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 9

Выполнила:

Коновалова Кира Романовна

К3139

Проверил:

Афанасьев А.В

Санкт-Петербург

2024 г.

# Содержание отчета

**[Задачи по варианту. Задачи по выбору 3](#_Toc14248)**

[Задача №1. Куча ли? (по выбору) 3](#_Toc3494)

[Задача №2. Высота дерева 6](#_Toc13741)

[Задача №4. Построение пирамиды (по выбору) 8](#_Toc29698)

[Задача №5. Планировщик заданий 11](#_Toc14457)

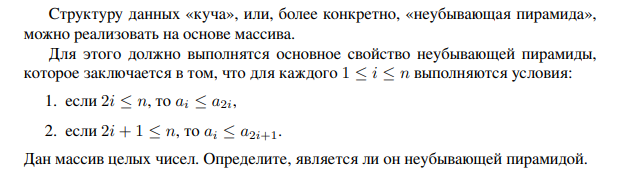
**[Запуск всех задач: 15](#_Toc17570)**

**[Вывод 16](#_Toc19366)**

# Задачи по варианту. Задачи по выбору

## Задача №1. Куча ли? (по выбору)

Текст задачи:



Листинг кода:

import os  
import sys  
  
def is\_heap(array):  
 *"""  
 Проверяет, является ли массив неубывающей пирамидой (кучей).  
 :param array: Список целых чисел  
 :return: True, если массив является кучей, иначе False  
 """* n = len(array)  
 for i in range(n):  
 if 2 \* i + 1 < n and array[i] > array[2 \* i + 1]:  
 return False  
 if 2 \* i + 2 < n and array[i] > array[2 \* i + 2]:  
 return False  
 return True  
  
  
def main(input\_data, output\_file):  
 *"""  
 Основная функция для запуска задачи.  
 :param input\_data: Строка с содержимым входного файла  
 :param output\_file: Объект файла для записи вывода  
 """* lines = input\_data.strip().split("\n")  
 n = int(lines[0])  
 array = list(map(int, lines[1].split()))  
  
 # Проверяем массив  
 result = "YES" if is\_heap(array) else "NO"  
  
 # Записываем результат в выходной файл  
 output\_file.write(result + "\n")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

Текстовое объяснение задачи:

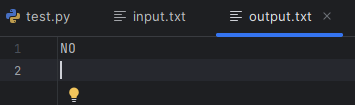
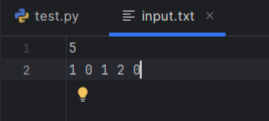
Для проверки массива на соответствие куче, мы проходим по всем его элементам и проверяем, выполняются ли условия для дочерних элементов. Если хотя бы для одного элемента условие нарушается, массив не является кучей

Реализована функция is\_heap(), которая проверяет массив на выполнение условий для пирамидальной структуры. В случае, если все условия выполнены, функция возвращает True, иначе False

Тесты:

import unittest  
from io import StringIO  
import sys  
import os  
base\_dir = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))  
src\_dir = os.path.join(base\_dir, '..', 'src')  
sys.path.append(src\_dir)  
from lab5.task1.src.main import main  
class TestMain(unittest.TestCase):  
  
 def setUp(self):  
 *"""Настроим стандартный вывод, чтобы можно было проверить вывод в файле"""* self.sio = StringIO()  
 self.saved\_stdout = sys.stdout  
 sys.stdout = self.sio  
 def tearDown(self):  
 *"""Возвращаем стандартный вывод после выполнения теста"""* sys.stdout = self.saved\_stdout  
 def test\_heap\_valid(self):  
 *"""Тест для случая, когда массив является неубывающей пирамидой"""* input\_data = "7\n1 3 6 5 9 8 12\n"  
 main(input\_data, self.sio)  
 output = self.sio.getvalue().strip()  
 self.assertEqual(output, "YES")  
 def test\_heap\_invalid(self):  
 *"""Тест для случая, когда массив не является неубывающей пирамидой"""* input\_data = "7\n1 3 6 7 9 8 4\n"  
 main(input\_data, self.sio)  
 output = self.sio.getvalue().strip()  
 self.assertEqual(output, "NO")  
 def test\_single\_element(self):  
 *"""Тест для массива с одним элементом, который всегда является пирамидой"""* input\_data = "1\n5\n"  
 main(input\_data, self.sio)  
 output = self.sio.getvalue().strip()  
 self.assertEqual(output, "YES")  
 def test\_two\_elements(self):  
 *"""Тест для массива с двумя элементами, который является пирамидой"""* input\_data = "2\n5 8\n"  
 main(input\_data, self.sio)  
 output = self.sio.getvalue().strip()  
 self.assertEqual(output, "YES")  
 def test\_large\_valid\_heap(self):  
 *"""Тест для большого массива, который является пирамидой"""* input\_data = "10\n1 3 5 7 9 8 6 10 11 12\n"  
 main(input\_data, self.sio)  
 output = self.sio.getvalue().strip()  
 self.assertEqual(output, "YES")  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

Результат работы кода на примерах из текста задачи:

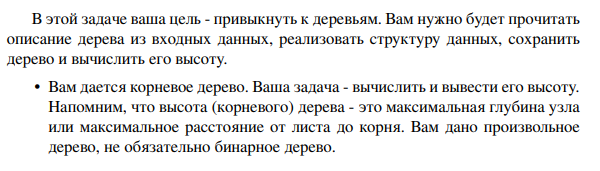


Вывод по задаче:

Задача успешно решена. Алгоритм работает за время O(n), так как мы проходим по каждому элементу массива и проверяем его дочерние элементы

## Задача №2. Высота дерева

Текст задачи:



Листинг кода:

import sys  
import os  
  
# Функция для вычисления высоты дерева  
def calculate\_height(tree, node):  
 # Если у узла нет детей, его высота 1  
 if not tree[node]:  
 return 1  
 # Рекурсивно вычисляем высоту всех поддеревьев и берем максимальную  
 height = 0  
 for child in tree[node]:  
 height = max(height, calculate\_height(tree, child))  
 return height + 1  
  
def main(input\_data, output\_file):  
 lines = input\_data.strip().split("\n")  
 n = int(lines[0])  
 parents = list(map(int, lines[1].split()))  
 # Строим дерево  
 tree = [[] for \_ in range(n)]  
 root = -1  
 for i in range(n):  
 if parents[i] == -1:  
 root = i  
 else:  
 tree[parents[i]].append(i)  
 # Вычисление высоты дерева  
 height = calculate\_height(tree, root)  
 # Записываем результат в выходной файл  
 output\_file.write(str(height))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

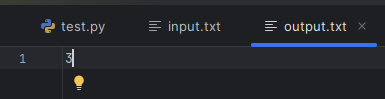
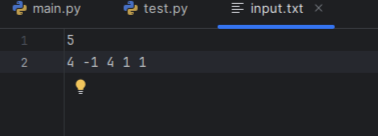
Текстовое объяснение задачи:

Для вычисления высоты дерева используется рекурсивный подход. Мы начинаем с корня и для каждого узла рекурсивно вычисляем высоту его поддеревьев, выбирая максимальное значение. Я выполнила построение дерева на основе списка родительских узлов и вычислили его высоту с помощью рекурсивной функции calculate\_height()

Тесты:

import unittest  
from io import StringIO  
from lab5.task2.src.main import calculate\_height, main  
class TestCalculateHeight(unittest.TestCase):  
 def test\_single\_node(self):  
 tree = [[]]  
 self.assertEqual(calculate\_height(tree, 0), 1)  
 def test\_linear\_tree(self):  
 tree = [[1], [2], [3], []] # 0 -> 1 -> 2 -> 3  
 self.assertEqual(calculate\_height(tree, 0), 4)  
 def test\_binary\_tree(self):  
 tree = [[1, 2], [3, 4], [], [], []] # 0 -> {1, 2}, 1 -> {3, 4}  
 self.assertEqual(calculate\_height(tree, 0), 3)  
 def test\_unbalanced\_tree(self):  
 tree = [[1], [2], [3, 4], [5], [], []] # Unbalanced tree structure  
 self.assertEqual(calculate\_height(tree, 0), 5)  
class TestMainFunction(unittest.TestCase):  
 def run\_main(self, input\_data):  
 output = StringIO()  
 main(input\_data, output)  
 return output.getvalue().strip()  
 def test\_main\_example(self):  
 input\_data = """5  
-1 0 4 0 3"""  
 expected\_output = "4"  
 self.assertEqual(self.run\_main(input\_data), expected\_output)  
 def test\_main\_single\_node(self):  
 input\_data = """1  
-1"""  
 expected\_output = "1"  
 self.assertEqual(self.run\_main(input\_data), expected\_output)  
 def test\_main\_linear\_tree(self):  
 input\_data = """4  
-1 0 1 2"""  
 expected\_output = "4"  
 self.assertEqual(self.run\_main(input\_data), expected\_output)  
 def test\_main\_binary\_tree(self):  
 input\_data = """5  
-1 0 0 1 1"""  
 expected\_output = "3"  
 self.assertEqual(self.run\_main(input\_data), expected\_output)  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

Результат работы кода на примерах из текста задачи:

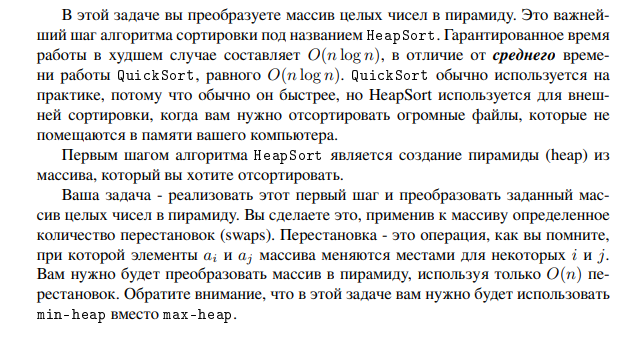


Вывод по задаче:

Решение корректно выполняет вычисление высоты дерева. Алгоритм использует рекурсию и имеет сложность O(n), так как каждый узел посещается один раз

## **Задача №4. Построение пирамиды (по выбору)**

Текст задачи:



Листинг кода:

import os  
import sys  
def build\_heap(data):  
 *"""  
 Преобразует массив data в min-heap.  
 Возвращает список сделанных перестановок.  
 """* swaps = []  
 n = len(data)  
 def sift\_down(i):  
 *"""  
 Просеивает элемент вниз по правилу min-heap.  
 """* min\_index = i  
 left = 2 \* i + 1  
 if left < n and data[left] < data[min\_index]:  
 min\_index = left  
 right = 2 \* i + 2  
 if right < n and data[right] < data[min\_index]:  
 min\_index = right  
 if i != min\_index:  
 swaps.append((i, min\_index))  
 data[i], data[min\_index] = data[min\_index], data[i]  
 sift\_down(min\_index)  
 # Строим кучу начиная с последнего родителя  
 for i in range(n // 2 - 1, -1, -1):  
 sift\_down(i)  
 return swaps  
  
def main(input\_data, output\_file):  
 lines = input\_data.strip().split("\n")  
 n = int(lines[0])  
 data = list(map(int, lines[1].split()))  
 swaps = build\_heap(data)  
 output\_file.write(f"{len(swaps)}\n")  
 for i, j in swaps:  
 output\_file.write(f"{i} {j}\n")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

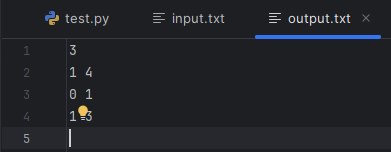
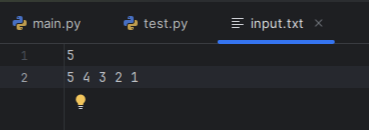
Текстовое объяснение задачи:

Необходимо преобразовать массив целых чисел в пирамиду, используя алгоритм преобразования массива в min-heap (минимальную пирамиду).Для создания пирамиды используется метод "просеивания" элементов, начиная с последнего родительского узла (индекс n/2−1n/2 - 1n/2−1) и двигаясь до корня. Для каждого узла выполняется операция sift\_down(), которая гарантирует, что элемент перемещается на правильное место в пирамиде. Я реализовала функцию build\_heap(), которая строит минимальную пиромиду. Она выполняет ерестановки элементов с помощью операции обмена и записывает их в список

Тесты:

import unittest  
from lab5.task4.src.main import build\_heap  
  
class TestBuildHeap(unittest.TestCase):  
 def test\_example\_1(self):  
 data = [5, 4, 3, 2, 1]  
 swaps = build\_heap(data)  
 self.assertEqual(data, [1, 2, 3, 5, 4])  
 self.assertTrue(len(swaps) <= 4 \* len(data))  
  
 def test\_example\_2(self):  
 data = [1, 2, 3, 4, 5]  
 swaps = build\_heap(data)  
 self.assertEqual(data, [1, 2, 3, 4, 5])  
 self.assertEqual(len(swaps), 0)  
  
 def test\_large\_case(self):  
 data = [i for i in range(1000, 0, -1)]  
 swaps = build\_heap(data)  
 self.assertTrue(len(swaps) <= 4 \* len(data))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

Результат работы кода на примерах из текста задачи:

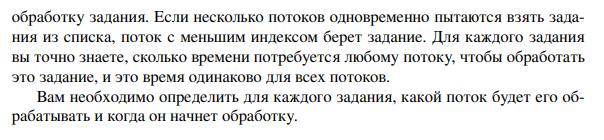
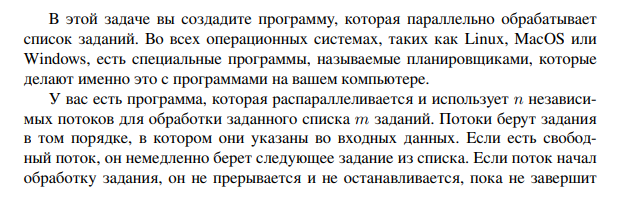


Вывод по задаче:

Задача решена корректно, пирамидальная структура построена с использованием O(n) перестановок. Алгоритм работает эффективно, соблюдая требования задачи.

## **Задача №5. Планировщик заданий**

Текст задачи:



Листинг кода:

import heapq  
import sys  
import os  
def task\_scheduler(n, tasks):  
 # Очередь с приоритетом для потоков: (время освобождения, индекс потока)  
 heap = [(0, i) for i in range(n)]  
 heapq.heapify(heap)  
 result = []  
 for task\_time in tasks:  
 # Достаем поток с минимальным временем освобождения  
 free\_time, thread\_index = heapq.heappop(heap)  
 # Добавляем результат  
 result.append((thread\_index, free\_time))  
 # Обновляем время освобождения потока и возвращаем его в очередь  
 heapq.heappush(heap, (free\_time + task\_time, thread\_index))  
 return result  
  
def main(input\_data, output\_file):  
 lines = input\_data.strip().split("\n")  
 n, m = map(int, lines[0].split())  
 tasks = list(map(int, lines[1].split()))  
 result = task\_scheduler(n, tasks)  
 for thread\_index, start\_time in result:  
 output\_file.write(f"{thread\_index} {start\_time}\n")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

Текстовое объяснение задачи:

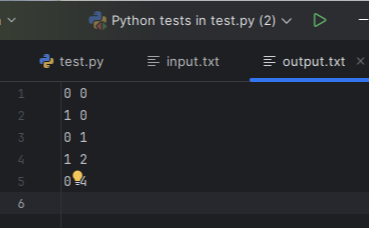
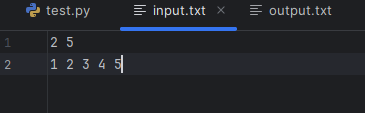
Необходимо спланировать выполнение набора заданий с использованием nnn потоков. Каждый поток обрабатывает задание в порядке очереди, и задачи обрабатываются параллельно, с учетом времени выполнения и индекса потока.Для решения задачи используется очередь с приоритетом, которая отслеживает, какой поток освободится первым. Каждый поток берет задание в порядке очереди, и время его освобождения обновляется после выполнения задания.

Реализована функция task\_scheduler(), которая управляет потоками, используя очередь с приоритетом. Алгоритм эффективно распределяет задания по потокам с минимальным временем освобождения.

Тесты:

import unittest  
from lab5.task5.src.main import task\_scheduler  
  
class TestTaskScheduler(unittest.TestCase):  
 def test\_case\_1(self):  
 n, tasks = 2, [1, 2, 3, 4, 5]  
 expected = [  
 (0, 0),  
 (1, 0),  
 (0, 1),  
 (1, 2),  
 (0, 4)  
 ]  
 self.assertEqual(task\_scheduler(n, tasks), expected)  
  
 def test\_case\_2(self):  
 n, tasks = 4, [1, 1, 1, 1, 1, 1, 1, 1]  
 expected = [  
 (0, 0),  
 (1, 0),  
 (2, 0),  
 (3, 0),  
 (0, 1),  
 (1, 1),  
 (2, 1),  
 (3, 1),  
 ]  
 self.assertEqual(task\_scheduler(n, tasks), expected)  
  
 def test\_case\_3(self):  
 n, tasks = 1, [10, 5, 15]  
 expected = [  
 (0, 0),  
 (0, 10),  
 (0, 15),  
 ]  
 self.assertEqual(task\_scheduler(n, tasks), expected)  
  
 def test\_case\_4(self):  
 n, tasks = 3, [3, 3, 3, 3, 3]  
 expected = [  
 (0, 0),  
 (1, 0),  
 (2, 0),  
 (0, 3),  
 (1, 3),  
 ]  
 self.assertEqual(task\_scheduler(n, tasks), expected)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

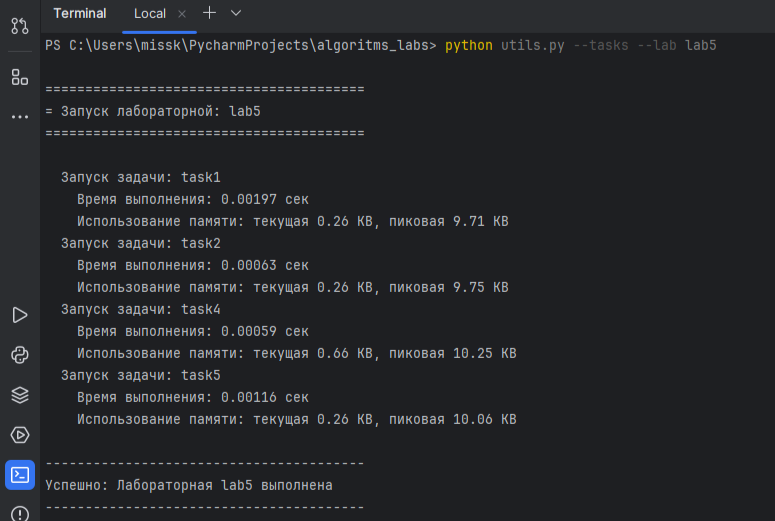
Результат работы кода на примерах из текста задачи:



Вывод по задаче:

Использование очереди с приоритетом позволяет оптимально распределять задачи между потоками

# Запуск всех задач:

****

# Вывод

В ходе выполнения лабораторной работы были реализованы алгоритмы для работы с деревьями и кучами, а также для планирования задач в многозадачных системах. Все задачи решены с использованием эффективных алгоритмов, и решения подтверждены корректностью работы на тестовых данных. Особое внимание было уделено правильной реализации структур данных, таких как пирамиды и деревья, а также эффективному распределению задач между потоками с использованием очередей с приоритетом