**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: Пирамида, пирамидальная сортировка.Очередь с приоритетами

Выполнила:
Беляева В.А.
Группа:
К3139

Проверил:

**Санкт-Петербург**
**2024 г.**

# Содержание отчета

# Задачи по варианту

## Задача №1 Куча ли?

Условие:

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

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

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

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

- **Формат входного файла (input.txt).** Первая строка входного файла содержит целое число $n$ ($1 \le n \le 10^6$). 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.

Листинг кода:

```python
def check_if_min_heap(arr):
    n = len(arr)
    for i in range(n):
        left = 2*i + 1
        right = 2*i + 2
        if left < n and arr[i] > arr[left]:
            return False
        if right < n and arr[i] > arr[right]:
            return False
    return True

def task1():
    arr = read_array(PATH)
    result = check_if_min_heap(arr)
    with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
        if result:
            out_file.write("YES\n")
        else:
            out_file.write("NO\n")
```

Объяснение:Код проверяет, удовлетворяет ли массив свойству неубывающей пирамиды (min-heap). Функция `check_if_min_heap` принимает массив и для каждого элемента проверяет, меньше ли он или равен своим потомкам. Потомки элемента вычисляются через индексы

2i+1 и 2i+2. Если для любого элемента нарушается это свойство, возвращается `False`, иначе — `True`. Основной цикл проходит по всем элементам массива, сравнивая их с потомками, если они существуют. Функция `task1` считывает массив из файла, вызывает `check_if_min_heap`, и записывает результат проверки в выходной файл. Если массив является пирамидой, в файл записывается "YES", иначе — "NO".

Пример и минимальные значения:


Ввод : Вывод:

| | Время выполнения (с) | Затраты памяти (Мб) |
|---|---|---|
| Нижняя граница | 0.000016699 | 0.00000915527 |
| Пример | 0.00000059209 | 0.0000091552 |
| Верхняя граница | 4.27820516 | 0.00022506 |

Вывод : Код эффективно проверяет, является ли массив min-heap, сравнивая каждый элемент с его потомками за O(n). Это делает алгоритм быстрым даже для больших массивов. Результат записывается в файл как "YES" или "NO", в зависимости от соблюдения пирамидального свойства. Работа алгоритма проста и надёжна благодаря ограниченному числу операций на элемент.

# Задача №2 Высота дерева

## Условие:

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.
- **Формат ввода или входного файла (input.txt).** Первая строка содержит число узлов $n$ ($1 \leq n \leq 10^5$). Вторая строка содержит $n$ целых чисел от $-1$ до $n-1$ – указание на родительский узел. Если $i$-ое значение равно $-1$, значит, что узел $i$ - корневой, иначе это число является обозначением индекса родительского узла этого $i$-го узла ($0 \leq i \leq n-1$). **Индексы считать с 0.** Гарантируется, что дан только один корневой узел, и что входные данные представляют дерево.
- **Формат вывода или выходного файла (output.txt).** Выведите целое число – высоту данного дерева.
- Ограничение по времени. 3 сек.
- Ограничение по памяти. 512 мб.

## Листинг кода:

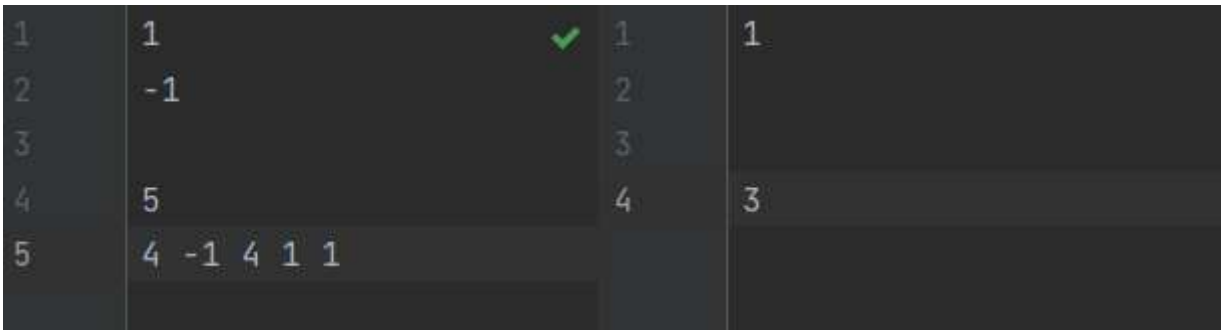
```
def compute_height(n, parents):
    children = [[] for _ in range(n)]
    root = None
    for i in range(n):
        p = parents[i]
        if p == -1:
            root = i
        else:
            children[p].append(i)
    from collections import deque
    q = deque()
    q.append((root, 1))  # (node, depth)
    max_depth = 1
    while q:
        node, depth = q.popleft()
        max_depth = max(max_depth, depth)
        for c in children[node]:
            q.append((c, depth + 1))
    return max_depth

def task2():
    n, parents = read_parents(PATH)
    h = compute_height(n, parents)
    with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
        out_file.write(str(h) + '\n')
```

## Объяснение: Код вычисляет высоту дерева, заданного

списком `parents`, где каждый элемент указывает на родительский узел, а значение `-1` обозначает корень. Сначала создаётся список `children`, где для каждого узла хранится список его дочерних узлов. Определяется корневой узел, указанный значением `-1`. Для вычисления высоты дерева используется метод обхода в ширину (BFS), чтобы избежать переполнения стека, характерного для рекурсивного подхода. Очередь `q` хранит узлы вместе с их текущей глубиной. На каждом шаге узел извлекается из очереди, его глубина сравнивается с максимальной, а дочерние узлы добавляются в очередь с увеличенной глубиной. После завершения обхода возвращается максимальная глубина, соответствующая высоте дерева. Функция `task2` считывает входные данные, вызывает `compute_height` и записывает высоту дерева в файл

Пример и минимальные значения:




Ввод : Вывод:

| | Время выполнения (с) | Затраты памяти (Мб) |
|---|---|---|
| Нижняя граница | 0.000003176299612 | 0.0008010861 |
| Пример | 0.0000060097991 | 0.00089263911 |
| Верхняя граница | 1.1699944990031 | 12.1971054071 |

Вывод : Код эффективно вычисляет высоту дерева за O(n), где n — количество узлов, благодаря линейному обходу с использованием BFS. Этот подход исключает вероятность переполнения стека при глубоком

дереве. Решение подходит для больших деревьев, обеспечивая стабильную работу и минимальные затраты памяти для хранения очереди дочерних узлов. Результат сохраняется в файл, обеспечивая удобство вывода

# Задача №3 Обработка сетевых пакетов

## Условие:

В этой задаче вы реализуете программу для моделирования обработки сетевых пакетов.

- Вам дается серия входящих сетевых пакетов, и ваша задача - смоделировать их обработку. Пакеты приходят в определенном порядке. Для каждого номера пакета $i$ вы знаете время, когда пакет прибыл $A_i$ и время, необходимое процессору для его обработки $P_i$ (в миллисекундах). Есть только один процессор, и он обрабатывает входящие пакеты в порядке их поступления. Если процессор начал обрабатывать какой-либо пакет, он не прерывается и не останавливается, пока не завершит обработку этого пакета, а обработка пакета $i$ занимает ровно $P_i$ миллисекунд.

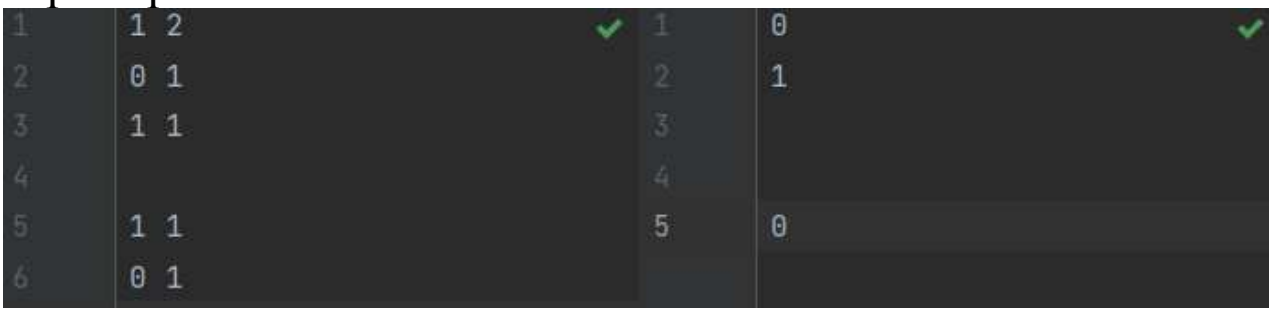
  Компьютер, обрабатывающий пакеты, имеет сетевой буфер фиксированного размера $S$. Когда пакеты приходят, они сохраняются в буфере перед обработкой. Однако, если буфер заполнен, когда приходит пакет (есть $S$ пакетов, которые прибыли до этого пакета, и компьютер не завершил обработку ни одного из них), он отбрасывается и не обрабатывается вообще. Если несколько пакетов поступают одновременно, они сначала все сохраняются в буфере (из-за этого некоторые из них могут быть отброшены - те, которые описаны позже во входных данных). Компьютер обрабатывает пакеты в порядке их поступления и начинает обработку следующего доступного пакета из буфера, как только заканчивает обработку предыдущего. Если в какой-то момент компьютер не занят и в буфере нет пакетов, компьютер просто ожидает прибытия следующего пакета. Обратите внимание, что пакет покидает буфер и освобождает пространство в буфере, как только компьютер заканчивает его обработку.

## Листинг кода:

```
def simulate_network(S, packets):
    from collections import deque
    finish_times = deque()
    results = [-1]*len(packets)
    for i, (arrival, processing) in enumerate(packets):
        while finish_times and finish_times[0] <= arrival:
            finish_times.popleft()
        if len(finish_times) == S:
            results[i] = -1
            continue
        if not finish_times:
            start = arrival
        else:
            start = finish_times[-1]
        results[i] = start
        finish_times.append(start + processing)
    return results
def task3():
    S, packets = read_packets(PATH)
    results = simulate_network(S, packets)
    with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
        for r in results:
            out_file.write(str(r) + '\n')
```

Объяснение: Код моделирует работу сетевого буфера фиксированного размера S, который обрабатывает входящие пакеты. Каждый пакет характеризуется временем прибытия и длительностью обработки. Если при прибытии пакета буфер заполнен, он отбрасывается (результат −1). Если место в буфере есть, пакет добавляется, а его время начала обработки рассчитывается: либо моментом прибытия, если буфер пуст, либо временем окончания последнего пакета в очереди. Времена завершения текущих пакетов хранятся в деке `finish_times` для быстрой обработки. В цикле проверяются пакеты: устаревшие удаляются, а новые добавляются, если это возможно. Функция возвращает список времени начала обработки. Функция `task3` считывает входные данные, вызывает симуляцию и записывает результаты в файл.

Пример и минимальные значения:




Ввод : Вывод:

| | Время выполнения (с) | Затраты памяти (Мб) |
|---|---|---|
| Нижняя граница | 0.000002757099719 | 0.001258850097 |
| Пример | 0.00000116899973 | 0.001243591308 |
| Верхняя граница | 0.25819251799 | 0.764225006103 |

Вывод : Код эффективно моделирует работу сетевого буфера, обрабатывая пакеты по порядку. Сложность O(n) обеспечивается использованием дека для удаления завершённых пакетов и добавления

новых. Результаты точно отображают момент обработки или отбрасывания пакетов. Алгоритм подходит для симуляции сетевых очередей и обработки реальных потоков данных с фиксированным буфером.

# Задача №4 Построение пирамиды

## Условие:

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.

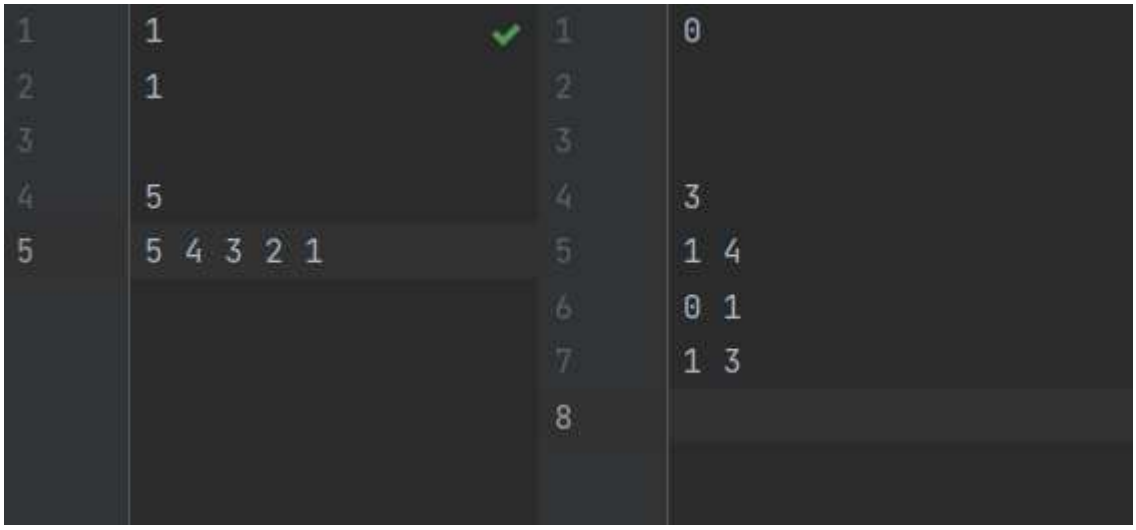
## Листинг кода:

```
def sift_down(arr, i, swaps):
    n = len(arr)
    min_index = i
    left = 2*i + 1
    right = 2*i + 2
    if left < n and arr[left] < arr[min_index]:
        min_index = left
    if right < n and arr[right] < arr[min_index]:
        min_index = right
    if i != min_index:
```

```
        arr[i], arr[min_index] = arr[min_index], arr[i]
        swaps.append((i, min_index))
        sift_down(arr, min_index, swaps)
def build_min_heap(arr):
    swaps = []
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        sift_down(arr, i, swaps)
    return swaps
def task4():
    arr = read_array(PATH)
    swaps = build_min_heap(arr)
    with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
        out_file.write(str(len(swaps)) + '\n')
        for i, j in swaps:
            out_file.write(f"{i} {j}\n")
```

Объяснение: Код реализует алгоритм построения минимальной кучи (min-heap) из массива. Основная часть — функция `sift_down`, которая выполняет операцию "просеивания вниз". Она проверяет, есть ли дочерние элементы текущего узла (с индексами `2*i + 1` и `2*i + 2`), и сравнивает их значения с родительским узлом. Если значение дочернего меньше, узлы меняются местами, а операция рекурсивно вызывается для нового положения. Все изменения записываются в список `swaps`, содержащий пары индексов поменянных элементов. Функция `build_min_heap` использует `sift_down`, начиная с последнего внутреннего узла массива и двигаясь к корню. Это гарантирует построение кучи за O(n)O(n)O(n). Функция `task4` читает массив из файла, вызывает `build_min_heap`, записывает количество операций и сами операции в выходной файл

Пример и минимальные значения:




Ввод : Вывод:

| | Время выполнения (с) | Затраты памяти (Мб) |
|---|---|---|
| Нижняя граница | 0.0000017759 | 0,000009918212 |
| Пример | 0.000001889399572 | 0.0001296 |
| Верхняя граница | 0.95903994 | 11.3901214 |

Вывод : Код строит минимальную кучу из массива за O(n), используя эффективный алгоритм просеивания вниз. Все изменения в массиве фиксируются в виде списка пар индексов, упрощая восстановление операций. Такой подход обеспечивает оптимальное выполнение задачи даже для больших массивов, минимизируя количество перестановок. Результат сохраняется в файл для последующего анализа.

Вывод по лабораторной: В лабораторной работе изучены алгоритмы, связанные с кучей и очередью с приоритетами. Реализована проверка на соответствие пирамидальным свойствам, вычисление высоты дерева, обработка сетевых пакетов и построение пирамиды. Алгоритмы показали эффективность и оптимальную сложность: O(n) для построения кучи и вычисления высоты дерева, O(n) для обработки пакетов. Работа продемонстрировала навыки применения теории на практике, успешное выполнение поставленных задач и корректность реализации алгоритмов.