

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №5
по курсу «Алгоритмы и структуры данных»
Тема: Пирамидальная сортировка
Вариант 5

Выполнил:
Артемов И. В.
К3141

Проверил:
Афанасьев А. В.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №1. Куча ли?	3
Задача №6: Очередь с приоритетами	7
Дополнительные задачи	12
Задача №4. Построение пирамиды	12
Задача №7. Снова сортировка	18
Вывод	22

Задачи по варианту

Задача №1. Куча ли?

Текст задачи.

1 задача. Куча ли?

Структуру данных «куча», или, более конкретно, «неубывающая пирамида», можно реализовать на основе массива.

Для этого должно выполняться основное свойство неубывающей пирамиды, которое заключается в том, что для каждого $1 \leq i \leq n$ выполняются условия:

1. если $2i \leq n$, то $a_i \leq a_{2i}$,
2. если $2i + 1 \leq n$, то $a_i \leq a_{2i+1}$.

Дан массив целых чисел. Определите, является ли он неубывающей пирамидой.

- **Формат входного файла (input.txt).** Первая строка входного файла содержит целое число n ($1 \leq n \leq 10^6$). Вторая строка содержит n целых чисел, по модулю не превосходящих $2 \cdot 10^9$.
- **Формат выходного файла (output.txt).** Выведите «YES», если массив является неубывающей пирамидой, и «NO» в противном случае.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Примеры:

№	input.txt	output.txt
1	5 1 0 1 2 0	NO
2	5 1 3 2 5 4	YES

Листинг кода.

```
# Импортируем библиотеки для отслеживания памяти и времени выполнения
программы
import tracemalloc
import time
import os
from lab5.utils import open_file, write_file, delete_prev_values,
get_output_path, print_output_file

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

current_dir = os.path.dirname(os.path.abspath(__file__)) # Текущая
директория
txtf_dir = os.path.join(os.path.dirname(current_dir), "txtf") # Директория
для input.txt и output.txt
input_path = os.path.join(txtf_dir, "input.txt")
```

```

def is_heap(array, n):
    """
    Проверяет, является ли массив неубывающей пирамидой.

    :param array: массив целых чисел
    :param n: размер массива
    :return: True, если массив - пирамида, иначе False
    """
    for i in range(n):
        left_child_index = 2 * (i + 1) - 1 # Индекс левого потомка
        right_child_index = 2 * (i + 1) # Индекс правого потомка

        # Проверяем условие для левого потомка
        if left_child_index < n and array[i] > array[left_child_index]:
            return False

        # Проверяем условие для правого потомка
        if right_child_index < n and array[i] > array[right_child_index]:
            return False

    return True

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    n_str, array_str = open_file(input_path)
    n = int(n_str.strip()) # Преобразуем первую строку в число n
    array = list(map(int, array_str.strip().split())) # Преобразуем
элементы массива в числа
    # Проверка корректности входных данных
    if (1 <= n <= 10 ** 6) and (all(abs(i) <= 2 * 10 ** 9 for i in array)):
        print(f"\nTask: 1\nInput:\n{n}\n{array}")
        delete_prev_values(1)

        # Проверяем массив на условие пирамиды
        result = "YES" if is_heap(array, n) else "NO"

        # Записываем результат в файл output.txt
        output_path = get_output_path(1)
        write_file(result, output_path)
        print_output_file(1)
    else:
        # Выводим сообщение об ошибке, если данные некорректны
        print('Введите корректные данные')

    # Выводим время работы программы
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))
    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

    # Останавливаем отслеживание памяти
    tracemalloc.stop()

```

Текстовое объяснение решения.

1. Импорт библиотек: `tracemalloc`, `time`, `os`, и функций из `lab5.utils`.
2. Запуск таймера для измерения времени выполнения.
3. Включение отслеживания памяти с помощью `tracemalloc`.
4. Определение текущей директории.
5. Определение пути к директории с файлами `input.txt` и `output.txt`.

6. Определение пути к файлу `input.txt`.
 7. Определение функции `is_heart` для проверки массива на соответствие свойствам пирамиды.
 8. Чтение данных из файла `input.txt` через `open_file`.
 9. Преобразование первой строки данных в целое число `n`.
 10. Преобразование второй строки данных в массив целых чисел.
 11. Проверка корректности входных данных: диапазон для `n` и элементов массива.
 12. Вывод задачи и входных данных, если они корректны.
 13. Очистка файла результатов для задачи 1 с помощью `delete_prev_values`.
 14. Проверка массива на соответствие свойствам пирамиды с использованием `is_heart`.
 15. Запись результата проверки в файл `output.txt`.
 16. Вывод содержимого файла результатов с помощью `print_output_file`.
 17. Вывод сообщения об ошибке, если входные данные некорректны.
 18. Вывод времени выполнения программы.
 19. Вывод объема использованной памяти.
 20. Остановка отслеживания памяти с помощью `tracemalloc.stop()`.
- Результат работы кода на примере из задачи:

task1.py input.txt × output.txt	
1	5
2	1 3 2 5 4

task1.py input.txt output.txt ×	
1	YES

Результат работы кода на максимальных и минимальных значениях:

task1.py input.txt × output.txt	
1	1
2	1

task1.py input.txt output.txt ×	
1	YES

The image shows two screenshots of a code editor interface. The top screenshot shows the 'input.txt' file with two lines of numbers. The bottom screenshot shows the 'output.txt' file with one line containing the word 'NO'.

```

task1.py  input.txt  output.txt
1 1000000
2 -252498802 -798391215 -808086011 -63994196 600622245 73906282 -9
  983823412 837629406 398991896 -875464930 -384805335 405347886 -

task1.py  input.txt  output.txt
1 NO
  
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0004983750404790044 секунд	15324 байт
Пример из задачи	0.0008398750214837492 секунд	15374 байт
Верхняя граница диапазона значений входных данных из текста задачи	0.4109084579977207 секунд	10765530 байт

Вывод по задаче: мною был изучен алгоритм определения, является ли массив неубывающей пирамидой.

Задача №6: Очередь с приоритетами

Текст задачи.

6 задача. Очередь с приоритетами

Реализуйте очередь с приоритетами. Ваша очередь должна поддерживать следующие операции: добавить элемент, извлечь минимальный элемент, уменьшить элемент, добавленный во время одной из операций.

- **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 10^6$) - число операций с очередью.

Следующие n строк содержат описание операций с очередью, по одному описанию в строке. Операции могут быть следующими:

- $A\ x$ – требуется добавить элемент x в очередь.
- X – требуется удалить из очереди минимальный элемент и вывести его в выходной файл. Если очередь пуста, в выходной файл требуется вывести звездочку «*».
- $D\ x\ y$ – требуется заменить значение элемента, добавленного в очередь операцией A в строке входного файла номер $x + 1$, на y . Гарантируется, что в строке $x + 1$ действительно находится операция A , что этот элемент не был ранее удален операцией X , и что y меньше, чем предыдущее значение этого элемента.

В очередь помещаются и извлекаются только целые числа, не превышающие по модулю 10^9 .

- **Формат выходного файла (output.txt).** Выведите последовательно результат выполнения всех операций X , по одному в каждой строке выходного файла. Если перед очередной операцией X очередь пуста, выведите вместо числа звездочку «*».
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
8	2
A 3	1
A 4	3
A 2	*
X	
D 2 1	
X	
X	
X	

Листинг кода.

```
# Импортируем библиотеки для отслеживания памяти и времени выполнения
программы
import time
import tracemalloc
from lab5.utils import *

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

current_dir = os.path.dirname(os.path.abspath(__file__)) # Текущая
директория
txtf_dir = os.path.join(os.path.dirname(current_dir), "txtf") # Путь к
директории с файлами
input_path = os.path.join(txtf_dir, "input.txt")

# Основная функция обработки операций
def process_priority_queue(operations):
    import heapq

    heap = [] # Минимальная куча
    element_map = {} # Словарь для отслеживания актуальных значений
    id_map = {} # Словарь: идентификатор строки → элемент
    removed = set() # Множество удаленных элементов
    result = [] # Для хранения выходных данных

    current_id = 0 # Уникальный идентификатор для операций A

    for i, operation in enumerate(operations):
        parts = operation.split()
        if parts[0] == "A":
            # Добавляем элемент в кучу
            x = int(parts[1])
            heapq.heappush(heap, (x, current_id))
            element_map[current_id] = x
            id_map[i + 1] = current_id # i+1 соответствует строке x + 1
            current_id += 1

        elif parts[0] == "X":
            # Удаляем минимальный элемент
            while heap and heap[0][1] in removed:
                heapq.heappop(heap) # Пропускаем удаленные элементы

            if heap:
                _, element_id = heapq.heappop(heap)
                result.append(str(element_map[element_id]))
                removed.add(element_id)
            else:
                result.append("*")

        elif parts[0] == "D":
            # Уменьшаем значение элемента
            x = int(parts[1]) + 1
            y = int(parts[2])
            element_id = id_map[x] # Получаем идентификатор элемента

            # Уменьшаем значение в словаре
            element_map[element_id] = y

            # Добавляем в кучу новое значение
```



```

        heapq.heappush(heap, (y, element_id))

    return result

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    lines = open_file(input_path)
    n = int(lines[0]) # Число операций
    operations = lines[1:] # Список операций

    # Проверка корректности входных данных
    if 1 <= n <= 10 ** 6 and all(len(op.split()) in (1, 2, 3) for op in
operations):
        print(f"\nTask: 6\nInput:\n{n}\n{operations}\n")
        delete_prev_values(6) # Очищаем предыдущие результаты

        # Обрабатываем операции
        results = process_priority_queue(operations)

        output_path = get_output_path(6)
        # Записываем результаты операций X в файл output.txt
        write_file("\n".join(results), output_path)
        print_output_file(6)
    else:
        print("Введите корректные данные")

    # Выводим время работы программы
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))
    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

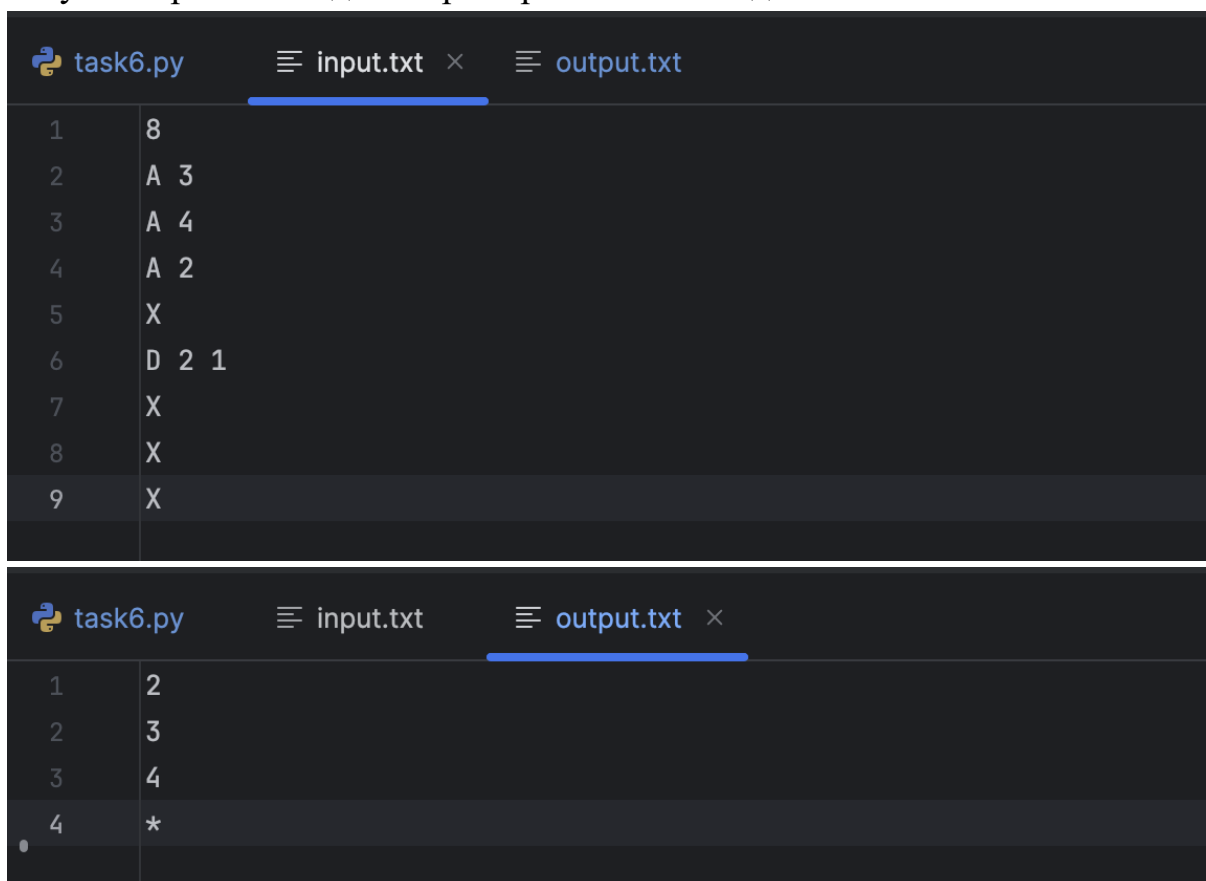
    # Останавливаем отслеживание памяти
    tracemalloc.stop()

```

Текстовое объяснение решения.

1. Импорт библиотек: `time`, `tracemalloc` и функций из `lab5.utils`.
2. Запуск таймера для измерения времени выполнения программы.
3. Включение отслеживания памяти с помощью `tracemalloc`.
4. Определение текущей директории.
5. Определение пути к директории с файлами `input.txt` и `output.txt`.
6. Определение пути к файлу `input.txt`.
7. Определение функции `process_priority_queue` для обработки операций с очередью с приоритетом.
8. Инициализация минимальной кучи.
9. Инициализация словаря для отслеживания значений элементов.
10. Инициализация словаря для соответствия идентификаторов строк и элементов.
11. Инициализация множества для удаленных элементов.
12. Инициализация списка для хранения результатов операций `X`.
13. Обработка операций `A` (добавление элемента).
14. Обработка операций `X` (удаление минимального элемента).

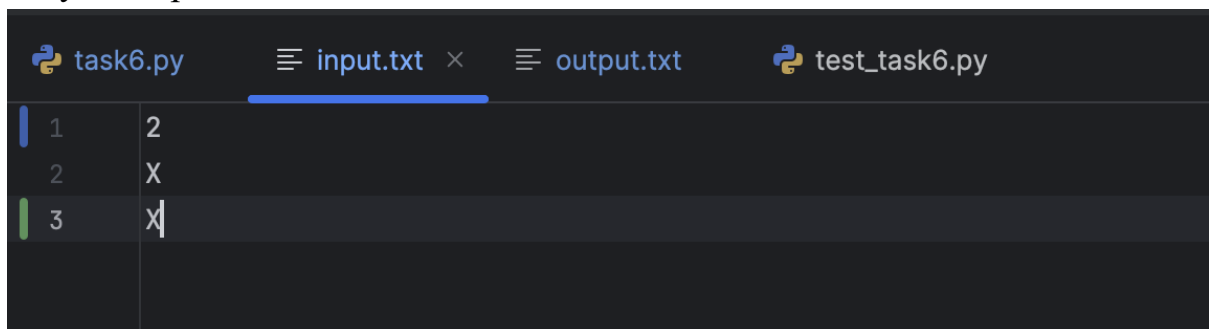
15. Обработка операций 'D' (изменение значения элемента).
 16. Возврат результатов операций.
 17. Чтение данных из файла 'input.txt' через 'open_file'.
 18. Преобразование первой строки в целое число 'n'.
 19. Формирование списка операций.
 20. Проверка корректности входных данных.
 21. Вывод задачи и входных данных, если они корректны.
 22. Очистка предыдущих результатов задачи 6 с помощью 'delete_prev_values'.
 23. Вызов функции 'process_priority_queue' для обработки операций.
 24. Запись результатов операций 'X' в файл 'output.txt'.
 25. Вывод содержимого файла результатов через 'print_output_file'.
 26. Вывод сообщения об ошибке, если входные данные некорректны.
 27. Вывод времени выполнения программы.
 28. Вывод объема использованной памяти.
 29. Остановка отслеживания памяти с помощью 'tracemalloc.stop()'.
- Результат работы кода на примере из текста задачи:



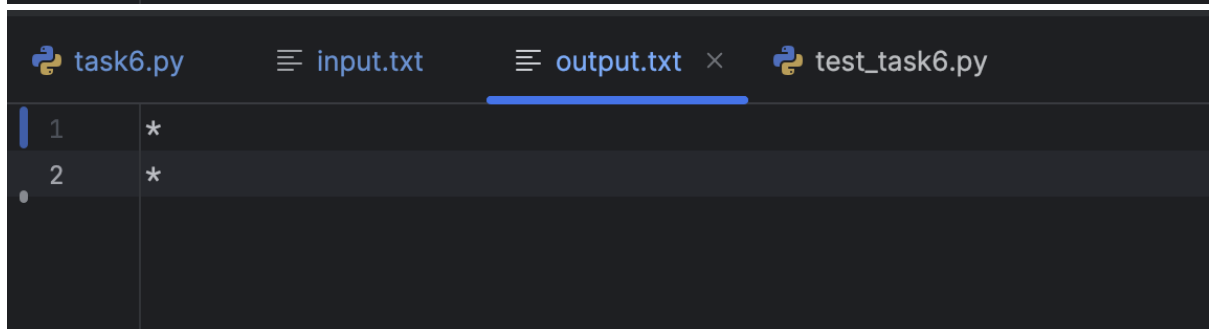
The image shows two screenshots of a code editor interface. The top screenshot shows the 'input.txt' file with 9 lines of input: 8, A 3, A 4, A 2, X, D 2 1, X, X, X. The bottom screenshot shows the 'output.txt' file with 4 lines of output: 2, 3, 4, *.

Line	Input	Output
1	8	2
2	A 3	3
3	A 4	4
4	A 2	*
5	X	
6	D 2 1	
7	X	
8	X	
9	X	

Результат работы кода на максимальных и минимальных значениях:



```
task6.py  input.txt  output.txt  test_task6.py
1 2
2 X
3 X
```



```
task6.py  input.txt  output.txt  test_task6.py
1 *
2 *
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0029671660158783197 секунд	52359 байт
Пример из задачи	0.0030947919585742056 секунд	52739 байт
Верхняя граница диапазона значений входных данных из текста задачи	0.22705929099993227 секунд	430401 байт

Вывод по задаче: мною был изучен алгоритм реализации очереди с приоритетами.

Дополнительные задачи

Задача №4. Построение пирамиды

Текст задачи.

4 задача. Построение пирамиды

В этой задаче вы преобразуете массив целых чисел в пирамиду. Это важнейший шаг алгоритма сортировки под названием HeapSort. Гарантированное время работы в худшем случае составляет $O(n \log n)$, в отличие от *среднего* времени работы QuickSort, равного $O(n \log n)$. QuickSort обычно используется на практике, потому что обычно он быстрее, но HeapSort используется для внешней сортировки, когда вам нужно отсортировать огромные файлы, которые не помещаются в памяти вашего компьютера.

Первым шагом алгоритма HeapSort является создание пирамиды (heap) из массива, который вы хотите отсортировать.

Ваша задача - реализовать этот первый шаг и преобразовать заданный массив целых чисел в пирамиду. Вы сделаете это, применив к массиву определенное количество перестановок (swaps). Перестановка - это операция, как вы помните, при которой элементы a_i и a_j массива меняются местами для некоторых i и j . Вам нужно будет преобразовать массив в пирамиду, используя только $O(n)$ перестановок. Обратите внимание, что в этой задаче вам нужно будет использовать min-heap вместо max-heap.

- **Формат ввода или входного файла (input.txt).** Первая строка содержит целое число n ($1 \leq n \leq 10^5$), вторая содержит n целых чисел a_i входного массива, разделенных пробелом ($0 \leq a_i \leq 10^9$, все a_i - различны.)

- **Формат выходного файла (output.txt).** Первая строка ответа должна содержать целое число m - количество сделанных свопов. Число m должно удовлетворять условию $0 \leq m \leq 4n$. Следующие m строк должны содержать по 2 числа: индексы i и j сделанной перестановки двух элементов, **индексы считаются с 0**. После всех перестановок в нужном порядке массив должен стать пирамидой, то есть для каждого i при $0 \leq i \leq n-1$ должны выполняться условия:

1. если $2i + 1 \leq n - 1$, то $a_i < a_{2i+1}$,
2. если $2i + 2 \leq n - 1$, то $a_i < a_{2i+2}$.

Обратите внимание, что все элементы входного массива различны. Любая последовательность свопов, которая менее $4n$ и после которой входной массив становится корректной пирамидой, считается верной.

- Ограничение по времени. 3 сек.
- Ограничение по памяти. 512 мб.
- Пример 1:

input.txt	output.txt
5	3
5 4 3 2 1	1 4
	0 1
	1 3

После перестановки элементов в позициях 1 и 4 массив становится следующим: 5 1 3 2 4.

Далее, перестановка элементов с индексами 0 и 1: 1 5 3 2 4. И напоследок, переставим 1 и 3: 1 2 3 5 4, и теперь это корректная неубывающая пирамида.

- Пример 2:

input.txt	output.txt
5	0
1 2 3 4 5	

Листинг кода.

```
# Импортируем библиотеки для отслеживания памяти и времени выполнения
# программы
import tracemalloc
import time
from lab5.utils import *

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

# Определяем пути к файлам
current_dir = os.path.dirname(os.path.abspath(__file__)) # Текущая
директория task1/src
txtf_dir = os.path.join(os.path.dirname(current_dir), "txtf") # Директория
task1/txtf
```

```

input_path = os.path.join(txtf_dir, "input.txt")

def build_min_heap(data):
    """Преобразует массив в min-heap и возвращает список свопов"""
    n = len(data)
    swaps = []

    def sift_down(i):
        """Функция просеивания вниз"""
        min_index = i
        left = 2 * i + 1 # индекс левого ребенка
        right = 2 * i + 2 # индекс правого ребенка

        # Если левый ребенок существует и меньше текущего узла
        if left < n and data[left] < data[min_index]:
            min_index = left

        # Если правый ребенок существует и меньше текущего минимального
        if right < n and data[right] < data[min_index]:
            min_index = right

        # Если текущий узел не минимальный, обмениваем местами и продолжаем
        if i != min_index:
            swaps.append((i, min_index)) # Запоминаем перестановку
            data[i], data[min_index] = data[min_index], data[i] # Перестановка
            sift_down(min_index) # Рекурсивно просеиваем вниз

    # Начинаем обработку с последнего родителя
    for i in range((n - 2) // 2, -1, -1):
        sift_down(i)

    return swaps

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    lines = open_file(input_path)
    n_str = lines[0].strip() # Убираем пробелы и символы новой строки из
    # первой строки
    m = lines[1].strip().split() # Разделяем вторую строку на элементы

    n = int(n_str) # Преобразуем первую строку в число n
    m = list(map(int, m)) # Преобразуем элементы массива в числа

    # Проверка корректности входных данных
    if (1 <= n <= 10 ** 5) and (all(0 <= i <= 10 ** 9 for i in m)):
        print(f"\nTask: 4\nInput:\n{n}\n{m}")
        delete_prev_values(4)

        # Строим min-heap и получаем список перестановок
        swaps = build_min_heap(m)

        output_path = get_output_path(4)
        # Формируем содержимое выходного файла
        output_data = [f"{len(swaps)}"] + [f"{i} {j}" for i, j in swaps]
        write_file("\n".join(output_data), output_path)
        print_output_file(4)
    else:
        # Выводим сообщение об ошибке, если данные некорректны
        print('Введите корректные данные')

```

```
# Выводим время работы программы
print("Время работы: %s секунд" % (time.perf_counter() - t_start))
# Выводим количество памяти, затраченной на выполнение программы
print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Текстовое объяснение решения.

1. Импорт библиотек: `tracemalloc`, `time`, и функций из `lab5.utils`.
2. Запуск таймера для измерения времени выполнения программы.
3. Включение отслеживания памяти с помощью `tracemalloc`.
4. Определение текущей директории.
5. Определение пути к директории с файлами `input.txt` и `output.txt`.
6. Определение пути к файлу `input.txt`.
7. Определение функции `build_min_heap` для преобразования массива в min-heap и сбора перестановок.
8. Определение вспомогательной функции `sift_down` для просеивания элементов вниз.
9. Просеивание узлов массива с последнего родительского элемента до корня.
10. Возврат списка перестановок из функции `build_min_heap`.
11. Чтение данных из файла `input.txt` через `open_file`.
12. Удаление пробелов и символов новой строки из первой строки данных.
13. Разделение второй строки данных на элементы.
14. Преобразование первой строки в целое число `n`.
15. Преобразование элементов второй строки в массив целых чисел.
16. Проверка корректности входных данных: диапазон для `n` и элементов массива.
17. Вывод задачи и входных данных, если они корректны.
18. Очистка предыдущих результатов задачи 4 с помощью `delete_prev_values`.
19. Вызов функции `build_min_heap` для построения min-heap и получения перестановок.
20. Определение пути к файлу результатов через `get_output_path`.
21. Формирование содержимого выходного файла с количеством перестановок и их парами.
22. Запись содержимого выходного файла через `write_file`.
23. Вывод содержимого файла результатов через `print_output_file`.
24. Вывод сообщения об ошибке, если входные данные некорректны.
25. Вывод времени выполнения программы.

26. Вывод объема использованной памяти.

27. Остановка отслеживания памяти с помощью `tracemalloc.stop()`.

Результат работы кода на примере из текста задачи:

```
task4.py  input.txt ×  output.txt  test_task4.py
1 5
2 5 4 3 2 1
```

```
task4.py  input.txt  output.txt ×  test_task4.py
1 3
2 1 4
3 0 1
4 1 3|
```

Результат работы кода на максимальных и минимальных значениях:

```
task4.py  input.txt ×  output.txt  test_task4.py
1 1
2 1|
```

```
task4.py  input.txt  output.txt ×  test_task4.py
1 0
```

```
task4.py  input.txt ×  output.txt  test_task4.py
1 100000
2 415135442 -429385671 323606245 315245939 -694721042 -790315520 34
  554101888 10453088 488652050 -192513403 157143581 959741818 -989
  -898358691 -667513100 455156951 -981707647 -722021365 -797944180
  88081266 835801821 756981885 -863479629 674754646 -406821374 670
```

	Время выполнения	Затраты памяти
--	------------------	----------------

Нижняя граница диапазона значений входных данных из текста задачи	0.0005116250249557197 секунд	15324 байт
Пример из задачи	0.0007557080243714154 секунд	15374 байт
Верхняя граница диапазона значений входных данных из текста задачи	0.1820174169843085 секунд	10481917 байт

Вывод по задаче: мною был изучен алгоритм построения пирамиды.

Задача №7. Снова сортировка

Текст задачи.

7 задача. Снова сортировка

Напишите программу пирамидальной сортировки на Python для последовательности в **убывающем порядке**. Проверьте ее, создав несколько случайных массивов, подходящих под параметры:

- **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 различных целых чисел, *по модулю* не превосходящих 10^9 .
- **Формат выходного файла (output.txt).** Одна строка выходного файла с отсортированным по невозрастанию массивом. Между любыми двумя числами должен стоять ровно один пробел.

Листинг кода.

```
import tracemalloc
import time
from lab5.utils import *
import os

# Запуск таймера для измерения времени выполнения
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

current_dir = os.path.dirname(os.path.abspath(__file__)) # Директория
task/src
txtf_dir = os.path.join(os.path.dirname(current_dir), "txtf") # Директория
task/txtf
input_path = os.path.join(txtf_dir, "input.txt")

def heapify(arr, n, i):
    """
    Вспомогательная функция для преобразования массива в max-heap.
    """
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

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

def heapsort(arr):
    """
    Пирамидальная сортировка массива.
    """
    n = len(arr)
```

```

# Построение max-heap
for i in range(n // 2 - 1, -1, -1):
    heapify(arr, n, i)

# Извлечение элементов из кучи
for i in range(n - 1, 0, -1):
    arr[0], arr[i] = arr[i], arr[0]
    heapify(arr, i, 0)

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    lines = open_file(input_path)
    n_str, m_str = lines[0].strip(), lines[1].strip()
    n = int(n_str)
    m = list(map(int, m_str.split())) # Преобразуем строки в список чисел

    # Проверка корректности входных данных
    if (1 <= n <= 10 ** 5) and (all(abs(i) <= 10 ** 9 for i in m)):
        print(f"\nTask 7\nInput:\n{n}\n{m}")
        delete_prev_values(7)

        # Сортируем массив m с помощью heapsort
        heapsort(m)
        m.reverse() # Инвертируем массив для получения убывающего порядка

        output_path = get_output_path(7)
        # Записываем результат в файл output.txt
        write_file(" ".join(map(str, m)), output_path)
        print_output_file(7)
    else:
        # Выводим сообщение об ошибке, если входные данные некорректны
        print('Введите корректные данные')

    # Выводим время выполнения программы
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))
    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

    # Останавливаем отслеживание памяти
    tracemalloc.stop()

```

Текстовое объяснение решения.

1 Импорт модулей:

- tracemalloc для отслеживания памяти.
- time для измерения времени выполнения.
- Функции из lab5.utils для работы с файлами.
- os для работы с файловой системой.

2 Инициализация замеров производительности:

- Запуск таймера с помощью time.perf_counter().
- Включение отслеживания памяти через tracemalloc.start().

3 Определение директорий:

- Получение текущей директории файла.
- Определение пути к директории txtf, содержащей файлы для ввода/вывода.

- Установка пути к файлу input.txt.

4 Реализация функций:

- `heapify(arr, n, i):`
 - Просеивает элемент вниз, чтобы преобразовать массив в `max-heap`.
 - Сравнивает узел с его левым и правым дочерними элементами и выполняет перестановки для сохранения структуры `max-heap`.
- `heapsort(arr):`
 - Преобразует массив в `max-heap`.
 - Последовательно извлекает наибольший элемент, перемещая его в конец массива, и восстанавливает `max-heap` на оставшейся части.

5 Основной блок программы:

- Чтение данных:
 - Входные данные читаются из файла `input.txt` с помощью `open_file`.
 - Первую строку преобразуют в число `n`, а вторую — в список чисел `m`.
- Проверка корректности данных:
 - Условие: $1 \leq n \leq 105$, $1 \leq m \leq 105$, все элементы массива `m` находятся в пределах $-10^9 \leq x \leq 10^9$.
 - Если данные некорректны, выводится сообщение об ошибке.
- Сортировка данных:
 - Сортировка массива `m` с использованием `heapsort`.
 - Инвертирование массива для получения убывающего порядка.
- Запись результата:
 - Результат записывается в файл `output.txt` через `write_file`.
 - Выводится содержимое выходного файла через `print_output_file`.



6 Вывод результатов производительности:

- Выводится время выполнения программы.
- Выводится максимальный объем памяти, использованной программой.



7 Завершение отслеживания памяти:

- Выключение отслеживания памяти с помощью `tracemalloc.stop()`.

Результат работы кода на примере из текста задачи:



 task7.py
 test_task7.py
☐ input.txt
☒ output.txt

15
21 2 3 4 5|



 task7.py
 test_task7.py
☐ input.txt
☒ output.txt

15 4 3 2 1|



Результат работы кода на максимальных и минимальных значениях:

 task7.py
 test_task7.py
☐ input.txt
☒ output.txt

11
21

 task7.py
 test_task7.py
☐ input.txt
☒ output.txt

11

 task7.py
 test_task7.py
☐ input.txt
☒ output.txt

11000000
2411622652 -274351482 -808057034 446216934 -959785437 -368795371 76
-273013715 -446851885 -302785270 304523685 484316502 -408695969 6

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0005130420322529972 секунд	15484 байт
Пример из задачи	0.0007835410069674253 секунд	15534 байт
Верхняя граница диапазона значений входных данных из текста задачи	0.19008412497350946 секунд	11521614 байт

Вывод по задаче: мною был изучен алгоритм пирамидальной сортировки.

Вывод

В ходе лабораторной работы был изучен алгоритм пирамидальной сортировки.