**Конспект Яндекс.Учебника по алгоритмам**

[Алгоритмы и их сложность 2](#_Toc195453041)

[Полный перебор и оптимизация перебора 3](#_Toc195453042)

[Жадные алгоритмы 5](#_Toc195453043)

[Рекурсивные алгоритмы 7](#_Toc195453044)

[Метод «Разделяй и властвуй» 9](#_Toc195453045)

[Рандомизированные алгоритмы 11](#_Toc195453046)

[Двоичный поиск 13](#_Toc195453047)

[Односвязный список 15](#_Toc195453048)

[Множество 18](#_Toc195453049)

[Словари 20](#_Toc195453050)

[Стек 22](#_Toc195453051)

[Очередь с приоритетом 25](#_Toc195453052)

[Дек 28](#_Toc195453053)

# Алгоритмы и их сложность

Основные понятия:

1. **Алгоритм** – последовательность шагов для решения задачи.
2. **Сложность алгоритма** – оценка времени работы или используемой памяти в зависимости от размера входных данных (n).

Виды сложности:

* **Временная сложность** – количество операций.
* **Пространственная сложность** – объем используемой памяти.

Нотация "О-большое":

* Оценивает верхнюю границу сложности.
* Примеры:
  + O(1) – константная (постоянная).
  + O(log n) – логарифмическая.
  + O(n) – линейная.
  + O(n²) – квадратичная.
  + O(2ⁿ) – экспоненциальная.

Примеры алгоритмов и их сложность:

1. **Линейный поиск** – O(n).
2. **Бинарный поиск** – O(log n).
3. **Сортировка пузырьком** – O(n²).
4. **Быстрая сортировка (QuickSort)** – O(n log n) в среднем.

Классы сложности задач:

* **P** – задачи, решаемые за полиномиальное время (O(nᵏ)).
* **NP** – задачи, где решение можно проверить за полиномиальное время.

# Полный перебор и оптимизация перебора

**1. Полный перебор (Brute Force)**

* **Определение:** Метод решения задачи, при котором проверяются все возможные варианты.
* **Применение:** используется, когда нет более эффективного алгоритма или когда входные данные небольшие.
* **Примеры задач:**
  + Поиск делителей числа.
  + Задача коммивояжёра (для малого числа городов).
  + Подбор пароля полным перебором.
* **Сложность:** часто экспоненциальная (O(2ⁿ), факториальная (O(n!)) или полиномиальная (O(nᵏ)).

**2. Оптимизация перебора**

Чтобы ускорить перебор, применяют методы, сокращающие количество проверяемых вариантов:

**2.1. Отсечение заведомо неподходящих вариантов (Backtracking)**

* **Идея:** если текущий путь решения заведомо не ведёт к правильному ответу, его можно отбросить.
* **Пример:**
  + Задача о расстановке ферзей на шахматной доске (если очередная расстановка приводит к конфликту, дальнейшие варианты не рассматриваются).

**2.2. Жадные алгоритмы**

* **Идея:** на каждом шаге выбирается локально оптимальное решение в надежде получить глобальный оптимум.
* **Пример:**
  + Задача о выборе монет для размена (жадный алгоритм работает не всегда, но иногда ускоряет перебор).

**2.3. Метод ветвей и границ (Branch and Bound)**

* **Идея:** Разбиение задачи на подзадачи (ветви) и оценка их потенциальной оптимальности (границы). Неперспективные ветви отбрасываются.
* **Пример:**
  + Оптимизация маршрута коммивояжёра (если текущий путь уже длиннее найденного минимума, его можно не рассматривать дальше).

**2.4. Динамическое программирование**

* **Идея:** Разбиение задачи на подзадачи с запоминанием промежуточных результатов.
* **Пример:**
  + Задача о рюкзаке (перебор с мемоизацией).

**3. Когда использовать перебор?**

* Данные небольшие (n ≤ 20–30).
* Нет известного эффективного алгоритма.
* Можно применить оптимизации (отсечения, жадные стратегии, кэширование).

# Жадные алгоритмы

**1. Определение**

**Жадный алгоритм** — это алгоритм, который на каждом шаге делает **локально оптимальный выбор** в надежде получить **глобально оптимальное решение**.

**2. Основные свойства**

* **Жадный выбор:** на каждом шаге выбирается лучшее решение в данный момент.
* **Оптимальная подструктура:** Оптимальное решение задачи содержит оптимальные решения подзадач.

**3. Примеры задач**

1. **Задача о выборе заявок (интервальное расписание)**
   * **Цель:** выбрать максимальное количество непересекающихся интервалов.
   * **Жадная стратегия:** выбирать интервал с самым ранним окончанием.
2. **Размен монет**
   * **Цель:** набрать нужную сумму минимальным количеством монет.
   * **Жадная стратегия:** брать наибольшую возможную монету (работает не для всех систем номиналов).
3. **Задача о рюкзаке (непрерывная версия)**
   * **Цель:** набрать предметы с максимальной суммарной стоимостью, не превышая вес.
   * **Жадная стратегия:** брать предметы с наибольшей удельной стоимостью (стоимость/вес).

**4. Когда жадный алгоритм работает?**

* Задача обладает **свойством жадного выбора** (локальный оптимум ведёт к глобальному).
* Доказана **корректность** жадного подхода (не всегда очевидно).

**5. Когда жадный алгоритм не подходит?**

* **Пример:** Дискретная задача о рюкзаке (не всегда можно разбить предметы).
* **Пример:** Задачи, где жадный выбор портит будущие варианты (например, некоторые графовые задачи).

# Рекурсивные алгоритмы

**1. Определение**

**Рекурсия** — это метод решения задачи, при котором функция **вызывает саму себя** для решения подзадач.

**2. Основные компоненты**

* **Базовый случай** — условие выхода из рекурсии (остановка).
* **Рекурсивный случай** — вызов функции с изменёнными параметрами (шаг к базовому случаю).

**3. Примеры рекурсивных алгоритмов**

A screenshot of a computer program

AI-generated content may be incorrect.

**4. Виды рекурсии**

* **Прямая рекурсия** — функция вызывает саму себя.
* **Косвенная рекурсия** — A → B → A.
* **Хвостовая рекурсия** — рекурсивный вызов в конце функции (можно заменить циклом).

**5. Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Простота записи для рекурсивных структур (деревья, графы) | Риск переполнения стека |
| Естественность для некоторых задач (разделяй и властвуй) | Большие накладные расходы |

**6. Когда использовать рекурсию?**

* Задача, естественно, разбивается на подзадачи (факториал, обход деревьев).
* Глубина рекурсии **невелика** (иначе — итеративное решение).

**7. Опасности рекурсии**

* **Переполнение стека** (если нет базового случая или слишком много вызовов).
* **Неэффективность** (например, наивный Фибоначчи — O(2ⁿ)).

**8. Оптимизации**

* **Мемоизация** — кеширование результатов (убирает повторные вычисления).
* **Итеративный подход** — замена рекурсии циклом (например, хвостовая рекурсия).

# Метод «Разделяй и властвуй»

**1. Определение**

**"Разделяй и властвуй"** — это метод решения задач, при котором:

1. **Разделение** — задача разбивается на меньшие подзадачи.
2. **Решение** — каждая подзадача решается рекурсивно.
3. **Комбинирование** — результаты подзадач объединяются в общее решение.

**2. Основные характеристики**

* **Рекурсивная структура** — естественно реализуется через рекурсию.
* **Базовый случай** — условие для остановки рекурсии (обычно тривиальная задача).
* **Эффективность** — часто даёт асимптотически оптимальные решения (например, O(n log n)).

**3. Классические примеры**

**1. Сортировка слиянием (Merge Sort)**

* **Разделение:** Массив делится пополам.
* **Решение:** Каждая половина сортируется рекурсивно.
* **Комбинирование:** Две отсортированные половины сливаются в один массив.
* **Сложность:** O(n log n).

**2. Быстрая сортировка (Quick Sort)**

* **Разделение:** Выбор опорного элемента и разбиение массива на две части.
* **Решение:** Рекурсивная сортировка левой и правой частей.
* **Комбинирование:** уже отсортированные части объединяются автоматически.
* **Сложность:** В среднем O(n log n), в худшем O(n²).

**3. Умножение матриц (алгоритм Штрассена)**

* **Разделение:** Каждая матрица разбивается на 4 подматрицы.
* **Решение:** Рекурсивное умножение подматриц с особым комбинированием.
* **Сложность:** O(n^log₂7) ≈ O(n²·⁸¹).

**4. Когда применять?**

* Задача **легко делится** на независимые подзадачи.
* Комбинирование решений **проще**, чем прямое решение.
* Подзадачи **того же типа**, что исходная задача.

# Рандомизированные алгоритмы

**1. Определение**

**Рандомизированные алгоритмы** — алгоритмы, которые используют **случайность** для достижения результата. Могут давать:

* **Всегда правильный ответ**, но со случайным временем работы (Лас-Вегас).
* **Приближённый/вероятностный ответ**, но с гарантией точности (Монте-Карло).

**2. Основные типы**

|  |  |  |
| --- | --- | --- |
| **Тип** | **Характеристика** | **Пример** |
| **Лас-Вегас** | Всегда корректен, время работы случайно | Быстрая сортировка с рандомизированным выбором опоры |
| **Монте-Карло** | Быстро работает, но может ошибаться | Тест Миллера-Рабина на простоту числа |

**3. Классические примеры**

**1. Быстрая сортировка (QuickSort) с рандомизацией**

* **Проблема:** В худшем случае работает за O(n²) (например, если опора выбирается неудачно).
* **Решение:** выбирать опорный элемент **случайно**.
* **Результат:** Среднее время работы — O(n log n) с высокой вероятностью.

**2. Тест Миллера-Рабина на простоту числа**

* **Проблема:** Проверка числа на простоту за полиномиальное время.
* **Решение:** Вероятностный тест (может ошибиться с крайне малой вероятностью).
* **Точность:** если число составное, алгоритм обнаруживает это с вероятностью ≥ 75% за один тест.

**3. Хеширование (Bloom filter)**

* **Проблема:** Эффективная проверка принадлежности элемента к множеству.
* **Решение:** Вероятностная структура данных (может давать ложноположительные ответы).
* **Применение:** Кэширование, сетевые фильтры.

**4. Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Часто проще и быстрее детерминированных аналогов | Нет 100% гарантии правильности (для Монте-Карло) |
| Позволяют решать задачи, для которых нет эффективных детерминированных решений | Требуют генератора хороших случайных чисел |

**5. Когда применять?**

* Нужно **ускорить** детерминированный алгоритм (QuickSort).
* Детерминированное решение **слишком сложно** (тесты на простоту).
* Допустима **малая вероятность ошибки** (Bloom filter).

# Двоичный поиск

**1. Определение**

**Бинарный поиск** — алгоритм поиска элемента в **отсортированном массиве**, который работает за время **O(log n)**.

**2. Принцип работы**

1. **Находим середину** массива.
2. **Сравниваем** искомый элемент с элементом в середине:
   * Если равны → элемент найден.
   * Если искомый меньше → ищем в **левой половине**.
   * Если искомый больше → ищем в **правой половине**.
3. **Повторяем**, пока не найдём элемент или интервал не станет пустым.

**3. Условия применения**

* Массив **должен быть отсортирован**.
* Работает только для структур со случайным **доступом** (например, массив).

**4. Сложность**

* **Время:** O(log n) — так как на каждом шаге область поиска уменьшается вдвое.
* **Память:** O(1) — не использует дополнительную память (итеративная реализация).

**5. Вариации бинарного поиска**

* **Поиск первого/последнего вхождения** (если есть дубликаты).
* **Поиск в монотонных функциях** (например, поиск корня уравнения).
* **Поиск в бесконечном массиве** (расширяем границы экспоненциально).

**6. Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Очень быстрый (O(log n)) | Требует предварительной сортировки |
| Простота реализации | Не работает на несортированных данных |

**7. Применение**

* Поиск в базах данных.
* Поиск в больших отсортированных файлах.
* Оптимизационные задачи (например, поиск минимального значения функции).

# Односвязный список

1. Определение

Односвязный список - это структура данных, состоящая из узлов, где каждый узел содержит:

* Данные (значение)
* Указатель (ссылку) на следующий узел

Последний элемент списка указывает на NULL.

2. Основные характеристики

* Динамическая структура (размер может изменяться во время выполнения)
* Последовательный доступ к элементам (в отличие от массива с произвольным доступом)
* Эффективные вставка и удаление (O(1) для операций в начале списка)

3. Операции и их сложность

|  |  |  |
| --- | --- | --- |
| **Операция** | **Сложность** | **Примечание** |
| Вставка в начало | O(1) |  |
| Удаление из начала | O(1) |  |
| Вставка в конец | O(n) | Требуется обход всего списка |
| Удаление из конца | O(n) |  |
| Поиск элемента | O(n) |  |
| Доступ по индексу | O(n) |  |

4. Реализация на Python (базовый пример)

A screen shot of a computer program

AI-generated content may be incorrect.

5. Преимущества и недостатки

**Преимущества:**

* Гибкость размеров
* Эффективные операции вставки/удаления в начале
* Не требует непрерывной области памяти

**Недостатки:**

* Нет произвольного доступа к элементам
* Требует дополнительной памяти для хранения указателей
* Медленный поиск элементов

6. Применение

* Когда количество элементов заранее неизвестно
* Когда важны частые вставки/удаления в начале
* В реализации стеков, очередей, хеш-таблиц
* В задачах, где важна гибкость структуры данных

# Множество

**1. Определение**

**Множество** — это структура данных, которая хранит **уникальные элементы** без определённого порядка (в общем случае).

**2. Основные свойства**

* **Уникальность** — все элементы различны.
* **Отсутствие порядка** (в базовой реализации).
* **Эффективные проверки** на принадлежность (O(1) в хеш-таблицах, O(log n) в деревьях).

**3. Операции и их сложность**

| **Операция** | **Сложность (в среднем)** |
| --- | --- |
| Добавление элемента | O(1) (хеш) / O(log n) (дерево) |
| Удаление элемента | O(1) / O(log n) |
| Проверка принадлежности | O(1) / O(log n) |
| Объединение, пересечение | O(n + m) (зависит от реализации) |

**4. Реализации множества**

1. **Хеш-таблица** (обычно в HashSet)
   * Основана на хеш-функциях.
   * O(1) для вставки/удаления/поиска (в среднем).
   * Нет порядка элементов.
2. **Сбалансированное дерево** (например, TreeSet)
   * Элементы хранятся отсортированно.
   * O(log n) для операций.
   * Поддерживает порядок.

**Пример кода (Python)**

A blue screen with white text

AI-generated content may be incorrect.

**6. Применение множеств**

* Удаление дубликатов из коллекции.
* Проверка уникальности данных.
* Множественные операции (объединение, пересечение, разность).
* Кэширование проверок (например, "уже обработанные элементы").

**7. Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Быстрые проверки (O(1)) | Нет порядка (в хеш-реализации) |
| Эффективное хранение уникальных данных | Требует больше памяти, чем список |

# Словари

**1. Определение**

**Словарь** — это структура данных, хранящая пары **ключ-значение**, где каждый ключ уникален.

**2. Основные свойства**

* **Уникальность ключей** (каждый ключ встречается только один раз).
* **Быстрый доступ** к значению по ключу (O(1) в среднем для хеш-таблиц).
* **Динамическое изменение** (можно добавлять, удалять и изменять элементы).

**3. Реализации словаря**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип реализации** | **Структура** | **Время доступа** | **Порядок элементов** |
| **Хеш-таблица** | Массив + хеш-функция | O(1) (в среднем) | Нет порядка |
| **Сбалансированное дерево** | Красно-чёрное/AVL-дерево | O(log n) | Элементы отсортированы |

**4. Основные операции**

|  |  |  |
| --- | --- | --- |
| **Операция** | **Описание** | **Сложность (хеш-таблица)** |
| d[key] = value | Добавление/изменение | O(1) |
| value = d[key] | Получение значения | O(1) |
| del d[key] | Удаление по ключу | O(1) |
| key in d | Проверка наличия ключа | O(1) |

**Пример кода (Python)**

A screen shot of a computer program

AI-generated content may be incorrect.

**6. Применение словарей**

* **Хранение настроек** (ключ — параметр, значение — настройка).
* **Подсчёт частот** (например, частоты слов в тексте).
* **Кэширование** (мемоизация, кеш данных).
* **Замена switch-case** (сопоставление ключей и функций).

**7. Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Быстрый доступ (O(1)) | Нет порядка (в хеш-таблице) |
| Гибкость (динамическое изменение) | Больший расход памяти, чем у массивов |
| Удобен для сопоставлений | Коллизии хешей могут замедлять доступ |

# Стек

1. Определение

Стек — это структура данных, работающая по принципу **LIFO** (Last In, First Out - "последним пришёл, первым вышел"). Элементы добавляются и удаляются только с одного конца, называемого **вершиной стека**.

2. Основные операции

* **push** (добавление элемента на вершину стека)
* **pop** (удаление элемента с вершины стека)
* **peek/top** (просмотр верхнего элемента без удаления)
* **isEmpty** (проверка на пустоту)
* **size** (получение количества элементов)

3. Временная сложность операций

Все основные операции выполняются за **O(1)** (константное время)

4. Реализации стека

1. **На массиве** (статический стек с фиксированным размером)
2. **На связном списке** (динамический стек)
3. **Встроенные реализации** (Stack в Java, list в Python)

5. Пример реализации на Python

A screen shot of a computer code

AI-generated content may be incorrect.

6. Применение стеков

* **Отмена операций** (undo в текстовых редакторах)
* **Парсинг** (проверка сбалансированности скобок)
* **Вызов функций** (стек вызовов в программировании)
* **Обход графов** (DFS - depth-first search)
* **Вычисление выражений** (обратная польская запись)

7. Преимущества и недостатки

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Быстрые операции (O(1)) | Ограниченный доступ (только к вершине) |
| Простая реализация | Может переполниться (в массиве фиксированного размера) |
| Полезен для многих алгоритмов | Не подходит для задач, требующих произвольного доступа |

8. Типичные задачи на стек

1. Проверка правильности скобочной последовательности
2. Реализация калькулятора выражений
3. Обработка отмены операций (undo/redo)
4. Преобразование инфиксной записи в постфиксную

# Очередь с приоритетом

**1. Определение**

**Очередь с приоритетом** — это структура данных, в которой элементы хранятся не в порядке добавления, а в соответствии с их **приоритетом**. Элемент с **наивысшим (или наименьшим)** приоритетом извлекается первым.

**2. Основные операции**

|  |  |  |
| --- | --- | --- |
| **Операция** | **Описание** | **Сложность** |
| insert | Добавление элемента | O(log n) |
| extract\_max / extract\_min | Извлечение элемента с максимальным/минимальным приоритетом | O(log n) |
| peek | Просмотр элемента с наивысшим приоритетом (без удаления) | O(1) |
| is\_empty | Проверка на пустоту | O(1) |

**3. Реализации очереди с приоритетом**

**3.1. Двоичная куча (Binary Heap)**

* **Структура:** Полное бинарное дерево, где каждый родитель больше (max-heap) или меньше (min-heap) своих потомков.
* **Преимущества:**
  + Эффективные вставка и извлечение (O(log n)).
  + Минимальные накладные расходы на память.
* **Недостатки:**
  + Не поддерживает быстрый поиск произвольного элемента (O(n)).

**3.2. Сбалансированное бинарное дерево поиска (AVL, Красно-чёрное дерево)**

* **Структура:** Дерево с автоматической балансировкой.
* **Преимущества:**
  + Поддерживает дополнительные операции (например, удаление произвольного элемента за O(log n)).
* **Недостатки:**
  + Более сложная реализация.

**3.3. Встроенные реализации**

* В Python: модуль heapq (реализация min-heap).
* В Java: PriorityQueue.
* В C++: priority\_queue из STL.

**4. Пример реализации на Python (**heapq**)**

A screenshot of a computer

AI-generated content may be incorrect.

**5. Применение**

* **Алгоритм Дейкстры** (поиск кратчайшего пути).
* **Huffman coding** (алгоритм сжатия данных).
* **Системы планирования задач** (например, в ОС).
* **Ивент-ориентированное моделирование** (обработка событий в порядке приоритета).

**6. Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Быстрое извлечение max/min (O(1) для peek, O(log n) для extract) | Неэффективный поиск произвольного элемента |
| Динамическое управление приоритетами | Нет стабильности порядка (элементы с одинаковым приоритетом могут извлекаться в произвольном порядке) |
| Широкое применение в алгоритмах | Требует реализации компаратора для сложных объектов |

**7. Сравнение с обычной очередью (FIFO)**

|  |  |
| --- | --- |
| **Очередь (FIFO)** | **Очередь с приоритетом** |
| Первым пришёл — первым вышел | Первым выходит элемент с наивысшим приоритетом |
| Реализуется через список/связный список | Реализуется через кучу/сбалансированное дерево |
| Операции за O(1) | Операции за O(log n) |

# Дек

**1. Определение**

**Дек** (от англ. *double-ended queue*) — это структура данных, которая позволяет **добавлять и удалять элементы с обоих концов** (головы и хвоста).

**2. Основные операции**

|  |  |  |
| --- | --- | --- |
| **Операция** | **Описание** | **Сложность (в среднем)** |
| push\_front(x) | Добавляет x в начало | O(1) |
| push\_back(x) | Добавляет x в конец | O(1) |
| pop\_front() | Удаляет первый элемент | O(1) |
| pop\_back() | Удаляет последний элемент | O(1) |
| front() | Возвращает первый элемент | O(1) |
| back() | Возвращает последний элемент | O(1) |
| is\_empty() | Проверяет, пуст ли дек | O(1) |

**3. Реализации дека**

**3.1. На основе двусвязного списка**

* **Плюсы:**
  + Все операции выполняются за O(1).
  + Динамически расширяется.
* **Минусы:**
  + Больше расход памяти на хранение указателей.

**3.2. На основе кольцевого буфера (массива с индексами)**

* **Плюсы:**
  + Эффективен по памяти (если размер фиксирован).
* **Минусы:**
  + При переполнении требует расширения (O(n) в худшем случае).

**3.3. Встроенные реализации**

* **Python:** collections.deque (реализован на двусвязном списке).

**4. Пример реализации на Python**

A screenshot of a computer

AI-generated content may be incorrect.

**5. Применение дека**

* **Алгоритм скользящего окна** (например, поиск максимума в подмассиве).
* **Обход графов (BFS)** с возможностью добавления элементов в начало/конец.
* **Реализация стека и очереди** одновременно.
* **Задачи с ограниченной историей** (например, undo/redo).

**6. Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Гибкость (добавление/удаление с двух сторон) | В некоторых реализациях (массив) может быть O(n) при расширении |
| Эффективные операции (O(1)) | На практике может быть медленнее, чем простая очередь |
| Может заменять стек и очередь | В C++ std::deque сложнее для понимания, чем vector |

**7. Сравнение с другими структурами**

|  |  |  |  |
| --- | --- | --- | --- |
| **Структура** | **Добавление** | **Удаление** | **Доступ** |
| **Дек** | С любого конца (O(1)) | С любого конца (O(1)) | O(1) к краям |
| **Очередь** | Только в конец (O(1)) | Только из начала (O(1)) | O(1) к началу |
| **Стек** | Только в конец (O(1)) | Только с конца (O(1)) | O(1) к вершине |

# Сортировки

**1. Встроенные методы сортировки**

**1.1.**sorted()

* Возвращает **новый отсортированный список**, не изменяя исходный.
* Работает с любыми итерируемыми объектами (списки, кортежи, строки).

**1.2.**list.sort()

* Сортирует **исходный список** (in-place).
* **Не возвращает новый список** (возвращает None).

**1.3. Параметры**key**и**reverse

* key — функция, определяющая порядок сортировки
* reverse – сортировка в обратном порядке

**2. Алгоритмы сортировки**

**2.1. Пузырьковая сортировка (Bubble Sort)**

* **Принцип:** сравнивает соседние элементы и меняет их местами.
* **Сложность:** O(n²) (медленный для больших данных).

**2.2. Быстрая сортировка (Quick Sort)**

* **Принцип:** Разделяй и властвуй (выбор опорного элемента).
* **Сложность:** O(n log n) в среднем, O(n²) в худшем случае.

**2.3. Сортировка слиянием (Merge Sort)**

* **Принцип:** Рекурсивное разбиение и слияние упорядоченных подмассивов.
* **Сложность:** O(n log n) (стабильно быстрый).

**2.4. Сортировка вставками (Insertion Sort)**

* **Принцип:** Постепенное построение отсортированного списка.
* **Сложность:** O(n²) (эффективен для почти упорядоченных данных).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Алгоритм** | **Лучший случай** | **Средний случай** | **Худший случай** | **Память** |
| **Bubble Sort** | O(n) | O(n²) | O(n²) | O(1) |
| **Quick Sort** | O(n log n) | O(n log n) | O(n²) | O(log n) |
| **Merge Sort** | O(n log n) | O(n log n) | O(n log n) | O(n) |
| **Insertion Sort** | O(n) | O(n²) | O(n²) | O(1) |

**4. Выбор алгоритма**

* **Малые данные** → Insertion Sort (простота).
* **Большие данные** → Quick Sort или Merge Sort.
* **Готовые решения** → sorted() или list.sort() (оптимизированы в Python).