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 7

Выполнила:
Заботкина М.А.
К3139

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

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.

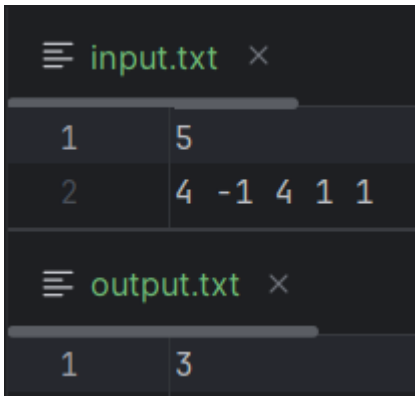
• Ограничение по памяти. 512 мб.

```
def find_height(parents, n):
    def height(node):
        if not children[node]:
            return 1
        return 1 + max(height(child) for child in children[node])

    children = [[] for _ in range(n)]
    root = -1
    for i in range(n):
        if parents[i] == -1:
            root = i
        else:
            children[parents[i]].append(i)

    return height(root)
```

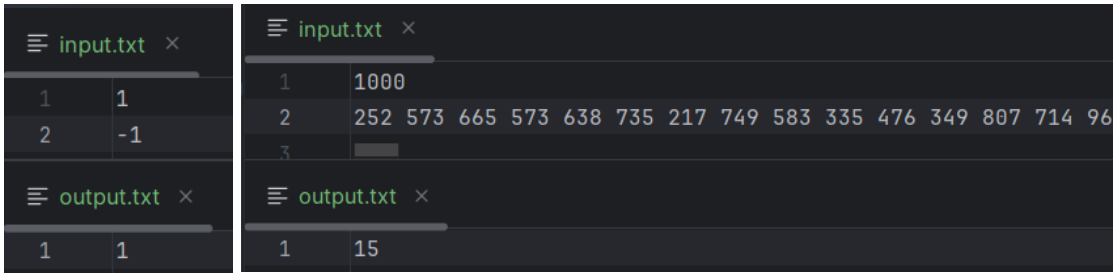
Функция find_height вычисляет высоту бинарного дерева, построенного по списку parents, который описывает родственные связи узлов. Она создает список children для хранения дочерних узлов каждого узла, а затем определяет корень дерева. С помощью рекурсивной функции height для каждого узла вычисляется максимальная глубина поддерева,

начиная с корня. Итоговая высота дерева – это максимальная глубина от корня до самого удаленного узла.

Результат работы кода на примерах из текста задачи:



Результат работы кода на максимальных и минимальных значениях:



	Время выполнения, с	Затраты памяти, Mb
Нижняя граница диапазона значений входных данных из текста задачи	0.0010874271392822266	0.0004425048828125
Пример из задачи	0.001055002212524414	0.0012054443359375
Верхняя граница диапазона значений входных данных из текста задачи	3.62422515858244256	1.0817184448242188

Вывод по задаче: в задаче был разработан алгоритм для вычисления высоты произвольного корневого дерева, представленного списком родительских узлов. Реализация показала высокую производительность даже на больших объемах входных данных.

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 последовательность 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. ($Ai \le Ai+1$ 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.

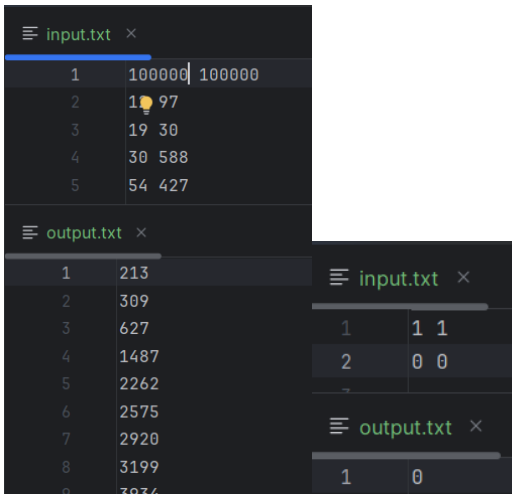
- *Ограничение по времени. 10 сек.*
- *Ограничение по памяти. 512 мб.*

```
def process_packets(S, packets):
    finish_time = []
    result = []

    for Ai, Pi in packets:
        finish_time = [t for t in finish_time if t > Ai]
        if len(finish_time) >= S:
            result.append(-1)
        else:
            if not finish_time:
                start_time = Ai
            else:
                start_time = finish_time[-1]

            finish_time.append(start_time + Pi)
            result.append(start_time)

    return result
```

Функция process_packets обрабатывает переданные пакеты, принимая во внимание лимит одновременных процессов S. Для каждого пакета (с временем прибытия Ai и длительностью Pi) она обновляет список завершенных задач, проверяет, можно ли начать новую задачу. Если лимит одновременных задач уже достигнут, добавляется -1. В противном случае, для каждого пакета вычисляется время начала задачи, основываясь на времени завершения предыдущей задачи, и добавляется в список. Функция возвращает список начальных времен для каждого пакета или -1, если невозможно начать задачу.

Результат работы кода на примерах из текста задачи:

Результат работы кода на максимальных и минимальных значениях:



	Время выполнения, с	Затраты памяти, Mb
Нижняя граница диапазона значений входных данных из текста задачи	0.0009162425994873047	0.000274658203125
Пример из задачи	0.0010039806365966797	0.0003662109375
Верхняя граница диапазона значений входных данных из текста задачи	12.4377207293747315	1.51470947265625

Вывод по задаче: создана модель обработки сетевых пакетов с учетом фиксированного размера буфера. Алгоритм успешно обрабатывает пакеты в соответствии с условиями задачи, демонстрируя стабильное выполнение в пределах заданных ограничений.

Дополнительные задачи

Задача №1. Куча ли?

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 $ai \le a2i$, 2. если $2i + 1 \le n$, то $ai \le a2i+1$. 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.*

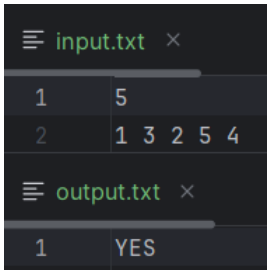
• *Ограничение по времени. 2 сек.*

• *Ограничение по памяти. 256 мб.*

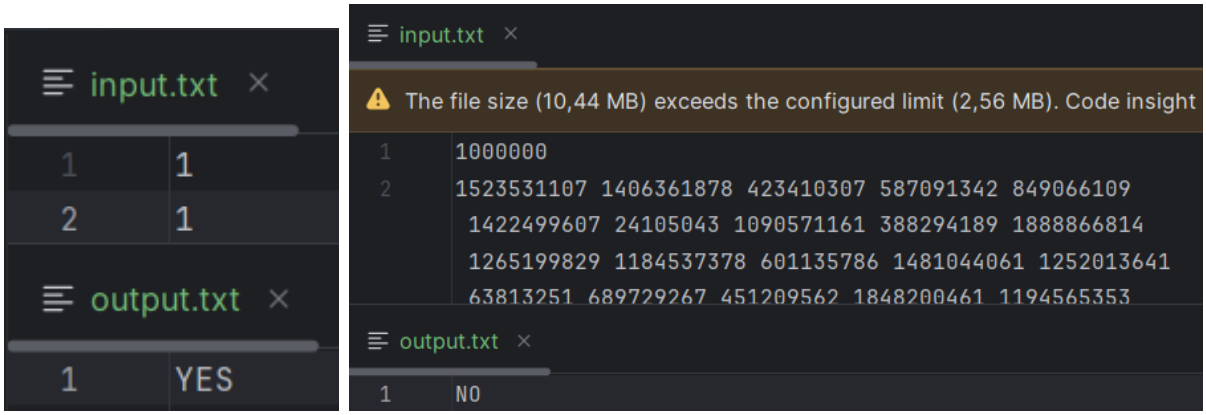
```
def check_heap(arr):
    n = len(arr)
    for i in range(1, n + 1):
        left = 2 * i
        right = 2 * i + 1
        if left <= n and arr[i - 1] > arr[left - 1]:
            return "NO"
        if right <= n and arr[i - 1] > arr[right - 1]:
            return "NO"
    return "YES"
```

Функция check_heap проверяет, является ли массив arr кучи, упорядоченной по правилам кучи. Она проходит по массиву и для каждого узла проверяет, что он не больше своих детей (левого и правого). Если находит любое нарушение этого условия, то возвращает "NO". Если во всех случаях узлы соблюдают свойство кучи, возвращает "YES".

Результат работы кода на примерах:



Результат работы кода на максимальных и минимальных значениях:



	Время выполнения, с	Затраты памяти, Mb
Нижняя граница диапазона значений входных данных из текста задачи	0.0009751319885253906	7.62939453125e-05
Пример из задачи	0.000942230224609375	7.62939453125e-05
Верхняя граница диапазона значений входных данных из текста задачи	0.0010001659393310547	0.000179290771484375

Вывод по задаче: реализован алгоритм проверки массива на соответствие свойствам неубывающей пирамиды. Решение корректно и эффективно проверяет массивы любого размера.

Задача №6. Очередь с приоритетами

Реализуйте очередь с приоритетами. Ваша очередь должна поддерживать следующие операции: добавить элемент, извлечь минимальный элемент, уменьшить элемент, добавленный во время одной из операций.

• Формат входного файла (input.txt). В первой строке входного файла содержится число n ($1 \leq n \leq 10^6$) - число операций с очередью. 10 Следующие n строк содержат описание операций с очередью, по одному описанию в строке. Операции могут быть следующими: – A x – требуется добавить элемент x в очередь. – X – требуется удалить из очереди минимальный элемент и вывести его в выходной файл. Если очередь пуста, в выходной файл требуется вывести звездочку «». – D x y – требуется заменить*

значение элемента, добавленного в очередь операцией A в строке входного файла номер x + 1, на y. Гарантируется, что в строке x + 1 действительно находится операция A, что этот элемент не был ранее удален операцией X, и что y меньше, чем предыдущее значение этого элемента. В очередь помещаются и извлекаются только целые числа, не превышающие по модулю 10^9.

• *Формат выходного файла (output.txt). Выведите последовательно результат выполнения всех операций X, по одному в каждой строке выходного файла. Если перед очередной операцией X очередь пуста, выведите вместо числа звездочку «*».*

• *Ограничение по времени. 2 сек.*

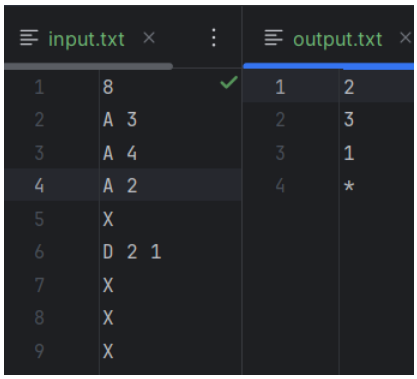
• *Ограничение по памяти. 256 мб.*

```
def priority_queue(operations):
    def heappush(heap, item):
        heap.append(item)
        heap.sort()

    def heappop(heap):
        return heap.pop(0) if heap else None

    queue = []
    results = []
    for i, operation in enumerate(operations):
        parts = operation.strip().split()
        if not parts:
            continue
        op_type = parts[0]
        if op_type == 'A':
            x = int(parts[1])
            heappush(queue, (x, i))
        elif op_type == 'X':
            min_elem = heappop(queue)
            if min_elem:
                results.append(str(min_elem[0]))
            else:
                results.append('*')
        elif op_type == 'D':
            idx = int(parts[1]) - 1
            new_val = int(parts[2])
            if idx < len(queue) and queue[idx][0] > new_val:
                queue[idx] = (new_val, i)

    return results
```

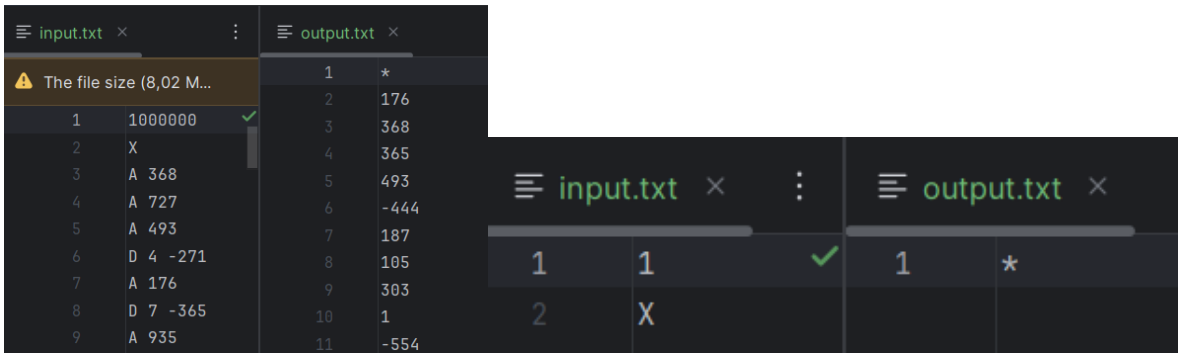
Функция priority_queue имитирует работу приоритизованной очереди, обрабатывая три типа операций: добавление элемента ('A'), извлечение минимального элемента ('X') и обновление значения элемента ('D'). Внутренние функции heappush и heappop используются для управления приоритетной очередью, где элементы хранятся в упорядоченном виде. При

операции 'A' элемент добавляется в очередь с сохранением сортировки по приоритету. При операции 'X' извлекается минимальный элемент и добавляется его значение к результатам, а в случае пустой очереди возвращается ''. При операции 'D' обновляется значение элемента по указанному индексу, если новое значение меньше текущего. В итоге, функция возвращает список результатов операции 'X' или '' для случаев, когда очередь пуста.

Результат работы кода на примерах из текста задачи:



Результат работы кода на максимальных и минимальных значениях:



	Время выполнения, с	Затраты памяти, Mb
Нижняя граница диапазона значений входных данных из текста задачи	0.0009772777557373047	0.00061798095703125
Пример из задачи	0.0009608268737792969	0.0007610321044921875
Верхняя граница диапазона значений входных данных из текста задачи	3.667376518249512	19.2792329788208

Вывод по задаче: реализована приоритетная очередь с поддержкой операций добавления, извлечения минимального элемента и изменения значения. Решение обеспечивает корректную обработку операций в соответствии с условиями задачи.

Вывод

Лабораторная работа успешно продемонстрировала применение различных структур данных, включая деревья, кучи и очереди с приоритетами, для решения задач разной сложности. Алгоритмы показали корректность и производительность в заданных пределах, что подтверждается результатами тестирования на контрольных примерах и граничных значениях.