Санкт-Петербургский Национальный Исследовательский
УниверситетИнформационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Лабораторная работа №2

Выполнили:
Стафеев И.А., Лапшина Ю.С., Килебе Нтангу Дьевина
Проверил
Мусаев А.А.

Санкт-Петербург,
2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: научиться сравнивать сложности различных алгоритмов, применять их оценку на практике.

Для достижения цели были поставлены следующие задачи:

- изучить алгоритмы со сложностью O(1), O(log n), O(n^2), O(2^n);
- построить графики, отображающие зависимость между количеством элементов и количеством шагов для алгоритмов с данными сложностями;
- написать программу для пузырьковой сортировки;
- произвести оценку сортировки методом пузырька и сравнить с методом sort();
- придумать и реализовать алгоритмы со сложностью O(3n), O(n*$\log n$), O(n!), O(n^3), O(3*$\log n$).

1 Построение зависимости между количеством элементом и количеством шагов для заданной сложностью

Чтобы построить графики, отображающие зависимость между количеством входных данных и числом шагов для выполнения алгоритма с такими входными данными были использованы библиотеки *numpy* и *matplotlib*. Соответствующий график показан на рисунке 1

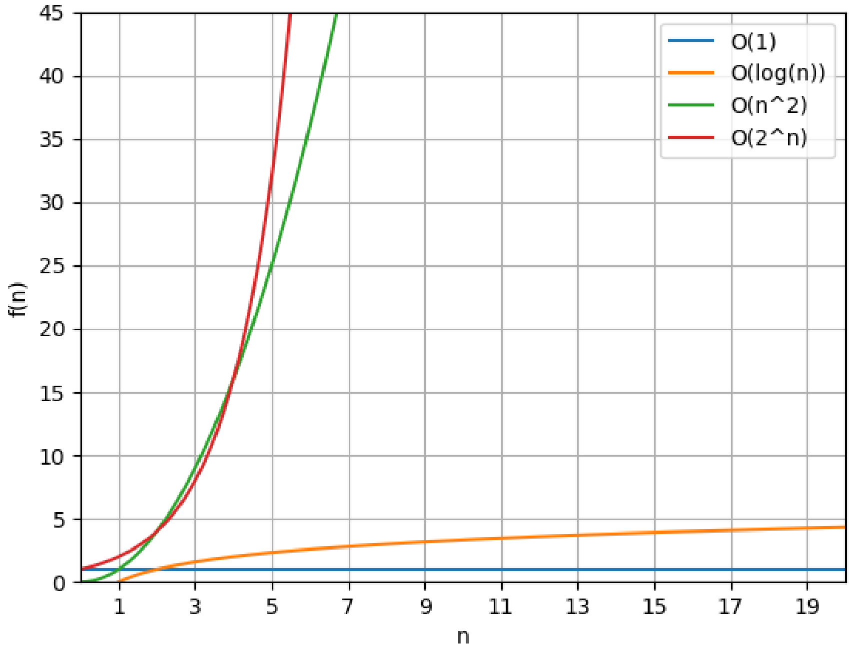

Рисунок 1 - Сравнение асимптотической сложности алгоритмов

Как можно заметить, представленные асимптотические сложности алгоритмов можно расположить в таком порядке по количеству требуемых шагов выполнения: $O(1)$, $O(logn)$, $O(n^2)$, $O(2^n)$, где алгоритм с первой сложностью выполняется наиболее быстро, а последний – наиболее долго, поскольку соответствующие графики становятся все более крутыми. Очевидно, что для решения каких-либо практических задач наиболее предпочтительным является использование алгоритма с наименьшей возможной для этой задачи асимптотической сложностью. Выполнение алгоритмов со сложностью $O(1)$ не зависит от объема входных данных, алгоритмы с $O(n)$ применяются для линейной обработки данных, алгоритмы с

$O(logn)$ применяются для задач с большим набором входных данных (таких, что даже линейная сложность не помогает быстро решить задачу). На практике также применяются алгоритмы с асимптотической сложностью $O(n^2)$ в тех случаях, когда алгоритмы с меньшей сложностью не существуют для данной задачи (например, некоторые операции над графами), и алгоритмы со сложностью $O(nlogn)$. Использование алгоритмов более высоких сложностей требует значительной вычислительной мощности и требует также много времени, поэтому вместо таких алгоритмов стараются использовать алгоритмы с меньшей сложностью, где это возможно.

2 Создание алгоритм для пузырьковой сортировки и оценка его сложности.

Идея метода сортировки пузырьком заключается в последовательном проходе по массиву и сравнении каждого элемента со следующим. Если текущий элемент больше следующего, то они меняются местами. Таким образом, на каждой итерации самый большой элемент "всплывает" на правильную позицию в конце массива. Процесс повторяется до тех пор, пока массив не будет полностью отсортирован.

Для реализации программы была написана функция sort_bubble, принимающая в качестве аргумента массив. Для обхода массива был использовал вложенный цикл, на каждой итерации которого происходит сравнение соседних элементов. Смена мест элементов реализована через множественное присваивание. Пример кода представлен на рисунке 2:

```
1 usage  dymonyx *
def sort_bubble(arr):
    for i in range(len(arr) - 1):
        for j in range(len(arr) - i - 1):
            if arr[j] < arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

if __name__ == '__main__':
    arr = list(map(float, input().split()))
    print(sort_bubble(arr))
```

Рисунок 2 – Алгоритм сортировки пузырьком

```
2 49 7 0 3 -5 3 6
[49.0, 7.0, 6.0, 3.0, 3.0, 2.0, 0.0, -5.0]
```

Рисунок 3 - пример работы метода сортировки пузырьком

Поскольку сортировка методом пузырька реализуется через вложенный список, её сложность можно оценить как O(n^2).

Встроенный метод sort() реализован через ЯП C и имеет сложность (в худшем случае) порядка O(n*log n).

На рисунке 4 представлены графики сложностей сортировки методом пузырька и встроенным методом sort(). Проанализировав графики, несложно заметить, что встроенный метод sort() имеет меньшую сложность, чем метод сортировки пузырьком, а значит, работает быстрее.

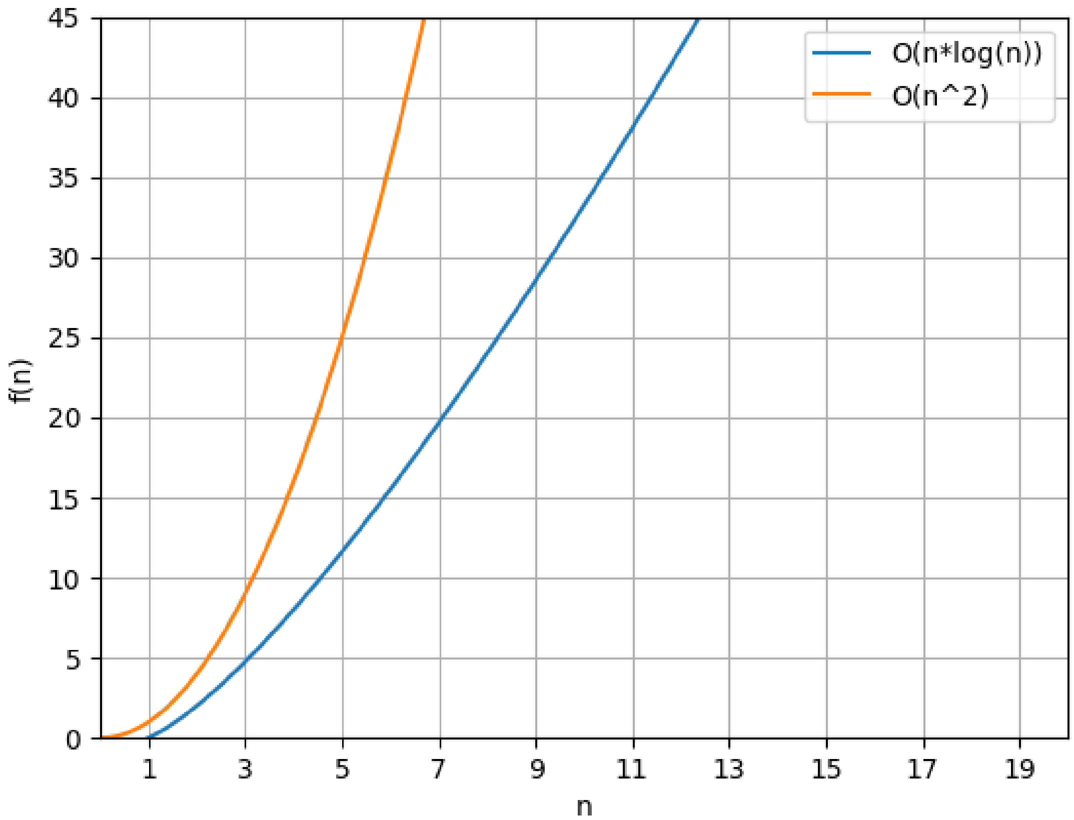

Рисунок 4 – Сравнение сложности двух алгоритмов сортировки

3 Реализация алгоритмов с заданной сложностью

Для каждого алгоритма с заданной сложностью была придумана небольшая задача, одним из решений которых является алгоритм с заданной сложностью (этот алгоритм может быть как оптимальным для придуманной задачи, так и не оптимальным). Код всех алгоритмов доступен по ссылке: https://github.com/staffeev/itmo_algos_labs/blob/main/Lab2/task3.py.

3.1 Алгоритм со сложностью O(3n)

Пусть имеется массив a длиной n, состоящих из целых чисел. Требуется определить такой максимальный индекс i (отсчет от 1), что префиксы длины i массива префиксных сумм, построенного для исходного массива, и массива префиксных сумм для развернутого массива отличаются не более чем в 3 позициях.

Для определения индекса i необходимо сначала построить массивы префиксных сумм для исходного и перевернутого массивов. Выполняющий эти операции код представлен на рисунке 5.

```
n = int(input())
a = list(map(int, input().split()))
pref = [0] * n
pref[0] = a[0]
for i in range(1, n):
    pref[i] = pref[i - 1] + a[i]
suf = [0] * n
suf[0] = a[-1]
for i in range(1, n):
    suf[i] = suf[i - 1] + a[n - i - 1]
```

Рисунок 5 - Вычисление массивов префиксных сумм

Построение массива префиксных сумм имеет асимптотическую сложность $O(n)$, поэтому последовательное выполнение двух построений массивов префиксных сумм имеет сложность $O(n+n) = O(2n)$.

Для нахождения индекса необходимо проитерироваться по парам соответствующих элементов массивов префиксных сумм, что можно сделать, например, используя функцию zip, и сравнить их. Если они равны или не равны и при этом количество отличий еще не равно 3-м, то к подсчитываемому индексу прибавляется один, иначе цикл прерывается. Сложность прохождения по массиву пар соответствующих элементов массивов имеет сложность $O(n)$, поэтому итоговая сложность алгоритма составляет $O(2n+n)=O(3n)$. Пример работы алгоритма представлен на рисунке 6, где в первой и второй строке написаны длина исходного массива и его элементы, в 3 и 4 строках написаны массивы префиксных сумм для исходного и перевернутого массива, и в 5 стоке написан искомый индекс.

```
PS D:\ProgrammingProjects\itmo_algos_
6
1 2 -1 4 8 1
[1, 3, 2, 6, 14, 15]
[1, 9, 13, 12, 14, 15]
6
PS D:\ProgrammingProjects\itmo_algos_
```

Рисунок 6 - Пример выполнения алгоритма со сложностью O(3n)

3.2 Алгоритм со сложностью O(nlogn)

На вход программе подается массив $bounds = [b_1, \cdots, b_n]$, состоящий из натуральных чисел. Пусть двоичное число s зависит от входного параметра k и определяется следующим образом: $s_i = \begin{cases} 1, & b_i \geq k \\ 0, & b_i < k \end{cases}$, где s_i-бит числа s. Требуется найти такое число, которое может быть создано названным способом, чтобы оно было наиболее близким ко входному числу p.

Для удобства обозначим построение двоичного числа от параметра k как $f(bounds, k)$. В новую переменную $sorted\_bounds$ занесем отсортированный по убыванию массив $bounds$. Тогда для любых sb_i и sb_j $(i < j)$ будет выполнено неравенство $f(bounds, sb_i) < f\left(bounds, sb_j\right)$. Тогда найти ближайшее к p число можно с помощью бинарного поиска: для числа sb_{mid} будем подсчитывать число $value = func(bounds, sb_{mid})$, и если $value \leq p$, то левую границу нужно сдвинуть на середину, иначе нужно сдвинуть правую границу. Параллельно будем сравнивать $|value - p|$ с минимальной разностью (изначально равной плюс бесконечности): если разность меньше минимальной, обновим минимальную разность и запомним соответствующее ей число $value$. Реализация бинарного поиска с построением числа представлена на рисунке 7.

```python
while left <= right:
    mid = (left + right) // 2
    bin_number = [1 if x >= sorted_bounds[mid] else 0 for x in bounds]
    value = int("".join(map(str, bin_number)), 2)
    if (cur_abs := abs(p - value)) < min_abs:
        min_abs = cur_abs
        closest_num = value
    if value <= p:
        left = mid + 1
    else:
        right = mid - 1

return closest_num
```

Рисунок 7 - Поиск наиболее близкого к p числа

Результатом выполнения алгоритма является наиболее близкое к p число, которое можно получить с помощью массива $bounds$ и параметра k. Сложность выполнения бинарного составляет $O(logn)$, сложность построения бинарного числа – $O(n)$, так как построение числа выполняется внутри тела цикла бинарного поиска, то результирующая сложность составляет $O(nlogn)$.

Пример выполнения программы представлен на рисунке 8 В первой строка написаны числа n и p, во второй – массив $bounds$, в 3-й – все числа, которые можно получить с помощью массива. В последней строке написано наиболее близкое к p число. Нетрудно заметить, что 26 действительно ближе всего к числу 27 среди чисел в 3 строке.

```
PS D:\ProgrammingProjects\itmo_
5 27
10 9 2 88 1

2 18 26 30 31

26
PS D:\ProgrammingProjects\itmo_
```

Рисунок 8 - Пример выполнения алгоритма со сложностью O(nlogn)

3.3 Алгоритм со сложностью O(n!)

Требуется подсчитать количество перестановок длины n таких, что сумма первого и последнего элементов делит сумму оставшихся.

Здесь будет описан наивный метод решения – полный перебор. Получить все перестановки длины n можно с помощью функции $permutations$ из модуля $itertools$. Сложность получения перестановок составляет $O(n!)$. Посчитаем сумму элементов перестановок (для всех одинаковую) по формуле: $s = n * (1 + n)/2$. Проитерируемся по всем перестановкам, и если $(s - p_1 - p_n)\%\,(p_1 + p_n) = 0$, где p_i – i –й элемент перестановки, то увеличим счетчик на 1. Так как проверка условия выполняется за $O(1)$, то результирующая сложность составляет $O(n! * 1) = O(n!)$.

На рисунке 9 показан пример работы алгоритмы для $n = 10$ и $n = 9$.

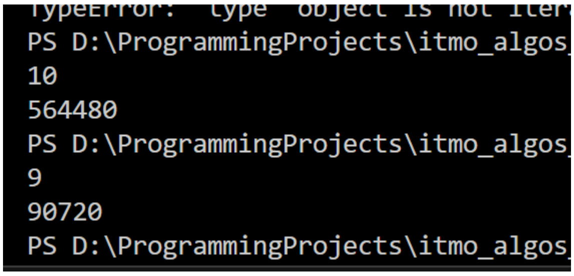

Рисунок 9 - Пример выполнения алгоритма со сложностью O(n!)

3.4 Алгоритм со сложностью O(n^3)

Пусть есть n точек на координатной плоскости. Сколько существует троек точек (p1, p2, p3), что сумма векторов, образованных точками (p1, p2) и (p2, p3) соответственно (где 1-я точка - начало вектора, 2-я - конец) является вектором, образующим тупой угол с положительным направлением оси Ox?

Пройдемся по всем тройкам точек, используя два вложенных цикла, и найдем координаты векторов, используя стандартные правила. Заметим, что если абсцисса результирующего вектора меньше 0, а ордината больше 0, то вектор образует тупой угол с положительным направлением оси Ox. На рисунке 10 представлена реализация этого алгоритма. Функция $\_make\_vector$ возвращает разность абсцисс и ординат конечной и начальной точек, т.е. координаты вектора.

```python
count = 0
for x in range(n):
    for y in range(n):
        for z in range(n):
            vec1 = _make_vector(points[x], points[y])
            vec2 = _make_vector(points[y], points[z])
            vec3 = (vec1[0] + vec2[0], vec1[1] + vec2[1])
            if vec3[0] < 0 and vec3[1] > 0:
                count += 1
return count
```

Рисунок 10 - Подсчет количества троек чисел по условию

На рисунке 11 показан пример выполнения алгоритма. В первых четырех строках написаны n и точки на плоскости, затем тройки точек, которые удовлетворяют условию и в конце их количество. Сложность построения вектором и проверки условия составляет $O(1)$, поэтому результирующая сложность составляет $O(n^3)$.

```
PS D:\ProgrammingProjects\itmo_algos_labs>
3
0 0
1 1
-10 10

((0, 0), (0, 0), (-10, 10))
((0, 0), (1, 1), (-10, 10))
((0, 0), (-10, 10), (-10, 10))
((1, 1), (0, 0), (-10, 10))
((1, 1), (1, 1), (-10, 10))
((1, 1), (-10, 10), (-10, 10))

6
PS D:\ProgrammingProjects\itmo_algos_labs>
```

Рисунок 11 - Пример выполнения программы со сложностью O(n^3)

3.5 Алгоритм со сложностью O(3logn)

Пусть даны три дискретные монотонные функции f, g, h, определенные на множестве $\{0,1,\cdots,n-1\}$. Найти для каждой функции такой x, что $f(x)$ наиболее близка к корню функции. Множества значений каждой функции задаются массивом вещественных чисел.

Найти x для функции можно за $O(logn)$ с помощью бинарного поиска, представленного на рисунке 12.

```
left, right = 0, n - 1
flag = False
while left  + 1 < right:
    mid = (left + right) // 2
    if _sign(s[left]) != _sign(s[mid]):
        flag = True
        right = mid
    else:
        left = mid
if not flag:
    return -1
return min(mid - 1, mid, mid + 1, key=lambda x: abs(s[x]))
```

Рисунок 12 - Нахождение наиболее близкого к корню дискретной функции числа

По сути, представленный алгоритм является разновидностью алгоритма бисекции. Если знак функции (реализуется через функцию $\_sign$) в точках $left$ и mid не совпадает, то правая граница сдвигается на середину, иначе левая граница сдвигается на середину. С помощью деления отрезка пополам удается найти x, которому соответствует наиболее близкое к корню функции значение функции в точке x. В ответ нужно взять $x \in \{mid - 1, mid, mid + 1\} \rightarrow |f(x)| = \min(|f(mid - 1)|, |f(mid)|, |f(mid + 1)|)$.

На рисунке 13 представлен пример выполнения этого алгоритма. Первая функция не имеет корней, вторая и третья имеют ближайшие к корню значения при $x = 3$ и $x = 1$ соответственно. Так как функций 3, то общая сложность выполнения будет равна $O(3logn)$.

```
algos_labs/Lab2/ta
5
1 2 3 4 5
10 5 3 -1 -2
-2 -1 1 2 3
(-1, 3, 1)
PS D:\ProgrammingP
```

Рисунок 13 - Пример выполнения алгоритма со сложностью O(3logn)

ЗАКЛЮЧЕНИЕ

Результатом выполнения лабораторной работы стало повышение навыков по написанию, оценке и сравнению алгоритмов с различной сложностью. В ходе работы были построены графики зависимости между количеством шагов и количеством элементов для алгоритмов различных сложностей. Были написаны программы, имеющие сложность O(3n), O(n*$\log n$), O(n!), O(n^3), O(3*$\log n$), O(n^2). Таким образом, полученные навыки позволят оптимизировать и упростить работу над рядом практических задач, в которых стоит выбор между алгоритмами с различной сложностью.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Ахо А.В., Хопкрофт Д.Э., Ульман Дж.Д. Алгоритмы и структуры данных [Электронный ресурс] – URL: https://vk.com/wall-114485185_260 (дата обращения 01.10.2023)
2. Python Sort Algorithms: A Comprehensive Guide [Электронный ресурс] – URL: https://ioflood.com/blog/python-sort-algorithms/ (дата обращения 02.10.2023)
3. Репозиторий на Github с выполненной работой [Электронный ресурс] – URL: https://github.com/staffeev/itmo_algos_labs (дата обращения 03.10.2023)
4. Википедия. Метод бисекции: [Электронный ресурс] – URL: https://ru.m.wikipedia.org/wiki/Метод_бисекции . (Дата обращения 02.10.2023)
5. Codility_. Prefix sums: [Электронный ресурс]: электронная книга. URL: https://codility.com/media/train/3-PrefixSums.pdf. (Дата обращения 02.10.2023)