Санкт-Петербургский Национальный Исследовательский Университет
Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Лабораторная работа №4

Выполнили:

Бабаев Р.С.

Садовая А.Р.

Абдулов И.А.

Проверил:

Мусаев А.А.

Санкт-Петербург,

2022

СОДЕРЖАНИЕ

Стр.

ВВЕДЕНИЕ

Данная работа представляет собой отчет о выполненных заданиях:

1. Написать программу с функциями для быстрой сортировки и сортировки расческой. Использовать данные функции п программу как модуль в другой программе. Пользователь выбирает один из двух методов сортировки. Оценить время выполнения программы с помощью модуля timeit.
2. Изучить блочную и пирамидальную сортировку. Написать соответствующие программы.
3. Оцените достоинства, недостатки и сложность изученных методов сортировок.

1 Задание 1

Быстрая сортировка заключается в том, что массив делится на две части относительно опорного элемента. В одну часть помещаются все элементы, величина которых меньше значения опорного элемента, а в правую часть — элементы со значением больше опорного. Таким образом каждая из частей так же рекурсивно делится на две части, результатом сортировки является слияние списков, возвращаемых рекурсией.

```python
import random
def quick_sort(nums): # Быстрая сортировка
    if len(nums) <= 1:
        return nums

    q = nums[random.randint(0, len(nums)-1)]
    low = [i for i in nums if i < q]
    eq = [i for i in nums if i == q]
    high = [i for i in nums if i > q]
    return quick_sort(low) + eq + quick_sort(high)
```

Рисунок 1.1 — Код функции для быстрой сортировки

Сортировка расческой заключается в том, чтобы “устранить” элементы с небольшими значениями в конце массива, которые замедляют работу алгоритма. При этой сортировке сначала берется большое расстояние между сравниваемыми значениями, а потом оно сужается вплоть до минимального.

```python
def comb(nums): # Сортировка расческой
    step, swap = int(len(nums)/1.247), 1
    while step > 1 or swap > 0:
        swap, i = 0, 0
        while i + step < len(nums):
            if nums[i] > nums[i+step]:
                nums[i], nums[i+step] = nums[i+step], nums[i]
                swap += 1
            i += 1
        if step > 1:
            step = int(step/1.247)
    return nums
```

Рисунок 1.2 — Код функции для сортировки расческой

Используем две данные функции сортировки как модуль в другой программе.

```
import task1_1 as mod # Используем функции как модуль
```

Рисунок 1.3 — Код импорта функций из другой программы

Рассмотрим процесс работы программы, в которой пользователь выбирает разные методы сортировки.

```
[4, -3.9, -1, 0, -8, 2.4, 1, -14.01]
Выберите метод сортировки: быстрая (0) / расчёской (1)? 0
Quick sort: [-14.01, -8, -3.9, -1, 0, 1, 2.4, 4]
Timeit: 0.00021349999951780774
```

Рисунок 1.4 — Пример работы быстрой сортировки

```
[4, -3.9, -1, 0, -8, 2.4, 1, -14.01]
Выберите метод сортировки: быстрая (0) / расчёской (1)? 1
Comb sort: [-14.01, -8, -3.9, -1, 0, 1, 2.4, 4]
Timeit: 0.0001007920000120066 1
```

Рисунок 1.5 — Пример работы сортировки расческой

Вывод: быстрая сортировка является одним из наиболее эффективных методов упорядочивания данных. На больших массивах данных сортировка расческой менее эффективна, чем быстрая, но на маленьких, как и показал пример, работает в несколько раз быстрее.

2 Задание 2

Блочная сортировка заключается в том, чтобы распределить элементы между конечным числом отдельных блоков так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше, чем в предыдущем. Каждый блок затем сортируется отдельно. Отсортированные элементы помещаются обратно в массив.

```python
def bucketSort(array): # Блочная сортировка
    bucket = []
    for i in range(len(array)):
        bucket.append([])
    for j in array:
        index_b = int(10 * j)
        bucket[index_b].append(j)
    for i in range(len(array)):
        bucket[i] = sorted(bucket[i])
    k = 0
    for i in range(len(array)):
        for j in range(len(bucket[i])):
            array[k] = bucket[i][j]
            k += 1
    return array
```

Рисунок 2.1 — Код функции блочной сортировки

Пирамидальная сортировка является усовершенствованной сортировкой пузырьком, в которой элемент всплывает / тонет по путям бинарного дерева.

```python
def heapSort(arr): # Пирамидальная сортировка
    def heapify(arr, n, i): # Наибольший в вершину дерева
        largest = i
        l, r = 2*i+1, 2*i+2
        if l < n and arr[i] < arr[l]:
            largest = l
        if r < n and arr[largest] < arr[r]:
            largest = r
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)
    n = len(arr)
    for i in range(n//2, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Рисунок 2.2 — Код функции пирамидальной сортировки

3 Задание 3

Алгоритм быстрой сортировки является одним из лучших и наиболее эффективных алгоритмов сортировки, а его рекурсивная реализация очень проста. К недостаткам можно отнести неустойчивость и деградацию скорости в худшем из случаев входных данных. Сложность $O(n^2)$.

Сортировка расческой является довольно упрощённым алгоритмом упорядочивания, улучшает сортировку пузырьком. В лучшем случае алгоритм имеет сложность $O(n \log 2n)$, в худшем случае работает за экспоненциальное $O(n^2)$ время.

Блочный тип сортировки может обладать линейным $O(n)$ временем исполнения на удачных входных данных. В худшем случае будет иметь экспоненциальное $O(n^2)$ время работы. Алгоритм применим к элементам, которые равномерно распределены.

Пирамидальная сортировка в среднем работает очень быстро, но на почти отсортированных массивах работает столь же долго, как и на хаотических данных. Зато, алгоритм эффективен по памяти, так как сортирует элементы на месте. Сложность алгоритма $O(n \log n)$

ЗАКЛЮЧЕНИЕ

Таким образом, для каждой задачи была написана программа на языке программирования Python. Были изучены 4 алгоритма сортировки массива и рассмотрены их достоинства, недостатки и сложность. Для каждого алгоритма приведены программы.

Все программы можно найти на репозитории в GitHub [1].

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. GitHub [Электронный ресурс]: `https://github.com/estle/itmo-uni` (дата обращения 02.12.2022).