

# Конспект по АиСД ЭАД

Версия от 23.11.2025

Третьяков Радомир, Петров Егор

## Содержание

<b>1</b>	<b>Немного теории чисел</b>	<b>3</b>
1.1	Базовые обозначения . . . . .	3
1.2	Простые числа . . . . .	3
<b>2</b>	<b>Алгоритм Евклида</b>	<b>3</b>
2.1	Классический алгоритм для НОД . . . . .	3
2.2	Реализация на Python . . . . .	4
2.3	Пример руками . . . . .	4
2.4	Оценка сложности . . . . .	4
<b>3</b>	<b>Расширенный алгоритм Евклида</b>	<b>4</b>
3.1	Реализация . . . . .	5
<b>4</b>	<b>Модульная арифметика и быстрое возведение в степень</b>	<b>5</b>
4.1	Что такое «по модулю» . . . . .	5
4.2	Обратные элементы по модулю . . . . .	6
<b>5</b>	<b>Как найти обратное число по модулю</b>	<b>6</b>
5.1	Быстрое возведение в степень по модулю . . . . .	7
5.2	Малая теорема Ферма . . . . .	8
<b>6</b>	<b>Малая теорема Ферма: интуитивное объяснение</b>	<b>9</b>
<b>7</b>	<b>Решето Эратосфена</b>	<b>10</b>
7.1	Идея алгоритма . . . . .	10
7.2	Реализация . . . . .	11
7.3	Сколько это работает . . . . .	11

<b>8</b>	<b>Факторизация чисел</b>	<b>11</b>
8.1	Наивный метод: простое деление . . . . .	11
8.2	Зачем нам всё это . . . . .	12
8.3	Решение задач по пройденным темам . . . . .	13
<b>9</b>	<b>Алгоритмы сортировки</b>	<b>18</b>
9.1	Сортировка пузырьком (Bubble sort) . . . . .	18
9.2	Шейкерная сортировка (Shaker sort) . . . . .	20
9.3	Сортировка расчёской (Comb sort) . . . . .	23
9.4	Сортировка вставками (Insertion sort) . . . . .	25
9.5	Сортировка Шелла (Shellsort) . . . . .	28
9.6	Сортировка выбором (Selection sort) . . . . .	30
9.7	Пирамидальная сортировка (Heapsort) . . . . .	31
9.8	Быстрая сортировка (Quicksort) . . . . .	32
9.9	Сортировка слиянием (Merge sort) . . . . .	34
9.10	Сортировка подсчётом (Counting sort) . . . . .	36
9.11	Поразрядные сортировки (Radix sort) . . . . .	37
9.12	Сортировка деревом (Tree sort) . . . . .	40
9.13	Timsort . . . . .	42
<b>10</b>	<b>Инверсии и связь с сортировками</b>	<b>43</b>
10.1	Определение и интерпретация . . . . .	43
10.2	Инверсии на массиве $[2, 4, 5, 6, 3, 1]$ . . . . .	43
10.3	Связь с сортировкой вставками . . . . .	44
10.4	Подсчёт инверсий через сортировку слиянием . . . . .	44

# 1 Немного теории чисел

Всю дорогу будем работать с целыми числами.

## 1.1 Базовые обозначения

- $\mathbb{Z}$  — множество всех целых чисел.
- $\mathbb{N}$  — множество натуральных чисел (будем считать, что это  $1, 2, 3, \dots$ ).
- $a \mid b$  — « $a$  делит  $b$ », то есть  $b = ak$  для некоторого целого  $k$ .
- $\gcd(a, b)$  — наибольший общий делитель чисел  $a$  и  $b$ .
- $a \equiv b \pmod{m}$  — числа  $a$  и  $b$  дают один и тот же остаток при делении на  $m$ .

Чуть-чуть про асимптотику (чтобы не гадать «быстро ли это работает»):

- $O(1)$  — время почти не зависит от входа (константное время);
- $O(\log n)$  — растёт медленно, логарифмически;
- $O(n)$  — время пропорционально размеру входа (сканируем массив, строку и т.п.);
- $O(n \log n)$  — типичная сложность «хороших» сортировок;
- $O(n^2)$  и больше — уже может тормозить, если  $n$  велико.

## 1.2 Простые числа

Напоминание: **простое** число  $p > 1$  — это такое  $p$ , у которого нет делителей, кроме 1 и самого  $p$ .

Факты, которые будем использовать без доказательства:

- Любое  $n > 1$  единственным образом раскладывается в произведение простых чисел (с точностью до порядка множителей).
- Простых чисел бесконечно много.

Дальше нам нужны инструменты, как с этим всем работать алгоритмически.

# 2 Алгоритм Евклида

## 2.1 Классический алгоритм для НОД

Задача: по двум целым  $a$  и  $b$  найти  $\gcd(a, b)$ .

**Лемма 2.1.** Для любых целых  $a, b$  (не обоих нулевых) верно:

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

*Идея.* Пусть  $a = qb + r$ , где  $r = a \bmod b$ .

- Если число  $d$  делит и  $a$ , и  $b$ , то оно делит и  $r = a - qb$ .

- Если  $d$  делит  $b$  и  $r$ , то оно делит и  $a = qb + r$ .

Значит, множества общих делителей  $(a, b)$  и  $(b, r)$  совпадают, а значит и их НОД одинаков.  $\square$

Из леммы следует простой алгоритм: пока второй аргумент не стал нулём, заменяем пару  $(a, b)$  на  $(b, a \bmod b)$ .

## 2.2 Реализация на Python

---

**Algorithm 1** Алгоритм Евклида

---

```
1 def gcd(a: int, b: int) -> int:
2     while b != 0:
3         a, b = b, a % b
4     return abs(a)
```

---

В стандартной библиотеке есть `math.gcd`, но лучше один раз написать самому, чтобы понимать, что происходит.

## 2.3 Пример руками

Найдём  $\gcd(252, 105)$ .

$$252 = 105 \cdot 2 + 42,$$

$$105 = 42 \cdot 2 + 21,$$

$$42 = 21 \cdot 2 + 0.$$

Последний ненулевой остаток — 21, значит  $\gcd(252, 105) = 21$ .

## 2.4 Оценка сложности

Если совсем грубо, то количество шагов алгоритма Евклида порядка  $O(\log \min(a, b))$ .

То есть даже для огромных чисел алгоритм будет работать очень быстро по сравнению, например, с «перебором всех делителей».

# 3 Расширенный алгоритм Евклида

Здесь та же идея, но бонусом хотим найти коэффициенты в линейной комбинации.

**Определение 3.1.** Коэффициенты Безу для чисел  $a$  и  $b$  — это такие целые  $x, y$ , что

$$ax + by = \gcd(a, b).$$

Расширенный алгоритм Евклида как раз и возвращает НОД и эти самые  $x, y$ .

### 3.1 Реализация

---

**Algorithm 2** Расширенный алгоритм Евклида

---

```
1 from typing import Tuple
2
3 def extended_gcd(a: int, b: int) -> Tuple[int, int, int]:
4     """
5     (d, x, y), d = gcd(a, b) ax + by = d.
6     """
7     if b == 0:
8         # gcd(a, 0) = |a|
9         return abs(a), 1 if a >= 0 else -1, 0
10
11     d, x1, y1 = extended_gcd(b, a % b)
12     # a = b * (a // b) + (a % b)
13     # d = x1 * b + y1 * (a % b)
14     x = y1
15     y = x1 - (a // b) * y1
16     return d, x, y
```

---

Эта функция нам пригодится для модульной арифметики.

## 4 Модульная арифметика и быстрое возведение в степень

### 4.1 Что такое «по модулю»

Говорим, что  $a \equiv b \pmod{m}$ , если  $m$  делит разность  $a - b$ . То есть  $a$  и  $b$  дают одинаковый остаток при делении на  $m$ .

Вычеты по модулю  $m$  ведут себя привычно:

$$a \equiv b \pmod{m}, \quad c \equiv d \pmod{m} \Rightarrow \begin{cases} a + c \equiv b + d \pmod{m}, \\ ac \equiv bd \pmod{m}. \end{cases}$$

Часто удобнее сразу работать только с остатками: хранить числа в диапазоне от 0 до  $m - 1$  и всё время брать  $\% m$ .

## 4.2 Обратные элементы по модулю

Если  $\gcd(a, m) = 1$ , то существует такое  $x$ , что

$$ax \equiv 1 \pmod{m}.$$

Число  $x$  называется *обратным* к  $a$  по модулю  $m$ .

Через расширенный алгоритм Евклида это делается очень быстро.

```
1 def mod_inverse(a: int, m: int) -> int | None:
2
3     d, x, _ = extended_gcd(a, m)
4     if d != 1:
5         return None
6     return x % m
```

## 5 Как найти обратное число по модулю

Пусть нам дано число  $a$  и модуль  $m$ . Мы хотим найти такое число  $x$ , что

$$ax \equiv 1 \pmod{m}.$$

Число  $x$  и называется **обратным к  $a$  по модулю  $m$** . Иными словами, мы хотим найти “деление по модулю”:

$$x = a^{-1} \pmod{m}.$$

### Когда обратный элемент существует

Обратный существует только тогда, когда

$$\gcd(a, m) = 1.$$

Это легко понять: если  $a$  и  $m$  имеют общий делитель  $d > 1$ , то левая часть  $ax$  всегда будет делиться на  $d$ , а правая часть  $(1)$  — нет. Значит, уравнение невозможно.

### Как найти обратный элемент

Есть два основных способа.

## 1. Через расширенный алгоритм Евклида

Расширенный алгоритм Евклида находит такие числа  $x$  и  $y$ , что

$$ax + my = \gcd(a, m).$$

Если  $\gcd(a, m) = 1$ , это принимает вид

$$ax + my = 1.$$

Переходим к классам по модулю  $m$ . Второй член пропадает:

$$ax \equiv 1 \pmod{m}.$$

Но это ровно то, что мы ищем! Значит,  $x$  — и есть обратный элемент (его нужно привести по модулю  $m$ ).

Например, хотим найти обратный элемент к  $a = 7$  по модулю  $m = 26$ . Расширенный алгоритм Евклида даёт:

$$7 \cdot 15 + 26 \cdot (-4) = 1.$$

Отсюда:

$$7 \cdot 15 \equiv 1 \pmod{26},$$

значит

$$7^{-1} \equiv 15 \pmod{26}.$$

### 5.1 Быстрое возведение в степень по модулю

Часто требуется вычислить величину  $a^e \bmod m$ , причём показатель  $e$  может быть очень большим. Наивный подход — перемножать  $a$  само на себя  $e$  раз — работает слишком медленно. Вместо этого используется простой приём, основанный на разборе чётности показателя.

Заметим, что для любого  $e$  справедливо:

- если  $e$  чётное, то

$$a^e = a^{2 \cdot (e/2)} = (a^2)^{e/2};$$

- если  $e$  нечётное, то

$$a^e = a \cdot a^{e-1},$$

причём показатель  $e - 1$  уже чётный.

Тем самым алгоритм выглядит так:

1. пока  $e > 0$ :

- если  $e$  нечётное, умножаем результат на  $a$  и уменьшаем показатель на 1;

- если  $e$  чётное, заменяем  $a$  на  $a^2 \bmod m$  и делим  $e$  пополам.

Так как на каждом шаге показатель либо уменьшается на 1, либо делится на 2, время работы алгоритма составляет порядка  $\mathcal{O}(\log e)$ .

---

**Algorithm 3** Быстрое возведение в степень по модулю
 

---

```

1 def pow_mod(a, e, m):
2     a %= m
3     result = 1
4
5     while e > 0:
6         if e % 2 == 1:
7             result = (result * a) % m
8
9         a = (a * a) % m
10        e //= 2
11
12    return result

```

---

В Python есть встроенная функция `pow(base, exponent, modulus)`, которая делает примерно то же самое, так что в реальном коде имеет смысл пользоваться именно ей.

## 5.2 Малая теорема Ферма

**Теорема 5.1** (Малая теорема Ферма). *Если  $p$  — простое число и  $p \nmid a$ , то*

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

*Эквивалентная формулировка:*

$$a^p \equiv a \pmod{p}$$

*для любого целого  $a$ .*

Идея доказательства: если взять все ненулевые остатки по модулю  $p$ , умножить их на  $a$  и посмотреть на произведение, получится та же самая совокупность классов, только переставленная. Отсюда и вылезает равенство.

Практическое применение: можно считать обратный элемент по модулю простого числа  $p$  как

$$a^{-1} \equiv a^{p-2} \pmod{p}.$$

То есть `pow(a, p-2, p)`.



## 6 Малая теорема Ферма: интуитивное объяснение

Рассмотрим простое число  $p$  и целое число  $a$ , которое не делится на  $p$ . Малая теорема Ферма утверждает:

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

Чтобы понять, почему это работает, посмотрим на всё “на пальцах”.

### Идея через остатки

Возьмём все ненулевые остатки по модулю  $p$ :

$$1, 2, 3, \dots, p-1.$$

Теперь умножим каждый из них на число  $a$ :

$$a, 2a, 3a, \dots, (p-1)a.$$

Рассмотрим эти величины по модулю  $p$ . Оказывается, при делении на  $p$  они дают *те же самые* остатки  $1, 2, \dots, p-1$ , только в другом порядке.

Почему? Если бы два разных числа давали один и тот же остаток:

$$ka \equiv la \pmod{p},$$

то можно было бы сократить на  $a$  (ведь оно не делится на  $p$ ), и получить:

$$k \equiv l \pmod{p},$$

что невозможно. Значит, умножение на  $a$  просто переставляет все остатки.

### Смотрим на произведения

Произведение исходных остатков:

$$1 \cdot 2 \cdot 3 \cdots (p-1).$$

Произведение новых остатков:

$$a \cdot 2a \cdot 3a \cdots (p-1)a = a^{p-1}(1 \cdot 2 \cdot 3 \cdots (p-1)).$$

А так как множества остатков одинаковые (просто в другом порядке), их произве-

дения по модулю  $p$  совпадают:

$$a^{p-1}(1 \cdot 2 \cdot \dots \cdot (p-1)) \equiv 1 \cdot 2 \cdot \dots \cdot (p-1) \pmod{p}.$$

Число  $1 \cdot 2 \cdot \dots \cdot (p-1)$  не делится на  $p$ , поэтому его можно сократить, получая итог:

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

## Пример для наглядности

Пусть  $p = 7$ , возьмём  $a = 3$ . Считаем степени по модулю 7:

$$3^1 \equiv 3,$$

$$3^2 \equiv 9 \equiv 2,$$

$$3^3 \equiv 6,$$

$$3^4 \equiv 18 \equiv 4,$$

$$3^5 \equiv 12 \equiv 5,$$

$$3^6 \equiv 15 \equiv 1.$$

Получили:

$$3^6 \equiv 1 \pmod{7},$$

что полностью совпадает с формулой  $3^{p-1} = 3^6$ .

## Практическое применение

Если модуль  $p$  — простое число, то мы можем находить обратный элемент:

$$a^{-1} \equiv a^{p-2} \pmod{p},$$

и это вычисляется с помощью быстрого возведения в степень.

## 7 Решето Эратосфена

Теперь хотим уметь быстро находить все простые числа до заданного  $N$ .

### 7.1 Идея алгоритма

1. Создаём булев массив `is_prime` размером  $N + 1$  и считаем, что все числа от 2 до  $N$  — простые.
2. Идём по  $p$  от 2 до  $\lfloor \sqrt{N} \rfloor$ . Если `is_prime[p]` всё ещё `True`, вычёркиваем все кратные  $p$ , начиная с  $p^2$ .

3. В конце все индексы, помеченные как `True`, — это простые числа.

Почему начинаем с  $p^2$ : всё, что меньше, уже вычеркнули на предыдущих шагах.

## 7.2 Реализация

---

**Algorithm 4** Решето Эратосфена

---

```
1 def sieve(n: int):
2     if n < 2:
3         return []
4
5     is_prime = [True] * (n + 1)
6     is_prime[0] = is_prime[1] = False
7
8     limit = n ** 0.5
9     i = 2
10    while i <= limit:
11        if is_prime[i]:
12            for j in range(i * i, n + 1, i):
13                is_prime[j] = False
14        i += 1
15
16    return [k for k in range(2, n + 1) if is_prime[k]]
```

---

## 7.3 Сколько это работает

- Память:  $O(N)$  на массив `is_prime`.
- Время:  $O(N \log \log N)$  — очень быстро для разумных  $N$  (скажем, до  $10^8$  при аккуратной реализации).

## 8 Факторизация чисел

Задача: по целому  $n$  найти его разложение на простые множители.

### 8.1 Наивный метод: простое деление

Самый прямолинейный способ — просто пробовать делители.

---

**Algorithm 5** Факторизация простым делением

---

```
1 def trial_division(n: int) -> list[int]:
2     factors = []
3     d = 2
4
5     while d * d <= n:
6         while n % d == 0:
7             factors.append(d)
8             n //= d
9         if d == 2:
10            d += 1
11        else:
12            d += 2
13    if n > 1:
14        factors.append(n)
15
16    return factors
```

---

Этот метод удивительно неплох, если числа не слишком большие.

## 8.2 Зачем нам всё это

Коротко, где всплывают эти алгоритмы:

- Алгоритм Евклида и его расширенная версия — основа почти всей модульной арифметики.
- Быстрое возведение в степень нужно в тестах простоты и криптографии.
- Решето Эратосфена — стандартный способ сгенерировать много простых чисел сразу.
- Факторизация (в т.ч. Pollard's Rho) — шаг в сторону реальных криптосистем и оценки их стойкости.

### 8.3 Решение задач по пройденным темам

#### Задача 1. Count The Divisors

**Условие задачи.**

Даны два целых числа  $l$  и  $r$ . Необходимо посчитать общее количество *различных простых делителей* всех целых чисел на отрезке  $[l, r]$  включительно.

Иными словами, нужно рассмотреть все числа  $l, l + 1, \dots, r$ , выписать все их простые делители, затем оставить только **уникальные** простые числа и посчитать их количество.

**Пример.**

Пусть  $l = 5, r = 15$ .

Числа на отрезке: 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.

Их простые делители:

5 : 5  
6 : 2, 3  
7 : 7  
8 : 2  
9 : 3  
10 : 2, 5  
11 : 11  
12 : 2, 3  
13 : 13  
14 : 2, 7  
15 : 3, 5

Множество различных простых делителей:

$\{2, 3, 5, 7, 11, 13\}$ ,

всего 6 чисел.

Значит, ответ для этого примера равен 6.

**Формат ввода**

Первая строка входных данных содержит два целых числа  $l$  и  $r$ :

$$1 \leq l \leq r \leq 2 \cdot 10^6.$$

**Формат вывода**

Выведите одно целое число — количество различных простых делителей всех целых чисел на отрезке  $[l, r]$ .

### Пример

Ввод	Вывод
5 15	6

### Python-код решения

```
1 l, r = map(int, input().split())
2
3 prime = [True] * (r + 1)
4 prime[0] = prime[1] = False
5 primes = []
6
7 for i in range(2, r + 1):
8     if prime[i]:
9         primes.append(i)
10        for j in range(i * i, r + 1, i):
11            prime[j] = False
12
13 ans = 0
14
15 for p in primes:
16     first = ((l + p - 1) // p) * p
17     if first <= r:
18         ans += 1
19
20 print(ans)
```

## Задача 2. Сложность двоичного поиска (Быки и коровы)

### Условие.

Загадано некоторое целое число от 1 до  $N$  включительно. Вы можете задавать вопросы вида: «Загаданное число это  $x$ ?» и получать один из трёх ответов:

- «Угадал число.»
- «Загаданное число больше.»
- «Загаданное число меньше.»

Используя стратегию **деления пополам** (двоичный поиск), нужно определить:

Какое наименьшее количество вопросов нужно гарантированно задать, чтобы точно угадать число?

### Формат ввода

Вводится одно целое число  $N$  ( $1 \leq N \leq 10^5$ ).

### Формат вывода

Выведите одно целое число — минимальное количество вопросов, которых гарантированно достаточно, чтобы угадать загаданное число на отрезке  $[1, N]$ .

### Комментарий к задаче (идея решения).

Метод деления пополам — это *двоичный поиск*. После каждого вопроса множество возможных чисел делится примерно пополам:

- после 1 вопроса остаётся не более  $N/2$  вариантов,
- после 2 вопросов — не более  $N/4$ ,
- после  $k$  вопросов — не более  $N/2^k$ .

Нужно, чтобы после некоторого количества шагов  $k$  гарантированно осталось не больше одного возможного числа:

$$\frac{N}{2^k} \leq 1 \iff 2^k \geq N.$$

Минимальное такое  $k$  — это

$$k = \lceil \log_2 N \rceil.$$

Именно столько вопросов в худшем случае потребуется при оптимальной стратегии двоичного поиска.

### Пример

Ввод	Вывод
8	3

Пояснение:  $2^3 = 8$ , значит за 3 вопроса можно гарантированно определить число от 1 до 8.

### Решение на Python

Ниже приведён код, который вычисляет минимальное  $k$  такое, что  $2^k \geq N$ , используя целочисленный цикл (без плавающей арифметики):

```
1 def min_questions(n: int) -> int:
2     k = 0
3     cur = 1
```

```

4   while cur < n:
5       cur *= 2
6       k += 1
7   return k

```

### Задача 3. Суммирование по составным

#### Условие.

Задано целое число  $N$  ( $4 \leq N \leq 10^6$ ). Нужно найти сумму *наименьших простых делителей* всех **составных** чисел, которые:

- строго больше 2,
- не превосходят  $N$ .

Напомним, что число называется *составным*, если оно больше 1 и не является простым (то есть имеет нетривиальные делители).

#### Формат ввода

Вводится одно целое число  $N$ .

#### Формат вывода

Выведите одно целое число — сумму наименьших простых делителей всех составных чисел  $x$ , таких что  $2 < x \leq N$ .

#### Примеры

Ввод	Вывод
5	2
9	9

Пояснение ко второму примеру: составные числа от 3 до 9 — это 4, 6, 8, 9.

4 : наименьший простой делитель = 2,

6 : 2,

8 : 2,

9 : 3.

Сумма:  $2 + 2 + 2 + 3 = 9$ .

**Основная идея** Вместо того чтобы для каждого числа отдельно искать его наименьший простой делитель, мы используем эффективный алгоритм, который заранее вычисляет наименьшие простые делители для всех чисел от 2 до  $N$ .



## Идея решения

Для эффективного решения задачи используется модифицированный алгоритм решета Эратосфена, который позволяет для каждого числа от 2 до  $N$  найти его наименьший простой делитель.

## Алгоритм:

1. Создаём массив `min_prime` размером  $N + 1$ , где `min_prime[i]` будет хранить наименьший простой делитель числа  $i$
2. Инициализируем массив: для каждого  $i$  от 2 до  $N$  полагаем `min_prime[i] = i`
3. Для каждого числа  $i$  от 2 до  $\sqrt{N}$ :
  - Если `min_prime[i] == i` (т.е.  $i$  — простое число)
  - Для всех чисел  $j = i^2, i^2 + i, i^2 + 2i, \dots \leq N$ :
    - Если `min_prime[j] == j`, то устанавливаем `min_prime[j] = i`
4. Суммируем значения `min_prime[i]` для всех составных чисел от 4 до  $N$

## Обоснование корректности:

- У каждого составного числа есть единственный наименьший простой делитель
- Решето Эратосфена эффективно находит все простые числа до  $N$
- Модификация позволяет сразу отмечать наименьшие простые делители для всех составных чисел

## Пример для $N = 9$ :

- Составные числа: 4, 6, 8, 9
- Наименьшие простые делители: 2, 2, 2, 3
- Сумма:  $2 + 2 + 2 + 3 = 9$

## Решение на Python

```
1 n = int(input())
2 min_prime = [0] * (n + 1)
3
4 for i in range(2, n + 1):
5     min_prime[i] = i
6
7 for i in range(2, int(n**0.5) + 1):
8     if min_prime[i] == i:
9         for j in range(i * i, n + 1, i):
10             if min_prime[j] == j:
11                 min_prime[j] = i
12 total = 0
13 for i in range(4, n + 1):
14     if min_prime[i] != i:
```

```

15         total += min_prime[i]
16
17 print(total)

```

## 9 Алгоритмы сортировки

В этом разделе рассматриваются классические алгоритмы сортировки, основанные на сравнениях и на работе с разрядами/значениями. Основной объект: массив (список) чисел  $a[0..n-1]$ , который требуется упорядочить по возрастанию.

В качестве базового примера работы алгоритмов будем использовать массив

$[2, 4, 5, 6, 3, 1]$ .

Мы будем обращать внимание на:

- инварианты алгоритма (что «поддерживается» на каждом шаге);
- поведение на различных типах входов (почти отсортированный, случайный, обратно отсортированный);
- асимптотику в разных случаях;
- стабильность и использование памяти;
- связь между алгоритмами (например, вставки vs инверсии, слияние vs подсчёт инверсий);

### 9.1 Сортировка пузырьком (Bubble sort)

#### Идея и инвариант

Сортировка пузырьком — прототип всех обменных алгоритмов.

Основной шаг: проход по массиву слева направо, для каждой пары соседей  $(a[i], a[i+1])$ :

$$a[i] > a[i+1] \Rightarrow \text{swap}(a[i], a[i+1]).$$

После одного полного прохода выполняется следующий инвариант:

максимальный элемент в  $a[0..n-1]$  находится в позиции  $n-1$ .

На  $k$ -й итерации внешний цикл «отсекает» правые  $k$  элементов: они уже стоят на своих окончательных позициях, и дальше алгоритм работает с суффиксом  $a[0..n-k-1]$ .

Если на очередном проходе не было ни одного обмена, это эквивалентно отсутствию инверсий в массиве, следовательно, массив уже отсортирован, и алгоритм может

остановиться раньше, чем через  $n - 1$  проходов.

**Пример на массиве**  $[2, 4, 5, 6, 3, 1]$

Начальный массив:

$$[2, 4, 5, 6, 3, 1].$$

Первый проход (слева направо):

$$[2, 4, 5, 6, 3, 1] \rightarrow [2, 4, 5, 3, 6, 1] \quad (6 > 3),$$

$$[2, 4, 5, 3, 6, 1] \rightarrow [2, 4, 5, 3, 1, 6] \quad (6 > 1).$$

После первого прохода максимум 6 стоит в конце:

$$[2, 4, 5, 3, 1, 6].$$

Второй проход:

$$[2, 4, 5, 3, 1, 6] \rightarrow [2, 4, 3, 5, 1, 6] \rightarrow [2, 4, 3, 1, 5, 6].$$

Третий:

$$[2, 4, 3, 1, 5, 6] \rightarrow [2, 3, 4, 1, 5, 6] \rightarrow [2, 3, 1, 4, 5, 6].$$

Четвёртый:

$$[2, 3, 1, 4, 5, 6] \rightarrow [2, 1, 3, 4, 5, 6].$$

Пятый:

$$[2, 1, 3, 4, 5, 6] \rightarrow [1, 2, 3, 4, 5, 6].$$

На следующем проходе обменов не будет — сортировка завершена.

### **Асимптотика и свойства**

$$T_{\text{best}}(n) = O(n), \quad T_{\text{avg}}(n) = O(n^2), \quad T_{\text{worst}}(n) = O(n^2).$$

Свойства:

- Память:  $O(1)$ .
- Стабильный алгоритм.

### **Реализация**

```

1 def bubble_sort(a):
2     n = len(a)
3     for i in range(n):
4         f = 0
5         for j in range(n - 1 - i):
6             if a[j] > a[j + 1]:
7                 a[j], a[j + 1] = a[j + 1], a[j]
8                 f = 1
9         if f == 0:
10            break

```

## 9.2 Шейкерная сортировка (Shaker sort)

### Идея и инвариант

Шейкерная (cocktail) сортировка — двунаправленная модификация пузырька. Пузырёк хорошо двигает максимальные элементы вправо, но минимальные (“черепахи”) двигаются влево очень медленно: на 1 позицию за проход.

Шейкерная сортировка решает это так:

- прямой проход (слева направо) продвигает максимум к правой границе;
- обратный проход (справа налево) продвигает минимум к левой границе.

Поддерживаются индексы **b** и **e** — левая и правая границы неотсортированного сегмента. После прямого прохода **e--**, после обратного — **b++**.

Инварианты после каждой «двойной» итерации:

$a[0..b - 1]$  содержит минимальные элементы в отсортированном порядке,  
 $a[e + 1..n - 1]$  содержит максимальные элементы в отсортированном порядке.

### Исходный массив

[7, 3, 9, 1, 6, 2, 8, 5, 4, 0]

## Шейкерная сортировка (Cocktail Shaker Sort)

### Прямой проход №1

Последовательно сравниваем пары элементов слева направо.

- $7 > 3$  — меняем местами: [3, 7, 9, 1, 6, 2, 8, 5, 4, 0]
- $7 < 9$  — ничего не делаем
- $9 > 1$  — меняем местами: [3, 7, 1, 9, 6, 2, 8, 5, 4, 0]

- $9 > 6$  — меняем местами:  $[3, 7, 1, 6, 9, 2, 8, 5, 4, 0]$
- $9 > 2$  — меняем местами:  $[3, 7, 1, 6, 2, 9, 8, 5, 4, 0]$
- $9 > 8$  — меняем местами:  $[3, 7, 1, 6, 2, 8, 9, 5, 4, 0]$
- $9 > 5$  — меняем местами:  $[3, 7, 1, 6, 2, 8, 5, 9, 4, 0]$
- $9 > 4$  — меняем местами:  $[3, 7, 1, 6, 2, 8, 5, 4, 9, 0]$
- $9 > 0$  — меняем местами:  $[3, 7, 1, 6, 2, 8, 5, 4, 0, 9]$

Наибольший элемент “всплыл” вправо.

### Обратный проход №1

Теперь идём справа налево (от предпоследнего до первого элемента).

- $0 < 4$  — меняем местами:  $[3, 7, 1, 6, 2, 8, 5, 0, 4, 9]$
- $0 < 5$  — меняем местами:  $[3, 7, 1, 6, 2, 8, 0, 5, 4, 9]$
- $0 < 8$  — меняем местами:  $[3, 7, 1, 6, 2, 0, 8, 5, 4, 9]$
- $0 < 2$  — меняем местами:  $[3, 7, 1, 6, 0, 2, 8, 5, 4, 9]$
- $0 < 6$  — меняем местами:  $[3, 7, 1, 0, 6, 2, 8, 5, 4, 9]$
- $0 < 1$  — меняем местами:  $[3, 7, 0, 1, 6, 2, 8, 5, 4, 9]$
- $0 < 7$  — меняем местами:  $[3, 0, 7, 1, 6, 2, 8, 5, 4, 9]$
- $0 < 3$  — меняем местами:  $[0, 3, 7, 1, 6, 2, 8, 5, 4, 9]$

Наименьший элемент “опустился” влево.

### Прямой проход №2

Идём слева направо (от второго до предпоследнего элемента).

- $3 < 7$  — ничего не делаем
- $7 > 1$  — меняем местами:  $[0, 3, 1, 7, 6, 2, 8, 5, 4, 9]$
- $7 > 6$  — меняем местами:  $[0, 3, 1, 6, 7, 2, 8, 5, 4, 9]$
- $7 > 2$  — меняем местами:  $[0, 3, 1, 6, 2, 7, 8, 5, 4, 9]$
- $7 < 8$  — ничего не делаем
- $8 > 5$  — меняем местами:  $[0, 3, 1, 6, 2, 7, 5, 8, 4, 9]$
- $8 > 4$  — меняем местами:  $[0, 3, 1, 6, 2, 7, 5, 4, 8, 9]$

### Обратный проход №2

Идём справа налево (от предпоследнего до второго элемента).

- $4 < 5$  — меняем местами:  $[0, 3, 1, 6, 2, 7, 4, 5, 8, 9]$
- $4 < 7$  — меняем местами:  $[0, 3, 1, 6, 2, 4, 7, 5, 8, 9]$
- $4 > 2$  — ничего не делаем
- $2 < 6$  — меняем местами:  $[0, 3, 1, 2, 6, 4, 7, 5, 8, 9]$
- $2 > 1$  — ничего не делаем
- $1 < 3$  — меняем местами:  $[0, 1, 3, 2, 6, 4, 7, 5, 8, 9]$

## Прямой проход №3

Идём слева направо (от второго до предпоследнего элемента).

- $1 < 3$  — ничего не делаем
- $3 > 2$  — меняем местами:  $[0, 1, 2, 3, 6, 4, 7, 5, 8, 9]$
- $3 < 6$  — ничего не делаем
- $6 > 4$  — меняем местами:  $[0, 1, 2, 3, 4, 6, 7, 5, 8, 9]$
- $6 < 7$  — ничего не делаем
- $7 > 5$  — меняем местами:  $[0, 1, 2, 3, 4, 6, 5, 7, 8, 9]$
- $7 < 8$  — ничего не делаем

## Обратный проход №3

Идём справа налево (от предпоследнего до третьего элемента).

- $5 < 6$  — меняем местами:  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$

Массив полностью отсортирован.

## Результат

$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$

\*Итог

$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$

Алгоритм завершил работу.

## Асимптотика и свойства

С точки зрения асимптотики шейкер аналогичен пузырьку:

$$T_{\text{best}}(n) = \Theta(n), \quad T_{\text{avg}}(n) = \Theta(n^2), \quad T_{\text{worst}}(n) = \Theta(n^2).$$

Свойства:

- Память:  $O(1)$ .
- Алгоритм стабилен.

## Реализация

```
1 def shaker_sort(a):
2     n = len(a)
3     if n <= 1:
4         return
5     b = 0
```

```

6     e = n - 1
7     while b < e:
8         f = 0
9         for i in range(b, e):
10            if a[i] > a[i + 1]:
11                a[i], a[i + 1] = a[i + 1], a[i]
12                f = 1
13        e -= 1
14        if f == 0:
15            break
16        f = 0
17        for i in range(e, b, -1):
18            if a[i - 1] > a[i]:
19                a[i - 1], a[i] = a[i], a[i - 1]
20                f = 1
21        b += 1
22        if f == 0:
23            break

```

## 9.3 Сортировка расчёской (Comb sort)

### Мотивация и схема

Комбинирует идею пузырька с «дальними» обменами. Главный недостаток пузырька: минимальный элемент из хвоста должен сделать  $O(n)$  локальных шагов, чтобы попасть в начало.

Comb sort вводит параметр `gap` — расстояние между сравниваемыми элементами. Изначально `gap = n`, затем на каждом шаге

$$\text{gap}_n := \left\lfloor \frac{\text{gap}_{n-1}}{\text{shrink}} \right\rfloor, \quad \text{shrink} \approx 1.3, \quad \text{gap} \geq 1.$$

На проходе по массиву сравниваются пары  $(a[i], a[i + \text{gap}])$  для всех  $i$ , где это возможно, и при необходимости меняются местами.

Получается последовательность «редких» обменов с большим шагом. По мере уменьшения `gap` массив становится всё более упорядоченным, и финальная стадия при `gap = 1` — фактически пузырёк на почти отсортированном массиве.

**Пример на массиве**  $[2, 4, 5, 6, 3, 1]$

Пусть `shrink = 1.3`.

$$\text{gap}_0 = 6, \quad \text{gap}_1 = \lfloor 6/1.3 \rfloor = 4.$$

Шаг с  $\text{gap} = 4$ :

$$(2, 3) \Rightarrow [2, 4, 5, 6, 3, 1], \quad (4, 1) \Rightarrow [2, 1, 5, 6, 3, 4].$$

Далее:

$$\text{gap}_2 = \lfloor 4/1.3 \rfloor = 3.$$

При  $\text{gap} = 3$ :

$$[2, 1, 5, 6, 3, 4] \rightarrow [2, 1, 4, 6, 3, 5].$$

$$\text{gap}_3 = \lfloor 3/1.3 \rfloor = 2.$$

При  $\text{gap} = 2$ :

Исходный массив:  $[2, 1, 4, 6, 3, 5]$

Сравниваем элементы на расстоянии 2:  $(2, 4)$ ,  $(1, 6)$ ,  $(4, 3)$ ,  $(6, 5)$

Обменяем где нужно:

- $(2, 4)$  - порядок верный
- $(1, 6)$  - порядок верный
- $(4, 3)$  - меняем местами  $\rightarrow [2, 1, 3, 6, 4, 5]$
- $(6, 5)$  - меняем местами  $\rightarrow [2, 1, 3, 5, 4, 6]$

$$\text{gap}_4 = \lfloor 2/1.3 \rfloor = 1.$$

Дальнейшие проходы при  $\text{gap}=1$  (по сути пузырьрёк) доводят массив до  $[1, 2, 3, 4, 5, 6]$ .

## Асимптотика и свойства

Теоретически худший случай:

$$T_{\text{worst}}(n) = O(n^2).$$

На практике для случайных данных наблюдается поведение, близкое к  $O(n \log n)$ , при малых константах.

Свойства:

- Память:  $O(1)$ .
- Стабильность: нет (перестановки на расстоянии могут менять относительный порядок равных элементов).

## Реализация



```

1 def comb_sort(a):
2     n = len(a)
3     gap = n
4     sh = 1.3
5     f = 1
6     while gap > 1 or f == 1:
7         gap = int(gap / sh)
8         if gap < 1:
9             gap = 1
10        f = 0
11        for i in range(n - gap):
12            j = i + gap
13            if a[i] > a[j]:
14                a[i], a[j] = a[j], a[i]
15                f = 1

```

## 9.4 Сортировка вставками (Insertion sort)

### Инвариант и описание

Сортировка вставками строит отсортированный префикс, расширяя его на один элемент за шаг.

Инвариант после шага  $k$ :

$a[0..k-1]$  отсортирован в неубывающем порядке.

Элемент  $a[k]$  рассматривается как ключ и вставляется в отсортированную часть  $a[0..k-1]$  так, чтобы сохранить порядок. Реализация обычно выполняет сдвиг последовательности элементов вправо до нахождения позиции для ключа.

### Пошаговая работа сортировки вставками

Исходный массив:

$[7, 3, 9, 1, 6, 2, 8, 5, 4, 0]$

**Шаг 1:**  $[7] \mid [3, 9, 1, 6, 2, 8, 5, 4, 0]$

↓

Берём  $3 \rightarrow 3 < 7 \rightarrow$  Сдвигаем

↓

$[3, 7] \mid [9, 1, 6, 2, 8, 5, 4, 0]$

**Шаг 2:**  $[3, 7] \mid [9, 1, 6, 2, 8, 5, 4, 0]$

↓

Берём  $9 \rightarrow 9 > 7 \rightarrow$  Не сдвигаем

↓

$[3, 7, 9] \mid [1, 6, 2, 8, 5, 4, 0]$

**Шаг 3:**  $[3, 7, 9] \mid [1, 6, 2, 8, 5, 4, 0]$

↓

Берём  $1 \rightarrow$  Сдвигаем  $9, 7, 3$

↓

$[1, 3, 7, 9] \mid [6, 2, 8, 5, 4, 0]$

**Шаг 4:**  $[1, 3, 7, 9] \mid [6, 2, 8, 5, 4, 0]$

↓

Берём  $6 \rightarrow$  Сдвигаем  $9, 7$

↓

$[1, 3, 6, 7, 9] \mid [2, 8, 5, 4, 0]$

**Шаг 5:**  $[1, 3, 6, 7, 9] \mid [2, 8, 5, 4, 0]$

↓

Берём  $2 \rightarrow$  Сдвигаем  $9, 8, 7, 6, 3$

↓

$[1, 2, 3, 6, 7, 8, 9] \mid [5, 4, 0]$

**Шаг 6:**  $[1, 2, 3, 6, 7, 8, 9] \mid [5, 4, 0]$

↓

Берём 5 → Сдвигаем 9, 8, 7, 6

↓

$[1, 2, 3, 5, 6, 7, 8, 9] \mid [4, 0]$

**Шаг 7:**  $[1, 2, 3, 5, 6, 7, 8, 9] \mid [4, 0]$

↓

Берём 4 → Сдвигаем 9, 8, 7, 6, 5

↓

$[1, 2, 3, 4, 5, 6, 7, 8, 9] \mid [0]$

**Шаг 8:**  $[1, 2, 3, 4, 5, 6, 7, 8, 9] \mid [0]$

↓

Берём 0 → Сдвигаем все элементы

↓

$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$

**Результат:**

$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$

Массив отсортирован с помощью сортировки вставками.

## Инверсии и сложность

Число сдвигов на шаге  $k$  равно числу элементов в префиксе  $a[0..k-1]$ , которые больше  $a[k]$ , то есть количеству новых инверсий, в которые входит позиция  $k$ .

Суммарно число сдвигов по всему алгоритму равно количеству инверсий  $I(a)$ :

$$T_{\text{insertion}}(a) = O(n + I(a)).$$

Отсюда:

- уже отсортированный массив:  $I(a) = 0 \Rightarrow T = O(n)$ ;
- обратно отсортированный массив:  $I(a) = O(n^2) \Rightarrow T = O(n^2)$ ;
- случайный массив: в среднем  $I(a) \approx n(n-1)/4 \Rightarrow T = O(n^2)$ .

## Свойства

- Стабильность: да (равные элементы не перескакивают через друг друга).

- Память:  $O(1)$ .
- Эффективен на почти отсортированных данных и для очень маленьких  $n$  (часто используется внутри других алгоритмов как «хвостовая» сортировка на небольших подмассивах).

## Реализация

```

1 def insertion_sort(a):
2     n = len(a)
3     for i in range(1, n):
4         x = a[i]
5         j = i - 1
6         while j >= 0 and a[j] > x:
7             a[j + 1] = a[j]
8             j -= 1
9         a[j + 1] = x

```

## 9.5 Сортировка Шелла (Shellsort)

### Мотивация

Shellsort — попытка ускорить сортировку вставками за счёт работы с элементами, удалёнными друг от друга. Если сделать вставки не по шагу 1, а по шагу  $h$ , то за одну операцию можно исправлять «грубые» нарушения порядка на больших расстояниях.

Идея: применять сортировку вставками не один раз, а для последовательности шагов  $h_1 > h_2 > \dots > 1$ .

### Схема

Пусть задана последовательность шагов  $h_k$ .

Для каждого  $h$ :

- массив разбивается на  $h$  подпоследовательностей:

$$(a[0], a[h], a[2h], \dots), (a[1], a[1+h], \dots), \dots, (a[h-1], a[2h-1], \dots);$$

- каждая подпоследовательность сортируется вставками.

Последняя фаза  $h = 1$  — обычные вставки на уже «почти» отсортированном массиве.

**Пример на массиве**  $[2, 4, 5, 6, 3, 1]$

Возьмём шаги:  $h = 3$ , затем  $h = 1$ .

**Шаг**  $h = 3$ . Подпоследовательности:

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

После сортировки вставками внутри:

$$(2, 6) \rightarrow (2, 6), \quad (4, 3) \rightarrow (3, 4), \quad (5, 1) \rightarrow (1, 5).$$

Возвращаем по индексам 0, 3, 1, 4, 2, 5:

$$[2, 3, 1, 6, 4, 5].$$

**Шаг**  $h = 1$ . Обычные вставки на  $[2, 3, 1, 6, 4, 5]$  приводят к  $[1, 2, 3, 4, 5, 6]$ .

### Сложность и свойства

Асимптотика зависит от выбранной последовательности  $h_k$ :

- классическая последовательность Шелла  $h_k = \lfloor n/2^k \rfloor$  даёт  $O(n^2)$  в худшем случае;
- существуют последовательности (Хиббард, Седжвик, Пратт), для которых худшая оценка существенно улучшена (например,  $O(n^{4/3})$ ,  $O(n(\log n)^2)$  и т.п.).

На практике при разумном выборе шагов Shellsort часто работает достаточно быстро.

### Свойства

- Память:  $O(1)$ .
- Стабильность: обычно нет (так как элементы могут перепрыгивать далеко).
- Подходит как «промежуточный» алгоритм: лучше квадратичных, проще и менее требователен к памяти, чем некоторые  $O(n \log n)$ -алгоритмы.

### Реализация (классический вариант)

```
1 def shell_sort(a):
2     n = len(a)
3     h = n // 2
4     while h > 0:
5         for i in range(h, n):
6             x = a[i]
7             j = i
8             while j >= h and a[j - h] > x:
9                 a[j] = a[j - h]
10                j -= h
```

```

11         a[j] = x
12     h //= 2

```

## 9.6 Сортировка выбором (Selection sort)

### Инвариант и схема

На шаге  $i$  сортировки выбором выполняется поиск минимального элемента на суффиксе  $a[i..n-1]$ , после чего он обменивается с  $a[i]$ .

Инвариант: после шага  $i$  префикс  $a[0..i]$  состоит из  $i+1$  минимальных элементов входного массива в отсортированном порядке.

**Пример на массиве**  $[2, 4, 5, 6, 3, 1]$

Шаг  $i = 0$ : минимум на  $[2, 4, 5, 6, 3, 1] — 1$ ; обмен с  $a[0]$ :

$[1, 4, 5, 6, 3, 2]$ .

Шаг  $i = 1$ : минимум на  $[4, 5, 6, 3, 2] — 2$ ; обмен с  $a[1]$ :

$[1, 2, 5, 6, 3, 4]$ .

Шаг  $i = 2$ : минимум на  $[5, 6, 3, 4] — 3$ ; обмен:

$[1, 2, 3, 6, 5, 4]$ .

Шаг  $i = 3$ : минимум на  $[6, 5, 4] — 4$ ; обмен:

$[1, 2, 3, 4, 5, 6]$ .

Дальнейшие шаги ничего не меняют.

### Сложность и свойства

$$T_{\text{best}}(n) = T_{\text{avg}}(n) = T_{\text{worst}}(n) = O(n^2).$$

- Число обменов  $O(n)$  (что мало по сравнению с пузырьком).
- Память:  $O(1)$ .
- Алгоритм нестабилен.

### Реализация

```

1 def selection_sort(a):
2     n = len(a)

```

```

3     for i in range(n):
4         m = i
5         for j in range(i + 1, n):
6             if a[j] < a[m]:
7                 m = j
8         if m != i:
9             a[i], a[m] = a[m], a[i]

```

## 9.7 Пирамидальная сортировка (Heapsort)

### Идея и структура данных

Heapsort использует двоичную max-heap (кучу): массив, удовлетворяющий свойству

$$a[i] \geq a[2i + 1], \quad a[i] \geq a[2i + 2]$$

для всех допустимых  $i$ . Максимальный элемент всегда находится в корне (индекс 0).

Алгоритм:

1. Построить кучу из массива за  $O(n)$ .
2. Повторять для  $i$  от  $n - 1$  до 1:
  - обменять  $a[0]$  и  $a[i]$ ;
  - уменьшить размер кучи до  $i$ ;
  - восстановить свойство кучи вниз от корня (heapify) за  $O(\log i)$ .

**Пример на массиве**  $[2, 4, 5, 6, 3, 1]$

После построения max-heap (одно из возможных представлений):

$$[6, 4, 5, 2, 3, 1].$$

Далее:

$$[6, 4, 5, 2, 3, 1] \rightarrow [1, 4, 5, 2, 3, 6] \rightarrow \text{heapify} \rightarrow [5, 4, 1, 2, 3, 6],$$

$$[5, 4, 1, 2, 3, 6] \rightarrow [3, 4, 1, 2, 5, 6] \rightarrow \text{heapify} \rightarrow [4, 3, 1, 2, 5, 6],$$

$$[4, 3, 1, 2, 5, 6] \rightarrow [2, 3, 1, 4, 5, 6] \rightarrow \text{heapify} \rightarrow [3, 2, 1, 4, 5, 6],$$

$$[3, 2, 1, 4, 5, 6] \rightarrow [1, 2, 3, 4, 5, 6].$$

### Сложность и свойства

$$T_{\text{best}}(n) = T_{\text{avg}}(n) = T_{\text{worst}}(n) = \Theta(n \log n).$$

- Память:  $O(1)$ .
- Нестабилен.

## Реализация

```

1 def _heapify(a, n, i):
2     while True:
3         l = 2 * i + 1
4         r = 2 * i + 2
5         m = i
6         if l < n and a[l] > a[m]:
7             m = l
8         if r < n and a[r] > a[m]:
9             m = r
10        if m == i:
11            break
12        a[i], a[m] = a[m], a[i]
13        i = m
14
15 def heap_sort(a):
16     n = len(a)
17     for i in range(n // 2 - 1, -1, -1):
18         _heapify(a, n, i)
19     for i in range(n - 1, 0, -1):
20         a[0], a[i] = a[i], a[0]
21         _heapify(a, i, 0)

```

## 9.8 Быстрая сортировка (Quicksort)

### Схема

Quicksort — рекурсивный алгоритм разделяй-и-властвуй:

1. Выбрать опорный элемент (pivot).
2. Разделить массив на части: элементы  $\leq \text{pivot}$  и элементы  $> \text{pivot}$ .
3. Рекурсивно отсортировать каждую часть.

Основная операция — *partition*, выполняющая разделение за  $O(n)$ .

Удачный выбор pivot'а (например, случайный или медиана из нескольких) обеспечивает хорошее поведение на практике.

**Пример на массиве** [2, 4, 5, 6, 3, 1]

Пусть в качестве pivot берётся последний элемент, 1.



Partition:

$$[2, 4, 5, 6, 3, 1] \rightarrow [1, 4, 5, 6, 3, 2]$$

(один из возможных вариантов; важен сам принцип).

Pivot оказывается в позиции 0, левый подмассив пуст, правый:  $[4, 5, 6, 3, 2]$  (или другое разбиение, в зависимости от реализации). Далее рекурсивные разбиения постепенно приводят к полностью отсортированному  $[1, 2, 3, 4, 5, 6]$ .

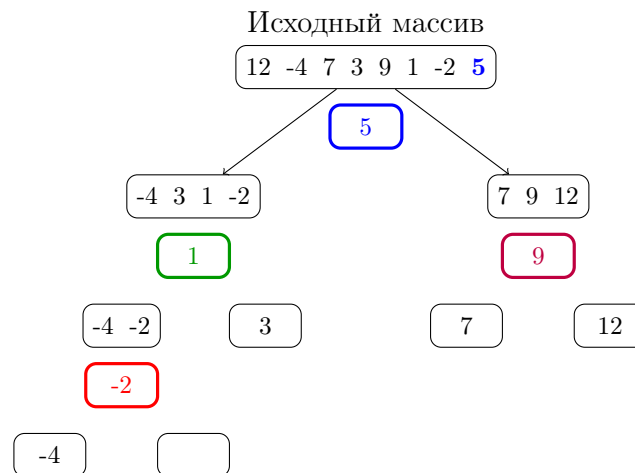


Рис. 1: Дерево разбиения QuickSort с корректным расположением -2 и стрелками

### Сложность и свойства

$$T_{\text{avg}}(n) = \Theta(n \log n), \quad T_{\text{best}}(n) = \Theta(n \log n), \quad T_{\text{worst}}(n) = \Theta(n^2).$$

Свойства:

- Память:  $O(\log n)$  в среднем (глубина рекурсии).
- Нестабилен.
- Одна из самых быстрых сортировок на практике (малая константа, хорошее локальное кеширование).

### Реализация (in-place Lomuto-like)

```

1 def _partition(a, l, r):
2     x = a[r]
3     i = l
4     for j in range(l, r):
5         if a[j] <= x:
6             a[i], a[j] = a[j], a[i]
7             i += 1
8     a[i], a[r] = a[r], a[i]
  
```

```

9     return i
10
11 def _quick_sort(a, l, r):
12     if l >= r:
13         return
14     p = _partition(a, l, r)
15     _quick_sort(a, l, p - 1)
16     _quick_sort(a, p + 1, r)
17
18 def quick_sort(a):
19     if a:
20         _quick_sort(a, 0, len(a) - 1)

```

## 9.9 Сортировка слиянием (Merge sort)

### Схема и инвариант

Классическая рекурсивная схема:

1. Если длина массива  $\leq 1$ , он уже отсортирован.
2. Разделить на две части  $L$  и  $R$ .
3. Рекурсивно отсортировать  $L$  и  $R$ .
4. Слить два отсортированных массива в один отсортированный за  $O(n)$ .

Инвариант: на каждом шаге рекурсии обе половины отсортированы, задача — корректно и стабильно их слить.

### Пример сортировки слиянием на массиве

Исходный массив:  $[2, 4, 5, 6, 3, 1]$

#### Шаг 1: Разделение

Делим массив на две примерно равные части:

- Левая половина:  $[2, 4, 5]$
- Правая половина:  $[6, 3, 1]$

#### Шаг 2: Рекурсивная сортировка левой половины $[2, 4, 5]$

- Делим:  $[2]$  и  $[4, 5]$
- $[2]$  — уже отсортирован (базовый случай)
- Сортируем  $[4, 5]$ :
  - Делим:  $[4]$  и  $[5]$
  - Оба массива отсортированы

- Сливаем: [4, 5]
  - Сливаем [2] и [4, 5]: [2, 4, 5]
- Результат левой половины: [2, 4, 5]

### Шаг 3: Рекурсивная сортировка правой половины [6, 3, 1]

- Делим: [6] и [3, 1]
  - [6] — уже отсортирован
  - Сортируем [3, 1]:
    - Делим: [3] и [1]
    - Оба массива отсортированы
    - Сливаем: [1, 3]
  - Сливаем [6] и [1, 3]: [1, 3, 6]
- Результат правой половины: [1, 3, 6]

### Шаг 4: Слияние отсортированных половин

Сливаем [2, 4, 5] и [1, 3, 6]:

- Сравниваем первые элементы:  $2 > 1 \rightarrow$  берём 1  
Результат: [1], осталось: [2, 4, 5] и [3, 6]
- Сравниваем:  $2 < 3 \rightarrow$  берём 2  
Результат: [1, 2], осталось: [4, 5] и [3, 6]
- Сравниваем:  $4 > 3 \rightarrow$  берём 3  
Результат: [1, 2, 3], осталось: [4, 5] и [6]
- Сравниваем:  $4 < 6 \rightarrow$  берём 4  
Результат: [1, 2, 3, 4], осталось: [5] и [6]
- Сравниваем:  $5 < 6 \rightarrow$  берём 5  
Результат: [1, 2, 3, 4, 5], осталось: [] и [6]
- Добавляем оставшиеся элементы: [6]  
Результат: [1, 2, 3, 4, 5, 6]

### Итоговый отсортированный массив:

[1, 2, 3, 4, 5, 6]

### Асимптотика и свойства

Рекуррентное соотношение:

$$T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = \Theta(n \log n)$$

в любом случае (худший/средний/лучший).

Свойства:

- Память:  $O(n)$  (надо хранить временный массив для слияния).
- Стабильность: да (если при равенстве брать элементы из левой части раньше правой).
- Предсказуемое время в худшем случае, хорошо ложится на внешнюю сортировку (файлы).

**Реализация (функция возвращает новый список)**

```
1 def merge_sort(a):
2     n = len(a)
3     if n <= 1:
4         return a
5     m = n // 2
6     left = merge_sort(a[:m])
7     right = merge_sort(a[m:])
8     i = j = 0
9     res = []
10    while i < len(left) and j < len(right):
11        if left[i] <= right[j]:
12            res.append(left[i])
13            i += 1
14        else:
15            res.append(right[j])
16            j += 1
17    res.extend(left[i:])
18    res.extend(right[j:])
19    return res
```

## 9.10 Сортировка подсчётом (Counting sort)

### Модель

Сортировка подсчётом предполагает, что элементы — целые числа в диапазоне  $[L, R]$  с небольшим размахом  $K = R - L + 1$ .

### Схема

1. Создать массив счётчиков  $c[0..K - 1]$ , обнулить.
2. Для каждого элемента  $x$  увеличить  $c[x - L]$ .
3. Восстановить исходный массив, проходя по  $c$  и выписывая каждое значение  $L + i$  ровно  $c[i]$  раз.

Для стабильной версии:

- сначала построить префиксные суммы  $p[i] = \sum_{j=0}^i c[j]$ ;
- затем пройти по исходному массиву справа налево, размещая элементы на позициях  $p[x - L] - 1$  и уменьшая  $p[x - L]$ .

**Пример на массиве  $[2, 4, 5, 6, 3, 1]$  при  $L = 1, R = 6$**

Массив счётчиков  $c[0..5]$  соответствует значениям 1..6.

Подсчёт:

$$a = [2, 4, 5, 6, 3, 1] \Rightarrow c = [1, 1, 1, 1, 1, 1].$$

Восстановление:

$$[1, 2, 3, 4, 5, 6].$$

### Сложность и свойства

$$T(n, K) = \Theta(n + K), \quad \text{память} = \Theta(K).$$

Свойства:

- Стабильность: только стабилизированная версия.
- Не опирается на сравнения, вырывается из нижней границы  $\Omega(n \log n)$ .

### Реализация (простая)

```
1 def counting_sort(a, L, R):
2     K = R - L + 1
3     c = [0] * K
4     for x in a:
5         c[x - L] += 1
6     i = 0
7     for v in range(K):
8         cnt = c[v]
9         x = L + v
10        for _ in range(cnt):
11            a[i] = x
12            i += 1
```

## 9.11 Поразрядные сортировки (Radix sort)

Поразрядные сортировки используют представление чисел в позиционной системе счисления и сортировку по разрядам, а не по целым значениям. Они опираются на

стабильные сортировки по ключу «разряд».

Рассмотрим двоичную систему (биты) для простоты.

### LSD radix sort через очереди 0–1

LSD (least significant digit first) обрабатывает разряды от младшего к старшему.

Пусть максимальный элемент имеет битовую длину  $w$ . Для каждого бита  $k = 0, 1, \dots, w - 1$ :

1. Создаём две очереди  $Q_0$  и  $Q_1$ .
2. Проходим по массиву слева направо. Для каждого  $x$ :

$$b = \left( \frac{x}{2^k} \right) \bmod 2 = (x \gg k) \& 1,$$

если  $b = 0$ , помещаем  $x$  в  $Q_0$ , иначе в  $Q_1$ .

3. Затем последовательно извлекаем все элементы из  $Q_0$ , затем из  $Q_1$ , переписывая их в массив.

За счёт стабильности на каждом шаге уже учтённый порядок по младшим разрядам сохраняется, и по завершении обработки всех  $w$  битов массив отсортирован.

### Пример на массиве $[2, 4, 5, 6, 3, 1]$

Бинарные представления:

$$2 = (010)_2, \quad 4 = (100)_2, \quad 5 = (101)_2, \quad 6 = (110)_2, \quad 3 = (011)_2, \quad 1 = (001)_2.$$

**Бит  $k = 0$**  (младший):

$$0 : \{2, 4, 6\}, \quad 1 : \{5, 3, 1\} \Rightarrow [2, 4, 6, 5, 3, 1].$$

**Бит  $k = 1$ :**

$$2 \rightarrow 1, \quad 4 \rightarrow 0, \quad 6 \rightarrow 1, \quad 5 \rightarrow 0, \quad 3 \rightarrow 1, \quad 1 \rightarrow 0.$$

Очереди:

$$0 : \{4, 5, 1\}, \quad 1 : \{2, 6, 3\} \Rightarrow [4, 5, 1, 2, 6, 3].$$

**Бит  $k = 2$ :**

$$4 \rightarrow 1, \quad 5 \rightarrow 1, \quad 1 \rightarrow 0, \quad 2 \rightarrow 0, \quad 6 \rightarrow 1, \quad 3 \rightarrow 0.$$

Очереди:

$$0 : \{1, 2, 3\}, \quad 1 : \{4, 5, 6\} \Rightarrow [1, 2, 3, 4, 5, 6].$$

### Сложность

Каждый проход —  $O(n)$ , всего  $w$  проходов. Если  $w$  — фиксировано (например, 32/64 бита), то:

$$T(n) = O(n), \quad \text{память} = \Theta(n).$$

### Реализация (LSD по битам, очереди 0/1)

```

1 from collections import deque
2
3 def radix_sort_lsd_binary(a):
4     if not a:
5         return
6     max_x = max(a)
7     w = max_x.bit_length()
8     for k in range(w):
9         q0 = deque()
10        q1 = deque()
11        for x in a:
12            if (x >> k) & 1:
13                q1.append(x)
14            else:
15                q0.append(x)
16        i = 0
17        while q0:
18            a[i] = q0.popleft()
19            i += 1
20        while q1:
21            a[i] = q1.popleft()
22            i += 1

```

### MSD radix sort

MSD (most significant digit first) сортирует по старшему разряду, затем рекурсивно применяет себя к подмассивам, соответствующим каждому значению старшего разряда.

В двоичном случае:

1. По биту  $k$  делим массив на два блока: с 0 и с 1 на этом бите.
2. Рекурсивно сортируем каждый блок по биту  $k - 1$ , и так далее.

По сути это частный случай блочной сортировки, где блоки — значения разрядов.

### Реализация (MSD по битам)

```

1 def _radix_sort_msd_binary(a, l, r, k):
2     if l >= r or k < 0:
3         return
4     i = l
5     j = r
6     while i <= j:
7         while i <= j and ((a[i] >> k) & 1) == 0:
8             i += 1
9         while i <= j and ((a[j] >> k) & 1) == 1:
10            j -= 1
11        if i < j:
12            a[i], a[j] = a[j], a[i]
13            i += 1
14            j -= 1
15        _radix_sort_msd_binary(a, l, j, k - 1)
16        _radix_sort_msd_binary(a, i, r, k - 1)
17
18 def radix_sort_msd_binary(a):
19     if not a:
20         return
21     max_x = max(a)
22     k = max_x.bit_length() - 1
23     _radix_sort_msd_binary(a, 0, len(a) - 1, k)

```

## 9.12 Сортировка деревом (Tree sort)

### Идея

Используем двоичное дерево поиска (BST):

1. Вставить все элементы массива в BST.
2. Выполнить симметричный обход (in-order): левое поддерево, корень, правое поддерево.

Симметричный обход BST возвращает элементы в отсортированном порядке.



### Пример на массиве [2, 4, 5, 6, 3, 1]

Вставляем элементы в порядке:

$$2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 1.$$

Получаем одно из возможных деревьев:

2 — корень, 1 — левый сын 2, 4 — правый сын 2,  
3 — левый сын 4, 5 — правый сын 4, 6 — правый сын 5.

Симметричный обход (лево–корень–право) даёт:

1, 2, 3, 4, 5, 6.

### Сложность

- При случайных данных и удачной балансировке дерева:  $O(n \log n)$ .
- В худшем случае (вставка уже отсортированного массива без балансировки):  $O(n^2)$ .

Если использовать самобалансирующиеся деревья (AVL, красно-чёрные), то худший случай  $O(n \log n)$  гарантированно.

### Реализация (наивный BST)

```
1 class Node:
2     __slots__ = ("x", "l", "r")
3     def __init__(self, x):
4         self.x = x
5         self.l = None
6         self.r = None
7
8 def _insert(root, x):
9     if root is None:
10         return Node(x)
11     if x <= root.x:
12         root.l = _insert(root.l, x)
13     else:
14         root.r = _insert(root.r, x)
15     return root
16
17 def _inorder(root, res):
```

```

18     if root is None:
19         return
20     _inorder(root.l, res)
21     res.append(root.x)
22     _inorder(root.r, res)
23
24 def tree_sort(a):
25     root = None
26     for x in a:
27         root = _insert(root, x)
28     res = []
29     _inorder(root, res)
30     for i in range(len(a)):
31         a[i] = res[i]

```

## 9.13 Timsort

### Идея

Timsort — гибридная сортировка, комбинирующая сортировку вставками и модифицированное слияние. Это стандартная сортировка в Python (`list.sort()`, `sorted`) и Java (частично).

Ключевые идеи:

- Выявление монотонных участков (*runs*) — уже отсортированных или строго убывающих, которые переворачиваются.
- Каждому run'у обеспечивается минимальная длина (через досортировку вставками).
- Run'ы сливаются в соответствии с определёнными инвариантами по длинам (чтобы избежать глубоких несбалансированных цепочек слияний и квадратичного времени).

Timsort очень эффективно пользуется уже имеющейся структурой данных (частичная отсортированность, длинные «плато», блоки равных элементов и т.п.).

**Пример на массиве** [2, 4, 5, 6, 3, 1]

Алгоритм найдёт возрастающий run:

[2, 4, 5, 6],

затем обнаружит убывающий кусок  $[3, 1]$ , развернёт его в

$$[1, 3],$$

получая два гup'a:

$$[2, 4, 5, 6], [1, 3].$$

Далее короткий гup доводится до минимальной длины (вставками, если нужно), и гup'ы сливаются (как в merge sort), что даёт:

$$[1, 2, 3, 4, 5, 6].$$

## Сложность и свойства

$$T_{\text{best}}(n) = \Theta(n), \quad T_{\text{avg}}(n) = T_{\text{worst}}(n) = \Theta(n \log n).$$

Свойства:

- Стабильность: да.
- Отличное поведение на «реальных» данных, и в худшем случае — не хуже сортировки слиянием.

Полная реализация достаточно сложна для включения в конспект; в Python она встроена на уровне интерпретатора.

## 10 Инверсии и связь с сортировками

### 10.1 Определение и интерпретация

Пусть дан массив  $a[0..n-1]$ . *Инверсией* называется пара индексов  $(i, j)$ ,  $0 \leq i < j < n$ , такая что

$$a[i] > a[j].$$

Количество инверсий  $I(a)$  измеряет «степень неотсортированности» массива:

- $I(a) = 0 \Leftrightarrow$  массив отсортирован неубывающим образом;
- максимум  $I(a) = \frac{n(n-1)}{2}$  достигается на строго убывающем массиве.

### 10.2 Инверсии на массиве $[2, 4, 5, 6, 3, 1]$

Перечислим все пары  $(i, j)$ ,  $i < j$ , такие что  $a[i] > a[j]$ :

$$[2, 4, 5, 6, 3, 1].$$

Инверсии:

$$(4, 3), (5, 3), (6, 3), \\ (4, 1), (5, 1), (6, 1), (3, 1), (2, 1).$$

Всего  $I(a) = 8$ .

### 10.3 Связь с сортировкой вставками

Как уже отмечалось, сортировка вставками на каждом шаге  $k$  сдвигает  $a[k]$  влево до первого элемента, меньшего или равного  $a[k]$ .

Каждый шаг «убирает» ровно столько инверсий, сколько было элементов  $a[i]$  с  $i < k$  и  $a[i] > a[k]$ . Сдвиг элемента вправо соответствует уничтожению одной инверсии, поэтому:

$$\text{число сдвигов} = I(a),$$

и

$$T_{\text{insertion}}(a) = \Theta(n + I(a)).$$

Для массива  $[2, 4, 5, 6, 3, 1]$  сортировка вставками сделает ровно 8 сдвигов.

### 10.4 Подсчёт инверсий через сортировку слиянием

Модифицируем сортировку слиянием так, чтобы параллельно с сортировкой считать инверсии.

При слиянии двух отсортированных массивов  $L$  и  $R$ :

- индексы  $i$  и  $j$  указывают текущие позиции в  $L$  и  $R$ ;
- если  $L[i] \leq R[j]$ , то берём  $L[i]$  без новых инверсий;
- если  $L[i] > R[j]$ , то все элементы  $L[i], L[i + 1], \dots$  образуют инверсии с  $R[j]$ , поэтому к ответу добавляется  $|L| - i$ .

#### Реализация подсчёта инверсий

```

1 def _merge_count(L, R):
2     i = j = 0
3     res = []
4     inv = 0
5     nL = len(L)
6     while i < nL and j < len(R):
7         if L[i] <= R[j]:
8             res.append(L[i])
9             i += 1
10        else:
```

```

11         res.append(R[j])
12         inv += nL - i
13         j += 1
14     res.extend(L[i:])
15     res.extend(R[j:])
16     return res, inv
17
18 def _sort_count(a):
19     n = len(a)
20     if n <= 1:
21         return a, 0
22     m = n // 2
23     L, invL = _sort_count(a[:m])
24     R, invR = _sort_count(a[m:])
25     M, invM = _merge_count(L, R)
26     return M, invL + invR + invM
27
28 def inversion_count(a):
29     _, inv = _sort_count(a)
30     return inv

```