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.

Вариант 1

Выполнила:
Гайдук А. С.
К3241

Проверила:
Ромакина О. М.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №1. 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 $a_i \leq a_{2i+1}$

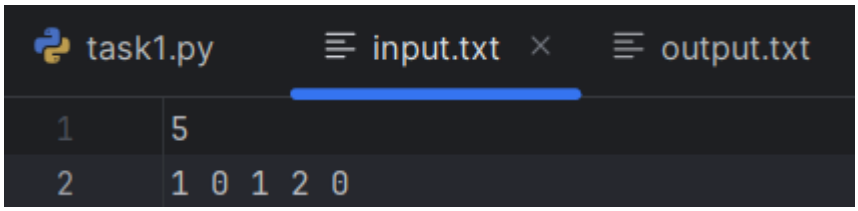
Дан массив целых чисел. Определите, является ли он неубывающей пирамидой.

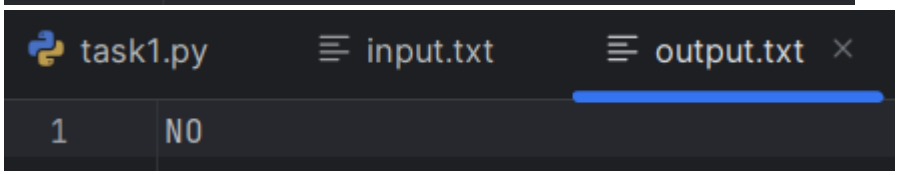
Листинг кода:

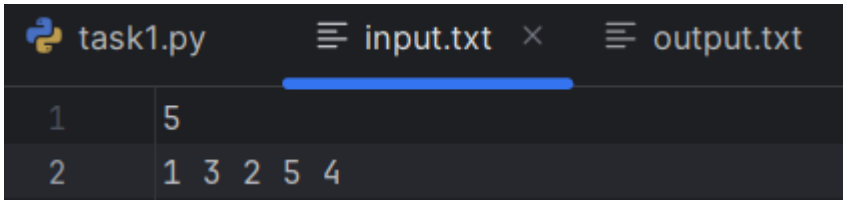
```
import tracemalloc
import time

tracemalloc.start()


def check_heap(n, array):
    for i in range(1, n):
        if 2 * i < n and array[i - 1] > array[2 * i - 1]:
            return "NO"
        if 2 * i + 1 < n and array[i - 1] > array[2 * i]:
            return "NO"
    return "YES"


with open('input.txt', 'r') as f:
    n = int(f.readline())
    array = [int(x) for x in f.readline().split()]

start_time = time.perf_counter()

result = check_heap(n, array)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as f:
    f.write(result)


print(f"Затраты памяти: {current / 10 ** 6:.6f} MB; Пиковое
использование: {peak / 10 ** 6:.6f} MB")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")
```

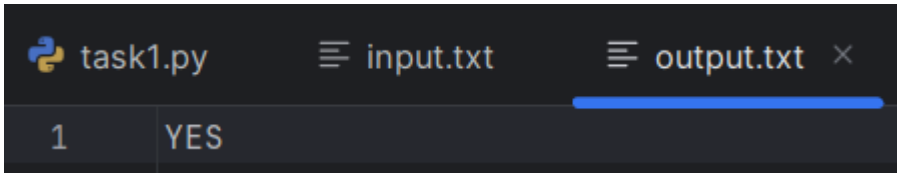
Текстовое объяснение решения:
Для решения этой задачи я создала метод check_heap, который проверяет входные данные на соответствие свойствам, указанным в условии.

Результат работы кода на примерах из текста задачи:

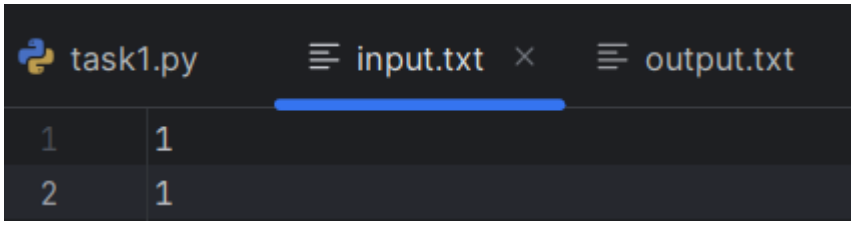


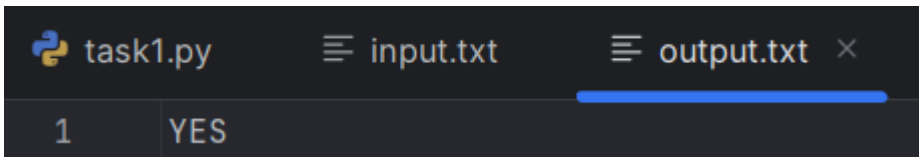


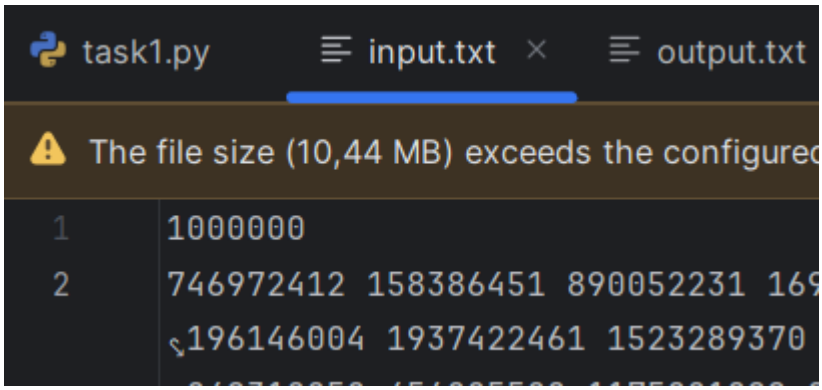


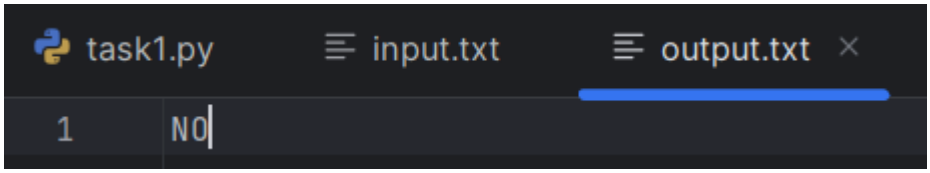


Результат работы кода на максимальных и минимальных значениях:









	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000005 сек	0.037525 MB
Пример из задачи	0.000010 сек	0.037583 MB
Пример из задачи	0.000006 сек	0.037583 MB
Верхняя граница диапазона значений входных данных из текста задачи	0.000014 сек	97.388708 MB

Вывод по задаче:
В ходе выполнения этой задачи я ознакомилась с такой структурой данных, как куча (или же неубывающая пирамида), её свойствами. Я научилась реализовывать алгоритм определения неубывающей пирамиды из массива.

Задача №2. Высота дерева

В этой задаче ваша цель - привыкнуть к деревьям. Вам нужно будет прочитать описание дерева из входных данных, реализовать структуру данных, сохранить дерево и вычислить его высоту.
• Вам дается корневое дерево. Ваша задача - вычислить и вывести его высоту. Напомним, что высота (корневого) дерева — это максимальная глубина узла или максимальное расстояние от листа до корня. Вам дано произвольное дерево, не обязательно бинарное дерево.

Листинг кода

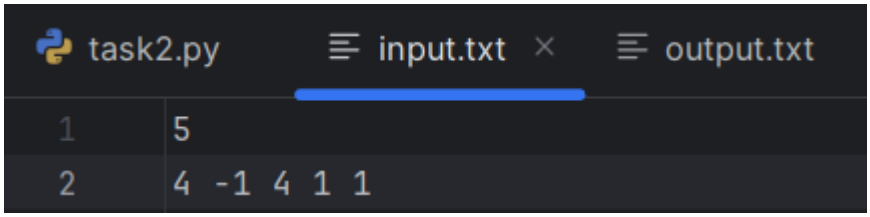
```
import tracemalloc
import time

tracemalloc.start()
```

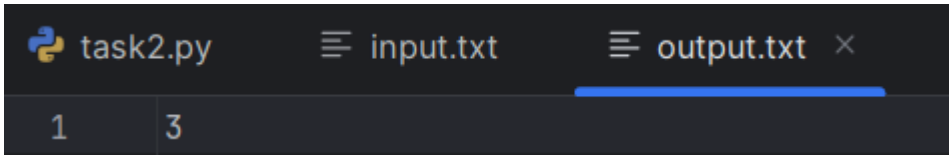
```
def tree_height(n, parents):
    tree = [[] for _ in range(n)]
    root = -1
    for child, parent in enumerate(parents):
        if parent == -1:
            root = child
        else:
            tree[parent].append(child)
    if root == -1:
        return 0
    queue = [(root, 1)]
    max_height = 0
    while queue:
        node, depth = queue.pop(0)
        max_height = max(max_height, depth)
        for child in tree[node]:
            queue.append((child, depth + 1))

    return max_height


with open('input.txt', 'r') as f:
    n = int(f.readline())
    array = [int(x) for x in f.readline().split()]

start_time = time.perf_counter()

result = tree_height(n, array)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as f:
    f.write(str(result))


print(f"Затраты памяти: {current / 10 ** 6:.6f} MB; Пиковое
использование: {peak / 10 ** 6:.6f} MB")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")
```

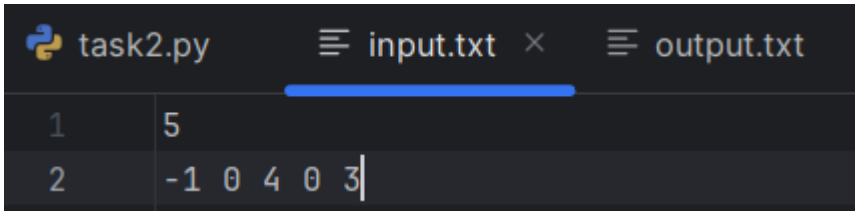
Текстовое объяснение решения:

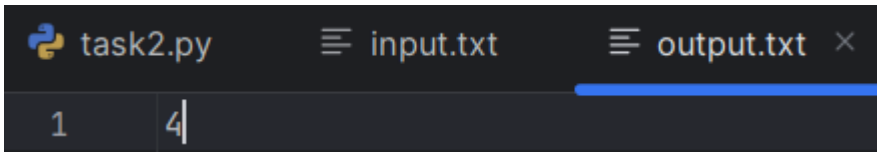
Я создала метод, который вычисляет высоту дерева на основе списка родителей. Для этого создается список смежности tree, где каждый индекс представляет узел, а список по этому индексу содержит дочерние узлы. Если родитель равен «-1», то мы определяем его как корень дерева. Если корень не найден, то высота дерева считается равной нулю. Далее идет обход в ширину: создаем список очереди из пары (узел, глубина), и пока очередь не пуста, продолжается обход дерева и извлекается первый элемент из очереди, обновляется максимальная глубина.

Результат работы кода на примерах из текста задачи:

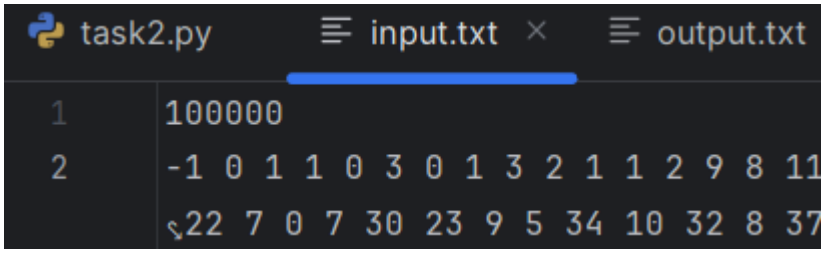


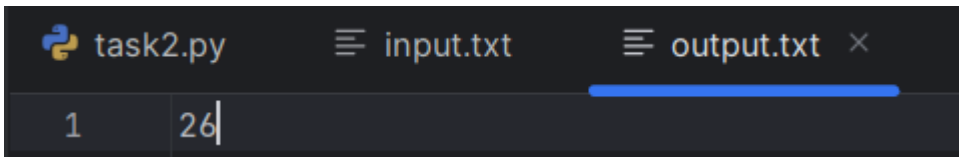


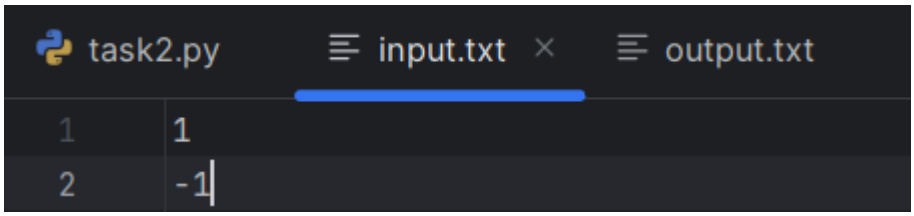


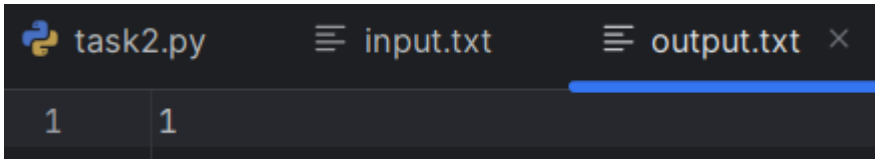


Результат работы кода на максимальных и минимальных значениях:









	Время выполнения	Затраты памяти
Нижняя граница	0.000013 сек	0.037569 MB

диапазона значений входных данных из текста задачи		
Пример из задачи	0.000033 сек	0.037585 MB
Пример из задачи	0.000023 сек	0.037585 MB
Верхняя граница диапазона значений входных данных из текста задачи	0.838931 сек	15.381051 MB

Вывод по задаче:
Я ознакомилась с такой структурой данных, как дерево, и смогла вычислить его высоту.

Дополнительные задачи

Задача №6. 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.

Листинг кода:

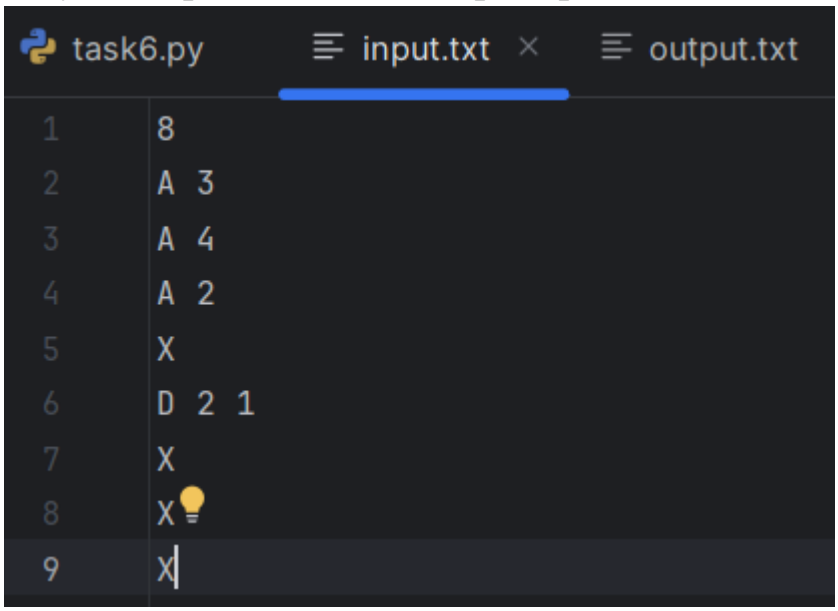
```
import heapq
import tracemalloc
import time

tracemalloc.start()

def queue_plus(operations):
    min_heap = []
    added_elements = {}
    element_indices = {}
    results = []

    for i, operation in enumerate(operations):
        operation_parts = operation.strip().split()
        command = operation_parts[0]

        if command == 'A':
            x = int(operation_parts[1])
            heapq.heappush(min_heap, x)
            added_elements[i + 1] = x
            if x not in element_indices:
                element_indices[x] = []
            element_indices[x].append(i + 1)

        elif command == 'X':
            if min_heap:
                min_el = heapq.heappop(min_heap)
                results.append(str(min_el))
                if min_el in element_indices:
                    element_indices[min_el].pop(0)
                    if not element_indices[min_el]:
                        del element_indices[min_el]
            else:
                results.append('*')

        elif command == 'D':
            x = int(operation_parts[1])
            y = int(operation_parts[2])

            if x in added_elements:
                prev_value = added_elements[x]
                if prev_value in min_heap:
                    min_heap.remove(prev_value)
                    heapq.heapify(min_heap)
                    heapq.heappush(min_heap, y)

                    added_elements[x] = y
```

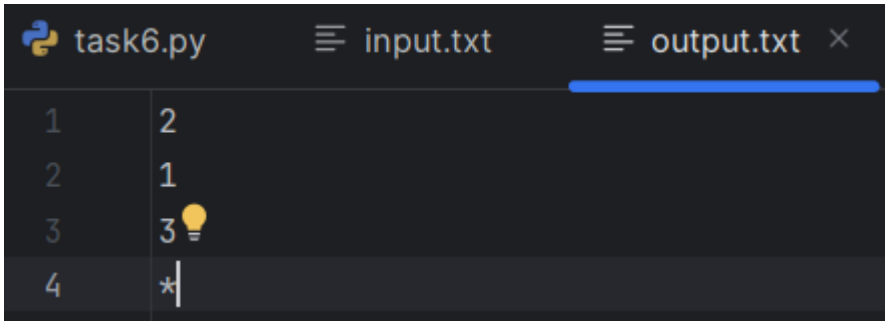
```
                    if prev_value in element_indices:
                        element_indices[prev_value].remove(x)
                        if not element_indices[prev_value]:
                            del element_indices[prev_value]
                    if y not in element_indices:
                        element_indices[y] = []
                    element_indices[y].append(x)

    return results


with open('input.txt', 'r') as f:
    n = int(f.readline().strip())
    operations = [f.readline().strip() for _ in range(n)]

start_time = time.perf_counter()


result = queue_plus(operations)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()


with open('output.txt', 'w') as f:
    f.write("\n".join(result))


print(f"Затраты памяти: {current / 10 ** 6:.6f} MB; Пиковое
использование: {peak / 10 ** 6:.6f} MB")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")
```

Текстовое объяснение решения:

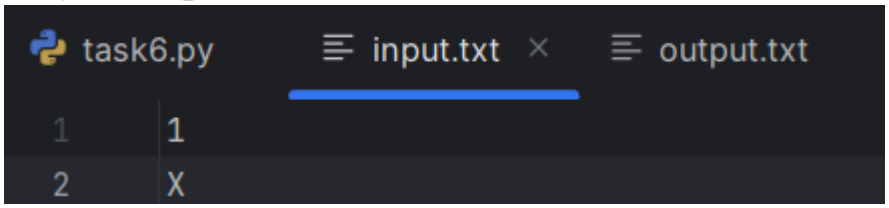
Для решения этой задачи я воспользовалась модулем heapq. Для операции A x я реализовала логику добавления элемента в кучу для поддержания очереди с минимальным элементом, словари хранят связь между строкой операции и добавленным элементом при заменах и удалениях. Для операции X минимальный элемент удаляется из кучи с помощью heapq.heappop(), а словарь element_indices обновляется, удаляя информацию об удаленном элементе. Для D x y я реализовала логику нахождения значения, добавленного операцией A в строке x + 1, а также удаления этого значения из кучи и замены его новым значением y. С помощью heapq.heapify() куча перестраивается.

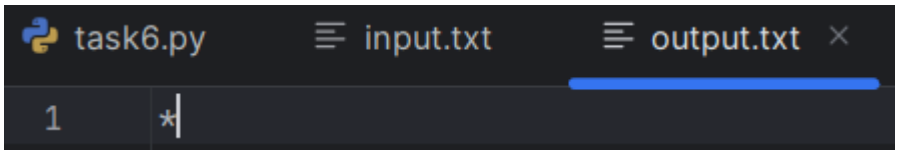
Результат работы кода на примерах из текста задачи:

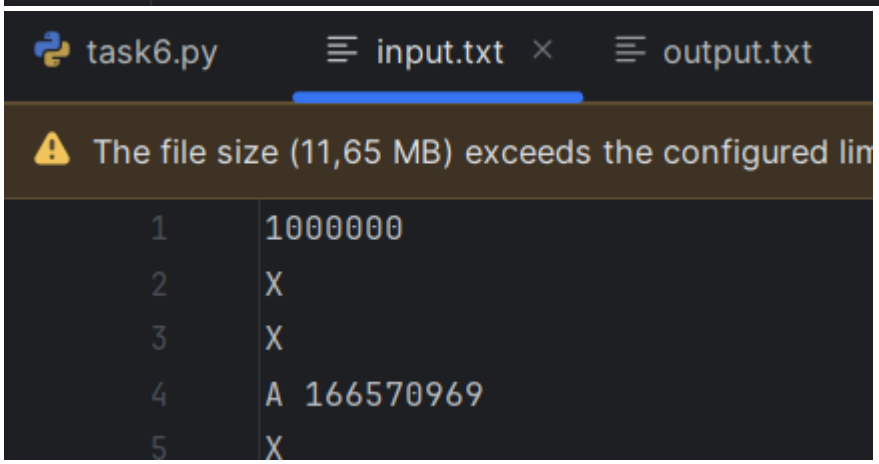


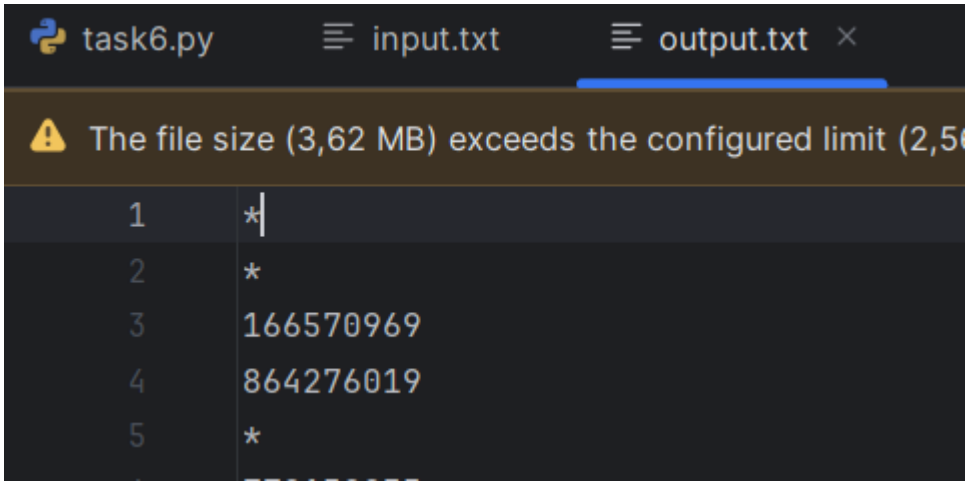


Результат работы кода на максимальных и минимальных значениях:









	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000009 сек	0.037557 MB
Пример из задачи	0.000081 сек	0.037830 MB
Верхняя граница диапазона значений входных данных из текста задачи	1.617778 сек	95.128170 MB

Вывод по задаче:
Я познакомилась с модулем heapq и научилась реализовывать алгоритм очереди с приоритетами.

Задача №5. Планировщик заданий

В этой задаче вы создадите программу, которая параллельно обрабатывает список заданий. Во всех операционных системах, таких как Linux, MacOS или Windows, есть специальные программы, называемые планировщиками, которые делают именно это с программами на вашем компьютере.
У вас есть программа, которая распараллеливается и использует n независимых потоков для обработки заданного списка m заданий. Потоки берут задания в том порядке, в котором они указаны во входных данных. Если есть свободный поток, он немедленно берет следующее задание из

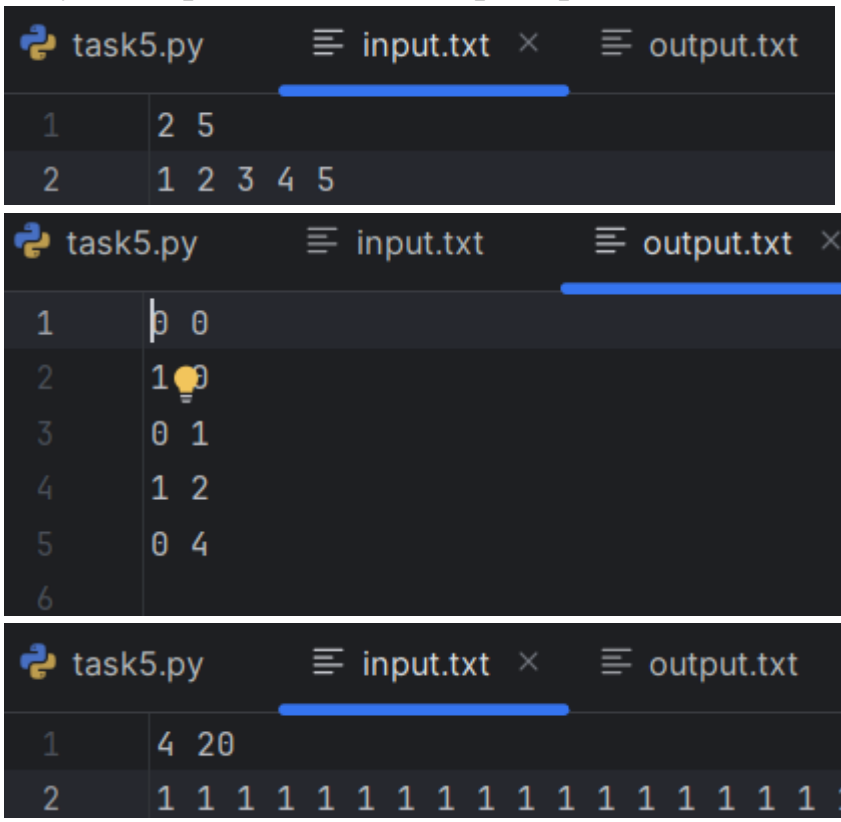
списка. Если поток начал обработку задания, он не прерывается и не останавливается, пока не завершит обработку задания. Если несколько потоков одновременно пытаются взять задания из списка, поток с меньшим индексом берет задание. Для каждого задания вы точно знаете, сколько времени потребуется любому потоку, чтобы обработать это задание, и это время одинаково для всех потоков.
Вам необходимо определить для каждого задания, какой поток будет его обрабатывать и когда он начнет обработку.

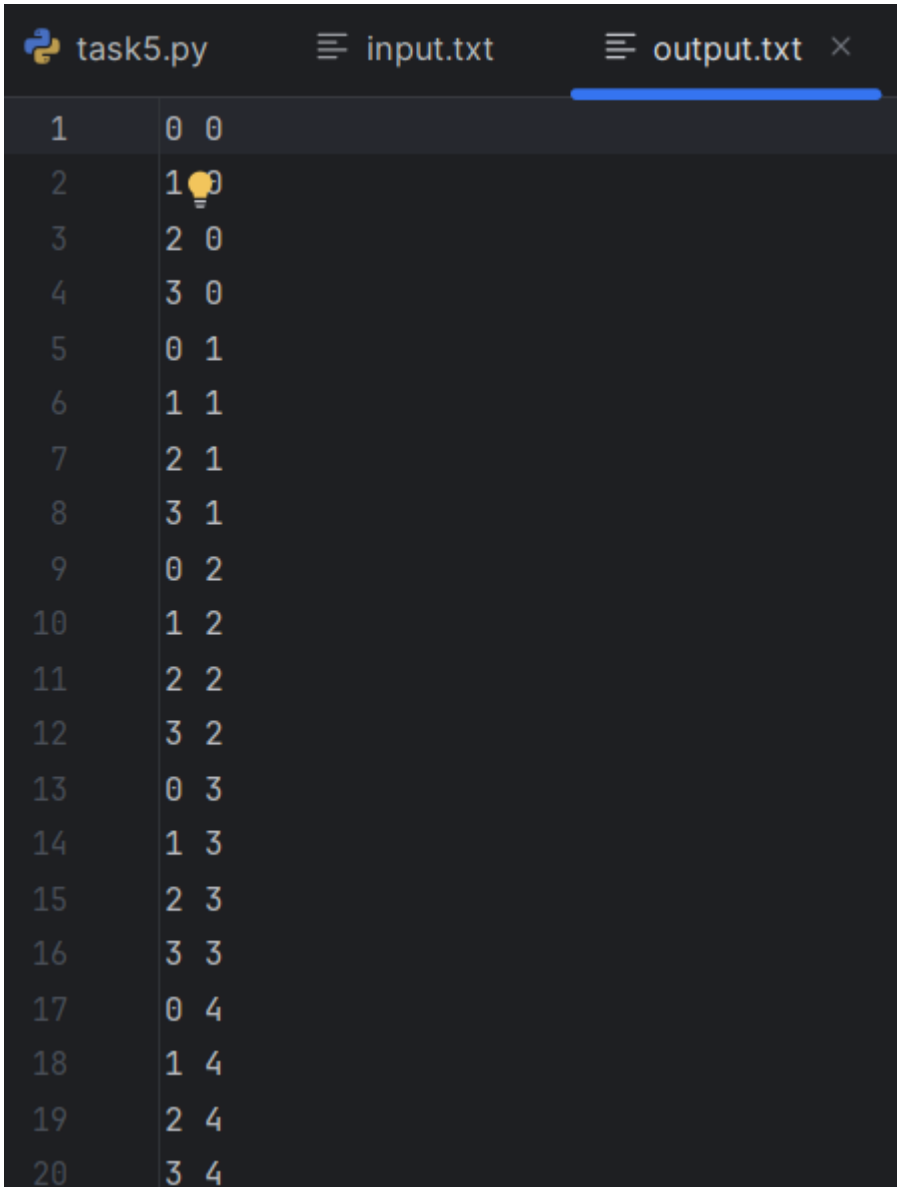
Листинг кода:

```
import heapq
import time
import tracemalloc

tracemalloc.start()

def task_scheduler(n, task_times):
    thread_heap = [(0, i) for i in range(n)]
    heapq.heapify(thread_heap)
    result = []

    for task_time in task_times:
        current_time, thread_index = heapq.heappop(thread_heap)
        result.append((thread_index, current_time))
        heapq.heappush(thread_heap, (current_time + task_time,
thread_index))

    return result



with open('input.txt', 'r') as f:
    n, m = map(int, f.readline().strip().split())
    task_times = list(map(int, f.readline().strip().split()))

start_time = time.perf_counter()

result = task_scheduler(n, task_times)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as f:
    for thread_index, pr_start_time in result:
        f.write(f"{thread_index} {pr_start_time}\n")

print(f"Затраты памяти: {current / 10 ** 6:.6f} MB; Пиковое
использование: {peak / 10 ** 6:.6f} MB")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")
```

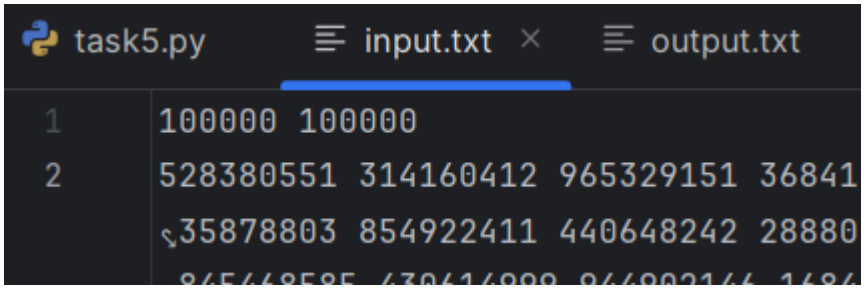
Текстовое объяснение решения:
Для решения этой задачи я вновь использовала модуль heapq. Я создала список из кортежей с временем завершения (изначально 0) и индексом потока, и преобразовала его с помощью heapq.heapify(). Проходя по каждой задаче, я извлекаю поток с минимальным временем завершения с помощью heappop(), добавляю результат для текущей задачи и обновляю время завершения потока.

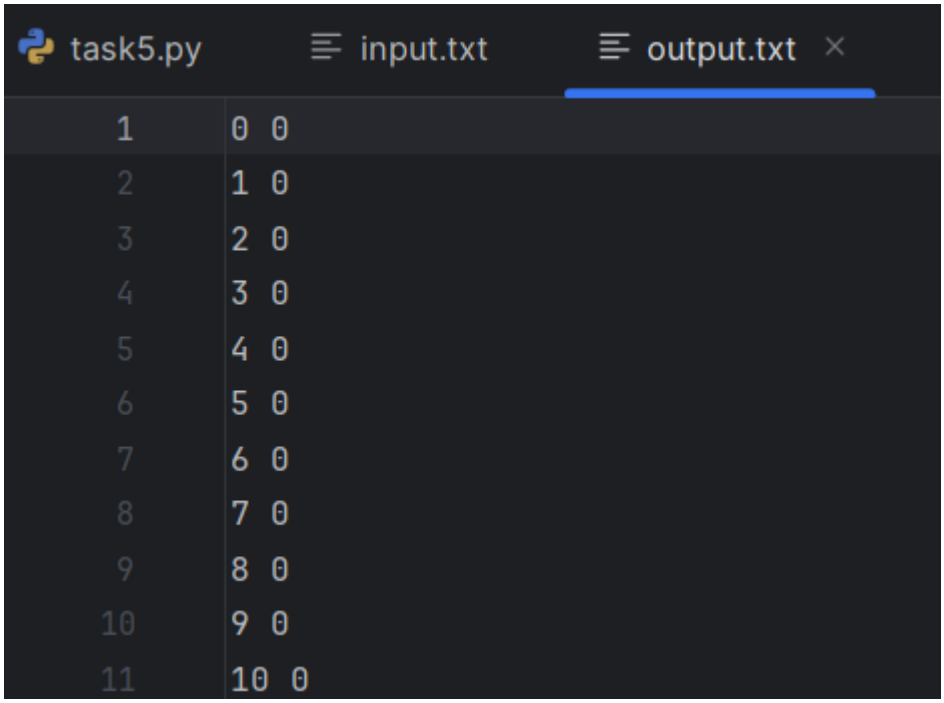
Результат работы кода на примерах из текста задачи:

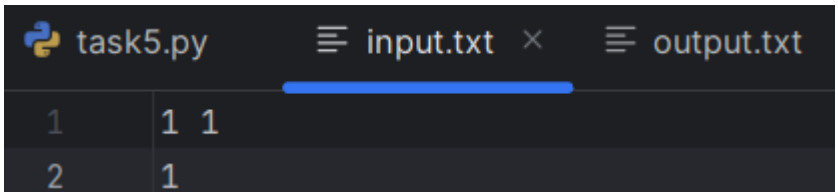


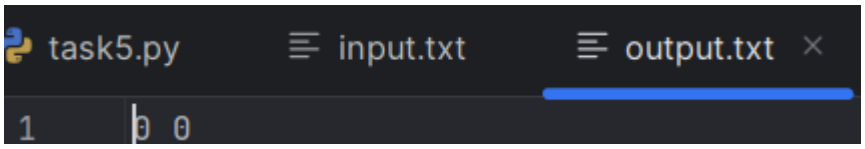


Результат работы кода на максимальных и минимальных значениях:









	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000019 сек	0.037575 MB
Пример из задачи	0.000027 сек	0.037633 MB
Пример из задачи	0.000032 сек	0.037694 MB
Верхняя граница диапазона значений входных данных из текста задачи	0.672271 сек	22.813371 MB

Вывод по задаче:

Я закрепила свои знания по работе с очередью с приоритетами. У меня получилось написать программу, реализующую функции планировщика заданий. Было интересно.

Вывод

В ходе выполнения данной лабораторной работы я ознакомилась с новыми структурами данных – кучей, пирамидой, очередью с приоритетами и деревом. Я попробовала реализовать различные алгоритмы по работе с этими структурами.