Конспект курса алгоритмов

СПбГУ, первый семестр, 2019 год

Алексей Гордеев

Оглавление

Введение

1. Анализ сложности алгоритмов

1.1 Вычисление чисел Фибоначчи

Последовательность чисел Фибоначчи

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

определяется следующим образом:

Определение 1.1.1 F_n — *n-ое число Фибоначчи*, где

$$F_0 = F_1 = 1,\ F_n = F_{n-1} + F_{n-2}, n > 1.$$

Числа Фибоначчи растут *экспоненциально* быстро:

Лемма 1.1.1 $2^{\lfloor n/2 \rfloor} \leqslant F_n \leqslant 2^n$.

Доказательство. Докажем утверждение по индукции.

База. Утверждение верно для $n = 0, 1$.

Переход. Пусть $n > 1$, тогда $F_n = F_{n-1} + F_{n-2} \leqslant 2^{n-1} + 2^{n-2} < 2^n$.

С другой стороны, $F_n \geqslant 2 \cdot F_{n-2} \geqslant 2 \cdot 2^{\lfloor (n-2)/2 \rfloor} = 2^{\lfloor n/2 \rfloor}$. ■

Экспоненциальный алгоритм

Следующий рекурсивный алгоритм вычисляет n-ое число Фибоначчи, точно следуя определению:

```
1 fib(n):
2   if n <= 1:
3     return 1
4   return fib(n - 1) + fib(n - 2)
```

Каково время работы этого алгоритма? Оценим $T(n)$ — суммарное количество вызовов `fib`, происходящих при выполнении `fib(n)`: если $n \leqslant 1$, то $T(n) = 1$; иначе

$$T(n) = 1 + T(n-1) + T(n-2).$$

Несложно доказать по индукции, что $F_n \leqslant T(n) < 2F_n$. В каждом вызове `fib` совершается ограниченное число (скажем, не больше пяти) операций, поэтому время работы алгоритма примерно пропорционально F_n (чуть позже у нас появится формальное определение этого "примерно").

Из леммы 1.1.1 следует, например, что, $F_{300} \geqslant 2^{150} > 10^{45}$. На компьютере, выполняющем 10^9 операций в секунду, `fib(300)` будет выполняться больше 10^{36} секунд.

Полиномиальный алгоритм

Посмотрим на дерево рекурсивных вызовов алгоритма `fib` (рис. 1.1). Видно, что алгоритм много раз вычисляет одно и то же. Давайте сохранять результаты промежуточных вычислений в массив:

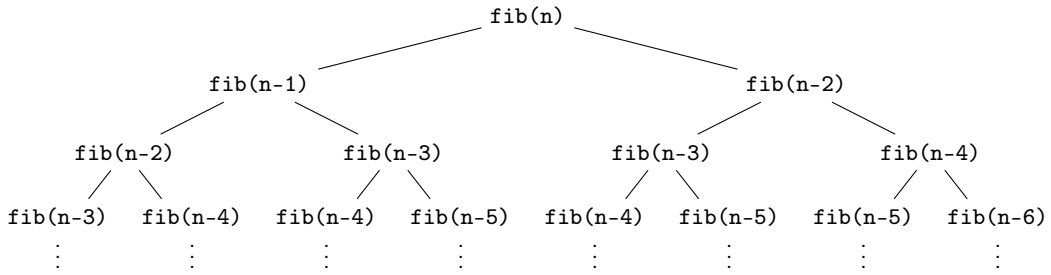


Рис. 1.1: Дерево рекурсивных вызовов `fib`

```
fibFast(n):
  if n <= 1:
    return 1
  int f[n + 1] # создаём массив с индексами 0..n
  f[0] = f[1] = 1
  for i = 2..n:
    f[i] = f[i - 1] + f[i - 2]
  return f[n]
```

На каждой итерации цикла совершается одно сложение, всего итераций примерно n, поэтому количество сложений, выполняемых в ходе нового алгоритма, примерно пропорционально n. Есть ещё одна тонкость — из леммы 1.1.1 следует, что двоичная запись F_n имеет длину порядка n. Чуть позже мы увидим, что сложение двух n-битовых чисел требует порядка n элементарных операций, значит общее время работы нового алгоритма примерно пропорционально n^2. С помощью `fibFast` можно уже за разумное время вычислить не только F_{300}, но и F_{100000}.

1.2 O-символика

Хорошо себя зарекомендовал и стал общепринятым следующий подход — оценивать время работы алгоритма некоторой функцией от входных параметров, при этом пренебрегая ограниченными множителями. Это позволяет эффективно сравнивать алгоритмы между собой, при этом не нужно заниматься точным подсчётом количества элементарных операций. Примерно так мы и рассуждали об алгоритмах вычисления чисел Фибоначчи. Введём несколько обозначений, которые помогут проводить подобные рассуждения более кратко и точно.

R Время работы алгоритма — это, конечно, всегда неотрицательная функция. Тем не менее, мы даём следующие определения для функций, принимающих произвольные вещественные значения, поскольку такие функции часто возникают в процессе рассуждений.

Определение 1.2.1 Рассмотрим функции $f, g : \mathbb{N} \to \mathbb{R}$.

- $f = O(g)$, если существуют такие $C > 0, N > 0$, что для любого $n > N$: $|f(n)| \leqslant C \cdot |g(n)|$.
- $f = \Omega(g)$, если существуют такие $C > 0, N > 0$, что для любого $n > N$: $|f(n)| \geqslant C \cdot |g(n)|$.
- $f = \Theta(g)$, если существуют такие $C_1 > 0, C_2 > 0, N > 0$, что для любого $n > N$: $C_1 \cdot |g(n)| \leqslant |f(n)| \leqslant C_2 \cdot |g(n)|$.

R Запись $f = O(g)$ можно понимать как "$|f| \leqslant |g|$ с точностью до константы". Аналогично, $\Omega(\cdot)$ можно считать аналогом $\geqslant$, а $\Theta(\cdot)$ — аналогом $=$.

Ещё один способ понимать запись $f = O(g)$ — отношение $\frac{|f(n)|}{|g(n)|}$ ограничено сверху некоторой константой.

Многие естественные свойства операторов сравнения $\leqslant, =, \geqslant$ выполняются и для их асимптотических аналогов. Сформулируем некоторые из этих свойств:

Предложение 1.2.1

1. $f = \Theta(g)$ тогда и только тогда, когда $f = O(g)$ и $f = \Omega(g)$.
2. $f = O(g)$ тогда и только тогда, когда $g = \Omega(f)$.
3. $f = \Theta(g)$ тогда и только тогда, когда $g = \Theta(f)$.

Доказательство. Докажем для примера п. 2.

Пусть $C > 0$, тогда $|f(n)| \leqslant C \cdot |g(n)|$ равносильно $|g(n)| \geqslant \frac{1}{C} \cdot |f(n)|$. Таким образом, $f = O(g)$ с константой C тогда и только тогда, когда $g = \Omega(f)$ с константой $\frac{1}{C}$. ■

Существуют также асимптотические аналоги $<$ и $>$: $o(\cdot)$ и $\omega(\cdot)$.

Определение 1.2.2 Рассмотрим функции $f, g : \mathbb{N} \to \mathbb{R}$.

- $f = o(g)$, если для любого $C > 0$ найдётся такое $N > 0$, что для любого $n > N$: $|f(n)| \leqslant C \cdot |g(n)|$.
- $f = \omega(g)$, если для любого $C > 0$ найдётся такое $N > 0$, что для любого $n > N$: $|f(n)| \geqslant C \cdot |g(n)|$.

R Запись $f = o(g)$ можно понимать как "отношение $\frac{|f(n)|}{|g(n)|}$ стремится к нулю с увеличением n", а запись $f = \omega(g)$ — как "отношение $\frac{|f(n)|}{|g(n)|}$ стремится к бесконечности с увеличением n".

Предложение 1.2.2

1. $f = o(g)$ тогда и только тогда, когда $g = \omega(f)$.
2. Если $f = O(g)$ и $g = O(h)$, то $f = O(h)$. То же верно для $\Omega, \Theta, o, \omega$.
3. Если $f = O(g)$, $g = o(h)$, то $f = o(h)$. Если $f = \Omega(g)$, $g = \omega(h)$, то $f = \omega(h)$.
4. Если $f = O(h)$ и $g = O(h)$, то $f + g = O(h)$. То же верно для o. Если $f, g \geqslant 0$, то то же верно и для Ω, Θ, ω.
5. Если $f = o(g)$, то $f + g = \Theta(g)$. Если $f, g \geqslant 0$, $f = O(g)$, то $f + g = \Theta(g)$.
6. Для любого $C \neq 0$ верно $C \cdot f = \Theta(f)$.

R Заметим, что у последних трёх свойств нет аналогов для обычных операторов сравнения.

Доказательство. Докажем для примера п. 3 (вариант с O и o).

Нужно показать, что для любого $C > 0$ найдётся $N > 0$, что для любого $n > N$: $|f(n)| \leqslant C \cdot |h(n)|$. Зафиксируем $C > 0$.

$f = O(g)$, поэтому найдутся $C_1 > 0, N_1 > 0$, что для любого $n > N_1$: $|f(n)| \leqslant C_1 \cdot |g(n)|$. $g = o(h)$, поэтому найдётся такое $N_2 > 0$, что для любого $n > N_2$: $|g(n)| \leqslant \frac{C}{C_1} \cdot |h(n)|$. Тогда для любого $n > \max(N_1, N_2)$: $|f(n)| \leqslant C_1 \cdot |g(n)| \leqslant C_1 \cdot \frac{C}{C_1} \cdot |h(n)| = C \cdot |h(n)|$. ■

Вернемся к алгоритмам вычисления чисел Фибоначчи. Пользуясь новыми обозначениями, мы показали, что время работы `fibFast(n)` можно оценить как $O(n^2)$. При этом время работы `fib(n)` можно оценить (сверху) как $O(2^n \cdot n)$, и (снизу) как $\Omega(2^{\lfloor n/2 \rfloor})$.

1.3 Многочлены, экспоненты и логарифмы

Очень часто время работы алгоритма удаётся оценить функцией, являющейся комбинацией каких-то из трёх базовых типов: многочленов, экспонент и логарифмов. Так, время работы `fibFast` мы оценили многочленом n^2, а время работы `fib` — произведением экспоненты на многочлен: $2^n \cdot n$. В связи с этим полезно изучить асимптотические свойства этих функций и научиться сравнивать их между собой.

Лемма 1.3.1 Для любых $l > k$ верно $n^k = o(n^l)$.

Доказательство. Для любого $C > 0$, при $n \geqslant \left(\frac{1}{C}\right)^{\frac{1}{l-k}}$ верно

$$n^k \leqslant C \cdot \frac{1}{C} \cdot n^k \leqslant C \cdot n^{l-k} \cdot n^k = C \cdot n^l.$$

■

Определение 1.3.1 *Многочлен* — это функция, которую можно записать в виде

$$f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$$

для некоторого $d \geqslant 0$, так, что $a_d \neq 0$. Это d называют *степенью* многочлена и обозначают как $\deg(f)$.

Предложение 1.3.2 Пусть $f(n)$ — многочлен степени d. Тогда $f(n) = \Theta(n^d)$.

Доказательство. Пусть $f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$. Из леммы 1.3.1 и п. 6 предложения 1.2.2 следует, что $a_j n^j = o(n^d)$ для любого $0 \leqslant j < d$, $a_d n^d = \Theta(n^d)$. Тогда, несколько раз применив п. 5 предложения 1.2.2, получаем $f(n) = \Theta(n^d)$. ■

Определение 1.3.2 Пусть $n > 0$, $0 < b \neq 1$. *Логарифм n по основанию b* ($\log_b n$) — это такое число x, что $b^x = n$.

Под $\log n$ без указания основания мы будем иметь в виду $\log_2 n$.

R Если $n \in \mathbb{N}$, то $1 + \lfloor \log_b n \rfloor$ — это количество цифр в b-ичной записи n.

Лемма 1.3.3 Пусть $n > 0$, $a, b > 0$, $a, b \neq 1$. Тогда $\log_a n = \frac{\log_b n}{\log_b a}$.

Доказательство. $b^{\log_b a \cdot \log_a n} = a^{\log_a n} = n$, поэтому $\log_b a \cdot \log_a n = \log_b n$. ■

Видим, что $\log_a n$ отличается от $\log_b n$ в константу ($\frac{1}{\log_b a}$) раз, то есть $\log_a n = \Theta(\log_b n)$ для любых оснований a, b. Поэтому для удобства часто используют записи вида $O(\log n)$, не уточняя основание.

Предложение 1.3.4 Для любых $x, y > 0$, $z > 1$ найдётся такое $N > 0$, что для любого $n > N$ верно $\log^x n < n^y < z^n$.

Доказательство. Начнём с первого неравенства. Обозначим $\log n = k$, тогда нам нужно показать, что для достаточно больших k верно $k^x < (2^k)^y = (2^y)^k = z^k$ для $z = 2^y > 1$. Таким образом, первое неравенство следует из второго.

Второе неравенство равносильно $n < z^{\frac{n}{y}} = 2^{\frac{n}{y} \log z}$. Обозначим $m = \frac{n}{y} \log z$, $C = \frac{y}{\log z}$. Тогда нам нужно доказать, что $C \cdot m < 2^m$, где $m = \frac{n}{C}$, C — некая константа, не зависящая от n.

Для любого $n > C^2$ верно $m > C$, то есть $C \cdot m < m^2$. Докажем по индукции, что при $m \geqslant 6$ выполняется $m^2 < 2^m$:

База. При $6 \leqslant m < 7$ верно, что $m^2 < 49 < 64 \leqslant 2^m$.

Переход. Пусть $r \leqslant m < r + 1$ для некоторого целого $r \geqslant 6$, тогда

$$2^{m+1} = 2 \cdot 2^m > 2m^2 > m^2 + 3m > m^2 + 2m + 1 = (m + 1)^2,$$

то есть утверждение верно и для $r + 1 \leqslant m < r + 2$.

Таким образом, для любого $n > \max(C^2, 6 \cdot C)$ выполняется $m > \max(C, 6)$, и

$$n = C \cdot m < m^2 < 2^m = z^{\frac{n}{y}}.$$

■

Теорема 1.3.5 Пусть $x, y > 0$; $b, z > 1$. Тогда $\log_b^x n = o(n^y)$, $n^y = o(z^n)$.

Доказательство. Мы уже знаем, что $\log_b n = \Theta(\log n)$ для любого $b > 1$, тогда верно и $\log_b^x n = \Theta(\log^x n)$ (соответствующие константы из определения $\Theta(\cdot)$ нужно возвести в степень x). Значит, достаточно доказать, что $\log^x n = o(n^y)$.

Рассмотрим $0 < \varepsilon < y$. По лемме 1.3.1, $n^{y-\varepsilon} = o(n^y)$, $n^y = o(n^{y+\varepsilon})$.

Из предложения 1.3.4 следует, что $\log^x n = O(n^{y-\varepsilon})$, $n^{y+\varepsilon} = O(z^n)$ (с константой, равной единице).

Тогда $\log^x n = O(n^{y-\varepsilon}) = o(n^y)$, $n^y = o(n^{y+\varepsilon}) = o(z^n)$. ■

Итак, мы видим, что любая полилогарифмическая функция растёт медленнее любой полиномиальной, а любая полиномиальная функция растёт медленнее любой экспоненциальной.

2. Элементарная арифметика

Современные компьютеры умеют за одну элементарную операцию складывать два 32(или 64)-битных числа. Часто (например, в криптографии) приходится работать с куда более длинными числами. Поговорим о том, как проводить с ними элементарные арифметические операции.

Для удобства будем считать, что на вход алгоритмам даются натуральные числа в двоичной записи. Мы будем работать с числами, имеющими двоичную запись длины n (возможно, с ведущими нулями).

R В произвольном случае можно применить соответствующий алгоритм для n равного максимуму из длин чисел, а в конце определить длину записи результата применения операции (удалить ведущие нули), что потребует $O(n)$ операций и не повлияет на оценку сложности алгоритма.

2.1 Сложение

Используем всем знакомый со школы способ сложения чисел в столбик:

```
add(a, b, n): # a и b - двоичные записи чисел
  int c[n + 1] = 0 # создаём и заполняем нулями массив, куда запишем ответ
  for i = 0..n - 1:
    c[i] = a[i] + b[i]
    if c[i] >= 2:
      c[i + 1] += 1, c[i] -= 2
  return c
```

Время работы алгоритма — $O(n)$, существенно быстрее нельзя, потому что столько времени занимает уже считывание входных данных или вывод ответа.

R Аналогичный алгоритм можно написать для чисел, записанных в b-ичной системе счисления. Если сумма трёх b-ичных чисел помещается в $32(64)$-битный тип данных, то алгоритм всё ещё будет корректен. При этом n-битное число будет иметь примерно $\frac{n}{\log b}$ цифр в b-ичной записи, то есть алгоритм будет работать за $O(\frac{n}{\log b}) = O(n)$, так как $\frac{1}{\log b}$ — это константа. Тем не менее, это может дать ускорение в несколько десятков раз, что безусловно бывает полезно на практике.

2.2 Умножение

Вспомним теперь и школьное умножение чисел в столбик (заметим лишь, что ответ имеет длину не больше $2n$, поскольку $(2^n - 1) \cdot (2^n - 1) < 2^{2n}$):

```
multiply(a, b, n): # a и b - двоичные записи чисел
  int c[2 * n] = 0 # создаём и заполняем нулями массив, куда запишем ответ
  for i = 0..n - 1:
    for j = 0..n - 1:
      c[i + j] += a[i] * b[j]
```

```
  for i = 0..2 * n - 2:
    if c[i] >= 2:
      c[i + 1] += c[i] / 2
      c[i] %= 2
  return c
```

Из-за двух вложенных циклов время работы этого алгоритма — уже $O(n^2)$.

Приведём альтернативный рекурсивный алгоритм умножения двух чисел, пользующийся следующим правилом:

$$a \cdot b = \begin{cases} 2 \cdot (a \cdot \lfloor \frac{b}{2} \rfloor), & \text{если } b \text{ чётно,} \\ a + 2 \cdot (a \cdot \lfloor \frac{b}{2} \rfloor), & \text{иначе.} \end{cases}$$

```
multiply(a, b): # a и b - двоичные записи чисел
  if b == 0:
    return 0
  c = multiply(a, b / 2) # деление нацело
  if b % 2 == 0:
    return 2 * c
  else:
    return 2 * c + a
```

В этой схематичной записи под делением на два имеется ввиду битовый сдвиг вправо (то есть взятие двоичной записи без младшего бита), под умножением на два — битовый сдвиг влево (добавление нуля в начало битовой записи). Остаток по модулю два — это младший бит числа.

Алгоритм произведёт $O(n)$ рекурсивных вызовов, поскольку при каждом вызове длина битовой записи b уменьшается на один. В каждом вызове функции происходят битовый сдвиг влево, битовый сдвиг вправо, и, возможно сложение — всего $O(n)$ элементарных операций. Таким образом, общее время работы снова $O(n^2)$.

Заметим, что если длины битовых записей a и b равны n и m, то время работы обоих алгоритмов можно оценить как $O(nm)$.

2.3 Деление

Пусть теперь мы хотим поделить a на b, то есть найти такие q, r, что $a = qb + r$ и $0 \leqslant r < b$. Здесь работает похожая идея: обозначим за q', r' результат деления $\lfloor \frac{a}{2} \rfloor$ на b, тогда:

$$(q, r) = \begin{cases} (2 \cdot q' + \lfloor \frac{2 \cdot r'}{b} \rfloor, 2 \cdot r' - \lfloor \frac{2 \cdot r'}{b} \rfloor \cdot b), & \text{если } a \text{ чётно,} \\ (2 \cdot q' + \lfloor \frac{2 \cdot r' + 1}{b} \rfloor, 2 \cdot r' + 1 - \lfloor \frac{2 \cdot r' + 1}{b} \rfloor \cdot b), & \text{иначе.} \end{cases}$$

При этом $\lfloor \frac{2 \cdot r'}{b} \rfloor, \lfloor \frac{2 \cdot r' + 1}{b} \rfloor \leqslant 1$. Получаем следующий рекурсивный алгоритм:

```
divide(a, b): # a и b - двоичные записи чисел
  if a == 0:
    return 0, 0 # q = r = 0
  q, r = divide(a / 2, b) # деление нацело
  q = 2 * q, r = 2 * r
  if a % 2 == 1:
    r += 1
  if r >= b:
    q += 1, r -= b
  return q, r
```

Снова имеем $O(n)$ рекурсивных вызовов, в каждом из которых происходит константное число битовых сдвигов и сложений (вычитаний), поэтому время работы снова оценивается как $O(n^2)$.

Альтернативный способ — школьное деление в столбик:

```
divide(a, b): # a и b - двоичные записи чисел
  n = len(a), m = len(b)
  q = 0
  for i = (n - m)..0:
    # умножение на 2 ** k - битовый сдвиг числа на k влево
    c = b * 2 ** i
    if a >= c:
      a = a - c
      q = q + 2 ** i
  return q, a
```

Время работы, как и в предыдущем случае, оценивается как $O(n^2)$. Есть и более точная в некоторых случаях оценка: если длины битовых записей a и b равны n и m, то количество итераций цикла не превосходит $n - m + 1$, поэтому получаем оценку $O(n(n - m + 1))$.

2.4 Алгоритм Карацубы

Можно ли перемножать числа быстрее? Оказывается, что да! Следующий алгоритм был придуман советским математиком Анатолием Карацубой в 1960 году.

Будем для удобства считать, что длина битовой записи чисел n — степень двойки (если это не так, добавим ведущих нулей, при этом длина чисел увеличится не более чем вдвое, что не повлияет на асимптотическую оценку).

Разобьём каждое из чисел на две равных половины: $a = 2^{\frac{n}{2}} a_l + a_r$, $b = 2^{\frac{n}{2}} b_l + b_r$.

Заметим, что

$$ab = (2^{\frac{n}{2}} a_l + a_r) \cdot (2^{\frac{n}{2}} b_l + b_r) = 2^n a_l b_l + 2^{\frac{n}{2}} (a_l b_r + a_r b_l) + a_r b_r.$$

Напишем рекурсивный алгоритм, пользующийся этим равенством: найдём четыре произведения вдвое более коротких чисел $(a_l b_l, a_l b_r, a_r b_l, a_r b_r)$ рекурсивно, после чего вычислим с их помощью ab. Для этого нам понадобится сделать константное число сложений и битовых сдвигов (и то, и другое делается за $\Theta(n)$). Обозначим за $T(n)$ время работы алгоритма на n-битовых числах, тогда мы получаем следующее рекуррентное соотношение:

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + \Theta(n). \tag{2.1}$$

Совсем скоро мы докажем теорему, из которой следует, что $T(n) = \Theta(n^2)$. Пока мы не получили никакого выигрыша по сравнению с предыдущими алгоритмами умножения. Но оказывается, что этот алгоритм можно ускорить, заметив, что

$$a_l b_r + a_r b_l = (a_l + b_l) \cdot (a_r + b_r) - a_l b_l - a_r b_r.$$

Значит, достаточно рекурсивно вычислить три произведения, и время работы алгоритма будет удовлетворять уже

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + \Theta(n). \tag{2.2}$$

Время работы такого алгоритма по той же теореме можно оценить уже как $\Theta(n^{\log_2 3}) = \Theta(n^{1.585\ldots})$.

R Строго говоря, длина чисел $a_l + b_l$, $a_r + b_r$ может оказаться равна $\frac{n}{2} + 1$. Чтобы честно получить рекуррентное соотношение 2.2, можно за линейное время свести умножение $(\frac{n}{2} + 1)$-битных чисел к умножению $\frac{n}{2}$-битных. На самом деле можно показать, что соотношение $T(n) = 3 \cdot T\left(\frac{n}{2} + 1\right) + \Theta(n)$ даёт такую же асимптотическую оценку.

```
multiply(a, b): # a и b - двоичные записи чисел
  n = max(len(a), len(b))
  if n == 1:
    return [a[0] * b[0]] # число из одного бита
  a --> al, ar # делим число a на две равные части
  b --> bl, br # делим число b на две равные части
  x = multiply(al, bl)
  y = multiply(ar, br)
  z = multiply(al + ar, bl + br)
  # умножение на 2 ** k - битовый сдвиг числа на k влево
  return x * 2 ** n + (z - x - y) * 2 ** (n / 2) + y
```

На практике, дойдя в рекурсии до $16(32)$-битных чисел, уже стоит воспользоваться стандартной операцией умножения.

Существуют и более быстрые методы умножения чисел (например, метод, основанный на быстром преобразовании Фурье), но о них мы поговорим позже.

3. Рекуррентные соотношения

3.1 Основная теорема о рекуррентных соотношениях

Алгоритм Карацубы — пример применения метода "разделяй и властвуй" ("divide-and-conquer"), с которым мы ещё не раз встретимся.

Идея этого метода состоит в том, чтобы свести решение задачи к решению нескольких подзадач в несколько раз меньшего размера, после чего восстановить решение исходной задачи с помощью решений подзадач. Сейчас мы докажем достаточно общую теорему, применимую к оценке многих алгоритмов, использующих метод "разделяй и властвуй".

Теорема 3.1.1 — Основная теорема о рекуррентных соотношениях.
Пусть $T(n) = a \cdot T(\lceil \frac{n}{b} \rceil) + \Theta(n^c)$, где $a > 0$, $b > 1$, $c \geqslant 0$. Тогда

$$T(n) = \begin{cases} \Theta(n^c), & \text{если } c > \log_b a, \\ \Theta(n^c \log n), & \text{если } c = \log_b a, \\ \Theta(n^{\log_b a}), & \text{если } c < \log_b a. \end{cases}$$

Доказательство. Пусть мы уже доказали утверждение теоремы для чисел вида $n = b^k$. Рассмотрим такую функцию $k(n)$, что для любого n верно $b^{k(n)-1} < n \leqslant b^{k(n)}$. При уменьшении n количество веток в дереве рекурсии и их размер могли только уменьшиться, поэтому

$$T(n) = O(T(b^{k(n)})) = \begin{cases} O(b^{k(n)c}) = O(n^c), & \text{если } c > \log_b a, \\ O(b^{k(n)c} \log(b^{k(n)})) = O(n^c \log n), & \text{если } c = \log_b a, \\ O(b^{k(n) \log_b a}) = O(n^{\log_b a}), & \text{если } c < \log_b a \end{cases}$$

(мы пользуемся тем, что $b^{k(n)} = b \cdot b^{k(n)-1} < bn = O(n)$).

Доказательство же точной асимптотической оценки (Θ) для произвольного n требует ещё некоторого количества технических выкладок, которые мы опустим. Желающие могут прочитать их в главе 4.6.2 в [1].

Теперь считаем, что $n = b^k$ для некоторого k.

Раскроем рекуррентность:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^c) = \Theta\left(n^c + a \cdot \left(\frac{n}{b}\right)^c\right) + a^2 T\left(\frac{n}{b^2}\right) =$$

$$= \cdots = \Theta\left(n^c \cdot \left(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^{k-1}\right)\right) + a^k T(1) =$$

$$= \Theta\left(n^c \cdot \left(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^{k}\right)\right)$$

(последнее слагаемое $a^k T(1) = \Theta(a^k) = \Theta\left(n^c \cdot \left(\frac{a}{b^c}\right)^k\right)$, так как $T(1) = \Theta(1)$, $n = b^k$).

Получаем геометрическую прогрессию со знаменателем $q = \frac{a}{b^c}$.

Если $c > \log_b a$, то $q < 1$, тогда

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

Если $c = \log_b a$, то $q = 1$, тогда

$$T(n) = \Theta(n^c \cdot (k + 1)) = \Theta(n^c \log_b n) = \Theta(n^c \log n).$$

Наконец, если $c < \log_b a$, то $q > 1$, тогда

$$T(n) = \Theta\left(n^c \cdot \left(q^k + \frac{q^k - 1}{q - 1}\right)\right) = \Theta(n^c q^k) = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}).$$

■

В алгоритме Карацубы $a = 3$, $b = 2$, $c = 1$, это соответствует третьему случаю теоремы, который и даёт $T(n) = \Theta(n^{\log_2 3})$.

II Базовые алгоритмы и структуры данных

4. Базовые структуры данных

4.1 Массив

Массив (array) — структура данных, позволяющая хранить набор значений в ячейках, пронумерованных индексами (или набором индексов в случае многомерного массива) из некоторого отрезка целых чисел. Встроен в большинство современных языков программирования.

Массив позволяет за константное ($O(1)$) время получать доступ к элементу по индексу, а также изменять этот элемент. При этом массив имеет фиксированный размер, поэтому не поддерживает операций вставки и удаления элементов. Если хочется вставить или удалить элемент, можно создать новый массив нужного размера и скопировать информацию в него, но это потребует $\Theta(n)$ операций, где n — длина массива.

Поиск элемента в массиве по значению также имеет сложность $\Theta(n)$, так как нужно проверить все элементы массива. Если массив специально упорядочен (например, элементы массива расположены в возрастающем порядке), то поиск можно делать быстрее (скоро мы изучим алгоритм двоичного поиска).

Немного о работе с массивом в C++:

```
int a[n]; // объявить массив длины n, нумерация ячеек от 0 до n-1
a[i]; // обращение к i-му элементу массива
a[i] = x; // присвоить x в i-ю ячейку массива

int b[n][m]; // объявить двумерный массив, нумерация ячеек двумя индексами,
             // от 0 до n-1 и от 0 до m-1
b[i][j]; // обращение к j-му элементу i-й строки массива
b[i][j] = y; // присвоение
```

4.2 Связный список

Связный список (linked list) состоит из узлов, каждый из которых содержит данные и ссылки на соседние узлы. В *двусвязном списке* поддерживаются ссылки на следующий и предыдущий узел, в *односвязном списке* — только на следующий.

Также поддерживаются ссылки на начало и конец списка (их часто называют головой и хвостом). Для того, чтобы посетить все узлы, можно начать с головы и переходить по ссылке на следующий узел, пока он существует.

Преимущество списка перед массивом — возможность вставлять и удалять элементы за $O(1)$.

Недостаток списка — невозможность быстрого доступа к произвольному элементу. Так, доступ к i-му элементу можно получить, лишь i раз пройдя по ссылке вперёд, начиная из головы списка (то есть за $\Theta(i)$).

Удобно делать голову и хвост фиктивными элементами и не хранить в них никаких данных, тогда функции вставки и удаления элементов пишутся проще.

Примерная реализация двусвязного списка на C++:

```
struct Node { // один узел списка
  Node *next, *prev; // указатели на следующий и предыдущий узлы
  int x; // данные, хранящиеся в узле
};

struct List {
  Node *head, *tail; // указатели на начало и конец списка

  List() { // инициализируем пустой список - создаём фиктивные head и tail
           // и связываем их друг с другом
    head = new Node();
    tail = new Node();
    head->next = tail;
    tail->prev = head;
  }

  void pushBack(int x) { // вставить x в конец списка
    Node *v = new Node();
    v->x = x, v->prev = tail->prev, v->next = tail;
    v->prev->next = v, tail->prev = v;
  }

  void insert(Node *v, int x) { // вставить x после v
    Nove *w = new Node();
    w->x = x, w->next = v->next, w->prev = v;
    v->next = w, w->next->prev = w;
  }

  void erase(Node *v) { // удалить узел v
    v->prev->next = v->next;
    v->next->prev = v->prev;
    delete v;
  }

  Node *find(int x) { // найти x в списке
    for (Node *v = head->next; v != tail; v = v->next) {
      if (v->x == x) {
        return v;
      }
    }
    return NULL;
  }
};
```

Можно хранить узлы списка в массиве, тогда вместо указателей можно использовать числа — номера ячеек массива. Но тогда нужно либо заранее знать количество элементов в списке, либо использовать динамический массив (который мы скоро изучим).

Можно также использовать встроенный в C++ двусвязный список — `std::list`.

Односвязный список использует меньше дополнительной памяти, но не позволяет перемещаться по списку в сторону начала. Также из него сложнее удалять элементы — не получится удалить элемент, имея ссылку только на него. Нужно как-то получить доступ к предыдущему элементу, чтобы пересчитать ссылку из него на следующий.

4.3 Динамический массив

Пусть мы хотим научиться вставлять новые элементы в конец массива. Можно попытаться сразу создать массив достаточно большого размера, и в отдельной переменной поддер-

живать его реальную длину. Но далеко не всегда максимальное количество элементов известно заранее.

Поступим следующим образом: если мы хотим вставить новый элемент, а место в массиве закончилось, то создадим новый массив вдвое большего размера, и скопируем данные туда.

```
int *a; // используемая память
int size; // размер памяти
int n; // реальный размер массива

void pushBack(int x) { // вставить x в конец массива
  if (n == size) {
    int *b = new int[size * 2];
    for (int i = 0; i < size; ++i) {
      b[i] = a[i];
    }
    delete[] a;
    a = b;
    size *= 2;
  }
  a[n] = x;
  n += 1;
}

void popBack() { // удалить последний элемент
  n -= 1;
}
```

В C++ можно и нужно использовать встроенную реализацию динамического массива — `std::vector`.

Каждая конкретная операция вставки элемента может работать долго — $\Theta(n)$, где n — длина массива. Зато можно показать, что *среднее время работы* операции вставки — $O(1)$. Докажем этот факт при помощи метода *амортизационного анализа*, который нам ещё не раз пригодится.

Предложение 4.3.1 Среднее время работы операции вставки — $O(1)$.

R Это утверждение равносильно тому, что суммарное время работы m операций вставки — $O(m)$.

Доказательство. Операция вставки работает не за $O(1)$ только тогда, когда происходит удвоение размера используемой памяти. Заметим, что если такая операция увеличила размер с $size$ до $2 \cdot size$, то хотя бы $size/2$ предыдущих операций вставки работали за $O(1)$ (может быть, и больше, если происходили также удаления элементов). Тогда суммарное время работы этих $size/2 + 1$ операций есть $O(size)$, поэтому среднее время их работы есть $O(1)$. ■

4.4 Стек, очередь, дек

Стек (stack) позволяет поддерживать список элементов, вставлять элемент в конец списка, а также удалять элемент из конца списка. Часто также говорят, что он организован по принципу LIFO (last in — first out).

Очередь (queue) организована по принципу FIFO (first in — first out). Она позволяет вставлять элемент в конец списка, а также удалять элемент из начала списка.

Дек (deque — double ended queue), или *двусторонняя очередь*, отличается от обычной очереди тем, что позволяет добавлять и удалять элементы как в начало, так и в конец списка.

Все эти структуры можно реализовать с помощью связного списка. Для стека и очереди хватит односвязного списка: в стеке достаточно поддерживать ссылку на предыдущий элемент, а в очереди — на следующий. Для реализации дека понадобится двусвязный список.

Также все три структуры можно реализовать с помощью динамического массива. В случае дека нужно научиться добавлять элементы в начало, для чего можно зациклить массив. То же самое можно сделать в реализации очереди, чтобы переиспользовать освободившееся место в начале массива. Приведём примерную реализацию дека на динамическом массиве:

```
int *a; // используемая память
int size; // размер памяти
int b, e; // дек использует ячейки в диапазоне [b,e), или,
          // если b >= e, в диапазонах [b,size) и [0,e)

int getSize() { // узнать количество элементов в деке
  if (b < e) {
    return e - b;
  } else {
    return e - b + size;
  }
}

void pushBack(int x) { // вставить x в конец дека
  if (getSize() == size) {
    ... // выделяем вдвое больше памяти
  }
  a[e] = x;
  e = (e + 1) % size;
}

void popBack() { // удалить элемент из конца дека
  e = (e - 1 + size) % size;
}

void pushFront(int x) { // вставить x в начало дека
  if (getSize() == size) {
    ... // выделяем вдвое больше памяти
  }
  a[b] = x;
  b = (b - 1 + size) % size;
}

void popFront() { // удалить элемент из начала дека
  b = (b + 1) % size;
}
```

В C++ существуют встроенные реализации этих структур — `std::stack`, `std::queue`, `std::deque`.

5. Двоичный поиск

5.1 Двоичный поиск

Если элементы массива расположены в возрастающем порядке (то есть массив *отсортирован*), то искать элемент в массиве можно быстрее, чем за линейное время.

Пусть мы хотим найти x в отсортированном массиве a длины n. Снова применим идею метода "разделяй и властвуй": посмотрим на элемент в середине массива — $a[\frac{n}{2}]$. Если $a[\frac{n}{2}] = x$, то мы нашли x в массиве a. Если $a[\frac{n}{2}] > x$, то x может найтись только в $a[0, 1, \ldots, \frac{n}{2} - 1]$. Если же $a[\frac{n}{2}] < x$, то x может найтись только в $a[\frac{n}{2} + 1, \ldots, n - 1]$. В любом из последних двух случаев, мы вдвое уменьшили длину отрезка, на котором нужно искать x.

Теперь можно рекурсивно повторить те же рассуждения — посмотреть на середину нового подоотрезка массива, и либо в середине найдётся x, либо мы поймём, в какой половине нового подотрезка нужно его искать. Будем повторять эти действия, пока не найдём x, либо не дойдём до подотрезка нулевой длины. Поскольку на каждом шаге длина отрезка уменьшается вдвое, это произойдёт через $O(\log n)$ шагов.

Также можно оценить время работы нашего алгоритма с помощью теоремы 3.1.1: время работы алгоритма в худшем случае (когда x так и не нашёлся, либо нашёлся в самом конце) удовлетворяет рекуррентному соотношению $T(n) = T(\frac{n}{2}) + \Theta(1)$. Это соответствует $a = 1$, $b = 2$, $c = 0$ в формулировке теоремы. $c = 0 = \log_2 1 = \log_b a$, тогда по теореме время работы алгоритма в худшем случае есть $\Theta(\log n)$.

Примерная реализация алгоритма двоичного поиска:

```
binarySearch(a, n, x): # ищет x в массиве a длины n, возвращает индекс,
                       # по которому лежит x, или -1, если x в массиве нет
  l = 0, r = n - 1 # границы интересующего нас подотрезка массива
  while l <= r: # пока подотрезок непуст
    m = (l + r) / 2
    if a[m] == x:
      return m
    if a[m] < x:
      l = m + 1
    else:
      r = m - 1
  return -1 # x так и не нашёлся
```

5.2 Левое вхождение

Возможно, x встречается в массиве несколько раз (так как массив отсортирован, эти вхождения образуют подотрезок массива). Научимся находить количество вхождений x в массив (то есть длину этого подотрезка).

Достаточно найти индексы самого левого и самого правого вхождений x в массив, тогда количество вхождений — это разность этих индексов плюс один. Начнём с левого вхождения.

Будем снова поддерживать границы l и r, но уже со следующим условием: пусть в любой момент времени выполняется $a[l] < x$ и $a[r] \geqslant x$. Удобно мысленно добавить

к массиву фиктивные элементы $a[-1] = -\infty$ и $a[n] = \infty$, тогда не нужно отдельно обрабатывать случаи, когда все элементы меньше x, или все элементы больше или равны x.

Сам алгоритм становится только проще: смотрим на середину отрезка (l, r), и, в зависимости от значения элемента массива с этим индексом, сдвигаем l или r так, чтобы требуемые от них условия не нарушились. Когда l и r указывают на соседние элементы (то есть $l = r - 1$), всё ещё верно, что $a[l] < x$, $a[r] \geqslant x$. Тогда если $a[r] = x$, то r — индекс левого вхождения x, иначе x в массиве не встречается.

Примерная реализация:

```
lowerBound(a, n, x): # возвращает минимальное i такое, что a[i] >= x
  l = -1, r = n # l и r изначально указывают на фиктивные элементы
  while r - l > 1:
    m = (l + r) / 2
    if a[m] < x:
      l = m # условие на l не нарушилось
    else:
      r = m # условие на r не нарушилось
  return r

getLeftEntry(a, n, x): # возвращает номер левого вхождения или -1
  i = lowerBound(a, n, x)
  if i == n or a[i] != x:
    return -1
  return i
```

Правое вхождение ищется примерно так же: нужно требовать от границ поиска $a[l] \leqslant x$ и $a[r] > x$, тогда в конце l — правое вхождение, если $a[l] = x$, иначе x в массиве не встречается. Заметим, что в реализации самого поиска нужно поправить всего одну строчку: `a[m] < x` заменить на `a[m] <= x`.

В C++ есть встроенные функции `std::lower_bound` и `std::upper_bound`. Первая возвращает итератор, указывающий на первый элемент, больше или равный x, вторая — на первый элемент, строго больший x. Пример использования:

```
i = lower_bound(a, a + n, x) - a;
// минимальное i такое, что a[i] >= x; n, если все элементы a меньше x
i = upper_bound(a, a + n, x) - a;
// минимальное i такое, что a[i] > x; n, если все элементы a меньше или равны x
i = upper_bound(a, a + n, x) - a - 1;
// максимальное i такое, что a[i] <= x; -1, если все элементы a больше x
cnt = upper_bound(a, a + n, x) - lower_bound(a, a + n, x);
// количество вхождений x в a
```

5.3 Двоичный поиск по функции

Можно рассмотреть и ещё более общую версию алгоритма. Пусть есть некоторая функция $f(i)$ такая, что $f(i) = 0$ для всех i, меньших некоторого порога t, и $f(i) = 1$ для всех $i \geqslant t$. Тогда этот порог можно найти с помощью алгоритма двоичного поиска.

Идея та же, что и при поиске левого вхождения: сначала возьмём такие границы поиска $l = L$, $r = R$, что $f(L) = 0$, $f(R) = 1$. После этого на каждом шаге будем сдвигать одну из границ l, r, в зависимости от значения $f\left(\frac{l+r}{2}\right)$. В тот момент, когда l и r отличаются на единицу, r — искомый порог. Если считать, что значение функции в точке вычисляется за $O(1)$, то время работы алгоритма — $O(\log(R - L))$, где L, R — начальные границы поиска.

Заметим, что алгоритмы поиска левого и правого вхождения являются частными случаями этого алгоритма: в случае левого вхождения можно взять функцию f такую,

что $f(i) = 1$ тогда и только тогда, когда $a[i] \geqslant x$; в случае правого вхождения — $f(i) = 1$ тогда и только тогда, когда $a[i] > x$ (и нас интересует максимальное i такое, что $f(i) = 0$, то есть в конце алгоритма нужно вернуть l, а не r).

5.4 Двоичный поиск по функции с вещественным аргументом

Двоичный поиск можно делать и по функции, принимающей значения во всех вещественных точках, а не только в целых. Решим для примера такую задачу: дана монотонно возрастающая непрерывная функция $f : \mathbb{R} \to \mathbb{R}$, и точки L, R такие, что $f(L) < 0$, $f(R) \geqslant 0$. Найдём точку x такую, что $f(x) = 0$.

R Если такие L, R не даны, но известно, что они существуют, то их можно найти алгоритмом *экспоненциального поиска*: начнём с $R = 1$ и будем удваивать R, пока $f(R) < 0$. Аналогично, начнём с $L = -1$ и будем удваивать L, пока $f(L) \geqslant 0$. При этом мы затратим $O(\log |R'| + \log |L'|)$ действий, где $R' \geqslant 0$, $L' \leqslant 0$ — любые такие, что $f(L') < 0$, $f(R') \geqslant 0$.

Поскольку f принимает вещественные значения, мы не всегда можем найти значение такого x абсолютно точно. Тем не менее, мы можем найти такое x с погрешностью ε: найдём такое x, что существует x' такое, что $f(x') = 0$, $|x - x'| < \varepsilon$.

Алгоритм практически не меняется: начинаем с границ $l = L$, $r = R$, на каждом шаге сдвигаем одну из границ в зависимости от значения $f\left(\frac{l+r}{2}\right)$. Единственное отличие: мы завершаем алгоритм в тот момент, когда $r - l < \varepsilon$. Поскольку в этот момент всё ещё $f(l) < 0$, $f(r) \geqslant 0$, а функция f непрерывна, на отрезке $[l, r]$ найдётся x' такое, что $f(x') = 0$. Поскольку длина отрезка меньше ε, любая из границ l, r подойдёт в качестве ответа. Время работы алгоритма (при условии, что значение f вычисляется за $O(1)$) — $O\left(\log\left(\frac{R-L}{\varepsilon}\right)\right)$.

Если вычисления проводятся при помощи стандартных вещественнозначных типов данных (например, `double` в C++), то из-за накопления погрешности при пересчёте границ условие $r - l < \varepsilon$ может никогда не выполниться. Чтобы избежать этого, вместо `while` напишем цикл, делающий столько итераций, сколько нужно, чтобы условие точно выполнилось. Пример реализации:

```
findRoot(L, R, eps):
  l = L, r = R
  k = log((r - l) / eps) # через k итераций условие r - l < eps выполнится
  for i = 0..(k - 1):
    m = (l + r) / 2
    if f(m) < 0:
      l = m # условие на l не нарушилось
    else:
      r = m # условие на r не нарушилось
  return r
```

5.5 Метод двух указателей

Иногда много двоичных поисков на одних и тех же данных можно соптимизировать с помощью так называемого *метода двух указателей*. Сделаем это на примере следующей задачи: дан массив a длины n, состоящий из неотрицательных целых чисел. Нужно найти максимальный по длине подотрезок массива, сумма на котором не превосходит M.

Удобнее работать не с отрезками, а с полуинтервалами: паре l, r будем сопоставлять полуинтервал $a[l, r)$, то есть элементы массива с индексами $l, l + 1, \ldots, r - 1$ (это прак-

тически всегда позволяет писать меньше плюс-минус единиц в индексах массивов, что делает код короче). Самое простое решение — перебрать все возможные полуинтервалы:

```
maxLen = 0, ansL = 0, ansR = 0 # здесь будем хранить ответ
for l = 0..(n - 1): # перебираем левую границу
  sum = 0 # будем считать сумму на текущем полуинтервале
  for r = (l + 1)..n:
    sum += a[r - 1]
    if sum <= M and r - l > maxLen:
      maxLen = r - l, ansL = l, ansR = r
```

Получилось решение за $\Theta(n^2)$. Заметим, что при фиксированной левой границе сумма на полуинтервале не уменьшается при увеличении правой границы. Значит сколько-то первых правых границ подойдут, а все последующие уже не подойдут. При этом нас интересует максимальная подходящая правая граница. Её можно найти двоичным поиском, таким образом соптимизировав решение до $\Theta(n \log n)$:

R Для того, чтобы быстро находить сумму на произвольном полуинтервале, заранее предподсчитаем *префиксные суммы* — массив p такой, что $p[i] = \sum_{j=0}^{i-1} a_j$. Тогда сумма на полуинтервале $a[l, r)$ равна $\sum_{i=l}^{r-1} a_i = p[r] - p[l]$.

```
maxLen = 0, ansL = 0, ansR = 0
int p[n + 1] # предподсчитываем префиксные суммы
p[0] = 0
for i = 1..n:
  p[i] = p[i - 1] + a[i - 1]

for l = 0..(n - 1):
  sl = l, sr = n + 1 # границы поиска, sum a[l..sl) <= M, sum a[l..sr) > M
  while sr - sl > 1:
    sm = (sl + sr) / 2
    if p[sm] - p[l] <= M:
      sl = sm
    else:
      sr = sm
  if sl - l > maxLen: # sl - максимальная подходящая правая граница
    maxLen = sl - l, ansL = l, ansR = sl
```

Перейдём теперь собственно к методу двух указателей. Пусть r — максимальная подходящая правая граница для левой границы $l - 1$. Тогда $M \geqslant \sum_{i=l-1}^{r-1} a_i \geqslant \sum_{i=l}^{r-1} a_i$, то есть граница r подойдёт и для l. Будем вместо двоичного поиска искать границу r так же, как и в самой первой версии алгоритма: просто перебирая все границы подряд. Но начнём перебор не с $l+1$, а с правой границы, найденной на предыдущем шаге. Поскольку теперь правая граница только увеличивается в ходе алгоритма, суммарно за всё время выполнения алгоритма она сдвинется не более, чем n раз. Тогда общее время работы алгоритма — $\Theta(n)$.

```
maxLen = 0, ansL = 0, ansR = 0
r = 0, sum = 0
for l = 0..(n - 1): # перебираем левую границу
  while r < n and sum + a[r] <= M:
    sum += a[r]
    r += 1
  if sum <= M and r - l > maxLen:
    maxLen = r - l, ansL = l, ansR = r
  sum -= a[l]
```

6. Сортировки

6.1 Квадратичные сортировки

Существует множество различных алгоритмов, сортирующих массив длины n за $\Theta(n^2)$. Мы поговорим лишь о двух из них.

Сортировка выбором

Сортировка выбором (selection sort) на i-м шаге находит i-й по возрастанию элемент и ставит его на i-ю позицию. Поскольку первые $i-1$ элементов в этот момент уже стоят на своих позициях, достаточно просто найти минимальный элемент в подотрезке $[i, n)$.

```
for i = 0..(n - 1):
  minPos = i
  for j = (i + 1)..(n - 1):
    if a[minPos] > a[j]:
      minPos = j
  swap(a[i], a[minPos])
```

Сортировка вставками

На i-м шаге сортировки вставками (insertion sort) первые i элементов массива (образующие *префикс* длины i) расположены в отсортированном порядке. i-й шаг состоит в том, что i-й элемент массива вставляется в нужную позицию остортированного префикса.

```
for i = 1..(n - 1):
  for (j = i; j > 0 and a[j] < a[j - 1]; --j):
    swap(a[j], a[j - 1])
```

Время работы

В обеих сортировках два вложенных цикла в худшем случае дают время работы $\Theta(n^2)$.

Сортировка выбором полезна тем, что делает $\Theta(n)$ операций `swap`. Это свойство пригождается, когда сортируются тяжёлые объекты, и каждая операция `swap` занимает много времени.

Инверсией называют такую пару элементов на позициях $i < j$, что $a_i > a_j$. Последовательность элементов является отсортированной тогда и только тогда, когда в ней нет инверсий. Время работы сортировки вставками можно оценить как $\Theta(n + Inv(a))$, где $Inv(a)$ — количество инверсий в массиве a, так как на каждом шаге внутреннего цикла количество инверсий в массиве уменьшается ровно на один. Значит, на отсортированном (или почти отсортированном) массиве время работы сортировки вставками составит $O(n)$.

Стабильность

Сортировка называется *стабильной*, если она оставляет равные элементы в исходном порядке. Обычно такое свойство сортировки нужно, когда помимо данных, по которым производится сортировка, в элементах массива хранятся какие-то дополнительные данные. Например, если дан список участников соревнований в алфавитном порядке, и хочется

отсортировать их по убыванию набранных баллов так, чтобы участники с равным числом баллов всё ещё шли в алфавитном порядке.

Сортировка выбором стабильной не является (например, массив $[5, 5, 3, 1]$ после первого шага изменится на $[1, 5, 3, 5]$, при этом порядок пятёрок поменялся). Сортировка вставками стабильна, так как меняет местами только соседние элементы, образующие инверсию, поэтому ни в какой момент времени не поменяет порядок равных элементов.

R Любую сортировку можно сделать стабильной, если вместо исходных элементов сортировать пары (элемент, его номер в исходном массиве), и при сравнении пар сначала сравнивать элементы, а при равенстве номера.

Применение на практике

Сейчас мы перейдём в алгоритмам сортировки, работающим за $\Theta(n \log n)$. Квадратичные сортировки, благодаря малой константе в оценке времени работы, работают быстрее более сложных алгоритмов на коротких массивах. На практике часто применяется гибридный подход: алгоритм сортировки, использующий метод "разделяй и властвуй", работает, пока массив не поделится на части достаточно малого размера, после чего на них запускается алгоритм квадратичной сортировки.

6.2 Сортировка слиянием (Merge sort)

Сортировка слиянием (Von Neumann, 1945) использует метод "разделяй и властвуй" следующим образом: массив делится на две части, каждая из них сортируется рекурсивно, после чего две отсортированных части *сливаются* при помощи метода двух указателей.

```
mergeSort(a, l, r): # сортирует a[l, r)
  if r - l <= 1:
    return
  m = (l + r) / 2
  mergeSort(a, l, m)
  mergeSort(a, m, r)
  merge(a, l, m, r)

# merge использует вспомогательный массив buf достаточно большого размера
merge(a, l, m, r): # сливает два отсортированных отрезка a[l, m) и a[m, r)
  for (i = l, j = m, k = 0; i < m or j < r; ):
    if i == m or (j < r and a[i] > a[j]):
      buf[k] = a[j]
      k += 1, j += 1
    else:
      buf[k] = a[i]
      k += 1, i += 1
  for i = 0..(r - l - 1):
    a[l + i] = buf[i]
```

Время работы сортировки слиянием можно оценить с помощью рекуррентного соотношения $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$. По теореме 3.1.1 получаем $T(n) = \Theta(n \log n)$.

Сортировка слиянием стабильна (поскольку функция `merge` не меняет относительный порядок равных элементов).

Недостатком сортировки слиянием является то, что она требует $\Theta(n)$ дополнительной памяти (вспомогательный массив в `merge`).

6.3 Быстрая сортировка (Quicksort)

Быстрая сортировка (Hoare, 1959) также использует метод "разделяй и властвуй", но немного по-другому. Возьмём какой-нибудь элемент массива — x. Поделим массив на три части так, что в первой все элементы меньше x, во второй равны x, в третьей больше x (это можно сделать за линейное от длины массива время, например, с помощью трёх вспомогательных массивов). Остаётся рекурсивно отсортировать первую и третью части.

```
quickSort(a):
  if len(a) <= 1:
    return
  x = randomElement(a) # x - случайный элемент a
  a --> l (< x), m (= x), r (> x) # делим a на три части
  return quickSort(l) + m + quickSort(r)
```

На практике, чтобы алгоритм работал быстрее и использовал меньше дополнительной памяти, используют более хитрый способ деления массива на части:

```
# partition выбирает x - случайный элемент a[l, r],
# переставляет местами элементы a[l, r] и возвращает m (l <= m < r) такое, что
# все элементы a[l, m] меньше или равны x, все элементы a[m + 1, r] больше или равны x
int partition(a, l, r):
  p = random(l, r), x = a[p]
  swap(a[p], a[l]) # теперь x стоит на l-й позиции
  i = l, j = r
  while i <= j:
    while a[i] < x:
      i += 1
    while a[j] > x:
      j -= 1
    if i >= j:
      break
    swap(a[i], a[j])
    i += 1, j -= 1
  return j

quickSort(a, l, r): # сортирует a[l, r]
  if l == r:
    return
  m = partition(a, l, r)
  quickSort(a, l, m)
  quickSort(a, m + 1, r)
```

Предложение 6.3.1 Функция `partition` работает корректно, то есть i и j не выходят за границы l, r, функция возвращает такое m, что $l \leqslant m < r$, все элементы $a[l, m]$ меньше или равны x, все элементы $a[m + 1, r]$ больше или равны x.

Доказательство. Заметим, что в любой момент времени все элементы $a[l, i)$ меньше или равны x, все элементы $a(j, r]$ больше или равны x.

На первой итерации внешнего цикла в момент проверки условия на 13-й строке верно, что $i = l \leqslant j$. Значит либо мы сразу выйдем из внешнего цикла (если $i = j = l < r$), либо выполнятся 15–16 строки, после чего всегда будет верно, что $a[l] \leqslant x$, $a[r] \geqslant x$, $j < r$. При этом 15–16 строки выполнились только при $i < j$, тогда даже после их выполнения i, j не вышли за границы l, r.

Отдельно отметим, что после первой итерации внешнего цикла точно выполняется неравенство $j < r$.

На всех последующих итерациях внешнего цикла после выполнения 9–12 строк будут верны неравенства $j \geqslant l$, $i \leqslant r$ (так как $a[l] \leqslant x$, $a[r] \geqslant x$). При этом если условие на

13-й строке не выполняется (то есть $i < j$), то даже после выполнения 15–16 строк i, j не выйдут за границы l, r.

Наконец, по итогам выполнения функции $l \leqslant m = j < r$, все элементы $a[m+1, r] = a(j, r]$ больше или равны x, все элементы $a[l, m] = a[l, j]$ меньше или равны x (так как это верно для элементов $a[l, i)$, $j \leqslant i$, причём $j = i$ только если мы вышли из внешнего цикла на 14-й строке, что возможно, только если $a[j] = x$). ■

Заметим, что такая реализация является нестабильной сортировкой.

Оценка времени работы

В худшем случае массив каждый раз будет делиться очень неравномерно, и почти все элементы будут попадать в одну из частей. Время работы в худшем случае можно оценить с помощью рекуррентного соотношения $T(n) = T(n-1) + \Theta(n)$, раскрыв которое, получаем $T(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$.

В лучшем же случае массив каждый раз будет делиться на две примерно равные части. Получаем рекуррентное соотношение $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$, тогда по теореме 3.1.1 получаем $T(n) = \Theta(n \log n)$.

Оказывается, время работы алгоритма в среднем намного ближе к лучшему случаю, чем к худшему. Для того, чтобы это доказать, нам понадобится терминология из теории вероятностей.

Определение 6.3.1 *Математическое ожидание* случайной величины X, принимающей значение x_1 с вероятностью p_1, x_2 с вероятностью p_2, …, x_n с вероятностью p_n ($p_1 + \cdots + p_n = 1$) есть

$$\mathbb{E}X = \sum_{i=1}^{n} p_i x_i.$$

Теорема 6.3.2 Математическое ожидание времени работы алгоритма быстрой сортировки есть $O(n \log n)$.

Доказательство. Мы докажем теорему только в случае, когда все элементы массива попарно различны. Кроме того, будем рассматривать версию алгоритма, делящую массив на три части (элементы меньше x; элементы, равные x; элементы больше x).

Для удобства будем использовать индексы элементов не в исходном, а в уже отсортированном массиве: пусть отсортированный массив имеет вид $z_1, z_2, \ldots, z_n$, исходный массив a — это какая-то перестановка элементов z_i.

Время работы алгоритма быстрой сортировки пропорционально количеству выполненных сравнений. Обозначим количество выполненных сравнений за $T(n)$, достаточно оценить его математическое ожидание.

z_i и z_j могли сравниваться, только если на каком-то шаге алгоритма один из них был выбран в качестве x. Заметим, что такой элемент не участвует в последующих рекурсивных вызовах, поэтому каждую пару элементов алгоритм сравнит не более, чем один раз.

Пусть Γ — множество всех возможных сценариев выполнения алгоритма, $p(A)$ — вероятность того, что произошёл сценарий $A \in \Gamma$. Для каждой пары i, j введём величину $\chi(A, i, j)$, равную единице, если z_i и z_j сравнивались в сценарии A, и нулю, если не сравнивались. Математическое ожидание количества выполненных алгоритмом сравнений равняется

$$\mathbb{E}T(n) = \sum_{A \in \Gamma} \left(p(A) \sum_{1 \leqslant i < j \leqslant n} \chi(A, i, j) \right) = \sum_{1 \leqslant i < j \leqslant n} \sum_{A \in \Gamma} p(A) \chi(A, i, j).$$

Остаётся для каждой пары $1 \leqslant i < j \leqslant n$ посчитать $\sum_{A \in \Gamma} p(A)\chi(A, i, j)$, то есть вероятность, с которой z_i и z_j сравнивались между собой.

Посмотрим на момент, в который z_i и z_j при делении массива попали в разные части. Заметим, что массив, который делился на части в этот момент, после сортировки будет являться подотрезком отсортированного массива z, тогда вместе с z_i и z_j он содержит весь подотрезок $z[i, j]$.

Поскольку z_i и z_j при делении попали в разные части, в качестве x точно был выбран один из элементов $z[i, j]$. При этом z_i и z_j сравнивались между собой только если в качестве x был выбран один из них. Поскольку x выбирается среди всех элементов равновероятно, вероятность того, что между z_i и z_j произошло сравнение, равняется $\frac{2}{j-i+1}$. Получаем

$$\mathbb{E}T(n) = \sum_{1 \leqslant i < j \leqslant n} \sum_{A \in \Gamma} p(A)\chi(A, i, j) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} =$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < 2n \sum_{k=1}^{n} \frac{1}{k} = O(n \log n).$$

Последний переход можно понять, например, следующим способом:

$$\sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \left(\frac{1}{2} + \frac{1}{3}\right) + \left(\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}\right) + \cdots <$$

$$< \frac{1}{1} + \left(\frac{1}{2} + \frac{1}{2}\right) + \left(\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}\right) + \cdots \leqslant \lfloor \log_2 n \rfloor + 1.$$

■

R Отсюда следует и более сильное утверждение: время работы алгоритма есть $O(n \log n)$ с вероятностью, близкой в единице. Это следует из следующего утверждения, известного как *неравенство Маркова*: для неотрицательной случайной величины X с математическим ожиданием $\mathbb{E}(X)$ вероятность того, что $X > k \cdot \mathbb{E}(X)$, не превосходит $\frac{1}{k}$. Это верно, так как в противном случае математическое ожидание X оказалось бы больше $\frac{1}{k} \cdot k \cdot \mathbb{E}(X) = \mathbb{E}(X)$.

В нашем случае, например, для $k = 100$ получаем, что с вероятностью 99% время работы не превосходит $O(100 \cdot n \log n) = O(n \log n)$.

Взятие случайного элемента — достаточно медленная операция, поэтому на практике вместо случайного элемента часто берут какой-то конкретный, например самый левый, самый правый, или средний; также часто используют средний по значению из этих трёх. Для подобных версий алгоритма можно построить массив, на котором сортировка будет работать за $\Theta(n^2)$. С этим борются разными способами, например, когда глубина рекурсии превышает $\log n$, переключаются на какой-нибудь другой алгоритм сортировки.

На практике алгоритм быстрой сортировки оказывается одним из самых быстрых и часто используемых. Встроенная в C++ сортировка — `std::sort`, использует алгоритм Introsort, который начинает сортировать массив алгоритмом быстрой сортировки, на большой глубине рекурсии переключается на heapsort (который мы скоро изучим), а массивы совсем небольшой длины сортирует сортировкой вставками.

6.4 Поиск k-й порядковой статистики

k-я порядковая статистика на массиве из n элементов — это k-й по возрастанию элемент. Например, при $k = 1$ это минимум, при $k = n$ — максимум. *Медиана* — это элемент, который оказался бы в середине массива, если бы его отсортировали. Если длина массива чётна, то в нём есть две медианы на позициях $\lfloor \frac{n+1}{2} \rfloor$ и $\lceil \frac{n+1}{2} \rceil$. Для определённости под медианой будем иметь в виду $\lfloor \frac{n+1}{2} \rfloor$-ю порядковую статистику.

Минимум и максимум в массиве очень легко ищется за $O(n)$ одним проходом по всем элементам массива. k-й по возрастанию элемент так просто уже не найти.

Можно отсортировать массив за $O(n \log n)$, тогда k-я порядковая статистика окажется на k-й позиции (если нумеровать элементы массива, начиная с единицы). Однако существуют и более быстрые алгоритмы, находящие k-ю порядковую статистику для произвольного k за $O(n)$.

Вернёмся к алгоритму быстрой сортировки. Когда мы поделили массив на две части, можно понять, в какой из этих частей находится k-й по возрастанию элемент: если размер левой части хотя бы k, то он находится в ней, иначе он находится в правой части. Тогда, если нас интересует не весь отсортированный массив, а только k-й по возрастанию элемент, можно сделать рекурсивный запуск только от той части, в которой он лежит.

```
kthElement(a, l, r, k): # находит k-ю порядковую статистику в a[l, r]
  if l == r:
    return a[l]
  m = partition(a, l, r)
  if m - l + 1 >= k:
    return kthElement(a, l, m, k)
  else:
    return kthElement(a, m + 1, r, k - (m - l + 1))
```

В худшем случае такой алгоритм будет работать по-прежнему за $\Theta(n^2)$. Однако оказывается, что оценка среднего времени работы после такой оптимизации улучшается с $O(n \log n)$ до $O(n)$.

Теорема 6.4.1 Математическое ожидание времени работы алгоритма `kthElement` есть $O(n)$.

Доказательство. Мы докажем теорему в случае, когда все элементы массива попарно различны.

Будем говорить, что алгоритм находится в *j-й фазе*, если размер текущего отрезка массива не больше $\left(\frac{3}{4}\right)^j n$, но строго больше $\left(\frac{3}{4}\right)^{j+1} n$. Оценим время работы алгоритма в каждой фазе отдельно.

Назовём элемент *центральным*, если хотя бы четверть элементов в текущем отрезке массива меньше его, и хотя бы четверть больше. Если в качестве разделителя x был выбран центральный элемент, то размер отрезка, от которого будет сделан рекурсивный запуск, будет не больше $\frac{3}{4}$ от размера текущего отрезка, то есть текущая фаза алгоритма точно закончится. При этом вероятность выбрать центральный элемент равна $\frac{1}{2}$ (так как ровно половина элементов отрезка являются центральными). Тогда математическое ожидание количества рекурсивных запусков, сделанных в течение j-й фазы, не превосходит

$$1 + \frac{1}{2}\left(1 + \frac{1}{2}\left(1 + \cdots\right)\right) = 1 + \frac{1}{2} + \frac{1}{4} + \cdots = 2.$$

При этом каждая итерация алгоритма на j-й фазе совершает $O\left(\left(\frac{3}{4}\right)^j n\right)$ действий. Математическое ожидание времени работы алгоритма равняется сумме математических

ожиданий времён работы каждой фазы, которая не превосходит

$$\sum_j O\left(\left(\frac{3}{4}\right)^j n\right) \cdot 2 = O\left(n \cdot \sum_j \left(\frac{3}{4}\right)^j\right) = O\left(n \cdot \frac{1}{1-3/4}\right) = O(n).$$

■

В C++ есть встроенная реализация этого алгоритма — `std::nth_element`.

R Алгоритм можно модифицировать так, чтобы он работал за $O(n)$ в худшем случае. Это делается так: поделим массив на $n/5$ групп по 5 элементов, в каждой за $O(1)$ найдём медиану. Теперь рекурсивным запуском алгоритма найдём медиану среди этих медиан, и уже её будем использовать в качестве разделителя. Тогда в половине групп хотя бы 3 элемента окажутся меньше разделителя, а в другой половине хотя бы 3 элемента окажутся больше разделителя. Значит каждая из частей, на которые поделился массив, будет иметь размер хотя бы $3n/10$. Получаем в худшем случае рекуррентное соотношение $T(n) = \Theta(n) + T(n/5) + T(7n/10)$. Можно показать, что в этом случае верно $T(n) = \Theta(n)$.

6.5 Оценка снизу на время работы сортировки сравнениями

Алгоритм сортировки сравнениями может копировать сортируемые объекты и сравнивать их друг с другом, но никак не использует внутреннюю структуру объектов. Все сортировки, изученные нами до этого момента, являются сортировками сравнения. Можно показать, что никакая сортировка сравнениями не может в общем случае работать быстрее, чем за $\Theta(n \log n)$.

Теорема 6.5.1 Любой алгоритм сортировки сравнениями имеет время работы $\Omega(n \log n)$ в худшем случае.

Доказательство. Алгоритм сортировки сравнениями должен уметь корректно сортировать любую перестановку чисел от 1 до n. Пусть алгоритм на каждой перестановке делает не более k сравнений. Заметим, что если зафиксировать результаты всех сравнений в ходе работы алгоритма, то он будет выдавать всегда одну и ту же перестановку данного на вход массива.

R Это не совсем верно для алгоритмов, использующих случайные числа (например, для алгоритма быстрой сортировки), поскольку сама последовательность сравнений может зависеть от того, какие случайные числа выпали. Однако это верно, если зафиксировать последовательность случайных чисел, которую получает алгоритм. Поскольку алгоритм должен корректно работать на любой данной ему последовательности случайных чисел, дальнейшие рассуждения остаются верны.

Поскольку алгоритм делает не более k сравнений, и равенств не бывает (поскольку мы сортируем перестановки), существует не более 2^k различных перестановок данного на вход массива, которые он может выдать. Поскольку алгоритм корректно сортирует произвольную перестановку, получаем $2^k \geqslant n!$. Тогда

$$k \geqslant \log(n!) \geqslant \log\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}\log\frac{n}{2} = \Omega(n \log n).$$

■

Тем не менее, если обладать какой-то дополнительной информацией о свойствах сортируемых объектов, иногда можно воспользоваться этими свойствами, чтобы отсортировать объекты быстрее, чем за $\Theta(n \log n)$.

6.6 Сортировка подсчётом

Если известно, что все числа во входном массиве целые, неотрицательные и меньше некоторого k, то их можно отсортировать за $\Theta(n + k)$ (при этом понадобится $\Theta(k)$ вспомогательной памяти). Для этого посчитаем, сколько раз встретилось каждое число от 1 до k, после чего просто выпишем каждое число в ответ столько раз, сколько он встречалось в исходном массиве.

```
int c[k]
countingSort(a, n):
  for i = 0..(k - 1):
    c[i] = 0
  for i = 0..(n - 1):
    c[a[i]] += 1
  p = 0
  for i = 0..(k - 1):
    for j = 0..(c[i] - 1):
      a[p] = i
      p += 1
```

Ясно, что таким же образом можно сортировать целые числа, лежащие в диапазоне $[L, R)$, за $\Theta(n + (R - L))$.

Если воспользоваться ещё одним вспомогательным массивом, сортировку можно сделать стабильной:

```
int c[k]
countingSort(a, n):
  for i = 0..(k - 1):
    c[i] = 0
  for i = 0..(n - 1):
    c[a[i]] += 1
  for i = 1..(k - 1):
    c[i] += c[i - 1]
  # теперь c[i] - количество элементов массива, не превосходящих i
  int b[n]
  for i = (n - 1)..0:
    c[a[i]] -= 1
    b[c[a[i]]] = a[i]
  for i = 0..(n - 1):
    a[i] = b[i]
```

Стабильная сортировка подсчётом пригодится нам в следующем алгоритме сортировки.

6.7 Поразрядная сортировка (Radix sort)

Пусть дан массив из n чисел, записанных в k-ичной системе счисления и имеющих не более d разрядов каждое. Отсортируем числа сортировкой подсчётом d раз — сначала по младшему разряду, потом по следующему, и так далее, в конце — по старшему разряду. При этом будем пользоваться стабильной версией сортировки подсчётом.

После первого шага числа будут отсортированы по 0-му разряду, после второго — по 1-му разряду, а при равенстве цифр в 1-м разряде — по 0-му. В конце числа будут отсортированы по $(d - 1)$-му разряду, при равенстве цифр в $(d - 1)$-м разряде — по $(d - 2)$-му,..., при равенстве цифр во всех разрядах, кроме 0-го — по 0-му. Значит числа просто окажутся отсортированы в порядке возрастания.

```
radixSort(a, n, d):
  for i = 0..(d - 1):
    countingSort(a, n, i) # стабильная сортировка подсчётом по i-му разряду
```

Каждый шаг алгоритма работает за $\Theta(n + k)$, тогда время работы всего алгоритма — $\Theta(d(n + k))$. Поразрядная сортировка является стабильной.

С помощью поразрядной сортировки можно сортировать любые объекты, которые можно лексикографически упорядочить. Так, можно лексикографически отсортировать n строк длины d каждая, в записи которых используется k различных символов, за $\Theta(d(n + k))$.

Пусть нам даны n неотрицательных целых чисел, меньших m. Если мы переведём их в k-ичную систему счисления, то сможем отсортировать их поразрядной сортировкой за $\Theta((1 + \log_k m)(n + k))$ (используя $\Theta(n(1 + \log_k m) + k)$ дополнительной памяти). При $n = k$ получаем время работы $\Theta(n + n \log_n m)) = \Theta(n + n\frac{\log m}{\log n})$.

7. Двоичная куча

Пусть мы хотим поддерживать множество элементов и быстро выполнять на нём следующие операции: добавлять и удалять элементы, а также находить минимум.

Можно попытаться сделать это с помощью динамического массива или списка. Тогда мы сможем добавлять и удалять элементы за $O(1)$ (чтобы быстро удалить элемент из массива, поменяем его местами с последним элементом, после чего уменьшим длину массива на один). Однако быстро находить минимум не получится: можно пытаться поддерживать указатель на минимальный элемент, но после каждого удаления минимума новый минимум получится найти лишь перебором всех оставшихся элементов за $\Theta(n)$ (где n — число элементов в множестве). Можно было бы поддерживать массив в отсортированном порядке (тогда минимум всегда будет в начале), но тогда не получится быстро добавлять или удалять произвольный элемент.

Двоичная куча (Williams, 1964) — структура данных, которая позволяет добавлять и удалять элементы за $O(\log n)$, а также в любой момент иметь доступ к минимальному элементу в множестве за $O(1)$.

Определение 7.0.1 *Двоичная куча (binary heap)* из n элементов — это массив a с индексами от 1 до n, образующий *двоичное дерево*: для любого $1 < i \leqslant n$ *родитель* i-го элемента — это элемент с номером $\lfloor \frac{i}{2} \rfloor$; соответственно, для любого $1 \leqslant i \leqslant n$ *дети* i-го элемента имеют номера $2i$, $2i+1$ (если элементы с такими номерами существуют).

При этом выполняется *свойство кучи*: значение любого элемента не меньше значения его родителя ($a[i] \geqslant a[\lfloor \frac{i}{2} \rfloor]$).

Заметим, что отсортированный массив всегда является кучей. Но фиксированный набор элементов обычно можно упорядочить и многими другими способами так, что тоже получится куча (например, см. рис. 7.1).

1	2	7	4	5	7	19	8	11	13	9	9

Рис. 7.1: Двоичная куча в виде массива

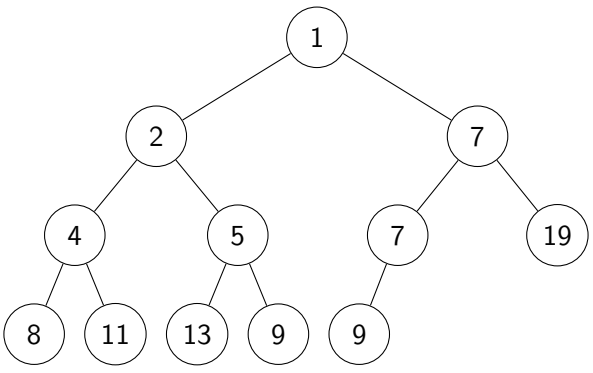


Рис. 7.2: Та же двоичная куча в виде дерева

Будем пользоваться терминами из теории графов. Элементы дерева (кучи) будем называть *вершинами*. *Корень дерева* — единственная вершина без родителя (в нашем случае это $a[1]$). *Потомки* вершины — это она сама, её дети, а также все их потомки; у *листа* (вершины без детей) потомков (кроме него самого) нет. *Предок* вершины v — любая такая вершина u, что v — потомок u. *Поддерево* вершины v состоит из всех её потомков; v — корень своего поддерева.

Иногда мы будем называть кучей не массив с индексами от 1 до n, а просто дерево, в котором свойство кучи выполняется для всех пар родитель-ребёнок. В этом смысле любое поддерево кучи тоже является кучей.

Глубина x — количество вершин на пути от корня дерева до x (все вершины являются потомками корня). *Высота* вершины x — максимальное количество вершин на пути от x до какого-либо её потомка. *Высота дерева* — максимум из высот вершин, то есть высота корня.

Глубина i-й вершины равна $\lfloor \log i \rfloor + 1$, так как i нужно поделить на два нацело $\lfloor \log i \rfloor$ раз, чтобы получить 1.

Высота корня — это максимум из глубин всех его потомков. Максимальная глубина у n-й вершины — $\lfloor \log n \rfloor + 1$. Значит, высота кучи из n элементов равна $\lfloor \log n \rfloor + 1$.

7.1 Базовые операции

Научимся выполнять три простых операции: `getMin()` — поиск минимального элемента в куче, `insert(x)` — добавление нового элемента x в кучу, и `extractMin()` — удаление минимума из кучи. Последние две операции будут пользоваться вспомогательными операциями `siftUp(i)` и `siftDown(i)`.

Поиск минимума

Из определения кучи сразу же следует, что минимальный элемент всегда будет находиться в корне. Тогда просто вернём $a[1]$, время работы — $O(1)$.

```
getMin():
  return a[1]
```

Добавление нового элемента

Пусть свойство кучи "практически" выполняется: для некоторого i известно, что $a[i] = x$ можно увеличить так, что массив a станет кучей. Такую почти кучу можно "починить", не меняя множество лежащих в ней значений, следующим образом: просто будем "поднимать" x вверх, пока x меньше значения в родителе.

```
siftUp(i):
  while i > 1 and a[i] < a[i / 2]:
    swap(a[i], a[i / 2])
    i /= 2
```

Предложение 7.1.1 Пусть известно, что $a[i] = x$ можно увеличить так, что массив a станет кучей. Тогда после выполнения `siftUp(i)` массив a станет кучей.

Доказательство. Докажем утверждение индукцией по i.

База. Если $i = 1$, то a уже является кучей (если корень можно увеличить так, что он станет не больше детей, то он и сейчас не больше детей). `siftUp(1)` не меняет массив a.

Переход. Заметим сначала, что если a уже является кучей, то `siftUp(i)` ничего не сделает, поэтому утверждение предложения верно.

Если a не является кучей, то единственная пара родитель-ребёнок, для которой не выполняется свойство кучи — это $x = a[i] < a[\lfloor \frac{i}{2} \rfloor] = y$ (поскольку для пары, в которой $a[i]$ является родителем, увеличение $a[i]$ может лишь начать нарушать неравенство). Тогда a точно станет кучей, если увеличить значение $a[i]$ до y.

После первой итерации цикла x и y поменяются местами: $y = a[i] > a[\lfloor \frac{i}{2} \rfloor] = x$. Заметим, что такой массив a станет кучей, если значение $a[\lfloor \frac{i}{2} \rfloor]$ увеличить до y (это равносильно увеличению $a[i]$ до y в исходном массиве). Тогда, поскольку последующие действия `siftUp` эквивалентны рекурсивному вызову `siftUp(i / 2)`, по индукции верно, что по завершении операции массив a станет кучей. ■

Количество итераций цикла в `siftUp(i)` не превосходит глубины i-й вершины, то есть время работы оценивается как $O(\log i + 1)$.

Чтобы добавить новый элемент x в кучу размера n, запишем x в $a[n + 1]$ (мы предполагаем, что массив a достаточно большого размера либо динамический). При этом a мог перестать быть кучей, но $a[n + 1]$ можно увеличить так, что a снова станет кучей (например, сделав его больше всех остальных элементов), поэтому достаточно запустить от него `siftUp`. Получаем время работы $O(\log n)$.

```
insert(x):
  sz += 1 # sz - размер кучи
  a[sz] = x
  siftUp(sz)
```

Удаление минимума

Пусть известно, что $a[i] = x$ можно уменьшить так, что массив a станет кучей. Такую почти кучу тоже можно "починить": будем "опускать" x вниз, пока x больше хотя бы одного из значений в детях, при этом будем менять x местами с тем ребёнком, значение в котором меньше.

```
siftDown(i):
  while 2 * i <= sz:
    j = 2 * i
    if 2 * i + 1 <= sz and a[2 * i + 1] < a[2 * i]:
      j = 2 * i + 1
    # j - номер минимального по значению ребёнка i-й вершины
    if a[j] >= a[i]:
      break
    swap(a[j], a[i])
    i = j
```

Предложение 7.1.2 Пусть известно, что $a[i] = x$ можно уменьшить так, что массив a размера n станет кучей. Тогда после выполнения `siftDown(i)` массив a станет кучей.

Доказательство. Докажем утверждение "обратной" индукцией по i.

База. Если $i > 2n$, то a уже является кучей (если лист можно уменьшить так, что он станет не меньше родителя, то он и сейчас не меньше родителя). `siftDown(i)` в таком случае не поменяет массив a.

Переход. Заметим сначала, что если a уже является кучей, то `siftDown(i)` ничего не сделает, поэтому утверждение предложения верно.

Если a не является кучей, то свойство кучи может не выполняться только для тех пар родитель-ребёнок, в которых $a[i]$ является родителем. Пусть j — номер минимального по значению из детей i-й вершины, $y = a[j] < x$. Тогда a точно станет кучей, если уменьшить значение $a[i]$ до y.

После первой итерации цикла x и y поменяются местами: $y = a[i] < a[j] = x$. Заметим, что такой массив a станет кучей, если значение $a[j]$ уменьшить до y (это равносильно уменьшению $a[i]$ до y в исходном массиве). Тогда, поскольку последующие действия `siftDown` эквивалентны рекурсивному вызову `siftDown(j)`, по индукции верно, что по завершении операции массив a станет кучей. ■

Количество итераций цикла в `siftDown(i)` не больше высоты i-й вершины, поэтому время работы оценивается как $O(\log n - \log i + 1) = O(\log \frac{n}{i} + 1)$, или, более грубо, $O(\log n)$.

Чтобы удалить минимум из кучи размера n, поменяем местами $a[1]$ и $a[n]$, и уменьшим размер кучи на один. Теперь в $a[1]$ находится возможно не минимальный элемент, но его можно уменьшить так, что a опять станет кучей (например, сделать его меньше всех остальных элементов). Тогда достаточно запустить `siftDown(1)`. Получаем время работы $O(\log n)$.

```
extractMin():
  swap(a[1], a[sz])
  sz -= 1
  if sz >= 1: # если куча непуста
    siftDown(1)
```

7.2 Построение кучи

Самый простой способ построить кучу из n элементов — начать с пустой кучи, и добавлять элементы по одному. В худшем случае (если все вызовы `siftUp` будут подниматься до корня, то есть если элементы добавляются в убывающем порядке) получим время работы $\Theta(n \log n)$.

Можно построить кучу быстрее: сложим элементы в массив в произвольном порядке, после чего поочерёдно запустим `siftDown` от всех элементов, не являющихся листьями, в порядке убывания их номеров.

```
build(a, n):
  sz = n
  for i = (n / 2)..1:
    siftDown(i)
```

Предложение 7.2.1 После выполнения `build` массив a станет кучей.

Доказательство. Докажем "обратной" индукцией по i, что после запуска `siftDown(i)` поддерево i-й вершины является кучей.

База. $i > \lfloor \frac{n}{2} \rfloor$ (на самом деле для таких i мы не вызываем `siftDown`, потому что он всё равно ничего не сделает). В этом случае поддерево состоит из одной вершины, поэтому утверждение верно.

Переход. По предположению индукции, поддеревья детей i-й вершины являются кучами. Значит, $a[i]$ можно уменьшить так, что и поддерево i-й вершины станет кучей (например, сделав $a[i]$ меньше значений в детях). Тогда после запуска `siftDown(i)` поддерево i-й вершины станет кучей. ■

Время работы `build` тоже складывается из $O(n)$ запусков функций, каждая из который работает за $O(\log n)$. Однако оценка времени работы $O(n \log n)$ в этом случае не является точной.

Теорема 7.2.2 Время работы `build` есть $\Theta(n)$.

Доказательство. Воспользуемся более точной оценкой времени работы `siftDown`: пусть $\lfloor \log n \rfloor = k$, то есть $2^k \leqslant n < 2^{k+1}$. Высоту один имеет $n - 2^k + 1 \leqslant 2^k$ вершин. Высоту $1 < h \leqslant k+1$ имеет $2^{k+2-h} - 2^{k+1-h} = 2^{k+1-h}$ вершин. Тогда время работы `build` есть

$$O\left(\sum_{h=1}^{k+1} 2^{k+1-h} \cdot h\right) = O\left(n \cdot \sum_{h=1}^{k+1} \frac{h}{2^h}\right) = O\left(n \cdot \sum_{j=1}^{k+1} \sum_{h=j}^{k+1} \frac{1}{2^h}\right) =$$

$$= O\left(n \cdot \sum_{j=1}^{k+1} \frac{1}{2^j} \cdot \frac{1}{1-\frac{1}{2}}\right) = O\left(n \cdot \frac{1}{1-\frac{1}{2}} \cdot \frac{1}{1-\frac{1}{2}}\right) = O(n).$$

■

7.3 Heapsort

С помощью двоичной кучи можно отсортировать массив за $O(n \log n)$: построим кучу на массиве, после чего n раз извлечём минимум. Если в корне кучи хранится максимум, а не минимум (свойство кучи в таком случае меняется на противоположное: любой элемент не больше своего родителя), то при извлечении максимума из кучи размера k он как раз оказывается на k-й позиции в массиве, поэтому алгоритму даже не нужна дополнительная память.

```
heapSort(a, n):
  build(a, n) # строим кучу, в корне которой хранится максимум
  for i = 0..(n - 1):
    extractMax()
```

Heapsort не является стабильной сортировкой.

7.4 Очередь с приоритетами

Очередь с приоритетами (priority queue) отличается от обычной очереди тем, что у каждого элемента есть *приоритет*, и извлекается не тот элемент, что был добавлен в очередь раньше всех, а элемент с максимальным приоритетом. Встроенная в C++ реализация очереди с приоритетами — `std::priority_queue`.

Потребность в использовании очереди с приоритетами возникает, например, при планировании задач в многозадачных операционных системах (процессы с большим приоритетом нужно выполнить раньше). Также очередь с приоритетами используется во многих алгоритмах, некоторые из которых нам ещё встретятся.

Очередь с приоритетами легко реализовать с помощью двоичной кучи с максимумом в корне: будем сравнивать элементы по их приоритету.

7.5 Удаление или изменение произвольного элемента

При использовании очереди с приоритетами часто хочется иметь возможность делать дополнительные операции: менять значение приоритета у объекта, лежащего в очереди, или удалять из очереди произвольный объект, не обязательно с максимальным приоритетом. Покажем, что двоичная куча поддерживает такие операции.

Если приоритет объекта не максимален, то сложно быстро понять, где именно в куче он находится. Поэтому когда возникает потребность в таких операциях, обычно для каждого

объекта дополнительно поддерживается ссылка на его позицию в куче (обычно это просто номер этой позиции), а для каждой позиции в куче поддерживается ссылка на объект, который находится в этой позиции. Так, если объекты пронумерованы числами от 1 до n, можно поддерживать массив `pos`, в i-й ячейке которого будет храниться позиция в куче элемента с номером i. При перестройке кучи этот массив несложно пересчитывать.

Итак, пусть мы хотим удалить объект из кучи a размера n, при этом мы знаем, что сейчас он находится на позиции i. Поступим так же, как и при удалении максимума: поменяем i-й элемент местами с последним, и уменьшим размер кучи. После этого a может перестать быть кучей, но i-й элемент можно либо увеличить, либо уменьшить так, что a снова станет кучей (например, можно вернуть старое значение i-го элемента). Значит после вызова `siftUp(i)` или `siftDown(i)` a снова станет кучей. Если мы вызовем обе операции, то одна из них "починит" кучу, а вторая ничего не сделает. Время работы — $O(\log n)$.

```
delete(i):
  swap(a[i], a[sz])
  sz -= 1
  if i <= sz:
    siftUp(i)
    siftDown(i)
```

Точно так же можно менять значение приоритета у произвольного объекта. Пусть объект находится в куче в позиции i, тогда поменяем значение приоритета и снова запустим `siftUp(i)`, `siftDown(i)`. Это работает ровно по тем же причинам, что и в случае удаления. Время работы снова $O(\log n)$.

```
change(i, x):
  a[i] = x
  siftUp(i)
  siftDown(i)
```

R Существуют и более сложные кучи, обладающие различными дополнительными полезными свойствами. *Фибоначчиевы кучи* (Fredman, Tarjan, 1987) известны тем, что позволяют в среднем за $O(1)$ не только находить минимум, но и добавлять новый элемент, а также уменьшать значение элемента. Удаление работает в среднем за $O(\log n)$. Использование фибоначчиевой кучи позволяет улучшить теоретическую оценку времени работы некоторых алгоритмов. Проблема в том, что из-за сложности реализации и большой константы в оценке времени работы на практике фибоначчиевы кучи, как правило, работают не быстрее обычных, поэтому практически не применяются и имеют скорее теоретическую ценность.

8. Хеширование

8.1 Хеш-таблица

Пусть мы хотим поддерживать множество A элементов — *ключей*, то есть уметь добавлять ключ в A, удалять ключ из A, а также искать ключ в A. Мы будем считать, что все ключи берутся из множества $U = \{0, 1, \ldots, |U| - 1\}$.

R В общем случае ключами могут быть какие угодно объекты, которым можно каким-либо образом сопоставить числа (например, строки).

Если $|U|$ не очень велико, можно просто создать массив размера $|U|$ и хранить каждый ключ в ячейке с номером, равным этому ключу. Чтобы понимать, каких ключей в A нет, в соответствующих ячейках будем хранить специальное значение, не равное ни одному из ключей, например, -1. Тогда все операции можно осуществлять за $O(1)$.

Недостаток этого подхода в том, что он он использует $|U|$ памяти, и поэтому работает, только если множество U не слишком велико. Кроме того, если в любой момент времени $|A| \leqslant n$, и n намного меньше $|U|$, то большая часть массива никак не будет использоваться, что непрактично.

Хеш-таблица позволяет решить ту же задачу, используя $\Theta(n)$ памяти, и осуществляя все операции *в среднем* за $O(1)$. Идея состоит в том, чтобы использовать массив размера m, где m намного меньше $|U|$, и ключ x хранить в ячейке с номером $h(x)$, пользуясь *хеш-функцией* $h : U \to M = \{0, 1, \ldots, m - 1\}$.

Может случиться так, что $x \neq y$, но $h(x) = h(y)$. Такую ситуацию мы будем называть *коллизией*. Хеш-функцию стараются выбрать так, чтобы минимизировать число коллизий. Тем не менее, поскольку $m < |U|$, всегда найдётся пара ключей из U с одинаковым значением хеш-функции, то есть какой бы хорошей ни была хеш-функция, коллизии иногда будут происходить.

8.2 Метод цепочек

Самый простой способ разрешения коллизий — в каждой ячейке массива хранить "цепочку" — список всех ключей, попавших в эту ячейку. Этот список можно реализовывать как с помощью связного списка, так и с помощью динамического массива, или даже ещё одной внутренней хеш-таблицы (чуть позже мы увидим пример, когда именно такой способ оказывается полезен).

```
1 vector<int> T[m] # хеш-таблица
2
3 find(x): # возвращает True, если x лежит в хеш-таблице
4   p = h(x)
5   for i = 0..(T[p].size() - 1):
6     if T[p][i] == x:
7       return True
8   return False
```

```
insert(x):
  if not find(x):
    T[h(x)].push_back(x)

delete(x):
  p = h(x)
  for i = 0..(T[p].size() - 1):
    if T[p][i] == x:
      swap(T[p][i], T[p].back())
      T[p].pop_back()
      break
```

Все операции работают за время, пропорциональное длине списка в соответствующей ячейке, то есть за $O(|T[h(x)]| + 1)$. Если ключи распределены по таблице равномерно, то есть в каждом списке оказалось примерно $\frac{n}{m}$ элементов, то операции будут работать в среднем за $O(\frac{n}{m} + 1)$, то есть за $O(1)$ при $m = \Omega(n)$ (мы докажем подобное утверждение более формально, когда будем рассматривать технику универсального хеширования).

Как добиться равномерного распределения? Подобрать "хорошую" хеш-функцию. Иногда это можно сделать, пользуясь информацией о том, с какими ключами придётся работать.

Так, если известно, что ключи выбираются из множества U случайно равновероятно, и $|U|$ делится на m, то подойдёт хеш-функция $h(x) = x \bmod m$: каждый ключ попадёт в каждую ячейку таблицы с равной вероятностью.

На практике часто используют хеш-функцию $h(x) = x \bmod m$, даже если о ключах, с которыми придётся работать, ничего не известно. При этом m обычно стараются выбирать простым.

8.3 Открытая адресация

Поговорим о ещё одном способе разрешения коллизий. В хеш-таблицах с открытой адресацией в каждой ячейке хранится не более одного ключа, а при поиске ключа ячейки проверяются в некотором порядке, пока не найдётся этот ключ или пустая ячейка. Этот порядок, конечно же, будет зависеть от того, какой ключ мы ищем. Поскольку в каждой ячейке хранится не более одного ключа, в таблице размера m не может храниться больше m ключей.

Более формально, теперь мы будем работать с хеш-функцией от двух аргументов $h : U \times \{0, \dots, m-1\} \to \{0, \dots, m-1\}$, и при поиске x перебирать ячейки в порядке $h(x, 0), h(x, 1), \dots, h(x, m-1)$. При этом мы будем требовать, чтобы эта последовательность (будем называть её *последовательностью проб*) была перестановкой чисел от 0 до $m-1$ (чтобы не проверять одну ячейку несколько раз, и чтобы рано или поздно проверить каждую ячейку).

Последовательность проб чаще всего строят одним из следующих трёх способов (здесь $h', h_1, h_2 : U \to \{0, \dots, m-1\}$ — вспомогательные хеш-функции):

- **Линейное пробирование**. $h(x, i) = (h'(x) + c \cdot i) \bmod m$. Обычно используют $c = 1$.
- **Квадратичное пробирование**. $h(x, i) = (h'(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$.
- **Двойное хеширование**. $h(x, i) = (h_1(x) + i \cdot h_2(x)) \bmod m$.

При этом c в первом способе и $h_2(x)$ в третьем должны быть взаимно просты с m (чтобы последовательность пробегала все ячейки). Во втором способе по тем же причинам тоже подойдут не все c_1, c_2.

Линейное пробирование самое простое и имеет наименьшую константу во времени работы, но страдает от кластеризации: блоки из лежащих в таблице подряд ключей со

временем становятся всё больше и больше (потому что всё больше и больше становится вероятность попасть в такой блок).

Двойное хеширование отличается тем, что может дать $\Theta(m^2)$ различных последовательностей (в первых двух способах вся последовательность проб однозначно определяется по $h'(x)$, поэтому всего возможно $\Theta(m)$ различных последовательностей), поэтому менее подвержено кластеризации. Квадратичное пробирование — некий компромисс между линейным пробированием и двойным хешированием.

```
int T[m] # Хеш-таблица

# считаем, что ключи - неотрицательные целые числа,
# в пустых ячейках лежит -1

getIndex(x): # возвращает индекс ячейки, в которой лежит x,
             # или индекс первой встретившейся пустой ячейки
  for i = 0..(m - 1):
    p = h(x, i)
    if T[p] == x or T[p] == -1:
      return p
  error "Hash table is full"

find(x): # возвращает True, если x лежит в хеш-таблице
  return T[getIndex(x)] == x

insert(x):
  p = getIndex(x)
  if T[p] != -1:
    return # x уже есть в таблице
  T[p] = x
```

Удалить ключ из хеш-таблицы теперь не так просто. Если просто пометить ячейку, где лежал ключ, как свободную, то поиск других ключей может перестать работать корректно (потому что при вставке другого ключа мы могли проходить через эту ячейку при поиске свободного места). Можно при удалении специальным образом помечать ячейку и пропускать её при дальнейшей работе с таблицей, но такую ячейку никогда нельзя будет переиспользовать.

```
delete(x):
  p = getIndex(x)
  if T[p] != -1:
    T[p] = -2 # getIndex всегда будет пропускать такую ячейку
```

Пусть в таблице размера m занято (в том числе удалёнными элементами) n ячеек. Предположим, что занятые ячейки расположены по таблице равномерно. Тогда оценим время работы `getIndex`: на каждом шаге вероятность **не** остановиться примерно равна $\alpha = \frac{n}{m} < 1$. Тогда математическое ожидание количества шагов примерно равно

$$1 + \alpha(1 + \alpha(1 + \alpha(1 + \cdots))) \leqslant \frac{1}{1-\alpha}.$$

Мы эвристически оценили время работы операций в среднем как $O(\frac{1}{1-\alpha})$. На практике хеш-таблицы с открытой адресацией действительно работают быстро, пока α достаточно далеко от единицы. Когда α приближается к единице (скажем, когда $\alpha > \frac{2}{3}$), можно построить новую таблицу вдвое большего размера, и перенести все элементы в неё (удалённые элементы при этом, конечно, переносить не надо).

8.4 Ассоциативный массив

В хеш-таблице можно хранить не просто ключи, а пары (ключ, значение). Получится *ассоциативный массив* — массив с произвольными индексами.

В C++ есть встроенные реализации хеш-таблицы и ассоциативного массива — это `std::unordered_set`, `std::unordered_map`.

Отметим, что множество ключей хеш-таблицы (и реализации ассоциативного массива через хеш-таблицу) никак не упорядочено.

В C++ есть и реализации упорядоченного множества и ассоциативного массива с упорядоченными ключами — `std::set`, `std::map`. Эти структуры позволяют совершать больше различных операций (например, находить следующий по значению ключ). Однако они устроены сильно сложнее (мы поговорим об их устройстве позже), а операции работают медленнее: даже поиск ключа требует в худшем случае $\Theta(\log n)$ времени.

8.5 Сортировка Киркпатрика-Рейша

С помощью хеш-таблицы можно соптимизировать поразрядную сортировку, улучшив время работы до $O(n \log \log C)$ на массиве из n целых чисел от 0 до $C - 1$.

Алгоритм (Kirkpatrick, Reisch, 1984) выглядит следующим образом: пусть $C = 2^{2^k}$ (если надо, округлим C вверх до ближайшего числа такого вида, при этом $\log \log C$ увеличится не более, чем на один). Если $C \leqslant n$, просто отсортируем числа подсчётом за $O(n)$, иначе представим каждое число в $\sqrt{C}$-ичной системе счисления; при этом каждое число будет состоять из двух 2^{k-1}-битных цифр ($\sqrt{C} = 2^{2^{k-1}}$). Осталось отсортировать числа по старшей цифре, а при равенстве по младшей рекурсивными вызовами.

Хеш-таблицы (точнее, ассоциативные массивы) нужны, чтобы сгруппировать числа в блоки с равной старшей цифрой.

```
sort(vector<int> a, int k):
  n = a.size()
  if n >= 2 ** (2 ** k):
    countingSort(a)
    return
  vector<int> l # сюда запишем все встретившиеся старшие цифры
  unordered_map<int, vector<int> > t
  # ассоциативный массив: в t[x] будем хранить список всех таких y,
  # что a[i] = x * 2 ** (2 ** (k - 1)) + y для некоторого i
  for i = 0..(n - 1):
    x = a[i] / (2 ** (2 ** (k - 1))) # x - старшие 2 ** (k - 1) бит a[i]
    y = a[i] % (2 ** (2 ** (k - 1))) # y - младшие 2 ** (k - 1) бит a[i]
    if t.find(x) == t.end(): # x встретилось впервые
      l.push_back(x)
    t[x].push_back(y)
  sort(l, k - 1) # отсортировали старшие цифры
  a.clear()
  for (x in l): # перебираем старшие цифры в порядке возрастания
    r = t[x]
    i = maxElementIndex(r) # найдём индекс максимального элемента в r
    rmax = r[i]
    swap(r[i], r.back())
    r.pop_back()
    sort(r, k - 1) # отсортировали все младшие цифры, кроме максимальной
    for (y in r):
      a.push_back(x * 2 ** (2 ** (k - 1)) + y)
    a.push_back(x * 2 ** (2 ** (k - 1)) + rmax)
```

Получаем рекуррентное соотношение

$$T(n) = \Theta(n) + T(|l|) + \sum_{x \in l} T(|t[x]| - 1).$$

Заметим, что

$$|l| + \sum_{x \in l} (|t[x]| - 1) = |l| - |l| + \sum_{x \in l} |t[x]| = n,$$

то есть на каждом уровне рекурсии суммарный размер подзадач равен n. При этом глубина рекурсии не превышает k, значит суммарно на всех уровнях совершено $O(kn) = O(n \log \log C)$ действий. Отметим, что это оценка времени работы в среднем, поскольку мы используем хеш-таблицы.

На практике из-за большой константы во времени работы хеш-таблиц алгоритм как правило оказывается не быстрее обычных алгоритмов сортировки.

R На похожей идее основано дерево ван Эмде Боаса (van Emde Boas, 1975), позволяющее делать те же операции, что и двоичная куча, на целых числах от 0 до $C - 1$ в среднем за $O(\log \log C)$. На практике оно практически не используется по тем же причинам.

8.6 Универсальное хеширование

Можно ли раз и навсегда выбрать для хеш-таблицы фиксированную хеш-функцию $h : U \to M$, которая будет хорошо работать на любых входных данных? Нет: поскольку $|U| > |M|$, всегда найдутся $\lceil |U|/|M| \rceil$ ключей с одинаковым значением хеш-функции (причём отношение $|U|/|M|$, как правило, достаточно велико). Тогда, если в запросах будет встречаться много таких ключей (например, если злоумышленник, знающий хеш-функцию, будет специально посылать такие запросы), эти запросы будут обрабатываться долго.

Эту проблему можно решить следующим образом: до начала работы хеш-таблицы выберем хеш-функцию случайно из некоторого семейства. Если правильно подобрать семейство, запросы по-прежнему будут обрабатываться в среднем быстро, вне зависимости от того, какие ключи подаются на вход.

Определение 8.6.1 Семейство $\mathcal{H}$ хеш-функций, действующих из множества U во множество $M = \{0, 1, \ldots, m - 1\}$, называется *универсальным*, если для любой пары различных ключей $k, l \in U$ количество таких хеш-функций $h \in \mathcal{H}$, что $h(k) = h(l)$, не превосходит $\frac{|\mathcal{H}|}{m}$.

Теорема 8.6.1 Пусть хеш-функция $h \in \mathcal{H}$, $h : U \to \{0, 1, \ldots, m - 1\}$ была случайно выбрана из универсального семейства $\mathcal{H}$, и использована при работе хеш-таблицы размера m. Пусть в хеш-таблицу уже были добавлены n ключей $l_1, \ldots, l_n$, коллизии разрешались методом цепочек. Тогда для любого ключа $k \in U$ математическое ожидание длины списка в ячейке с индексом $h(k)$ не превосходит $1 + \frac{n}{m}$.

Доказательство. Для каждой пары ключей $a, b \in U$ введём величину $\chi(h, a, b)$, равную единице, если $h(a) = h(b)$, и нулю иначе. $\chi(h, a, a) = 1$ для любой h. При этом для $a \neq b$ математическое ожидание $\chi(h, a, b)$ не превосходит $\frac{1}{m}$ по определению универсального семейства:

$$\mathbb{E}(\chi(h, a, b)) = \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \chi(h, a, b) \leqslant \frac{1}{|\mathcal{H}|} \cdot \frac{|\mathcal{H}|}{m} = \frac{1}{m}.$$

Обозначим $L = \{l_1, \ldots, l_n\}$. Математическое ожидание длины списка в ячейке с индексом $h(k)$ равняется

$$\mathbb{E}\left(\sum_{l \in L} \chi(h,k,l)\right) = \sum_{l \in L} \mathbb{E}(\chi(h,k,l)) \leqslant 1 + \sum_{l \in L \setminus \{k\}} \mathbb{E}(\chi(h,k,l)) \leqslant 1 + \frac{n}{m}.$$

■

Следствие 8.6.2 Пусть хеш-таблица размера m использует технику универсального хеширования и метод цепочек для разрешения коллизий. Математическое ожидание суммарного времени работы k операций `insert`, `find`, `delete`, среди которых $O(m)$ операций `insert`, есть $\Theta(k)$.

Доказательство. В любой момент времени в таблице лежат не более $O(m)$ элементов. Время работы любой операции не превосходит времени вычисления хеш-функции (которое мы считаем константным) и времени прохода по списку в ячейке с индексом, равным значению хеш-функции. По предыдущей теореме, математическое ожидание времени выполнения каждой операции не превосходит $O(1 + \frac{O(m)}{m}) = O(1)$. Тогда математическое ожидание суммарного времени работы всех операций не превосходит $O(k)$, то есть равняется $\Theta(k)$. ■

Ещё раз отметим, что теорема и следствие выполняются для **любых** последовательностей запросов; математическое ожидание берётся не по входным данным, а по случайному выбору хеш-функции.

8.7 Построение универсального семейства хеш-функций

Построим универсальное семейство хеш-функций для чисел, влезающих в машинное слово (считаем, что арифметические операции над числами выполняются за $O(1)$).

Теорема 8.7.1 Пусть p — такое простое, что $U \subset \{0, 1, \ldots, p-1\}$; пусть $m < p$. Для любых целых $1 \leqslant a < p$, $0 \leqslant b < p$ определим хеш-функцию

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m.$$

Семейство хеш-функций

$$\mathcal{H}_{p,m} = \{h_{a,b} : \ 1 \leqslant a < p, 0 \leqslant b < p\}$$

является универсальным.

Доказательство. Пусть $k \neq l$ — два различных ключа. Поймём, для каких a, b верно $h_{a,b}(k) = h_{a,b}(l)$.

Заметим сначала, что если $r = (ak + b) \bmod p$, $s = (al + b) \bmod p$, то $r \neq s$, так как $r - s \equiv a(k - l) \not\equiv 0 \pmod p$.

Более того, разным парам (a, b) соответствуют разные пары (r, s), так как по (r, s) можно восстановить (a, b): $a \equiv (r - s)(k - l)^{-1} \pmod p$, $b \equiv (r - ak) \pmod p$.

Различных пар (a, b) с $1 \leqslant a < p$, $0 \leqslant b < p$ всего $(p - 1) \cdot p$. Различных пар (r, s) с $0 \leqslant r \neq s < p$ всего $p^2 - p$, то есть столько же. Значит, мы установили взаимно однозначное соответствие между множествами пар (a, b) и пар (r, s).

Остаётся заметить, что $h_{a,b}(k) = h_{a,b}(l)$ тогда и только тогда, когда $r \equiv s \pmod m$. Для фиксированного $0 \leqslant r < p$ количество $0 \leqslant s < p$ таких, что $s \neq r$, но $r \equiv s \pmod m$, не превосходит $\lceil \frac{p}{m} \rceil - 1 \leqslant \frac{p+m-1}{m} - 1 = \frac{p-1}{m}$.

Тогда количество хеш-функций $h_{a,b} \in \mathcal{H}_{p,m}$, для которых $h_{a,b}(k) = h_{a,b}(l)$ равняется количеству пар (r, s) таких, что $0 \leqslant r \neq s < p$, $r \equiv s \pmod m$, которое не превосходит $p \cdot \frac{p-1}{m} = \frac{|\mathcal{H}_{p,m}|}{m}$. ■

8.8 Совершенное хеширование

Если множество используемых ключей известно заранее (например, в языках программирования множество зарезервированных слов фиксировано), то можно построить хеш-таблицу, операции в которой будут работать за $O(1)$ в **худшем случае**. При этом можно добиться того, чтобы хеш-таблица имела размер $O(n)$, где n — число используемых ключей.

Снова считаем, что все ключи — целые неотрицательные числа, меньшие некоторого простого p. Построим хеш-таблицу с $m = n$ ячейками, при этом хеш-функцию аккуратно выберем из универсального семейства $\mathcal{H}_{p,m}$. В каждой ячейке этой таблицы вместо списка мы заведём хеш-таблицу второго уровня, причём, если в ячейку с индексом j попало n_j ключей, то внутреннюю хеш-таблицу в этой ячейке сделаем размера $m_j = n_j^2$. Хеш-функцию для внутренней таблицы мы аккуратно выберем из универсального семейства $\mathcal{H}_{p,m_j}$.

Оказывается, можно подобрать такие хеш-функции, что во внутренних таблицах не будет коллизий, а суммарный размер всех таблиц будет оцениваться как $O(n)$.

В доказательстве оценок мы будем пользоваться следующим фактом (под $\mathbb{P}(A)$ имеется в виду вероятность того, что произошло событие A):

Теорема 8.8.1 — Неравенство Маркова.
Пусть X — неотрицательная случайная величина с конечным математическим ожиданием. Тогда для любого $a > 0$ верно

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

Доказательство. Поскольку X неотрицательно, $\mathbb{E}X \geqslant 0 \cdot \mathbb{P}(X < a) + a \cdot \mathbb{P}(X \geqslant a)$, откуда следует требуемое неравенство. ■

Теорема 8.8.2 Пусть в хеш-таблице размера $m = n^2$ хранятся n ключей, причём хеш-функция $h(\cdot)$ была случайно выбрана из $\mathcal{H}_{p,m}$. Тогда с вероятностью более чем $\frac{1}{2}$ коллизий нет (все ключи хранятся в разных ячейках).

Доказательство. Для каждой пары различных ключей введём величину $\chi(h, a, b)$, равную единице, если $h(a) = h(b)$, и нулю иначе. Как и в теореме 8.6.1, $\mathbb{E}(\chi(h, a, b)) \leqslant \frac{1}{m} = \frac{1}{n^2}$. Рассмотрим случайную величину X — количество коллизий. Тогда

$$\mathbb{E}X = \mathbb{E}\left(\sum_{a \neq b}(\chi(h, a, b))\right) = \sum_{a \neq b}\mathbb{E}(\chi(h, a, b)) \leqslant \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

Поскольку X принимает только целые неотрицательные значения, по неравенству Маркова

$$\mathbb{P}(X > 0) = \mathbb{P}(X \geqslant 1) \leqslant \mathbb{E}X < \frac{1}{2}.$$

■

Из этой теоремы сразу же следует, что если мы можем позволить себе использовать хеш-таблицу размера n^2, то, сделав несколько попыток, мы подберём такую хеш-функцию, что все ключи попадут в разные ячейки таблицы (вероятность того, что мы не найдём такую хеш-функцию за s попыток, меньше 2^{-s}; в среднем понадобится не более двух попыток). Наша двухуровневая схема нужна, чтобы уменьшить количество используемой памяти до $O(n)$.

Те же рассуждения показывают, что, сделав несколько попыток, мы подберём хеш-функцию без коллизий для каждой внутренней хеш-таблицы. Осталось понять, почему суммарный размер таблиц можно сделать линейным от числа ключей.

Теорема 8.8.3 Пусть в хеш-таблице размера $m = n$ хранятся n ключей, причём хеш-функция $h(\cdot)$ была случайно выбрана из $\mathcal{H}_{p,m}$. Пусть в j-ю ячейку попало n_j ключей, которые были помещены во внутреннюю хеш-таблицу размера $m_j = n_j^2$. Тогда математическое ожидание суммарного размера внутренних хеш-таблиц меньше $2n$.

Доказательство. Заметим сначала, что

$$\mathbb{E}\left(\sum_{j=0}^{m-1} m_j\right) = \mathbb{E}\left(\sum_{j=0}^{m-1} n_j^2\right) = \mathbb{E}\left(\sum_{j=0}^{m-1}\left(n_j + 2\cdot\frac{n_j(n_j-1)}{2}\right)\right) =$$

$$= \mathbb{E}\left(\sum_{j=0}^{m-1} n_j\right) + 2\cdot\mathbb{E}\left(\sum_{j=0}^{m-1}\frac{n_j(n_j-1)}{2}\right) = n + 2\cdot\mathbb{E}\left(\sum_{j=0}^{m-1}\frac{n_j(n_j-1)}{2}\right).$$

Теперь заметим, что $\sum_{j=0}^{m-1}\frac{n_j(n_j-1)}{2}$ — это суммарное количество случившихся коллизий. Из доказательства предыдущей теоремы мы знаем, что математическое ожидание количества коллизий не превосходит $\frac{n(n-1)}{2}\cdot\frac{1}{m} = \frac{n-1}{2}$, так как $m = n$. Получаем

$$\mathbb{E}\left(\sum_{j=0}^{m-1} m_j\right) \leqslant n + 2\cdot\frac{n-1}{2} = 2n - 1 < 2n.$$

■

Следствие 8.8.4 В предположениях теоремы суммарный размер внутренних хеш-таблиц окажется больше или равен $4n$ с вероятностью меньше $\frac{1}{2}$.

Доказательство. Воспользуемся неравенством Маркова:

$$\mathbb{P}\left(\sum_{j=0}^{m-1} m_j \geqslant 4n\right) \leqslant \frac{\mathbb{E}\left(\sum_{j=0}^{m-1} m_j\right)}{4n} < \frac{2n}{4n} = \frac{1}{2}.$$

■

Снова получаем, что за s попыток с вероятностью хотя бы $1 - 2^{-s}$ (и не больше чем за две попытки в среднем) мы подберём такую хеш-функцию для внешней хеш-таблицы, что суммарный размер всех внутренних хеш-таблиц будет оцениваться как $O(n)$ (будет меньше $4n$).

III Теоретико-числовые алгоритмы

9. Арифметика сравнений

Начиная с этой главы, при оценке сложности алгоритмов будем использовать обозначение $M(n)$, имея в виду сложность умножения двух чисел длины n. Можно показать (с помощью метода Ньютона), что деление нацело имеет ту же сложность, что и умножение, поэтому для деления отдельного обозначения вводить не будем.

9.1 Сложение и умножение по модулю N

Сложение и умножение по n-битному модулю N имеет ту же сложность, что и обычные сложение и умножение, то есть $O(n)$ и $O(M(n))$: нужно произвести вычисление без модуля, при этом результат будем иметь не более $2n$ бит, после чего вычислить остаток от деления результата на N. В случае сложения результат меньше $2N$, поэтому достаточно просто (возможно) вычесть N, что требует ещё $O(n)$ операций и не увеличивает оценку времени работы. В случае умножения осуществляем деление с остатком за $O(M(n))$.

9.2 Возведение в степень по модулю N

Воспользуемся той же идеей, что и в рекурсивном алгоритме умножения:

$$a^b = \begin{cases} (a^{\lfloor \frac{b}{2} \rfloor})^2, & \text{если } b \text{ чётно,} \\ a \cdot (a^{\lfloor \frac{b}{2} \rfloor})^2, & \text{иначе.} \end{cases}$$

```
modExp(a, b, N): # a и b - двоичные записи чисел, N - модуль
  if b == 0:
    return 1
  c = modExp(a, b / 2, N) # деление нацело
  if b % 2 == 0:
    return c ** 2 mod N
  else:
    return a * c ** 2 mod N
```

Если n — максимальная длина чисел a, b, N, то происходит $O(n)$ рекурсивных вызовов, на каждом из которых не более двух умножений по модулю N. Получаем оценку сложности $O(n \cdot M(n))$.

9.3 Алгоритм Евклида

Алгоритм Евклида находит *наибольший общий делитель (НОД, greatest common divisor, gcd)* двух чисел a и b.

Предложение 9.3.1 Для любых $a, b \geqslant 0$ выполняется $\gcd(a, b) = \gcd(a \bmod b, b)$.

Доказательство. Ясно, что любой общий делитель a и b является и делителем $a - b$. Наоборот, любой общий делитель $a - b$ и b является делителем a. Тогда $\gcd(a, b) = \gcd(a - b, b)$. Остаётся $\lfloor \frac{a}{b} \rfloor$ раз вычесть b из a. ■

Алгоритм Евклида пользуется вышеописанным правилом, пока не окажется, что $b = 0$. Ясно, что $\gcd(a, 0) = a$ для любого a.

```
gcd(a, b): # a, b >= 0
  if b == 0:
    return a
  return gcd(b, a % b)

gcd(a, b): # нерекурсивная версия
  while b > 0:
    a %= b
    swap(a, b)
  return a
```

Оценим время работы алгоритма:

Лемма 9.3.2 Если $a \geqslant b \geqslant 0$, то $a \bmod b < \frac{a}{2}$.

Доказательство. Если $b \leqslant \frac{a}{2}$, то $a \bmod b < b \leqslant \frac{a}{2}$. Если $b > \frac{a}{2}$, то $a \bmod b = a - b < \frac{a}{2}$. ■

Следствие 9.3.3 Время работы алгоритма Евклида на числах длины не более n есть $O(n \cdot M(n))$.

Доказательство. За каждые две итерации алгоритма a и b уменьшаются хотя бы вдвое, поэтому число итераций не превосходит $2n$. На каждой происходит одно деление с остатком, имеющее сложность $O(M(n))$. ■

Если использовать алгоритм деления в столбик, можно показать оценку получше:

Теорема 9.3.4 Время работы использующего деление в столбик алгоритма Евклида на числах длины не более n есть $O(n^2)$.

Доказательство. Вспомним, что если мы пользуемся алгоритмом деления в столбик, то остаток от деления k-битного числа a на l-битное число b вычисляется за $O(k(k - l + 1))$.

Пусть в процессе выполнения алгоритма Евклида мы работали с числами $z_1 = a, z_2 = b, z_3 = a \bmod b, \ldots, z_{k+1} = z_{k-1} \bmod z_k = 0$, имеющими длины $n_1, n_2, \ldots, n_k, n_{k+1}$. Суммарное время выполнения всех делений —

$$\sum_{i=1}^{k-1} O(n_i(n_i - n_{i+1} + 1)) = O\left(nk + n \cdot \sum_{i=1}^{k-1}(n_i - n_{i+1})\right) = O(nk + n(n_1 - n_k)) = O(n^2).$$

■

9.4 Расширенный алгоритм Евклида

Расширенный алгоритм Евклида одновременно с $d = \gcd(a, b)$ находит такие x и y, что $ax + by = d$. Эти x и y можно использовать как *сертификат*, подтверждающий, что d — действительно НОД a и b:

Лемма 9.4.1 Если a и b делятся на d, и $d = ax + by$ для некоторых x, y, то $d = \gcd(a, b)$.

Доказательство. a и b делятся на d, тогда $d \leqslant \gcd(a, b)$. С другой стороны a и b делятся на $\gcd(a, b)$, тогда и $d = ax + by$ делится на $\gcd(a, b)$, то есть $d \geqslant \gcd(a, b)$. ■

Такие x и y всегда можно найти следующим алгоритмом:

```
extendedEuclid(a, b): # возвращает x,y такие, что ax + by = gcd(a, b)
  if b == 0:
    return 1, 0
  x, y = extendedEuclid(b, a % b)
  return y, x - (a / b) * y # деление нацело
```

Предложение 9.4.2 Вышеописанный алгоритм возвращает такие x, y, что $ax + by = \gcd(a, b)$.

Доказательство. Докажем утверждение индукцией по b.

База. Если $b = 0$, то $a = a \cdot 1 + b \cdot 0$.

Переход. Пусть рекурсивный вызов вернул x, y. По предположению индукции выполняется равенство $\gcd(b, a \bmod b) = b \cdot x + (a \bmod b) \cdot y$.

Тогда $\gcd(a, b) = \gcd(b, a \bmod b) = b \cdot x + (a \bmod b) \cdot y = b \cdot x + (a - \lfloor \frac{a}{b} \rfloor b) \cdot y = a \cdot y + b \cdot (x - \lfloor \frac{a}{b} \rfloor y)$. ■

Ясно, что число итераций этого алгоритма совпадает с числом итераций обычного алгоритма Евклида (так как рекурсивные переходы устроены точно так же). Чтобы оценить время работы расширенного алгоритма Евклида, нужно ещё понять, насколько большими могут быть x и y:

Предложение 9.4.3 Если $a, b > 0$, то для возвращаемых алгоритмом x и y верно, что $|x| \leqslant b$, $|y| \leqslant a$.

Доказательство. Снова воспользуемся индукцией по b.

База. При $b = 0$ алгоритм возвращает $x = 1$, $y = 0$ (случай $b = 0$ не подходит под условие предложения, но нужен нам как база индукции).

Переход. Если $a \bmod b = 0$, то `extendedEuclid(b, a mod b)` вернул $x' = 1$, $y' = 0$. Тогда `extendedEuclid(a, b)` вернёт $x = 0 \leqslant b$, $y = 1 \leqslant a$.

Если же $a \bmod b \neq 0$, то по предположению индукции `extendedEuclid(b, a mod b)` вернул такие x', y', что $|x'| \leqslant a \bmod b$, $|y'| \leqslant b$. Тогда `extendedEuclid(a, b)` вернёт $|x| = |y'| \leqslant b$, $|y| = |x' - \lfloor \frac{a}{b} \rfloor y'| \leqslant |x'| + \lfloor \frac{a}{b} \rfloor |y'| \leqslant a \bmod b + \lfloor \frac{a}{b} \rfloor b = a$. ■

Пусть длина битовой записи a и b не превосходит n. Поскольку a и b только уменьшаются при рекурсивных вызовах, ясно, что все числа, возникающие в процессе промежуточных вычислений, имеют длину не более n. Теперь уже ясно, что оценка времени работы алгоритма Евклида $O(n \cdot M(n))$ остаётся верной и для расширенной версии.

Покажем, что остаётся верной и оценка $O(n^2)$:

> **Следствие 9.4.4** Время работы использующего деление в столбик расширенного алгоритма Евклида на числах длины не более n есть $O(n^2)$.

Доказательство. Для того, чтобы повторить доказательство теоремы 9.3, достаточно показать, что суммарное время работы всех операций, кроме рекурсивного вызова, в `extendedEuclid(a,b)` на числах длины n и m соответственно, есть $O(n(n - m + 1))$.

Частное $\lfloor \frac{a}{b} \rfloor$ можно вычислить одновременно с остатком $a \bmod b$ за $O(n(n - m + 1))$. Также нужно произвести умножение $\lfloor \frac{a}{b} \rfloor$ на y, но, поскольку длина $\lfloor \frac{a}{b} \rfloor$ не превосходит $n - m + 1$, а длина y не превосходит n, это умножение имеет ту же сложность, что и деление выше. Помимо умножения и деления, происходит ещё лишь линейное от n число действий. ■

9.5 Нахождение обратного по модулю N

Число, *обратное к a по модулю N* (об. a^{-1}) — это такое x, что $ax \equiv 1 \pmod N$.

Такое x существует тогда и только тогда, когда $ax + Ny = 1$ для некоторого y. Как мы уже знаем, такое уравнение имеет решение только когда $\gcd(a, N) = 1$, то есть когда a взаимно просто с N.

С помощью расширенного алгоритма Евклида мы можем за $O(n^2)$ проверить, существует ли обратное к a по модулю N, и если существует, то найти его.

Если для числа a существует обратное по модулю N число a^{-1}, то можно осуществлять деление на a по модулю N: это равносильно умножению на a^{-1}.

10. Проверка на простоту и факторизация

10.1 Решето Эратосфена

Решето Эратосфена — алгоритм, находящий все простые числа, не превосходящие некоторой границы n. Алгоритм рассматривает последовательно все числа от 2 до n. Если очередное число ещё не помечено как составное, он помечает его как простое, а все числа, делящиеся на него, помечает как составные.

Поскольку у любого составного числа x есть делитель, не больший корня из x, достаточно перебирать кратные простого числа, начиная с его квадрата.

```
vector<bool> isPrime(n + 1, 1)
isPrime[0] = isPrime[1] = 0
for i = 2..n:
  if isPrime[i]:
    for (j = i * i; j <= n; j += i):
      isPrime[j] = 0
```

Время работы алгоритма есть

$$\sum_{\substack{p\text{-простое,}\\ p\leqslant n}} \frac{n}{p} = n \cdot \left(\sum_{\substack{p\text{-простое,}\\ p\leqslant n}} \frac{1}{p} \right) = O(n \log\log n),$$

поскольку

$$\sum_{\substack{p\text{-простое,}\\ p\leqslant n}} \frac{1}{p} = O(\log\log n)$$

(факт из теории чисел, который мы доказывать не будем).

Решето с линейным временем работы

Существует алгоритм со временем работы $O(n)$ (Gries, Misra, 1978), который устроен похожим образом, но помечает каждое составное число ровно один раз. Более того, для каждого числа, не большего n, он находит его наименьший простой делитель, что позволяет потом быстро факторизовать (разложить на простые множители) любое из чисел.

Обозначим наименьший простой делитель x за $p[x]$. Алгоритм помечает составное число x в тот момент, когда он рассматривает число $\frac{x}{p[x]}$. Заметим, что $p[x] \leqslant p[\frac{x}{p[x]}]$.

Чтобы сделать все такие пометки, достаточно, рассматривая число y, пометить все числа вида yq, где q — простое, меньшее или равное $p[y]$.

```
vector<int> primes # список простых чисел
vector<int> d(n + 1, -1) # d[i] - номер p[i] в списке простых
for i = 2..n:
  if d[i] == -1:
    d[i] = sz(primes)
    primes.push_back(i)
  for (j = 0; j <= d[i] and i * primes[j] <= n; ++j):
    d[i * primes[j]] = j
```

10.2 Перебор делителей

Пусть теперь мы хотим проверить на простоту конкретное число n. Самый простой способ — это перебрать все числа, большие единицы, но меньшие n, и проверить, делится ли n хоть на одно из них. На самом деле достаточно перебирать числа, не большие $\sqrt{n}$, поскольку у составного n точно найдётся нетривиальный делитель не больше $\sqrt{n}$. Тем не менее, если b — длина битовой записи n, такой алгоритм всё ещё будет иметь сложность, экспоненциально зависящую от b: он совершит $\Omega(\sqrt{n}) = \Omega(2^{b/2}) = \Omega((\sqrt{2})^b)$ итераций.

В криптографии часто возникает потребность в больших простых числах. Соответственно, хочется иметь алгоритм, умеющий проверять число на простоту за время, полиномиально зависящее от длины числа.

10.3 Тест Ферма

Вспомним малую теорему Ферма:

Теорема 10.3.1 — Малая теорема Ферма.
Для любого простого p и любого $1 \leqslant a \leqslant p-1$ выполняется $a^{p-1} \equiv 1 \pmod{p}$.

На основе этой теоремы можно проверять n на простоту следующим тестом: возьмём случайное $1 \leqslant a \leqslant n-1$ и проверим, правда ли, что $a^{n-1} \equiv 1 \pmod{n}$. Если сравнение не выполняется (то есть n не прошло тест), то n точно составное.

Проблема в том, что из того, что сравнение выполняется, совсем не следует, что n — простое. Например, $341 = 31 \cdot 11$ — составное, но $2^{340} = 1024^{34} \equiv 1 \pmod{341}$. Можно пытаться бороться с этой проблемой, запуская тест несколько (s) раз с различными a.

```
# возвращает False, если n составное;
# возвращает True, если все проверки пройдены успешно
FermatTest(n):
  for i = 0..(s - 1):
    a = random(1, n - 1) # случайное число от 1 до n - 1
    if modExp(a, n - 1, n) != 1:
      return False # n точно составное
  return True # надеемся, что n простое
```

Время работы теста Ферма на b-битном числе n — $O(sb \cdot M(b))$.

Как нам поможет несколько запусков теста? Заметим во-первых, что если a окажется не взаимно просто с n (то есть $\gcd(a, n) \neq 1$), то $a^{n-1} \not\equiv 1 \pmod{n}$, потому что такое a не обратимо по модулю n. Это замечание, однако, не очень полезно, потому что таких a может быть мало (например, если n — произведение двух больших простых чисел), тогда вероятность случайно наткнуться на них невелика.

Зато можно доказать, что если $a^{n-1} \not\equiv 1 \pmod{n}$ хотя бы для одного a, взаимно простого с n, то сравнение не выполняется и для многих других a тоже:

Напомним, что $\mathbb{Z}_n$ — кольцо вычетов по модулю n; $\mathbb{Z}_n^*$ — мультипликативная группа обратимых элементов кольца вычетов по модулю n.

Предложение 10.3.2 Пусть нашлось a' такое, что $\gcd(a', n) = 1$, $(a')^{n-1} \not\equiv 1 \pmod{n}$. Тогда

$$\left|\left\{a:\ 1 \leqslant a < n,\ a^{n-1} \not\equiv 1 \pmod{n}\right\}\right| \geqslant \frac{n-1}{2}.$$

Доказательство. Рассмотрим множество из всех остальных a:

$$A = \left\{a:\ 1 \leqslant a < n,\ a^{n-1} \equiv 1 \pmod{n}\right\}.$$

Нужно показать, что $|A| \leqslant \frac{n-1}{2}$. Заметим, что A — подгруппа $\mathbb{Z}_n^*$. При этом по условию леммы $a' \in \mathbb{Z}_n^* \setminus A$, поэтому A — собственная подгруппа. Поскольку порядок подгруппы делит порядок группы, $|A| \leqslant \frac{|\mathbb{Z}_n^*|}{2} \leqslant \frac{n-1}{2}$. ■

Таким образом, если существует хоть одно a, взаимно простое с n и такое, что $a^{n-1} \not\equiv 1 \pmod n$, то n не пройдёт тест со случайным a с вероятностью хотя бы $\frac{1}{2}$. Значит, алгоритм, запустивший тест s раз, ошибётся и назовёт такое составное число n простым с вероятностью не больше, чем $\frac{1}{2^s}$. Скажем, при $s = 50$ вероятность ошибки не превосходит 2^{-50}, что уже пренебрежимо мало.

Числа Кармайкла

К сожалению, существуют составные числа n, для которых сравнение $a^{n-1} \equiv 1 \pmod n$ выполняется для всех a, взаимно простых с n. Такие n называются *числами Кармайкла*. Наименьшее число Кармайкла равняется $561 = 3 \cdot 11 \cdot 17$. Тест Ферма поймёт, что число Кармайкла n составное, только если случайно наткнётся на a, имеющее с n общий делитель, вероятность чего мала.

Чисел Кармайкла бесконечно много, но при этом они встречаются достаточно редко. Пусть $C(n)$ — количество чисел Кармайкла, не превосходящих n. Известно, что $C(n) > n^{2/7}$ для достаточно больших n, но, с другой стороны, $\frac{C(n)}{n}$ стремится к нулю с ростом n. Поэтому большие простые числа иногда генерируют следующим образом: берут случайное число заданной длины, и проверяют его тестом Ферма (иногда даже не случайными a, а $a = 2, 3, \dots$). Если длина числа большая, то вероятность того, что оно оказалось числом Кармайкла, невелика, тогда теста Ферма вполне достаточно.

Если же хочется проверить на простоту конкретное число, тестом Ферма не обойтись.

10.4 Тест Миллера-Рабина

Тест Миллера-Рабина (Artjuhov, 1967; Miller, 1976; Rabin, 1980) делает более тонкую проверку, которая работает и в случае чисел Кармайкла.

Определение 10.4.1 Назовём a *нетривиальным корнем из единицы* по модулю n, если $a^2 \equiv 1 \pmod n$, но $a \not\equiv \pm 1 \pmod n$.

Лемма 10.4.1 Нетривиальных корней по простому модулю p не существует.

Доказательство. По модулю 2 есть всего два остатка — 0 и $1 \equiv -1$, ясно, что 0 — не корень из единицы. Дальше считаем $p \geqslant 3$.

Если $a^2 \equiv 1 \pmod p$, то $(a-1)(a+1)$ делится на p. $\gcd(a-1, a+1) \leqslant 2$, а $p \geqslant 3$, тогда либо $a-1$ делится на p, и $a \equiv 1 \pmod p$, либо $a+1$ делится на p, и $a \equiv -1 \pmod p$. ■

Тест Миллера-Рабина помимо проверки условия из теоремы Ферма пытается найти нетривиальный корень из единицы по модулю n. Делается это следующим образом: пусть дано нечётное n (чётные числа проверять на простоту очень просто). Из чётного числа $n-1$ выделяются все степени двойки: $n - 1 = 2^k m$ для $k \geqslant 1$ и нечётного m. После этого вычисляется последовательность $a^m \bmod n, a^{2m} \bmod n, \dots, a^{2^k m} \bmod n = a^{n-1} \bmod n$.

Если $a^{n-1} \not\equiv 1 \pmod n$, то, как и в тесте Ферма, мы делаем вывод, что число составное. Иначе найдём в последовательности первую единицу, и посмотрим на элемент перед ней. Если он есть (то есть последовательность состоит не только из единиц), и он не равен $-1 \bmod n$, то мы нашли нетривиальный корень из единицы по модулю n, то есть n — составное.

```
# возвращает False, если n составное;
# возвращает True, если все проверки пройдены успешно
MillerRabinTest(n):
  m = n - 1, k = 0 # найдём k, m такие, что n - 1 = m * 2 ** k
  while m % 2 == 0:
    m /= 2, k += 1
  for i = 0..(s - 1):
    a = random(1, n - 1) # случайное число от 1 до n - 1
    x = modExp(a, m, n)
    for j = 0..(k - 1):
      y = x * x mod n
      if y == 1 and x != 1 and x != n - 1:
        return False # нашли нетривиальный корень из единицы, т.е. n - составное
      x = y
    if x != 1:
      return False # не выполнилось условие теоремы Ферма, т.е. n - составное
  return True # надеемся, что n простое
```

Время работы s раундов теста Миллера-Рабина на b-битном числе — снова $O(sb \cdot M(b))$.

Оценка вероятности ошибки

Назовём $1 \leqslant a \leqslant n-1$ *свидетелем непростоты* n, если n не проходит тест Миллера-Рабина при использовании a, то есть если $a^{n-1} \not\equiv 1 \pmod n$; либо если для некоторого $0 \leqslant j < k$ $a^{2^j m} \not\equiv \pm 1 \pmod n$, но $a^{2^{j+1} m} \equiv 1 \pmod n$.

Теорема 10.4.2 Пусть n — нечётное составное; $n - 1 = 2^k m$, где $k \geqslant 1$, m — нечётное. Тогда существует хотя бы $\frac{n-1}{2}$ свидетелей непростоты n.

Доказательство. Мы можем считать, что n — число Кармайкла, так для остальных n утверждение теоремы следует из предложения 10.3.2. Рассмотрим два случая — либо $n = p^t$ для некоторого простого p и $t \geqslant 2$, либо $n = n_1 n_2$ для некоторых взаимно простых $n_1 > 1$ и $n_2 > 1$.

Пусть $n = p^t$, тогда рассмотрим $a = 1 + p^{t-1}$. Заметим, что

$$a^n = (1 + p^{t-1})^{p^t} = 1 + p^{t-1} \cdot p^t + \left(p^{t-1}\right)^2 \cdot (\cdots) \equiv 1 \pmod{p^t}.$$

Тогда $a^{n-1} \not\equiv 1 \pmod n$ (так как в этом случае было бы верно $a^n \equiv a \pmod n$), то есть мы нашли a, взаимно простое с n, но не проходящее даже тест Ферма. Это противоречит предположению, что n — число Кармайкла.

Значит, $n = n_1 n_2$ для некоторых взаимно простых нечётных n_1, n_2. Рассмотрим максимальное $0 \leqslant j < k$, для которого найдётся v такое, что $v^{2^j m} \equiv -1 \pmod n$.

Если B — множество чисел от 1 до $n-1$, не являющихся свидетелями непростоты, то

$$B \subset B' = \left\{ b \in \mathbb{Z}_n^* : \; b^{2^j m} \equiv \pm 1 \pmod n \right\}.$$

Заметим, что B' — подгруппа $\mathbb{Z}_n^*$, поэтому если мы покажем, что $\mathbb{Z}_n^* \setminus B'$ непусто, то сразу получим

$$|B| \leqslant |B'| \leqslant \frac{|\mathbb{Z}_n^*|}{2} \leqslant \frac{n-1}{2}.$$

По китайской теореме об остатках, существует такое w, что

$$w \equiv v \pmod{n_1}, \; w \equiv 1 \pmod{n_2}.$$

Заметим, что

$$w^{2^j m} \equiv v^{2^j m} \equiv -1 \pmod{n_1}, \; w^{2^j m} \equiv 1 \pmod{n_2},$$

значит, $w^{2^j m} \not\equiv \pm 1 \pmod{n}$.

С другой стороны,

$$w^{2^{j+1}m} \equiv 1 \pmod{n_1},\ w^{2^{j+1}m} \equiv 1 \pmod{n_2},$$

то есть $w^{2^{j+1}m} \equiv 1 \pmod{n}$. Значит, $w \in \mathbb{Z}_n^* \setminus B'$, что завершает доказательство теоремы. ■

Из теоремы следует, что для любого нечётного составного n (в том числе для чисел Кармайкла), доля a, являющихся свидетелями непростоты, есть хотя бы $\frac{1}{n-1} \cdot \frac{n-1}{2} = \frac{1}{2}$, то есть вероятность ошибки после s раундов теста не превосходит 2^{-s}. На самом деле можно показать, что свидетелей непростоты ещё больше, и вероятность ошибки на каждом шаге не превосходит $\frac{1}{4}$.

R В 2002 году был изобретён детерминированный тест простоты, работающий за полиномиальное от длины числа время. Время работы теста AKS (Agrawal, Kayal, Saxena, 2002) на b-битном числе в первой публикации оценивалось как $O(b^{12+\varepsilon})$, впоследствии оценка была улучшена до $O(b^{6+\varepsilon})$ (ε — сколь угодно малое положительное число, под $O(b^{\varepsilon})$ имеется в виду какая-то степень $\log b$). Однако вероятностные тесты всё равно работают намного быстрее, поэтому на практике этот тест не используется.

R Пусть $\pi(n)$ — количество простых чисел, не превосходящих n. Известно, что $\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1$. Значит, вероятность того, что случайное число из b бит окажется простым, стремится к $\frac{1}{\ln 2^b} \approx \frac{1.44}{b}$ с ростом b. Тогда большие простые числа можно находить, просто много раз генерируя случайное число и проверяя его каким-нибудь тестом на простоту, пока тест не окажется пройден. Если генерировать b-битное число, в среднем понадобится порядка b попыток.

10.5 ρ-алгоритм Полларда

Задачу факторизации числа не умеют решать за время, полиномиальное от длины битовой записи числа. Тем не менее, существуют алгоритмы, работающие эффективнее перебора всех потенциальных делителей до корня из числа.

ρ-алгоритм Полларда (1975) с вероятностью, близкой к единице, находит нетривиальный делитель составного b-битного числа n за $O(n^{\frac{1}{4}} b^2) = O(2^{\frac{b}{4}} b^2)$ шагов. Делается это следующим образом: выбирается специальная функция $f : \mathbb{Z}_n \to \mathbb{Z}_n$, действующая на остатках по модулю n (обычно используют $f(x) = (x^2 + 1) \bmod n$), а также остаток $0 \leqslant x_0 < n$. После этого строится последовательность по правилу $x_i = f(x_{i-1})$.

Пусть p — минимальный нетривиальный делитель n, обозначим $x'_i = x_i \bmod p$. Рано или поздно какое-то значение x'_i встретится второй раз (то есть совпадёт с x'_j, $j < i$). Если предположить, что функция f ведёт себя как "случайная" функция, то можно считать, что каждое x'_i — случайный остаток по модулю p, тогда по парадоксу дней рождения первый повтор с большой вероятностью случится за первые $O(\sqrt{p})$ шагов (ниже мы докажем это утверждение более формально).

Если $x'_i = x'_j$, то $x_i - x_j$ делится на p, тогда $\gcd(x_j - x_i, n) > 1$. Более того, поскольку вычисление $f(x)$ состоит из применения нескольких сложений и умножений, из $x_i \bmod p = x_j \bmod p$ следует, что $f(x_i) \bmod p = f(x_j) \bmod p$, то есть $x'_{i+1} = x'_{j+1}$. Тогда по индукции $\gcd(x_{j+k} - x_{i+k}, n) > 1$ для любого $k \geqslant 0$.

Тогда можно поступить следующим образом: будем вычислять одновременно две последовательности: $x_i = f(x_{i-1})$ и $y_0 = x_0$, $y_i = f(f(y_{i-1}))$, и на каждом шаге вычислять $\gcd(y_i - x_i, n)$, пока он не окажется больше единицы.

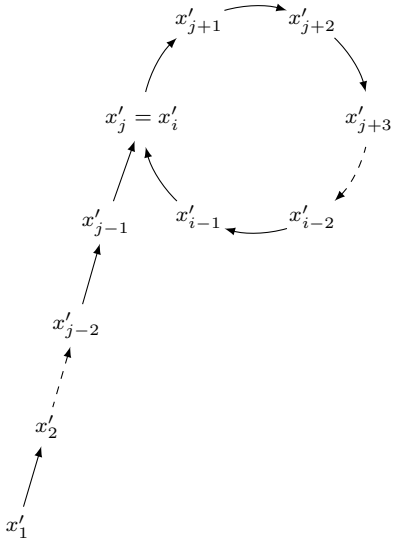


Рис. 10.1: Траектория последовательности x'_i напоминает букву ρ, отсюда и пошло название алгоритма

За первые $O(\sqrt{p})$ шагов значения x_i по модулю p с большой вероятностью начнут повторяться (то есть x_i попадёт на цикл на рис. 10.1). В этот момент y_i тоже уже будет находиться на этом цикле, причём на каждой следующей итерации y_i будет делать два шага вперёд по циклу, а x_i только один. Тогда ещё за $O(\sqrt{p})$ шагов y_i "догонит" x_i, то есть для какого-то i окажется верным $x_i \bmod p = y_i \bmod p$, откуда $\gcd(y_i - x_i, n) > 1$, значит мы найдём делитель n, больший единицы.

Заметим, что совсем не обязательно $\gcd(y_i - x_i, n)$ окажется равным p. y_i и x_i могут в тот же момент времени оказаться на одной позиции в цикле и по какому-нибудь другому модулю q, где q — делитель n. Может даже оказаться, что $\gcd(y_i - x_i, n) = n$, тогда алгоритм не даст нам никакой полезной информации. Но можно показать, что для достаточно больших n вероятность этого мала (мы должны одновременно попасть на одну позицию в цикле по всем нетривиальным делителям n). В случае, если это случается, можно перезапустить алгоритм с другим x_0.

Как уже говорилось, в качестве f обычно используют $f(x) = (x^2 + 1) \bmod n$ или $f(x) = (x^2 - 1) \bmod n$. Эти функции, конечно, не являются случайными, поэтому описанный выше анализ не является строгим доказательством времени работы алгоритма, а лишь эвристическими рассуждениями. Тем не менее, время работы алгоритма на практике хорошо согласуется с этими рассуждениями.

```
rhoAlgorithm(n):
  y = x = random(0, n - 1) # случайное число от 0 до n - 1
  d = 1
  while d == 1:
    x = (x * x + 1) mod n
    y = (y * y + 1) mod n
    y = (y * y + 1) mod n
    d = gcd(x - y, n)
  if d == n:
    return rhoAlgorithm(n) # запустим алгоритм заново
  return d
```

Обычно число n проверяют на простоту, прежде чем запускать на нём алгоритм Полларда. Если нужно целиком факторизовать n, то, получив нетривиальный делитель d,

нужно проверить d и $\frac{n}{d}$ на простоту, и, если надо, запустить алгоритм на них.

Алгоритм с большой вероятностью сделает не более $O(\sqrt{p}) = O(n^{\frac{1}{4}})$ шагов (так как минимальный делитель p не больше корня из n), на каждом шаге происходит вычисление gcd, поэтому время работы на b-битном числе n можно оценить как $O(n^{\frac{1}{4}}b^2)$.

Оценка времени работы

Покажем, что если бы f действительно была случайной функцией, то значения $x'_i = x_i \bmod n$ с большой вероятностью начали бы повторяться за первые $O(\sqrt{p})$ шагов. Поскольку мы будем работать с последовательностью x'_i, для удобства будем считать, что x_0 — случайный остаток по модулю p, f определена на остатках по модулю p.

Теорема 10.5.1 Пусть p — минимальный нетривиальный делитель составного n. Пусть $f : \mathbb{Z}_p \to \mathbb{Z}_p$ и $x_0 \in \mathbb{Z}_p$ выбраны случайно, $\lambda > 0$. Рассмотрим последовательность, определённую правилом $x_i = f(x_{i-1})$, $0 < i \leqslant l = 1 + \lfloor\sqrt{2\lambda p}\rfloor$. Вероятность того, что все x_i попарно различны, не превосходит $e^{-\lambda}$.

Доказательство. Всего различных пар (f, x_0) — $p^p \cdot p = p^{p+1}$. Всего пар (f, x_0), для которых все элементы последовательности окажутся попарно различны — $p \cdot (p-1) \cdots (p-l) \cdot p^{p-l}$, так как x_0 мы выбираем произвольно; $f(x_0) \neq x_0$; $f(x_1) \neq x_0, x_1$;…; $f(x_l) \neq x_0, x_1, \ldots, x_l$; остальные значения f произвольные.

Тогда доля пар (f, x_0), соответствующих последовательностям без повторений, равна

$$\prod_{i=1}^{l} \left(1 - \frac{i}{p}\right).$$

Чтобы показать, что эта доля не превосходит $e^{-\lambda}$, достаточно показать, что

$$\sum_{i=1}^{l} \ln\left(1 - \frac{i}{p}\right) \leqslant -\lambda.$$

Докажем это неравенство:

$$\sum_{i=1}^{l} \ln\left(1 - \frac{i}{p}\right) < -\sum_{i=1}^{l} \frac{i}{p} = -\frac{l(l+1)}{2p} < -\frac{l^2}{2p} < -\frac{2\lambda p}{2p} = -\lambda.$$

Первое неравенство следует из того, что функция $f(x) = ln(1-x) + x$ отрицательна при $0 < x < 1$, так как $f(0) = 0$, $f'(x) = -\frac{1}{1-x} + 1 = -\frac{x}{1-x} < 0$ при $0 < x < 1$. Последнее неравенство следует из того, что $l > \sqrt{2\lambda p}$. ■

R Существуют и более быстрые алгоритмы факторизации. Например, время работы алгоритма GNFS (general number field sieve) эвристически оценивается как $O(e^{(64/9)^{1/3}(\log n)^{1/3}(\log\log n)^{2/3}})$.

11. Криптография

Одна из классических задач криптографии звучит следующим образом. Есть Алиса и Боб, которые хотят поговорить без свидетелей. Также есть Ева (от англ. eavesdropper), которая их подслушивает. Алиса и Боб хотят, чтобы Ева, даже если она подслушает все сообщения, не поняла, о чём был разговор.

Более формально, пусть Алиса хочет послать Бобу секретное сообщение — битовую строку x. Она шифрует её с помощью некоторой функции $E(\cdot)$ (от англ. encoder) и посылает Бобу зашифрованное сообщение $E(x)$. Боб использует функцию $D(\cdot)$ (от англ. decoder), чтобы восстановить исходное сообщение: $D(E(x)) = x$.

Алиса и Боб хотят, чтобы Ева, даже подслушав $E(x)$, не получила никакой информации о x.

Обычно считается, что функции $E(\cdot)$, $D(\cdot)$ вычисляются при помощи некоторых известных всем (в том числе Еве) алгоритмов, но результат вычислений зависит от некоторых параметров — ключей, известных только Алисе и Бобу. Это правило известно как *принцип Керкгоффса* (1883). Такая схема обмена сообщениями оказывается более гибкой: если бы вся сложность расшифровки сообщения состояла в том, что Еве неизвестны алгоритмы, которыми пользуются Алиса и Боб, то, если бы Ева откуда-то узнала эти алгоритмы, Алисе и Бобу пришлось бы с нуля договариваться о новой схеме. Если же алгоритмы известны всем, и Ева откуда-то узнаёт ключи, то достаточно поменять ключи, не меняя всю схему целиком.

11.1 Схемы с закрытым ключом

Схемы с закрытым ключом (также их называют *симметричными схемами*) используют один и тот же ключ для шифрования и расшифровки сообщения. Такие схемы подразумевают, что Алиса и Боб заранее договариваются о некотором секретном ключе, который больше никому не известен.

Одноразовый блокнот

В этом протоколе шифрования Алиса и Боб заранее встречаются и выбирают битовую строку r той же длины, что и будущее сообщение x. Тогда шифрование сообщения состоит в побитовом сложении (`xor`) сообщения с ключом: $E_r(x) = x \oplus r$. Расшифровка устроена ровно так же: $D_r(x) = x \oplus r$, тогда $D_r(E_r(x)) = D_r(x \oplus r) = x \oplus r \oplus r = x$.

Если выбирать каждый бит r случайно равновероятно, то все биты $E_r(x) = x \oplus r$ также будут равны 0 или 1 равновероятно, так что зашифрованное сообщение не даст Еве никакой информации (оно могло получиться из каждого сообщения x с одной и той же вероятностью).

Недостаток схемы состоит в том, что таким r можно воспользоваться лишь один раз: если передать два сообщения x и z, пользуясь одним r, то Ева узнает $x \oplus z = (x \oplus r) \oplus (z \oplus r)$. Отсюда уже можно извлечь полезную информацию: например, если в одном из сообщений есть длинная последовательность нулей, то в $x \oplus z$ на том же месте будет просто кусок другого сообщения. Известна история проекта “Venona”, в рамках которого американцы во

время холодной войны расшифровали многие сообщения советских разведчиков, пользуясь в том числе тем, что те иногда переиспользовали одноразовые блокноты.

Если Алиса и Боб хотят передать много сообщений, им нужно заранее запастись очень большим общим набором случайных бит, что непрактично.

Блочные шифры

Существует целое семейство *блочных шифров*, устроенных по следующей схеме: Алиса и Боб заранее выбирают секретный ключ — случайную строку r, но уже фиксированной длины (например, 128 бит). Сообщение делится на блоки одинаковой длины (например, тоже по 128 бит), после чего каждый из блоков шифруется одной и той же сложно устроенной обратимой функцией $E_r(\cdot)$.

Примерно так устроены, например, следующие схемы шифрования:

- AES (advanced encryption standard) — симметричный алгоритм шифрования, принятый в 2001 году в качестве стандарта шифрования правительством США;
- симметричные шифры "Кузнечик" и "Магма" входят в ГОСТ 34.12-2018, принятый в качестве стандарта шифрования на территории России и СНГ.

Предполагается, что несмотря на то, что один и тот же ключ используется для шифрования многих блоков, воспользоваться этим для расшифровки сообщений, как в случае с одноразовым блокнотом, не удастся, потому что функция $E_r(\cdot)$ устроена достаточно сложно. Одним из важных свойств такой функции является *лавинный эффект* — при изменении одного бита x должны меняться в среднем порядка половины бит $E_r(x)$. Если это не так, то по зашифрованным данным можно пытаться восстановить исходные сообщения с помощью статистического анализа.

11.2 Схемы с открытым ключом

У симметричных схем есть следующий недостаток: для того, чтобы договориться о секретном ключе, Алисе и Бобу всё равно нужно как-то поговорить без свидетелей до того, как схема начнёт ими использоваться. Непонятно, что же им делать, если они никогда раньше не общались, и Ева может читать их переписку с самого начала.

В 1970-е годы было открыто сразу несколько схем, позволяющих Алисе и Бобу начать общаться, не договариваясь ни о каких секретных ключах заранее. Эти схемы стали называть *асимметричными*, или *схемами с открытым ключом*.

В общих чертах схемы с открытым ключом устроены примерно так: для того, чтобы Алиса могла передавать Бобу сообщения, Боб выбирает два ключа — закрытый, который он не показывает никому (даже Алисе); и открытый, который он показывает Алисе или даже публикует в открытом доступе. При этом функция шифрования сообщения $E(\cdot)$ использует открытый ключ, а функция расшифровки $D(\cdot)$ — закрытый.

Пусть Алиса хочет отправить Бобу сообщение x. Алиса, пользуясь открытым ключом, отправляет Бобу зашифрованное сообщение $E(x)$. Боб, пользуясь известным только ему закрытым ключом, расшифровывает сообщение: $D(E(x)) = x$.

Важно, чтобы только Боб мог вычислять значения функции $D(\cdot)$ за разумное время (то есть, не обладая закрытым ключом, функцию $E(\cdot)$ должно быть очень сложно обратить). Для достижения такого эффекта обычно используют задачи, которые не умеют решать быстро (например, задачу факторизации числа).

Заметим, что, поскольку открытый ключ доступен вообще всем, не только Алиса, а вообще кто угодно может посылать сообщения Бобу, не опасаясь, что их сможет прочитать кто-то, кроме Боба.

Для того, чтобы Боб мог посылать сообщения Алисе, ей тоже нужно выбрать открытый и закрытый ключи, и опубликовать открытый ключ.

11.3 RSA

Одной из первых асимметричных схем была RSA (Rivest, Shamir, Adleman, 1977). Она основана на том, что генерировать большие простые числа умеют быстро, а вот быстро раскладывать числа на простые множители не умеют.

Протокол RSA

- Боб выбирает открытый и закрытый ключи:
  - Боб выбирает два случайных больших простых числа $p \neq q$, вычисляет $n = pq$, $\varphi(n) = (p-1)(q-1)$.
  - Боб выбирает e, взаимно простое с $\varphi(n)$. Обычно берётся небольшое e, чтобы шифрование занимало меньше времени. Боб публикует пару (n, e) — *открытый ключ*.
  - Боб вычисляет d, обратное к e по модулю $\varphi(n)$: $de \equiv 1 \pmod{\varphi(n)}$. Пара (n, d) — *закрытый ключ* Боба.
- Алиса хочет передать Бобу сообщение x. Считаем, что $0 \leqslant x < n$ (иначе поделим сообщение на части и пошлём каждую отдельно). Используя открытый ключ Боба, Алиса вычисляет $y = E(x) = x^e \bmod n$ и передаёт его Бобу.
- Боб получает y, и, пользуясь закрытым ключом, вычисляет $D(y) = y^d \bmod n = x^{de} \bmod n$.

Лемма 11.3.1 Пусть $n = pq$, где $p \neq q$ — простые; d, e — такие, что $de \equiv 1 \pmod{\varphi(n)}$. Тогда $x^{de} \equiv x \pmod{n}$ для любого $0 \leqslant x < n$.

Доказательство. Если x делится на p, то $x^{de} \equiv 0 \equiv x \pmod{p}$. Иначе $x^{de} \equiv x \pmod{p}$ по малой теореме Ферма, так как $de - 1$ делится на $\varphi(n)$, то есть и на $p - 1$. Аналогично, $x^{de} \equiv x \pmod{q}$, тогда $x^{de} - x$ делится на $pq = n$. ■

R Заметим, что доля x, не взаимно простых с n, не превосходит $\frac{1}{p} + \frac{1}{q}$, что для больших p, q пренебрежимо мало (если мы наткнулись на такой x, то мы случайно факторизовали n). Для x, взаимно простых с n, утверждение леммы сразу же следует из теоремы Эйлера: $x^{de} = x^{k\varphi(n)+1} \equiv x \pmod{n}$.

Предполагается, что Ева, даже если перехватит сообщение, не сможет по доступным ей n, e, $x^e \bmod n$ восстановить x за разумное время. Можно было бы факторизовать n, после чего найти d так же, как это сделал Боб, но факторизовать числа быстро не умеют. Также неизвестно, можно ли решить задачу восстановления x по n, e, $x^e \bmod n$ быстрее каким-то другим способом.

При этом все шаги, выполняемые Алисой и Бобом, можно делать быстро (пусть n состоит из b бит):

- Быстро генерировать случайные числа заданной длины мы уже умеем;
- d, обратное к e по модулю $\varphi(n)$, можно вычислить с помощью расширенного алгоритма Евклида за $O(b^2)$;
- шифрование и расшифровка сообщения осуществляется возведением в степень по модулю за $O(b \cdot M(b))$.

R Есть ещё множество тонкостей, к которым прибегают, чтобы защититься от известных методов атак на этот алгоритм шифрования (например, d должно быть не слишком мало, а p и q должны быть большими, но не слишком близкими друг к другу; $\varphi(n) = (p-1)(q-1)$ не должно состоять только из маленьких делителей;...). Мы о них подробно говорить не будем. Отметим лишь, что в настоящее время рекомендуется использовать n, состоящее хотя бы из 2048 бит.

На практике RSA и другие схемы с открытым ключом обычно используют не для передачи самого сообщения, а для передачи ключа, который будет использоваться более быстрой схемой с закрытым ключом.

11.4 Цифровая подпись

Пусть Бобу пришло сообщение якобы от Алисы, и он хочет убедиться, что его послала именно Алиса, а не кто-то другой. С помощью схемы с открытым ключом Алиса может "подписать" своё сообщение так, как это не может сделать никто другой (то есть такую подпись можно использовать как доказательство того, что автор сообщения — Алиса). Делается это следующим образом:

Протокол цифровой подписи

- У Алисы есть открытый и закрытый ключи, она хочет послать Бобу сообщение x. Алиса вычисляет цифровую подпись $\sigma = D(x)$, пользуясь закрытым ключом, и посылает Бобу пару (x, σ), состоящую из сообщения и подписи.
- Боб, пользуясь открытым ключом, проверяет равенство $E(\sigma) = x$. Если равенство выполняется, сообщение действительно пришло от Алисы; если нет, то сообщение было повреждено при пересылке, либо было отправлено кем-то другим.

R От схемы требуется, чтобы выполнялось равенство $E(D(x)) = x$ для любых x. Мы же ранее требовали немного другое условие: $D(E(x)) = x$ для любого x. Но в случае, когда $D(\cdot)$, $E(\cdot)$ — биекции (а это, как правило, так) первое следует из второго.

Цифровая подпись подтверждает не только то, что автор сообщения именно тот, за кого он себя выдаёт, но ещё и то, что сообщение не было изменено в процессе пересылки.

Можно одновременно подписывать сообщение x и пересылать его в зашифрованном виде: Алиса, пользуясь своим закрытым ключом, вычисляет подпись, после чего шифрует сообщение и подпись с помощью открытого ключа Боба и отправляет зашифрованные сообщение и подпись Бобу. Боб расшифровывает сообщение и подпись с помощью своего закрытого ключа, после чего проверяет подпись с помощью открытого ключа Алисы.

Криптографические хеш-функции

На практике обычно подпись вычисляют не от самого сообщения x, а от $h(x)$, где $h(\cdot)$ — специальная хеш-функция. Тогда получатель может вычислить ту же хеш-функцию от полученного сообщения, и проверить корректность подписи. Смысл в том, что такая подпись намного короче подписи, вычисленной от самого сообщения.

Что требуется от используемой хеш-функции $h(\cdot)$? Одно из требований звучит так: нужно, чтобы по значению $h(x)$ было сложно подобрать такое y, что $h(x) = h(y)$ (иначе сообщение можно будет подделать).

11.5 Цифровой сертификат

Есть ещё одна проблема: Ева может попытаться выдать себя за Алису, например, подменив лежащий в открытом доступе ключ Алисы на свой.

Эту проблему решают следующим образом: создаётся сертификат, в котором содержится информация об открытом ключе Алисы и о самой Алисе, после чего этот сертификат подписывается некоторой стороной, которой все доверяют, и открытый ключ которой всем известен. Поскольку сертификат подписан тем, кому Боб доверяет, он может быть уверен, что открытый ключ в сертификате действительно принадлежит Алисе.

На практике в качестве этой стороны выступают специальные *центры сертификации (англ. certification authority)*.

11.6 Протокол Диффи-Хеллмана

Поговорим о схемах с открытым ключом, пользующихся отсутствием известных эффективных решений другой задачи — *задачи дискретного логарифмирования*. Схема Диффи-Хеллмана (Diffie, Hellman, 1976) была опубликована на год раньше RSA; она позволяет Алисе и Бобу создать общий секретный ключ, общаясь по открытым каналам (которые может подслушивать Ева).

Протокол Диффи-Хеллмана

- Алиса и Боб договариваются использовать простой модуль p, а также g с достаточном большим $\mathrm{ord}_p(g)$ — порядком по модулю p (минимальным $k > 0$ таким, что $g^k \equiv 1 \pmod p$).
- Алиса выбирает случайное число a, а Боб выбирает случайное число b.
- Алиса отправляет Бобу $x = g^a \bmod p$, а Боб отправляет Алисе $y = g^b \bmod p$.
- Алиса вычисляет $y^a \bmod p = g^{ab} \bmod p$, Боб вычисляет $x^b \bmod p = g^{ab} \bmod p$. $g^{ab} \bmod p$ — их общий секретный ключ.

R Часто утверждают, что в качестве g нужно брать первообразный корень по модулю p. На самом деле это не обязательное условие, достаточно, чтобы $\mathrm{ord}_p(g)$ имел большой простой делитель.

Предполагается, что по p, g, $g^a \bmod p$, $g^b \bmod p$ не получится найти $g^{ab} \bmod p$ за разумное время. Можно было бы решить задачу дискретного логарифмирования, то есть восстановить a по g и $g^a \bmod p$, аналогично восстановить b, после чего вычислить $g^{ab} \bmod p$ уже будет несложно. a и b выбираются достаточно большими, чтобы это нельзя было сделать простым перебором. Неизвестно, можно ли решать задачу дискретного логарифмирования быстро, и можно ли быстро найти $g^{ab} \bmod p$ каким-то другим способом.

p и g обычно выбираются сильно заранее, и считаются известными всем (в том числе Еве). Популярный вариант — использовать такое простое p, что $q = \frac{p-1}{2}$ тоже простое (такие q называются *числами Софи Жермен*). Тогда в качестве g можно взять случайное число из отрезка $[2, p-2]$ — порядок любого взаимно простого с p числа (кроме $p-1$) будет не меньше q.

Все остальные шаги протокола мы умеем осуществлять быстро.

Все те же действия можно совершать не над группой остатков по модулю p, а над произвольной циклической группой с порождающим элементом g. Важно только, чтобы не было известно быстрых алгоритмов, вычисляющих дискретный логарифм в этой группе.

11.7 Схема Эль-Гамаля

Ещё одна асимметричная схема для передачи шифрованных сообщений — схема Эль-Гамаля (Elgamal, 1985), основанная на схеме Диффи-Хеллмана.

Протокол Эль-Гамаля

- Боб выбирает открытый и закрытый ключи:
  - Алиса и Боб заранее договариваются о таких же p и g, как в протоколе Диффи-Хеллмана.
  - Боб выбирает случайное число a — *закрытый ключ* Боба.
  - Боб передаёт Алисе (или публикует) *открытый ключ* — $h = g^a \bmod p$.
- Алиса хочет передать Бобу сообщение x. Считаем, что x — элемент циклической группы, порождённой g (детали того, как сопоставлять произвольному сообщению элемент группы, порождённой g, мы опустим). Алиса выбирает случайное число b, и передаёт Бобу пару $(y, z) = (g^b \bmod p, xh^b \bmod p)$.
- Боб получает пару (y, z) и, пользуясь закрытым ключом, вычисляет $y^{-a}z \bmod p = g^{-ab}g^{ab}x \bmod p = x$.

Как и в протоколе Диффи-Хеллмана, a и b выбирают достаточно большими, чтобы их нельзя было восстановить по $g^a \bmod p$, $g^b \bmod p$ простым перебором. Как и схема Диффи-Хеллмана, схема Эль-Гамаля безопасна в предположении того, что не существует эффективных решений задачи дискретного логарифмирования.

Если Алиса будет использовать одно и то же b много раз, и Еве как-то удастся узнать содержание одного из старых сообщений, то она сможет перехватывать все новые сообщения (так как если Ева узнала x, то она может вычислить $zx^{-1} \bmod p = h^b \bmod p$, с помощью чего она сможет расшифровывать все последующие сообщения, зашифрованные с помощью того же b). Поэтому важно каждый раз использовать новое b, в связи с чем b часто называют *эфемерным ключом*.

IV Базовые алгоритмы на графах

12. Графы и способы их хранения

12.1 Определения

Граф задаётся множеством *вершин* V и множеством *рёбер* E, соединяющих пары вершин (в английском языке граф — *graph*, вершина — *vertex* или *node*, ребро — *edge*).

В *ориентированном (directed graph, digraph)* графе каждое ребро имеет направление, то есть является упорядоченной парой вершин (может быть ребро из вершины a в вершину b, но не быть ребра из b в a). В *неориентированном (undirected)* графе рёбра направлений не имеют, то есть являются неупорядоченными парами (считаем, что ребро, ведущее из a в b, ведёт и из b в a тоже).

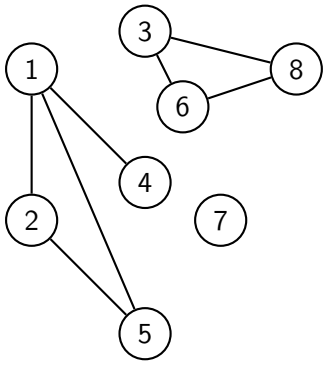


Рис. 12.1: Неориентированный граф

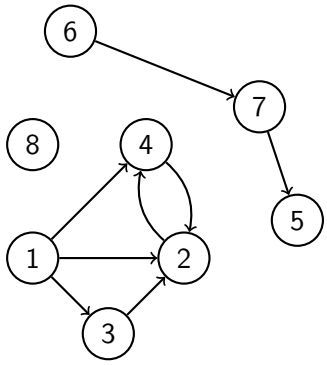


Рис. 12.2: Ориентированный граф

Пример неориентированного графа — карта стран (вершины — страны, рёбра есть между странами, имеющими общую границу). Пример ориентированного графа — интернет (вершины — сайты, рёбра — гиперссылки). Ещё примеры графов — транспортные сети (автомобильные, железнодорожные, авиационные), графы друзей в соцсетях.

В графе может быть несколько рёбер между одной парой вершин (несколько рёбер в одном направлении в ориентированном графе). Такие рёбра называются *кратными (multiple edges)*. Ребро, концы которого совпадают, называют *петлёй (loop)*. Графы без петель и кратных рёбер называют *простыми (simple)*.

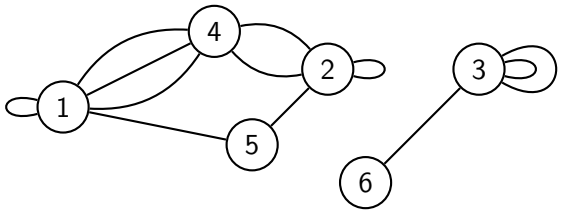


Рис. 12.3: Граф с петлями и кратными рёбрами

Для ребра $e = (a, b)$ вершины a и b называют его *концами (endpoints)* (или *началом и концом* в случае ориентированного графа). Ребро *инцидентно (incident)* каждому из

своих концов. Две вершины неориентированного графа, соединённые ребром, называют *смежными (adjacent)*. Два ребра неориентированного графа, имеющие общий конец, тоже называют *смежными*.

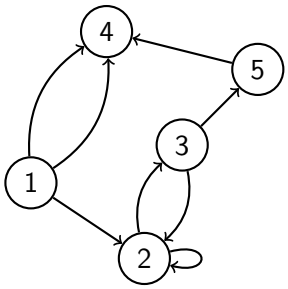
Степень (degree) вершины неориентированного графа $\deg(v)$ — это количество инцидентных ей рёбер (при этом каждая петля считается два раза). В ориентированном графе у вершины v есть *входящая степень (indegree)* $\deg_{in}(v)$ — количество рёбер, входящих в v, и *исходящая степень (outdegree)* $\deg_{out}(v)$ — количество рёбер, исходящих из v.

Подграф (subgraph) $H = (W, F)$ графа $G = (V, E)$ — это граф на подмножестве вершин $W \subseteq V$ и подмножестве рёбер $F \subseteq E$ графа G, так что оба конца любого ребра из F лежат в W. *Индуцированный подграф (induced subgraph)* $H = (W, F)$ содержит все рёбра исходного графа, оба конца которых лежат в W. *Остовный подграф (spanning subgraph)* содержит все вершины исходного графа ($W = V$), но не обязательно все рёбра.

12.2 Способы хранения графа

Чаще всего граф хранят одним из следующих двух способов (считаем, что $|V| = n$, $V = \{v_0, \ldots, v_{n-1}\}$):

- *Матрица смежности (adjacency matrix)* графа — это матрица (двумерный массив) a размера $n \times n$, где $a_{i,j}$ — количество рёбер, идущих из v_i в v_j. Матрица смежности простого графа состоит из нулей и единиц, а её главная диагональ (множество элементов вида $a_{i,i}$) состоит из нулей. Матрица смежности неориентированного графа симметрична ($a_{i,j} = a_{j,i}$).
- *Списки смежности (adjacency lists)* графа — это набор списков, по одному для каждой вершины: в списке смежности вершины v хранится список вершин, в которые ведут рёбра из v (или список самих рёбер). В графе с кратными рёбрами вершина u встречается в списке v столько раз, сколько рёбер из v в u есть в графе. В неориентированном графе каждое ребро входит в списки обоих своих концов; в ориентированном — только в список своего начала. Размер списка смежности вершины v равен $\deg(v)$ ($\deg_{out}(v)$ в ориентированном графе).



$$\begin{pmatrix} 0 & 1 & 0 & 2 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

1:	2, 4, 4
2:	2, 3
3:	2, 5
4:	
5:	4

Рис. 12.4: Граф, его матрица смежности и списки смежности

Матрица смежности позволяет для любой пары вершин за $O(1)$ проверить, соединены ли эти вершины ребром. Однако она занимает $O(n^2) = O(V^2)$ памяти (здесь и далее для краткости в асимптотических оценках вместо $|V|$, $|E|$ будем писать просто V, E). Если хочется проверять наличие ребра за $O(1)$, используя меньше памяти, можно вместо всей матрицы смежности хранить в хеш-таблице пары вершин, связанных ребром.

Списки смежности можно реализовывать как динамическими массивами, так и связными списками. В любом случае, они занимают суммарно $O(V+E)$ памяти, и позволяют для любой вершины быстро просмотреть всех её соседей. Проверка наличия ребра между парой вершин u, v при использовании только списка смежности займёт $O(\min(\deg(u), \deg(v)))$ ($\deg_{out}$ в ориентированном графе).

13. Поиск в глубину

Большая часть рассуждений применима как к ориентированным, так и к неориентированным графам (если не сказано обратное); допускаются кратные рёбра и петли (если не сказано обратное).

13.1 Обход вершин, достижимых из данной

Поиск в глубину (depth-first search, DFS) по вершине $w \in V$ находит множество вершин, достижимых из неё, то есть таких, в которые можно попасть, сделав несколько переходов по рёбрам, начиная из вершины w.

Изначально алгоритм запускается от вершины w. Он перебирает исходящие из текущей вершины рёбра и смотрит, куда они ведут. Каждый раз, когда алгоритм встречает ещё не посещённую вершину, он запускается от неё рекурсивно, а после возврата из рекурсии продолжает перебирать рёбра, исходящие из текущей вершины.

```
dfs(v):
  visited[v] = True
  for u in es[v]: # es[v] - список смежности вершины v
    if not visited[u]:
      dfs(u)

fill(visited, False)
dfs(w)
# visited[u] = True, если u достижима из w
```

Почему алгоритм посетил все достижимые из w вершины? Пусть u достижима из w, тогда существует *путь (path)* из w в u — последовательность вершин $w = v_1, v_2, \ldots, v_k = u$, в которой для $1 \leqslant i < k$ есть ребро из v_i в v_{i+1} (путём, в зависимости от контекста, мы будем называть как такую последовательность вершин, так и последовательность соединяющих их рёбер). Пусть v_i — первая вершина на пути, которую не посетил алгоритм. Тогда он посетил v_{i-1} и проверил ребро, ведущее из v_{i-1} в v_i, что противоречит тому, что он не посетил v_i.

Время работы алгоритма (при использовании списков смежности) — $O(V+E)$, так как каждая вершина посещена не более одного раза, время обработки каждой посещённой вершины пропорционально её исходящей степени, сумма исходящих степеней не превосходит числа рёбер в графе (удвоенного числа рёбер в неориентированном графе).

13.2 Компоненты связности

Неориентированный граф называется *связным (connected)*, если в нём существует путь между любой парой вершин.

На произвольном графе можно ввести отношение достижимости: $a \sim b$, если b достижимо из a. Это отношение рефлексивно ($a \sim a$) и транзитивно (если $a \sim b$ и $b \sim c$, то $a \sim c$). В неориентированных графах это отношение ещё и симметрично (если $a \sim b$, то $b \sim a$), поэтому оно является отношением эквивалентности: все вершины однозначно

разбиваются на классы эквивалентности — множества, в каждом из которых любые две вершины достижимы друг из друга; при этом любые две вершины из разных классов эквивалентности не достижимы друг из друга (класс эквивалентности вершины — это просто множество вершин, достижимых из неё; из рефлексивности, транзитивности и симметричности следует, что для разных вершин такие множества либо не пересекаются, либо совпадают).

Индуцированные подграфы на каждом из этих классов эквивалентности называют *компонентами связности (connected components)* (иногда компонентами связности будем называть сами классы эквивалентности, то есть множества вершин этих подграфов). Каждая компонента связности является связным подграфом, при этом рёбер между вершинами из разных компонент связности в графе нет. Связный граф состоит из одной компоненты связности, содержащей все вершины.

В ориентированных графах отношение $\sim$ не симметрично, поэтому понятие "компонента связности" для них не определено. Есть понятие *сильной связности*, которое мы обсудим чуть позже.

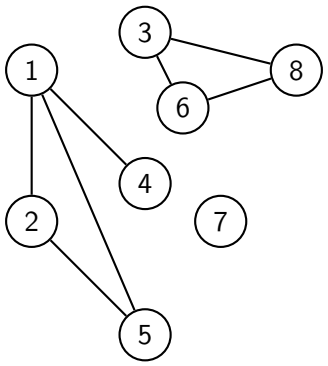


Рис. 13.1: Компоненты связности этого графа — $\{1, 2, 4, 5\}$, $\{3, 6, 8\}$ и $\{7\}$

Алгоритм поиска в глубину по вершине находит все достижимые из неё вершины, то есть её компоненту связности. Тогда для того, чтобы найти все компоненты связности, запустим поиск в глубину из каждой вершины по очереди. Для того, чтобы не обрабатывать одну компоненту связности несколько раз, просто не будем обнулять пометки того, была ли вершина посещена, между запусками алгоритма от разных вершин. Если очередная рассматриваемая вершина уже посещена, то она входит в одну из уже найденных компонент связности. Значит, запускать алгоритм от неё не надо.

```
vector<vector<int> > components

dfs(v):
  visited[v] = True
  components.back().push_back(v)
  for u in es[v]:
    if not visited[u]:
      dfs(u)

fill(visited, False)
for v = 0..(n - 1): # вершины пронумерованы от 0 до n - 1
  if not visited[v]:
    components.push_back(vector<int>())
    dfs(v)
```

Время работы алгоритма — по-прежнему $O(V + E)$, так как каждую вершину алгоритм теперь посещает ровно один раз.

13.3 Дерево(лес) поиска в глубину неориентированного графа

Пусть $G = (V, E)$ — связный граф, из вершины $v \in V$ которого был запущен поиск в глубину. Рассмотрим остовный подграф $H = (V, F)$, где F — множество рёбер, при рассмотрении которых алгоритм делал рекурсивный запуск (рёбер, которые вели в ещё не посещённые вершины). Заметим, что H — связный граф, так как поиск в глубину на связном графе G посетил все вершины.

Цикл — это путь с совпадающими концами и ненулевым числом рёбер, все рёбра в котором различны. Граф H не содержит циклов: ориентируем рёбра H в сторону вершины, которая была непосещённой в момент прохода по ребру алгоритмом; в каждую вершину входит не более одного ребра H. Но если бы нашёлся цикл, то в ту вершину цикла, которая была посещена алгоритмом позже всех, вело бы сразу два ребра.

Итак, H — связный граф без циклов, то есть *дерево*. Если ориентировать рёбра этого дерева, как сказано выше, получится *корневое дерево* (корень — вершина v). Мы снова будем пользоваться терминами, которыми пользовались при изучении двоичных куч: ребёнок, родитель, потомок, предок, лист и так далее.

H называют *деревом поиска в глубину* графа G. Конечно, дерево поиска в глубину не единственно: оно зависит от начальной вершины и порядка рёбер в списках смежности.

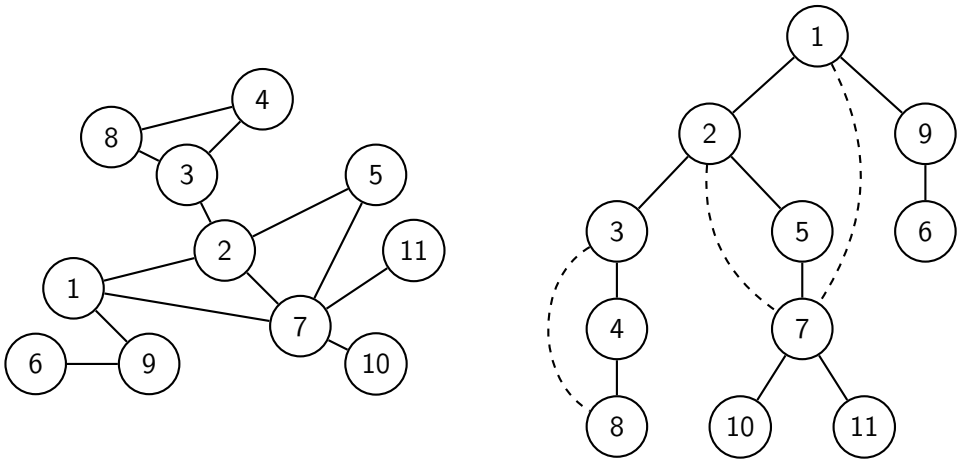


Рис. 13.2: Связный граф и его дерево поиска в глубину (поиск запущен из вершины 1, в списках смежности вершины упорядочены по возрастанию номеров)

Ребро графа G, не являющееся ребром дерева поиска в глубину (то есть любое ребро из $E \setminus F$) будем называть *обратным (back edge)*. Рассмотрим любое обратное ребро $e = (v, u)$. Пусть вершину v алгоритм посетил раньше вершины u. Тогда вершина u была посещена после вершины v, но до того, как алгоритм рассмотрел ребро e, то есть u была посещена в одном из рекурсивных вызовов, произошедших во время обработки v. Значит, в дереве поиска в глубину u является потомком v.

Итак, любое обратное ребро соединяет пару "предок-потомок". Значит, в графе нет *перекрёстных* рёбер — рёбер, соединяющих пары вершин, ни одна из которых не является потомком другой в дереве поиска в глубину.

Лес поиска в глубину

При поиске компонент связности в несвязном графе получается своё дерево поиска в глубину в каждой компоненте связности. Их совокупность — *лес поиска в глубину*. Все рёбра графа снова делятся на рёбра деревьев и обратные рёбра; рёбер между разными деревьями нет.

13.4 Поиск цикла в неориентированном графе

Как проверить, есть ли в неориентированном графе цикл? Поскольку деревья поиска в глубину не имеют циклов, любой цикл будет проходить хотя бы через одно обратное ребро. С другой стороны, любое обратное ребро $e = (v, u)$ соответствует циклу в графе: если v — предок u, то поднимемся по дереву из u в v, и вернёмся в u по ребру e. Значит, в графе есть цикл тогда и только тогда, когда при поиске в глубину нашлось хотя бы одно обратное ребро.

Как отличить обратное ребро от ребра дерева? Если граф простой, то обратное ребро — любое ребро, которое ведёт в уже посещённую вершину, не являющуюся родителем текущей в дереве поиска в глубину. То есть достаточно для каждой вершины запомнить её родителя (или просто передавать его отдельным параметром при рекурсивном запуске).

Если в графе имеются кратные рёбра, то надо хранить (передавать в рекурсивный запуск) не просто родителя вершины, а ещё и указатель на ребро дерева (или просто его номер, если рёбра пронумерованы), по которому мы пришли в текущую вершину из родителя. Так мы сможем отличить ребро дерева от других рёбер между той же парой вершин.

Если цикл нужно явно найти и вывести, то, обнаружив обратное ребро $e = (u, v)$, будем переходить, начиная из текущей вершины u, по ссылкам на родителя, пока не встретим v, и выписывать вершины на пути.

13.5 Времена входа и выхода

В процессе работы поиска в глубину для каждой вершины v запомним *время входа* (время начала обработки вершины) $t_{in}[v]$ и *время выхода* (время конца обработки вершины) $t_{out}[v]$. Для этого заведём специальный счётчик, который будем увеличивать на единицу каждый раз, когда происходит одно из этих событий; текущее значение этого счётчика и будет текущим временем.

```
T = 0 # счётчик времени

dfs(v):
  tin[v] = T, T += 1
  visited[v] = True
  for u in es[v]:
    if not visited[u]:
      dfs(u)
  tout[v] = T, T += 1
```

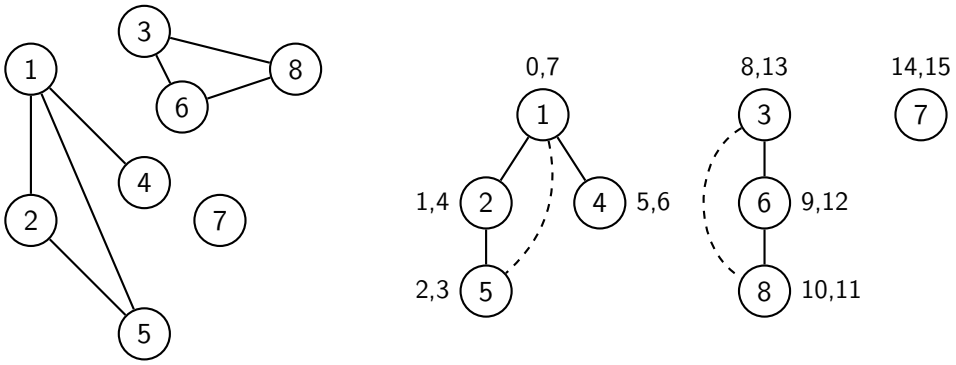


Рис. 13.3: Граф и его лес поиска в глубину с отмеченными временами входа и выхода

Заметим, что если вершина u начала обрабатываться в одном из рекурсивных вызовов, произошедших во время обработки другой вершины v, то u закончит обрабатываться

раньше, чем v. Значит, для любых вершин $v, u \in V$ либо отрезки $[t_{in}(v), t_{out}(v)]$ и $[t_{in}(u), t_{out}(u)]$ не пересекаются, либо один из них содержится в другом. Более того, отрезок u содержится в отрезке v тогда и только тогда, когда u является потомком v в дереве поиска в глубину.

Времена входа и выхода имеют множество различных применений. Простой пример: пусть G — корневое дерево; запустим поиск в глубину из корня G и вычислим времена входа и выхода каждой вершины. Получившееся дерево поиска в глубину совпадает с графом G. Тогда мы можем за $O(1)$ проверить для любой пары вершин, является ли одна из них предком другой в G: достаточно проверить, вложен ли отрезок времени обработки одной из вершин в отрезок другой вершины.

13.6 Дерево(лес) поиска в глубину ориентированного графа, типы рёбер

Дерево(лес) поиска в глубину определяются для ориентированных графов так же, как и для неориентированных: будем перебирать все вершины и запускать поиск в глубину из ещё не посещённых, не обнуляя пометки о посещённости в процессе.

Времена входа и выхода для ориентированных графов вычисляются так же, как и для неориентированных, при этом по прежнему либо отрезки времён обработки вершин не пересекаются, либо один из них вложен в другой.

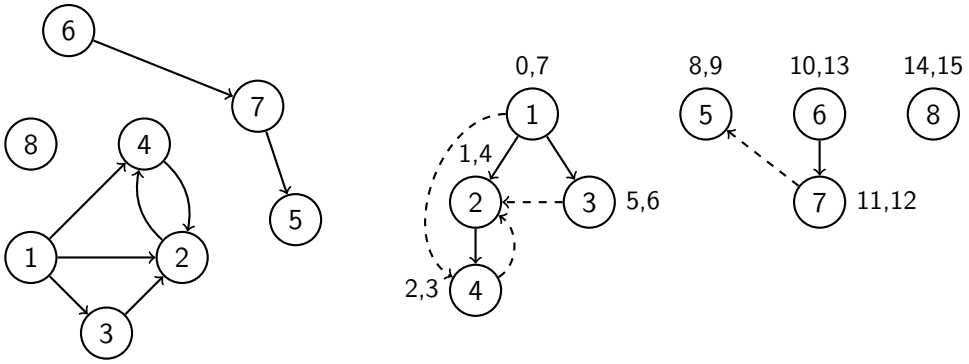


Рис. 13.4: Граф и его лес поиска в глубину с отмеченными временами входа и выхода

В ориентированных графах рёбра, не вошедшие в лес поиска в глубину, делятся уже на три типа:

- *прямые (forward)* — ведущие от вершины к её потомку (ребро $1 \to 4$ на рис. 13.4);
- *обратные (back)* — ведущие от вершины к её предку (ребро $4 \to 2$ на рис. 13.4);
- *перекрёстные (cross)* — ведущие от вершины к другой вершине, не являющейся ни предком, ни потомком первой (рёбра $3 \to 2$ и $7 \to 5$ на рис. 13.4).

Понять тип ребра можно по временам входа и выхода его концов: ребро $v \to u$ является

- прямым (либо ребром леса), если $t_{in}(v) < t_{in}(u)$, $t_{out}(u) < t_{out}(v)$;
- обратным, если $t_{in}(u) < t_{in}(v)$, $t_{out}(v) < t_{out}(u)$;
- перекрёстным, если $t_{out}(u) < t_{in}(v)$.

Заметим, что ситуация $t_{out}(v) < t_{in}(u)$ невозможна: в момент рассмотрения такого ребра вершина u была бы непосещённой, тогда ребро вошло бы в дерево поиска в глубину, и было бы верно $t_{in}(u) < t_{out}(u) < t_{out}(v)$.

13.7 Поиск цикла в ориентированном графе

Как и в случае неориентированного графа, в ориентированном графе есть цикл тогда и только тогда, когда при поиске в глубину нашлось хотя бы одно обратное ребро: по

любому обратному ребру цикл строится точно так же, как и в неориентированном случае; с другой стороны, пусть в графе есть цикл $v_1 \to v_2 \to \cdots \to v_k \to v_1$, и вершину v_i алгоритм обработал первой среди вершин цикла. Остальные вершины цикла достижимы из v_i, поэтому все они будут потомками v_i в дереве поиска в глубину. Но тогда ребро $v_{i-1} \to v_i$ ($v_k \to v_1$ при $i = 1$) является обратным.

Обратное ребро можно обнаружить при помощи времён входа и выхода. Альтернативный способ — для каждой вершины поддерживать пометку одного из трёх типов: "непосещённая", "в обработке", "обработана". Тогда обратные рёбра — ровно те, что ведут в вершину с пометкой "в обработке".

13.8 Топологическая сортировка

Ориентированный ациклический граф (directed acyclic graph, dag) — это ориентированный граф без циклов. С помощью ориентированных ациклических графов удобно представлять отношения зависимости: иерархические отношения ("руководитель-подчинённый"), причинно-следственные связи.

Топологическая сортировка (topological sort) ориентированного графа — это такой порядок на вершинах, что любое ребро графа ведёт из меньшей вершины в большую. Ясно, что граф, в котором есть цикл, топологически отсортировать невозможно.

Возникает вопрос: какие ациклические графы можно топологически отсортировать? Оказывается, что все: достаточно расположить вершины в порядке убывания времён выхода. Действительно, если нашлось такое ребро $v \to u$, что $t_{out}(v) < t_{out}(u)$, то это ребро обратное, значит, граф не ациклический.

Можно не вычислять времена входа и выхода напрямую, а просто добавлять вершину в конец списка в конце её обработки. Тогда вершины в списке окажутся в порядке возрастания времён выхода. Остаётся развернуть список в конце, чтобы получить порядок топологической сортировки.

```
vector<int> topOrder # сюда запишем вершины в порядке топологической сортировки

dfs(v):
  visited[v] = True
  for u in es[v]:
    if not visited[u]:
      dfs(u)
  topOrder.push_back(v)

fill(visited, False)
for v = 0..(n - 1): # вершины пронумерованы от 0 до n - 1
  if not visited[v]:
    dfs(v)
reverse(topOrder.begin(), topOrder.end())
```

Исток (source) — это вершина ориентированного графа, в которую не входит рёбер. *Сток (sink)* — вершина, из которой, наоборот, не выходит рёбер. Заметим, что первая вершина в порядке топологической сортировки всегда является истоком, а последняя — стоком. В частности, в любом ациклическом ориентированном графе всегда есть хотя бы по одному истоку и стоку (это утверждение несложно доказать и без использования топологической сортировки).

13.9 Компоненты сильной связности

Рассмотрим на произвольном ориентированном графе следующее отношение: $a \sim b$ (a *сильно связно (strongly connected)* с b), если одновременно a достижимо из b, и b

достижимо из a. Такое отношение рефлексивно, транзитивно и симметрично, поэтому оно однозначно разбивает вершины графа на классы эквивалентности, которые называют *компонентами сильной связности (strongly connected components)*. Граф, состоящий из одной такой компоненты, называют *сильно связным (strongly connected graph)*.

Пусть $V_1, \ldots, V_k$ — компоненты сильной связности графа G. Построим вспомогательный граф, вершины которого — $V_1, \ldots, V_k$, а ребро $V_i \to V_j$ проведено, только если в G было ребро $v_i \to v_j$ для некоторых $v_i \in V_i$, $v_j \in V_j$ (при этом не будем проводить петли и кратные рёбра). Получившийся граф называют *метаграфом (meta-graph) или конденсацией* графа G. Метаграф не содержит циклов, так как все вершины компонент сильной связности, вошедших в цикл, были бы одновременно достижимы друг из друга, то есть должны были бы лежать в одной компоненте.

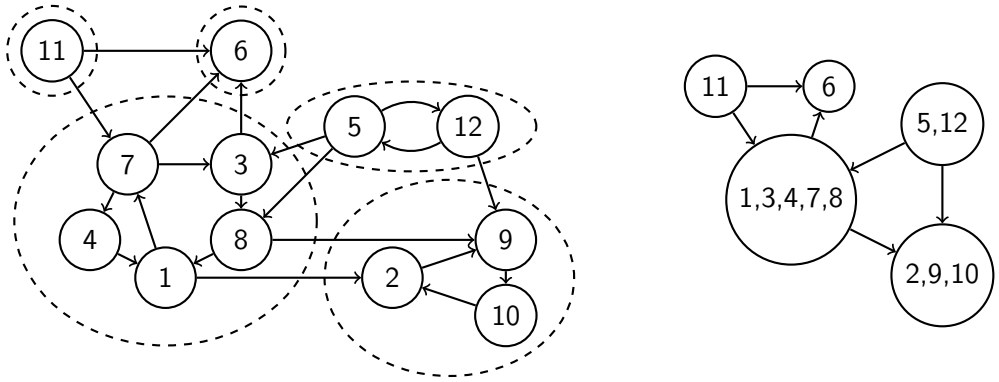


Рис. 13.5: Ориентированный граф и его метаграф

Наша цель — научиться находить компоненты сильной связности произвольного ориентированного графа G за $O(V + E)$. Первое, что мы сделаем — запустим алгоритм топологической сортировки на графе G. Он вернёт список вершин в порядке убывания времён выхода; обозначим этот список за $v_1, \ldots, v_n$. Этот список в общем случае не является порядком топологической сортировки (так как в G могут быть циклы), но всё равно обладает полезными свойствами.

Предложение 13.9.1 Пусть A, B — компоненты сильной связности графа G, в метаграфе есть ребро $A \to B$. Тогда максимальное время выхода среди вершин в A больше, чем максимальное время выхода среди вершин в B.

Доказательство. Пусть $v \in A \cup B$ — первая вершина из $A \cup B$, посещённая поиском в глубину. Если $v \in A$, то во время обработки вершины v будут посещены все вершины обеих компонент, то есть $t_{out}(v)$ будет строго больше времён выхода остальных вершин из $A \cup B$.

Если $v \in B$, то к концу обработки v будут посещены все вершины B, но не будет посещено ни одной вершины из A. Значит, время выхода любой вершины из A окажется строго больше времени выхода любой вершины из B. ■

Отсюда следует, что метаграф можно топологически упорядочить, если расположить компоненты сильной связности в порядке убывания максимального значения времён выхода их вершин. В частности, v_1 принадлежит первой в топологическом порядке компоненте, то есть компоненте-истоку. Обозначим эту компоненту за A_1.

Как по одной вершине из компоненты-истока найти все вершины, лежащие в этой компоненте? Рассмотрим *транспонированный граф (transpose graph)* G^T, полученный из G изменением направления всех рёбер. Заметим, что компоненты сильной связности

G^T совпадают (как множества вершин) с компонентами G, а метаграф G^T является транспонированным графом метаграфа G. В частности, компонента-исток в метаграфе G является компонентой-стоком в метаграфе G^T.

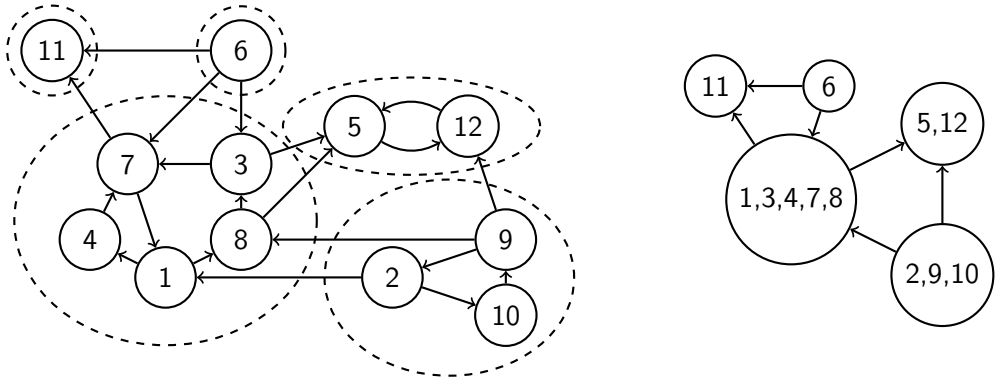


Рис. 13.6: Транспонированный граф и его метаграф

Множество вершин, достижимых из любой вершины компоненты-стока, совпадает с этой компонентой. Значит, для того, чтобы найти A_1, можно запустить поиск в глубину в графе G^T из v_1.

Как найти все остальные компоненты? Удалим из графа G компоненту A_1 (обозначим полученный граф за G_1) и повторим те же действия: найдём v_i — вершину с максимальным временем выхода из ещё не удалённых; v_i лежит в компоненте-истоке A_2 графа G_1, которую можно найти, запустив поиск в глубину из v_i в графе G_1^T. После этого удалим компоненту A_2 и тем же способом найдём следующую компоненту, и так далее, пока вершины не закончатся.

Удалять вершины и перестраивать списки смежности не нужно, можно просто считать посещённые вершины удалёнными.

```
# уже найден topOrder - список вершин в порядке убывания времени выхода
vector<vector<int> > components

dfsT(v):
  visited[v] = True
  components.back().push_back(v)
  for u in esT[v]: # список смежности транспонированного графа
    if not visited[u]:
      dfsT(u)

fill(visited, False)
for v in topOrder: # перебираем вершины в порядке убывания времени выхода
  if not visited[v]:
    # v - вершина с максимальным временем выхода из ещё не посещённых
    components.push_back(vector<int>())
    dfsT(v)
```

Алгоритм топологической сортировки работает за $O(V+E)$, суммарное время поисков в глубину по транспонированному графу также есть $O(V+E)$, так как каждая вершина посещается ровно один раз.

13.10 Поиск мостов и компонент рёберной двусвязности

Мост (bridge) — ребро в неориентированном графе, удаление которого увеличивает количество компонент связности.

Введём на вершинах графа отношение *рёберной двусвязности*: $a \sim b$, если между a и b есть два не пересекающихся по рёбрам пути. Это отношение симметрично, рефлексивно (из вершины в неё саму есть два пути длины ноль, которые совпадают, но не пересекаются по рёбрам), а также транзитивно: пусть $a \sim b$, $b \sim c$, покажем, что $a \sim c$. Два не пересекающихся по рёбрам пути между b и c образуют цикл C. Тогда пусть u, v — первые вершины на двух не пересекающихся по рёбрам путях из a в b, которые лежат на цикле C; на цикле всегда найдутся не пересекающиеся по рёбрам пути из u и v в c.

Итак, $\sim$ — отношение эквивалентности. Классы эквивалентности, на которые оно разбивает вершины графа, называют *компонентами рёберной двусвязности (2-edge-connected components)*. Граф, состоящий из одной такой компоненты, называют *рёберно двусвязным (2-edge-connected graph)*.

Множество мостов совпадает со множеством рёбер, концы которых лежат в разных компонентах рёберной двусвязности: это рёбра, которые являются единственным путём между своими концами. Отсюда также следует, что все мосты всегда входят в любой лес поиска в глубину.

Поиск мостов

Здесь и далее будем обозначать ребро дерева поиска в глубину между вершиной u и её родителем за e_u.

Как понять по ребру $e_u = (v, u)$ дерева поиска в глубину, является ли оно мостом? Если e_u — мост, то из поддерева u нет обратных рёбер, ведущих за пределы поддерева. Если же e_u — не мост, то существует путь из u в v, не проходящий по ребру e_u, то есть содержащий обратное ребро, ведущее в v или её предка.

Пусть $low(u)$ — минимальное время входа среди всех вершин поддерева u и всех концов обратных рёбер, инцидентных вершинам этого поддерева. Тогда e_u — мост тогда и только тогда, когда $low(u) > t_{in}(v)$. Значение $low(u)$ можно вычислить во время поиска в глубину:

$$low(u) = \min\left(t_{in}(u), \min_{(u,w)-\text{обратное}} t_{in}(w), \min_{w\ -\ \text{ребёнок}\ u} low(w)\right).$$

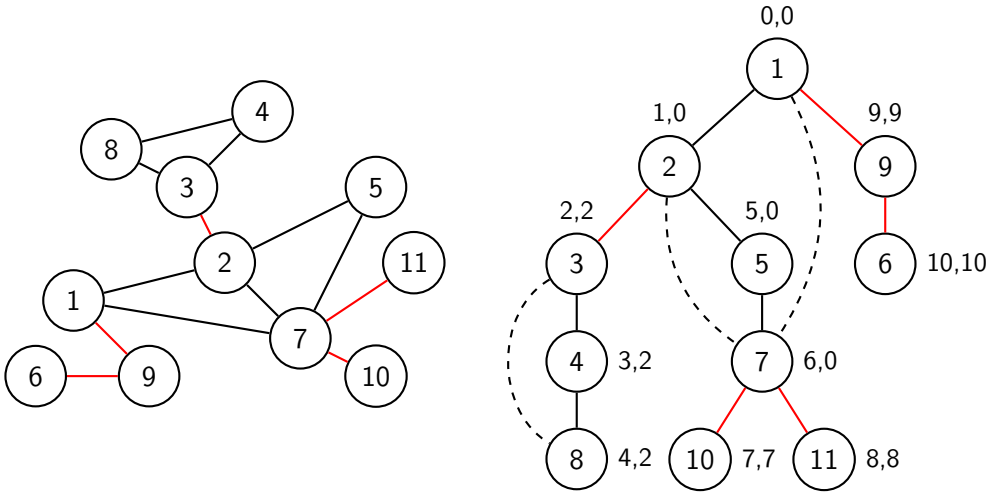


Рис. 13.7: Граф и его дерево поиска в глубину с отмеченными временами входа и значениями low. Мосты помечены красным цветом. Компоненты рёберной двусвязности: $\{1, 2, 5, 7\}$, $\{3, 4, 8\}$, $\{6\}$, $\{9\}$, $\{10\}$, $\{11\}$.

Поиск компонент рёберной двусвязности

Зная мосты, несложно найти и компоненты рёберной двусвязности: компонента, содержащая корень дерева, состоит из вершин, на пути в дереве от корня до которых нет мостов; если ребро дерева $e_u = (v, u)$ — мост, то компонента, содержащая вершину u, состоит из вершин поддерева u, на пути от u до которых нет мостов. Значит, компоненты можно найти, запуская поиски в глубину на уже построенном лесе поиска в глубину, в котором помечены мосты.

Можно найти компоненты и в процессе поиска мостов: будем складывать вершину в стек в начале её обработки. Каждый раз, когда мы находим мост $e_u = (v, u)$, будем доставать вершины из стека, пока не достанем u. При этом мы достанем вершины, лежащие в поддереве u, на пути в дереве от u до которых не было моста (иначе мы достали бы их раньше). Значит, мы достанем ровно компоненту вершины u. По тем же причинам в конце в стеке останется ровно компонента корня дерева.

Получаем алгоритм поиска мостов и компонент рёберной двусвязности за $O(V + E)$.

```
vector<vector<int> > components
vector<int> stack
T = 0

dfs(v, parent):
  visited[v] = True
  tin[v] = low[v] = T, T += 1
  stack.push_back(v)
  for u in es[v]:
    if u != parent: # проверяем, что это не ребро, по которому пришли в v;
                    # если есть кратные рёбра, нужно проверять аккуратнее
      if not visited[u]:
        dfs(u, v)
        if low[u] > tin[v]: # (v, u) - мост
          vector<int> component
          while component.size() == 0 or component.back() != u:
            component.push_back(stack.back())
            stack.pop_back()
          components.push_back(component)
        low[v] = min(low[v], low[u])
      else: # обратное ребро
        low[v] = min(low[v], tin[u])

fill(visited, False)
for v = 0..(n - 1):
  if not visited[v]:
    stack.clear()
    dfs(v, -1)
    components.push_back(stack) # компонента корня дерева
```

По любому графу можно построить лес, вершинами которого будут компоненты рёберной двусвязности графа, а рёбрами — мосты.

13.11 Поиск точек сочленения и компонент вершинной двусвязности

Точка сочленения (cut vertex, articulation point) — вершина неориентированного графа, удаление которой увеличивает количество компонент связности.

Введём на рёбрах графа отношение *вершинной двусвязности*: $e_1 \sim e_2$, если $e_1 = e_2$, или если существует *простой цикл* (цикл, в котором вершины не повторяются), проходящий одновременно через e_1 и e_2 (или, что то же самое, есть два не пересекающихся по вершинам пути из концов e_1 в концы e_2). Это отношение рефлексивно, симметрично и транзитивно:

покажем, что если $e_1 \sim e_2$ и $e_2 \sim e_3$, то $e_1 \sim e_3$. Можно считать, что e_1, e_2, e_3 попарно различны (иначе доказывать нечего). Пусть C — простой цикл, проходящий через e_2 и e_3. Пусть u, v — первые вершины на не пересекающихся по вершинам путях из концов e_1 в концы e_2, которые лежат на цикле C; $u \neq v$, на цикле найдётся два не пересекающихся по вершинам пути из u, v в концы e_3.

Значит, $\sim$ — отношение эквивалентности; классы эквивалентности, на которые разбиваются рёбра, называют *компонентами вершинной двусвязности*, или просто *компонентами двусвязности (biconnected components)*, а также *блоками (blocks)*. Граф, состоящий из одной компоненты двусвязности, называют *вершинно двусвязным*, или просто *двусвязным (biconnected graph)*.

Для каждой компоненты двусвязности F можно рассмотреть подграф $H = (W, F)$, где W — множество концов рёбер из F. Этот подграф тоже будем называть компонентой двусвязности или блоком. Заметим, что такой подграф всегда связен.

Любые два блока (как подграфы) либо не пересекаются по вершинам, либо пересекаются ровно по одной вершине: пусть компоненты H_1, H_2 пересеклись по вершинам $a \neq b$; между a и b есть два пути — один в H_1, другой в H_2. Объединение этих путей даёт простой цикл, на котором лежат рёбра из разных компонент двусвязности, что невозможно.

Множество вершин, лежащих в пересечении нескольких компонент двусвязности, совпадает со множеством точек сочленения: это вершины, у которых есть соседи из разных компонент двусвязности.

Рассмотрим вспомогательный граф, вершины которого — это точки сочленения и блоки, а рёбра соответствуют парам из точки сочленения и блока, в котором эта точка лежит. В этом графе нет циклов (точки сочленения на таком цикле не были бы точками сочленения). При этом если исходный граф связен, то этот граф тоже связен: в этом случае он является деревом, и его называют *деревом блоков и точек сочленения (block-cut tree)*. В случае несвязного графа такое дерево есть у каждой компоненты связности.

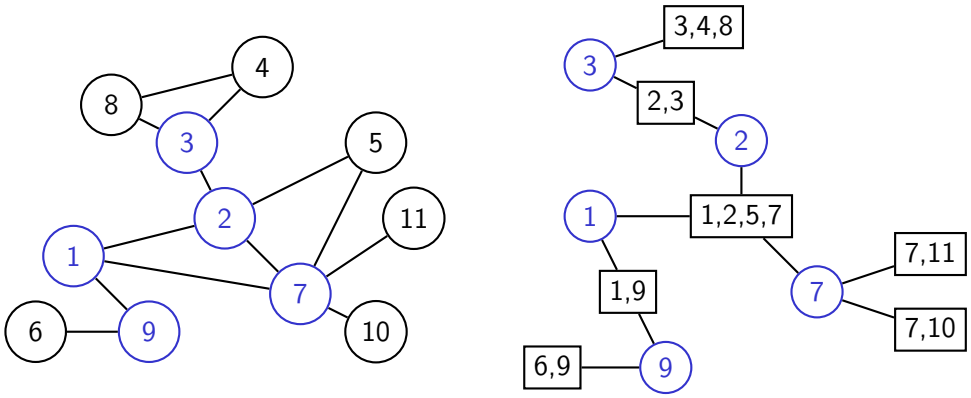


Рис. 13.8: Граф и его дерево блоков и точек сочленения. Точки сочленения помечены синим цветом.

Заметим, что мосты соответствуют компонентам двусвязности, состоящим из одного ребра.

Поиск точек сочленения

Для поиска точек сочленения снова воспользуемся значениями low. Пусть v — вершина, не являющаяся корнем дерева поиска в глубину. v — точка сочленения тогда и только тогда, когда у вершины v есть такой ребёнок u, что $low(u) \geqslant t_{in}(v)$ (то есть из поддерева u нет обратных рёбер, ведущих за пределы поддерева v). Действительно, это условие

равносильно тому, что при удалении v корень и некоторая вершина u, соседняя с v, окажутся в разных компонентах связности.

Корень дерева же является точкой сочленения тогда и только тогда, когда он имеет хотя бы два ребёнка (это следует из того, что перекрёстных рёбер не бывает).

Поиск компонент двусвязности

Как найти компоненты двусвязности? Во-первых, заметим, что любое обратное ребро между вершиной u и её предком w лежит в той же компоненте, что и ребро дерева e_u (они лежат на простом цикле, состоящем из пути в дереве от w до u и обратного ребра).

Остаётся разбить на компоненты рёбра дерева. Назовём ребро дерева $e_u = (v, u)$ *интересным*, если $low(u) \geqslant t_{in}(v)$. В частности, все рёбра дерева между корнем и его ребенком — интересные; при поиске точек сочленения, не являющихся корнем, мы искали как раз интересные рёбра.

Заметим, что если ребро $e_u = (v, u)$ — не интересное, то e_u и e_v лежат в одной компоненте двусвязности. Если же ребро $e_u = (v, u)$ — интересное, то все рёбра дерева из той же компоненты двусвязности, что и e_u, находятся в поддереве вершины u.

Получаем следующий алгоритм: будем обходить рёбра дерева в порядке поиска в глубину, тогда ребро дерева $e_u = (v, u)$ либо лежит в той же компоненте, что и e_v (если e_u не интересное), либо лежит в новой компоненте, рёбра из которой мы до этого не встречали.

Можно строить компоненты и прямо во время поиска точек сочленения (подробности на следующей странице).

```
# для удобства считаем, что граф простой (если это не так, некоторые проверки
# нужно проводить аккуратнее, а в стеке нужно различать кратные рёбра)
vector<vector<pair<int, int> > > components
vector<pair<int, int> > stack
T = 0

dfs(v, parent):
  visited[v] = True
  tin[v] = low[v] = T, T += 1
  count = 0 # считаем количество детей
  for u in es[v]:
    if u != parent: # проверяем, что это не ребро, по которому пришли в v
      if not visited[u] or tin[u] < tin[v]: # видим ребро в первый раз
        stack.push_back({v,u})
      if not visited[u]:
        count += 1
        dfs(u, v)
        if low[u] >= tin[v]: # (v, u) - интересное
          vector<pair<int, int> > component
          while component.size() == 0 or component.back() != {v,u}:
            component.push_back(stack.back())
            stack.pop_back()
          components.push_back(component)
          if parent != -1 or count >= 2:
            ... # v - точка сочленения
        low[v] = min(low[v], low[u])
      else: # обратное ребро
        low[v] = min(low[v], tin[u])

fill(visited, False)
for v = 0..(n - 1):
  if not visited[v]:
    dfs(v, -1)
```

Будем класть ребро в стек, когда видим его в первый раз (в простом графе это равносильно тому, что ребро ведёт в непосещённую вершину, либо в вершину с меньшим временем входа, чем у текущей). Каждый раз, когда мы находим интересное ребро дерева $e_u = (v, u)$, будем доставать рёбра из стека, пока не достанем e_u. При этом любое обратное ребро (a, b) (a — потомок b) мы достанем одновременно с ребром e_a (и они действительно должны лежать в одной компоненте). Что касается рёбер дерева, мы достанем все рёбра из поддерева u, которые не интересные сами, и на пути от которых до e_u нет других интересных рёбер (все остальные мы достали раньше). Это ровно те рёбра дерева, которые лежат в одной компоненте с e_u. Поскольку все рёбра из корня в его детей — интересные, по завершении обработки корня стек будет пустой.

Получаем алгоритм поиска точек сочленения и компонент двусвязности за $O(V + E)$.

13.12 2-SAT

Булева переменная (boolean variable) — переменная, принимающая одно из двух значений: истина или ложь; `true` или `false`; 1 или 0. *Литерал (literal)* — это булева переменная или её отрицание. Если x — булева переменная, то x, $\neg x$ (так обозначается отрицание) — литералы.

Конъюнктивная нормальная форма (КНФ, conjunctive normal form, CNF) представляет собой *конъюнкцию (conjunction)*, то есть логическое "И", нескольких *дизъюнктов*. Каждый *дизъюнкт (clause)* представляет собой *дизъюнкцию (disjunction)*, то есть логическое "ИЛИ", нескольких литералов. Пример КНФ от переменных x_1, x_2, x_3, x_4 (конъюнкция обозначается как $\wedge$, дизъюнкция — как $\vee$):

$$(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_4) \wedge (x_3) \wedge (\neg x1 \vee x_2 \vee \neg x_3 \vee \neg x_4).$$

Дизъюнкт называется *выполненным (satisfied)*, если хотя бы один его литерал имеет значение `true`. КНФ *выполнима (satisfiable)*, если существует набор значений переменных, при котором выполнены все дизъюнкты. Такой набор значений называют *выполняющим набором*. Один из выполняющих наборов для формулы выше — `true,false,true,false`.

В 2-КНФ каждый дизъюнкт состоит из двух литералов. Задача 2-SAT (2-satisfiability) формулируется следующим образом: дана 2-КНФ; нужно проверить, выполнима ли она, и предъявить выполняющий набор, если он существует. Примеры 2-КНФ:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_4)$$

выполнима (выполняющий набор — `true,false,false,true`);

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

невыполнима.

Решение задачи 2-SAT

Пусть дана 2-КНФ A на n переменных $x_1, \ldots, x_n$, состоящая из m дизъюнктов. Построим по ней *граф импликаций* G_A. Вершинами этого графа будут все $2n$ литералов. Для каждого дизъюнкта $(a \vee b)$ (a, b — литералы) проведём два ребра: $\neg a \to b$ и $\neg b \to a$ (если $a = \neg x_i$, то $\neg a = \neg(\neg x_i) = x_i$).

Заметим, что дизъюнкт $(a \vee b)$ эквивалентен любой из двух *импликаций (implications, "если. . . , то. . .")* $\neg a \Rightarrow b$ и $\neg b \Rightarrow a$. Таким образом, для любого пути в графе импликаций из c в d и значений c, d из любого выполняющего набора верно, что $c \Rightarrow d$.

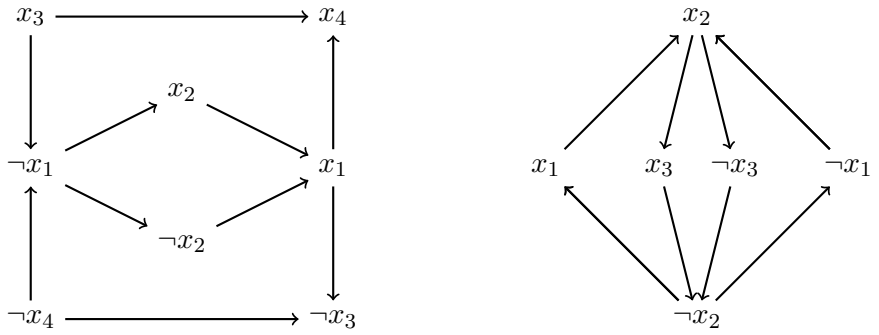


Рис. 13.9: Графы импликаций 2-КНФ из примеров выше

Если для какой-то переменной x_i литералы x_i и $\neg x_i$ оказались в одной компоненте сильной связности графа импликаций, то A невыполнима. Действительно, если бы нашёлся выполняющий набор, для него было бы верно одновременно $x_i \Rightarrow \neg x_i$ и $\neg x_i \Rightarrow x_i$. Но если x_i равно `true`, то неверна первая импликация, а если x_i равно `false`, то вторая.

Пусть это не так, то есть никакая компонента сильной связности не содержит одновременно литерал и его отрицание. Тогда построим выполняющий набор следующим образом: пронумеруем компоненты сильной связности в порядке топологической сортировки (алгоритм нахождения компонент сильной связности находит их именно в таком порядке); пусть $k[a]$ — номер компоненты литерала a. $k[x_i] \neq k[\neg x_i]$ для любой переменной x_i; присвоим переменной x_i значение `true`, если $k[x_i] > k[\neg x_i]$, и значение `false` иначе.

Почему получился выполняющий набор? Пусть нашёлся невыполненный дизъюнкт $(a \vee b)$ (a, b — литералы). В графе импликаций ему соответствуют рёбра $\neg a \to b$ и $\neg b \to a$, откуда следует $k[b] \geqslant k[\neg a]$, $k[a] \geqslant k[\neg b]$. С другой стороны, из того, что дизъюнкт невыполнен, следует, что $k[\neg a] > k[a]$, $k[\neg b] > k[b]$. Получаем $k[b] \geqslant k[\neg a] > k[a] \geqslant k[\neg b] > k[b]$, что невозможно.

Таким образом, получаем следующий алгоритм: строим граф импликаций; находим его компоненты сильной связности и значения k для всех вершин графа; проверяем, верно ли, что $k[x_i] \neq k[\neg x_i]$ для всех переменных. Если это не так, то A невыполнима; если это так, то выполняющий набор строится следующим образом: присвоим `true` всем x_i таким, что $k[x_i] > k[\neg x_i]$, остальным присвоим `false`.

Поскольку в графе импликаций $2n$ вершин и $2m$ рёбер, время работы алгоритма — $O(n + m)$.

14. Алгоритмы поиска кратчайших путей

Во *взвешенном графе (weighted graph)* каждое ребро имеет вес (weight) w_e. В зависимости от задачи, веса могут быть как целыми, так и вещественными; иногда допускаются отрицательные веса.

Длина (length) пути во взвешенном графе — это сумма весов рёбер на пути. Можно считать, что все рёбра в невзвешенном графе имеют вес, равный единице. Тогда длина пути в невзвешенном графе — это просто количество рёбер в этом пути. *Кратчайший путь (shortest path)* между двумя вершинами — это путь минимальной длины между этими вершинами (путь с минимальным числом рёбер в невзвешенном случае). *Расстояние (distance)* между вершинами — длина кратчайшего пути между ними. Если пути между вершинами нет, расстояние считается бесконечным.

Алгоритмы поиска кратчайших путей, которые мы изучим, применимы как для неориентированных, так и для ориентированных графов. Отличие лишь в том, что в ориентированном графе расстояние от вершины a до вершины b может не совпадать с расстоянием от вершины b до вершины a.

14.1 Поиск в ширину

Поиск в ширину (breadth-first search, BFS) находит расстояния в невзвешенном графе от одной вершины s до всех остальных вершин. Пусть d_v — расстояние от s до вершины v; A_k — множество вершин, расстояние до которых равняется k: $A_k = \{v \in V : \ d_v = k\}$.

Будем находить множества A_k последовательно: $A_0 = \{s\}$; пусть уже найдены $A_0, \dots, A_{k-1}$, тогда A_k — это множество вершин из $V \setminus A_0 \setminus \dots \setminus A_{k-1}$, в которые ведут рёбра из A_{k-1}. Быстро понимать, лежит ли уже вершина v в одном из множеств, можно с помощью массива расстояний $dist$: $dist[v] = k$, если $v \in A_k$; $dist[v] = \infty$, если множество, в котором лежит v, ещё не найдено (или если v не достижима из s). Здесь ∞ — бесконечность; на практике в качестве ∞ можно использовать число, заведомо большее, чем длина любого кратчайшего пути; поскольку мы работаем с невзвешенными графами, подойдёт $\infty = |V|$.

```
vector<vector<int> > A
vector<int> dist(n) # в графе n вершин
fill(dist, inf) # inf - "бесконечность"
dist[s] = 0
A.push_back({s})
for (i = 0; A[i].size() > 0; i += 1):
  A.push_back({})
  for v in A[i]:
    for u in es[v]:
      if dist[u] == inf:
        dist[u] = i + 1
        A[i + 1].push_back(u)
```

Этот алгоритм уже работает за $O(V+E)$, так как он просматривает список смежности каждой вершины не более одного раза. Можно ещё упростить алгоритм: представим, что множества $A_0, A_1, \dots$ хранятся подряд в одном массиве: $q = A_0 A_1 A_2 \dots$. Тогда

вышеописанный алгоритм просматривает элементы этого массива по порядку (то есть по очереди достаёт вершины из начала массива), и иногда добавляет новые вершины в конец массива. Значит, q — это просто очередь; можно не хранить множества A_k явно, вместо этого поддерживая очередь q. Получившийся алгоритм и называют поиском в ширину.

```
vector<int> dist(n) # в графе n вершин
fill(dist, inf)
dist[s] = 0
q <-- s # кладём s в очередь
while not q.empty():
  q --> v # достаём v из очереди
  for u in es[v]:
    if dist[u] == inf:
      dist[u] = dist[v] + 1
      q <-- u
```

Дерево кратчайших путей

Пока мы научились только находить расстояния от s до остальных вершин. Что делать, если нужно восстановить сами кратчайшие пути? Аналогично дереву поиска в глубину, можно построить дерево поиска в ширину: в него войдут рёбра, при просмотре которых алгоритм добавлял новые вершины в очередь. В простом графе достаточно для каждой вершины u запомнить её предка в дереве поиска в ширину; в графе с кратными рёбрами нужно запомнить, какое именно из кратных рёбер алгоритм просматривал при добавлении u в очередь.

Путь в этом дереве из s в любую вершину u — кратчайший; поэтому дерево поиска в ширину называют также *деревом кратчайших путей*. Для того, чтобы восстановить кратчайший путь из s в u, нужно подниматься по рёбрам дерева из u, пока не попадём в s.

```
    if dist[u] == inf:
      dist[u] = dist[v] + 1
      p[u] = v # v - родитель u в дереве кратчайших путей
      q <-- u
```

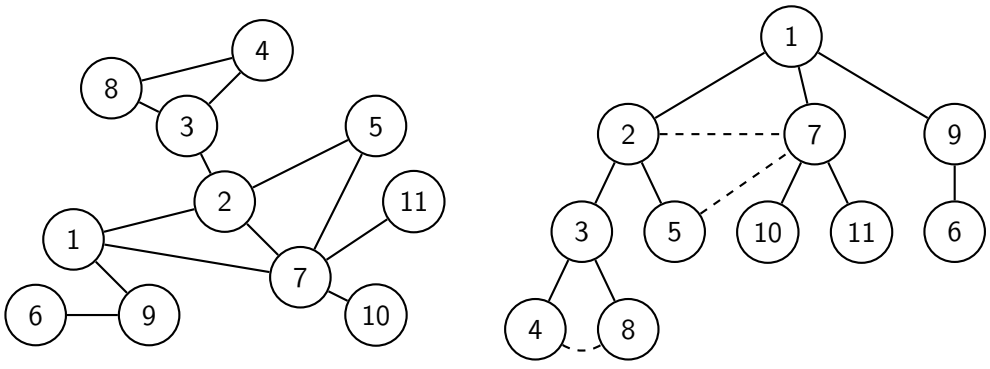


Рис. 14.1: Граф и его дерево кратчайших путей

```
vector<int> path
while u != s:
  path.push_back(u)
  u = p[u]
path.push_back(s)
reverse(path.begin(), path.end())
```

0-1-BFS

Пусть теперь каждое ребро в графе имеет вес, равный 0 или 1. Модифицируем алгоритм поиска в ширину следующим образом: вместо очереди будем использовать дек; при просмотре ребра e из v в u будем класть u в дек, если нашёлся более короткий путь до u, чем был известен ранее (если $dist[u] > dist[v] + w_e$). При этом положим u в начало дека, если $w_e = 0$, и в конец дека, если $w_e = 1$.

```
vector<int> dist(n) # в графе n вершин
fill(dist, inf)
dist[s] = 0
deque <-- s # кладём s в дек
while not deque.empty():
  v <-- deque # достаём v из начала дека
  for u, w in es[v]: # u - конец ребра, w - его вес
    if dist[u] > dist[v] + w:
      dist[u] = dist[v] + w
      if w == 0:
        u --> deque # кладём u в начало дека
      else:
        deque <-- u # кладём u в конец дека
```

Заметим сначала, что для любой вершины v значение $dist[v]$ равно ∞, либо соответствует длине какого-то пути из s в v, поэтому $dist[v]$ не может оказаться меньше длины кратчайшего пути.

Посмотрим на все вершины в порядке, в котором мы достаём их из дека в первый раз (пока для удобства будем считать, что алгоритм игнорирует повторные попадания вершины в дек). Сначала мы достанем вершину s, затем все вершины, достижимые из s только по рёбрам веса 0. К этому моменту мы достали ровно множество A_0, при этом в момент извлечения этих вершин из дека расстояние до них уже посчитано корректно.

После этого мы достанем из дека вершины из $V \setminus A_0$, достижимые из A_0 по ребру веса 1, а также все вершины, достижимые из этих по рёбрам веса 0, то есть ровно множество A_1; расстояние до этих вершин также уже посчитано корректно в момент извлечения их из дека. Аналогично, после этого мы достанем множества A_2, A_3, и так далее; в момент извлечения любой вершины из дека расстояние до неё уже посчитано корректно.

Поймём теперь, почему каждая вершина попала в дек не более двух раз (тогда время работы алгоритма оценивается как $O(V + E)$). Пусть u попала в дек в первый раз в момент обработки ребра $e = (v, u)$, в $dist[u]$ при этом было записано $dist[v] + w_e$. К этому моменту уже была хотя бы раз обработана любая вершина t с $d_t < d_v$. Значит, $d_u \geqslant d_v$. Тогда, если $w_e = 0$, то $dist[u] = d_u$, то есть u больше никогда не будет добавлена в дек. Если же $w_e = 1$, то $dist[u] \leqslant d_u + 1$, тогда u может быть добавлена в дек ещё не более одного раза.

1-k-BFS

Пусть теперь рёбра в графе могут иметь любой целый вес от 1 до k. Как найти кратчайшие пути от s до остальных вершин в таком графе?

Первый способ — заменим каждое ребро длины $l > 1$ на l рёбер длины 1, добавив при этом в граф $l - 1$ новую вершину: ребро $e = (u, v)$, $w_e = l > 1$ заменим на l рёбер $e_1 = (u, a_1), e_2 = (a_1, a_2), \ldots, e_l = (a_{l-1}, v)$, где $a_1, \ldots, a_{l-1}$ — новые вершины (свои для каждого такого ребра). Расстояния между вершинами исходного графа в новом графе не изменились. При этом все рёбра нового графа имеют длину 1, поэтому расстояния в нём можно найти поиском в ширину. Поскольку в новом графе $O(V + (k - 1)E)$ вершин и $O(kE)$ рёбер, получаем время работы $O(V + kE)$.

Второй способ — будем поддерживать свою очередь q_i для каждого расстояния i. Вначале $q_0 = \{s\}$, остальные очереди пустые; $dist[s] = 0$, $dist[v] = \infty$ для $v \neq s$. Будем рассматривать очереди q_i в порядке возрастания i. Для каждой вершины v из q_i такой, что $dist[v] = i$, и каждого ребра $e = (v, u)$ такого, что $dist[u] > dist[v] + w_e$, обновим $dist[u]$ и положим u в $q_{dist[v]+w_e}$.

```
vector<int> dist(n) # в графе n вершин
fill(dist, inf)
dist[s] = 0
q[0] <-- s
for i = 0..((n - 1) * k): # (n - 1) * k - максимально возможное расстояние
  while not q[i].empty():
    v <-- q[i]
    if dist[v] != i:
      continue
    for u, w in es[v]:
      if dist[u] > dist[v] + w:
        dist[u] = dist[v] + w
        q[i + w] <-- u
```

$dist[v]$ снова всегда равняется длине какого-то пути от s до v (либо ∞). Покажем по индукции, что вершины v такие, что $dist[v] = i$ в момент извлечения v из q_i — это ровно вершины множества A_i. Это верно для $i = 0$; для $i > 0$ в q_i попадают вершины из $V \setminus A_0 \setminus \cdots \setminus A_{i-1}$, в которые ведёт ребро веса 1 из A_{i-1}, веса 2 из A_{i-2},... или веса k из A_{i-k} (то есть вершины множества A_i), а также, возможно, часть вершин из $A_0 \cup \cdots \cup A_{i-1}$, в которые ведут такие же рёбра. Тогда такие вершины v, что $dist[v] = i$ в момент извлечения v из q_i — это ровно вершины множества A_i.

При этом каждая вершина попадёт в не более чем k различных очередей: пусть вершина u попала в $q_{dist[v]+w_e}$ при рассмотрении ребра $e = (v, u)$, до этого было верно $dist[u] = \infty$. Тогда в этот момент $dist[v] = d_v$, откуда $d_u > d_v$ (иначе u уже попала бы до этого в очередь q_i для $i \leqslant d_v$). Но тогда $dist[u] = dist[v] + w_e < d_u + k$, значит, u попадёт в другие очереди ещё не более $k - 1$ раза.

Получаем время работы $O(kV + E)$, так как каждая вершина попала не более, чем в k очередей; при этом список смежности каждой вершины был рассмотрен ровно один раз.

Произвольные веса

Что делать, если веса рёбер — произвольные вещественные числа? Попробуем модифицировать алгоритм поиска в ширину так, чтобы он всё ещё работал. Первая идея: будем класть вершину v в очередь каждый раз, когда значение $dist[v]$ уменьшается; теперь каждую вершину, мы, возможно, будем обрабатывать несколько раз. Другая идея: вместо обычной очереди будем пользоваться очередью с приоритетами, извлекая на каждом шаге такую вершину v, что значение $dist[v]$ минимально среди всех вершин в очереди.

Обе эти идеи при правильной реализации приводят к рабочим алгоритмам: первая — к алгоритму Форда-Беллмана, вторая — к алгоритму Дейкстры. Изучением этих алгоритмов мы сейчас и займёмся.

14.2 Алгоритм Дейкстры

Алгоритм Дейкстры (Dijkstra, 1959) находит расстояния от вершины s до всех остальных вершин в графе с неотрицательными весами рёбер. Алгоритм постепенно расширяет множество A вершин, до которых уже найдены расстояния и кратчайшие пути. При этом для любой вершины v алгоритм поддерживает значение $dist[v]$, равное ∞ либо длине какого-то пути из s в v.

Как и раньше, за d_v будем обозначать длину кратчайшего пути от s до v.

Вначале $dist[s] = 0$, $dist[v] = \infty$ для $v \neq s$, множество A пусто. На каждом шаге алгоритм добавляет в множество A вершину v с минимальным значением $dist[v]$ среди всех вершин из $V \setminus A$ (на первом шаге в A всегда добавляется вершина s). После этого алгоритм просматривает все исходящие из только что добавленной вершины v рёбра, и для каждого ребра $e = (v, u)$ обновляет значение $dist[u]$ значением $\min(dist[u], dist[v] + w_e)$; будем называть эту операцию *релаксацией* ребра e.

Докажем по индукции, что в момент добавления вершины v в множество A расстояние от s до v уже найдено: $dist[v] = d_v$. Это верно на первом шаге алгоритма, когда в A добавляется сама вершина s. Далее, пусть для всех вершин из A расстояние до них уже найдено, $dist[v] = \min_{u \in V \setminus A} dist[u]$; нужно показать, что $dist[v] = d_v$.

Пусть P — кратчайший путь из s в v; здесь и дальше длину пути P будем обозначать как $l(P)$. Пусть u — первая вершина на пути P, не лежащая в множестве A; x — вершина на пути перед ней, $e = (x, u)$ — соединяющее их ребро. Обозначим часть пути от s до x за P_1, часть пути от u до v за P_2, тогда $l(P) = l(P_1) + w_e + l(P_2) \geqslant l(P_1) + w_e$ (так как веса рёбер неотрицательны, то есть $l(P_2) \geqslant 0$).

$l(P_1) \geqslant d_x$, при этом $d_x = dist[x]$ по предположению индукции (так как $x \in A$). Кроме того, $dist[u] \leqslant dist[x] + w_e$, так как при добавлении x в A была произведена релаксация ребра e. Соберём все неравенства вместе:

$$d_v = l(P) \geqslant l(P_1) + w_e \geqslant d_x + w_e = dist[x] + w_e \geqslant dist[u] \geqslant dist[v].$$

С другой стороны, $dist[v] \geqslant d_v$, поскольку $dist[v]$ равно ∞ или соответствует длине какого-то пути из s в v.

Реализация

Удобно воспользоваться очередью с приоритетами: будем хранить в очереди ещё не рассмотренные вершины с приоритетами, равными значениям $dist$. Тогда в ходе алгоритма нужно $O(V)$ раз извлечь из очереди вершину с минимальным приоритетом, и $O(E)$ раз уменьшить приоритет вершины в очереди.

```
vector<int> dist(n) # в графе n вершин
fill(dist, inf)
dist[s] = 0
for v = 0..(n - 1):
  q <-- (v, dist[v]) # кладём в очередь вершину v с приоритетом dist[v]
for i = 0..(n - 1):
  v <-- q # достаём из очереди вершину с минимальным приоритетом
  for u, w in es[v]:
    if dist[u] > dist[v] + w:
      dist[u] = dist[v] + w
      q.decreaseKey(u, dist[u]) # уменьшаем значение приоритета u до dist[u]
```

Очередь с приоритетами можно реализовывать разными способами: можно просто хранить значения $dist$ в массиве, и искать минимум проходом по массиву за $O(V)$ (дополнительно надо хранить пометки о том, была ли вершина уже рассмотрена). В этом случае время работы алгоритма составит $O(V \cdot V + E \cdot 1) = O(V^2 + E)$.

Если реализовать очередь с приоритетами с помощью двоичной кучи, то получим время работы $O(V \cdot \log V + E \cdot \log V) = O((V + E) \log V)$.

Первый способ эффективнее в случае плотного графа ($E = \Omega(V^2)$), второй — при $E = o(V^2)$.

R При реализации очереди с приоритетами при помощи фибоначчиевой кучи получается теоретическая оценка времени работы $O(V \log V + E)$, так как амортизированное время выполнения операции уменьшения приоритета в таких кучах равно $O(1)$.

Если известно, что веса рёбер и длины путей — целые числа, не превосходящие C, реализация очереди с приоритетами с помощью дерева Ван-Эмде-Боаса даст оценку времени работы $O((V + E) \log \log C)$.

Восстановление кратчайшего пути

Восстановление кратчайшего пути осуществляется полностью аналогично тому, как это делалось в алгоритме поиска в ширину: запомним для каждой вершины v последнее ребро, при релаксации которого $dist[v]$ уменьшилось. Эти рёбра снова образуют дерево кратчайших путей, поднимаясь по ним, можно восстановить кратчайший путь от s до любой вершины v за время, пропорциональное количеству рёбер на этом пути.

14.3 Алгоритм A*

Алгоритм A* (произносится как "A-star"; Hart, Nilsson, Raphael, 1968) находит расстояния от вершины s до вершины t в графе с неотрицательными весами рёбер. В каком-то смысле он является модификацией алгоритма Дейкстры, более эффективной в случае, когда нужно найти расстояние от s лишь до одной вершины t, а не до всех остальных вершин.

Алгоритм использует вспомогательные значения $f[v]$ для каждой вершины графа v. f — какая-то дополнительная информация о графе. Обычно (хотя не всегда) под $f[v]$ имеется в виду какая-то оценка снизу на расстояние от v до t в графе. Например, пусть вершины графа — это точки на плоскости, а вес ребра — расстояние на плоскости между концами этого ребра. Тогда в качестве $f[v]$ логично взять расстояние на плоскости между v и t: для любого пути в графе из v в t длина этого пути будет не меньше $f[v]$.

Единственное отличие алгоритма A* от алгоритма Дейкстры: в качестве приоритета вершины v используется не $dist[v]$, а $dist[v] + f[v]$.

```
vector<int> dist(n) # в графе n вершин
fill(dist, inf)
dist[s] = 0
for v = 0..(n - 1):
  q <-- (v, dist[v] + f[v])
for i = 0..(n - 1):
  v <-- q
  if v == t:
    break # нас интересовало только расстояние до t
  for u, w in es[v]:
    if dist[u] > dist[v] + w:
      dist[u] = dist[v] + w
      q.decreaseKey(u, dist[u] + f[u])
```

Интуитивно это можно понимать так: алгоритм пытается в первую очередь рассматривать вершины, которые больше похожи на лежащие на кратчайшем пути от s к t. Поскольку $dist[v] + f[v]$ — это оценка на длину пути из s в t, проходящего через v, логично сначала рассматривать вершины, для которых эта величина минимальна.

Неравенство треугольника

Будем говорить, что для f выполняется *неравенство треугольника*, если $f[a] \leqslant f[b] + w_e$ для любого ребра e из a в b. В примере выше f удовлетворяет неравенству треугольника, поскольку оно эквивалентно неравенству треугольника для расстояний между точками на плоскости (нужно рассмотреть треугольник на вершинах a, b, t).

Предложение 14.3.1 Если для f выполняется неравенство треугольника, то алгоритм A* корректно найдёт расстояние от s до t.

Доказательство. Ход рассуждений тот же, что и в доказательстве алгоритма Дейкстры. Покажем, что в момент, когда алгоритм достаёт из очереди вершину v, расстояние до неё уже найдено: $dist[v] = d_v$. Это верно на первом шаге, когда $v = s$.

Рассмотрим один из последующих шагов. Пусть v — вершина с минимальным значением $dist[v] + f[v]$ из ещё не рассмотренных. Пусть P — кратчайший путь из s в v; u, x, e, P_1, P_2 имеют тот же смысл, что и в доказательстве алгоритма Дейкстры. Кроме того, пусть P_2 состоит из рёбер $e_1, \ldots, e_k$, соединяющих вершины $u, a_1, \ldots, a_{k-1}, v$.

По неравенству треугольника,

$$f[u] \leqslant f[a_1] + w_{e_1} \leqslant \cdots \leqslant w_{e_1} + \cdots + w_{e_k} + f[v] = l(P_2) + f[v].$$

Кроме того, снова верно, что $dist[u] \leqslant l(P_1) + w_e$, так как при обработке вершины x была произведена релаксация ребра e. Тогда

$$dist[v] + f[v] \leqslant dist[u] + f[u] \leqslant l(P_1) + w_e + l(P_2) + f[v] = l(P) + f[v],$$

откуда $dist[v] \leqslant l(P)$. ■

Алгоритм A* останавливается, когда находит расстояние до вершины t. Такое же условие остановки можно добавить в алгоритм Дейкстры, если расстояние до остальных вершин нас не интересует; будем называть такую версию алгоритмом Дейкстры с `break`.

Предложение 14.3.2 Пусть для f выполняется неравенство треугольника, $f[v] \geqslant 0$ для любой $v \in V$, $f[t] = 0$. Тогда алгоритм A* переберёт подмножество тех вершин, которые перебрал бы алгоритм Дейкстры с `break`.

Доказательство. Алгоритм Дейкстры перебирает вершины в порядке увеличения расстояния от s до них: на каждом шаге он достаёт из очереди вершину v с минимальным значением $dist[v]$, после чего при релаксации исходящих из v рёбер обновляет значения $dist$ других вершин величинами, больше или равными $dist[v]$. Значит, алгоритм Дейкстры с `break` переберёт вершины v с $dist[v] \leqslant dist[t]$.

Алгоритм A* перебирает вершины в порядке увеличения $dist[v] + f[v]$: на каждом шаге он достаёт из очереди вершину v с минимальным значением $dist[v] + f[v]$. При релаксации исходящего из v ребра $e = (v, u)$ алгоритм пытается обновить значение $dist[u]$ величиной $dist[v] + w_e$; заметим однако, что по неравенству треугольника

$$dist[v] + w_e + f[u] \geqslant dist[v] + f[v],$$

поэтому $dist[u] + f[u] \geqslant dist[v] + f[v]$ для любой извлечённой на последующих шагах вершины u. Значит, алгоритм A* переберёт вершины с $dist[v] + f[v] \leqslant dist[t] + f[t]$.

Остаётся доказать, что если $dist[v] + f[v] \leqslant dist[t] + f[t]$, то $dist[v] \leqslant dist[t]$.

Пусть $dist[v] + f[v] \leqslant dist[t] + f[t]$. Тогда

$$dist[v] \leqslant dist[t] + f[t] - f[v] \leqslant dist[t],$$

так как $f[v] \geqslant 0$, $f[t] = 0$. ■

Заметим, что если f удовлетворяет условию последнего предложения, то $f[v]$ является оценкой снизу на расстояние от v до t: пусть кратчайший путь P из v в t состоит из рёбер $e_1, \ldots, e_k$. Тогда, применив несколько раз неравенство треугольника, получаем

$$f[v] \leqslant w_{e_1} + \cdots + w_{e_k} + f[t] = l(P) + f[t] = l(P),$$

поскольку $f[t] = 0$.

14.4 Алгоритм Форда-Беллмана

Алгоритм Форда-Беллмана (Shimbel, 1955; Bellman, 1958; Ford, 1956; Moore, 1957) находит расстояние от вершины s до всех остальных вершин в графе с произвольными вещественными весами, но без *отрицательных циклов* (циклов, длина которых отрицательна).

Если из s достижим отрицательный цикл, то кратчайшего пути до любой вершины на этом цикле (и до любой вершины, достижимой из вершин цикла) не существует: к любому пути можно добавлять проходы по циклу, при этом с каждым таким проходом длина пути будет уменьшаться.

R Заметим, что в неориентированном графе любое ребро отрицательного веса соответствует отрицательному циклу, который получается при проходе по этому ребру вперёд и назад. Поэтому в неориентированных графах понятие кратчайшего пути имеет смысл, только если веса всех рёбер неотрицательны.

Пусть в графе нет (достижимых из s) отрицательных циклов. Тогда для любой достижимой из s вершины v найдётся *простой* (без повторяющихся вершин) кратчайший путь из s в v. Если бы это было не так, то путь содержал бы цикл. Тогда, удалив цикл из пути, мы не увеличим длину пути. Значит, достаточно искать простые кратчайшие пути из s в другие вершины. С этой задачей справляется следующий алгоритм:

```
fill(dist, inf)
dist[s] = 0
for k = 0..(n - 2):
  for v = 0..(n - 1):
    for u, w in es[v]:
      dist[u] = min(dist[u], dist[v] + w)
```

Всё, что делает этот алгоритм — $|V|-1$ раз проводит релаксацию всех рёбер графа. Почему он найдёт расстояния до всех вершин корректно? Пусть Q — любой простой путь из s в v; пусть он состоит из рёбер $e_1, \dots, e_k$, $k \leqslant |V|-1$. На первом шаге внешнего цикла алгоритм проведёт релаксацию ребра e_1, на втором шаге — ребра e_2,..., на k-м — ребра e_k. Тогда после k шагов внешнего цикла

$$dist[v] \leqslant w_{e_1} + \dots + w_{e_k} = l(Q).$$

Мы знаем, что если v достижима из s, то существует простой кратчайший путь P из s в v, тогда после $|V|-1$ шага внешнего цикла $dist[v] \leqslant l(P)$. С другой стороны, $dist[v]$ в любой момент — либо ∞, либо длина какого-то пути из s в v. Значит, $dist[v] \geqslant l(P)$.

Время работы алгоритма — $O(VE)$.

Восстановление кратчайшего пути

Восстановление кратчайшего пути полностью аналогично алгоритмам поиска в ширину и Дейкстры — запоминаем для каждой вершины последнее ребро, при релаксации которого $dist[v]$ уменьшилось; эти рёбра образуют дерево кратчайших путей и позволяют восстановить кратчайший путь от s до любой достижимой из неё вершины v за время, пропорциональное количеству рёбер на этом пути.

Поиск отрицательного цикла

Что делать, если неизвестно, есть ли в графе отрицательные циклы? С помощью алгоритма Форда-Беллмана можно проверить их наличие, и даже найти один из отрицательных циклов, если они есть. Всё, что нужно сделать — ещё одну ($|V|$-ю по счёту) релаксацию всех

рёбер графа в конце алгоритма. Будем называть релаксацию ребра $e = (v, u)$ *успешной*, если в результате её выполнения $dist[u]$ строго уменьшилось.

Если в графе нет отрицательных циклов, то успешных релаксаций на дополнительном шаге алгоритма не произойдёт, поскольку после $|V| - 1$ шага все расстояния уже были найдены. С другой стороны, пусть в графе есть достижимый из s отрицательный цикл, состоящий из вершин $v_1, \ldots, v_k$ и рёбер $e_1, \ldots, e_k$ (e_i соединяет v_i и v_{i+1}, $v_{k+1} = v_1$). Тогда, если на последнем шаге алгоритма не было успешных релаксаций, то $dist[v_{i+1}] \leqslant dist[v_i] + w_{e_i}$ для $1 \leqslant i \leqslant k$, откуда

$$\sum_{i=1}^{k} dist[v_i] \leqslant \sum_{i=1}^{k} dist[v_i] + \sum_{i=1}^{k} w_{e_i},$$

то есть $w_{e_1} + \cdots + w_{e_k} \geqslant 0$, что противоречит отрицательности цикла.

Итак, для того, чтобы проверить, если ли в графе достижимые из s отрицательные циклы, достаточно проверить, произойдёт ли на дополнительном шаге алгоритма хотя бы одна успешная релаксация.

R Для того, чтобы проверить, есть ли вообще в графе отрицательные циклы (не обязательно достижимые из какой-то фиксированной вершины графа), можно добавить в граф новую вершину, провести из неё во все остальные вершины рёбра веса 0, и использовать эту вершину в качестве s.

Восстановить отрицательный цикл можно примерно так же, как восстанавливается кратчайший путь: пусть $e[v]$ — последнее ребро, при релаксации которого $dist[v]$ уменьшилось, $p[v]$ — его второй конец. $p[v]$ и $e[v]$ не определены, если $dist[v]$ ни разу не уменьшалось в ходе алгоритма.

Лемма 14.4.1 Если $p[v]$, $e[v]$ определены, то $dist[v] \geqslant dist[p[v]] + w_{e[v]}$.

Доказательство. $dist[v]$ не менялось после успешной релаксации $e[v]$, а $dist[p[v]]$ могло только уменьшиться. ■

Пусть на последнем шаге произошла успешная релаксация ребра из u в v. Будем выписывать последовательность вершин $v, u, p[u], p[p[u]], \ldots$, пока она не зациклится, то есть пока какая-то вершина x в последовательности не повторится (отметим, что это не обязательно будет вершина v). Подотрезок последовательности с первого по второе вхождение x и будет образовывать отрицательный цикл.

Почему описанный выше алгоритм действительно найдёт отрицательный цикл? Сначала нужно понять, почему последовательность действительно зациклится. Предположим, что повтора вершины так и не произошло. Переобозначим выписанную последовательность за $v = v_1, u = v_2, \ldots, v_k$. Тогда $p[v_k]$ не определено, при этом $dist[v_k] \neq \infty$, поскольку v_k участвовала в успешной релаксации ребра $e[v_{k-1}]$. Тогда $v_k = s$, $dist[v_k] = 0$. Значит, выписанная последовательность образует простой путь P из s в v. При этом по лемме 14.4.1 выполняется неравенство

$$dist[v] = dist[v_1] \geqslant dist[v_2] + w_{e[v_1]} \geqslant \cdots \geqslant dist[v_k] + w_{e[v_1]} + \cdots + w_{e[v_{k-1}]} = l(P).$$

С другой стороны, поскольку P — простой путь, $dist[v] \leqslant l(P)$ после $|V| - 1$ шага внешнего цикла. Но это противоречит тому, что $dist[v]$ уменьшилось при релаксации ребра $e[v]$ на дополнительном шаге алгоритма.

Итак, последовательность зациклилась; обозначим вершины в подотрезке последовательности за $v_1, \ldots, v_k, v_{k+1} = v_1$ ($p[v_i] = v_{i+1}$); эти вершины вместе с рёбрами

$e[v_1], \ldots, e[v_k]$ образуют цикл C. Осталось показать, что длина этого цикла отрицательна. По лемме 14.4.1, $dist[v_i] \geqslant dist[v_{i+1}] + w_{e[v_i]}$. Просуммируем эти неравенства по i от 1 до k и получим

$$\sum_{i=1}^{k} dist[v_i] \geqslant \sum_{i=1}^{k} dist[v_i] + \sum_{i=1}^{k} w_{e[v_i]},$$

откуда $l(C) = w_{e[v_1]} + \cdots + w_{e[v_k]} \leqslant 0$. Нам же нужно показать, что $l(C) < 0$. Пусть v_j — вершина на цикле C, значение $dist[v_j]$ которой менялось в последний раз позже остальных вершин на цикле. Тогда $dist[v_{j-1}] > dist[v_j] + w_{e[v_{j-1}]}$, поскольку $dist[v_j]$ уменьшилось после релаксации ребра $e[v_{j-1}]$. Значит, одно из k неравенств выше строгое, и $l(C) < 0$.

Алгоритм Форда-Беллмана с очередью

Вернёмся к графам без отрицательных циклов. Алгоритм Форда-Беллмана можно соптимизировать, избавившись от повторного выполнения одних и тех же действий. При этом получается алгоритм, который называют алгоритмом Форда-Беллмана с очередью; также он известен как SPFA (Shortest path faster algorithm; Moore, 1959). Он имеет ту же теоретическую оценку времени, что и алгоритм Форда-Беллмана, но на практике часто работает намного быстрее.

Первая идея — на k-м шаге алгоритма нет смысла проводить релаксацию исходящих из v рёбер, если $dist[v]$ не поменялось с $(k-1)$-го шага. Обозначим за B_k множество таких вершин v, что $dist[v]$ уменьшилось в течение $(k-1)$-го шага; $B_0 = \{s\}$. На k-м шаге алгоритма достаточно произвести релаксацию рёбер, исходящих из вершин множества B_k.

```
fill(dist, inf)
dist[s] = 0
B[0] <-- s
for k = 0..(n - 2):
  for v in B[k]:
    for u, w in es[v]:
      if dist[u] > dist[v] + w:
        dist[u] = dist[v] + w
        if u not in B[k + 1]:
          B[k + 1] <-- u
```

Получившийся алгоритм очень похож на версию алгоритма поиска в ширину, в которой множества $A_k = \{v \in V : \ d_v = k\}$ строились явно. Как и в том случае, множества B_k можно моделировать с помощью очереди: будем складывать вершину v в очередь, если $dist[v]$ уменьшилось, и v в данный момент ещё не лежит в очереди. Такой алгоритм с очередью эквивалентен предыдущему алгоритму, но с ещё одной оптимизацией: теперь мы не будем класть вершину v в B_{k+1}, если с момента её обработки в B_k до конца обработки вершин из B_k $dist[v]$ не менялось. Эту оптимизацию можно сделать, так как релаксация рёбер, исходящих из такой вершины v, на $(k+1)$-м шаге бессмысленна, поскольку значение $dist[v]$ не поменялось с предыдущей релаксации тех же рёбер.

```
fill(dist, inf)
fill(inQueue, False)
dist[s] = 0
q <-- s, inQueue[s] = True
while not q.empty():
  q --> v, inQueue[v] = False
  for u, w in es[v]:
    if dist[u] > dist[v] + w:
      dist[u] = dist[v] + w
      if not inQueue[u]:
        q <-- u, inQueue[u] = True
```

Время работы получившегося алгоритма можно оценить как $O(VE)$, поскольку он делает не больше релаксаций рёбер, чем делал обычный алгоритм Форда-Беллмана.

14.5 Алгоритм Флойда

Алгоритм Флойда (Floyd, 1962; Roy, 1959; Warshall, 1962) находит расстояния между всеми парами вершин в графе с произвольными вещественными весами, но без отрицательных циклов.

Алгоритм последовательно для $0 \leqslant k \leqslant n$ и для каждой пары вершин $0 \leqslant i, j < n$ вычисляет $dist[k, i, j]$ — минимальную длину пути из i в j, *промежуточные вершины* (все вершины, кроме концов пути) в котором имеют номера меньше k. Будем считать, что $dist[k, i, j] = \infty$, если ни одного такого пути не существует. $dist[n, i, j]$ — это расстояние от i до j.

Если $i \neq j$, то $dist[0, i, j] = w_e$, где e — минимальное по весу ребро из i в j; $dist[0, i, j] = \infty$, если из i в j нет рёбер. $dist[0, i, i] = 0$ для любого i.

Пусть теперь $k \geqslant 0$, для всех пар вершин i, j уже известно $dist[k, i, j]$; чему равняется $dist[k + 1, i, j]$? Пусть P — минимальный по длине путь из i в j, промежуточные вершины в котором имеют номера меньше $k+1$. Если вершина k не является промежуточной вершиной P, то $dist[k + 1, i, j] = dist[k, i, j]$. Если же k является промежуточной вершиной P, то можно считать, что P проходит через k ровно один раз (так как в графе нет отрицательных циклов). Тогда P распадается на два пути: путь P_1 из i в k, и путь P_2 из k в j. При этом промежуточные вершины в путях P_1, P_2 имеют номера меньше k; P_1, P_2 имеют минимально возможную длину среди таких путей (иначе P был бы не минимальным по длине). Значит, $l(P_1) = dist[k, i, k]$, $l(P_2) = dist[k, k, j]$. Итак,

$$dist[k + 1, i, j] = \min(dist[k, i, j], dist[k, i, k] + dist[k, k, j]).$$

Получаем алгоритм со временем работы $O(V^3)$, использующий $O(V^3)$ дополнительной памяти. На самом деле можно ограничиться $O(V^2)$ памяти: для любого k будем использовать одно и то же $dist[i, j]$ вместо $dist[k, i, j]$.

```
# dist[i, j] равняется минимальному весу ребра из i в j или inf, если рёбер из i в j нет
# dist[i, i] = 0
for k = 0..(n - 1):
  for i = 0..(n - 1):
    for j = 0..(n - 1):
      dist[i, j] = min(dist[i, j], dist[i, k] + dist[k, j])
```

Почему алгоритм с двумерным массивом вместо трёхмерного остаётся корректным? В любой момент $dist[i, j]$ — это длина какого-то пути из i в j; при этом после k итераций внешнего цикла $dist[i, j]$ не больше длины любого пути из i в j, промежуточные вершины в котором имеют номера меньше k (это было верно для $dist[k, i, j]$, а $dist[i, j] \leqslant dist[k, i, j]$ после k итераций). Значит, после n итераций $dist[i, j]$ — это расстояние от i до j.

Восстановление пути

Для каждой пары i, j запомним $maxNum[i, j]$ — последнее k, при котором $dist[i, j]$ уменьшилось. Тогда $maxNum[i, j]$ — максимальная по номеру вершина на кратчайшем пути из i в j; кратчайший путь из i в j состоит из двух путей: кратчайшего пути из i в k и из k в j. Каждый из этих путей восстановим рекурсивно теми же рассуждениями.

Альтернативный способ: для каждой пары i, j запомним $first[i, j]$ — первую после i вершину на кратчайшем пути. Вначале $first[i, j] = j$, если $dist[i, j]$ равняется весу ребра из i в j, и -1, если рёбер из i в j нет. В момент, когда $dist[i, j]$ обновляется значением

$dist[i,k] + dist[k,j]$, обновим $first[i,j]$ значением $first[i,k]$. Теперь для того, чтобы восстановить путь из i в j, надо заменять i на $first[i,j]$, пока i не станет равно j.

Графы с отрицательными циклами

Что делать, если в графе могут быть отрицательные циклы? Заметим, что массив $dist$, получившийся в результате выполнения алгоритма, удовлетворяет следующим утверждениям:

- **Вершина i лежит на отрицательном цикле тогда и только тогда, когда $dist[i,i] < 0$.**
  Действительно, поскольку $dist[i,i]$ — это длина какого-то пути из i в i, если $dist[i,i] < 0$, то i лежит на отрицательном цикле. С другой стороны, пусть i лежит на отрицательном цикле C, и k — вершина с максимальным номером на этом цикле, не считая i. Тогда после k-го шага алгоритма $dist[i,i] \leqslant dist[i,k] + dist[k,i] \leqslant l(C) < 0$.
- **В графе не существует кратчайшего пути из a в b тогда и только тогда, когда существует такая вершина i, что $dist[i,i] < 0$, $dist[a,i] < \infty$, $dist[i,b] < \infty$.**
  Действительно, в графе не существует кратчайшего пути из a в b тогда и только тогда, когда из a достижим отрицательный цикл, из которого достижима вершина b; вершина i из условия выше — любая вершина на этом цикле.

Заметим, что, пользуясь первым утверждением, можно не только понять, есть ли в графе отрицательные циклы, но и восстановить какой-то из них: это делается так же, как восстанавливается кратчайший путь в графе без отрицательных циклов.

Пользуясь вторым утверждением, можно за $O(V^3)$ для каждой пары вершин a, b понять, существует ли кратчайший путь из a в b. Если путь существует, то его длина была найдена алгоритмом корректно, и путь может быть восстановлен обычным способом.

R В графе с неотрицательными весами рёбер расстояния между всеми парами вершин можно найти несколькими запусками алгоритма Дейкстры за $O(V^2 \log V + VE)$ (если использовать фибоначчиеву кучу). В разреженных графах ($E = o(V^2)$) этот способ даёт лучшую теоретическую оценку, чем алгоритм Флойда. В графах с произвольными вещественными весами рёбер, но без отрицательных циклов, можно добиться той же оценки времени работы, специальным образом модифицировав веса рёбер (Johnson, 1977). Мы изучим эту идею подробно, когда будем изучать задачу поиска максимального потока минимальной стоимости.

R В **неориентированном** графе с неотрицательными весами рёбер расстояния от одной вершины до всех остальных можно находить за $O(V + E)$ (Thorup, 1999). Несколькими запусками этого алгоритма можно найти расстояния между всеми парами вершин за $O(V^2 + VE)$.

V Жадные алгоритмы

15. Жадные алгоритмы

15.1 Алгоритм Хаффмана

Алгоритм Хаффмана (Huffman, 1952) — один из самых известных алгоритмов сжатия текста. Пусть дан текст, состоящий из символов алфавита Σ, который мы хотим закодировать как можно более короткой последовательностью бит.

Префиксные коды

Самый простой способ — кодировать каждый символ уникальной последовательностью из k бит (будем называть такую последовательность *кодовым словом (codeword)*), где $2^k \geqslant |\Sigma|$ (чтобы такое кодирование вообще было возможно). Такой способ не всегда эффективен: например, пусть $\Sigma = \{a, b, c, d\}$, текст имеет длину 10000, но большая часть символов текста равна a (скажем, a встречается в тексте 7000 раз, а остальные символы — по 1000 раз). Тогда вышеописанный способ кодирования потребует 20000 бит. Если же кодовое слово символа a будет равно 0, символа b — 10, c — 110, d — 111, то суммарно потребуется лишь $7000 \cdot 1 + 1000 \cdot 2 + 1000 \cdot 3 + 1000 \cdot 3 = 15000$ бит.

Итак, иногда оказывается выгодным использовать кодовые слова разной длины для разных символов. Заметим, что при этом нельзя использовать одновременно кодовые слова, одно из которых является префиксом другого: в этом случае нельзя будет понять, где в закодированной последовательности бит заканчивается одно кодовое слово, и начинается следующее. Коды, в которых ни одно кодовое слово не является префиксом другого, называют *префиксными (prefix codes)* (логичнее было бы называть их беспрефиксными, но термин "префиксный код" является общепринятым).

Какой префиксный код будет оптимальным для данного текста t? Пусть $cnt(t, a)$ — количество вхождений символа a в текст (будем называть это количество *частотой*), $len(C, a)$ — длина кодового слова символа a в префиксном коде C. Тогда количество бит, которое понадобится для того, чтобы закодировать текст t кодом C, равняется

$$L(C, t) = \sum_a cnt(t, a) \cdot len(C, a).$$

Мы хотим найти префиксный код C, минимизирующий $L(C, t)$.

Связь с двоичными деревьями

Любому префиксному коду можно сопоставить двоичное дерево: кодовые слова будут соответствовать листьям дерева; для того, чтобы по листу получить кодовое слово, нужно пройти по пути из корня в этот лист, и выписывать 0 при спуске в левого ребёнка, 1 при спуске в правого.

В дереве оптимального префиксного кода ни у какой вершины не может быть ровно один ребенок: такую вершину можно просто удалить, поставив её ребёнка на её место; при этом новое дерево будет всё ещё соответствовать префиксному коду, но длина некоторых кодовых слов уменьшится.

a	0
b	1000
c	1001
d	101
e	11

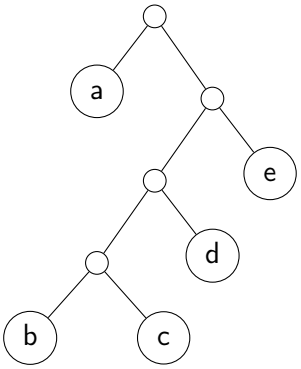


Рис. 15.1: Префиксный код и соответствующее ему двоичное дерево

Жадный алгоритм

Посмотрим на самый глубокий лист v в дереве оптимального префиксного кода. У родителя v есть ещё один ребёнок; обозначим его за u. Пути от корня до v и u являются кодовыми словами каких-то двух символов. Если это не два самых редко встречающихся в тексте t символа, то поменяем их местами с самыми редкими. При этом длина закодированного текста может только уменьшиться. Значит, всегда найдётся оптимальный префиксный код, в дереве которого два самых редких символа находятся в соседних листьях на максимальной глубине.

Пусть C — такой оптимальный код, a и b — два самых редких символа в t; они находятся в листьях v, u с общим родителем w. Введём новый символ c и рассмотрим текст t_1, полученный из t заменой всех вхождений a и b на c. Также рассмотрим префиксный код C_1, соответствующий дереву кода C, из которого удалили v и u, а в ставшую листом вершину w поместили символ c. Заметим, что $L(C,t) = L(C_1,t_1) + cnt(t,a) + cnt(t,b)$.

Код C_1 оптимален для текста t_1: пусть нашёлся такой код D_1, что $L(D_1,t_1) < L(C_1,t_1)$. Проделаем с деревом D_1 обратную операцию: добавим два ребёнка листу, в котором находится символ c, и поместим в них символы a и b. Получится дерево такого кода D, что

$$L(D,t) = L(D_1,t_1) + cnt(t,a) + cnt(t,b) < L(C_1,t_1) + cnt(t,a) + cnt(t,b) = L(C,t).$$

Но это противоречит оптимальности C, значит такого D_1 не существует, и C_1 — оптимален. Кроме того, по любому оптимальному коду для t_1 вышеописанной операцией мы можем построить оптимальный код для t.

Будем применять эти рассуждения рекурсивно, и получать строки, в которых всё меньше и меньше различных символов. В конце концов мы получим строку, в которой встречается всего два различных символа. Оптимальный префиксный код для такой строки кодирует каждый символ одним битом.

Реализация

Будем хранить символы в очереди с приоритетами: приоритет символа равен его частоте. Пока в очереди больше одного символа, будем доставать из очереди два символа a, b с минимальными приоритетами cnt_a, cnt_b, и складывать в очередь новый символ c с приоритетом $cnt_c = cnt_a + cnt_b$. В этот момент будем подвешивать вершины, соответствующие символам a и b, детьми к вершине, соответствующей символу c. В конце в очереди останется только один символ; вершина, соответствующая этому символу, будет корнем дерева оптимального префиксного кода. Получаем время работы $O(|\Sigma| \log |\Sigma|)$ (если считать, что частоты символов уже предподсчитаны).

```
# для удобства считаем, что символы - числа от 0 до n - 1
for i = 0..(n - 1):
  q <-- (i, cnt[i]) # кладём в очередь i с приоритетом cnt[i]
for c = n..(2 * n - 2): # n - 1 итерация
  a <-- q, b <-- q # извлекаем два элемента с минимальными приоритетами
  cnt[c] = cnt[a] + cnt[b]
  v[c]->l = v[a], v[c]->r = v[b] # подвешиваем a и b детьми к c
  q <-- (c, cnt[c])
r <-- q
return v[r] # возвращаем корень дерева
```

R Для того, чтобы закодированный текст можно было потом раскодировать, необходимо вместе с ним в каком-то виде хранить дерево префиксного кода или таблицу частот, по которой дерево можно будет построить заново.

15.2 Оптимальное кэширование

Под *кэшированием (caching)* обычно имеют в виду хранение малого количества данных в быстрой памяти, производимое для того, чтобы реже взаимодействовать с медленной памятью. В современных компьютерах кэширование происходит одновременно на многих уровнях: есть кэш процессора, более медленная оперативная память, ещё более медленный жёсткий диск. Сам жёсткий диск используется браузерами для кэширования часто посещаемых веб-страниц: чтение с диска быстрее, чем их повторная загрузка из интернета.

Кэширование тем эффективнее, чем чаще оказывается, что запрашиваемые данные уже находятся в кэше. Алгоритм управления кэшом определяет, какую информацию хранить в кэше, и какую информацию удалять из него, если требуется записать в кэш новые данные.

Сформулируем задачу в абстрактном виде: есть множество из n фрагментов данных, хранящихся в основной памяти. Более быстрая кэш-память способна хранить $k < n$ фрагментов данных; можно считать, что в начале работы алгоритма кэш пустой, либо что он уже содержит какие-то k элементов (дальнейшие рассуждения от этого не зависят). Нужно обработать последовательность обращений к памяти $d_1, \ldots, d_m$; алгоритм должен постоянно принимать решение о том, какие элементы хранить в кэше. Запрашиваемый элемент d_i читается очень быстро, если он уже находится в кэше. В противном случае его нужно скопировать в кэш из основной памяти; при этом, если кэш заполнен, нужно предварительно удалить из кэша какой-то другой элемент. Такая ситуация называется *кэш-промахом (cache miss)*. Требуется минимизировать количество операций записи в кэш при обработке последовательности запросов к памяти.

Заметим, что, вообще говоря, количество операций записи в кэш может не совпадать с количеством кэш-промахов: алгоритм управления кэшом мог бы записывать в кэш элементы, которые понадобятся не прямо сейчас, а когда-нибудь потом. Поймём, почему такие действия бессмысленны: пусть в какой-то момент алгоритм записывает в кэш элемент x; если x ни разу не будет запрошен до момента его удаления из кэша (или до конца работы алгоритма), то эту запись можно было просто не производить. Если же x будет запрошен позже, то эту запись можно отложить непосредственно до момента, когда он будет запрошен: ячейка кэш-памяти, которую он занимает, всё равно до этого момента не будет никак использоваться.

Таким образом, по любому алгоритму мы можем построить его "ленивую" версию, которая записывает элемент в кэш, только если сразу после этого он будет запрошен. При этом она делает не больше операций записи в кэш, чем исходный алгоритм; количество операций записи в кэш для неё совпадает с количеством кэш-промахов.

R Конечно, на практике алгоритм управления кэшом не обладает информацией о будущих запросах. Тем не менее, обладая этой информацией, можно решить задачу оптимально и получить теоретически минимально возможное количество промахов для данной последовательности запросов. Таким образом, алгоритм, который мы сейчас изучим, используется для оценки качества применяемых на практике алгоритмов.

Алгоритм Белади

Алгоритм Белади (Bélády, 1966) следует следующему правилу: когда нужно записать в кэш элемент d_i, и свободного места в кэше нет, он удаляет из кэша элемент, который понадобится в следующий раз позже всех остальных.

Обозначим алгоритм Белади за B. Почему B оптимален? Заметим, что B — ленивый. Пусть S_0 — ленивый алгоритм, делающий минимально возможное количество кэш-промахов на последовательности запросов $d_1, \ldots, d_m$. Для каждого $1 \leqslant i \leqslant m$ построим S_i — ленивый алгоритм, делающий не больше промахов, чем S_0, и при этом при обработке первых i запросов делающий те же действия, что и B. Тогда S_m делает то же количество промахов, что и B, а значит, B делает не больше промахов, чем S_0.

Пусть $1 \leqslant i \leqslant m$, S_{i-1} уже построен. При обработке первых $i-1$ запросов B и S_{i-1} делали одни и те же действия, в частности, к моменту обработки i-го запроса содержимое кэша для этих алгоритмов совпадает. Если d_i уже находится в кэше, или если B и S_{i-1} при записи d_i в кэш удаляют из кэша один и тот же элемент, можно взять $S_i = S_{i-1}$.

Пусть при записи d_i в кэш S_{i-1} удаляет из кэша элемент a, а B — элемент $b \neq a$. Заметим, что a будет запрошен раньше, чем b (либо они оба больше не будут запрошены). Определим S_i следующим образом: при обработке первых $i-1$ запросов он ведёт себя как S_{i-1} и B; при обработке i-го запроса оне ведёт себя как B, то есть удаляет из кэша b и записывает на его место d_i. Далее он ведёт себя как S_{i-1}, пока не произойдёт одно из двух событий:

- S_{i-1} удаляет b из кэша, и записывает на его место запрошенный элемент d_j. В этот момент S_i запишет d_j в кэш на место a; после этого содержимое кэша для S_{i-1} и S_i совпадает, поэтому дальше S_i просто повторяет те же действия, что и S_{i-1}.
- Запрашивается элемент $d_j = a$, и S_i удаляет из кэша элемент x, чтобы записать на его место a. Если $x = b$, то S_{i-1} никак не меняет кэш; если $x \neq b$, то S_{i-1} удаляет x, и записывает на его место b. В любом случае, после этого содержимое кэша для S_{i-1} и S_i совпадает, поэтому дальше S_i ведёт себя, как S_{i-1}.
  Здесь есть тонкость: в случае $x \neq b$ алгоритм S_i ведёт себя не лениво, так как он записывает в кэш b перед запросом $d_j = a \neq b$. Однако такой S_{i-1} можно сделать ленивым, не увеличивая количество операций записи в кэш; переобозначим за S_i его ленивую версию.

В любом из вышеописанных случаев полученный алгоритм S_i является ленивым и делает не больше кэш-промахов, чем S_{i-1}, то есть не больше, чем S_0.

R Многие используемые на практике алгоритмы кэширования основаны на *принципе LRU (least recently used)*: из кэша удаляется элемент, который дольше всех не запрашивался. В каком-то смысле это алгоритм Белади, но с изменённым направлением времени: вместо будущих запросов изучаются предыдущие. Этот принцип часто оказывается эффективным, поскольку для многих приложений характерна *локальность обращений (locality of reference)* — программа часто продолжает обращаться к данным, к которым обращалась недавно.

Библиография

Книги

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest и Clifford Stein. *Introduction to Algorithms (3rd edition)* (цитируется на странице 15).