Отчет по лабораторной работе 1

Введение в алгоритмы. Сложность. Поиск.

Дата: 2025-09-26 **Семестр:** 3 курс 5 семестр **Группа:** ПИЖ-б-о-23-2(2) **Дисциплина:** Анализ сложности алгоритмов **Студент:** мальцев виталий Игоревич

Цель работы

Цель работы: Освоить понятие вычислительной сложности алгоритма. Получить практические навыки реализации и анализа линейного и бинарного поиска. Научиться экспериментально подтверждать теоретические оценки сложности O(n) и O(log n)

Практическая часть

Выполненные задачи

- ☐ Задача 1: Реализовать функцию линейного поиска элемента в массиве.
- ☐ Задача 2: Реализовать функцию бинарного поиска элемента в отсортированном массиве.
- ☐ Задача 3: Провести теоретический анализ сложности обоих алгоритмов.
- ☐ Задача 4: Экспериментально сравнить время выполнения алгоритмов на массивах разного размера.
- ☐ Задача 5: Визуализировать результаты, подтвердив асимптотику O(n) и O(log n).

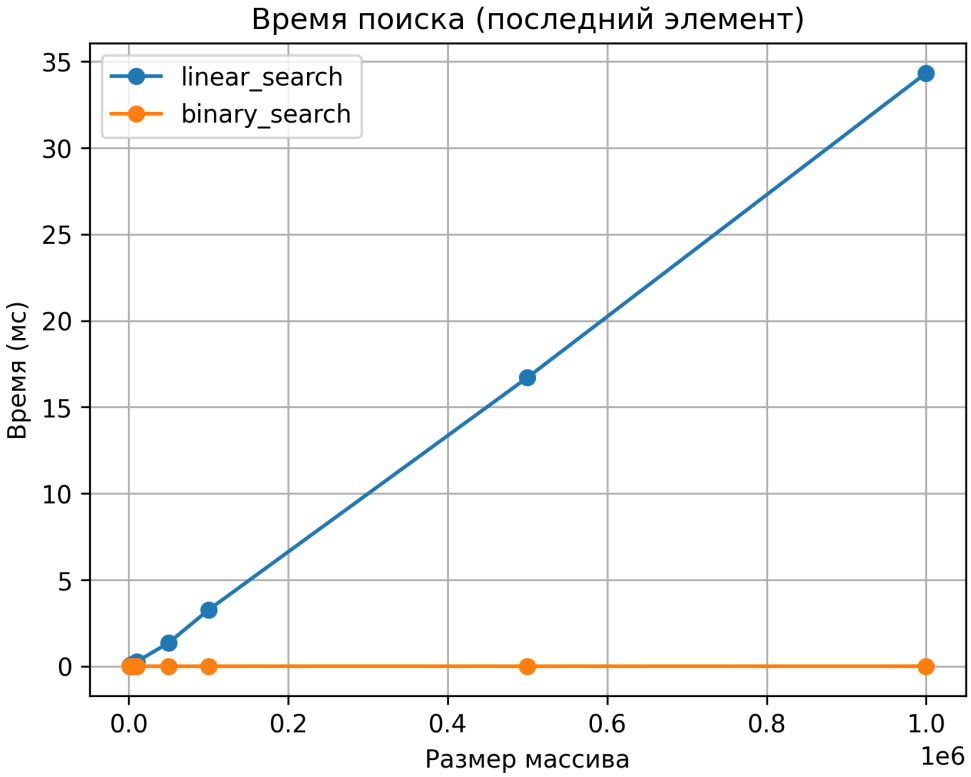
Ключевые фрагменты кода

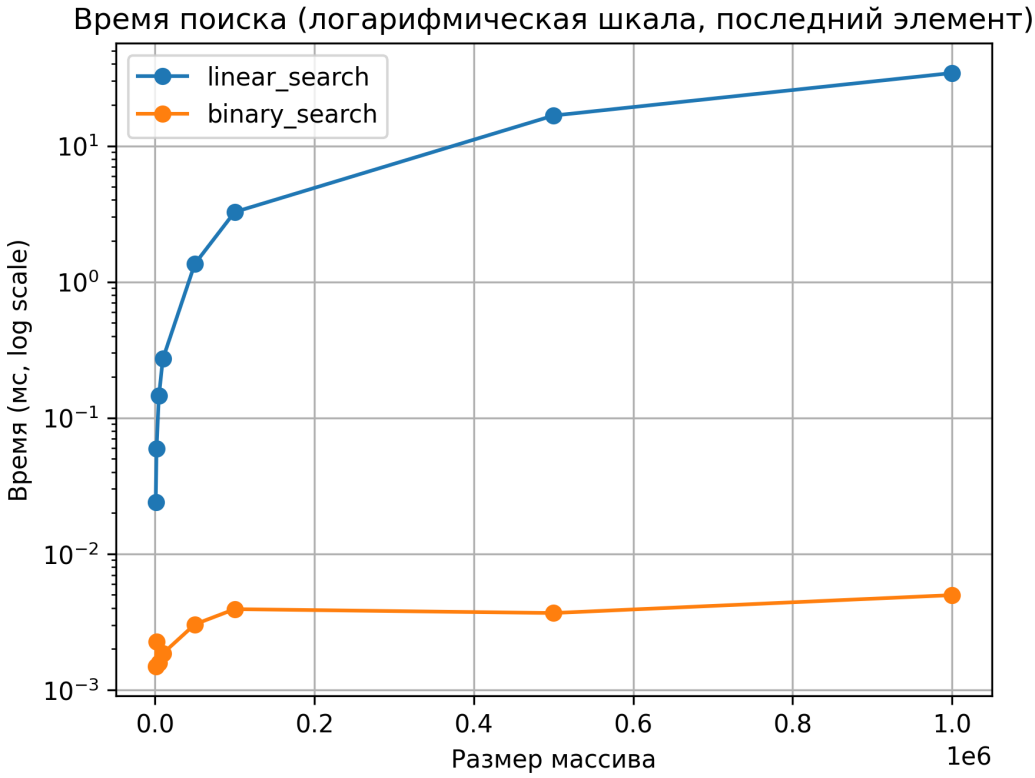
```
# Программирование на языке высокого уровня (Python).
# Задание № 01_lab01.
# Выполнил: Мальцев Виталий Игоревич
# Группа: ПИЖ-б-о-23-2(2)

# search_comparison.py

# Импорт необходимых библиотек
import matplotlib.pyplot as plt
import timeit


def linear_search(arr, target):
    """
    Линейный поиск элемента в массиве.
    Возвращает индекс target или -1, если не найден.
    Сложность: O(n), где n - длина массива.
    """
    for i in range(len(arr)):      # O(n) - проход по всем элементам
        if arr[i] == target:       # O(1) - сравнение
            return i               # O(1) - возврат индекса
```

```
    return -1                       # O(1) - если не найден
    # Общая сложность: O(n)


def binary_search(arr, target):
    """
    Бинарный поиск элемента в отсортированном массиве.
    Возвращает индекс target или -1, если не найден.
    Сложность: O(log n), где n - длина массива.
    """
    left = 0                        # O(1) - инициализация
    right = len(arr) - 1            # O(1) - инициализация
    while left <= right:            # O(log n) - деление диапазона
        mid = (left + right) // 2   # O(1) - вычисление середины
        if arr[mid] == target:      # O(1) - сравнение
            return mid              # O(1) - возврат индекса
        elif arr[mid] < target:     # O(1) - сравнение
            left = mid + 1          # O(1) - сдвиг границы
        else:
            right = mid - 1         # O(1) - сдвиг границы
    return -1                       # O(1) - если не найден
    # Общая сложность: O(log n)


sizes = [1000, 2000, 5000, 10000, 50000, 100000, 500000, 1000000]


def generate_test_data(sizes):
    """
    Генерирует отсортированные массивы заданных размеров и целевые элементы.
    Возвращает словарь: {size: {'array': [...], 'targets': {...}}}
    """
    data = {}
    for size in sizes:
        arr = list(range(size))
        targets = {
            'first': arr[0],
            'middle': arr[size // 2],
            'last': arr[-1],
            'absent': -1
        }
        data[size] = {'array': arr, 'targets': targets}
    return data


test_data = generate_test_data(sizes)


def measure_time(search_func, arr, target, repeat=10):
    """
    Замеряет среднее время выполнения функции поиска по массиву.
    Возвращает: float: Среднее время поиска (в миллисекундах)
    за repeat запусков.
    """
```

```
    times = []
    for _ in range(repeat):
        t = timeit.timeit(lambda: search_func(arr, target), number=1)
        times.append(t * 1000)
    return sum(times) / len(times)


# Новый способ хранения:
# для каждого алгоритма — словарь {size: [first, middle, last, absent]}
element_keys = ['first', 'middle', 'last', 'absent']
results_linear = {}
results_binary = {}
for size, info in test_data.items():
    arr = info['array']
    targets = info['targets']
    # Сохраняем времена в фиксированном порядке: [first, middle, last, absent]
    results_linear[size] = [
        measure_time(linear_search, arr, targets['first']),
        measure_time(linear_search, arr, targets['middle']),
        measure_time(linear_search, arr, targets['last']),
        measure_time(linear_search, arr, targets['absent'])
    ]
    results_binary[size] = [
        measure_time(binary_search, arr, targets['first']),
        measure_time(binary_search, arr, targets['middle']),
        measure_time(binary_search, arr, targets['last']),
        measure_time(binary_search, arr, targets['absent'])
    ]


def plot_results(sizes, results_linear, results_binary):
    """
    Рисует графики в нормальном формате и в логарифмическом по оси y
    В результате работы функции сохраняются два изображения в рабочую
    директорию
    """
    y_linear = [results_linear[size][2] for size in sizes]  # last
    y_binary = [results_binary[size][2] for size in sizes]  # last
    plt.plot(sizes, y_linear, marker='o', label='linear_search')
    plt.plot(sizes, y_binary, marker='o', label='binary_search')
    plt.xlabel('Размер массива')
    plt.ylabel('Время (мс)')
    plt.title('Время поиска (последний элемент)')
    plt.legend()
    plt.grid(True)
    # plt.ticklabel_format(style='plain', axis='x')
    plt.savefig('./ОТЧЁТ/time_complexity_plot.png',
                dpi=300, bbox_inches='tight')
    plt.show()

    # log scale
    plt.plot(sizes, y_linear, marker='o', label='linear_search')
    plt.plot(sizes, y_binary, marker='o', label='binary_search')
    plt.xlabel('Размер массива')
```

```
    plt.ylabel('Время (мс, log scale)')
    plt.yscale('log')
    plt.title('Время поиска (логарифмическая шкала, последний элемент)')
    plt.legend()
    plt.grid(True)
    # plt.ticklabel_format(style='plain', axis='x')
    plt.savefig('./ОТЧЁТ/time_complexity_plot_log.png',
                dpi=300, bbox_inches='tight')
    plt.show()


plot_results(sizes, results_linear, results_binary)

# 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)
```






```
Характеристики ПК для тестирования:
- Процессор: Intel Core i5-12100f
- Оперативная память: 16 GB DDR4
- ОС: Windows 11
- Python: 3.12
```

Линейный поиск (linear_search): теоретически O(n), время растет линейно с размером массива. Практически: время поиска первого элемента минимально, последнего/отсутствующего — максимально, график близок к прямой. Для последнего элемента требуется n сравнений.
Бинарный поиск (binary_search): теоретически O(log n), время растет медленно, логарифмически. Практически: время почти не зависит от позиции элемента, график близок к логарифмической кривой. Для последнего элемента требуется log(n) сравнений.

Ответы на контрольные вопросы

1. Что такое асимптотическая сложность алгоритма и зачем она нужна?

Асимптотическая сложность — это способ оценки эффективности алгоритма в зависимости от размера входных данных. Она показывает, как быстро растёт время выполнения (или объём памяти) при увеличении количества элементов.
Нужна она для того, чтобы сравнивать алгоритмы независимо от конкретных компьютеров и измерять их производительность в общем виде, не проводя реальные тесты для всех возможных размеров данных.

---

2. Разница между O(1), O(n) и O(log n) с примерами

- **O(1)** — постоянное время: выполнение не зависит от размера данных.
  Пример: доступ к элементу массива по индексу.
- **O(n)** — линейное время: время выполнения растёт прямо пропорционально количеству элементов.
  Пример: линейный поиск элемента в неотсортированном массиве.
- **O(log n)** — логарифмическое время: время растёт медленно, при каждом шаге сокращается количество проверяемых данных.
  Пример: бинарный поиск в отсортированном массиве.

---

3. Отличие линейного поиска от бинарного и условия выполнения бинарного поиска

- **Линейный поиск** проверяет элементы последовательно — от начала до конца. Сложность O(n).
- **Бинарный поиск** делит отсортированный массив пополам, исключая половину элементов на каждом шаге. Сложность O(log n).

Условия для бинарного поиска:

1. Массив должен быть **отсортирован**.
2. Должна быть возможность **доступа по индексу** к элементам.

---

4. Почему практическое время выполнения может отличаться от O-большого

Асимптотическая оценка показывает поведение при **больших n** и не учитывает:

- константные факторы (например, оптимизация компилятора или кэширование);
- различия в архитектуре процессора;
- влияние среды выполнения (операционная система, фоновая нагрузка);
- конкретные входные данные (лучший, худший, средний случаи).

Из-за этого реальное время может отличаться, особенно при малых размерах входа.

---

5. Как экспериментально подтвердить, что сложность равна O(n) или O(log n)

План эксперимента:

1. **Выбрать алгоритм** (например, линейный или бинарный поиск).
2. **Сгенерировать тестовые данные** разных размеров: N = 100, 1000, 10000, 100000 и т.д.
3. **Измерить время выполнения** алгоритма для каждого размера.
4. **Построить график** зависимости времени от N.
5. **Сравнить форму графика**:
   - Прямая линия → O(n)
   - Кривая, растущая медленно (похожа на логарифм) → O(log n)
