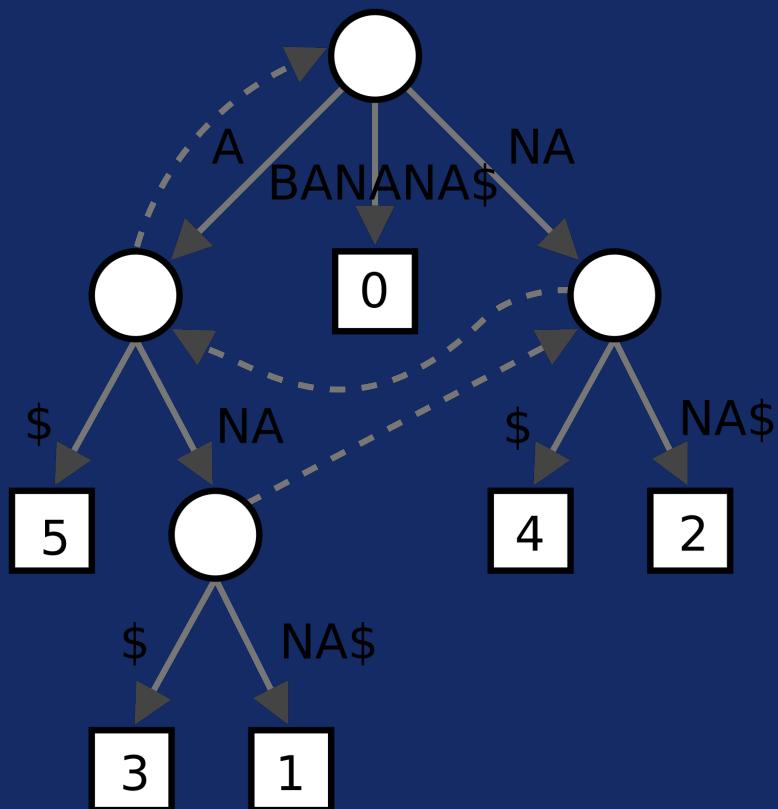


Математические основы алгоритмов

Билеты



Содержание

| | |
|--|----|
| 0 Асимптотические обозначения | 4 |
| 1 Метод «разделяй и властвуй». Быстрое умножение Карацубы, анализ времени работы. | 5 |
| 2 Метод динамического программирования. Задача о разделении стержня. Задача о порядке умножения матриц. | 7 |
| 3 Нахождение наибольшей общей подпоследовательности: «народный» алгоритм, алгоритм Хиршберга. | 11 |
| 4 Сортировка вставками, сортировка слиянием, куча и сортировка кучей | 15 |
| 5 «Быстрая сортировка», ожидаемое время работы при случайному выборе опорного элемента. | 20 |
| 6 Нижняя оценка числа сравнений при сортировке. Сортировка подсчётом. Поразрядная сортировка. | 23 |
| 7 Нахождение i -го по величине элемента массива. | 27 |
| 8 Поиск в ориентированном графе: поиск в ширину, поиск в глубину, топологическая сортировка, нахождение компонентов сильной связности. | 29 |
| 9 Кратчайшие пути в графе с весами: алгоритм Беллмана–Форда, алгоритм Дейкстры. Очередь с приоритетами и её реализация. | 35 |
| 10 Кратчайшие пути в графе между всеми парами вершин: алгоритм Варшалла. Кратчайшие пути с весами: алгоритм Флойда–Варшалла. Нахождение всех путей с помощью умножения матриц. | 39 |
| 11 Быстрое умножение матриц: алгоритм Штрассена. | 42 |
| 12 Быстрое умножение булевых матриц через числовые: метод четырёх русских. | 45 |
| 13 Структуры данных для представления множеств: вектор, список, двоичное дерево поиска. Основные операции и сложность их реализации. АВЛ-деревья, операции над ними, их сложность. | 47 |
| 14 В-деревья. Реализация операций над ними, их сложность. Понятие о красно-чёрных деревьях | 56 |
| 15 Полиномиальное хэширование строк. Алгоритм Рабина–Карпа. Нахожде- | |

| | |
|---|-----|
| ние наибольшей общей подстроки. Нахождение самого длинного палиндрома | 61 |
| 16 Поиск в строке: алгоритм Кнута–Морриса–Пратта и его реализация на конечном автомате | 67 |
| 17 Префиксное дерево для множества строк. Алгоритм Ахо–Корасик  | 72 |
| 18 Сжатие данных методом Хаффмана. Арифметическое кодирование | 76 |
| 19 Суффиксное дерево и его применение. Алгоритм Укконена построения суффиксного дерева.  | 83 |
| 20 Сжатие по повторяющимся подстрокам: LZ77 и LZ78 | 97 |
| 21 Преобразование Берроуза–Вилера (BWT), реализация | 100 |
| 22 Геометрические алгоритмы: ближайшая пара точек, выпуклая оболочка, наиболее удалённая пара точек | 103 |

0 Асимптотические обозначения

0.1 Большое O

Определение

Пусть $f(n)$ — это какая-то функция. Говорят, что функция $g(n) = O(f(n))$, если существуют константы $C > 0$ и N , такие что

$$g(n) \leq C \cdot f(n), \quad \forall n \geq N.$$

Это означает, что $g(n)$ растёт не быстрее, чем $f(n)$.

Примеры:

- $\frac{n}{3} = O(n)$
- $\frac{n(n-1)(n-2)}{6} = O(n^3)$
- $1 + 2 + 3 + \dots + n = O(n^2)$
- $\log_2 n + 3 = O(\log n)$
- $10^{100} = O(1)$

0.2 Большое Ω

Определение

Функция $g(n) = \Omega(f(n))$, если существуют константы $C > 0$ и N , такие что

$$g(n) \geq C \cdot f(n), \quad \forall n \geq N.$$

Это означает, что $g(n)$ растёт не медленнее, чем $f(n)$.

Примеры:

- $n^2 + 5n = \Omega(n^2)$
- $3n^3 + 2n = \Omega(n^3)$
- $2^n + n^2 = \Omega(2^n)$
- $n \log n = \Omega(n)$

0.3 Большое Θ

Определение

$g(n) = \Theta(f(n))$, если одновременно выполняется $g(n) = O(f(n))$ и $g(n) = \Omega(f(n))$. Это означает, что $g(n)$ растёт с тем же порядком, что и $f(n)$.

Примеры:

- $3n^2 + 10n + 7 = \Theta(n^2)$
- $5n \log n + 2n = \Theta(n \log n)$
- $7 \cdot 2^n + 3n^5 = \Theta(2^n)$
- $4n^3 - n = \Theta(n^3)$

0.4 Малое o

Определение

Функция $g(n) = o(f(n))$, если для любых $C > 0$ найдётся N , такое что

$$g(n) < C \cdot f(n), \quad \forall n \geq N.$$

Это означает, что $g(n)$ растёт строго медленнее, чем $f(n)$.

Примеры:

- $n = o(n^2)$
- $\log n = o(n)$
- $n^2 = o(2^n)$
- $\sqrt{n} = o(n)$

0.5 Малое ω

Определение

$g(n) = \omega(f(n))$, если для любых $C > 0$ найдётся N , такое что

$$g(n) > C \cdot f(n), \quad \forall n \geq N.$$

Это означает, что $g(n)$ растёт строго быстрее, чем $f(n)$.

Примеры:

- $n^2 = \omega(n)$
- $n \log n = \omega(n)$
- $2^n = \omega(n^3)$
- $n! = \omega(2^n)$

1 Метод «разделяй и властвуй». Быстрое умножение Кацаубы, анализ времени работы.

1.1 Метод «разделяй и властвуй»

«Разделяй и властвуй» (англ. *divide and conquer*) — это парадигма разработки алгоритмов, заключающаяся в рекурсивном разбиении задачи на более мелкие подзадачи до тех пор, пока они не станут элементарными, с последующим объединением их решений.

Структура метода:

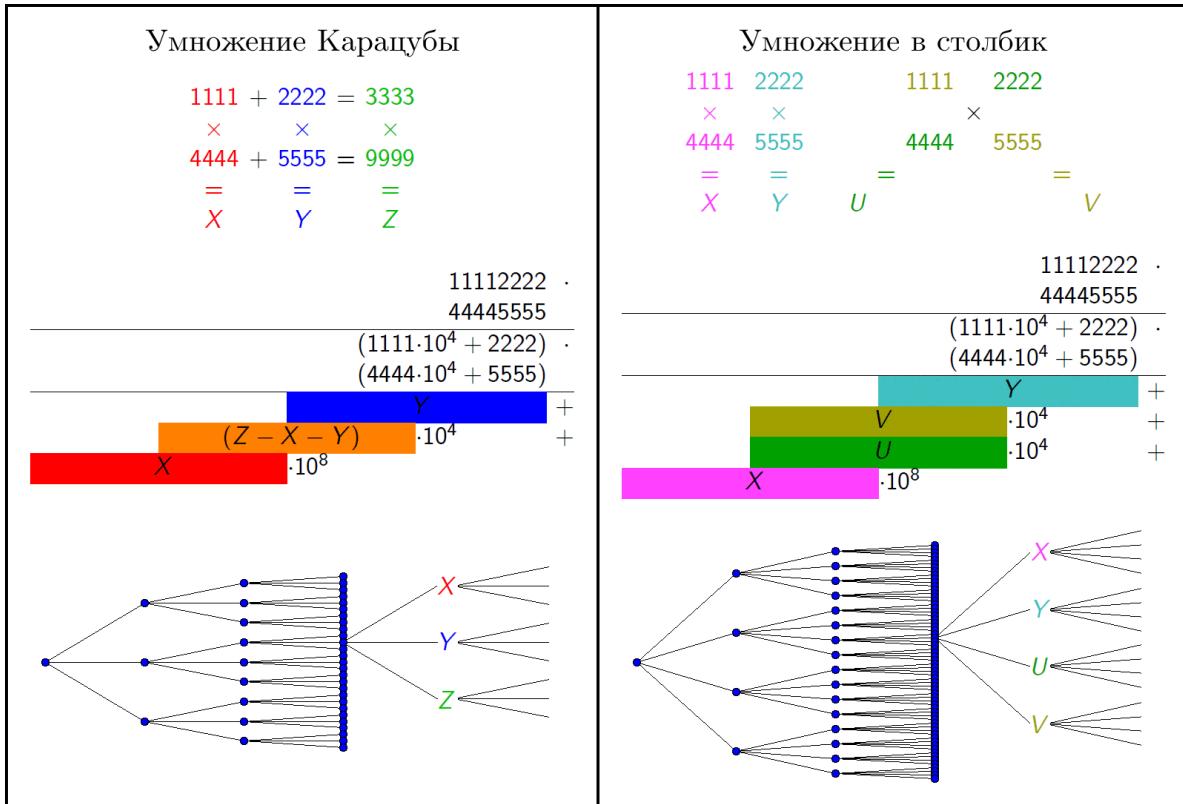
Алгоритм состоит из трёх основных шагов:

- Разделяй: разделение задачи на более мелкие подзадачи с помощью рекурсии.
- Властвуй: решение подзадач рекурсивно, когда они станут достаточно малы.
- Объединяй: комбинирование решений всех подзадач для получения решения исходной задачи.

1.2 Быстрое умножение Карацубы

$$\begin{array}{r}
 & a_1 & a_0 \\
 & b_1 & b_0 \\
 \hline
 & a_1 b_0 & a_0 b_0 \\
 a_1 b_1 & a_0 b_1 \\
 \hline
 a_1 b_1 & a_1 b_0 + a_0 b_1 & a_0 b_0
 \end{array}$$

- Первая идея:** Два двухзначных числа можно перемножить не за 4, а за 3 умножения. Произведения $a_1 b_1$ и $a_0 b_0$ вычисляются как обычно, а чтобы вычислить $a_1 b_0 + a_0 b_1$, перемножим $(a_1 + a_0)(b_1 + b_0) = a_1 b_1 + a_0 b_1 + a_1 b_0 + a_0 b_0$ и вычтем $a_1 b_1$ и $a_0 b_0$.
- Вторая идея:** Точно так же можно перемножить два n -разрядных числа, представленных в двоичной форме. Числа представляются в двоичной форме $a = a_1 \cdot 2^{\frac{n}{2}} + a_0$ и $b = b_1 \cdot 2^{\frac{n}{2}} + b_0$, $a_1, a_0, b_1, b_0 \in \{0, \dots, 2^{\frac{n}{2}} - 1\}$. Поскольку числа представлены в двоичной форме, не надо ничего делить, достаточно взять $\frac{n}{2}$ старших и $\frac{n}{2}$ младших разрядов их двоичной записи. И дальше всё то же самое, потребуется вычислить 3 произведения чисел длины не более чем $\frac{n}{2} + 1$ и $O(n)$ битовых сложений.
- Третья идея:** Для вычисления каждого из произведений чисел вдвое меньшей длины рекурсивно используется *тот же самый метод*. В итоге получится три рекурсивных вызова задач размерности $\frac{n}{2}$, то есть на каждом уровне рекурсии длина уменьшается вдвое, и потому глубина рекурсии – $\log_2 n$.



1.3 Анализ времени работы

Рекурсивный вызов образует дерево.

Общее время работы – это сумма времени выполнения во всех вершинах дерева.

| Размер | Сколько | Время |
|-----------------|----------|-----------------------------------|
| n | 1 | $C \cdot n$ |
| $\frac{n}{2}$ | 3 | $C \cdot \frac{n}{2} \cdot 3$ |
| $\frac{n}{4}$ | 9 | $C \cdot \frac{n}{4} \cdot 9$ |
| \vdots | \vdots | \vdots |
| $\frac{n}{2^i}$ | 3^i | $C \cdot \frac{n}{2^i} \cdot 3^i$ |

$$i = \log_2 n$$

Время работы для каждой подзадачи линейно относительно её размера и для подзадачи $\frac{n}{2^i}$ оно составляет $O(\frac{n}{2^i})$

$T(n)$ – время умножения чисел длины n

$T(n) = 3 \cdot T(\frac{n}{2}) + O(n)$, перенос и сумма выполняются за линейное время

Тогда общее время:

$$T(n) = \sum_{i=0}^{\log_2 n} 3^i \cdot \frac{n}{2^i} = n \sum_{i=0}^{\log_2 n} \left(\frac{3}{2}\right)^i = n \cdot \frac{\left(\frac{3}{2}\right)^{1+\log_2 n} - 1}{\frac{3}{2} - 1} = O(3^{\log_2 n}) = O(n^{\log_2 3})$$

2 Метод динамического программирования. Задача о разделении стержня. Задача о порядке умножения матриц.

2.1 Метод динамического программирования

Это подход, при котором сложные задачи разбивают на более простые подзадачи и сохраняют результаты решения этих подзадач для последующего использования.

Основная идея — избегать повторного вычисления результатов за счет сохранения промежуточных данных (мемоизации). Это позволяет сократить время выполнения задачи.

2.2 Задача о разделении стержня (Rod Cutting Problem)

Условие. Дан стержень длины n . Если разрезать его на куски длины i , то каждый кусок можно продать по цене p_i . Требуется определить, как разрезать стержень, чтобы получить максимальную выручку.

Идея. На каждом шаге мы должны принять решение:

- Какой длины кусок отрезать первым?
- Как потом оптимально разрезать оставшуюся часть?

Замечание.

- Рекурсивное решение возможно, но оно делает слишком много повторных вычислений.
- Жадный алгоритм (*всегда отрезать кусок с наибольшей текущей выгодой*) работает быстро, но не всегда оптимален.

Пример

Пусть $n = 6$, $p_1 = 0$, $p_2 = 2$, $p_3 = 7$, $p_4 = 10$.

Жадный алгоритм выберет кусок длины 4 (цена 10). Оставшийся кусок длины 2 принесет $p_2 = 2$, итого 12.

Оптимальный разрез: 3 + 3 (по $p_3 = 7$), итого 14.

Algorithm 1 Динамическое программирование для задачи о разделении стержня

Длина стержня n , цены на отрезки p_1, \dots, p_n

Для всех $j \in \{0, \dots, n\}$ вычислить T_j — наибольшую сумму, которую можно выручить, разрезав и продав стержень длины j .

```
1:  $T_0 = 0$                                 ▷ стержень длины 0 не стоит никакого
2: for  $j = 1$  to  $n$  do
3:    $T_j = 0$                                 ▷ пока не найдено никаких способов продать стержень
4:   for  $i = 1$  to  $j$  do                  ▷ для всякой длины отрезаемого куска
5:      $r = p_i + T_{j-i}$                       ▷ сложить его цену с выручкой за остаток
6:     if  $r > T_j$  then                  ▷ если так можно выручить больше известного
7:        $T_j = r$                           ▷ оценка цены стержня длины  $j$  улучшается
8: return  $T_n$                         ▷ столько можно выручить за весь стержень
```

Время работы: $O(n^2)$, так как для каждого j перебираем все $i \leq j$.

2.3 Задача о порядке умножения матриц (Matrix Chain Multiplication)

Условие. Даны n матриц M_1, M_2, \dots, M_n , где M_i имеет размер $m_{i-1} \times m_i$. Ассоциативность умножения позволяет расставлять скобки как угодно: результат один и тот же, но число операций зависит от порядка.

Чтобы перемножить матрицы быстрее, ставится задача заранее определить наилучший порядок их умножения.

Пример

Даны матрицы размера $A = 100 \times 2$, $B = 2 \times 100$, $C = 100 \times 3$. Если умножить $A(BC)$ потребуется $2 \cdot 100 \cdot 3 + 100 \cdot 2 \cdot 3 = 1200$ умножений и примерно столько же сложений. Если умножить $(AB)C$ потребуется $100 \cdot 2 \cdot 100 + 100 \cdot 100 \cdot 3 = 50000$ умножений

Решение методом динамического программирования

Идея: Построить верхнетреугольную матрицу T размера $n \times n$.

Где $T_{i,j}$ хранит минимальное число операций, необходимых для вычисления произведения $M_i \times \dots \times M_j$

Шаги алгоритма:

1. Если умножается одна матрица:

$i = j$, то $T_{i,j} = 0$, т.к. ничего вычислять не нужно.

2. Перебор длины поддиапазонов:

Для всех длин поддиапазонов $l = 2, 3, \dots, n$

l – количество матриц в текущем диапазоне, которое мы перемножаем

3. Перебор начального диапазона:

Для каждого начального индекса i (от 1 до $n - l$)

Вычисляем правую границу диапазона: $j = i + l - 1$

4. Перебор точки разбиения:

Для всех возможных точек разбиения k ($i \leq k \leq j$)

Разбиваем диапазон матрицы M_i, M_{i+1}, \dots, M_j на две группы:

Первая часть: от M_i до M_k

Вторая часть: от M_{k+1} до M_j

Считаем стоимость текущего разбиения

$T_{i,k} + T_{k+1,j} + m_{i-1} \cdot m_k \cdot m_j$, где $m_{i-1} \cdot m_k \cdot m_j$ – размеры матрицы, задающие стоимость умножения результирующих матриц.

Обновляем минимальную стоимость для текущего диапазона:

$$T_{i,j} = \min(T_{i,k} + T_{k+1,j} + m_{i-1} \cdot m_k \cdot m_j)$$

Algorithm 2 Нахождение наилучшего порядка умножения матриц и их умножение в соответствии с этим порядком

Вход: матрицы M_1, \dots, M_n , размер каждой $M_i = m_{i-1} \times m_i$

Нахождение числа действий

```
1: for  $i = 1$  to  $n$  do
2:    $T_{i,i} = 0$                                 ▷ произведение одной матрицы — считать нечего
3: for  $\ell = 2$  to  $n$  do
4:   for  $i = 1$  to  $n - \ell + 1$  do
5:      $j = i + \ell$                             ▷ правая граница
6:      $T_{i,j} = \infty$                       ▷ пока не найден ни один способ
7:     for  $k = i$  to  $j - 1$  do
8:        $t = T_{i,k} + T_{k+1,j} + m_{i-1} m_k m_j$     ▷ разрезаем между  $k$  и  $k + 1$ 
9:       if  $t < T_{i,j}$  then                  ▷ нашли более экономичный способ
10:       $T_{i,j} = t$ 
```

Произведение $M_i \cdots M_j$ восстанавливается процедурой:

Процедура умножить(i, j)

```
1: if  $i = j$  then
2:   return  $M_i$                                 ▷ одна матрица
3: for  $k = i$  to  $j - 1$  do
4:   if  $T_{i,j} = T_{i,k} + T_{k+1,j} + m_{i-1} m_k m_j$  then    ▷ для какого-то  $k$  равенство выполняется
5:     return умножить( $i, k$ ) × умножить( $k + 1, j$ )
```

Время работы.

$O(n^3)$, т.к. 3 вложенных цикла по длине куска l , начальной точки i и точки разбиения k .

Пример

Есть 4 матрицы

$M_1(10 \times 20) \quad M_2(20 \times 30) \quad M_3(30 \times 40) \quad M_4(40 \times 50)$

1. Инициализация таблицы:

Для одной матрицы $T_{i,i} = 0$

2. Когда $l = 2$ (перемножаем две матрицы)

Мы смотрим пары матриц:

$$M_1 M_2 \quad T_{1,2} = 10 \cdot 20 \cdot 30 = 6000$$

$$M_2 M_3 \quad T_{2,3} = 20 \cdot 30 \cdot 40 = 24000$$

$$M_3 M_4 \quad T_{3,4} = 30 \cdot 40 \cdot 50 = 60000$$

3. Когда $l = 3$ (перемножаем три матрицы)

Теперь перебираем начальные индексы i :

$$\text{Если } i = 1, \text{ то } j = i + l - 1 = 3 \quad (M_1 M_2 M_3)$$

$$\text{Если } i = 2, \text{ то } j = i + l - 1 = 4 \quad (M_2 M_3 M_4)$$

Для диапазона $(M_1 M_2 M_3)$:

Ищем k :

$$\text{Если } k = 1 ((M_1)(M_2 M_3))$$

$$T_{1,1} + T_{2,3} + (10 \cdot 20 \cdot 40) = 0 + 24000 + 8000 = 32000$$

$$\text{Если } k = 2 ((M_1 M_2)(M_3))$$

$$T_{1,2} + T_{3,3} + (10 \cdot 30 \cdot 40) = 6000 + 0 + 12000 = 18000$$

$$\text{Минимальная стоимость } T_{1,3} = \min(32000, 18000) = 18000$$

Для диапазона $(M_2 M_3 M_4)$:

Ищем k :

$$\text{Если } k = 2 ((M_2)(M_3 M_4))$$

$$T_{2,2} + T_{3,4} + (20 \cdot 30 \cdot 50) = 0 + 60000 + 30000 = 90000$$

$$\text{Если } k = 3 ((M_2 M_3)(M_4))$$

$$T_{2,3} + T_{4,4} + (20 \cdot 40 \cdot 50) = 24000 + 0 + 40000 = 64000$$

$$\text{Минимальная стоимость } T_{2,4} = \min(90000, 64000) = 64000$$

Когда $l = 4$ (весь набор $M_1 M_2 M_3 M_4$):

Ищем $k = 1, 2, 3$:

$$\text{Если } k = 1 ((M_1)(M_2 M_3 M_4))$$

$$T_{1,1} + T_{2,4} + (10 \cdot 20 \cdot 50) = 0 + 64000 + 10000 = 74000$$

$$\text{Если } k = 2 ((M_1 M_2)(M_3 M_4))$$

$$T_{1,2} + T_{3,4} + (10 \cdot 30 \cdot 50) = 6000 + 60000 + 15000 = 81000$$

$$\text{Если } k = 3 ((M_1 M_2 M_3)(M_4))$$

$$T_{1,3} + T_{4,4} + (10 \cdot 40 \cdot 50) = 18000 + 0 + 20000 = 38000$$

$$\text{Минимальная стоимость } T_{1,4} = \min(74000, 81000, 38000) = 38000$$

Итого: оптимальная скобочная структура и порядок умножения:

Поскольку $T_{1,4}$ достигается при $k = 3$, разбиваем

как $(M_1 M_2 M_3) M_4$.

Далее $T_{1,3}$ достигается при $k = 2$, значит $(M_1 M_2 M_3) = (M_1 M_2) M_3$.

Итого оптимально:

$$((M_1 M_2) M_3) M_4$$

Пошагово с затратами:

$$1) M_1 M_2: 10 \cdot 20 \cdot 30 = 6000$$

$$2) (M_1 M_2) M_3: 10 \cdot 30 \cdot 40 = 12000$$

$$3) ((M_1 M_2) M_3) M_4: 10 \cdot 40 \cdot 50 = 20000$$

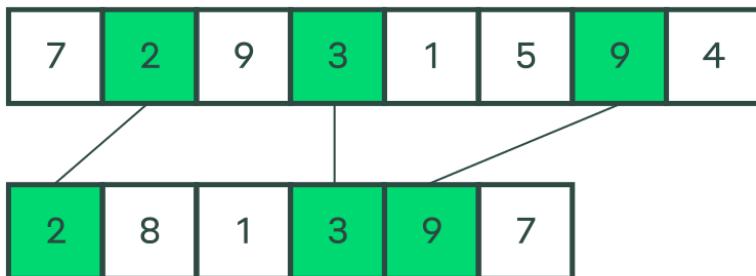
Итого 38 000

Таблица значений $T_{i,j}$ (в скалярных умножениях):

| $T_{i,j}$ | $j = 1$ | $j = 2$ | $j = 3$ | $j = 4$ |
|-----------|---------|---------|---------|---------|
| $i = 1$ | 0 | 6 000 | 18 000 | 38 000 |
| $i = 2$ | | 0 | 24 000 | 64 000 |
| $i = 3$ | | | 0 | 60 000 |
| $i = 4$ | | | | 0 |

3 Нахождение наибольшей общей подпоследовательности: «народный» алгоритм, алгоритм Хиршберга.

Задача. Сравнить две строки на похожесть.



Определение

Σ — Множество символов, называемое алфавитом.

$w = a_1 \dots a_l$ — Стока над алфавитом Σ , где $l \geq 0$, и $a_1, \dots, a_l \in \Sigma$ — символы.

$|w| = l$ — Длина строки обозначается так.

ε — Стока длины 0.

Σ^* — Множество всех строк над алфавитом.

Подпоследовательность строки — это строка, полученная удалением 0 или более символов из исходной строки при сохранении порядка оставшихся символов.

3.1 «Народный» алгоритм

Даны две строки a_1, \dots, a_m и b_1, \dots, b_n . Нужно найти наибольшую общую подпоследовательность (Longest Common Subsequence, LCS).

Подход к решению:

- Перебрать все варианты подпоследовательностей обеих строк — слишком долго (экспоненциальное время)
- Использовать метод динамического программирования, который позволяет эффективно найти LCS за $O(mn)$

Идея "народного" алгоритма:

1. Для любого префикса (начального куска) строки $u_i = a_1 \dots a_i$ и $v_j = b_1 \dots b_j$ хранит длину LCS в таблице $T_{i,j}$.
2. Заполняем таблицу $T_{i,j}$, начиная с коротких префиксов и постепенно увеличиваем их длину.

Как заполняется таблица $T_{i,j}$:

Базовый случай: Если $i = 0$ или $j = 0$, то $T_{i,j} = 0$, т.к. одна из строк пуста.

Общий случай:

(a) Если последние символы префиксов совпадают ($a_i = b_j$):

- Тогда LCS для u_i и v_j можно построить на основе LCS для более коротких префиксов u_{i-1} и v_{j-1} , добавив к ним текущий символ.

$$T_{i,j} = T_{i-1,j-1} + 1$$

(b) Если последние символы префиксов не совпадают ($a_i \neq b_j$):

Тогда последний символ одного из префиксов не входит в LCS.

У нас есть два варианта:

- Игнорировать последний символ строки a : тогда LCS определяется как LCS для u_{i-1} и v_j и длина равна $T_{i-1,j}$.
- Игнорировать последний символ строки b : тогда LCS определяется как LCS для u_i и v_{j-1} и длина равна $T_{i,j-1}$.

Для наибольшей общей подпоследовательности:

$$T_{i,j} = \max(T_{i-1,j}, T_{i,j-1})$$

Итоговая формула:

$$T_{i,j} = \begin{cases} T_{i-1,j-1} + 1, & \text{если } a_i = b_j, \\ \max(T_{i-1,j}, T_{i,j-1}), & \text{если } a_i \neq b_j. \end{cases}$$

Algorithm 3 Нахождение длины наибольшей общей подпоследовательности

На входе: строки $u = a_1 \dots a_m$ и $v = b_1 \dots b_n$.

Строится матрица T размера $(m+1) \times (n+1)$,

где $T_{i,j}$ — длина наибольшей общей подпоследовательности $u_i = a_1 \dots a_i$ и $v_j = b_1 \dots b_j$.

```

1: for  $j = 0$  to  $n$  do
2:    $T_{0,j} = 0$ 
3: for  $i = 1$  to  $m$  do
4:    $T_{i,0} = 0$ 
5:   for  $j = 1$  to  $n$  do
6:     if  $a_i = b_j$  then
7:        $T_{i,j} = T_{i-1,j-1} + 1$ 
8:     else
9:        $T_{i,j} = \max(T_{i-1,j}, T_{i,j-1})$ 
10: return  $T_{m,n}$ 
```

Пример

$$u = ABC, v = AC$$

| | | | |
|-----------|---|---|---|
| $T_{i,j}$ | — | A | C |
| — | 0 | 0 | 0 |
| A | 0 | 1 | 1 |
| B | 0 | 1 | 1 |
| C | 0 | 1 | 2 |

1. $T_{1,1} : a_1 = b_1 = A$ совпадает, поэтому
 $T_{1,1} = T_{0,0} + 1 = 1$
2. $T_{2,2} : a_2 = B, b_2 = C$ не совпадают, берем максимум:
 $T_{2,2} = \max(T_{1,2}, T_{2,1}) = 1$
3. $T_{3,2} : a_3 = b_2 = C : T_{3,2} = T_{2,1} + 1 = 2$

Достоинства:

Работает за $O(m \cdot n)$

Недостатки:

Требует $O(m \cdot n)$ памяти для хранения таблицы

Чтобы найти саму подпоследовательность:

Идем по таблице $T_{i,j}$ от $T_{m,n}$ к $T_{0,0}$

Если $a_i = b_j$ добавляем этот символ в LCS и идем в $T_{i-1,j-1}$

Если $T_{i-1,j} \geq T_{i,j-1}$: идем вверх ($i - 1$)

Иначе идем влево: ($j - 1$)

Algorithm 4 Построение наибольшей общей подпоследовательности

На входе: строки $u = a_1 \dots a_m$ и $v = b_1 \dots b_n$, матрица T размера $(m + 1) \times (n + 1)$, где $T_{i,j}$ — длина наибольшей общей подпоследовательности $u_i = a_1 \dots a_i$ и $v_j = b_1 \dots b_j$. Построить одну из подпоследовательностей u и v наибольшей длины.

```
1:  $i = m$ 
2:  $j = n$ 
3:  $w = \varepsilon$ 
4: while  $i > 0$  и  $j > 0$  do
5:   if  $a_i = b_j$  then
6:      $w = a_i w$ 
7:      $i = i - 1$ 
8:      $j = j - 1$ 
9:   else if  $T_{i-1,j} > T_{i,j-1}$  then
10:     $i = i - 1$ 
11:   else
12:      $j = j - 1$ 
13: return  $w$ 
```

3.2 Алгоритм Хиршберга

Построение наибольшей общей подпоследовательности за время $O(m \cdot n)$, используя память $O(\min(m, n))$, где $m = |u|, n = |v|$

Идея алгоритма:

1. Деление строки u :

Строчку u делим пополам: $u = u_1 u_2$, где $|u_1| = \lfloor \frac{m}{2} \rfloor$, $|u_2| = \lceil \frac{m}{2} \rceil$.

2. Оптимальное разбиение строки v :

Строчку v нужно разбить на две части $v = v'v''$, чтобы можно было отдельно работать с u_1 и u_2 .

Посчитаем LCS для строки u_1 и строки v :

Используем динамическое программирование.

- Итоговая строка таблицы $T^{u_1,v}$ содержит длину LCS для всех префиксов строки v :

Если $T_j^{u_1,v} = k$, то это означает, что длина LCS между u_1 и $v_{[1:j]}$ равна k .

- Посчитаем LCS для строки u_2 и v :

Теперь считаем LCS для строк u_2 и v , но в обратном порядке.

Строчки u_2 и v перевернуты, и снова используем динамическое программирование.

- Последняя строка таблицы также будет содержать длину LCS для всех суффиксов строки v :

Если $T_j^{u_2, v} = k$, то это означает, что длина LCS между u_2 и суффиксом $v_{[j+1:n]}$ равна k .

3. Суммируем результаты для всех разбиений:

Для каждого j (позиции в строке v), считаем:

$$LCS(u_1, v_{[1:j]}) + LCS(u_2, v_{[j+1:n]})$$

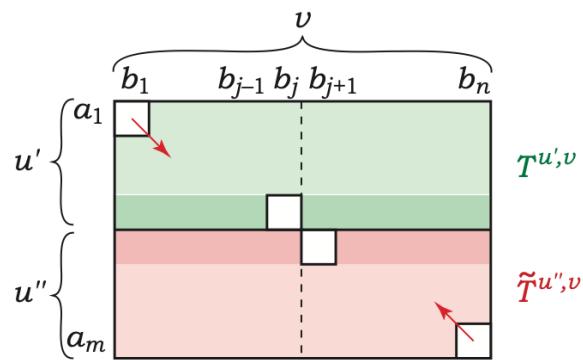
Эта сумма:

Длины LCS первой половины строки u_1 и префикса строки $v_{[1:j]}$

Длины LCS второй половины строки u_2 и суффикса строки $v_{[j+1:n]}$

4. Выбираем j , при котором сумма максимальна.

Преимущества: Линейная память $O(\min(m, n))$, поскольку при каждом шаге используется только последняя строка таблицы.



Лемма

Алгоритм Хиршберга работает за $O(mn)$.

Доказательство. Пусть $T(m, n)$ – время работы алгоритма для строк длиной m и n .

Покажем, что для всех m, n выполняется $T(m, n) \leq 2mn$.

При каждой рекурсии алгоритм заполняет две таблицы динамического программирования размера примерно $m \times n$ (одну вперёд, одну назад) за $O(mn)$ операций. После этого он делит строку длины m пополам (в позиции $\lceil \frac{m}{2} \rceil$) и рекурсивно решает задачу для двух половин: строк длины $\lfloor \frac{m}{2} \rfloor$ и $\lceil \frac{m}{2} \rceil$ и соответствующих частей второй строки (суммарная длина второго остаётся n). Таким образом получаем рекуррентное соотношение вида:

$$T(m, n) \leq C \cdot mn + T\left(\left\lfloor \frac{m}{2} \right\rfloor, k\right) + T\left(\left\lceil \frac{m}{2} \right\rceil, n - k\right),$$

где C – некоторая константа, а k – индекс разбиения второй строки.

По индукции:

База. Если $m = 0$ или $n = 0$, алгоритм сразу завершает работу, и $T(m, n) = 0 \leq 2mn$. Если один из параметров равен 1, например $m = 1$, алгоритм сравнивает одну букву со всеми n буквами другой строки за $O(n) \leq 2mn$. Таким образом для $m \leq 1$ или $n \leq 1$ неравенство $T(m, n) \leq 2mn$ очевидно.

Предположение. Пусть для всех пар длин (m', n') с $m' + n' < m + n$ верно $T(m', n') \leq 2m'n'$. Мы рассматриваем случай $m, n \geq 2$.

Переход. По рекурренте имеем $T(m, n) \leq Cmn + T\left(\frac{m}{2}, k\right) + T\left(\frac{m}{2}, n - k\right)$.

Применяя индукционное предположение к обоим рекурсивным вызовам (каждый из них на строках меньшей суммарной длины), получаем

$$T\left(\frac{m}{2}, k\right) \leq 2 \cdot \frac{m}{2} k = mk,$$

$$T\left(\frac{m}{2}, n - k\right) \leq 2 \cdot \frac{m}{2} (n - k) = m(n - k).$$

$$\text{Отсюда } T(m, n) \leq C mn + mk + m(n - k) = C mn + mn = (C + 1) mn.$$

Если в исходном определении считать константу $C = 1$ (или включить её в оценки), то получаем $T(m, n) \leq 2mn$. Более строго, как показывается в литературе, можно выбрать параметр $C_0 = 2C$, тогда неравенство превращается в $(C + C_0/2)mn = 2C mn$. Так или иначе, мы выводим, что $T(m, n) \leq 2mn$, что завершает индуктивное доказательство. \square

Таким образом, по принципу полной индукции доказано, что в худшем случае $T(m, n) \leq 2mn$. Это даёт асимптотику $O(mn)$.

4 Сортировка вставками, сортировка слиянием, куча и сортировка кучей

4.1 Сортировка вставкой

Сортировка вставками (Insertion Sort) — это простой алгоритм сортировки.

Главная мысль: сперва отсортировать все элементы с 1-го по $(i - 1)$ -й, потом вставить i -й элемент на своё место среди элементов с 1-го по $(i - 1)$ -й. Для этого нужно сдвинуть на одну позицию вперёд все элементы между правильным местом и i -м местом. И так далее, для всех i от 2 до n .

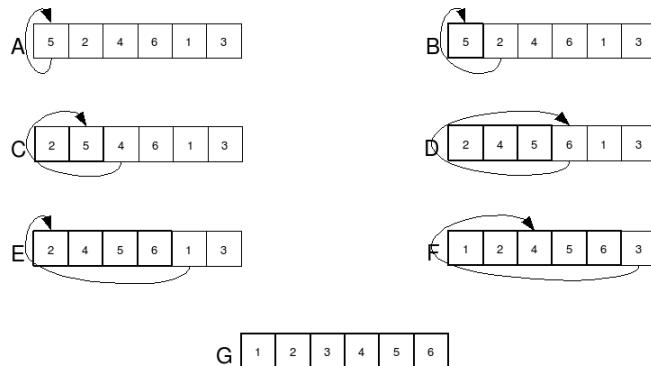
Массив из одного элемента считается уже отсортированным.

Algorithm 5 Сортировка вставкой

```

1: for  $j = 2$  to  $n$  do
2:    $t = x_j$ 
3:    $i = j - 1$ 
4:   while  $i \geq 1$  и  $x_i > t$  do
5:      $x_{i+1} = x_i$ 
6:      $i = i - 1$ 
7:    $x_{i+1} = t$ 

```



Порядок работы алгоритма:

1. Начинаем с элемента на позиции 2.
2. Сравниваем текущий элемент с предыдущим.
3. Если текущий элемент меньше предыдущего, перемещаем его влево, пока не найдём подходящее место.
4. Продолжаем процесс для всех последующих элементов.

Лемма

Алгоритм сортировки вставкой правильно сортирует массив, полученный на входе.

Доказательство. Докажем утверждение по индукции, используя следующий **инвариант внешнего цикла**:

После итерации j текущие значения x_1, \dots, x_j образуют отсортированную последовательность элементов исходного массива.

База индукции. Для $j = 1$ утверждение верно, поскольку первый элемент сам по себе отсортирован.

Переход. Предположим, что в начале итерации j элементы x_1, \dots, x_{j-1} уже отсортированы (предположение индукции). Тело цикла вставляет элемент x_j в нужное место, сохраняя упорядоченность. Следовательно, после завершения итерации элементы x_1, \dots, x_j также будут отсортированы.

Заключение. Для $j = n$ (по окончании работы алгоритма) все элементы x_1, \dots, x_n будут отсортированы. Тем самым алгоритм сортирует массив корректно. \square

Асимптотика. Внешний цикл выполняется $n - 1$ раз, и на каждой итерации совершается не более n действий. Следовательно, время работы алгоритма: $O(n^2)$.

Оценка в худшем случае. Обозначим $T(w)$ — время работы алгоритма на входе w длины n . В худшем случае: $T(n) = \max_{|w|=n} T(w)$.

Для сортировки вставкой время работы в худшем случае имеет порядок: $T(n) = \Theta(n^2)$.

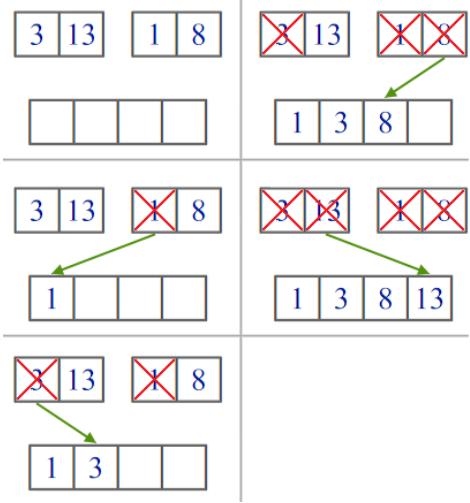
Такой случай достигается, например, на массиве, отсортированном по убыванию. На удачно подобранных (почти отсортированных) входных данных алгоритм может работать быстрее.

4.2 Сортировка слиянием

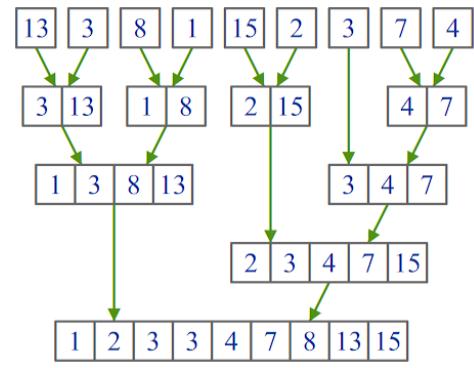
Дан массив a_1, a_2, \dots, a_n из целых чисел. Требуется упорядочить его по неубыванию: $a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}$.

Алгоритм использует принцип «разделяй и властвуй»:

1. Если в массиве один элемент — он уже отсортирован.
2. Иначе массив разбивается на две части и они сортируются рекурсивно.
3. После сортировки применяется процедура *слияния*, которая объединяет две отсортированные части в один отсортированный массив.



(a) Пример работы процедуры слияния.



(b) Пример работы рекурсивного алгоритма сортировки слиянием

Algorithm 6 Сортировка слиянием (фон Нейман)

Дано: массив a_1, a_2, \dots, a_n . **Цель:** упорядочить его по возрастанию.

Процедура $\text{merge_sort}(\ell, m)$

▷ Включительно слева и исключительно справа

```

1: if  $m - \ell \geq 2$  then
2:    $\text{merge\_sort}(\ell, \lfloor \frac{\ell+m}{2} \rfloor)$            ▷ сортировка первой половины
3:    $\text{merge\_sort}(\lfloor \frac{\ell+m}{2} \rfloor, m)$        ▷ сортировка второй половины
4:    $\text{merge}(\ell, \lfloor \frac{\ell+m}{2} \rfloor, m)$         ▷ слияние

```

Процедура $\text{merge}(\ell, mid, m)$

```

1:  $L = a[\ell \dots \lfloor \frac{\ell+m}{2} \rfloor - 1]$       ▷ Копируем левую отсортированную половину: индексы от  $\ell$  до
    $\lfloor \frac{\ell+m}{2} \rfloor - 1$  (включительно).
2:  $R = a[\lfloor \frac{\ell+m}{2} \rfloor \dots m - 1]$ 
3:  $i = 0, j = 0, k = \ell$                       ▷  $i, j$  — указатели в  $L$  и  $R$ ;  $k$  — позиция в итоговом массиве  $a$ 
   (начинаем с  $\ell$ ).
4: while  $i < |L|$  и  $j < |R|$  do
5:   if  $L[i] \leq R[j]$  then          ▷ Сравниваем первые невставленные элементы двух частей.
6:      $a[k] = L[i]$ 
7:      $i = i + 1$ 
8:   else
9:      $a[k] = R[j]$ 
10:     $j = j + 1$ 
11:     $k = k + 1$                   ▷ Сдвигаем позицию записи вправо в массиве  $a$ 
12: while  $i < |L|$  do          ▷ Если после основного цикла в  $L$  остались элементы
13:    $a[k] = L[i]$ 
14:    $i = i + 1, k = k + 1$         ▷ копируем их подряд в  $a$  (они уже отсортированы).
15: while  $j < |R|$  do          ▷ Если в  $R$  остались элементы
16:    $a[k] = R[j]$ 
17:    $j = j + 1, k = k + 1$         ▷ ...копируем их в  $a$ 

```

Время работы

Процедура merge работает за $\Theta(m - \ell)$, так как выполняется $m - \ell$ итераций, каждая за $O(1)$.

Время работы всего алгоритма:

Глубина рекурсии – $\log_2 n$

Задач размера n – одна

Задач размера $\frac{n}{2}$ – две

Задач размера $\frac{n}{2^i} = 2^i$

Всего:

$$\sum_{i=0}^{\log_2 n} 2^i \frac{n}{2^i} = n \log_2 n$$

4.3 Куча

Куча (heap) — почти сбалансированное двоичное дерево, представленное в виде массива, с условием:

$$x_i \leq x_{2i}, \quad x_i \leq x_{2i+1} \quad (\text{min-heap})$$

Для *max-heap* знак неравенства меняется.

Вершины на каждом уровне упорядочены.

Дерево почти сбалансировано: на каждом i -м уровне, кроме, может быть, последнего, есть все 2^i вершин, а на последнем уровне есть сколько-то самых левых элементов.

Свойства:

- Корень — вершина 1.
- Родитель вершины i : $[i/2]$.
- Левый потомок: $2i$.
- Правый потомок: $2i + 1$.

Превращение массива в кучу

Массив рассматривается в качестве кучи, в которой условие кучи нарушено в каких-то вершинах. Затем эта неправильная куча исправляется: индукцией по высоте поддерева последовательно обеспечивается, что каждое поддерево — правильная куча.

Каждый лист — уже сам по себе куча, ничего исправлять не надо.

Если у какой-то внутренней вершины i оба потомка — кучи, то вызывается процедура *heapify()* с аргументом i , делающая кучу из всего этого поддерева: элемент i «утопляет» до своего законного места.

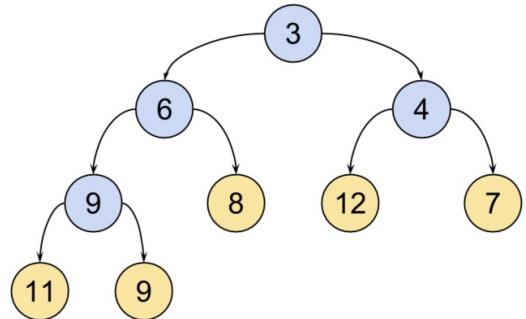


Рис. Пример кучи для минимума

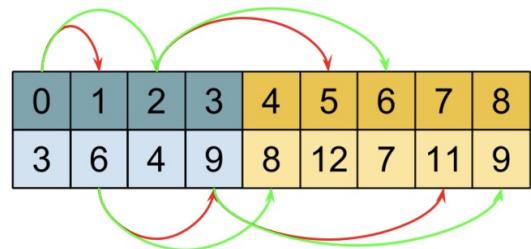


Рис. Хранение кучи в массиве, красная стрелка — левый сын, зеленая — правый

Algorithm 7 Преобразование массива в кучу

Процедура `heapify(i)`

```
1: while значение в i больше минимального потомка do
2:   j — индекс наименьшего потомка i
3:   обменять  $x_i$  и  $x_j$ 
4:   i = j
```

Процедура `make_heap()`

```
1: for i =  $\lfloor \frac{n}{2} \rfloor$  downto 1 do
2:   heapify(i)
```

Оценка времени работы процедуры make_heap()

Грубая верхняя оценка: если считать, что `heapify()` вызывается для $\Theta(n)$ узлов и каждый вызов в худшем случае работает за $O(\log n)$, то получаем $O(n \log n)$. Но это слишком грубо: большинство узлов находятся близко к листьям, и для них `heapify()` выполняется значительно быстрее.

Рассмотрим процедуру построения кучи:

1. Стоимость одной операции heapify

Пусть вершина находится на высоте h (расстояние до листа). В худшем случае элемент может опуститься по дереву до самого низа, поэтому

$$T(i) = O(h).$$

2. Сколько вершин имеют высоту h

Для бинарного дерева:

- на высоте 0 (листи) — $\approx \frac{n}{2}$ вершин
- на высоте 1 — $\approx \frac{n}{4}$
- на высоте 2 — $\approx \frac{n}{8}$
- ...
- на высоте h — $\approx \frac{n}{2^{h+1}}$

3. Суммарная стоимость процедуры make_heap

Просуммируем трудоёмкость по всем уровням:

$$T = \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left(\frac{n}{2^{h+1}} \cdot O(h) \right).$$

Вынесем n :

$$T = O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h}\right).$$

4. Оценка суммы

Известно, что сходящийся ряд

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2.$$

Следовательно,

$$T = O(n \cdot 2) = O(n).$$

5. Итог

Время работы процедуры `make_heap` составляет $\Theta(n)$.

Таким образом, построение кучи выполняется за линейное время, несмотря на то, что операция `heapify` в худшем случае занимает $O(\log n)$.

4.4 Сортировка кучей

Сортировка кучей: сперва построить кучу за линейное время, а потом на каждом шаге приписывать x_1 к сортированному массиву и удалять его из кучи, заменяя x_1 на x_n и вызывая `heapify(1)` — это займёт логарифмическое время для каждого элемента, так что общее время работы будет $n \log n$.

5 «Быстрая сортировка», ожидаемое время работы при случайному выборе опорного элемента.

Быстрая сортировка использует метод «разделяй и властвуй», как и сортировка слиянием, но деление на подзадачи зависит от опорного элемента (*pivot*).

Идея алгоритма:

1. Выбирается опорный элемент (*pivot*), может быть выбран случайно или по определенному правилу.

2. Массив делится на две части:

Левая часть содержит элементы, меньше или равные опорному.

Правая часть содержит элементы, больше или равные опорному. (Элементы обмениваются за линейное время)

3. Обе части массива сортируются рекурсивно тем же алгоритмом.

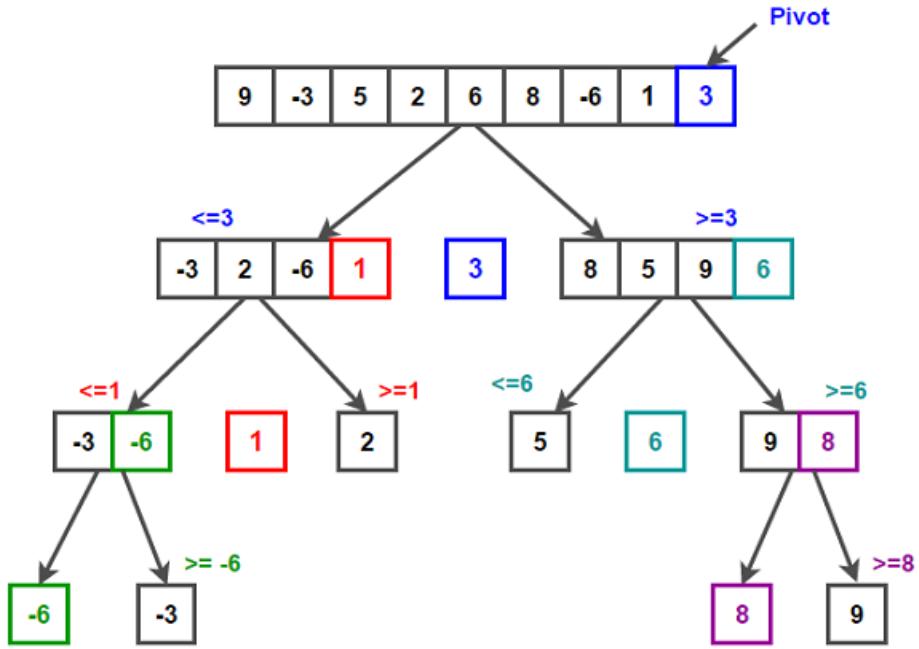
- Опорный элемент может выбираться произвольно, в том числе случайно.
- При удачном выборе *pivot* массив делится примерно пополам, сложность $O(n \log n)$.
- При неудачном выборе — $O(n^2)$.

Разделение с применением дополнительной памяти:

Делается проход по исходному массиву, каждый элемент сравнивается с опорным и размещается в левой или правой части нового массива того же размера, потом новый массив переписывается поверх старого.

Алгоритм Хоара (разделение без дополнительной памяти)

С разных концов массива запускаются два указателя: левый движется вправо, пропуская элементы меньше опорного значения, правый движется влево, пропуская элементы больше опорного значения. Когда оба указателя останавливаются на элементах, стоящих «не в своей половине», эти элементы меняются местами, и движение продолжается. Когда указатели пересекаются ($i \geq j$), процедура завершает работу и возвращает индекс j как границу разбиения.



Algorithm 8 Быстрая сортировка

Процедура $\text{quicksort}(\ell, m)$

```

1: if  $\ell < m$  then
2:    $j = \text{partition}(\ell, m)$ 
3:    $\text{quicksort}(\ell, j)$ 
4:    $\text{quicksort}(j + 1, m)$ 

```

Процедура $\text{partition}(\ell, m)$

```

1: случайно выбрать  $p \in \{\ell, \dots, m\}$                                 ▷ можно выбрать как угодно
2:  $a = x_p$                                                                ▷ запомнить значение опорного элемента
3: if  $p = m$  then
4:    $x_m \leftrightarrow x_{m-1}$                                               ▷ чтобы опорный не был последним
5:  $i = \ell - 1$                                                         ▷ левый указатель, -1 чтобы в repeat не выйти за границы
6:  $j = m + 1$                                                         ▷ правый указатель, +1 чтобы в repeat не выйти за границы
7: loop
8: repeat
9:    $i = i + 1$ 
10: until  $x_i \geq a$                                                  ▷ пока не найдём элемент, не меньший опорного значения  $a$ 
11: repeat
12:    $j = j - 1$ 
13: until  $x_j \leq a$ 
14: if  $i \geq j$  then
15:   return  $j$                                                        ▷ здесь и дальше  $\geq a$ , до этого места —  $\leq a$ 
16:  $x_i \leftrightarrow x_j$ 

```

Лемма (Разделение Хоара)

Процедура разделения Хоара работает корректно. А именно:

1. Указатели, войдя в отрезок массива $[\ell, r]$ на первой итерации внешнего цикла, не выходят за границы массива.
2. В результате получается разбиение массива на два непустых куска: существует индекс $j \in \{\ell, \dots, r - 1\}$, такой что

$$a_\ell, \dots, a_j \leq a_{j+1}, \dots, a_r.$$

Кроме того, в момент завершения процедуры выполняется одно из условий:

$$j = i \quad \text{или} \quad j = i - 1.$$

Доказательство. В начале каждой итерации внешнего цикла выполняются два инварианта:

- все элементы, до которых доехал левый указатель i — то есть все элементы a_ℓ, \dots, a_i меньше либо равны опорному;
- все элементы, до которых доехал правый указатель j — то есть все элементы a_j, \dots, a_r больше либо равны опорному.

После выполнения внутренних циклов указатели i и j останавливаются на элементах, которые либо равны опорному, либо находятся не в «своей» половине. Тогда:

- все элементы, которые проехал левый указатель i — то есть все элементы a_ℓ, \dots, a_{i-1} не превосходят опорного, а a_i — больше или равен ему;
- все элементы, которые проехал правый указатель j — то есть все элементы a_{j+1}, \dots, a_r больше или равны опорному, а a_j — меньше или равен ему.

После обмена a_i и a_j второе условие приводится к первому. Если при этом $i < j$, процесс продолжается; если $i \geq j$, то цикл завершается.

Случай $i = j$. Элементы a_ℓ, \dots, a_{i-1} не больше опорного, а a_i — первый элемент правой части. Тогда $j = i$, и условие выполнено.

Случай $i > j$. Правый указатель не может проехать дальше $i - 1$, значит $j = i - 1$. Элементы a_ℓ, \dots, a_j не больше опорного, и условие также выполнено.

Непустота частей. Опорный элемент не выбирается последним: на первой итерации левый указатель не доходит до конца, а правый — не уходит левее $r - 1$. Следовательно, обе части содержат хотя бы один элемент.

□

«Быстрая сортировка» — неустойчивая, то есть не сохраняет порядок одинаковых элементов.

5.1 Анализ времени работы при случайному выборе pivot

При случайному выборе pivot алгоритм является вероятностным. Ожидаемое время работы: $n \log n$

В худшем случае — $O(n^2)$, в лучшем — $O(n \log n)$.

Математическое ожидание времени работы алгоритма

Пусть (x_1, x_2, \dots, x_n) – входной массив

π – одно из возможных вычислений алгоритма

$|\pi|$ – количество шагов в этом вычислении

$Pr(\pi)$ – вероятность того, что из всех вычислений произойдет это.

Тогда математическое ожидание времени работы алгоритма

$$\sum_{\pi - \text{вычислений на } (x_1, \dots, x_n)} Pr(\pi) \cdot |\pi|$$

Теорема

Если все элементы массива различны и опорный элемент каждый раз выбирается случайно, то для любого входного массива математическое ожидание времени работы составит: $\Theta(n \log n)$

Доказательство. Время работы пропорционально числу сравнений между элементами. Пусть y_i и y_j – элементы отсортированного массива. Сколько раз они могут сравниваться за время работы алгоритма?

На нулевом уровне рекурсии y_i и y_j , и все элементы y_{i+1}, \dots, y_{j-1} между ними находятся в одном и том же куске. Если теперь в качестве опорного элемента выбирается элемент, меньший, чем y_i , или больший, чем y_j , то на следующем уровне рекурсии все элементы y_i, y_{i+1}, \dots, y_j остаются в одном куске. В какой-то момент один из этих элементов выбирается в качестве опорного. Если это будет один из промежуточных элементов y_k , где $i < k < j$, то y_i попадёт в левую половину, y_j в правую, и потому сравнение между ними никогда не будет.

Если же за опорный элемент будет взят y_i или y_j , то алгоритм сравнивает их один раз. Поскольку выбор одного из элементов y_i, y_{i+1}, \dots, y_j равновероятен, это будет y_i или y_j с вероятностью $\frac{2}{j-i+1}$, и потому ожидаемое число сравнений между этими двумя элементами – это $\frac{2}{j-i+1}$.

Ожидаемое число сравнений:

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

На первом шаге замена переменных $k = j - i$, на втором – оценка $\sum_{k=1}^n \frac{1}{k} \approx \ln n$. \square

6 Нижняя оценка числа сравнений при сортировке. Сортировка подсчётом. Поразрядная сортировка.

Теорема

Всякий алгоритм сортировки, основанный на сравнениях, на входе длины n в худшем случае производит не менее $\Omega(n \log n)$ сравнений.

Доказательство. Любому алгоритму сортировки сравнениями можно сопоставить дерево. Необходимо доказать, что высота такого дерева для любого алгоритма сортировки сравнениями не меньше, чем $\Omega(n \log n)$, где n – количество элементов.

При сравнении некоторых двух из них, существует два возможных исхода ($a_i \leq a_j$ и $a_i > a_j$), значит, каждый узел дерева имеет не более двух сыновей. Всего существует $n!$ различных перестановок n элементов, значит, число листьев нашего дерева не менее $n!$ (иначе некоторые перестановки были бы не достижимы из корня, а, значит, алгоритм неправильно работал бы).

Докажем, что двоичное дерево с не менее чем $n!$ листьями имеет глубину $\Omega(n \log n)$. Двоичное дерево высоты h имеет не более чем 2^h листьев. Значит, имеем неравенство $n! \leq \ell \leq 2^h$, где ℓ — число листьев. Прологарифмировав его, получим:

$$h \geq \log_2(n!) = \log_2 1 + \log_2 2 + \dots + \log_2 n > \frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \frac{n}{2}(\log_2 n - 1) = \Omega(n \log n)$$

Итак, для любого алгоритма сортировки сравнениями, существует такая перестановка, на которой он выполнит $\Omega(n \log n)$ сравнений. \square

6.1 Сортировка подсчетом

Алгоритм основанный не на сравнении элементов

Дан массив x_1, \dots, x_n , где $x_i \in \{1, \dots, k\}$ для достаточно малого k .

1. Создаём вспомогательный массив счётчиков

Создаём массив $c[1..k]$ и заполняем его нулями. Этот массив будет считать, сколько элементов с каждым значением встречается в исходном массиве.

2. Проходим по исходному массиву и заполняем счётчики

Для каждого элемента x_i увеличиваем счётчик для его значения на единицу:

$$c[x_i] := c[x_i] + 1$$

После этого в $c[m]$ хранится количество элементов, равных m .

3. Преобразуем счётчики в позиции (кумулятивная сумма)

Суммируем счётчики слева направо, чтобы получить информацию о том, где кончается каждое значение в выходном массиве:

Для каждого i от 2 до k :

$$c[i] := c[i] + c[i - 1]$$

Теперь $c[m]$ содержит позицию, куда попадёт **последний** элемент со значением m .

4. Заполняем результирующий массив справа налево

Проходим по исходному массиву **от конца к началу** (это важно для стабильности):

Для каждого элемента x_i (где i идёт от n до 1):

- Получаем текущее значение: $m := x_i$
- Берём позицию для этого значения из $c[m]$: $j := c[m]$
- Помещаем элемент в результирующий массив: $\text{result}[j] := m$
- Уменьшаем счётчик: $c[m] := c[m] - 1$

5. Возвращаем отсортированный массив

Массив result содержит элементы, отсортированные по возрастанию, с сохранением порядка равных элементов.

Пример

Предположим, исходный массив: $x = [3, 1, 2, 1, 3, 2]$, $n = 6$, $k = 3$.

После пункта 2 (простые счётчики):

$c[1] = 2$ (две единицы)

$c[2] = 2$ (две двойки)

$c[3] = 2$ (две тройки)

Выходной массив должен быть отсортирован: $[1, 1, 2, 2, 3, 3]$.

Суммируем счётчики накопительно слева направо:

| | | | |
|-------------|------------|--------------------|--------------------|
| Было: | $c[1] = 2$ | $c[2] = 2$ | $c[3] = 2$ |
| В пункте 3: | $c[1] = 2$ | $c[2] = 2 + 2 = 4$ | $c[3] = 4 + 2 = 6$ |

Теперь $c[m]$ содержит **последнюю позицию**, где должен находиться элемент со значением m .

На Шаге 4 проходим по исходному массиву от $i = 6$ до $i = 1$.

Итерация 1: $i = 6$

$$x_6 = 2$$

- Значение: $m := 2$;
- Позиция: $j := c[2] = 4$;
- Размещение: $\text{result}[4] := 2$;
- Декремент: $c[2] := 3$.

Состояние: $c = [2, 3, 6]$, $\text{result} = [-, -, -, 2, -, -]$.

Итерация 2: $i = 5$

$$x_5 = 3 \quad (\text{переходим к другому числу})$$

- Значение: $m := 3$;
- Позиция: $j := c[3] = 6$;
- Размещение: $\text{result}[6] := 3$;
- Декремент: $c[3] := 5$.

Состояние: $c = [2, 3, 5]$, $\text{result} = [-, -, -, 2, -, 3]$.

:

Итерация 6: $i = 1$

$$x_1 = 3$$

- Значение: $m := 3$;
- Позиция: $j := c[3] = 5 \leftarrow \text{но уже другая позиция!}$;
- Размещение: $\text{result}[5] := 3$;
- Декремент: $c[3] := 4$.

Состояние: $c = [0, 2, 4]$, $\text{result} = [1, 1, 2, 2, 3, 3]$.

Algorithm 8 Сортировка подсчётом (Counting Sort)

```
1: Создать вспомогательный массив  $c[1 \dots k]$  и заполнить его нулями
2: for  $i := 1$  to  $n$  do
3:    $c[x_i] := c[x_i] + 1$            /* подсчитываем количество каждого значения */
4: for  $i := 2$  to  $k$  do
5:    $c[i] := c[i] + c[i - 1]$        /* преобразуем в кумулятивные позиции */
6: Создать результирующий массив  $result[1 \dots n]$ 
7: for  $i := n$  downto 1 do
8:    $m := x_i$                    /* текущее значение */
9:    $j := c[m]$                  /* позиция для этого значения */
10:   $result[j] := m$             /* помещаем элемент */
11:   $c[m] := c[m] - 1$         /* уменьшаем счётчик */
12: return  $result[1 \dots n]$     /* возвращаем отсортированный массив */
```

Время работы: $O(n + k)$

6.2 Поразрядная сортировка

Сначала делим данные по разрядам, а потом сортируем внутри каждого разряда. Для этого нужно знать, сколько элементов в массиве и сколько разрядов у самого длинного элемента.

1. Узнать количество элементов массива и определить, сколько разрядов у самого длинного числа.
2. Создать промежуточный массив размера, равного основанию системы счисления, с пустыми ячейками.
3. Найти все значения первого разряда у каждого числа и поместить соответствующие числа во вспомогательный массив в ячейки с этими номерами.
4. Заменить содержимое исходного массива непустыми значениями из вспомогательного массива.
5. Повторять шаги, пока не будут обработаны все разряды.

Пример

Исходный массив:

$$A = [225, 145, 244, 125, 143, 123, 243]$$

Сортировка по последнему разряду:

Единицы: 225 (5), 145 (5), 244 (4), 125 (5), 143 (3), 123 (3), 243 (3)

Распределение по корзинам:

- Корзина 3: 143, 123, 243
- Корзина 4: 244
- Корзина 5: 225, 145, 125

Новый массив после сортировки по единицам:

$$A_1 = [143, 123, 243, 244, 225, 145, 125]$$

Сортировка по десяткам:

Десятки: 143 (4), 123 (2), 243 (4), 244 (4), 225 (2), 145 (4), 125 (2)

Распределение по корзинам:

- Корзина 2: 123, 225, 125
- Корзина 4: 143, 243, 244, 145

Новый массив после сортировки по десяткам:

$$A_2 = [123, 225, 125, 143, 243, 244, 145]$$

Сортировка по сотням:

Сотни: 123 (1), 225 (2), 125 (1), 143 (1), 243 (2), 244 (2), 145 (1)

Распределение по корзинам:

- Корзина 1: 123, 125, 143, 145
- Корзина 2: 225, 243, 244

Новый массив после сортировки по сотням:

$$A_3 = [123, 125, 143, 145, 225, 243, 244]$$

Время работы: $O(m(n + k))$

m – количество разрядов

n – количество объектов

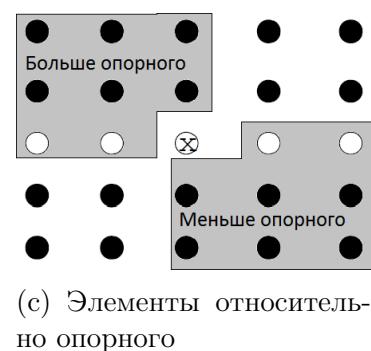
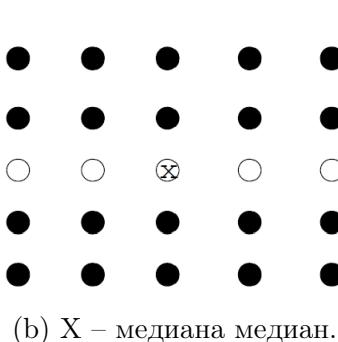
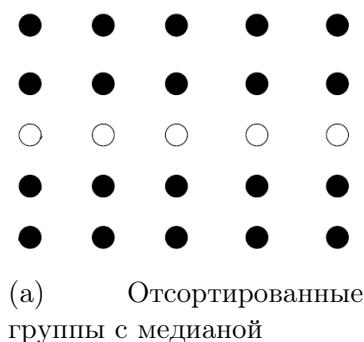
k – система счисления

7 Нахождение i -го по величине элемента массива.

Дан массив a_1, \dots, a_n . Алгоритм делит массив на $\lceil \frac{n}{5} \rceil$ пятерок элементов, последняя может быть не полной.

$$\underbrace{a_1, a_2, a_3, a_4, a_5}_{1\text{-я пятёрка}}, \quad \underbrace{a_6, a_7, a_8, a_9, a_{10}, \dots}_{2\text{-я пятёрка}}, \quad \underbrace{\dots, a_n}_{\lceil \frac{n}{5} \rceil\text{-я пятёрка}}$$

1. В каждой пятерке за константное время находится медиана.
2. Берется массив из $\lceil \frac{n}{5} \rceil$ медиан и в нем тем же методом рекурсивно определяется медиана y . Это первый рекурсивный вызов.
3. Ко всему массиву применяется процедура разбиения, подобная процедуре *partition()* из *quicksort*, используя медиану в качестве опорного элемента.
4. Если медиана медиан оказалась i -м элементом массива, она возвращается.
5. Иначе берется часть массива в которую попадает i -ая позиция. Это второй рекурсивный вызов.



Algorithm 9 Нахождение i -го по величине элемента массива (Блум, Флойд, Пратт, Ривест, Тарьян 1973)

Процедура $select(a_1, \dots, a_n; i)$

```
1: if  $n = 1$  then
2:   return  $a_1$ 
3: Пусть  $k = \lceil \frac{n}{5} \rceil$ 
4: Отсортировать каждую пятёрку  $a_{5j+1}, \dots, a_{5j+5}$ 
5: Пусть  $b_1, \dots, b_k$  — массив, где  $b_j$  — медиана  $j$ -й пятёрки.
6:  $y = select(b_1, \dots, b_k; \lceil \frac{k}{2} \rceil)$  /* медиана медиан */
7:  $j = partition(a_1, \dots, a_n; y)$ 
8: if  $j = i$  then
9:   return  $a_j$ 
10: else if  $j > i$  then
11:   return  $select(a_1, \dots, a_{j-1}; i)$ 
12: else
13:   return  $select(a_{j+1}, \dots, a_n; i - j - 1)$ 
```

Пример

Входные данные:

Массив $A = [12, 5, 8, 20, 3, 9, 15, 2, 7, 11, 18, 1, 6]$ (13 элементов).

Ищем $k = 6$ (элемент с рангом 6 в 0-индексации, т.е. 7-й по величине).

Шаг 1. Разбиение на пятерки

Делим массив на группы по 5 элементов:

- $G_1 = [12, 5, 8, 20, 3]$
- $G_2 = [9, 15, 2, 7, 11]$
- $G_3 = [18, 1, 6]$ (остаток)

Шаг 2. Поиск медиан групп

Сортируем каждую группу и берем средний элемент:

- $sorted(G_1) = [3, 5, 8, 12, 20] \Rightarrow m_1 = 8$
- $sorted(G_2) = [2, 7, 9, 11, 15] \Rightarrow m_2 = 9$
- $sorted(G_3) = [1, 6, 18] \Rightarrow m_3 = 6$

Массив медиан: $M = [8, 9, 6]$.

Шаг 3. Выбор опорного элемента (Pivot)

Рекурсивно ищем медиану массива M . Сортируем $M: [6, 8, 9]$. Медиана медиан $x = 8$.

Шаг 4. Разбиение (Partition) относительно $x = 8$

Переупорядочиваем исходный массив A так, чтобы элементы < 8 были слева, а > 8 — справа:

- Меньше 8 (L): $[5, 3, 2, 7, 1, 6]$ (6 элементов)
- Равно 8 (E): $[8]$ (1 элемент)
- Больше 8 (R): $[12, 20, 9, 15, 11, 18]$ (6 элементов)

Шаг 5. Анализ позиции

Длина L равна 6. Опорный элемент (8) занимает позицию с индексом 6 (после шести элементов из L).

Так как мы искали индекс $k = 6$, то опорный элемент и есть искомый.

Результат: 6-й по величине элемент — 8.

7.1 Процедура разбиения

Сперва все элементы массива сравниваются с опорным. Отдельно записываются элементы меньше опорного, равные опорному и больше, чем опорный.

Размещаются в массиве последовательно и возвращается номер опорного элемента.

7.2 Оценка времени работы

1. $T(n)$ – максимальное число сравнений при n -элементном массиве.
2. Медиана медиан находится за $T(\lceil \frac{n}{5} \rceil)$ действий.
3. Время работы для поиска i -го элемента в одной из двух частей массива $T(s)$, где s – количество элементов в этой части. Но s не превосходит $\frac{7n}{10}$, так как чисел меньше рассекающего элемента не менее $\frac{3n}{10}$ – это $\frac{n}{10}$ медиан, меньше медианы медиан и плюс не менее $\frac{2n}{10}$ элементов меньше этих медиан. С другой стороны чисел больше рассекающего элемента тоже не менее $\frac{3n}{10} \Rightarrow s \leq \frac{7n}{10}$
4. Время на сортировку групп и разбиение по рассекающему элементу Cn .

Итого: $T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + Cn$

Покажем, что для всех n выполняется неравенство $T(n) \leq 10Cn$

По индукции

1. Предположим, что наше неравенство $T(n) \leq 10Cn$ выполняется при малых n , для некоторой достаточно большой константы C .
2. Тогда, по предположению индукции, $T\left(\frac{n}{5}\right) \leq 10C\frac{n}{5} = 2Cn$ и $T\left(\frac{7n}{10}\right) \leq 10C\frac{7n}{10} = 7Cn$, тогда

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + Cn = 2Cn + 7Cn + Cn = 10Cn \Rightarrow T(n) \leq 10Cn$$

Так как $T(n) \leq 10Cn$, то время работы алгоритма $O(n)$.

8 Поиск в ориентированном графе: поиск в ширину, поиск в глубину, топологическая сортировка, нахождение компонентов сильной связности.

8.1 Поиск в ширину

BFS (breadth-first search): в ориентированном графе $G = (V, E)$ обойти все вершины, достижимые из данной вершины $s \in V$, каждую по одному разу.

Алгоритм

В каждый момент времени вершина может быть помеченной или непомеченной.

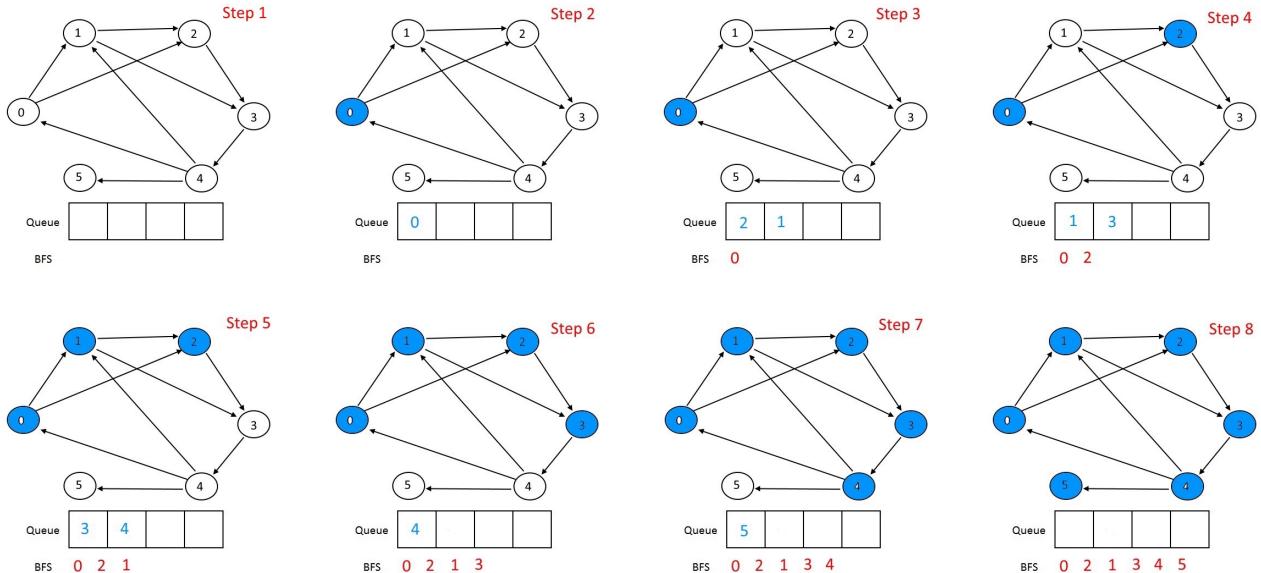
Если вершина помечена, то алгоритм уже нашёл путь из s в неё. Кроме пометок на вершинах, алгоритм хранит очередь, в которой находятся все те помеченные вершины, для которых ещё не обработаны исходящие дуги.

Вершина может быть:

- (1) не помеченной
- (2) помеченной, но ещё не обработанной (в очереди)

(3) помеченной и обработанной (все исходящие дуги рассмотрены)

Изначально помечена только вершина s и очередь состоит из s . На каждом шаге из очереди извлекается следующая вершина u , рассматриваются все исходящие из неё дуги (u, v) , если v не помечена – пометить и поместить в конец очереди.



Algorithm 10 Поиск в ширину

Структуры данных: очередь вершин для обработки, множество всех встреченных вершин.

```

1: пометить  $s$ 
2: поместить  $s$  в конец очереди
3: while очередь непуста do
4:   извлечь из очереди первый элемент,  $u$ 
5:   for all  $v: (u, v) \in E$  do
6:     if  $v$  не помечена then
7:       пометить  $v$ 
8:       поместить  $v$  в конец очереди
9:   теперь  $u$  обработана

```

Время работы: $O(|E|)$

Чтобы проверить, что алгоритм действительно найдёт все вершины достижимые из s , докажем следующие утверждения:

Утверждение

1. Очередь содержит вершины двух уровней:
Сначала идут вершины находящиеся на расстоянии ℓ от s .
Затем вершины, находящиеся на расстоянии $\ell + 1$ от s .
2. Все вершины с расстоянием, меньшим, чем ℓ , уже обработаны.
3. Все вершины на расстоянии ℓ , которых нет в очереди тоже обработаны.
4. Вершины на расстоянии $\ell + 1$, которые в очереди – это потомки уже обработанных вершин с расстоянием ℓ .

Набросок доказательства

Базис: в очереди только s , она находится на расстоянии 0, и это все вершины на расстоянии 0.

Переход: пусть извлекается u , она на расстоянии ℓ . Все новые вершины v , добавленные в очередь по дуге (u, v) , будут на расстоянии $\ell + 1$. Действительно, будь для v более короткий путь, она, по предположению индукции, уже была бы помечена ранее.

Утверждение

- (I) алгоритм помечает вершину $v \Leftrightarrow$ есть путь из s в v ;
- (II) если алгоритм находит v по дуге (u, v) , то один из кратчайших путей из s в v идёт через u ;
- (III) все пройденные дуги (u, v) образуют дерево.

Доказательство.

(I) (обходит \Rightarrow есть путь) индукция по длине вычисления.

Базис: помечена в первой строчке алгоритма – тогда это s .

Переход: когда v помечается, есть ранее помеченная вершина u и дуга (u, v) . Путь из s в u есть по предположению индукции, он продолжается дугой до искомого.

(есть путь \Rightarrow обходит) индукция по длине пути.

Базис: 0, помечается в первой строке.

Переход: пусть u — предпоследняя вершина на пути из s в v . Тогда путь в неё короче, и по предположению индукции u когда-то помечается. Одновременно u заносится в очередь, откуда она рано или поздно будет извлечена, и тогда при рассмотрении дуги (u, v) будет обнаружена вершина v .

(II) Из утверждения 1. u находится на расстоянии ℓ , а v на $\ell + 1$.

(III) Всякий раз, когда алгоритм переходит по дуге, он переходит в ранее не рассматривавшуюся вершину. Поэтому все рассмотренные дуги образуют ориентированный граф, в котором у одной вершины степень захода 0, а у остальных — 1. В таком графе не может быть циклов, в том числе неориентированных. \square

Время обхода всего графа: $O(|E| + |V|)$

8.2 Поиск в глубину

DFS (depth-first search): пройти до конца самый длинный путь, затем вернуться на шаг назад и продолжить следующий путь — и так далее, пока все вершины, достижимые из данной, не будут найдены.

Статусы вершин:

непомеченная («белая»);

помеченная и обрабатываемая («серая»)

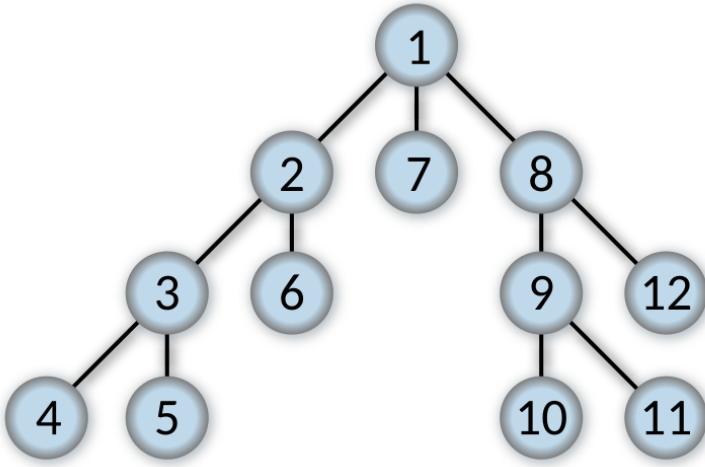
помеченная и обработанная («чёрная»).

Процесс обхода

Изначально все вершины — непомеченные.

Алгоритм произвольно выбирает одну не помеченную вершину, помечает её, и не помеченные соседи один за другим рекурсивно обходятся.

Когда обходы всех соседей заканчиваются и последний рекурсивный вызов возвращается, обработка текущей вершины завершается (помечается чёрным), процедура возвращается.



Порядок обхода дерева в глубину

Процесс повторяется для следующей вершины.

Algorithm 11 Поиск в глубину

Процедура $DFS(u)$

```

1: пометить  $u$                                 /* «серая» */
2: for all  $v: (u, v) \in E$  do           /* «белая» */
3:   if  $v$  не помечена then
4:      $DFS(v)$ 
5: обработка  $u$  закончена                      /* «чёрная» */

```

Время работы: $O(|V| + |E|)$

Лемма о корректности обхода

Если перед запуском $DFS(s)$ ни одна вершина графа не помечена, то после завершения работы алгоритма помеченными окажутся в точности те вершины, которые достижимы из вершины s .

Доказательство. Обозначим множество помеченных вершин как M , а множество достижимых из s вершин как R . Нам нужно доказать, что $M = R$. Докажем два включения:

1. **Все помеченные вершины достижимы ($M \subseteq R$).**

Докажем индукцией по порядку вызова процедуры DFS .

- **База:** Первая помеченная вершина — это s . Она достижима из самой себя (путь длины 0).
- **Шаг:** Пусть v — вершина, для которой был вызван $DFS(v)$. Этот вызов мог произойти только из процедуры $DFS(u)$, где (u, v) — ребро графа. По предположению индукции, вершина u достижима из s . Следовательно, v также достижима из s (через путь до u , продолженный ребром (u, v)).

2. **Все достижимые вершины будут помечены ($R \subseteq M$).**

Докажем индукцией по длине кратчайшего пути k от s до вершины v .

- **База ($k = 0$):** $v = s$. Вершина s помечается явно при первом вызове алгоритма.
- **Шаг:** Предположим, что все вершины на расстоянии k от s помечаются. Рассмотрим вершину v , находящуюся на расстоянии $k + 1$.

Существует путь длины $k + 1$ из s в v . Пусть u — предпоследняя вершина этого пути. По предположению индукции, $DFS(u)$ будет вызван.

В $DFS(u)$ алгоритм перебирает все исходящие ребра, включая ребро (u, v) . В этот момент:

- Либо вершина v еще не помечена \Rightarrow будет вызван $DFS(v)$, и она станет помеченной.
- Либо вершина v уже была помечена ранее \Rightarrow она уже находится в множестве M .

В обоих случаях по завершении работы $DFS(s)$ вершина v окажется помеченной.

□

Поиск в глубину приспособливается к обходу всего графа так же, как и поиск в ширину.

Сперва произвольно выбирается начальная вершина, проводится поиск в глубину из неё, а по завершении обхода выбирается следующая непомеченная вершина, и повторяется то же самое, пока все вершины не окажутся помеченными.

Algorithm 12 Поиск в глубину

Основная процедура

```

1:  $t = 0$ 
2: for all  $s \in V$  do
3:   if  $s$  не помечена then
4:      $DFS(s)$ 
```

Процедура $DFS(u)$

```

1: пометить  $u$ 
2:  $t = t + 1$ 
3: начало( $u$ ) =  $t$ 
4: for all  $v : (u, v) \in E$  do
5:   if  $v$  не помечена then
6:      $DFS(v)$ 
7:  $t = t + 1$ 
8: конец( $u$ ) =  $t$ 
```

В момент перекраски серой вершине назначается время *обнаружения*, а в момент перекраски в чёрный — время *окончания*

Время работы: $O(|V| + |E|)$;

8.3 Топологическая сортировка

Дан ориентированный граф $G = (V, E)$.

Задача: расположить его вершины в таком порядке, чтобы для каждой дуги $(u, v) \in E$ вершина u была записана раньше, чем v . Это возможно тогда и только тогда, когда граф ациклический.

Решение: запустить поиск в глубину, потом отсортировать вершины по времени окончания. Законченные позже всего должны быть в начале.

Лемма

Граф ациклический тогда и только тогда, когда при поиске в глубину никогда не рассматривается дуга, ведущая в вершину, находящуюся в стеке возврата (дуга из «серой» вершины в «серую»).

Доказательство. Поскольку последовательность вершин в стеке возврата образует путь в графе, такая дуга замкнёт этот путь в цикл.

И обратно: если есть цикл, то какая-то из его вершин, v , будет рассмотрена первой. Тогда рекурсивные вызовы, делаемые процедурой $DFS(v)$, обойдут весь цикл и рассмотрят замыкающее цикл ребро. \square

Лемма

Если в ациклическом графе есть дуга (u, v) , то время завершения v меньше, чем время завершения u .

Доказательство. Эта дуга однажды будет рассматриваться. В этот момент v нет в стеке возврата. Если v помечена, то у неё уже есть время завершения, а у u ещё только будет (позднее). Если же v ранее не рассматривалась, то в неё спустится рекурсия, и по возвращении из рекурсии v будет уже обработана — а u только предстоит получить время завершения. \square

8.4 Нахождение компонентов сильной связности

Ставится задача найти в данном ориентированном графе $G = (V, E)$ его компоненты сильной связности.

Задача решается алгоритмом Косараджу–Шарира.

Алгоритм работает так. Сперва запускается поиск в глубину для G . Затем запускается поиск в глубину для *обращённого графа* G^R , в котором направления всех дуг изменены на обратные (у G и G^R одни и те же компоненты сильной связности).

При поиске в глубину для G^R во внешнем цикле вершины рассматриваются *в порядке времени их завершения* при первом поиске в глубину, *от конца к началу*. После этого оказывается, что каждый запуск процедуры DFS во внешнем цикле будет находить очередную компоненту сильной связности исходного графа.

Правильность алгоритма основана на следующем свойстве.

Лемма

Пусть C и D — различные компоненты сильной связности ориентированного графа $G = (V, E)$. Пусть существует ребро $(u, v) \in E$, такое что $u \in C$, а $v \in D$. Тогда максимальное время завершения обработки вершин в C больше, чем в D :

$$\max_{x \in C} f[x] > \max_{y \in D} f[y],$$

где $f[v]$ — время выхода из вершины v (время завершения обработки) при поиске в глубину.

Доказательство. Рассмотрим два возможных случая старта обхода:

1. Первой посещена вершина из C .

Пусть $x \in C$ — первая вершина, в которую зашёл DFS . В этот момент все вершины и в C ,

и в D ещё не посещены («белые»). Так как $C \rightarrow D$, то из x есть путь ко всем вершинам C (по определению сильной связности) и ко всем вершинам D (через связующее ребро). Запущенная процедура $\text{DFS}(x)$ не завершится, пока не обойдёт все доступные из x непосещённые вершины. Значит, рекурсия спустится во все вершины C и D , обработает их, и только после этого завершится вызов $\text{DFS}(x)$. Следовательно, время выхода из x будет самым последним.

2. Первой посещена вершина из D .

Пусть $y \in D$ — первая вершина, в которую зашёл DFS . Процедура $\text{DFS}(y)$ обойдёт все вершины, доступные из y . Поскольку C и D — разные компоненты сильной связности и ребро идёт $C \rightarrow D$, то обратного пути из D в C не существует (иначе это была бы одна большая компонента). Значит, DFS из y обойдёт только вершины D , но никак не сможет попасть в C . Обработка D завершится полностью, время выхода для всех вершин D будет простирано. И только спустя какое-то время алгоритм дойдёт до вершин из C . Таким образом, времена выхода в D будут меньше, чем в C . \square

Теорема

Вершины каждого дерева, найденного алгоритмом Косараджу–Шарира при втором поиске в глубину — это и есть сильно связные компоненты исходного графа.

Доказательство. Индукция по количеству найденных компонентов сильной связности. Надо доказать, что если первые k выданных компонентов связности действительно будут таковыми, то и следующий тоже будет компонентом связности. На этом этапе поиск в глубину в графе G^R находит вершину y , позже всех закончившуюся среди ещё не рассмотренных вершин. Эта вершина, в частности, закончилась позже всех в своём сильно связном компоненте D .

Утверждается, что при поиске в G^R из y не удастся пройти ни в какой другой ССК. Действительно, если можно пройти в C по дуге (v, u) , то в графе G была дуга (u, v) , и потому, согласно предыдущей лемме, самое позднее время завершения в C — позднее чем в u . Поэтому все вершины из C к этому моменту уже рассмотрены, туда больше не перейти. \square

Время работы алгоритма: $O(|V| + |E|)$.

9 Кратчайшие пути в графе с весами: алгоритм Беллмана–Форда, алгоритм Дейкстры. Очередь с приоритетами и её реализация.

9.1 Поиск в графе с весами: алгоритм Беллмана–Форда

Пусть в ориентированном графе $G = (V, E)$ для каждой дуги $(u, v) \in E$ задан её вес — число $w_{u,v} \in \mathbb{R}$. Если дуги нет, можно положить $w_{u,v} = \infty$.

Нужно найти *пути наименьшего веса* из данной вершины $s \in V$ во все вершины графа. В графе допускаются дуги отрицательного веса, и такие дуги могут сократить уже проделанный путь. Если же в графе есть цикл отрицательного веса, то тогда, проходя его много раз, можно сделать вес пути сколь угодно малым — и потому пути наименьшего веса в вершины, достижимые из этого цикла, окажутся неопределёнными. Поэтому алгоритм ищет пути наименьшего веса из s во все вершины, коль скоро из s нельзя прийти ни в какой *цикл отрицательного веса*. Если же это условие не выполняется, то алгоритм должен об этом сообщить.

Алгоритм Беллмана–Форда решает эту задачу, вычисляя для каждой вершины $v \in V$ сле-

дующие значения:

- d_v : наименьший вес пути из s в v ;
- π_v : предыдущая вершина на пути наименьшего веса из s в v .

Значения π_v образуют дерево кратчайших путей (для достижимых вершин).

Изначально полагается $d_v = \infty$, $\pi_v = \text{NULL}$ для всех вершин, и $d_s = 0$. Это первое приближение: найдена только начальная вершина, пути в другие вершины неизвестны.

Алгоритм постепенно находит пути меньшего веса в другие вершины, запоминая веса лучших из найденных путей в этих переменных. Значения уменьшаются с помощью элементарной операции *улучшения пути*, используя некоторую дугу $(u, v) \in E$.

```
1: if  $d_u + w_{u,v} < d_v$  then
2:    $d_v = d_u + w_{u,v}$ 
3:    $\pi_v = u$ 
```

Лемма

После i -й итерации внешнего цикла алгоритм Беллмана–Форда находит все пути наименьшего веса длины не более чем i .

Доказательство. Индукция по i .

База: $i = 0$. Вес пути длины 0 равен 0.

Переход. Пусть верно для i -го прохода. Всякий путь наименьшего веса длины $i + 1$ в вершину v на предыдущем шаге проходит через некоторую вершину u , и потому содержит путь наименьшего веса длины i в u . Тогда на $(i + 1)$ -м проходе искомый путь в вершину v будет найден при рассмотрении вершины u . \square

Algorithm 13 Алгоритм Беллмана–Форда

```
1:  $d_s = 0$ 
2:  $d_v = \infty$  для всех  $v \neq s$ 
3: for  $i = 1$  to  $|V| - 1$  do
4:   for all  $(u, v) \in E$  do
5:     if  $d_u + w_{u,v} < d_v$  then           ▷ если путь из  $s$  в  $v$  можно улучшить этой дугой
6:        $d_v = d_u + w_{u,v}$ 
7:        $\pi_v = u$ 
8:   for all  $(u, v) \in E$  do
9:     if  $d_u + w_{u,v} < d_v$  then
10:    return есть достижимый цикл отрицательного веса
11: return достижимых циклов отрицательного веса нет
```

Теорема

Алгоритм Беллмана–Форда за $|V| \cdot |E|$ шагов или правильно вычисляет пути наименьшего веса из вершины s во все вершины, или сообщает о наличии достижимого цикла отрицательного веса.

Доказательство. По лемме, после $|V| - 1$ итераций найден кратчайший путь до каждой вершины среди путей, содержащих не более $|V| - 1$ дуг, а для недостижимых вершин расстояние остаётся бесконечным.

Сначала рассмотрим случай, когда на $|V|$ -й итерации некоторая дуга (u, v) всё ещё может быть релаксирована, то есть $d_u + w_{u,v} < d_v$. Это означает, что существует путь P из s в v длиной не более $|V|$ дуг с весом строго меньше текущего значения d_v . Если бы в P было не более $|V| - 1$ дуг, то алгоритм уже нашёл бы такой путь по лемме, что противоречит неравенству выше, поэтому в P ровно $|V|$ дуг и $|V|+1$ вершин. Среди этих $|V|+1$ вершин две совпадают, значит, в P есть цикл C . Если удалить из P этот цикл, длина пути уменьшится, а вес не увеличится, иначе P не был бы самым выгодным путём длины не более $|V|$ дуг, следовательно, вес цикла C отрицателен, и цикл достижим из s .

Теперь предположим наоборот, что после $|V| - 1$ итераций ни одна дуга (u, v) уже не может быть релаксирована, то есть для всех дуг выполнено $d_u + w_{u,v} \geq d_v$. Пусть существует достижимый из s цикл $v_0, v_1, \dots, v_{k-1}, v_k = v_0$ отрицательного веса. Для каждой дуги цикла (v_i, v_{i+1}) имеем $d_{v_i} + w_{v_i, v_{i+1}} \geq d_{v_{i+1}}$. Складывая эти неравенства по всем i и сокращая одинаковые суммы $\sum d_{v_i}$, получаем $\sum_{i=0}^{k-1} w_{v_i, v_{i+1}} \geq 0$, что противоречит предположению об отрицательном весе цикла.

Значит, если после $|V| - 1$ итераций релаксаций больше нет, то достижимых отрицательных циклов не существует, а найденные значения d_v — это длины кратчайших путей из s до всех вершин. Совместно с первым случаем это доказывает корректность алгоритма Беллмана–Форда. \square

9.2 Поиск в графе с весами: алгоритм Дейкстры

Тоже находит все пути наименьшего веса из s во все вершины. Работает быстрее, но требует, чтобы не было дуг отрицательного веса.

Алгоритм Дейкстры в каждый момент времени помнит для каждой вершины v вес d_v лучшего из известных ему путей из s , а также предпоследнюю вершину на этом пути, и пытается находить более короткие пути.

В момент обработки очередной вершины u путь наименьшего веса в u к этому моменту уже известен. Тогда алгоритм рассматривает все дуги, исходящие из вершины u , и с их помощью улучшает пути, ведущие на одну вершину дальше. Это позволяет алгоритму просматривать каждую дугу графа лишь однажды.

На каждом шаге алгоритм помнит набор вершин $Q \subset V$, пути наименьшего веса из которых ещё не продолжены. Изначально это все вершины, $Q = V$.

Алгоритм на каждом шаге находит такую вершину u из Q , что известная длина d_u кратчайшего пути в неё — наименьшая, и перебирает исходящие из неё дуги, улучшая пути по ним.

Algorithm 14 Алгоритм Дейкстры

- 1: $d_s = 0$
 - 2: $d_v = \infty$ для всех $v \neq s$
 - 3: $Q = V$
 - 4: **while** $Q \neq \emptyset$ **do**
 - 5: пусть $u \in Q$ — вершина с минимальным значением d_u
 - 6: $Q = Q \setminus \{u\}$
 - 7: **for all** $v : (u, v) \in E$ **do**
 - 8: **if** $d_u + w_{u,v} < d_v$ **then** /* если путь из s в v можно улучшить этой дугой */
 - 9: $d_v = d_u + w_{u,v}$
 - 10: $\pi_v = u$
-

Теорема

Алгоритм Дейкстры работает правильно.

Доказательство. Пусть D_v — наименьший вес путей из s в v .

Утверждение о правильности. В начале каждой итерации внешнего цикла, для всякой вершины $v \notin Q$, путь наименьшего веса в ней уже построен, то есть $d_v = D_v$.

Индукцией по длине вычисления.

В начале работы условие выполняется, поскольку вершин не из Q нет. Пусть из Q удаляется вершина u , и при удалении всех предшествовавших вершин условие выполнялось.

Надо доказать, что значение d_u в этот момент — это наименьший вес пути из s в u ; иными словами, что $d_u = D_u$.

Какой-то путь наименьшего веса D_u из s в u есть. Поскольку $s \notin Q$ и $u \in Q$ на этом пути есть пара последовательных вершин $x \notin Q$, $y \in Q$, соединённых ребром $(x, y) \in E$ (ничто не мешает вершине x совпадать с s , а вершине y — с u). Вершина x была удалена из Q раньше, чем u , и, по предположению индукции, в тот момент для неё уже был построен путь наименьшего веса. Тогда же этот путь был продолжен до вершины y — и это продолжение даёт путь наименьшего веса в y : действительно, существуй в y путь меньшего веса, существовал бы и путь меньшего веса в u .

Тогда известно следующее. Во-первых, раз алгоритм выбрал из Q именно вершину u , а не y , то $d_u \leq d_y$. Во-вторых, $d_y = D_y$, так как путь наименьшего веса в y уже найден. В-третьих, $D_y \leq D_u$ поскольку эти две вершины находятся на одном пути. Из цепочки неравенств $d_u \leq d_y = D_y \leq D_u$ следует, что $d_u = D_u$ — и, стало быть, к моменту извлечения вершины u из очереди ведущий в неё путь наименьшего веса уже найден. \square

Время работы:

Алгоритм Дейкстры реализуется с использованием очереди с приоритетами для хранения множества Q и значений d_v для всех элементов $v \in Q$. Тогда во время работы он проделывает $|V|$ операций *extract_min()* и не более $|E|$ операций *decrease()*. Каждая операция работает за время $O(\log |V|)$. Поэтому итоговое время работы — $O((|E| + |V|) \log |V|)$. Если граф связный, то можно записать короче: $O(|E| \log |V|)$.

9.3 Очередь с приоритетами

Для представления множества Q алгоритм использует особую структуру данных: очередь с приоритетами (priority queue). Каждый элемент Q находится там вместе со своим текущим значением d_v . Операции:

- *insert(x)*: вставить новый элемент.
- *min()*: выдать минимальный элемент.
- *extract_min()*: выдать минимальный элемент и удалить его.
- *decrease(x, k)*: изменить значение элемента $x \in Q$ на k , коль скоро k меньше его текущего значения.

Сложность алгоритма Дейкстры зависит от того, как реализована очередь с приоритетами.

Наивная реализация: хранить массив x_v , индексированный по $v \in V$.

Время работы *decrease* будет $O(1)$, но *extract_min* требует времени $|V|$.

Отсюда общее время работы $|V|^2 + |E| = O(|V|^2)$.

Другая наивная реализация — хранить сортированный массив. Теперь $extract_min$ работает за время $O(1)$, а для $decrease$ нужно будет переставлять элемент в нужное место, что потребует времени $|V|$. Выйдет время $|E| \cdot |V|$ — ещё хуже.

Реализация очереди с приоритетами с помощью кучи

Используется куча, в которой значение в каждой внутренней вершине не больше, чем значение в любом из её потомков (min-heap).

- $insert(x)$: добавить в конец кучи новый элемент (он становится листом), после чего дать ему всплыть наверх до его законного места.
- $min()$: просто вернуть x_1 .
- $extract_min()$: переместить x_n в x_1 , убрав его в конце; затем запустить исправление кучи из корня, то есть, $heapify(1)$.
- $decrease(i, k)$: изменить значение x_i на k , после чего дать элементу x_i всплыть наверх, пока возможно.

Важно, что процедура $decrease$ получает на входе номер ячейки i , в которой лежит элемент, а любая операция над кучей может перемещать элементы между ячейками. Поэтому необходимо поддерживать дополнительную структуру данных — отображение номеров вершин исходного массива в номера ячеек.

10 Кратчайшие пути в графе между всеми парами вершин: алгоритм Варшалла. Кратчайшие пути с весами: алгоритм Флойда–Варшалла. Нахождение всех путей с помощью умножения матриц.

Дан ориентированный граф $G = (V, E)$, где $V = \{1, \dots, n\}$, а $E \subseteq V \times V$.

Задача: проверить существование пути из каждой вершины в каждую — то есть для каждой пары вершин (i, j) определить, есть ли путь из i в j .

Эти сведения составляют новый граф с тем же множеством вершин, называемый транзитивным замыканием графа G : это граф $G^* = (V, E^*)$, где $(i, j) \in E^*$, если в G есть путь из i в j .

10.1 Очевидный алгоритм для транзитивного замыкания

Очевидный алгоритм будет перебирать все пары вершин (i, j) , а для каждой пары — все промежуточные вершины k , для которых есть дуги (i, k) и (k, j) . Если при этом дуги (i, j) ещё нет, она будет добавляться. Перебор всех пар (i, j) будет продолжаться, пока можно добавить новые дуги.

Algorithm 15 Очевидный алгоритм для транзитивного замыкания

```

1: while можно добавить дуги do
2:   for  $i = 1$  to  $n$  do
3:     for  $j = 1$  to  $n$  do
4:       for  $k = 1$  to  $n$  do
5:         if  $(i, k) \in E \wedge (k, j) \in E$  then
6:           добавить дугу  $(i, j)$  к  $E$ 
```

Грубая верхняя оценка времени работы — $O(n^5)$, потому что на каждой итерации внешнего цикла, кроме последней, добавляется по меньшей мере одно ребро из $O(n^2)$ возможных. Но можно оценить лучше, используя следующий инвариант внешнего цикла.

Лемма

После каждой t -й итерации внешнего цикла в E добавлены все такие дуги (i, j) , что в исходном графе есть путь из i в j длины не более 2^t . И обратно: если дуга когда-либо добавляется, то путь есть.

Доказательство. По индукции

База ($t = 0$). До запуска цикла в E лежат исходные дуги, то есть все пути длины $1 = 2^0$. Утверждение верно.

Переход. Пусть после t -й итерации утверждение верно. Рассмотрим путь длины $L \leq 2^{t+1}$ из i в j :

- Если $L \leq 2^t$, дуга (i, j) уже в E по предположению.
- Если $2^t < L \leq 2^{t+1}$, разобьём путь на $i \rightarrow k$ длины $\leq 2^t$ и $k \rightarrow j$ длины $\leq 2^t$. Тогда после t -й итерации в E есть дуги (i, k) и (k, j) , а на $(t+1)$ -й итерации алгоритм добавит (i, j) .

Обратное направление.

Алгоритм добавляет (i, j) только если есть (i, k) и (k, j) . Они соответствуют путям длины $\leq 2^t$, значит существует путь $i \rightarrow j$ длины $\leq 2^{t+1}$. \square

10.2 Алгоритм Варшалла

Транзитивное замыкание можно построить за время n^3 .

Algorithm 16 Алгоритм Варшалла для транзитивного замыкания

```

1: for  $k = 1$  to  $n$  do
2:   for  $i = 1$  to  $n$  do
3:     for  $j = 1$  to  $n$  do
4:       if  $(i, k) \in E \wedge (k, j) \in E$  then
5:         добавить дугу  $(i, j)$  в  $E$ 

```

Время работы: $O(n^3)$

Лемма (Варшалл 1962).

После k -й итерации внешнего цикла переменная E содержит все пары вершин (i, j) , для которых в исходном графе существует путь из i в j , проходящий только через промежуточные вершины из множества $\{1, \dots, k\}$.

Доказательство. Доказывается индукцией по числу итераций.

База $k = 0$: перед началом работы алгоритма переменная E содержит все ребра графа, то есть все пути, проходящие через промежуточные вершины из пустого множества.

Переход. К началу k -й итерации все пути, проходящие через промежуточные вершины из $\{1, \dots, k-1\}$, уже найдены. Пусть кратчайший путь из i в j проходит только через вершины из $\{1, \dots, k\}$. Если через k он не проходит, то он уже построен. Если же он проходит через

k , то он когда-то приходит в неё впервые — и, стало быть, есть путь из i в k , проходящий только через промежуточные вершины из $\{1, \dots, k-1\}$ — и когда-то в последний раз её покидает, так что есть путь из k и j , тоже проходящий только через $\{1, \dots, k-1\}$. Эти два пути уже построены по предположению индукции, и тогда на k -й итерации находится и искомый путь. \square

Из леммы сразу следует правильность алгоритма: действительно, после n -й итерации внешнего цикла будут найдены все пути, проходящие через промежуточные вершины $\{1, \dots, n\}$, то есть совсем все пути.

10.3 Кратчайшие пути с весами: алгоритм Флойда–Варшалла.

Теперь граф взвешенный, то есть для каждой дуги задан её вес $w_{i,j}$. Если дуги нет, можно положить $w_{i,j} = \infty$. Допускаются дуги с отрицательным весом, но предполагается, что нет циклов отрицательного веса. Нужно найти пути наименьшего веса между всеми парами вершин.

Алгоритм Флойда–Варшалла решает эту задачу за время $O(n^3)$.

Algorithm 17 Алгоритм Флойда–Варшалла для нахождения всех кратчайших путей

1: Массив $d_{i,j}$, начальные значения: $d_{i,j} = \begin{cases} 0, & \text{если } i = j \\ w_{i,j}, & \text{если } (i, j) \in E \\ \infty, & \text{в противном случае} \end{cases}$

2: **for** $k = 1$ to n **do**

3: **for** $i = 1$ to n **do**

4: **for** $j = 1$ to n **do**

5: **if** $d_{i,j} > d_{i,k} + d_{k,j}$ **then**

6: $d_{i,j} = d_{i,k} + d_{k,j}$

Лемма

Пусть в графе нет циклов отрицательного веса. Тогда после каждой k -й итерации алгоритма 17 для каждой пары вершин (i, j) переменная $d_{i,j}$ содержит наименьший вес пути из i в j среди путей, проходящих только через промежуточные вершины из множества $\{1, \dots, k\}$.

Доказательство. Доказательство такое же, как для случая невзвешенного графа. Индукция по k .

База $k = 0$ выполнена.

В индукционном переходе берётся путь наименьшего веса из i в j , проходящий только через промежуточные вершины из $\{1, \dots, k\}$ и заходящий в k . Тогда участок этого пути между первым и последним заходом в k можно убрать — ведь его вес не может быть отрицательным, так как в графе нет отрицательных циклов по предположению. Тогда путь состоит из двух кусков, для которых наименьший вес пути уже получен. \square

Чтобы построить сами кратчайшие пути, для каждой пары вершин можно запоминать вершину $\pi_{i,j}$ — следующую после i вершину на кратчайшем пути из i в j .

10.4 Нахождение всех путей с помощью умножения матриц.

Матрица смежности — булева матрица $A \in \mathbb{B}^{n \times n}$, в которой элемент $a_{i,j}$ равен 1, если в графе есть дуга (i, j) , и равен 0, если дуги нет.

Произведение булевых матриц $A \in \mathbb{B}^{m \times \ell}$ и $B \in \mathbb{B}^{\ell \times n}$ — это булева матрица $A \times B = C \in \mathbb{B}^{m \times n}$, со следующими значениями элементов:

$$c_{i,j} = \bigvee_{k=1}^{\ell} a_{i,k} \wedge b_{k,j}$$

Если $A \in \mathbb{B}^{n \times n}$ — матрица смежности графа, то A^ℓ — матрица достижимости по путям длины ровно ℓ . Пусть $I \in \mathbb{B}^{n \times n}$ — единичная матрица. Тогда $(A \vee I)^\ell$ — матрица достижимости по путям длины не более ℓ , а A^{n-1} — матрица достижимости. Обозначение:

$$A^* = (A \vee I)^{n-1} \text{ — рефлексивно-транзитивное замыкание.}$$

Вычисляется возведением в квадрат $\log n$ раз. Если умножать матрицы напрямую, получится $n^3 \log n$, что медленно.

11 Быстрое умножение матриц: алгоритм Штассена.

11.1 Быстрое умножение матриц

Произведение двух матриц размера $n \times n$ по определению вычисляется за n^3 умножений и $n^2(n - 1)$ сложений. Однако существуют алгоритмы умножения матриц, работающие быстрее, чем по определению, *быстрее очевидного* — и уже сама такая возможность удивительна.

11.2 Алгоритм Штассена

Пусть \mathcal{R} — кольцо, то есть множество с заданными на нём ассоциативными операциями сложения и умножения, причём сложение коммутативно, сложение и умножение дистрибутивные, есть нейтральные элементы по сложению и умножению, а также у каждого элемента есть противоположный элемент относительно сложения (без противоположного элемента — полукольцо).

Сумма матриц $A \in \mathcal{R}^{m \times n}$, $B \in \mathcal{R}^{m \times n}$ — матрица $A + B = C \in \mathcal{R}^{m \times n}$:

$$c_{i,j} = a_{i,j} + b_{i,j}$$

Сложение матриц ассоциативно, коммутативно, есть нейтральный и противоположный элементы.

Произведение матриц $A \in \mathcal{R}^{m \times \ell}$, $B \in \mathcal{R}^{\ell \times n}$ — матрица $A \times B = C \in \mathcal{R}^{m \times n}$:

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

Умножение матриц ассоциативно, есть нейтральный элемент.

Утверждение

Квадратные матрицы размера $n \times n$ над кольцом сами образуют кольцо.

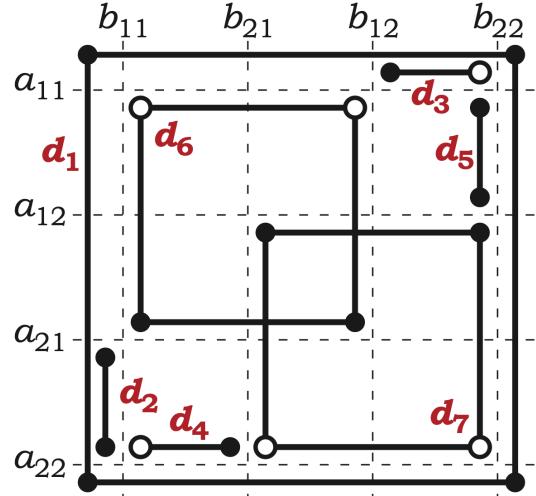
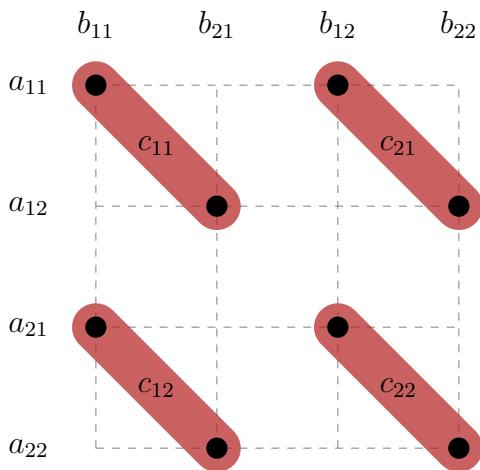
Произведение двух матриц 2×2 , согласно определению, выражается за 8 умножений и 4 сложения:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Но есть удивительный способ вычисления этого же произведения за 7 умножений и 18 сложений.

Лемма

(Штрассен [1969]). Пусть $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ и $B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$ — две матрицы размера 2×2 над кольцом R . Тогда их произведение $A \times B = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$ можно вычислить за 7 умножений и 18 сложений в R .



Доказательство. Восемь произведений, соответствующие определению произведения матриц, где горизонтальные пунктирные линии соответствуют четырём элементам A , вертикальные — четырём элементам B , а пересечение линий соответствует произведению элементов. Произведения разбиваются на 4 пары — суммы, соответствующие элементам искомой матрицы $A \times B$.

Чтобы вычислить эти же 4 суммы иначе, сперва вычисляют следующие 7 произведений.

$$\begin{aligned} d_1 &= (a_{11} + a_{22})(b_{11} + b_{22}), \\ d_2 &= (a_{21} + a_{22})b_{11}, \\ d_3 &= a_{11}(b_{12} - b_{22}), \\ d_4 &= a_{22}(b_{21} - b_{11}), \\ d_5 &= (a_{11} + a_{12})b_{22}, \\ d_6 &= (a_{21} - a_{11})(b_{11} + b_{12}), \\ d_7 &= (a_{12} - a_{22})(b_{21} + b_{22}). \end{aligned}$$

Тогда искомое произведение $C = AB$ выражается так:

$$\begin{aligned} c_{11} &= d_1 + d_4 + d_7 - d_5, \\ c_{12} &= d_3 + d_5, \\ c_{21} &= d_2 + d_4, \\ c_{22} &= d_1 + d_3 + d_6 - d_2. \end{aligned}$$

□

Смысл умножения матриц 2×2 за 7 умножений и много сложений в том, что эту формулу для произведения матриц можно применять рекурсивно, сводя умножение матриц размера $n \times n$ к семи умножениям матриц вдвое меньшего размера — *семи*, а не восьми.

Сперва матрицы $n \times n$ представляются в виде блочных матриц, состоящих из четырёх подматриц размера $\frac{n}{2} \times \frac{n}{2}$ каждая:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right) \times \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right) = \left(\begin{array}{c|c} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ \hline A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{array} \right).$$

Поскольку матрицы размера $\frac{n}{2} \times \frac{n}{2}$ над кольцом сами образуют кольцо, здесь записано произведение двух матриц размера 2×2 над этим кольцом. Согласно лемме, вычисление этого произведения сводится к умножению семи пар матриц вдвое меньшего размера, а также некоторому количеству сложений таких матриц. Наконец, для умножения матриц размера $\frac{n}{2} \times \frac{n}{2}$ рекурсивно используется этот же алгоритм.

На каждом уровне рекурсии, кроме самого нижнего, полученный алгоритм занимается исключительно сложениями и вычитаниями блочных матриц. Никакие числа при этом не умножаются. И лишь на самом нижнем уровне рекурсии умножаются матрицы размера 1×1 , то есть просто числа.

Время работы:

Глубина рекурсии — $\log_2 n$. Задач размера n^2 — одна. Задач размера $\frac{n}{2} \times \frac{n}{2}$ — семь. Задач размера $\frac{n}{4} \times \frac{n}{4}$ — всего 7^2 . Для каждой задачи на глубине i , время, затрачиваемое на внутренние вычисления, сводится к рекурсивным вызовам плюс $O\left(\left(\frac{n}{2^i}\right)^2\right)$ — это константное число сложений и вычитаний матриц размера $\frac{n}{2^i} \times \frac{n}{2^i}$. Поэтому общее количество действий оценивается по следующей формуле:

$$\sum_{i=0}^{\log_2 n} 7^i \cdot \left(\frac{n}{2^i}\right)^2 = n^2 \sum_{i=0}^{\log_2 n} \left(\frac{7}{4}\right)^i = n^2 \cdot \frac{\left(\frac{7}{4}\right)^{1+\log_2 n} - 1}{\frac{7}{4} - 1} = O\left(n^2 \cdot \left(\frac{7}{4}\right)^{\log_2 n}\right).$$

Так как $\left(\frac{7}{4}\right)^{\log_2 n} = n^{\log_2 7 - 2}$, получаем:

$$O\left(n^2 \cdot n^{\log_2 7 - 2}\right) = O\left(n^{\log_2 7}\right).$$

Теорема

(Штрассен). Пусть \mathcal{R} — кольцо, пусть $k \geq 0$, и пусть A и B — две матрицы $2^k \times 2^k$ над \mathcal{R} . Тогда произведение AB можно вычислить за 7^k умножений и $\Theta(7^k)$ сложений.

Если размер умножаемых матриц — не степень двойки, то их можно дополнить до следующий степени двойки нулями, что ухудшит время работы лишь в константное число раз. Поэтому матрицы любого размера $n \times n$ можно перемножить за время $O(n^{\log_2 7})$.

Метод Штрассена применим к любым кольцам (например, целым числам), но с `float` возможны ошибки округления, так как они не образуют кольца.

Виноград доказал, что для матриц 2×2 нельзя использовать менее 7 умножений. Улучшения ищут через умножение матриц большего размера (например, 3×3), но пока безуспешно: алгоритм Ладермана для 3×3 требует 23 умножения, что медленнее Штрассена ($\log_3 23 > \log_2 7$).

Самый быстрый теоретически — метод Копперсмита–Винограда, улучшенный до $O(n^{2.3716})$. Но он обгоняет Штрассена только на астрономически больших n , поэтому на практике Штрассен остаётся лучшим.

Нетривиальных нижних оценок сложности умножения матриц нет (кроме очевидной n^2). Возможно, истинная сложность близка к $n^2(\log n)^{O(1)}$. Минимальный показатель степени обозначают ω ; известно, что $2 \leq \omega < 2.3716$.

12 Быстрое умножение булевых матриц через числовые: метод четырёх русских.

12.1 Умножение булевых матриц через числовые

Булевые матрицы определены над *булевым полукольцом*, а не над кольцом, поскольку нет противоположного элемента для сложения. Поэтому алгоритм Штрассена неприменим. Однако можно умножить булевые матрицы через числовые следующим образом.

Сперва булевые матрицы записываются как числовые матрицы с элементами 0 и 1, и затем они перемножаются в кольце вычетов по модулю $n + 1$ — или по любому удобному модулю, превосходящему n . Тогда вместо дизъюнкции конъюнкций

$$\bigvee_{k=1}^n (a_{i,k} \wedge b_{k,j})$$

будет вычислена сумма произведений

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

по модулю $n+1$. Сама по себе эта сумма в точности равна количеству истинных конъюнкций в дизъюнкции конъюнкций, и потому она лежит в диапазоне от 0 до n — откуда следует, что по модулю $n + 1$ она вычислится точно. Чтобы узнать значение дизъюнкции конъюнкций, достаточно будет проверить вычисленную сумму произведений на неравенство нулю.

12.2 Метод четырёх русских.

Простое решение

Если мы будем считать произведение матриц $C = A \cdot B$ по определению ($c_{i,j} = \sum_{k=1}^n a_{i,k} \cdot b_{k,j}$), то сложность работы алгоритма составит $O(n^3)$ — каждый из n^2 элементов результирующей матрицы C вычисляется за время, пропорциональное n .

Сейчас будет показано, как немного уменьшить это время.

Сжатие матриц

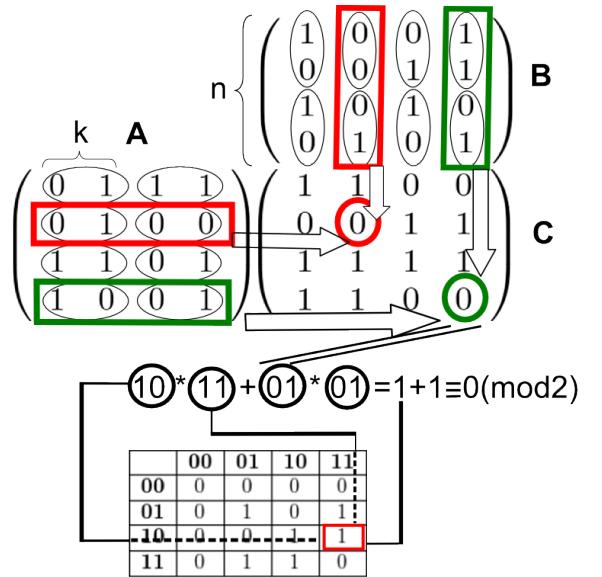
Предварительно строится таблица всех возможных скалярных произведений по модулю 2 для пар двоичных векторов длины k . Таких векторов 2^k , значит, потребуется 2^{2k} значений.

Матрица A преобразуется следующим образом: каждая её строка делится на блоки длины k , последний блок дополняется нулями при необходимости. Каждый блок интерпретируется как двоичное число. В итоге получается матрица

$$A'_{n \times \lceil \frac{n}{k} \rceil}.$$

Матрица B обрабатывается аналогично, только блокируются её столбцы. Получаем

$$B'_{\lceil \frac{n}{k} \rceil \times n}.$$



Теперь, если вместо произведения матриц A и B считать произведение новых матриц A' и B' , воспользовавшись посчитанными скалярными произведениями, то каждый элемент матрицы C будет получаться уже за время, пропорциональное $\lceil \frac{n}{k} \rceil$ вместо n , и время произведения матриц сократится с $O(n^3)$ до $O(n^2 \cdot \frac{n}{k}) = O(\frac{n^3}{k})$.

Оценка сложности алгоритма и выбор k

Оценим асимптотику данного алгоритма.

- Предподсчёт скалярных произведений всех пар бинарных векторов длины k требует: $O(2^{2k} \cdot k)$.
- Построение сжатых матриц A' и B' занимает: $O(n^2)$.
- Умножение A' и B' — $O\left(\frac{n^3}{k}\right)$.

Суммарная сложность:

$$O(2^{2k}k) + O(n^2) + O\left(\frac{n^3}{k}\right).$$

Оптимальный выбор k достигается при $k = \log n$, поскольку тогда

$$2^{2k} = n^2, \quad 2^{2k}k = O(n^2 \log n),$$

и итоговая сложность равна

$$O(n^2 \log n) + O\left(\frac{n^3}{\log n}\right) = O\left(\frac{n^3}{\log n}\right).$$

Пример

Рассмотрим работу алгоритма на примере перемножения двух матриц A и B , где

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$k = \log_2 n = \log_2 4 = 2,$$

то предподсчитаем все скалярные произведения:

Для удобства каждому битовому вектору будет соответствовать двоичное число с ведущими нулями, т.е. в данном случае имеем числа 00, 01, 10, 11. Ниже приведена таблица, в которой записаны все искомые произведения:

| | 00 | 01 | 10 | 11 |
|----|----|----|----|----|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 1 | 0 | 1 |
| 10 | 0 | 0 | 1 | 1 |
| 11 | 0 | 1 | 1 | 0 |

Согласно соглашению относительно битовых векторов и двоичных чисел получим новые матрицы A' и B' :

$$A' = \begin{pmatrix} 01 & 11 \\ 01 & 00 \\ 11 & 01 \\ 10 & 01 \end{pmatrix}, \quad B' = \begin{pmatrix} 10 & 00 & 01 & 11 \\ 10 & 01 & 10 & 01 \end{pmatrix}$$

Перемножим эти матрицы по модулю два с использованием нашего предподсчета:

$$C = A' \times B' = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Матрица C — искомая.

13 Структуры данных для представления множеств: вектор, список, двоичное дерево поиска. Основные операции и сложность их реализации. АВЛ-деревья, операции над ними, их сложность.

13.1 Структуры данных для представления множеств

Множества представляют собой абстрактный тип данных, предназначенный для хранения элементов произвольного вида, на которых задано отношение порядка \leq (линейный порядок, допускающий эквивалентные элементы).

Различные структуры данных позволяют реализовывать операции над такими множествами

с различной эффективностью.

Поддерживаются базовые операции:

- поиск элемента, эквивалентного данному;
- вставка элемента во множество;
- удаление элемента из множества.

Часто также реализуются дополнительные операции:

- нахождение минимального или максимального элемента;
- нахождение элемента, предшествующего или следующего по порядку данному.

Вектор

Вектор или массив — структура данных, расположенная в последовательных ячейках памяти компьютера. Выделяется блок памяти фиксированного размера.

Динамический вектор — более сложная структура данных, умеющая при переполнении расширяться — для этого выделяется новый блок побольше и в него переносятся все значения.

Поиск: в векторе с n значениями поиск выполняется за время $O(n)$, потому что приходится просмотреть все элементы.

Вставка: можно присвоить элемент в конец, и потому время $O(1)$, коль скоро в векторе есть место.

Удаление: после того, как удаляемый элемент найден (за время $O(n)$), удалить его можно за время $O(1)$, перенеся в освободившееся место последний элемент.

Сортированный вектор

Вектор, элементы которого всегда поддерживаются в сортированном состоянии.

Поиск элемента: двоичный поиск за время $O(\log n)$.

Вставка и удаление: $O(n)$, потому что приходится сдвигать-раздвигать.

Список

Как и вектор, содержит значения в произвольном порядке. Состоит из структур, содержащих один элемент множества, указатель на предыдущий элемент и указатель на следующий элемент.

Поиск: $O(n)$, поскольку надо просмотреть весь список.

Вставка и удаление: после того, как элемент найден, выполняется за время $O(1)$ переключением указателей.

Двоичное дерево поиска

Двоичное дерево поиска (binary search tree, BST) — это структура данных, которая представляет множество в виде дерева.

Каждая вершина дерева содержит одно значение и три указателя:

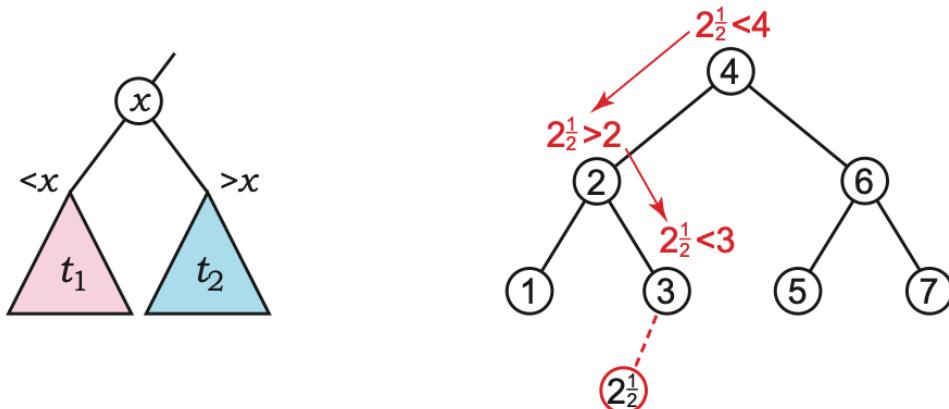
на предка (если это корень, то указатель равен NULL),

на левое поддерево,

на правое поддерево (если поддерева нет, то ставится NULL).

При этом для каждой вершины действует простое правило: все элементы в её левом поддереве имеют значения меньше или равные значению самой вершины, а в правом поддереве — больше или равные. Благодаря этому нужный элемент можно находить быстро: мы начинаем с корня, сравниваем его значение с тем, что ищем, и если они не совпадают, просто спускаемся либо влево, либо вправо — туда, где он может находиться.

Также все элементы дерева можно вывести в порядке возрастания. Для этого достаточно пройти его «слева направо»: сначала левое поддерево, затем текущую вершину, затем правое поддерево. Такой обход удобно делать рекурсивно, но это не обязательно — можно реализовать и без рекурсии, если хранить текущую вершину и понимать, откуда мы в неё пришли.



Двоичное дерево поиска: (слева) условие для каждого поддерева; (справа) поиск элемента $2\frac{1}{2}$ с последующей вставкой.

Другие операции над деревом.

Найти наименьший (наибольший) элемент. Начиная с корня, переходить всё время к левому (правому) потомку. Никаких сравнений не потребуется.

Найти следующий по порядку элемент. На входе — указатель на текущий элемент. Если у него есть правое поддерево, то это — наименьший элемент правого поддерева. Если же правого под дерева нет, то надо подниматься по дереву наверх, и как только текущее дерево окажется левым поддеревом чего-то, следующий элемент найден.

Предыдущий элемент — симметрично.

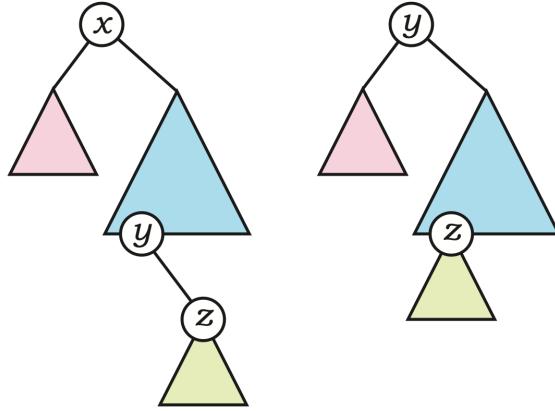
Добавление элемента. Сперва найти место, где он должен был быть — поиском вниз до листа. А далее сделать новый элемент левым или правым потомком этого листа, в зависимости от значения.

Удаление элемента x (дан указатель на него).

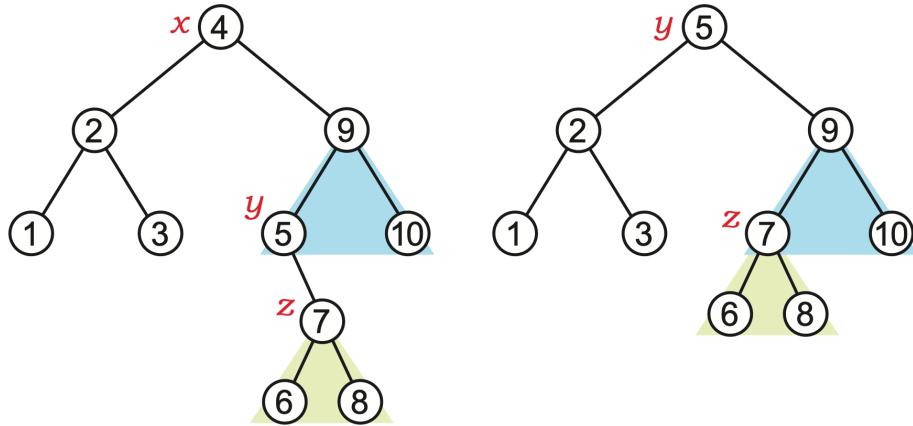
Если у элемента x нет потомков, то его можно просто удалить. Если потомок один, то он занимает место x .

Если потомка два, то, как показано ниже на рисунке, находится элемент y , следующий по порядку за x (он в его правом поддереве). Он и занимает место удаляемого элемента.

Если у y есть правый потомок, то тот в свою очередь занимает освободившееся место, где был y .



Двоичное дерево поиска: удаление вершины x , у которой есть оба потомка.



Удаление в двоичном дереве поиска: удаляется $x = 4$.

Сложность каждой операции — не более чем высота дерева. Если дерево сбалансировано — то есть все пути от корня до листа примерно одинаковой длины — то все операции выполняются за логарифмическое время.

13.2 АВЛ-деревья, операции над ними, их сложность.

АВЛ-деревья — это усложнённая разновидность двоичных деревьев поиска.

АВЛ-деревья «почти сбалансираны»: высота поддеревьев-потомков каждой вершины отличается не более чем на 1. Этого ограничения достаточно, чтобы дерево имело логарифмическую высоту. В каждой вершине хранится высота её поддерева — достаточно даже не высоты, а разности между высотой правого и левого поддеревьев.

Высота АВЛ-дерева. Числа Фибоначчи

Лемма

АВЛ-дерево высоты h содержит не менее чем $F_{h+3} - 1$ вершин, где F_n — n -е число Фибоначчи.

Доказательство. Индукция по h .

База: $h = 0$, это одинокая вершина, и $F_{0+3} - 1 = 1$.

Переход: если дерево имеет высоту h , то один из его потомков имеет высоту $h - 1$, и потому содержит не менее чем $F_{h+2} - 1$ вершину, а другой — высоту не менее чем $h - 2$, и, стало быть, не менее чем $F_{h+1} - 1$ вершину. Всего, с учётом корня, $1 + F_{h+2} - 1 + F_{h+1} - 1 = F_{h+3} - 1$ вершин. \square

Утверждение

$$F_n = \frac{1}{\sqrt{5}}(\varphi^n - \psi^n), \text{ где } \varphi = \frac{1+\sqrt{5}}{2} \text{ и } \psi = \frac{1-\sqrt{5}}{2}$$

Отсюда $F_n = \frac{1}{\sqrt{5}}\varphi^n - o(1)$. Приближённые значения: $\varphi \approx 1.62$ и $\psi \approx -0.62$.

Лемма

Высота АВЛ-дерева с n вершинами не превосходит $\log_\varphi n \approx 1.44 \log_2 n$.

Поэтому при поиске в АВЛ-дереве используется логарифмическое число сравнений — и хотя это число в 1.44 раз хуже теоретически оптимального времени поиска, благодаря этому небольшому ухудшению удается обеспечить выполнение вставок и удалений также за логарифмическое время.

13.3 Операции над АВЛ-деревом

Элемент вставляется, как в обычное двоичное дерево поиска: по пути сравнения ключей спускаемся влево или вправо, пока не найдём пустое место и не создадим там вершину.

Однако после вставки дерево может разбалансироваться, то есть у некоторых вершин высота поддеревьев может начать различаться на 2 (но не более чем на 2). Поэтому после вставки всегда проводится *исправление после вставки*, чтобы дерево осталось правильным АВЛ-деревом.

Элемент удаляется, как из обычного дерева поиска, с последующим *исправлением после удаления*.

Преобразования дерева при балансировке проводятся применением элементарной операции *вращения* (также называемой *поворотом*).

Рассмотрим две соседние вершины x и y , где x — левый потомок вершины y . Обозначим левое и правое поддеревья вершины x через t_1 и t_2 , а правое поддерево вершины y — через t_3 , как изображено на рисунке 4 (левом).

Тогда эту структуру внутри дерева можно изменить, как показано на рисунке 4 (правом), поставив x сверху и опустив y на уровень ниже, так что дерево t_1 приподнимется на уровень, а дерево t_3 , соответственно, опустится. Такое дерево останется правильным деревом поиска, однако высота поддеревьев изменится. Это — вращение между x и y .

Точно так же можно вращать и в обратном направлении. Ниже приведены подробности применения вращения для балансировки дерева после операций вставки и удаления.

13.4 Исправление после вставки

При вставке в дерево добавляется новый лист. Высота поддеревьев на пути от корня к этому листу может увеличиться на единицу, поэтому некоторые вершины на этом пути могут стать разбалансированными: разность высот их левого и правого поддеревьев становится равной 2 по модулю. Во всех остальных частях дерева структура не менялась, поэтому все

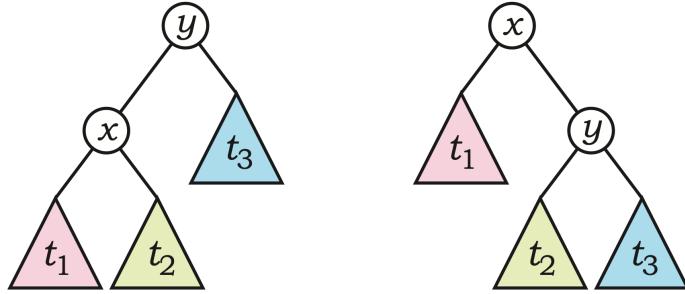


Рис. 4: Вращение между вершинами x и y в AVL-деревьях.

разбалансированные вершины лежат именно на этом пути.

Поиск разбалансированной вершины

Исправление начинается с добавленного листа. Алгоритм поднимается вверх, пока не найдёт самую нижнюю разбалансированную вершину, обозначим её y . Пусть до вставки высота вершины y была равна h , а высоты её двух поддеревьев — $h - 1$ и $h - 2$. Обозначим через x корень более высокого под дерева высоты $h - 1$.

После вставки высота под дерева с корнем в x увеличилась до h , так что вершина y стала разбалансированной. При этом вставка произошла в том под дереве вершины x , которое до вставки имело высоту $h - 2$ и стало иметь высоту $h - 1$. Другой потомок вершины x по-прежнему имеет высоту $h - 2$: если бы его высота отличалась, то дисбаланс возник бы уже в вершине x , а не в y .

Четыре варианта пути

Путь от y к тому под дереву, где произошла вставка и где до вставки была высота $h - 2$, может проходить через левого или правого потомка на каждом из двух уровней. Отсюда возникают четыре случая:

- левый–левый (LL);
- левый–правый (LR);
- правый–левый (RL);
- правый–правый (RR).

Для каждого случая применяется одно или два вращения, после которых под дерево с корнем в y снова становится корректным АВЛ-деревом высоты h .

Случай левый–левый

На рис. 5 показан случай, когда вставка происходит в левом потомке левого потомка вершины y , то есть в её левом–левом потомке. После вставки вершина x получает высоту h , а вершина y — высоту $h + 1$. Под дерево t'_1 , содержащее новый элемент, становится выше двух других, но «подвешено» ниже их уровня.

Для исправления достаточно одного правого вращения между вершинами x и y . В результате под дерево перестраивается так, что все три под дерева t'_1 , t_2 и t_3 оказываются прикреплены

на одном уровне, а всё поддерево с корнем в новой вершине снова имеет высоту h . Тем самым разбалансировка устраняется одной операцией.

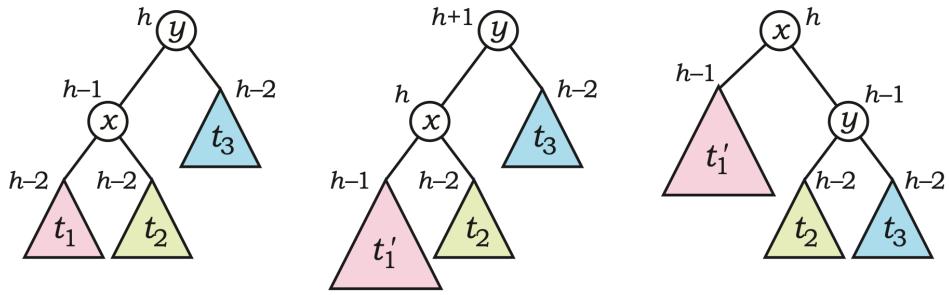


Рис. 5: Исправление после вставки: случай «левый–левый». Слева исходное дерево; посередине — дерево после вставки в поддерево t_1 ; справа — дерево после одного вращения между вершинами x и y .

Случай вставки в правом–правом потомке (*RR*) симметричен описанному: выполняется одно левое вращение.

Случай левый–правый

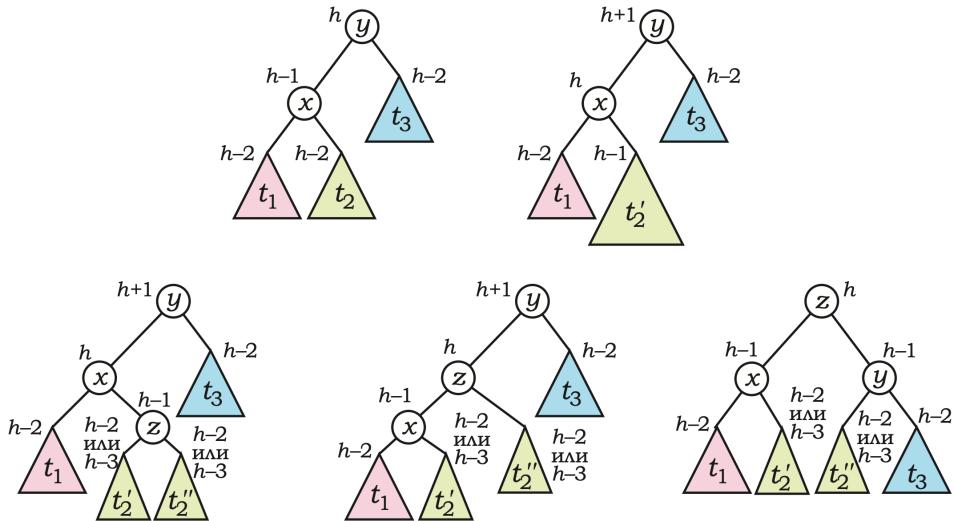


Рис. 6: Исправление после вставки: случай «левый–правый». Сверху: слева исходное дерево до вставки, справа — после вставки. Снизу: слева структура лево-правого потомка z , посередине — после первого вращения, справа — после второго вращения.

Пусть новый элемент вставлен в левом–правом потомке вершины y , как на рис. 6. Обозначим этот потомок через z . После вставки высота вершины z становится равной $h - 1$, а высоты её поддеревьев — либо $h - 2$, либо $h - 3$.

Цель — перестроить локальное поддерево так, чтобы четыре поддерева вокруг z встали на один уровень. Для этого сначала выполняется вращение между вершинами x и z , после чего структура принимает вид, показанный на рис. 6 (снизу посередине). Затем выполняется второе вращение между вершинами z и y .

Итоговое поддерево (рис. 6, снизу справа) является корректным АВЛ-деревом высоты h , поэтому дальнейших исправлений не требуется.

Случай правый–левый (*RL*) полностью симметричен этому: также используется комбинация из двух вращений, только в зеркальном направлении.

Высота дерева после исправления

Во всех четырёх случаях высота исправленного поддерева с корнем в y становится равной его прежней высоте h . Следовательно, выше по дереву разбалансированность уже не возникает.

Отсюда видно, что если вставка приводит к разбалансировке какой-то вершины, то общая высота дерева при этом не увеличивается. Высота всего дерева увеличивается только тогда, когда до вставки у корня высоты h оба под дерева имели высоту $h - 1$ и высота одного из них увеличилась на единицу.

13.5 Исправление после удаления

После удаления вершина убирается из АВЛ-дерева так же, как из обычного дерева поиска. Затем на пути от удалённой вершины к корню ищется самая нижняя разбалансированная вершина высоты h . У неё одно поддерево имеет высоту $h - 1$, а другое, в котором происходило удаление, — высоту $h - 3$. Структура поддерева высоты $h - 1$ определяет, какой из трёх вариантов исправления нужно применить.

Случай 1: поддеревья одинаковой высоты

Пусть у более высокого под дерева высоты $h - 1$ оба дочерних под дерева имеют высоту $h - 2$, как на рис. 7. В этом случае для восстановления условий АВЛ-дерева достаточно одного вращения относительно разбалансированной вершины. После вращения высота рассматриваемого под дерева остаётся равной h , поэтому выше по дереву никаких изменений не требуется.

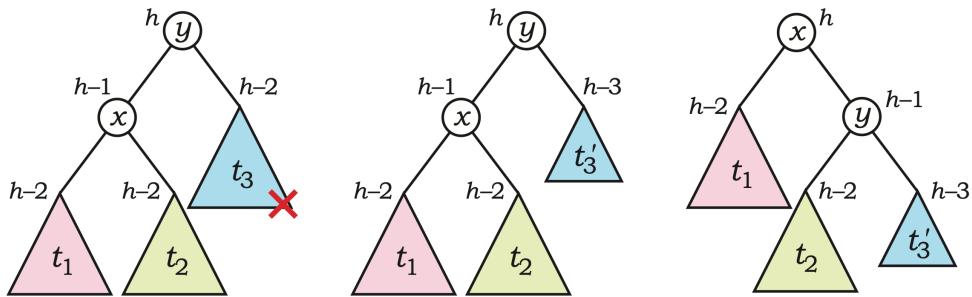


Рис. 7: Исправление после удаления, случай «слева одинаковые», одно вращение: слева исходное дерево; посередине дерево после удаления из t_3 ; справа дерево после вращения.

Случай 2: левый потомок выше правого

Пусть поддерево высоты $h - 1$ расположено слева, и его левый потомок выше правого. Тогда высоты двух поддеревьев этого левого потомка равны $h - 2$ и $h - 3$, как показано на рис. 8. Одного вращения достаточно, чтобы восстановить баланс, но высота исходной вершины уменьшается с h до $h - 1$.

Так как высота уменьшилась, дисбаланс может возникнуть у предков этой вершины. Поэтому алгоритм продолжает подъём вверх по дереву и повторяет те же шаги исправления. В предельном случае он может дойти до корня; уменьшение высоты корня уже не нарушает никаких условий.

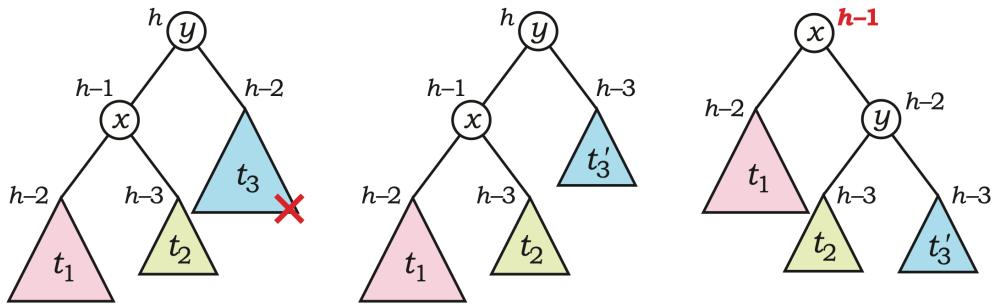


Рис. 8: Исправление после удаления, случай «слева больше левое», одно вращение: слева исходное дерево; посередине дерево после удаления из t_3 ; справа дерево после вращения, при котором высота уменьшилась.

Случай 3: правый потомок выше левого

Пусть поддерево высоты $h - 1$ также левое, но теперь его правый потомок выше левого. Тогда левый-правый потомок имеет высоту $h - 2$ и состоит из двух поддеревьев высоты $h - 3$ или $h - 4$. Как видно на рис. 9, четыре этих поддерева можно выровнять на один уровень с помощью двух последовательных вращений.

После двух вращений получается корректное АВЛ-поддерево, а высота исходной вершины уменьшается до $h - 1$. Дальнейшее исправление аналогично случаю 2: алгоритм поднимается уровнем выше и при необходимости повторяет балансировку.

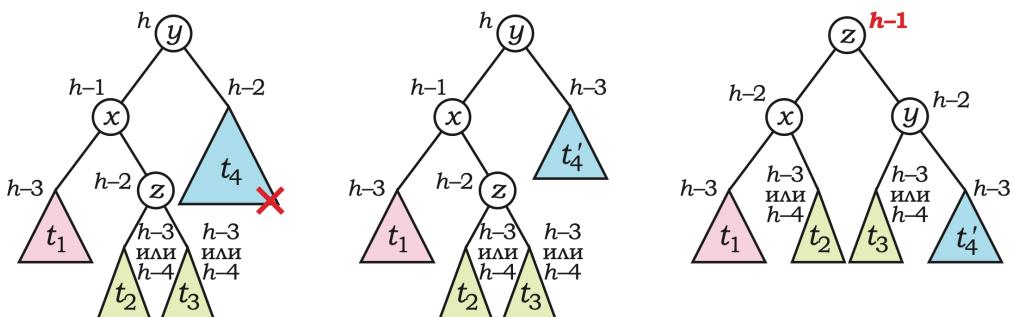


Рис. 9: Исправление после удаления, случай «слева больше правое», два вращения: слева исходное дерево; посередине дерево после удаления из t_4 ; справа дерево после двух вращений, при которых высота уменьшилась.

Общее поведение высоты дерева

Во всех трёх случаях локальное поддерево после исправления либо сохраняет высоту h , либо уменьшает её до $h - 1$. Поэтому при удалении высота всего дерева может уменьшиться, но никогда не увеличивается.

14 В-деревья. Реализация операций над ними, их сложность. Понятие о красно-чёрных деревьях

14.1 В-деревья

Интуитивная идея

Двоичные деревья поиска удобны в оперативной памяти, но плохо работают во внешней: каждый переход к вершине требует чтения блока с диска.

Идея В-дерева:

- сделать вершины *многодетными* (большой степени), так чтобы одна вершина занимала один дисковый блок;
- за счёт этого сильно уменьшить высоту дерева (и число обращений к диску).

Например, если все вершины имеют степень 1000, то для 10^9 ключей высота всего около 3, и путь от корня до листа требует прочитать около 4 блоков.

Структура В-дерева

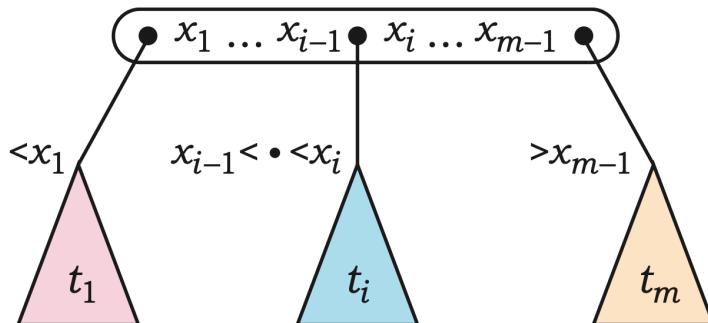
Пусть вершины двоичного дерева называются «*2-вершинами*»: у каждой 2 потомка и 1 значение.

Обобщим это понятие.

- **m -вершина** имеет m поддеревьев t_1, \dots, t_m (возможно, пустых) и $m - 1$ ключей (значений):

$$x_1 < x_2 < \dots < x_{m-1}.$$

Все ключи в поддереве t_i строго больше x_{i-1} и строго меньше x_i (для краёв: всё в t_1 меньше x_1 , всё в t_m больше x_{m-1}).



m -вершина в В-дереве

- В-дерево задаётся параметром $k \geq 2$.
 - Корень может иметь степень от 2 до $2k$, если он не является листом.
 - Каждая другая вершина имеет степень от k до $2k$.
- Балансировка: все пути от корня до листьев имеют одинаковую длину.

Дерево «дышит» за счёт того, что степени вершин меняются от k до $2k$, но высота при этом остаётся логарифмической.

Поиск в В-дереве

Поиск элемента x происходит сверху вниз.

Пусть мы стоим в некоторой m -вершине с ключами

$$x_1 < x_2 < \dots < x_{m-1}$$

и поддеревьями t_1, \dots, t_m .

1. Внутри массива (x_1, \dots, x_{m-1}) выполняем двоичный поиск значения x .
2. Если x найден среди ключей вершины, поиск завершён.
3. Если нет, двоичный поиск определяет индекс i , такой что

$$x_{i-1} < x < x_i,$$

и мы рекурсивно переходим в поддерево t_i .

4. Процесс продолжается до листа.

Число вершин на пути от корня до листа — $O(\log n)$, а поиск внутри одной вершины занимает $O(\log k)$, что считается константой при фиксированном k . Итого: поиск в В-дереве за $O(\log n)$.

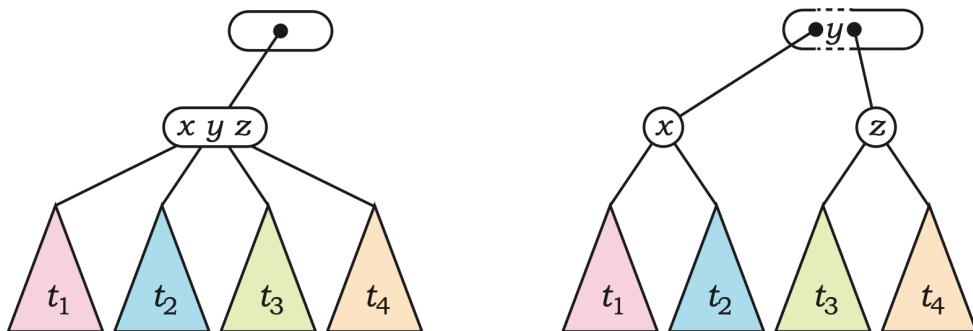
Вставка в В-дерево

В отличие от АВЛ-дерева, где баланс правится *после* вставки (поднимаясь от листа к корню), в В-дереве баланс правится *на спуске*.

Пусть нужно вставить новый ключ в лист.

Идея.

- Никогда не спускаться в *полную* вершину (степени $2k$).
- По пути вниз все встреченные $2k$ -вершины *разделять* на две k -вершины, выталкивая средний ключ вверх.



Разделение вершины при вставке в В-дереве, для $k = 2$: (слева) переполненная 4-вершина; (справа) выталкивание переполнения наверх.

Разделение вершины. Пусть у вершины степени $2k$ ключи

$$x_1 < \dots < x_{2k-1}.$$

- Берём средний ключ x_k и поднимаем его в родительскую вершину.
- Слева остаётся вершина с ключами x_1, \dots, x_{k-1} (степени k).
- Справа — вершина с ключами x_{k+1}, \dots, x_{2k-1} (тоже степени k).

Важно, что когда мы спускаемся из родителя в эту вершину, в родителе *не может* быть степени $2k$: её уже разделили раньше. Значит, для нового ключа x_k в родителе есть свободное место.

Алгоритм вставки.

1. Начинаем с корня.
2. Если текущая вершина полная (степени $2k$), разделяем её.
3. После этого выбираем нужное поддерево и спускаемся в него.
4. Дойдя до листа, вставляем ключ в массив ключей листа (он уже не полон).

Особый случай корня. Если корень был вершиной степени $2k$, то его разделение создаёт новый корень степени 2 , в котором один ключ и две k -дочерние вершины. Высота дерева увеличивается ровно на 1 , и только в этом случае.

Удаление из В-дерева

Удаление также делается *сверху вниз*, при этом на спуске мы следим за тем, чтобы в вершинах не становилось слишком мало ключей.

Пусть параметр дерева равен k .

Инвариант на спуске. При спуске к ребёнку мы гарантируем:

- текущая вершина *не* является k -вершиной;
- если ребёнок является k -вершиной, то мы *усиливаем* его за счёт сестёр (или объединяем с сестрой), прежде чем спускаться дальше.

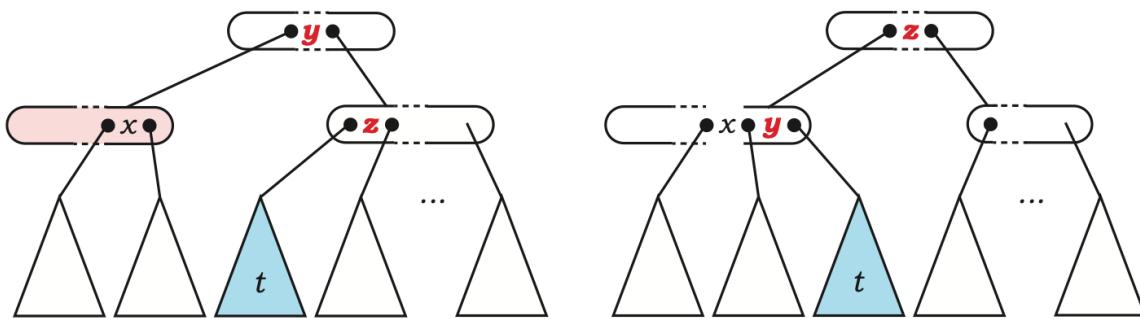
Отсюда следует, что когда мы дойдём до листа, он будет *как минимум* $(k + 1)$ -вершиной и выдержит удаление одного ключа.

Увеличение k -вершины за счёт сестёр. Рассмотрим вершину v степени k . Из инварианта следует, что её родитель имеет степень не меньше $k + 1$, значит, у v есть хотя бы одна сестра.

- Если соседняя сестра имеет степень не меньше $k + 1$, можно *заимствовать* из неё одно поддерево и один ключ родителя (смещение ключей и поддеревьев как на рисунке).
- Если же соседняя сестра — k -вершина, то она объединяется с текущей k -вершиной в одну $2k$ -вершину — это в точности обратная операция к разделению вершины. При этом у родительницы станет на одно поддерево меньше, но она это переживёт, поскольку она не k -вершина.

Удаление ключа. Когда найдена вершина, содержащая удаляемый ключ:

- Если это лист, просто удаляем ключ (лист имеет степень хотя бы $k + 1$).
- Если это внутренняя вершина:



Задействование поддерева у соседней сестры при удалении в В-дереве: (слева) малоимущая k -вершина, выделенная красным; (справа) поддерево заимствовано у сестры справа.

- рассматриваем предыдущее поддерево (левое относительно ключа); если его корень имеет степень как минимум $k + 1$, рекурсивно удаляем *максимальный* ключ в этом поддереве и переносим его на место удаляемого;
- иначе рассматриваем следующее поддерево (правое относительно ключа); если его корень имеет степень как минимум $k + 1$, аналогично берём минимальный ключ;
- если оба соседних поддерева имеют корни – k -вершины, объединяем их вместе с удаляемым ключом в одну $2k$ -вершину и продолжаем процедуру в получившейся вершине.

Особый случай корня. Корень не обязан удовлетворять нижней границе степени. Однако может случиться, что:

- корень является 2-вершиной;
- оба его потомка являются k -вершинами.

Тогда их исправление приведёт к тому, что у корня не останется ключей. В этом случае три вершины (корень и два его ребёнка) объединяются в одну $2k$ -вершину, которая становится новым корнем, и высота дерева уменьшается на 1.

Сложность операций в В-дереве

Во всех операциях (поиск, вставка, удаление):

- мы проходим один путь от корня к листу;
- высота В-дерева с n ключами и параметром k равна $O(\log_k n) = O(\log n)$;
- работа внутри одной вершины требует $O(1)$ машинных операций (число ключей в вершине ограничено константой $2k - 1$).

Поэтому:

$$\text{поиск, вставка, удаление в В-дереве} = O(\log n).$$

14.2 Понятие о красно-чёрных деревьях

2–3–4-деревья как частный случай В-деревьев

В-дерево с параметром $k = 2$ называется 2–3–4-деревом.

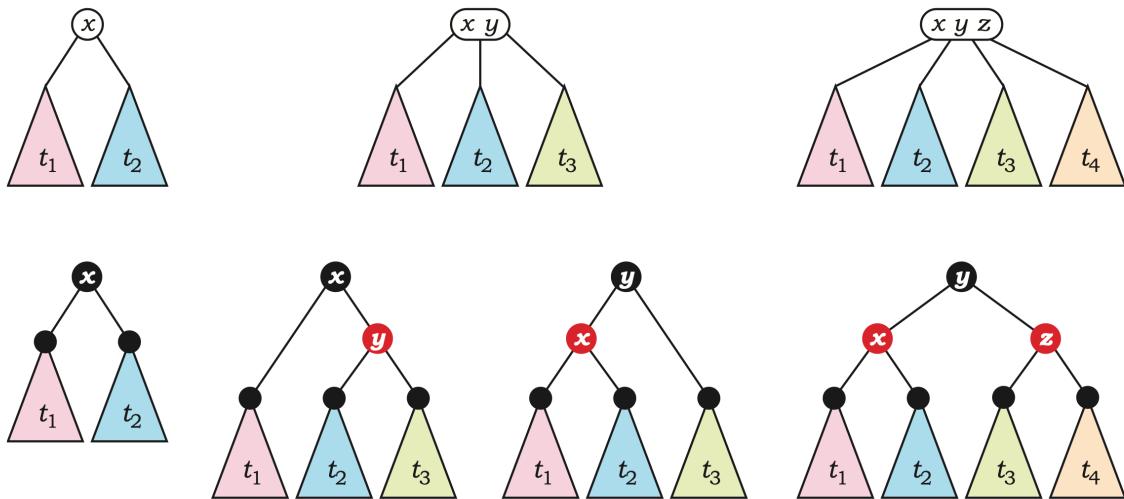
- 2-вершина: 1 ключ, 2 поддерева;
- 3-вершина: 2 ключа, 3 поддерева;

- 4-вершина: 3 ключа, 4 поддерева.

Высота такого дерева меньше, чем у АВЛ-дерева, но в одной вершине приходится делать несколько сравнений, и реализация заметно сложнее.

Представление 2–3–4-дерева в виде красно-чёрного

Идея. Красно-чёрное дерево представляет каждую вершину 2–3–4-дерева набором связанных двоичных вершин, помеченных цветом: красный или чёрный.



Преобразование вершин 2-3-4-деревьев (сверху) во фрагменты красно-чёрных деревьев (снизу)

- Каждая 2-вершина соответствует одной чёрной вершине двоичного дерева.
- Каждая 3-вершина представляется: чёрный корень поддерева + один красный ребёнок (левый или правый); вместе они хранят те же два ключа и три поддерева.
- Каждая 4-вершина представляется: чёрный корень + два красных ребёнка, к которым присоединены те же четыре поддерева.

Верхняя вершина в этом «пучке» всегда чёрная.

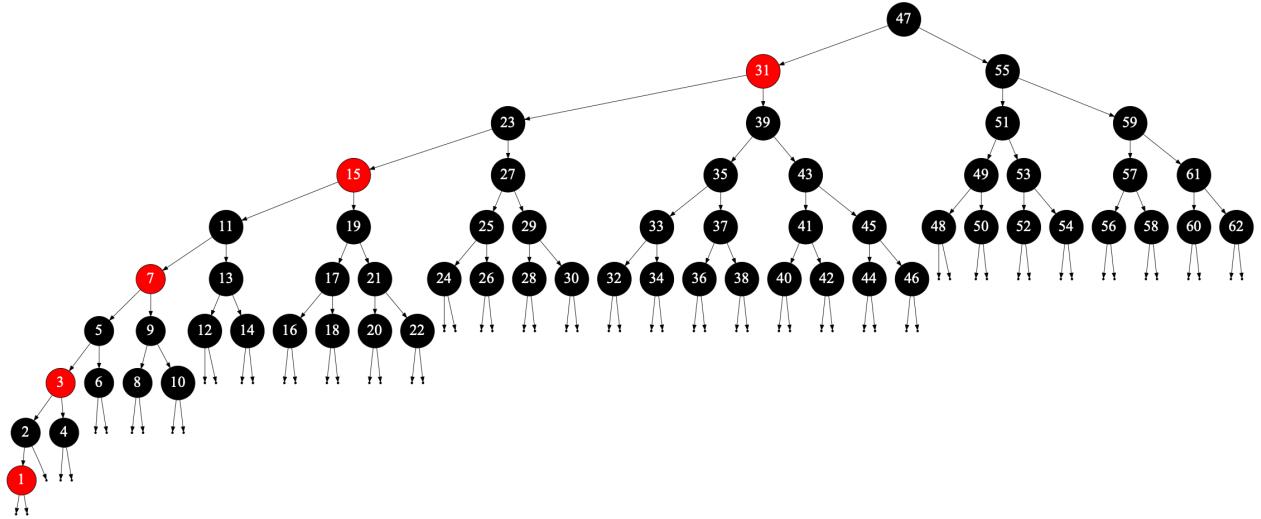
Инварианты красно-чёрного дерева

На красно-чёрное дерево накладываются два основных ограничения:

- **Нет двух красных подряд:** две красных вершины не могут быть соединены ребром. То есть у красной вершины оба ребёнка чёрные.
- **Однаковое число чёрных:** на любом пути от корня к любому листу количество чёрных вершин одинаково (чёрная высота дерева фиксирована).

Из этих инвариантов следует:

- красно-чёрные деревья взаимно однозначно соответствуют 2–3–4-деревьям (с точностью до выбора представления каждой 3-вершины);
- высота красно-чёрного дерева с n вершинами не превосходит $2 \log_2(n + 1)$, то есть все операции выполняются за $O(\log n)$.



Худший случай красно-чёрного дерева — с путём длины порядка $2 \log_2 n$.

Практические замечания

Использование красно-чёрных деревьев вместо прямой реализации 2–3–4-деревьев позволяет:

- работать с обычным двоичным деревом поиска (по два потомка у вершины);
- заменить сложные операции над В-деревом на комбинации *поворотов* и перекрасок в красно-чёрном дереве;
- получить на практике небольшое, но заметное ускорение по сравнению с АВЛ-деревьями.

Поэтому, например, стандартные контейнеры вроде `std::set` и `std::map` в большинстве реализаций стандартной библиотеки C++ построены именно на красно-чёрных деревьях.

15 Полиномиальное хэширование строк. Алгоритм Рабина–Карпа. Нахождение наибольшей общей подстроки. Нахождение самого длинного палиндрома

15.1 Задача поиска подстроки и мотивация хэширования

Пусть задан алфавит Σ и строки над этим алфавитом. Напоминание:

- **подстрока** строки $w = a_1 \dots a_n$ — это $a_i a_{i+1} \dots a_j$ для некоторых $1 \leq i \leq j \leq n$;
- **префикс** — подстрока вида $a_1 \dots a_j$;
- **суффикс** — подстрока вида $a_i \dots a_n$.

Классическая задача:

Поиск подстроки в строке. Дан текст $w = a_1 \dots a_n$ и шаблон $x = b_1 \dots b_m$. Требуется найти все смещения s такие, что

$$w_s = a_{s+1} \dots a_{s+m} = b_1 \dots b_m.$$

Наивный алгоритм

Для каждого $s = 0, 1, \dots, n - m$ посимвольно сравнивать подстроку w_s с x :

- на каждое смещение — до m сравнений;
- всего — до $m(n - m + 1) = O(mn)$ операций.

В худшем случае эта оценка достигается, например, при

$$x = a^{m-1}b, \quad w = a^{n-1}b,$$

где приходится каждый раз почти полностью пробегать подстроку.

Хотим ускорить сравнение подстрок — для этого и вводится хэширование строк.

15.2 Хэш-функции для строк

Идея: сопоставить каждой строке w некоторое число $h(w)$ фиксированного размера (например, 32 или 64 бита).

- Если $h(u) \neq h(v)$, то строки u и v точно *разные*.
- Если $h(u) = h(v)$, строки *могут* быть одинаковыми, а могут и различаться (это называется *коллизия*). В этом случае можно сделать дополнительную посимвольную проверку.

Хочется, чтобы:

- значения $h(w)$ были *равномерно* распределены,
- вероятность коллизии для случайных строк была очень мала.

Плохой пример хэш-функции

Самая простая и одновременно неудачная идея:

$$h(w) = \sum_{i=1}^{\ell} a_i \pmod{2^{32}},$$

где a_i — код i -го символа строки w длины ℓ .

Недостаток: строки, совпадающие по *мультимножеству* символов, но в другом порядке (например, "abc" и "cba") будут иметь один и тот же хэш.

Полиномиальное хэширование

Более удачный подход — интерпретировать строку как коэффициенты многочлена.

Пусть

$$w = a_1 a_2 \dots a_\ell, \quad a_i — \text{числовой код символа.}$$

Выберем:

- основание степени p (часто простое число: 31, 37, 53, 257 и т.п.),
- модуль M — большое число (часто простое: $10^9 + 7$, $10^9 + 9$, либо 2^{64}).

Определим хэш:

$$h(w) = (a_1 p^{\ell-1} + a_2 p^{\ell-2} + \dots + a_\ell p^0) \bmod M.$$

Свойства:

- вычисляется за $\Theta(\ell)$ для строки длины ℓ ;
- строки, отличающиеся на ранних позициях, дают сильно разные значения;
- для случайных строк коллизии редки (вероятность порядка $1/M$).

На практике часто используют ещё и *двойной* хэш (два разных (p, M)), чтобы ещё сильнее уменьшить вероятность коллизий.

Пример

Для иллюстрации пусть $p = 31$, $M = 10^9 + 7$, строка "abc", коды:

$$a = 97, \quad b = 98, \quad c = 99.$$

Тогда

$$\begin{aligned} h(\text{abc}) &= 97 \cdot 31^2 + 98 \cdot 31^1 + 99 \cdot 31^0 \pmod{10^9 + 7} \\ &= 97 \cdot 961 + 98 \cdot 31 + 99 \\ &= 93217 + 3038 + 99 = 96354. \end{aligned}$$

15.3 Алгоритм Рабина–Карпа

Рассматриваем опять задачу поиска подстроки $x = b_1 \dots b_m$ в тексте $w = a_1 \dots a_n$.

Идея алгоритма

Вместо посимвольного сравнения на каждом смещении:

- один раз считаем хэш шаблона x ;
- считаем хэши всех подстрок текста длины m с помощью «скользящего» хеша (rolling hash);
- сравниваем только числа. Если хэши совпали, делаем дополнительную посимвольную проверку.

Определения хешей

Пусть модуль обозначен через q . Тогда:

$$X = \sum_{i=1}^m b_i p^{m-i} \pmod{q}$$

— хэш шаблона x .

Для каждого смещения $s = 0, \dots, n - m$ определим хэш подстроки текста:

$$W_s = \sum_{i=1}^m a_{s+i} p^{m-i} \pmod{q},$$

то есть W_s — хэш подстроки $a_{s+1} \dots a_{s+m}$.

Переход к следующему окну (rolling hash)

Пусть уже известно W_s :

$$W_s = a_{s+1} p^{m-1} + a_{s+2} p^{m-2} + \dots + a_{s+m} p^0.$$

Хотим получить W_{s+1} — хэш подстроки $a_{s+2} \dots a_{s+m+1}$.

1. Домножим W_s на p :

$$pW_s = a_{s+1}p^m + a_{s+2}p^{m-1} + \dots + a_{s+m}p^1.$$

2. Вычтем «лишний» член $a_{s+1}p^m$ (первый символ выпадает из окна).

3. Прибавим новый символ a_{s+m+1} с коэффициентом p^0 .

Получаем формулу:

$$W_{s+1} = (pW_s - a_{s+1}p^m + a_{s+m+1}) \bmod q.$$

Все операции — по модулю q . Переход от W_s к W_{s+1} занимает $O(1)$ арифметических действий.

Полный алгоритм Рабина–Карпа

1. Вычислить X — хэш шаблона.
2. Вычислить W_0 — хэш первой подстроки $a_1 \dots a_m$.
3. Для всех $s = 0, \dots, n - m$:
 - если $W_s = X$, выполнить посимвольное сравнение $a_{s+1} \dots a_{s+m}$ и $b_1 \dots b_m$;
 - если подстроки совпадают — нашли вхождение;
 - затем вычислить W_{s+1} по формуле rolling hash.

Сложность алгоритма Рабина–Карпа

- Подготовка: вычисление X и W_0 занимает $\Theta(m)$.
- Основной проход по окнам: $n - m + 1$ окон, обновление хэша за $O(1)$ и, как правило, без посимвольной проверки.

В среднем, если коллизии редки, время работы:

$$\Theta(n + m).$$

В худшем случае (если почти все окна дают одинаковый хэш и приходится постоянно делать посимвольное сравнение) время может выродиться в $O(mn)$, как и у наивного алгоритма.

15.4 Нахождение наибольшей общей подстроки

Постановка задачи

Даны две строки:

$$u = a_1 \dots a_m, \quad v = b_1 \dots b_n.$$

Найти самую длинную строку x , для которой существуют разбиения

$$u = u_1 x u_2, \quad v = v_1 x v_2.$$

То есть x — *наибольшая общая подстрока* (Longest Common Substring).

Наивный алгоритм

Для каждой пары позиций (i, j) :

- сравнивать символы u_i, v_j , затем u_{i+1}, v_{j+1} и т.д.,
- пока совпадают или пока не выйдем за пределы строки.

Всего пар (i, j) порядка mn , на каждую может уйти до $\min(m, n)$ символов. Общая оценка сверху:

$$O((m + n)^3).$$

Проверка существования общей подстроки длины ℓ

Используем полиномиальное хэширование.

1. Для строки u вычисляем хэши всех подстрок длины ℓ и сохраняем их начальные позиции в некоторой структуре (например, в сбалансированном двоичном дереве поиска или в хэш-таблице), упорядоченной по значению хэша. Это занимает $O(m \log m)$.
2. Затем последовательно рассматриваем все подстроки строки v длины ℓ :
 - для каждой считаем хэш;
 - ищем этот хэш в структуре, построенной по строке u ;
 - если такого хэша нет — значит, таких подстрок в u нет;
 - если хэш найден, сравниваем текущую подстроку v с каждой подстрокой u с тем же хэшем (по позициям из структуры).

При «хороших» хэшах число настоящих коллизий мало, поэтому ожидаемое время процедуры:

$$O((m + n) \log(m + n)).$$

Двоичный поиск по длине

Наконец, используется двоичный поиск по ℓ . Его можно записать в виде рекурсивной процедуры, отвечающей на вопрос «Найти длину наибольшей общей подстроки, если известно, что её длина не меньше, чем ℓ_1 , и строго меньше, чем ℓ_2 ?».

Algorithm 18 Нахождение длины наибольшей общей подстроки двоичным поиском

Процедура $f(\ell_1, \ell_2)$

- 1: **if** $\ell_2 - \ell_1 = 1$ **then**
 - 2: **return** ℓ_1
 - 3: $k = \lfloor \frac{\ell_1 + \ell_2}{2} \rfloor$
 - 4: **if** есть общая подстрока длины k **then**
 - 5: **return** $f(k, \ell_2)$
 - 6: **else**
 - 7: **return** $f(\ell_1, k)$
-

Итоговая сложность:

$$O((m + n)(\log(m + n))^2).$$

Далее в курсе будет показано, что эту задачу можно решать и за $O(m + n)$, используя более сложные структуры.

15.5 Нахождение самого длинного палиндрома

Определения

Обращение строки:

$$(a_1a_2 \dots a_{n-1}a_n)^R = a_na_{n-1} \dots a_2a_1.$$

Строка u называется **палиндромом**, если $u = u^R$.

Задача: по данной строке

$$w = a_1a_2 \dots a_n$$

найти самую длинную её подстроку, являющуюся палиндромом.

Наивный алгоритм

Рассмотреть все возможные центры палиндрома:

- каждый символ как центр нечётного палиндрома,
- каждую «щель» между символами как центр чётного палиндрома;

и для каждого центра расширяться влево и вправо, пока символы совпадают.

Такой алгоритм работает за $O(n^2)$.

Алгоритм с полиномиальными хэшами

Хотим ускориться до примерно $O(n \log n)$.

Идея:

- построить строку w^R — обращение исходной строки;
- с помощью полиномиального хэширования уметь за $O(1)$ получать хэш любой подстроки w и любой подстроки w^R ;
- проверять, существует ли палиндром заданной длины ℓ , сравнивая подходящие подстроки в w и w^R .

Пусть у нас есть префиксные хэши для w и w^R . Тогда:

- подстрока $a_{i+1} \dots a_{i+\ell}$ строки w — палиндром, если она совпадает с строкой $a_{i+\ell} \dots a_{i+1}$;
- строка $a_{i+\ell} \dots a_{i+1}$ — это подстрока строки w^R , начинающаяся с позиции $n - i - \ell + 1$.

Значит, для каждого i можно:

- взять хэш подстроки длины ℓ в w с позиции $i + 1$;
- взять хэш подстроки длины ℓ в w^R с позиции $n - i - \ell + 1$;
- сравнить хэши (и при совпадении при необходимости сделать посимвольную проверку).

Если для какого-то i хэши совпали и посимвольная проверка (если делается) прошла, — значит, палиндром длины ℓ в строке есть.

Проверка для фиксированной ℓ требует всего $O(n)$ времени.

Двоичный поиск по длине палиндрома

Заметим: если строка содержит палиндром длины ℓ , то она содержит палиндром и длины $\ell - 2$ (тот же самый палиндром, но без крайних символов).

Это позволяет применять двоичный поиск по ℓ :

- отдельно по нечётным длинам палиндромов;
- отдельно по чётным длинам палиндромов.

На каждом шаге двоичного поиска вызывается проверка «есть ли палиндром длины $\ell?$ », которая работает за $O(n)$.

Итоговая сложность:

$$O(n \log n).$$

Существуют и более быстрые алгоритмы:

- алгоритм Манахера находит самый длинный палиндром за $O(n)$;
- существуют и варианты на хэшах с линейным временем, но для курса достаточно алгоритма $O(n \log n)$ и знания о существовании линейного.

16 Поиск в строке: алгоритм Кнута–Морриса–Пратта и его реализация на конечном автомате

16.1 Задача и идея алгоритма Кнута–Морриса–Пратта

Пусть дана строка (текст) $w = a_1a_2\dots a_n$ и шаблон (искомая подстрока) $x = b_1b_2\dots b_m$. Требуется найти все вхождения x в w .

Наивный алгоритм для каждого сдвига s сравнивает x с подстрокой $w_s = a_{s+1}\dots a_{s+m}$ по символу. В худшем случае это занимает время $\Theta(mn)$.

Алгоритм Кнута–Морриса–Пратта (КМП) улучшает наивный поиск: он не использует хэширование и работает за линейное время $\Theta(m + n)$.

Рассмотрим пример. Пусть в строке $w = ababaab$ ищется подстрока $x = aba$. Сначала $w_1 = abab$ сравнивается с $x = aba$ и выясняется, что они различны в четвёртом символе:

$$abab \neq aba.$$

Имеет ли смысл после этого сравнивать $w_2 = baba$ с aba ? Нет: раз подстрока w_1 совпала с x в первых трёх символах, а первый и второй символы x различны, это значит, что первый символ w_2 не совпадает с первым символом x . Значит, w_2 точно не подходит. Можно сразу переходить к w_3 .

Идея КМП:

- алгоритм читает текст w слева направо;
- после чтения первых i символов w он хранит число j — длину **самого длинного префикса** строки x , который является **суффиксом** префикса $a_1\dots a_i$,

$$a_{i-j+1}\dots a_i = b_1\dots b_j;$$

- при чтении нового символа a_i алгоритм пытается «продлить» совпадение ещё на один символ: сравнивает a_i с b_{j+1} ;
- если $a_i \neq b_{j+1}$, алгоритм не возвращается назад по тексту, а **уменьшает** j по специальной функции так, чтобы снова получился какой-то префикс x , являющийся суффиксом прочитанного текста, и снова пробует его продлить.

Чтобы уметь быстро «откатывать» j к меньшему подходящему значению, заранее строится **префиксная функция** для строки x .

16.2 Префиксная функция

Определение. Префиксная функция

Пусть $x = b_1 b_2 \dots b_m$ — фиксированная строка.

Префиксная функция (или π -функция) для x — это функция

$$\pi : \{1, \dots, m\} \rightarrow \{0, \dots, m-1\},$$

которая для каждого i даёт длину наибольшего суффикса подстроки $b_1 \dots b_i$, являющегося одновременно префиксом строки x :

$$\pi(i) = \max\{j \mid 0 \leq j < i, b_1 \dots b_j = b_{i-j+1} \dots b_i\}.$$

Иначе говоря, $\pi(i)$ — это длина максимального собственного префикса строки $b_1 \dots b_i$, который одновременно является её суффиксом.

В примере выше для $x = abab$ имеем $\pi(3) = 1$: у префикса aba единственный нетривиальный префикс-суффикс — это строка «а».

Повторяя применение π , можно получить все длины префиксов x , которые являются суффиксами $b_1 \dots b_i$:

$$j, \pi(j), \pi(\pi(j)), \dots$$

Именно по этой цепочке алгоритм КМП «откатывает» j , пока не найдёт префикс, который можно продолжить текущим символом текста.

Пример

Для строки $x = abaaba$ посчитаем значения префиксной функции π .

- $i = 1, x[1..1] = a$.

Нетривиальных префиксов-суффиксов нет, поэтому $\pi(1) = 0$.

- $i = 2, x[1..2] = ab$.

Суффиксы: **b**, **ab**. Префиксы длины < 2 : только **a**.

Совпадений нет, значит $\pi(2) = 0$.

- $i = 3, x[1..3] = aba$.

Суффиксы: **a**, **ba**, **aba**.

Префиксы длины < 3 : **a**, **ab**.

Совпадает только **a** длины 1, поэтому $\pi(3) = 1$.

- $i = 4, x[1..4] = abaa$.

Суффиксы: **a**, **aa**, **baa**, **abaa**.

Префиксы длины < 4 : **a**, **ab**, **aba**.

Совпадает только **a**, значит $\pi(4) = 1$.

- $i = 5, x[1..5] = abaab$.

Суффиксы: **b**, **ab**, **aab**, **baab**, **abaab**.

Префиксы длины < 5 : **a**, **ab**, **aba**, **abaa**.

Совпадает **ab** длины 2, поэтому $\pi(5) = 2$.

- $i = 6, x[1..6] = abaaba$.

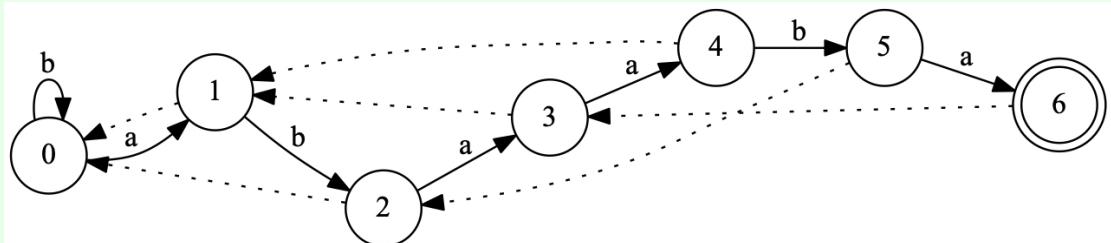
Суффиксы: **a**, **ba**, **aba**, **aaba**, **baaba**, **abaaba**.

Префиксы длины < 6 : **a**, **ab**, **aba**, **abaa**, **abaab**.

Совпадает **aba** длины 3, значит $\pi(6) = 3$.

Итого:

$$\pi(1) = 0, \quad \pi(2) = 0, \quad \pi(3) = 1, \quad \pi(4) = 1, \quad \pi(5) = 2, \quad \pi(6) = 3.$$



16.3 Алгоритм Кнута–Морриса–Пратта

Будем считать, что префиксная функция π для строки x уже построена.

Algorithm 19 Алгоритм Кнута–Морриса–Пратта

Require: текст $w = a_1 \dots a_n$, шаблон $x = b_1 \dots b_m$, массив $\pi[1..m]$

```

1:  $j = 0$                                       $\triangleright$  длина совпавшего префикса  $x$ 
2: for  $i = 1$  to  $n$  do
3:   while  $j > 0$  и  $b_{j+1} \neq a_i$  do
4:      $j = \pi(j)$                           $\triangleright$  переход к более короткому префиксу
5:   if  $b_{j+1} = a_i$  then
6:      $j = j + 1$                             $\triangleright$  префикс  $x$  успешно продлился
7:   if  $j = m$  then
8:     сообщить: подстрока найдена по смещению  $i - m$ 
9:    $j = \pi(j)$                             $\triangleright$  ищем следующее вхождение

```

Инвариант. В начале каждой i -й итерации цикла (перед строкой 7) выполнено: наибольший префикс x , являющийся суффиксом $a_1 \dots a_i$, равен $b_1 \dots b_j$.

Лемма

Имея готовую таблицу значений префиксной функции π , алгоритм КМП выполняется за время $\Theta(n)$, где n — длина текста w .

Доказательство. На первый взгляд, строчка 4 при неудачном стечении обстоятельств может выполняться $m n$ раз. Но это не так.

Каждый раз, когда выполняется строка 4, значение j уменьшается, а $j \geq 0$ всегда. За всё время работы алгоритма j увеличивается не более чем на n раз (строка 6), поэтому уменьшаться оно тоже может не более n раз. Значит, внутренний цикл (строка 4) выполняется не более n раз за всю работу алгоритма. Остальные действия занимают очевидно линейное время. \square

16.4 Построение префиксной функции

Осталось научиться быстро строить саму префиксную функцию π для строки $x = b_1 \dots b_m$.

Algorithm 20 Построение префиксной функции

Require: строка $x = b_1 \dots b_m$

- 1: $\pi(1) = 0$
- 2: $j = 0$
- 3: **for** $i = 2$ **to** m **do** ▷ обрабатываем символ b_i
- 4: **while** $j > 0$ **и** $b_{j+1} \neq b_i$ **do** ▷ сжимаем префикс
- 5: $j = \pi(j)$
- 6: **if** $b_{j+1} = b_i$ **then** ▷ можем продлить совпадение
- 7: $j = j + 1$
- 8: $\pi(i) = j$

После каждой i -й итерации цикла значение $\pi(i)$ вычислено корректно, и текущее j всегда равно $\pi(i)$. Время работы алгоритма — $\Theta(m)$ по той же схеме оценки, что и для основного алгоритма КМП.

В итоге: подготовка (вычисление π) занимает время $\Theta(m)$, а затем поиск шаблона x в любом тексте w — время $\Theta(n)$. Совокупная сложность $\Theta(m + n)$.

16.5 Поиск с помощью конечных автоматов

Детерминированные конечные автоматы

Определение. Детерминированный конечный автомат

(Клини). Детерминированный конечный автомат (deterministic finite automaton, DFA) — это пятёрка

$$\mathcal{A} = (\Sigma, Q, q_0, \delta, F),$$

где:

- Σ — конечный алфавит;
- Q — конечное множество состояний;
- $q_0 \in Q$ — начальное состояние;
- $\delta : Q \times \Sigma \rightarrow Q$ — функция переходов: если автомат находится в состоянии $q \in Q$ и читает символ $a \in \Sigma$, то следующее состояние равно $\delta(q, a)$;
- $F \subseteq Q$ — множество **принимающих состояний**.

Для любой входной строки $w = a_1 \dots a_\ell$ (где $\ell \geq 0$, $a_i \in \Sigma$) вычисление — это последовательность состояний p_0, \dots, p_ℓ :

$$p_0 = q_0, \quad p_i = \delta(p_{i-1}, a_i), \quad i = 1, \dots, \ell.$$

Строка **принимается**, если $p_\ell \in F$, и **отвергается** иначе.

Множество всех принимаемых строк обозначают $L(\mathcal{A})$ и называют **языком**, распознаваемым автоматом \mathcal{A} .

Множество строк в теоретической информатике называют **формальным языком** (formal language) или просто **языком** (language).

На практике DFA удобно реализовать таблицей значений функции δ , хранящейся в массиве: при чтении каждого символа делается один доступ к таблице и изменяется одна переменная — текущее состояние автомата.

Конечный автомат, реализующий алгоритм Кнута–Морриса–Пратта

Идея: состояние автомата будет хранить то же самое число j , что и алгоритм КМП — длину самого длинного префикса x , уже совпавшего с суффиксом прочитанной части текста. Тогда один переход автомата соответствует одной итерации КМП.

Теорема

Для любой строки $x = b_1 \dots b_m$ над алфавитом Σ существует детерминированный конечный автомат с $m + 1$ состояниями, распознающий язык Σ^*x (все строки, оканчивающиеся на x). Такой автомат можно построить за время $\Theta(|\Sigma| \cdot m)$.

Идея построения.

- Сначала строим префиксную функцию π для строки x .
- Берём множество состояний

$$Q = \{0, 1, \dots, m\},$$

где состояние j означает: уже прочитан суффикс входной строки длины j , совпадающий с префиксом $b_1 \dots b_j$.

- Начальное состояние $q_0 = 0$ соответствует пустому суффиксу.

Переходы:

$$\delta(0, a) = \begin{cases} 1, & b_1 = a, \\ 0, & b_1 \neq a; \end{cases}$$

для $1 \leq j < m$:

$$\delta(j, a) = \begin{cases} j + 1, & b_{j+1} = a, \\ \delta(\pi(j), a), & b_{j+1} \neq a; \end{cases}$$

и для полного совпадения ($j = m$):

$$\delta(m, a) = \delta(\pi(m), a).$$

Принимающее множество:

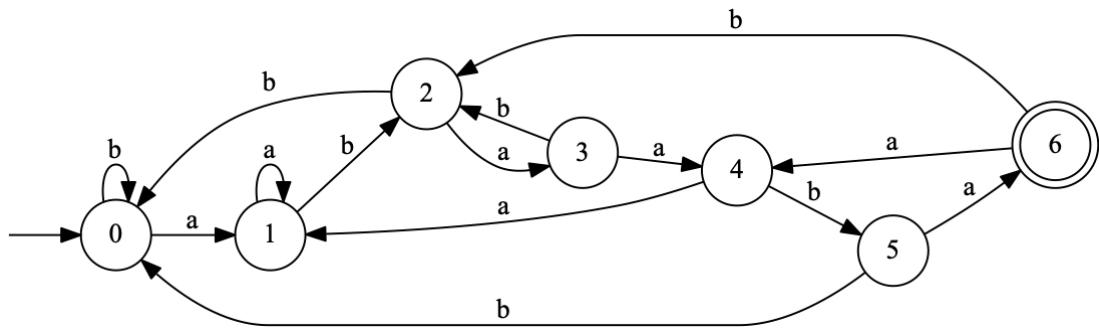
$$F = \{m\}.$$

Таким образом, на каждом шаге состояние автомата хранит ту же величину j , что и алгоритм КМП: длину максимального префикса x , совпадающего с хвостом прочитанной части входа. Поэтому состояние m достигается тогда и только тогда, когда последний суффикс входного слова равен всей строке x , то есть когда вход лежит в языке Σ^*x . Это даёт корректность распознавания.

Оценка времени построения таблицы переходов также повторяет анализ КМП. Префиксная функция π строится за $\Theta(m)$. Затем для каждого символа $a \in \Sigma$ можно последовательно вычислить значения $\delta(j, a)$ для $j = 0, 1, \dots, m$, используя те же переходы по $\pi(j)$ при несовпадениях. Каждое значение j для фиксированного символа может увеличиться и уменьшиться ограниченное число раз, поэтому на один символ алфавита требуется $\Theta(m)$ времени, а на весь алфавит — $\Theta(|\Sigma| \cdot m)$.

Пример

Пусть $x = abaaba$. Тогда префиксная функция равна $\pi(1) = 0, \pi(2) = 0, \pi(3) = 1, \pi(4) = 1, \pi(5) = 2, \pi(6) = 3$ (см. Пример 16.2). По этим значениям строится автомат с $m+1 = 7$ состояниями.



Такой автомат посимвольно читает текст слева направо и в точности повторяет работу алгоритма КМП, но уже в терминах конечного автомата: состояние j соответствует длине совпавшего префикса шаблона, а достижение состояния t означает, что очередное вхождение шаблона в текст найдено.

17 Префиксное дерево для множества строк. Алгоритм Ахо–Корасик

Пусть дан текст

$$w = a_1 a_2 \dots a_n$$

над алфавитом Σ и конечное множество строк

$$K = \{x_1, x_2, \dots, x_k\} \subseteq \Sigma^*$$

Нужно найти все вхождения каждой строки x_j в текст w .

17.1 Постановка задачи и идея алгоритма

При $k = 1$ задачу решает алгоритм Кнута–Морриса–Пратта (КМП): после чтения первых i символов текста он знает длину самого длинного префикса строки x_1 , который совпадает с суффиксом $a_1 \dots a_i$.

Теперь k может быть больше 1. Алгоритм Ахо–Корасик делает примерно то же самое, но сразу для множества строк K :

- после обработки первых i символов текста алгоритм хранит некоторую строку u ;
- эта строка u :
 - является префиксом некоторой строки из K ;
 - совпадает с суффиксом $a_1 \dots a_i$;
 - и среди всех таких строк имеет максимальную длину.

Другими словами, в каждый момент мы знаем *самый длинный подходящий «кусок»* из множества K , который сейчас прочитан на конце текста.

Чтобы эффективно поддерживать этот префикс u , используются:

- префиксное дерево (trie) для множества K ;
- обобщённая префиксная функция π (так называемые *ссылки неудачи*).

17.2 Префиксное дерево

Сначала введём структуру для хранения множества строк.

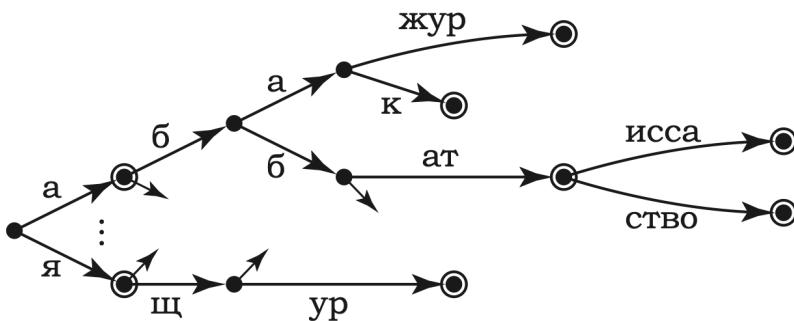
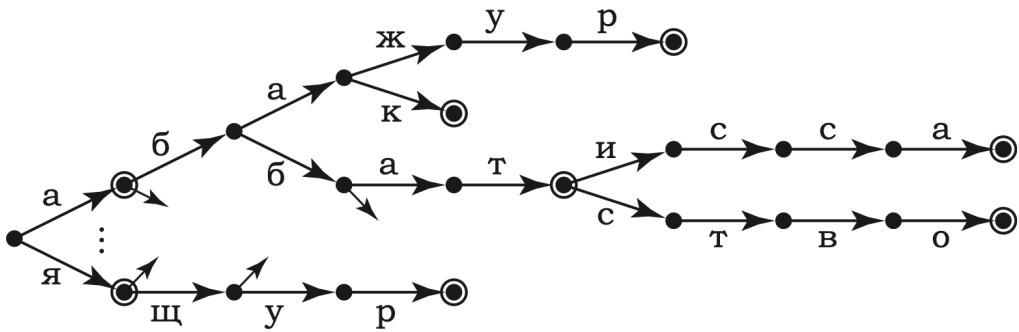
Определение.

Пусть Σ — конечный алфавит. *Префиксное дерево* (prefix tree, trie) для множества строк $S \subseteq \Sigma^*$ — это корневое дерево, в котором:

- рёбра помечены символами алфавита Σ ;
- корень соответствует пустой строке ε ;
- каждой вершине однозначно соответствует строка w — конкатенация надписей на рёбрах по пути от корня до этой вершины;
- в вершине храним бит: принадлежит ли соответствующая строка w множеству S (при необходимости вместе с дополнительной информацией о w).

Если вершина соответствует строке w , а из неё есть ребро по символу $a \in \Sigma$, то это ребро ведёт в вершину, соответствующую строке wa .

Для любой строки длины ℓ операции поиска, вставки и удаления в префиксном дереве выполняются за время $O(\ell)$ и не зависят от числа строк в множестве S .



(сверху) Префиксное дерево и (снизу) компактное префиксное дерево

В контексте алгоритма Ахо–Корасик префиксное дерево строится для множества K искомых строк; каждая вершина соответствует некоторому префиксу строки из K .

17.3 Обобщённая префиксная функция

Интуитивно: мы находимся в вершине префиксного дерева, соответствующей строке x . Следующий символ текста — a . Если в дереве есть ребро по a , переходим к вершине, соответствующей строке xa . Если ребра нет, надо «откатиться» к более короткому префиксу

какой-то строки из K и попробовать прочитать символ a оттуда. Именно это и кодирует функция π .

Сначала обозначим $\text{prefixes}(K) = \{\text{все префиксы всех строк из } K\} \cup \{\varepsilon\}$.

Определение.

Префиксная функция для множества строк $K = \{x_1, \dots, x_k\}$ — это функция

$$\pi : \text{prefixes}(K) \rightarrow \text{prefixes}(K),$$

такая что для любого $x \in \text{prefixes}(K)$, $x \neq \varepsilon$, значение $\pi(x)$ — это наибольший по длине *собственный* суффикс x , который одновременно является префиксом некоторой строки из K :

$$\pi(x) = \max \{ x' \mid |x'| < |x|, x' — \text{суффикс } x, x' \in \text{prefixes}(K) \}.$$

Для корня полагаем $\pi(\varepsilon) = \varepsilon$.

То есть:

- $\pi(x)$ говорит, в какую вершину префиксного дерева нужно перейти, если чтение следующего символа из вершины x завершилось неудачей;
- в префиксном дереве значение $\pi(x)$ удобно изображать пунктирным ребром из вершины x в вершину, расположенную ближе к корню.

Каждая вершина префиксного дерева также помечается *метками строк* из K — это строки, которые полностью заканчиваются в этой вершине. Метки «наследуются» по ссылкам π : если в вершине $\pi(x)$ распознаётся строка x_j , то она считается распознанной и в вершине x .

17.4 Поиск в тексте: алгоритм Ахо–Корасик

Теперь, когда префиксное дерево и функция π построены для множества K , можно искать все вхождения строк из K в тексте w .

Algorithm 21 Алгоритм Ахо–Корасик

множество строк K , префиксное дерево и функция π уже построены

```

1:  $u = \varepsilon$                                      ▷ корень префиксного дерева
2: for  $i = 1$  to  $n$  do                   ▷ читаем очередной символ  $a_i$ 
3:   while  $u \neq \varepsilon$  и из  $u$  нет ребра по символу  $a_i$  do
4:      $u = \pi(u)$                            ▷ откат по префиксным ссылкам
5:   if из  $u$  есть ребро по символу  $a_i$  then
6:      $u = ua_i$ 
7:   for all строк  $x_j \in K$ , помеченных в вершине  $u$  do
8:     сообщить о вхождении  $x_j$ , заканчивающемся в позиции  $i$ 
9:   (начало в позиции  $i - |x_j| + 1$ )

```

Инвариант: после обработки i -го символа текста вершина u соответствует самому длинному префиксу некоторой строки из K , который совпадает с суффиксом $a_1 \dots a_i$.

Как и в алгоритме КМП, каждый символ текста обрабатывается за $O(1)$ амортизированно: шаги по π уменьшают длину текущей строки и в сумме дают $O(n)$. Время поиска: $O(n + z)$, где z — число всех найденных вхождений.

17.5 Построение функции π

Осталось построить значения π для всех вершин префиксного дерева. Будем делать это в порядке увеличения длины строки (обход дерева в ширину).

Algorithm 22 Построение префиксной функции π для множества K

Require: множество строк $K = \{x_1, \dots, x_k\}$ над алфавитом Σ

```

1: построить префиксное дерево для  $K$ ; корень соответствует строке  $\varepsilon$ 
2:  $\pi(\varepsilon) = \varepsilon$ 
3: создать пустую очередь  $Q$ 
4: for all  $a \in \Sigma$  do
5:   if в дереве есть вершина  $a$  (ребро из корня по  $a$ ) then
6:      $\pi(a) = \varepsilon$ 
7:     добавить вершину  $a$  в очередь  $Q$ 
8:   else
9:     добавить в корне петлю по символу  $a$  (переход из корня в корень по  $a$ )
10:  while  $Q$  не пуста do
11:    извлечь вершину  $u$  из начала  $Q$ 
12:    for all  $a \in \Sigma$  do
13:      if в дереве есть вершина  $ua$  (ребро из  $u$  по  $a$ ) then
14:         $v = \pi(u)$ 
15:        while в дереве нет вершины  $va$  do
16:           $v = \pi(v)$ 
17:         $\pi(ua) = va$ 
18:        к меткам вершины  $ua$  добавить все метки вершины  $va$ 
19:        добавить вершину  $ua$  в конец очереди  $Q$ 

```

Обозначим через

$$m = \sum_{j=1}^k |x_j|$$

суммарную длину всех строк из K .

Лемма

Алгоритм построения функции π работает за время $O(m)$, то есть линейно по суммарной длине строк из множества K .

Идея доказательства. Рассмотрим любую строку $x = a_1 \dots a_\ell \in K$ и соответствующую ей цепочку вершин

$$u_1 = a_1, u_2 = a_1 a_2, \dots, u_\ell = a_1 \dots a_\ell.$$

Каждая вершина u_i попадает в очередь ровно один раз, и для неё значение $\pi(u_i)$ вычисляется один раз.

Внутренний цикл

$$v = \pi(v)$$

на каждом шаге уменьшает длину строки, соответствующей v . Поэтому по каждой вершине можно «спуститься» по ссылкам π ограниченное число раз — не больше её длины. Суммарное число таких шагов по всем вершинам не превосходит $O(m)$.

Остальные действия алгоритма занимают константное время на вершину, поэтому общая сложность построения π равна $O(m)$. \square

Итог.

- Построение префиксного дерева и функции π занимает время $O(m)$.
- Поиск всех вхождений в тексте длины n занимает время $O(n + z)$.

В целом алгоритм Ахо–Корасик работает за время $O(n + m + z)$ и требует памяти $O(m)$.

18 Сжатие данных методом Хаффмана. Арифметическое кодирование

18.1 Постановка задачи сжатия

Задача: построить функцию

$$f: \Sigma^* \rightarrow \Sigma^*,$$

которая

- переводит разные строки в разные (отображение инъективно, код однозначно декодируется);
- переводит «типичные» (часто встречающиеся) строки в более короткие строки.

Очевидно, нельзя сделать короче *каждую* строку, но можно добиться выигрыша на реальных данных.

Обычно исходный алфавит Σ — это байты:

$$\Sigma = \{0, 1, \dots, 255\}.$$

Для описания двоичного кодирования удобно дополнительно выделить целевой алфавит

$$\Omega = \{0, 1\},$$

и говорить о кодировании гомоморфизмом $h: \Sigma^* \rightarrow \Omega^*$.

18.2 Гомоморфизмы и коды

Даны два алфавита: исходный Σ и целевой Ω .

Определения.

Рассматриваются слова над алфавитами Σ^* и Ω^* .

- **Гомоморфизм**

$$h: \Sigma^* \rightarrow \Omega^*$$

задаётся образами всех символов: для каждого $a \in \Sigma$ задано слово $h(a) \in \Omega^*$.

Для любых строк $u, v \in \Sigma^*$ выполняется

$$h(uv) = h(u)h(v).$$

- Гомоморфизм h называется **кодом**, если по $h(w)$ можно однозначно восстановить w ; то есть

$$u \neq v \Rightarrow h(u) \neq h(v).$$

- Гомоморфизм h называется **префиксным кодом**, если образ ни одного символа не является префиксом образа другого:

$$\forall a \neq b \in \Sigma, \forall x \in \Omega^*: h(a) \neq h(b)x.$$

Тогда можно декодировать слева направо: первый символ исходного слова однозначно определяется по первым k символам кода, где k — некоторая константа.

Пример

- $\Sigma = \{a, b, c\}$, $\Omega = \{0, 1\}$, $h(a) = 0$, $h(b) = 01$, $h(c) = 10$.

Это *не* код, потому что

$$h(ac) = 010 = h(ba).$$

- $\Sigma = \{a, b, c\}$, $\Omega = \{0, 1\}$, $h(a) = 0$, $h(b) = 10$, $h(c) = 11$.

Это **префиксный** код: никакой образ не является префиксом другого.

- $\Sigma = \{a, b, c\}$, $\Omega = \{0, 1\}$, $h(a) = 0$, $h(b) = 01$, $h(c) = 11$.

Это код, но не префиксный: разные строки однозначно декодируются, если читать код *справа налево*, но, например,

$$h(accc) = 01111111, \quad h(bccc) = 01111111.$$

18.3 Код Хаффмана

Идея: часто встречающиеся символы кодировать короткими двоичными словами, а редкие — длинными, причём код остаётся префиксным.

Пусть Σ — алфавит, $w \in \Sigma^*$ — строка. Через $|w|_a$ обозначим число вхождений символа a в w .

Определение.

Код Хаффмана для строки w — это префиксный код $h: \Sigma \rightarrow \{0, 1\}^*$, построенный по частотам $|w|_a$ следующим образом.

Построение дерева Хаффмана.

- Для каждого символа $a \in \Sigma$, встречающегося в w , создаём лист (одиночное дерево) с весом $|w|_a$.
- Пока деревьев больше одного:
 - берём два дерева с наименьшими весами корней;
 - соединяем их новым корнем, вес корня равен сумме двух весов;
 - ребру к одному поддереву приписываем бит 0, к другому — бит 1.
- В полученном двоичном дереве каждый символ соответствует листу; его код — последовательность нулей и единиц на пути от корня к листу.

Полученный код всегда префиксный: пути до разных листьев не могут быть префиксами друг друга.

Хранить поддеревья удобно в очереди с приоритетами; тогда время построения

$$O(|\Sigma| \log |\Sigma|).$$

Теорема

Пусть Σ — алфавит, $w \in \Sigma^*$ — строка. Тогда код Хаффмана — префиксный код, минимизирующий длину $h(w)$ для строки w .

Доказательство. Доказательство по индукции по размеру алфавита $|\Sigma|$.

База. $|\Sigma| = 2$. Тогда единственный префиксный код — закодировать один символ нулём, другой единицей. Кратче быть не может.

Переход. Пусть теорема верна для любых алфавитов размера $|\Sigma| - 1$. Возьмём два самых редких символа $a, b \in \Sigma$. Сольём их в новый «супер-символ» \hat{c} , заменив в строке w все вхождения a и b на \hat{c} . Получим строку w' над алфавитом $\Sigma' = \Sigma \setminus \{a, b\} \cup \{\hat{c}\}$.

По предположению индукции код Хаффмана h' для w' оптимальен среди всех префиксных кодов над Σ' . При построении кода Хаффмана для исходного Σ пары a, b на первом шаге также склеиваются в общее поддерево, а далее дерево строится так же, как для w' . Отсюда

$$h(a) = h'(\hat{c})0, \quad h(b) = h'(\hat{c})1, \quad h(c) = h'(c) \ (c \neq a, b).$$

Значит, код h для строки w длиннее кода h' для w' ровно на $|w|_a + |w|_b$ бит:

$$|h(w)| = |h'(w')| + |w|_a + |w|_b.$$

Пусть теперь g — любой другой префиксный код для Σ . Можно перестроить его в код \tilde{g} так, чтобы коды a и b отличались только последним битом ($\tilde{g}(a) = x0$, $\tilde{g}(b) = x1$), не увеличивая длину кода: $|g(w)| \geq |\tilde{g}(w)|$. Тогда по \tilde{g} строим код g' для алфавита Σ' , положив $g'(\hat{c}) = x$ и $g'(c) = \tilde{g}(c)$ для остальных c . Получаем

$$|\tilde{g}(w)| = |g'(w')| + |w|_a + |w|_b.$$

По оптимальности h' имеем $|g'(w')| \geq |h'(w')|$, поэтому

$$|g(w)| \geq |\tilde{g}(w)| = |g'(w')| + |w|_a + |w|_b \geq |h'(w')| + |w|_a + |w|_b = |h(w)|.$$

То есть ни один префиксный код не короче кода Хаффмана. \square

18.4 Арифметическое кодирование

Бывает, что код Хаффмана «упирается» в целое число бит на символ. Например, $\Sigma = \{a, b\}$ и a встречается вдвое чаще b ; Хаффман всё равно даст им коды одинаковой длины (по 1 биту) и сжатия не получится.

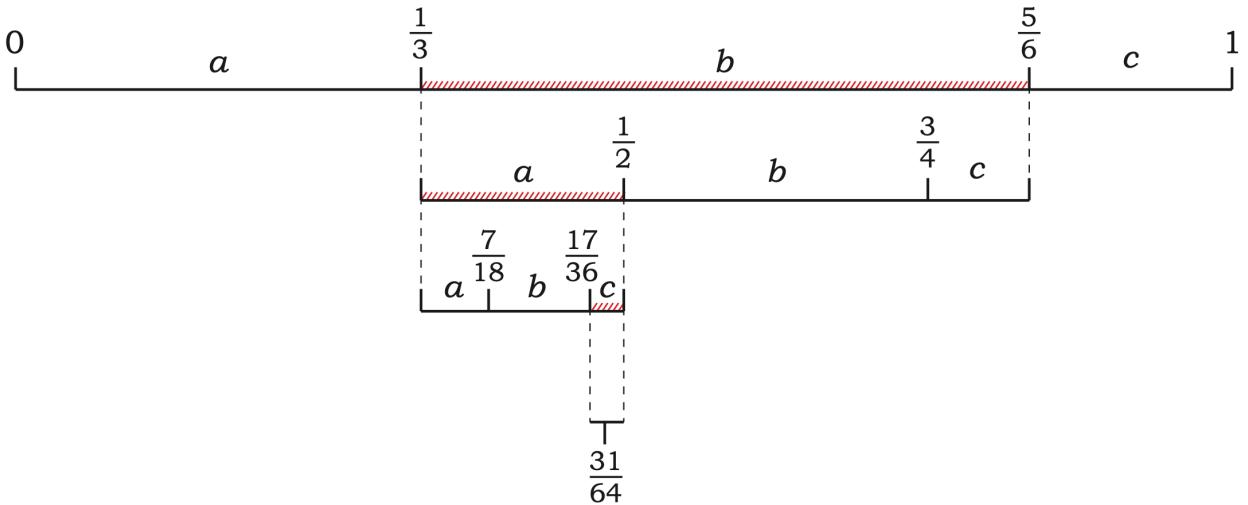
Идея арифметического кодирования.

- Код всего сообщения — одно число $x \in [0, 1)$ (рациональное в теории).
- Алгоритм хранит текущий интервал $[\ell, m) \subseteq [0, 1)$.
- Читая очередной символ, мы **сужаем** интервал до подотрезка, длина которого пропорциональна вероятности этого символа.

Пусть алфавит Σ **линейно упорядочен** (например, $a < b < c$), и каждому символу $a \in \Sigma$ задана вероятность $P(a) \in \mathbb{Q}$, причём $\sum_{a \in \Sigma} P(a) = 1$. Если текущий интервал $[\ell, m)$ и прочитан символ $a \in \Sigma$, то делим $[\ell, m)$ на куски в порядке возрастания символов и берём кусок для a :

$$[\ell, m) \longrightarrow \left[\ell + (m - \ell) \sum_{b < a} P(b), \ell + (m - \ell) \sum_{b \leq a} P(b) \right).$$

Стартуем с $[\ell, m) = [0, 1)$ и последовательно обрабатываем все символы строки. В конце можно выбрать любое $x \in [\ell, m)$ и выдать **достаточное число первых битов** двоичной записи x , чтобы однозначно попасть внутрь этого интервала; также запоминается длина строки n .



Пример работы арифметического кодирования, строка $w = bac$ (пример).

Пример

$P(a) = \frac{1}{3}$, $P(b) = \frac{1}{2}$, $P(c) = \frac{1}{6}$. Рассмотрим $w = bac$.

Кодирование:

$$[0, 1) \xrightarrow{b} \left[\frac{1}{3}, \frac{5}{6}\right] \xrightarrow{a} \left[\frac{1}{3}, \frac{1}{2}\right] \xrightarrow{c} \left[\frac{17}{36}, \frac{1}{2}\right].$$

Берём, например,

$$x = \frac{31}{64} = (0.011111)_2,$$

то есть код 011111. Также запоминаем, что $n = 3$.

Декодирование кода 011111: считаем, что закодированное число $x = \frac{31}{64}$.

- Старт $[0, 1)$. Делим на $[0, \frac{1}{3})$ (для a), $[\frac{1}{3}, \frac{5}{6})$ (для b), $[\frac{5}{6}, 1)$ (для c). Число $x = \frac{31}{64}$ попадает в $[\frac{1}{3}, \frac{5}{6})$, значит первый символ — b .
- Переходим к $[\frac{1}{3}, \frac{5}{6})$ и делим его так же пропорционально. x попадает в первый кусок, значит второй символ — a . Получаем интервал $[\frac{1}{3}, \frac{1}{2})$.
- Делим $[\frac{1}{3}, \frac{1}{2})$; x попадает в кусок для c , значит третий символ — c .

Так как $n = 3$, останавливаемся: строка bac восстановлена.

18.5 Целочисленная реализация арифметического кодирования

Работать с рациональными числами напрямую медленно, поэтому используют **целочисленную** реализацию.

Выбираем $N = 2^B$ (шкала). Число вида $\frac{i}{N}$ храним как целое i . Начальный интервал: $[\ell, m) = [0, N)$. Вероятности заменяем целыми **весами** $P(a)$, сумма которых равна N .

Переход по символу. Пусть считали символ a_j

$$\ell' = \ell + \left\lfloor \frac{m - \ell}{N} \sum_{t=1}^{j-1} P(a_t) \right\rfloor, \quad m' = \ell + \left\lfloor \frac{m - \ell}{N} \sum_{t=1}^j P(a_t) \right\rfloor.$$

Чтобы можно было постепенно выводить старшие биты, используются три случая «исправления» интервала (E1/E2/E3):

- Если $m < \frac{N}{2}$ (интервал целиком в нижней половине $[0, \frac{N}{2})$), то все числа из интервала

имеют следующий бит результата 0.

Выводим **шаблон** 01^k , где k — число “запомненных” неопределённых разрядов из Е3. После этого делаем $(\ell, m) = (2\ell, 2m)$ и сбрасываем k в 0.

- Если $\ell \geq \frac{N}{2}$ (интервал целиком в верхней половине), то следующий бит обязан быть 1. Выводим **шаблон** 10^k (бит 1 и затем k нулей), затем делаем $(\ell, m) = (2(\ell - \frac{N}{2}), 2(m - \frac{N}{2}))$ и сбрасываем k в 0.
- Если $\frac{N}{4} \leq \ell < \frac{N}{2}$ и $\frac{N}{2} \leq m < \frac{3N}{4}$, (интервал «около середины»): следующий бит пока *неясен*. Чтобы не терять точность, делаем $(\ell, m) = (2(\ell - \frac{N}{4}), 2(m - \frac{N}{4}))$ и увеличиваем счётчик $k := k + 1$.

Пример

Округлим вероятности до шага $\frac{1}{16}$:

$$P(a) = \frac{5}{16}, \quad P(b) = \frac{8}{16}, \quad P(c) = \frac{3}{16}.$$

Берём $N = 16$, веса $P(a) = 5$, $P(b) = 8$, $P(c) = 3$. Кодируем $w = bac$.

- Старт: $[0, 16)$.
- После b : $[5, 13)$ (исправлений нет).
- После a : $[5, 7)$. Так как $7 < 8$, это Е1: выводим 0 (здесь $k = 0$), масштабируем $\rightarrow [10, 14)$. Теперь $\ell = 10 \geq 8$, это Е2: выводим 1, масштабируем $\rightarrow [4, 12)$.
- После c : из $[4, 12)$ получаем $[10, 12)$. Здесь $\ell = 10 \geq 8$, Е2: выводим 1, масштабируем $\rightarrow [4, 8)$.
- Интервал $[4, 8)$ попадает в Е3: $[4, 8) \rightarrow [0, 8)$ ($k = 1$), затем $[0, 8) \rightarrow [0, 16)$ ($k = 2$).

Дальше символы кончились, выводим финальные биты. Так как $\ell = 0 < \frac{N}{4} = 4$, выводим $01^{k+1} = 011$. Итого код: 011 011, то есть 011011, соответствующий числу $(0.011011)_2 = \frac{27}{64}$, близкому к числу из рациональной реализации.

Algorithm 23 Арифметическое кодирование (целочисленная реализация)

Require: строка $w = a_{i_1} \dots a_{i_n}$ над линейно упорядоченным $\Sigma = \{a_1, \dots, a_k\}$; целые веса $P(a_1), \dots, P(a_k)$, $\sum P(a_j) = N = 2^B$.

1: $\ell = 0$
2: $m = N$
3: $t = 0$
4: **for** $i = 1$ to n **do**
5: $a_j = i$ -й символ строки w /* $a_j \in \Sigma$ */
6: $L = \sum_{t=1}^{j-1} P(a_t)$, $R = \sum_{t=1}^j P(a_t)$
7: $(\ell, m) = \left(\ell + \frac{m - \ell}{N} L, \ell + \frac{m - \ell}{N} R \right)$ /* сузили интервал под символ */
8: **while** выполнен E1 или E2 или E3 **do**
9: **if** $\frac{N}{4} \leq \ell$ **and** $m < \frac{3N}{4}$ **then**
10: $\ell = 2(\ell - \frac{N}{4})$ /* E3: сдвиг из “середины” */
11: $m = 2(m - \frac{N}{4})$
12: $t = t + 1$ /* запомнили ещё один “неясный” разряд */
13: **else if** $m < \frac{N}{2}$ **then**
14: вывести 0 и затем t единиц /* E1: выводим 0 1^t */
15: $t = 0$
16: $\ell = 2\ell$, $m = 2m$ /* масштабируем обратно к $[0, N]$ */
17: **else if** $\ell \geq \frac{N}{2}$ **then**
18: вывести 1 и затем t нулей /* E2: выводим 1 0^t */
19: $t = 0$
20: $\ell = 2(\ell - \frac{N}{2})$, $m = 2(m - \frac{N}{2})$
21: /* Добиваемся, чтобы выданные биты точно попали внутрь текущего интервала */
22: **if** $\ell < \frac{N}{4}$ **then**
23: вывести 0 и затем $t + 1$ единиц /* 0 1^{t+1} */
24: **else**
25: вывести 1 и затем $t + 1$ нулей /* 1 0^{t+1} */

18.6 Декодирование

При декодировании поддерживаются те же ℓ, m и число x , всегда лежащее в интервале $[\ell, m]$. Интуиция: x — это **текущее “окно” из B битов** закодированного числа (мы его постоянно сдвигаем).

- Сначала читаем $B = \log_2 N$ первых битов кода и записываем в $x \in [0, N]$.
- На каждом шаге делим $[\ell, m]$ на части пропорционально весам $P(a_j)$ и по тому, куда попадает x , определяем очередной символ.
- Затем сужаем интервал, и делаем те же исправления E1/E2/E3. При каждом исправлении масштабируем не только ℓ, m , но и x , и **дочитываем следующий бит** из входа в младший разряд x .

Algorithm 24 Арифметическое кодирование: обратное преобразование

Require: битовая последовательность, длина исходного слова n ; алфавит $\Sigma = \{a_1, \dots, a_k\}$; целые веса $P(a_1), \dots, P(a_k)$, сумма которых равна $N = 2^B$.

- 1: $\ell = 0$
- 2: $m = N$
- 3: $x = (\text{первые } \log_2 N \text{ битов входа})$
- 4: **for** $i = 1$ to n **do**
- 5: найти такое j , что

$$\ell + \frac{m - \ell}{N} \sum_{t=1}^{j-1} P(a_t) \leq x < \ell + \frac{m - \ell}{N} \sum_{t=1}^j P(a_t);$$

- 6: вывести символ a_j
 - 7: $(\ell, m) = \left(\ell + \frac{m - \ell}{N} \sum_{t=1}^{j-1} P(a_t), \ell + \frac{m - \ell}{N} \sum_{t=1}^j P(a_t) \right)$
 - 8: **while** верен один из трёх случаев **do**
 - 9: **if** $\ell \geq \frac{N}{4}$ **and** $m < \frac{3N}{4}$ **then**
 - 10: $\ell = 2(\ell - \frac{N}{4})$
 - 11: $m = 2(m - \frac{N}{4})$
 - 12: $x = 2(x - \frac{N}{4})$
 - 13: **else if** $m < \frac{N}{2}$ **then**
 - 14: $\ell = 2\ell$
 - 15: $m = 2m$
 - 16: $x = 2x$
 - 17: **else if** $\ell \geq \frac{N}{2}$ **then**
 - 18: $\ell = 2(\ell - \frac{N}{2})$
 - 19: $m = 2(m - \frac{N}{2})$
 - 20: $x = 2(x - \frac{N}{2})$
 - 21: **else**
 - 22: **break**
 - 23: прочитать следующий бит входа; если он равен 1, то $x = x + 1$
-

Пример

Пусть $N = 16$, $P(a) = 5$, $P(b) = 8$, $P(c) = 3$, и декодируется код 011011.

- Читаем первые $B = 4$ бита: $x = (0110)_2 = 6$, интервал $[0, 16]$.
- Делим по весам: $[0, 5)$ (a), $[5, 13)$ (b), $[13, 16)$ (c). $x = 6$ попадает в (b), значит первый символ b , новый интервал $[5, 13)$.
- Делим $[5, 13)$: получаем $[5, 7)$ (a), $[7, 11)$ (b), $[11, 13)$ (c). $x = 6$ попадает в (a), значит второй символ a , новый интервал $[5, 7)$.
- Дальше начинают срабатывать те же E1/E2, при этом x масштабируется вместе с интервалом и каждый раз дочитывается следующий бит из входа — в итоге третий символ получается c .

После трёх шагов исходная строка bas восстановлена; оставшиеся биты кода могут не понадобиться.

Замечание

Набор вероятностей/весов не обязан быть одним и тем же на каждом шаге. Можно использовать любую модель, которая по уже прочитанному префикску оценивает веса символов;

важно лишь, чтобы при кодировании и декодировании они вычислялись одинаково на соответствующих шагах.

19 Суффиксное дерево и его применение. Алгоритм Укконена построения суффиксного дерева.

19.1 Суффиксные деревья

Суффиксное дерево (Вайнер, 1973) — это структура данных, которая строится по одной строке w и позволяет очень быстро искать в ней подстроки.

Типичный сценарий: фиксируется длинная строка w (“текст”), а затем многоократно задаются запросы вида “входит ли строка x в w и где именно?”. Если заранее построить суффиксное дерево для w , то каждый такой запрос можно обрабатывать за время $O(|x|)$, не зависящее от длины текста $|w|$.

19.2 Упрощённое суффиксное дерево

В упрощённом виде суффиксное дерево — это префиксное дерево (бор) для множества суффиксов данной строки.

Пусть $w = a_1 \dots a_n$ — строка. Тогда у неё $n + 1$ суффиксов:

$$\text{suffixes}(w) = \{a_i a_{i+1} \dots a_n \mid i = 1, \dots, n\} \cup \{\varepsilon\}.$$

Если построить префиксное дерево для множества $\text{suffixes}(w)$, то:

- каждая вершина соответствует какому-то префиксу одного из суффиксов, то есть подстроке w ;
- всякая подстрока w — это префикс *какого-то* суффикса, значит её можно найти, спустившись от корня по символам подстроки.

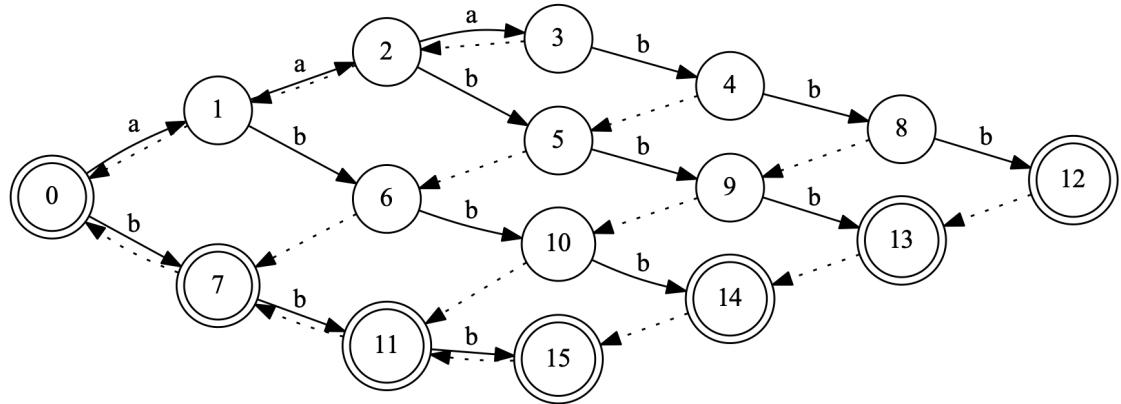
Кроме того, в таком дереве удобно вводятся *суффиксные ссылки*. Пусть $w = uav$, где $u, v \in \Sigma^*$, $a \in \Sigma$. Суффиксам av и v соответствуют вершины дерева. Из вершины av проводится суффиксная ссылка в вершину v (“убрали первый символ”).

Если $w = a_1 \dots a_n$, то, следуя по суффиксным ссылкам из вершины, соответствующей w , можно последовательно обойти все n вершин, соответствующих суффиксам w .

Но у такого несжатого дерева есть серьёзный недостаток.

Пример

Префиксное дерево для множества суффиксов строки $w = a^n b^n$ содержит $(n + 1)^2$ вершин. Схема дерева для $n = 3$ изображена на рисунке ниже.)



Упрощённое суффиксное дерево для строки $w = aaabbb$

Размер дерева в худшем случае $\Theta(n^2)$, а значит и построение, и хранение такой структуры обходятся слишком дорого.

19.3 Суффиксное дерево и его построение

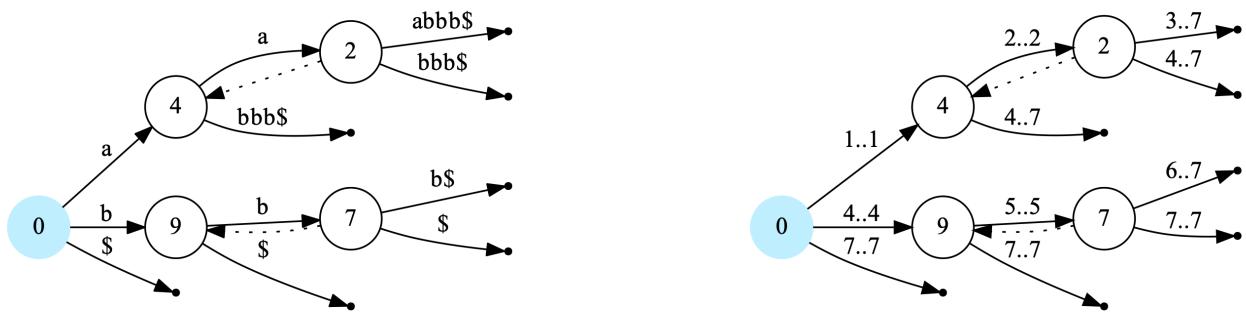
Настоящее (компактное) суффиксное дерево.

Определение

Пусть дана строка w .

Суффиксное дерево для w — это **сжатое (компактное)** префиксное дерево для множества всех суффиксов строки w .

- Вместо того, чтобы хранить по одному символу на ребре, каждое ребро помечается **подстрокой** $w[i..j]$.
- На практике ребро хранится как пара позиций (i, j) в исходной строке w , а не как отдельная строка.
- Каждая внутренняя вершина имеет не менее двух исходящих рёбер.
- Суффиксные ссылки хранятся только между внутренними вершинами.



суффиксное дерево для $w\$$; компактное представление рёбер суффиксного дерева.

Число вершин. Число листьев в суффиксном дереве не превосходит $n + 1$ (по одному листу на каждый суффикс, включая пустой). Каждая внутренняя вершина обязана ветвиться

(степень ≥ 2), поэтому таких вершин не больше, чем листьев. Отсюда общее число вершин — $O(n)$.

Почему нужна особая адресация подстрок. В несжатом префиксном дереве каждая под строка соответствует вершине. В компактном дереве под строки “внутри ребра” отдельной вершины не имеют.

Удобно представлять произвольную под строку **тройкой**:

(вершина v , исходящее из неё ребро e , длина $j \geq 0$ вдоль e).

Интуитивно: стоим в вершине v , выбираем исходящее ребро e и идём по нему j символов.

Все операции, которые в несжатом дереве делались бы с вершинами, в компактном дереве выполняются с такими тройками.

Идея алгоритма Укконена

Есть несколько линейных алгоритмов построения суффиксного дерева. Алгоритм Укконена (1995) строит дерево **онлайн**:

- строка $w = a_1 a_2 \dots a_n$ читается слева направо;
- после чтения префикса $a_1 \dots a_i$ в памяти поддерживается почти готовое суффиксное дерево для этого префикса;
- на шаге i все уже присутствующие суффиксы продлеваются символом a_i (неявно), и при необходимости добавляются новые вершины и листья.

Чтобы добиться времени $O(n)$, алгоритм делает два ключевых трюка:

1. Использует рёбра с “бесконечным” концом для всех листьев.
2. Хранит только один “самый длинный” недостроенный суффикс как *текущее положение* (v, j) , а остальные короткие суффиксы достраивает автоматически.

Незавершённые суффиксы и текущее положение. На шаге i алгоритм хранит *неполное* суффиксное дерево для префикса $a_1 \dots a_i$: из него **исключены** несколько последних суффиксов — а именно *самый длинный суффикс*, который *уже встречался в дереве* (совпадает с некоторой предыдущей под строкой), и все более короткие суффиксы. Этот самый длинный суффикс алгоритм помнит в виде *положения в дереве*.

Текущее положение задаётся парой (v, j) :

- v — вершина дерева;
- $j \geq 0$ — длина, отсчитанная вдоль исходящей из v дуги.

Важно: запоминать, по какой именно дуге идёт отсчёт, не нужно. Это всегда дуга, которая начинается символом a_{i-j+1} , то есть j -м с конца символом уже прочитанной части строки. После обработки символа a_i текущее положение указывает ровно на этот символ.

Как продлеваются рёбра к листьям за $O(1)$. Наивно, чтобы продлить все суффиксы на один символ, пришлось бы для каждого листа увеличивать на 1 правую границу под строки на соответствующей дуге. Листьев $O(n)$, и если так делать на каждом шаге, получится $O(n^2)$.

Укконен делает иначе:

- рёбра, ведущие в листья, помечаются парами вида (i, ∞) ;
- символ ∞ означает, что подстрока идёт “до конца” текущего прочитанного префикса;
- при чтении каждого нового символа все такие рёбра автоматически удлиняются на один, **без явного прохода по листьям**.

Пример

Пример. Построение суффиксного дерева по алгоритму Укконена для строки

$$w = ababbabbba.$$

Во время работы алгоритма будем явно фиксировать:

- что уже прочитано;
- текущее положение (v, j) : вершина v и длина j вдоль исходящей из неё дуги;
- какие суффиксы текущего префикса ещё не достроены.

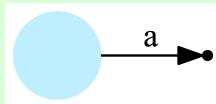
Шаг 1. Читаем первую букву а.

В дереве появляется единственная дуга из корня в лист, помеченная $(1, \infty)$: она соответствует суффиксу **a**.

Прочитано: а

Положение: корень, длина 0.

Недостроенных суффиксов: нет.



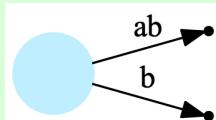
Шаг 2. Читаем вторую букву б.

Все дуги, ведущие в листья, автоматически продлеваются на один символ. Единственная дуга **a** становится дугой **ab** (метка $(1, \infty)$ просто растягивается до позиции 2). Кроме того, появляется новая дуга из корня, соответствующая суффиксу **b**, тоже с меткой $(2, \infty)$.

Прочитано: ab.

Положение: корень, длина 0.

Недостроенных суффиксов: нет.



Шаг 3. Читаем третью букву а.

Обе дуги, идущие в листья, автоматически продлеваются:

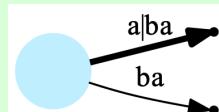
$$ab \rightarrow aba, \quad b \rightarrow ba.$$

Теперь суффикс **a** префикса **aba** уже встречался раньше (это первый символ строки). Алгоритм пытается прочитать **a** из корня — это удаётся, и он **запоминает** недостроенный суффикс **a** как текущее положение на дуге **a**, где пометка $a|ba$ означает длину 1, отсчитанную на дуге **aba**.

Прочитано: aba.

Положение: корень, дуга **a**, длина 1.

Недостроенный суффикс: **a**.



Шаг 4. Читаем четвёртую букву б.

Дуги к листьям снова продлеваются:

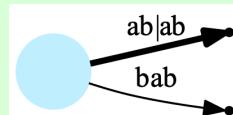
$$aba \rightarrow abab, \quad ba \rightarrow bab.$$

Алгоритм пытается продлить недостроенный суффикс **a** до **ab**: из корня по **a** есть дуга, следующий символ на ней — **b**, он совпадает с читаемой буквой. Значит, **ab** уже содержится в дереве, и алгоритм лишь смещает положение на один символ вперёд.

Прочитано: abab.

Положение: корень, дуга **a**, длина 2.

Недостроенный суффикс: **ab**.



Шаг 5. Читаем пятую букву **b**. Первая «ошибка» и разрезы дуг.

Дуги к листьям продлеваются:

$$abab \rightarrow ababb, \quad bab \rightarrow babb.$$

Теперь рассматриваем недостроенный суффикс **ab** и пытаемся дописать к нему новую букву **b**, то есть построить суффикс **abb**.

5.1. Несовпадение внутри дуги ababb. Разрез и новая внутренняя вершина. Активная точка находится на дуге, которая из корня ведёт по строке **ababb**. Мы стоим на расстоянии 2 от корня (наподстроке **ab**) и смотрим на следующий символ на дуге:

ababb.

Он равен **a**, а нам нужно прочитать **b**. Возникает **несовпадение внутри дуги**.

Алгоритм устраняет его так:

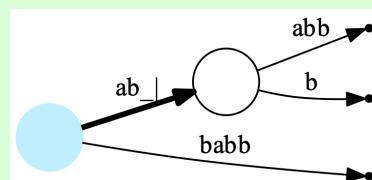
- дуга **ababb** делится пополам в точке после **ab**;
- получается новая внутренняя вершина, соответствующая подстроке **ab**;
- из корня теперь идёт дуга, помеченная **ab**, в эту вершину;
- из новой вершины выходят две дуги:
 - старая продолженная дуга **abb**, ведущая к листу суффикса **ababb**;
 - новая короткая дуга **b**, ведущая к листу суффикса **abb** (путь корень → **ab** → **b** задаёт строку **abb**).

Это первая внутренняя вершина, поэтому суффиксную ссылку пока никуда не ставим: переменная «последняя добавленная вершина» просто запоминает эту вершину **ab**.

Прочитано: ababb.

Положение: корень, дуга **a**, длина 3 (идёт обработка).

Недостроенные суффиксы: **abb, bb, b.**

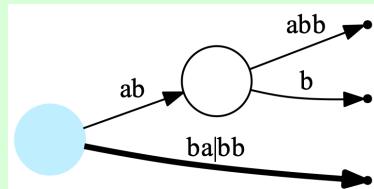


Суффикс **abb** мы только что внесли в дерево (он заканчивается в новом листе). Следующий по длине недостроенный суффикс — **bb**. Алгоритму нужно перейти к его обработке.

Как перейти к следующему суффиксу **bb.**

Текущее положение было на расстоянии 3 от корня по пути **ab a**. Чтобы перейти от **abb** к **bb**, нужно отбросить первый символ **a**, то есть уменьшить длину на 1. При этом:

- длина от корня становится равной 2;
- путь длины 2 от корня теперь идёт по дуге **b**, а не по дуге **a**;
- активное положение автоматически «перескакивает» на дугу **b** на расстояние 2 (подстрока **bb**).



Получаем активный суффикс **bb** и снова пытаемся дописать к нему букву **b**.

5.2. Разрез дуги **babb** и суффиксная ссылка **ab** → **b**.

Теперь смотрим на дугу, идущую из корня по строке **babb**. Мы отмечаем на ней два символа **bb** и смотрим на следующий:

babb.

Последний отмеченный символ (второй) — **a**, а читаем **b**. Снова несовпадение внутри дуги, поэтому:

- дуга **babb** делится пополам в точке после первого символа **b**;
- появляется новая внутренняя вершина, соответствующая подстроке **b**;
- из корня теперь есть короткая дуга **b** в эту вершину;
- от новой вершины уходят дуги **abb** и **b**, соответствующие суффиксам **babb** и **bb**.

А теперь важный момент: только что мы создали *вторую* внутреннюю вершину.

- Первая внутренняя вершина соответствовала подстроке **ab**.
- Вторая — подстроке **b**.

Междуд ними алгоритм сразу ставит суффиксную ссылку:

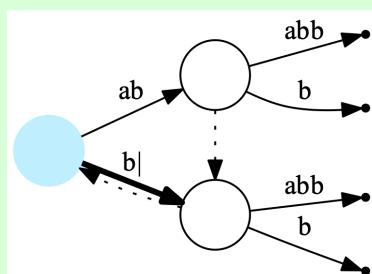
ab → **b**.

Это логично: если из строки **ab** удалить первый символ, остаётся **b**.

Прочитано: ababb.

Положение: корень, дуга **b**, длина 2 (идёт обработка).

Недостроенные суффиксы: **bb**, **b**.



Теперь суффикс **bb** тоже достроен, остался последний недостроенный суффикс — **b**.

5.3. Обработка последнего суффикса **b**.

Переход к суффиксу **b** устроен так же: мы снова уменьшаем длину пути от корня на 1. Было 2, становится 1, то есть активное положение — корень, дуга **b**, длина 1 (подстрока **b**).

Теперь пытаемся продлить этот суффикс буквой **b**. Последний отмеченный символ на дуге **b** и читаемый символ совпадают (оба **b**), поэтому:

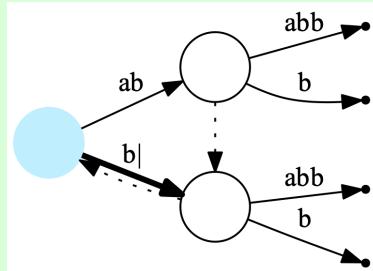
- никаких новых дуг и вершин создавать не нужно;
- алгоритм лишь ставит суффиксную ссылку из *последней созданной* внутренней вершины (той, что соответствует **b**) в *текущую* вершину активного положения.

После этого все суффиксы префикса **ababb** уже представлены в дереве, и обработка пятого символа завершается.

Прочитано: ababb.

Положение: корень, дуга **b**, длина 1.

Недостроенных суффиксов: нет — суффиксы **ababb**, **babb**, **abb**, **bb** и **b** все содержатся в дереве.



Шаг 6. Читаем шестую букву **a**.

Дуги к листьям продлеваются:

$$ababb \rightarrow ababba, \quad babb \rightarrow babba, \quad abb \rightarrow abba, \dots$$

Пытаемся продлить активный суффикс.

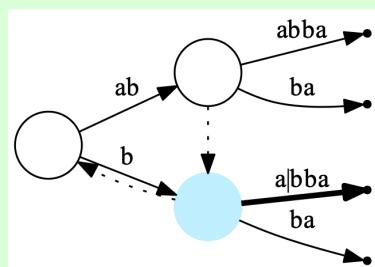
Активный суффикс сейчас — **b**. Мы читаем новую букву **a** и пытаемся дописать её к этому суффиксу, т.е. построить **ba**.

- Сначала просто увеличиваем длину j на 1, получая $j = 2$ на дуге **b**.
- Но сама дуга **b** имеет длину 1, поэтому $j > |x|$, и срабатывает *нормализация положения*: мы полностью «проходим» дугу **b**, переходим в вершину, помеченную **b**, и уменьшаем j на длину дуги. В итоге j становится равным 1.
- Теперь отсчёт идёт уже по другой дуге — исходящей из вершины **b** и начинавшейся буквой **a**. На этой дуге первая буква как раз **a**, то есть она совпадает с входным символом.

Так как символ совпал, больше ничего делать не нужно, просто запоминается новое активное положение.

Прочитано: ababba.

Положение: вершина **b**, дуга **a**, длина 1.



Шаг 7. Читаем седьмую букву **b**.

Все листовые дуги снова автоматически удлиняются на один символ.

$$ababba \rightarrow ababbab, \quad babba \rightarrow babbab, \dots$$

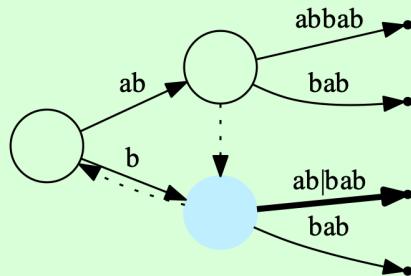
Продлеваем активный суффикс.

Активный суффикс, который мы помним, — **a** (после вершины **b**). Нужно добавить к нему новую букву **b**, т.е. получить **ab**.

- Увеличиваем j до 2 на той же дуге **a**....

- Смотрим второй символ на этой дуге: он **b**.
- Он совпадает с входным символом **b**, поэтому разрезать дугу и создавать вершины не нужно.

Прочитано: ababbab.
Положение: вершина **b**,
дуга **a**, длина 2.



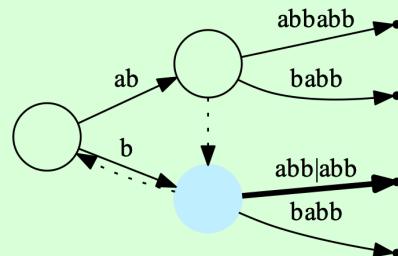
Шаг 8. Читаем восьмую букву **b**.

Все листы ещё раз автоматически удлиняются на букву **b**:

$$ababbab \rightarrow ababbabb, \quad babbab \rightarrow babbabb, \dots$$

Восьмой символ **b** читается точно так же.

Прочитано: ababbabb.
Положение: вершина **b**,
дуга **a**, длина 3.



К этому моменту *внутри одной дуги* накоплены несколько недостроенных суффиксов (самый длинный и все его суффиксы). Они ещё не вынесены в отдельные листья — это произойдёт позже, когда на следующем шаге возникнет первое несовпадение.

Шаг 9. Читаем девятую букву **b**.

Как и раньше, все дуги, ведущие в листья, сами удлиняются ещё на один символ:

$$ababbabb \rightarrow ababbabbb, \quad babbabb \rightarrow babbabbb, \dots$$

9.1. Пытаемся продлить активный суффикс. Первый разрез дуги и новая внутренняя вершина abb.

Активный суффикс сейчас — **abb**. Нужно дописать к нему новую букву **b**, то есть получить **abbb**.

- Увеличиваем длину на дуге: пытаемся перейти к четвёртому символу.
- Четвёртый символ на этой дуге оказывается **a**, а читаемая буква — **b**.

Получаем **несовпадение внутри дуги**.

Чтобы исправить несовпадение, алгоритм делит текущую дугу пополам:

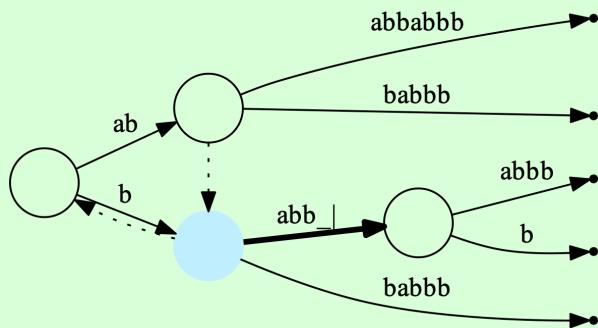
- в точке после трёх символов **abb** создаётся новая внутренняя вершина, соответствующая подстроке **abb**;
- дуга из вершины **b** теперь идёт до этой вершины и помечена **abb**;
- из новой вершины выходят две дуги:

- старая хвостовая часть (начинается с **a** — продолжение старого пути);
- новая дуга, начинающаяся с **b**, ведущая к листу для суффикса **abbb**.

Это первая внутренняя вершина, созданная на этом шаге, поэтому переменная «последняя добавленная вершина» запоминает именно её.

Прочитано: ababbabbb.

Положение (пока): вершина **b**,
дуга **a**,
длина 4 (идёт обработка).



9.2. Переход к следующему суффиксу **abbb**.

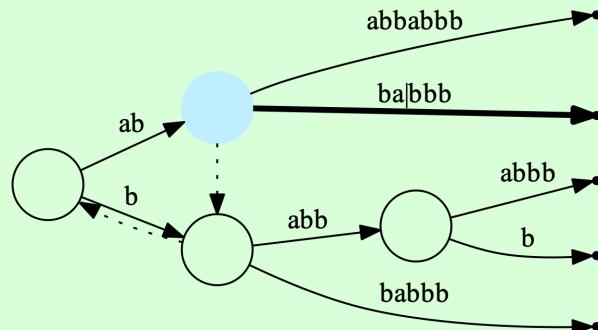
Следующий суффикс, который нужно достроить, — **abbb** (он начинается на один символ правее предыдущего).

Переход к нему выглядит так:

- идём по суффиксной ссылке из текущей вершины **b** в корень, длину j при этом не меняем (она остаётся равной 3);
- от корня с длиной $j = 3$ нормализуем положение: спускаемся по дугам, читая **abb**; в итоге оказываемся в вершине, соответствующей подстроке **ab**;

Прочитано: ababbabbb.

Положение: вершина **ab**, дуга
b, длина 2 (идёт обработка).



9.3. Несовпадение на дуге из вершины **ab** и разрез.

Сейчас рассматриваем следующий суффикс **abbb**.

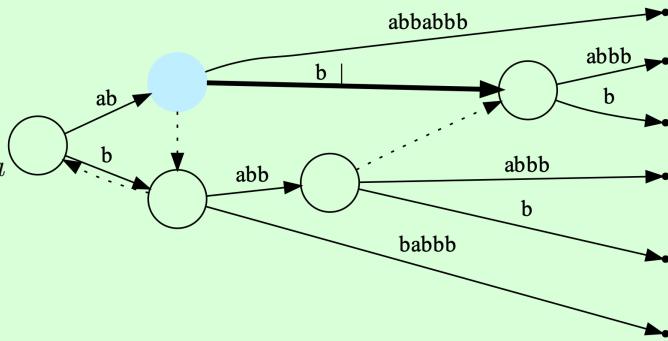
Пытаемся отметить третью букву **b**. Оказывается, на дуге в этой позиции стоит символ **a**, а читаемая буква — **b**. Возникает новое несовпадение внутри дуги.

Алгоритм делает то же самое, что и раньше:

- дуга из вершины **ab** делится на две части;
- в этой точке создаётся промежуточная вершина, соответствующая подстроке **abb**;
- из предыдущей промежуточной вершины (для подстроки **babbb**) сразу ставится суффиксная ссылка в новую вершину **abb**.

После разреза суффикс **abbb** полностью внесён в дерево.

Прочитано: ababbabbb,
Положение: вершина ab, дуга
b, длина 2 (идёт обработка).



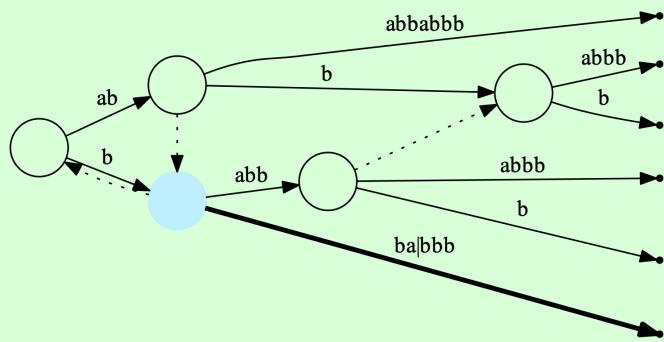
9.4. Суффикс bbb. Завершение шага.

Суффикс abbb уже внесён в дерево. Следующий по длине суффикс — bbb.

Внесение суффикса bbb.

Алгоритм идёт по суффиксной ссылке из вершины ab в вершину b, **сохраняя** число отсчитанных символов $j = 2$.

Прочитано: ababbabbb,
Положение: вершина b, дуга
b, длина 2.



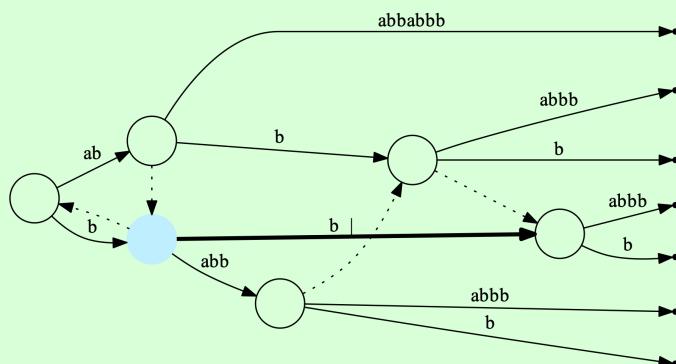
1) Разрез дуги и добавление суффикса bbb.

Мы пытаемся отметить на этой дуге следующий (третий) символ для суффикса bbb. На дуге в этой позиции стоит буква a, а читаемая буква — b. То есть **символы не совпадают**.

Алгоритм делает то же самое, что и раньше при несоответствии внутри дуги:

- дуга из вершины b делится на две части в точке после bb;
- в этой точке создаётся новая внутренняя вершина, соответствующая подстроке bb;
- от неё отходят две дуги: старая хвостовая (начинается символом a) и новая дуга по b в лист — это как раз суффикс bbb;
- из предыдущей промежуточной вершины (со строкой abb) ставится суффиксная ссылка в новосозданную вершину bb.

Прочитано: ababbabbb,
Положение: вершина b, дуга
b, длина 2.



После этого суффикс **bbb** внесён в дерево.

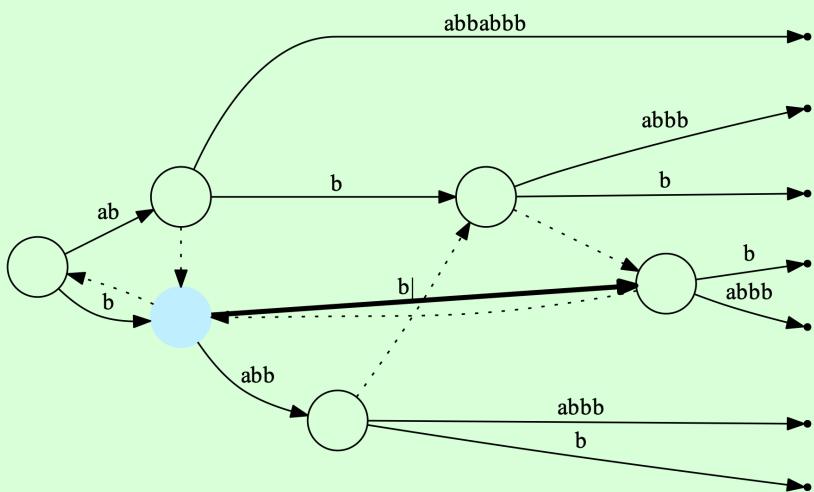
9.5. Переход к суффиксу **bb**.

Чтобы перейти к нему, алгоритм:

- идёт по соответствующей суффиксной ссылке с сохранением числа отмеченных позиций $j = 2$ и попадает в корень, на дугу **b**;
- на дуге **b** всего один символ, поэтому дуга «проходится» целиком, и мы оказываемся в вершине **b**, при этом j уменьшается до 1;
- теперь отмечен один символ **b** на дуге **b** — это суффикс **bb**; следующий символ на дуге совпадает с читаемой буквой, разрезать больше нечего;
- остаётся только проставить суффиксную ссылку из последней созданной вершины **bb** в текущую вершину **b**.

Прочитано: **ababbabbb**.

Положение: вершина **b**, дуга **b**, длина 1.



После этого все суффиксы префикса **ababbabbb** обработаны на шаге 9.

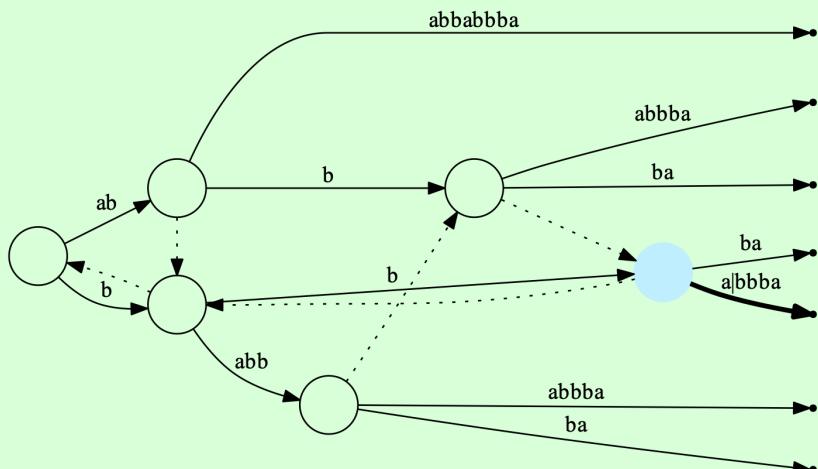
Шаг 10. Читаем последнюю букву **a**.

Дуги, ведущие в листья, автоматически продлеваются на символ **a**.

Алгоритм пытается прочитать **a** из текущего положения (вершина **b**, дуга **b**, длина 1). На текущей дуге следующий символ действительно равен **a**, поэтому достаточно сдвинуться по ней ещё на одну позицию вперёд — в новую вершину, соответствующую подстроке **bb**. Разрезов дуг и новых суффиксных ссылок больше не требуется.

Прочитано: **ababbabba**.

Положение: вершина **bb**, дуга **a**, длина 1.



В дереве всё ещё остаются три недостроенных суффикса: bba , ba , a — они «спрятаны» в текущем положении и ещё не доведены до отдельных листьев.

Завершение: добавляем специальный символ \$.

Чтобы аккуратно учесть все недостроенные суффиксы, к строке приписывают особый символ $$$, не встречающийся в алфавите:

$$w' = ababbabbbba$.$$

После прочтения символа $$$ каждый суффикс строки w' оканчивается в своём отдельном листе, и получается полноценное суффиксное дерево для исходной строки w .

19.4 Алгоритм Укконена: псевдокод

Перед формальной записью зафиксируем обозначения.

- Положение алгоритма в дереве задаётся парой (v, j) , где
 - v — вершина суффиксного дерева;
 - $j \geq 0$ — сколько символов мы прошли по исходящей из v дуге.
Если $j = 0$, мы находимся точно в вершине v .
- Пусть из v выходит дуга в вершину, соответствующую строке vx , и эта дуга помечена подстрокой x (заданной как пара позиций в строке w). Тогда:
 - вершину-назначение обозначаем через vx ;
 - через x обозначаем строку на дуге, её длина $|x|$ используется для “нормализации” положения (v, j) .

Algorithm 25 Алгоритм Укконена для построения суффиксного дерева

```
1: Вход: строка  $w = a_1 \dots a_n$ 
2: строим дерево, состоящее из одного корня (подстрока  $\varepsilon$ )
3:  $j = 0$ 
4:  $v = \text{корень } \varepsilon$ 
5: for  $i = 1$  to  $n$  do
6:    $u = \text{NULL}$                                  $\triangleright$  на текущей букве ещё не добавляли новых вершин
7:    $j = j + 1$                                  $\triangleright$  пробуем продвинуть текущий суффикс на символ  $a_i$ 
8:   while  $j > 0$  do                       $\triangleright$  перебираем все недостроенные суффиксы
9:     while  $j > |x|$  do                   $\triangleright$  положение на дуге  $(v, vx)$  выходит за край
10:     $j = j - |x|$ 
11:     $v = vx$                                  $\triangleright$  полностью прошли дугу
12:    if из  $v$  нет перехода по символу  $a_{i-j+1}$  then
13:       $\triangleright$  здесь всегда  $j = 1$ , значит  $a_{i-j+1} = a_i$ 
14:      добавить переход из  $v$  по  $a_i$  в новый лист  $v_{a_i}$ 
15:      if  $u \neq \text{NULL}$  then
16:        добавить суффиксную ссылку из  $u$  в  $v$ 
17:      else if  $j$ -й символ на дуге совпадает с  $a_i$  then           $\triangleright$  можно просто продлить
         текущий суффикс
18:        if  $u \neq \text{NULL}$  then
19:          добавить суффиксную ссылку из  $u$  в  $v$ 
20:        continue for           $\triangleright$  символ  $a_i$  успешно дописан, переходим к следующему  $i$ 
21:      else                       $\triangleright$  несовпадение внутри дуги
22:        разрезать дугу, вставив промежуточную вершину  $v'$ 
23:        if  $u \neq \text{NULL}$  then
24:          добавить суффиксную ссылку из  $u$  в  $v'$ 
25:         $u = v'$                    $\triangleright$  запоминаем последнюю добавленную вершину
            $\triangleright$  переход к следующему (более короткому) суффиксу
26:        if  $v$  — корень then
27:           $j = j - 1$ 
28:        else
29:           $v = \text{суффиксная\_ссылка}(v)$ 
```

Как понимать цикл строк 8–26. Интуитивно: после прочтения символа a_i мы должны продлить на него все ещё не отражённые в дереве суффиксы префикса $a_1 \dots a_i$. Алгоритм делает это «сверху вниз»:

- **Нормализация положения (v, j) .**
 - Пока $j > |x|$, мы полностью проходим дугу (v, vx) : переходим в vx и уменьшаем j на $|x|$.
 - После этого либо $j = 0$ (мы в вершине), либо $1 \leq j \leq |x|$ и стоим внутри дуги.
- **Дальше возможны три случая:**
 - A. (строчка 12) Из вершины v нет перехода по символу a_{i-j+1} (на практике здесь всегда $j = 1$, то есть мы смотрим на a_i). Тогда:
 - добавляем новую дугу в лист — тем самым достраиваем один недостающий суффикс;
 - если на текущем шаге мы уже создавали промежуточную вершину u , добавляем из неё суффиксную ссылку в v .
 - B. (строчка 17) j -й символ на текущей дуге совпадает с a_i . Значит, дерево уже содер-

жит нужное продолжение:

- просто продлеваем текущий суффикс (никаких новых вершин не нужно);
- ставим суффиксную ссылку из u в v , если u уже был;
- обработка символа a_i на этом заканчивается, переходим к следующему i .

C. (строчка 21) Несовпадение внутри дуги:

- разбиваем дугу: создаём новую промежуточную вершину v' в позиции (v, j) ;
- от v' исходят две дуги: старая (хвост разрезанной дуги) и новая дуга с символом a_i в новый лист;
- если была предыдущая новая вершина u , проводим суффиксную ссылку из u в v' ;
- обновляем $u \leftarrow v'$.

• **Переход к следующему суффиксу.**

- Если v — корень, то следующий суффикс получается просто уменьшением j на 1 (мы «обрезаем» первый символ).
- Иначе переходим по суффиксной ссылке из v .

Внутренний цикл заканчивается, когда для текущего i все нужные суффиксы либо достроены (случаи А и С), либо оказалось, что их продолжение уже представлено в дереве (случай В).

Почему в строке 12 всегда $j = 1$. Предположим противное: пусть в момент проверки строки 12 $j > 1$, но из v нет перехода по символу a_{i-j+1} .

- На предыдущей итерации цикла мы либо уже прошли по дуге, либо нормализовали положение, так что текущее (v, j) указывает на подстроку $a_{i-j+1} \dots a_i$, которая уже встречалась раньше.
- Это значит, что в дереве есть дуга из v , начинающаяся с символа a_{i-j+1} : иначе эта подстрока не могла бы быть представлена.

Получаем противоречие. Следовательно, случай А может возникнуть только тогда, когда $j = 1$, то есть мы смотрим ровно на новый символ a_i и видим, что из v нет дуги, начинающейся с него.

Лемма

Алгоритм Укконена работает за время $O(n)$.

Доказательство. Рассмотрим, сколько раз могут выполняться вложенные циклы.

1. При каждом переходе к новому символу a_i мы один раз увеличиваем j в строке 6. Значит, суммарно за всё время работы алгоритма j увеличивается не более чем на n .
2. В цикле нормализации (строки 8–10) мы только уменьшаем j : каждый проход уменьшает j хотя бы на 1, и суммарное число таких уменьшений не может быть больше суммарного числа увеличений, то есть не больше n .
3. Внутренний цикл (строка 7) перебирает недостроенные суффиксы. Каждая его итерация, кроме последней для данного i :
 - либо добавляет новый лист (случай А),
 - либо разрезает дугу и добавляет внутреннюю вершину (случай С).

Всего в суффиксном дереве может появиться не более $O(n)$ листьев и внутренних вершин, значит, операций типов А и С тоже не больше $O(n)$. Между этими операциями выполняется только константное число действий.

Таким образом, каждая из трёх групп действий (увеличения j , нормализация положения, добавление вершин и переходы по суффиксным ссылкам) выполняется суммарно $O(n)$ раз. Значит, общее время работы алгоритма — $O(n)$.

□

19.5 Примеры применения суффиксного дерева

Подсчёт числа вхождений подстроки x в w .

- По символам строки x спускаемся от корня по подходящим дугам.
- Если в какой-то момент нужной дуги нет — строка x не входит в w ни разу.
- Если путь по x пройден полностью (мы оказались либо в вершине, либо внутри дуги), то:
 - все суффиксы, которые начинаются с x , дают листья в поддереве ниже;
 - число вхождений x равно количеству листьев в соответствующем поддереве.

Наибольшая общая подстрока строк w и z . Один из вариантов алгоритма:

- строим суффиксное дерево для строки w ;
- читаем строку z слева направо, поддерживая текущую позицию в дереве;
- при совпадении символа продолжаем спуск по дугам;
- при несовпадении:
 - поднимаемся по суффиксной ссылке;
 - продолжаем попытку прочитать тот же символ строки z с новой позиции;
- по ходу работы запоминаем максимальную достигнутую глубину (длину общей подстроки) и место, где она была достигнута.

Инвариант: переход по суффиксной ссылке заменяет текущую подстроку на ту же, но без первого символа. Поэтому после подъёма по суффиксной ссылке мы не теряем уже просмотренную часть строки z , а просто сдвигаем начало подстроки.

Каждый символ строки z обрабатывается за $O(1)$ амортизированного времени:

- суммарное число спусков и подъёмов по суффиксным ссылкам $O(|z|)$;
- поэтому просмотр строки z занимает $O(|z|)$, а весь алгоритм — $O(|w| + |z|)$ (построение дерева + просмотр).

20 Сжатие по повторяющимся подстрокам: LZ77 и LZ78

20.1 Почему повторяющиеся подстроки помогают

Метод Хаффмана и арифметическое кодирование используют разную частоту символов. Но в реальных текстах часто повторяются целые подстроки. Идея: заменять повтор на «ссылку» (LZ77) или постепенно строить «словарь» повторов (LZ78).

Ключевая мысль

Если текущий фрагмент уже встречался раньше, выгоднее записать:

(где было раньше) + (сколько символов) вместо самих символов.

20.2 Метод Лемпеля–Зива LZ77 (Ziv–Lempel, 1977)

Идея и формат выхода

LZ77 читает строку слева направо и заменяет её последовательностью фраз (троек) (d, ℓ, a) .

Определение

Тройка (d, ℓ, a) означает:

- взять ℓ символов, начиная на расстоянии d символов *назад* от текущей позиции;
- затем приписать один «буквальный» символ a .

Если подходящего повтора нет, обычно берут $(0, 0, a)$: «просто вывести символ a ». Допускается $\ell > d$: тогда копирование продолжается «по кругу», т.е. с периодическим продолжением.

Пример (LZ77)

Для строки $w = abaababa$ один из возможных кодов:

$$\underbrace{(0, 0, a)}_{a} \underbrace{(0, 0, b)}_{b} \underbrace{(2, 1, a)}_{aa} \underbrace{(3, 2, b)}_{bab} \underbrace{(0, 0, a)}_{a} .$$

То же можно записывать, отделяя «литералы» от ссылок:

$$ab (2, 1) (3, 3) (2, 2).$$

Проверка разбиения:

$$\underbrace{ab}_{ab} \underbrace{(2, 1)}_a \underbrace{(3, 3)}_{aba} \underbrace{(2, 2)}_{ba}$$

Жадный вариант

На каждом шаге есть свобода: какую из ранее встречавшихся подстрок взять. Жадная стратегия: *всегда брать самую длинную возможную подстроку*, даёт наилучшее сжатие. Следующая теорема приводится без доказательства.

Теорема

Жадный LZ77 оптимален.

«Скользящее окно»

Проблема «чистого» LZ77: расстояние d может становиться очень большим, и это дорого кодировать, а поиск по всей истории замедляется.

Скользящее окно (sliding window)

Обычно ищут повторы не во всей предыдущей строке, а только среди последних m символов:



Старые символы «забываются». Это чуть ухудшает сжатие, но упрощает и ускоряет реализацию.

Чтобы искать подстроки эффективно, можно воспользоваться суффиксным деревом — но для этого нужно научиться удалять из него все суффиксы длиннее m . Для этого в недостроенном суффиксном дереве, создаваемом алгоритмом Укконена, надо на каждом шаге стирать самый длинный суффикс за совокупное линейное время. Алгоритм будет помнить лист, соответствующий самому длинному суффиксу, и на каждом шаге будет удаляться дуга, ведущая в этот лист; после этого алгоритм, следуя по суффиксной ссылке, будет запоминать следующий по порядку суффикс. Если же самый длинный недостроенный суффикс находится как раз на этой дуге, то дуга будет делиться.

20.3 Метод Лемпеля–Зива LZ78 (Ziv–Lempel, 1978)

Идея: словарь строится на лету

В отличие от LZ77, метод LZ78 постепенно строит *словарь* подстрок. Словарь удобно хранить в виде префиксного дерева (trie): это позволяет быстро искать самое длинное совпадение с текущим префиксом входа.

В сжатом файле словарь не хранится: декодер строит *точно такой же* словарь, читая выходные пары.

Алгоритм LZ78 (кодирование)

Пусть словарь содержит строки T_0, T_1, \dots , где $T_0 = \varepsilon$.

На каждом шаге (пока вход не закончился):

1. найдём в словаре **самую длинную** строку $v = T_j$, совпадающую с текущим префиксом входа;
2. прочитаем следующий символ α , идущий сразу после v ;
3. выведем пару (j, α) ;
4. добавим в словарь новую строку $T_{\text{new}} = v\alpha$.

Пример (LZ78)

Пусть $w = ababaaabb$, а в начале $T_0 = \varepsilon$.

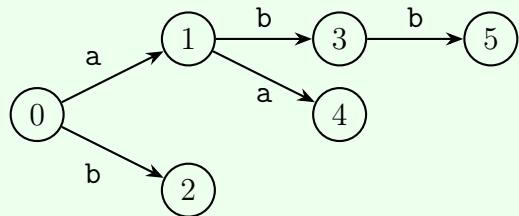
Кодирование (шаги):

| шаг | прочитали v | индекс j | символ α | добавили |
|-----|---------------|------------|-----------------|-------------|
| 1 | ε | 0 | a | $T_1 = a$ |
| 2 | ε | 0 | b | $T_2 = b$ |
| 3 | a | 1 | b | $T_3 = ab$ |
| 4 | a | 1 | a | $T_4 = aa$ |
| 5 | ab | 3 | b | $T_5 = abb$ |

Итоговый код:

$$(0, a) (0, b) (1, b) (1, a) (3, b) \text{ то есть } 0a \ 0b \ 1b \ 1a \ 3b.$$

Предфиксное дерево (один из вариантов):



Декодирование: на каждой паре (j, α) выводим $T_j \alpha$ и добавляем его в словарь. Для данного кода:

$$0a \mapsto a, \quad 0b \mapsto b, \quad 1b \mapsto ab, \quad 1a \mapsto aa, \quad 3b \mapsto abb.$$

Ограниченный словарь и варианты поведения

На практике размер словаря ограничивают (например, 2^{12} элементов). Когда словарь заполнен, возможны варианты:

- продолжать работу с текущим словарём (перестать добавлять новые строки);
- очистить словарь и начать заново;
- удалять записи по эвристике (например, «наименее полезные»).

Существует много вариантов LZ-методов, например LZW (Welch, 1984) как развитие LZ78. Часто LZ-схему комбинируют с последующим энтропийным кодированием (например, арифметическим).

21 Преобразование Берроуза–Вилера (BWT), реализация

21.1 Определение

Пусть дана строка $w = a_1 a_2 \dots a_n$ над линейно упорядоченным алфавитом Σ . Рассмотрим все её циклические сдвиги

$$\rho_i = a_i a_{i+1} \dots a_n a_1 a_2 \dots a_{i-1}, \quad i = 1, \dots, n.$$

Отсортируем ρ_1, \dots, ρ_n лексикографически и составим таблицу T размера $n \times n$, в каждой строке которой стоит один циклический сдвиг.

Обозначим через

$$L = b_1 b_2 \dots b_n$$

последний столбец таблицы T (последние символы отсортированных сдвигов), а через t — номер строки, где стоит исходная строка $w = \rho_1$. Тогда

$$\text{BWT}(w) = (L, t) = (b_1 \dots b_n, t).$$

21.2 Зачем это нужно при сжатии

На первый взгляд, это даже «увеличивает» данные: если исходная строка кодируется cn битами, где $c = \log_2 |\Sigma|$, то пара (L, t) требует примерно $cn + \log_2 n$ бит.

Но важна *структура* строки L . Если в w часто встречается подстрока вида au , то среди циклических сдвигов много строк, начинающихся с u и заканчивающихся на a ; после сортировки такие строки окажутся рядом, поэтому в L появляются длинные серии одинаковых символов. Дальше к L применяют простой алгоритм вроде RLE (run-length encoding), который сжимает повторяющиеся серии.

RLE: Максимальную подстроку вида $a^{\ell+2}$ можно закодировать тремя символами $aa\ell$.

Пример

Пример

Пусть $w = abaababa$.

Все циклические сдвиги: Отсортированные:

| | |
|----------|------------------------|
| abaababa | aabaabab |
| baababaa | aababaab |
| aababaab | abaabaab |
| ababaaba | abaababa ✓ ($t = 4$) |
| babaabaa | ababaaba |
| abaabaab | baabaaba |
| baabaaba | baababaa |
| aabaabab | babaabaa |

Последний столбец:

$$L = w' = \text{bbb}aaaa, \quad t = 4.$$

То есть $\text{BWT}(w) = (\text{bbb}aaaa, 4)$.

Тогда RLE даст на выходе $bb1aa3$.

21.3 Обратимость: «идея через восстановление таблицы»

По последнему столбцу $L = b_1 \dots b_n$ можно восстановить всю таблицу T (и значит, строку w).

- Отсортируем L — получим первый столбец $F = c_1 \dots c_n$ таблицы T , потому что это те же символы, только в отсортированном виде.
- Для каждого i пара $b_i c_i$ встречается в исходной строке как подстрока длины 2.
- Если отсортировать все $b_i c_i$, получится первые два столбца таблицы T .
- Повторяя (приписали L , отсортировали), можно нарастить 3 столбца, 4 столбца, …, пока не восстановим всю таблицу.

$$\begin{pmatrix} c_1 & ? & \dots & ? & b_1 \\ \vdots & \ddots & & & \vdots \\ c_n & ? & \dots & ? & b_n \end{pmatrix}$$

Это доказывает обратимость, но такой способ не годится как быстрый алгоритм.

21.4 Эффективное обратное преобразование (перестановка p)

Лемма

BWT можно вычислить за линейное время.

Доказательство. Строится суффиксное дерево для $ww\$$. Затем дерево обходится, и в лексикографическом порядке рассматриваются $n = |w|$ суффиксов строго большей длины, чем $w\$$. Первые n символов каждого суффикса — это один из циклических сдвигов строки w . При обходе для каждого суффикса выводится его n -й символ — это и получается последний столбец отсортированного массива циклических сдвигов. Время: $O(n)$. \square

Лемма

Обратное к BWT преобразование можно вычислить за линейное время.

Доказательство. Идея. Определим перестановку $p : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$: если i -я строка таблицы T имеет вид ub (то есть её последний символ $b = b_i$), то $p(i)$ — номер строки, содержащей следующий циклический сдвиг bu .

Тогда, зная t , можно сразу выписать конец исходной строки:

$$a_n = b_t, \quad a_{n-1} = b_{p(t)}, \quad a_{n-2} = b_{p^2(t)}, \dots, \quad a_{n-i} = b_{p^i(t)}.$$

Формула для $p(i)$.

$$p(i) = 1 + \left| b_1 b_2 \dots b_{i-1} \right|_{b_i} + \sum_{a < b_i} \left| b_1 b_2 \dots b_n \right|_a.$$

Почему так.

$$p(i) = 1 + \#\{\text{циклических сдвигов, лексикографически меньших, чем } bu\}.$$

Сдвиги, меньшие bu , бывают двух типов.

- все сдвиги, начинающиеся с символов $a < b$: их ровно $\sum_{a < b} |L|_a$;
- среди сдвигов, начинающихся с b , строка bu занимает то же место, что строка ub среди строк, оканчивающихся на b , поскольку $bu < bu \iff vb < ub$. Это место равно $\left| b_1 \dots b_{i-1} \right|_{b_i}$.

Добавляем $+1$, потому что строки нумеруются с единицы.

Как посчитать все $p(i)$ за $O(n + |\Sigma|)$. Создаём массив счётчиков C_a для всех $a \in \Sigma$ и делаем первый проход по L . В начале i -й итерации значение $\left| b_1 \dots b_{i-1} \right|_{b_i}$ лежит в C_{b_i} , его сохраняем в $p(i)$, затем увеличиваем C_{b_i} . После первого прохода знаем все $|L|_a$ и можем посчитать для каждого символа x сумму $\sum_{a < x} |L|_a$ (префиксные суммы по упорядоченному алфавиту). На втором проходе просто прибавляем эти суммы и 1 к уже сохранённым $p(i)$. \square

Пример

Дана преобразованная строка $w' = bbbaaaa$, а также номер строки 4 . Сперва находится число вхождений каждого символа в каждый префикс, а на их основе — перестановка p .

| | $b(1)$ | $b(2)$ | $b(3)$ | $a(4)$ | $a(5)$ | $a(6)$ | $a(7)$ | $a(8)$ |
|-----|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| a | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| b | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 |
| p | $1 + 0 + 5$ | $1 + 1 + 5$ | $1 + 2 + 5$ | $1 + 0 + 0$ | $1 + 1 + 0$ | $1 + 2 + 0$ | $1 + 3 + 0$ | $1 + 4 + 0$ |

Отсюда $p(4) = 1$, $p(1) = 6$, $p(6) = 3$, $p(3) = 8$, $p(8) = 5$, $p(5) = 2$, $p(2) = 7$ (и далее $p(7) = 4$).

По строке w' мы уже нашли перестановку p . Далее восстанавливаем исходную строку, двигаясь по индексам.

Начинаем с $i_{\text{idx}} = t = 4$.

Цепочка индексов:

$$4 \rightarrow 1 \rightarrow 6 \rightarrow 3 \rightarrow 8 \rightarrow 5 \rightarrow 2 \rightarrow 7 \rightarrow 4.$$

Выписываем символы $b_{i_{\text{idx}}}$ в этом порядке:

$$b_4 b_1 b_6 b_3 b_8 b_5 b_2 b_7 = a b a b a a b a = ababaaba.$$

Это w , записанная с конца к началу, поэтому разворачиваем:

$$w = abaababa.$$

22 Геометрические алгоритмы: ближайшая пара точек, выпуклая оболочка, наиболее удалённая пара точек

22.1 Вычислительная геометрия: общий контекст

Вычислительная геометрия изучает алгоритмы, которые получают на вход геометрические объекты (точки, отрезки, многоугольники) и должны:

- либо ответить на вопрос (пересекаются ли?),
- либо вычислить величину (расстояние/площадь),
- либо построить объект (выпуклую оболочку и т.п.).

Типичные приложения: навигация/ориентация, компьютерная графика и т.д.

22.2 Ближайшая пара точек (closest pair)

Постановка

Дано множество P из n точек на плоскости. Требуется найти пару (p, q) , с минимальным расстоянием между ними.

Наивное решение. Перебрать все $\binom{n}{2}$ пар $\Rightarrow O(n^2)$.

Цель. $O(n \log n)$ методом *разделяй и властвуй* (алгоритм Шеймоса).

Ключевая идея divide & conquer (алгоритм Шеймоса)

Сначала один раз подготовим два массива одной и той же выборки:

- X — точки, отсортированные по x (при равном x — по y);
- Y — точки, отсортированные по y (при равном y — по x).

Дальше работает рекурсия `БЛИЖАЙШАЯ_ПАРА(X, Y)`, где X и Y содержат одно и то же множество точек, просто в разных порядках.

1. **Делим по X .** Берём середину массива X и проводим вертикальную прямую $x = x_0$, разделяющую точки на две половины одинакового размера:

$$X_\ell = \text{левая половина } X, \quad X_r = \text{правая половина } X.$$

(Точки, лежащие ровно на прямой, можно отнести к любой половине.)

2. **Делим Y относительно той же прямой.** Чтобы не пересортировывать по y на каждом уровне, мы одним проходом по Y (сохраняя порядок) раскладываем точки в два массива:

$$Y_\ell := \{p \in Y : p \in X_\ell\}, \quad Y_r := \{p \in Y : p \in X_r\}.$$

Тогда Y_ℓ и Y_r уже отсортированы по y , и это делается за $O(n)$.

3. **Рекурсивно ищем минимум внутри половин.** Находим ближайшие пары *внутри* половин $\Rightarrow d_\ell, d_r$ и берём

$$d = \min(d_\ell, d_r).$$

4. **Проверяем пары по разные стороны разреза через полосу.** Если ближайшая пара лежит по разные стороны прямой $x = x_0$, то обе точки обязаны лежать на расстоянии $< d$ от этой прямой, то есть внутри полосы ширины $2d$.

Выделим из Y все такие точки, сохраняя порядок по y :

$$Y' := \{p \in Y : |p_x - x_0| < d\}.$$

(Это тоже один линейный проход по Y .) Теперь нужно искать улучшение только среди пар внутри Y' .

5. **«7 точек».** Пусть точки Y' идут по возрастанию y : $Y'[1], Y'[2], \dots$. Тогда для каждой точки $p = Y'[i]$ достаточно сравнить расстояние только с *семью следующими в этом массиве*:

$$Y'[i+1], Y'[i+2], \dots, Y'[i+7] \quad (\text{если они существуют}).$$

Геометрическое обоснование: все точки с расстоянием меньше, чем d , от (x, y) , лежат в прямоугольнике размером $2d \times d$. В квадрате $d \times d$ слева от прямой их не больше четырёх, и в квадрате справа от прямой их тоже не больше четырёх. Следовательно, больше восьми их там не разместить никак, и одна из них — сама точка (x, y) .

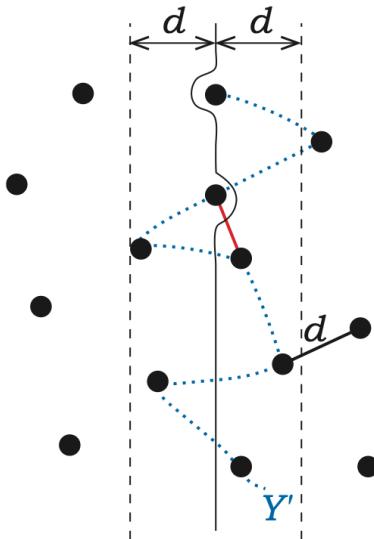
Algorithm 26 Алгоритм Шеймоса

Дано: множество точек P на плоскости.

- 1: $X = P$, отсортированное по координате x
- 2: $Y = P$, отсортированное по координате y
- 3: ближайшая_пара(X, Y)

процедура ближайшая_пара(X, Y)

- 1: Провести вертикальную прямую, делящую множество точек X пополам.
 - 2: Разделить массивы: $X = X_\ell \uplus X_r$ и $Y = Y_\ell \uplus Y_r$.
 - 3: ближайшая_пара(X_ℓ, Y_ℓ)
 - 4: ближайшая_пара(X_r, Y_r)
 - 5: Пусть d — меньшее из двух полученных расстояний.
 - 6: Пусть Y' — точки на расстоянии менее чем d от вертикальной прямой, отсортированные по y .
 - 7: **for all** (x, y) — точка из Y' **do**
 - 8: Проверить расстояние от (x, y) до 7 следующих точек из Y'
-



Нахождение ближайшей пары точек.

Оценка сложности

Сортировки X и Y : $O(n \log n)$ один раз. Рекурсия:

$$T(n) = 2T(n/2) + O(n),$$

так как разбиение Y и проход по Y' линейны. Следовательно, $T(n) = O(n \log n)$.

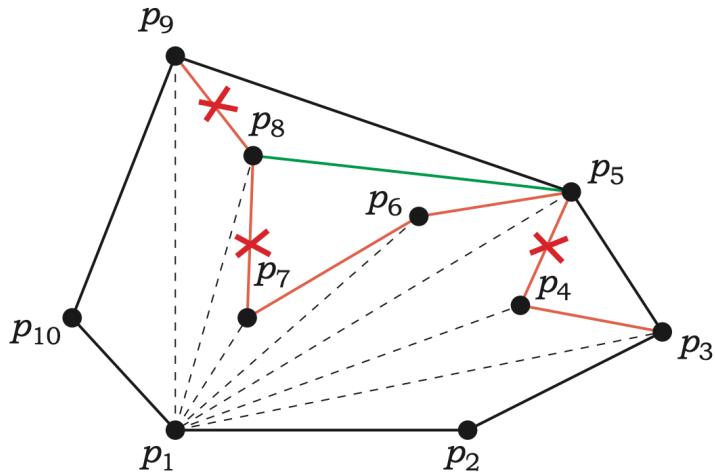
22.3 Выпуклая оболочка (convex hull): просмотр Грэма

Определение

Выпуклая оболочка множества точек P на плоскости — это выпуклый многоугольник, вершины которого являются некоторыми точками из P , и который содержит все точки P (внутри или на границе). Обычно оболочку возвращают как последовательность вершин в порядке против часовой стрелки (CCW).

Идея алгоритма

- Берём опорную точку p_1 — **самую нижнюю** (минимальный y), а при равенстве — **самую левую** (минимальный x).
- Сортируем остальные точки по направлению луча из p_1 **против часовой стрелки**. Если на одном луче несколько точек, оставляем **самую дальнюю** (остальные заведомо внутри/на ребре).
- Идём по отсортированным точкам и поддерживаем стек вершин оболочки: добавляя новую точку p_i , выкидываем вершины сверху стека, пока последние три точки не дают поворот CCW.



Построение выпуклой оболочки: просмотр Грэма.

Algorithm 27 Нахождение выпуклой оболочки: просмотр Грэма

Дано: множество точек P . Найти выпуклую оболочку.

- Пусть $p_1 \in P$ — точка с наименьшей координатой y , самая левая из них.
 - Отсортировать $P \setminus \{p_1\}$ по направлению лучей из p_1 в эти точки, против часовой стрелки. Если на каком-то луче несколько точек, оставить из них самую дальнюю.
 - Пусть $P = \{p_1, \dots, p_n\}$ — полученный отсортированный массив.
 - if** $n < 3$ **then**
 - Все точки на одной прямой (или точек слишком мало), завершить работу.
 - Поместить в стек p_1
 - Поместить в стек p_2
 - Поместить в стек p_3
 - for** $i = 4, \dots, n$ **do**
 - while** отрезки (второй сверху элемент стека) \rightarrow (вершина стека) $\rightarrow p_i$ поворачивают по часовой стрелке **do**
 - Извлечь элемент из стека
 - Поместить в стек p_i
 - return** содержимое стека (в порядке снизу вверх) — выпуклая оболочка.
-

Пример (как работает удаление из стека)

Рассмотрим точки в порядке после сортировки по углу: $p_1, p_2, p_3, p_4, p_5, \dots$

- Сначала стек: (p_1, p_2, p_3) . Переходим к вершине p_4 .

- Отрезки (p_2, p_3) и (p_3, p_4) поворачивают, как и положено, против часовой стрелки, так что вершина p_4 добавляется в стек:
 (p_1, p_2, p_3, p_4) .
- Смотрим на p_5 и видим, что отрезки (p_3, p_4) и (p_4, p_5) развернулись по часовой стрелке. Тогда p_4 не может быть вершиной оболочки для уже рассмотренных точек, поэтому удаляем p_4 :
стек снова (p_1, p_2, p_3) .
- Проверяем ещё раз: теперь рёбра (p_2, p_3) и (p_3, p_5) поворачивают против часовой стрелки, и p_5 добавляется в стек:
 (p_1, p_2, p_3, p_5) .
И так далее.

То есть алгоритм «разочаровывается» в вершинах, которые дали правый поворот, и выкидывает их, пока обход снова не станет CCW.

Лемма

После i -й итерации цикла в стеке находится выпуклая оболочка множества точек $\{p_1, \dots, p_i\}$.

Доказательство. Докажем по индукции по i .

База: при $i = 3$ в стеке лежат (p_1, p_2, p_3) , то есть выпуклая оболочка трёх точек.

Переход: пусть после шага $i - 1$ стек содержит выпуклую оболочку множества $\{p_1, \dots, p_{i-1}\}$. Рассмотрим добавление точки p_i .

Пока последние две вершины стека вместе с p_i образуют поворот *по часовой стрелке*, верхняя вершина стека не может быть вершиной оболочки для множества $\{p_1, \dots, p_i\}$: отрезок, соединяющий предпоследнюю вершину напрямую с p_i , проходит “снаружи” относительно этой верхней вершины, значит она оказывается внутри (или на границе) новой оболочки и её можно удалить.

Алгоритм удаляет такие вершины до тех пор, пока поворот не станет против часовой стрелки. После этого добавление p_i продолжает обход границы оболочки против часовой стрелки, и полученный стек задаёт выпуклую оболочку множества $\{p_1, \dots, p_i\}$.

Переход доказан. □

Сложность. Сортировка по углам: $O(n \log n)$. Каждый элемент заносится в стек один раз и извлекается не более одного раза, поэтому суммарно цикл со стеком работает за $O(n)$. Итого время: $O(n \log n)$.

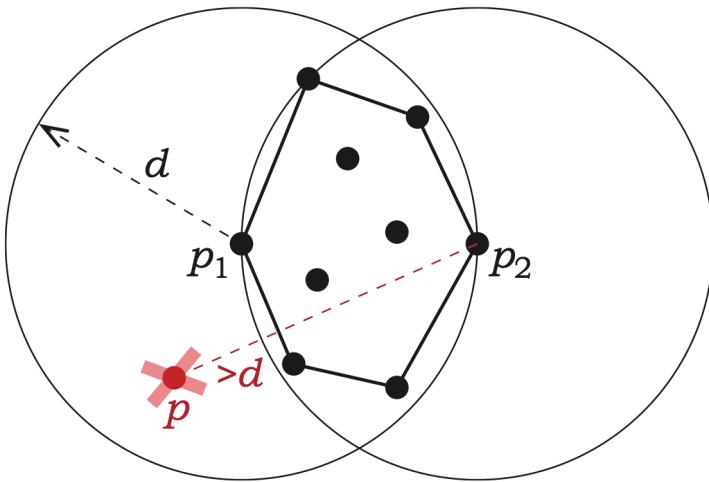
Замечание. В классе алгоритмов, основанных на сравнениях, быстрее $O(n \log n)$ для выпуклой оболочки в общем случае получить нельзя. Также существуют алгоритмы со временем $O(n \log h)$, где h — число вершин оболочки.

22.4 Нахождение наиболее отдалённой пары точек: метод вращающегося штангенциркуля

Постановка

Расстояние между наиболее отдалёнными двумя точками множества P называют **диаметром** этого множества:

$$D(P) = \max_{p,q \in P} \text{dist}(p, q).$$



Наиболее отдалённая пара точек и выпуклая оболочка.

Лемма

Наиболее отдалённая пара точек множества P — это некоторые две вершины выпуклой оболочки.

Доказательство. Пусть p_1, p_2 — наиболее отдалённая пара, и $d = \text{dist}(p_1, p_2)$.

Тогда для любой точки $p \in P$ должно быть одновременно

$$\text{dist}(p, p_1) \leq d \quad \text{и} \quad \text{dist}(p, p_2) \leq d,$$

иначе пара (p, p_1) или (p, p_2) была бы дальше, чем (p_1, p_2) , что невозможно по выбору p_1, p_2 .

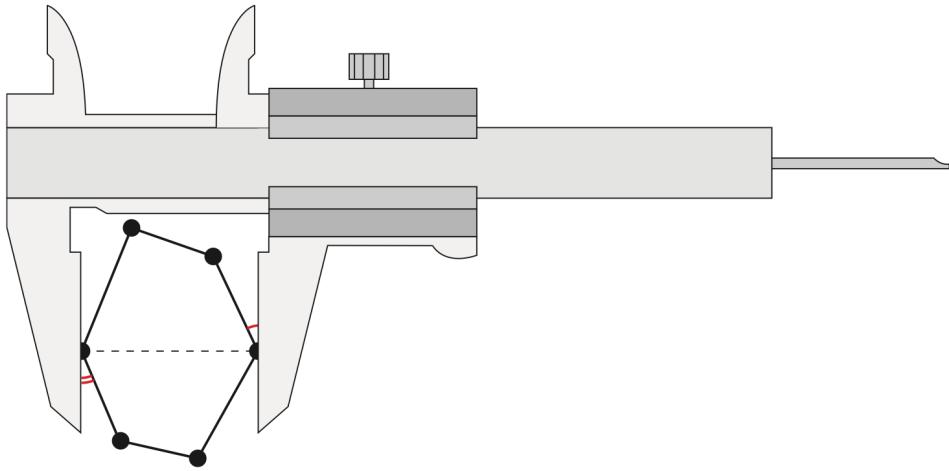
Значит, все точки множества P лежат в пересечении двух кругов радиуса d с центрами в p_1 и p_2 . Следовательно, вся выпуклая оболочка лежит в этом пересечении.

Если бы, например, p_1 не была вершиной выпуклой оболочки (то есть лежала строго внутри оболочки), то крайняя точка оболочки в направлении от p_2 к p_1 находилась бы ещё дальше от p_2 , чем p_1 . Тогда нашлась бы пара точек с расстоянием $> d$ — противоречие.

Аналогично p_2 — вершина оболочки. Значит, искомая наиболее отдалённая пара — две вершины выпуклой оболочки. \square

По лемме достаточно искать диаметр среди вершин выпуклой оболочки. Поэтому:

построить выпуклую оболочку \Rightarrow найти диаметр выпуклого h -угольника.



Противолежащая пара точек и измерение её штангенциркулем.

Противолежащие пары и штангенциркуль

Наивно диаметр выпуклого многоугольника можно искать перебором всех пар вершин за $O(h^2)$, но можно быстрее.

Противолежащая (antipodal) пара — это пара вершин (p_i, p_j) такая, что существуют две параллельные касательные прямые к многоугольнику, одна касается в p_i , другая — в p_j , и весь многоугольник лежит между этими прямыми (как между губками штангенциркуля).

Диаметр выпуклого многоугольника достигается на некоторой antipodal-паре. Значит, достаточно перебрать все antipodal-пары и взять максимум расстояния.

Метод вращающегося штангенциркуля (rotating calipers). Пусть выпуклый многоугольник задан вершинами p_1, \dots, p_h .

Рассмотрим две параллельные прямые (“губки”), которые касаются многоугольника с двух сторон.

В некотором положении губки касаются двух рёбер (p_i, p_{i+1}) и (p_j, p_{j+1}) ; соответствующая пара вершин (p_i, p_j) — противолежащая.

Дальше алгоритм *поворачивает* обе прямые, сохраняя параллельность, на минимальный угол так, чтобы **одно из двух касаемых рёбер сменилось на следующее**:

либо $(p_i, p_{i+1}) \rightarrow (p_{i+1}, p_{i+2})$ (тогда $i = i + 1$),

либо $(p_j, p_{j+1}) \rightarrow (p_{j+1}, p_{j+2})$ (тогда $j = j + 1$).

После каждого такого шага получаем новую противолежащую пару и можем обновить максимум расстояния $\text{dist}(p_i, p_j)$.

Так как на каждом шаге увеличивается ровно один из индексов i или j , а каждый из них делает не более h увеличений за полный оборот, все противолежащие пары перечисляются за $O(h)$.

Сложность

- Построение выпуклой оболочки: $O(n \log n)$ (например, просмотр Грэма).
- Перебор antipodal-пар методом штангенциркуля: $O(h)$,

Итого: $O(n \log n)$.