

(САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №5
по курсу «Алгоритмы и структуры данных»
Тема: Деревья. Пирамида, пирамидальная сортировка.
Очередь с приоритетами.
Вариант 12

Выполнил:
Колпаков А.С.
К3139

Проверил:
Афанасьев А.В

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №3. Обработка сетевых пакетов	3
Задача №4. Построение пирамиды	6
Дополнительные задачи	10
Задача №1. Куча ли?	10
Задача №5. Планировщик заданий	12
Вывод	25

Задачи по варианту

Задача №3. Обработка 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.

Листинг кода:

```
import utils

def network_packets(buffer_size, packets):
    buffer = []
    start_times = []

    for arrival_time, processing_time in packets:
        while buffer and buffer[0] <= arrival_time:
            buffer.pop(0)
```

```

        if len(buffer) >= buffer_size:
            start_times.append(-1)
        else:
            if not buffer:
                start_time = arrival_time
            else:
                start_time = buffer[-1]

            start_times.append(start_time)

            buffer.append(start_time +
processing_time)

    return start_times

if __name__ == '__main__':
    data =
utils.read_data('lab5/task3/textf/input.txt')
    res = network_packets(data[0][0], data[1:])
    utils.print_task_data(3, data, res)
    utils.write_file("lab5/task3/textf/output.txt",
[res])

```

Текстовое объяснение решения:

1. Создаем пустой массив буфера и начального времени обработки пакетов.
2. Проходимся по значениям пакетов, также удаляем пакеты из буфера, которые уже были обработаны.
3. Если количество элементов в буфере больше максимального, то записываем -1 в результат,

4. В ином случае записываем в буфер начальное время исполнения + время на обработку пакета.

Результат работы кода на примерах из текста задачи:

```
input.txt U
itmo > algo > lab5 > task3

1  2  3
2  0  1
3  3  1
4 10  1
```

```
output.txt U
itmo > algo > lab5 > tas

1  0  3 10
```

```
Тест примера
Время работы: 0.0005062920099589974 секунд
Память: 0.005377769470214844 МБ
```

```
-----
Ran 1 test in 0.001s
```

```
OK
```

	Время выполнения	Затраты памяти
Пример из задачи	0.000506292009958997 4 секунд	0.005377769470214844 МБ

Вывод по задаче:

Программа успешно реализует алгоритм обработки сетевых пакетов.

Задача №4. Построение пирамиды

В этой задаче вы преобразуете массив целых чисел в пирамиду. Это важнейший шаг алгоритма сортировки под названием HeapSort. Гарантированное время работы в худшем случае составляет $O(n \log n)$, в отличие от *среднего* времени работы QuickSort, равного $O(n \log n)$. QuickSort обычно используется на практике, потому что обычно он быстрее, но HeapSort используется для внешней 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 массив 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 $O(n)$ перестановок. Обратите внимание, что в этой задаче вам нужно будет использовать min-heap вместо max-heap.

Листинг кода:

```
import utils

def build_pyramid(n, data):
    swaps = []

    def move_down(i):
```

```

        min_index = i
        left = 2 * i + 1
        right = 2 * i + 2

        if left < n and data[left] <
data[min_index]:
            min_index = left

        if right < n and data[right] <
data[min_index]:
            min_index = right

        if i != min_index:
            swaps.append(f'{i} {min_index}\n')
            data[i], data[min_index] =
data[min_index], data[i]
            move_down(min_index)

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

    swaps.insert(0, str(len(swaps)) + '\n')
    return swaps

if __name__ == '__main__':
    data =
utils.read_data('lab5/task4/textf/input.txt')
    res = build_pyramid(data[0], data[1])

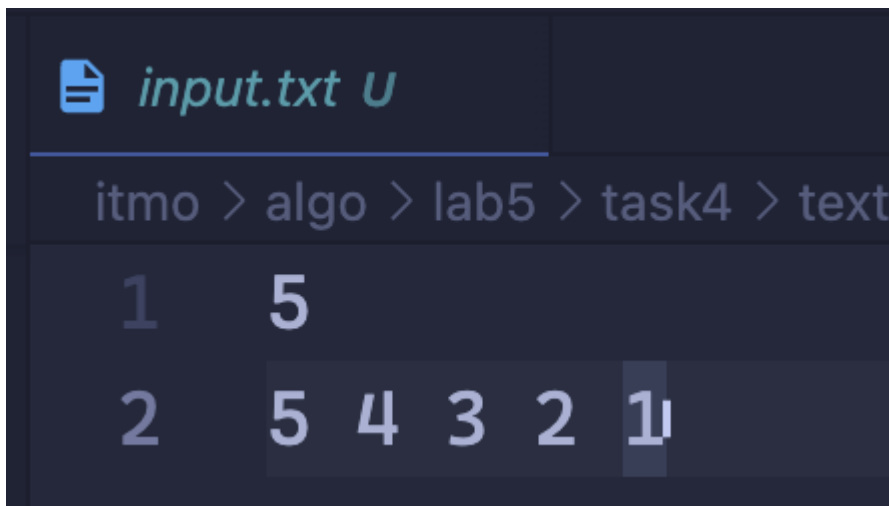
```

```
utils.print_task_data(4, data, res)
utils.write_file("lab5/task4/textf/output.txt",
res)
```

Текстовое объяснение решения:

1. Создаем массив перестановок.
2. Создаем функцию `move_down`, которая отвечает за перестановку элемента пирамиды с минимальным элементом из потомков. Также индексы перестановки записываются в результат.
3. В конце проходимся циклом по значениям массива и вызываем функцию `move_down`.
4. Добавляем количество элементов в `swar` и возвращаем результат.

Результат работы кода на примерах из текста задачи:




```
output.txt U
itmo > algo > lab5 > task4
1 3
2 1 4
3 0 1
4 1 3
```

```
Тест примера
Время работы: 0.00029983301647007465 секунд
Память: 0.0057315826416015625 МБ
.
-----
Ran 1 test in 0.001s
OK
```

	Время выполнения	Затраты памяти
Пример из задачи	0.00029983301647007465 секунд	0.0057315826416015625 МБ

Вывод по задаче:
Программа эффективно реализует алгоритм построения пирамиды из массива целых чисел.

Дополнительные задачи

Задача №1. 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.

Листинг кода:

```
import utils

def is_heap(n, array):
    for i in range(n):
        if 2 * i + 1 < n and array[i] > array[2 * i + 1]:
            return "NO"
        if 2 * i + 2 < n and array[i] > array[2 * i + 2]:
            return "NO"
    return "YES"

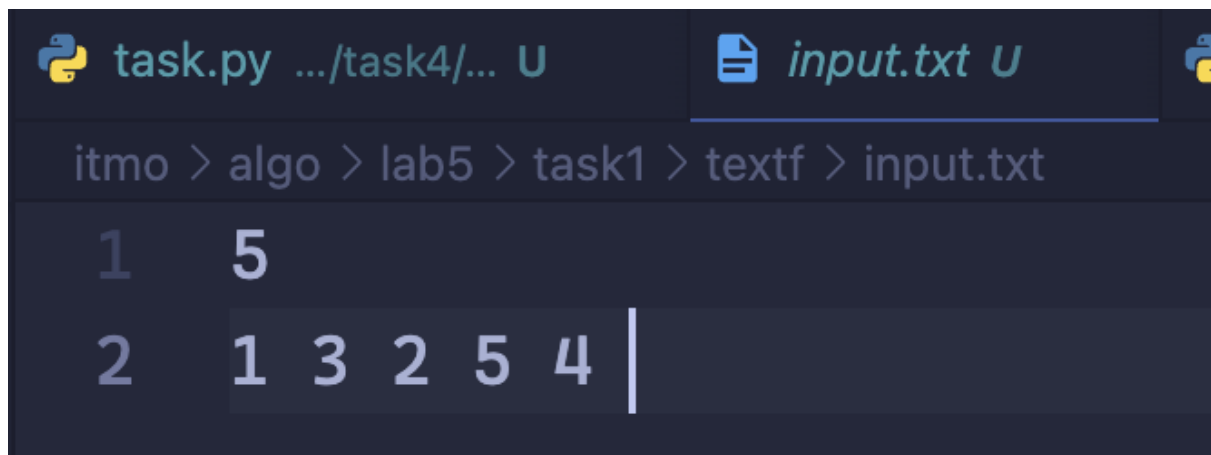
if __name__ == '__main__':
    data = utils.read_data('lab5/task1/textf/input.txt')
    res = is_heap(data[0], data[1])
    utils.print_task_data(1, data, res)
```

```
utils.write_file("lab5/task1/textf/output.txt",  
[res])
```

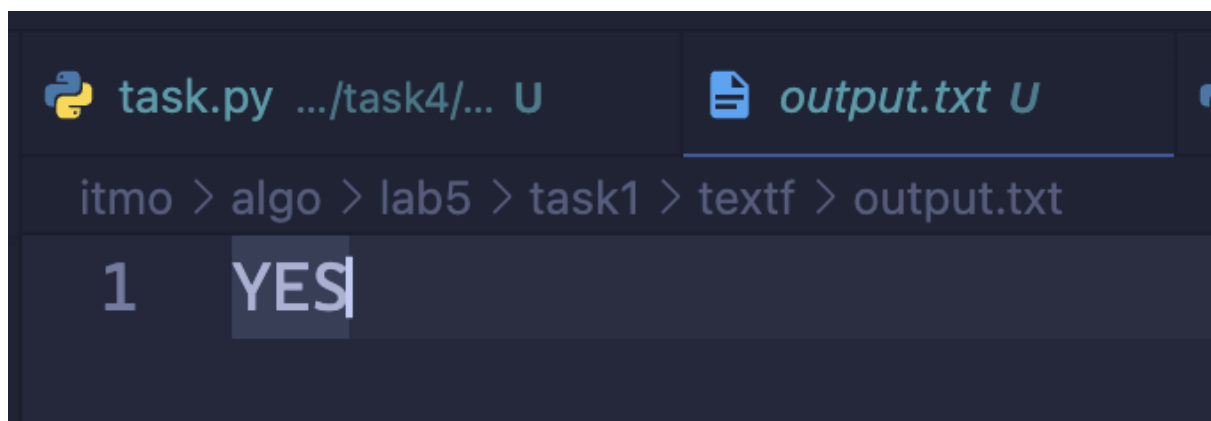
Текстовое объяснение решения:

1. Циклом проходимся по каждому элементу изначального массива.
2. Если какое то условие не выполняется, то выводим NO, в ином случае выводим YES.

Результат работы кода:



```
task.py .../task4/... U  input.txt U  
itmo > algo > lab5 > task1 > textf > input.txt  
1      5  
2      1 3 2 5 4
```



```
task.py .../task4/... U  output.txt U  
itmo > algo > lab5 > task1 > textf > output.txt  
1      YES
```

```
Тест примера  
Время работы: 0.0006839580019004643 секунд  
Память: 0.0052337646484375 МБ  
.  
-----  
Ran 1 test in 0.001s  
OK
```

	Время выполнения	Затраты памяти
Пример из задачи	0.0006839580019004643 секунд	0.0052337646484375 МБ

Вывод по задаче:

Программа успешно проверяет является ли заданный массив кучей.
Тестирование показало правильную работу алгоритма.

Задача №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 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.

Листинг кода:

```
import utils

class Heap:
    def __init__(self, n):
        self.heap = [(0, i) for i in range(n)]

    def push(self, time, thread_index):
```

```

        self.heap.append((time, thread_index))
        self.move_up(len(self.heap) - 1)

def pop(self):
    self.swap(0, len(self.heap) - 1)
    min_thread = self.heap.pop()
    self.move_down(0)
    return min_thread

def move_up(self, idx):
    while idx > 0:
        parent = (idx - 1) // 2
        if self.heap[idx] >= self.heap[parent]:
            break
        self.swap(idx, parent)
        idx = parent

def move_down(self, idx):
    size = len(self.heap)
    while 2 * idx + 1 < size:
        left = 2 * idx + 1
        right = 2 * idx + 2
        smallest = left
        if right < size and self.heap[right] <
self.heap[left]:
            smallest = right
        if self.heap[idx] <=
self.heap[smallest]:

```

```

        break

        self.swap(idx, smallest)
        idx = smallest

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j],
self.heap[i]

def task_manager(n, m, jobs):
    heap = Heap(n)
    result = []

    for i in range(m):
        current_time, thread_index = heap.pop()
        result.append(f'{thread_index}
{current_time}\n')
        heap.push(current_time + jobs[i],
thread_index)

    return result

if __name__ == '__main__':
    data =
utils.read_data('lab5/task5/textf/input.txt')
    res = task_manager(data[0][0], data[0][1],
data[1])
    utils.print_task_data(5, data, res)

```

```
utils.write_file("lab5/task5/textf/output.txt",  
res)
```

Текстовое объяснение решения:

1. Создаем класс Heap, в котором мы описываем логику работы кучи.
2. В классе Heap создаем функции push и pop, которые дают возможность добавить элемент в кучу и получить первый элемент из нее.
3. Также создаем функции move_up и move_down, которые проталкивают элемент вверх и вниз соответственно.
4. Задаем функцию swap, которая меняет два элемента местами.
5. В конце задаем функцию task_manager, которая проходится по каждому элементу массива заданий и добавляет в результат значения индекса потока для задания и время начала его выполнения.

Результат работы кода на примерах из текста задачи:



task.py U



input.txt U

itmo > algo > lab5 > task5 > textf > input

1 2 5

2 1 2 3 4 5



task.py U



output.txt U

itmo > algo > lab5 > task5 > textf > outp

1 0 0

2 1 0

3 0 1

4 1 2

5 0 4

Тест примера

Время работы: 0.0009032499947352335 секунд

Память: 0.0052032470703125 МБ

.

Ran 1 test in 0.001s

OK

	Время выполнения	Затраты памяти
Пример из задачи	0.000903249994735233 5 секунд	0.0052032470703125 МБ

Вывод по задаче:

Программа эффективно выполняет алгоритм планировщика заданий, который параллельно обрабатывает список заданий.

Вывод

В ходе лабораторной работы были изучены и реализованы различные структуры данных: деревья, двоичная куча (пирамида) и очередь с приоритетами. Эти структуры помогают эффективно управлять данными и решать задачи сортировки, поиска и упорядочивания. Также была рассмотрена пирамидальная сортировка (heapsort), которая обеспечивает сортировку за $O(n \log n)$ благодаря использованию свойств кучи. Работа помогла понять принципы работы этих структур и их применение в алгоритмах.