Санкт-Петербургский Национальный Исследовательский Университет

Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий и систем связи

Лабораторная работа №4

Вариант №1

Выполнил(и:)

Гусев Я.А.

Проверил

Мусаев А.А.

Санкт-Петербург,

2022

Задание 1

На рисунке 1 изображён написанный мной алгоритм сортировки расчёской (combSort).

```
def combsort(given):
    space = len(given) - 1
    while space >= 1:
        for j in range(len(given) - space):
            if given[j] > given[j + space]:
                given[j], given[j + space] = given[j + space], given[j]
        space = int(space // 1.247)
    return given
```

Рисунок 1 – Сортировка расчёской

На рисунке 2 изображён написанный мной алгоритм быстрой сортировки (quickSort)

```
from random import randint as r
def quicksort(given, start, end):
    if start >= end:
        return given
    i, j = start, end
    pivot = given[r(i, j)]
    while i <= j:
        while given[i] < pivot:
            i += 1
        while given[j] > pivot:
            j -= 1
        if i <= j:
            given[i], given[j] = given[j], given[i]
            i, j = i + 1, j - 1
    quicksort(given, start, j)
    quicksort(given, i, end)
    return given
```

Рисунок 2 – Быстрая сортировка

Далее я импортирую эти функции сортировки в другую программу, в которой можно самому выбрать, какой сортировкой воспользоваться (Рисунок 3).

```
from combSort import combsort
from quickSort import quicksort

ex = [-82, -96, -46, 97, 95, 86, 44, -93, 31, -33, -9, 96, 8, 56, 72, 51, -67, -12, 55,
      26, 29, 67, 80, -90, -11, -2, -43, -73, -73, -7, 47, -91, 45, -94, 88, 65, -67, 99, 95, -2, 17]
method = input('Выберите метод сортировки (quick | comb) ')
#print('Далее введите ваш массив для сортировки через пробел')
#given = list(map(int, input().split()))
given = ex
if method == 'quick':
    print(quicksort(given, 0, len(given) - 1))
else:
    print(combsort(given))
```

Рисунок 3 – Программа с двумя доступными видами сортировок

Импортирую библиотеки time и timeit (Рисунок 4) и с их помощью узнаю время выполнения обеих сортировок (Рисунок 5).

```
print(timeit.timeit(stmt='combsort([-82, -96, -46, 97, 95, 86, 44, -93, 31, -33, -9, 96, 8, 56, 72, 51, -67, -12, 55,'
                         ' 26, 29, 67, 80, -90, -11, -2, -43, -73, -73, -7, 47, -91, 45, -94, 88, 65, -67, 99, 95, -2, 17])',
                    setup='from combSort import combsort', timer=time.perf_counter, number=1, globals=globals()))
print(timeit.timeit(stmt='quicksort([-82, -96, -46, 97, 95, 86, 44, -93, 31, -33, -9, 96, 8, 56, 72, 51, -67, -12, 55,'
                         ' 26, 29, 67, 80, -90, -11, -2, -43, -73, -73, -7, 47, -91, 45, -94, 88, 65, -67, 99, 95, -2, 17], 0, 40)',
                    setup='from quickSort import quicksort', timer=time.perf_counter, number=1, globals=globals()))
```

Рисунок 4 – Измерение времени выполнения сортировок с помощью библиотек timeit и time

```
3.0499999999999278e-05
5.199999999999996e-05
```

Рисунок 5 – Время выполнения quickSort и combSort

Как мы видим, быстрая сортировка быстрее сортировки расчёской примерно в 1.7 раз.

Задание 2

На рисунке 6 изображён написанный мной алгоритм блочной сортировки (bucketSort).

```
def bucketsort(given):
    blocks = [[] for i in range((max(given) - min(given)) // 10)]
    negative_blocks = [[] for i in range((max(given) - min(given)) // 10)]
    for i in range(len(given)):
        if given[i] >= 0:
            blocks[given[i] // 10].append(given[i])
        else:
            negative_blocks[given[i] // 10].append(given[i])
    blocks = negative_blocks + blocks
    blocks = [sorted(elem) for elem in blocks]
    return [item for sublist in blocks for item in sublist]

ex = [-82, -96, -46, 97, 95, 86, 44, -93, 31, -33, -9, 96, 8, 56, 72, 51, -67, -12, 55,
      26, 29, 67, 80, -90, -11, -2, -43, -73, -73, -7, 47, -91, 45, -94, 88, 65, -67, 99, 95, -2, 17]
print(bucketsort(ex))
```

Рисунок 6 – Алгоритм блочной сортировки

Среднее время выполнения – 1.989999999999978e-05.

На рисунке 7 изображена пирамидальная сортировка (heapSort).

```python
def heapify(arr, n, i):
    largest = i  # Самый большой элем - в корне
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)
    for i in range(n, -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 7 – Пирамидальная сортировка

Время выполнения - 1.2599979527294636e-05

Задание 3

Для сравнения времени всех реализованных сортировок, я построил их график с помощью библиотек numpy, matplotlib и pandas (Рисунок 8).

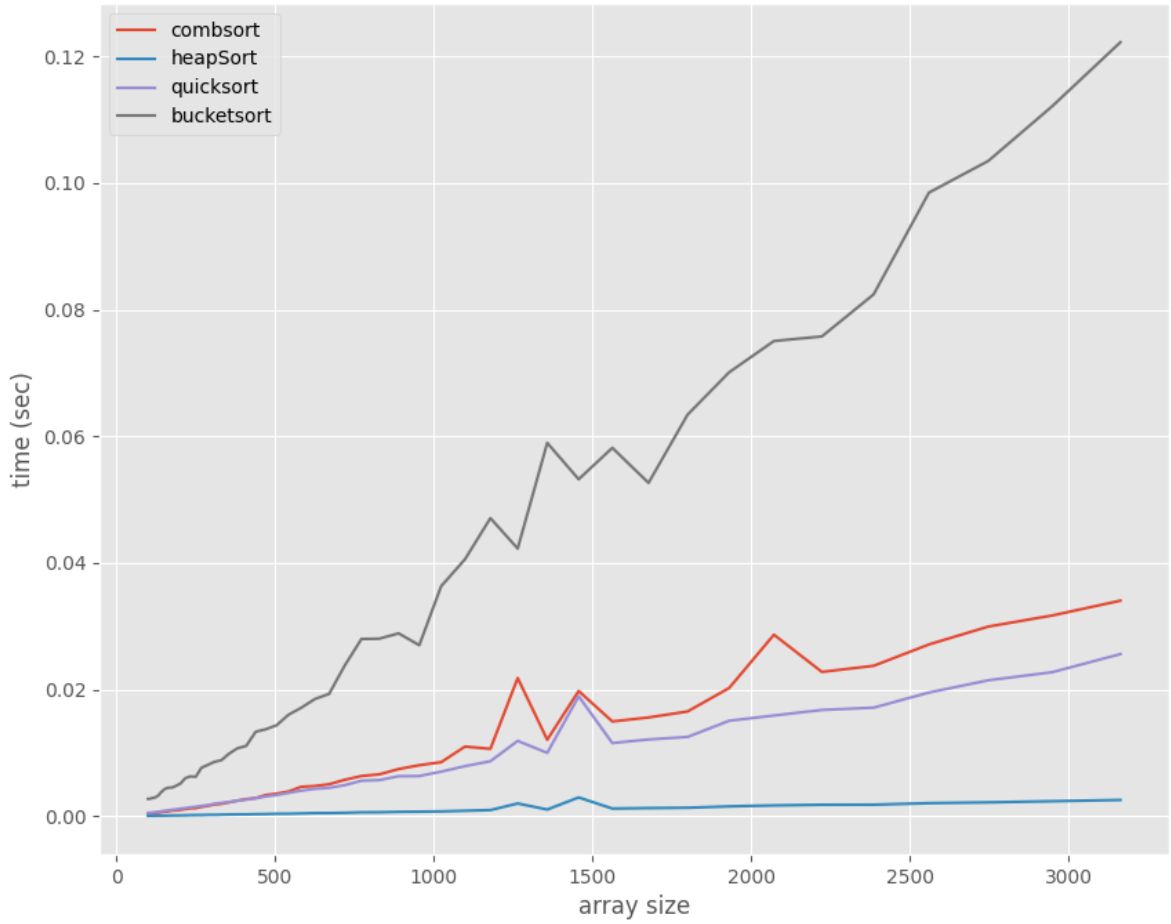

Рисунок 8 – Зависимость времени работы различных алгоритмов сортировок от размера массива входных данных

Как мы видим, quicksort и combsort находятся относительно на одном уровне, однако quicksort всё же быстрее. Самый быстрый алгоритм сортировки – heapsort. Bucketsort же в разы медленнее других алгоритмов, так он эффективен лишь на равномерно распределённых данных (там у него сложность O(n). Мы могли видеть это во втором задании, где блочная сортировка оказалась быстрее сортировки расчёской и быстрой сортировки, потому что входной массив был достаточно равномерно распределён. Для графика же я параллельно с увеличением размера массива увеличивал его

распределённость (Рисунок 9) (r – randint), чтобы показать несовершенство bucketsort.

```
b = [r(r(-50*j, -1000), r(1000, 50*j)) for i in range(j)]
```

Рисунок 9 – Генерация массива чисел для построения графиков

Общая оценка алгоритмов:

1) Сортировка расчёской. Улучшенная версия сортировки пузырьком. Сложность в лучшем случае – O(nlogn), в худшем сложность равна сложности сортировки пузырьком (O(n^2)).
2) Быстрая сортировка. Один из самых быстрых алгоритмов сортировки на массиве с неизвестными данными. Худшее время работы – O(n^2), среднее время – O(nlogn). Из минусов – на почти отсортированном массиве работает так же долго как и на неотсортированном. Из плюсов: короткий алгоритм, малый расход памяти.
3) Блочная сортировка. Как говорилось выше, хорошо работает на удачных входных данных – O(n). В остальных случаях O(n^2).
4) Пирамидальная сортировка. В основе лежит построение двоичного дерева. Сложность алгоритма в лучшем, худшем и среднем случае равна O(nlogn), поэтому он является крайне эффективным. Выделяемая память линейна. Из-за сложности алгоритма выигрыш получается только на больших n.