Отчет по лабораторной работе 4

Алгоритмы сортировки

Дата: 2025-10-10 **Семестр:** 3 курс 5 семестр **Группа:** ПИЖ-б-о-23-2(1) **Дисциплина:** Анализ сложности алгоритмов **Студент:** Черников Дмитрий Дмитриевич

Цель работы

Изучить и реализовать основные алгоритмы сортировки. Провести их теоретический и практический сравнительный анализ по временной и пространственной сложности. Исследовать влияние начальной упорядоченности данных на эффективность алгоритмов. Получить навыки эмпирического анализа производительности алгоритмов.

Практическая часть

Выполненные задачи

- ☐ Задача 1: Реализовать 5 алгоритмов сортировки.
- ☐ Задача 2: Провести теоретический анализ сложности каждого алгоритма.
- ☐ Задача 3: Экспериментально сравнить время выполнения алгоритмов на различных наборах данных.
- ☐ Задача 4: Проанализировать влияние начальной упорядоченности данных на эффективность сортировок.

Ключевые фрагменты кода

```
# sorts.py

def bubble_sort(ar):
    """
    Сортировка пузырьком.
    Сравнивает соседние элементы и меняет их местами,
    если они идут в неправильном порядке.
    Сложность: худший O(n^2), средний O(n^2), лучший O(n). Память: O(1).
    """

    for i in range(len(ar) - 1, 0, -1):
        for j in range(0, i):
            if (ar[j] > ar[j+1]):
                temp = ar[j+1]
                ar[j+1] = ar[j]
                ar[j] = temp

    return ar

# Время: худший O(n^2), средний O(n^2), лучший O(n)
```

```python
# Память: O(1)


def selection_sort(ar):
    """
    Сортировка выбором.
    Находит минимальный элемент в неотсортированной
    части и меняет его с текущим.
    Сложность: худший/средний/лучший O(n^2). Память: O(1).
    """
    for i in range(len(ar)):
        min = 2**100
        min_ind = -1
        for j in range(i, len(ar)):
            if min > ar[j]:
                min = ar[j]
                min_ind = j
        if min_ind != -1:
            temp = ar[i]
            ar[i] = ar[min_ind]
            ar[min_ind] = temp

    return ar

# Время: худший O(n^2), средний O(n^2), лучший O(n^2)
# Память: O(1)


def insertion_sort(ar):
    """
    Сортировка вставками.
    Вставляет каждый элемент на своё место в отсортированной части массива.
    Сложность: худший/средний O(n^2), лучший O(n). Память: O(1).
    """
    for i in range(1, len(ar)):
        key = ar[i]
        j = i - 1
        while j >= 0 and ar[j] > key:
            ar[j + 1] = ar[j]
            j -= 1
        ar[j + 1] = key
    return ar

# Время: худший O(n^2), средний O(n^2), лучший O(n)
# Память: O(1)


def merge_sort(ar):
    """
    Сортировка слиянием.
    Рекурсивно делит массив пополам и сливает отсортированные части.
    Сложность: худший/средний/лучший O(n log n). Память: O(n).
    """
    if len(ar) <= 1:
```

```
        return ar
    mid = len(ar) // 2
    left = merge_sort(ar[:mid])
    right = merge_sort(ar[mid:])
    return merge(left, right)

# Время: худший O(n log n), средний O(n log n), лучший O(n log n)
# Память: O(n)


def merge(left, right):
    """
    Сливает два отсортированных массива в один отсортированный.
    Используется в merge_sort.
    """
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result


def quick_sort(ar):
    """
    Быстрая сортировка (quick sort).
    Делит массив относительно опорного элемента и рекурсивно сортирует части.
    Сложность: худший O(n^2), средний/лучший O(n log n).
    Память: O(log n) (стек рекурсии).
    """
    if len(ar) <= 1:
        return ar
    pivot = ar[len(ar) // 2]
    left = [x for x in ar if x < pivot]
    middle = [x for x in ar if x == pivot]
    right = [x for x in ar if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

# Время: худший O(n^2), средний O(n log n), лучший O(n log n)
# Память: O(log n) (стек рекурсии)
```

```
# generate_data.py

import random
```

```python
def generate_random_data(size):
    """
    Генерирует список случайных целых чисел заданной длины.
    Аргументы:
        size: целое число - размер возвращаемого списка.
    Возвращает:
        Список целых чисел длины size,
        заполненный случайными значениями от 0 до 1_000_000.
    """
    ar = []
    for i in range(size):
        ar.append(random.randint(0, 1_000_000))
    return ar


def generate_sorted_data(size):
    """
    Генерирует отсортированный по возрастанию список случайных целых чисел.
    Аргументы:
        size: целое число - размер возвращаемого списка.
    Возвращает:
        Список целых чисел длины size, отсортированный по возрастанию.
    """
    ar = generate_random_data(size)
    ar.sort()
    return ar


def generate_reversed_data(size):
    """
    Генерирует список случайных целых чисел и выполняет его реверс.
    Аргументы:
        size: целое число - размер возвращаемого списка.
    Возвращает:
        Список целых чисел длины size, элементы которого расположены в
        обратном порядке (по сравнению с исходным случайным списком).
    """
    ar = generate_sorted_data(size)
    return list(reversed(ar))


def generate_almost_sorted_data(size):
    """
    Генерирует почти отсортированный список целых чисел.
    Аргументы:
        size: целое число - размер возвращаемого списка.
    Возвращает:
        Список целых чисел длины size, в котором примерно 5% элементов
        случайным образом перемешаны (остальные элементы отсортированы).
    """
    ar = generate_sorted_data(size)
    for i in range(size // 20):
```

```
        ind1 = random.randint(0, len(ar) - 1)
        ind2 = random.randint(0, len(ar) - 1)
        temp = ar[ind1]
        ar[ind1] = ar[ind2]
        ar[ind2] = temp
    return ar
```

```
# perfomance_test.py

import timeit
import modules.sorts as sorts
import modules.generate_data as gen_data
import matplotlib.pyplot as plt


def perf_test(gen_data, sizes):
    """
    Запускает тестирование производительности всех сортировок на данных,
    сгенерированных функцией gen_data, для каждого размера из sizes.
    Аргументы:
        gen_data: функция-генератор данных (например, generate_random_data).
        sizes: список целых чисел — размеры тестируемых массивов.
    Возвращает:
        Словарь, где ключ — название метода сортировки, значение — список
        времени выполнения (мс) для каждого размера.
    """

    data = {}
    for size in sizes:
        data[size] = gen_data(size)

    measures = {"bubble sort": measure_time(data, sorts.bubble_sort),
                "select sort": measure_time(data, sorts.selection_sort),
                "insert sort": measure_time(data, sorts.insertion_sort),
                "merge sort": measure_time(data, sorts.merge_sort),
                "quick sort": measure_time(data, sorts.quick_sort)}
    return measures


def measure_time(data, function):
    """
    Измеряет время выполнения сортировки для каждого массива из data.
    Аргументы:
        data: словарь {размер: массив}.
        function: функция сортировки (например, bubble_sort).
    Возвращает:
        Список времени выполнения (мс) для каждого массива.
    """
    measures = []
    for ar in data.values():
```

```
        ar_copy = ar.copy()
        start = timeit.default_timer()
        function(ar_copy)
        end = timeit.default_timer()
        measures.append((end-start) * 1000)

    return measures


def Visualization(sizes):
    """
    Строит и сохраняет графики сравнения всех методов сортировки
    на разных типах данных и размерах.
    Аргументы:
        sizes: список целых чисел — размеры тестируемых массивов.
    Возвращает:
        None. Сохраняет графики в папку ОТЧЁТ и отображает их на экране.
    """
    random_data_measures = perf_test(gen_data.generate_random_data, sizes)
    sorted_data_measures = perf_test(gen_data.generate_sorted_data, sizes)
    reversed_data_measures = perf_test(gen_data.generate_reversed_data, sizes)
    almost_sorted_data_measures = perf_test(
        gen_data.generate_almost_sorted_data, sizes)

    Create_plot(random_data_measures, sizes,
                "Сравнение методов сортировки на рандомных данных",
                "./report/random_data_all_methods.png")
    Create_plot(sorted_data_measures, sizes,
                "Сравнение методов сортировки на отсортированных данных",
                "./report/sorted_data_all_methods.png")
    Create_plot(reversed_data_measures, sizes,
                "Сравнение методов сортировки на реверсных данных",
                "./report/reversed_data_all_methods.png")
    Create_plot(almost_sorted_data_measures, sizes,
                "Сравнение методов сортировки на почти отсортированных данных",
                "./report/almost_sorted_data_all_methods.png")


def Create_plot(data, sizes, title, path):
    """
    Строит и сохраняет график времени работы сортировок для одного типа данных.
    Аргументы:
        data: словарь {название метода: список времени}.
        sizes: список размеров массивов.
        title: строка — заголовок графика.
        path: строка — путь для сохранения PNG-файла.
    Возвращает:
        None. Сохраняет график и отображает его.
    """
    plt.plot(sizes, data["bubble sort"],
             marker="o", color="red", label="bubble",)
    plt.plot(sizes, data["select sort"],
             marker="o", color="green", label="select",)
    plt.plot(sizes, data["insert sort"],
```

```
            marker="o", color="blue", label="insert",)
    plt.plot(sizes, data["merge sort"],
            marker="o", color="purple", label="merge",)
    plt.plot(sizes, data["quick sort"],
            marker="o", color="brown", label="quick",)
    plt.xlabel("Размер массива n")
    plt.ylabel("Время выполнения ms")
    plt.title(title)
    plt.legend(loc="upper left", title="Методы")
    plt.savefig(path, dpi=300, bbox_inches="tight")
    plt.show()

# , path_csv="./ОТЧЁТ/summary.csv"


def Create_summary_table(sizes):
    """
    Формирует и выводит сводную таблицу результатов тестирования сортировок
    для всех типов данных и размеров.
    Аргументы:
        sizes: список целых чисел — размеры тестируемых массивов.
    Возвращает:
        None. Печатает таблицу в консоль
    """
    # Характеристики вычислительной машины
    pc_info = """
    Характеристики ПК для тестирования:
    - Процессор: Intel Core i5-12500H @ 2.50GHz
    - Оперативная память: 32 GB DDR4
    - ОС: Windows 11
    - Python: 3.12
    """
    print(pc_info)
    generators = {
        "random": gen_data.generate_random_data,
        "sorted": gen_data.generate_sorted_data,
        "reversed": gen_data.generate_reversed_data,
        "almost_sorted": gen_data.generate_almost_sorted_data,
    }

    # вычисление измерений для всех типов данных
    all_measures = {}
    for name, gen in generators.items():
        all_measures[name] = perf_test(gen, sizes)

    # Подгодовка csv линий
    # import csv
    csv_rows = [("data_type", "size", "method", "time_ms")]
    methods = [
        "bubble sort",
        "select sort",
        "insert sort",
        "merge sort",
        "quick sort",
```

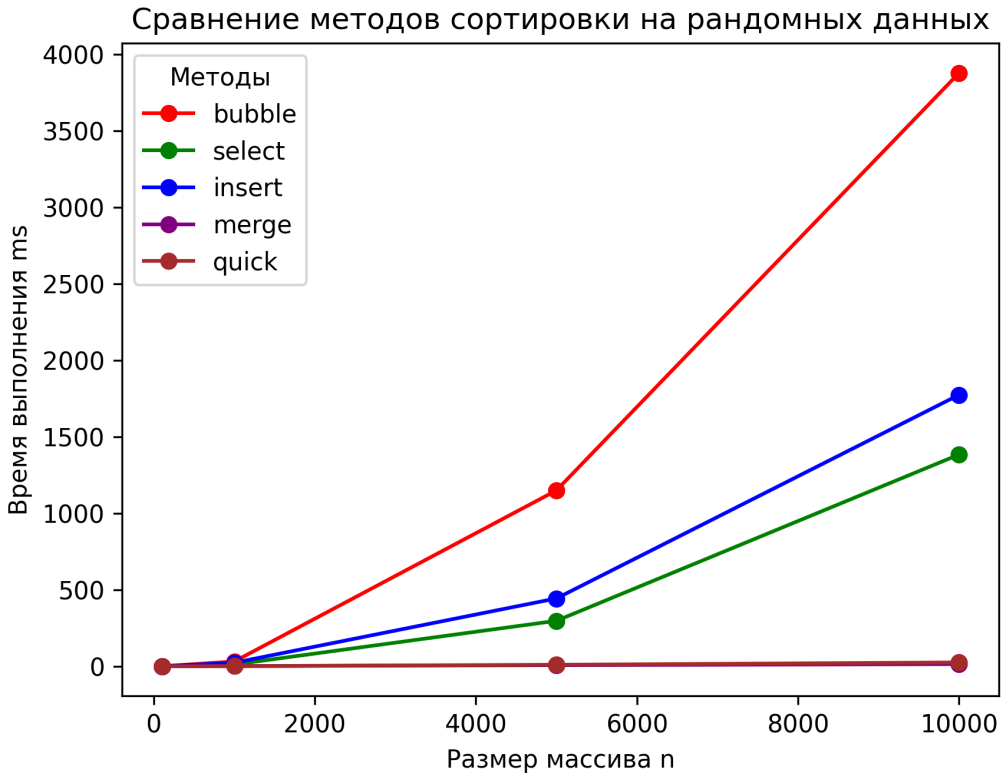
```
    ]

    for data_type, measures in all_measures.items():
        for i, size in enumerate(sizes):
            for method in methods:
                time_ms = measures[method][i]
                csv_rows.append((data_type, size, method, f"{time_ms:.6f}"))

    # Создание csv файла
    # import os
    # os.makedirs(os.path.dirname(path_csv), exist_ok=True)
    # with open(path_csv, "w", newline='', encoding='utf-8') as f:
    #     writer = csv.writer(f)
    #     writer.writerows(csv_rows)

    # Вывод таблицы в консоль
    col_widths = [max(len(str(r[i])) for r in csv_rows) for i in range(4)]
    sep = " | "
    header = csv_rows[0]
    print(sep.join(str(header[i]).ljust(col_widths[i]) for i in range(4)))
    print('-' * (sum(col_widths) + 3 * (len(col_widths) - 1)))
    for row in csv_rows[1:]:
        print(sep.join(str(row[i]).ljust(col_widths[i]) for i in range(4)))

    # print(f"\nSummary CSV written to: {path_csv}")
```

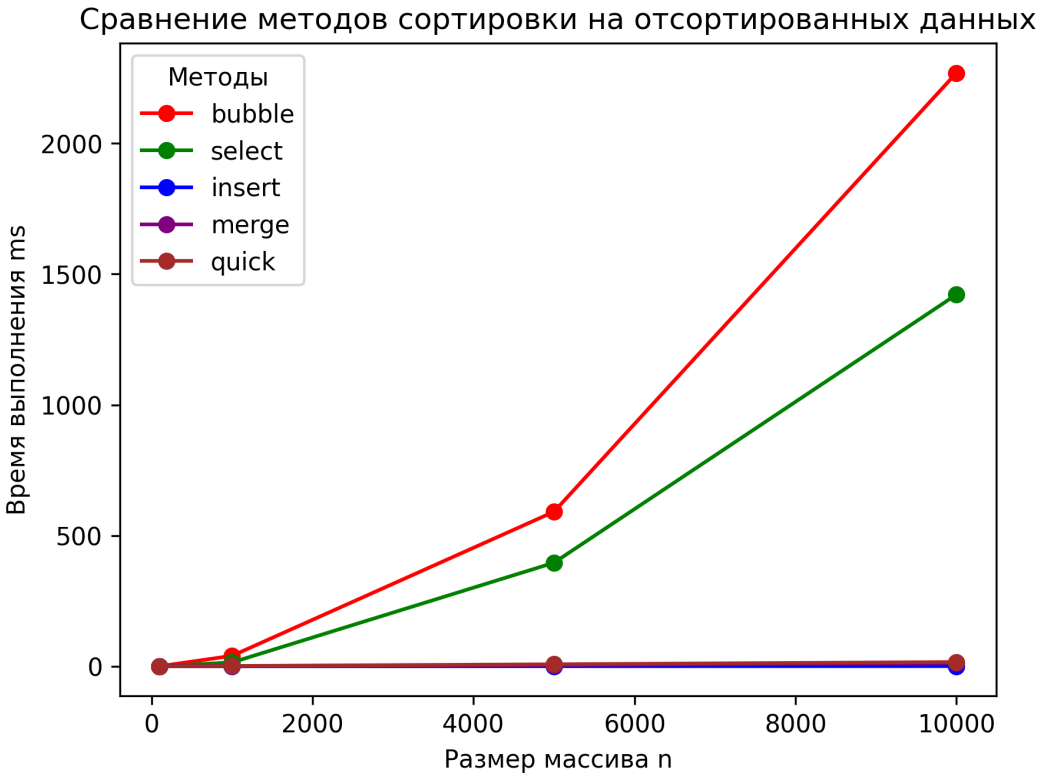
```
# main.py

import modules.perfomance_test as perf_test

sizes = [100, 1000, 5000, 10000]
perf_test.Visualization(sizes)
perf_test.Create_summary_table(sizes)
```

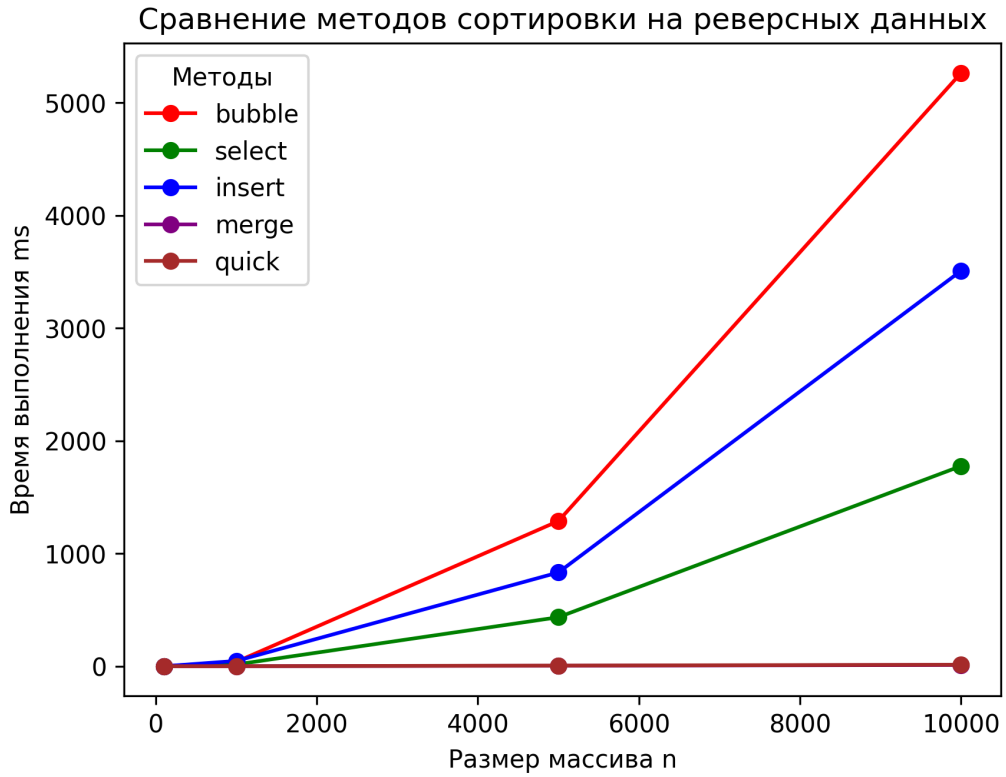
```
    Характеристики ПК для тестирования:
    - Процессор: Intel Core i5-12500H @ 2.50GHz
    - Оперативная память: 32 GB DDR4
    - ОС: Windows 11
    - Python: 3.12

data_type     | size  | method      | time_ms
-------------------------------------------------
random        | 100   | bubble sort | 0.887500
random        | 100   | select sort | 0.318100
random        | 100   | insert sort | 0.118000
random        | 100   | merge sort  | 0.100300
random        | 100   | quick sort  | 0.123200
```

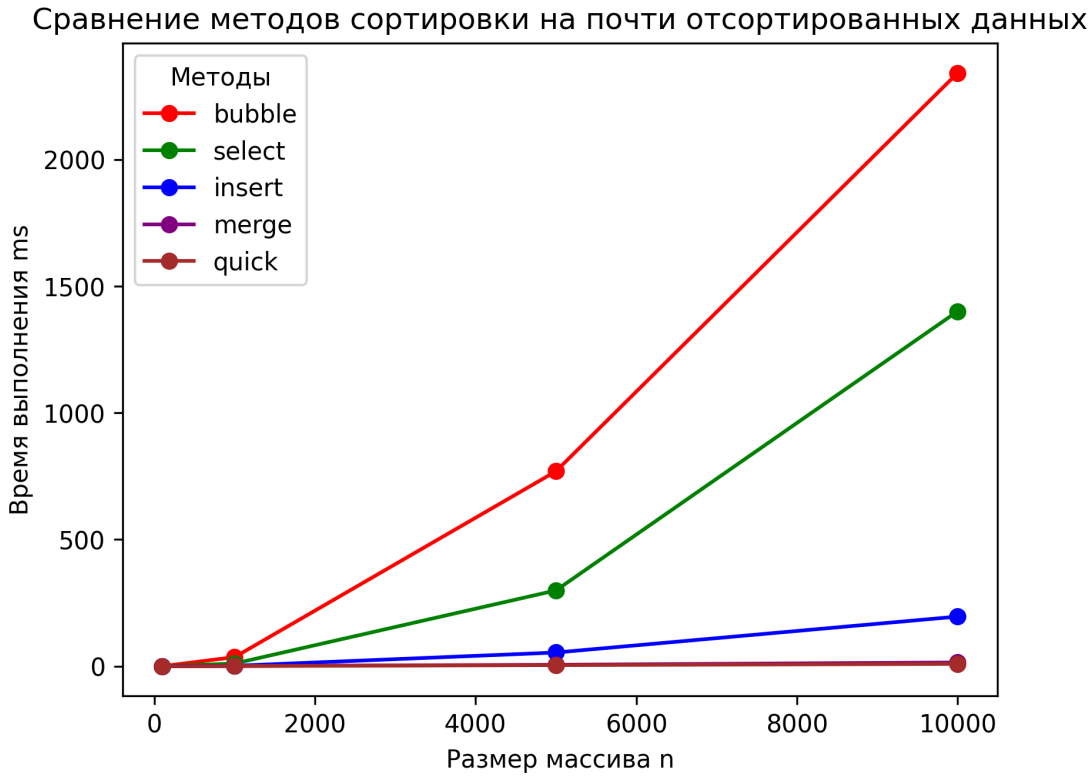
```
random        | 1000  | bubble sort | 43.948800
random        | 1000  | select sort | 26.232100
random        | 1000  | insert sort | 13.895900
random        | 1000  | merge sort  | 1.059200
random        | 1000  | quick sort  | 1.409800
random        | 5000  | bubble sort | 1007.825300
random        | 5000  | select sort | 370.983300
random        | 5000  | insert sort | 386.780600
random        | 5000  | merge sort  | 6.781700
random        | 5000  | quick sort  | 9.354000
random        | 10000 | bubble sort | 4129.669800
random        | 10000 | select sort | 1351.992800
random        | 10000 | insert sort | 1710.378900
random        | 10000 | merge sort  | 15.633100
random        | 10000 | quick sort  | 23.419100
sorted        | 100   | bubble sort | 0.253100
sorted        | 100   | select sort | 0.154600
sorted        | 100   | insert sort | 0.009200
sorted        | 100   | merge sort  | 0.092600
sorted        | 100   | quick sort  | 0.082400
sorted        | 1000  | bubble sort | 26.395600
sorted        | 1000  | select sort | 12.506900
sorted        | 1000  | insert sort | 0.079000
sorted        | 1000  | merge sort  | 0.977000
sorted        | 1000  | quick sort  | 0.880100
sorted        | 5000  | bubble sort | 526.580600
sorted        | 5000  | select sort | 316.926500
sorted        | 5000  | insert sort | 0.393500
sorted        | 5000  | merge sort  | 4.921200
sorted        | 5000  | quick sort  | 4.311700
sorted        | 10000 | bubble sort | 2410.782600
sorted        | 10000 | select sort | 1454.704400
sorted        | 10000 | insert sort | 0.767700
sorted        | 10000 | merge sort  | 12.701700
sorted        | 10000 | quick sort  | 10.356500
reversed      | 100   | bubble sort | 0.269000
reversed      | 100   | select sort | 0.164700
reversed      | 100   | insert sort | 0.473600
reversed      | 100   | merge sort  | 0.128400
reversed      | 100   | quick sort  | 0.170400
reversed      | 1000  | bubble sort | 44.018000
reversed      | 1000  | select sort | 14.707200
reversed      | 1000  | insert sort | 41.668100
reversed      | 1000  | merge sort  | 1.267700
reversed      | 1000  | quick sort  | 0.677200
reversed      | 5000  | bubble sort | 1314.268300
reversed      | 5000  | select sort | 449.457700
reversed      | 5000  | insert sort | 818.359300
reversed      | 5000  | merge sort  | 8.229200
reversed      | 5000  | quick sort  | 5.414800
reversed      | 10000 | bubble sort | 5413.766200
reversed      | 10000 | select sort | 1684.056000
reversed      | 10000 | insert sort | 3490.517200
reversed      | 10000 | merge sort  | 15.460200
```

```
reversed      | 10000 | quick sort  | 10.066100
almost_sorted | 100   | bubble sort | 0.146200
almost_sorted | 100   | select sort | 0.182100
almost_sorted | 100   | insert sort | 0.027500
almost_sorted | 100   | merge sort  | 0.125500
almost_sorted | 100   | quick sort  | 0.067800
almost_sorted | 1000  | bubble sort | 22.912000
almost_sorted | 1000  | select sort | 15.258000
almost_sorted | 1000  | insert sort | 1.704900
almost_sorted | 1000  | merge sort  | 1.181400
almost_sorted | 1000  | quick sort  | 0.718200
almost_sorted | 5000  | bubble sort | 618.788700
almost_sorted | 5000  | select sort | 406.144500
almost_sorted | 5000  | insert sort | 51.530500
almost_sorted | 5000  | merge sort  | 6.893100
almost_sorted | 5000  | quick sort  | 5.051000
almost_sorted | 10000 | bubble sort | 2687.900400
almost_sorted | 10000 | select sort | 1411.839800
almost_sorted | 10000 | insert sort | 210.462300
almost_sorted | 10000 | merge sort  | 14.076700
almost_sorted | 10000 | quick sort  | 10.871100
```




Сравнение методов сортировки на отсортированных данных
Методы
bubble
select
insert
merge
quick
Время выполнения ms
0
500
1000
1500
2000
0
2000
4000
6000
8000
10000
Размер массива n


Сравнение методов сортировки на реверсных данных
Методы
bubble
select
insert
merge
quick
Время выполнения ms
0
1000
2000
3000
4000
5000
0
2000
4000
6000
8000
10000
Размер массива n



Анализ эффективности алгоритмов сортировки

1. Эффективность алгоритмов для разных типов данных

Алгоритм	**Сложность (средняя)**	**Сложность (худшая)**	**Подходит для**	**Примечания**
Bubble Sort	$O(n^2)$	$O(n^2)$	Малые массивы, почти отсортированные	Прост в реализации, но крайне неэффективен на больших данных
Selection Sort	$O(n^2)$	$O(n^2)$	Малые массивы	Количество сравнений не зависит от входных данных
Insertion Sort	$O(n^2)$	$O(n^2)$	Почти отсортированные, малые массивы	Очень эффективен на почти отсортированных данных
Merge Sort	O(n log n)	O(n log n)	Любые типы данных	Стабильный, требует дополнительную память
Quick Sort	O(n log n)	$O(n^2)$	Случайные и почти случайные данные	Быстрее Merge Sort на практике, но чувствителен к выбору опорного элемента

2. Поведение алгоритмов на краевых случаях

Некоторые алгоритмы сильно зависят от порядка элементов во входном массиве. Ниже рассмотрено, как они ведут себя в лучших и худших случаях.

Bubble Sort

- **Лучший случай:** массив уже отсортирован — O(n) (если реализована проверка флага обмена).
  Пример: `[1, 2, 3, 4, 5]`
- **Худший случай:** массив отсортирован в обратном порядке — $O(n^2)$.
  Пример: `[5, 4, 3, 2, 1]`

Selection Sort

- Выполняет одинаковое количество сравнений в любом случае → **$O(n^2)$**.
  Пример: `[1, 2, 3, 4, 5]`, `[5, 4, 3, 2, 1]`, `[3, 1, 4, 2, 5]` — одинаковое поведение.
  Разница может быть только в количестве обменов, но не в общем времени.

Insertion Sort

- **Лучший случай:** массив отсортирован — O(n).
  Пример: `[1, 2, 3, 4, 5]`
- **Худший случай:** массив отсортирован в обратном порядке — $O(n^2)$.
  Пример: `[5, 4, 3, 2, 1]`
- **Почти отсортированный массив (например, 95% упорядочено):** работает близко к линейному времени.

Quick Sort

- **Лучший случай:** массив случайный или опорный элемент выбран удачно (делит массив примерно пополам) — O(n log n).
  Пример: `[8, 3, 1, 9, 6, 2, 10, 4]`
- **Худший случай:** массив отсортирован или обратно отсортирован при выборе первого/последнего элемента в качестве опорного — $O(n^2)$.
  Пример: `[1, 2, 3, 4, 5, 6, 7]`
  (для классического выбора pivot = последний элемент)

Merge Sort

- Не зависит от порядка входных данных.
  Всегда работает за **O(n log n)**.
  Пример: `[1, 2, 3, 4, 5]`, `[5, 4, 3, 2, 1]` — одинаковая производительность.

---

3. Итоги

Тип данных	**Наиболее эффективный алгоритм**
Случайный массив	Quick Sort
Уже отсортированный	Insertion Sort
Отсортированный в обратном порядке	Merge Sort / Quick Sort (с рандомизацией pivot)
Почти отсортированный	Insertion Sort

Тип данных	Наиболее эффективный алгоритм
Малый объем данных	Insertion Sort / Selection Sort

Вывод

На практике **Quick Sort** обычно показывает лучшие результаты для случайных данных за счет эффективного разделения,
но при плохом выборе опорного элемента может деградировать до **$O(n^2)$**.
Для почти отсортированных массивов предпочтителен **Insertion Sort**, а для стабильной гарантии — **Merge Sort**.

Ответы на контрольные вопросы

1. Какие алгоритмы сортировки имеют сложность $O(n^2)$ в худшем случае, а какие — O(n log n)?

Алгоритмы со сложностью **$O(n^2)$** в худшем случае — это простые методы сортировки: **Bubble Sort**, **Selection Sort**, **Insertion Sort**, а также **Quick Sort**, если опорный элемент выбирается неудачно (например, при сортировке уже отсортированного массива).
Алгоритмы со сложностью **O(n log n)** в худшем и среднем случае — это **Merge Sort**, а также **Quick Sort** при хорошем выборе опорного элемента.

---

2. Почему сортировка вставками (Insertion Sort) эффективна для маленьких или почти отсортированных массивов?

Сортировка вставками работает, последовательно вставляя элементы на свои места в уже частично отсортированную последовательность.
Если массив **уже почти отсортирован**, количество необходимых перестановок минимально, и алгоритм выполняет близко к **O(n)** операций.
Для **малых массивов** накладные расходы на организацию сложных алгоритмов (например, рекурсии) не оправданы, поэтому Insertion Sort показывает высокую скорость благодаря своей простоте и локальности данных.

---

3. В чем разница между устойчивой (stable) и неустойчивой (unstable) сортировкой?

Устойчивая сортировка сохраняет относительный порядок элементов с одинаковыми ключами.
Неустойчивая сортировка может изменять этот порядок.

Пример:
Пусть есть два объекта с одинаковым значением ключа — `A(1)` и `B(1)`. После устойчивой сортировки они останутся в том же порядке, что и были (`A(1), B(1)`), а при неустойчивой — порядок может измениться (`B(1), A(1)`).

Примеры устойчивых алгоритмов: Insertion Sort, Merge Sort, Bubble Sort.
Примеры неустойчивых алгоритмов: Quick Sort, Selection Sort, Heap Sort.

---

4. Опишите принцип работы алгоритма быстрой сортировки (Quick Sort). Что такое "опорный элемент" и как его выбор влияет на производительность?

Quick Sort основан на принципе **разделяй и властвуй**:

1. Из массива выбирается **опорный элемент (pivot)**.
2. Массив делится на две части — элементы меньше опорного и элементы больше опорного.
3. Каждая часть рекурсивно сортируется тем же методом.
4. Результаты объединяются в один отсортированный массив.

Выбор опорного элемента сильно влияет на производительность.
Если pivot выбран так, что массив делится на две **равные части**, сложность будет **O(n log n)**.
Если же массив делится неравномерно (например, pivot — всегда минимальный или максимальный элемент), сложность ухудшается до **$O(n^2)$**.
Поэтому часто используют **рандомизацию pivot**.

---

5. Сортировка слиянием (Merge Sort) гарантирует время O(n log n), но требует дополнительной памяти. В каких ситуациях этот алгоритм предпочтительнее быстрой сортировки?

Merge Sort предпочтителен в ситуациях, когда:

- Необходима **устойчивость сортировки** (например, при сортировке записей с одинаковыми ключами).
- Требуется **предсказуемое время работы**, независимо от входных данных.
- Данные слишком **крупные** и хранятся **внешне** (на диске), где важна последовательность чтения.
- Нельзя рисковать деградацией производительности, как в Quick Sort.