

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 9

Выполнила:

Левчук С.А.

К3141

Проверил:

Афанасьев А. В.

Санкт-Петербург

2024 г.

## Содержание отчета

<b>Содержание отчета</b>	<b>2</b>
<b>Задачи по варианту</b>	<b>3</b>
Задание №2. Высота дерева	3
Задание №5. Планировщик заданий	6
<b>Дополнительные задачи</b>	<b>10</b>
Задание №1. Куча ли?	10
Задание №3. Обработка сетевых пакетов	13
<b>Вывод</b>	<b>18</b>

## Задачи по варианту

### Задание №2. Высота дерева

#### 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 4 -1 4 1 1	3

Решение:

```
from utils import write, read

def tree_height(n, parents):
    children = [[] for _ in range(n)]
    root = -1
    for child, parent in enumerate(parents):
        if parent != -1:
            children[parent].append(child)
        else:
            root = child

    def calc_height(node):
        if not children[node]:
            return 1
        max_height = 0
        for child in children[node]:
            max_height = max(max_height, calc_height(child))
        return max_height + 1

    return calc_height(root)
```

```
def main():
    write(end='')
    (n,), array = read(type_convert=int)
    write(tree_height(n, array), to_end=True)

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

Функция `tree_height` вычисляет высоту дерева, представленного массивом родителей (`parents`), где каждый элемент массива указывает на родителя соответствующего узла.

Алгоритм:

- 1) Создается список `children`, который будет хранить детей для каждого узла. Изначально он инициализируется пустыми списками/
- 2) Инициализируем переменную `root` значением `-1`. Она будет использоваться для хранения индекса корня дерева.
- 3) Проходим циклом по всем узлам и их родителям. Если родитель не равен `-1`, то текущий узел добавляется к списку детей этого родителя. В противном случае, если родитель равен `-1`, значит текущий узел является корнем дерева, и его индекс сохраняется в переменную `root`.
- 4) Внутри функции `calc_height` рекурсивно рассчитывается высота поддерева, начиная от текущего узла. Если у узла нет детей, возвращается `1` (высота листа). Иначе перебираются все дети узла, и находится максимальная высота среди всех поддеревьев плюс `1` (учитывается сам узел).
- 5) Возвращается результат вызова `calc_height` для корня дерева.

Результат работы программы:

Входные данные:

```
5
-1 0 4 0 3
```

Выходные данные:

```
4
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab5.task2.src.tree_height import main, tree_height

class TestTreeHeight(unittest.TestCase):
```

```

def test_should_check_example1_data(self):
    # given
    n = 5
    data = [4, -1, 4, 1, 1]
    expected_result = 3

    # when
    result = tree_height(n, data)

    # then
    self.assertEqual(result, expected_result)

def test_should_check_example2_data(self):
    # given
    n = 5
    data = [-1, 0, 4, 0, 3]
    expected_result = 4

    # when
    result = tree_height(n, data)

    # then
    self.assertEqual(result, expected_result)

def test_should_check_time_data(self):
    # given
    expected_time = 3

    # when
    time = time_data(main)

    # then
    self.assertTrue(time < expected_time)

def test_should_check_memory_data(self):
    # given
    expected_memory = 512

    # when
    current, peak = memory_data(main)

    # then
    self.assertTrue(current < expected_memory)
    self.assertTrue(peak < expected_memory)

if __name__ == "__main__":
    unittest.main()

```

Вывод по задаче:

В ходе решения данной задачи мы реализовали структуру данных “дерево”, сохранили его и вычислили высоту.

## Задание №5. Планировщик заданий

### 5 задача. Планировщик заданий

В этой задаче вы создадите программу, которая параллельно обрабатывает список заданий. Во всех операционных системах, таких как Linux, MacOS или Windows, есть специальные программы, называемые планировщиками, которые делают именно это с программами на вашем компьютере.

У вас есть программа, которая распараллеливается и использует  $n$  независимых потоков для обработки заданного списка  $m$  заданий. Потоки берут задания в том порядке, в котором они указаны во входных данных. Если есть свободный поток, он немедленно берет следующее задание из списка. Если поток начал обработку задания, он не прерывается и не останавливается, пока не завершит

обработку задания. Если несколько потоков одновременно пытаются взять задания из списка, поток с меньшим индексом берет задание. Для каждого задания вы точно знаете, сколько времени потребуется любому потоку, чтобы обработать это задание, и это время одинаково для всех потоков.

Вам необходимо определить для каждого задания, какой поток будет его обрабатывать и когда он начнет обработку.

- **Формат ввода или входного файла (input.txt).** Первая строка содержит целые числа  $n$  и  $m$  ( $1 \leq n \leq 10^5$ ,  $1 \leq m \leq 10^5$ ). Вторая строка содержит  $m$  целых чисел  $t_i$  - время в секундах, которое требуется для выполнения  $i$ -ой задания любым потоком ( $0 \leq t_i \leq 10^9$ ). Все эти значения даны в том порядке, в котором они подаются на выполнение. **Индексы потоков начинаются с 0.**
- **Формат выходного файла (output.txt).** Выведите в точности  $m$  строк, причем  $i$ -ая строка (начиная с 0) должна содержать два целочисленных значения: индекс потока, который выполняет  $i$ -ое задание, и время в секундах, когда этот поток начал выполнять задание.
- Ограничение по времени. 6 сек.
- Ограничение по памяти. 512 мб.
- Пример 1:

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

Решение:

```
from utils import read, write
import heapq

def scheduler(n, tasks):
    threads = [(0, i) for i in range(n)]
    results = []
    for task in tasks:
        end_time, thread_index = heapq.heappop(threads)
        start_time = end_time
        results.append((thread_index, start_time))
        new_end_time = start_time + task
        heapq.heappush(threads, (new_end_time, thread_index))
    return results
```

```
def main():
    write(end='')
    data = [list(line) for line in read(type_convert=int)]
    n, m = data[0]
    tasks = data[1]
    result = scheduler(n, tasks)
    for line in result:
        write(*line, to_end=True)

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

Функция scheduler реализует планировщик для распределения задач между потоками.

Алгоритм:

1. Создаем список threads, содержащий начальные времена завершения задач для каждого потока. Изначально они равны нулю, так как ни одна задача еще не выполнялась. Каждый элемент списка представляет собой кортеж (end\_time, thread\_index), где end\_time — время окончания задачи, а thread\_index — номер потока.
2. Инициализируем пустой список results, куда будем добавлять результаты планирования.
3. Для каждой задачи в списке tasks:
  - 1) Извлекаем первый элемент из кучи threads с помощью heapq.heappop. Это будет поток с минимальным временем завершения текущей задачи.
  - 2) Сохраняем время начала задачи как текущее время завершения (start\_time = end\_time).
  - 3) Добавляем в results кортеж (thread\_index, start\_time), указывающий, какой поток выполняет задачу и когда она начинается.
  - 4) Рассчитываем новое время завершения задачи для данного потока: new\_end\_time = start\_time + task.
  - 5) Помещаем обновленный поток обратно в кучу с помощью heapq.heappush.
4. По завершении цикла возвращаем список результатов.

Результат работы программы:

Входные данные:

2 5

```
1 2 3 4 5
```

Выходные данные:

```
0 0
1 0
0 1
1 2
0 4
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab5.task5.src.scheduler import main, scheduler

class TestScheduler(unittest.TestCase):

    def test_should_check_example1_data(self):
        # given
        n, m = 2, 5
        data = [1, 2, 3, 4, 5]
        expected_result = [(0, 0), (1, 0), (0, 1), (1, 2), (0, 4)]

        # when
        result = scheduler(n, data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_example2_data(self):
        # given
        n, m = 4, 20
        data = [1] * 20
        expected_result = [(0, 0), (1, 0), (2, 0), (3, 0), (0, 1), (1, 1), (2,
1), (3, 1), (0, 2), (1, 2), (2, 2), (3, 2), (0, 3), (1, 3), (2, 3), (3, 3),
(0, 4), (1, 4), (2, 4), (3, 4)]

        # when
        result = scheduler(n, data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_empty_data(self):
        # given
        n = 4
        data = []
        expected_result = []

        # when
        result = scheduler(n, data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_time_data(self):
```



```

    # given
    expected_time = 6

    # when
    time = time_data(main)

    # then
    self.assertTrue(time < expected_time)

def test_should_check_one_task_data(self):
    # given
    n = 2
    data = [5]
    expected_result = [(0, 0)]

    # when
    result = scheduler(n, data)

    # then
    self.assertEqual(result, expected_result)

def test_should_check_memory_data(self):
    # given
    expected_memory = 512

    # when
    current, peak = memory_data(main)

    # then
    self.assertTrue(current < expected_memory)
    self.assertTrue(peak < expected_memory)

if __name__ == "__main__":
    unittest.main()

```

Вывод по задаче:

В ходе решения данной задачи мы создали программу, которая параллельно обрабатывает список заданий.

## Дополнительные задачи

### Задание №1. Куча ли?

#### 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

#### Решение:

```
from utils import read, write

def is_heap(array_len, array):
    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'

def main():
    write(end='')
    (n, ), array = read(type_convert=int)
    write(is_heap(n, array), to_end=True)

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

Функция `is_heap` проверяет, удовлетворяет ли массив свойствам бинарной кучи:

- 1) Каждый родительский элемент должен быть меньше или равен своим дочерним элементам.
- 2) Для этого мы проходим по всем индексам массива до половины его длины (так как индексы после этой точки будут листьями дерева).
- 3) Проверяем условия для левого и правого потомков текущего элемента.
- 4) Если условие нарушается, возвращаем строку "NO". В противном случае, если все элементы соответствуют требованиям, возвращаем "YES".

Результат работы программы:

Входные данные:

```
5
1 3 2 5 4
```

Выходные данные:

```
YES
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab5.task1.src.is_heap import main, is_heap

class TestIsHeap(unittest.TestCase):

    def test_should_check_incorrect_data(self):
        # given
        n = 5
        data = [1, 0, 1, 2, 0]
        expected_result = 'NO'

        # when
        result = is_heap(n, data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_correct_data(self):
        # given
        n = 5
        data = [1, 3, 2, 5, 4]
        expected_result = 'YES'

        # when
        result = is_heap(n, data)

        # then
```

```
self.assertEqual(result, expected_result)

def test_should_check_time_data(self):
    # given
    expected_time = 2

    # when
    time = time_data(main)

    # then
    self.assertTrue(time < expected_time)

def test_should_check_memory_data(self):
    # given
    expected_memory = 256

    # when
    current, peak = memory_data(main)

    # then
    self.assertTrue(current < expected_memory)
    self.assertTrue(peak < expected_memory)

if __name__ == "__main__":
    unittest.main()
```

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм проверки на то, является ли массив “кучей” (или “неубывающей пирамидой”).

## Задание №3. Обработка сетевых пакетов

### 3 задача. 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  $A_i$  и время, необходимое процессору для его обработки  $P_i$  (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  $P_i$  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 ( $1 \leq S \leq 10^5$ ) и количество  $n$  ( $1 \leq n \leq 10^5$ ) входящих сетевых пакетов. Каждая из следующих  $n$  строк содержит два числа,  $i$ -ая строка содержит время прибытия пакета  $A_i$  ( $0 \leq A_i \leq 10^6$ ) и время его обработки  $P_i$  ( $0 \leq P_i \leq 10^3$ ) 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. ( $A_i \leq A_{i+1}$  для  $1 \leq i \leq n - 1$ .)
- 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.
- Пример 1:

input.txt	output.txt
1 0	

Если нет пакетов, ничего выводить не нужно.

• Пример 2:

input.txt	output.txt
1 1	0
0 0	

Единственный пакет поступил в момент времени 0, и компьютер обработал его сразу.

• Пример 3:

input.txt	output.txt
1 2	0
0 1	-1
0 1	

Первый пакет поступил в момент времени 0, второй пакет также, но был отброшен, так как сетевой буффер имеет размер 1 и он был полон (т.к. место занято первым пакетом). Первый пакет начал обрабатываться в момент времени 0, а второй не обрабатывался.

## Решение:

```
from utils import read, write

def network_packets(s, n, packets):
    if n == 0:
        return None
    deque = []
    result = []
    head = 0
    buffer_time = packets[0][0]
    for p in packets:
        start_time, p_i = p
        head = max([head] + [index for index in range(len(deque)) if deque[index] <= start_time])
        if len(deque[head+1:]) < s:
            result += [max(buffer_time, start_time)]
            buffer_time += p_i
            deque.append(buffer_time)
        else:
            result += [-1]
    return result

def main():
    data = [tuple(line) for line in read(type_convert=int)]
    s, n = data[0]
    packets = data[1:]
    write(end='')
    result = network_packets(s, n, packets)
    for res in result:
        write(res, to_end=True)
```

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

Функция `network_packets` имеет следующий алгоритм:

1. Проверка пустого списка: если количество пакетов равно нулю ( $n == 0$ ), то возвращается `None`.
2. Инициализация переменных:
  - 1) `deque`: пустой список, используемый для хранения времени завершения передачи каждого пакета.
  - 2) `result`: список, куда будут добавляться результаты обработки каждого пакета.
  - 3) `head`: указатель на первый элемент очереди, который еще не завершен.
  - 4) `buffer_time`: начальное время завершения первого пакета.
3. Основной цикл: Проходим по каждому пакету в списке `packets`.
  - 1) Разбираем текущий пакет на два значения: `start_time` (время начала передачи) и `p_i` (длительность передачи).
  - 2) Определяем новое значение `head` как максимальный индекс среди всех элементов `deque`, которые завершаются раньше или одновременно со временем начала нового пакета.
4. Обработка пакета:
  - 1) Если длина оставшейся части очереди (от позиции `head`) меньше размера буфера `s`, то есть место для размещения нового пакета:
    - 1.1 Добавляем в `result` максимальное значение между текущим временем буфера и временем начала пакета.
    - 1.2 Обновляем время завершения буфера (`buffer_time`), прибавляя к нему длительность пакета.
    - 1.3 Добавляем новое время завершения в очередь `deque`.
  - 2) Иначе добавляем в `result` значение `-1`, так как пакет не может быть обработан из-за переполнения буфера.
5. Возврат результата: После обработки всех пакетов возвращаем список `result`.

Результат работы программы:

Входные данные:

```
3 6  
0 2  
1 2  
2 2
```

```
3 2
4 2
5 2
```

Выходные данные:

```
0
2
4
6
8
-1
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab5.task3.src.network_packets import main, network_packets

class TestNetworkPackets(unittest.TestCase):

    def test_should_check_empty_data(self):
        # given
        s, n = 1, 0
        data = []
        expected_result = None

        # when
        result = network_packets(s, n, data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_example2_data(self):
        # given
        s, n = 1, 1
        data = [(0, 0)]
        expected_result = [0]

        # when
        result = network_packets(s, n, data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_example13_data(self):
        # given
        s, n = 2, 3
        data = [(0, 1), (3, 1), (10, 1)]
        expected_result = [0, 3, 10]

        # when
        result = network_packets(s, n, data)

        # then
        self.assertEqual(result, expected_result)
```



```

def test_should_check_example14_data(self):
    # given
    s, n = 3, 6
    data = [(0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2)]
    expected_result = [0, 2, 4, 6, 8, -1]

    # when
    result = network_packets(s, n, data)

    # then
    self.assertEqual(result, expected_result)

def test_should_check_time_data(self):
    # given
    expected_time = 10

    # when
    time = time_data(main)

    # then
    self.assertTrue(time < expected_time)

def test_should_check_memory_data(self):
    # given
    expected_memory = 512

    # when
    current, peak = memory_data(main)

    # then
    self.assertTrue(current < expected_memory)
    self.assertTrue(peak < expected_memory)

if __name__ == "__main__":
    unittest.main()

```

Вывод по задаче:

В ходе решения данной задачи мы реализовали программу для моделирования обработки сетевых пакетов, используя очередь.

## **Вывод**

В ходе выполнения лабораторной работы №4 мы изучили и реализовали такие структуры данных, как стек, очередь и связанный список.