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

Факультет инфокоммуникационных технологий

Лабораторная работа №3
Вариант №1

Выполнил:
Бацанова Е. А.
Проверил
Мусаев А.А.

Санкт-Петербург,
2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....3
Задание 1.....4
Задание 2.....6
Задание 3.....9
ЗАКЛЮЧЕНИЕ.....11
СПИСОК ЛИТЕРАТУРЫ.....12
ПРИЛОЖЕНИЕ.....13

ВВЕДЕНИЕ

Целью данной работы является изучение сложности различных алгоритмов. В ходе лабораторной работы были реализованы следующие задачи:

1) Реализация алгоритма пузырьковой сортировки, сравнение сложности ее работы со сложностью встроенного метода *sort()*;
2) Реализация собственных алгоритмов, имеющих заданную сложность;
3) Сравнение сложности различных алгоритмов и построение графиков зависимости количества шагов от количества элементов для них.

Задание 1

Задание: Написать программу для пузырьковой сортировки. Оценить сложность данного метода. Сравнить с методом *sort()*.

Решение:

Пузырьковая сортировка работает путем последовательного сравнения пар соседних элементов списка. Если текущий элемент больше следующего, то они меняются местами. Этот процесс продолжается до тех пор, пока весь список не будет отсортирован.

В данной программе, сначала определяется длина списка *n*, затем два вложенных цикла выполняют итерации по всем элементам списка. При этом после *k*-той итерации внешнего цикла последние *k* элементов будут стоять в отсортированном порядке.

После завершения внешнего цикла, список будет отсортирован в порядке возрастания. Отсортированный список возвращается в качестве результата работы программы.

```python
def bubble_sort(mas):
    n = len(mas)
    for i in range(n):
        for j in range(0, n - i - 1):
            if mas[j] > mas[j+1]:
                mas[j], mas[j+1] = mas[j+1], mas[j]
    return mas


# example
mas = [64, 34, 25, 12, 22, 11, 90]
print("Array: ", mas)
sorted_mas = bubble_sort(mas)
print("Sorted array:", sorted_mas)
```

Рисунок 1 – Реализация алгоритма пузырьковой сортировки и примера для тестирования её работы

```
Array:  [64, 34, 25, 12, 22, 11, 90]
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

Рисунок 2 – Вывод программы для пузырьковой сортировки – исходный и отсортированный массивы

Сложность пузырьковой сортировки – $O(n^2)$.

Сложность сортировки методом *sort()* – $O(nlogn)$.

Пузырьковая сортировка значительно превосходит по сложности сортировку методом *sort()*. Особенно это заметно при больших объемах массива.

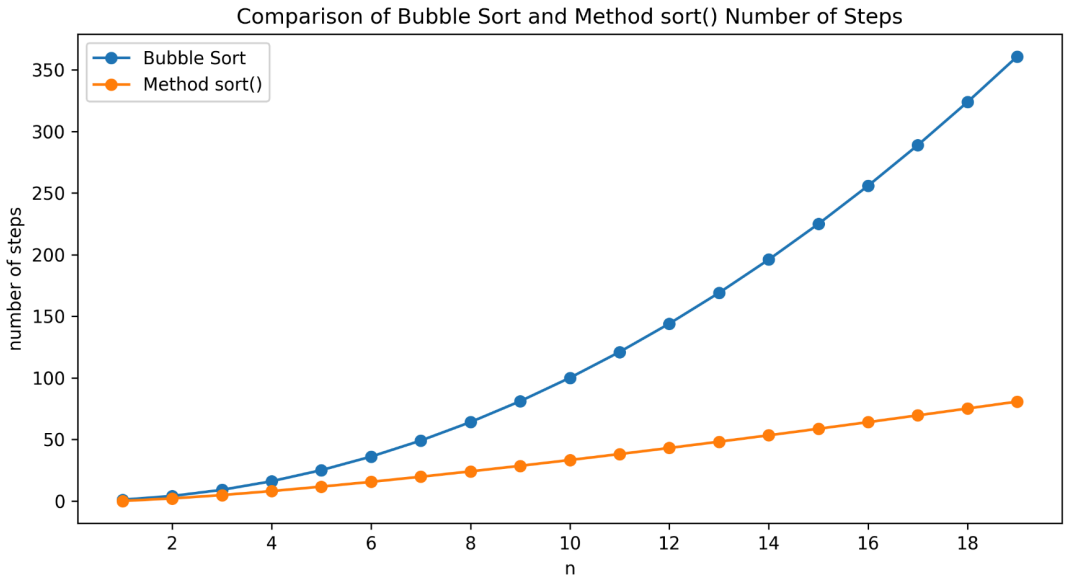

Рисунок 3 – Графики зависимости количества шагов от количества элементов в массиве для пузырьковой сортировки и метода *sort()*

Задание 2

Задание: Придумать и реализовать алгоритмы, имеющие сложность $O(3n)$, $O(nlogn)$, $O(n!)$, $O(n^3)$, $O(3logn)$

Решение:

1) Алгоритм со сложностью $O(3n)$

Примером такого алгоритма может быть суммирование всех элементов массива размера n, возведенных в степень 3 – всего внешний цикл совершает n итераций, внутри каждой из которых совершается еще 3 итерации внутреннего цикла. Поэтому общая сложность алгоритма равна $O(3n)$.

```
def o_3n(mas):
    summary = 0
    for i in mas:
        temp = 1
        for j in range(3):
            temp *= i
        summary += temp
    return summary
```

Рисунок 4 – Программа, имеющая сложность $O(3n)$ – сумма кубов элементов массива размера n

2) Алгоритм со сложностью $O(nlogn)$

Примером такого алгоритма может быть сортировка слиянием для массива размера n. На каждом уровне рекурсии массив делится пополам, пока не будет достигнута длина в 1 элемент. Затем каждый отсортированный подмассив объединяется с помощью функции *merge()*, на которую также требуется $O(n)$ операций.

Таким образом, на каждом уровне рекурсии требуется $O(n)$ операций, а таких уровней $logn$ из-за разделения массива пополам. Поэтому общая сложность алгоритма равна $O(nlogn)$.

```python
def merge_sort(mas):
    if len(mas) <= 1:
        return mas
    middle = len(mas) // 2
    left = merge_sort(mas[:middle])
    right = merge_sort(mas[middle:])
    return merge(left, right)


def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Рисунок 5 – Программа, имеющая сложность $O(nlogn)$ – сортировка слиянием массива размера n

3) Алгоритм со сложностью $O(n!)$

Примером такого алгоритма может быть алгоритм полного перебора всех перестановок n элементов. Сам алгоритм является рекурсивным и вызывает себя же внутри цикла для всех перестановок начиная с i-ой позиции. Сложность увеличивается факториально, так как на i-ой итерации задействуются оставшиеся n-i элементов.

```python
def generate_permutations(mas, n, i):
    if i == n:
        print(mas)
    else:
        for j in range(i, n):
            mas[i], mas[j] = mas[j], mas[i]
            generate_permutations(mas, n, i + 1)
            mas[i], mas[j] = mas[j], mas[i]
```

Рисунок 6 – Программа, имеющая сложность $O(n!)$ – поиск всех перестановок элементов массива размера n

4) Алгоритм со сложностью $O(n^3)$

Примером такого алгоритма может быть алгоритм перемножения двух матриц размерности $n \times n$. Сложность $O(n^3)$ достигается из-за наличия трёх вложенных циклов, каждый из которых выполняется n раз.

```python
def multiplying_matrices(matrix1, matrix2):
    n = len(matrix1)
    result = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                result[i][j] += matrix1[i][k] * matrix2[k][j]
    return result
```

Рисунок 7 – Программа, имеющая сложность $O(n^3)$ – перемножения двух матриц размерности $n \times n$

5) Алгоритм со сложностью $O(3logn)$

Примером такого алгоритма может быть алгоритм двоичного поиска 3 элементов в массиве с размерностью n. Для каждого из трёх элементов мы применяем бинарный поиск, имеющий сложность $O(logn)$.

```python
def binary_search(array, target1, traget2, target3):
    targets = [target1, traget2, target3]
    for target in targets:
        left, right = 0, len(array) - 1
        cnt = 0
        while left < right:
            cnt += 1
            m = (left + right) // 2
            if target > m:
                left = m + 1
            else:
                right = m
        return cnt
```

Рисунок 8 – Программа, имеющая сложность $O(3logn)$ – алгоритм двоичного поиска 3 элементов в массиве с размерностью n

Задание 3

Задание: Построить зависимость между количеством элементов и количеством шагов для алгоритмов со сложностью $O(1)$, $O(logn)$, $O(n^2)$, $O(2^n)$. Сравнить сложность данных алгоритмов.

Решение:

Для начала составим таблицу, в которой будут содержаться различные значения количества элементов и соответствующие им значения количества шагов для каждого из алгоритмов заданной сложности.

Таблица 1. Значения количества шагов при заданном количестве элементов для алгоритмов различной сложности

n	$O(1)$	$O(logn)$	$O(n^2)$	$O(logn)$
1	1	0	1	2
2	1	1	4	4
3	1	2	9	8
4	1	2	16	16
5	1	3	25	32
6	1	3	36	64
7	1	3	49	128
8	1	4	64	256
9	1	4	81	512
10	1	4	100	1024
11	1	4	121	2048
12	1	4	144	4096
13	1	4	169	8192
14	1	4	196	16384
15	1	4	225	32768

Из графиков и таблицы видно, что больше всего шагов при рассмотрении для достаточно большого количества элементов (больше 4) совершает алгоритм со сложностью $O(2^n)$, далее идут алгоритмы $O(n^2)$, $O(logn)$ и $O(1)$.

Значение для алгоритма со сложностью $O(1)$ не умаляя общности было взято равным единице. В любом случае данный алгоритм реализует постоянное количество шагов, которое не зависит от количества элементов, и рано или поздно остальные графики окажутся выше графика для $O(1)$.

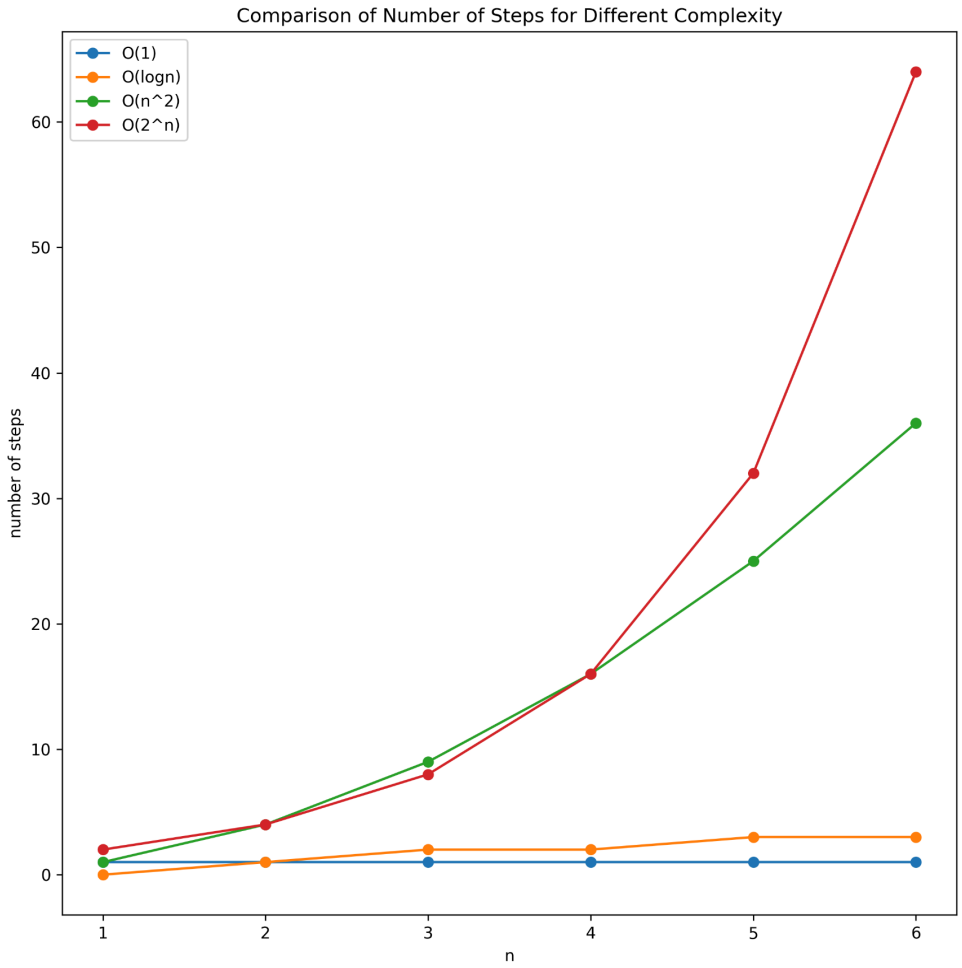

Рисунок 9 – Графики зависимости количеством шагов от количества элементов для алгоритмов со сложностью $O(1)$, $O(logn)$, $O(n^2)$, $O(2^n)$

ЗАКЛЮЧЕНИЕ

В ходе работы был рассмотрен алгоритм сортировки пузырьком и встроенный алгоритм сортировки *sort()*. Для каждой из этих сортировок был построен график зависимости количества шагов от количества элементов в массиве. С помощью данных графиков было наглядно продемонстрировано преимущество алгоритма сортировки *sort()*.

Также были придуманы и реализованы собственные функции имеющие заданную сложность.

В конце был проведен сравнительный анализ эффективности работы алгоритмов с различной сложностью в зависимости от объема данных. Полученные результаты позволяют сделать вывод о том, что выбор алгоритма с учетом его сложности играет ключевую роль в оптимизации времени выполнения программы.

В целом работа помогла понять как формируется сложность алгоритма, как она влияет на его работу и как сильно отличается количество шагов в зависимости от количества элементов для алгоритмов с различной сложностью.

СПИСОК ЛИТЕРАТУРЫ

1) Habr. Оценка сложности алгоритмов. [Электронный ресурс] – (Дата последнего обращения 19.05.2024).

ПРИЛОЖЕНИЕ

Для удобства все файлы выгружены на GitHub: https://github.com/kathykkKk/Algorithms-and-Data-Structures-ICT.git