САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ
ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИФАКУЛЬТЕТ
ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе № 2

по курсу «Алгоритмы и структуры данных»

Тема: Алгоритмы сортировки и поиска.

Вариант 14

Выполнил:

Мурашов Никита Александрович

Группа К3141

Проверил:

Афанасьев А. В.

Санкт-Петербург2024 г.

Введение

Данная лабораторная работа включает реализацию и тестирование нескольких алгоритмов сортировки и поиска, используемых в информатике для решения различных задач. Основные цели работы заключаются в изучении и применении методов сортировки, поиска, а также в применении подхода "разделяй и властвуй" для эффективного решения задач.

Задачи по варианту

Задача 1: Сортировка слиянием

Реализация алгоритма сортировки слиянием, который рекурсивно делит массив на две половины и затем объединяет их в отсортированном порядке. Сортировка слиянием работает эффективно и имеет сложность $O(n\log n)$.

Код программы

```
def merge_sort(array, left, right):
    if left < right:
        middle = (left + right) // 2
        merge_sort(array, left, middle)
        merge_sort(array, middle + 1, right)
        merge(array, left, middle, right)
```

Тесты и анализ

Для тестирования использовались массивы разного размера и степени отсортированности. Алгоритм показал себя эффективным для всех протестированных массивов, корректно выполняя сортировку и обеспечивая стабильную производительность благодаря использованию рекурсии и подхода "разделяй и властвуй". Основное преимущество — это логарифмическая сложность слияния подмассивов.

```
def test_basic_sort(self):
    array = [12, 11, 13, 5, 6, 7]
    merge_sort(array, 0, len(array) - 1)
    self.assertEqual(array, [5, 6, 7, 11, 12, 13])
```

Задача 2: Сортировка слиянием с отслеживанием индексов

Расширенная версия сортировки слиянием, где дополнительно выводятся индексы и значения подмассивов на этапе слияния для отслеживания процесса выполнения алгоритма. Это позволяет понять, как именно происходят объединения и какая часть массива изменяется на каждом этапе.

Код программы

```
def merge_with_indices(array, left, middle, right):
    print(f"Слияние: индексы {left + 1}-{right + 1}, значения: {array[left]}-{array[right]}")
    # Слияние подмассивов
```

Тесты и анализ

Алгоритм был протестирован на массивах с различными элементами для отслеживания корректности индексации. Вывод индексов помогает визуализировать процесс слияния, что особенно полезно для понимания работы алгоритма и учебных целей. Тесты подтвердили корректность алгоритма для всех случаев.

```
def test_basic_sort_with_indices(self):
    array = [9, 7, 5, 8]
    merge_sort_with_indices(array, 0, len(array) - 1)
    self.assertEqual(array, [5, 7, 8, 9])
```

Задача 3: Подсчет инверсий в массиве

Алгоритм сортировки слиянием, модифицированный для подсчета количества инверсий в массиве. Инверсии — это пары элементов, которые стоят в неправильном порядке. Подсчет инверсий позволяет оценить, насколько сильно массив отличается от отсортированного.

Код программы

```
def merge_and_count(array, temp_array, left, middle, right):
    inv_count = 0
    if array[i] > array[j]:
        inv_count += (middle - i + 1)
    # Слияние подмассивов с подсчетом инверсий
```

Тесты и анализ

Были протестированы различные массивы: отсортированные, обратные и случайные. Алгоритм корректно подсчитал количество инверсий во всех случаях, подтверждая правильность своей работы. Подсчет инверсий важен для анализа перестановок и оптимизации сортировки.

```
def test_inversions(self):
    array = [1, 20, 6, 4, 5]
    temp_array = [0] * len(array)
    result = merge_sort_and_count(array, temp_array, 0, len(array) - 1)
    self.assertEqual(result, 5)
```

Задача 4: Бинарный поиск

Реализация бинарного поиска в отсортированном массиве для нахождения индекса заданного элемента или определения, что элемент отсутствует в массиве. Бинарный поиск работает за логарифмическое время $O(\log n)$ и является одним из самых быстрых методов поиска в отсортированных данных.

Код программы

```
def binary_search(array, target):
    left, right = 0, len(array) - 1
    while left <= right:
        middle = (left + right) // 2
        if array[middle] == target:
            return middle
```

Тесты и анализ

Алгоритм был протестирован на массивах различной длины и для разных значений целевых элементов. Бинарный поиск показал отличные результаты на отсортированных массивах, эффективно находя элементы или определяя их отсутствие. Это подтверждает высокую эффективность данного алгоритма для поиска в больших объемах данных.

```
def test_binary_search_found(self):
    array = [1, 2, 3, 4, 5]
    self.assertEqual(binary_search(array, 3), 2)
```

Задача 5: Элемент большинства

Алгоритм для нахождения элемента, который встречается более чем $n/2$ раз в массиве, используя метод Бойера-Мура. Этот метод работает за линейное время и использует константную память, что делает его очень эффективным для данной задачи.

Код программы

```
def majority_element(array):
    candidate = None
    count = 0
    for num in array:
        if count == 0:
            candidate = num
        count += 1 if num == candidate else -1
```

Тесты и анализ

Алгоритм был протестирован на массивах, содержащих элемент большинства, а также на массивах, где такого элемента нет. Во всех случаях алгоритм корректно находил элемент или возвращал $None$, если элемента большинства не было. Метод Бойера-Мура доказал свою эффективность за счет линейной сложности и минимальных требований к памяти.

```
def test_majority_element_exists(self):
    array = [2, 3, 9, 2, 2]
    self.assertEqual(majority_element(array), 2)
```

Заключение

В ходе выполнения лабораторной работы были реализованы и протестированы несколько классических алгоритмов сортировки и поиска. Каждый алгоритм был подробно рассмотрен, реализован на языке Python и протестирован с использованием модуля unittest. Были изучены их принципы работы, достоинства и недостатки, а также проведена оценка их временной сложности.

Основные выводы:

- Простые алгоритмы сортировки, такие как пузырьковая сортировка и сортировка выбором, просты в реализации, но неэффективны для больших массивов из-за квадратичной сложности.
- Сортировка вставками более эффективна на почти отсортированных массивах и позволяет отслеживать изменения позиций элементов.
- Линейный поиск прост в реализации, но неэффективен для больших массивов; для улучшения производительности стоит использовать более сложные алгоритмы поиска, такие как бинарный поиск.
- Тестирование алгоритмов позволяет убедиться в их корректности и выявить возможные ошибки на ранних этапах разработки.

Выполнение данной лабораторной работы способствовало углублению понимания основных алгоритмов сортировки и поиска, развитию навыков программирования на языке Python и умению проводить модульное тестирование программ.