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 12

Выполнила:
Мкртчян.К.Г.
К3141

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Оглавление

Задачи по варианту

Задача №3. Обработка сетевых пакетов. [N баллов]

Текст задачи:

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.

Решение:

```
def net_packet_processing(buffer_size: int, packages_count: int, packages:
tp.List[tp.Tuple[int, int]]) -> tp.List[int]:
    if packages_count == 0:
        return []
    deque = []
    result = []
    head = 0
    buffer_time = packages[0][0]
    for p in packages:
        start_time, duration = p
        head = max([head] + [index for index in range(len(deque)) if
deque[index] <= start_time])
        if len(deque[head + 1:]) < buffer_size:
            result += [max(buffer_time, start_time)]
            buffer_time += duration
            deque.append(buffer_time)
        else:
            result += [-1]
    return result
```

Объяснение решения:

Функция net_packet_processing предназначена для обработки сетевых пакетов, которая управляет временной очередью пакетов, основываясь на заданном размере буфера и временных параметрах.

Параметры:

- buffer_size (int): максимальное количество пакетов, которые могут храниться в буфере одновременно.
- packages_count (int): общее количество пакетов, которые нужно обработать.
- packages (List[Tuple[int, int]]): список пакетов, где каждый пакет представлен кортежем из двух целых чисел:
- start_time (int): время начала обработки пакета.
- duration (int): продолжительность обработки пакета.

Логика работы функции:

Проверка на пустой список пакетов: Если packages_count равен 0, функция возвращает пустой список, так как обрабатывать нечего.

Инициализация переменных:

- deque — список, представляющий очередь времени завершения обработки пакетов.
- result — список для хранения времени начала обработки текущего пакета или -1, если пакет не может быть обработан.
- head — индекс для отслеживания очередных пакетов, указывающий на последний обработанный пакет.

-buffer_time — время, до которого буфер занят (значение инициализируется временем начала первого пакета).

Обработка каждого пакета: Функция перебирает все пакеты, выполняя следующие шаги:

- Для каждого пакета извлекается start_time и duration.

-head обновляется до индекса, где завершенные пакеты закончились (то есть их моменты завершения меньше или равны start_time).

- Если количество пакетов в буфере после завершения обработанных пакетов (определяемое как len(deque[head+1:])) меньше размера буфера:

-Записываем в result максимальное значение между buffer_time и start_time, чтобы проверить, когда можно начать обработку текущего пакета.

-Обновляем buffer_time, добавляя к нему продолжительность текущего пакета.

-Добавляем в deque новое значение buffer_time, представляющее время завершения текущего пакета.

-Если буфер заполнен, добавляем -1 в result, указывая на то, что пакет не может быть обработан.

Возвращаемое значение:

Функция возвращает список result, который содержит время, когда начинается обработка каждого пакета, или -1, если пакет не может быть обработан.

Время выполнения программы, затраты памяти и результат:

	данные	время, сек.	память, МБ	результат
Значения из примера	3 6 [(0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2)]	0.0	32.24	0 2 4 6 8 -1
Значения из примера	2 3 [(0, 1), (3, 1), (10, 1)]	0.0	32.26	0 3 10
Значения из примера	1 2 [(0, 1), (2, 1)]	0.0	32.26	0 2
Значения из примера	1 1 [(0, 0)]	0.0	32.26	0
Максимальные значения	1000 100000 [(100005, 28), (100006, 769), (100007, 699), (100008, 481), (100009, 368)]	2.67	45.91	-1 -1 -1 -1 -1

• Ограничение по времени. 10 сек. • Ограничение по памяти. 512 мб.

Вывод по задаче: мы вспомнили материал предыдущей лекции и использовали дек для решения задачи по обработке сетевых пакетов.

Задача №4. Построение пирамиды. [N баллов]

Текст задачи:

В этой задаче вы преобразуете массив целых чисел в пирамиду. Это важнейший шаг алгоритма сортировки под названием HeapSort. 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.

Решение:

```
class MinHeap:
    def __init__(self, array_len: int, array: tp.List[int]) -> None:
        self.heap = array
        self.high = array_len
        self.swaps = []

    def heap_sort(self) -> (int, tp.List[tp.Tuple[int]]):
        for i in range(self.high // 2 - 1, -1, -1):
            self.swap(i)
        return len(self.swaps), self.swaps

    def swap(self, index: int) -> None:
        smallest = index
        left = 2 * index + 1
        right = 2 * index + 2

        if left < self.high and self.heap[left] < self.heap[smallest]:
            smallest = left

        if right < self.high and self.heap[right] < self.heap[smallest]:
            smallest = right

        if smallest != index:
            self.swaps += [(index, smallest)]
            self.heap[index], self.heap[smallest] = self.heap[smallest],
self.heap[index]
            self.swap(smallest)
```

Объяснение решения:

Класс MinHeap реализует структуру данных неубывающая куча/пирамида и предоставляет функциональность для выполнения сортировки с

использованием метода кучей (heap sort). Вот краткое описание работы класса:

1. Инициализация (Конструктор)

- Метод __init__ принимает два параметра:
array_len: длина входного массива.
array: список целых чисел.
- Инициализирует внутреннее представление кучи (self.heap) и фиксирует её длину (self.high). Также инициализирует пустой список для хранения операций замены (self.swaps).

2. Сортировка кучи (heap_sort)

- Метод heap_sort осуществляет основное действие сортировки:
- Последовательно вызывает метод swap для создания структуры - неубывающей кучи с помощью элементов массива. Начинает с индекса, равного array_len // 2 - 1, и проходит до корня кучи (индекс 0).
- Возвращает количество произведённых операций замены и сами операции в виде списка кортежей.

3. Метод замены (swap)

- Метод swap выполняет перераспределение элементов для поддержания свойств неубывающей кучи:
- Находит значения левого и правого дочерних элементов.
- Сравнивает значения текущего элемента с его дочерними элементами и находит наименьший.
- Если наименьший элемент не совпадает с текущим индексом, производит замену (swap) и рекурсивно вызывает сам себя для дочернего элемента, чтобы обеспечить правильность кучи.

Время выполнения, затраты памяти и результат:

	данные	время, сек.	память, МБ	результат
Минимальные значения	5 [1, 2, 3, 4, 5]	0.0	32.26	0 []
Значения из примера	5 [1, 2, 3, 5, 4]	0.0	32.29	3 [(1, 4), (0, 1), (1, 3)]
Максимальные значения	100005 [2195, 6072, 10235, 44175]	0.04	45.03	74348 [(50001, 100003), (49999, 100000), (49998, 99998), (49997, 99996)]

• Ограничение по времени. 3 сек. • Ограничение по памяти. 512 мб.

Вывод по задаче: мы научились считать количество произведённых операций замены во время алгоритма построения пирамиды.

Дополнительные задачи

Задача №1. Куча ли? [N баллов]

Текст задачи:

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 $1 \leq i \leq n$ выполняются условия: 1. если $2i \leq n$, то $ai \leq a2i$, 2. если $2i + 1 \leq n$, то $ai \leq a2i+1$. Дан массив целых чисел. Определите, является ли он неубывающей пирамидой.

Решение:

```
def is_heap(array_len: int, array: tp.List[int]) -> str:
    for index in range(array_len // 2):
        left_index = 2 * index + 1
        right_index = 2 * index + 2
        if array[index] > array[left_index]:
            return "NO"
        if right_index < len(array) and array[index] > array[right_index]:
            return "NO"
    return "YES"
```

Объяснение решения:

Функция is_heap предназначена для проверки, является ли заданный массив неубывающей кучей (min-heap). В мин-куче каждый родительский узел должен быть меньше или равен своим дочерним узлам. Вот подробное описание работы функции:

Параметры:

- array_len (int): длина массива, который нужно проверить.
- array (List[int]): список целых чисел, представляющий собой структуру данных.

Возвращаемое значение:

Функция возвращает строку:

"YES" — если массив является неубывающей кучей,

"NO" — если не является.

Логика работы функции:

- Итерация по родительским узлам:
    - Функция проходит по всем родительским узлам в массиве. Поскольку в неубывающей куче каждый родительский узел находится в индексе i, левые и правые дочерние узлы находятся на индексах $2i + 1$ и $2i + 2$ соответственно.

Проверка на условие неубывающей кучи:

- Для каждого родительского узла (по индексу index):

- Вычисляется индекс левого дочернего узла (left_index) и правого дочернего узла (right_index).
- Если родительский узел больше левого дочернего узла, функция возвращает "NO".
- Если правый дочерний узел существует (проверка с помощью right_index < len(array)) и родительский узел больше правого дочернего узла, также возвращает "NO".

Возврат результата:

Если все родительские узлы удовлетворяют условию для неубывающей кучи, функция возвращает "YES".

Время выполнения, затраты памяти и результат:

	данные	время, сек.	память, МБ	результат
Значения из примера	5 [1, 0, 1, 2, 0]	0.0	32.07	NO
Значения из примера	5 [1, 3, 2, 5, 4]	0.0	32.1	YES
Максимальные значения	1000000 [1638320326, 1614147333, -380180974]	0.0	70.77	NO

• Ограничение по времени. 2 сек. • Ограничение по памяти. 256 мб

Вывод по задаче: Функция is_heap эффективно проверяет, является ли заданный массив неубывающей кучей, используя простую итерацию и сравнение значений. Это предоставляет полезный инструмент для работы с кучами в различных алгоритмах, таких как сортировка или приоритетные очереди.

Задача №7. Снова сортировка. [N баллов]

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

Решение:

```
def heap_sort_max(high: int, array: tp.List[int]) -> tp.List[int]:
    heap = array.copy()
    build_heap(high, heap)
    for i in range(high - 1, -1, -1):
        heap[i], heap[0] = heap[0], heap[i]
        swap(heap, i, 0)
    return heap


def swap(heap: tp.List[int], high: int, index: int) -> None:
    left = 2 * index + 1
    right = 2 * index + 2

    if left < high and heap[left] < heap[index]:
        largest = left
    else:
        largest = index

    if right < high and heap[right] < heap[largest]:
        largest = right

    if largest != index:
        heap[index], heap[largest] = heap[largest], heap[index]
        swap(heap, high, largest)


def build_heap(high: int, heap: tp.List[int]) -> None:
    for i in range((len(heap) - 1) // 2, -1, -1):
        swap(heap, high, i)
```

Объяснение решения:

Краткое объяснение работы сортировки (Heap Sort)

Функция heap_sort_max выполняет сортировку массива чисел в порядке убывания, используя алгоритм сортировки с использованием кучи (heap sort). Рассмотрим, как она работает шаг за шагом:

- Копирование массива: heap = array.copy()

  Создается копия входного массива, чтобы не изменять исходные данные.

- Построение кучи: build_heap(high, heap)

  Функция build_heap преобразует массив в пирамиду. Для этого она вызывает функцию swap, которая помогает поддерживать свойства кучи.

Сортировка:

- Цикл начинает с конца массива и проходит к началу.

- На каждой итерации происходит обмен первого элемента (максимального элемента в куче) с текущим элементом на позиции i.
- Затем восстанавливается свойство кучи внутри массива (в функции swap) для элементов, которые не отсортированы. Это происходит с помощью рекурсивной перестановки элементов, чтобы убедиться, что оставшаяся часть массива снова соответствует условиям кучи.

Функция swap:

- Эта функция отвечает за перестановку элементов и поддержание свойства кучи. Она сравнивает текущий элемент с его дочерними элементами и при необходимости производит обмен для установления корректного порядка.
- При использовании рекурсии она проверяет дочерние элементы, что поддерживает кучу до тех пор, пока все элементы не будут правильно расположены.

Функция build_heap:

- Строит кучу, начиная с последнего родительского узла и перемещаясь к корню. Это гарантирует, что все элементы в массиве будут соответствовать свойству кучи.

Время выполнения, затраты памяти и результат:

	данные	время, сек.	память, МБ	результат
Минимальные значения	1000	0.001349099911749363	32.35	[999964251, 995790628, 994693236, 994341023]
	[-228140335, -258391466, -750710185, 172164125]			
Средние значения	10000	0.01883130008354783	33.01	[999720431, 999124673, 999085893, 999070278]
	[-201366236, -872046159, -288878434, -685310700]			
Максимальные значения	100000	0.27189780003391206	37.71	[999983651, 999975793, 999964347, 999948909]
	[-936273498, -446287534, 166006819, 221888846]			
Повторяющиеся значения	100000	0.11590969981625676	36.74	[1000000000, 1000000000, 1000000000, 1000000000]
	[-839694192, 1000000000, -839694192, 1000000000]			

Вывод по задаче: таким образом, алгоритм сортировки кучи (heap sort) работает следующим образом: преобразует массив в пирамиду. Проходит от конца массива к началу, извлекая максимальные элементы и помещая их в конец массива. Поддерживает свойство кучи на каждом шаге, повторяя это до тех пор, пока весь массив не будет отсортирован. Эта сортировка имеет временную сложность O(n log n) в худшем, среднем и лучшем случае, и использует O(1) дополнительной памяти, что делает её эффективной для различных задач сортировки.

Вывод по лабораторной работе

В ходе лабораторной работы мы научились реализовывать алгоритм пирамидальной сортировки и его отдельные составляющие, а также вспомнили структуру данных дек.