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: Деревья. Пирамида, пирамидальная сортировка. Очередь с приоритетами

Выполнила:

Просветова Валерия Дмитриевна

К3141

Проверила:

Афанасьев.А.В.

Санкт-Петербург

2024г.

# Содержание отчета

Задача №1: Куча ли ?

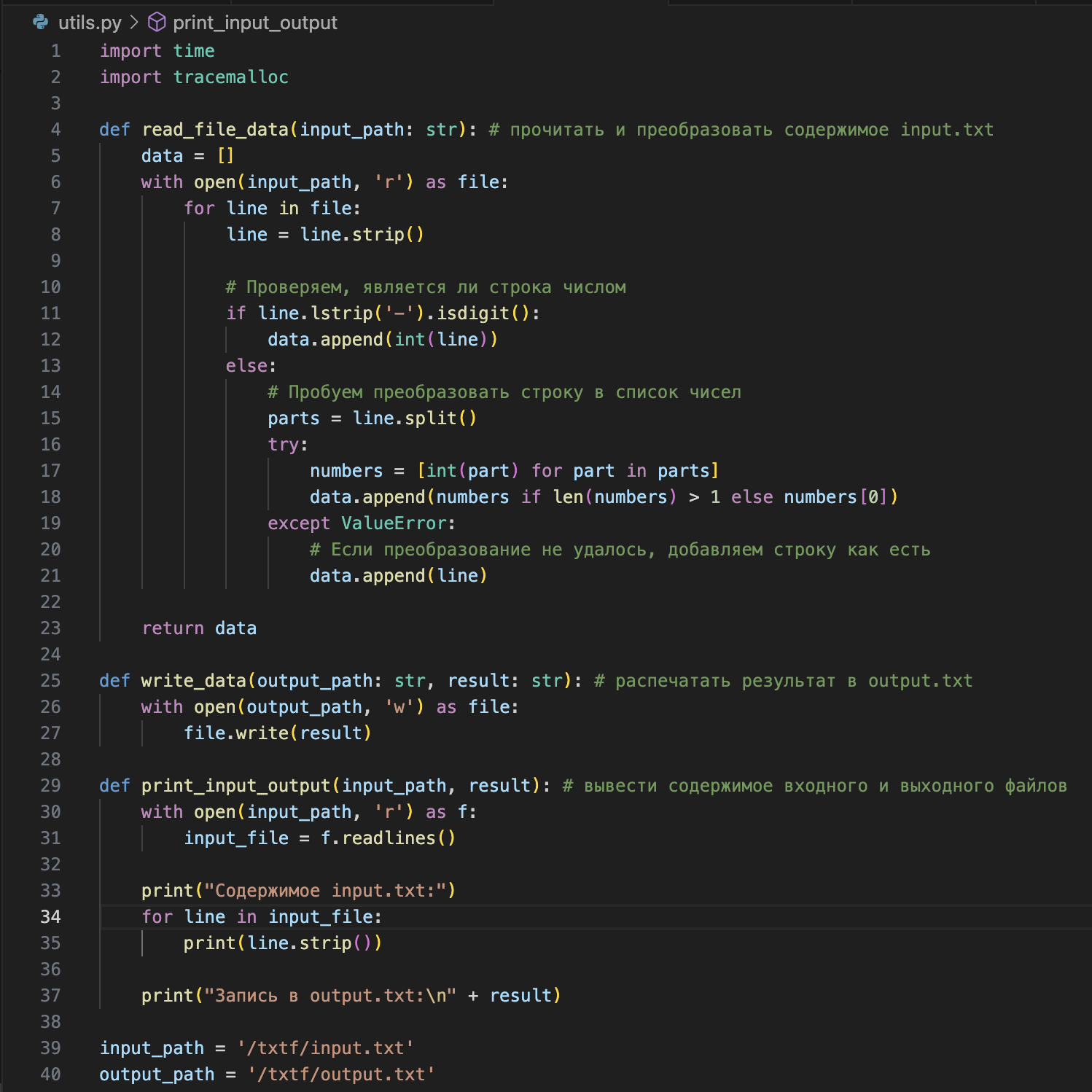
Задача №5: Планировщик заданий

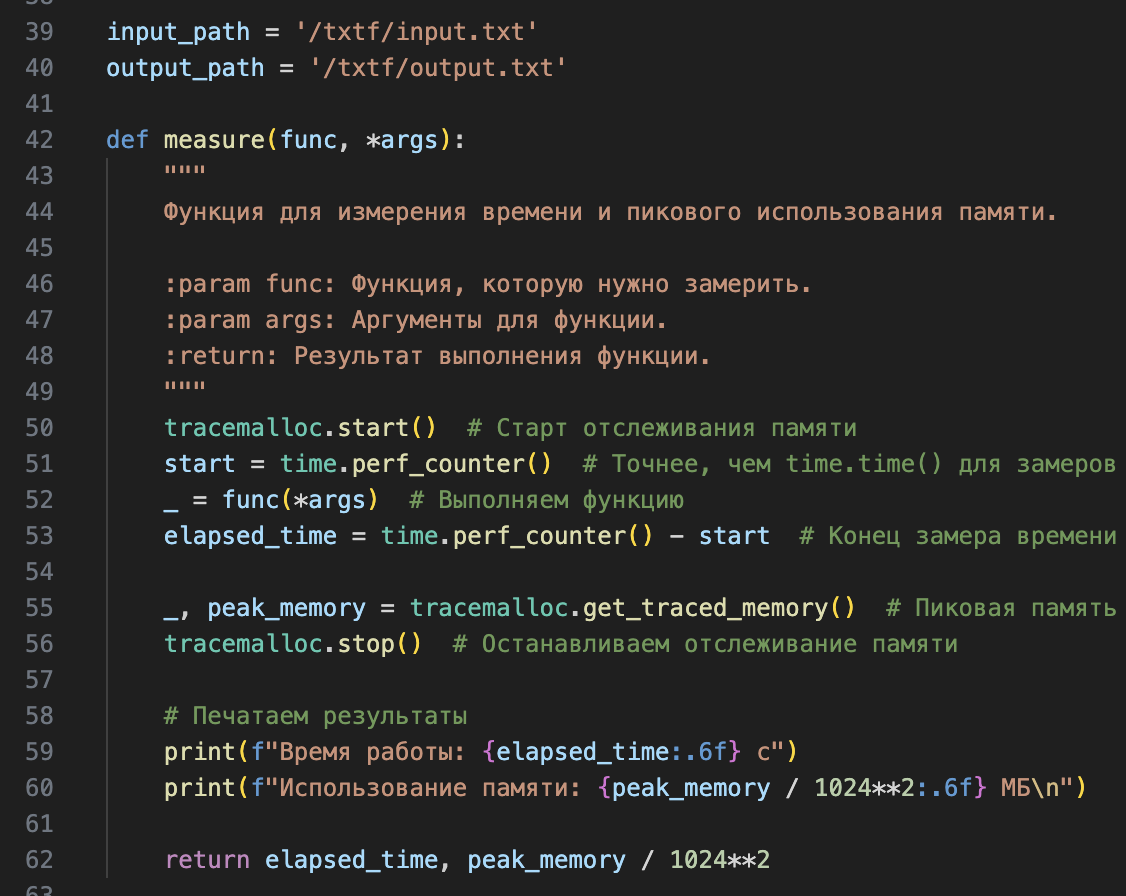
Задача №6: Очередь с приоритетами

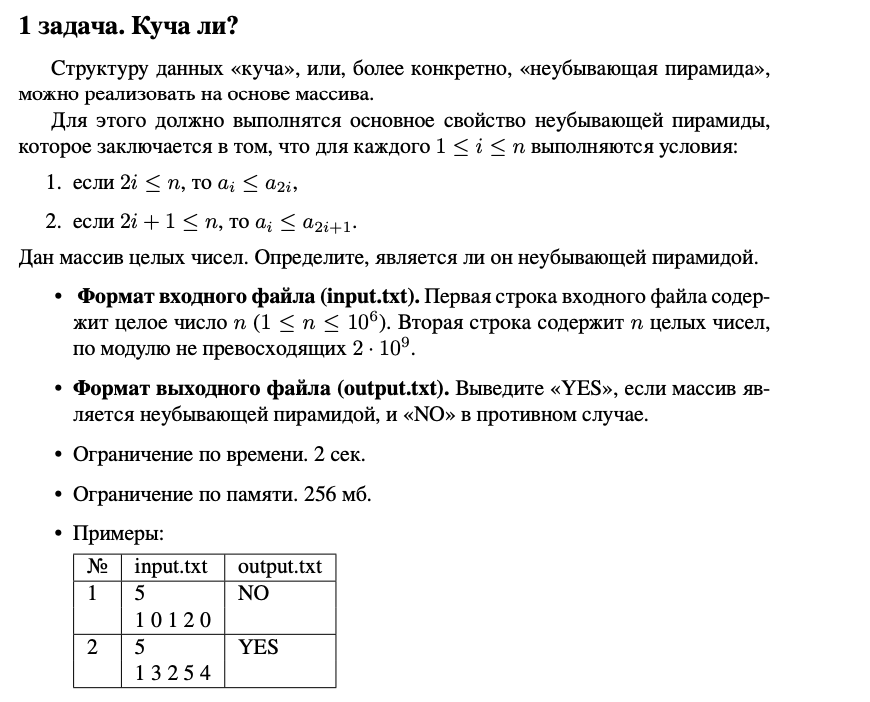
Задача №7: Снова сортировка

Вывод

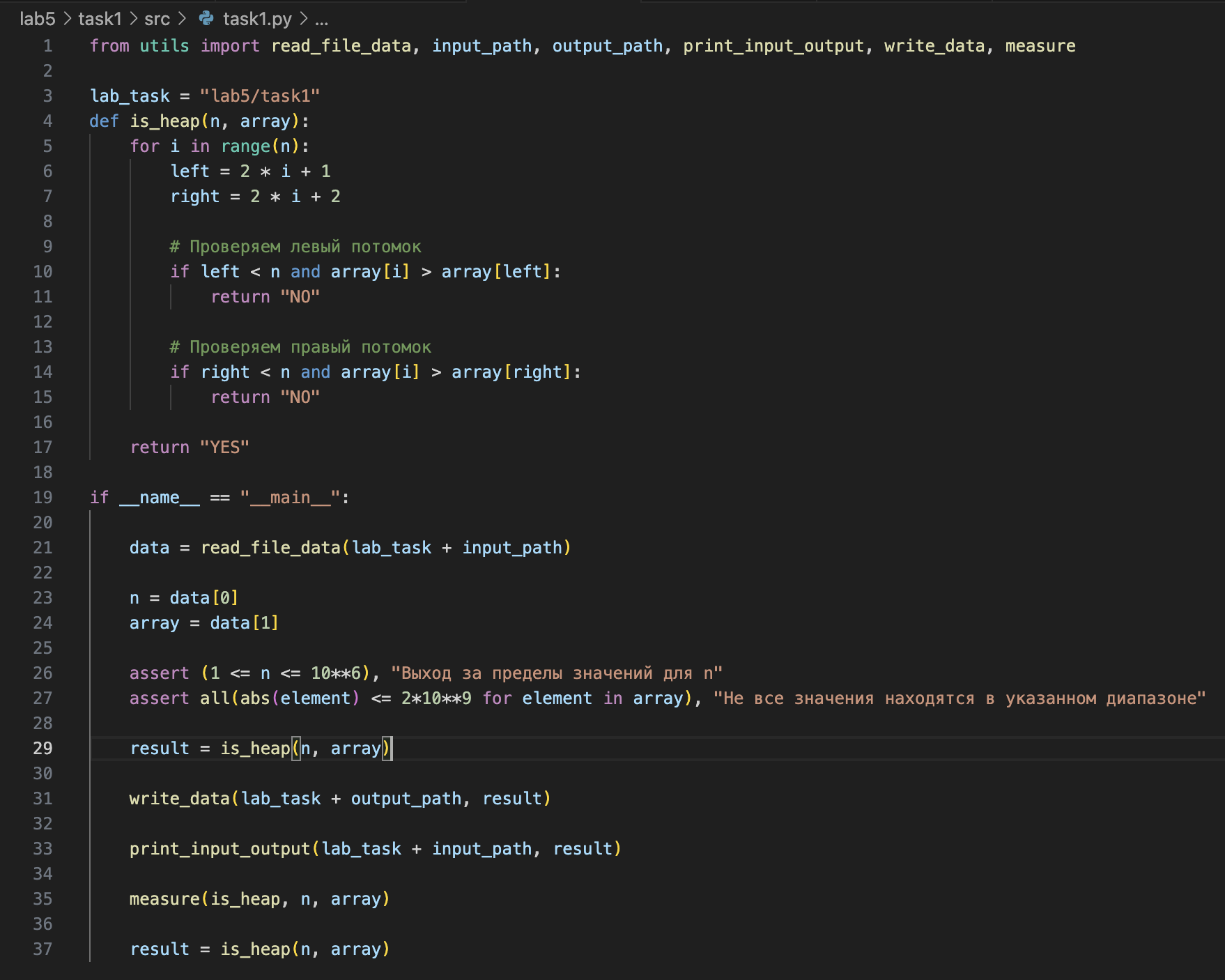
Использовала файл utils.py для исключения повторения строчек кода:





Задача№1:

Решение:



Функция is\_heap(n, array)

Эта функция принимает два аргумента:

1. n — количество элементов в массиве.
2. array — сам массив, который нужно проверить на соответствие свойству минимальной кучи.

Цикл for i in range(n):

Цикл проходит по всем узлам массива. Каждый элемент массива можно рассматривать как узел дерева:

* Узел с индексом i имеет:
  + Левого потомка с индексом 2 \* i + 1
  + Правого потомка с индексом 2 \* i + 2

*left = 2 \* i + 1*

Вычисляется индекс левого потомка текущего узла с индексом i.

*right = 2 \* i + 2*

Вычисляется индекс правого потомка текущего узла с индексом i.

Проверка левого потомка

*if left < n and array[i] > array[left]:*

*return "NO"*

1. Проверяем, существует ли левый потомок. Для этого сравниваем left с n (количество элементов в массиве). Если left >= n, значит у текущего узла нет левого потомка.
2. Если левый потомок существует, проверяем, больше ли значение текущего узла (array[i]), чем значение левого потомка (array[left]).
   * Если это условие выполняется, то массив нарушает свойство минимальной кучи (где родительское значение должно быть меньше или равно значениям потомков).
   * В этом случае функция сразу возвращает "NO".

Проверка правого потомка

*if right < n and array[i] > array[right]:*

*return "NO"*

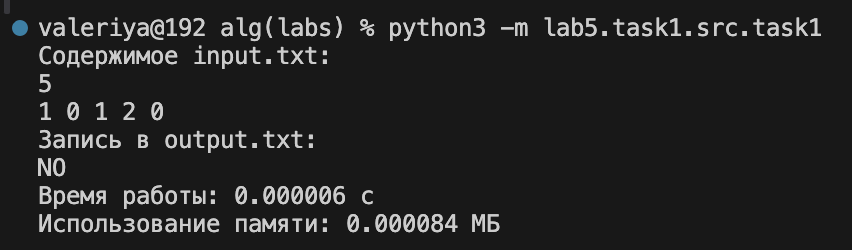
1. Проверяем, существует ли правый потомок. Для этого сравниваем right с n. Если right >= n, значит у текущего узла нет правого потомка.
2. Если правый потомок существует, проверяем, больше ли значение текущего узла (array[i]), чем значение правого потомка (array[right]).
   * Если это условие выполняется, то массив нарушает свойство минимальной кучи.
   * В этом случае функция сразу возвращает "NO".

Если все проверки пройдены

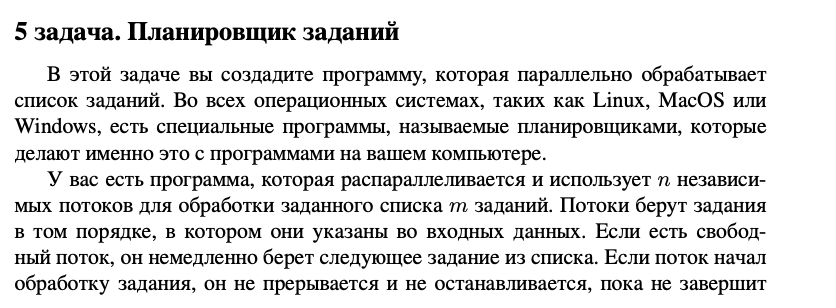
*return "YES"*

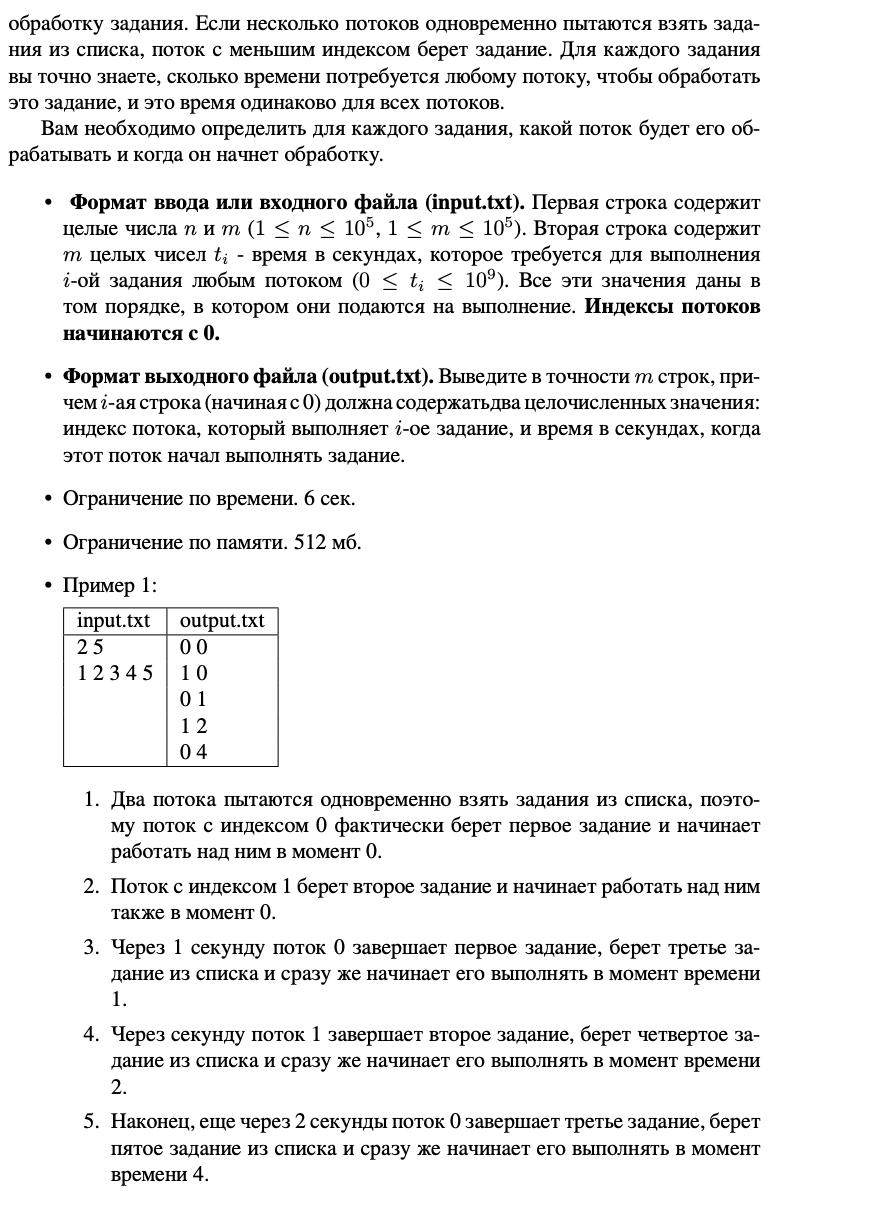
Если цикл завершился, и ни одно из условий не нарушилось (значение каждого узла было меньше или равно значениям его потомков), то массив соответствует свойству минимальной кучи, и функция возвращает "YES".

Запуск программы:

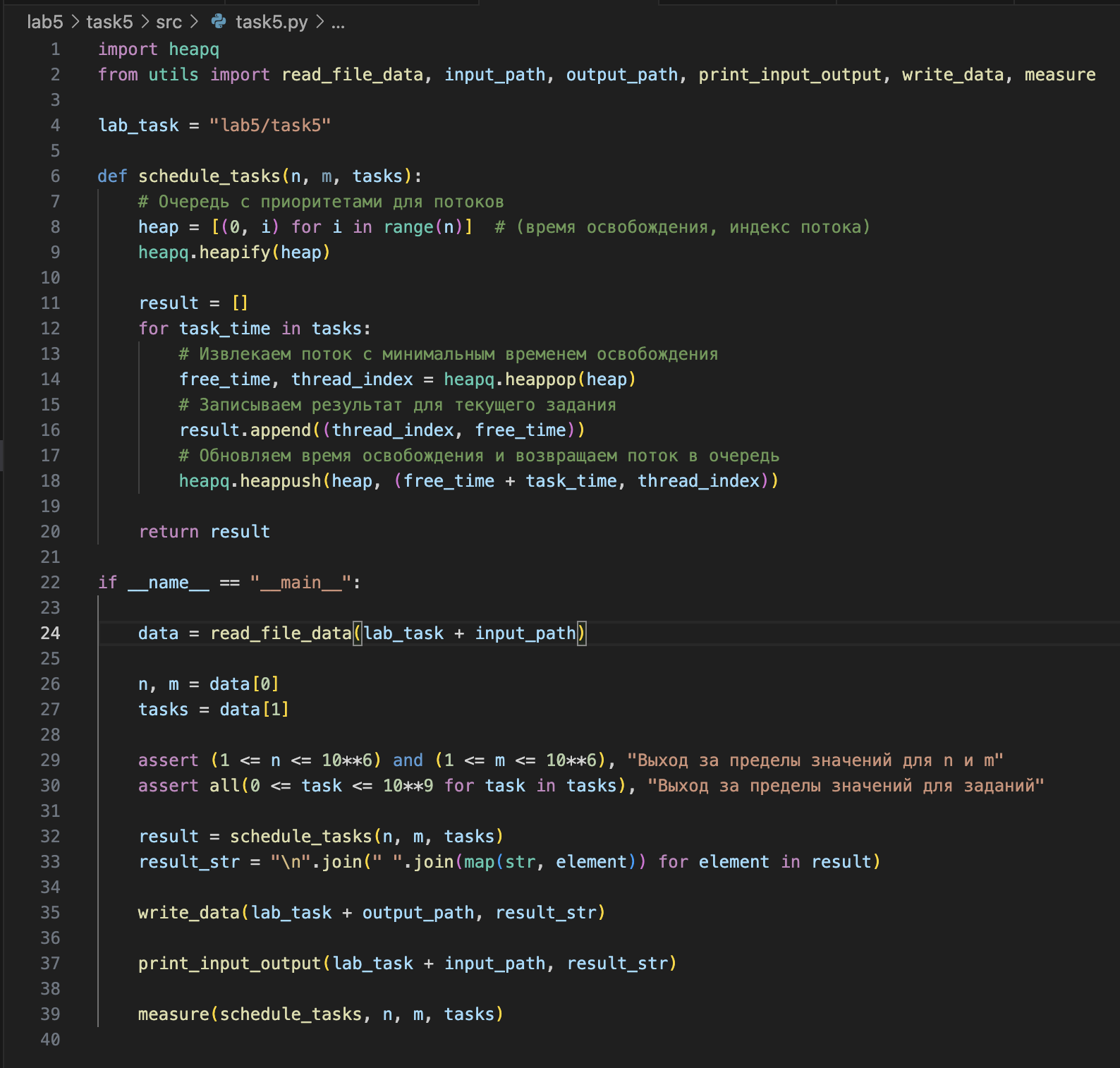


Задача №5:





Решение:



1. Входные параметры функции schedule\_tasks

* **n** — количество потоков (например, процессоров или исполнителей задач).
* **m** — количество задач.
* **tasks** — список, где каждый элемент — это время выполнения соответствующей задачи.

**2**. Инициализация очереди с приоритетами

*heap = [(0, i) for i in range(n)] # (время освобождения, индекс потока)*

*heapq.heapify(heap)*

* **heap** — это список кортежей, где каждый кортеж содержит два значения:
  1. **Время освобождения потока** (начальное значение — 0, т.к. потоки свободны в начале).
  2. **Индекс потока** (от 0 до n-1).
* **heapq.heapify** — преобразует список heap в **мин-кучу**, обеспечивая, что на вершине (в начале списка) всегда находится элемент с минимальным временем освобождения. Мин-куча оптимальна для извлечения и обновления минимального элемента.

3. Итерация по задачам и их распределение

Цикл проходит по каждой задаче в списке tasks.

*for task\_time in tasks:*

* **task\_time** — время, которое требуется для выполнения текущей задачи.

3.1 Извлечение потока с минимальным временем освобождения

*free\_time, thread\_index = heapq.heappop(heap)*

* **heapq.heappop** — извлекает элемент с минимальным временем освобождения из кучи.
* Возвращает два значения:
  + **free\_time** — текущее время, когда поток освободится.
  + **thread\_index** — индекс потока, который освободится первым.

3.2 Запись результата

*result.append((thread\_index, free\_time))*

* Добавляем кортеж (thread\_index, free\_time) в список result, чтобы зафиксировать:
  + Какой поток (по индексу) будет обрабатывать задачу.
  + В какое время он начнет выполнять эту задачу.

3.3 Обновление времени освобождения потока

*heapq.heappush(heap, (free\_time + task\_time, thread\_index))*

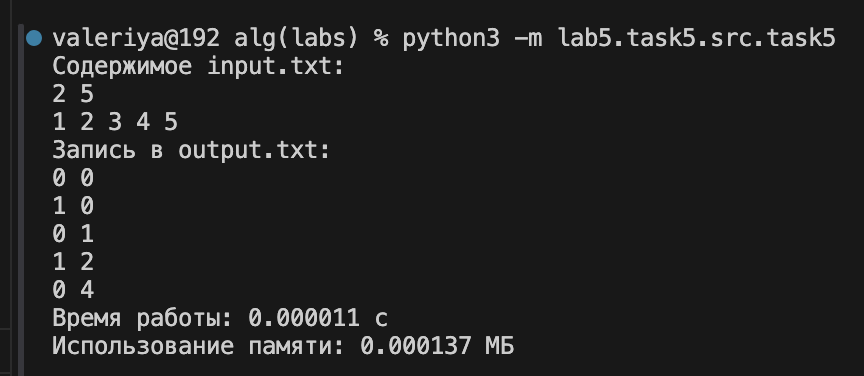
* **heapq.heappush** — добавляет новый кортеж в кучу, обновляя её структуру.
* Новый кортеж:
  + **free\_time + task\_time** — обновленное время, когда поток освободится (предыдущее время + длительность текущей задачи).
  + **thread\_index** — индекс потока, которому была назначена задача.

4. Возврат результата

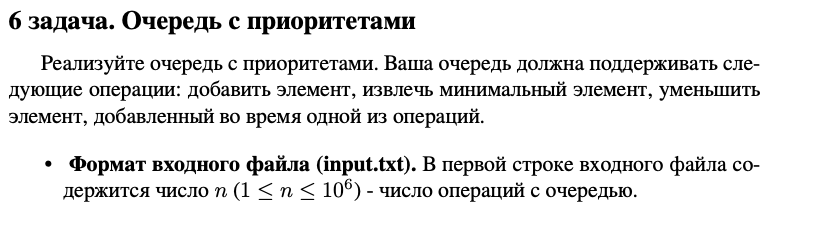
return result

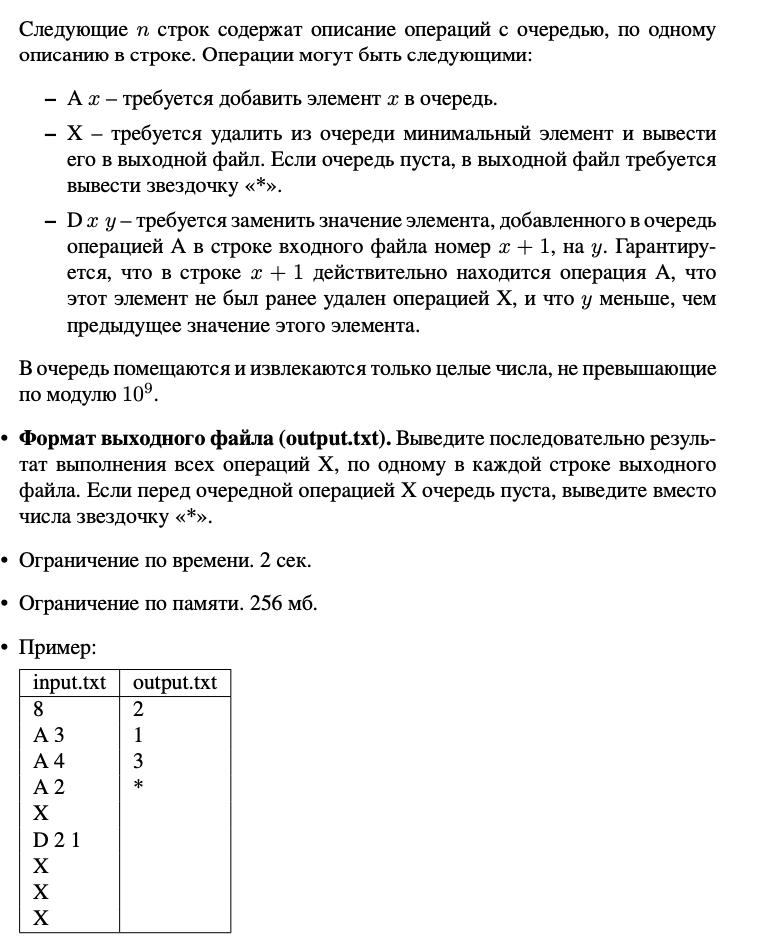
* Список result содержит информацию о том, какой поток и в какое время начал выполнение каждой задачи.
* Формат результата: список кортежей (индекс\_потока, время\_начала\_задачи)

Запуск программы:

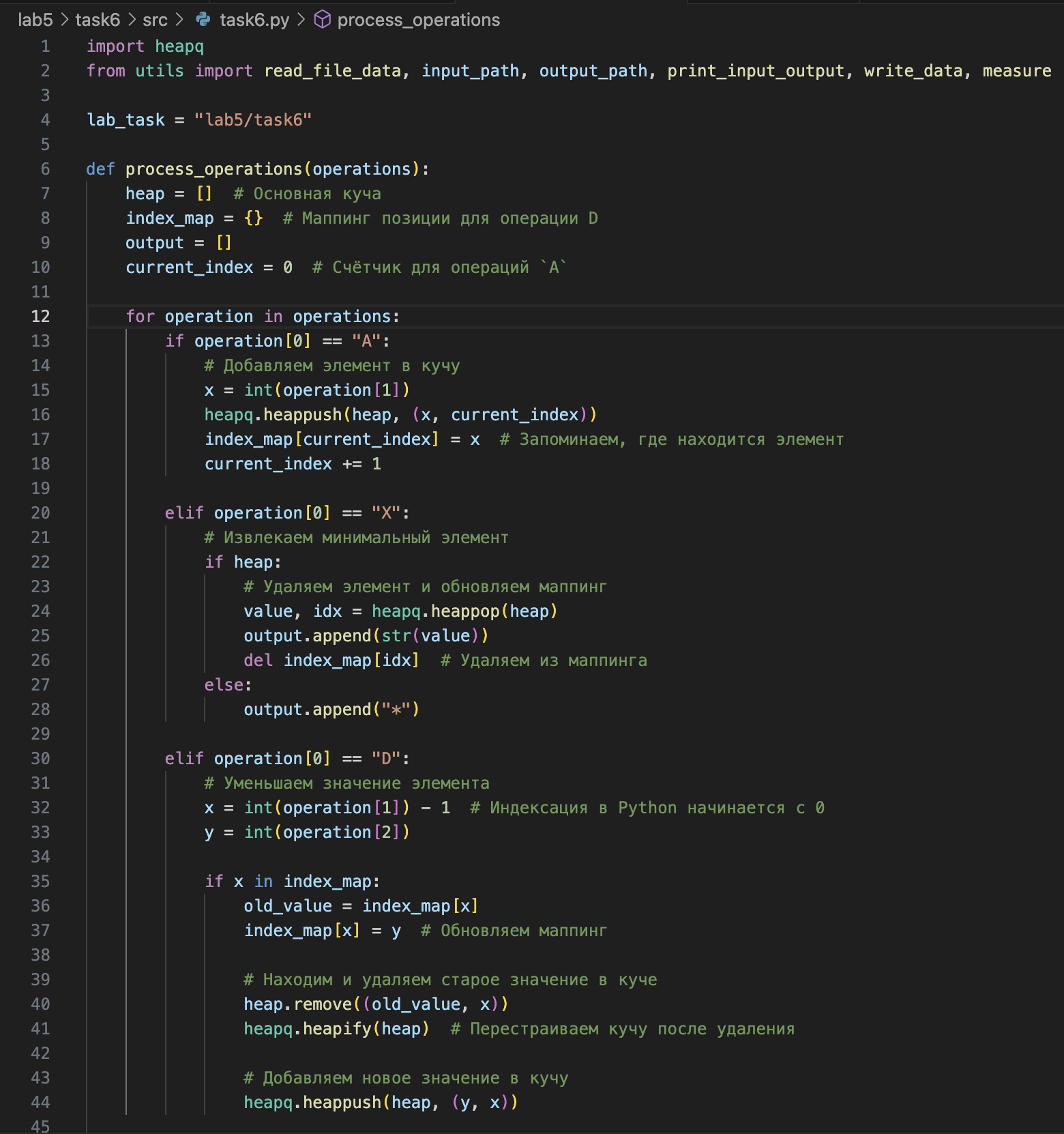


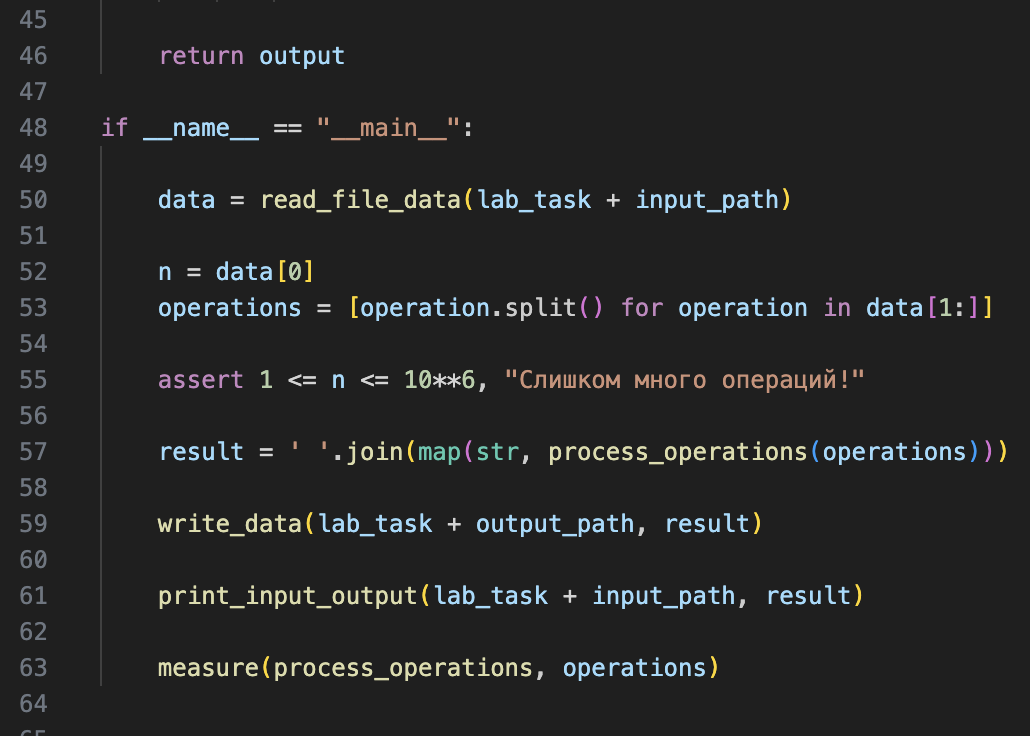
Задача№6:





Решение:



**import heapq -** Импортируем модуль heapq, который предоставляет функции для работы с кучами (очередями с приоритетами). В данном случае, мы будем использовать его для добавления элементов в кучу, извлечения минимальных элементов и поддержания порядка.

**def priority\_queue(operations): -** Определение функции priority\_queue, которая принимает список строк operations, каждая из которых представляет собой одну из операций с очередью.

**heap = [] # Очередь с приоритетами -** Инициализируем пустой список heap, который будет представлять собой нашу очередь с приоритетами. В Python это фактически будет минимальной кучей, где минимальный элемент всегда будет на первом месте.

**added\_elements = {} # Хранение индексов добавленных элементов {индекс: значение} -**

Создаем пустой словарь added\_elements, который будет хранить пары "индекс элемента — его значение". Это нужно для того, чтобы отслеживать добавленные элементы и правильно работать с операцией изменения элементов.

**removed = set() # Удалённые элементы (по индексу) -** Создаем пустое множество removed, которое будет хранить индексы элементов, которые были логически удалены (например, после изменения значений).

**result = [] -** Создаем пустой список result, который будет содержать результат выполнения операции извлечения элементов (операция "X").

**for index, operation in enumerate(operations, 1): -** Это цикл по всем операциям в списке operations, где index — это индекс операции (нумерация с 1), а operation — это строка с самой операцией. Нумерация с 1 позволяет легко идентифицировать элементы, когда мы добавляем их в очередь.

**parts = operation.split() -** Разбиваем строку операции на отдельные части (по пробелам). Например, операция "A 5" будет разбита на список ["A", "5"].

**if parts[0] == "A": -** Проверяем, является ли первая часть строки операцией "A", которая добавляет элемент в очередь. Если да, выполняем блок кода внутри.

**value = int(parts[1]) -** Извлекаем значение элемента, которое мы добавляем в очередь. Это значение находится во второй части строки (в parts[1]), и преобразуем его в целое число с помощью int.

**heapq.heappush(heap, (value, index)) -** Добавляем кортеж (value, index) в очередь с приоритетами (в кучу). Кортеж состоит из значения элемента и его индекса, что позволяет нам поддерживать уникальность элементов и отслеживать их индексы.

**added\_elements[index] = value -** Добавляем элемент в словарь added\_elements, где ключ — это индекс элемента, а значение — это его значение. Это нужно для отслеживания, какие элементы были добавлены, и для корректной работы с операцией изменения значений.

**elif parts[0] == "X": -** Проверяем, является ли операция "X" (извлечение минимального элемента). Если да, выполняем блок кода внутри.

**while heap and heap[0][1] in removed: -** Этот цикл удаляет все элементы, которые были логически помечены как удалённые. Мы проверяем, если в куче есть элементы, и если индекс первого элемента в куче (heap[0][1]) находится в множестве removed (то есть он был удален), мы его удаляем.

**heapq.heappop(heap) -** Извлекаем минимальный элемент из кучи. Если элемент был логически удалён, то просто продолжаем извлекать его.

**if heap: -** Проверяем, есть ли ещё элементы в куче после того, как мы удалили все логически удалённые. Если куча не пуста, извлекаем минимальный элемент.

**value, idx = heapq.heappop(heap) -** Извлекаем минимальный элемент из кучи и сохраняем его значение в value, а индекс в idx.

**removed.add(idx) -** Добавляем индекс элемента в множество removed, чтобы пометить его как удалённый.

**result.append(str(value)) -** Добавляем значение извлечённого элемента в список result. Мы преобразуем его в строку с помощью str(), так как результаты должны быть строками.

**else: -** Если куча пуста (то есть извлечь элемент невозможно), выполняем блок кода ниже.

**result.append("\*") -** В случае, если в очереди нет элементов, добавляем в результат символ "\*", который сигнализирует об ошибке (попытка извлечь элемент из пустой очереди).

**elif parts[0] == "D": -** Проверяем, является ли операция "D" (изменение значения элемента). Если да, выполняем блок кода внутри.

**target\_index = int(parts[1]) + 1 -** Извлекаем индекс элемента для изменения. Мы прибавляем 1, так как индексация начинается с 1, а Python использует индексацию с 0.

**new\_value = int(parts[2]) -** Извлекаем новое значение, которое будет присвоено элементу с индексом target\_index.

**if target\_index in added\_elements: -** Проверяем, существует ли элемент с таким индексом в словаре added\_elements. Если да, продолжаем.

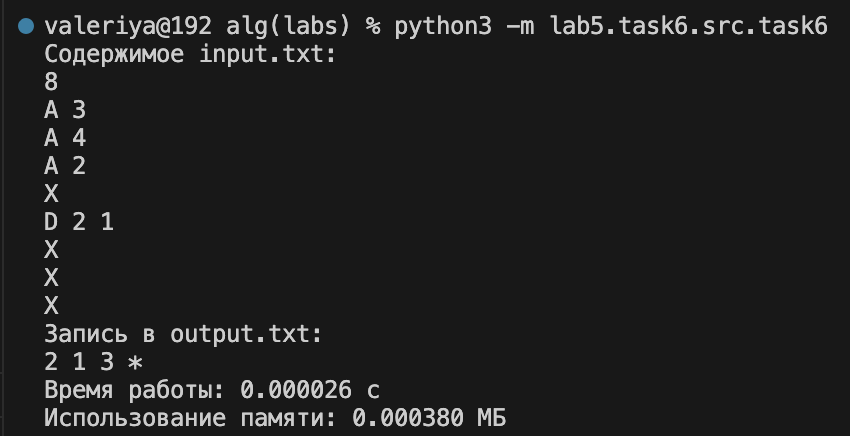
**removed.add(target\_index) -** Помечаем старое значение как удалённое, добавляя индекс в множество removed.

**heapq.heappush(heap, (new\_value, target\_index)) -** Добавляем новый элемент с новым значением в кучу. Кортеж состоит из нового значения и того же индекса, чтобы элемент оставался тем же, но с обновлённым значением.

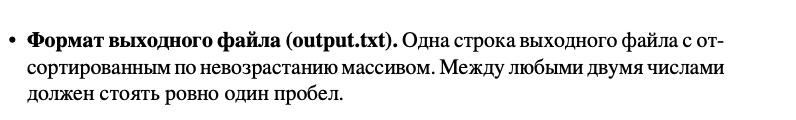
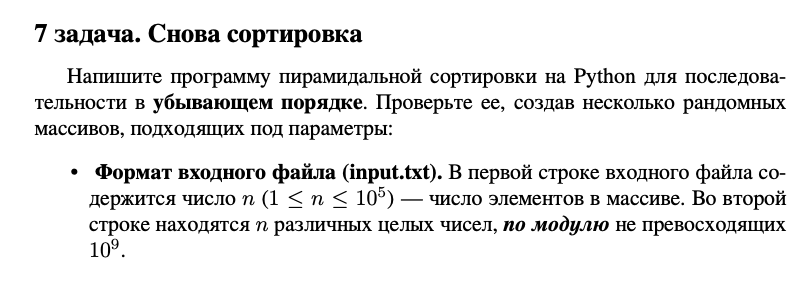
**added\_elements[target\_index] = new\_value -** Обновляем значение элемента в словаре added\_elements, чтобы оно отражало новое значение.

**return result -** После выполнения всех операций возвращаем список result, который содержит все результаты извлечений и ошибок.

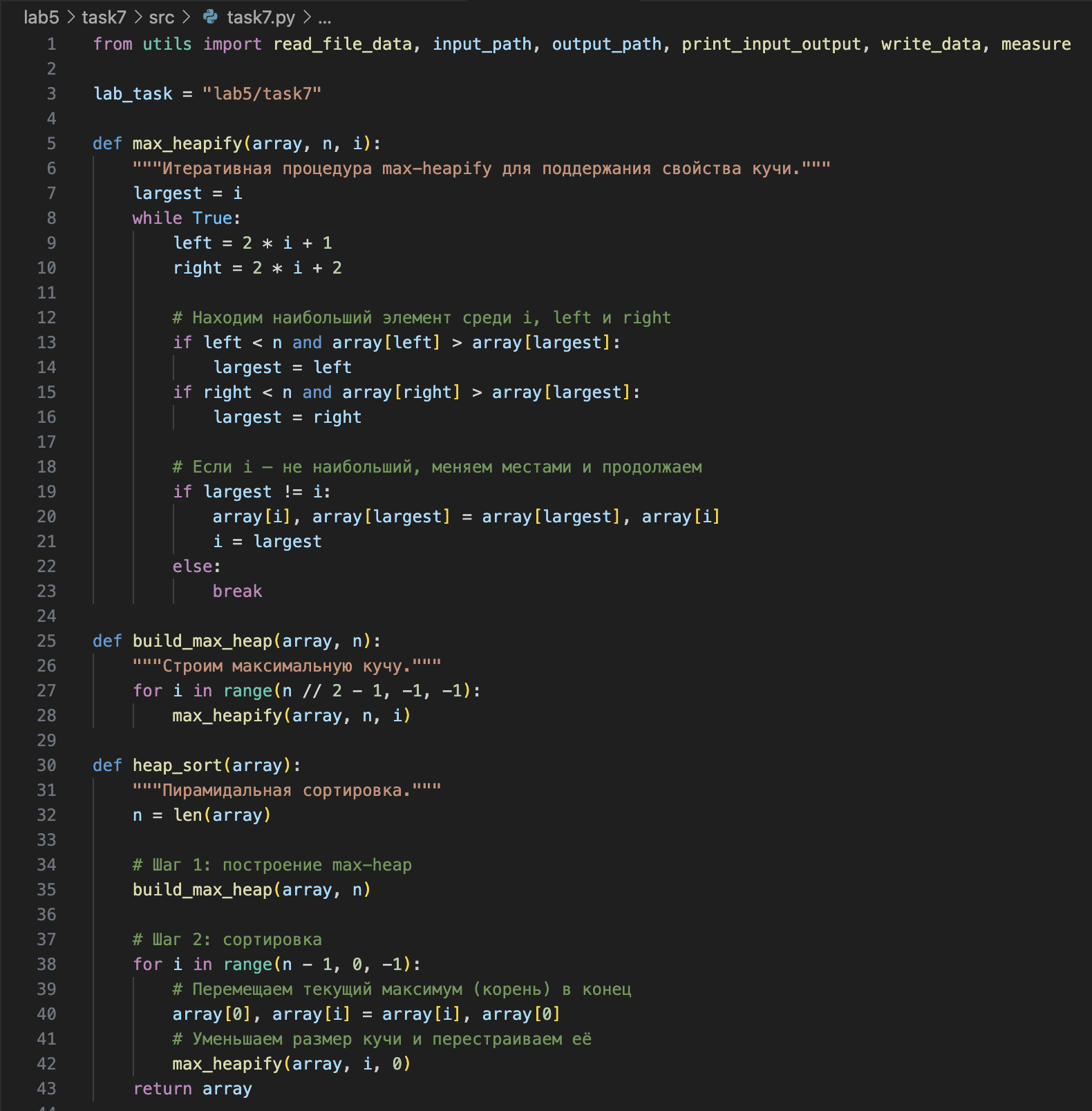
Запуск программы:

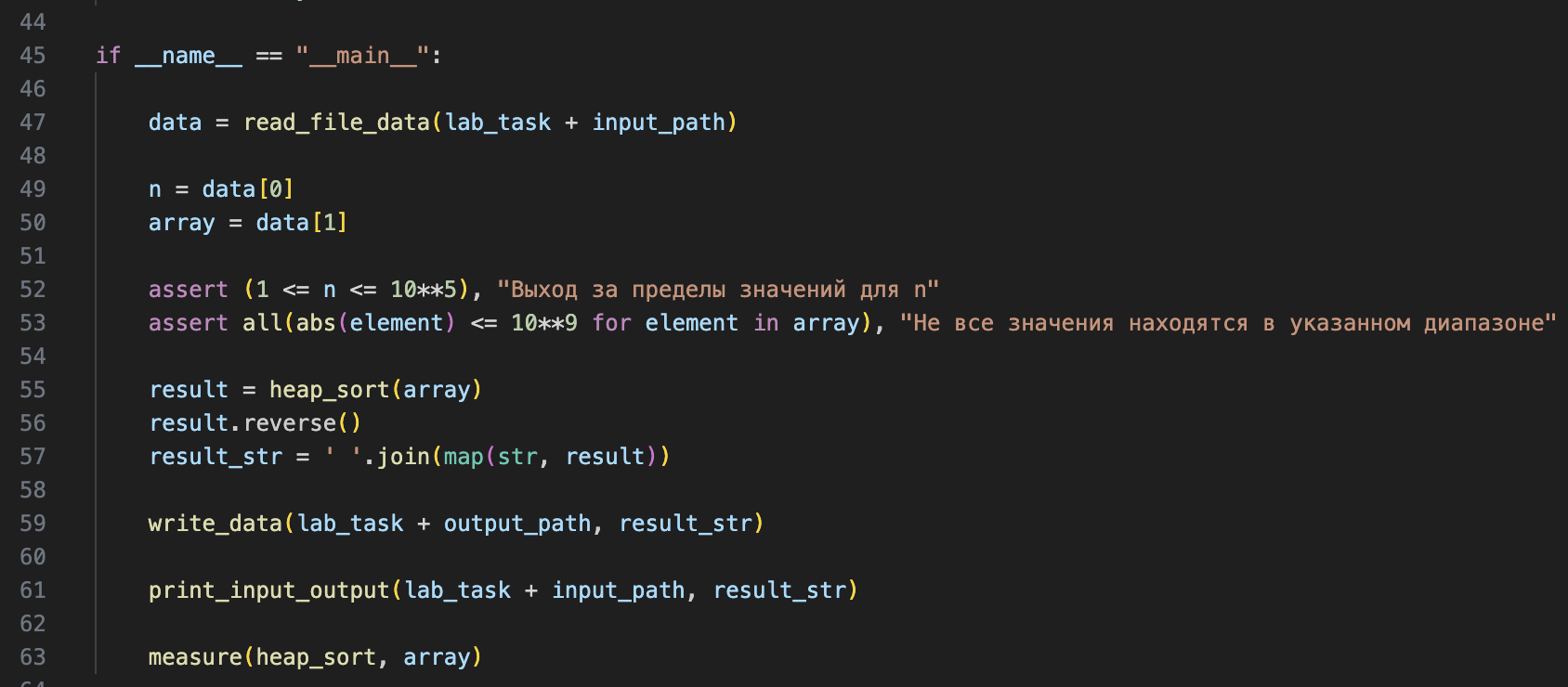


Задача №7:



Решение:





**Функции:**

*1. max\_heapify(array, n, i)*

Эта функция поддерживает свойство максимальной кучи на поддереве с корнем в узле i.

array: массив, представленный в виде двоичной кучи.

n: текущий размер кучи (может уменьшаться при сортировке).

i: индекс текущего узла, для которого нужно выполнить процедуру max-heapify.

* **Как работает функция :**
  1. Определяется индекс текущего узла как i.
  2. Находятся индексы левого (left = 2\*i + 1) и правого (right = 2\*i + 2) дочерних узлов.
  3. Сравнивается текущий узел с его дочерними узлами:
     + Если левый или правый дочерний узел больше текущего узла, обновляется индекс largest (индекс наибольшего элемента среди узла и его потомков).
  4. Если текущий узел (i) не является наибольшим, он меняется местами с наибольшим узлом, а затем процедура повторяется для нового индекса largest.
  5. Процесс продолжается, пока узел не окажется на своём правильном месте (свойство максимальной кучи восстановлено).

*2. build\_max\_heap(array, n)*

Эта функция строит максимальную кучу из массива.

array: массив, из которого строится куча. n: длина массива.

* **Как работает функция:**
  + В двоичной куче листья уже удовлетворяют свойству кучи. Поэтому процедура начинается с последнего узла, который имеет дочерние элементы (n//2 - 1).
  + Для каждого узла (двигаясь в обратном порядке от последнего не-листового узла до корня), вызывается max\_heapify, чтобы привести поддерево с корнем в этом узле к состоянию максимальной кучи.
  + После выполнения этой функции весь массив становится максимальной кучей.

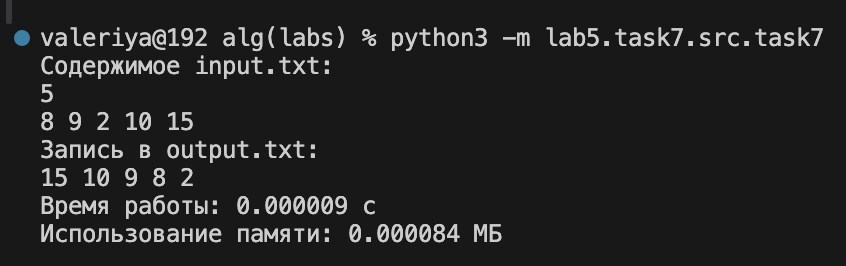
*3. heap\_sort(array)*

Эта функция выполняет сортировку массива с использованием алгоритма пирамидальной сортировки.

array: массив, который нужно отсортировать.

* **Как работает функция:**
  1. **Построение максимальной кучи:**
     + Вызовом build\_max\_heap из массива строится максимальная куча, где корень (array[0]) содержит максимальный элемент.
  2. **Сортировка:**
     + Идем с конца массива к началу:
       - Максимальный элемент (корень) перемещается в конец массива путём обмена array[0] и array[i].
       - Размер кучи уменьшается на 1 (n уменьшается).
       - Вызывается max\_heapify для восстановления свойства кучи.
  3. После завершения всех итераций массив оказывается отсортированным.

Запуск программы :



Вывод:

В данной лабораторной работе я научилась работать с кучей, пирамидальной сортировкой. Отработала навыки по решению задач на эти темы. Все необходимые задачи решила.