МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Алгоритмы и Структуры Данных»
Тема: Очереди с приоритетами. Параллельная обработка.

Студент гр. 1303 _____ Кузнецов Н.А.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург
2022

Цель работы.

Изучение параллельной обработки с применением очереди с приоритетом.

Задание.

На вход программе подается число процессоров n и последовательность чисел t0 , . . . , tm−1 , где ti — время, необходимое на обработку i-й задачи.

Требуется для каждой задачи определить, какой процессор и в какое время начнёт её обрабатывать, предполагая, что каждая задача поступает на обработку первому освободившемуся процессору.

Примечание #1: в работе необходимо использовать очередь с приоритетом (т.е. min или max-кучу)

Примечание #2: в работе запрещено использовать библиотечные реализации алгоритмов и структур.

Формат входа

Первая строка входа содержит числа n и m. Вторая содержит числа t0 , . . . , tm−1 , где ti — время, необходимое на обработку i-й задачи. Считаем, что и процессоры, и задачи нумеруются с нуля.

Формат выхода

Выход должен содержать ровно m строк: i-я (считая с нуля) строка должна содержать номер процессора, который получит i-ю задачу на обработку, и время, когда это произойдёт.

Выполнение работы.

Class Processor — класс, который хранит индекс текущего процессора, время начала обработки информации и текущее время. В класс перегружен

метод *__lt__* (<), определяющий меньший элемент либо по времени окончания обработки задачи, либо, при равенстве времени, по индексу.

Class Heap — класс двоичной мини-кучи. В конструкторе задаётся массив, в котором будут храниться элементы кучи. Принципы хранения и обращения к элементам кучи реализованы в методах класса:

- *get_parent(index)* — возвращает индекс родителя элемента;
- *get_left_child(index)* — возвращает индекс первого (левого) потомка элемента;
- *get_right_child(index)* — возвращает индекс второго (правого) потомка элемента;
- *insert(self, element)* — вставка нового элемента в данною структуру;
- *sift_up(self, index)* — просеивание вверх;
- *extract_min(self)* — возвращает и извлекает корневой элемент дерева;
- *sift_down(self, index)* — просеивание вниз.

В функции *process()* создаётся экземпляр класса *Heap*. Далее с помощью цикла экземпляры класса Processor добавляются в очередь с приоритетом. В список ans на каждом шаге заносятся индекс процессора, который получил задачу на обработку и время, за сколько это произойдёт. Данная функция возвращает список ans.

В функции *main()* считываются данные, вызывается функция *process()* и вызывается печать, полученных данных.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 — результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	2 5 1 2 3 4 5	0 0 1 0	Верно

		0 1 1 2 0 4	
2.	0 0		Верно
3.	10 2		Верно
4.	3 7 11 9 8 7 6 3 1	0 0 1 0 2 0 2 8 1 9 0 11 0 14	Верно
5.	5 5 1 2 3 4 5	0 0 1 0 2 0 3 0 4 0	Верно
6.	1 3 6 21 8	0 0 0 6 0 27	Верно

Вывод.

В ходе данной лабораторной работы было изучена параллельная обработка с применением очереди с приоритетом.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from modules.processimport process


def main():
    proc_kol, time_kol = map(int, input().split())
    times = list(map(int, input().split()))
    answer = process(proc_kol, time_kol, times)

    for i in answer:
        print(*i)


if __name__ == '__main__':
    main()
```

Название файла: tests.py

```
ffrom modules.process import process


def test1():
    assert process(2, 5, [1, 2, 3, 4, 5]) == [[0, 0], [1, 0], [0, 1],
[1, 2], [0, 4]]


def test2():
    assert process(0, 0, []) == []


def test3():
    assert process(10, 2, []) == []


def test4():
    assert process(3, 7, [11, 9, 8, 7, 6, 3, 1]) == [[0, 0], [1, 0],
[2, 0], [2, 8], [1, 9], [0, 11], [0, 14]]


def test5():
    assert process(5, 5, [1, 2, 3, 4, 5]) == [[0, 0], [1, 0], [2, 0],
[3, 0], [4, 0]]


def test6():
    assert process(1, 3, [6, 21, 8]) == [[0, 0], [0, 6], [0, 27]]
```

Название файла: heap.py

```
class Heap:
    def __init__(self):
        self.max_size = 10**5
        self.heap = [None] * self.max_size
        self.size = 0

    @staticmethod
    def get_parent(index):
        return (index - 1) // 2

    @staticmethod
    def get_left_child(index):
        return 2 * index + 1

    @staticmethod
    def get_right_child(index):
        return 2 * index + 2

    def insert(self, element):
        if self.size == self.max_size:
            return -1
        self.heap[self.size] = element
        self.sift_up(self.size)
        self.size += 1

    def extract_min(self):
        min_element = self.heap[0]
         self.heap[0], self.heap[self.size - 1] = self.heap[self.size -
1], None
        self.size -= 1
        self.sift_down(0)
        return min_element

    def sift_up(self, index):
        parent = self.get_parent(index)
        while index > 0 and self.heap[parent] > self.heap[index]:
                self.heap[parent], self.heap[index] = self.heap[index],
self.heap[parent]
            index = parent
            parent = self.get_parent(index)

    def sift_down(self, index):
        left = self.get_left_child(index)
        right = self.get_right_child(index)
        if left >= self.size and right >= self.size:
            return
        if right >= self.size:
                min_index = left if self.heap[left] < self.heap[index]
else index
        else:
                min_index = left if self.heap[left] < self.heap[right]
else right
                    min_index = min_index if self.heap[min_index] <
self.heap[index] else index
        if min_index != index:
```

```
            self.heap[min_index], self.heap[index] = self.heap[index],
self.heap[min_index]
            self.sift_down(min_index)
```

Название файла: processor.py

```
class Processor:
    def __init__(self, index, start, time):
        self.index = index
        self.start = start
        self.time = time

    def __lt__(self, other):
        if self.time + self.start == other.time + other.start:
            return self.index < other.index
        return self.time + self.start < other.time + other.start

    def __str__(self):
        return f'{self.index} {self.start}'
```

Название файла: process.py

```
from src.modules.heap import Heap
from src.modules.processor import Processor


def process(proc_kol, time_kol, times):
    answer = list()

    heap = Heap()
    for i in range(min(nproc_kol, len(times))):
        heap.insert(Processor(i, 0, times[i]))
        answer.append([i, 0])

    for i in range(proc_kol, time_kol):
        cur_proc = heap.extract_min()
        cur_time = cur_proc.start + cur_proc.time
        heap.insert(Processor(cur_proc.index, curr_time, times[i]))
        answer.append([current_proc.index, current_time])

    return answer
```