено к увеличивающему пути s в u, противоречие. Аналогично, каждое ребро с $e^- \in T$ и $e^+ \in S$ должно быть пустым. Отсюда получаем $w(f) = c(S, T)$, так что f максимальный. □

Вывод 4.50. Пусть f — поток в сети потоков $N = (G, c, s, t)$, обозначим за S_f множество вершин, лежащих на увеличивающих путях для f из s; положим $T_f = V \setminus S_f$. Тогда f — максимальный поток титтк $t \in T_f$. В этом случае (S_f, T_f) является минимальным разрезом: $w(f) = c(S_f, T_f)$.

Теорема 4.51 (о целочисленном потоке)**.** *Пусть $N = (G, c, s, t)$ — сеть, все емкости () которой являются целыми числами. Тогда существует максимальный поток на N такой, что все значения $f(e)$ являются целыми.*

Доказательство. Установив $f_0(e) = 0$ для всех e, получим целочисленный поток f_0 на N с нулевым значением. Если этот тривиальный поток не максимален, то существует увеличивающий путь для f_0. В таком случае число d из доказательства теоремы 4.49 — положительное целое, и мы можем построить целочисленный поток f_1 со значением d. Продолжая увеличивать поток через N, мы каждый раз увеличиваем его на некоторое положительное целое; поскольку величина минимального разреза ограничивает сверху размер потока, то рано или поздно мы получим поток f, для которого не существует увеличивающих путей. По теореме 4.49 этот поток f–максимален. □

Метод Форда-Фалкерсона. Алгоритм Эдмондса–Карпа

Теперь напишем метод Форда–Фалкерсона.

Require: G — орграф, $c(\cdot)$ — пропускные способности, s, t — исток и сток сети
Ensure: f — максимальный поток в G
  $f(u, v) := 0\ \forall u, v \in V$;
  $G_f := G$, $c_f(u, v) := c(u, v)\ \forall u, v \in V$;
  while в G_f существует увеличивающий путь **do**
    найти увеличивающий путь p в G_f;
    $f := f + f_p$;
    вычислить G_f, c_f.
  end while

Заметим, что это именно метод, а не алгоритм — мы не поясняем, как именно искать увеличивающие пути. Кроме того, в таком виде даже сходимость метода не гарантируется.

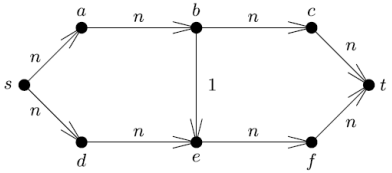
Пользуясь методом Форда-Фалкерсона, построим два алгоритма поиска максимального потока. Начнем с алгоритма Форда-Фалкерсона.

```
1:  procedure Ford-Fulkerson(G = (V, E), c(·), s, t)
2:      for e ∈ E do
3:          f(e) ← 0
4:      end for
5:      пометить s парой (−, ∞)
6:      for v ∈ V do
7:          u(v) ← false, d(v) ← ∞
8:      end for
9:      while ∃v ∈ V : v помечена u(v) = false do
10:         выбрать отмеченную v, для которой u(v) = false
11:         for e ∈ {e ∈ E : e⁻ = v} do
12:             if w = e⁺ не помечена и f(e) < c(e) then
13:                 d(w) ← min{c(e) − f(e), d(v)}, пометить w тройкой (v, +, d(w))
14:             end if
15:         end for
16:         for e ∈ {e ∈ E : e⁺ = v} do
17:             if w = e⁻ не помечено и f(e) > 0 then
18:                 d(w) ← min{f(e), d(v)}, пометить w тройкой (v, −, d(w))
19:             end if
20:         end for
21:         u(v) ← true
22:         if t помечена then
23:             взять d — последнюю компоненту метки t
24:             w ← t
25:             while w ≠ s do
```

взять v, x — первые две компоненты метки t,
if $x = +$ **then**
$f(e) \leftarrow f(e) + d$ для $e = vw$
else
$f(e) \leftarrow f(e) - d$ для $e = wv$
end if
$w \leftarrow v$
end while
удалить метки для всех вершин в $V \setminus \{s\}$
for $v \in V$ **do**
$d(v) \leftarrow \infty$, $u(v) =$ **false**
end for
end if
end while
S — множество всех помеченных вершин
return f, S, $V \setminus S$
end procedure

Из теоремы о целочисленном потоке немедленно следует следующая

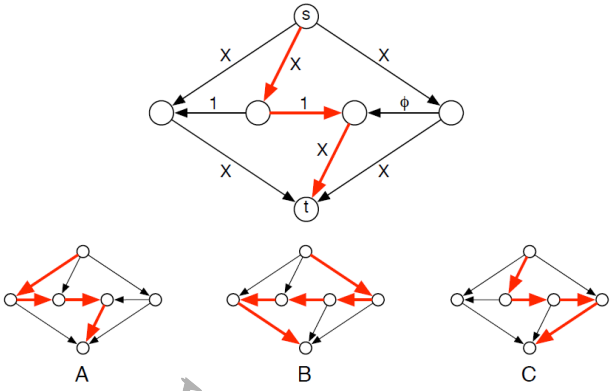
Теорема 4.52. *Пусть $N = (G, c, s, t)$ — сеть, для которой $c(e) \in \mathbb{Q}$ для любого $e \in E$. Тогда алгоритм* Ford-Fulkerson *определяет максимальный поток f и минимальный разрез (S, T). Более того, $w(f) = c(S, T)$.*



Хотя алгоритм Ford-Fulkerson корректно работает на целочисленных сетях, он не полиномиален: количество увеличиваний зависит не только от $|V|$ и $|E|$, но и от значений c. Рассмотрим пример сети на картинке слева. Если мы используем пути $s \to a \to b \to e \to f \to t$ и $s \to d \to e \to b \to c \to t$, то нам понадобится $2n$ итераций для получения максимального потока.

Если же емкости ребер могут быть иррациональны, алгоритм Ford-Fulkerson может никогда

не остановиться. Следуя работе [**zwi**], приведем пример сети, на которой алгоритм Форда-Фалкерсона никогда не сойдется к правильному ответу. На картинке справа $\phi = \frac{\sqrt{5}-1}{2}$, $X = 100$. Алгоритм ФФ сперва найдет центральный увеличиваюший путь (выше на картинке).



Остаточные емкости ребер посередине равны $1, 0, \phi$. Допустим, что в некоторый момент остаточные емкости этих равны $\phi^{k-1}, 0, \phi^k$. Мы проделываем следующие увеличения:

(1) Увеличить вдоль пути B, добавляя ϕ^k к потоку; остаточные емкости равны ϕ^{k+1}, ϕ^k, 0.

(2) Увеличить вдоль пути C, добавляя ϕ^k к потоку; остаточные емкости равны ϕ^{k+1}, 0, ϕ^k.

(3) Увеличить вдоль пути B, добавляя ϕ^{k+1} к потоку; остаточные емкости равны 0, ϕ^{k+1}, ϕ^{k+2}.

(4) Увеличить вдоль пути A, добавляя ϕ^{k+1} к потоку; остаточные емкости равны ϕ^{k+1}, 0, ϕ^{k+2}.

Тогда после $4n+1$ итераций остаточные емкости равны ϕ^{2n-2}, 0, ϕ^{2n-1}. При $n \to \infty$ получаем предельный поток

$$1 + 2\sum_{i=0}^{\infty} \phi^i = 1 + \frac{2}{1-\phi} = 4 + \sqrt{5}$$

Между тем очевидно, что максимальный поток равен $2X + 1 = 201$.

Однако легкой модернизацией можно из алгоритма Ford-Fulkerson получить полиномиальный алгоритм, находящий максимальный поток через сеть.

Теорема 4.53 (Edmonds-Karp). *Заменим шаг (10) алгоритма* Ford-Fulkerson *следующим образом:*

10': среди v, для которых $u(v) =$ **false**, *выбрать отмеченную первой*

Полученный алгоритм (который мы будем называть Edmonds-Karp*) имеет сложность $O(|V||E|^2)$.*

Доказательство. Нетрудно видеть, что поток f всегда увеличивается с помощью увеличения пути наименьшей длины, при условии, что мы заменяем шаг (10) на (10'). Пусть f_0 будет потоком значения 0, определенного изначально, а $f_1, f_2, f_3, \ldots$ — последовательность потоков, построенная впоследствии. Обозначим самую короткую длину увеличивающего пути от s до v для f_k как $x_v(k)$. Докажем, что для всех k и v выполнено

$$x_v(k+1) \geqslant x_v(k)$$

Допустим обратное, а именно, что неравенство нарушается для некоторой пары (v, k); можно рассмотреть пару (v, k) с минимальным $x_v(k+1)$. Рассмотрим последнее ребро e на кратчайшем пути увеличения от s до v относительно $f_k + 1$. Допустим сначала, что e является передним ребром, поэтому $e = uv$ для некоторой вершины u; обратите внимание, что для этого требуется $f_k + 1(e) < c(e)$. Таким образом, $x_v(k+1) = x_u(k+1) + 1$, так что $x_u(k+1) \geqslant x_u(k)$ в силу нашего выбора v. Следовательно, $x_v(k+1) \geqslant x_u(k) + 1$. С другой стороны, $f_k(e) = c(e)$, иначе $x_v(k) \leqslant x_u(k)+1$ и $x_v(k+1) \geqslant x_v(k)$, что противоречит выбору v. Следовательно, e было обратным ребром в момент, когда f_k заменился на f_k+1. Поскольку мы использовали путь кратчайшей длины, мы заключаем $x_u(k) = x_v(k) + 1$ и, следовательно, $x_v(k+1) \geqslant x_v(k) + 2$, противоречие. Случай, когда uv — обратное, разбирается аналогично. Более того, похожие аргументы также приводят к неравенству

$$y_v(k+1) \geqslant y_v(k)$$

для всех k и v. Здесь $y_v(k)$ — длина кратчайшего увеличивающего пути от v до t для f_k.

При увеличении потока в увеличивающем пути найдется как минимум одино *критическое* ребро: поток через этот ребро либо увеличивается до максимально возможного [емкости ребра], либо уменьшается до нуля. Пусть $e = uv$ — критическое ребро в увеличивающем пути для f_k; этот путь состоит из

$$x_v(k) + y_v(k) = x_u(k) + y_u(k)$$

ребер. Если используется в следующий раз в каком-либо увеличивающем пути (для некоторого f_h), оно должно быть использовано в обратном направлении: если было передним ребром для f_k, оно должно быть обратным для f_h, и наоборот.

Допустим, что e было передним ребром для f_k. Тогда

$$x_v(k) = x_u(k) + 1, \quad x_u(h) = x_v(h) + 1.$$

Согласно неравенствам, доказанным выше, получаем

$$x_v(h) \geqslant x_v(k), \quad y_u(h) \geqslant y_u(k).$$

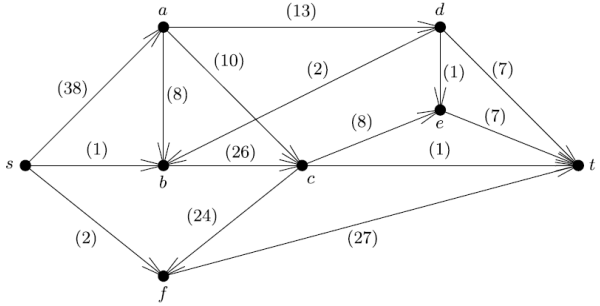
Следовательно, мы получаем

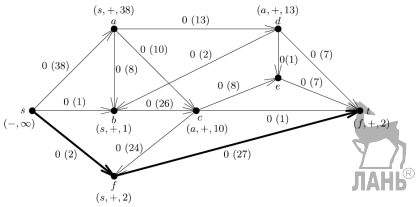
$$x_u(h) + y_u(h) = x_v(h) + 1 + y_u(h) \geqslant x_v(k) + 1 + y_u(k) = x_u(k) + y_u(k) + 2.$$

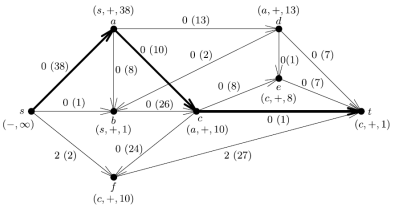
Таким образом, увеличивающий путь относительно f_h содержит по крайней мере на два ребра больше, чем увеличивающий путь по отношению к f_k. Это также верно, если e — обратное ребро для f_h.

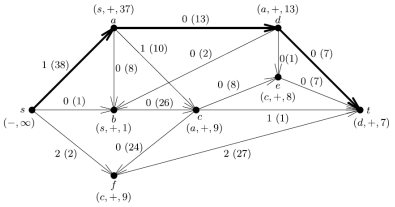
Очевидно, что никакой путь увеличивающих не может содержать больше чем $|V|-1$ ребер. Следовательно, каждое ребро может быть критическим не более $\frac{|V|+1}{2}$ раза, и, таким образом, поток может быть изменен не более $O(|V||E|)$ раз. [В частности, отсюда следует, что алгоритм Эдмондса-Карпа останавливается даже в случае иррациональных емкостей.] Каждая итерация — поиск увеличивающего пути и обновление потока — выполняет только $O(|E|)$ шагов, поскольку каждое ребро обрабатывается не более трех раз: дважды в процессе маркировки и один раз при изменении потока. Отсюда следует, что сложность алгоритма Эдмондса-Карпа равна $O(|V||E|^2)$. □

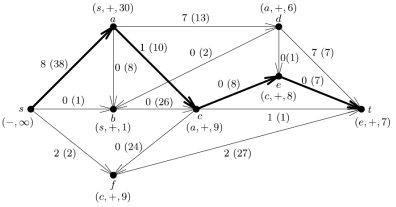
Пример 4.54. Мы используем алгоритм Эдмондса-Карпа, чтобы определить максимальный поток и минимальный разрез в сети на картинке ниже [итерации – на следующей странице]. Емкости ребер указаны в скобках; числа без скобок — соответствующие значения потока. Увеличивающий путь, используемый для построения следующего потока, выделен жирным шрифтом.

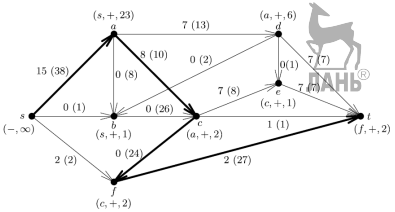


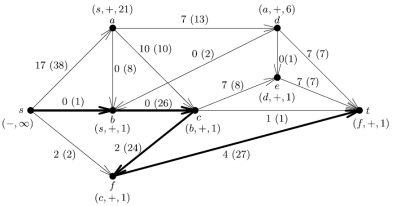


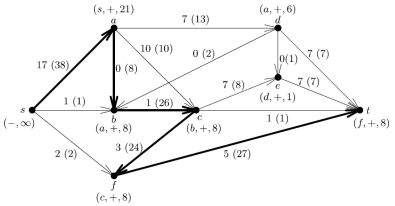


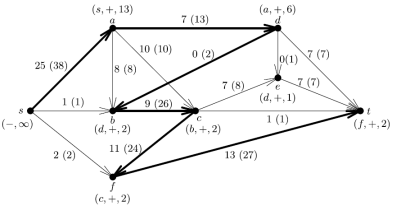

(s, +, 37)
a
0 (13)
(a, +, 13)
d
1 (38)
1 (10)
0 (2)
0 (7)
0 (8)
0(1)
e
0 (8)
0 (7)
s
0 (1)
0 (26)
(c, +, 8)
(−, ∞)
b
(s, +, 1)
c
(a, +, 9)
1 (1)
t
(d, +, 7)
2 (2)
0 (24)
2 (27)
f
(c, +, 9)

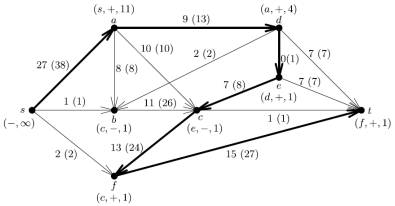

(s, +, 30)
a
7 (13)
(a, +, 6)
d
8 (38)
1 (10)
0 (2)
7 (7)
0 (8)
0(1)
e
0 (8)
0 (7)
s
0 (1)
0 (26)
(c, +, 8)
(−, ∞)
b
(s, +, 1)
c
(a, +, 9)
1 (1)
t
(e, +, 7)
2 (2)
0 (24)
2 (27)
f
(c, +, 9)

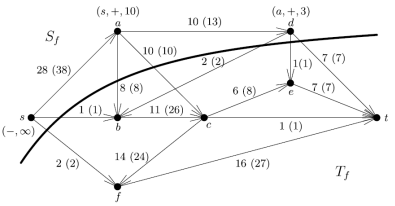

(s, +, 23)
a
7 (13)
(a, +, 6)
d
15 (38)
8 (10)
0 (2)
7 (7)
0 (8)
0(1)
e
7 (8)
7 (7)
s
0 (1)
0 (26)
(c, +, 1)
(−, ∞)
b
(s, +, 1)
c
(a, +, 2)
1 (1)
t
(f, +, 2)
2 (2)
0 (24)
2 (27)
f
(c, +, 2)


(s, +, 21)
a
7 (13)
(a, +, 6)
d
17 (38)
10 (10)
0 (2)
7 (7)
0 (8)
0(1)
e
7 (8)
7 (7)
s
0 (1)
0 (26)
(d, +, 1)
(−, ∞)
b
(s, +, 1)
c
(b, +, 1)
1 (1)
t
(f, +, 1)
2 (2)
2 (24)
4 (27)
f
(c, +, 1)


(s, +, 21)
a
7 (13)
(a, +, 6)
d
17 (38)
10 (10)
0 (2)
7 (7)
0 (8)
0(1)
e
7 (8)
7 (7)
s
1 (1)
1 (26)
(d, +, 1)
(−, ∞)
b
(a, +, 8)
c
(b, +, 8)
1 (1)
t
(f, +, 8)
2 (2)
3 (24)
5 (27)
f
(c, +, 8)


(s, +, 13)
a
7 (13)
(a, +, 6)
d
25 (38)
10 (10)
0 (2)
7 (7)
8 (8)
0(1)
e
7 (8)
7 (7)
s
1 (1)
9 (26)
(d, +, 1)
(−, ∞)
b
(d, +, 2)
c
(b, +, 2)
1 (1)
t
(f, +, 2)
2 (2)
11 (24)
13 (27)
f
(c, +, 2)


(s, +, 11)
a
9 (13)
(a, +, 4)
d
27 (38)
10 (10)
2 (2)
7 (7)
8 (8)
0(1)
e
7 (8)
7 (7)
s
1 (1)
11 (26)
(d, +, 1)
(−, ∞)
b
(c, −, 1)
c
(e, −, 1)
1 (1)
t
(f, +, 1)
2 (2)
13 (24)
15 (27)
f
(c, +, 1)


(s, +, 10)
a
10 (13)
(a, +, 3)
d
S_f
10 (10)
28 (38)
2 (2)
7 (7)
8 (8)
1(1)
e
6 (8)
7 (7)
s
1 (1)
11 (26)
(−, ∞)
b
c
1 (1)
t
2 (2)
14 (24)
16 (27)
T_f
f

Метод проталкивания предпотока. Алгоритм Тарьяна-Голдберга

Теперь мы представим другой алгоритм поиска максимального потока, разработанный Голдбергом и Тарьяном. Вместо итеративного увеличения потока путем нахождения новых увеличивающих путей этот алгоритм работает с так называемыми предпотоками. Основная идея алгоритма состоит в том, чтобы протолкнуть поток из вершин с избытком к стоку t, используя пути от s до t, которые не обязательно являются кратчайшими, но только текущие оценки для таких путей. Конечно, может случиться так, что лишний поток не может быть вытолкнут из некоторой вершины v; в этом случае его необходимо отправить обратно к источнику по подходящему пути. Выбор всех этих путей контролируется определенной функцией маркировки на вершинах.

Определение 4.55. Пусть $(G = (V, E), c, s, t)$ — сеть. *Предпоток* f — функция $f : V \times V \to \mathbb{R}$, удовлетворяющая свойствам:

- $f(v, w) \leqslant c(v, w)$ для всех $(v, w) \in V \times V$
- $f(v, w) = -f(w, v)$ для всех $(v, w) \in V \times V$
- $U \in Vf(u, v) \geqslant 0$ для всех $v \in V \setminus \{s\}$

Последнее условие означает, что количество потока, входящего в вершину $v \neq s$, не меньше, но может быть больше потока, исходящего из v.

Определение 4.56. Величина

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

называется *избытком* предпотока f в v.

Нетрудно видеть, что для потока f избыток $e(v) = 0$ для любой вершины v.

Определение 4.57. Пусть на сети $(G = (V, E), c, s, t)$ задан предпоток f. Определим *остаточную емкость* $r_f : V \times V \to \mathbb{R}$:

$$r_f(v, w) = c(v, w) - f(v, w).$$

Если ребро vw удовлетворяет $r_f(v, w) > 0$, мы можем пустить некоторый поток через него; такое ребро называется *остаточным ребром*.

Остаточное ребро либо vw является передним, но еще не насыщенным: $0 \leqslant f(v, w) < c(v, w)$, либо обратным, то есть $0 < f(w, v) \leqslant c(w, v)$.

Определение 4.58. *Остаточный граф* предпотока f определяется как

$$G_f = (V, E_f), \quad E_f = \{vw \in E \mid r_f(v, w) > 0\}$$

Определение 4.59. Отображение $d : V \to \mathbb{N}_0 \cup \{\infty\}$ называется *корректной маркировкой* для предпотока f, если выполняются следующие два условия:

- $d(s) = |V|$ и $d(t) = 0$;
- $d(v) \leqslant d(w) + 1$ для всех $vw \in E_f$.

Будем также называть вершину v *активной*, если $v \neq s, t$, $(v) > 0$ и $d(v) < \infty$.

```
1: procedure Goldgberg(G = (V, E), c : V × V → ℝ, s, t)
2:     for (v, w) ∈ (V \ {s}) × (V \ {s}) do
3:         f(v, w) ← 0, r_f(v, w) ← c(v, w)
4:     end for
5:     d(s) ← |V|
6:     for v ∈ V \ {s} do
7:         f(s, v) ← c(s, v), r_f(s, v) ← 0
8:         f(v, s) ← −c(s, v), r_f(v, s) ← c(v, s) + c(s, v)
9:         d(v) ← 0
10:        e(v) ← c(s, v)
11:    end for
12:    while ∃v ≠ s, t, (v) > 0 и d(v) < ∞ do
13:        выполнить допустимую операцию для v
14:    end while
15:    return f
16: end procedure
```

В качестве допустимых операций могут использоваться Push и Label [при условии, что она допустимы]:

```
1: procedure Push(G = (V, E), c : V × V → ℝ, s, t, f, v, w)
2:     δ ← min(e(v), r_f(v, w))
3:     f(v, w) ← f(v, w) + δ, f(w, v) ← f(w, v) − δ
4:     r_f(v, w) ← r_f(v, w) − δ, r_f(w, v) ← r_f(w, v) + δ
5:     e(v) ← e(v) − δ, e(w) ← e(w) + δ
6:     return f
7: end procedure
```

```
1: procedure Relabel(G = (V, E), c : V × V → ℝ, s, t, f, v, w)
2:     d(v) ← min{d(w) + 1 | r_f(v, w) > 0}
3:     return f
4: end procedure
```

Процедура Push допустима при условии, что v активна,

$$r_f(v, w) > 0, \quad d(v) = d(w) + 1$$

Процедура Relabel допустима при условии, что v активна и

$$\forall w \in V \quad r_f(v, w) > 0 \to d(v) \leqslant d(w)$$

Мы покажем, что алгоритм Тарьяна-Голдберга находит максимальный поток в сети N за конечное число шагов независимо от того, в каком порядке мы выбираем активные вершины и допустимые операции. Это разительно отличает метод Тарьяна-Голдберга от метода Форда-Фалкерсона, сложность которого существенно зависит от порядка выполнения операций.

Лемма 4.60. *Пусть f — предпоток на N, d — корректная маркировка на V относительно к f, и v активная вершина. Тогда либо* Push, *либо операция* Relabel *допустимы для v.*

Доказательство. Поскольку d допустима, то $d(v) \leqslant d(w)+1$ для всех w с $r_f(v,w) > 0$. Если $\textsc{Push}(N, f, v, w)$ не допустимо для любого w, то $d(v) \leqslant d(w)$ для всех w с $r_f(v,w) > 0$, так как d принимает только целые значения. Но тогда Relabel допустима. □

Лемма 4.61. *Во время выполнения алгоритма* Goldberg f *всегда является препотоком, а d — корректной маркировкой для f.*

Доказательство. Мы уже используем индукцию по числу выполненных допустимых операций k. Утверждение справедливо для базиса индукции $k = 0$: очевидно, f инициализируется как предпоток на этапах (7) и (8); маркировка d, определенная на шагах (5) и (9), корректна для f, поскольку $d(v) = 0$ для $v \neq s$ и все ребра sv насыщены на шаге (7). Кроме того, остаточные емкости и избыток потока инициализированы правильно на шагах (7), (8) и (10).

Доказывая шаг индукции, допустим, что утверждение выполняется после k операций. Положим сначала, что следующая операция - это $\textsc{Push}(v, w)$. Легко проверить, что f остается предпотоком и что остаточные мощности и превышения потока обновляются правильно. Пометки не изменяются, а vw и wv — единственные ребра, на которых изменился f. Следовательно, нам нужно разобраться с этими ребрами, чтобы показать, что d — корректная пометка. По определению $vw \in E_f$ перед Push. Теперь vw может быть удалено из остаточного графа G_f (если оно насыщено после применения Push), но тогда маркировка останется корректной.

Теперь рассмотрим антипараллельное ребро wv. Если это ребро уже есть в G_f, доказывать нечего. Допустим, что wv добавляется к G_f Push; опять же, d остается корректной, так как условия допустимости для $\textsc{Push}(v, w)$ требуют $d(w) = d(v) - 1$.

Осталось рассмотреть случай, когда следующей операцией является $\textsc{Relabel}(v)$. Тогда требование допустимости $d(v) \leqslant d(w)$ для всех вершин w с $r_f(v,w) > 0$. При увеличении $d(v)$ до минимума всех $d(w)+1$, условие $d(v) \leqslant d(w)+1$ выполняется для всех w с $r_f(v,w) > 0$ после этого изменения. Все остальные метки остаются неизменными, так что новая метка d остается корректной для f. □

Лемма 4.62. *Пусть f — предпоток на N, пусть d — допустимая маркировка для f, а v и w — две вершины в N такие, что w достижима из v в остаточном графе G_f. Тогда*

$$d(v) - d(w) \leqslant d(v, w).$$

Доказательство. Пусть

$$P : v = v_0 - v_1 - \cdots - v_k = w$$

кратчайший путь из v в w в G_f. Поскольку d — корректная маркировка, для всех $i = 0, \ldots, k-1$ получаем $d(v_i) \leqslant d(v_{i+1}) + 1$. Поскольку P имеет длину $k = d(v, w)$, мы получаем требуемое неравенство. □

Утверждение 4.63. *Пусть f — предпоток на N, а d — корректная маркировка для f. Тогда t не достижима из s в остаточном графе G_f.*

Доказательство. Допустим обратное. Тогда $d(s) \leqslant d(t) + d(s, t)$ по лемме, доказанной выше. Но это противоречит $d(s) = |V|$, $d(t) = 0$ и $d(s, t) \leqslant |V| - 1$. □

Теорема 4.64. *Если алгоритм* Goldberg *заканчивает работу, а все метки принимают конечное значение, то построенный предпоток f на самом деле максимальный поток на N.*

Доказательство. По лемме 4.60 алгоритм может завершиться титтк больше нет активных вершин. Поскольку все метки конечны по предположению, $e(v) = 0$ верно для каждой вершины $v \neq s, t$; таким образом, итоговый предпоток действительно является потоком на N. Из утверждения 4.60 нет пути из s в t в G_f, поэтому не существует увеличивающего пути от s к t относительно f. По теореме об увеличивающем пути это эквивалентно максимальности потока. □

Лемма 4.65. *Пусть f — предпоток на N. Если v — вершина, для которой $e(v) > 0$, то s достижима из v в G_f.*

Доказательство. Обозначим множество вершин, достижимых из v в G_f [по ориентированному пути] на S и положим $T = V \setminus S$. Тогда $f(u, w) \leqslant 0$ для всех вершин u, w с $u \in T$ и $w \in S$, так как

$$0 = r_f(w, u) = c(w, u) - f(w, u) \geqslant 0 + f(u, w).$$

В силу антисимметричности f получим

$$\sum_{w \in S} e(w) = \sum_{u \in V, w \in S} f(u, w) = \sum_{u \in T, w \in S} f(u, w) + \sum_{u, w \in S} f(u, w) = \sum_{u \in T, w \in S} f(u, w) \leqslant 0$$

По определению предпотока $e(w) \geqslant 0$ для всех $w \neq s$. Но $(v) > 0$, и, поскольку $\sum_{w \in S} e(w) \leqslant 0$ следует $s \in S$. □

Лемма 4.66. *В алгоритме Тарьяна-Голдберга* $d(v) \leqslant 2|V| - 1$ *для всех* $v \in V$.

Доказательство. Очевидно, что утверждение выполняется после инициализации во время шагов (1) - (11). Метка $d(v)$ вершины v может быть изменена только операцией Relabel(v), такая операция допустима титтк v активна. В частности $v \neq s, t$, так что утверждение тривиально для s и t, более того, $e(v) > 0$. Согласно лемме 4.65 s доступно из v в остаточном графе G_f. Теперь лемма 4.62 дает

$$d(v) \leqslant d(s) + d(v, s) \leqslant d(s) + |V| - 1 = 2|V| - 1.$$

□

Лемма 4.67. *Общее количество вызовов* Relabel *не более* $(2|V| - 1)(|V| - 2) < 2|V|^2$.

Доказательство. Каждый Relabel(v) увеличивает $d(v)$. Поскольку $d(v)$ ограничено $2|V|-1$ на протяжении всего алгоритма (см. лемму 4.65). Таким образом, при выполнении алгоритма Тарьяна-Голдберга не более $2|V| - 1$ Relabel-операции выполняются для любой заданной вершины $v \neq s, t$. □

Для оценки количества вызова Push нам нужно различать два случая: Push(v, w) будем называть *насыщающим* Push, если после его исполнения $r_f(v, w) = 0$ (то есть для $\delta = r_f(v, w)$ на первом шаге Push) и *ненасыщающим* Push в противном случае.

Лемма 4.68. *При выполнении алгоритма Тарьяна-Голдберга общее количество вызовов насыщающих* Push *не более* $|V||E|$.

Доказательство. По определению для любого Push(v, w) требуется $vw \in E_f$ и $d(v) = d(w) + 1$. Если Push насыщает ребро, дальнейшее Push(v, w) может произойти только после Push (w, v), так как $r_f(v, w) = 0$ после насыщения Push(v, w). Push(w, v) не допустимо [в введенном нами смысле] до того, как метки были изменены таким образом, что $d(w) = d(v) + 1$; следовательно, $d(w)$ должна была увеличена хотя бы на 2 перед Push(w, v). Точно так же никакой Push(v, w) не может стать допустимым прежде, чем $d(v)$ также увеличится на не менее 2. В частности $d(v) + d(w)$ должно увеличиться как минимум на 4 между любыми двумя последовательными применениями Push(v, w).

С другой стороны, как только выполняется первый Push(v, w) или Push(w, v), выполняется

$$d(v) + d(w) \geqslant 1$$

. Кроме того, по лемме 4.65 $d(v), d(w) \leqslant 2|V| - 1$ на протяжении работы алгоритма; следовательно, во время последней Push-операции с участием v и w выполняется

$$d(v) + d(w) \leqslant 4|V| - 2$$

. Таким образом, насыщающих Push(v, w) -операций не более $|V| - 1$, так что общее количество насыщающих Push-операций не превышает $(|V| - 1)|E|$. □

Лемма 4.69. *При выполнении алгоритма Тарьяна-Голдберга выполняется не более* $2|V|^2|E|$ *ненасыщающим* Push*-операции.*

Доказательство. Введем потенциал

$$\Phi = \sum_{v \text{активна}} d(v)$$

и рассмотрим его изменение в течение работы алгоритма Тарьяна-Голдберга. После инициализации $\Phi = 0$, если алгоритм останавливается, снова $\Phi = 0$. Любой ненасыщенный Push(v, w) уменьшает Φ хотя бы на один: поскольку $r_f(v, w) > e(v)$, вершина v становится неактивной, Φ уменьшился на $d(v)$; даже если вершина w стала активной из-за Push, Φ снова увеличивается только на $d(w) = d(v) - 1$ в силу условий допустимости. Точно так же любой насыщающий Push(v, w) увеличивает Φ на не более чем $2|V| - 1$, поскольку метка вершины w [которая может снова стать активной благодаря Push] удовлетворяет $d(w) \leqslant 2|V| - 1$ по лемме 4.65.

По 4.68 насыщающие Push-операции увеличивают Φ максимум на $(2|V| - 1)|V||E|$ в целом, по лемме 4.67 Relabel-операции увеличивают Φ на большинство $(2|V|-1)(|V|-2)$ единиц. Отсюда следует верхняя оценка для числа ненасыщенных Push-операций: $(2|V|-1)(|V||E| + |V| - 2)$. □

Теорема 4.70. *Алгоритм Тарьяна-Голдберга заканчивается после того, как допустимо не более* $O(|V|^2|E|)$ *операции (с максимальным потоком).*

Доказательство. В силу лемм 4.67, 4.68, 4.69 число допустимых операций ограничено, следовательно, число итераций последнего цикла **while** ограничено. Поскольку первые два цикла **for** совершают конечное число операций, алгоритм Тарьяна-Голдберга останавливается на любой сети. Более того, таким же образом проверяется оценка на сложность. Из теоремы 4.64 следует, что построенный предпоток в самом деле является максимальным потоком. □

0-1 потоки

Теперь мы сосредоточимся на частном случае *0-1 потоков*, то есть потоков, которые на ребрах равны либо 0, либо 1.

Определение 4.71. *Блокирующий поток* — поток, для которого любой увеличивающий путь содержит заднее ребро.

Мы заинтересованы в построении блокирующего 0-1 потока — это оказывается полезным в некоторых комбинаторных применениях.

Определение 4.72. *0-1-сеть* $N = (G, c, s, t)$ — сеть, в которой $c(e) = 1$ для всех ребер e.

Лемма 4.73. *Пусть $N = (G, c, s, t)$ — 0-1-сеть. Тогда*

$$d(s,t) \leqslant 2\frac{|V|}{\sqrt{M}},$$

где M — максимальное значение потока на N. Более того, если для каждой вершины $v \in V \setminus \{s,t\}$ верно по крайней мере одно из двух условий $\operatorname{indeg}(v) \leqslant 1$ *и* $\operatorname{outdeg}(v) \leqslant 1$, *то*

$$d(s,t) \leqslant 1 + \frac{|V|}{M}$$

Доказательство. Пусть $D = d(s,t)$, и пусть V_i будет множеством всех вершин $v \in V$ с $d(s,v) = i$ для $i = 0, \ldots, D$. Вершины с большим расстоянием до s могут существовать, но не важны в наших рассуждениях; для упрощения обозначений можно предположить, что таких вершин нет. Затем

$$(S_i, T_i) = (V_0 \cup V_1 \cup \ldots \cup V_i, V_{i+1} \cup \ldots \cup V_D)$$

является разрезом для каждого $i < D$. Поскольку каждое ребро e, для которого $e^- \in S_i$ и $e^+ \in T_i$, удовлетворяет условиям $e^- \in V_i$ и $e^+ \in V_{i+1}$, а N является 0-1-сетью, из мы получаем

$$M \leqslant c(S_i, T_i) \leqslant |V_i| \times |V_i + 1|, \quad i = 0, \ldots, D-1.$$

Таким образом, хотя бы одно из двух значений $|V_i|$ или $|V_{i+1}|$ не может быть меньше $\sqrt{}$. Следовательно, по крайней мере половина слоев V_i содержит $\sqrt{M}$ или более вершин. Это дает

$$D\frac{\sqrt{M}}{2} \leqslant |V_0| + \ldots + |V_D| \leqslant |V|,$$

и, следовательно,

$$d(s,t) \leqslant 2\frac{|V|}{\sqrt{M}}.$$

Теперь предположим, что N удовлетворяет дополнительному условию, указанному в утверждении. Тогда поток через любую данную вершину не может превышать единицы, и мы получаем более сильное неравенство

$$M \leqslant |V_i|, \quad i = 0, \ldots, D-1.$$

Тогда

$$M(D-1) \leqslant |V_1| + \ldots + |V_D - 1| \leqslant |V|.$$

□

1: **procedure** Block01Flow($G = (V, E)$, $c : V \times V \to \{0, 1\}$, s, t)
2: $L \leftarrow \varnothing$

```
    for v ∈ V do
        ind(v) ← 0
    end for
    for e ∈ E do
        g(e) ← 0, ind(e⁺) ← ind(e⁺) + 1
    end for
    while ind(t) ≠ 0 do
        v ← t
        for i ∈ {d ... 1} do
            выбрать ребро e = uv and удалить e из E
            ind(v) ← ind(v) − 1, g(e) ← 1
            if ind(v) = 0 then
                добавить v в L
                while L ≠ ∅ do
                    удалить the first vertex w из L
                    for  do{e ∈ E | e⁻ = w}
                        удалить e из E, ind(e⁺) ← ind(e⁺) − 1
                        if ind(e⁺) = 0 then
                            добавить e⁺ в L
                        end if
                    end for
                end while
            end if
            v ← u
        end for
    end while
    return g
end procedure
```

Очевидно, что каждое ребро e обрабатывается, а затем удаляется не более одного раза во время повторения большого цикла в этой процедуре, так что сложность Block01Flow есть $O(|E|)$.

Теорема 4.74. *Пусть $N = (G, c, s, t)$ — 0-1-сеть. Тогда алгоритм* Block01Flow *может быть использован для вычисления максимального 0-1-потока на N со сложностью $O(|V|^{\frac{2}{3}}|E|)$.*

Доказательство. В силу линейности Block01Flow по $|E|$ достаточно показать, что блокируюший поток нужно вычислять $O(|V|^{\frac{2}{3}})$ раз. Пусть максимальное значение потока на N. Поскольку значение потока увеличивается как минимум на 1 во время каждой фазы алгоритма, утверждение тривиально в случае $M \leqslant |V|^{\frac{2}{3}}$; положим, что

$M > |V|^{\frac{2}{3}}$. Рассмотрим фазу [однозначно определенную], на которой значение потока увеличивается до значения, превышающего $M - |V|^{\frac{2}{3}}$, и пусть f будет потоком 0-1 на N, который алгоритм построил в непосредственно предшествующей фазе. Тогда $w(f) \leqslant M - |V|^{\frac{2}{3}}$, и, следовательно, значение M' максимального потока на $N'(f)$ удовлетворяет $' = M - w(f) \geqslant |V|^{\frac{2}{3}}$. Очевидно, что N' также 0-1-сеть, так что расстояние $d(s,t)$ от s до t в $N'(f)$ удовлетворяет неравенству

$$d(s,t) \leqslant 2|V|\sqrt{M}? \leqslant 2|V|^{\frac{2}{3}}$$

в силу леммы 4.73. Поскольку расстояние между s и t в соответствующей вспомогательной сети увеличивается в каждой фазе, и, следовательно, конструкция f может занять не более $2|V|^{\frac{2}{3}}$ фаз. В силу выбора f мы достигаем значения потока, превышающего $M-|V|^{\frac{2}{3}}$ на следующем этапе, так что для увеличения значения потока до M необходимо не более $|V|^{\frac{2}{3}}$ фаз, пока не достигнет M. □

Анлогичным способом можно получить и следующую теорему.

Теорема 4.75. *Пусть $N = (G, c, s, t)$ - 0-1-сеть. Если каждая вершина $v \neq s, t$ удовлетворяет хотя бы одному из двух условий* $\operatorname{indeg}(v) \leqslant 1$ *и* $\operatorname{outdeg}(v) \leqslant 1$, *то тогда алгоритм Диница можно использовать для определения максимального 0-1-потока на N со сложностью $O(|V|^{\frac{1}{2}}|E|)$.*

Теперь рассмотрим одно комбинаторное применение поиска 0-1-потока. Пусть дан двудольный граф $G = (S \sqcup T, E)$, мы ищем паросочетание максимальной мощность в G. Покажем, что это проблема эквивалентна нахождению максимального потока 0-1 в соответствующей сети потока. Определим орграф $H = (S \cup T \cup \{s,t\}, E \cup E_s \cup E_t)$, где

$$E_s = \{sx \min x \in S\}, E_t = \{yt \mid y \in T\}.$$

Тем самым мы определяем 0-1-сеть N. Отметим, что ребра x_iy_i $(i = 1, ..., k)$ произвольного совпадения мощности k для G индуцирует поток значения k на N: положим $f(e) = 1$ для всех ребер $e = sx_i, e = x_iy_i$ и $e = y_it$ для $i = 1, ..., k$). И наоборот, 0-1-поток величины k дает соответствие, состоящее из k ребер: надо выбрать k ребер типа xy, которые на самом деле переносят ненулевой поток.

Только что описанное преобразование проблемы приводит к эффективному алгоритму определения максимальных паросочетания в двудольных графах: как мы видели выше, максимальный 0-1-поток на N может быть определен со сложностью $O(|V|^{\frac{1}{2}}|E|)$. Таким образом, сложность не превышает $O(|V|^{\frac{5}{2}})$.

Вершинная и реберная связности

В этой секции графы по умолчанию связны.

Определение 4.76. Два пути в графе будем *реберно непересекающимися*, если у них нет ни одного общего ребра, и *вершинно непересекающимися*, если у них нет ни одной общей вершины.

Определение 4.77. *Реберный разделитель* вершин u и v в графе G — множество ребер, удаление которых нарушит линейную связность u и v. *Вершинный разделитель* вершин u и v в графе G — множество вершин, удаление которых [вместе с инцидентными ребрами] нарушит линейную связность u и v.

Определение 4.78. Пусть G — граф, а $u \neq v$ — его вершины. Определим $\lambda(u, v)$ как минимальное число ребер, удаление которых нарушит линейную связность u и v. Также определим $\kappa(u, v)$ как максимальное число реберно непересекающихся путей u и v.

Определение 4.79. *Вершинная связность* $\lambda(G)$ графа G определяется как

$$\kappa(G) = \min_{u,v \in G} \kappa(u, v)$$

Граф G *m-вершинно связный*, если $\kappa(G) \geqslant m$.

Определение 4.80. *Реберная связность* $\lambda(G)$ графа G определяется как

$$\lambda(G) = \min_{u,v \in G} \lambda(u, v)$$

Граф G *m-реберно связный*, если $\lambda(G) \geqslant m$.

Теорема 4.81. *Пусть G — граф или орграф. Тогда $\kappa(G) \leqslant \lambda(G) \leqslant \delta(G)$, где $\delta(G)$ — минимальная степень вершины в графе.*

Доказательство. Достаточно рассмотреть только неориентированный случай; ориентированный случай разбирается аналогично. Возьмем вершину v с $\deg v = \delta(G)$. Удаление всех ребер, инцидентных с v, дает несвязный граф, так что $\lambda(G) \leqslant \delta(G)$. Если $\lambda(G) = 1$, G содержит мост $e = uv$. Тогда G не может быть 2-связным, потому что удаление u из G дает либо K_1, либо несвязный граф. Если $\lambda(G) = k \geqslant 2$, то пусть $\{e_1, \ldots e_k\}$ — множество ребер, которые надо удалить, тогда удаление $k-1$ ребра $_2, \ldots, e_k$ в G приводит к графу H, содержащему мост $e_1 = uv$. Поэтому если мы удалим из G одну из конечных вершин каждого из e_i, отличных от u и v (для $i = 2, \ldots, k$), мы получаем либо несвязный граф, либо граф, где e_1 является мостом (так граф без этого ребра несвязен). В любом случае $\kappa(G) \leqslant k = \lambda(G)$. □

Теорема 4.82 (Menger). *Пусть G — граф или орграф, s и t — две вершины графа G. Тогда максимальное число реберно непересекающиеся путей от s до t равно $\lambda(u, v)$.*

Доказательство. Положим G орграфом. Можно предположить, что t достижима из s, в противном случае утверждение тривиально. Рассмотрим сеть N на G, где каждое

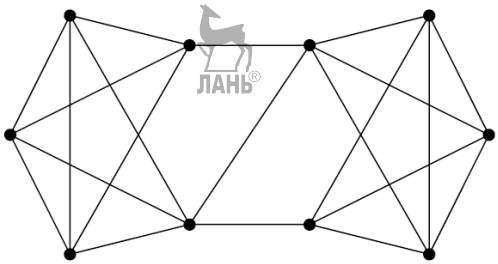


Рис. 16. Для этого графа $\kappa(G) = 2$, $\lambda(G) = 3$, $\delta(G) = 4$.

ребро имеет емкость $c(e) = 1$, пусть k — максимальное число реберно непересекающихся ориентированных путей от s до t. Очевидно, что любая система таких путей дает 0-1-поток f значения k [$f(e) = 1$, если e встречается в одном из путей, и $f(e) = 0$ в противном случае]. Следовательно, максимальное значение потока на N — некоторое целое число $k' \geqslant k$. В силу теоремы Форда-Фалкерсона мы можем построить максимальный целочисленный поток, используя увеличивающие пути пропускной способности 1. Обратите внимание, что эти пути не обязательно ориентированы в одну сторону — могут возникнуть обратные ребра. Тем не менее, всегда можно найти k увеличивающих путей без обратных ребер: предположим, что $e = uv$ — обратное ребро, встречающееся на пути W; тогда должен существовать путь W', который был построен до W и содержит e в качестве переднего ребра. Таким образом, пути W и W' имеют вид

$$W = s \overset{W_1}{-} v - u \overset{e}{-} t, \quad W' = s \overset{W_1'}{-} u - v \overset{W_2'}{-} t$$

Теперь мы можем заменить пути W и W' на пути W_1W_2' и $W_1'W_2$, устранив таким образом ребро . Мы можем предположить, что является обратным ребром для W, тогда W_1, W_2' и W_1' содержит только передние ребра (тогда какW_2 все еще может содержать обратные ребра). Повторяя эту конструкцию, получаем k увеличивающих путей из s в t, которые состоят из только передних ребер. Любые два увеличивающих пути, состоящие из передних ребер, должны быть ребрно непересекающимся, так как все ребра имеют единичную вместимость. Отсюда $k' \leqslant k$, и, следовательно, $k' = k$.

Таким образом, максимальное число реберно непересекающихся путей из s в t в G равно максимальному значению потока на N и, следовательно, по теореме Форда-Фалкерсона, равно мощности минимального разреза в N. Осталось показать, что минимальный разрез в N имеет мощность $\lambda(u, v)$. Очевидно, что любой разрез (S, T) в N

задает множество ребер, удаление которых нарушает связность s и t:

$$A = \{e \in E \mid e^- \in S, e^+ \in T\}$$

И наоборот, пусть A — минимальное множество ребер, разделяющее для s и t. Пусть S_A — множество вершин v, достижимых из s ориентированным путем, не содержащим ребер из A; положим $T_A = V \setminus S_A$. Тогда (S_A, T_A) — разрез в N. Ясно, что каждое ребро e, для которого $e^- \in S_A$ и $e^+ \in T_A$ должно содержаться в A. Так как A минимальное, A состоит именно из этих ребер и поэтому индуцируется разрезом.

Теперь сведем случай неориентированного графа к ориентированному. Построим по графу G ориентированный граф $\overrightarrow{G}$, ориентировав каждое ребро G в обе стороны (по ребру uv графа G построим ребра $u \to v$ и $v \to u$). Очевидно, что система реберно непересекающихся путей в G индуцирует соответствующую систему реберно непересекающихся ориентированных путей в $\overrightarrow{G}$. Обратное также верно при условии, что реберно непересекающиеся ориентированные пути в $\overrightarrow{G}$ не содержат пары антипараллельных ребер. Но такие пары ребер можно устранить способом, аналогичны устранению обратных ребер в первой части доказательства.

Теперь пусть k — максимальное число реберных непересекающихся ориентированных ребер в G и, следовательно, также в $\overrightarrow{G}$. Тогда существует реберный разделитель $\overrightarrow{A}$ в $\overrightarrow{G}$ мощности k [удаление ребер $\overrightarrow{A}$ нарушит связность s и t]; соответствующее ему множество ребер в G является разделителем для s и t в G мощности не более k. Поскольку минимальная мощность ребра разделителя s и t должен быть не меньше максимального числа непересекающихся путей от s до t, получаем утверждение. □

Мы видели раньше, что максимальный 0-1-поток в 0-1 сети может найден за $O(|V|^{\frac{2}{3}}|E|)$. Таким образом, из теоремы Менгера очевидно следует

Теорема 4.83. *Пусть G — граф, u и v — две вершины G. Тогда $\lambda(u, v)$ может быть определена со сложностью $O(|V|^{\frac{2}{3}}|E|)$.*

Нетрудно видеть, что вычисление $\lambda(G)$ можно осуществить за $O(|V|^{\frac{8}{3}}|E|)$ — можно посчитать $\lambda(u, v)$ для всех пар $u \neq v \in V$. Однако следующее утверждение помогает понизить сложность.

Лемма 4.84. *Пусть $G = (\{v_1, \dots v_n\}, E)$ — граф или орграф. Тогда*

$$\lambda(G) = \min\{\lambda(v_i, v_{i+1}) \mid i \in \{1, \dots, n\}\}$$

где $v_{n+1} = v_1$.

Доказательство. Пусть u и v — вершины G, для которых $\lambda(G) = \lambda(u, v)$, и T — реберный разделитель мощности $\lambda(u, v)$ для u и v. Пусть X — множество вершин w, для которых существует путь (направленный, если G — орграф) от u к w, не содержащий

ребер T. Введем Y — множество вершин w, для которого каждый путь (направленный, если G — орграф) от u к w содержит некоторое ребро из T. Тогда (X, Y) — разрез G, более того, $u \in X$ и $v \in Y$. Теперь является реберным разделителем x и y для каждой пары вершин $x \in X$ и $y \in Y$ — в противном случае существовал бы путь от u до y, не содержащий ребер из T. Отсюда

$$|T| = \lambda(G) \leqslant \lambda(x, y) \in\leqslant |T|,$$

то есть $\lambda(x, y) = \lambda(G)$. Очевидно, для некоторого i $v_i \in X$ и $v_{i+1} \in Y$, тогда $\lambda(G) = \lambda(v_i, v_{i+1})$. □

Отсюда следует, что $\lambda(G)$ графа или орграфа G может быть определена со сложностью $O(|V|^{5/3}|E|)$.

Утверждение теоремы Менгера верно и для вершинного, и для реберного разделителя.

Теорема 4.85 (Menger). *Пусть G — граф или орграф, а s и t — любые две несмежные вершины. Тогда максимальное число вершинно непересекающихся путей от s до t равно минимальной мощности разделителя вершин для s и t.*

Доказательство. Сведем доказательство этого утверждения к реберной версии теоремы Менгера. По графу $G = (V, E)$ строим граф

$$G' = \big(\{s, t\} \cup (V \setminus \{s, t\} \times \{0, 1\}), E_s \cup E_t \cup E' \cup E_V\big)$$

, где

$$E_s = \{s(v, 0) \mid v \in V \setminus \{s, t\}\}, \quad E_t = \{(v, 1)t \mid v \in V \setminus \{s, t\}\},$$
$$E_V = \{(v, 0)(v, 1) \mid v \in V \setminus \{s, t\}\}, \quad E' = \{(u, 1)(v, 0) \mid uv \in E\}$$

Грубо говоря, мы разделяем каждую вершину, отличную от s и t, на две части, соединенные ребром; это приведет к трансформации вершинно непересекающихся путей в реберно непересекающиеся пути.

По теореме Менгера максимальное число реберно непересекающихся путей от s до t в G равно минимальной мощности реберного разделителя s и t, скажем, A. Конечно, A может содержать ребра не вида $(v, 0)(v, 1)$, и в этом случае не напрямую соответствует разделителю вершин в G. Однако если некоторое ребро вида $(u, 1)(v, 0)$ входит в A, мы можем заменить его на $(u, 1)(u, 0)$ в A и снова получить минимальный реберный разделитель. . □

Сеть, построенная в прошлой теореме, удовлетворяет условиям теоремы 4.75, а значит, верно следующее утверждение.

Утверждение 4.86. *Пусть G — граф или орграф, а s и t — любые две несмежные вершины G. Тогда максимальное число вершинно непересекающихся путей от s до t — и само множество таких путей — может быть определено со сложностью $O(|V|^{\frac{1}{2}}|E|)$.*

Закончим эту секцию алгоритмом, вычисляющим $\kappa(G)$. В алгоритме ниже Pathnr(G, s, t) — процедура, вычисляющая максимальное число вершинно непересекающихся путей от s до t в графе G.

```
 1: procedure Kappa(G = (V, E))
 2:     n ← |V|, k ← 0, y ← n − 1, S ← V
 3:     while k ⩽ y do
 4:         выбрать v ∈ S и удалить v из S
 5:         for w ∈ S \ A_v do
 6:             x = Pathnr(()G, v, w)
 7:             y ← min{y, x}
 8:         end for
 9:         k ← k + 1
10:     end while
11:     return y
12: end procedure
```

Теорема 4.87. *Пусть $G = (V, E)$ — связный граф. Тогда* Kappa *корректно вычисляет $\kappa(G)$ со сложностью $O(|V|^{\frac{1}{2}}|E|^2)$.*

Доказательство. Если $G = K_n$, алгоритм завершает работу (после удаления всех n вершин) с $\kappa = y = n - 1$. Теперь допустим, что G не полный. Во время цикла **while** вершины $v_1, v_2, \ldots, v_k$ выбираются до тех пор, пока минимум γ всех значений $\kappa(v_i, w_i)$ не станет меньше k, где w_i пробегает вершины, не смежные с v_i, тогда

$$k \geqslant \gamma + 1 \geqslant \kappa(G) + 1.$$

По определению существует вершинный разделитель T в G мощности $\kappa(G)$. Поскольку $k \geqslant \kappa(G) + 1$, существует хотя бы одна вершина $v_i \notin$. Поскольку $G \setminus T$ не связно; следовательно, существует вершина v в $G \setminus T$ такая, что каждый путь из v к v_i проходит через T. В частности, v_i и v не могут быть смежными; таким образом,

$$\gamma \leqslant \kappa(v_i, v) \leqslant |T| = \kappa(G),$$

так что $\gamma = \kappa(G)$. Тем самым установлена корректность алгоритма. Сложность алгоритма, как нетрудно заметить, равна $O(\kappa(G)|V|^{\frac{3}{2}}|E|)$: во время каждой из $\kappa(G)$ итераций цикла **while** процедура Pathnr вызывается $O(|V|)$ раз, каждый из этих вызовов имеет

сложность $O(|V|^{\frac{1}{2}}|E|)$. Очевидно $\kappa(G) \leqslant \deg v$, тогда

$$\kappa(G) \leqslant \min \deg v \leqslant 2\frac{|E|}{|V|},$$

что дает желаемую сложность $O(|V|^{\frac{1}{2}}|E|^2)$. □


ЛАНЬ®


ЛАНЬ®

Часть 5

Элементы теории сложности

Со школы каждому знакомо интуитивное определение алгоритма — «набор инструкций, описывающих порядок действий исполнителя для достижения некоторого результата». Считается, что алгоритм принимает на вход некоторый *конструктивный объект*, выполняет определенную последовательность действий и возвращает (если останавливается) новый конструктивный объект. Конструктивный объект — тот, для которого существует конструктивный способ задать его, например, натуральное число, матрица или граф.

Такое понятие алгоритма довольно расплывчато и не позволяет конкретно ответить на такие вопросы:

- *любая ли задача разрешима алгоритмически?*
- *если некоторая задача не разрешается никаким алгоритмом, как это доказать?* Иными словами, как показать, что любой алгоритм разрешает задачу, отличную от данной?
- *как вычислить время, совершенных алгоритмом, или требуемую память?* Нельзя вычислить время работы алгоритма, просто рассмотрев некоторую его реализацию (например, на языке C), и сложив количества выполений каждой операции. Команды не унитарны ни по времени, ни по ресурсоемкости, время выполнения одной и той же команды может быть разным. Лучшее, что можно сделать в данном случае — оценить (если возможно) время работы команды (в худшем случае) сверху некоторой константой. Но тут важно знать, какие операции мы считаем элементарными и у каких операций время можно оценить сверху константой (в частности, сложение чисел нельзя a priori считать операцией, выполянемой за не более чем постоянное время — если мы не считаем, например, что сами слагаемые ограничены сверху).
- *как показать, что данная задача не может быть решена точно за относительно малое время?* Само по себе существование некоторого алгоритма не гарантирует, что нельзя построить более быстрый алгоритм, разрешающий ту же задачу.

Четкие ответы на эти вопросы требуют наличия логического понятия алгоритма. Первый вопрос в списке выше (Entschiedungsproblem, дословно «проблема разрешимости») сформулировал в 1928 году Гильберт:

> Формула чистого исчисления предикатов, т. е. формула, в которой нет никаких индивидуальных знаков, называется выполнимой в некоторой области индивидуумов, если можно заменить переменные высказывания значениями «истина» и «ложь», переменные предикаты — какими-либо специальными предикатами, определенными в соответствующей области индивидуумов, и свободные предметные переменные—индивидуальными предметами таким образом, чтобы формула перешла в истинное высказывание. Если о некоторой формуле говорят

просто, что она выполнима, то при этом имеют в виду, что вообще существует область индивидуумов, в которой имеет место выполнимость. Если формула $\mathfrak{A}$ в какой-нибудь области индивидуумов не общезначима, то, очевидно, $\overline{\mathfrak{A}}$ в соответствующей области выполнима, и наоборот. Аналогично простая общезначимость формулы $\mathfrak{A}$ и выполнимость $\overline{\mathfrak{A}}$ находятся в отношении утверждения и отрицания.

Обе проблемы общезначимости и выполнимости, эквивалентные друг другу, называют также обычно одним общим именем: проблемой разрешимости (Entscheidungsproblem) узкого исчисления предикатов. На основании замечаний, сделанных в § 11, мы вправе считать ее главной проблемой математической логики.

Д. Гильберт, В. Аккерман *Основы теоретической логики* [25]
пер. А. Ерофеев, М. ИЛ, 1947

В 1936 году Алонсо Черч и Алан Тьюринга независимо друг от друга дали отрицательный ответ на проблему разрешимости Гильберта. Каждый из них пришел к такому выводу, построив свои *модели вычислений*: Черч построил в работе [7] *λ-анализ*, определил эффективно вычислимые (effective calculable) функции и показал, что не существует эффективно вычислимой функции, определяющей эквивалентность данных λ-выражений. Тьюринг же в чуть более поздней работе [21] определил *вычислительную машину* (известную сегодня как *(детерминированная) машина Тьюринга*) и показал, что не существует машины Тьюринга, определяющей по любой машине Тьюринга, останавливается ли она на пустом входе. Более того, Тьюринг показал эквивалентность вычислимости, введенной в [21], и эффективной вычислимости Черча.

Машина Тьюринга оказалась удобной моделью вычислений не только для определения понятия алгоритма, но также и для определения *временной* и *пространственной сложности*, поэтому на ней мы остановимся подробнее.

Машина Тьюринга. Прежде чем сформулировать понятие машины Тьюринга, введем пару базовых определений.

Определение 5.1. *Алфавит* есть конечное множество, по умолчанию будем обозначать его $\Sigma = \{a_1, \ldots, a_N\}$. Элементы алфавита мы будем называть *буквами*.

Определение 5.2. *Слово* есть конечная последовательность символов алфавита. По умолчанию слова мы будем обозначать $x, y, z, \ldots$. Под $w[i]$ мы будем в дальнейшем подразумевать i-ый элемент этой последовательности, то есть i-ую букву. Длину слова w мы будем обозначать как $|w|$, а $\#_x(w)$ — количество вхождений буквы x в w. *Пустую строчку* ϵ определим как слово нулевой длины.

Пример 5.3. Над алфавитом $\{0, 1\}$ словами являются $w_1 = 0010$ и $w_2 = 1010110$. В слове w_1 ровно одна единица, то есть $\#_1(w_1) = 1$. Во втором слове 7 букв, то есть $|w_2| =$

7. В обоих словах одинаковая третья буква, то есть $w_1[3] = w_2[3]$. При этом $w_1[5]$ не определена, так как $|w_1| = 4$.

Определение 5.4. Множество всех слов над алфавитом $\Sigma = \{a_1, \ldots, a_N\}$ будем обозначать $\Sigma^* = \{a_1, \ldots, a_N\}^*$. *Язык* L есть подмножество Σ^*.

Определение 5.5. *Машина Тьюринга* (сокращенно МТ) (или *недетерминированная машина Тьюринга*, НМТ) — кортеж $(Q, \Sigma, \Delta, \#, \vdash, q_0, F, \delta)$, где

- Q — *конечное* множество состояний;
- Σ — входной алфавит, $\Delta \supset \Sigma$ — алфавит МТ (конечный);
- $\# \in \Delta \setminus \Sigma$ — пробельный символ;
- $q_0 \in Q$ — стартовое состояние,
- $F \subset Q$ — финальные состояния,
- $\delta \subset \big((Q \setminus F) \times \Delta\big) \times \big(Q \times \Delta \times \{-1, 0, 1\}\big)$ — отношение перехода

Машина Тьюринга $(Q, \Sigma, \Delta, \#, \vdash, q_0, F, \delta)$ называется *детерминированной*, если из каждого состояния по каждой букве существует не более одного перехода:

$$\forall p \in Q, a \in \Delta \quad |\{q \in Q \mid \exists b \in \Delta, i \in \{-1, 0, 1\}\ ((p, a), (q, b, i)) \in \delta\}| \leqslant 1$$

Иногда мы будем считать, что $F = \{q_A, q_R\}$, где $q_A \subset Q$ — состояние *принятия*, $q_R \subset Q$ — состояние *отклонения*.

В начальный момент на ленте записано *входное слово* (input) $w \in \Sigma^*$, машина находится в состоянии q_0 и головка расположена над пробелом перед первым символом входа. На каждом шаге (такте) машина, находящаяся в состоянии q над символом a, осуществляет переход по правилу δ: если $\delta(q, a) = (p, b, i)$, то МТ записывает в текущую ячейку символ b вместо a, переходит в состояние p и затем сдвигается в соответствии с i: влево, если $i = -1$, вправо, если $i = +1$, остается на месте, если $i = 0$. Переходов из q_A и q_R нет, то есть МТ, оказавшись в таких состояниях, прекращает работать.

Мы будем говорить, что МТ M *останавливается на слове* w, если M, работая на w, завершает работу в одном из состояний F. В случае, если $F = \{q_A, q_R\}$, мы будем говорить, что МТ M *принимает слово* w, если M, работая на w, достигает состояния q_A; если же на слове w МТ M достигает состояния q_R, будем говорить, что M *отвергает слово* w. В принципе машина Тьюринга может не останавливаться на слове — например, если попадет на нем в вечный цикл.

В конце работы МТ на ленте остается некоторое слово $f = f(w)$; будем говорить, что $f(w)$ — *выход* (output) машины Тьюринга, а МТ *вычисляет функцию* $f(\cdot)$.

Формально работу машины Тьюринга можно описать следующим образом.

Определение 5.6. *Конфигурация МТ* — элемент множества $Q \times \Delta^* \times \Delta \times \Delta^*$.

ПИКЧА! Элементы конфигурации (q, w_1, a, w_2) мы интерпретируем следующим образом: q — состояние, в котором находится машина, w_1 — слово, находящееся слева от

головки, a — буква, на которую указывает головка, w_2 — слово, находящееся справа от головки.

Определение 5.7. Для машины Тьюринга $M = (Q, \Sigma, \Delta, \#, q_0, F, \delta)$ определим отношение $\vdash \subset (Q \times \Delta^* \times \Delta \times \Delta^*)^2$:

- если $((p, a), (q, b, 0)) \in \delta$, то $(p, w_1, a, w_2) \vdash^* (q, w_1, b, w_2)$ для любых $w_1, w_2 \in \Delta^*$;
- если $((p, a), (q, b, -1)) \in \delta$, то $(p, w_1 c, a, w_2) \vdash^* (q, w_1, c, bw_2)$ и $(p, \epsilon, a, w_2) \vdash^* (q, \epsilon, \#, bw_2)$ для любых $w_1, w_2 \in \Delta^*$ и $c \in \Delta$;
- если $((p, a), (q, b, 1)) \in \delta$, то $(p, w_1, a, cw_2) \vdash^* (q, w_1 b, c, w_2)$ и $(p, w_1, a, \epsilon) \vdash^* (q, w_1 b, \#, \epsilon)$ для любых $w_1, w_2 \in \Delta^*$ и $c \in \Delta$;

Отношение $\vdash^* \subset (Q \times \Delta^* \times \Delta \times \Delta^*)^2$ определим как рефлексивное и транзитивное замыкание $\vdash$.

Определение 5.8. Будем говорить, что машина Тьюринга $M = (Q, \Sigma, \Delta, \#, q_0, q_A, q_R, \delta)$ *принимает* слово w, если $(q_0, \epsilon, \#, w) \vdash^* (q_A, \#^n, \#, \#^m)$ для некоторых $m, n \in \mathbb{N}$. Аналогично будем говорить, что машина Тьюринга M *отвергает* слово w, если $(q_0, \epsilon, \#, w) \vdash^* (q_R, \#^n, \#, \#^m)$ для некоторых $m, n \in \mathbb{N}$.

Определение 5.9. *Язык $L(M)$ машины Тьюринга M* — множество слов, принимаемых M. Будем говорить, что машина Тьюринга M *разрешает* L, если она принимает любое слово из L и отвергает любое слово из $\Sigma^* \setminus L$.

Пример 5.10. Построим МТ, которая распознает слова из алфавита $\Sigma = \{a, b, c\}$, в которых равное количество букв a и b (язык $L_=$). Она будет искать каждому символу a или b парный символ, который еще не занят, и ставить на них метки, что они заняты. Если какому-то символу не нашлось парного, то машина отвергает слово, а если она дошла до конца входа, и все символы спарены, то принимает слово.

- $\Delta = \{a, b, c, \hat{a}, \hat{b}, \tilde{a}, \tilde{b}\}$;
- $Q = \{q_0, q_a, q_b, q_1, q_A, q_R\}$, $F = \{q_A, q_R\}$;

- функция перехода:

$$\delta(q_0, a) = (q_b, \hat{a}, +1);$$
$$\delta(q_0, b) = (q_a, \hat{b}, +1);$$
$$\delta(q_0, \Delta) = (q_0, \Delta, +1),\ \Delta \neq a, b, \#;$$
$$\delta(q_0, \#) = (q_A, \#, 0);$$
$$\delta(q_\alpha, \alpha) = (q_1, \tilde{\alpha}, -1),\ \alpha \in \{a, b\};$$
$$\delta(q_\alpha, \beta) = (q_\alpha, \beta, +1),\ \beta \neq \alpha, \#;$$
$$\delta(q_\alpha, \#) = (q_R, \#, 0);$$
$$\delta(q_1, \Delta) = (q_1, \Delta, -1),\ \Delta \neq \hat{a}, \hat{b};$$
$$\delta(q_1, \hat{\alpha}) = (q_0, \hat{\alpha}, +1).$$

Почему эта МТ корректна? Пусть на каком-то шаге она находится в конфигурации вида $u\,[q_0]\,av\alpha bw$, где в слове $v\alpha$ нет символа b, и в слове $av\alpha bw$ справа от головки нет символа с меткой $\hat{}$. Тогда

$$u\,[q_0]\,av\alpha bw \vdash u\hat{a}\,[q_b]\,v\alpha bw \vdash^* u\hat{a}v\alpha\,[q_b]\,bw \vdash u\hat{a}v\,[q_1]\,\alpha\tilde{b}w \vdash^* \ \vdash^* u\,[q_1]\,\hat{a}v\alpha\tilde{b}w \vdash u\hat{a}\,[q_1]\,v\alpha\tilde{b}w.$$

То есть, за один такой цикла МТ находит одну пару символов a, b, ставит на них метки, и сдвигается на один шаг вправо. Если МТ находится в конфигурации вида $u\,[q_0]\,av$, где в слове v нет символа b, то

$$u\,[q_0]\,av \vdash u\hat{a}\,[q_b]\,v \vdash^* u\hat{a}v\,[q_b] \vdash u\hat{a}v\,[q_R],$$

и машина отвергает входное слово. В то же время, из конфигурации $u\,[q_0]\,v$, где в слове v нет символов a и b, машина переходит в состояние q_A:

$$u\,[q_0]\,v \vdash^* uv\,[q_0] \vdash uv\,[q_A].$$

Оставшиеся случаи разбираются аналогично, и отсюда мы видим, что построенная МТ разрешает язык $L_=$.

Можно построить также МТ с бесконечной в обе стороны лентой, МТ с многими лентами (на каждой из которых работает отдельная головка, а при переходе несколько головок могут передвигаться одновременно). Такие конструкции эквивалентны определенной нами МТ: язык L принимается МТ титтк L принимается МТ с бесконечной в обе стороны лентой (или со многими лентами). Доказательство можно прочитать, например, в **[kozen]**.

Может показаться, что МТ — очень слабая модель вычислений. Но на машине Тьюринга можно, например, реализовать сложение и умножение чисел, записанных в двоичной записи (подумайте, как!), копирование слов, сравнение и т.д. Более того, тезис Тьюринга (∼тезис Черча, Черча-Тьюринга, Поста) утверждает, что любой алгоритм можно

реализовать на машине Тьюринга. В теории вычислений это принимается как постулат, либо вообще как определение алгоритма. Этот тезис обосновывается тем фактом, что пока неизвестно ни одного алгоритма, который *нельзя* было бы реализовать с помощью некоторой МТ.

В сеттинге машины Тьюринга можно формально определить не только алгоритм, но также время работы алгоритма и его ресурсоемкость.

В следующих двух определениях считаем, что машина Тьюринга M останавливается на слове w.

Определение 5.11. *Время работы* $\mathrm{time}_M(w)$ машины Тьюринга M на слове w — длина корректного вычисления, начинающегося с конфигурации $(q, \epsilon, \epsilon, w)$. *Временная сложность* (time demand) машины Тьюринга M — функция $\mathrm{time}_M(n) : \mathbb{N} \to \mathbb{N}$, определенная следующей формулой:

$$\mathrm{time}_M(n) = \sup_{w \in \Sigma^n} \mathrm{time}_M(w)$$

Определение 5.12. *Количество памяти* $\mathrm{space}_M(w)$, использованное машиной Тьюринга M при работе на слове w — количество задействованных в корректном вычислении, начинающегося с конфигурации $(q, \epsilon, \epsilon, w)$. *Ресурсоемкость* (space demand) машины Тьюринга M — функция $\mathrm{space}_M(n) : \mathbb{N} \to \mathbb{N}$, определенная следующей формулой:

$$\mathrm{space}_M(n) = \sup_{w \in \Sigma^n} \mathrm{space}_M(w)$$

Как правило на практике проще оценить асимптотическое поведение функций $\mathrm{time}_M(n)$ и $\mathrm{space}_M(n)$, а сами эти функции вычислять очень трудно.

Закончим парой последних определений.

Определение 5.13. Пусть зафиксирован алфавит Σ. Скажем, что $L \in \mathbf{DTIME}(f(n))$, если для любого $w \in \Sigma^*$ число шагов, которые делает T на входе w, не превышает $f(|w|)$. Детерминированная машина Тьюринга T называется *полиномиальной*, если $L(T) \in \mathbf{DTIME}(p(n))$ для некоторого многочлена $p(x) \in \mathbb{Q}[x]$.

Определение 5.14. Язык $L \subset \Sigma^*$ имеет *полиномиальную сложность*, если он может быть разрешен полиномиальной детерминированной машиной Тьюринга. Класс языков, имеющих полиномиальную сложность, будем обозначать **PTIME**, или же просто **P**. Говоря иначе,

$$\mathbf{P} = \bigcup_{k \geqslant 0} \mathbf{DTIME}(O(n^k))$$

Определение 5.15. Пусть зафиксирован алфавит Σ. Язык $L \subset \Sigma^*$ принадлежит $\mathbf{NTIME}(f(n$ если для любого слова $w \in L$ существует корректное вычисление длины не более $f(n)$. Мы

говорим, что $L \in \mathbf{NP}$, если $L \in \mathbf{NTIME}(p(n))$ для некоторого многочлена $p(x) \in \mathbb{Q}[x]$; иными словами,

$$\mathbf{NP} = \bigcup_{k \geqslant 0} \mathbf{NTIME}(O(n^k))$$

Альтернативное определение класса **NP** (которым мы будем пользоваться гораздо чаще) такое:

Определение 5.16. Будем говорить, что $L \in \mathbf{NTIME}(p(n))$, если существует вычислимый за полиномиальное от длины аргументов время предикат $R(\cdot,\cdot) : \Sigma^* \times \Sigma^* \to \{0,1\}$ такой, что

$$x \in L \iff \exists y \in \Sigma^* \quad (|y| \leqslant p(|x|)) \wedge (R(x,y))$$

Слово $y \in \Sigma^*$ в правой части равенства выше называется *сертификатом* слова x.

Почему эти определения эквивалентны? Пусть L принимается некоторой НМТ M за полиномиальное время. Имея последовательность переходов НМТ y можно эмулировать ее работу на входе x (также за полиномиальное время). Тогда можно рассмотреть предикат $R(x,y)$, который равен 1, если y является корректной последовательностью шагов для M на входе x, и после ее исполнения машина останавливается в состоянии q_A, и равен 0 иначе. Как несложно видеть, $x \in L$ только если существует соответствующий «сертификат» y. Теперь докажем утверждение в обратную сторону. Построим НМТ M', которая сначала генерирует произвольную последовательность y длины не больше $P(|x|)$, а затем вычисляет значение $R(x,y)$. Очевидно, она удовлетворяет всем необходимым условиям.

Разные задачи можно также сводить друг к другу. Мы заинтересованы в анализе с учетом сложности вычислений, поэтому будем рассматривать полиномиальные сводимости.

Определение 5.17. Пусть $L_1 \subset \Sigma^*$, $L_2 \subset \Gamma^*$, говорим, что L_1 *полиномиально сводится к* L_2, если существует вычислимая всюду определенная функция $f : \Sigma^* \to \Gamma^*$ такая, что

$$\forall x \in \Sigma^* \quad x \in L_1 \iff f(x) \in L_2$$

Выражение $L_1 \leqslant_p L_2$ понимается как «L_1 p-сводится к L_2».

Если $L_1 \leqslant_p L_2$ и машина Тьюринга Alg разрешает L_2, то $Alg \circ f$ разрешает L_1: $Alg(f(x)) = \mathbf{YES}$ титтк $f(x) \in L_2$, что в свою очередь эквивалентно $x \in L_1$. Отсюда следует, например, что

$$L_2 \in \mathbf{NP} \Rightarrow L_1 \in \mathbf{NP}, \quad L_2 \in \mathbf{P} \Rightarrow L_1 \in \mathbf{P}, \quad L_1 \notin \mathbf{P} \Rightarrow L_2 \notin \mathbf{P}$$

Определение 5.18. Язык L является **NP**-*полным*, если $L \in \mathbf{NP}$ и для любого $A \in \mathbf{NP}$ имеет место сводимость $A \leqslant_p L$.

Подробности про классы **P** и **NP**, а также про полиномиальную сводимость можно прочитать в приложении, а также в [11, 27, 3, 33, 30].

RAM. Теперь мы перейдем к другой модели вычислений, более близкой к реальным компьютерам. Это машина с неограниченными регистрами, или RAM (random access machine).

Определение 5.19. *Машина с неограниченными регистрами* (RAM) — набор из бесконечной последовательности регистров $x[0], x[1], \ldots$, содержащих натуральных чисел, и конечной последовательности команд c_i вида

- $Z(n)$: $x[n] \leftarrow 0$ (регистр $x[n]$ обнуляется);
- $S(n)$: $x[n] \leftarrow x[n] + 1$ (значение регистра $x[n]$ увеличивается на единицу);
- $T(m, n)$: $x[n] \leftarrow x[m]$ (регистру $x[n]$ присваевается значение регистра $x[m]$);
- $J(m, n, k)$: если $x[n] = x[m]$, то выполняется команда c_k, иначе исполняется следующая в последовательности команда

Важная особенность — команды, которые дают возможность обращаться к произвольной ячейке памяти по ее *адресу* за константное время. Из-за этого свойства модель и называется *random access* machine. Команды RAM выполняются по порядку. Если следующую команду нельзя выполнить, то RAM прекращает свою работу. (Это происходит либо после выполнения последней команды в последовательности команд RAM, либо после перехода вида $J(m, n, k)$, где k превышает суммарное число команд RAM).

Определение 5.20. Будем говорить, что RAM M *вычисляет* функцию $f : \mathbb{N}^k \to \mathbb{N}$, если на входном наборе значений регистров

$$x[i] = \begin{cases} a_{i+1}, & i < k \\ 0, & i \geqslant k \end{cases}$$

RAM совершает конечное число шагов, а после последнего шага $x[0] = f(a_1, \ldots a_k)$.

Нетрудно убедиться в том, что на RAM можно реализовать элементарную арифметику.

Определение 5.21. Рассмотрим следующую RAM:

(1) $J(1, 2, 5)$
(2) $S(0)$
(3) $S(2)$
(4) $J(1, 1, 1)$

Убедимся в том, что эта RAM вычисляет f: $(a, b) \to a + b$. Команды 1 и 4 образуют цикл — поскольку $x[1] = x[1]$, то всегда будет осуществляться переход к первой команде; RAM прекратит работу только после перехода к пятой команде, если $x[1] = x[2]$. За каждый проход цикла значения $x[1]$ и $x[2]$ увеличиваются на единицу, и к моменту выполнения первой команды $x[0] = a + n$, $x[1] = b$, $x[2] = n$. Следовательно, RAM остановится титтк $x[1] = x[2] = b$, в этот момент $x[0] = a + b$.

Определение 5.22. Рассмотрим следующую RAM:

(1) $J(0, 2, 5)$
(2) $J(1, 2, 6)$
(3) $S(2)$
(4) $J(1, 1, 1)$
(5) $T(1, 0)$

Убедимся в том, что эта RAM вычисляет $f\colon (a, b) \to \max(a, b)$. RAM изменяет значение только третьего регистра. Если $x[2] \neq x[0]$ и $x[2] \neq x[1]$, то третья строчка выполняется и $x[2]$ увеличивается на единицу, после чего RAM снова возвращается к исполнению первой команды. Если $a \leqslant b$, то среди значений $x[2]$ раньше встретится a, тогда совершается переход к команде $T(2, 0)$, после чего RAM завершает работу и $x[0] = b = \max(a, b)$. Если $a < b$, то b встретится раньше среди значений $x[2]$, и тогда RAM, выполняя вторую команду, перейдет на шестую строчку и завершит тем самым работу; тогда $x[0] = a = \max(a, b)$.

Определение 5.23. Рассмотрим следующую RAM:

(1) $J(0, 2, 9)$
(2) $J(1, 3, 5)$
(3) $S(3)$
(4) $S(4)$
(5) $J(1, 1, 1)$
(6) $Z(3)$
(7) $S(2)$
(8) $J(1, 1, 1)$
(9) $T(4, 0)$

Убедимся в том, что эта RAM вычисляет $f\colon (a, b) \to ab$. Нетрудно убедиться, что в любой момент до выполнения девятой команды $x[0] = a$, $x[1] = b$, $x[2] = i$, $x[3] = j$, $x[4] = b(i - 1) + j$. Выход из большого цикла (с 1 по 8 команду) совершается титтк $x[2] = a$, в этот момент $x[4] = b(a - 1) + b = ab$.

Реализацию деления с остатком, вычисления НОДа и НОКа и прочей арифметики оставим как упражнение.

RAM и машина Тьюринга — эквивалентные модели, их можно моделировать друг на друге. Например, чтобы смоделировать МТ на RAM, достаточно хранить в ячейке $x[i]$ число 0 или 1, соответствующее символу в ячейке ленты МТ, и в выделенном регистре хранить положение головки. Переходы МТ, очевидно, легко смоделировать на RAM. В то же время, МТ может хранить целые числа и выполнять арифметические операции, поэтом на ней можно моделировать RAM.

Чтобы определить время работы RAM, нужно присвоить каждой команде определенное время выполнения, вес. Тогда время работы $t(x)$ есть сумма времени выполнения

всех команд, исполнявшихся RAM. Существуют различные способы это сделать. Например, можно присвоить всем командам одинаковое время выполнения — тогда мы получим *равномерный весовой критерий*. Его разумно применять, если мы считаем, что в ходе работы программы не будут возникать слишком большие числа. Например, мы анализируем алгоритм, который будет реализован на C++, и знаем, что в реальных задачах хватит типа данных int64. Однако надо учесть, что равномерный весовой критерий, вообще говоря, позволяет производить некоторые операции гораздо быстрее, чем это возможно в реальности: например, вектора или матрицы можно записать как очень большие целые числа, тогда их сложение в нашей модели будет занимать $O(1)$ времени.

Другой распространенный подход — *логарифмический весовой критерий*. В списке команд нет умножения и деления, а сложение целых чисел с n битами занимает $O(\log n)$ времени. Этот критерий соответствует случаю, когда программа работает с большими числами, выходящими за пределы разрядной сетки *(длинная арифметика)*.

Очевидно, RAM может выигрывать у МТ в быстродействии, за счет произвольного доступа к памяти. Имеют место следующие соотношения между временем работы RAM и МТ.

Теорема 5.24. *Пусть (многоленточная) машина Тьюринга M делает $t_M(x)$ шагов на входе x. Тогда можно построить RAM R, которая будет работать за время $t_R(x) = O\big(t_M(x)\big)$.*

Если дана RAM R с логарифмическим *весовым критерием, которая работает за время $t_R(x)$, то можно построить многоленточную МТ, работающую за $O\big(t_R^2(x)\big)$.*

Отсюда следует, что сложности RAM и МТ полиномиально связаны. Как мы увидим, для нас это принципиально, так как при изучении классов сложности принято считать, что *эффективные алгоритмы* — это то же самое, что *полиномиальные* (с полиномиально ограниченным временем работы). Это, конечно, очень большое допущение, но такая, довольно грубая, дифференциация по сложности оказалась очень плодотворной.

В дальнейшем мы будем неявно предполагать при анализе сложности, что рассматриваемые нами алгоритмы реализованы на RAM: либо с равномерным весовым критерием, если мы не учитываем длину записи чисел, либо с логарифмическим. Другой вариант — учитывать только число *элементарных операций* в ходе алгоритма — например, сложений и/или умножений чисел (это будет обговорено в условии задачи).

Вероятностные алгоритмы: определения

На практике порой оказывается, что детерминированные алгоритмы работают слишком долго (например, в теории чисел — детерминированная проверка числа на простоту, даже полиномиальная, занимает слишком много времени). Один из способов обойти это — построить вероятностный алгоритм, то есть, неформально говоря, алгоритм, совершающий некоторые шаги случайно.

Определение 5.25. *Вероятностная машина Тьюринга* (сокращенно ВМТ) — кортеж $(Q, \Sigma, \Delta, \#, \vdash, q_0, F, \delta_1, \delta_2)$, где

- Q — *конечное* множество состояний;
- Σ — входной алфавит, $\Delta \supset \Sigma$ — алфавит МТ (конечный);
- $\# \in \Delta \setminus \Sigma$ — пробельный символ;
- $q_0 \in Q$ — стартовое состояние,
- $F \subset Q$ — финальные состояния,
- $\delta_1, \delta_2 \colon (Q \setminus \{q_A, q_R\}) \times \Delta \to Q \times \Delta \times \{-1, 0, 1\}$ — функции перехода некоторых детерминированных МТ

Вероятностная машина Тьюринга обрабатывает входное слово, совершая каждый переход случайно: в каждом состоянии по каждой букве есть по одному переходу δ_1 и δ_2, с вероятностью $\frac{1}{2}$ совершается один из этих переходов. В отличие от недетерминированной машины Тьюринга, которая «одновременно находится в нескольких состояниях», ВМТ находится ровно в одном, но случайно определенном состоянии.

Формально говоря, по ВМТ $M = (Q, \Sigma, \Delta, \#, \vdash, q_0, F, \delta_1, \delta_2)$ и входному слову w строится ориентированное дерево $T_{M,w}$, корень которого помечен конфигурацией $(q_0, \epsilon, \#, w)$, а ребра имеют вид

$$\begin{cases} \big((p, w_1 c, a, w_2), (q, w_1, c, bw_2)\big), & \delta_i(p, a) = (q, b, -1) \\ \big((p, w_1, a, cw_2), (q, w_1 b, c, w_2)\big), & \delta_i(p, a) = (q, b, 1) \\ \big((p, w_1, a, w_2), (q, w_1, b, bw_2)\big), & \delta_i(p, a) = (q, b, 0) \end{cases}$$

Вес каждого ребра равен $\frac{1}{2}$, вес пути равен произведению входящих в него ребер. ВМТ M *останавливается на слове* w *с вероятностью* p, если сумма весов всех путей в $T_{M,w}$ из корня в висячие вершины (помеченные $(q, \#^n, \#, \#^m)$) равна p.

Определение 5.26. По аналогии с МТ будем говорить, что ВМТ $M = (Q, \Sigma, \Delta, \#, \vdash, q_0, \{q_A, q_R\}, \delta_1, \delta_2)$ *принимает слово* w *с вероятностью* p (*отвергает слово* w *с вероятностью* p), если сумма весов всех путей в $T_{M,w}$ из корня в висячие вершины, помеченные $(q_A, \#^n, \#, \#^m)$ (соответственно помеченные $(q_R, \#^n, \#, \#^m)$), равна p.

Определение 5.27. ВМТ $M = (Q, \Sigma, \Delta, \#, q_0, q_A, q_R, \delta)$ *слабо разрешает* (или *разрешает в смысле Монте-Карло*) язык L, если для любого входа $x \in L$ с вероятностью хотя бы $\frac{3}{4}$ машина M остановится в состоянии q_A, в противном случае машина M остановится в состоянии q_R с вероятностью не менее $\frac{3}{4}$.

В этом определении число $\frac{3}{4}$ можно заменить на любое число в интервале $(0, \frac{1}{2})$, читатель может убедиться в том, что класс распознаваемых языков от этого не изменится.

Определение 5.28. ВМТ $M = (Q, \Sigma, \Delta, \#, q_0, \{q_A, q_R\}, \delta)$ *принимает* язык L, если она всегда отвергает слово, не лежащее в L, а на слове из L она останавливается в q_A с вероятностью не менее $\frac{1}{2}$.

В этом определении число $\frac{3}{4}$ можно заменить на любое число в интервале $(0,1)$, читатель может убедиться в том, что класс распознаваемых языков от этого не изменится.

Определение 5.29. ВМТ $M = (Q, \Sigma, \Delta, \#, q_0, \{q_A, q_R\}, \delta)$ *сильно разрешает* (или *разрешает в смысле Лас-Вегас*) язык L, если для любого входа $x \in \Sigma^*$ с вероятностью 1 машина M остановится в состоянии q_A, в противном случае машина M остановится в состоянии q_R с вероятностью 1.

Теперь определим время работы вероятностной машины Тьюринга M.

Определение 5.30. *Время работы* $\mathrm{time}_M(w)$ машины Тьюринга M на слове w — случайная величина на множестве листьев $T_{M,w}$, равная на каждом листе его глубине.

По аналогии с классами **P** и **NP** определим вероятностные полиномиальные классы сложности.

Определение 5.31. Пусть ВМТ $M = (Q, \Sigma, \Delta, \#, \vdash, q_0, \{q_A, q_R\}, \delta_1, \delta_2)$ принимается (слабо принимается, сильно принимается) за ожидаемое время $f(n)$, если

$$\mathbb{E}[\sup_{w \in \Sigma^n} \mathrm{time}_M(w)] = f(n)$$

Определение 5.32. Будем говорить, что $L \in$ **BPP** (соответственно $L \in$ **RP** и $L \in$ **ZPP**), если L слабо разрешается (принимается, сильно разрешается) некоторой ВМТ за ожидаемое время $O(n^d)$ для некоторого $d \in \mathbb{N}$.

Нетрудно видеть, что $\mathbf{P} \subset \mathbf{ZPP} \subset \mathbf{RP} \subset \mathbf{BPP}$. Возможность включения в обратную сторону до сих пор остается открытым вопросом.

Классы P, NP и co − NP

Во-первых, полиномы над любым кольцом образуют кольцо, то есть замкнуты относительно операций сложения и умножения. Во-вторых, многочлены замкнуты относительно композиции: если $p(x)$, $q(x)$ — многочлены, то $p(q(x))$ — также многочлен. Этим мы будем пользоваться, чтобы доказывать замкнутость класса **P** относительно естественных языковых операций [собственно, ради таких свойств замкнутости этот класс и стали рассматривать].

Утверждение 5.33. $\mathbf{P} = \mathbf{co}{-}\mathbf{P}$, *то есть если* $L \in \mathbf{P}$, *то* $\overline{L} \in \mathbf{P}$.

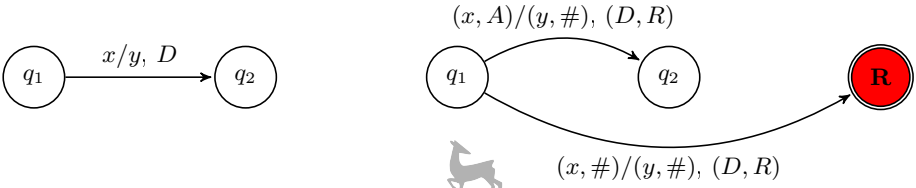
Доказательство. Здесь нельзя просто поменять местами q_A и q_R: существуют слова, на которых машина может в какой-то момент попасть в состояние, из которого ни q_A, ни q_R не достижимы [например, машина вылетает в вечный цикл на некотором слове]. Ключевым здесь становится ограничение по времени: для M по определению существует многочлен $p(x)$ такой, что на любом слове $w \in L(M)$ время работы машины M на w не превышает $p(|w|)$. Согласно определению ДМТ из каждого нефинального состояния

по каждой букве определен переход, поэтому если слово $w \notin L$, то возможны лишь две альтернативы:

- M останавливается в состоянии q_R за не более $p(|w|)$ тактов;
- M не выдает булевского ответа за $p(|w|)$ шагов, то есть не останавливается ни в одном из финальных состояний.

Поэтому модифицируем M следующим образом: пусть $\widetilde{M}$ — двухленточная детерминированная машина Тьюринга, на первой ленте эмулирующая M, а на второй «отсчитывающая время работы». [Двухленточная машина Тьюринга эквивалентна одноленточной, подробнее об этом можно прочитать в Вялом.] А именно, пусть $\Gamma = \Sigma \cup \{A\}$, тогда новая машина $\widetilde{M}$ есть композиция двух подмашин $Poly_{p(x)}$ и M', вход $w = w_1 \dots w_{|w|}$ (где $w_i \in \Sigma$ — буквы) подается на первую ленту, и

- $Poly_{p(x)}$ — блок, вычисляющий $p(|w|)$ [записывает на вторую ленту $A^{|w|}$, на второй ленте вычисляет $p(|w|)$, оставляя на второй ленте слово $A^{p(|w|)}$, и возвращает головки на обоих лентах в начальное положение]
- M' — подмашина Тьюринга, полученная из M заменой переходов



Фактически мы просто добавили к каждому переходу отсчитывание времени и остановку в q_R после любых $p(|w|)$ переходов оригинальной машины Тьюринга M.

Время работы M' очевидно не превосходит $p(|w|)$, а $Poly_{p(x)}$ за $|w|$ записывает $A^{|w|}$ на вторую ленту, за полиномиальное время вычисляет сам многочлен и за $O(|w| + p(|w|))$ возвращает головки в стартовое положение. Таким образом, суммарное время работы полиномиально по длине входного слова. □

Кроме того, полиномиальные языки замкнуты относительно объединения, конкатенации и звездочки Клини. Доказательство, впрочем, совершенно не аналогичное доказательству замкнутости регулярных языков. На примере следующей задачи мы поймем, как его делать.

Утверждение 5.34. *Если $L \subset \Sigma^*$ имеет полиномиальную сложность, то и $L^* \subset \Sigma^*$ имеет полиномиальную сложность.*

Доказательство. Решение, конечно же, заключается в том, чтобы перестроить оригинальную машину Тьюринга M, принимавшую L, получить новую машину для языка L^* и установить ее полиномиальность. Основным аргументом будет полиномиальная ограниченность по времени.

Для начала по ДМТ M построим двухленточную ДМТ $\widetilde{M}$, принимающую тот же язык L, но останавливающуюся в q_R на всех словах из $\overline{L}$, и будем в дальнейшем использовать именно ее. Теперь заметим, что $x \in L^*$ тогда и только тогда существуют непустые слова $x_1, \dots x_n \in L \setminus \{\epsilon\}$, такие, что $x = x_1 \dots x_n$. Сначала проверим, равен ли первый символ $\#$, при равенстве совершаем переход в q_A, в противном переходим в другое состояние и запустим «динамику» на слове w, то есть будем, как в динамическом программировании, для каждого префикса $w[1:k]$ проверив его принадлежность языку L^*. Действительно,

$$w[1:k] \in L^* \iff \exists l \in (1,k),\ w[1:l] \in L^*,\ w[l+1:k] \in L,$$

поэтому можно хранить слово $u \in \{1,0\}^*$ такое, что $u[i] = 1$ тогда и только тогда $w[1:i] \in L^*$, для каждого k вычислять $w[i:k] \stackrel{?}{\in} L$, затем проверить для каждого $l \in (1,k)$ условие $(u[l] = 1) \wedge (w[l+1:k] \in L)$ и вычислить таким образом новую букву $u[k]$. [При $k = 1$ проверка сводится к проверке $w[1] \in L$.]

Машина $\widetilde{M}$ за полиномиальное время разрешит принадлежность любого $w[i:j]$, то есть за $p(|w[i:j]|)$ даст положительный или отрицательный ответ, и машина, построенная в этой задаче, корректно разрешит принадлежность любого префикса языку L^* [то есть всегда даст ответ «да» или «нет»]. Мы вызываем оригинальную ДМТ $\widetilde{M}$ для всех подслов w, то есть $\binom{|w|}{2} = O(|w|^2)$ раз; так как многочлен монотонно возрастает, начиная с некоторого x, то суммарное время работы подмашины $\widetilde{M}$ оценивается как

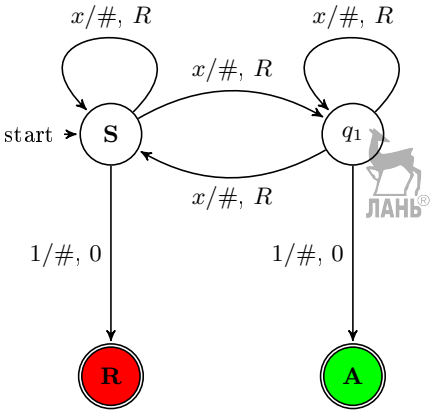
$$\sum_{1 \leqslant i < j \leqslant |w|} p(|w[i:j]|) = O(\binom{|w|}{2} p(|w|)).$$

Остальные операции проделываются за не более чем квадратичное время [вычисление $u[k]$ занимает $O(k)$ операций]. Поэтому время работы полученной машины Тьюринга полиномиально. □

Утверждение 5.35. *Класс* **NP** *замкнут относительно $*$-операции Клини.*

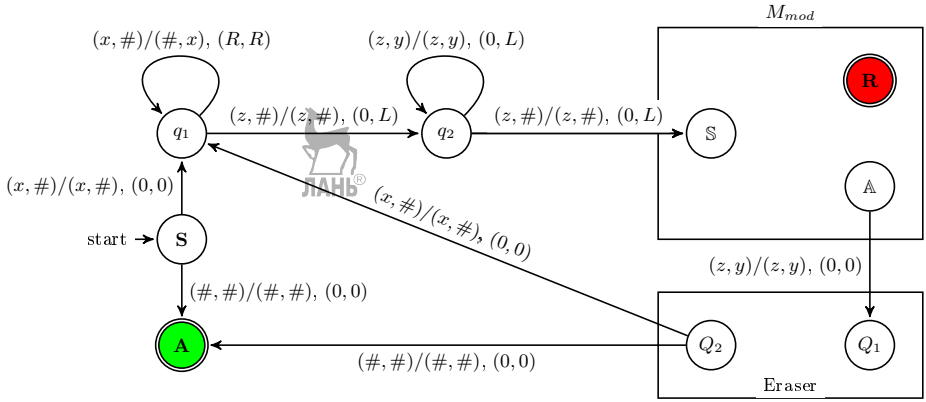
Решение. Дан L, принимающийся некоторой недетерминированной МТ M, работающей на всех $w \in L$ за не более чем $p(|w|)$ тактов [для некоторого многочлена $p(\cdot)$]; построим НМТ $\widetilde{M}$, принимающую L^*. Заметим, что $x \in L^*$ тогда и только тогда существуют непустые слова $x_1, \dots x_n \in L \setminus \{\epsilon\}$, такие, что $x = x_1 \dots x_n$. Мы сделаем нашу конструкцию способом, похожим на решение задачи 26. Тут же, впрочем, возникнет некоторая трудность: первым шагом в том решении была замена оригинальной ДМТ на машину, которая за $O(p(|w|))$ тактов останавливается на слове w и определяет принадлежность языку слова

w. Здесь же a priori не очевидно, что для НМТ M существует НМТ M', принимающая тот же язык и останавливающаяся за полиномиальное время на всех $w \in \Sigma^*$.



Корректных вычислений длины $p(|w|)$ для некоторого многочлена $p(\cdot)$ может быть $2^{p(|w|)-1}$, как, например, в примере слева [здесь x пробегает множество $\{0,1\}$] — все корректные последовательности вычислений длины k здесь есть последовательности состояний $\{S, q_1\}$ длины k. НМТ, которая принимает $\overline{L}$, должна проверить, что все вычисления длины $p(|w|)$ дают отрицательный ответ. В частных случаях можно придумать способ проверить за полиномиальное время, что на всех вычислениях на слове w НМТ дает отрицательный ответ. Общей конструкции для любой полиномиальной НМТ пока нет, но в данной задаче можно эту проблему обойти.

Сделаем нашу НМТ $\widetilde{M}$ двухленточной: на первой ленте будем оставлять «непрочитанный» префикс слова w, а на второй — эмулировать M. Если формально, то мы рассмотрим $\Gamma = \Sigma \cup \{\perp\}$ и следующую НМТ с таким алфавитом ленты. В переходах x и y пробегают все буквы алфавита, z пробегает множество $\Sigma \cup \{\#\}$, а M_{mod} получилась из M следующей модификацией переходов:



- $\perp$ считается как «новый символ» пустой ячейки, то есть переход $(q_i, y_i) \to (q_j, \#, D_{ij})$ в оригинальной НМТ M заменяется на $(q_i, y_i) \to (q_j, \perp, D_{ij})$, а к переходу вида $(q_i, \#) \to (q_j, y_j, D_{ij})$ добавляется переход $(q_i, \perp) \to (q_j, y_j, D_{ij})$. Мы хотим испрользовать новый «пустой символ», чтобы после завершения M_{mod} во всех посещенных ячейках были записаны буквы, и «стирающая подмашина» работала корректно: она просто найдет левый конец использованной памяти и, проходя вправо, сотрет все символы. Более того, наш garbage collector работает полиномиально, как мы выясним чуть позже.
- Все переходы сохраняют головку первой ленты на месте и не меняют буквы, то есть переход $(q_i, y_i) \to (q_j, y_j, D_{ij})$ в оригинальной НМТ преобразуется в

$$(q_i, (x, y_i)) \to (q_j, (x, y_j), (0, D_{ij})).$$

Сначала проверим, что не примем никакого слова из $\Sigma^* \setminus L^*$. Попасть в $\mathbf{A}$ можно либо из старта, если входное слово пусть [и $\epsilon \in L^*$], либо из Q_2; в таком случае M_{mod} приняла слово $u_1 \in \Sigma^*$, которое является суффиксом входного слова [в состоянии $\mathbb{S}$ на первой ленте было пустое слово, а на второй — u_1, значит, аналогичное состояние лент было и в q_1]. В состояние q_1 можно было попасть только из Q_2, значит, M_{mod} успела принять скопированное с ленты слово $u_2 \in \Sigma^*$ и $u_2 u_1$ — суффикс входного слова. Аналогично получим, что в состоянии q_1 на первой ленте записано слово $u_k u_{k-1} \dots u_2 u_1$, вторая лента пустая, а $u_i \in L$. То есть попасть в $\mathbf{A}$ из q_1 можно лишь по слову $u_k u_{k-1} \dots u_2 u_1$, в котором все $u_i \in L$. При этом по любому $u_k u_{k-1} \dots u_2 u_1$, в котором все $u_i \in L$, можно попасть в финал: соответствующее разбиение на слова из L существует. То есть принимаемые слова образуют в точности L^*.

Осталось проверить полиномиальность построенной машины. Время работы на пустом слове есть $O(1)$; на слове $x = x_1 \dots x_n$, $x_i \in L$ ветка вычислений, дающая ответ «$x_1 \dots x_n \in L$», по очереди переписывает x_i на вторую ленту за $|x_i|$ тактов, за $|x_i|$ тактов сдвигает головку на второй ленте в начало за время, не превышающее $p(|x_i|)$, принимает x_i, и за не более чем $p(|x_i|)$ стирает память второй ленты: за не более чем $p(|x_i|)$ тактов МТ в одной последовательности вычислений может посетить не более $p(|x_i|)$ ячеек ленты. Еще нужно учесть переходы из Q_2, их не более x. Таким образом, суммарное время работы не превышает

$$|x| + 2\sum_{i=1}^{n}(|x_i| + p(|x_i|)) = 3|x| + 2\sum_{i=1}^{n} p(|x_i|) \leqslant 3|x| + 2p(|x|),$$

так $x_1^n + \ldots + x_k^n \leqslant (x_1 + \ldots + x_k)^n$ как для любого $n \geqslant 1$.

Пусть $R_L(\cdot, \cdot)$ — вычислимый предикат, задающий L [в смысле предикатного определения **NP**]. Тогда $R(\cdot, \cdot)$, предикат для L^*, удовлетворяет двум соотношениям:

- ϵ — сертификат слова ϵ;
- если y_i — сертификаты слов $x_i \in L$, то $y_1 \dots y_n$ — сертификат $x_1 \dots x_n$.

□

Определение 5.36. Пусть зафиксирован алфавит Σ. Скажем, что $L \in \mathbf{DSPACE}(f(n))$, если для любого $w \in \Sigma^*$ число ячеек, которые использует машина Тьюринга T на входе w, не превышает $f(|w|)$. Детерминированная машина Тьюринга T называется *полиномиальной*, если $L(T) \in \mathbf{DSPACE}(p(n))$ для некоторого многочлена $p(x) \in \mathbb{Q}[x]$.

Определение 5.37. Класс языков **PSPACE** определим как

$$\mathbf{PSPACE} = \bigcup_{k \geqslant 0} \mathbf{DSPACE}(O(n^k))$$

Определение 5.38. Пусть зафиксирован алфавит Σ. Язык $L \subset \Sigma^*$ принадлежит классу $\mathbf{NSPACE}(f(n))$, если для любого слова $w \in L$ существует корректное вычисление, использующее не более $f(n)$ ячеек. Мы говорим, что $L \in \mathbf{NP}$, если $L \in \mathbf{NSPACE}(p(n))$ для некоторого многочлена $p(x) \in \mathbb{Q}[x]$; иными словами,

$$\mathbf{NPSPACE} = \bigcup_{k \geqslant 0} \mathbf{NSPACE}(O(n^k))$$

Теорема 5.39 (Savitch). *Если к тому же* $f(n) \geq \log n$, *то*

$$\mathbf{NSPACE}(f(n)) \subset \mathbf{DSPACE}(f^2(n)).$$

В частности, $\mathbf{NPSPACE} = \mathbf{PSPACE}$

Опять же, для любой *достаточно легко* вычислимой функции $f(n)$ имеют место следующие соотношения между различными классами:

$$\mathbf{DTIME}(f(n)) \subset \mathbf{NTIME}(f(n)) \subset \mathbf{DSPACE}(f(n)) \subset \mathbf{NSPACE}(f(n)) \subset \bigcup_{c>0} \mathbf{DTIME}(2^{cf(n)}).$$

Из сказанного выше следуют следующие соотношения между классами:

$$\mathbf{P} \subset \mathbf{NP} \cap \mathbf{co} - \mathbf{NP},$$
$$\mathbf{NP} \subset \mathbf{PSPACE} \subset \mathbf{EXPTIME}.$$

Однако, вопрос о том, верно ли $\mathbf{P} = \mathbf{co} - \mathbf{NP}$, остается открытым — это одна из «проблем тысячелетия». Есть и другие вопросы, например, равны ли классы **NP** и $\mathbf{co} - \mathbf{NP}$ (если $\mathbf{P} = \mathbf{NP}$ то это, конечно, так).

Пример 5.40. В качестве простейшего примера задачи из **NP** можно привести *SAT* — задача определения того, является ли заданная булева формула выполнимой (satisfiability). Соответствующий язык L_{sat} состоит из кодировок всех выполнимых формул (можно считать, что формула есть корректная последовательность из символов $\{(,), \wedge, \vee, \neg\}$ и номеров переменных). Достаточно очевидно, что он лежит в **NP**: предикатом $R(F, x)$ является значение формулы F на наборе переменных x, а сертификатом — любой выполняющий набор (если он существует).

$$PRIMES \subset \mathbf{NP} \cap \mathbf{co-NP}$$

Язык, состоящий из двоичных записей простых чисел, лежит в $\mathbf{co-NP}$. Действительно, его дополнение принадлежит **NP**: в качестве сертификата для составного числа можно предъявить его сомножители. Гораздо сложнее показать следующее.

Утверждение 5.41 ([18]). *Язык простых чисел принадлежит классу* **NP**.

Доказательство. Построим предикат $R(\cdot,\cdot)$ и докажем полиномиальность размера сертификата и времени проверки истинности R. Число p простое титтк все ненулевые числа, меньшие p, взаимнопросты с ним, то есть все ненулевые остатки по модулю p образуют мультипликативную группу. Если же p не просто и делится на некоторый $d < p$, то d не обратим по модулю p и не может быть элементом мультипликативной группы $(\mathbb{Z}/p\mathbb{Z})^\times$. Более того, для простого p мультипликативная группа $(\mathbb{Z}/p\mathbb{Z})^\times$ циклична, а значит, существует порождающий этой группы $g \in (1;p)$. Таким образом,

$$p - \text{простое} \iff \exists g \in (1;p), \quad \{1, g \bmod p, \dots g^{p-1} \bmod p\} \equiv \{1, 2, \dots p-1\}.$$

Последнее верно титтк порядок элемента g в $(\mathbb{Z}/p\mathbb{Z})^\times$ равен $p-1$, то есть

$$\forall p_i - \text{простое}, p_i | p-1 \qquad g^{p-1} = 1, g^{\frac{p-1}{p_i}} \neq 1$$

Теперь построим сертификат простоты p и предъявим детерминированный полиномиальный алгоритм, позволяющий проверить истинность сертификата. Рассмотрим корневое дерево (V, E, r, f), где

- $f : V \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}^\infty$ — функция, помечающая каждую вершину набором чисел $[g, p, p_1, \dots p_k]$, где p_i — простые числа, встречающиеся в разложении $p-1$;
- $(v_1 v_2) \in E$, если вершина v_1 помечена $[g, q, q_1, \dots q_k]$, а v_2 помечена $[g', q_i, q'_1, \dots q'_k]$ для некоторого i;
- корень $r \in V$ помечается $[g, p, p_1, \dots p_k]$, то есть вторая компонента в ней суть исходное число p, простота которого проверяется.

Суть заключается в том, чтобы в вершине $[g, p, p_1, \dots p_k]$ установить, что g — первообразный корень по модулю p, а все остальные p_i, встречающиеся в разложении $p-1$, также являются простыми. Для этого и нужно дерево: проверка простоты запускается рекурсивно для каждого p_i.

Сначала докажем, что дерево имеет полиномиальный по $\log p$ размер. Согласно правилам построения дерева по списку пометок вершин можно однозначно построить дерево: мы знаем корень, помеченный исходным p, можем восстановить его непосредственных потомков, а дальше продолжаем достраивать дерево рекурсивно. Поэтому достаточно лишь показать, что суммарная длина всех пометок полиномиально ограничена. Действительно,

$g \in (1; p)$, поэтому длина его записи ограничена $\log(p)$. Далее, для некоторых $\alpha_i \geqslant 1$

$$p - 1 = \prod_{i=1}^{k} p_i^{\alpha_i} \geqslant \prod_{i=1}^{k} p_i \Rightarrow \log(p-1) \geqslant \sum_{i=1}^{k} \log(p_i).$$

Таким образом, суммарная длина простых делителей $p-1$ не превышает $\log(p)$, и тогда $|[g, p, p_1, \ldots p_k]| \leqslant 3\log(p)$. Пусть $L(p)$ — суммарная длина всех пометок сертификатного дерева для p, тогда $L(p) \leqslant 3\log(p) + \sum\limits_{i=1}^{k} L(p_i)$. Раскроем неравенство:

$$L(p) \leqslant 3\log(p) + \sum_{i=1}^{k} L(p_i) \leqslant 3\log(p) + 3\sum_{i=1}^{k} \log(p_i) + \sum_{i,j} L(q_{ij}) \leqslant \ldots,$$

где q_{ij} — простые числа, встречающиеся в разложении $p_i - 1$. Таким образом, мы ограничиваем $L(p)$ суммой по всем уровням дерева сертификата длин простых чисел [простоту которых мы проверяем]. Выше мы уже доказали, что суммарная длина простых делителей $p-1$ не превышает $\log(p)$, поэтому сумма длин простых чисел на каждом уровне не превышает аналогичной суммы на предыдущем уровне, то есть

$$\log(p) \geqslant \sum_{i=1}^{k} \log(p_i) \geqslant \sum_{i,j} \log(q_{ij}) \geqslant \ldots$$

Самих же уровней в дереве не более $\log(p)$: максимальный простой делитель $p-1$ не превышает $\frac{p-1}{2}$ [минимальный простой делитель равен хотя бы 2], таким образом, на k-ом уровне максимальное простое число не превышает $\frac{p-1}{2^k}$. Таким образом, длина всех меток дерева не превышает $3\log(p) \cdot \log(p) = 3\log(p)^2$ и, таким образом, сертификат простоты полиномиален по длине.

Теперь докажем, что можно проверить корректность сертификата полиномиальным алгоритмом. Для корня дерева, помеченного $[g, p, p_1, \ldots p_k]$, нужно проверить, что $g^{p-1} = 1$, $g^{\frac{p-1}{p_i}} \neq 1$ для всех p_i — простых делителей $p-1$, а также то, что простые делители $p-1$ — указанные p_i и только они. Числа не длиннее $\log(p)$ можно умножать и делить друг на друга с остатком за не более чем $O(\log(p)^2)$ тактов. Возводить число в степень можно быстро, возводя число в квадрат и затем беря остаток по модулю p; быстрое возведение в степень можно осуществить, используя $O(\log(p))$ операций умножения и $O(\log(p))$ операций взятия остатка [после каждого умножения], сложность каждой операции есть $O(\log(p)^2)$, поэтому за $O(\log(p)^3)$ можно вычислить g^{p-1}. Сравнить с 1 можно за $O(\log(p))$:

нужно найти младший не равный 1 разряд, в котором стоит не 0. Поэтому проверить истинность $g^{p-1} = 1$ равенства $O(\log(p)^3)$. Далее, простых делителей $p-1$ не больше $\log(p)$:

$$p - 1 = \prod_{i=1}^{k} p_i^{\alpha_i} \geqslant \prod_{i=1}^{k} p_i \geqslant 2 \cdot 3 \cdot \ldots P_k,$$

где P_k — k-ое простое число, а $P_k \geqslant k$ для $k \geqslant 3$, поэтому $2 \cdot 3 \cdot \ldots P_k \geqslant 2 \cdot \ldots \cdot k = k!$; следовательно, $p - 1 \geqslant k!$, и если $k > \log(p)$, то $\log(p)! > p - 1$ при больших p: из анализа известно $n! > \frac{n^n}{e^n}$, тогда при $n > 10$ имеем

$$n! \geqslant \frac{n^n}{e^n} > e^n \geqslant e^n - 1 \Rightarrow \log(p)! > p - 1.$$

Тогда чисел $g^{\frac{p-1}{p_i}}$ всего $O(\log(p))$, значит, проверить истинность $g^{\frac{p-1}{p_i}} \neq 1$ для всех p_i можно за $O(\log(p)^4)$. Осталось узнать, что все p_i делят p, а других простых делителей нет. Будем сначала делить p на p_1 с остатком, $p_1 | p$ титтк мы будем делить на p_1 хотя бы два раза [во второй раз может получиться нетривиальный остаток]. Когда полученное число не будет делиться на p_1, будем делить на p_2 с остатком. В итоге получится число, не делящееся на все p_i, надо сравнить его с единицей. Каждое деление с остатком можно проделать за $O(\log(p)^2)$, сравнение остатков с нулем проделывается за $O(\log(p))$, а самих делений нужно произвести $O(\log(p))$:

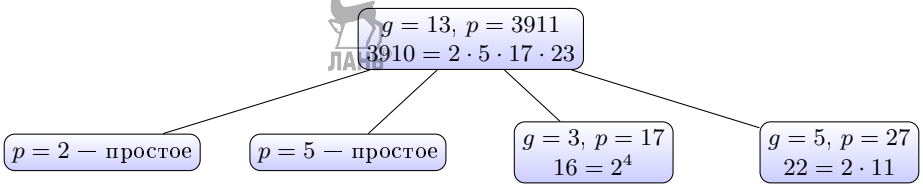
$$p - 1 = \prod_{i=1}^{k} p_i^{\alpha_i} \geqslant 2^{\sum \alpha_i} \Rightarrow \sum \alpha_i \leqslant \log(p-1) < \log(p)$$

Поэтому проверка того, что простые делители $p-1$ есть p_i и только они, требует $O(\log(p)^3)$. Таким образом, в каждой вершине дерева проделывается $O(\log(p)^4)$ битовых операций, а так как проверка простоты запускается для всех p_i, то имеет место рекурсия

$$T(p) = O(\log(p)^4) + \sum_{i=1}^{k} T(p_i).$$

Раскрывая рекурсию, получим $T(p) = O(\log(p)^4) + \sum\limits_{i=1}^{k} O(\log(p_i)^4) + \ldots$ — сумму сумм $O(\log(p_i)^4)$ на каждом уровне, причем константу для ограничения сверху можно взять одну и ту же. Тогда так как $x_1^n + \ldots + x_k^n \leqslant (x_1 + \ldots + x_n)^n$, то на каждом уровне сумму $\sum\limits_{i=1}^{k} O(\log(p_i)^4)$ можно оценить $O(\log(p)^4)$. Глубина сертификатного дерева не превышает $\log(p)$, поэтому сложность проверки корректности сертификата есть $O(\log(p)^5)$, то есть полиномиальна. □

Пример 5.42. Построим **NP**-сертификат простоты для числа $p = 3911$, $g = 13$. Простыми в рекурсивном построении будем считать только числа $2, 3, 5$ (они сами являются своими сертификатами). Построим дерево и предъявим проверку корректности.



Проверка корректности записана также в вершинах дерева. 2 и 5 — простые по условию числа. Пояснения к вычислениям записаны ниже.

Проверка того, что 3 — первообразный корень по модулю 17:

$$3^2 = 9,\ 3^4 = 81 = -4 \bmod 17,\ 3^8 = (-4)^2 = 16 = -1 \bmod 17,\ 3^{16} = (-1)^2 = 1 \bmod 17$$

Проверка того, что 5 — первообразный корень по модулю 23:

$$5^2 = 25 = 2 \bmod 23,\ 5^4 = 4 \bmod 23,\ 5^8 = 16 \bmod 23$$
$$5^{11} = 5^{8+2+1} = 16 \cdot 2 \cdot 5 = 160 = -1 + 23 \cdot 7 = -1 \bmod 23$$
$$5^{22} = (-1)^2 = 1 \bmod 23$$

Проверка того, что 13 — первообразный корень по модулю 3911:

$$13^2 = 169,\ 13^4 = 1184 \bmod 3911,\ 13^8 = 1718 \bmod 3911,\ 13^{16} = 2630 \bmod 3911$$
$$13^{32} = 2252 \bmod 3911,\ 13^{64} = 2848 \bmod 3911,\ 13^{128} = 3601 \bmod 3911$$
$$13^{256} = 2236 \bmod 3911,\ 13^{512} = 1438 \bmod 3911,\ 13^{1024} = 2836 \bmod 3911$$
$$13^{5 \cdot 17 \cdot 23} = 13^{1955} = 13^{1024+512+256+128+32+2+1} = 3910 \bmod 3911$$
$$13^{2 \cdot 17 \cdot 23} = 13^{782} = 13^{512+256+8+4+2} = 3349 \bmod 3911$$
$$13^{2 \cdot 5 \cdot 23} = 13^{230} = 13^{128+64+32+4+2} = 1346 \bmod 3911$$
$$13^{2 \cdot 5 \cdot 17} = 13^{170} = 13^{128+32+8+2} = 1790 \bmod 3911$$
$$13^{3910} = (-1)^2 = 1 \bmod 3910$$

Системы линейных неравенств

Теперь перейдем к решению системы линейных неравенств над $\mathbb{Q}$

$$\begin{cases} a_{11}x_1 + \ldots + a_{1n}x_n \leqslant b_1 \\ \ldots \\ a_{m1}x_1 + \ldots + a_{mn}x_n \leqslant b_m \end{cases}$$

В будущем будем писать $x \leqslant y$, если каждая компонента x не больше каждой компоненты y.

Определение 5.43. Пусть $A \in \mathrm{Mat}_{n\times n}(\mathbb{Q})$, $b \in \mathbb{Q}^n$. Будем говорить, что система линейных неравенств $Ax \leqslant b$ *совместна*, если существует хотя бы один вектор $u \in \mathbb{Q}^n$, удовлетворяющий $Au \leqslant b$.

Рассмотрим язык L_{cons}, состоящий из кодировок всех совместных систем линейных неравенств ($Ax \leq b$) с целыми коэффициентами. Длиной входа считаем количество коэффициентов умноженное на логарифм максимального по модулю среди них, т.е. $|(A,b)| = m(n+1)\log h$, где $A \in \mathbb{Z}^{m\times n}$, $b \in \mathbb{Z}^m$, $h = \max\{\max_{i,j}|A_{ij}|, \max_i |b_i|\}$. Покажем, что данный язык принадлежит **NP**. Казалось бы, это очевидно: можно предъявить в качестве сертификата такой x, что $Ax \leq b$, причем проверка тоже выполняется за полиномиальное время. В этом рассуждении есть недочет: нам нужно предъявить не произвольный допустимый вектор $x \in \mathbb{Q}^n$, а такой, длина записи которого полиномиально ограничена размером входа. Почему это возможно? Без ограничения общности будем считать, что все столбцы матрицы A линейно независимы (иначе некоторые переменные можно всегда считать равными 0). Заметим, что так как допустимое множество $\{x : Ax \leq b\}$ является полиэдром, то существует решение x, находящееся в его вершине. Это означает, что при соответствующей перестановке строк

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

и $A_1x = b_1$, $A_2x \leq b_2$, где A_1 — невырожденная квадратная матрица. Тогда $x = A_1^{-1}b_1$. Так элементы обратной матрицы выражаются через отношение детерминантов миноров к детерминанту всей матрицы, то длина их битовой записи полиномиально ограничена. Следовательно, полиномиально ограничена и длина записи коэффициентов вектора x. Заметим, что знать конкретное представление через A_1 и A_2, как и линейную зависимость столбцов матрицы нам не нужно — это просто позволяет показать, что найдется подходящий сертификат x, который и можно предъявить.

Теперь покажем, что язык несовместных систем линейных неравенств принадлежит классу **NP**. Сначала оговорим обозначение $x \geqslant y$ для векторов $x, y \in \mathbb{R}^n$: скажем, что $x \geqslant y$, если $x_i \geqslant y_i$ для всех i, то есть x покоординантно больше y. Неравенства в условии

полагаются строгими; любую систему линейных неравенств можно привести к виду $Ax \leqslant b$, домножив некоторые строки на -1.

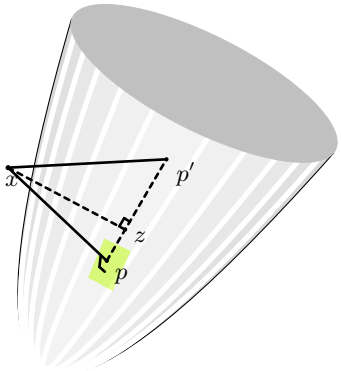
Здесь нужно предъявить предикат вида $\exists x P(x)$, то есть проверить существование некоторого объекта. Мы сможем его построить, если докажем следующую лемму.

Лемма 5.44 (Farkás). *Имеет место следующая альтернатива.*

- *Система* $Ax = b,\ x \geqslant 0$ *совместна;*
- *Существует* $y \in \mathbb{R}^n$ *такой, что* $A^T y \geqslant 0,\ y^T b < 0$.

Сначала мы докажем следующее

Утверждение 5.45. *Даны* S *— замкнутое выпуклое подмножество* $\mathbb{R}^n$ *и* $x \in \mathbb{R}^n$ *лежит вне* S. *Пусть* $p \in S$ *— самая близкая точка* S *к* x, *то есть* $p = \arg\min_{u \in S} ||x - u||$. *Тогда для любой* $z \in S$ *верно* $\langle x - p, z - p \rangle \leqslant 0$.



Доказательство. Условие $\forall z \in S\ \langle x - p, z - p \rangle \leqslant 0$ эквивалентно тому, что либо x и все точки S находятся по разные стороны от касательной в точке p гиперплоскости γ [на иллюстрации отмечена салатовым цветом], либо $\overrightarrow{xp} \perp \overrightarrow{zp}$. Если бы это было не так, то пусть $p' \in S$ лежит по ту же сторону от γ‘, что и x. Опустим перпендикуляр из x на прямую $(p'p)$ и обозначим пересечение за z. Так как p и p' лежат по одну сторону γ, то есть

$$\langle x - p, p' - p \rangle > 0,$$

то z лежит на отрезке $[p'p]$. В силу выпуклости S так как $p, p' \in S$, то $z \in S$. Катет xz является кратчайшей стороной в Δxpz, что противоречит тому, что p — самая близкая точка S к x. □

Доказательство леммы Фаркаша. Пусть система $Ax = b,\ x \geqslant 0$ совместна, докажем наличие y, удовлетворяющего указанным условиям.

Обозначим $A\mathbb{R}^n_+ \stackrel{def}{=} \{Ax | x \geqslant 0\}$, любая точка этого множества имеет вид Aw для $w \geqslant 0$. Очевидно, $A\mathbb{R}^n_+$ — замкнутое множество. Более того, $A\mathbb{R}^n_+$ — выпуклое: для любых $Ax, Ay \in A\mathbb{R}^n_+$ все точки соединяющего их отрезка имеют вид $\lambda Ax + (1 - \lambda)Ay = A\big(\lambda x + (1 - \lambda)y\big)$ [для $\lambda \in [0; 1]$]; так как $\lambda, 1 - \lambda \geqslant 0$, то $\lambda x + (1 - \lambda)y \in \mathbb{R}^n_+$. Кроме того, система $Ax \leqslant b$ несовместна, значит, $b \notin A\mathbb{R}^n_+$.

Теперь пусть $w = \arg\min_{p \in \mathbb{R}^n_+} ||b - Ap||$, тогда $\langle b - Aw, Ax - Aw \rangle \leqslant 0$ для любого $x \geqslant 0$ согласно доказанному выше утверждению. Теперь докажем, что $y = Aw - b$ — искомый.

Действительно,

$$0 \geqslant \langle b - Aw, Ax - Aw \rangle = (b - Aw)^T (Ax - Aw) = (b - Aw)^T A(x - w) \Rightarrow (Aw - b)^T A(x - w) \geqslant 0$$

для любого $x \in \mathbb{R}^n_+$; тогда рассмотрим $x = w + e_i$, где $e_i = (\underbrace{0, \dots 0}_{i-1}, 1, \underbrace{0, \dots 0}_{n-i})$, и получим $(Aw - b)^T Ae_i \geqslant 0$, но $(Aw - b)^T Ae_i$ равен i-ой компоненте, поэтому $y^T A \geqslant 0$. В то же время

$$y^T b = y^T (Aw - y) = y^T Aw - y^T y.$$

Но $(b - Aw)^T A(x - w) \leqslant 0$, подставим $x = 0$ и получим $-(b - Aw)^T Aw = y^T Aw \leqslant 0$; так как $y^T y > 0$ [ведь $Aw - b \neq 0$], то $y^T b < 0$.

Теперь докажем, что в условии действительно альтернатива, то есть оба условия не могут выполняться одновременно. Иными словами, если есть $y \in \mathbb{R}^n$ такой, что $A^T y \geqslant 0$, $y^T b < 0$, то система неравенств не может быть совместной. Если бы система $Ax = b$ была совместна, то $(y^T A)x = y^T (Ax) = y^T b$, $y^T A \geqslant 0$, $y^T b < 0$, тогда $y^T Ax \geqslant 0$ и $y^T Ax < 0$ одновременно. □

Теперь выведем из леммы Фаркаша следующее следствие:

Имеет место следующая альтернатива.

- *Система $Ax \leqslant b$ совместна;*
- *Существует $y \in \mathbb{R}^n_+$ такой, что $A^T y = 0$, $y^T b = -1$.*

Действительно, пусть не существует такого $y \in \mathbb{R}^n_+$, что $A^T y = 0$, $y^T b = -1$. Тогда система

$$z^T \begin{bmatrix} A_{11} & \dots & A_{1n} & b_1 \\ A_{21} & \dots & A_{2n} & b_2 \\ \vdots & \ddots & \vdots & \vdots \\ A_{n1} & \dots & A_{nn} & b_n \end{bmatrix} = \begin{bmatrix} 0 & \dots & 0 & -1 \end{bmatrix}, \quad z \geqslant 0$$

несовместна. Транспонируем равенство выше, получим, что система $A_b z = \begin{bmatrix} 0 & \dots & 0 & -1 \end{bmatrix}$ не имеет решения. Следовательно, существует такой вектор $\begin{bmatrix} y_1 & y_2 & \dots & y_{n+1} \end{bmatrix}^T$ такой, что

$$\begin{bmatrix} A_{11} & \dots & A_{1n} & b_1 \\ A_{21} & \dots & A_{2n} & b_2 \\ \vdots & \ddots & \vdots & \vdots \\ A_{n1} & \dots & A_{nn} & b_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{bmatrix} \geqslant 0, \quad \begin{bmatrix} 0 & \dots & 0 & -1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{bmatrix} < 0.$$

Здесь $y_i \in \mathbb{R}$. Тогда в силу последнего равенства $-y_{n+1} < 0$, то есть $y_{n+1} > 0$; тогда в силу первого равенства

$$\begin{bmatrix} A_{11} & \dots & A_{1n} & b_1 \\ A_{21} & \dots & A_{2n} & b_2 \\ \vdots & \ddots & \vdots & \vdots \\ A_{n1} & \dots & A_{nn} & b_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} A_{11} & \dots & A_{1n} \\ A_{21} & \dots & A_{2n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \dots & A_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} + y_{n+1} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \geqslant 0$$

Тогда $A \begin{bmatrix} y_1 & \dots & y_n \end{bmatrix}^T \geqslant -y_{n+1} b$, и система $Ax \leqslant b$ имеет решение $x = -\begin{bmatrix} \frac{y_1}{y_{n+1}} & \dots & \frac{y_n}{y_{n+1}} \end{bmatrix}^T$. Разумеется, если система $Ax \leqslant b$ совместна, то не существует такого $y \in \mathbb{R}^n_+$, что $A^T y = 0$, $y^T b = -1$; альтернативность доказывается так же, как в лемме Фаркаша.

Итак, получается, что язык несовместных систем неравенств можно задачть предикатом

$$R(\{A, b\}, y) = \exists y \in \mathbb{R}^N, \quad A^T y = 0, y^T b = -1$$

Осталось только проверить, что y полиномиально ограничен по входу, а истинность $R(\{A, b\}, y)$ можно проверить за не более чем полиномиальное время; тем самым мы докажем, что язык несовместных систем неравенств лежит в **NP**.

На вход подаются $A_{ij}, b_i \in \mathbb{Z}$, искомый $y \in \mathbb{Q}^N$; докажем полиномиальность размера сертификата. Пусть $h = \max A_{ij}$; тогда существует такое решение $A^T z = 0$, что

$$\max z_i \leqslant \max_{j_1, \dots j_{\mathrm{rk}(A)}} \det[A_{j_1, \dots j_{\mathrm{rk}(A)}}],$$

где $A_{j_1, \dots j_{\mathrm{rk}(A)}}$ — подматрица A, составленная из столбцов с номерами $j_1, \dots j_{\mathrm{rk}(A)}$. Таким образом, достаточно найти z в кубе $[0; \max_{j_1, \dots j_{\mathrm{rk}(A)}} \det[A_{j_1, \dots j_{\mathrm{rk}(A)}}]]^N$, а затем указать $y = \frac{-z}{\langle z, b \rangle}$ [конечно, если все координаты положительны]. Граница на координаты z полиномиально ограничена [некоторым минором поданной матрицы], таким образом, координаты y также ограничены полиномиально по входу.

Истинность же предиката на A, y, b можно проверить за полиномиальное время: нужно лишь вычислить $A^T y$ и $y^T b$, на вычисление их нужно не более $O(n^3)$ арифметических операций [сложения и умножения], арфиметические операции на числах не более M по сложности не превышают $(\log M)^2$. При все компоненты $A^T y$ полиномиально ограничены [например, $n \max |A_{ij}| \max |y_i|$], аналогично

$$|y^T b| = |y_1 b_1 + \dots + y_n b_n| \leqslant n \max |y_i| \max |b_i|.$$

Значит, проверка истинности $R(\{A, b\}, y)$ полиномиальна по времени.

Полиномиальная сводимость

Алгоритмы, решающие конкретную задачу, можно частично упорядочить, основываясь на их сложности. Классы задач можно частично упорядочить по включению, например, $\mathbf{P} \in \mathbf{NP}$, а $\mathbf{NP}$, в свою очередь, вложен в $\mathbf{PSPACE}$. Но как сравнить между собой разные задачи? Например, очевидно, что задача поиска медианы проще, чем задача сортировки массива, но как это формализовать? Для этих целей вводится понятие *сводимости*.

Первое, вполне естественное определение — это сводимость по Куку.

Определение 5.46. Будем говорить, что язык L_1 *сводится по Куку* к L_2, если существует алгоритм, который может обращаться к оракулу для L_2 (т.е. вызывать функцию $f(y) = \mathbb{I}_{L_2}(y)$) и разрешает L_1 за *полиномиальное* время при условии, что каждый вызов оракула требует $\Theta(1)$ времени.

То есть, если мы каким-то образом можем быстро узнать, принадлежит ли произвольное слово y языку L_2, то можно написать программу, которая проверяет $x \in L_1$ за полиномиальное время. Однако, оказывается, что эта сводимость зачастую слишком сильная (так же, как сводимость по Тьюрингу): она действует между разными уровнями полиномиальной иерархии, но не позволяет детально исследовать один уровень. Например, очевидно, что любой язык L сводится по Куку к своему дополнению $\overline{L}$, поэтому, в частности, любой язык из $\mathbf{NP}$ сводится к некоторому языку из $\mathbf{co-NP}$, и наоборот. Это означает, что если мы хотим различать классы $\mathbf{NP}$ и $\mathbf{co-NP}$ (с оговоркой, что мы не знаем, совпадают они или нет), то нельзя отождествлять языки, которые сводятся друг к другу по Куку.

Другой вид сходимости, который мы и будем использовать — это *полиномиальная сводимость* (или *сводимость по Карпу*).

Определение 5.47. Пусть $L_1 \subset \Sigma^*$, $L_2 \subset \Gamma^*$, говорим, что L_1 *полиномиально сводится к* L_2, если существует вычислимая всюду определенная функция $f : \Sigma^* \to \Gamma^*$ такая, что

$$\forall x \in \Sigma^* \quad x \in L_1 \iff f(x) \in L_2$$

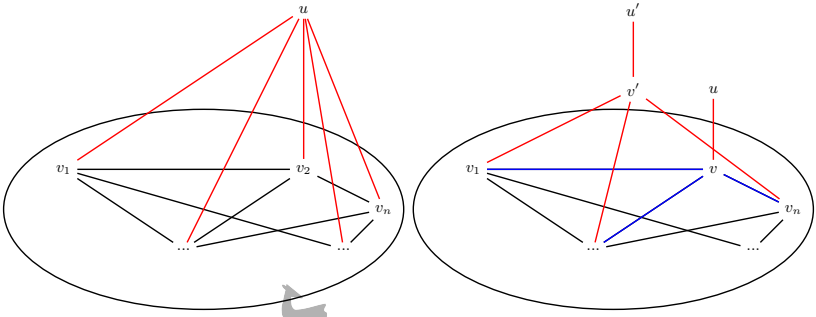
Выражение $L_1 \leqslant_m L_2$ понимается как «L_1 p-сводится к L_2».

Полиномиальная сводимость обладает следующими свойствами:

(1) $L \leqslant_P L$ для любого языка L (рефлексивность);
(2) если $L_1 \leqslant_P L_2$ и $L_2 \leqslant_P L_3$, то $L_1 \leqslant_P L_3$ (транзитивность);
(3) если $L_1 \leqslant_P L_2$, то $\overline{L}_1 \leqslant_P \overline{L}_2$;
(4) если $L_1 \leqslant L_2 \in \mathbf{P}$, то $L_1 \in \mathbf{P}$;
(5) если $L_1 \leqslant L_2 \in \mathbf{NP}$, то $L_1 \in \mathbf{NP}$;
(6) если $L_1 \leqslant L_2 \in \mathbf{co-NP}$, то $L_1 \in \mathbf{co-NP}$.

Сложность конструктивного решения задачи можно «распределить» между конструкцией и доказательством: либо с большими трудностями доказывать корректность тривиальной конструкции, либо создать громоздкую конструкцию, доказательство корректности которой лежит на поверхности. И обычно, строя сводимости, люди избирают второй путь.

Пример 5.48. Язык ГП состоит из всех графов, имеющих гамильтонов путь (несамопересекающийся путь, проходящий через все вершины графа). Язык ГЦ состоит из всех графов, имеющих гамильтонов цикл (цикл, проходящий через все вершины, в котором все вершины, кроме первой и последней, попарно различны). Построим явную сводимость языков ГЦ и ГП друг к другу.



(а) HP $\leqslant_p$ HC (в) HC $\leqslant_p$ HP

Сводимость ГЦ к ГП строится тривиально: графу $G = (V, E)$ сопоставим граф $G_u = (V \cup \{u\}, E \cup \{(xu) | x \in V\})$; иными словами, получим новый граф из G добавлением новой вершины, смежной со всеми вершинами старого графа. Тогда G_u содержит гамильтонов цикл титтк G содержит гамильтонов путь: пусть в гамильтоновом цикле в G_u вершины расположены в порядке $u, v_1, \ldots, v_n, u$, тогда $v_1, \ldots, v_n$ — гамильтонов путь; если же $v_1, \ldots, v_n$ — гамильтонов путь в G, то u замыкает этот путь в гамильтонов цикл в G. Несложно убедиться, что такая сводимость полиномиальна [детерминированный алгоритм добавляет новую строчку и новый столбец в матрицу смежности графа, заполненные единицами].

Теперь по $G = (V, E)$ и некоторой v [например, первой по порядковому номеру — положим, что вершины занумерованы, а v — та, что имеет номер 1] построим новый граф

$$G_v = (V \cup \{u, v', u'\}, E \cup \{(v'x) | x \in V,\ \exists (vx) \in E\} \cup \{(vu), (v'u')\}).$$

Говоря проще, мы «копируем» вершину v, добавляя новую вершину v' с теми же соседями, что и v, а затем достраиваем две висячие вершины. Конструкция здесь, как вообще в сходимостях, относительно сложна, зато доказательство корректности становится тривиальным и практически очевидным. Действительно, если в G существовал гамильтонов цикл, то пусть он прходил вершины графа в последовательности $v_1 \to \ldots \to v \to v_1$ (то есть v — последняя непосещенная вершина графа), тогда

$$u' \to v' \to v_1 \to \ldots \to v \to u$$

— гамильтонов путь: мы посетили u', v' и u лишь однажды, прошли по гамильтонову циклу в G, не проходя последнее ребро, то есть посетили все вершины оригинального графа. Напротив, если в G_v имеется гамильтонов путь, то его концы есть непременно u и u' (эти вершины висячие), и если гамильтонов путь начинается в u', то вторая по порядку вершина равна v'; пусть гамильтонов путь в G_v имеет вид

$$u' \to v' \to v_1 \to \ldots \to v \to u,$$

тогда

$$v_1 \to \ldots \to v \to v_1$$

есть гамильтонов цикл в G: мы просто убрали u', v' и u, получили часть гамильтонова пути, проходящую по всем вершинам G и замыкаем ее в цикл, так как v_1 и v смежны.

Определение 5.49. Пусть дан некоторый класс языков $\mathcal{C}$. Язык L называется $\mathcal{C}$-*трудным*, если любой язык $L' \in \mathcal{C}$ полиномиально сводится к L. Язык L называется $\mathcal{C}$-*полным*, если $L \in \mathcal{C}$ и он является $\mathcal{C}$-трудным.

Понятие полноты аналогично понятию универсального множества для перечислимых (или коперечислимых) языков.

Очевидны следующие свойства полных и трудных языков:

(1) если $L_1 \leqslant_P L_2$ и L_1 — $\mathcal{C}$-трудный, то L_2 тоже $\mathcal{C}$-трудный в силу транзитивности;
(2) если L — $\mathcal{C}$-трудный (полный), то $\overline{L}$ — $co-\mathcal{C}$-трудный (полный).

Рассмотрим класс $\mathbf{P}$. Кажется, все языки должны быть $\mathbf{P}$-трудными, ведь мы можем просто проверить, принадлежит ли слово языку $L \in \mathbf{P}$, за полиномиальное время. Однако, следует помнить о двух исключениях — тривиальных языках $\emptyset$ и Σ^*. Из определения сводимости по Карпу сразу следует, что $L \leqslant_P \emptyset$ тогда и только тогда, когда $L = \emptyset$. Аналогично, единственный язык, сводящийся к Σ^* — это он сам. Все же остальные языки действительно являются $\mathbf{P}$-трудными.

Менее очевидно, как обстоят дела с классами $\mathbf{NP}$, $\mathbf{co-NP}$ и т.д., есть ли для них трудные или полные языки? Оказывается, есть. Классическим примером $\mathbf{NP}$-полного языка служит SAT, т.е. язык всех выполнимых булевых формул. Приведем набросок доказательства того, что SAT является $\mathbf{NP}$-трудным.

Теорема 5.50 (теорема Кука-Левина). *Определим SAT — язык булевых формул в конъюктивной нормальной форме, которые* выполнимы, *то есть равны 1 хотя бы на одном наборе значений переменных. Язык SAT является NP-полным.*

Доказательство. Пусть L — некоторый язык из **NP**, а M — недетерминированная МТ, распознающая его за время $T_M(n) \leqslant P(n)$. Пусть Q — множество состояний данной МТ, A — ее алфавит ($\Sigma \subset A$). Для слова $x \in \Sigma^*$ построим соответствующую булеву формулу $F_{M,x} = f(x)$. Во-первых, модифицируем МТ таким образом, чтобы после перехода в финальное состояние она зацикливалась в нем, т.е. добавим переходы вида $(q_A, a) \to (q_A, a, 0)$ и $(q_R, a) \to (q_R, a, 0)$. Затем определим переменные, над которыми мы построим формулу. Для начала введем переменные $y^t_{i,a}$, $t, |i| \leqslant P(|x|)$, $a \in A$, которые будут отвечать за то, что на шаге t в позиции i на ленте записан символ a. Переменные z^t_i будут индикаторами того, что на шаге t головка M находится в позиции i. Наконец, u^t_q будут показывать, что на t-м шаге состояние машины есть q. Формулу $F_{M,x}$ будем строить как конъюнкцию элементарных выражений, отвечающих за начальное состояние и переходы МТ. Во-первых, построим часть формулы, отвечающую за начальное состояние:

$$F_{start} := \left[\bigwedge_{i=0}^{|x|-1} \left(y^0_{i,x_i} \wedge \bigwedge_{a \neq x_i} \overline{y^0_{i,a}}\right)\right] \wedge \left[z^0_0 \wedge \bigwedge_{|i| \leqslant P(|x|),\, i \neq 0} \overline{z^0_i}\right] \wedge \left[u^0_{q_0} \bigwedge_{q \in Q \setminus \{q_0\}} \overline{u^0_q}\right].$$

Затем для каждого шага $t \leqslant P(|x|)$ будем добавлять подформулу F^t_{trans}, отвечающую за корректность перехода. Заметим, что МТ может менять состояние ленты только локально, поэтому для ячеек, которые достаточно далеко от головки, нужно просто сохранить текущий символ:

$$\overline{\left(z^t_{i-1} \vee z^t_i \vee z^t_{i+1}\right)} \to (y^{t+1}_{i,a} \equiv y^t_{i,a} \wedge z^{t+1}_i \equiv z^t_i).$$

Для тех же ячеек, которые находятся близко к головке, требуется записать все допустимые изменения согласно правилам перехода данной МТ:

$$(z^t_i \wedge u^t_q \wedge y^t_{i,a}) \to \bigvee_{(q',b's) \in \delta(q,a)} \ldots,$$

где $\delta(q, a)$ — множество всех допустимых переходов из состояния головки q и символа на ленте a. Наконец, добавим проверку на то, что машина приняла слово: $u^{P(|x|)}_{q_A} \equiv 1$. Построенная таким образом формула

$$F_{M,x} = F_{start} \wedge \left(\bigwedge_{0 \leqslant t < P(|x|)} F^t_{trans}\right) \wedge u^{P(|x|)}_{q_A}$$

будет, как несложно убедиться, выполнимой тогда и только тогда, когда x принимается M. При этом, данную формулу можно построить за полиномиальное по $|x|$ время. Следовательно, $L \leqslant_P \mathsf{SAT}$. В то же время, $\mathsf{SAT} \in \mathbf{NP}$, поэтому он является **NP**-полным (множество всех **NP**-полных языков будем обозначать **NPC**).

Более того, можно показать, что язык всех выполнимых формул, заданных в КНФ, тоже является **NP**-полным. Действительно, мы построили формулу, которая имеет вид конъюнкции некоторых клауз, каждая из которых имеет длину $O(1)$. Следовательно, если мы заменим каждую из них на КНФ, то длина ее записи все равно будет ограничена константой (напомню, что длина КНФ, в общем случае, может быть экспоненциально больше, чем длина исходной формулы), и длина всей полученной формулы (в КНФ) останется полиномиально ограниченной. *В дальнейшем под* SAT *мы будем подразумевать язык выполнимых КНФ!* Можно упрощать задачу и дальше, например, **NP**-полными являются следующие языки:

- $3 - \mathsf{SAT}$, состоящий из выполнимых КНФ, в каждом дизъюнкте которых не больше трех литералов;
- $3\mathsf{E} - \mathsf{SAT}$, состоящий из выполнимых КНФ, в каждом дизъюнкте которых ровно 3 литерала;
- язык выполнимых КНФ, в каждом дизъюнкте которых не больше трех литералов, и при этом каждая переменная входит в формулу не больше трех раз, а каждый литерал — не более двух.

При этом, задача $2 - \mathsf{SAT}$, когда в каждый дизъюнкт входит не больше двух литералов, является полиномиально разрешимой. Можно построить следующий полиномиальный алгоритм. Заметим, что конъюнкция эквивалентна импликациям: $(x \vee y) = (\bar{x} \to y) = (\bar{y} \to x)$. Построим орграф, где вершины соответствуют переменным и их отрицаниям, а ребро между вершинами отвечает какой-то импликации. Затем сконденсируем данный граф, то есть найдем его компоненты сильной связности, что можно сделать за полиномиальное время. Соответственно, исходная КНФ выполнима тогда и только тогда, когда никакая переменная x не находится в одной компоненте сильной связности с ее отрицанием $\bar{x}$, что можно проверить за, опять же, полиномиальное время.

Еще одна связанная задача — $\mathsf{MAX} - 2 - \mathsf{SAT}$: дана КНФ, в каждом дизъюнкте которой не больше 2 литералов, требуется найти число выполняющих наборов переменных для данной КНФ. Эта задача является **NP**-трудной, точнее, **NP**-трудна соответствующая задача разрешения: верно ли, что число выполняющих наборов не меньше k. Сама же оптимизационная задача является полной в классе $\mathcal{APX}$ (оптимизационные задачи, которые могут быть решены с точностью до мультипликативной константы за полиномиальное время). □

Из этой теоремы можно вывести, что и некоторые другие языки также **NP**-полны.

- Протыкающее множество — язык

$$\Big\{(\{A_1,\ldots,A_m\},k)\ \Big|\ A_i \text{ — конечное множество, } k\in\mathbb{N},\ \exists B,\ |B|=k,\ B\cap A_i\neq\varnothing\Big\};$$

- Клика — язык

$$\Big\{(G,k)\ \Big|\ G \text{ — граф, } k\in\mathbb{N},\ \exists K_k,\ K_k \text{ изоморфен некоторому подграфу } G\Big\}.$$

  (G,k) лежит в *Клика*, если в G есть полный подграф на k вершинах;
- Вершинное покрытие — язык

$$\Big\{(G,k)\ \Big|\ G \text{ — граф, в котором существует вершинное покрытие мощности } k\Big\}.$$

  Напомним, что вершинное покрытие мощности k в графе G — множество $S\subset V$ мощности k такое, что любое ребро смежно хотя бы одной вершине S;
- Гамильтонов цикл — язык графов, содержащих гамильтонов цикл, то есть допускающих циклический обход всех вершин графа, проходящий каждую вершину однажды;
- Разбиение — язык

$$\Big\{(\{a_1,\ldots a_n\})\ \Big|\ \forall i\ a_i\in\mathbb{N},\ \exists B\subset[1;n],\ \sum_{i\in B}a_i=\sum_{i\notin B}a_i\Big\}.$$

  Иными словами, множество $\{a_1,\ldots a_n\}$ принадлежит этому языку, если его можно разбить на два множества одинакового веса;
- 3-сочетание — язык

$$\Big\{(M,W,X,Y)\ \Big|\ |W|=|X|=|Y|=q,\ M\subseteq W\times X\times Y,\ \exists M'\subseteq M,\ |M'|\in q,$$
$$\forall(x_1,y_1,z_1),(x_2,y_2,z_3)\in M'\quad x_1\neq x_2,y_1\neq y_2,z_1\neq z_2\Big\}.$$

  Задача поиска 3-сочетания формулируется так: дано множество $M\subseteq W\times X\times Y$, где W,X и Y — непересекающиеся множества, содержащие одинаковое число элементов q, надо найти трехмерное сочетание, то есть такое подмножество $M'\subseteq M$ мощности q, никакие два элемента которого не имеют ни одной одинаковой координаты;
- Рюкзак — язык

$$\Big\{(\{a_1,\ldots,a_n\},b)\ \Big|\ a_1,\ldots a_n,b\in\mathbb{N},\ \exists i_1,\ldots i_k\in[1;n]\quad a_{i_1}+\ldots+a_{i_k}=b\Big\};$$

- $\mathsf{MAX}-2-\mathsf{SAT}$ — язык

$$\Big\{(c_1\wedge\ldots c_n,k)\ \Big|\ c_i \text{ — 2-конъюнкция, } k\in\mathbb{N},\ \exists j_1,\ldots j_k\in[1;n]$$
$$c_{j_1},\ldots c_{j_k} \text{ выполняются при некотором значении переменных}\Big\};$$

- Максимальный разрез — язык

$$\Big\{(G,k)\ \Big|\ G=(V,E),\ k\in\mathbb{N},\exists A\subset V,\ |\{(xy)\in E|x\in A, y\notin A\}|\geqslant k\Big\}.$$

(G,k) лежит в языке тогда и только тогда, когда множество вершин графа можно разбить на два непересекающихся подмножества, между которыми можно провести не менее k ребер.

Покажем, что некоторые из этих языков также **NP**-полны.

Теорема 5.51. *Пусть* Ровно-3-выполнимость *— язык КНФ, в которых каждый дизъюнкт содержит ровно три различных литерала. Язык* $3-\mathsf{SAT}$ *полиномиально сводится к языку* Ровно-3-выполнимость.

Доказательство. Пусть нам дана КНФ ϕ, в которой некоторые дизъюнкты, вообще говоря, могут содержать не более трех различных переменных (одну или две, например), построим по ней $\hat{\phi}$ и покажем, что

$$\phi\in 3-\mathsf{SAT}\Leftrightarrow\hat{\phi}\in\text{Ровно-3-выполнимость}$$

Приведем ϕ к КНФ ϕ' заменой каждого дизъюнкта на дизъюнкт, содержащий только различные переменные. Например, $x\vee y\vee\overline{y}$ перейдет в x, а $x\vee x\vee y$ перейдет в $x\vee y$.

Теперь построим $\hat{\phi}$ по ϕ', достроив каждый дизъюнкт до 3-дизъюнкта, не меняя области истинности. Пусть $\phi'=\alpha_1\wedge\ldots\alpha_n\wedge\beta_1\wedge\ldots\wedge\beta_k\wedge\gamma_1\wedge\ldots\wedge\gamma_l$, где α_i содержат одну переменную, β_i — две, а γ_i — три. Добавим переменные A,a,b и для каждого β_i введем

$$\widetilde{\beta_i}=(\beta\vee\neg A)\wedge(A\vee a\vee b)\wedge(A\vee\neg a\vee b)\wedge(A\vee a\vee\neg b)\wedge(A\vee\neg a\vee\neg b).$$

Аналогично добавим переменные B,C,c,d,e,f и заменим каждый α_i на $\widetilde{\alpha_i}$ [получится конъюнкция $\alpha\vee\neg B\vee\neg C$ и дизъюкнтов вида $B\vee c^{\pm1}\vee d^{\pm1}$ и $C\vee e^{\pm1}\vee f^{\pm1}$].

Докажем, что $\hat{\phi}=\widetilde{\alpha_1}\wedge\ldots\widetilde{\alpha_n}\wedge\widetilde{\beta_1}\wedge\ldots\wedge\widetilde{\beta_k}\wedge\gamma_1\wedge\ldots\wedge\gamma_l$ — искомая. Действительно, $\hat{\phi}$ является ровно-3-КНФ, а ее область истинности совпадает с областью истинности ϕ: так как все дизъюнкты вида $A\vee a^{\pm1}\vee\neg b^{\pm1}$ одновременно верны только при $A=1$, то дизъюнкт $\beta_i\vee\neg A$ выполнен тогда выполнен β_i; ситуация с α_i разбирается аналогично. □

В следующих нескольких утверждениях $\psi(x_1,x_2,x_3)=x_1\vee x_2\vee\neg x_3$ и $\chi(x_1,x_2)=(x_1\vee x_2)\wedge(x_1\vee\neg x_2)\wedge\neg x_1$. Нам предлагается на этих конкретных примерах понять полиномиальные сводимости. Заметим, что $\psi(1,1,1)=1$, поэтому ψ выполнима, а χ никогда не выполнима: первые два монома выполнимы одновременно только при $x_1=1$, но тогда неверен последний дизъюнкт.

Теорема 5.52. *Язык* $3-\mathsf{SAT}$ *полиномиально сводится к языку* Вершинное покрытие.

Доказательство. Считаем без ограничения общности, что нам дана ровно-3-КНФ

$$\phi = \phi_1 \wedge \ldots \wedge \phi_m, \quad \phi_i = \phi_i[1] \vee \phi_i[2] \vee \phi_i[3], \quad \phi_i[j] \in \{x_1, \ldots x_n\}.$$

По формуле ϕ строится граф $G_\phi = (V_\phi, E_\phi)$, где множество вершин

$$V_\phi = \left(\bigcup_{i \in [1;n]} \{[x_i], [\neg x_i]\} \right) \cup \left(\bigcup_{i \in [1;m]} \{[\phi_i[1]], [\phi_i[2]], [\phi_i[3]]\} \right),$$

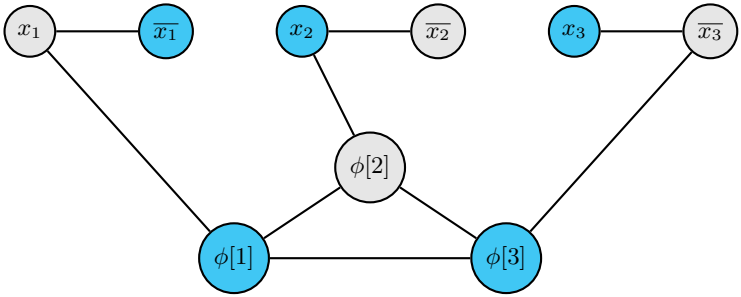
а ребра

$$E_\phi = \left(\bigcup_{i \in [1;n]} \left\{ ([x_i], [\neg x_i]) \right\} \right) \cup \left(\bigcup_{i \in [1;m], j_1, j_2 \in \{1,2,3\}} \left\{ ([\phi_i[j_1]], [\phi_i[j_2]]) \right\} \right) \cup$$
$$\left(\bigcup_{j \in [1;m],\ k \in \{1,2,3\}} \left\{ ([x_i], [\phi_j[k]]) | \phi_j[k] \equiv x_i \right\} \right).$$

Говоря проще, мы строим граф G_ϕ таким образом. Для каждой логической переменной x_i образуем пару смежных «литеральных» вершин, помеченных, соответственно, x_i и $\neg x_i$, а для каждого 3-дизъюнкта ϕ_i образуем три смежные «дизъюнктные» вершины, помеченных переменными этого дизъюнкта; каждую дизъюнктную вершину соединим с соответствующей литеральной вершиной, имеющей ту же метку. Если ϕ имела m дизъюктов, то, по построению, G_ϕ имеет $2n + 3m$ вершин и не более $\binom{2n+3m}{2}$ ребер. Поэтому построение графа G_ϕ — полиномиальная по длине формулы ϕ процедура: $n = O(|\phi|)$, $m = O(|\phi|)$.

Читатель приглашается самостоятельно проверить, что отображение $\phi \mapsto G_\phi$ действительно задает сводимость, а в этой задаче мы покажем, как такое доказательство работает на конкретных примерах.

(а) Для $\psi(x_1, x_2, x_3) = x_1 \vee x_2 \vee \neg x_3$ надо найти 5-вершинное покрытие. Вот оно:



Несложно убедиться в том, что выделенные голубым цветом вершины доставляют 5-покрытие. Заметим, что мы выбрали набор значений $x_1 = 0$, $x_2 = 1$, $x_3 = 1$

и соответствующим образом выбрали вершины: если $x_i = 1$, то мы выбираем $[x_i]$ в покрытие, в противном случае выбираем $[\neg x_i]$. Выбрав по n «литеральных» вершин, мы просто добавляем еще $2m$ «дизъюнктных» вершин, чтобы покрыть остальные ребра — по два в каждый «дизъюнктный» треугольник.

(b) Заметим сначала, что любое ребро $\big([x_i], [\neg x_i]\big)$ должно быть покрыто хотя бы одной вершиной, а каждый треугольник с вершинами в $[\psi_i[1]]$, $[\psi_i[2]]$ и $[\psi_i[3]]$ должен быть покрыт хотя бы двумя вершинами — одна вершина покрывает ровно две стороны из трех. Тогда независимо от того, на какую эквивалентную ровно-3-КНФ мы заменим исходную χ (а таких могло быть очень много, например, $\chi(x_1, x_2) \vee \psi(y_1^{(1)}, y_2^{(1)}, y_3^{(1)}) \vee \ldots \vee \psi(y_1^{(N)}, y_2^{(N)}, y_3^{(N)})$, где ψ — 3-КНФ, которая рассматривалась пунктом выше, $y_j^{(i)}$ — отличные от x_1 и x_2 переменные, а N может быть любым), вершинное покрытие будет содержать не менее $n_{new} + 2m_{new}$ вершин. Покажем, что ровно столько вершин в вершинном покрытии G_χ не может содержаться (опять-таки, независимо от ровно-3-КНФ, на которую мы заменили ξ).

Воистину, если в покрытии ровно $n_{new} + 2m_{new}$ вершин, то из каждых двух $[x_i]$ и $[\neg x_i]$ выбрана ровно одна (покрывающая ребро $\big([x_i], [\neg x_i]\big)$). Тогда подставим в χ значения $x_i = 1$, если в покрытии выбрана вершина $[x_i]$, и $x_i = 0$, если в покрытии выбрана вершина $[\neg x_i]$, и выберем из каждого треугольника по две верштны, чтобы покрыть оставшиеся ребра вида $\big([x_i^{\pm}], [\phi_j[k]]\big)$; покажем, что при полученном наборе значений переменных $\chi = 1$. Действительно, минимальное покрытие покрывает все ребра вида $\big([x_i], [\neg x_i]\big)$ и треугольники вида $\{\phi_i[1], \phi_i[2], \phi_i[3]\}$, две из трех выбранных вершин в треугольнике покрывают два «исходящих» ребра $\big([x_i], [\phi_j[k]]\big)$, поэтому третье «исходящее» ребро должно быть покрыто «литеральной вершиной» $[x_i]$ (или $[\neg x_i]$). Поэтому каждый дизъюнкт удовлетворяется хотя бы одной переменной (или ее отрицанием), таким образом, $\phi = 1$ на текущем наборе. Выше мы уже показали, что такого набора переменных не существует.

Резюмируя, можно сказать следующее: на языке графов выполнимость КНФ выражается как «хотя бы одно исходящее ребро каждого дизъюнктного треугольника покрывается литеральной вершиной».

□

Теорема 5.53. *Язык* $3 - \mathsf{SAT}$ *полиномиально сводится к языку* Клика.

Доказательство. Вообще мы можем ни черта не делать, имея на руках решение прошлой задачи, ведь следующие два утверждения эквивалентны:

- $R \subset V(G)$ — вершинное покрытие графа G;

- $V(G)\backslash R \subset V(G^{\complement})$ — клика в дополнении графа $G^{\complement} = (V, E^{\complement})$, где $E^{\complement} = \{(xy) | x, y \in V, (xy) \notin E\}$.

Действительно, $R \subset V(G)$ — вершинное покрытие графа G тогда $V(G) \setminus R \subset V(G)$ является независимым множеством: если некоторые две вершины $v_1, v_2 \in V(G) \setminus R$ соединены ребром, то хотя бы одна из этих двух вершин лежит в R, что невозможно; обратное верно аналогично. И $V(G) \setminus R \subset V(G)$ является независимым множеством тогда $V(G) \setminus R \subset V(G^{\complement})$ образует клику в дополнении к графу: все несмежные вершины в G будут смежны в $G^{\complement}$ и наоборот.

(a) Для $\psi(x_1, x_2, x_3) = x_1 \vee x_2 \vee \neg x_3$ надо найти 4-клику в $\tilde{G}_\psi$. Возьмем $\tilde{G}_\psi = G_\psi^{\complement}$, а на рисунке в прошлой задаче были не выбраны вершины $[\neg x_1]$, $[\neg x_2]$, $[x_3]$ и $[\phi[2]]$. Они попарно несмежны в G_ψ, следовательно, в $\tilde{G}_\psi$ они образуют 4-клику.

(b) Комбинируем второй пункт предыдущей задачи с доказанным в начале этой задачи утверждение и сразу приходим к успеху: любое вершинное покрытие G должно было иметь более $n_{new} + 2m_{new}$, значит, в дополнении клика будет содержать не более $n_{new} + m_{new}$ вершин.

□

Теорема 5.54. *Язык* Гамильтонов цикл **NP**-*полон.*

Доказательство. Можно напрямую построить полиномиальную сводимость $3-\mathsf{SAT}$ к Гамильтонов цикл. Доказательство можно найти в [36] или [34]. □

Теорема 5.55. *Язык* $3-\mathsf{SAT}$ *полиномиально сводится к языку* $\mathsf{MAX}-2-\mathsf{SAT}$.

Доказательство. По ровно-3-КНФ $\phi = \phi_1 \wedge \ldots \wedge \phi_n$ построим 2-КНФ $\Phi = \Phi_1 \wedge \ldots \wedge \Phi_n$, где для $\phi_i = a_i \vee b_i \vee c_i$ (a_i, b_i, c_i — переменные или их отрицания) Φ_i — конъюнкция следующих десяти дизъюнктов

$$a_i, b_i, c_i, d_i, \quad \neg a_i \vee \neg b_i, \neg b_i \vee \neg c_i, \neg c_i \vee \neg a_i, \quad a_i \vee \neg d_i, b_i \vee \neg d_i, c_i \vee \neg d_i.$$

Доказательство сводимости основано на следующем утверждении: ϕ_i выполнима тогда при некотором значении d_i в Φ_i выполнимо не менее q дизъюнктов. Утверждается, что $q = 7$. Действительно, если хотя бы какой-то из дизъюнктов a_i, b_i и c_i равен 1 и влечет таким образом выполнение ϕ_i, то можно подобрать некоторое d_i:

- один из трех a_i, b_i, c_i верен, считаем без ограничения общности, что $a_i = 1$, $b_i = c_i = 0$, тогда при $d_i = 0$ верны

$$a_i, \neg a_i \vee \neg b_i, \neg b_i \vee \neg c_i, \neg c_i \vee \neg a_i, a_i \vee \neg d_i, b_i \vee \neg d_i, c_i \vee \neg d_i,$$

то есть 7 штук [а при $d_i = 1$ станет выполнимым d_i и перестанет $b_i \vee \neg d_i$, так что число выполнимых дизъюнктов не увеличится];

- два из трех a_i, b_i, c_i верен, считаем без ограничения общности, что $a_i = b_i = 1$, $c_i = 0$, тогда при $d_i = 1$ верны

$$a_i, b_i, d_i, \neg b_i \vee \neg c_i, \neg c_i \vee \neg a_i, \quad a_i \vee \neg d_i, b_i \vee \neg d_i,$$

  и снова выполнены 7 дизъюнктов [а при $d_i = 0$ перестанет выполняться d_i и станет верным $c_i \vee \neg d_i$, число выполнимых дизъюкнтов не превысит 7];
- все три a_i, b_i, c_i выполнены, тогда при $d_i = 1$ верны

$$a_i, b_i, c_i, d_i, a_i \vee \neg d_i, b_i \vee \neg d_i, c_i \vee \neg d_i,$$

  то есть 7 штук [а при $d_i = 0$ не выполнялся бы d_i, истинность остальных дизъюнктов сохранилась].

Нерассмотренные случаи значения переменных симметричны рассмотренным.

А если $a_i = b_i = c_i = 0$, то при $d_i = 1$ дизъюнкты a_i, b_i, c_i, d_i не будут выполнены, а при $d_i = 0$ не выполнены a_i, b_i, c_i и все дизъюнкты, содержащие d_i; таким образом, не более 6 дизъюнктов будут выполнены. Более того, в $\Phi = \Phi_1 \wedge \ldots \wedge \Phi_n$ будут выполнимы $7n$ дизъюнктов титтк в каждом Φ_i выполнимо 7 дизъюнктов [мы перебрали все варианты выше, больше 7 дизъюнктов не могут верны одновременно], что эквивалентно одновременной выполнимости всех ϕ_i.

Продемонстрируем эту сводимость на конкретных примерах. Для $\psi = x_1 \vee x_2 \vee \neg x_3$ имеем

$$\tilde{\psi} = x_1 \wedge x_2 \wedge \neg x_3 \wedge y \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_3 \vee \neg x_1) \wedge (x_1 \vee \neg y) \wedge (x_2 \vee \neg y) \wedge (\neg x_3 \vee \neg y$$

При $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, $y = 1$ выполняется хотя бы 7 дизъюнктов формулы $\tilde{\psi}$. □

Теорема 5.56. *Язык* Разбиение **NP**-*полон.*

Доказательство. Сведём язык $3 - \mathsf{SAT}$ к языку Разбиение. Рассмотрим произвольную 3-КНФ φ, в которой есть n переменных $x_1, \ldots, x_n$ и m дизъюнктов $c_1, \ldots, c_m$. Построим множество S и число σ следующим образом. Для каждой переменной построим два числа длины $n+m$ в десятичной системе счисления: t_i и f_i. Зададим их по правилам:

- для $i = 1, 2, \ldots, n$ i-е цифры чисел t_i и f_i равны единице;
- число t_i имеет j-ю цифру равную 1 для $j = n+1, n+2, \ldots, n+m$, если литерал x_i есть в дизъюнкте c_{j-n};
- число f_i имеет j-ю цифру равную 1 для $j = n+1, n+2, \ldots, n+m$, если литерал $\neg x_i$ есть в дизъюнкте c_{j-n};
- остальные цифры чисел t_i и f_i положим равными нулю.

Получили $2n$ чисел. Добавим во множество S ещё $2m$ чисел длины $n+m$. Для каждого дизъюнкта c_i построим числа y_i и z_i, которые имеют $(n+i)$-ю цифры равную единице, а остальные цифры — нули. Например, для 3-КНФ $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$ мы получим следующие числа (для удобства запишем это всё в

виде таблицы: в каждой строчке, начиная со второй, записаны соответствующие числа; после первых n цифр запишем разделитель опять-таки исключительно для удобства).

Числом σ у нас будет число длины $n+m$ в десятичной записи, у которого i-я цифра равна 1 для $1 \leqslant i \leqslant n$ и j-я цифра равна 3 для $n+1 \leqslant j \leqslant n+m$. Таким образом, мы построили полиномиально вычислимую сводящую функцию. Утверждается, что исходная 3-КНФ выполнима тогда и только тогда, когда полученная пара (S,t) лежит в Разбиение.

	1	2	3	#	1	2	3	4
t_1	1	0	0	#	1	0	0	1
f_1	1	0	0	#	0	1	1	0
t_2	0	1	0	#	1	0	1	0
f_2	0	1	0	#	0	1	0	1
t_3	0	0	1	#	1	1	0	1
f_3	0	0	1	#	0	0	1	0
#	#	#	#	#	#	#	#	#
y_1	0	0	0	#	1	0	0	0
z_1	0	0	0	#	1	0	0	0
y_2	0	0	0	#	0	1	0	0
z_2	0	0	0	#	0	1	0	0
y_3	0	0	0	#	0	0	1	0
z_3	0	0	0	#	0	0	1	0
y_4	0	0	0	#	0	0	0	1
z_4	0	0	0	#	0	0	0	1

Во-первых, если формула выполнима, то нужно взять в сумму t_i, если $x_i = 1$ на выполняющем наборе, и f_i — в противном случае; кроме того, возьмём y_i в сумму, если в c_i не более двух литералов равно единице, а если ровно один литерал — возьмём ещё и y_i.

Тогда первые n цифр суммы будет равны единице, а последние m цифр будут равны 3, что и требовалось, то есть (S,σ) лежит в Разбиение.

Обратно: пусть (S,σ) лежит в Разбиение. Покажем, что формула выполнима. Выполняющий набор строится по слагаемым, входящим в сумму точно так же, как мы выбирали слагаемые в сумму по выполняющему набору (нужно только всё в обратном порядке проделать). □

Утверждение предыдущей теоремы верно, если мощность алфавита хотя бы 2. В унарном алфавите задача разбиения в сумму множеств разрешима за полиномиальное время.

Утверждение 5.57. *Пусть* Унарное разбиение — *язык*

$$\left\{(\{a^{a_1},\dots a^{a_n}\}) \;\middle|\; \forall i\ a_i \in \mathbb{N},\ \exists B \subset [1;n],\ \sum_{i\in B} a_i = \sum_{i\notin B} a_i\right\}.$$

Этот язык принадлежит классу **P**.

Доказательство. Заметим, что размер входа вырос экспоненциально. Если перевести его в двоичную систему счисления, то получим задачу Разбиение, которая полна в NP. Поскольку **NP** $\subseteq$ **EXPTIME** и экспоненциальное по входу языка Разбиение

время — это полиномиальное по входу языка Унарное разбиение время, то Унарное разбиение лежит в **P**. □

Остовные деревья с ограничениями. Итак, как мы видели выше, задача нахождения минимального остовного дерева может быть решена за полиномиальное время. Однако задачи поиска минимального остовного дерева с некоторыми ограничениями [на число листьев, степень вершин итд.] оказываются **NP**-сложны.

Остовное дерево со степенью ограничения: Пусть G — связный граф, а k положительное целое число. Существует ли остовное дерево T для G с максимальной степенью не более k?

Максимальное листовое остовное дерево: Пусть G — связный граф, а k положительное целое число. Есть ли остовное дерево для G, имеющее по крайней мере k листьев?

Минимальное листовое остовное дерево: Пусть G — связный граф, а k положительное целое число. Есть ли остовное дерево для G, имеющее не более k листьев?

Изоморфное остовное дерево: Пусть G — связный граф, а T — дерево (скажем, оба определены на n вершинах). Есть ли у G остовное дерево, изоморфное T?

Формально определим

- $L_1 = \{(G,k) \mid \exists T$ — остовное дерево $G,\ \max \deg_T(u) \leqslant k\}$
- $L_2 = \{(G,k) \mid \exists T$ — остовное дерево $G,\ T$ имеет не менее k листьев$\}$
- $L_3 = \{(G,k) \mid \exists T$ — остовное дерево $G,\ T$ имеет не более k листьев$\}$
- $L_4 = \{(G,H) \mid \exists T$ — остовное дерево $G,\ T \simeq H\}$

Теорема 5.58. *Все веденные выше языки* **NP**-*полны.*

Доказательство. Нетрудно построить полиномиальную сводимость языка Гамильтонов путь к языкам L_1, L_3 и L_4. Действительно, остовное дерево T является гамильтоновым путем титтк выполнены следующие условия:

- Все вершины имеют степень 1 или 2 в T.
- имеет ровно два листа.
- — это путь.

Таким образом, отображения

$$f_1 : G \mapsto (G,2), \quad f_3 : G \mapsto (G,2),$$

$$f_4 : (\{v_1, \dots v_n\}, E) \mapsto ((\{v_1, \dots v_n\}, E), (\{v_1, \dots v_n\}, \{v_1v_2, v_2v_3, \dots v_{n-1}v_n\}))$$

действительно задают полиномиальные сводимости Гамильтонов путь к языкам L_1, L_3 и L_4.

Далее мы построим Доминирующее множество $\leqslant_p L_2$. Достаточно показать, что G имеет связное доминирующее множество размером не более k тогда и только тогда, когда G имеет остовное дерево, имеющее $|V| - k$ листьев.

Сначала пусть D — связное доминирующее множество G с размером не более k. Так как G_D (индуцированный граф на подмножестве вершин D) связен, у него есть остовное дерево T_D. Мы расширяем T_D до остовного дерева T из G следующим образом. По определению любая вершина $u \notin D$ имеет сосела $v \in D$; мы добавляем соответствующее ребро $eu = uv$ к T_U. Сделав так для всех $u \notin D$, мы получим остовное дерево T для G, и все эти вершины u являются листьями по построению. Таким образом, G действительно имеет остовное дерево по крайней мере с $|V| - k$ листьями.

В обратную сторону, пусть T - остовное дерево для G с хотя бы $|V| - k$ листьями. Выберем D в качестве множества всех вершин, не являющихся листьями T, так что $|D| \leqslant k$. Кроме того, D является доминирующим множеством, так как каждый лист T смежен с не листом, так как мы можем предположить, что G имеет как минимум три вершины. Очевидно, что T_D является остовным деревом для G_D, так что D действительно является связным доминирующим множеством. □

Вычисление пересечения групп. Теперь рассмотрим задачу Пересечение групп: для данных $G = \langle A \rangle$ и $H = \langle B \rangle$ — подгрупп в Sym_n — найти $G \cap H$.

Пока не известен полиномиальный алгоритм, разрешающий эту задачу, и мы не ожидаем, что он существует, из-за следующей теоремы.

Теорема 5.59. set-stab *полиномиально сводится к* Пересечение групп *и наоборот.*

Доказательство. set-stab $\leqslant_P$ Пересечение групп:

Предположим, Δ — множество, который мы хотим стабилизировать, все, что нам нужно сделать — вычислить $G \cap \{Sym_\Delta \times Sym_{\Omega \setminus \Delta}\}$. Можно выбрать транспозиции как порождающее множество произведения. Пересечение двух множеств дает в точности $Stab_\Delta(G)$.

Пересечение групп $\leqslant_P$set-stab:

Рассмотрим произведение $G \times H \leqslant Sym_\Omega \times Sym_\Omega \leqslant Sym_{\Omega \times \Omega}$ и пусть действие будет покоординатным, $(g, h)(i, j) = (i^g, j^h)$. Теперь стабилизируем диагональ:

$$\Omega \times \Omega = \{(i, i) \mid i \in \Omega\},$$

и это даст $G \cap H$. □

И поскольку мы говорим, что изоморфизм графов сводится к set-stab, маловероятно, что у нас есть полиномиальный алгоритм для решения задачи пересечения.

Однако для частного случая, когда G нормализует H (то есть $G \leqslant N_{Sym_\Omega}(H)$), мы можем решить задачу пересечения за полиномиальное время.

Утверждение 5.60. *Если G нормализует H, то мы можем вычислить $G \cap H$ за полиномиальное время.*

Доказательство. Мы ищем башню подгрупп с хорошими свойствами. И поскольку G нормализует H, основная идея заключается в том, что $GH = \{gh \mid g \in G, h \in H\}$ является подгруппой в Sym_n. А также для всех i множество $G^{(i)}H$ также является подгруппой. Далее, поскольку G нормализует H, то $H \lhd GH$.

Башня, которую мы ищем, есть

$$G \cap H = G \cap G^{(n-1)}H \leqslant G \cap G^{(n-2)}H \leqslant \cdots \leqslant G \cap GH = G$$

А так как порождающий набор для $G^{(i)}H$ является объединением порождающих множеств для $G^{(i)}$ и H, мы можем проверить принадлежность $G \cap G^{(i)}H$. Таким образом, используя лемму Шрейера и алгоритм Reduce, мы можем спуститься в башню вычислений порождающийные установки.

Дополнительное свойство, которое нам нужно, это то, что индексы последовательных элементов башни малые. Читатель может проверить, что $[G \cap G^{(i-1)}H : G \cap G^{(i)}H] \leqslant n - i$. □


ЛАНЬ®


ЛАНЬ®

Часть 6

Избранные задачи и решения

Задача 2. Симметричная монета бросается 10 раз. Подсчитайте вероятности следующих событий:

(a) число выпавших «орлов» равно числу «решек»;
(b) выпало больше «орлов», чем «решек»;
(c) при $i = 1, \ldots, 5$ одинаковы результаты i-го и $11 - i$-го бросаний;
(d) «орел» выпал не менее четырех раз подряд.

Решение. **(a)** Выпало 5 орлов, соответствующих последовательностей бросков $\binom{10}{5}$, значит, вероятность равна $\binom{10}{5}2^{-10}$.

(b) Орлов и решек либо поровну, либо кого-то из них больше, и число комбинаций, в которых орлов больше, равно числу комбинаций, в которых решек больше. Пользуемся предыдущим пунктом и получаем

$$\frac{1}{2} - \binom{10}{5}2^{-11}.$$

(c) Палиндромов длины 10 всего 2^5, поэтому вероятность такого события равна $2^{-5} = \frac{1}{32}$.

(d) Воспользуемся формулой полной вероятности: максимальный отрезок из орлов может иметь длину от 4 до 10, аккуратно пересчитаем все эти вероятности:

- 10 орлов выпадают лишь в 1 случае;
- 9 орлов подряд выпадают лишь в 2 случаях — выпало 9 орлов и одна решка либо в начале, либо в конце;
- 8 орлов подряд выпадают в 5 случаях — последовательность есть либо XO$\underbrace{\text{P...P}}_{8\text{ раз}}$, либо O$\underbrace{\text{P...P}}_{8\text{ раз}}$O, либо $\underbrace{\text{P...P}}_{8\text{ раз}}$OX (на месте X может стоять либо P, либо O);
- 7 орлов подряд выпадают в $2\cdot(4+2) = 12$ случаях — рассуждения аналогичны («отрезок» из орлов должен быть «ограничен» решками, в остальных позициях может стоять все, что угодно — мы учитываем те случаи, в которых «отрезок» из семи орлов максимален);
- 6 орлов подряд выпадают в $2 \cdot (8 + 4) + 4 = 28$ случаях;
- 5 орлов подряд выпадают в $2 \cdot (16 + 8 + 8) = 64$ случаях;
- 4 орла подряд выпадают в $2\cdot(32+16+16)+16-5 =$ случаях — это придется учесть иначе, здесь могут случаи, уже учтенные ранее, то есть сначала надо рассмотреть все случаи с отрезком из 4 орлов, а затем исключить два случая, когда встречаются отрезки длин 5 и 4 (таких ровно два),а также случаи, где есть два отрезка длины 4;

В итоге получим $\frac{251}{1024}$.

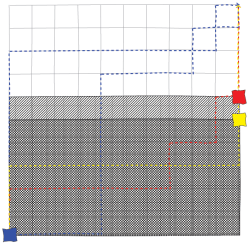
□

Задача 3. При двух бросках игральной кости выпало X_1 и X_2, соответственно. Вычислите $\mathbb{E}[\max\{X_1, X_2\}] + \mathbb{E}[\min\{X_1, X_2\}]$.

Решение. Воспользуемся линейностью матожидания:

$\mathbb{E}[\max\{X_1, X_2\}] + \mathbb{E}[\min\{X_1, X_2\}] = \mathbb{E}[\max\{X_1, X_2\} + \min\{X_1, X_2\}] = \mathbb{E}[X_1 + X_2] = 2\mathbb{E}[X_1]$, так как две кости одинаковые. Заметим, что $\mathbb{E}[X_1] = \frac{1+2+3+4+5+6}{6} = \frac{7}{2}$, поэтому ответ 7. □

Задача 4. В двух спичечных коробках имеется по n спичек. Каждый раз, когда человеку нужна спичка, он берет ее из случайного коробка (равновероятно). Найдите вероятность того, что когда один коробок опустеет, а в другом останется ровно k спичек.



Решение. Можно считать, что речь идет о путях в квадрате $n \times n$. Сначала вероятность того, что как только первый коробок опустеет, во втором останется k спичек. В терминах квадрата событие «один коробок опустеет, а в другом останется ровно k спичек» есть просто «путь впервые достиг правой стороны на высоте k». То есть это все пути, которые стартуют в синей точке, посещают красную точку, минуя желтую: в противном случае человек вынул последнюю спичку из первого коробка, во втором осталось больше k спичек. Таких возможностей ровно $\binom{2n-k-1}{n-1}$. Между тем, всего путей в квадрате $\binom{2n}{n}$. Тогда получится

$$\mathbb{P} = \frac{2\binom{2n-k-1}{n-1}}{\binom{2n}{n}}$$

□

Задача 5. Имеется генератор случайных битов, который выдает 0 или 1 с вероятностью $\frac{1}{2}$. Предложите алгоритм, который, используя данный генератор, возвращает 0 с вероятностью $\frac{1}{3}$ и 1 с вероятностью $\frac{2}{3}$.

Решение. Здесь вероятностное пространство есть $\Omega = \{0,1\}^{\mathbb{N}}$. Нам дан генератор

$$Gen_2 \text{ — генератор } \xi,\ \mathbb{P}(\xi = 0) = \mathbb{P}(\xi = 1) = \frac{1}{2},$$

надо построить

$$Gen_3 \text{ — генератор } \xi,\ \mathbb{P}(\xi = 0) = \frac{1}{3},\ \mathbb{P}(\xi = 1) = \frac{2}{3}.$$

Предложим следующий алгоритм:

1: **procedure** Gen3(Gen_2)

```
2:     ξ1 ← 0; ξ2 ← 0
3:     ξRet ← 0
4:     while ξ1 = 1 ∧ ξ2 = 0 do
5:         ξ1 = Gen2()
6:         ξ2 = Gen2();
7:     end while
8:     if ξ1 = ξ2 then
9:         ξRet = 1
10:    end ifreturn ξRet
11: end procedure
```

Здесь мы в течение каждого такта запускаем дважды генератор случайной величины. Если выпало дважды одинаковое значение, то есть 11 или 00, мы возвращаем 1 и покидаем цикл, если выпало 01, то возвращаем 0 и останавливаемся, а если выпало 10, то мы продолжаем цикл.

Вероятность того, что результат двух бросков станет впервые отличен от 10 и будет равен некоторой $x \in \{01, 11, 00\}$, равна $\frac{1}{4} + \frac{1}{4^2} + \ldots + \frac{1}{4^k} + \ldots = \frac{\frac{1}{4}}{1-\frac{1}{4}} = \frac{1}{3}$.

Теперь оценим *среднее время работы*: понятно, что этот алгоритм может никогда не остановиться, выдавая последовательность $1010\ldots$, но такое происходит с нулевой вероятностью. Итак, k бросков совершается с вероятностью $(\frac{1}{4})^{k-1}\frac{3}{4}$: сначала $k-1$ раз выпадает 10, в последний раз выпадает что-то отличное от 10. Считаем сумму:

$$\mathbb{E}t = \sum_{k=1}^{\infty} k\Big(\frac{1}{4}\Big)^{k-1}\frac{3}{4} = \frac{3}{4}\Big(\sum_{k=1}^{\infty} kz^{k-1}\Big)\Big|_{z=\frac{1}{4}} = \frac{3}{4}\Big(\sum_{k=1}^{\infty} z^k\Big)'\Big|_{z=\frac{1}{4}} = \frac{3}{4}\Big(\frac{1}{1-z}\Big)'\Big|_{z=\frac{1}{4}} =$$

$$= \frac{3}{4}\frac{1}{(1-z)^2}\Big|_{z=\frac{1}{4}} = \frac{4}{3}$$

Итак, в среднем нужно совершить $\frac{4}{3}$ бросков. □

Задача 6. Дан массив из n элементов, на которых определено отношение равенства. Постройте on-line алгоритм, который в «потоковом режиме обработки данных» определяет, есть ли в массиве элемент, повторяющийся больше $\frac{n}{2}$ раз. Считается, что в Вашем распоряжении есть память объемом $O(\log n)$ битов.

Решение. Формальное описание алгоритма находится ниже (в честь изобретателя данного алгоритма назовем процедуру Moore).

```
1: procedure Moore(w[1 : n])
2:     k = 0, h = 0
3:     for i = 1; i ⩽ n; i+ = 1 do
4:         if h = 0 then
```

```
            k = i
            h+ = 1
        end if
        if h > 0 и w[i] = w[k] then
            h+ = 1
        end if
        if h > 0 и w[i] ≠ w[k] then
            h− = 1
            if h = 0 then
                k = 0
            end if
        end if
    end for
    i = 0
    for i = 1; i ⩽ n; i+ = 1 do
        if w[i] = w[k] then
            i+ = 1
        end if
    end for
    if i > n/2 then
        return w[k]
    else
        print(NO x EXISTS)
    end if
end procedure
```

Предложим следующий алгоритм, использующий всего один проход по массиву и $O(\log n)$ битов. Мы будем хранить кандидата C, который может повторяться более $\frac{n}{2}$ раз, и «высоту» h. Каждый раз, встречая новый элемент $w[i]$, если высота положительная, то мы увеличим высоту, если $w[i] = C$, и уменьшим в ином случае. Если же высота равна 0, то мы присваиваем C значение первого встреченного $w[i]$.

Прежде чем доказать корректность алгоритма, продемонстрируем его работу на некоторых примерах (здесь A от absent, отсутствие кандидата):

Массив		1	2	1	3	1	2	1
Кандидат	A	1	A	1	A	1	A	1
Счетчик	0	1	0	1	0	1	0	1

Элемент 1 повторяется 4 раза в массиве из 7 элементов

Массив		1	2	2	2	3	1
Кандидат	A	1	A	2	2	2	A
Счетчик	0	1	0	1	2	1	0

Элемент 2 повторяется 3 раза в массиве из 6 элементов, никто не повторяется хотя бы 4 раза

Массив		1	2	3	3	1	1	1	1
Кандидат	A	1	A	3	3	3	A	1	1
Счетчик	0	1	0	1	2	1	0	1	2

Элемент 1 повторяется 5 раз в массиве из 9 элементов

Массив		1	2	3	3	3	7
Кандидат	A	1	A	3	3	3	3
Счетчик	0	1	0	1	2	3	2

Элемент 3 повторяется 3 раз в массиве из 6 элементов; он выдается как кандидат, но не повторяется более 4 раз — такие коллизии нужно устранять вторым проходом

Теперь докажем, что для любого $i \in [1; n]$ данный алгоритм после n шагов позволяет представить $w[1:i]$ как объединение множеств A и B, где

- A состоит из K копий C (в дальнейшем будем именовать K *высотой*),
- B — множество, в котором любой элемент повторяется не более $\frac{i}{2}$.

Это можно сделать индукцией по i. Действительно, база при $i = 1$ устанавливается тривиально; шаг же доказывается рассмотрением трех возможностей, возникающих при чтении $w[i+1]$:

- высота равна 0, тогда любой элемент здесь повторяется здесь не более $\frac{i}{2}$, добавление нового элемента делает множество A непустым (содержащим один элемент) и не меняет B;
- высота положительна и $w[i+1] = C$, тогда к множеству A добавляется $w[i+1]$, а множество B не меняется, разбиение в $A \cup B$ снова имеет место;
- высота положительная и $w[i+1] \neq C$, тогда мощность A уменьшается на 1 (за счет некоторого элемента, равного C), а мощность B — увеличивается на 2, при этом элементов, равных C, было не более $\frac{i-1}{2}$ (до прочтения $w[i+1]$ было $|A| = h$), значит, теперь их стало не более $\frac{i+1}{2}$ (добавляются два разных элемента). Действительно, за шаг до этого $|A|$ могла либо увеличиваться, либо уменьшаться на 1, и в обоих случаях получается, что любой элемент встречался не более $\frac{i-1-|A|}{2}$ раз (по предположению индукции).

  Например, если высота на i-ом шаге равнялась 1, то по предположению любой элемент в B повторяется не более $\frac{i-1}{2}$ раз, на $i+1$ шаге добавляются два разных элемента, получается не более $\frac{i+1}{2}$ повторений.

С помощью этого утверждения докажем корректность алгоритма. Пусть в массиве есть элемент X, повторяющийся более $\frac{n}{2}$ раз; покажем, что кандидат будет равен X. Допустим, это не так, и такой элемент равен $Y \neq C$. Тогда согласно лемме есть K вхождений C и, следовательно, не более $\frac{n-C}{2}$ вхождений этого $Y \neq C$, но Y должен входить более $\frac{n}{2}$. Если же нет элемента, встречающегося более половины раз, то второй проход по массиву это выявит.

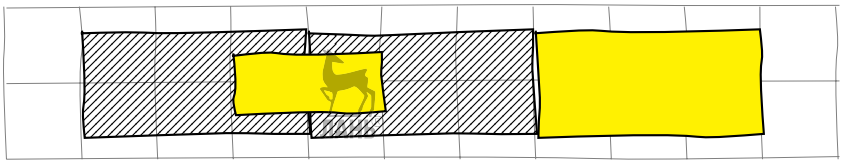
Последний момент: мы храним не самого кандидата, а номер ячейки массива, где впервые встретили его. Тем самым мы используем не более $2\log(n)$ *битов, и наш алгоритм не зависит от величины элементов массива.* □

Задача 7. На вход подается описания n событий в формате (s, f) — время начала и время окончания. Требуется составить расписание для человека, который хочет принять участие в максимальном количестве событий. Например, события это доклады на конференции или киносеансы на фестивале, которые проходят в разных аудиториях. Предположим, что участвовать можно только с начала события и до конца. Рассмотрим три жадных алгоритма.

- **(a)** Выберем событие кратчайшей длительности, добавим его в расписание, исключим из рассмотрения события, пересекающиеся с выбранным. Продолжим делать то же самое далее.
- **(b)** Выберем событие, наступающее раньше всех, добавим его в расписание, исключим из рассмотрения события, пересекающиеся с выбранным. Продолжим делать то же самое далее.
- **(c)** Выберем событие, завершающееся раньше всех, добавим его в расписание, исключим из рассмотрения события, пересекающиеся с выбранным. Продолжим делать то же самое далее.

Какой алгоритм вы выберете? В качестве обоснования для каждой процедуры проверьте, что она является оптимальной или постройте конкретный контрпример.

Решение. **(a)** Этот алгоритм не является оптимальным, здесь можно привести следующий контрпример:

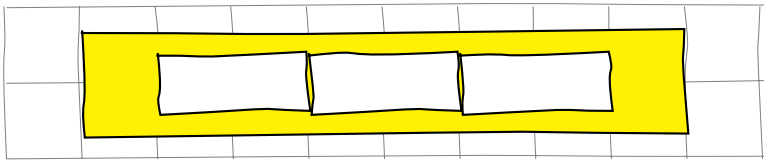


Формально, начальные данные есть четыре интервала

$$(0, 3),\ (2, 4),\ (3, 6),\ (6, 9)$$

Нетрудно видеть, что алгоритм выбирает лишь желтые интервалы и дает максимум из двух событий: на первом шаге выбирается интервал $(2,4)$ и исключаются из рассмотрения $(0,3)$ и $(3,6)$, на втором шаге берется последний интервал $(6,9)$. Между тем, выбор трех интервалов длины 3 дает ответ 3 и является оптимальным в данной ситуации.

(b) Контрпример построен ниже и аналогичен предыдущему:



Формально, начальные данные есть четыре интервала

$$(0,8),\ (1,3),\ (3,5),\ (5,7)$$

Нетрудно видеть, что алгоритм выбирает лишь один желтый интервал и дает максимум из одного события, в то время как можно посетить три события.

(c) А в этом пункте алгоритм является оптимальным. Докажем это от противного. Пусть для некоторых начальных данных $\{a_1, \dots a_k\}$ — множество интервалов, выданное нашим алгоритмом, а максимально возможное число непересекающихся событий равно n, соответствующая выборка (или хотя бы одна из возможных) есть $\{b_1, \dots b_n\}$. Тогда b_1 заканчивается не раньше a_1, так как событие a_1 заканчивается раньше всех. Заметим также, что b_2 начинается не раньше конца a_1 и заканчивается не раньше a_2 — ведь из всех событий, начинающихся после конца a_1, a_2 заканчивается раньше всех. Аналогично получим, что b_i начинается не раньше, чем заканчивается a_{i-1}, и заканчивается не раньше a_i. Тогда если $k < n$, то b_{k+1} начинается после конца a_k; в таком случае алгоритм мог выбрать хотя бы $n - k$ событий.

□

Задача 8. Покажите, что язык факторизации

$$L_{factor} = \{(N, M) \in \mathbb{Z}^2 \mid 1 < M < N \text{ и } N \text{ имеет делитель } d,\ 1 < d \leqslant M\}$$

принадлежит $\mathbf{NP} \cap \mathbf{co} - \mathbf{NP}$.

Решение. Сначала покажем, что $L_{factor} \in \mathbf{NP}$. Предъявим для этого всюду определенный предикат

$$R\big((N, M), d\big) = (1 < d \leqslant M) \wedge (1 < M < N) \wedge (d|N)$$

Ясно, что $(N, M) \in L_{factor}$ титтк существует d такой, что $R\big((N, M), d\big) = 1$. Так как $d < M$, то он полиномиален по входу (M, N); проверить же истинность $R\big((N, M), d\big)$

можно детерминированным полиномиальным алгоритмом за $O(\log(N)\log(M))$, сначала сравнив d с M за $O(\log(M))$, а затем поделив N на d с остатком за $O(\log(N)\log(M))$.

Теперь покажем, что $L_{factor} \in \mathbf{co}-\mathbf{NP}$. Для этого заметим, что $(N,M) \notin L_{factor}$ титтк выполнено хотя бы одно из следующих условий:

- $M = 1$ или $M \geqslant N$;
- N — простое;
- N имеет делитель $d \in (M;N)$.

Действительно, $(N,M) \notin L_{factor}$ титтк либо не выполнено ограничение на M и N, либо не существует делителя N, не превышающего M; тогда либо делитель больше M и таким образом лежит в интервале $(M;N)$, либо делителя нет вообще. Тогда покажем, что $L_{factor} \in \mathbf{co}-\mathbf{NP}$, представив дополнение как L_1, L_2, L_3 и показав принадлежность каждого из трех языков $\mathbf{NP}$. Действительно, пусть

- $L_1 = \{(N,M) \in \mathbb{Z}^2 | M = 1 \ \vee \ M \geqslant N\}$;
- $L_2 = \{(N,M) \in \mathbb{Z}^2 | N - \text{простое}\}$;
- $L_3 = \{(N,M) \in \mathbb{Z}^2 | \exists d, \ M < d < N, \ d | N\}$.

Выше уже было показано, что $(N,M) \notin L_{factor}$ титтк $(N,M) \in L_i$ для некоторого $i \in \{1,2,3\}$. Первый язык $L_1 \in \mathbf{P}$, так как можно детерминированной полиномиальной МТ проверить оба условия $M = 1$ и $M \geqslant N$ [оба за $O(\log M)$ и $O(\log N)$, соответственно]. Второй язык $L_2 \in \mathbf{NP}$ согласно предыдущей задаче. Третий язык $L_3 \in \mathbf{NP}$, так как существует предикат

$$R_{L_3}\big((N,M),d\big) = (M < d < N) \wedge (1 < M < N) \wedge (d|N),$$

истинность которого можно проверить детерминированным полиномиальным алгоритмом за $O(\log(N)^2)$, сначала сравнив d с M за $O(\log(N))$, а затем поделив N на d с остатком за $O(\log(N)^2)$; при этом $(N,M) \in L_3$ титтк при некотором $M < d < N$ принимает значение 1 предикат $R_{L_3}\big((N,M),d\big)$. Так как объединение языков из $\mathbf{NP}$ само лежит в $\mathbf{NP}$, то и $\overline{L_{factor}} = L_1 \cup L_2 \cup L_3 \in \mathbf{NP}$. □

Задача 9. Оценить асимптотику $T(n)$, если $T(n) = T(n-1) + 2T(n-2) + \ldots + 2^{n-1}T(0)$, где $T(0) = 1$.

Решение. Из условия получаем:

$$\begin{aligned} T(n+1) &= T(n) + 2T(n-1) + \ldots + 2^n T(0) \\ &= T(n) + 2\underbrace{\big(T(n-1) + 2T(n-2) + \ldots + 2^{n-1}T(0)\big)}_{T(n)} \\ &= 3T(n), \end{aligned}$$

откуда и из начального условия следует, что $T(n) = 3^n$. □

Задача 10. Пусть $L \in \mathbf{co}-\mathbf{NP}$, а N — непустой подъязык L. Верно ли, что $N \in \mathbf{co}-\mathbf{NP}$?

Решение. Неверно. Пусть $L = \Sigma^*$. Тогда $L \in \mathbf{P} \subseteq \mathbf{co} - \mathbf{NP}$. Так как любой язык — это подмножество языка Σ^*, то в качестве N можно взять любой язык не из $\mathbf{co} - \mathbf{NP}$. Например, подойдёт любой неразрешимый язык в качестве N. □

Задача 11. Язык EUCLID состоит из троек (a, b, c) таких, что $\gcd ab = c$ (числа задаются в двоичном виде). Язык NOHAMPATH состоит из описаний графов, которые не содержат гамильтонова цикла. Пусть NOHAMPATH полиномиально сводится к EUCLID. Верно ли, что тогда $\mathbf{NP} = \mathbf{co} - \mathbf{NP}$?

Доказательство. Верно. Из условия следует, что $\mathbf{co} - \mathbf{NP} = \mathbf{P}$, т.к. язык NOHAMPATH полон в классе $\mathbf{co} - \mathbf{NP}$ (так как его дополнение (HAMPATH) полно в $\mathbf{NP}$). Но тогда $\mathbf{P} = \mathbf{NP} = \mathbf{co} - \mathbf{NP}$ в силу замкнутости класса $\mathbf{P}$ относительно взятия дополнения. □

Задача 12. Язык $\mathrm{supw}L$ надслов языка L определим по формуле:

$$\mathrm{supw}L = \{u = wsv \mid w, v \in \Sigma^*,\ s \in L\}$$

. Верно ли, что если $L \in \mathbf{P}$, то и $\mathrm{supw}L \in \mathbf{P}$?

Доказательство. Верно. Нужно найти подстроку из языка L. Для этого будем перебирать все подстроки длины 1, длины 2, длины 3 и т. д. Пусть $p(n)$ — полином, ограничивающий время работы алгоритма, распознающего язык L, на входе длины n. Тогда время работы построенного алгоритма не превосходит

$$n \cdot p(1) + (n-1) \cdot p(n) + \ldots + 2 \cdot p(n-1) + p(n) \leqslant n^2 \cdot p(n).$$

Последнее неравенство верно не для всех полиномов, но мы можем взять в качестве $p(n)$ возрастающий полином (просто взять $p(n)$ так, чтобы все его коэффициенты были положительными; это не уменьшит его значения во всех положительных точках). □

Задача 13. Определим граф G_n на множестве $\{1, 2, \ldots, n\}$, где вершины — это числа, а две вершины соединены ребром тогда и только тогда, когда одно число делится на другое. Верно ли, что язык $L = \{(n, k) \mid$ в G_n есть клика размера $k\}$ принадлежит классу $\mathbf{NPC}$? Числа задаются двоичной записью.

Решение. Неверно. Оценим размер максимальной клики в графе G_n. Для этого пройдём в ней от минимума до максимума. Получим последовательность $1 \to k_1 m \to k_1 k_2 m \to \ldots \to k_1 k_2 \ldots k_l m$, где все $k_j \geqslant 2$ (в максимальную клику всегда входит число 1, потому что оно делит любое число). Поэтому чисел в максимальной клике не более $\lfloor \log_2 n \rfloor + 1$. Поэтому, если $k > \lfloor \log_2 n \rfloor + 1$, то выдаём ответ 0. В противном случае в графе есть клика нужного размера: числа $1, 2, 2^2, \ldots, 2^{k-1}$ образуют клику либо нужного размера. При этом наибольшее число в этой клике меньше n, так как $2^{k-1} \leqslant 2^{\lfloor \log_2 n \rfloor} \leqslant n$. Таким образом, разрешающий алгоритм работает за одно сравнение. □

Задача 14. В массиве $a[1 \dots N]$ записано N целых чисел. Все встречаются по 2 раза, кроме одного, которое встречается 3 раза. Требуется найти число, встречающееся 3 раза. Ограничения следующие: время работы — $O(N \log(\max\limits_i a[i]))$, память — $O(\log(\max\limits_i a[i]))$.

Решение. Сделаем побитовую операцию XOR всех записанных чисел (это делается за время $O(N \log(\max\limits_i a[i]))$; при этом память, которую мы используем есть $O(\log(\max\limits_i a[i]))$). Полученное число — искомое. □

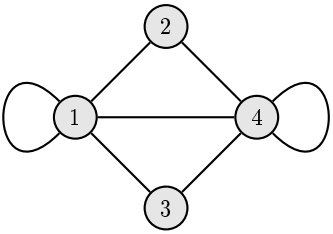
Задача 15. В массиве $a[1 \dots N]$ записано N целых чисел. Все встречаются по 3 раза, кроме одного, которое встречается 1 раз. Требуется найти число, встречающееся 1 раз. Ограничения следующие: время работы — $O(N \log^3(\max\limits_i a[i]))$, память — $O(\log(\max\limits_i a[i]))$.

Решение. Переведём числа в троичную систему счисления. Для каждого отдельного числа это делается за $O(\log(a[i]))$ делений на число 3 (вычитания мы не учитываем). В итоге за $O(N \log^3(\max\limits_i a[i]))$ можно преобразовать все числа в троичную систему счисления. Далее мы складываем числа поразрядно по модулю 3. То, что получится в итоге, и есть искомое число. □

Задача 16. Покажите, что если на вход подаётся двоичное число, гарантированно являющееся полным квадратом, то извлечь квадратный корень из него можно за полиномиальное время.

Решение. Будем использовать бинарный поиск: делим отрезок на две равные (или почти равные части) и берём «среднее» число c (договоримся, что будем брать $c = \lfloor \frac{a+b}{2} \rfloor$, где a и b — левая и правая границы текущего отрезка соответственно; в самом начале $a = 1$ и $b = n$). Если $c^2 > n$, то переходим к правой половине: $a := c$ и $b := b$; если $c^2 < n$, то переходим к левой половине: $a := a$ и $b := c$; если же $c^2 = n$, то останавливаемся и выдаём ответ c. Заметим, что делений отрезка пополам мы сделаем не больше $\log_2 n$ делений отрезка пополам. После каждого деления отрезка пополам мы вычисляем результат деления одного числа на другое, возводим число в квадрат и сравниваем два числа (длины всех получаемых в процессе работы чисел не больше $2 \log_2 n$, так как $\log_2 n^2 = 2 \log_2 n$). Сложность алгоритма: $O(\log_2^3 n)$. Отметим, что этот алгоритм может ответить также и на вопрос, является ли целое число полным квадратом, если в конце возвращать «нет», если не будет найдено нужно число, когда левая и правая границы текущего отрезка совпадут. □

Задача 17. На картинке ниже изображен граф. Пусть g_n — это число путей в G длины n, которые начинаются в вершине 1. Из определения следует, что $g(0) = 1$ (единственный путь: $0 \to 0$), а $g(1) = 4$ (пути: $1 \to 1$, $1 \to 2$, $1 \to 4$, $1 \to 3$).



Рассмотрим A, матрицу смежности графа G:

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}.$$

(a) Вычислите число $g(2)$ путей в G длины 2 и проверьте, что оно равно сумме элементов первой строки матрицы A^2. Объясните это совпадение и докажите общую формулу для $g(n)$.

(b) Найдите рекуррентное соотношение, которому удовлетворяет последовательность $\{g_n,\ n = 0, 1, \ldots\}$.

(c) Непосредственное вычисление по рекуррентной формуле. Оцените его трудоемкость при вычислении $A = g_{20000} \bmod 29$.

(d) Докажите, что последовательность $\{g_n\}$ периодична по любому модулю. Оцените ее период для $\bmod 29$ и найдите трудоемкость вычисления этим способом.

Решение. **(a)** «Совпадение» на самом деле инкорпорировано в определение умножения матриц, поэтому сразу к доказательству утверждения:

$$\forall n \geqslant 1 \quad (A^n)_{ij} = \text{число путей длины } n \text{ из вершины } i \text{ в вершину } j$$

Доказываем его индукцией по $n \in \mathbb{N}$: для $n = 1$ утверждение верно по определению матрицы A, а переход корректен, так как

$$(A^n)_{ij} = \sum_{k=1}^{4} (A^{n-1})_{ik} A_{kj},$$

а $(A^{n-1})_{ik} A_{kj}$ — число путей длины n из i в j, предпоследняя вершина которых — k; соответствующие множества не пересекаются и в объединении дают все искомые пути.

Из вышесказанного сразу следует, что $g(n) = \sum\limits_{i=1}^{4} A_{1i}$, а $g(2) = 4+2+2+4 = 12$.

(b) Согласно предыдущему пункту

$$g(n) = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} A^n \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Если мы получим реккурентное выражение на A^n, то из него сразу же получим реккурентное уравнение на $g(n)$.

Мы найдем реккурентное полиномиальное уравнение на A, если вычислим минимальный многочлен A, то есть полином $f(X) = \sum b_i X^i$ минимальной степени со старшим коэффициентом 1 такой, что $f(A)$. Заметим, что $A = A^T$, то есть A — матрица самосопряженного линейного оператора, следовательно, в некотором базисе A может быть записана диагональной матрицей, и минимальный многочлен A будет делить характеристический. [Отдельно стоит отметить, что такая схема рассуждений работает с любым неориентированным графом.]

Здесь же

$$\begin{vmatrix} 1-\lambda & 1 & 1 & 1 \\ 1 & -\lambda & 0 & 1 \\ 1 & 0 & -\lambda & 1 \\ 1 & 1 & 1 & 1-\lambda \end{vmatrix} = \lambda^4 - 2\lambda^3 + 4\lambda^2 = (\lambda^2 - 2\lambda + 4)\lambda^2$$

Заметим, что $A^3 - 2A^2 + 4A = 0$:

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 4 & 2 & 2 & 4 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 4 & 2 & 2 & 4 \end{bmatrix}, \quad A^3 = \begin{bmatrix} 12 & 8 & 8 & 12 \\ 8 & 4 & 4 & 8 \\ 8 & 4 & 4 & 8 \\ 12 & 8 & 8 & 12 \end{bmatrix}$$

При этом $A^2 - 2A + 4\mathrm{Id} \neq 0$, поэтому задачу о поиске реккуренты нужно ставить так:

$$\forall n \geqslant 3 \quad g(n) = 2g(n-1) + 4g(n-2), \quad g(1) = 4, g(2) = 12.$$

(c) На производящую функцию $G(z) = \sum\limits_{k=1}^{\infty} g(n) z^{n-1}$ этой последовательности получается уравнение

$$G(z) - 4 - 12z = 2z(G(z) - 4) + 4z^2 G(z),$$

откуда $G(z) = 4\frac{1+z}{1-2z-4z^2}$. Заметим, что корни $1-2z-4z^2$ есть $1 \pm \sqrt{5}$, и решения можно искать в виде $g(n) = \alpha(1+\sqrt{5})^n + \beta(1-\sqrt{5})^n$. Из условий

$$\begin{cases} \alpha(1+\sqrt{5}) + \beta(1-\sqrt{5}) = 4 \\ \alpha(1+\sqrt{5})^2 + \beta(1-\sqrt{5})^2 = 12 \end{cases}$$

имеем $\alpha = \frac{5+3\sqrt{5}}{10}$, $\alpha = \frac{5-3\sqrt{5}}{10}$. Итого

$$g(n) = \frac{5+3\sqrt{5}}{10}\left(1+\sqrt{5}\right)^n + \frac{5-3\sqrt{5}}{10}\left(1-\sqrt{5}\right)^n.$$

Уравнение на производящую функцию можно записать и по модулю 29, получится формальный ряд с коэффициентами $g_n \in \mathbb{Z}/29\mathbb{Z}$, вычислим их явно. Заметим для начала, что 5 является квадратичным вычетом по этому модулю: в силу квадратичного закона взаимности

$$\left(\frac{5}{29}\right)\left(\frac{29}{5}\right) = (-1)^{\frac{5-1}{2}\frac{29-1}{2}} = (-1)^{28} = 1,$$

а $29 \equiv 2^2 \bmod 5$. Решений $x^2 = 5$ в $\mathbb{Z}/29\mathbb{Z}$ ровно 2, найдем хотя бы одно. Можно честно найти первообразный корень и возвести в нужную степень, но можно заметить, что $11^2 = 121 = 29 \cdot 4 + 5$, и тогда корни есть 11 и 18. Тогда заменим $\sqrt{5}$ на 11 и получим

$$g(n) \bmod 29 = \frac{38}{10}12^n + \frac{-28}{10}\Big(-10\Big)^n = -2 \cdot 12^n + 3 \cdot 19^n$$

[если бы мы заменили $\sqrt{5}$ на 18, получилась бы та же формула — слагаемые поменялись бы местами].

Можно напрямую возвести в степень и взять остаток по модулю 29. Однако мы воспользуемся иным способом: если известны два соседних члена последовательности, то следующий член можно вычислить за $O(1)$, совершив несколько сложений и взятий остатка по модулю 29. Все эти операции можно считать за постоянное по n время: нам достаточно работать с остатками g_n по модулю 29, мы можем вычислить $(2g_{n-1} + 4g_{n-2}) \bmod 29$, прибавляя пошагово $g_{n-1} \bmod 29$ или $g_{n-2} \bmod 29$ и затем снова беря остаток по модулю 29 [либо сразу получается остаток, либо придется вычесть 29]. Например, для $g(1) = 4$ и $g(2) = 12$ вычисления пройдут следующим образом: надо найти $g(3) = 2g(2) + 4g(1)$, тогда мы начинаем с 0 и

- прибавляем $g(2) \bmod 29 = 12$, получаем $\underline{12}$, вычитать 29 не надо;
- прибавляем $g(2) \bmod 29 = 12$, получаем $\underline{24}$, вычитать 29 не надо;
- прибавляем $g(1) \bmod 29 = 4$, получаем $\underline{28}$, вычитать 29 не надо;
- прибавляем $g(1) \bmod 29 = 4$, получаем 32, вычитаем 29 и получаем $\underline{3}$;
- прибавляем $g(1) \bmod 29 = 4$, получаем $\underline{7}$, вычитать 29 не надо;
- прибавляем $g(1) \bmod 29 = 4$, получаем $\underline{11}$, вычитать 29 не надо.

В итоге $g(3) \bmod 29 = 11$.

Описанным выше способом можно вычислить $g(n) \bmod 29$ за $O(n)$ шагов.

(d) Мы уже знаем, что $g(n) = 2g(n-1) + 4g(n-2)$, поэтому пара остатков

$$(g(n-1) \bmod 29, g(n-2) \bmod 29)$$

однозначно определяет $g(n) \bmod 29$. Тогда период ограничен числом пар остатков по модулю 29, то есть $29^2 = 841$. Отметим, что совершенно не обязательно период должен равняться 29^2, он может быть существенно меньше, как и для

других аналогичных реккурент второго порядка. Например, период $F_n \bmod 11$ равен 10 [проверьте самостоятельно]. Но для сложности наших вычислений это несущественно: нам достаточно найти период $T \leqslant 841$, взять остаток n по модулю T и вычислить $g_{n \bmod T} \bmod 29$. Согласно предыдущему пункту, вычисление периода потребует $O(T)$ арифметических операций в любом случае, что не зависит от n. Взятие остатка n по модулю T можно осуществить за $O(\log(n)\log(T))$ операций, просто разделив с остатком. Затем за постоянное по n время проделать операции из предыдущего пункта — нам придется считать не более T последовательных членов. Итак, сложность по n получается равной $O(\log(n))$.

Итак, период мог быть существенно меньше 29^2, но он ограничен константой, не зависящей от n, и поэтому мы легко проделали наши вычисления.

□


ЛАНЬ®


ЛАНЬ®

Библиография

Стандартная литература по теме — [32, 28]. Хорошее введение в алгоритмы с точки зрения программиста написано Седжвиком [37], можно также посетить страничку его курса.

[1] Manindra Agrawal, Neeraj Kayal и Nitin Saxena. “PRIMES is in P”. В: *Annals of Mathematics* 160 (2004), с. 781—793. URL: https://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf.

[2] Manindra Agrawal и Chandan Saha. “A Survey of Techniques Used in Algebraic and Number Theoretic Algorithms”. В: (). URL: https://drona.csa.iisc.ac.in/~chandan/research/survey_CNT.pdf.

[3] S. Arora и B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[4] J. A. Bondy и U.S.R. Murty. *Graph theory*. Springer, 2008.

[5] Joan Boyar. “Inferring Sequences Produced by Pseudo-random Number Generators”. В: *J. ACM* 36.1 (1989), с. 129—141. ISSN: 0004-5411. DOI: 10.1145/58562.59305. URL: http://doi.acm.org/10.1145/58562.59305.

[6] Alfred Brauer. “On addition chains”. В: *Bulletin American Mathematical Society* 45 (1939), с. 736—739.

[7] Alonzo Church. “An Unsolvable Problem of Elementary Number Theory”. В: *American Journal of Mathematics* 58.2 (1936). JSTOR, www.jstor.org/stable/2371045, с. 345—363.

[8] Neill Michael Clift. “Calculating optimal addition chains”. В: *Computing* 91.3 (2011), с. 265—284. ISSN: 1436-5057. DOI: 10.1007/s00607-010-0118-8. URL: https://doi.org/10.1007/s00607-010-0118-8.

[9] Leonard Eugene Dickson. *History of the Theory of Numbers*. Carnegie Institue of Washington, 1919. URL: https://archive.org/details/historyoftheryo01dick/page/1.

[10] Bakir Farhi. “An identity involving the least common multiple of binomial coefficients and its application”. В: *arXiv e-prints*, arXiv:0906.2295 (июнь 2009), arXiv:0906.2295. arXiv: 0906.2295 [math.NT].

[11] P. Gács и L. Lovász. *Complexity of algorithms*. lecture notes, 1999.

[12] M. Goresky и A. Klapper. *Algebraic Shift Register Sequences*. CUP, 2012.

[13] R.L. Graham и Pavol Hell. “On the History of the Minimum Spanning Tree Problem”. В: *Annals of the History of Computing* 7 (февр. 1985), с. 43—57. DOI: 10.1109/MAHC.1985.10011.

[14] G. Grimmett и D. Stirzaker. *One thousand exercises in probability*. Oxford University Press, 2001.

[15] D. Jungnickel. *Graphs, Networks and Algorithms*. Springer, 2013.

[16] R. Motwani и P. Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.

[17] M. Nair. “On Chebyshev-Type Inequalities for Primes”. В: *The American Mathematical Monthly* 89.2 (1982), с. 126—129. URL: http://www.jstor.org/stable/2320934.

[18] Vaughn Pratt. “Every prime has a succint sertificate”. В: *SIAM Journal of Computing* 4.3 (1975), с. 214—220. URL: http://boole.stanford.edu/pub/SucCert.pdf.

[19] J. Barkley Rosser и Lowell Schoenfeld. “Approximate formulas for some functions of prime numbers”. В: *Illinois J. Math.* 6.1 (март 1962), с. 64—94. DOI: 10.1215/ijm/1255631807. URL: https://doi.org/10.1215/ijm/1255631807.

[20] V. Shoup. *A computuational introduction to group theory*. 2009. URL: http://e-maxx.ru/bookz/files/shoup.pdf.

[21] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction”. В: *Proceedings of the London Mathematical Society* s2-43.1 (1938), с. 544—546. DOI: 10.1112/plms/s2-43.6.544. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-43.6.544. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-43.6.544.

[22] Н. Алон и Дж. Спенсер. *Веротяностный метод*. Москва: Бином, 2011.

[23] М. Балазар. *Асимптотический закон распределения простых чисел*. МЦНМО, 2013. URL: http://www.mccme.ru/free-books/dubna/balazard.pdf.

[24] И. Виноградов. *Основы теории чисел*. Москва: Гостехиздат, 1952.

[25] Д. Гильберт и В. Аккерман. *Основы теоретической логики*. Москва: Иностранная литература, 1947.

[26] А.А. Глибичук и др. *Элементы математики в задачах*. МЦНМО, 2016. URL: https://www.mccme.ru/circles/oim/discrbook.pdf.

[27] М. Гэри и Д. Джонсон. *Вычислительные машины и труднорешаемые задачи*. Москва: Мир, 1982.

[28] С. Дасгупта, Х. Пападимитриу и У. Вазирани. *Алгоритмы*. Москва: МЦНМО, 2014.

[29] А. Звонкин и С. Ландо. *Графы на поверхностях и их приложения*. Москва: МЦНМО, 2010.

[30] А. Китаев, А. Шень и М. Вялый. *Классические и квантовые вычисления*. Москва: МЦНМО, 1999.

[31] Д. Кнут. *Искусство программирования для ЭВМ*. Москва: Вильямс, 2002.

[32] Т. Кормен и др. *Алгоритмы: построение и анализ*. 3-е изд. Москва: Вильямс, 2013.

[33] Н. Кузюрин и С. Фомин. *Эффективные алгоритмы и сложность вычислений*. Москва: Мир, 1979.

[34] Д. В. Мусатов. *Теория сложности*. конспект лекций. МФТИ, 2016. URL: http://ru.discrete-mathematics.org/fall2016/3/complexity/compl-book.pdf.

[35] Ал. Омельченко. *Теория графов*. Москва: МЦНМО, 2018.

[36] Х. Пападимитриу и К. Стайглиц. *Комбинаторная оптимизация. Алгоритмы и сложность*. Москва: Мир, 1985.

[37] Р. Седжвик. *Фундаментальные алгоритмы на C++*. М.: Диа-Софт, 2001.

[38] А. Ширяев. *Вероятность*. Москва: МЦНМО, 2003.