

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 12

Выполнила:
Мкртчян.К.Г.
К3141

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №1. Сортировка слиянием [N баллов]	3
Задача №3. Число инверсий [N баллов]	6
Задача №8. Умножение многочленов [N баллов]	9
Дополнительные задачи	12
Задача №2. Сортировка слиянием+ [N баллов]	12
Задача №5. Представитель большинства [N баллов]	14
Задача №7. Поиск максимального подмассива за линейное время [N баллов]	16 16
Вывод по всей лабораторной	19

Задачи по варианту

Задача №1. Сортировка слиянием [N баллов]

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 рандомных массивов, подходящих под параметры.

Решение:

```
def verification(n, array, attempt=1):
    res = 1
    if type(n) is int and 1 <= n <= 10 ** 5:
        if type(array) is list and n == len(array) and all(type(x) is int
and abs(x) <= 10 ** 9 for x in array):
            return merge_sort(n, array)
        else:
            res *= 0
    else:
        res *= 0
    if res == 0:
        if attempt == 3:
            return 'Ошибка!'
        else:
            print("Введите данные ещё раз, соблюдая ограничения: ")
            try:
                new_n = int(input())
                new_array = list(map(int, input().split(" ")))
                return verification(new_n, new_array, attempt + 1)
            except:
                return 'Ошибка!'

def merge_sort_main(*args):
    if len(args) == 1:
        path = args[0]
        file_input = open(path, 'r')
        len_arr = int(file_input.readline().strip())
        array = list(map(int, file_input.readline().strip().split(" ")))

        file_output = open('output' + path[5:], 'w')
        result = " ".join(map(str, verification(len_arr, array)))
        file_output.write(result)
    else:
        len_arr, array = args
        return verification(len_arr, array)

def merge_sort(len_arr, array):
    middle = len_arr // 2
    list_a, list_b = array[:middle], array[middle:]
    len_a, len_b = middle, len_arr - middle
    if len_a > 1:
        list_a = merge_sort(len_a, list_a)
    if len_b > 1:
        list_b = merge_sort(len_b, list_b)
    return merge(len_a, len_b, list_a, list_b)
```

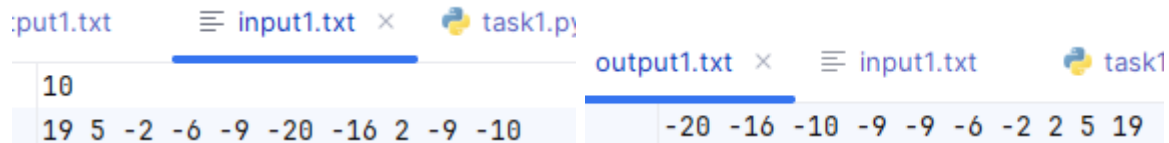
```
def merge(len_a, len_b, array_a, array_b):
    len_c = len_a + len_b
    array_c = [0] * len_c
    index_a, index_b = 0, 0
    for index_c in range(len_c):
        if index_b >= len_b:
            array_c[index_c] = array_a[index_a]
            index_a += 1
        elif index_a >= len_a:
            array_c[index_c] = array_b[index_b]
            index_b += 1
        else:
            if array_a[index_a] <= array_b[index_b]:
                array_c[index_c] = array_a[index_a]
                index_a += 1
            else:
                array_c[index_c] = array_b[index_b]
                index_b += 1
    return array_c
```

Объяснение решения:

1. В первой функции `verification`, которая принимает на вход количество элементов в массиве, сам массив и число попыток ввода данных, мы проверяем тип переменных и соответствие их значений ограничениям. Если все данные удовлетворяют условиям, то переходим к выполнению функции `merge_sort`, если это не так, то просим пользователя ввести данные ещё 2 раза. Если пользователь ввёл неподходящие нам данные уже в третий раз, то программа сообщит об ошибке и завершится. Также мы считываем две строки из терминала, поэтому, если пользователь ввёл все данные в одну строку, то программа выдаст ошибку.
2. В функции `merge_sort` мы находим середину массива и делим его пополам, если длина массива всё ещё больше единицы, то повторяем эту операцию. К двум получившимся половинкам применяем функцию `merge`.
3. В функции `merge` мы соединяем два списка следующим образом: создаем массив `array_c` и заполняем его нулями (длина списка равна сумме длин списков `array_a` и `array_b`). Создадим также переменные для хранения индекса текущего элемента из первого и второго списков. Затем в цикле будем записывать в итоговой массив элемент из `array_a` и прибавлять единицу к `index_a`, если `array_a[index_a] <= array_b[index_b]`. Если это не так, то запишем элемент из второго списка и увеличим `index_b`. Если мы добавили все элементы из первого (второго) массива, тогда добавим оставшиеся из второго (первого). Функция возвращает массив `array_c` – результат слияния двух списков.

4. Функция `merge_sort_main` нужна для того, чтобы считать данные из файла, проверить данные из терминала или файла, запустив функцию `verification`, которая запустит `merge_sort`, если значения удовлетворяют условиям задачи.

Результат работы кода на примере:

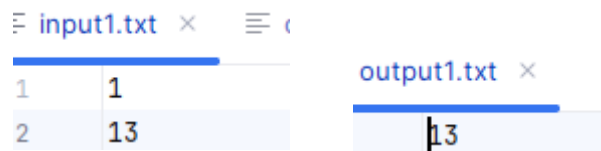


```

input1.txt
10
19 5 -2 -6 -9 -20 -16 2 -9 -10

output1.txt
-20 -16 -10 -9 -9 -6 -2 2 5 19
  
```

Результат работы кода на минимальных значениях:

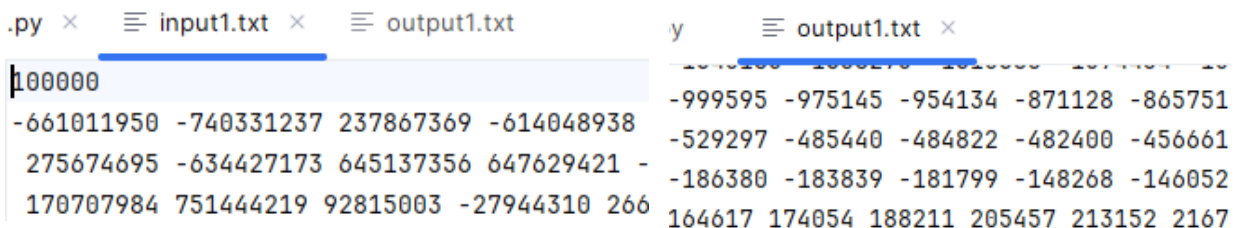


```

input1.txt
1
2

output1.txt
1
13
  
```

Результат работы кода на максимальных значениях:



```

input1.txt
100000
-661011950 -740331237 237867369 -614048938
275674695 -634427173 645137356 647629421 -
170707984 751444219 92815003 -27944310 266

output1.txt
-999595 -975145 -954134 -871128 -865751
-529297 -485440 -484822 -482400 -456661
-186380 -183839 -181799 -148268 -146052
164617 174054 188211 205457 213152 2167
  
```

	Время выполнения, сек	Затраты памяти, МБ
1 0	0.000011	19.62
1000 -349649722987715729...	0.00109	20.02
10000 750431581265752675 - 102766916...	0.01324	20.68
100000 -1951942875553582...	0.1596	23.98

Вывод по задаче:

Мы научились применять сортировку слиянием, которая эффективна по времени работы и используемой памяти. Поняли принцип «разделяй и властвуй», а также повторили ввод и вывод данных из файла. Сортировка слиянием на массиве из 1000 элементов работает в 20 раз быстрее

сортировки вставками. На таких данных оба алгоритма используют одинаковый объем памяти – около 20 МБ.

Задача №3. Число инверсий [N баллов]

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 сортированном 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.

Решение:

```
result = 0
def verification(n, array, attempt=1):

    res = 1
    if type(n) is int and 1 <= n <= 10 ** 5:
        if type(array) is list and n == len(array) and all(type(x) is int
and abs(x) <= 10 ** 9 for x in array):
            return merge_sort(n, array)
        else:
            res *= 0
    else:
        res *= 0
    if res == 0:
        if attempt == 3:
            return 'Ошибка!'

        else:
            print("Введите данные ещё раз, соблюдая ограничения: ")
            try:
                new_n = int(input())
                new_array = list(map(int, input().split(" ")))
                return verification(new_n, new_array, attempt)
            except:
                return 'Ошибка!'

def merge_sort_main(args):
    global result
    result = 0
    if len(args) == 1:
        path = args[0]
        file_input = open(path, 'r')
        len_arr = int(file_input.readline().strip())
        array = list(map(int, file_input.readline().strip().split(" ")))

        path = 'output' + path[5:]
        file_output = open(path, 'w')
        result = str(number_of_permutations(len_arr, array))
        file_output.write(result)
        file_output.close()
    else:
        len_arr, array = args
        return verification(len_arr, array)
```

```

def merge_sort(len_arr, array):
    middle = len_arr // 2
    list_a, list_b = array[:middle], array[middle:]
    len_a, len_b = middle, len_arr - middle
    if len_a > 1:
        list_a = merge_sort(len_a, list_a)
    if len_b > 1:
        list_b = merge_sort(len_b, list_b)
    return merge(len_a, len_b, list_a, list_b)

def merge(len_a, len_b, array_a, array_b):
    global result
    len_c = len_a + len_b
    array_c = [0] * len_c
    count = 0
    index_a, index_b = 0, 0
    for index_c in range(len_c):
        if index_b >= len_b:
            array_c[index_c] = array_a[index_a]
            step = abs(index_c - index_a)
            index_a += 1
        elif index_a >= len_a:
            array_c[index_c] = array_b[index_b]
            step = abs(index_c - len_a - index_b)
            index_b += 1
        else:
            if array_a[index_a] <= array_b[index_b]:
                array_c[index_c] = array_a[index_a]
                step = abs(index_c - index_a)
                index_a += 1
            else:
                array_c[index_c] = array_b[index_b]
                step = abs(index_c - len_a - index_b)
                index_b += 1
        count += step
    result += count//2
    return array_c

def number_of_permutations(*args):
    merge_sort_main([i for i in args])
    return result

```

Объяснение решения:

- 1) Создадим глобальную переменную `result`, которая будет хранить число инверсий.
- 2) Функции `merge_sort_main`, `merge_sort`, `verification` работают аналогично одноименным функциям в предыдущей задаче, только в функции `merge_sort_main` мы записываем результат функции `number_of_permutations`.
- 3) В функцию `merge` добавим переменную-счётчик – `count` и переменную шаг – `step`. После каждого действия сортировки мы считаем шаг – количество перестановок для данной переменной: для левой части массива он будет равен разности нового и старого индексов (`index_c - index_a`), а для правой – разности нового и старого индекса минус длина левой части массива (`index_c - len_a - index_b`).

После каждой итерации прибавляем значение шага к общему числу перестановок (`count += step`). По завершение сортировки добавляем `count // 2` к результату. Делим значение `count` для того, чтобы не учитывать изменение индексов элементов, которые автоматически попали бы на свое место при, например, сортировке вставками.

- 4) Для того, чтобы посчитать число перестановок вызовем одноименную функцию – `number_of_permutations`. Она принимает на вход либо имя файла, в котором содержатся данные, либо длину массива и сам массив. Возвращает функция переменную `result`.

Результат работы кода на примере из задачи:

input3.txt × task3.py

10

1 8 2 1 4 7 3 2 3 6

= output3.txt ×

1 17

Результат работы кода на минимальных значениях:

input3.txt ×

1 1

2 18

= output3.txt ×

1 0

Результат работы кода на максимальных значениях:

output3.txt × input3.txt × task3.py

100000

-73156353 932485320 72148939 387238881 338789520 63836749 -69536553 -72718342 919276068 677027373 -971379661 -451805

= output3.txt × input3.txt ×

1 2503949796

	Время выполнения, сек	Затраты памяти, МБ
1 0	0.0000071	19.96
1000 -511304447 -95059240 373123010...	0.00138	19.98
10000 679843547 -348593807...	0.01723	20.53
100000 -5269212 916664534 -608053615...	0.2159	23.85

Вывод по задаче: мы научились считать число перестановок необходимых для сортировки массива и дополнили функцию сортировки слиянием, а также проверили получившиеся значения с помощью сортировки пузырьком.

Задача №8. Умножение многочленов [N баллов]

Текст задачи: Задача. Даны 2 многочлена порядка $n-1$: $a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x + a_0$ и $b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \dots + b_1x + b_0$. Нужно получить произведение: $c_{2n-2}x^{2n-2} + c_{2n-3}x^{2n-3} + \dots + c_1x + c_0$.

Решение:

```
from math import log2, ceil

def verification(n, A, B, attempt=1):
    res = 1
    if type(n) is int and n > 0 and type(A) is list and type(B) is list:
        if len(A) == len(B) == n:
            pass
        else:
            res *= 0
    else:
        res *= 0
    if res == 0:
        if attempt == 3:
            return 'Ошибка!'
        else:
            print("Введите данные ещё раз, соблюдая ограничения: ")
            try:
                new_n = int(input())
                new_A = list(map(int, input().split(" ")))
                new_B = list(map(int, input().split(" ")))
                return verification(new_n, new_A, new_B, attempt + 1)
            except:
                return 'Ошибка!'
    else:
        return res

def karatsuba_polynomial_multiply(*args):
    if len(args) == 1:
        path = args[0]
        file_input = open(path, 'r')
        n = int(file_input.readline())
        A = list(map(int, file_input.readline().strip().split()))
        B = list(map(int, file_input.readline().strip().split()))
        if verification(n, A, B):
            file_output = open('output' + path[5:], 'w')
            file_output.write(reformat(multiply(power_of_two(A),
power_of_two(B))))
        else:
            n, A, B = args
            if verification(n, A, B):
                return reformat(multiply(power_of_two(A), power_of_two(B)))

def multiply(poly1, poly2):
    n = len(poly1)
    if n == 1:
```

```

        return [poly1[0] * poly2[0]]

    mid = n // 2
    a = poly1[:mid]
    b = poly1[mid:]
    c = poly2[:mid]
    d = poly2[mid:]

    ac = multiply(a, c)
    bd = multiply(b, d)
    third_product = multiply([a[i] + b[i] for i in range(mid)], [c[i] +
d[i] for i in range(mid)])

    middle = [third_product[i] - ac[i] - bd[i] for i in
range(len(third_product))]
    result = []

    n = []
    i = 3
    while i <= len(ac):
        n.append(i)
        i = 2*i + 1
    if len(ac) in n:
        result += union(ac, middle, bd)
        return result
    else:
        return ac + middle + bd

def union(a, b, c):
    n = len(a) // 2
    return a[:n] + [a[-n + i] + b[i] for i in range(n)] + b[n:-n] + [b[-n
+ i] + c[i] for i in range(n)] + c[n:]

def power_of_two(m):
    n = len(m)
    return [0] * (2 ** ceil(log2(n)) - n) + m

def reformat(x):
    for i in range(len(x)):
        if x[i] != 0:
            return " ".join(map(str, x[i:]))

```

Объяснение решения:

- 1) karatsuba_polynomial_multiply – главная функция, с ее помощью мы считываем данные из файла, при необходимости, и вызываем функцию проверки, а затем функцию умножения полиномов.
- 2) Функция verification также, как и предыдущие одноименные функции проверяет полученные данные. В этой задаче нет строгих ограничений, поэтому проверим, чтобы n – порядок многочленов являлся натуральным числом, а длины массивов A , B были равны друг другу и равны n .
- 3) Если введенные значения удовлетворяют условиям задачи, то переходим к выполнению функции multiply, которая принимает на

вход два массива – коэффициенты полиномов. Для того, чтобы массивы всегда делились ровно пополам, добавим в начало каждого массива нули, чтобы их длина равнялась двойке в какой-либо степени. Для этого при вызове функции `multiply` применим функцию `power_of_two` к обоим массивам.

- 4) В функции `multiply` разделим первый полином на a, b , а второй – на c, d . Переменные ac, bd содержат произведение этих переменных соответственно. Переменная `third_product` будет содержать произведение $(a + b)(c + d)$, а переменная `middle` содержит разность `third_product` и ac, bd . $third_product = (a + b)(c + d) - ac - bd = \underline{ac} + ad + bc + \underline{bd} - \underline{ac} - \underline{bd} = ad + bc$ – это и есть коэффициенты из середины многочлена. Использование трех произведений вместо четырех, помогает быстрее выполнить перемножение многочленов. Такой способ предложил советский математик – Анатолий Карацуба. Алгоритм Карацубы имеет сложность $n^{\log_2 3}$ вместо n^2 .
- 5) Затем мы объединяем результаты трех произведений с помощью функции `union`.
- 6) С помощью функции `reformat` мы убираем лишние нули в начале массива и преобразуем массив в строку, где коэффициенты разделены пробелами.

Результат работы кода на примерах из текста задачи:

```

input8.txt × task8.py
3
3 2 5
5 1 2
output8.txt × input8.txt
1 15 13 33 9 10

```

Результат работы кода на минимальных значениях:

```

input8.txt ×
1 1
2 4
3 5
output8.txt ×
20

```

	Время выполнения, сек	Затраты памяти, МБ
1 4 5	0.000012	19.79
100 -744 647 -629... 279 365 -545...	0.00162	20.00

1000 73334 501893616... -64417 -58002 -95495...	0.0428	20.37
-------------------------------------------------------	--------	-------

Вывод по задаче: в этой задаче мы научились перемножать многочлены, используя принцип «разделяй и властвуй», а также алгоритм Карацубы.

Дополнительные задачи

Задача №2. Сортировка слиянием+ [N баллов]

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.

Решение:

```
def verification(n, array, attempt, source):

    res = 1
    if type(n) is int and 1 <= n <= 10 ** 5:
        if type(array) is list and n == len(array) and all(type(x) is int
and abs(x) <= 10 ** 9 for x in array):
            return merge_sort(0, n-1, array, source)
        else:
            res *= 0
    else:
        res *= 0
    if res == 0:
        if attempt == 3:
            return 'Ошибка!'

        else:
            print("Введите данные ещё раз, соблюдая ограничения: ")
            try:
                new_n = int(input())
                new_array = list(map(int, input().split(" ")))
                return verification(new_n, new_array, attempt + 1, '')
            except:
                return 'Ошибка!'

def merge_sort_main(*args):
    if len(args) == 1:
        path = args[0]
        file_input = open(path, 'r')
        len_arr = int(file_input.readline().strip())
        array = list(map(int, file_input.readline().strip().split(" ")))

        path = 'output' + path[5:]
        file_output = open(path, 'w')
        file_output.close()
        verification(len_arr, array, 1, path)
    else:
        len_arr, array = args
```

```

        return verification(len_arr, array, 1, '')

def merge_sort(start, end, array, source):

    middle = (start + end + 1) // 2
    list_a, list_b = array[start:middle], array[middle:end+1]

    len_a, len_b = middle - start, end + 1 - middle
    if len_a > 1:
        list_a = merge_sort(start, middle-1, array, source)
    if len_b > 1:
        list_b = merge_sort(middle, end, array, source)
    return merge(start, end, list_a, list_b, source)

def merge(start, end, array_a, array_b, source):

    array_c = []
    index_a, index_b = 0, 0
    for i in range(end - start + 1):
        if index_b == len(array_b):
            array_c.extend(array_a[index_a:])
            break
        elif index_a == len(array_a):
            array_c.extend(array_b[index_b:])
            break
        else:
            if array_a[index_a] <= array_b[index_b]:
                array_c.append(array_a[index_a])
                index_a += 1
            else:
                array_c.append(array_b[index_b])
                index_b += 1
    if len(source) == 0:
        print(start+1, end+1, *array_c)
    else:
        file = open(source, 'a')
        file.write(f'{start+1} {end+1} {' '.join(map(str, array_c))}\n')
        file.close()
    return array_c

```

Объяснение решения:

- 1) В функции merge_sort_main мы считываем данные из файла, если это необходимо, и проверяем их, вызывая функцию verification. Если данные удовлетворяют заданным ограничениям, то переходим к функции merge_sort, которая разделяет массив и вызывает функцию merge для половинок массива.
- 2) Мы немного изменили функцию merge из первого задания. Теперь помимо двух массивов она принимает на вход индекс начала и конца первоначального массива (до деления), а также source – источник данных, если это файл, то мы запишем результаты в него, если данные введены из терминала, то source = '' и мы выведем данные в консоль.
- 3) После сортировки мы выводим в консоль или записываем в файл индекс начала и конца части массива, а также эту, уже отсортированную часть.

Результат работы кода на примере из задачи:

	Время выполнения, сек	Затраты памяти, МБ
1 0	0.000011	19.62
1000 -349649722 987715729...	0.00109	20.02
10000 750431581 265752675 - 102766916...	0.01324	20.68
100000 -195194287 5553582...	0.1596	23.98

Задача №5. Представитель большинства [N баллов]

Текст задачи:

Правило большинства — это когда выбирается элемент, имеющий больше половины голосов.

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

- Формат входного файла (input.txt). В первой строке входного файла содержится число n ($1 \leq n \leq 10^5$) — число элементов в массиве. Во второй строке находятся n положительных целых чисел, по модулю не превосходящих 10^9 , $0 \leq a_i \leq 10^9$.

- Формат выходного файла (output.txt). Выведите 1, если во входной последовательности есть элемент, который встречается строго больше половины раз; в противном случае - 0.

Решение:

```
def frequent(high: int, array: tp.List[int]) -> int:
    dictionary = {}
    res = 1
    for index in range(high):
        elem = array[index]
        if elem in dictionary:
            dictionary[elem] += 1
        else:
            dictionary[elem] = 1
        if dictionary[elem] > high / 2:
            return 1
        else:
            res = 1
    if res:
        return 0
```

Объяснение решения:

Функция `frequent` принимает два аргумента: `high` (целое число) и `array` (список целых чисел). Она предназначена для определения, есть ли в массиве элемент, который встречается более чем в половине всех элементов массива.

Вот как работает функция:

- Создается пустой словарь `dictionary`, который будет использоваться для подсчета количества вхождений каждого элемента в массиве.
- Переменная `res` инициализируется значением 1, но в данном контексте она не используется должным образом.
- В цикле `for` перебираются элементы массива до индекса `high`.
 - Для каждого элемента `elem` проверяется, есть ли он уже в словаре: если да, то увеличивается его счетчик. Если нет, то элемент добавляется в словарь с начальным значением 1.
- Если количество вхождений элемента превышает половину размера массива ($high / 2$), функция немедленно возвращает 1, что означает, что такой элемент найден.
- Если ни один элемент не встречается более чем в половине массива, функция возвращает 0.

Таким образом, функция возвращает 1, если существует элемент, который встречается более чем в половине всех элементов массива, и 0 в противном случае.

Результат работы программы:

	данные	время, сек.	память, МБ	результат
Минимальные значения	1 0	0.0	31.64	1
Значения из примера	5 2 3 9 2 2	0.0	31.66	1
Значения из примера	4 1 2 3 4	0.0	31.66	0
Максимальные значения	100000 868256720 637618372 173848204 637618372	0.01	33.92	1
Максимальные значения	100000 -694496944 -350085909 -207223832 872254026	0.01	35.42	0

Вывод по задаче: Мы научились использовать словари в Python для эффективного подсчета количества вхождений элементов в массиве. Это позволяет быстро проверять, сколько раз каждый элемент встречается.

Задача №7. Поиск максимального подмассива за линейное время [N баллов]

Текст задачи: можно найти максимальный подмассив за линейное время, воспользовавшись следующими идеями. Начните с левого конца массива и двигайтесь вправо, отслеживая найденный к данному моменту максимальный подмассив. Зная максимальный подмассив массива $A[1..j]$, распространите ответ на поиск максимального подмассива, заканчивающегося индексом $j + 1$, воспользовавшись следующим наблюдением: максимальный подмассив массива $A[1..j + 1]$ представляет собой либо максимальный подмассив массива $A[1..j]$, либо подмассив $A[i..j + 1]$ для некоторого $1 \leq i \leq j + 1$. Определите максимальный подмассив вида $A[i..j + 1]$ за константное время, зная максимальный подмассив, заканчивающийся индексом j .

Решение:

```
def line_find_max_subarray_main(*args):
    if len(args) == 1:
        path = args[0]
        file_input = open(path, 'r')
        n = int(file_input.readline().strip())
        array = list(map(int, file_input.readline().strip().split(" ")))
        result = " ".join(map(str, line_find_max_subarray(n, array)))
        file_output = open('output' + path[5:], 'w')
        file_output.write(result)
        file_output.close()
    else:
        n, array = args
        line_find_max_subarray(n, array)
```



```

def line_find_max_subarray(n, array):
    if verification(n, array):
        max_sum = 0
        start_index = 0
        end_index = 0
        sums = 0
        for i in range(n):

            if sums == 0:
                start_index = i
            sums += array[i]
            if max_sum < sums:
                max_sum = sums
                end_index = i
            if sums < 0:
                sums = 0
        return start_index, end_index, max_sum

def verification(n, array, attempt=1):
    res = 1
    if type(n) is int and 1 <= n <= 10 ** 5:
        if type(array) is list and n == len(array) and all(type(x) is int
and abs(x) <= 10 ** 9 for x in array):
            res = 1
        else:
            res *= 0
    else:
        res *= 0
    if res == 0:
        if attempt == 3:
            return 'Ошибка!'

        else:
            print("Введите данные ещё раз, соблюдая ограничения: ")
            try:
                new_n = int(input())
                new_array = list(map(int, input().split(" ")))
                return verification(new_n, new_array, attempt + 1)
            except:
                return 'Ошибка!'
    else:
        return res

```

Объяснение решения:

- 1) В функции `line_find_max_subarray_main` мы считаем данные и передадим их в функцию `line_find_max_subarray`. В ней мы проверим данные с помощью функции `verification`, если они удовлетворяют условиям задачи, то перейдем к следующему алгоритму.
- 2) Создадим переменные: `max_sum` – максимальная сумма подмассива, `start_index` – индекс первого элемента подмассива, `end_index` – индекс последнего элемента подмассива, `sums` – сумма элементов подмассива на данный момент. Изначально все они равны нулю. Затем пройдемся в цикле по массиву, если сумма равно нулю, то обновим индекс начала, затем прибавим к сумме *i*-ый элемент массива. Если сумма стала больше максимальной, то обновим максимальную сумму и индекс последнего элемента, если сумма

стала отрицательной, то обнулим ее. После окончания цикла выведем индексы начала и конца подмассива, а также сумму элементов максимального подмассива.

Результат работы кода на примере:

```
input7.txt × task7.py output7.txt
1 10
2 -17 -14 -3 18 6 7 -10 20 -16 -17
output7.txt ×
1 3 7 41
```

Результат работы кода на минимальных значениях:

```
input7.txt ×
1 1
2 14
output7.txt ×
1 0 0 14
```

Результат работы кода на максимальных значениях:

```
input7.txt input7.txt × task7.py test7.py
100000
570496247 484711979 -773373644 644428553 847731437 36
-260282652 -433591703 340789315 -451966027 673703434
837242167 -952290561 -841993967 -387709819 147208911
915409675 824391261 579691215 592972977 313491242 69
output7.txt × input7.txt task
1 24670 56514 230842448993
```

	Время выполнения, сек	Затраты памяти, МБ
1 0	0.0000042	19.98
1000 484787009 - 933307925...	0.000145	20.00
100000 -370554242 -561616364...	0.0131	24.57

Вывод по задаче: мы научились искать максимальный подмассив за линейное время при помощи алгоритма Кадане.

Вывод по всей лабораторной

В ходе лабораторной работы №2 мы научились использовать быстрый алгоритм сортировки – сортировку слиянием, а также алгоритм поиска в отсортированном массиве – бинарный поиск. Также мы познакомились с алгоритмом Карацубы и Кадане и научились применять принцип «разделяй и властвуй» для разных задач.