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

Отчет по лабораторной работе №2
по курсу «Алгоритмы и структуры данных»
Тема: Сортировка слиянием. Метод декомпозиции.

Выполнил:
Мезенцев Богдан
К3139
Проверил:
Афанасьев Антон Владимирович

Санкт-Петербург
2024 г.

Содержание отчёта

Задачи по варианту

Задание 1. Сортировка слиянием

1) Для начала создаём функцию *merge_sort(A, n)*, в которую мы передаем два параметра(*A* - сортируемый массив, *n* - количество элементов массива).
   Далее стоит сделать проверку на длину массива, так как, если массив состоит из одного элемента, то он уже является отсортированным, в таком случае функция будет сразу возвращать массив.

```
def merge_sort(A, n):
    if len(A) > 1:
        mid = len(A) // 2
        left = A[:mid]
        right = A[mid:]
        merge_sort(left, n)
        merge_sort(right, n)
        # i - индекс в списке left
        # j - индекс в списке right
        # k - индекс в исходном списке A
        i = j = k = 0
```

   также мы находим элемента, который делит наш исходный массив на два подмассива (*left* и *right*). Так как метод сортировки слиянием заключается в рекурсивном разделении сортируемого массива на подмассивы, поэтому мы вызываем функцию *merge_sort(),* но аргументами уже будут подмассивы *left* и *right.* Еще нам необходимо создать переменные для индексов в массиве и подмассивах.

2) Далее рекурсивно сравниваем элементы отсортированных массивов и добавляем их в исходный массив *A*

```
while i < len(left) and j < len(right):
    if left[i] < right[j]:
        A[k] = left[i]
        i += 1
    else:
        A[k] = right[j]
        j += 1
    k += 1

while j < len(right): # если в правом массиве остались элементы, а в левом нет
    A[k] = right[j]
    j += 1
    k += 1

while i < len(left): # если в левом массиве остались элементы, а в правом нет
    A[k] = left[i]
    i += 1
    k += 1
```

также необходимо предусмотреть такой вариант, когда в одном из подмассивов элементов не останется. За это отвечают два последних цикла *while* см. скриншот

3) В результате функция возвращает массив *A*, отсортированный по возрастанию.

Тестирование:

На вход функции был дан массив состоящий из 20000 элементов, в диапазоне от 10**8 до 10**9, важно отметить, что массив был отсортирован в обратном порядке. Данный массив соответствует ограничениям входных данных.

```
A = [random.randint(10**8, 10**9) for _ in range(20000)]
A_reverse_sort = sorted(A, reverse=True)
```

Результаты модульного теста:

```
tests.py::AlgorithmsSortTestCase::test1_merge_sort PASSED
Время работы алгоритма_1: 0.016349 секунд
Использование памяти: 144.000000 Кб
```

Задание 3. Число инверсий

1) Для подсчета инверсий в не отсортированном массиве будем использовать всё тот же алгоритм сортировки слиянием и функцию *merge_sort()*, нам необходимо немного дополнить данный алгоритм

```
def merge_sort_and_count(A, n):
    if len(A) > 1:
        mid = len(A) // 2
        left = A[:mid]
        right = A[mid:]
        inv_count = merge_sort_and_count(left, n) + merge_sort_and_count(right, n)
        # i - индекс в списке left
        # j - индекс в списке right
        # k - индекс в исходном списке A
        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                A[k] = left[i]
                i += 1
            else:
                A[k] = right[j]
                j += 1
                inv_count += len(left) - i  # количество инверсий
            k += 1
```

Для подсчета инверсий создаем переменную *inv_count,* в которой будем рекурсивно вызывать функцию *merge_sort_and_count()*. Аргументами данной функции уже будут подмассивы *left* и *right*. Теперь при сравнении элементов подмассивов *left* и *right,* в случае, когда элемент из массива right будет меньше чем из массива left, то это будет являться инверсией.

2) В результате функция возвращает число инверсий в массиве. В случае, если массив состоит из одного элемента, функция возвращает значение 0.

Тестирование:

На вход функции был дан не отсортированный массив, который содержит 8 инверсий.

```
A = [5, 3, 2, 4, 1]  # неотсортированный массив, в котором 8 инверсий!!!
n = len(A)  # кол-во элементов массива A
```

Сам модульный тест:

```
result = merge_sort_and_count(A, n)
(self.assertEqual(result, 8))
```

Результаты модульного теста:

```
tests.py::AlgorithmsSortTestCase::test2_inversion_count PASSED
Время работы алгоритма_2: 0.000039 секунд
Использование памяти: 0.000000 Кб
```

Количество используемой памяти очень мало, поэтому имеет значение 0Кб!

Задание 4. Бинарный поиск

```python
def binary_search(value, lst):
    left = 0
    right = len(lst) - 1
    mid = len(lst) // 2

    while lst[mid] != value and left <= right:
        if value > lst[mid]:
            left = mid + 1
        else:
            right = mid - 1
        mid = (left + right) // 2

    if left > right:
        return -1
    else:
        return mid
```

1) Задаем переменные left (начало списка) и right (конец списка). mid устанавливается как середина списка.
2) цикл while продолжается до тех пор, пока значение в середине списка (lst[mid]) не совпадает с искомым (value) и индексы left не превышают right.
   - Если value больше, чем lst[mid], это означает, что искомое значение находится в правой части списка. Поэтому left смещается вправо (mid + 1).
   - Если value меньше, чем lst[mid], значение находится слева, и right смещается влево (mid - 1).
3) Обновляем индекс mid, после изменения границ left и right новая середина пересчитывается.
4) Если left превышает right, это означает, что значение не найдено, и функция возвращает -1. Если же значение найдено, функция возвращает индекс mid, где находится искомое значение.

Задание 5. Представитель большинства

Представителем большинства будет являться то число, которое встречается в массиве больше, чем *n/2* раз (*n* - количество элементов в массиве).

1) Сначала создадим функцию *element_of_majority(A)*, параметром которой будет массив, в котором ищем элемент.

```
def element_of_majority(A):
    A.sort()
    element = A[len(A) // 2]
    count = A.count(element)
    if count > len(A) // 2:
        return 1
    else:
        return 0
```

2) Заметим, что если в отсортированном массиве, есть такой элемент, который встречается в нем больше, чем *n/2* раз, то она всегда будет находится по середине массива. Зная это, мы можем найти его, поделив длину массива на 2, и тогда получим индекс данного элемента.

3) Далее, используя метод *count*, мы подсчитываем то, сколько раз этот элемент встречается в массиве *A*. И следом делаем проверку, если количество повторов элемента больше, чем *n/2*, то функция возвращает значение 1, иначе значение 0.

Тестирование:

1) Случай, когда представитель большинства есть в массиве:
   На вход функции подается массив, в котором 1 является представителем большинства.

```
A = [1]*10000 + [2]*5000 + [3]*4000
```

   Результаты модульного теста:

```
Время работы алгоритма_2: 0.000725 секунд
Использование памяти: 64.000000 Кб
```

2) Теперь случай, когда представителя большинства в массиве нет:

```
A = [1]*5000 + [2]*5000 + [3]*5000
```

Все различные элементы массива встречаются равное количество раз
Результаты модульного теста:

```
tests.py::AlgorithmsSortTestCase::test3_2_majority_element PASSED
Время работы алгоритма_2: 0.000064 секунд
Использование памяти: 0.000000 Кб
```

Количество используемой памяти очень мало, поэтому имеет значение 0Кб!

Задание 10*

Функция merge_sort_ref отличается от традиционной реализации сортировки слиянием тем, что в ней реализована проверка на отсортированность подмассивов left и right перед их объединением:

```
# Проверка, нужно ли объединять массивы в том случае, если массивы отсортированы!
if left[-1] <= right[0]:
    A[:] = left + right
```

Если массивы уже отсортированы, то они просто объединяются без дополнительной обработки. Это увеличивает эффективность в ситуациях, когда массивы уже частично отсортированы.

Проверка на сортированность перед слиянием позволяет избежать дополнительной работы, если оба подмассива уже отсортированы. Это может существенно ускорить выполнение, особенно при работе с частично отсортированными данными.

В остальном, логика самого алгоритма остается прежней: рекурсивно делится массив на меньшие подмассивы, которые затем сливаются в отсортированном порядке.

Вывод

В ходе данной лабораторной работы был изучен алгоритм сортировки слиянием и метод “Разделяй и властвуй”. Также был изучен процесс написания модульных тестов для замера времени работы алгоритмов и количества используемой памяти.