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.
Вариант 15

Выполнил:
Скворцов Д.А.
К3140

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №3. Обработка сетевых пакетов

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 Ai и время, необходимое процессору для его обработки Pi (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 Pi 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.
- **Формат ввода или входного файла (input.txt).** Первая строка содержит размер S буфера ($1 \le S \le 10^5$) и количество n ($1 \le n \le 10^5$) входящих сетевых пакетов. Каждая из следующих n строк содержит два числа, i-ая строка содержит время прибытия пакета Ai ($0 \le Ai \le 10^6$) и время его обработки Pi ($0 \le Pi \le 10^3$) в миллисекундах. Гарантируется, что последо вательность времени прибытия входящих пакетов – неубывающая, однако, она может содержать

одинаковые значения времени прибытия нескольких пакетов, в этом случае рассматривается пакет, записанный в входном файле раньше остальных, как прибывший ранее. ($A_i \le A_{i+1}$ для $1 \le i \le n-1$.)

- **Формат вывода или выходного файла (output.txt).** Для каждого пакета напечатайте время (в миллисекундах), когда процессор начал его обрабатывать; или-1, если пакет был отброшен. Вывести ответ нужно в том же порядке, как как пакеты были описаны во входном файле.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.

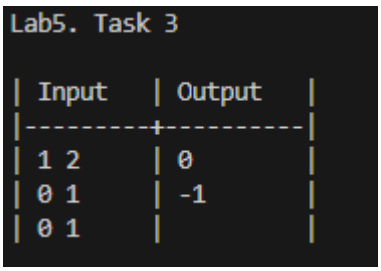
Листинг кода.

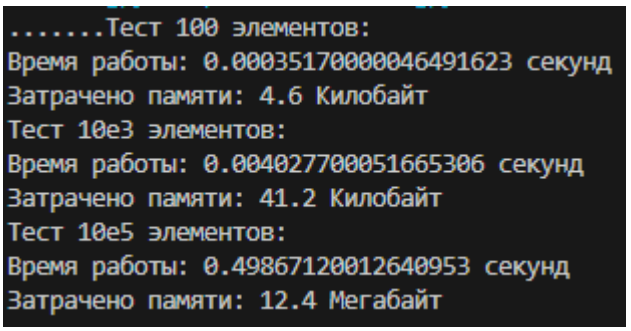
```
class Node:
    def __init__(self, time, next=None, prev=None):
        self.time = time
        self.next = next
        self.prev = prev

    def copy(self):
        return Node(self.time, self.next, self.prev)


class Queue:
    def __init__(self):
        self.head = None
        self.tail = None

    def put(self, node: Node):
        if self.head is None:
            self.head = node
        elif self.tail is None:
            self.tail = node
            self.tail.next = self.head
            self.head.prev = self.tail
        else:
            prev_tail = self.tail
            self.tail = node
            self.tail.next = prev_tail
            prev_tail.prev = self.tail
```

```
    def pop(self):
        prev_head = self.head
        if prev_head is not None:
            new_head = self.head.prev
            if new_head is not None:
                new_head.next = None
            self.head = new_head

        return prev_head

    def peek(self):
        if self.head is not None:
            return self.head.copy()
        return None

    def peek_tail(self):
        if self.tail is not None:
            return self.tail.copy()
        return self.peek()

    def print(self):
        cur = self.head
        res = []
        while cur != None:
            res.append((cur.time))
            cur = cur.prev
        print(res)


class Buffer(Queue):
    def __init__(self):
        super().__init__()
        self.len = 0

    def put(self, node: Node):
        super().put(node)
        self.len += 1

    def pop(self):
        ret = super().pop()
        if ret is not None:
            self.len -= 1
        return ret
```

```
def solution(buffer_size, packages):
    buffer = Buffer()
    ans = []

    for package in packages:
        arrive_time, process_time = map(int, package.split())
            while buffer.peek() is not None and buffer.head.time <=
arrive_time:
            buffer.pop()

        if buffer.len >= buffer_size:
            ans.append(-1)
        else:
            tail = buffer.peek_tail()
            if tail is None:
                start_time = arrive_time
            else:
                start_time = tail.time

            ans.append(start_time)

            buffer.put(Node(start_time + process_time))

    return ans
```

Обработчик пакетов реализован через очередь. Пакеты накапливаются в ней до тех пор, пока их время прибытия не становится большим или равным времени выполнения первого пакета в очереди.





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0003517000046491623 секунд	4.6 Килобайт
Медиана диапазона значений входных данных из текста задачи	0.004027700051665306 секунд	41.2 Килобайт
Верхняя граница диапазона значений входных данных из текста задачи	0.49867120012640953 секунд	12.4 Мегабайт

Вывод по задаче:
Алгоритм работает за линейное время, но затрачивает много памяти для сохранения всех пакетов в очереди. Использование не просто стэка, а очереди значительно повышает эффективность.

Задача №7. Снова сортировка

Напишите программу пирамидальной сортировки на Python для последова тельности в убывающем порядке. Проверьте ее, создав несколько рандомных массивов, подходящих под параметры:

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

- Формат выходного файла (output.txt). Одна строка выходного файла с отсортированным по возрастанию массивом.Между любыми двумя числами должен стоять ровно один пробел.
- Для проверки можно выбрать случай, когда сортируется массив размера 10^3,10^4,10^5 чисел порядка 10^9, отсортированных в обратном порядке; когда массив уже отсортирован в нужном порядке; когда много одинаковых элементов, всего 4-5 уникальных; средний-случайный. Сравните на данных сетах Randomized-QuickSort, MergeSort, HeapSort, InsertionSort.
- Есть ли случай, когда сортировка пирамидой выполнится за O(n)?
- Напишите процедуру Max-Heapify, в которой вместо рекурсивного вызова использовалась бы итеративная конструкция (цикл).

Листинг кода.

```
def left(i):
    return 2 * (i+1) - 1

def right(i):
    return 2 * (i+1)



def MinHeapify(lst, i, heap_size=None):
    if heap_size is None:
        heap_size = len(lst)

    while True:
        _l = left(i)
        r = right(i)
        if _l > r:
            break
        if _l < heap_size and lst[_l] < lst[i]:
            smallest = _l
        else:
            smallest = i

        if r < heap_size and lst[r] < lst[smallest]:
            smallest = r

        if smallest != i:
            lst[i], lst[smallest] = lst[smallest], lst[i]
            i = smallest
```

```
        else:
            break

def BuildMinHeap(lst):
    heap_size = len(lst)
    swaps = []
    for i in range(heap_size//2, -1, -1):
        MinHeapify(lst, i)

    return swaps

def HeapSort(lst):
    BuildMinHeap(lst)
    heap_size = len(lst)
    for i in range(len(lst)-1, 0, -1):
        lst[0], lst[i] = lst[i], lst[0]
        heap_size = heap_size - 1
        MinHeapify(lst, 0, heap_size)


def solution(lst, sort_func=HeapSort):
    res = lst.copy()
    if sort_func == HeapSort:
        sort_func(res)
    else:
        sort_func(res, 0, len(res)-1)
    return res
```

Используем функционал стека, чтобы запоминать порядок наложения скобок. При любом несоответствии возвращаем индекс символа, на котором всё сломалось, иначе - *“Success”*.

```
Lab5. Task 7

| Input              | Output             |
|--------------------+--------------------|
| 10                 | 9 8 7 6 5 4 3 2 1 0 |
| 1 8 4 2 3 7 5 6 9 0 |                    |
```

```
 Merge sort:
Тест 1000 элементов:
Время работы: 0.0035690004006028175 секунд
Затрачено памяти: 15.9 Килобайт
Тест 10000 элементов:
Время работы: 0.07726950012147427 секунд
Затрачено памяти: 156.5 Килобайт
Тест 100000 элементов:
Время работы: 0.5647536003962159 секунд
Затрачено памяти: 1.5 Мегабайт
```

```
 Randomized Quick sort:
Тест 1000 элементов:
Время работы: 0.002564399503171444 секунд
Затрачено памяти: 9.9 Килобайт
Тест 10000 элементов:
Время работы: 0.03708400018513203 секунд
Затрачено памяти: 81.0 Килобайт
Тест 100000 элементов:
Время работы: 0.3198051992803812 секунд
Затрачено памяти: 785.0 Килобайт
```

```
Heap sort:
Тест 1000 элементов:
Время работы: 0.0044610993936657906 секунд
Затрачено памяти: 8.1 Килобайт
Тест 10000 элементов:
Время работы: 0.07376760058104992 секунд
Затрачено памяти: 78.4 Килобайт
Тест 100000 элементов:
Время работы: 0.9011797001585364 секунд
Затрачено памяти: 781.5 Килобайт
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0044610993936657906 секунд	8.1 Килобайт
Медиана диапазона значений входных данных из текста задачи	0.07376760058104992 секунд	78.4 Килобайт
Верхняя граница диапазона значений входных данных из текста задачи	0.9011797001585364 секунд	781.5 Килобайт

Вывод по задаче:

Сортировка работает медленнее, чем MergeSort и QuickSort, но тратит в среднем меньше памяти, так как проводит операции над пирамидой, возвращаясь к обычному состоянию массива для каждого элемента.

Дополнительные задачи

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.
- **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.

```
class Node:
    def __init__(self, idx):
        # self.parent = None
        self.children = []
        self.idx = idx

    def add_children_connection(self, node: 'Node'):
        self.children.append(node.idx)
        # node.parent = self.idx

    def get_children(self):
        return self.children.copy()

    def print(self):
        print(self.idx, "-", self.children)
```

```
class Tree:
    def __init__(self, tree_list):
        self.index_tree = tree_list
        self.fill_tree()
        self.root = self.node_tree[-1]

    def fill_tree(self):
         self.node_tree = [None] * (len(self.index_tree) + 1) # for root
node (last)
        fast_search_parent = {}

        for node_idx, parent_idx in enumerate(self.index_tree):
                                        fast_search_parent[parent_idx]    =
fast_search_parent.get(parent_idx, []) + [node_idx]


        stack = [-1]
        while stack:
            parent_idx = stack.pop()

            if self.node_tree[parent_idx] is None:
                self.node_tree[parent_idx] = Node(parent_idx)

            if parent_idx in fast_search_parent:
                for child_idx in fast_search_parent[parent_idx]:
                    stack.append(child_idx)
                    child_node = Node(child_idx)

self.node_tree[parent_idx].add_children_connection(child_node)
                    self.node_tree[child_idx] = child_node

    def get_node_by_index(self, idx) -> Node:
        if idx == -1:
            return self.root
        if 0 <= idx < len(self.node_tree):
            return self.node_tree[idx]
        return None

    def get_height(self):
        stack = [(self.root.idx, 0)]
        max_height = 0
```

```
        while stack:
            node_idx, height = stack.pop()
            node = self.get_node_by_index(node_idx)

            for child_node in node.get_children():
                stack.append((child_node, height + 1))

            max_height = max(max_height, height)

        return max_height


def solution(lst):
    tree = Tree(lst)
    return tree.get_height()
```

Реализован класс дерева, в котором метод нахождения высоты проходится в глубину по всем ветвям и находит наибольшую по длине.

```
Lab5. Task 2

| Input       | Output   |
|-------------+----------|
| 5           | 3        |
| 4 -1 4 1 1  |          |
```

```
......Тест 100 элементов:
Время работы: 0.00023579970002174377 секунд
Затрачено памяти: 17.8 Килобайт
Тест 10e3 элементов:
Время работы: 0.0047380998730659485 секунд
Затрачено памяти: 195.7 Килобайт
Тест 10e5 элементов:
Время работы: 0.25733030028641224 секунд
Затрачено памяти: 19.1 Мегабайт
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000235799700021743 секунд	17.8 Килобайт
Медиана диапазона	0.0047380998730659485	195.7 Килобайт

значений входных данных из текста задачи	секунд	
Верхняя граница диапазона значений входных данных из текста задачи	0.25733054597855 секунд	19.1 Мегабайт

Вывод по задаче:
Время работы алгоритма - O(n), но он расходует память на сохранение ссылок на объекты. Связные ноды дерева позволяют быстрее искать их детей и спускаться на уровень ниже.

Задача №4. 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(nlogn), в отличие от среднего времени работы QuickSort, равного O(nlogn). 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 ai и aj 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 ($1 \le n \le 10^5$), вторая содержит n целых чисел входного массива, разделенных пробелом ($0 \le a_i \le 10^9$, все ai - различны.)

- **Формат выходного файла (output.txt).** Первая строка ответа должна содержать целое число m - количество сделанных свопов. Число m должно удовлетворять условию $0 \le m \le 4n$. 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 $0 \le i \le n-1$ должны выполняться условия:
    1. если $2i+1 \le n-1$, то $a_i < a_{2i+1}$,
    2. если $2i+2 \le n-1$, 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 left(i): # 0 -> 1
    return 2 * (i+1) - 1

def right(i): # 0 -> 2
    return 2 * (i+1)

def MinHeapify(lst, i):
    swaps = []
    while True:
        _l = left(i)
        r = right(i)
        if _l > r:
            break
        if _l < len(lst) and lst[_l] < lst[i]:
            smallest = _l
        else:
            smallest = i

        if r < len(lst) and lst[r] < lst[smallest]:
            smallest = r

        if smallest != i:
```

```
            lst[i], lst[smallest] = lst[smallest], lst[i]
            swaps.append((i, smallest))
            i = smallest
        else:
            break

    return swaps


def BuildMinHeap(lst):
    heap_size = len(lst)
    swaps = []
    for i in range(heap_size//2, -1, -1):
        swaps += MinHeapify(lst, i)

    return swaps


def solution(lst):
    return BuildMinHeap(lst)
```

Пирамида реализована на основе списка значений с данной зависимостью элементов. Функция MinHeapify восстанавливает структуру пирамиды для выбранного элемента, а BuildMinHeap вызывает Heapify для элементов второй половины массива.

```
Lab5. Task 4

| Input     | Output   |
|-----------+----------|
| 5         | 3        |
| 5 4 3 2 1 | 1 4      |
|           | 0 1      |
|           | 1 3      |
```

```
Тест 100 элементов:
Время работы: 9.62996855378151e-05 секунд
Затрачено памяти: 80 Байт
Тест 1000 элементов:
Время работы: 0.0033111004158854485 секунд
Затрачено памяти: 188 Байт
Тест 10e5 элементов:
Время работы: 0.13157770037651062 секунд
Затрачено памяти: 188 Байт
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0000962996855378 секунд	80 Байт
Медиана диапазона значений входных данных из текста задачи	0.0033111004158854485 секунд	188 Байт
Верхняя граница диапазона значений входных данных из текста задачи	0.13157770037651062 секунд	188 Байт

Вывод по задаче:
Алгоритм работает быстро, за линейное время и при этом затрачивает минимум памяти. поскольку не создает копий массива, а лишь перемещает элементы.

Вывод

Такие структуры данных, как пирамида и дерево позволяют решать многие классы задач, связанные с распределением и сортировкой сравнимых типов данных. Они существенно уменьшают время выполнения программы и упрощают задачи для понимания. Также в случае пирамидальной сортировки присутствует заметная экономия памяти, хоть и меньшая эффективность по времени.