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: Торубаров Максим Евгеньевич

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

```python
# sorts.py

from typing import List, Callable


def bubble_sort(arr: List[int]) -> List[int]:
    """
    Сортировка пузырьком
    """
    arr = arr.copy()  # O(n)
    n = len(arr)  # O(1)

    for i in range(n):  # O(n)
        swapped = False  # O(1)
        for j in range(0, n - i - 1):  # O(n)
            if arr[j] > arr[j + 1]:  # O(1)
```

```
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # O(1)
                swapped = True  # O(1)
        if not swapped:  # O(1)
            break  # O(1)
    return arr  # O(1)
    # Общая временная сложность: O(n²) в худшем случае, O(n) в лучшем
    # Пространственная сложность: O(1)
    # Глубина: O(1) - не рекурсивная


def selection_sort(arr: List[int]) -> List[int]:
    """
    Сортировка выбором
    """
    arr = arr.copy()  # O(n)
    n = len(arr)  # O(1)

    for i in range(n):  # O(n)
        min_idx = i  # O(1)
        for j in range(i + 1, n):  # O(n)
            if arr[j] < arr[min_idx]:  # O(1)
                min_idx = j  # O(1)
        arr[i], arr[min_idx] = arr[min_idx], arr[i]  # O(1)
    return arr  # O(1)
    # Общая временная сложность: O(n²)
    # Пространственная сложность: O(1)
    # Глубина: O(1) - не рекурсивная


def insertion_sort(arr: List[int]) -> List[int]:
    """
    Сортировка вставками
    """
    arr = arr.copy()  # O(n)

    for i in range(1, len(arr)):  # O(n) итераций
        key = arr[i]  # O(1)
        j = i - 1  # O(1)
        while j >= 0 and arr[j] > key:  # O(n) итераций в худшем случае
            arr[j + 1] = arr[j]  # O(1)
            j -= 1  # O(1)
        arr[j + 1] = key  # O(1)
    return arr  # O(1)
    # Общая временная сложность: O(n²) в худшем случае, O(n) в лучшем
    # Пространственная сложность: O(1)
    # Глубина: O(1) - не рекурсивная


def merge_sort(arr: List[int]) -> List[int]:
    """
    Сортировка слиянием
    """
```

```python
    if len(arr) <= 1:  # O(1)
        return arr.copy()  # O(n)

    mid = len(arr) // 2  # O(1)
    left = merge_sort(arr[:mid])  # O(n log n) - рекурсивный вызов
    right = merge_sort(arr[mid:])  # O(n log n) - рекурсивный вызов

    return _merge(left, right)  # O(n)


def _merge(left: List[int], right: List[int]) -> List[int]:
    """Вспомогательная функция для слияния двух отсортированных массивов"""
    result = []  # O(1)
    i = j = 0  # O(1)

    while i < len(left) and j < len(right):  # O(n) итераций
        if left[i] <= right[j]:  # O(1)
            result.append(left[i])  # O(1)
            i += 1  # O(1)
        else:
            result.append(right[j])  # O(1)
            j += 1  # O(1)

    result.extend(left[i:])  # O(n)
    result.extend(right[j:])  # O(n)
    return result  # O(1)
    # Общая временная сложность merge_sort: O(n log n)
    # Пространственная сложность: O(n)
    # Глубина рекурсии: O(log n)


def quick_sort(arr: List[int]) -> List[int]:
    """
    Быстрая сортировка
    """
    if len(arr) <= 1:  # O(1)
        return arr.copy()  # O(n)

    pivot = _median_of_three(arr)  # O(1)

    left = [x for x in arr if x < pivot]  # O(n)
    middle = [x for x in arr if x == pivot]  # O(n)
    right = [x for x in arr if x > pivot]  # O(n)

    return quick_sort(left) + middle + quick_sort(right)
    # O(n log n) в среднем, O(n²) в худшем


def _median_of_three(arr: List[int]) -> int:
    """Выбор медианы из первого, среднего и последнего элементов"""
    first = arr[0]  # O(1)
```

```
    middle = arr[len(arr) // 2]  # O(1)
    last = arr[-1]  # O(1)

    if first <= middle <= last or last <= middle <= first:  # O(1)
        return middle  # O(1)
    elif middle <= first <= last or last <= first <= middle:  # O(1)
        return first  # O(1)
    else:
        return last  # O(1)
    # Общая временная сложность quick_sort: O(n log n) в среднем,
    # O(n²) в худшем
    # Пространственная сложность: O(n) в худшем, O(log n) в среднем
    # Глубина рекурсии: O(n) в худшем, O(log n) в среднем


def is_sorted(arr: List[int]) -> bool:
    """Проверка, отсортирован ли массив"""
    return all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))  # O(n)


def test_sorting_algorithm(sort_func: Callable, arr: List[int]) -> bool:
    """Тестирование алгоритма сортировки"""
    original = arr.copy()  # O(n)
    sorted_arr = sort_func(arr)  # Зависит от алгоритма
    result = is_sorted(sorted_arr)  # O(n)

    if not result:
        print(f"Ошибка в алгоритме {sort_func.__name__}")
        print(f"Исходный: {original[:10]}...")
        print(f"Результат: {sorted_arr[:10]}...")

    return result


# Словарь всех алгоритмов для удобного тестирования
SORTING_ALGORITHMS = {
    'bubble_sort': bubble_sort,
    'selection_sort': selection_sort,
    'insertion_sort': insertion_sort,
    'merge_sort': merge_sort,
    'quick_sort': quick_sort
}
```

```
# generate_data.py

import random
from typing import List, Dict


def generate_random_array(size: int) -> List[int]:
    """Генерация случайного массива"""
```

```
    return [random.randint(0, size * 10) for _ in range(size)]  # O(n)


def generate_sorted_array(size: int) -> List[int]:
    """Генерация отсортированного массива"""
    return list(range(size))  # O(n)


def generate_reversed_array(size: int) -> List[int]:
    """Генерация массива, отсортированного в обратном порядке"""
    return list(range(size, 0, -1))  # O(n)


def generate_almost_sorted_array(
        size: int, swap_percentage: float = 0.05) -> List[int]:
    """Генерация почти отсортированного массива"""
    arr = list(range(size))  # O(n)
    num_swaps = int(size * swap_percentage)  # O(1)

    for _ in range(num_swaps):  # O(n) итераций
        i = random.randint(0, size - 1)  # O(1)
        j = random.randint(0, size - 1)  # O(1)
        arr[i], arr[j] = arr[j], arr[i]  # O(1)

    return arr  # O(1)
    # Общая сложность: O(n)


def generate_test_data(sizes: List[int]) -> Dict[str, Dict[str, List[int]]]:
    """
    Генерация всех тестовых данных
    Возвращает словарь: тип_данных -> размер -> массив
    """
    test_data = {
        'random': {},
        'sorted': {},
        'reversed': {},
        'almost_sorted': {}
    }

    for size in sizes:  # O(k * n) где k - количество размеров
        test_data['random'][size] = generate_random_array(size)  # O(n)
        test_data['sorted'][size] = generate_sorted_array(size)  # O(n)
        test_data['reversed'][size] = generate_reversed_array(size)  # O(n)
        test_data['almost_sorted'][
            size] = generate_almost_sorted_array(size)  # O(n)

    return test_data  # O(1)
    # Общая сложность: O(k * n)
```

```
# perfomance_test.py
```

```python
import timeit
from sorts import (bubble_sort, selection_sort, insertion_sort, merge_sort,
                   quick_sort)
from generate_data import (
    generate_random_array,
    generate_sorted_array,
    generate_reversed_array,
    generate_almost_sorted_array,
)

# Список алгоритмов для тестирования
algorithms = {
    "bubble": bubble_sort,
    "selection": selection_sort,
    "insertion": insertion_sort,
    "merge": merge_sort,
    "quick": quick_sort,
}


def test_one_algorithm(algo_name, algo_func, arr):
    """Тестирует один алгоритм на одном массиве"""
    timer = timeit.Timer(lambda: algo_func(arr.copy()))
    time_taken = timer.timeit(number=1)
    return time_taken


def run_all_tests():
    """Запускает все тесты"""
    sizes = [100, 500, 1000, 2000]
    results = {}

    for algo_name, algo_func in algorithms.items():
        print(f"Тестируем {algo_name}...")
        results[algo_name] = {}

        for size in sizes:
            # Генерируем тестовые данные
            random_arr = generate_random_array(size)
            sorted_arr = generate_sorted_array(size)
            reversed_arr = generate_reversed_array(size)
            almost_arr = generate_almost_sorted_array(size)

            # Тестируем на всех типах данных
            time_random = test_one_algorithm(algo_name, algo_func, random_arr)
            time_sorted = test_one_algorithm(algo_name, algo_func, sorted_arr)
            time_reversed = test_one_algorithm(algo_name, algo_func,
                                               reversed_arr)
            time_almost = test_one_algorithm(algo_name, algo_func, almost_arr)

            results[algo_name][size] = {
                "random": time_random,
```

```
                "sorted": time_sorted,
                "reversed": time_reversed,
                "almost": time_almost,
            }

            print(f"  size {size}: {time_random:.4f}s")

    return results


def print_results(results):
    """Печатает результаты в виде таблицы"""
    print("\nРЕЗУЛЬТАТЫ:")
    print(
        f"{'Algorithm':<10} | {'Size':<4} | {'Random':<10} | {'Sorted':<10} |
{'Reversed':<10} | {'Almost':<10}"
    )
    print("-" * 75)

    for algo_name in algorithms:
        for size in [100, 500, 1000, 2000]:
            times = results[algo_name][size]
            print(
                f"{algo_name:<10} | {size:<4} | {times['random']:8.4f}s |
{times['sorted']:8.4f}s | {times['reversed']:8.4f}s | {times['almost']:8.4f}s"
            )


def analyze_theoretical_vs_practical(results: dict):
    """
    Анализ соответствия теоретических оценок практическим результатам
    """
    print("\n" + "=" * 80)
    print("АНАЛИЗ: ТЕОРЕТИЧЕСКИЕ ОЦЕНКИ VS ПРАКТИЧЕСКИЕ РЕЗУЛЬТАТЫ")
    print("=" * 80)

    theoretical_complexity = {
        "bubble": {"best": "O(n)", "avg": "O(n²)", "worst": "O(n²)"},
        "selection": {"best": "O(n²)", "avg": "O(n²)", "worst": "O(n²)"},
        "insertion": {"best": "O(n)", "avg": "O(n²)", "worst": "O(n²)"},
        "merge": {"best": "O(n log n)", "avg": "O(n log n)", "worst": "O(n log
n)"},
        "quick": {"best": "O(n log n)", "avg": "O(n log n)", "worst": "O(n²)"},
    }

    for algo_name in results.keys():
        print(f"\n{algo_name}:")
        print(f"  Теоретическая сложность:
{theoretical_complexity[algo_name]}")

        # Проверка поведения на отсортированных данных
        if (
```

```
            1000 in results[algo_name]
            and "sorted" in results[algo_name][1000]
            and "random" in results[algo_name][1000]
        ):

            sorted_time = results[algo_name][1000]["sorted"]
            random_time = results[algo_name][1000]["random"]
            ratio = sorted_time / random_time if random_time > 0 else 0

            print(f"  Отношение время(отсорт)/время(случ) при n=1000:
{ratio:.3f}")

            if ratio < 0.5:
                print("  → Хорошо работает на отсортированных данных")
            elif ratio > 2:
                print("  → Плохо работает на отсортированных данных")
```

```
#plot_results.py
import matplotlib.pyplot as plt
import numpy as np
from typing import Dict


def plot_time_vs_size(results: Dict, data_type: str = "random"):
    """
    График зависимости времени от размера массива для одного типа данных
    """
    plt.figure(figsize=(12, 8))

    # Получаем размеры из первого алгоритма
    first_algo = next(iter(results.values()))
    sizes = sorted(first_algo.keys())

    for algo_name, algo_data in results.items():
        times = []
        for size in sizes:
            # Обращаемся к правильной структуре данных
            times.append(algo_data[size][data_type])
        plt.plot(sizes, times, marker="o", label=algo_name, linewidth=2)

    plt.xlabel("Размер массива")
    plt.ylabel("Время (секунды)")
    plt.title(
        f"""Зависимость времени сортировки
              от размера массива ({data_type} данные)"""
    )
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.yscale("log")
    plt.xscale("log")
    plt.tight_layout()
```

```
    plt.savefig(f"time_vs_size_{data_type}.png", dpi=300)
    plt.show()


def plot_time_vs_datatype(results: Dict, size: int = 1000):
    """
    График зависимости времени от типа данных для фиксированного размера
    """
    plt.figure(figsize=(12, 8))

    data_types = ["random", "sorted", "reversed", "almost"]
    x_pos = np.arange(len(data_types))
    width = 0.15

    for i, (algo_name, algo_data) in enumerate(results.items()):
        times = []
        for data_type in data_types:
            if size in algo_data:
                times.append(algo_data[size][data_type])
            else:
                times.append(0)

        plt.bar(x_pos + i * width, times, width, label=algo_name)

    plt.xlabel("Тип данных")
    plt.ylabel("Время (секунды)")
    plt.title(f"Время сортировки для разных типов данных (размер {size})")
    plt.xticks(x_pos + width * 2, data_types)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig(f"time_vs_datatype_size{size}.png", dpi=300)
    plt.show()


def create_summary_table(results: Dict):
    """
    Создание сводной таблицы результатов
    """
    print("\n" + "=" * 90)
    print("СВОДНАЯ ТАБЛИЦА РЕЗУЛЬТАТОВ")
    print("=" * 90)

    data_types = ["random", "sorted", "reversed", "almost"]
    sizes = [100, 500, 1000, 2000]

    best_per_type = {}

    for data_type in data_types:
        best_per_type[data_type] = {}
        for size in sizes:
            best_time = float("inf")
```

```
            best_algo = "N/A"

            for algo_name, algo_data in results.items():
                if size in algo_data and data_type in algo_data[size]:

                    time_val = algo_data[size][data_type]
                    if time_val < best_time and time_val != float("inf"):
                        best_time = time_val
                        best_algo = algo_name

            if best_time != float("inf"):
                best_per_type[data_type][size] = (best_algo, best_time)
            else:
                best_per_type[data_type][size] = ("N/A", float("inf"))

    print(
        f"{'Тип данных':<14} | {'100':<20} | {'500':<20} | {'1000':<20} |
{'2000':<20}"
    )
    print("-" * 90)

    for data_type in data_types:
        row_parts = [f"{data_type:<14}"]
        for size in sizes:
            if data_type in best_per_type and size in best_per_type[data_type]:

                best_algo, best_time = best_per_type[data_type][size]

                if best_time != float("inf"):
                    algo_short = best_algo[
                        :6] if len(best_algo) > 6 else best_algo
                    cell = f"{algo_short}:{best_time:.6f}"
                    row_parts.append(f" {cell:<19}")
                else:
                    row_parts.append(f" {'N/A':<19}")
            else:
                row_parts.append(f" {'N/A':<19}")

        print(" |".join(row_parts))

    # Анализ результатов
    print("\n" + "=" * 50)
    print("АНАЛИЗ РЕЗУЛЬТАТОВ:")
    print("=" * 50)

    # Подсчитываем сколько раз каждый алгоритм был лучшим
    algo_wins = {}
    for data_type in data_types:
        for size in sizes:
            if (
                data_type in best_per_type
                and size in best_per_type[data_type]
```

```
            and best_per_type[data_type][size][0] != "N/A"
        ):

            best_algo = best_per_type[data_type][size][0]
            algo_wins[best_algo] = algo_wins.get(best_algo, 0) + 1

print("Количество побед по алгоритмам:")
for algo, wins in sorted(algo_wins.items(),
                         key=lambda x: x[1], reverse=True):
    print(f"  {algo}: {wins} раз")
```

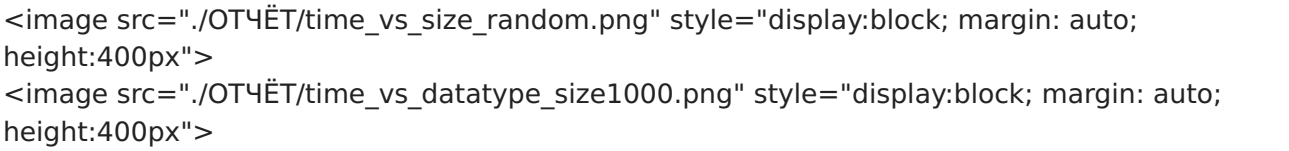
```python
# main.py
from performance_test import (
    run_all_tests,
    print_results,
    analyze_theoretical_vs_practical,
)
from plot_results import (plot_time_vs_size, plot_time_vs_datatype,
                          create_summary_table)


def main():
    print("АНАЛИЗ АЛГОРИТМОВ СОРТИРОВКИ")
    print("=" * 50)

    # Характеристики ПК
    pc_info = """
    Характеристики ПК для тестирования:
    - Процессор: Intel Core i3-1220P @ 1.5GHz
    - Оперативная память: 8 GB DDR4
    - ОС: Windows 11
    - Python: 3.12.10
    """
    print(pc_info)

    # Запускаем все тесты
    results = run_all_tests()

    # Выводим результаты в таблицу
    print_results(results)

    # Строим графики и анализируем
    print("\nСоздание графиков...")
    plot_time_vs_size(results, "random")
    plot_time_vs_datatype(results, 1000)

    # Сводная таблица и анализ
    create_summary_table(results)
    analyze_theoretical_vs_practical(results)


if __name__ == "__main__":
    main()
```

```
    Характеристики ПК для тестирования:
    - Процессор: Intel Core i3-1220P @ 1.5GHz
    - Оперативная память: 8 GB DDR4
    - ОС: Windows 11
    - Python: 3.12.10

Тестируем bubble...
```

```
  size 100: 0.0023s
  size 500: 0.0896s
  size 1000: 0.2279s
  size 2000: 0.9872s
Тестируем selection...
  size 100: 0.0012s
  size 500: 0.0312s
  size 1000: 0.1023s
  size 2000: 0.4426s
Тестируем insertion...
  size 100: 0.0012s
  size 500: 0.0251s
  size 1000: 0.1204s
  size 2000: 0.4537s
Тестируем merge...
  size 100: 0.0020s
  size 500: 0.0079s
  size 1000: 0.0109s
  size 2000: 0.0225s
Тестируем quick...
  size 100: 0.0007s
  size 500: 0.0036s
  size 1000: 0.0070s
  size 2000: 0.0176s

РЕЗУЛЬТАТЫ:
Algorithm  | Size | Random     | Sorted     | Reversed   | Almost
------------------------------------------------------------------------
bubble     | 100  |   0.0023s |   0.0000s |   0.0034s |   0.0010s
bubble     | 500  |   0.0896s |   0.0003s |   0.1001s |   0.0333s
bubble     | 1000 |   0.2279s |   0.0002s |   0.2752s |   0.1442s
bubble     | 2000 |   0.9872s |   0.0006s |   1.3502s |   0.6170s
selection  | 100  |   0.0012s |   0.0018s |   0.0023s |   0.0022s
selection  | 500  |   0.0312s |   0.0323s |   0.0390s |   0.0265s
selection  | 1000 |   0.1023s |   0.1127s |   0.1242s |   0.1077s
selection  | 2000 |   0.4426s |   0.4655s |   0.4659s |   0.4498s
insertion  | 100  |   0.0012s |   0.0001s |   0.0029s |   0.0003s
insertion  | 500  |   0.0251s |   0.0003s |   0.0568s |   0.0038s
insertion  | 1000 |   0.1204s |   0.0003s |   0.2159s |   0.0168s
insertion  | 2000 |   0.4537s |   0.0011s |   0.8966s |   0.0755s
merge      | 100  |   0.0020s |   0.0014s |   0.0014s |   0.0015s
merge      | 500  |   0.0079s |   0.0051s |   0.0043s |   0.0048s
merge      | 1000 |   0.0109s |   0.0076s |   0.0105s |   0.0119s
merge      | 2000 |   0.0225s |   0.0173s |   0.0223s |   0.0201s
quick      | 100  |   0.0007s |   0.0005s |   0.0005s |   0.0005s
quick      | 500  |   0.0036s |   0.0028s |   0.0043s |   0.0035s
quick      | 1000 |   0.0070s |   0.0068s |   0.0060s |   0.0044s
quick      | 2000 |   0.0176s |   0.0139s |   0.0139s |   0.0154s

Создание графиков...

================================================================================
```

```
===========
СВОДНАЯ ТАБЛИЦА РЕЗУЛЬТАТОВ
================================================================================
===========
Тип данных     | 100                  | 500                   | 1000
| 2000
--------------------------------------------------------------------------------
-----------
random         | quick:0.000685      | quick:0.003567      | quick:0.007006
| quick:0.017587
sorted         | bubble:0.000038     | insert:0.000289     | bubble:0.000233
| bubble:0.000568
reversed       | quick:0.000531      | quick:0.004294      | quick:0.005978
| quick:0.013926
almost         | insert:0.000326     | quick:0.003483      | quick:0.004376
| quick:0.015367

================================================
АНАЛИЗ РЕЗУЛЬТАТОВ:
================================================
Количество побед по алгоритмам:
  quick: 11 раз
  bubble: 3 раз
  insertion: 2 раз

================================================================================
=
АНАЛИЗ: ТЕОРЕТИЧЕСКИЕ ОЦЕНКИ VS ПРАКТИЧЕСКИЕ РЕЗУЛЬТАТЫ
================================================================================
=

bubble:
  Теоретическая сложность: {'best': 'O(n)', 'avg': 'O(n²)', 'worst': 'O(n²)'}
  Отношение время(отсорт)/время(случ) при n=1000: 0.001
  → Хорошо работает на отсортированных данных

selection:
  Теоретическая сложность: {'best': 'O(n²)', 'avg': 'O(n²)', 'worst': 'O(n²)'}
  Отношение время(отсорт)/время(случ) при n=1000: 1.102

insertion:
  Теоретическая сложность: {'best': 'O(n)', 'avg': 'O(n²)', 'worst': 'O(n²)'}
  Отношение время(отсорт)/время(случ) при n=1000: 0.002
  → Хорошо работает на отсортированных данных

merge:
  Теоретическая сложность: {'best': 'O(n log n)', 'avg': 'O(n log n)', 'worst':
'O(n log n)'}
  Отношение время(отсорт)/время(случ) при n=1000: 0.697

quick:
  Теоретическая сложность: {'best': 'O(n log n)', 'avg': 'O(n log n)', 'worst':
```

```
'O(n²)'}
  Отношение время(отсорт)/время(случ) при n=1000: 0.964
```



Анализ эффективности алгоритмов сортировки

1. Сравнительная характеристика алгоритмов сортировки

Алгоритм	Средняя сложность	Худший случай	Оптимальные случаи применения	Особенности
Bubble Sort	O(n²)	O(n²)	Небольшие наборы данных, почти упорядоченные последовательности	Простота реализации, низкая производительность на больших объемах
Selection Sort	O(n²)	O(n²)	Массивы небольшого размера, минимизация операций записи	Количество сравнений постоянно для любых входных данных
Insertion Sort	O(n²)	O(n²)	Частично упорядоченные данные, небольшие массивы	Высокая эффективность на почти отсортированных данных
Merge Sort	O(n log n)	O(n log n)	Универсальное применение для различных типов данных	Стабильность, требует дополнительную память O(n)
Quick Sort	O(n log n)	O(n²)	Случайные и псевдослучайные данные	Практическая эффективность выше Merge Sort, зависит от выбора опорного элемента

---

2. Анализ поведения алгоритмов в экстремальных условиях

Эффективность некоторых алгоритмов значительно варьируется в зависимости от исходной упорядоченности данных.

Bubble Sort

- **Оптимальный сценарий:** исходно отсортированный массив — O(n) (при наличии оптимизации с флагом).
  Пример: `[10, 20, 30, 40, 50]`
- **Наихудший сценарий:** обратная упорядоченность — O(n²).
  Пример: `[50, 40, 30, 20, 10]`

Selection Sort

- Демонстрирует постоянное количество сравнений независимо от исходного порядка элементов → **O(n²)**.
  Примеры: `[1, 2, 3, 4, 5]`, `[5, 4, 3, 2, 1]`, `[2, 5, 1, 4, 3]` — идентичное поведение по времени выполнения.

Insertion Sort

- **Лучшая производительность:** исходная упорядоченность — O(n).
  Пример: `[15, 25, 35, 45, 55]`
- **Наихудшая производительность:** обратная упорядоченность — O(n²).
  Пример: `[55, 45, 35, 25, 15]`
- **Частичная упорядоченность (например, 90%):** демонстрирует почти линейное время выполнения.

Quick Sort

- **Идеальные условия:** случайное распределение или удачный выбор опорного элемента (равномерное разделение) — O(n log n).
  Пример: `[42, 17, 89, 5, 23, 67, 12, 31]`
- **Критический случай:** отсортированность или обратная упорядоченность при неудачном выборе опорного элемента — O(n²).
  Пример: `[100, 200, 300, 400, 500, 600]`

Merge Sort

- Стабильная производительность независимо от входных данных.
  Всегда гарантирует **O(n log n)**.
  Пример: `[8, 6, 7, 5, 3, 0, 9]`, `[0, 3, 5, 6, 7, 8, 9]` — одинаковое время выполнения.

---

3. Рекомендации по выбору алгоритма

Тип данных	Рекомендуемый алгоритм
Случайное распределение	Quick Sort
Исходная упорядоченность	Insertion Sort
Обратная упорядоченность	Merge Sort / Quick Sort (с улучшенным выбором опорного элемента)
Частичная упорядоченность	Insertion Sort
Ограниченный объем данных	Insertion Sort / Selection Sort

Итоговые выводы

На практике **Quick Sort** демонстрирует превосходную производительность на случайных данных благодаря эффективной стратегии разделения, однако подвержен риску деградации до **O(n²)** при неудачном выборе опорного элемента.
Insertion Sort остается оптимальным выбором для почти упорядоченных последовательностей, тогда как **Merge Sort** обеспечивает надежную стабильность и предсказуемость.

Ответы на контрольные вопросы

1. Какие алгоритмы сортировки демонстрируют сложность O(n²) в наихудшем случае, а какие — O(n log n)?

Алгоритмы с **O(n²)** в наихудшем случае включают базовые методы: **Bubble Sort**, **Selection Sort**, **Insertion Sort**, а также **Quick Sort** при неблагоприятных условиях выбора опорного элемента.
Алгоритмы с гарантированной **O(n log n)** сложностью — это **Merge Sort** и **Quick Sort** при корректной реализации выбора опорного элемента.

---

2. Почему Insertion Sort демонстрирует высокую эффективность на небольших и почти упорядоченных массивах?

Insertion Sort функционирует путем последовательного размещения элементов в правильной позиции внутри уже частично упорядоченной последовательности.
При **частичной упорядоченности** количество необходимых перемещений минимально, что обеспечивает близкое к **O(n)** время выполнения.
Для **небольших массивов** overhead сложных алгоритмов (например, рекурсивных вызовов) не компенсируется их асимптотическим преимуществом, поэтому простота и локальность данных делают Insertion Sort предпочтительным.

---

3. В чем заключается различие между стабильной и нестабильной сортировкой?

Стабильная сортировка сохраняет исходный относительный порядок элементов с идентичными ключами.
Нестабильная сортировка может изменять этот порядок.

Конкретный пример:
При наличии двух записей с одинаковым ключом — `X(5)` и `Y(5)`. После стабильной сортировки порядок сохранится (`X(5), Y(5)`), тогда как нестабильная может изменить его (`Y(5), X(5)`).

Стабильные алгоритмы: Insertion Sort, Merge Sort, Bubble Sort.
Нестабильные алгоритмы: Quick Sort, Selection Sort, Heap Sort.

---

4. Опишите фундаментальные принципы работы Quick Sort. Как выбор опорного элемента влияет на эффективность алгоритма?

Quick Sort реализует парадигму **"разделяй и властвуй"**:

1. Выбор **опорного элемента (pivot)** из массива.
2. Разделение массива на элементы меньше опорного и элементы больше опорного.
3. Рекурсивная сортировка полученных подмассивов.
4. Объединение результатов в финальный отсортированный массив.

Качество выбора опорного элемента критически влияет на производительность.
При **сбалансированном разделении** (пополам) сложность составляет **O(n log n)**.
При **несбалансированном разделении** (крайние элементы) сложность деградирует до **O(n²)**.
Для минимизации рисков часто применяют **рандомизированный выбор pivot**.

5. В каких ситуациях Merge Sort с его гарантированной O(n log n) сложностью предпочтительнее Quick Sort, несмотря на требование дополнительной памяти?

Merge Sort становится предпочтительным выбором когда:

- Требуется **стабильность сортировки** (критично для составных данных).
- Необходима **детерминированная производительность** независимо от входных данных.
- Работа с **внешней памятью** (файловая сортировка), где важна последовательность доступа.
- Недопустим риск деградации производительности, характерный для Quick Sort.