

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:  
Авдиенко Данила Андреевич  
Группа К3140

Проверил:  
Афанасьев А. В.

Санкт-Петербург  
2024 г.

# Содержание отчета

## Оглавление

## Оглавление

<i>Содержание отчета.....</i>	<i>2</i>
<i>Задачи по варианту.....</i>	<i>3</i>
Задание № 1. Сортировка вставкой.....	3
Задание №3. Обратная сортировка вставкой.....	6
Задание №4. Линейный поиск.....	8
Задание №6. Пузырьковая сортировка .....	10
Задание №8. Секретарь Свop. ....	11

## Задачи по варианту

### Задание № 1. Сортировка вставкой

Текст задачи.

Используя *псевдокод* процедур Merge и Merge-sort из презентации к Лекции 2 (страницы 6-7), напишите программу сортировки слиянием на Python и проверьте сортировку, создав несколько случайных массивов, подходящих под параметры:

Код:

```
import time
import resource
from lab2.utils import write_output, read_input

# Function to print memory usage in MB
def print_memory_usage():
    usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
    print(f"Memory usage: {usage / 1024:.2f} MB") # Convert to MB

def merge_sort(arr: list):
    if len(arr) > 1: # проверяем, не является ли длина массива единицей
        l = arr[:len(arr) // 2] # делим текущий массив на две части
        r = arr[len(arr) // 2:]
        # рекурсивно вызываем функцию на каждую из частей, пока она внутри
        # не дойдет до условия, что длина == 1 и не начнет выполнять код ниже
        merge_sort(l)
        merge_sort(r)
        templist = [] # временный список для слияния двух частей
        i = j = 0 # задаем индексы для работы с массивами
        while len(l) > i and len(
            r) > j: # условие для прекращения (пока индекс
# сравниваемого элемента не станет равным длине списка)
            if l[i] <= r[j]: # сравнение
                templist.append(l[i])
                i += 1
            else:
                templist.append(r[j])
                j += 1
            if len(l) == i: # если мы подошли к концу списка l, то добавляем
# все элементы списка r и наоборот в else
                templist.extend(r[j:])
            else:
                templist.extend(l[i:])
        for i in range(len(templist)): # обновляем исходный лист
            arr[i] = templist[i]

if __name__ == "__main__":
    time_start = time.perf_counter()
    n, arr = read_input("input.txt")
    merge_sort(arr)
    write_output(arr, "output.txt")
    time_elapsed = (time.perf_counter() - time_start)
    mmry = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / 1024.0 /
1024.0
    print("Время:", time_elapsed)
    print("Память:%5.1f MB" % (mmry))
```

Текстовое объяснение функции merge\_sort.

Получаем список в качестве аргумента функции, проверяем базовый случай, определяем две половины массива и рекурсивно от них вызываем эту же функцию. Когда доходим до самого глубокого уровня рекурсии, то берем и проводим поэлементное сравнение элементов левой и правой части, в конце итерации обновляем исходный список и возвращаемся на уровень выше

Примеры работы кода в худшем случае + сравнение с сортировкой вставками:

merge_sort	insertion_sort
<pre>/usr/bin/python3 /Users/danilaa Время: 0.223002792 Память: 25.4 МБ Process finished with exit code</pre>	<pre>/usr/bin/python3 /Users/danilaa Время: 354.956595375 Память: 21.6 МБ</pre>

Обе функции протестированы на идентичных наборах данных, где  $10^9$  чисел отсортированы в обратном порядке.

Тесты с использованием отдельных текстовых файлов для тест-кейсов:

```
import unittest
from lab2.task1.first import merge_sort

class TestFirstTask(unittest.TestCase):
    pass

with open("test_data.txt", "r") as f:
    for i, line in enumerate(f):
        if line.startswith("#") or not line.strip():
            continue

        unsorted_str, expected_str = line.strip().split("|")
        unsorted = list(map(int, unsorted_str.split(",")))
        expected = list(map(int, expected_str.split(",")))

        def create_test_function(unsorted, expected):
            def test_should_merge_sort(self):
                merge_sort(unsorted)
                self.assertEqual(unsorted, expected)
            return test_should_merge_sort
        test_name = f"test_should_merge_sort{i+1}"
        test_should_merge_sort = create_test_function(unsorted, expected)
        setattr(TestFirstTask, test_name, test_should_merge_sort)

if __name__ == "__main__":
    unittest.main()
```

Вывод: был реализован алгоритм сортировки слиянием, функция протестирована отдельным файлом.

## Задание №2. Сортировка слиянием+

Текст задачи.

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

Чтобы убедиться, что Вы действительно используете сортировку слиянием, мы просим Вас, после каждого осуществленного слияния (то есть, когда соответствующий подмассив уже отсортирован!), выводить индексы граничных элементов и их значения.

Код:

```
import time
import resource
from lab2.utils import write_output, read_input

# Function to print memory usage in MB
def print_memory_usage():
    usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
    print(f"Memory usage: {usage / 1024:.2f} MB") # Convert to MB

def merge_sort(arr: list, index_log: list, start_idx=0):
    if len(arr) > 1: # проверяем, не является ли длина массива единицей
        l = arr[:len(arr) // 2] # делим текущий массив на две части
        r = arr[len(arr) // 2:]
        # рекурсивно вызываем функцию на каждую из частей, пока она внутри
        # не дойдет до условия, что длина == 1 и не начнет выполнять код ниже
        merge_sort(l, index_log, start_idx)
        merge_sort(r, index_log, start_idx + len(l))
        templist = [] # временный список для слияния двух частей
        i = j = 0 # задаем индексы для работы с массивами
        while len(l) > i and len(
            r) > j: # условие для прекращения (пока индекс
                # сравниваемого элемента не станет равным длине списка)
                if l[i] <= r[j]: # сравнение
                    templist.append(l[i])
                    i += 1
                else:
                    templist.append(r[j])
                    j += 1
            if len(l) == i: # если мы подошли к концу списка л, то добавляем
                # все элементы списка р и наоборот в елсе
                templist.extend(r[j:])
            else:
                templist.extend(l[i:])

            # out.write(f"{arr.index(templist[0]) + 1} {arr.index(templist[-1])
            # + 1} {templist[0]} {templist[-1]}\n")
            # Записываем индексы начала и конца объединенного списка и значения
            first_idx = start_idx
            last_idx = start_idx + len(templist) - 1
            index_log.append((first_idx + 1, last_idx + 1, templist[0],
                templist[-1])) # +1 для человеческого индекса

            for i in range(len(templist)): # обновляем исходный лист
                arr[i] = templist[i]

if __name__ == "__main__":
    time_start = time.perf_counter()
    n, arr = read_input("input.txt")
```

```

index_log = [] # Хранение индексов и значений для каждой итерации
слияния
merge_sort(arr, index_log)
write_output(arr, "output.txt", index_log)
time_elapsed = (time.perf_counter() - time_start)
mmry = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / 1024.0 /
1024.0
print("Время:", time_elapsed)
print("Память:%5.1f МБ" % (mmry))

```

### Текстовое объяснение функции merge\_sort (+)

Тот же самый merge\_sort, что и в первом задании, только теперь записываем в файл индексы граничных элементов и их значения.

### Примеры работы кода и затраты:

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

```

/usr/bin/python3 /Users/danilaa
Время: 0.0008688339999999989
Память: 8.2 МБ

```

Вывод: был реализован алгоритм сортировки слиянием, который записывает индексы граничных элементов и их значения.

### Задание №3. Число инверсий

Текст задачи:

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, отсортированном наоборот - каждые два элемента будут составлять инверсию (всего  $n(n-1)/2$ ).

Код:

```
import time
import resource
from lab2.utils import write_output, read_input

# Function to print memory usage in MB
def print_memory_usage():
    usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
    print(f"Memory usage: {usage / 1024:.2f} MB") # Convert to MB

def merge_sort(arr: list):
    number_of_inversions = 0
    if len(arr) > 1: # проверяем, не является ли длина массива единицей
        l = arr[:len(arr) // 2] # делим текущий массив на две части
        r = arr[len(arr) // 2:]
        # рекурсивно вызываем функцию на каждую из частей, пока она внутри
        # не дойдет до условия, что длина == 1 и не начнет выполнять код ниже
        number_of_inversions += merge_sort(l)
        number_of_inversions += merge_sort(r)
        templist = [] # временный список для слияния двух частей
        i = j = 0 # задаем индексы для работы с массивами
        while len(l) > i and len(
            r) > j: # условие для прекращения (пока индекс
# сравниваемого элемента не станет равным длине списка)
            if l[i] <= r[j]: # сравнение
                templist.append(l[i])
                i += 1
            else:
                templist.append(r[j])
                j += 1
                number_of_inversions += len(l) - i
        if len(l) == i: # если мы подошли к концу списка л, то добавляем
# все элементы списка р и наоборот в елсе
            templist.extend(r[j:])
        else:
            templist.extend(l[i:])
        for i in range(len(templist)): # обновляем исходный лист
            arr[i] = templist[i]
        return number_of_inversions

if __name__ == "__main__":
    time_start = time.perf_counter()
    n, arr = read_input("input.txt")
    number_of_inversions = merge_sort(arr)
    with open("output.txt", "w") as out:
        out.write(str(number_of_inversions))
    time_elapsed = (time.perf_counter() - time_start)
    mmry = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / 1024.0 /
1024.0
```



```
print("Время:", time_elapsed)
print("Память:%5.1f МБ" % (mmry))
```

Текстовое объяснение решения.

Алгоритм реализован при помощи функции `merge_sort`, но с некоторыми доработками: если выполняется условие инверсии, то мы сохраняем текущее количество инверсий для каждого элемента в переменную.

Пример работы кода и затраты:

third.py	input.txt	:	output.txt
1	10	✓	1 17
2	1 8 2 1 4 7 3 2 3 6		

Время: 0.00043524999999999999
Память: 8.5 МБ

Вывод по задаче: был реализован алгоритм, который считает количество инверсий.

## Задание №4. Бинарный поиск.

Текст задачи.

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.

Код:

```
import time
import resource
from lab2.utils import write_output, read_input_for_binary_search

# Function to print memory usage in MB
def print_memory_usage():
    usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
    print(f"Memory usage: {usage / 1024:.2f} MB") # Convert to MB

def binary_search(sorted_arr: list, number: int):
    l, r = 0, len(sorted_arr) - 1
    while l <= r:
        mid = (l + r) // 2
        if sorted_arr[mid] == number:
            return mid
        elif sorted_arr[mid] < number:
            l = mid + 1
        else:
            r = mid - 1
    return -1

if __name__ == "__main__":
    time_start = time.perf_counter()
    n_sorted, sorted_arr, n, arr =
    read_input_for_binary_search("input.txt")
    answer = []
    for i in arr:
        g = str(binary_search(sorted_arr, i))
        answer.append(g)
    write_output(answer, "output.txt")
    time_elapsed = (time.perf_counter() - time_start)
    mmry = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / 1024.0 /
1024.0
    print("Время:", time_elapsed)
    print("Память:%5.1f МБ" % (mmry))
```

Текстовое объяснение решения.

Реализован алгоритм бинарного поиска для нахождения заданного числа в заданном массиве.

Пример работы и затраты:

```
/usr/bin/python3 /Users/danilaavdienko
Время: 0.0009153340000000003
Память: 8.5 МБ
```

four.py		input.txt ×		output.txt ×	
1	5	✓	1	2	0 -1 0 -1
2	1 5 8 12 13		2		
3	5				
4	8 1 23 1 11				

Вывод по задаче: был реализован алгоритм бинарного поиска.

## Задание №5. Представитель большинства.

### Текст задачи:

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. Наивный метод это сделать:

### Код:

```
import time
import resource
from lab2.utils import write_output, read_input_for_binary_search,
read_input

# Function to print memory usage in MB
def print_memory_usage():
    usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
    print(f"Memory usage: {usage / 1024:.2f} MB") # Convert to MB

def majority_element(A, left, right):
    if left == right:
        return A[left]

    # массив пополам
    mid = (left + right) // 2
    # ищем элемент большинства в левой и правой половинах
    left_majority = majority_element(A, left, mid)
    right_majority = majority_element(A, mid + 1, right)

    # если оба подмассива возвращают одинаковый элемент
    if left_majority == right_majority:
        return left_majority

    # количество вхождений каждой цифры в текущем подмассиве
    left_count = sum(1 for i in range(left, right + 1) if A[i] ==
left_majority)
    right_count = sum(1 for i in range(left, right + 1) if A[i] ==
right_majority)

    # возвращаем элемент
    if left_count > (right - left + 1) // 2:
        return left_majority
    if right_count > (right - left + 1) // 2:
        return right_majority

    # нет подходящего элемента
    return 0

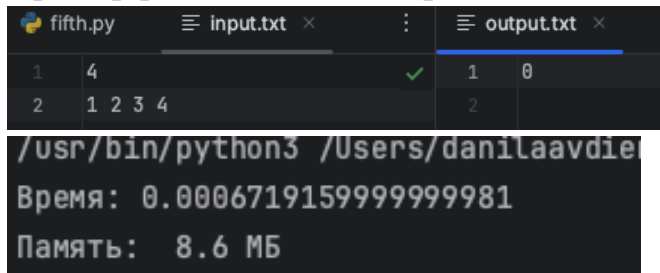
if __name__ == "__main__":
    time_start = time.perf_counter()
    time_elapsed = (time.perf_counter() - time_start)
    n, arr = read_input("input.txt")
    res = majority_element(arr, 0, n - 1)
    if res != 0:
        res = 1
    write_output(str(res))
    mmry = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / 1024.0 /
1024.0
```

```
print("Время:", time_elapsed)
print("Память:%5.1f МБ" % (mmry))
```

Текстовое объяснение решения.

Реализован алгоритм, который определяет, существует ли в списке элемент, который занимает более половины позиций в этом списке. Проверяется базовый случай, рекурсивно вызываем функцию от левой и правой части и если там один и тот же элемент, то он является элементом большинства, если нет то подсчитываем количество элементов равных представителю большинства левой и правой частей.

Пример работы кода и затраты:



The screenshot shows a code editor with two tabs: 'input.txt' and 'output.txt'. The 'input.txt' tab contains the following content:

1	4
2	1 2 3 4

The 'output.txt' tab contains the following content:

1	0
2	

Below the editor, a terminal window shows the command `/usr/bin/python3 /Users/danilaavdie` and the output:

```
Время: 0.00067191599999999981
Память: 8.6 МБ
```

Вывод по задаче: реализован алгоритм, который ищет элемент большинства.

**Вывод:**

В лабораторной работе были решены задания при помощи алгоритма merge\_sort и его модификаций. Создан readme файл, файл utils.