

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
К3241

Проверила:
Ромакина О. М.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета.....	2
Задачи по варианту	3
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 презентации Лекции 2 (страницы 6–7), напишите программу сортировки слиянием на Python и проверьте сортировку, создав несколько случайных массивов, подходящих под параметры.

Листинг кода

```
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) > 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]:
            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}МВ; Пиковое использование: {peak / 10**6 }МВ")
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:

```
task1.py  input.txt  output.txt
1 10
2 1 8 2 1 4 7 3 2 3 6

task1.py  input.txt  output.txt
1 [1, 1, 2, 2, 3, 3, 4, 6, 7, 8]
```

Результат работы кода на максимальных и минимальных значениях:

```
task1.py  input.txt  output.txt
1 1
2 0

task1.py  input.txt  output.txt
1 [0, 1]
```

```
task1.py  input.txt  output.txt
1 20000
2 581308046 558203431 -234559966 725577172
  -987100663 493557662 770565887 86582853
  -641172824 -60070452 -440424515 -409533
```

```
task1.py  input.txt  output.txt
1 [-999966223, -999600759, -999556097, 2
  -999495368, -999480038, -999413473, 2
  -999329710, -999319267, -999181653, 2
```

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

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.

Листинг кода

```
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
    j = 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} МВ; Пиковое
использование: {peak / 10 ** 6:.6f} МВ")
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:

task2.py	input.txt	output.txt
1	10	
2	1 8 2 1 4 7 3 2 3 6	

task2.py	input.txt	output.txt
1	1 3 1 8	
2	1 3 1 4	
3	1 4 2 4	
4	1 6 1 4	
5	1 3 7 3	
6	1 3 3 6	
7	1 4 2 6	
8	1 6 3 6	
9	1 11 1 7	
10	1 1 2 2 3 3 4 6 7 8	

Результат работы кода на максимальных и минимальных значениях:

```
task2.py  input.txt  output.txt
1 1
2 -1000000000
```

```
task2.py  input.txt  output.txt
1 -1000000000
```

```
task2.py  input.txt  output.txt
1 100000
2 -410838225 -168241014 939192359 162
  752703982 -312502555 418060572 -55
  645004281 322682769 -472316299 -72
```

```
task2.py  input.txt  output.txt
⚠ The file size (3,63 MB) exceeds the configured limit (2,5

1 1 3 -168241014 939192359
2 1 4 -410838225 939192359
3 1 3 -600181715 594528395
4 1 4 162166800 594528395
5 1 7 -410838225 594528395
6 1 3 752703982 -312502555
```

	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

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.

Листинг кода

```
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):
        if a[i] <= 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}МВ; Пиковое использование: {peak / 10**6 }МВ")
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:

task3.py	input.txt	output.txt
1	10	
2	1 8 2 1 4 7 3 2 3 6	

task3.py	input.txt	output.txt
1	17	

Результат работы кода на максимальных и минимальных значениях:

```
task3.py  input.txt  output.txt
1 100000
2 -869974541 43114000 56754273 695245739 -21713705
  -548617516 -594710266 -279887496 -324630518 874
  -527821988 -961800209 561059822 -559229686 -8406
```

```
task3.py  input.txt  output.txt
1 2504883289
```

```
task3.py  input.txt  output.txt
1 1
2 0
```

```
task3.py  input.txt  output.txt
1 0
```

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

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:
        mid = (right + left) // 2
        guess = arr[mid]
        if guess == item:
            return mid
        if guess > item:
            right = mid - 1
        else:
            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}МВ")
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:

```
task4.py  input.txt  output.txt
1 5
2 1 5 8 12 13
3 5
4 8 1 23 1 11
```

```
task4.py  input.txt  output.txt
1 2 0 -1 0 -1
```

Результат работы кода на максимальных и минимальных значениях:

```
task4.py  input.txt  output.txt
1 1
2 1
3 1
4 1
```

```
task4.py  input.txt  output.txt
1 0
```

```
task4.py  input.txt  output.txt
1 100000
2 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

```
task4.py  input.txt  output.txt
1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

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

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, \dots, 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 $O(n \log n)$.

Листинг кода

```
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}МВ; Пиковое использование: {peak / 10**6 }МВ")
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:

task5.py	input.txt	output.txt
1	5	
2	2 3 9 2 2	

task5.py	input.txt	output.txt
1	1	


```
task5.py  input.txt  output.txt
1 4
2 1 2 3 4
```

```
task5.py  input.txt  output.txt
1 0
```

Результат работы кода на максимальных и минимальных значениях:

```
task5.py  input.txt  output.txt
1 1
2 0
```

```
task5.py  input.txt  output.txt
1 1
```

```
task5.py  input.txt  output.txt
1 100000
2 1000000000 1000000000 1000000000 1000000000 1000000000
   1000000000 1000000000 1000000000 1000000000 1000000000
```

```
task5.py  input.txt  output.txt
1 1
```

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

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

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».