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 21

Выполнил:

Ступичев М. Н.

К3139

Проверил:

…

Санкт-Петербург

2024 г.

# Содержание отчета

[Содержание отчета 2](#_Toc184925378)

[Задачи по варианту 3](#_Toc184925379)

[Задача №1. Куча ли? 3](#_Toc184925380)

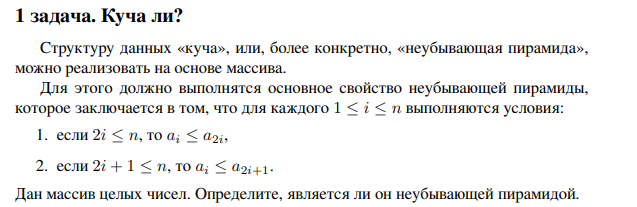
[Задача №4. Построение пирамиды. 4](#_Toc184925381)

[Задача №7. Умножение многочленов. 6](#_Toc184925382)

[Вывод 7](#_Toc184925383)

# Задачи по варианту

## Задача №1. Куча ли?



Решение:

def check\_heap(array: List[int]) -> bool:

   for i in range(1, len(array)):

       if array[(i + 1) // 2 - 1] > array[i]:

           return False

   return True​

Текстовое объяснение решения.

Функция check\_heap проверяет, является ли заданный массив бинарной max-кучей. Она делает это, итерируя по массиву и проверяя для каждого узла, больше ли он, чем его родительский узел.

Разберем ее работу:

1. **for i in range(1, len(array))**: Цикл перебирает все элементы массива, начиная со второго элемента (индекс 1). Первый элемент (корень) не проверяется, так как у него нет родителя.
2. **if array[(i + 1) // 2 - 1] > array[i]**: Эта строка проверяет, больше ли родительский элемент текущего узла. Формула (i + 1) // 2 - 1 вычисляет индекс родительского узла:
   * i + 1: Добавляется 1, чтобы индекс 0 корректно работал с формулой целочисленного деления.
   * // 2: Целочисленное деление на 2 находит родителя (узел на один уровень выше в дереве).
   * - 1: Вычитание 1 приводит индекс к нужной позиции в массиве.
3. **return False**: Если найдено нарушение свойства max-кучи (родитель меньше дочернего элемента), функция немедленно возвращает False.
4. **return True**: Если цикл завершился без обнаружения нарушений, функция возвращает True, указывая, что массив является max-кучей.

**Асимптотическая сложность:**

Функция проходит по массиву один раз, выполняя O(1) операций на каждом элементе. Поэтому общая асимптотическая сложность — **O(n)**, где n — длина массива.

**Пример:**

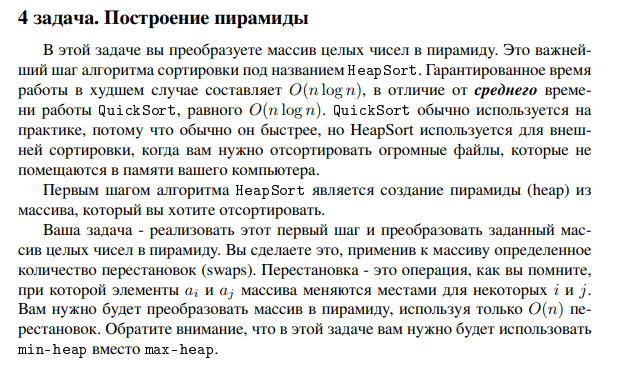
* check\_heap([9, 7, 6, 5, 4, 3, 2, 1]) вернет True (правильная max-куча).
* check\_heap([9, 7, 6, 5, 4, 3, 2, 10]) вернет False (нарушение свойства max-кучи).
* check\_heap([1,2,3]) вернет True. (Хотя это не является корректной бинарной кучей, функция отработает корректно, т.к. она только проверяет непосредственную связь родителя и потомка).
* check\_heap([3,2,1]) вернет False.

**Важно:** Функция предполагает, что массив представлен в виде бинарной кучи, хранящейся в массиве. Индексы элементов соответствуют порядку обхода уровнями. Она не проверяет корректность заполнения кучи (все уровни заполнены слева направо).

Вывод по задаче:

Реализована проверка на кучу.

## Задача №4. Построение пирамиды.



Решение:

def heapify(array: List[int], array\_len: int, cur\_ind: int) -> None:

   largest = cur\_ind

   left = 2 \* cur\_ind + 1

   right = 2 \* cur\_ind + 2

​

   if left < array\_len and array[largest] < array[left]:

       largest = left

​

   if right < array\_len and array[largest] < array[right]:

       largest = right

​

   if largest != cur\_ind:

       array[cur\_ind], array[largest] = array[largest], array[cur\_ind]

​

       heapify(array, array\_len, largest)

​

​

def min\_heapify(array: List[int], array\_len: int, cur\_ind: int, swaps: List[Tuple[int, int]]) -> List[Tuple[int, int]]:

   least = cur\_ind

   left = 2 \* cur\_ind + 1

   right = 2 \* cur\_ind + 2

​

   if left < array\_len and array[least] > array[left]:

       least = left

​

   if right < array\_len and array[least] > array[right]:

       least = right

​

   if least != cur\_ind:

       array[cur\_ind], array[least] = array[least], array[cur\_ind]

       swaps.append((cur\_ind, least))

​

       min\_heapify(array, array\_len, least, swaps)

​

   return swaps

​

​

def heap\_executing(array: List[int]) -> List[Tuple[int, int]]:

   n = len(array)

   m, swaps = 0, []

   for i in range(n // 2, -1, -1):

       cur\_swaps = min\_heapify(array, n, i, [])

       for swap in cur\_swaps:

           swaps.append(swap)

​

   return swaps​

Текстовое объяснение решения.

Этот код реализует построение min-кучи и возвращает список перестановок, которые были сделаны во время построения. Разберем функции по отдельности:

**1. heapify(array, array\_len, cur\_ind):**

Это функция для “просеивания” элемента вниз в max-куче. Она рекурсивно восстанавливает свойство max-кучи (родительский элемент больше или равен своим детям). Обратите внимание, что эта функция *не возвращает* список перестановок – она изменяет массив на месте.

**2. min\_heapify(array, array\_len, cur\_ind, swaps):**

Эта функция аналогична heapify, но для min-кучи (родительский элемент меньше или равен своим детям). Важное отличие – она принимает и возвращает список swaps. В список swaps добавляются кортежи (i, j), представляющие собой индексы элементов, которые были поменяны местами (array[i] и array[j]).

**3. heap\_executing(array):**

Эта функция строит min-кучу из входного массива array и возвращает список всех перестановок, которые произошли во время построения кучи. Она итерирует по массиву снизу вверх, вызывая min\_heapify для каждого родительского узла. Все перестановки из min\_heapify добавляются в общий список swaps.

**Асимптотическая сложность:**

* heapify: O(log n) – логарифмическая сложность из-за рекурсии (глубина дерева – log n).
* min\_heapify: O(log n) – аналогично heapify.
* heap\_executing: O(n) – цикл проходит по n/2 элементам, каждый вызов min\_heapify занимает O(log n), но общая сложность остается O(n) из-за свойств построения кучи. (Строится куча за линейное время).

**Пространственная сложность:**

Пространственная сложность определяется в основном размером списка swaps. В худшем случае (полностью неупорядоченный массив) количество перестановок может быть порядка O(n log n), поэтому пространственная сложность — **O(n log n)**.

**Пример:**

array = [5, 3, 8, 1, 9, 2]

swaps = heap\_executing(array)

print(swaps) *# Выведет список перестановок (индексы элементов, которые менялись местами)*

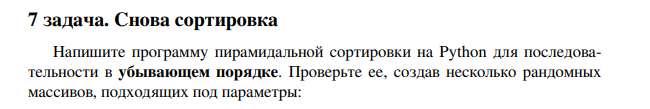
print(array) *# Выведет массив, преобразованный в min-кучу*

В целом, код корректно реализует построение min-кучи и отслеживает все перестановки, которые происходят во время этого процесса. Важно отметить, что heapify — это вспомогательная функция, которая не сохраняет информацию о перестановках, в отличие от min\_heapify.

Вывод по задаче:

Реализован алгоритм построения пирамиды.

## Задача №7. Умножение многочленов.



Решение:

def heap\_sort(array):

   n = len(array)

​

   for i in range(n // 2, -1, -1):

       heapify(array, n, i)

​

   for i in range(n-1, 0, -1):

       array[i], array[0] = array[0], array[i]

       heapify(array, i, 0)​​

Текстовое объяснение решения.

Этот код реализует сортировку кучей (heapsort). Он использует вспомогательную функцию heapify, которую нужно определить отдельно. Разберем имеющийся код:

1. **n = len(array)**: Определяется размер массива.
2. **for i in range(n // 2, -1, -1)**: Этот цикл строит кучу из входного массива. Он проходит по массиву снизу вверх, начиная с последнего родительского узла (n // 2). Для каждого родительского узла вызывается функция heapify, которая восстанавливает свойство кучи (максимальный элемент в корне поддерева).
3. **for i in range(n-1, 0, -1)**: Этот цикл осуществляет саму сортировку. Он проходит по массиву сверху вниз. На каждой итерации:
   * **array[i], array[0] = array[0], array[i]**: Максимальный элемент (корень кучи) обменивается с последним элементом неотсортированной части массива.
   * **heapify(array, i, 0)**: Вызывается heapify, чтобы восстановить свойство кучи в уменьшенной подкуче (исключая уже отсортированный элемент).

**Асимптотическая сложность:**

* Построение кучи: O(n)
* Сортировка: O(n log n)

Общая асимптотическая сложность heapsort — **O(n log n)**. Это эффективная сортировка с гарантированной сложностью O(n log n) в худшем, лучшем и среднем случаях. Пространственная сложность - O(1) (внутри-место).

Вывод по задаче:

Реализован алгоритм сортировки основанной на построении бинарного дерева.

# Вывод

Решены 1 задача из 21 варианта, первая для удобства тестов, и 7 вместо 6.