Отчет по лабораторной работе 1

Введение в алгоритмы. Сложность. Поиск.

Дата: 2025-09-19
Семестр: 3 курс 1 полугодие - 5 семестр
Группа: ПИЖ-б-о-23-2
Дисциплина: Анализ сложности алгоритмов
Студент: Мальцев Виталий Игоревич

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)

Задание:

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

```
# Отчет по лабораторной работе 1
# [Название работы]

**Дата:** [YYYY-MM-DD]
**Семестр:** [Номер, например 3 курс 2 полугодие - 6 семестр]
**Группа:** [Номер группы]
**Дисциплина:** [Наименование]
**Студент:** [ФИО]

# 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': {...}}}
    Сложность: O(k*n), где k - количество размеров, n - размер массива.
    """
    data = {}
    for size in sizes:                # O(k)
        arr = list(range(size))       # O(n)
        targets = {
            'first': arr[0],              # O(1)
            'middle': arr[size // 2],     # O(1)
            'last': arr[-1],              # O(1)
            'absent': -1                  # O(1)
        }
        data[size] = {'array': arr, 'targets': targets}  # O(1)
    return data                       # O(1)
# Общая сложность: O(k*n)


test_data = generate_test_data(sizes)


def measure_time(search_func, arr, target, repeat=10):
    times = []
```

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


results = {
    'linear_search': {},
    'binary_search': {}
}
for size, info in test_data.items():
    arr = info['array']
    targets = info['targets']
    results['linear_search'][size] = {}
    results['binary_search'][size] = {}
    for key, target in targets.items():
        results['linear_search'][size][key] = measure_time(
            linear_search, arr, target)
        results['binary_search'][size][key] = measure_time(
            binary_search, arr, target)


def plot_results(results, sizes):
    plt.figure(figsize=(12, 6))
    for alg in ['linear_search', 'binary_search']:
        y = [results[alg][size]['last'] for size in sizes]
        plt.plot(sizes, y, marker='o', label=alg)
    plt.xlabel('Размер массива')
    plt.ylabel('Время (мс)')
    plt.title('Время поиска (последний элемент)')
    plt.legend()
    plt.grid(True)
    plt.savefig('time_complexity_plot.png', dpi=300, bbox_inches='tight')
    plt.show()

    plt.figure(figsize=(12, 6))
    for alg in ['linear_search', 'binary_search']:
        y = [results[alg][size]['last'] for size in sizes]
        plt.plot(sizes, y, marker='o', label=alg)
    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.savefig('time_complexity_plot_log.png', dpi=300, bbox_inches='tight')
    plt.show()

    # График в log-log масштабе
    plt.figure(figsize=(12, 6))
    for alg in ['linear_search', 'binary_search']:
        y = [results[alg][size]['last'] for size in sizes]
```

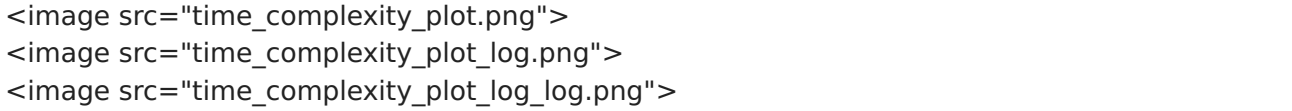
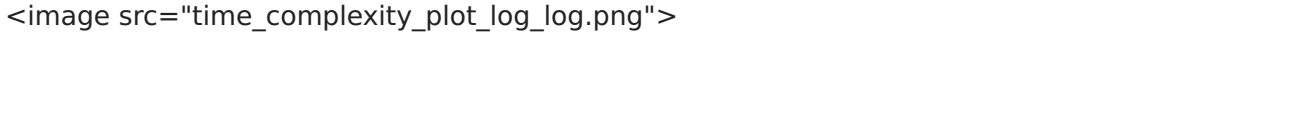
```
        plt.plot(sizes, y, marker='o', label=alg)
    plt.xlabel('Размер массива (log scale)')
    plt.ylabel('Время (мс, log scale)')
    plt.xscale('log')
    plt.yscale('log')
    plt.title('Время поиска (log-log scale, последний элемент)')
    plt.legend()
    plt.grid(True)
    plt.savefig('time_complexity_plot_log_log.png', dpi=300,
bbox_inches='tight')
    plt.show()


plot_results(results, sizes)


def print_table(alg_name, results, sizes, keys):
    print(f"\nТаблица результатов для {alg_name}:")
    header = "Размер ".ljust(10) + "| " + \
        " | ".join([k.ljust(10) for k in keys])
    print(header)
    print("-" * len(header))
    for size in sizes:
        row = str(size).ljust(10) + "| "
        row += " | ".join([f"{results[alg_name][size][key]:10.3f}" for key in
keys])
        print(row)


element_keys = ['first', 'middle', 'last', 'absent']
print_table('linear_search', results, sizes, element_keys)
print_table('binary_search', results, sizes, element_keys)

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

<div style="display:flex; justify-content:center;">
<image src="out.png">
</div>

Контрольные вопросы

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

Асимптотическая сложность алгоритма — это оценка роста времени выполнения или объёма памяти, требуемого алгоритмом, в зависимости от размера входных данных при стремлении этого размера к бесконечности. Обычно выражается с помощью нотации O («большое O»), Ω («омега») или Θ («тета»).

Зачем нужна:

- Позволяет сравнивать эффективность алгоритмов независимо от аппаратного обеспечения и языка программирования.
- Помогает понять, как алгоритм масштабируется с ростом входных данных.
- Упрощает анализ, игнорируя константы и младшие члены, что делает оценку универсальной для больших n.

---

2. Объясните разницу между O(1), O(n) и O(log n). Приведите примеры алгоритмов с такой сложностью.

- **O(1)** — константная сложность. Время выполнения не зависит от размера входных данных.

    - *Пример:* доступ к элементу массива по индексу, операция push/pop в стеке (при реализации на массиве).

- **O(n)** — линейная сложность. Время выполнения пропорционально размеру входных данных.

    - *Пример:* линейный поиск в неупорядоченном массиве, обход всех элементов списка.

- **O(log n)** — логарифмическая сложность. Время выполнения растёт пропорционально логарифму размера данных (обычно $\log_2 n$).

    - *Пример:* бинарный поиск в отсортированном массиве, поиск в сбалансированном бинарном дереве.

---

3. В чем основное отличие линейного поиска от бинарного? Какие предварительные условия необходимы для выполнения бинарного поиска?

Основное отличие:

- Линейный поиск последовательно проверяет каждый элемент до тех пор, пока не найдёт нужный или не дойдёт до конца. Сложность — O(n).
- Бинарный поиск работает только с отсортированными данными и на каждом шаге сокращает область поиска вдвое, используя сравнение со средним элементом. Сложность — O(log n).

Предварительные условия для бинарного поиска:

- Массив (или другая структура данных) должен быть **отсортирован** (по возрастанию или убыванию).
- Должна быть возможность **быстрого доступа к произвольному элементу** (например, массив или вектор, а не связный список без индексации).

---

4. Почему на практике время выполнения алгоритма может отличаться от теоретической оценки O-большое?

Теоретическая оценка O-большое:

- Игнорирует константы и младшие члены.
- Предполагает идеальные условия и асимптотическое поведение при $n \to \infty$.

Практические причины расхождения:

- **Константы:** Алгоритм O(n) с большой константой может работать медленнее, чем $O(n^2)$ с маленькой константой при малых n.
- **Ограничения памяти и кэширования:** Алгоритмы с хорошей локальностью данных могут работать быстрее из-за кэша CPU.
- **Накладные расходы:** Вызовы функций, выделение памяти, управление потоками могут замедлять работу.
- **Размер данных:** При малых n асимптотика не имеет значения — важны реальные коэффициенты.
- **Влияние ОС и других процессов:** Конкурентное использование ресурсов может влиять на время выполнения.

---

5. Как экспериментально подтвердить, что сложность алгоритма равна O(n) или O(log n)? Опишите план эксперимента.

План эксперимента:

Шаг 1: Подготовка

- Реализуйте алгоритм.
- Подготовьте набор тестовых данных разного размера: n_1, n_2, ..., n_k (например, 100, 1000, 10000, 100000).
- Для каждого размера n сгенерируйте несколько наборов данных (например, 10–100) и усредните время выполнения.

Шаг 2: Измерение времени

- Запустите алгоритм на каждом наборе данных.
- Измерьте время выполнения (в миллисекундах или наносекундах) с помощью таймера (например, `time` в Python, `System.nanoTime()` в Java).
- Запишите результаты в таблицу: `n | t(n)`.

Шаг 3: Анализ

- Постройте график: ось X — размер данных `n`, ось Y — время выполнения `t(n)`.
- Для O(n): график должен быть примерно линейным (прямая линия).
- Для O(log n): график должен расти очень медленно, почти горизонтально; можно построить график `t(n)` vs `log(n)` — он должен быть линейным.

Шаг 4: Проверка через отношение

- Рассчитайте отношение времени к теоретической сложности:
    - Для O(n): `t(n) / n` должно быть примерно постоянным.
    - Для O(log n): `t(n) / log(n)` должно быть примерно постоянным.
- Если значения стабильны — гипотеза подтверждается.

Шаг 5: Визуализация и вывод

- Постройте графики и выведите средние отношения.
- Сделайте вывод: если зависимость соответствует ожидаемой, то сложность подтверждена экспериментально.

> **Важно:** Исключите влияние внешних факторов (запускать на чистой системе, повторять измерения, использовать warm-up).