Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)

Факультет Инфокоммуникационных технологий (ИКТ)

Образовательная программа Мобильные и сетевые технологии

О Т Ч Е Т

по Лабораторной работе 2

Дисциплина: Алгоритмы и структуры данных

Специальность: 09.03.03 Прикладная информатика

Проверил:

Мусаев А. А. __________

Дата: «__»_________2023 г.

Оценка: ________________

Выполнил:

Балцат К. И.,

студент группы К33401

Санкт-Петербург
2023

ВЫПОЛНЕНИЕ

1 Задача 1

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 построения зависимости между количеством элементов и количеством шагов для алгоритмов с разной сложностью, нужно рассмотреть каждый алгоритм отдельно.

Алгоритм со сложностью O(1):

Алгоритмы с константной сложностью имеют постоянное количество шагов, которое не зависит от количества элементов. Например, присваивание значения переменной - это операция с O(1) сложностью.

Алгоритм со сложностью O(logn):

Алгоритмы с логарифмической сложностью имеют количество шагов, пропорциональное логарифму от количества элементов. Такие алгоритмы характерны для работы с отсортированными данными. Например, бинарный поиск имеет O(logn) сложность.

Алгоритм со сложностью O(n^2):

Алгоритмы с квадратичной сложностью имеют количество шагов, пропорциональное квадрату количества элементов. Например, алгоритм сортировки пузырьком имеет O(n^2) сложность.

Алгоритм со сложностью O(2^n):

Алгоритмы с экспоненциальной сложностью имеют количество шагов, растущее экспоненциально с ростом количества элементов. Например, алгоритм перебора всех подмножеств множества имеет O(2^n) сложность.

Зависимость количества шагов от количества элементов для каждого из алгоритмов будет выглядеть следующим образом:

Алгоритм со сложностью O(1): Количество шагов не зависит от количества элементов, поэтому зависимость представляет собой горизонтальную прямую.

Алгоритм со сложностью O(logn): Количество шагов увеличивается логарифмически при увеличении количества элементов. Зависимость представляет собой кривую, которая начинается у некоторой точки на вертикальной оси и затем растет медленно и постепенно, стремясь к горизонтальной прямой.

Алгоритм со сложностью O(n^2): Количество шагов увеличивается квадратично при увеличении количества элементов. Зависимость представляет собой параболу, которая стремится к вершине, находящейся на нулевой точке на оси абсцисс, и затем растет очень быстро, становясь почти вертикальной.

Алгоритм со сложностью O(2^n): Количество шагов увеличивается экспоненциально при увеличении количества элементов. Зависимость представляет собой кривую, которая начинается у некоторой точки на вертикальной оси и затем растет очень быстро, почти вертикально.

Сравнение сложности данных алгоритмов показывает, что алгоритмы с константной и логарифмической сложностью наиболее эффективны при работе с большими объемами данных. Алгоритмы с квадратичной и экспоненциальной сложностью могут быть эффективны только при работе с небольшими объемами данных. При работе с большими объемами данных они могут быть слишком медленными и неэффективными, что может привести к проблемам с производительностью и временем выполнения. Поэтому для больших объемов данных лучше использовать алгоритмы с меньшей сложностью, а для маленьких объемов данных можно использовать любой алгоритм в зависимости от конкретной задачи.

2 Задача 2

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

Решение:

Сложность данного метода можно оценить как O(n^2), так как имеется двойной цикл по элементам массива. Это значит, что время выполнения алгоритма растет квадратично с ростом количества элементов.

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

if __name__ == '__main__':
    import random
    import timeit

    numbers = [random.randint(-1000, 1000) for i in range(1, 1000)]

    print(f"Sorted time: {timeit.timeit('sorted(numbers)', number=100, setup='from __main__ import numbers')}")
    print(f"Bubble_sort time: {timeit.timeit('bubble_sort(numbers)', number=100, setup='from __main__ import nu
```

Сравнение с методом sort() в Python, который использует алгоритм сортировки Тима, показывает, что пузырьковая сортировка обычно менее эффективна, чем метод sort(). Сложность алгоритма сортировки Тима в среднем составляет O(n log n), что является более быстрой и эффективной в сравнении с пузырьковой сортировкой. Кроме того, метод sort() в Python является реализацией сортировки на месте, т.е. изменяет исходный массив, в то время как пузырьковая сортировка создает новый массив, что также влияет на производительность. Поэтому в большинстве случаев лучше использовать метод sort() вместо пузырьковой сортировки, особенно при работе с большими объемами данных.

```
konstantinbaltsat@MacBook-Pro-Konstantin-3 HW2 % python3 task2.py
Sorted time: 0.009444869000000002
Bubble_sort time: 4.560763993
```

3 Задача 3

Придумать и реализовать алгоритмы, имеющие сложность O(3n), O(nlogn), O(n!), O(n3), O(3log(n))

Решение:

1. Алгоритм со сложностью O(3*n):

```
# Алгоритм со сложностью O(3*n):
def print_matrix(matrix):
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            for k in range(len(matrix[i][j])):
                print(matrix[i][j][k], end=" ")
            print()
        print()
# Пример использования
matrix = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
print_matrix(matrix)
```

2. Алгоритм со сложностью O(n*log(n)):

```python
# Алгоритм со сложностью O(n*log(n)):
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    left = merge_sort(left)
    right = merge_sort(right)

    return merge(left, right)

def merge(left, right):
    res = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            res.append(left[i])
            i += 1
        else:
            res.append(right[j])
            j += 1

    res += left[i:]
    res += right[j:]

    return res
```

3. Алгоритм со сложностью O(n!):

```python
# Алгоритм со сложностью O(n!):
def permute(nums):
    if len(nums) == 0:
        return []
    if len(nums) == 1:
        return [nums]
    res = []
    for i in range(len(nums)):
        for j in permute(nums[:i] + nums[i+1:]):
            res.append([nums[i]] + j)
    return res
```

4. Алгоритм со сложностью O(n^3):

```python
# Алгоритм со сложностью O(n^3):
def matrix_multiplication(a, b):
    m = len(a)
    n = len(b)
    p = len(b[0])

    res = [[0 for j in range(p)] for i in range(m)]

    for i in range(m):
        for j in range(p):
            for k in range(n):
                res[i][j] += a[i][k] * b[k][j]

    return res

# Пример использования
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
b = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]
c = matrix_multiplication(a, b)
print(c)
```

5. Алгоритм со сложностью O(3*log(n)):

```
# Алгоритм со сложностью O(3*log(n)):
def binary_search(arr, x):
    l = 0
    r = len(arr) - 1
    while l <= r:
        mid = l + (r - l) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            l = mid + 1
        else:
            r = mid - 1
    return -1

def log_loop(n):
    arr = [i for i in range(n)]
    for i in range(int(math.log(n, 2))):
        for j in range(int(math.log(n, 2))):
            x = binary_search(arr, j)
            if x != -1:
                print(i, j, x)

# Пример использования
log_loop(16)
```

ВЫВОД

Я изучил первую главу учебника, разобрался с оценкой сложностей алгоритмов и выполнил задания лабораторной.