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 Вариант 1

Выполнила: Гайдук Алина

K3241

Проверила:

Ромакина О. М.

Санкт-Петербург 2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	
Задача №1. Сортировка слиянием	3
Задача №2. Сортировка слиянием+	5
Задача №3. Число инверсий	9
Дополнительные задачи	13
Задача №4. Бинарный поиск	13
Задача №5. Представитель большинства	15
Вывод	19

Задачи по варианту

Задача №1. Сортировка слиянием

Используя псевдокод процедуры Merge и Merge-sort из презентациик Лекции 2 (страницы 6–7), напишите программу сортировки слиянием на Python и проверьте сортировку, создав несколько рандомных массивов, подходящих под параметры.

```
import time
import tracemalloc
tracemalloc.start()
def split and merge(arr):
    if len(arr) == 1:
       return arr
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    if len(left) > 1:
        left = split and merge(left)
    if len(right) > \overline{1}:
        right = split and merge(right)
    return merge(left, right)
def merge(a, b):
    result = []
    i = 0
    j = 0
    while i < len(a) and j < len(b):
        if a[i] < b[j]:</pre>
            result.append(a[i])
            i += 1
        else:
            result.append(b[j])
            j += 1
    result += a[i:] + b[j:]
    return result
with open('input.txt', 'r') as f:
   n = int(f.readline())
   a = [int(x) for x in f.readline().split()]
start time = time.perf counter()
sorted list = split and merge(arr=a)
end time = time.perf counter()
current, peak = tracemalloc.get traced memory()
with open('output.txt', 'w') as f:
```

```
f.write(str(sorted_list))

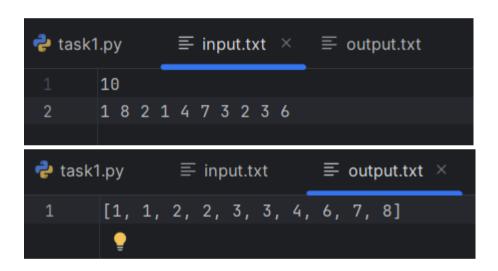
print(f"Затраты памяти: {current / 10**6}MB; Пиковое использование: {peak / 10**6 }MB")

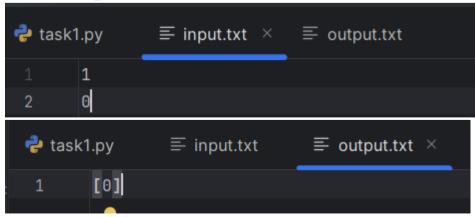
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")
```

Текстовое объяснение решения

В решении я использовала две функции — split and merge и merge. Функция split and merge проверяет количество элементов в массиве (что их больше 1), и затем делит массив надвое. Для каждого разделенного массива срабатывает рекурсия. Далее, когда в разделенных массивах остается по 1 элементу, вызывается функция merge, которая сравнивает элементы двух массивов между собой и добавляет меньший в массив-результат.

Результат работы кода на примерах из текста задачи:





```
      Image: absolute the large of the large
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000003 сек	0.037821 MB
Пример из задачи	0.000425 сек	0.0379 MB
Верхняя граница диапазона значений входных данных из текста задачи	0.304792 сек	1.9434 MB

Вывод по задаче:

В ходе выполнения данной задачи я ознакомилась с алгоритмом «сортировка слиянием». Здесь мы использовали метод декомпозиции или «разделяй и властвуй», разделяя исходный массив необходимое количество раз надвое. Таким образом исходную задачу удалось поделить на меньшие экземпляры той же задачи. «Властвование» заключается в рекурсивном решении этих подзадач. Затем мы объединили решения подзадач в решение исходной задачи.

Задача №2. Сортировка слиянием+

Дан массив целых чисел. Ваша задача — отсортировать его в порядке неубывания с использованием **сортировки слиянием**.

Для проверки, что действительно используется сортировка слиянием, требуется после каждого завершенного слияния (то есть, когда соответствующий подмассив уже отсортирован) выводить индексы граничных элементов и их значения.

```
import time
import tracemalloc
merge steps = []
def split and merge(arr):
    if len(arr) == 1:
       return arr
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    left = split_and_merge(left)
    right = split and merge (right)
    return merge(left, right)
def merge(a, b):
    global merge steps
    result = []
   i = 0
    i = 0
    I f = 1
    I l = I f + len(a) + len(b)
    V f = a[0] if a else b[0]
    V l = b[-1] if b else a[-1]
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            result.append(a[i])
            i += 1
        else:
            result.append(b[j])
            j += 1
    result += a[i:] + b[j:]
    merge steps.append((I f, I l, V f, V l))
    return result
with open('input.txt', 'r') as f:
   n = int(f.readline())
    a = [int(x) for x in f.readline().split()]
start time = time.perf counter()
tracemalloc.start()
sorted_list = split and merge(a)
```

```
current, peak = tracemalloc.get_traced_memory()
end_time = time.perf_counter()

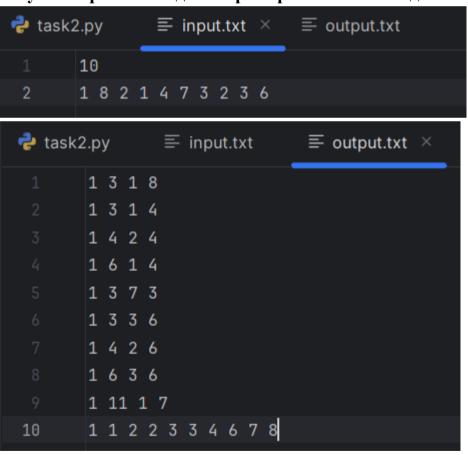
with open('output.txt', 'w') as f:
   for step in merge_steps:
        f.write(f"{step[0]} {step[1]} {step[2]} {step[3]}\n")
   f.write(" ".join(map(str, sorted_list)))

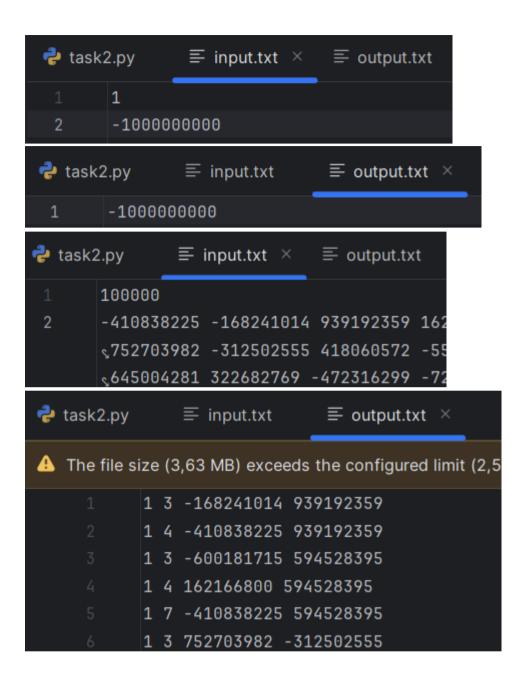
print(f"Затраты памяти: {current / 10 ** 6:.6f} MB; Пиковое
использование: {peak / 10 ** 6:.6f} MB")
print(f"Время выполнения программы: {end time - start time:.6f} секунд")
```

Текстовое объяснение решения

В рекурсивной функции split_and_merge я делю массив на две части, а затем объединяю части обратно в отсортированный массив через функцию merge. Сама функция merge выполняет слияние и сохраняет данные о каждом шаге (начальный и конечный индексы массива, первое и последнее его значения). Эти данные записываются в merge_steps.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000018 сек	0.000001 MB
Пример из задачи	0.000062 сек	0.000744 MB
Верхняя граница диапазона значений входных данных из	1.923654 сек	9.745056 MB

текста задачи

Вывод по задаче:

В результате выполнения этой задачи у меня получилось выполнить сортировку слиянием и описать каждый шаг слияния.

Задача №3. Число инверсий

Инверсией в последовательности чисел A называется такая ситуация, когда i < j, а $A_i > A_j$. Количество инверсий в последовательности в некотором роде определяет, насколько близка данная последовательность к отсортированной. Например, в отсортированном массиве число инверсий равно 0, а в массиве, сортированном наоборот — каждые два элемента будут составлять инверсию (всего n(n-1)/2).

Дан массив целых чисел. Ваша задача — подсчитать число инверсий в нем. **Подсказка**: чтобы сделать это быстрее, можно воспользоваться модификацией сортировки слиянием.

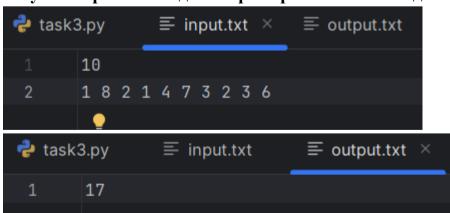
```
import time
import tracemalloc
tracemalloc.start()
def split and merge(arr):
    if len(arr) == 1:
        return arr, 0
   mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    left inversions, right inversions = 0, 0
    if len(left) > 1:
        left, left inversions = split and merge(left)
    if len(right) > 1:
        right, right inversions = split and merge(right)
    merged, merge_inversions = merge(left, right)
    total inversions = left inversions + right inversions +
merge inversions
    return merged, total inversions
def merge(a, b):
    result = []
    i = 0
    j = 0
    inversions = 0
    while i < len(a) and j < len(b):</pre>
        if a[i] \le b[j]:
            result.append(a[i])
```

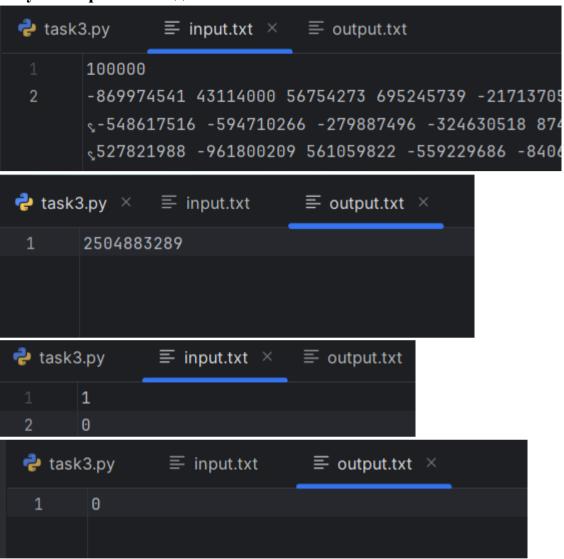
```
i += 1
        else:
            result.append(b[j])
            j += 1
            inversions += len(a) - i
    result += a[i:] + b[j:]
    return result, inversions
with open('input.txt', 'r') as f:
   n = int(f.readline())
   a = [int(x) for x in f.readline().split()]
start time = time.perf counter()
sorted list, inversions quantity = split and merge(arr=a)
end time = time.perf counter()
current, peak = tracemalloc.get traced memory()
with open('output.txt', 'w') as f:
   f.write(str(inversions quantity))
print(f"Затраты памяти: {current / 10**6}MB; Пиковое использование: {peak
/ 10**6 }MB")
print(f"Время выполнения программы: {end time - start time:.6f} секунд")
```

Текстовое объяснение решения:

Для решения данной задачи я лишь модифицировала код, написанный для задачи 1. Теперь для каждой половины массива подсчитываются инверсии, пока массив не станет единичного размера. При слиянии, если элемент из левого массива больше элемента из правого массива, мы выделяем инверсию. Число инверсий в этом случае равно числу оставшихся элементов в левом массиве. Далее я сложила инверсии с левой и с правой половин, а так же инверсии, возникшие при слиянии.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000003 сек	0.037821 MB
Пример из задачи	0.000065 сек	0.0379 MB
Верхняя граница диапазона значений входных данных из текста задачи	1.811412 сек	9.4714 MB

Вывод по задаче

В ходе выполнения данной задачи я ознакомилась с принципом подсчета количества инверсий в сортировке слиянием.

Дополнительные задачи

Задача №4. Бинарный поиск

В этой задаче вы реализуете алгоритм бинарного поиска, который позволяет очень эффективно искать (даже в огромных) списках при условии, что список отсортирован. Цель — реализация алгоритма двоичного (бинарного) поиска.

Листинг кода

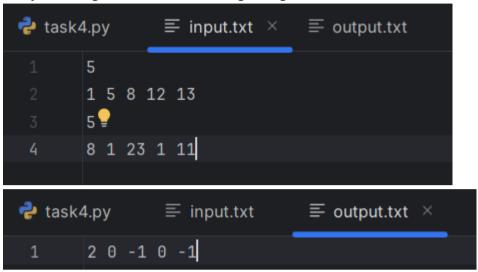
```
import time
import tracemalloc
tracemalloc.start()
def binary search(arr, item):
    left = 0
    right = len(arr)-1
    while left <= right:</pre>
        mid = (right + left) // 2
        quess = arr[mid]
        if quess == item:
            return mid
        if guess > item:
            right = mid - 1
            left = mid + 1
    return -1
with open('input.txt', 'r') as f:
    n = int(f.readline())
    a = [int(x) for x in f.readline().split()]
    k = int(f.readline())
    b = [int(x) for x in f.readline().split()]
start time = time.perf counter()
result = [binary search(arr=a, item=item) for item in b]
end time = time.perf counter()
current, peak = tracemalloc.get traced memory()
with open('output.txt', 'w') as f:
    f.write(' '.join(map(str, result)))
print(f"Затраты памяти: {current / 10**6}МВ; Пиковое использование: {peak
/ 10**6 }MB")
print(f"Время выполнения программы: {end time - start time:.6f} секунд")
```

Текстовое объяснение решения

Метод binary_search определяет индексы левой и правой границ массива, а также, пока элементы для поиска есть, средний индекс и значение среднего

элемента. Далее метод сравнивает это значение с искомым числом и двигает границы пока число не будет найдено.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000005 сек	0.037699 MB
Пример из задачи	0.000011 сек	0.037803 MB
Верхняя граница диапазона значений входных данных из текста задачи	1.658469 сек	13.024574 MB

Вывод по задаче:

В ходе выполнения этой задачи я научилась реализовывать алгоритм бинарного поиска, который является очень эффективным при работе с сортированными списками.

Задача №5. Представитель большинства

Правило большинства — это когда выбирается элемент, имеющий больше половины голосов. Допустим, есть последовательность A элементов $a_1, a_2, ..., a_n$ и нужно проверить, содержит ли она элемент, который появляется больше, чем n/2 раз.

Очевидно, время выполнения этого алгоритма квадратично. Ваша цель — использовать метод "Разделяй и властвуй" для разработки алгоритма проверки, содержится ли во входной последовательности элемент, который встречается больше половины раз, за время $O(n \log n)$.

```
import time
import tracemalloc

tracemalloc.start()

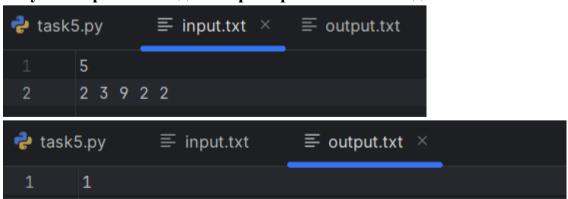
def majority(arr):
    n = len(arr)
    if n == 1:
        return arr[0]
    mid = n // 2
    left_majority = majority(arr[:mid])
    right_majority = majority(arr[mid:])
    if left_majority == right_majority:
        return left majority
```

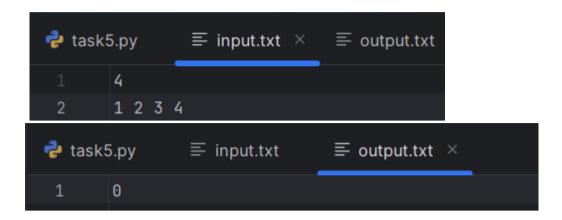
```
left count = sum(1 for element in arr if element==left majority)
    right count = sum(1 for element in arr if element == right majority)
    if left count + right count > mid:
        return 1
    return 0
with open('input.txt', 'r') as f:
   n = int(f.readline())
   a = [int(x) for x in f.readline().split()]
start time = time.perf counter()
result = majority(a)
end time = time.perf counter()
current, peak = tracemalloc.get traced memory()
with open('output.txt', 'w') as f:
   f.write(str(result))
print(f"Затраты памяти: {current / 10**6}MB; Пиковое использование: {peak
/ 10**6 }MB")
print(f"Время выполнения программы: {end time - start time:.6f} секунд")
```

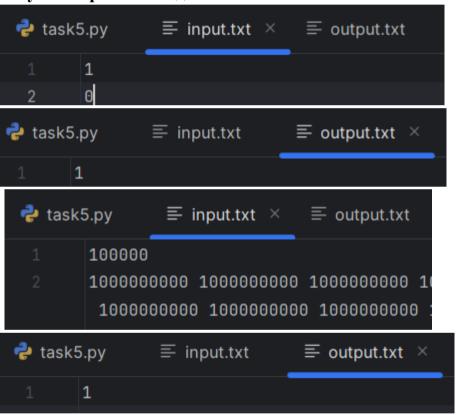
Текстовое объяснение решения

Функция majority рекурсивна. Изначально мы определяем длину массива, если массив единичен, то возвращаем «1» и значение единственного элемента массива. Далее рекурсивно находим представителя большинства для левого и правого разбиения массива и считаем, сколько раз он встречается в исходном массиве. Если в обеих половинах массива нет представителя большинства, то возвращается «0». В конце проверяем, превышает ли каждый из кандидатов половину длины массива.

Результат работы кода на примерах из текста задачи:







	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000004 сек	0.037661 MB
Пример из задачи	0.000029 сек	0.037715 MB

Пример из задачи	0.000041 сек	0.037719 MB
Верхняя граница диапазона значений входных данных из текста задачи	0.272692 сек	9.937066 MB

Вывод по задаче:

В ходе выполнения этой задачи я ознакомилась с правилом большинства и научилась реализовывать алгоритм проверки, содержится ли в последовательности элемент, который встречается больше половины раз, используя метод «Разделяй и властвуй».

Вывод

В ходе выполнения данной лабораторной работы я ознакомилась с более эффективными и быстрыми алгоритмами сортировки данных, а именно с сортировкой слиянием, а также с методом декомпозиции — «Разделяй и властвуй».