Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №10
дисциплины «Алгоритмизация»

Выполнил:
Кожуховский Виктор Андреевич
2 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и
вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств
вычислительной
техники и автоматизированных систем
», очная форма обучения

_____
(подпись)

Руководитель практики:
Воронкин Роман Александрович

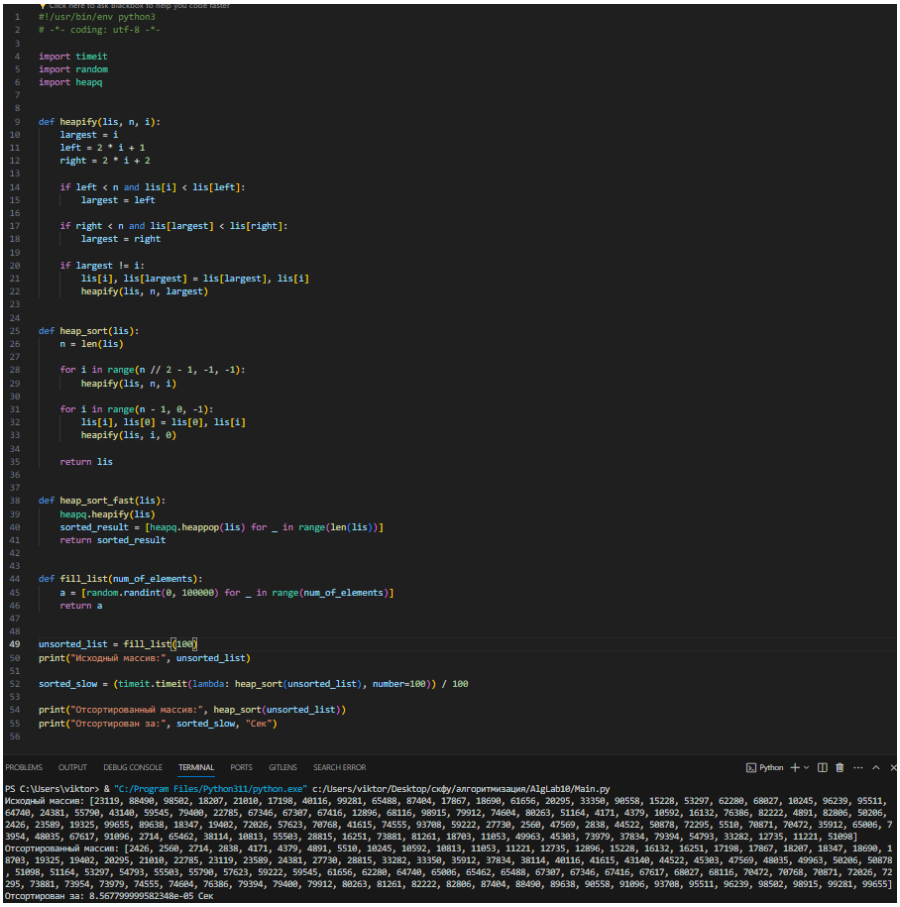
_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

Порядок выполнения работы:

Написал программу поиска элемента в массиве, автоматического заполнения массива, расчёта тысячи точек, показывающих время поиска элемента в массиве в худшем и среднем случае, вывода графиков, составленных из этих точек, и подсчета корреляции:



```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import timeit
import random
import heapq


def heapify(lis, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and lis[i] < lis[left]:
        largest = left

    if right < n and lis[largest] < lis[right]:
        largest = right

    if largest != i:
        lis[i], lis[largest] = lis[largest], lis[i]
        heapify(lis, n, largest)


def heap_sort(lis):
    n = len(lis)

    for i in range(n // 2 - 1, -1, -1):
        heapify(lis, n, i)

    for i in range(n - 1, 0, -1):
        lis[i], lis[0] = lis[0], lis[i]
        heapify(lis, i, 0)

    return lis


def heap_sort_fast(lis):
    heapq.heapify(lis)
    sorted_result = [heapq.heappop(lis) for _ in range(len(lis))]
    return sorted_result


def fill_list(num_of_elements):
    a = [random.randint(0, 100000) for _ in range(num_of_elements)]
    return a


unsorted_list = fill_list(100)
print("Исходный массив:", unsorted_list)

sorted_slow = (timeit.timeit(lambda: heap_sort(unsorted_list), number=100)) / 100

print("Отсортированный массив:", heap_sort(unsorted_list))
print("Отсортирован за:", sorted_slow, "Сек")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS  SEARCH ERROR

PS C:\Users\viktor> & "C:/Program Files/Python311/python.exe" c:/Users/viktor/Desktop/скфу/алгоритмизация/AlgLab10/Main.py
Исходный массив: [23119, 88490, 98502, 18207, 21010, 17198, 40116, 99281, 65488, 87404, 17867, 18690, 61656, 20295, 33350, 98558, 15228, 53297, 62280, 68027, 10245, 96239, 95511,
64740, 24381, 55790, 43140, 59545, 79400, 22785, 67346, 67307, 67416, 12896, 68116, 98915, 79912, 74604, 80263, 51164, 4171, 4379, 10592, 16132, 76386, 82222, 4891, 82806, 50206,
2426, 23589, 19325, 99655, 89638, 18347, 19402, 72026, 57623, 70768, 41615, 74555, 93708, 59222, 27730, 2560, 47569, 2838, 44522, 50878, 72295, 5510, 70871, 70472, 35912, 65006, 7
3954, 48035, 67617, 91096, 2714, 65462, 38114, 10813, 55503, 28815, 16251, 73881, 81261, 18703, 11053, 49963, 45303, 73979, 37834, 79394, 54793, 33282, 12735, 11221, 51098]
Отсортированный массив: [2426, 2560, 2714, 2838, 4171, 4379, 4891, 5510, 10245, 10592, 10813, 11053, 11221, 12735, 12896, 15228, 16132, 16251, 17198, 17867, 18207, 18347, 18690, 1
8703, 19325, 19402, 20295, 21010, 22785, 23119, 23589, 24381, 27730, 28815, 33282, 33350, 35912, 37834, 38114, 40116, 41615, 43140, 44522, 45303, 47569, 48035, 49963, 50206, 50878
, 51098, 51164, 53297, 54793, 55503, 55790, 57623, 59222, 59545, 61656, 62280, 64740, 65006, 65462, 65488, 67307, 67346, 67416, 67617, 68027, 68116, 70472, 70768, 70871, 72026, 72
295, 73881, 73954, 73979, 74555, 74604, 76386, 79394, 79400, 79912, 80263, 81261, 82222, 82806, 87404, 88490, 89638, 90558, 91096, 93708, 95511, 96239, 98502, 98915, 99281, 99655]
Отсортирован за: 8.567799999582348e-05 Сек
```

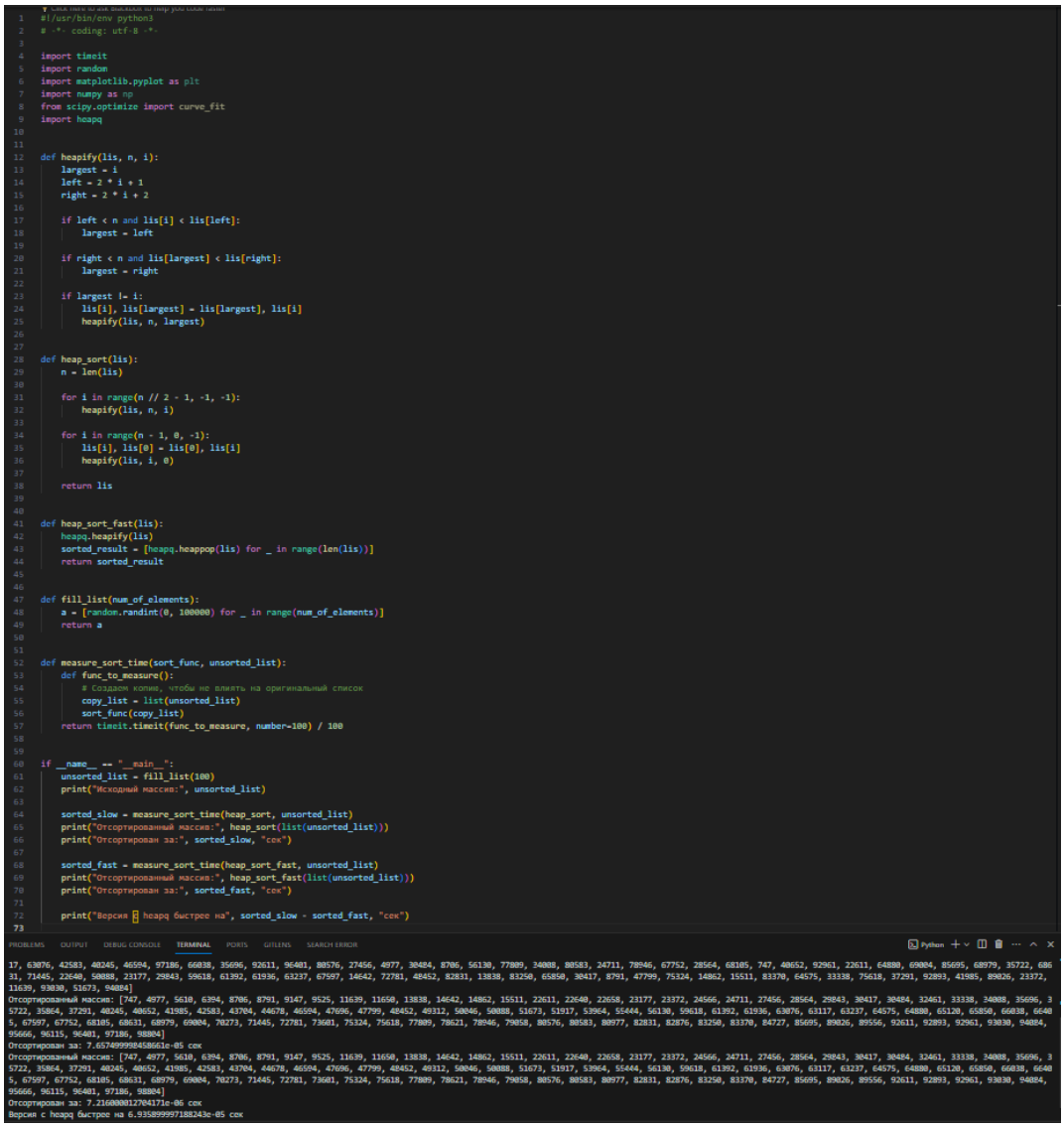
Рисунок 1. Код неоптимизированного алгоритма heapsort

Таблица 1. Сравнение алгоритма Heap Sort с Quick Sort и Merge Sort

Характеристика	Heap Sort	Quick Sort	Merge Sort
Сложность времени	O(n log n)	O(n^2) в худшем случае, O(n log n) в среднем	O(n log n)
Сложность по памяти	O(1) или O(log n)	O(log n) в среднем	O(n)
Стабильность	Нестабильная	Нестабильная	Стабильная
Необходимость доп. памяти	Нет или	O(log n) в среднем, но может быть O(n)	Нет
Лучший случай	O(n log n)	O(n log n)	O(n log n)
Худший случай	O(n log n)	O(n^2)	O(n log n)
Средний случай	O(n log n)	O(n log n)	O(n log n)

Heapsort не требует доп. память, занимает меньше всех места, но считается нестабильным. Quick Sort медленный в худшем случае, занимает больше места, требует доп. память и является нестабильным. Merge Sort стабилен, быстр, не требует доп памяти, но имеет наибольшую сложность по памяти.

Произвел оптимизацию алгоритма при помощи встроенной библиотеки heapq:



```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import timeit
import random
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
import heapq


def heapify(lis, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and lis[i] < lis[left]:
        largest = left

    if right < n and lis[largest] < lis[right]:
        largest = right

    if largest != i:
        lis[i], lis[largest] = lis[largest], lis[i]
        heapify(lis, n, largest)


def heap_sort(lis):
    n = len(lis)

    for i in range(n // 2 - 1, -1, -1):
        heapify(lis, n, i)

    for i in range(n - 1, 0, -1):
        lis[i], lis[0] = lis[0], lis[i]
        heapify(lis, i, 0)

    return lis


def heap_sort_fast(lis):
    heapq.heapify(lis)
    sorted_result = [heapq.heappop(lis) for _ in range(len(lis))]
    return sorted_result


def fill_list(num_of_elements):
    a = [random.randint(0, 100000) for _ in range(num_of_elements)]
    return a


def measure_sort_time(sort_func, unsorted_list):
    def func_to_measure():
        # Создаем копию, чтобы не влиять на оригинальный список
        copy_list = list(unsorted_list)
        sort_func(copy_list)
    return timeit.timeit(func_to_measure, number=100) / 100


if __name__ == "__main__":
    unsorted_list = fill_list(100)
    print("Исходный массив:", unsorted_list)

    sorted_slow = measure_sort_time(heap_sort, unsorted_list)
    print("Отсортированный массив:", heap_sort(list(unsorted_list)))
    print("Отсортирован за:", sorted_slow, "сек")

    sorted_fast = measure_sort_time(heap_sort_fast, unsorted_list)
    print("Отсортированный массив:", heap_sort_fast(list(unsorted_list)))
    print("Отсортирован за:", sorted_fast, "сек")

    print("Версия с heapq быстрее на", sorted_slow - sorted_fast, "сек")
```

```
17, 63076, 42583, 40245, 46594, 97186, 66038, 35696, 92611, 96481, 80576, 27456, 4977, 30484, 8706, 56130, 77809, 34088, 80583, 24711, 78946, 67752, 28564, 68105, 747, 40652, 92961, 22611, 64880, 69004, 85695, 68979, 35722, 686
31, 71445, 22640, 50888, 23177, 29843, 59618, 61392, 61936, 63237, 67597, 14642, 72781, 48452, 82831, 13838, 83250, 65850, 30417, 8791, 47799, 75324, 14862, 15511, 83370, 64575, 33338, 75618, 37291, 92893, 41985, 89026, 23372,
11639, 93030, 51673, 94084]
Отсортированный массив: [747, 4977, 5610, 6394, 8706, 8791, 9147, 9525, 11639, 11650, 13838, 14642, 14862, 15511, 22611, 22640, 22658, 23177, 23372, 24566, 24711, 27456, 28564, 29843, 30417, 30484, 32461, 33338, 34088, 35696, 3
5722, 35864, 37291, 40245, 40652, 41985, 42583, 43704, 44678, 46594, 47696, 47799, 48452, 49312, 50046, 50888, 51673, 51917, 53964, 55444, 56130, 59618, 61392, 61936, 63076, 63117, 63237, 64575, 64880, 65120, 65850, 66038, 6640
5, 67597, 67752, 68105, 68631, 68979, 69004, 70273, 71445, 72781, 73601, 75324, 75618, 77809, 78621, 78946, 79058, 80576, 80583, 80977, 82831, 82876, 83250, 83370, 84727, 85695, 89026, 89556, 92611, 92893, 92961, 93030, 94084,
95666, 96115, 96481, 97186, 98804]
Отсортирован за: 7.657499998458661e-05 сек
Отсортированный массив: [747, 4977, 5610, 6394, 8706, 8791, 9147, 9525, 11639, 11650, 13838, 14642, 14862, 15511, 22611, 22640, 22658, 23177, 23372, 24566, 24711, 27456, 28564, 29843, 30417, 30484, 32461, 33338, 34088, 35696, 3
5722, 35864, 37291, 40245, 40652, 41985, 42583, 43704, 44678, 46594, 47696, 47799, 48452, 49312, 50046, 50888, 51673, 51917, 53964, 55444, 56130, 59618, 61392, 61936, 63076, 63117, 63237, 64575, 64880, 65120, 65850, 66038, 6640
5, 67597, 67752, 68105, 68631, 68979, 69004, 70273, 71445, 72781, 73601, 75324, 75618, 77809, 78621, 78946, 79058, 80576, 80583, 80977, 82831, 82876, 83250, 83370, 84727, 85695, 89026, 89556, 92611, 92893, 92961, 93030, 94084,
95666, 96115, 96481, 97186, 98804]
Отсортирован за: 7.216080012704171e-06 сек
Версия с heapq быстрее на 6.935899997188243e-05 сек
```

Рисунок 2. Оптимизированный алгоритм heapsort

Применение в реальной жизни:

Системы с ограниченной памятью: Эффективен там, где важно минимизировать использование дополнительной памяти.

Приоритетные очереди: Используется для эффективного управления данными с приоритетом, например в обработке событий.

Потоковая обработка: Подходит для сортировки данных в реальном времени, когда данные постоянно поступают.

Базы данных: Помогает в сортировке больших объемов данных вне основной памяти, через внешнюю сортировку.

Вычислительные задачи с таймингом: Хорош для задач в реальном времени, где важна предсказуемость времени выполнения операций.

Heap Sort выбирают из-за надёжности и предсказуемости, когда стоит избегать риска существенного замедления из-за худшего случая выполнения, как, например, у Quick Sort. Также он полезен, если требуется сортировать данные без дополнительного пространства, например, при выполнении сортировки прямо на физических носителях или в ситуациях с ограниченной доступной памятью.

Анализ сложности:

Добавил рисование графиков зависимости времени сортировки с помощью оптимизированного и не оптимизированного heapsort от количества элементов в массиве.

Поскольку Heap Sort может выполняться in-place, для его работы теоретически не требуется дополнительное пространство кроме самого массива, который сортируется. При этом методе сортировки:

- Не требуется выделять дополнительный массив для разделения данных.
- Манипуляции с элементами осуществляются в пределах того же массива.
- Требуется ограниченное количество переменных для хранения индексов и временных значений в процессе выполнения алгоритма.

Heap Sort возможно реализовать без рекурсии, при этом алгоритм получается с чистой пространственной сложностью O(1), то есть он будет занимать константное дополнительное пространство, не зависимо от размеров

входных данных. Эта модификация использует только циклы и не требует дополнительной памяти для рекурсивного стека, что делает пространственную сложность чисто константной.

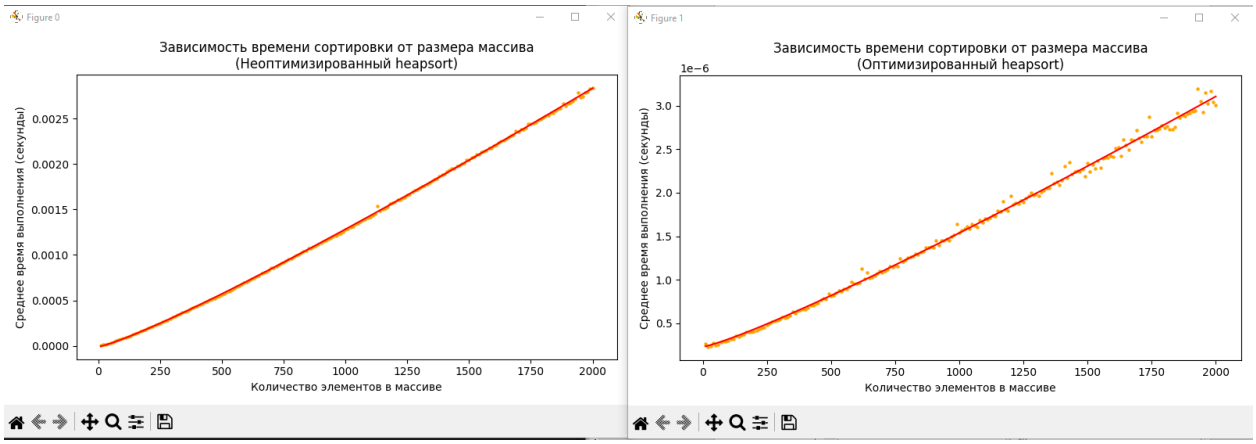


Рисунок 3. Графики зависимости времени сортировки с помощью оптимизированного и не оптимизированного heapsort от количества элементов в массиве

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import timeit
import random
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
import heapq


amount_of_dots = 100  # Количество точек
aod = (amount_of_dots + 1) * 10
median_time = {}
graph_stuff = [i for i in range(10, aod, 10)]
xlabel = "Количество элементов в массиве"
ylabel = "Среднее время выполнения (секунды)"


def heapify(lis, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and lis[i] < lis[left]:
        largest = left

    if right < n and lis[largest] < lis[right]:
        largest = right

    if largest != i:
        lis[i], lis[largest] = lis[largest], lis[i]
        heapify(lis, n, largest)


def heap_sort(lis):
    n = len(lis)

    for i in range(n // 2 - 1, -1, -1):
        heapify(lis, n, i)

    for i in range(n - 1, 0, -1):
        lis[i], lis[0] = lis[0], lis[i]
        heapify(lis, i, 0)

    return lis


def heap_sort_fast(lis):
    heapq.heapify(lis)
    sorted_result = [heapq.heappop(lis) for _ in range(len(lis))]
    return sorted_result


def fill_list(num_of_elements):
    a = [random.randint(0, 100000) for _ in range(num_of_elements)]
    return a


def measure_sort_time(sort_func, unsorted_list):
    def func_to_measure():
        # Создаем копию, чтобы не влиять на оригинальный список
        copy_list = list(unsorted_list)
        sort_func(copy_list)
    return timeit.timeit(func_to_measure, number=100) / 100


def n_log_n_model(x, a, b):
    return a * x * np.log(x) + b


def lsm(name, time, graph_num):
    plt.figure(graph_num).set_figwidth(8)
    plt.title(
        f"Зависимость времени сортировки от размера массива\n({name})")
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.tight_layout()
    plt.grid(False)

    x_data = np.array(graph_stuff)
    y_data = np.array(list(time.values()))

    params, _ = curve_fit(n_log_n_model, x_data, y_data)

    a_fit, b_fit = params
    print(f"Коэффициенты уравнения ({name}): a = {a_fit}, b = {b_fit}")
    x_fit = np.linspace(min(x_data), max(x_data), 100)
    y_fit = n_log_n_model(x_fit, *params)

    plt.plot(x_fit, y_fit, "r-", label="n log n Fit")

    plt.scatter(graph_stuff, time.values(), s=5, c="orange")
    plt.tight_layout()


def results(name, func, graph_index):
    for i in range(10, aod, 10):
        a = fill_list(i)
        median_time[i] = timeit.timeit(lambda: func(a),
                                       number=100) / 100

    lsm(name, median_time, graph_index)


if __name__ == "__main__":
    unsorted_list = fill_list(10000)

    sorted_slow = measure_sort_time(heap_sort, unsorted_list)
    print("Отсортирован обычным heapsort за:", sorted_slow, "сек")

    sorted_fast = measure_sort_time(heap_sort_fast, unsorted_list)
    print("Отсортирован heapsort с heapq за:", sorted_fast, "сек")

    print("Версия с heapq быстрее на", sorted_slow - sorted_fast, "сек")

    results("Неоптимизированный heapsort", heap_sort, 0)
    results("Оптимизированный heapsort", heap_sort_fast, 1)

    plt.show()
```

Рисунок 4. Полный код с двумя алгоритмами heapsort и выводом графиков скорости их сортировки

Решение задания:

```python
import heapq
import random


def print_ascending_sums(A, B):

    # Сортируем оба массива
    A.sort()
    B.sort()

    # min-heap хранит выражения вида (сумма, индекс в A, индекс в B)
    heap = [(A[i] + B[0], i, 0) for i in range(len(A))]
    heapq.heapify(heap)

    # Вывод суммы в возрастающем порядке
    for _ in range(len(A)**2):
        sum, i, j = heapq.heappop(heap)
        print(sum, end=' ')
        if j + 1 < len(B):
            heapq.heappush(heap, (A[i] + B[j+1], i, j+1))


def fill_list(num_of_elements):
    a = [random.randint(1, 10) for _ in range(num_of_elements)]
    return a


n = 5
A = fill_list(n)
B = fill_list(n)
print(sorted(A), "\n", sorted(B))
print_ascending_sums(A, B)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS  SEARCH ERROR
PS C:\Users\viktor> & "C:/Program Files/Python311/python.exe" "c:/Users/viktor/De
[1, 6, 8, 9, 10]
 [4, 4, 8, 8, 10]
5 5 9 9 10 10 11 12 12 13 13 14 14 14 14 16 16 16 17 17 18 18 18 19 20
```

Рисунок 5. Код решения задания 6 и его вывод

Вывод: в результате выполнения лабораторной работы был изучен алгоритм heap sort и проведено исследование зависимости времени поиска от количества элементов в массиве, показавшее что зависимость время поиска линейно увеличивается с добавлением элементов в массив.