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»

Выполнила:
Михеева Елена Александровна
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 Р.А., канд. техн. наук,
Доцент кафедры инфокоммуникаций

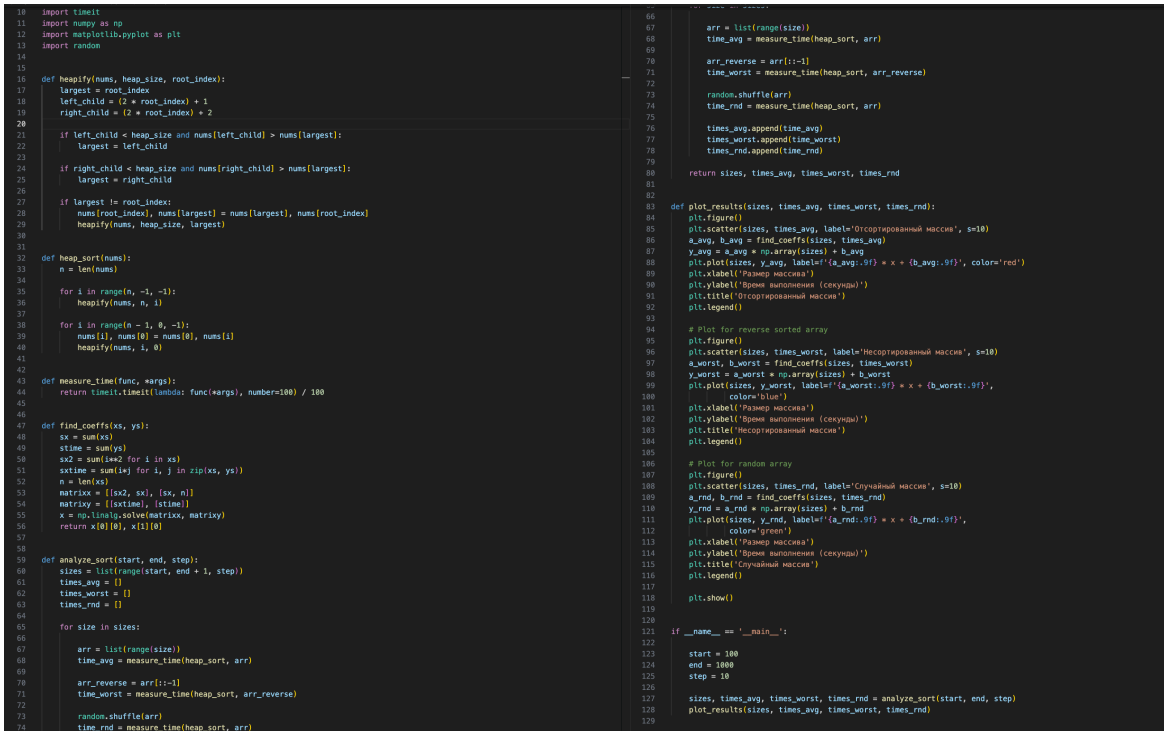
_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

Порядок выполнения работы.

1. Была реализована программа, которая использует алгоритм сортировки Heap Sort — это алгоритм сортировки, который использует структуру данных под названием "куча" (heap). Куча представляет собой дерево, где каждый узел имеет значение не меньше (или не больше, в зависимости от задачи) значений его дочерних узлов. В случае с сортировкой кучей, это "макс-куча" (max-heap), где значение в каждом узле больше или равно значения его дочерних узлов. В программе heap_sort.py было исследовано время работы алгоритма в случаях, когда на вход идут: отсортированный массив, массив обратный отсортированному и массив с рандомными значениями.



```python
import timeit
import numpy as np
import matplotlib.pyplot as plt
import random


def heapify(nums, heap_size, root_index):
    largest = root_index
    left_child = (2 * root_index) + 1
    right_child = (2 * root_index) + 2

    if left_child < heap_size and nums[left_child] > nums[largest]:
        largest = left_child

    if right_child < heap_size and nums[right_child] > nums[largest]:
        largest = right_child

    if largest != root_index:
        nums[root_index], nums[largest] = nums[largest], nums[root_index]
        heapify(nums, heap_size, largest)


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

    for i in range(n, -1, -1):
        heapify(nums, n, i)

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


def measure_time(func, *args):
    return timeit.timeit(lambda: func(*args), number=100) / 100


def find_coeffs(xs, ys):
    sx = sum(xs)
    stime = sum(ys)
    sx2 = sum(i**2 for i in xs)
    sxtime = sum(i*j for i, j in zip(xs, ys))
    n = len(xs)
    matrixx = [[sx2, sx], [sx, n]]
    matrixy = [[sxtime], [stime]]
    x = np.linalg.solve(matrixx, matrixy)
    return x[0][0], x[1][0]


def analyze_sort(start, end, step):
    sizes = list(range(start, end + 1, step))
    times_avg = []
    times_worst = []
    times_rnd = []

    for size in sizes:

        arr = list(range(size))
        time_avg = measure_time(heap_sort, arr)

        arr_reverse = arr[::-1]
        time_worst = measure_time(heap_sort, arr_reverse)

        random.shuffle(arr)
        time_rnd = measure_time(heap_sort, arr)

        times_avg.append(time_avg)
        times_worst.append(time_worst)
        times_rnd.append(time_rnd)

    return sizes, times_avg, times_worst, times_rnd


def plot_results(sizes, times_avg, times_worst, times_rnd):
    plt.figure()
    plt.scatter(sizes, times_avg, label='Отсортированный массив', s=10)
    a_avg, b_avg = find_coeffs(sizes, times_avg)
    y_avg = a_avg * np.array(sizes) + b_avg
    plt.plot(sizes, y_avg, label=f'{a_avg:.9f} * x + {b_avg:.9f}', color='red')
    plt.xlabel('Размер массива')
    plt.ylabel('Время выполнения (секунды)')
    plt.title('Отсортированный массив')
    plt.legend()

    # Plot for reverse sorted array
    plt.figure()
    plt.scatter(sizes, times_worst, label='Несортированный массив', s=10)
    a_worst, b_worst = find_coeffs(sizes, times_worst)
    y_worst = a_worst * np.array(sizes) + b_worst
    plt.plot(sizes, y_worst, label=f'{a_worst:.9f} * x + {b_worst:.9f}',
             color='blue')
    plt.xlabel('Размер массива')
    plt.ylabel('Время выполнения (секунды)')
    plt.title('Несортированный массив')
    plt.legend()

    # Plot for random array
    plt.figure()
    plt.scatter(sizes, times_rnd, label='Случайный массив', s=10)
    a_rnd, b_rnd = find_coeffs(sizes, times_rnd)
    y_rnd = a_rnd * np.array(sizes) + b_rnd
    plt.plot(sizes, y_rnd, label=f'{a_rnd:.9f} * x + {b_rnd:.9f}',
             color='green')
    plt.xlabel('Размер массива')
    plt.ylabel('Время выполнения (секунды)')
    plt.title('Случайный массив')
    plt.legend()

    plt.show()


if __name__ == '__main__':

    start = 100
    end = 1000
    step = 10

    sizes, times_avg, times_worst, times_rnd = analyze_sort(start, end, step)
    plot_results(sizes, times_avg, times_worst, times_rnd)
```

Рисунок 1. Код программы heap_sort.py

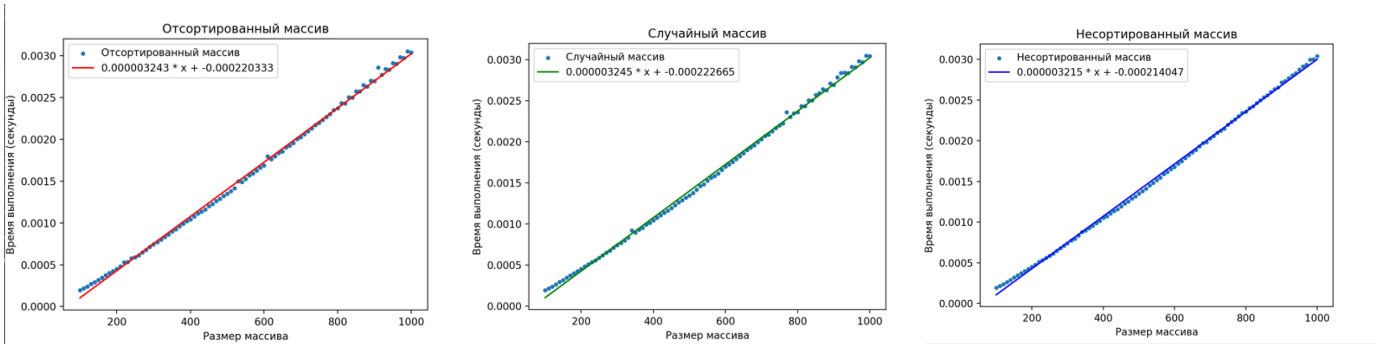


Рисунок 2. Графики зависимости длины массива от времени работы

2. Был проведен сравнительный анализ сортировки кучей с другими методами сортировки.

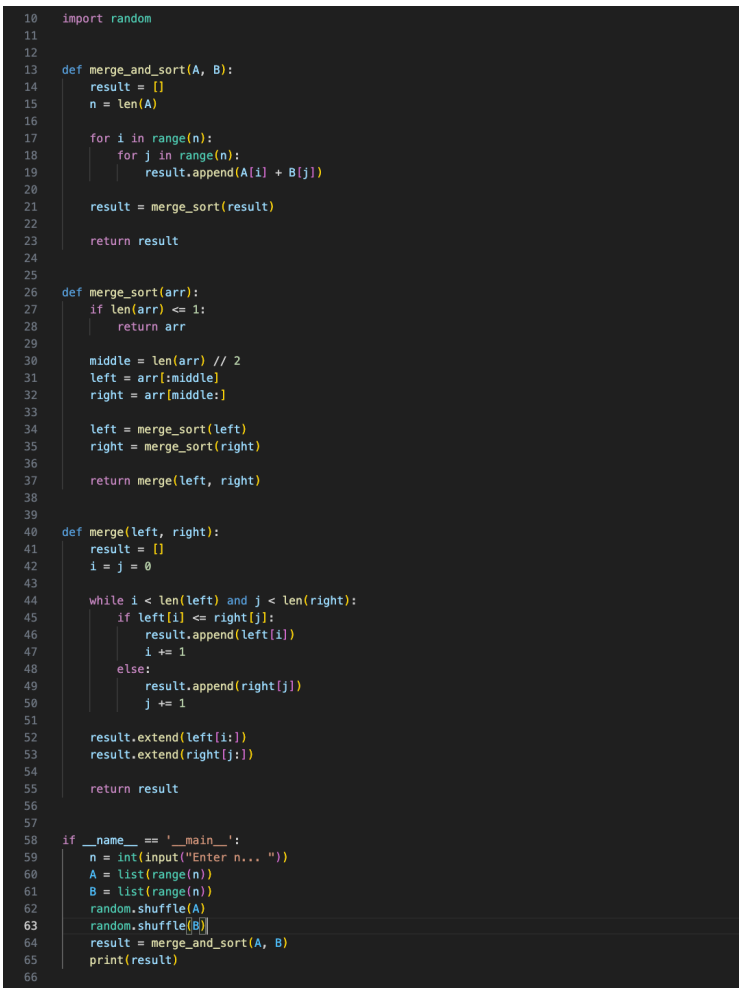
Сортировка	Лучший случай	Средний случай	Худший случай
Heap Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)

Сортировка Heap Sort всегда имеет временную сложность O(n log n) в любом случае, так как строит кучу и извлекает максимальные элементы.

Сортировка Quick Sort в лучшем и среднем случаях, имеет временную сложность O(n log n), но в худшем случае может быть O(n^2). Однако, оптимизации могут сделать его эффективным.

Сортировка Merge Sort также имеет временную сложность O(n log n) в любом случае. Этот метод сортировки эффективен и стабилен, но требует дополнительной памяти для буфера при слиянии

3. Была написана программа для решения задачи: Даны массивы A[1...n] и B[1...n]. Мы хотим вывести все n2 сумм вида A[i]+B[j] в возрастающем порядке. Наивный способ — создать массив, содержащий все такие суммы, и отсортировать его. Соответствующий алгоритм имеет время работы O(n^2logn) и использует O(n^2) памяти. Приведите алгоритм с таким же временем работы, который использует линейную память.



```python
import random


def merge_and_sort(A, B):
    result = []
    n = len(A)

    for i in range(n):
        for j in range(n):
            result.append(A[i] + B[j])

    result = merge_sort(result)

    return result


def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    middle = len(arr) // 2
    left = arr[:middle]
    right = arr[middle:]

    left = merge_sort(left)
    right = merge_sort(right)

    return merge(left, right)


def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result


if __name__ == '__main__':
    n = int(input("Enter n... "))
    A = list(range(n))
    B = list(range(n))
    random.shuffle(A)
    random.shuffle(B)
    result = merge_and_sort(A, B)
    print(result)

```

Рисунок 3. Код программы task6.py

4. Временная сложность данного кода будет составлять O(n^2 log n), так как вложенные циклы привносят O(n^2), а вызовы merge_sort и merge обеспечивают O(log n) на каждом уровне рекурсии.