

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 3

Выполнила:
Блинова П. В.
К3139 (номер группы)

Проверил:
Афанасьев А. В.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №1. Куча ли?	3
Задача №4. Построение пирамиды	5
Дополнительные задачи	7
Задача №2. Высота дерева	7
Задача №7. Снова сортировка	10
Вывод	12

Задачи по варианту

Задача №1. Куча ли?

Текст задачи.

1 задача. 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.

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

№	input.txt	output.txt
1	5 1 0 1 2 0	NO
2	5 1 3 2 5 4	YES

Листинг кода.

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

Текстовое объяснение решения.

Этот код реализует функцию `is_heap`, которая проверяет, является ли заданный массив кучей (пирамидой) или нет. Функция принимает два аргумента: • n :

Размер массива (количество элементов). • `array`: Сам массив, представляющий потенциальную кучу. Куча — это структура данных, удовлетворяющая свойству кучи: значение родительского узла всегда больше или равно (для `max`-кучи) значениям его дочерних узлов. В этом коде предполагается `max`-куча. Функция работает следующим образом: Она итерируется по массиву `array` от начала до конца (от 0 до $n-1$). Для каждого элемента `array[i]` она проверяет: • `if 2 * i + 1 < n and array[i] > array[2 * i + 1]`:: Эта строка проверяет, существует ли левый дочерний узел (индекс $2 * i + 1$) и больше ли значение родительского узла (`array[i]`) значения левого дочернего узла. Если это так, это нарушает свойство кучи, и функция немедленно возвращает "NO". • `if 2 * i + 2 < n and array[i] > array[2 * i + 2]`:: Аналогично, эта строка проверяет правый дочерний узел (индекс $2 * i + 2$). Если значение родительского узла больше значения правого дочернего узла, свойство кучи нарушено, и функция возвращает "NO". Если цикл завершается без обнаружения нарушений свойства кучи, это означает, что массив является кучей, и функция возвращает "YES". В отчете можно описать алгоритм так: Функция `is_heap` проверяет корректность `max`-кучи, используя линейный проход по массиву. Для каждого узла она сравнивает его значение со значениями его левого и правого потомков. Если хотя бы один потомок имеет значение большее, чем родительский узел, функция возвращает "NO", указывая на нарушение свойства кучи. В противном случае, после проверки всех узлов, функция возвращает "YES", подтверждая, что массив представляет собой корректную `max`-кучу.

Результат работы кода на примерах из текста задачи:

```
C:\Users\Slawa\AppData\Local\Programs\
Входные данные:
5
1 0 1 2 0
Результат:
NO
Время работы: 1.7300015315413475e-05
Память: 0.00016021728515625 M6

Process finished with exit code 0
```

Вывод по задаче:

В ходе работы над задачей мной была изучена куча. Алгоритм имеет временную сложность $O(n)$, поскольку он проходит по массиву один раз.

Задача №4. Построение пирамиды

Текст задачи.

4 задача. Построение пирамиды

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.)

Листинг кода.

```
def move_down(i, n, data, swaps):
    min_index = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and data[left] < data[min_index]:
        min_index = left

    if right < n and data[right] < data[min_index]:
        min_index = right

    if i != min_index:
        swaps.append((i, min_index)) # Сохраняем пару обменов
        data[i], data[min_index] = data[min_index], data[i] # Перестановка
        move_down(min_index, n, data, swaps) # Рекурсивный вызов

def build_heap(n, nums):
    swaps = []
```

```
for i in range(n // 2 - 1, -1, -1):  
    move_down(i, n, nums, swaps)  
  
return swaps
```

Текстовое объяснение решения.

Алгоритм `build_heap` строит `min`-кучу снизу вверх. Он использует рекурсивную функцию `move_down`, которая просеивает элементы вниз по куче, восстанавливая свойство `min`-кучи. Алгоритм начинается с проверки нелистовых узлов (с индекса $n // 2 - 1$ до 0) и просеивает каждый узел вниз до тех пор, пока не будет удовлетворено свойство `min`-кучи. Список `swaps` позволяет отслеживать все произведенные перестановки. Временная сложность алгоритма составляет $O(n)$, что делает его эффективным для построения кучи.

Результат работы кода на примерах из текста задачи.

```
Входные данные:  
5  
5 4 3 2 1  
Результат:  
3  
1 4  
0 1  
1 3  
Время работы: 1.9700033590197563e-05  
Память: 0.00016021728515625 МБ
```

Вывод по задаче:

В ходе работы над задачей мной был изучен способ построения пирамиды. Алгоритм имеет линейную временную сложность $O(n)$.

Дополнительные задачи

Задача №2. Высота дерева

Текст задачи.

2 задача. 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.
- **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.
- Пример 1:

input.txt	output.txt
5	3
4 -1 4 1 1	

- **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.
- Пример 1:

input.txt	output.txt
5	3
5 4 3 2 1	1 4
	0 1
	1 3

После перестановки элементов в позициях 1 и 4 массив становится следующим: 5 1 3 2 4.

Далее, перестановка элементов с индексами 0 и 1: 1 5 3 2 4. И напоследок, переставим 1 и 3: 1 2 3 5 4, и теперь это корректная неубывающая пирамида.

- Пример 2:

input.txt	output.txt
5	0
1 2 3 4 5	

Листинг кода.

```
def build_tree(n, parents):
    tree = [[] for i in range(n)]
    root = None
    for child in range(n):
        parent = parents[child]
        if parent == -1:
            root = child
```



```

        else:
            tree[parent].append(child)
    return tree, root

def find_tree_height(tree, node):
    if not tree[node]: # Если у узла нет детей
        return 1
    else:
        height = 0
        for child in tree[node]:
            height = max(height, find_tree_height(tree, child))
        return height + 1

```

Текстовое объяснение решения.

Код эффективно строит и обходит дерево. `build_tree` использует представление дерева в виде списка смежности, что является экономичным по памяти. `find_tree_height` рекурсивно вычисляет высоту, эффективно проходя по дереву. Рекурсивный подход понятен, но для очень глубоких деревьев может привести к переполнению стека. Для больших деревьев можно рассмотреть итеративный вариант. В целом, код демонстрирует корректную и эффективную работу с древовидными структурами данных. Функции `build_tree` и `find_tree_height` представляют собой классический подход к построению и обработке деревьев в программировании.

Результат работы кода на примерах из текста задачи.

```

Входные данные:
5
-1 0 4 0 3
Результат:
4
Время работы: 3.400002606213093e-05
Память: 0.00148773193359375 Мб

Process finished with exit code 0

```

Вывод по задаче:

В ходе работы над задачей мной был изучен способ определения высоты дерева.

Задача №7. Снова сортировка

Текст задачи.

7 задача. Снова сортировка

Напишите программу пирамидальной сортировки на Python для последовательности в **убывающем порядке**. Проверьте ее, создав несколько случайных массивов, подходящих под параметры:

- **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 10^5$) — число элементов в массиве. Во второй строке находятся n различных целых чисел, *по модулю* не превосходящих 10^9 .

11

-
- **Формат выходного файла (output.txt).** Одна строка выходного файла с отсортированным по невозрастанию массивом. Между любыми двумя числами должен стоять ровно один пробел.

Листинг кода.

```
def heapify(lst, n, i):  
    smallest = i  
    left = 2 * i + 1  
    right = 2 * i + 2
```

```

    if left < n and lst[left] < lst[smallest]:
        smallest = left

    if right < n and lst[right] < lst[smallest]:
        smallest = right

    if smallest != i:
        lst[i], lst[smallest] = lst[smallest], lst[i]
        heapify(lst, n, smallest)

def heap_sort(n, lst):
    for i in range(n // 2 - 1, -1, -1):
        heapify(lst, n, i)

    for i in range(n - 1, 0, -1):
        lst[i], lst[0] = lst[0], lst[i]
        heapify(lst, i, 0)

    return lst

```

Текстовое объяснение решения.

Представленный код реализует алгоритм сортировки кучей (пирамидальная сортировка). Алгоритм основан на использовании структуры данных «куча» (в данном случае, минимальная куча). Код состоит из двух функций: `heapify` и `heap_sort`. Функция `heapify(lst, n, i)`: Эта вспомогательная функция поддерживает свойство минимальной кучи в поддереве с корнем в узле с индексом `i`. Она принимает на вход: • `lst`: список, который нужно отсортировать (внутри процедуры). • `n`: размер кучи (или релевантной части списка, рассматриваемой как куча). • `i`: индекс узла, который необходимо «просеять» (`heapify`). Функция находит минимальный элемент среди узла с индексом `i` и его потомков (левого и правого). Если минимальный элемент не находится в узле `i`, происходит обмен, и функция рекурсивно вызывается для поддерева с корнем в индексе поменянного элемента, чтобы восстановить свойство минимальной кучи. Функция `heap_sort(n, lst)`: Эта функция выполняет алгоритм пирамидальной сортировки. Она принимает на вход: • `n`: размер списка. • `lst`: список, который нужно отсортировать (внутри процедуры). Сначала функция строит минимальную кучу из входного списка, используя функцию `heapify`. Затем, в цикле, она многократно извлекает минимальный элемент (который находится в корне кучи, индекс 0) и помещает его в конец отсортированной части. После извлечения минимального элемента функция `heapify` вызывается снова, чтобы восстановить свойство

минимальной кучи для оставшихся элементов. Этот процесс повторяется до тех пор, пока весь список не будет отсортирован.

Результат работы кода на примерах из текста задачи.

```
Входные данные:
5
98 67 35 3 24
Результат:
98 67 35 24 3
Время работы: 7.71000050008297e-05
Память: 0.001026153564453125 Мб

Process finished with exit code 0
```

Вывод по задаче:

В ходе работы над задачей мной был изучен способ пирамидальной сортировки.

Вывод

В ходе лабораторной работы был изучен метод работы со следующими структурами данных: деревья, пирамида или двоичная куча, очередь с приоритетами, а также еще одному виду сортировки за $n \log n$: пирамидальной (heapsort).