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. Оче- редь с приоритетами

Выполнил:
Нгуен Хыу Жанг
К3140

Проверила:
Афанасьев А.В

Санкт-Петербург
2024 г

Содержание

Задачи по варианту

Задание 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 $a_i \le a_{2i}$,
2. если $2i + 1 \le n$, то $a_i \le a_{2i+1}$.

Дан массив целых чисел. Определите, является ли он неубывающей пирамидой.

- **Формат входного файла (input.txt).** Первая строка входного файла содер- жит 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 яв- ляется 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

```python
def is_min_heap(arr):
    n = len(arr)
    for i in range(n):
        left = 2 * i + 1
        right = 2 * i + 2

        if left < n and arr[i] > arr[left]:
            return "NO"

        if right < n and arr[i] > arr[right]:
            return "NO"

    return "YES"

input_file_path = '../txtf/input.txt'
output_file_path = '../txtf/output.txt'

with open(input_file_path) as f:
    n = int(f.readline().strip())
    arr = list(map(int, f.readline().strip().split()))

result = is_min_heap(arr)

with open(output_file_path, 'w') as f:
    f.write(result)
```

input.txt: 

output.txt: 

Функция `is_min_heap(arr)`

Эта функция принимает на вход массив `arr` и проверяет, соблюдается ли свойство неубывающей пирамиды:

1. Родительский элемент массива должен быть меньше или равен своему левому потомку.
2. Родительский элемент массива должен быть меньше или равен своему правому потомку.

Пошаговое описание:

1. **Определение размера массива**:
   `n = len(arr)`
   Здесь вычисляется длина массива, чтобы знать количество элементов.
2. **Цикл по всем элементам массива**:
   `for i in range(n):`
   Перебираются все индексы массива, начиная с 0 до `n-1`.
3. **Определение индексов потомков**:
   `left = 2 * i + 1`
   `right = 2 * i + 2`
   Для каждого элемента массива `arr[i]`:
   - Индекс левого потомка вычисляется как 2i + 1.
   - Индекс правого потомка вычисляется как 2i + 22.
4. **Проверка свойств неубывающей пирамиды**:
   ```
   if left < n and arr[i] > arr[left]:
       return "NO"
   if right < n and arr[i] > arr[right]:
       return "NO"
   ```
   - Если индекс левого потомка `left` находится в пределах массива (`left < n`) и родительский элемент `arr[i]` больше левого потомка `arr[left]`, возвращается `"NO"`, так как свойство пирамиды нарушено.
   - Аналогично проверяется правый потомок: если `arr[i] > arr[right]`, возвращается `"NO"`.
5. **Возврат результата**: Если ни одно из условий нарушения пирамиды не выполнено, функция завершает проверку и возвращает `"YES"`, что означает, что массив является неубывающей пирамидой.

Основная программа

Основная программа считывает данные из файла, вызывает функцию проверки и записывает результат в файл.

1. **Считывание входных данных**:
   `with open(input_file_path) as f:`

```
n = int(f.readline().strip())
arr = list(map(int, f.readline().strip().split()))
```

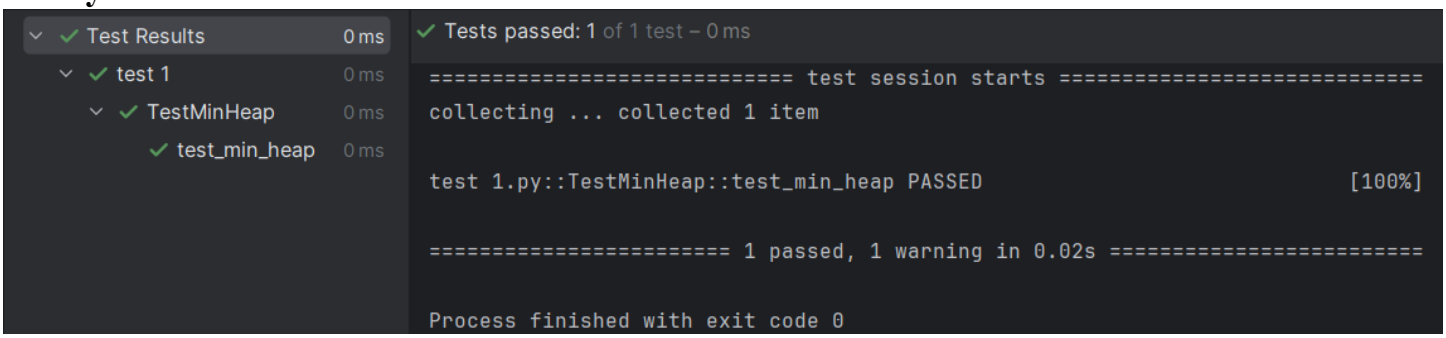
- Из файла `input.txt` считывается количество элементов `n`.
- Затем считывается массив чисел, который преобразуется из строки в список целых чисел.

2. **Вызов функции `is_min_heap`**:
   `result = is_min_heap(arr)`
   Проверяется, является ли массив `arr` неубывающей пирамидой.
3. **Запись результата в файл**:
   ```
   with open(output_file_path, 'w') as f:
       f.write(result)
   ```
   Результат работы программы (`"YES"` или `"NO"`) записывается в файл `output.txt`.

Unittest для задание 1:

```
import unittest
import sys
import os

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from c1 import is_min_heap

class TestMinHeap(unittest.TestCase):

    def test_min_heap(self):
        # given
        self.assertEqual(is_min_heap([1, 0, 1, 2, 0]), "NO")
        self.assertEqual(is_min_heap([1, 3, 2, 5, 4]), "YES")

        # when
        self.assertEqual(is_min_heap([1]), "YES")

        # then
        self.assertEqual(is_min_heap([1, 2, 3, 4, 5]), "YES")
        self.assertEqual(is_min_heap([5, 4, 3, 2, 1]), "NO")
        self.assertEqual(is_min_heap([1, 1, 1, 1, 1]), "YES")
        self.assertEqual(is_min_heap([2, 1, 2, 1, 2]), "NO")

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

*** Результат :**



1. Импорт необходимых модулей
`import unittest`

```
import sys
import os
```
Здесь импортируются стандартные модули Python:

- `unittest`: встроенный модуль для создания и запуска тестов
- `sys` и `os`: используются для манипуляции с путями файловой системы и настройки окружения.

2. Настройка пути к модулю
```
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src')))
from c1 import is_min_heap
```
Эти строки добавляют в начало списка путей поиска модулей (`sys.path`) путь к директории `src`, расположенной на уровень выше текущего файла. Это необходимо для импорта функции `is_min_heap` из модуля `c1`, находящегося в этой директории.

3. Определение класса тестов
```
class TestMinHeap(unittest.TestCase):
```
Создается класс `TestMinHeap`, наследующийся от `unittest.TestCase`. Это позволяет использовать встроенные методы для тестирования, такие как `assertEqual`

4. Метод тестирования `test_min_heap`
```
def test_min_heap(self):
```
Внутри класса определен метод `test_min_heap`, который будет автоматически распознан фреймворком `unittest` как тестовый, поскольку его имя начинается с `test_`.

5. Тестовые случаи

Внутри метода `test_min_heap` представлены несколько тестовых случаев:

- **Проверка с неупорядоченным списком:**
  `self.assertEqual(is_min_heap([1, 0, 1, 2, 0]), NO)`
  Ожидается, что функция `is_min_heap` вернет `NO` для данного списка, так как он не соответствует свойствам мин-кучи.
- **Проверка с корректной мин-кучей:**
  `self.assertEqual(is_min_heap([1, 3, 2, 5, 4]), YES)`
  Ожидается, что функция вернет `YES`, поскольку список соответствует свойствам мин-кучи.
- **Проверка с одним элементом:**
  `self.assertEqual(is_min_heap([1]), YES)`
  Ожидается, что список с одним элементом является мин-кучей.
- **Проверка с возрастающим порядком:**
  `self.assertEqual(is_min_heap([1, 2, 3, 4, 5]), YES)`
  Ожидается, что отсортированный по возрастанию список является мин-кучей.
- **Проверка с убывающим порядком:**
  `self.assertEqual(is_min_heap([5, 4, 3, 2, 1]), NO)`
  Ожидается, что отсортированный по убыванию список не является мин-кучей.
- **Проверка с одинаковыми элементами:**

```
self.assertEqual(is_min_heap([1, 1, 1, 1, 1]), YES)
```
Список с одинаковыми элементами должен быть признан мин-кучей.

- **Проверка с чередующимися элементами:**
```
self.assertEqual(is_min_heap([2, 1, 2, 1, 2]), NO)
```
Ожидается, что такой список не является мин-кучей.

6. Запуск тестов
```
if __name__ == '__main__':
    unittest.main()
```
Этот блок кода проверяет, запущен ли скрипт напрямую, и в этом случае вызывает `unittest.main()`, который запускает все тесты в модуле.

Задание 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 ($1 \le n \le 10^5$). 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 ($0 \le i \le n-1$). **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

- **Объяснение примера**. Данный входной файл задает 5 узлов дерева с числа- ми от 0 до 4. Узел под индексом 0 является дочерним узлом узла с индексом
4. Узел под индексом 1 - корневой узел. Узел с индексом 2 - тоже дочерний узел четвертого узла, а узлы с индексами 3 и 4 - дочерние узлы первого (кор- невого) узла. Можно записать данные узлы с соответствующми индексами, чтобы увидеть наглядно:

0	1	2	3	4
4	-1	4	1	1

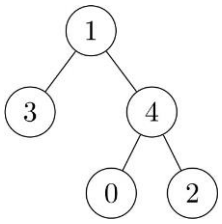
Давайте построим это дерево: Узел «1» – корневой, у него двое дочерних узла: «3»

и «4», и у узла «4» – тоже два дочерних: «0» и «2».

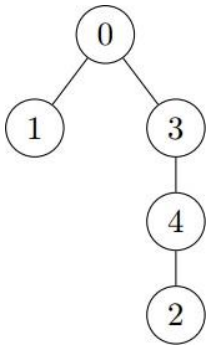
Таким образом, высота этого дерева равна 3, т.к. количество вершин на пути от корня 1 к листу 2 равно 3.

- Пример 2:

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



- **Объяснение примера**. Здесь также 5 узлов со значениями от 0 до 4, причем узел «0»



– корневой, узел «1» – дочерний узла «0», узел «2» – дочерний от узла «4», узел «3» – также дочерний узла «0» (корневого), и узел «4» – дочерний узла «3». Высота дерева равно 4, т.к. количество узлов на пути от корневого узла к листу «2» равно 4.

- Что делать. Высоту ***бинарного*** дерева можно посчитать рекурсивно:

```
def height(root):
                    if root is None:
                            return 0

        return max(height(root.left), height(root.right)) + 1
```

Однако дерево в задаче не обязательно бинарное, поэтому вам нужно адап- тировать подсчет высоты дерева для произвольного дерева. В этом случае дерево может быть очень глубоким, не допускайте переполения стека, если вы используете рекурсию, и тестируйте ваш алгоритм для максимальных данных и максимально возможной глубины.

Используйте тот факт, что значения каждого узла дерева - это его индекс, т.е. целочисленное значение от 0 до $n - 1$, и вы можете хранить значения каждого узла в массиве, при этом доступ к любому узлу будет $O(1)$.

```
import os

def read_tree_from_file(input_file_path):
    with open(input_file_path, 'r') as f:
        n = int(f.readline().strip())
```

```
        parents = list(map(int, f.readline().strip().split()))
    return n, parents

def build_tree(n, parents):
    tree = [[] for _ in range(n)]

    for child in range(n):
        parent = parents[child]
        if parent != -1:
            tree[parent].append(child)

    return tree

def calculate_height(tree, node, height):
    if not tree[node]:
        return height

    max_height = height
    for child in tree[node]:
        max_height = max(max_height, calculate_height(tree, child, height + 1))

    return max_height

def main():
    input_file_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'txtf', 'input.txt'))
    output_file_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'txtf', 'output.txt'))

    n, parents = read_tree_from_file(input_file_path)
    tree = build_tree(n, parents)

    root_index = parents.index(-1)

    height = calculate_height(tree, root_index, 1)

    with open(output_file_path, 'w') as f:
        f.write(str(height))

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

input.txt:
```
5
4 -1 4 1 1
```

output.txt:
```
3
```

1. Импорт необходимых модулей:

`import os`

Модуль `os` используется для работы с путями к файлам, обеспечивая совместимость с различными операционными системами.

2. Функция `read_tree_from_file(input_file_path)`:

```
def read_tree_from_file(input_file_path):
    with open(input_file_path, 'r') as f:
        n = int(f.readline().strip())
        parents = list(map(int, f.readline().strip().split()))
```

```
    return n, parents
```
Эта функция читает данные из файла:

- Первая строка содержит число узлов `n`.
- Вторая строка содержит список из `n` целых чисел, представляющих родительские связи.
- Функция возвращает количество узлов и список родительских связей.

3. Функция `build_tree(n, parents)`:

```
def build_tree(n, parents):
    tree = [[] for _ in range(n)]
    for child in range(n):
        parent = parents[child]
        if parent != -1:
            tree[parent].append(child)
    return tree
```
Эта функция строит представление дерева в виде списка списков:

- Создается список `tree` длиной `n`, где каждый элемент — пустой список.
- Итерация по всем узлам: если значение в `parents[child]` не равно `-1`, то добавляем текущий узел в список дочерних узлов его родителя.
- Функция возвращает построенное дерево.

4. Функция `calculate_height(tree, node, height)`:

```
def calculate_height(tree, node, height):
    if not tree[node]:
        return height
    max_height = height
    for child in tree[node]:
        max_height = max(max_height, calculate_height(tree,
child, height + 1))
    return max_height
```
Рекурсивная функция для вычисления высоты дерева:

- Если текущий узел не имеет дочерних (`tree[node]` пуст), возвращается текущая высота.
- Иначе инициализируется `max_height` текущей высотой и рекурсивно вычисляется высота для каждого дочернего узла, увеличивая текущую высоту на 1.
- Функция возвращает максимальную найденную высоту.

5. Функция `main()`:

```
def main():
    input_file_path =
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'txtf', 'input.txt'))
    output_file_path =
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'txtf', 'output.txt'))
    n, parents = read_tree_from_file(input_file_path)
    tree = build_tree(n, parents)
```

```
    root_index = parents.index(-1)
    height = calculate_height(tree, root_index, 1)
    with open(output_file_path, 'w') as f:
        f.write(str(height))
```

Основная функция, выполняющая последовательность шагов:

- Определяет пути к входному и выходному файлам.
- Считывает данные из входного файла с помощью `read_tree_from_file`.
- Строит дерево с использованием `build_tree`.
- Находит индекс корневого узла (значение `-1` в списке `parents`).
- Вычисляет высоту дерева, начиная с корня и высоты 1.
- Записывает результат в выходной файл.

6. Запуск программы:

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

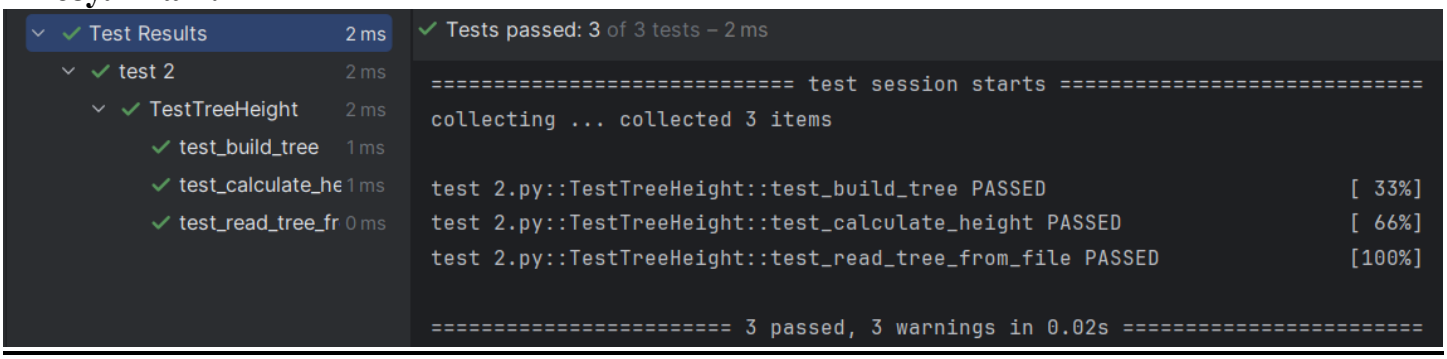
Эта конструкция обеспечивает запуск функции `main()` при выполнении скрипта напрямую.

Unittest для задание 2:

```
import unittest
import os
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from c2 import read_tree_from_file, build_tree, calculate_height

class TestTreeHeight(unittest.TestCase):

    def setUp(self):
        # given
        self.test_input_file = 'test_input.txt'
        self.test_output_file = 'test_output.txt'
        # then
        with open(self.test_input_file, 'w') as f:
            f.write('5\n4 -1 4 1 1\n')

    def tearDown(self):
        # given
        if os.path.exists(self.test_input_file):
            os.remove(self.test_input_file)
        # then
        if os.path.exists(self.test_output_file):
            os.remove(self.test_output_file)

    def test_read_tree_from_file(self):
        # given
        n, parents = read_tree_from_file(self.test_input_file)
        # when
        self.assertEqual(n, 5)
        # then
        self.assertEqual(parents, [4, -1, 4, 1, 1])

    def test_build_tree(self):
        # given
        n, parents = read_tree_from_file(self.test_input_file)
```

```
        # when
        tree = build_tree(n, parents)
        expected_tree = [
            [],
            [3, 4],
            [],
            [],
            [0, 2]
        ]
        # then
        self.assertEqual(tree, expected_tree)

    def test_calculate_height(self):
        # given
        n, parents = read_tree_from_file(self.test_input_file)
        # when
        tree = build_tree(n, parents)
        root_index = parents.index(-1)
        height = calculate_height(tree, root_index, 1)
        # then
        self.assertEqual(height, 3)

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

*** Результат :**



1. Импорт необходимых модулей и настройка пути:

```
import unittest
import os
import sys

sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src')))
from c2 import read_tree_from_file, build_tree, calculate_height
```

Здесь импортируются стандартные модули `unittest`, `os` и `sys`. С помощью `sys.path.insert` в список путей добавляется директория `'../src'`, что позволяет импортировать модуль `c2`.

2. Класс `TestTreeHeight`:

```
class TestTreeHeight(unittest.TestCase):
```

Класс `TestTreeHeight` наследуется от `unittest.TestCase`, что позволяет использовать встроенные методы для тестирования.

3. Методы `setUp` и `tearDown`:

```
def setUp(self):
```

```
    self.test_input_file = 'test_input.txt'
    self.test_output_file = 'test_output.txt'
    with open(self.test_input_file, 'w') as f:
        f.write('5\n4 -1 4 1 1\n')

def tearDown(self):
    if os.path.exists(self.test_input_file):
        os.remove(self.test_input_file)
    if os.path.exists(self.test_output_file):
        os.remove(self.test_output_file)
```

Метод `setUp` выполняется перед каждым тестом и создает файл `test_input.txt` с содержимым `'5\n4 -1 4 1 1\n'`. Метод `tearDown` выполняется после каждого теста и удаляет созданные файлы, обеспечивая чистоту тестового окружения.

4. Тестирование функции `read_tree_from_file`:

```
def test_read_tree_from_file(self):
    n, parents = read_tree_from_file(self.test_input_file)
    self.assertEqual(n, 5)
    self.assertEqual(parents, [4, -1, 4, 1, 1])
```

Этот тест проверяет, правильно ли функция `read_tree_from_file` считывает количество узлов и список родительских связей из файла. Ожидается, что `n` будет равно 5, а `parents` — `[4, -1, 4, 1, 1]`.

5. Тестирование функции `build_tree`:

```
def test_build_tree(self):
    n, parents = read_tree_from_file(self.test_input_file)
    tree = build_tree(n, parents)
    expected_tree = [
        [],
        [3, 4],
        [],
        [],
        [0, 2]
    ]
    self.assertEqual(tree, expected_tree)
```

Здесь проверяется, правильно ли функция `build_tree` строит дерево на основе входных данных. Ожидаемая структура дерева задается в `expected_tree`, и результат работы функции сравнивается с ней.

6. Тестирование функции `calculate_height`:

```
def test_calculate_height(self):
    n, parents = read_tree_from_file(self.test_input_file)
    tree = build_tree(n, parents)
    root_index = parents.index(-1)
    height = calculate_height(tree, root_index, 1)
    self.assertEqual(height, 3)
```

Этот тест проверяет, корректно ли функция `calculate_height` вычисляет высоту дерева. Корень дерева определяется по индексу элемента `-1` в списке `parents`, и ожидается, что высота дерева будет равна 3.

7. Запуск тестов:

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

Этот блок кода позволяет запускать тесты при выполнении скрипта напрямую. Функция `unittest.main()` обнаруживает и запускает все методы, начинающиеся с `test`.

Задание 3 : Обработка сетевых пакетов

В этой задаче вы реализуете программу для моделирования обработки сетевых пакетов.

- Вам дается серия входящих сетевых пакетов, и ваша задача - смоделиро- вать их обработку. Пакеты приходят в определенном порядке. Для каждого номера пакета i вы знаете время, когда пакет прибыл A_i и время, необходи- мое 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 обра- боткой. 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 несколь- ко 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 буфе- ра, как только заканчивает обработку предыдущего. Если в какой-то момент компьютер не занят и в буфере нет пакетов, компьютер просто ожидает при- бытия 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 A_i ($0 \le A_i \le 10^6$) и время его обработки P_i ($0 \le P_i \le 10^3$) 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 обраба- тывать; 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 1 0 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 занят первый пакет). Первый пакет начал обрабатываться в момент времени 0, а второй не обрабатывался.

- Пример 4:

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

Первый пакет поступил в момент времени 0, компьютер сразу начал его обрабатывать и закончил в момент времени 1. Второй пакет поступил во время 1, и компьютер так же начал его обрабатывать.

- Что делать. Для решения этой задачи вы можете использовать массив или очередь (точнее, дек, чтобы у вас был доступ к последнему элементу).

  Одно из возможных решений - сохранить в списке или очереди время, когда компьютер завершит обработку пакетов (`finish_time`), которые в настоя- щее время хранятся в сетевом буфере, в порядке возрастания. Когда при- бывает новый пакет, вам сначала нужно удалить в начале `finish_time` все пакеты, которые уже обработаны к моменту прибытия нового пакета. Затем вы добавляете время окончания для нового пакета в `finish_time`. Если буфер заполнен (в *очереди* `finish_time` уже S элементов), пакет отбрасы- вается. В противном случае время завершения его обработки добавляется к `finish_time`.

  Если при поступлении нового пакета очередь `finish_time` пуста, компью- тер начнет обработку нового пакета немедленно, как только он поступит.

В противном случае компьютер начнет обработку нового пакета, как толь- ко он закончит обработку последнего из пакетов, находящихся в настоя- щее время в `finish_time` (здесь вам нужно получить доступ к последнему элементу `finish_time`, чтобы определить, когда компьютер начнет обраба- тывать новый пакет). Вам также нужно будет вычислить время окончания обработки, добавив P_i к времени начала обработки и поместив его в конец `finish_time`.

- Еще примеры.

№	input.txt	output.txt
5	1 1 0 1	0
6	1 1 1 0	1
7	1 2 0 0 0 0	0 0
8	1 2 0 0 0 1	0 0
9	1 2 0 1 0 0	0 -1
10	1 2 0 1 0 1	0 -1
11	1 2 0 1 1 1	0 1
12	1 2 0 1 2 1	0 2
13	2 3 0 1 3 1 10 1	0 3 10
14	3 6 0 2 1 2 2 2 3 2 4 2	0 2 4 6 8 -1

```python
import os

def process_packets(buffer_size, packets):
    finish_time = []
    results = []
    current_time = 0

    for arrival_time, process_time in packets:
        while finish_time and finish_time[0] <= arrival_time:
            finish_time.pop(0)

        if len(finish_time) < buffer_size:
            if not finish_time:
                start_time = arrival_time
            else:
                start_time = finish_time[-1]

            finish_time.append(start_time + process_time)
            results.append(start_time)
        else:
            results.append(-1)

    return results

def main():
    input_file_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'txtf', 'input.txt'))
    output_file_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'txtf', 'output.txt'))

    with open(input_file_path, 'r') as f:
        buffer_size, n = map(int, f.readline().strip().split())
        packets = [tuple(map(int, f.readline().strip().split())) for _ in range(n)]

    results = process_packets(buffer_size, packets)
```

```
    with open(output_file_path, 'w') as f:
        for result in results:
            f.write(f"{result}\n")

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

input.txt:

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

output.txt:

```
0
-1
```

1. Общая структура

Код состоит из двух основных функций:

- **`process_packets`**: Обрабатывает входящие пакеты с учетом размера буфера и времени обработки.
- **`main`**: Читает данные из файла, вызывает функцию обработки пакетов и записывает результаты в файл.

2. Функция `process_packets`

Эта функция выполняет основную логику обработки пакетов. Ее входные параметры:

- `buffer_size`: размер буфера.
- `packets`: список пакетов, где каждый пакет представлен кортежем `(arrival_time, process_time)`.

Локальные переменные:

- **`finish_time`**: список времени завершения обработки пакетов, находящихся в буфере.
- **`results`**: список времени начала обработки пакетов, либо `-1`, если пакет был отброшен.
- **`current_time`**: текущее время обработки.

Этапы обработки:

1. **Удаление обработанных пакетов из буфера**:
   ```
   while finish_time and finish_time[0] <= arrival_time:
       finish_time.pop(0)
   ```
   - Удаляются пакеты, завершенные к моменту прибытия нового пакета.
   - `finish_time[0]` хранит время завершения первого пакета в очереди.
2. **Проверка на возможность добавления нового пакета в буфер**:
   ```
   if len(finish_time) < buffer_size:
   ```
   Если текущий размер буфера меньше `buffer_size`, пакет добавляется.
3. **Обработка пакета**:
   - Если буфер пуст:
     ```
     if not finish_time:
         start_time = arrival_time
     ```
     Обработка начинается сразу после прибытия пакета.
   - Если в буфере есть пакеты:
     ```
     else:
     ```

```
start_time = finish_time[-1]
```
Обработка начинается после завершения последнего пакета в буфере.

4. **Добавление времени завершения обработки в буфер**:
   `finish_time.append(start_time + process_time)`
   Рассчитывается время завершения текущего пакета.
5. **Обработка ситуации переполнения буфера**:
   ```
   else:
       results.append(-1)
   ```
   Если буфер переполнен, пакет отбрасывается.
6. **Сохранение результата**:
   `results.append(start_time)`
   Время начала обработки сохраняется в список `results`.

3. Функция `main`

Эта функция выполняет чтение входных данных, вызывает обработку пакетов и записывает результаты.

Этапы выполнения:

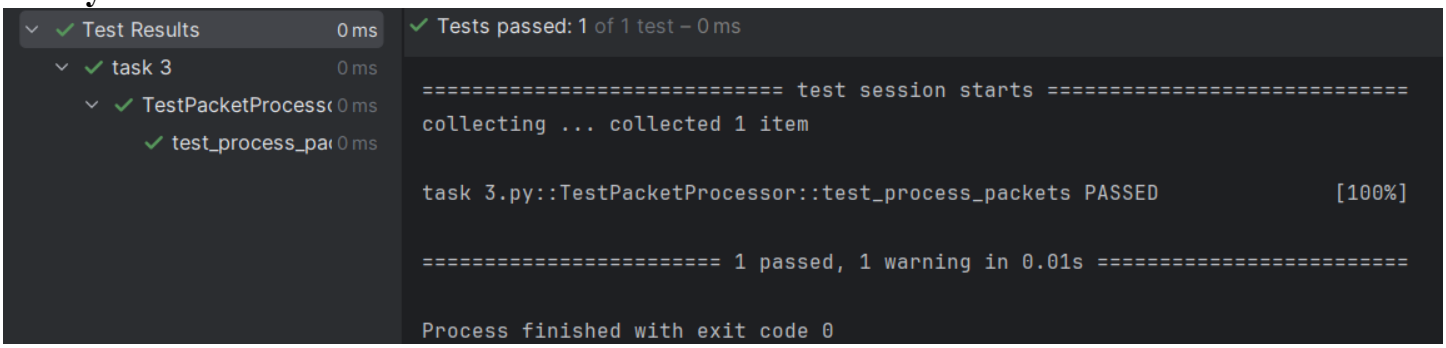
1. **Чтение данных из файла**:
   ```
   with open(input_file_path, 'r') as f:
       buffer_size, n = map(int, f.readline().strip().split())
       packets = [tuple(map(int,
   f.readline().strip().split())) for _ in range(n)]
   ```
   Считываются:
   - Размер буфера `buffer_size`.
   - Количество пакетов `n`.
   - Список пакетов `packets`.
2. **Вызов функции обработки**:
   `results = process_packets(buffer_size, packets)`
3. **Запись результатов в файл**:
   ```
   with open(output_file_path, 'w') as f:
       for result in results:
           f.write(f"{result}\n")
   ```

Unittest для задание 3:

```
import unittest
import os
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from c3 import process_packets

class TestPacketProcessor(unittest.TestCase):

    def test_process_packets(self):
        # given
        buffer_size = 1
        packets = [(0, 1)]
        expected = [0]
        result = process_packets(buffer_size, packets)
```

```
        self.assertEqual(result, expected)

        # when
        buffer_size = 1
        packets = [(0, 1), (0, 1)]
        expected = [0, -1]
        result = process_packets(buffer_size, packets)
        self.assertEqual(result, expected)

        # when
        buffer_size = 1
        packets = [(0, 1), (1, 1)]
        expected = [0, 1]
        result = process_packets(buffer_size, packets)
        self.assertEqual(result, expected)

        # when
        buffer_size = 2
        packets = [(0, 1), (0, 1)]
        expected = [0, 1]
        result = process_packets(buffer_size, packets)
        self.assertEqual(result, expected)

        # when
        buffer_size = 3
        packets = [(0, 1), (1, 1), (2, 1)]
        expected = [0, 1, 2]
        result = process_packets(buffer_size, packets)
        self.assertEqual(result, expected)

        # then
        buffer_size = 2
        packets = [(0, 1), (0, 1), (0, 1)]
        expected = [0, 1, -1]
        result = process_packets(buffer_size, packets)
        self.assertEqual(result, expected)

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

*** Результат :**



1. Общая структура

Код состоит из двух основных функций:

- **`process_packets`**: Обрабатывает входящие пакеты с учетом размера буфера и времени обработки.
- **`main`**: Читает данные из файла, вызывает функцию обработки пакетов и записывает результаты в файл.

2. Функция `process_packets`

Эта функция выполняет основную логику обработки пакетов. Ее входные параметры:

- `buffer_size`: размер буфера.
- `packets`: список пакетов, где каждый пакет представлен кортежем `(arrival_time, process_time)`.

Локальные переменные:

- **`finish_time`**: список времени завершения обработки пакетов, находящихся в буфере.
- **`results`**: список времени начала обработки пакетов, либо `-1`, если пакет был отброшен.
- **`current_time`**: текущее время обработки.

Этапы обработки:

1. **Удаление обработанных пакетов из буфера**:
   ```
   while finish_time and finish_time[0] <= arrival_time:
       finish_time.pop(0)
   ```
   - Удаляются пакеты, завершенные к моменту прибытия нового пакета.
   - `finish_time[0]` хранит время завершения первого пакета в очереди.
2. **Проверка на возможность добавления нового пакета в буфер**:
   `if len(finish_time) < buffer_size:`
   Если текущий размер буфера меньше `buffer_size`, пакет добавляется.
3. **Обработка пакета**:
   - Если буфер пуст:
     ```
     if not finish_time:
         start_time = arrival_time
     ```
     Обработка начинается сразу после прибытия пакета.
   - Если в буфере есть пакеты:
     ```
     else:
         start_time = finish_time[-1]
     ```
     Обработка начинается после завершения последнего пакета в буфере.
4. **Добавление времени завершения обработки в буфер**:
   `finish_time.append(start_time + process_time)`
   Рассчитывается время завершения текущего пакета.
5. **Обработка ситуации переполнения буфера**:
   ```
   else:
       results.append(-1)
   ```
   Если буфер переполнен, пакет отбрасывается.
6. **Сохранение результата**:
   `results.append(start_time)`
   Время начала обработки сохраняется в список `results`.

3. Функция `main`

Эта функция выполняет чтение входных данных, вызывает обработку пакетов и записывает результаты.

Этапы выполнения:

1. **Чтение данных из файла**:

```
with open(input_file_path, 'r') as f:
    buffer_size, n = map(int, f.readline().strip().split())
    packets = [tuple(map(int,
f.readline().strip().split())) for _ in range(n)]
```

Считываются:

- Размер буфера `buffer_size`.
- Количество пакетов `n`.
- Список пакетов `packets`.

2. **Вызов функции обработки**:

```
results = process_packets(buffer_size, packets)
```

3. **Запись результатов в файл**:

```
with open(output_file_path, 'w') as f:
    for result in results:
        f.write(f"{result}\n")
```

Задание 4 : Построение пирамиды

В этой задаче вы преобразуете массив целых чисел в пирамиду. Это важней- ший шаг алгоритма сортировки под названием HeapSort. Гарантированное время работы в худшем случае составляет $O(n \log n)$, в отличие от ***среднего*** време- ни работы QuickSort, равного $O(n \log n)$. QuickSort обычно используется на практике, потому что обычно он быстрее, но HeapSort используется для внеш- ней 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 мас- сив целых чисел в пирамиду. Вы сделаете это, применив к массиву определенное количество перестановок (swaps). Перестановка - это операция, как вы помните, при которой элементы a_i и a_j массива меняются местами для некоторых i и j. Вам нужно будет преобразовать массив в пирамиду, используя только $O(n)$ пе- рестановок. 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 a_i входного массива, разделенных пробелом ($0 \le a_i \le 10^9$, все a_i - различны.)
- **Формат выходного файла (output.txt).** Первая строка ответа должна со- держать целое число m - количество сделанных свопов. Число m должно удовлетворять условию $0 \le m \le 4n$. Следующие m строк должны содер- жать по 2 числа: индексы i и j сделанной перестановки двух элементов, **индексы считаются с 0**. После всех перестановок в нужном порядке мас- сив 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 мас- сив

становится корректной пирамидой, считается верной.

- Ограничение по времени. 3 сек.
- Ограничение по памяти. 512 мб.
- Пример 1:

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

После перестановки элементов в позициях 1 и 4 массив становится следую- щим: 5 1 3 2 4.

Далее, перестановка элементов с индексами 0 и 1: 1 5 3 2 4. И напоследок, переставим 1 и 3: 1 2 3 5 4, и теперь это корректная неубывающая пирамида.

- Пример 2:

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

```python
import os

def swap(arr, i, j, swaps):
    arr[i], arr[j] = arr[j], arr[i]
    swaps.append((i, j))

def heapify(arr, n, i, swaps):
    smallest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] < arr[smallest]:
        smallest = left
    if right < n and arr[right] < arr[smallest]:
        smallest = right

    if smallest != i:
        swap(arr, i, smallest, swaps)
        heapify(arr, n, smallest, swaps)

def build_min_heap(arr):
    n = len(arr)
    swaps = []
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i, swaps)
    return swaps

def main():
    input_file_path = os.path.join('..', 'txtf', 'input.txt')
    output_file_path = os.path.join('..', 'txtf', 'output.txt')

    with open(input_file_path, 'r') as f:
        n = int(f.readline().strip())
        arr = list(map(int, f.readline().strip().split()))
```

```
    swaps = build_min_heap(arr)

    with open(output_file_path, 'w') as f:
        f.write(f"{len(swaps)}\n")
        for i, j in swaps:
            f.write(f"{i} {j}\n")

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

input.txt:

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

output.txt:

```
3
1 4
0 1
1 3
```

1. Функция `swap`: Меняет местами два элемента массива и записывает эту операцию в список `swaps`.

```
def swap(arr, i, j, swaps):
    arr[i], arr[j] = arr[j], arr[i]
    swaps.append((i, j))
```

2. Функция `heapify`: Рекурсивно восстанавливает свойства минимальной кучи, начиная с индекса `i`. Она сравнивает родительский элемент с дочерними и, при необходимости, меняет их местами, вызывая себя для поддерева, в котором произошла перестановка.

```
def heapify(arr, n, i, swaps):
    smallest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] < arr[smallest]:
        smallest = left
    if right < n and arr[right] < arr[smallest]:
        smallest = right
    if smallest != i:
        swap(arr, i, smallest, swaps)
        heapify(arr, n, smallest, swaps)
```

3. Функция `build_min_heap`: Преобразует весь массив в минимальную кучу, начиная с последнего узла, который имеет дочерние элементы. Она вызывает `heapify` для каждого узла, начиная с индекса `n // 2 - 1` и двигаясь к корню.

```
def build_min_heap(arr):
    n = len(arr)
    swaps = []
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i, swaps)
```

```
    return swaps
```

4. Функция `main`: Читает входные данные из файла `input.txt`, вызывает `build_min_heap` для преобразования массива в кучу и записывает результат в файл `output.txt`.

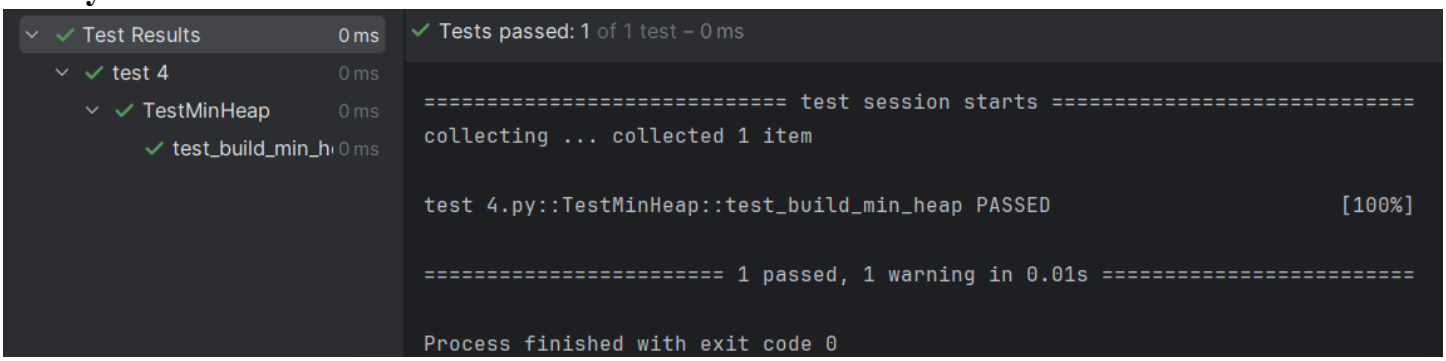
```
def main():
    input_file_path = os.path.join('..', 'txtf', 'input.txt')
    output_file_path = os.path.join('..', 'txtf', 'output.txt')
    with open(input_file_path, 'r') as f:
        n = int(f.readline().strip())
        arr = list(map(int, f.readline().strip().split()))
    swaps = build_min_heap(arr)
    with open(output_file_path, 'w') as f:
        f.write(f"{len(swaps)}\n")
        for i, j in swaps:
            f.write(f"{i} {j}\n")
```

Unittest для задание 4:

```
import unittest
import os
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from c4 import build_min_heap

class TestMinHeap(unittest.TestCase):
    def is_min_heap(self, arr):
        n = len(arr)
        for i in range(n // 2):
            left = 2 * i + 1
            right = 2 * i + 2
            if left < n and arr[i] > arr[left]:
                return False
            if right < n and arr[i] > arr[right]:
                return False
        return True

    def test_build_min_heap(self):
        # given
        arr = [5, 4, 3, 2, 1]
        swaps = build_min_heap(arr)
        expected_swaps = [(1, 4), (0, 1), (1, 3)]

        # when
        self.assertTrue(len(swaps) <= 4 * len(arr))

        # then
        for s in expected_swaps:
            self.assertIn(s, swaps)

        # given
        arr = [1, 2, 3, 4, 5]
        # when
        swaps = build_min_heap(arr)
        # then
        self.assertEqual(len(swaps), 0)
```

```
        # given
        arr = [3, 1, 5, 2, 4]
        # when
        swaps = build_min_heap(arr)
        # then
        self.assertTrue(self.is_min_heap(arr))

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

*** Результат :**



1. Импортирование необходимых модулей

```
import unittest
import os
import sys
```

- `unittest`: встроенный модуль Python для создания и выполнения тестов.
- `os`: модуль для взаимодействия с операционной системой, например, для работы с путями файлов.
- `sys`: модуль для доступа к некоторым переменным, используемым или поддерживаемым интерпретатором Python, включая пути поиска модулей.

2. Добавление пути к исходному коду в системный путь

```
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src')))
```

Эта строка добавляет директорию `src` в начало системного пути поиска модулей. Это необходимо, чтобы Python мог найти модуль `c4`, расположенный в этой директории.

3. Импортирование функции `build_min_heap` из модуля `c4`

```
from c4 import build_min_heap
```

Здесь импортируется функция `build_min_heap` из модуля `c4`, которая, предположительно, строит минимальную кучу из переданного списка.

4. Определение класса теста

```
class TestMinHeap(unittest.TestCase):
```

Создается класс `TestMinHeap`, наследующий от `unittest.TestCase`. Это позволяет использовать все возможности фреймворка `unittest` для организации и выполнения тестов.

5. Вспомогательный метод для проверки свойства минимальной кучи

```
def is_min_heap(self, arr):
    n = len(arr)
```

```
    for i in range(n // 2):
        left = 2 * i + 1
        right = 2 * i + 2
        if left < n and arr[i] > arr[left]:
            return False
        if right < n and arr[i] > arr[right]:
            return False
    return True
```

Этот метод проверяет, является ли переданный список `arr` минимальной кучей. В минимальной куче каждый родительский элемент не больше своих дочерних элементов.

6. Тестирование функции `build_min_heap`

```
def test_build_min_heap(self):
    # given
    arr = [5, 4, 3, 2, 1]
    swaps = build_min_heap(arr)
    expected_swaps = [(1, 4), (0, 1), (1, 3)]

    # when
    self.assertTrue(len(swaps) <= 4 * len(arr))

    # then
    for s in expected_swaps:
        self.assertIn(s, swaps)

    # given
    arr = [1, 2, 3, 4, 5]
    # when
    swaps = build_min_heap(arr)
    # then
    self.assertEqual(len(swaps), 0)

    # given
    arr = [3, 1, 5, 2, 4]
    # when
    swaps = build_min_heap(arr)
    # then
    self.assertTrue(self.is_min_heap(arr))
```

В этом методе выполняются следующие шаги:

- **Подготовка данных**:
  - Создается список `arr = [5, 4, 3, 2, 1]`.
  - Вызвается функция `build_min_heap`, которая должна преобразовать `arr` в минимальную кучу, и возвращает список выполненных обменов `swaps`.
  - Определяется список ожидаемых обменов `expected_swaps = [(1, 4), (0, 1), (1, 3)]`.

- **Проверка количества обменов**:
  - Проверяется, что количество выполненных обменов не превышает четырех умноженных на длину списка `arr`. Это условие может быть связано с теоретическими пределами сложности алгоритма.
- **Проверка конкретных обменов**:
  - Для каждого ожидаемого обмена из `expected_swaps` проверяется, что он присутствует в списке выполненных обменов `swaps`.
- **Тестирование на уже отсортированном списке**:
  - Создается отсортированный список `arr = [1, 2, 3, 4, 5]`.
  - Проверяется, что функция `build_min_heap` не выполняет никаких обменов, то есть `swaps` пуст.
- **Проверка на произвольном списке**:
  - Создается список `arr = [3, 1, 5, 2, 4]`.
  - Проверяется, что после выполнения `build_min_heap` список `arr` является минимальной кучей, используя вспомогательный метод `is_min_heap`.

7. Запуск тестов

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

Этот блок кода позволяет запускать тесты, если скрипт выполняется напрямую. Функция `unittest.main()` обнаруживает все методы, начинающиеся с `test`, и выполняет их.

Задание 5 : 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 независи- мых потоков для обработки заданного списка m заданий. Потоки берут задания в том порядке, в котором они указаны во входных данных. Если есть свобод- ный 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 зада- ния 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 об- рабатывать и когда он начнет обработку.

- **Формат ввода или входного файла (input.txt).** Первая строка содержит целые числа n и m ($1 \le n \le 10^5$, $1 \le m \le 10^5$). Вторая строка содержит m целых чисел t_i - время в секундах, которое требуется для выполнения i-ой задания любым потоком ($0 \le t_i \le 10^9$). 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, при- чем i-ая строка (начиная с 0) должна содержатьдва 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 1 2 3 4 5	0 0 1 0 0 1 1 2 0 4

1. Два потока пытаются одновременно взять задания из списка, поэто- му поток с индексом 0 фактически берет первое задание и начинает работать над ним в момент 0.
2. Поток с индексом 1 берет второе задание и начинает работать над ним также в момент 0.
3. Через 1 секунду поток 0 завершает первое задание, берет третье за- дание из списка и сразу же начинает его выполнять в момент времени 1.
4. Через секунду поток 1 завершает второе задание, берет четвертое за- дание из списка и сразу же начинает его выполнять в момент времени 2.
5. Наконец, еще через 2 секунды поток 0 завершает третье задание, берет пятое задание из списка и сразу же начинает его выполнять в момент времени 4.

- Пример 2:

input.txt
4 20 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

output.txt
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

Задания берутся 4 потоками по 4 штуки за раз, обрабатываются за 1 секунду, а затем приходит следующий набор из 4 заданий. Это происходит 5 раз, начиная с моментов 0, 1, 2, 3 и 4. После этого обрабатываются все $5 \times 4 = 20$ заданий.

- Что делать? Подумайте о последовательности событий, когда один из по- токов становится свободным (в самом начале и позже, после завершения некоторого задания). Как применить очередь с приоритетами, чтобы имити- ровать обработку этих заданий в нужном порядке? Не забудьте рассмотреть случай, когда одновременно освобождаются несколько потоков.

```python
import heapq
import os

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    with open(input_path, 'r') as f:
        n, m = map(int, f.readline().strip().split())
        tasks = list(map(int, f.readline().strip().split()))

    min_heap = []
    for i in range(n):
        heapq.heappush(min_heap, (0, i))

    results = []

    for task_time in tasks:
        finish_time, thread_index = heapq.heappop(min_heap)

        results.append((thread_index, finish_time))

        new_finish_time = finish_time + task_time

        heapq.heappush(min_heap, (new_finish_time, thread_index))

    with open(output_path, 'w') as f:
        for thread_index, start_time in results:
            f.write(f"{thread_index} {start_time}\n")

if __name__ == "__main__":
    task_scheduler()
```

input.txt:

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

output.txt:

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

1. Вводные данные:

- **n**: количество потоков.
- **m**: количество заданий.

- $t_1, t_2, ..., t_k$: время в секундах, необходимое для выполнения каждого из m заданий.

2. Выходные данные:

Для каждого задания выводится:

- Индекс потока, который будет его выполнять.
- Время в секундах, когда этот поток начнёт выполнение задания.

3. Алгоритм решения:

Для эффективного распределения заданий между потоками и учёта времени их завершения используется очередь с приоритетами (куча). Каждый элемент в очереди представляет собой кортеж из двух элементов: времени, когда поток освободится, и индекса потока. При добавлении нового задания в очередь учитывается время его завершения, что позволяет определить, когда поток будет доступен для следующего задания.

4. Пояснение к коду:

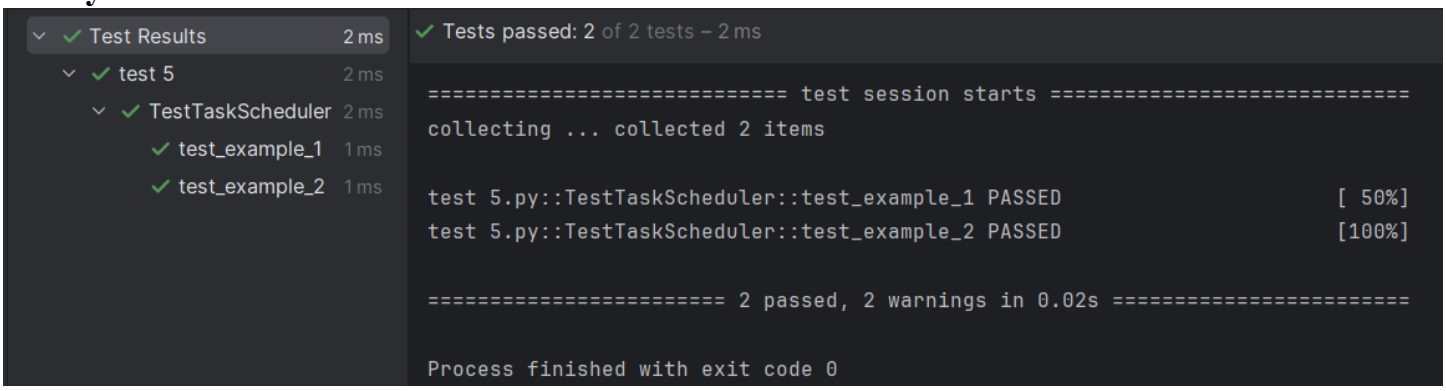
- **Чтение входных данных:** Открываем файл `input.txt`, считываем количество потоков и заданий, а также список времён выполнения заданий.
- **Инициализация кучи:** Создаём кучу, где каждый элемент — это кортеж из времени, когда поток освободится, и индекса потока. Изначально все потоки доступны с момента времени 0.
- **Обработка заданий:** Для каждого задания извлекаем поток с минимальным временем завершения, записываем его индекс и время начала выполнения задания, затем обновляем время завершения этого потока и возвращаем его в кучу.
- **Запись результатов:** Открываем файл `output.txt` и записываем для каждого задания индекс потока и время его начала выполнения.

Unittest для задание 5:

```
import unittest
import os
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from c5 import task_scheduler

class TestTaskScheduler(unittest.TestCase):
    def test_example_1(self):
        # given
        input_data = "2 5\n1 2 3 4 5\n"
        expected_output = "0 0\n1 0\n0 1\n1 2\n0 4\n"

        # when
        with open(os.path.join('..', 'txtf', 'input.txt'), 'w') as f:
            f.write(input_data)

        # when
        task_scheduler()

        # when
        with open(os.path.join('..', 'txtf', 'output.txt'), 'r') as f:
            result = f.read()

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

```
    def test_example_2(self):
        # given
        input_data = "4 20\n" + "1 " * 20 + "\n"
        expected_output = "\n".join([
            f"{i % 4} {i // 4}"
            for i in range(20)
        ]) + "\n"

        # when
        with open(os.path.join('..', 'txtf', 'input.txt'), 'w') as f:
            f.write(input_data)

        # when
        task_scheduler()

        # when
        with open(os.path.join('..', 'txtf', 'output.txt'), 'r') as f:
            result = f.read()

        # then
        self.assertEqual(result, expected_output)

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

*** Результат :**



1. Импорт необходимых модулей

```
import unittest
import os
import sys
```

- `unittest`: встроенный модуль Python для создания и выполнения тестов.
- `os`: модуль для взаимодействия с операционной системой, включая работу с путями файлов.
- `sys`: модуль для доступа к некоторым переменным, используемым или поддерживаемым интерпретатором Python, включая `sys.path`, который определяет пути поиска модулей.

2. Добавление пути к исходному коду в `sys.path`

```
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src')))
```

Эта строка добавляет директорию `src` в начало списка путей поиска модулей Python. `os.path.dirname(__file__)` получает путь к текущему файлу, затем

`os.path.join` поднимается на уровень выше (`..`) и переходит в папку `src`. Это необходимо для того, чтобы Python мог импортировать модули из этой директории, даже если она не находится в стандартных путях поиска.

3. Импорт функции `task_scheduler` из модуля `c5`

```
from c5 import task_scheduler
```

Здесь происходит импорт функции `task_scheduler` из модуля `c5`, который должен находиться в директории `src`. Предполагается, что эта функция реализует некоторую задачу, которую мы будем тестировать.

4. Определение класса теста

```
class TestTaskScheduler(unittest.TestCase):
```

Создается класс `TestTaskScheduler`, наследующий от `unittest.TestCase`. Это позволяет использовать все возможности фреймворка `unittest` для организации и выполнения тестов.

5. Определение первого теста

```
def test_example_1(self):
    # given
    input_data = '2 5\n1 2 3 4 5\n'
    expected_output = '0 0\n1 0\n0 1\n1 2\n0 4\n'

    # when
    with open(os.path.join('..', 'txtf', 'input.txt'), 'w') as
f:
        f.write(input_data)

    # when
    task_scheduler()

    # then
    with open(os.path.join('..', 'txtf', 'output.txt'), 'r') as
f:
        result = f.read()

    self.assertEqual(result, expected_output)
```

- **Подготовка данных**: В переменную `input_data` записываются входные данные, которые будут использованы функцией `task_scheduler`. `expected_output` содержит ожидаемый результат работы этой функции.
- **Запись входных данных в файл**: Открывается файл `input.txt` в директории `txtf` для записи и записываются подготовленные входные данные.
- **Выполнение тестируемой функции**: Вызывается функция `task_scheduler`, которая должна обработать данные из `input.txt` и записать результат в `output.txt`.

- **Чтение и проверка результата**: Открывается файл `output.txt`, читается его содержимое и сравнивается с ожидаемым результатом с помощью метода `assertEqual`.

6. Определение второго теста

```
def test_example_2(self):
    # given
    input_data = '4 20\n + 1  * 20 + \n'
    expected_output = '\n'.join([f'{i % 4} {i // 4}' for i in
range(20)]) + '\n'

    # when
    with open(os.path.join('..', 'txtf', 'input.txt'), 'w') as
f:
        f.write(input_data)

    # when
    task_scheduler()

    # then
    with open(os.path.join('..', 'txtf', 'output.txt'), 'r') as
f:
        result = f.read()

    self.assertEqual(result, expected_output)
```

Этот тест аналогичен первому, но с другими входными данными и ожидаемым результатом. Он проверяет, правильно ли функция `task_scheduler` обрабатывает другой набор данных.

7. Запуск тестов

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

Этот блок кода позволяет запускать тесты, если файл выполняется как основная программа. Метод `unittest.main()` обнаруживает все методы, начинающиеся с `test`, и выполняет их.

Задание 6 : Очередь с приоритетами

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

- **Формат входного файла (input.txt).** В первой строке входного файла со- держится число n ($1 \le n \le 10^6$) - число операций с очередью.
  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*. Гарантиру- ется, 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 резуль- тат 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.
- Пример:

input.txt	output.txt
8 A 3 A 4 A 2 X D 2 1 X X X	2 1 3 *

```
import heapq
import os

class PriorityQueue:
    def __init__(self):
        self.elements = []
        self.entry_finder = {}
        self.REMOVED = '<removed>'
        self.counter = 0

    def add(self, x):
        if x in self.entry_finder:
            self.remove(x)
        entry = [x, self.counter]
        self.entry_finder[x] = entry
        heapq.heappush(self.elements, entry)
        self.counter += 1

    def remove(self, x):
        entry = self.entry_finder.pop(x)
        entry[-1] = self.REMOVED

    def pop(self):
```

```
        while self.elements:
            x, count = heapq.heappop(self.elements)
            if count != self.REMOVED:
                del self.entry_finder[x]
                return x
        return None

def process_operations(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    pq = PriorityQueue()
    results = []

    with open(input_path, 'r') as f:
        n = int(f.readline().strip())
        operations = f.readlines()

    for i, operation in enumerate(operations):
        parts = operation.strip().split()
        cmd = parts[0]

        if cmd == 'A':
            x = int(parts[1])
            pq.add(x)
        elif cmd == 'X':
            min_elem = pq.pop()
            if min_elem is None:
                results.append('*')
            else:
                results.append(str(min_elem))
        elif cmd == 'D':
            idx = int(parts[1]) - 1
            new_value = int(parts[2])

            old_value = int(operations[idx].strip().split()[1])
            pq.remove(old_value)
            pq.add(new_value)

    with open(output_path, 'w') as f:
        f.write('\n'.join(results) + '\n')

if __name__ == "__main__":
    process_operations()
```

input.txt:

```
8
A 3
A 4
A 2
X
D 2 1
X
X
X
```

output.txt:

```
2
1
3
*
```

```
import heapq
import os
```

Импорт библиотек:

- `heapq`: Это стандартная библиотека Python, которая предоставляет функции для работы с кучами (heap). Мы будем использовать ее для реализации очереди с приоритетами.
- `os`: Библиотека для взаимодействия с операционной системой, используется для работы с путями к файлам.

Класс `PriorityQueue`:

```
class PriorityQueue:
    def __init__(self):
        self.elements = []
        self.entry_finder = {}
        self.REMOVED = '<removed>'
        self.counter = 0
```

- **`elements`**: Список, который будет хранить элементы очереди.
- **`entry_finder`**: Словарь, который связывает значения с их записями в очереди, чтобы мы могли легко найти и удалить их.
- **`REMOVED`**: Специальное значение, чтобы пометить удаленные элементы.
- **`counter`**: Счетчик, используемый для обеспечения уникальности каждого элемента в очереди.

Метод `add`:

```
def add(self, x):
    if x in self.entry_finder:
        self.remove(x)
    entry = [x, self.counter]
    self.entry_finder[x] = entry
    heapq.heappush(self.elements, entry)
    self.counter += 1
```

- Этот метод добавляет элемент `x` в очередь.
- Если элемент уже существует, он помечается как удаленный.
- Создается новая запись `entry`, которая состоит из значения и счетчика.
- Элемент добавляется в кучу с помощью `heapq.heappush`.
- Счетчик увеличивается для следующих добавлений.

Метод `remove`:

```
def remove(self, x):
    entry = self.entry_finder.pop(x)
    entry[-1] = self.REMOVED
```

- Удаляет элемент `x` из очереди путем пометки его как удаленного.

Метод pop:

```
def pop(self):
    while self.elements:
        x, count = heapq.heappop(self.elements)
        if count != self.REMOVED:
            del self.entry_finder[x]
            return x
    return None
```

- Извлекает минимальный элемент из очереди.
- Если элемент помечен как удаленный, он продолжает извлекать, пока не найдет действительный элемент.
- Возвращает минимальный элемент или `None`, если очередь пуста.

Функция `process_operations`:

```
def process_operations(input_file='input.txt',
output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)
    pq = PriorityQueue()
    results = []
```

- Функция для обработки операций с очередью.
- Определяются пути к входному и выходному файлам.
- Создается экземпляр `PriorityQueue` и инициализируется список `results` для хранения результатов операций.

Чтение входного файла:

```
    with open(input_path, 'r') as f:
        n = int(f.readline().strip())
        operations = f.readlines()
```

- Открывается файл для чтения.
- Читается количество операций `n` и все операции в список `operations`.

Обработка операций:

```
    for i, operation in enumerate(operations):
        parts = operation.strip().split()
        cmd = parts[0]
```

- Перебираем все операции, разбиваем каждую на части для определения команды.

Обработка команд:

- **Добавление элемента**:

  ```
  if cmd == 'A':
      x = int(parts[1])
      pq.add(x)
  ```

  - Если команда `A`, добавляем элемент `x` в очередь.
- **Удаление минимального элемента**:

  ```
  elif cmd == 'X':
      min_elem = pq.pop()
      if min_elem is None:
  ```

```
            results.append('*')
        else:
            results.append(str(min_elem))
```

- Если команда `X`, извлекаем минимальный элемент. Если очередь пуста, добавляем `*` в результаты.

- **Уменьшение элемента**:

```
elif cmd == 'D':
    idx = int(parts[1]) - 1
    new_value = int(parts[2])
    old_value = int(operations[idx].strip().split()[1])
    pq.remove(old_value)
    pq.add(new_value)
```

- Если команда `D`, получаем индекс элемента, который нужно изменить. Удаляем старое значение и добавляем новое.

Запись результатов в выходной файл:

```
    with open(output_path, 'w') as f:
        f.write('\n'.join(results) + '\n')
```

- Открываем файл для записи и записываем все результаты.

Главный блок:

```
if __name__ == "__main__":
    process_operations()
```

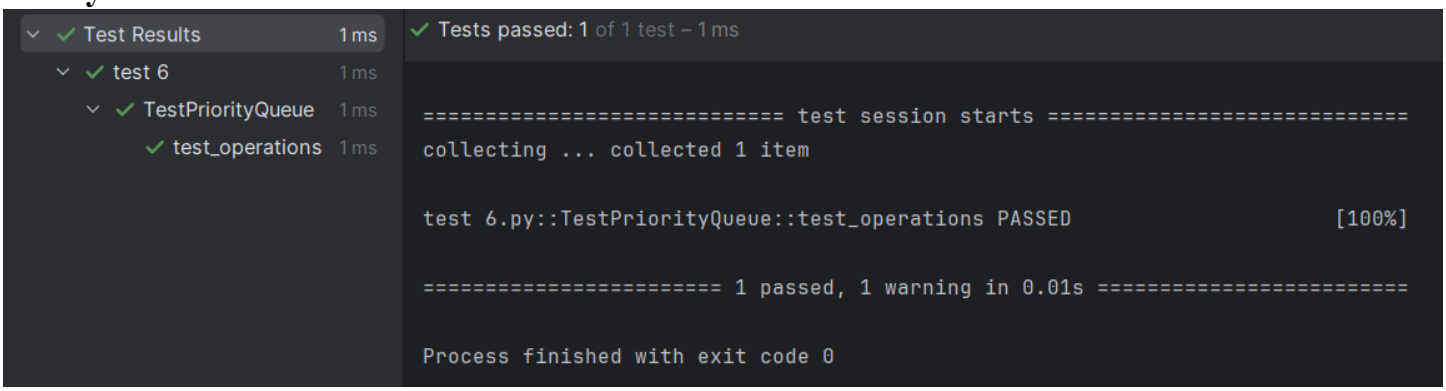
- Проверка, запускается ли скрипт напрямую. Если да, вызывается функция `process_operations`.

Unittest для задание 6:

```
import unittest
import os
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from c6 import process_operations

class TestPriorityQueue(unittest.TestCase):

    def test_operations(self):
        # given
        input_data = "8\nA 3\nA 4\nA 2\nX\nD 2 1\nX\nX\nX\n"
        expected_output = "2\n1\n3\n*\n"

        # when
        with open(os.path.join('..', 'txtf', 'input.txt'), 'w') as f:
            f.write(input_data)

        # when
        process_operations()

        # when
        with open(os.path.join('..', 'txtf', 'output.txt'), 'r') as f:
            result = f.read()

        # then
```

```
        self.assertEqual(result, expected_output)

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

*** Результат :**




```
import unittest
import os
import sys
```

Импорт библиотек:

- **unittest**: Стандартная библиотека Python для написания и выполнения тестов.
- **os**: Библиотека для работы с операционной системой, используется для работы с файловой системой.
- **sys**: Позволяет взаимодействовать с интерпретатором Python, в данном случае используется для изменения пути поиска модулей.

```
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src')))
from c6 import process_operations
```

- С помощью sys.path.insert добавляем путь к папке src в список путей, по которым Python ищет модули. Это позволяет импортировать функцию process_operations из модуля c6, который находится в папке src.

Класс `TestPriorityQueue`:

```
class TestPriorityQueue(unittest.TestCase):
```

- Создается класс TestPriorityQueue, который наследует от unittest.TestCase. Это позволяет использовать все функции для тестирования, предоставляемые unittest.

Метод `test_operations`:

```
def test_operations(self):
```

- Определяется метод test_operations, который будет содержать тестовые сценарии.

Подготовка данных:

```
# given
input_data = "8\nA 3\nA 4\nA 2\nX\nD 2 1\nX\nX\nX\n"
expected_output = "2\n1\n3\n*\n"
```

- **input_data**: Строка, представляющая входные данные для теста. Содержит 8 операций с очередью.
- **expected_output**: Ожидаемый результат выполнения операций, который будет сравнен с фактическим результатом.

Запись входных данных в файл:

```
with open(os.path.join('..', 'txtf', 'input.txt'), 'w') as f:
    f.write(input_data)
```

- Открывается файл input.txt для записи в режиме w (write). Путь к файлу создается с помощью os.path.join, чтобы обеспечить кросс-платформенность.
- В input.txt записываются входные данные.

Вызов функции обработки операций:

```
process_operations()
```

- Вызывается функция process_operations, которая будет обрабатывать входные данные и записывать результаты в выходной файл.

Чтение результатов из выходного файла:

```
with open(os.path.join('..', 'txtf', 'output.txt'), 'r') as f:
    result = f.read()
```

- Открывается файл output.txt для чтения в режиме r (read).
- Содержимое файла читается и сохраняется в переменной result.

Проверка результата:

```
self.assertEqual(result, expected_output)
```

- Используется метод assertEqual для сравнения фактического результата (result) с ожидаемым (expected_output). Если они не равны, тест не пройдет.

Главный блок:

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

- Проверка, запущен ли скрипт напрямую. Если да, вызывается функция unittest.main(), которая запускает все тесты, определенные в классе.

Задание 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 со- держится 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 от- сортированным по невозрастанию 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; ко- гда 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).

```python
import os

def max_heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        max_heapify(arr, i, 0)

def read_input(file_path):
    with open(file_path, 'r') as f:
        n = int(f.readline().strip())
        arr = list(map(int, f.readline().strip().split()))
    return arr

def write_output(file_path, arr):
    with open(file_path, 'w') as f:
        f.write(' '.join(map(str, arr)) + '\n')

def main():
    input_path = os.path.join('..', 'txtf', 'input.txt')
    output_path = os.path.join('..', 'txtf', 'output.txt')

    arr = read_input(input_path)
    heap_sort(arr)
    write_output(output_path, arr)

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

input.txt:

```
5
5 2 9 1 5
```

output.txt:

```
1 2 5 5 9
```

1. Импорт библиотек

```
import os
```

- **os**: Библиотека для работы с операционной системой, используется для работы с путями к файлам.

2. Функция `max_heapify`

```
def max_heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, n, largest)
```

- **`max_heapify`**: Эта функция поддерживает свойство кучи. Она принимает массив `arr`, размер кучи `n` и индекс `i`.
- **Переменные**:
  - `largest`: Изначально считается, что текущий элемент `i` является наибольшим.
  - `left` и `right`: Индексы левого и правого дочерних элементов.
- **Проверки**:
  - Если левый дочерний элемент больше текущего наибольшего, обновляем `largest`.
  - Аналогично проверяем правый дочерний элемент.
- **Обмен**: Если `largest` изменился, выполняется обмен элементов и рекурсивный вызов `max_heapify` для нового поддерева.

3. Функция `heap_sort`

```
def heap_sort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        max_heapify(arr, i, 0)
```

- **`heap_sort`**: Основная функция для сортировки массива.
- **Построение кучи**:
  - Цикл от `n // 2 - 1` до `0` для создания максимальной кучи. Это гарантирует, что все родительские элементы удовлетворяют свойству кучи.
- **Сортировка**:
  - Цикл от `n - 1` до `1` для извлечения элементов из кучи. Самый большой элемент (находится в корне) обменивается с последним элементом массива.

- После этого вызывается `max_heapify` для восстановления свойства кучи, но только для уменьшенного массива (без последнего элемента).

4. Функция `read_input`

```
def read_input(file_path):
    with open(file_path, 'r') as f:
        n = int(f.readline().strip())
        arr = list(map(int, f.readline().strip().split()))
    return arr
```

- **`read_input`**: Функция для чтения входных данных из файла.
- Открывается файл, считывается количество элементов `n` и сам массив, который преобразуется из строки в список целых чисел.

5. Функция `write_output`

```
def write_output(file_path, arr):
    with open(file_path, 'w') as f:
        f.write(' '.join(map(str, arr)) + '\n')
```

- **`write_output`**: Функция для записи отсортированного массива в выходной файл.
- Массив преобразуется в строку, где элементы разделены пробелами, и записывается в файл.

6. Основная функция `main`

```
def main():
    input_path = os.path.join('..', 'txtf', 'input.txt')
    output_path = os.path.join('..', 'txtf', 'output.txt')

    arr = read_input(input_path)
    heap_sort(arr)
    write_output(output_path, arr)
```

- **`main`**: Основная функция, которая выполняет следующие действия:
  - Определяет пути к входному и выходному файлам.
  - Читает массив из входного файла.
  - Вызывает функцию сортировки.
  - Записывает отсортированный массив в выходной файл.

7. Главный блок

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

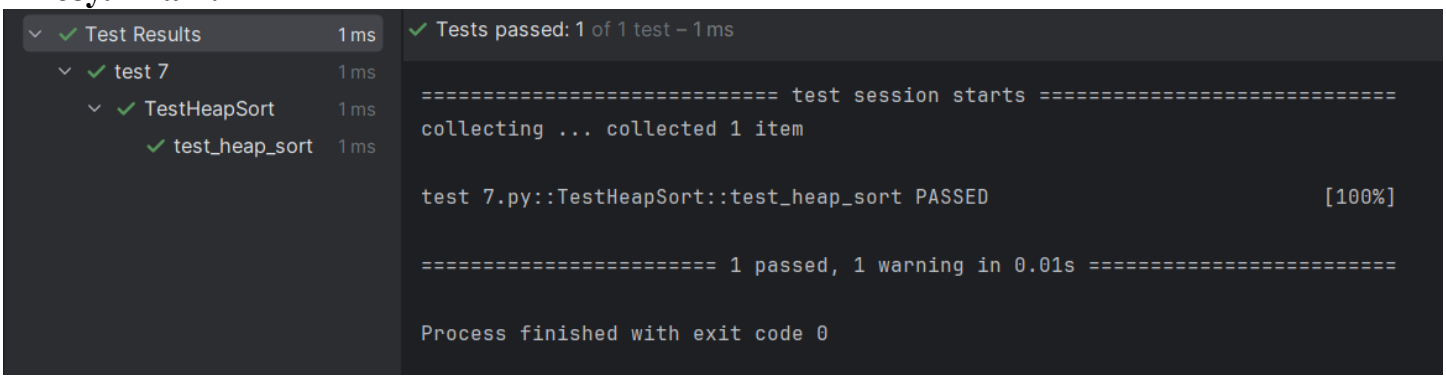
- Проверка, запущен ли скрипт напрямую. Если да, вызывается функция `main`.

Unittest для задание 7:

```
import unittest
import os
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from c7 import heap_sort, read_input, write_output

class TestHeapSort(unittest.TestCase):
```

```
    def test_heap_sort(self):
        # given
        input_data = "5\n5 2 9 1 5\n"
        expected_output = "1 2 5 5 9\n"

        # when
        with open(os.path.join('..', 'txtf', 'input.txt'), 'w') as f:
            f.write(input_data)

        # when
        arr = read_input(os.path.join('..', 'txtf', 'input.txt'))

        # when
        heap_sort(arr)

        # when
        write_output(os.path.join('..', 'txtf', 'output.txt'), arr)

        # when
        with open(os.path.join('..', 'txtf', 'output.txt'), 'r') as f:
            result = f.read().strip()

        # then
        self.assertEqual(result, expected_output.strip())

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

*** Результат :**



1. Импорт библиотек

```
import unittest
import os
import sys
```

- **unittest**: Стандартная библиотека Python для написания и выполнения тестов.
- **os**: Библиотека для работы с операционной системой, используется для работы с файловой системой.
- **sys**: Позволяет взаимодействовать с интерпретатором Python, в данном случае используется для изменения пути поиска модулей.

2. Изменение пути поиска модулей

```
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src')))
from c7 import heap_sort, read_input, write_output
```

- С помощью sys.path.insert добавляется путь к папке src в список путей, по которым Python ищет модули. Это позволяет импортировать функции heap_sort, read_input и write_output из модуля c7, который находится в папке src.

3. Класс `TestHeapSort`

```
class TestHeapSort(unittest.TestCase):
```

- Создается класс TestHeapSort, который наследует от unittest.TestCase. Это позволяет использовать все функции для тестирования, предоставляемые unittest.

4. Метод `test_heap_sort`

```
def test_heap_sort(self):
```

- Определяется метод test_heap_sort, который будет содержать тестовый сценарий для проверки работы функции сортировки.

Подготовка данных

```
# given
input_data = "5\n5 2 9 1 5\n"
expected_output = "1 2 5 5 9\n"
```

- **input_data**: Строка, представляющая входные данные для теста. В первой строке указано количество элементов, во второй — сами элементы массива.
- **expected_output**: Ожидаемый результат выполнения сортировки, который будет сравнен с фактическим результатом.

Запись входных данных в файл

```
with open(os.path.join('..', 'txtf', 'input.txt'), 'w') as f:
    f.write(input_data)
```

- Открывается файл input.txt для записи в режиме w (write). Путь к файлу создается с помощью os.path.join, чтобы обеспечить кросс-платформенность.
- В input.txt записываются входные данные.

Чтение данных из файла

```
arr = read_input(os.path.join('..', 'txtf', 'input.txt'))
```

- Вызывается функция read_input, которая считывает массив из файла input.txt.

Сортировка массива

```
heap_sort(arr)
```

- Вызывается функция heap_sort, которая сортирует массив.

Запись результатов в выходной файл

```
write_output(os.path.join('..', 'txtf', 'output.txt'), arr)
```

- Вызывается функция write_output, которая записывает отсортированный массив в файл output.txt.

Чтение результатов из выходного файла

```
with open(os.path.join('..', 'txtf', 'output.txt'), 'r') as f:
    result = f.read().strip()
```

- Открывается файл output.txt для чтения в режиме r (read).
- Содержимое файла читается и сохраняется в переменной result, при этом удаляются лишние пробелы с помощью strip().

Проверка результата

```
self.assertEqual(result, expected_output.strip())
```

- Используется метод assertEqual для сравнения фактического результата (result) с ожидаемым (expected_output). Если они не равны, тест не пройдет.

5. Главный блок

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

- Проверка, запущен ли скрипт напрямую. Если да, вызывается функция unittest.main(), которая запускает все тесты, определенные в классе.