

Лекция 7. Сортировки и O-нотация

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

- ❶ Что такое сортировка? Зачем она нужна?
- ❷ Сортировка пузырьком (Bubble Sort)
- ❸ Сортировка выбором (Selection Sort)
- ❹ Сортировка вставками (Insertion Sort)
- ❺ Быстрая сортировка (Quick Sort)
- ❻ Оценка сложности алгоритмов (О-нотация)
- ❼ Сравнение алгоритмов сортировки

Что такое сортировка?

Определение

Сортировка — это процесс упорядочения элементов коллекции по заданному правилу (обычно по возрастанию или убыванию).

- Вход: массив чисел (или других объектов, для которых определено отношение порядка).
- Выход: тот же массив, но элементы расположены в порядке возрастания (или убывания).
- Сортировка — одна из самых фундаментальных задач в программировании.

Пример

Исходный массив: 5, 2, 8, 1, 3

Отсортированный: 1, 2, 3, 5, 8

Зачем нужна сортировка?

- Упорядоченные данные легче искать (бинарный поиск).
- Упорядоченные данные удобны для анализа (медиана, квантили).
- Многие алгоритмы требуют отсортированных данных.
- Сортировка используется в базах данных, поисковых системах, обработке данных.

Важно

Сортировка — это не только про числа. Можно сортировать строки, структуры, любые объекты, для которых определено сравнение.

- **По устойчивости:**

- Устойчивая — сохраняет относительный порядок равных элементов.
- Неустойчивая — порядок равных элементов может измениться.

- **По использованию памяти:**

- In-place — использует постоянный объём дополнительной памяти, $O(1)$.
- Не in-place — требуется дополнительная память (например, $O(n)$).

- **По сложности:**

- Квадратичные ($O(n^2)$) — пузырьрёк, выбор, вставки.
- Линейно-логарифмические ($O(n \log n)$) — быстрая (в среднем), сортировка слиянием, пирамидальная.

Общая структура задачи сортировки

- На входе: массив `arr` размера `n`.
- Требуется переставить элементы так, чтобы `arr[i] ≤ arr[j]` для всех `i < j`.
- Используем только статические массивы и указатели, т.к. с динамикой и `std::vector` еще не работали.

Стандартная сигнатура функции сортировки

```
void sort(int* arr, int size);  
// arr - указатель на первый элемент  
// size - количество элементов
```

- Все рассматриваемые функции переставляют элементы внутри исходного массива
- Для обмена используем временную переменную.

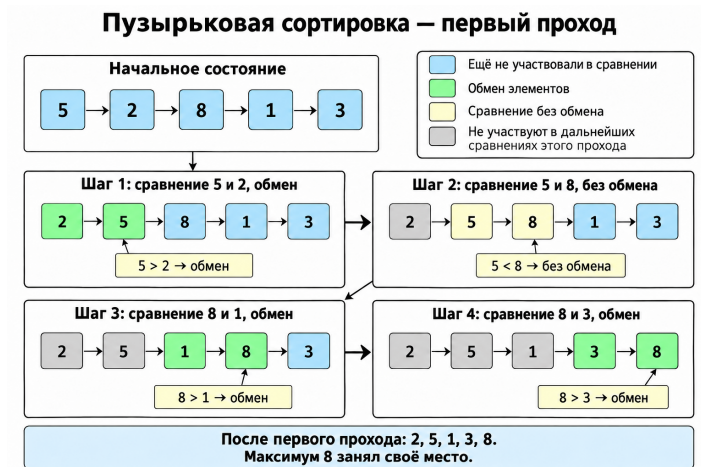
Идея сортировки пузырьком

Основная идея

Последовательно сравниваются соседние элементы, и если они стоят в неправильном порядке, они меняются местами. В результате каждого прохода максимальный элемент оставшейся неотсортированной части «всплывает» в конец массива (возрастание).

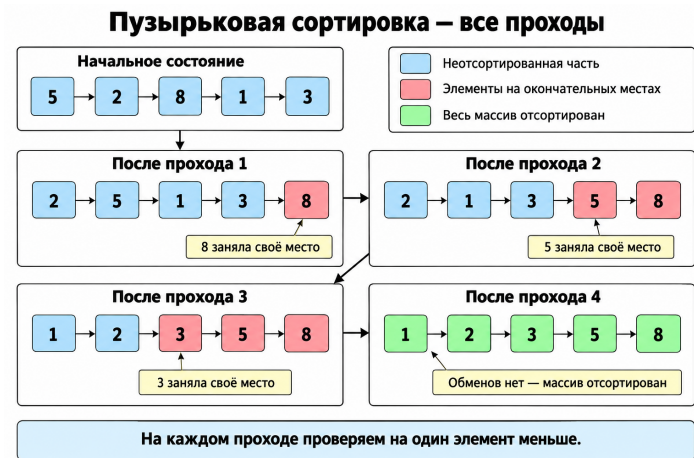
- Алгоритм делает несколько проходов по массиву.
- На каждом проходе сравниваются пары $\text{arr}[i]$ и $\text{arr}[i+1]$.
- Если $\text{arr}[i] > \text{arr}[i+1]$, они меняются местами.
- После каждого прохода на место становится максимум оставшейся неотсортированной части.
- Количество проходов — $n-1$.

Визуализация пузырька: первый проход



На первом проходе максимальный элемент 8 «всплывает» в конец.

Визуализация пузырька: все проходы



После каждого прохода массив становится более упорядоченным.

Реализация пузырька

```
void bubbleSort(int* arr, int size) {  
    for (int i = 0; i < size - 1; ++i) {  
        // На каждом проходе последние i элементов уже отсортированы  
        for (int j = 0; j < size - 1 - i; ++j) {  
            if (arr[j] > arr[j+1]) {  
                // обмен соседних элементов  
                int temp = arr[j];  
                arr[j] = arr[j+1];  
                arr[j+1] = temp;  
            }  
        }  
    }  
}
```

- Внешний цикл — количество проходов ($n-1$).
- Внутренний цикл — сравнения пары, до $size-1-i$.
- Можно оптимизировать: если за проход не было обменов, массив уже отсортирован.

Оптимизированный пузырьрёк

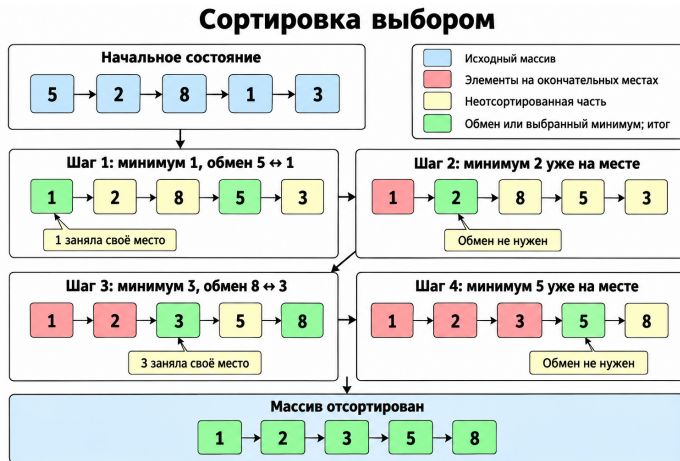
```
void bubbleSortOptimized(int* arr, int size) {
    bool swapped;
    for (int i = 0; i < size - 1; ++i) {
        swapped = false;
        for (int j = 0; j < size - 1 - i; ++j) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = true;
            }
        }
        // Если не было обменов, массив отсортирован
        if (!swapped) break;
    }
}
```

- Флаг `swapped` отслеживает, были ли обмены.
- Если обменов не было, алгоритм завершается досрочно.
- В лучшем случае (отсортированный массив) — $O(n)$.

Основная идея

На каждом шаге находим минимальный элемент в неотсортированной части массива и меняем его местами с первым элементом этой части.

- Разбиваем массив на две части: отсортированную (слева) и неотсортированную (справа).
- На каждом шаге ищем минимальный элемент в неотсортированной части.
- Меняем его местами с первым элементом неотсортированной части.
- Граница отсортированной части сдвигается вправо.



На каждом шаге находится минимальный элемент и перемещается в начало неотсортированной части.

Реализация сортировки выбором

```
void selectionSort(int* arr, int size) {  
    for (int i = 0; i < size - 1; ++i) {  
        int minIndex = i; // индекс минимального элемента  
        for (int j = i + 1; j < size; ++j) {  
            if (arr[j] < arr[minIndex]) {  
                minIndex = j;  
            }  
        }  
        // обмен минимального с текущим  
        if (minIndex != i) {  
            int temp = arr[i];  
            arr[i] = arr[minIndex];  
            arr[minIndex] = temp;  
        }  
    }  
}
```

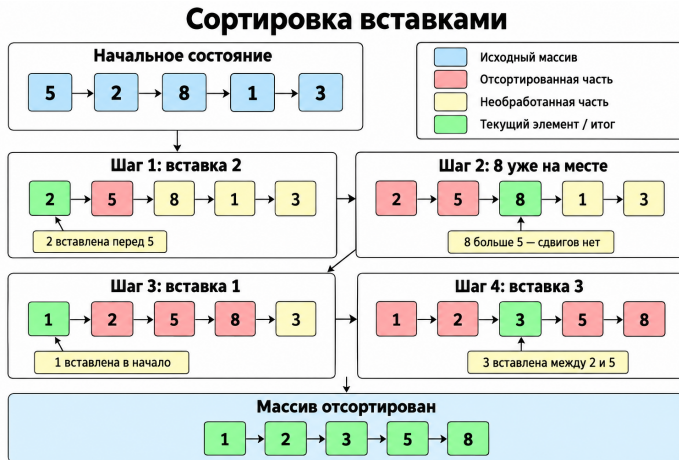
- Внешний цикл проходит по позициям от 0 до $n-2$.
- Внутренний цикл ищет минимальный элемент в неотсортированной части.
- После обмена элемент `arr[i]` становится на своё место.

- Всегда выполняется $n \times (n-1) / 2$ сравнений, независимо от входных данных.
- Количество обменов: не более $n-1$ (если минимум уже на месте, обмена нет).
- **Сложность:** $O(n^2)$ по времени, $O(1)$ по памяти (in-place).
- **Неустойчивая** — порядок равных элементов может измениться.
- **Преимущество:** простота, малое количество обменов.
- **Недостаток:** всегда квадратичная сложность, неэффективен для больших массивов.

Основная идея

Элементы поочерёдно вставляются в уже отсортированную часть массива, сдвигая большие элементы вправо.

- Массив разбивается на отсортированную (слева) и неотсортированную (справа) части.
- Берём первый элемент из неотсортированной части.
- Сдвигаем элементы в отсортированной части, которые больше текущего, вправо.
- Вставляем текущий элемент на освободившееся место.



Элемент 3 вставляется между 2 и 5 в отсортированную часть [1, 2, 5, 8]. Элементы 5 и 8 сдвигаются вправо.

Реализация сортировки вставками

```
void insertionSort(int* arr, int size) {  
    for (int i = 1; i < size; ++i) {  
        int key = arr[i];    // текущий элемент для вставки  
        int j = i - 1;  
        // сдвиг элементов, больших key  
        while (j >= 0 && arr[j] > key) {  
            arr[j + 1] = arr[j];  
            --j;  
        }  
        arr[j + 1] = key;    // вставка  
    }  
}
```

- Внешний цикл начинается с 1 (первый элемент уже отсортирован).
- Внутренний цикл сдвигает элементы вправо.
- После сдвига key вставляется на своё место.

Анализ сложности сортировки вставками

- **Лучший случай:** массив уже отсортирован — $O(n)$ (только сравнения).
- **Худший случай:** массив отсортирован в обратном порядке — $O(n^2)$.
- **Средний случай:** $O(n^2)$.
- **Память:** $O(1)$ (in-place).
- **Устойчивая:** порядок равных элементов сохраняется.
- **Преимущество:** эффективен на почти отсортированных данных.
- **Недостаток:** квадратичная сложность на больших данных.

Идея быстрой сортировки

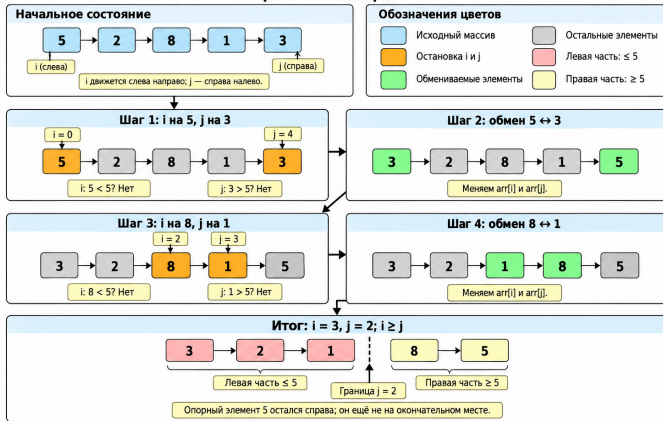
Основная идея

Алгоритм «разделяй и властвуй»:

1. Выбрать опорный элемент (pivot).
 2. Разбить массив на две части: элементы $\leq \text{pivot}$ и элементы $\geq \text{pivot}$.
 3. Рекурсивно отсортировать обе части.
- После разбиения элементы левой части не больше опорного значения, элементы правой — не меньше. Сам опорный элемент не обязательно занимает окончательное место. Рекурсивно сортируются обе полученные части.
 - Базовый случай: подмассив из 0 или 1 элемента уже отсортирован.

Разбиение массива — схема Хоара

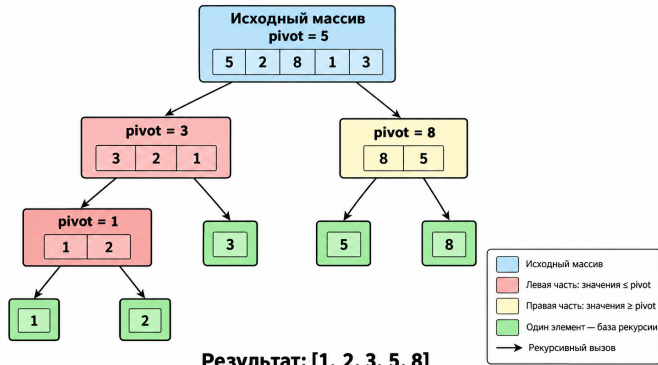
Опорное значение $\text{pivot} = 5$



Элементы меньше опорного перемещаются влево, больше — вправо.

Быстрая сортировка — дерево рекурсии

Схема Хоара; опорное значение — первый элемент части



Граница частей не обязана совпадать с позицией опорного элемента.

Рекурсивная сортировка подмассивов.

Реализация разбиения (схема Хоара)

```
int partition(int* arr, int low, int high) {  
    int pivot = arr[low]; // опорный - первый элемент  
    int i = low - 1;  
    int j = high + 1;  
    while (true) {  
        do { ++i; } while (arr[i] < pivot);  
        do { --j; } while (arr[j] > pivot);  
        if (i >= j) return j;  
        // обмен arr[i] и arr[j]  
        int temp = arr[i];  
        arr[i] = arr[j];  
        arr[j] = temp;  
    }  
}
```

- Использует два индекса: i движется вправо, j — влево.
- Останавливаются на элементах, которые нужно поменять местами.
- Возвращает индекс, разделяющий части.

Реализация быстрой сортировки

```
void quickSort(int* arr, int low, int high) {  
    if (low < high) {  
        int split = partition(arr, low, high);  
        quickSort(arr, low, split);  
        quickSort(arr, split + 1, high);  
    }  
}
```

- Функция принимает границы подмассива.
- После разбиения рекурсивно сортируются левая и правая части.
- Базовый случай: $low \geq high$.

Запуск

```
quickSort(arr, 0, size - 1);
```

Выбор опорного элемента (pivot)

- От выбора опорного элемента сильно зависит производительность.
- **Варианты:**
 - Первый элемент — просто, но может быть неудачным (например, на отсортированном массиве).
 - Случайный элемент — уменьшает вероятность худшего случая.
 - Медиана трёх — выбирается медиана из первого, среднего и последнего — обычно улучшает разбиение.
- В нашей реализации мы использовали первый элемент для простоты.
- На практике лучше выбирать случайный или медиану трёх.

- **Лучший и средний случай:** $O(n \log n)$ — при хорошем разбиении.
- **Худший случай:** $O(n^2)$ — повторяющееся разбиение на части из 1 и $n - 1$ элементов.
- **Память:** $O(\log n)$ для рекурсивного стека (в среднем), $O(n)$ в худшем (если неудачное разбиение).
- **Неустойчивая:** порядок равных элементов не гарантируется.
- **Преимущество:** очень быстрая на практике благодаря хорошей локальности и эффективному разбиению.
- **Недостаток:** худший случай $O(n^2)$, но вероятность его крайне мала при случайном выборе pivot.

Оптимизация быстрой сортировки

- Для малых подмассивов (например, размер не более 10) использовать сортировку вставками — это уменьшает накладные расходы на рекурсию.
- Выбор медианы трёх в качестве опорного.
- Альтернативная реализация разбиения (схема Ломута) может быть проще, но схема Хоара эффективнее.

Оптимизированная версия

```
void quickSortOpt(int* arr, int low, int high) {  
    if (high - low < 10) {  
        insertionSort(arr + low, high - low + 1);  
        return;  
    }  
    int split = partition(arr, low, high);  
    quickSortOpt(arr, low, split);  
    quickSortOpt(arr, split + 1, high);  
}
```

Что такое сложность алгоритма?

Сложность алгоритма

Это мера количества ресурсов (времени и памяти), которые требуются для выполнения алгоритма в зависимости от размера входных данных.

- **Временная сложность** — сколько операций (или времени) нужно.
- **Пространственная сложность** — сколько дополнительной памяти требуется.
- Оценивается в виде функции от n (размера входных данных).

Определение

Говорят, что $f(n) = O(g(n))$, если существуют константы $C > 0$ и n_0 , такие что для всех $n > n_0$ выполняется:

$$f(n) \leq C \cdot g(n)$$

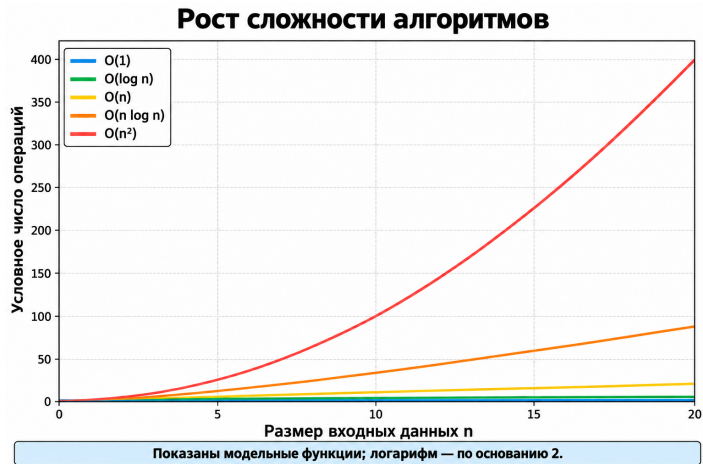
- $O(g(n))$ — класс функций, которые растут не быстрее, чем $g(n)$ (с точностью до константы).
- Игнорируем константы и младшие степени.
- Например: $3n + 2 = O(n)$, $5n^2 + 3n + 1 = O(n^2)$.

Сложность	Название и примеры
$O(1)$	Константная — доступ к элементу массива по индексу.
$O(\log n)$	Логарифмическая — бинарный поиск.
$O(n)$	Линейная — поиск максимума в неотсортированном массиве.
$O(n \log n)$	Линейно-логарифмическая — быстрая сортировка (в среднем), сортировка слиянием.
$O(n^2)$	Квадратичная — пузырьрёк, выбор, вставки (худший случай).
$O(2^n)$	Экспоненциальная — рекурсивное вычисление чисел Фибоначчи (без мемоизации).
$O(n!)$	Факториальная — перебор всех перестановок.

Сравнение роста числа операций

n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
10	1	3	10	33	100
100	1	7	100	664	10 000
1000	1	10	1000	9966	1 000 000
10 000	1	13	10 000	132 877	100 000 000
100 000	1	17	100 000	1 660 964	10^{10}

При увеличении n квадратичное число операций растёт значительно быстрее, чем $n \log n$.



Наглядно видно, как быстро растут $O(n^2)$.

Как вычислять сложность алгоритма?

- **Циклы:** сложность цикла равна сложности тела, умноженной на количество итераций.
- **Вложенные циклы:** считаем общее число выполнений тела. Если число внутренних итераций одинаково для каждого внешнего шага, количества итераций перемножаются.
- **Рекурсия:** решаем рекуррентное соотношение (например, для быстрой сортировки (при разбиении примерно пополам) — $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$).
- **Анализ операций:** считаем количество сравнений, обменов и т.д.
- **Пример:** пузырьрёк — два вложенных цикла, каждый примерно n итераций $\rightarrow O(n^2)$.

Вопрос: что можно сказать о сложности?

```
int sum = 0;
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        sum += arr[i][j];
```

Вопрос: что можно сказать о сложности?

```
int sum = 0;
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        sum += arr[i][j];
```

Ответ: $O(n^2)$ — два вложенных цикла по n итераций.

Вопрос: что можно сказать о сложности?

```
int count = 0;
for (int i = 1; i < n; i *= 2)
    for (int j = 0; j < n; ++j)
        count++;
```

Вопрос: что можно сказать о сложности?

```
int count = 0;
for (int i = 1; i < n; i *= 2)
    for (int j = 0; j < n; ++j)
        count++;
```

Ответ: $O(n \log n)$ — внешний цикл выполняется $\log_2 n$ раз, внутренний — n раз.

Вопрос: что можно сказать о сложности?

```
int sum = 0;
for (int i = 0; i < n; ++i)
    for (int j = 0; j < i; ++j)
        sum += a[i][j];
```

Вопрос: что можно сказать о сложности?

```
int sum = 0;
for (int i = 0; i < n; ++i)
    for (int j = 0; j < i; ++j)
        sum += a[i][j];
```

Ответ: $O(n^2)$ — внутренний цикл выполняется в среднем $n/2$ раз, общее число итераций $\frac{n(n-1)}{2}$, что даёт $O(n^2)$.

Вопрос: что можно сказать о сложности?

```
int fib(int n) {  
    if (n <= 1) return n;  
    return fib(n-1) + fib(n-2);  
}
```

Вопрос: что можно сказать о сложности?

```
int fib(int n) {  
    if (n <= 1) return n;  
    return fib(n-1) + fib(n-2);  
}
```

Ответ: $O(2^n)$ — каждое рекурсивное ветвление даёт два вызова, образуя бинарное дерево высоты n .

Сводная таблица алгоритмов сортировки

Алгоритм	Лучший случай	Средний случай	Худший случай	Память
Пузырёк	$O(n)$ (слайд 11)	$O(n^2)$ (слайд 10)	$O(n^2)$	$O(1)$
Выбор	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Вставки	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Быстрая	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ (в среднем)

Быстрая сортировка эффективна на больших массивах, но выбор алгоритма зависит от данных и требований.

Алгоритм	Устойчивый?	Переставляет элементы внутри исходного массива?
Пузырёк	Да	Да
Выбор	Нет	Да
Вставки	Да	Да
Быстрая	Нет (обычно)	Да

- Устойчивость важна, когда нужно сохранить порядок равных элементов.
- In-place экономит память.
- Быстрая сортировка нестабильна, но очень быстрая.
- Рекурсивная быстрая сортировка использует стек: в среднем — $O(\log n)$, в худшем случае — $O(n)$.

Когда какой алгоритм использовать?

- **Маленькие массивы ($n < 10-20$):** Сортировка вставками — быстрее из-за малых накладных расходов.
- **Почти отсортированные данные:** Сортировка вставками обычно особенно эффективна.
- **Большие массивы ($n > 1000$):** Быстрая сортировка (в среднем $O(n \log n)$).
- **Требуется стабильность:** Сортировка вставками или Пузырьковая.
- **Ограниченная память:** любой in-place алгоритм.

Важно

В распространённых реализациях `std::sort` используются комбинированные алгоритмы, например `introsort`. Стандарт требует $O(n \log n)$ сравнений, но не задаёт конкретный алгоритм.

- Познакомились с четырьмя алгоритмами сортировки: пузырьрёк, выбор, вставки, быстрая.
- Изучили их идеи, реализации и сложность.
- Разобрали O -нотацию — способ оценки сложности алгоритмов.
- Научились сравнивать алгоритмы по времени и памяти.
- Теперь вы можете выбирать подходящий алгоритм для своих задач.

Следующая лекция — исключения и обработка ошибок.

Вопросы?