Санкт-Петербургский Государственный Технический Университет
(Технологический институт)

Кафедра системного анализа и информационных технологий

Лабораторная работа №3

Выполнили:

Лелякин А. А., Люцай В.Н.

Проверил

Мусаев А.А.

Санкт-Петербург,

2022

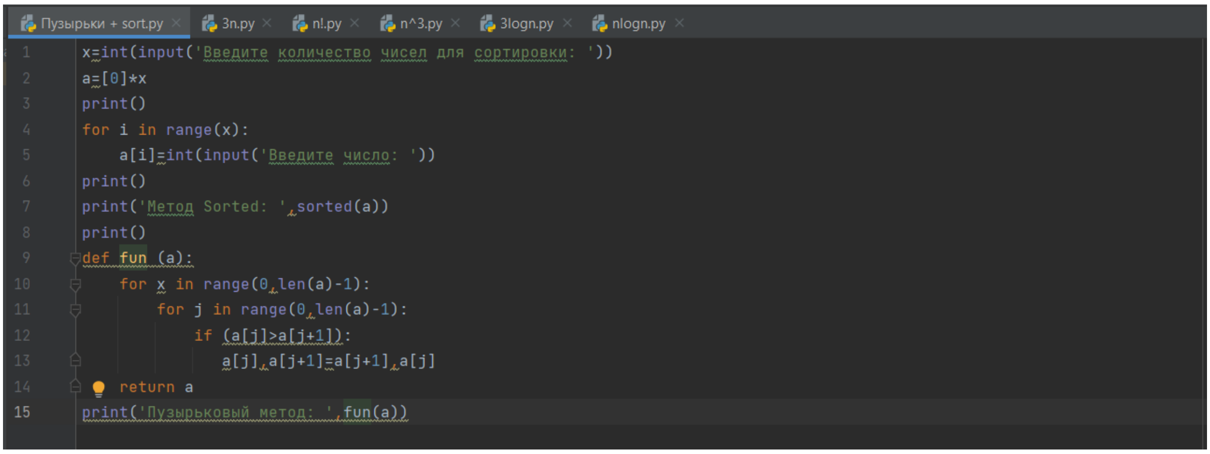
Ход работы

Задание 1

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

Выполнение:

Мы написали программу для сортировки методом “пузырька”. Код программы представлен ниже (рис.1).

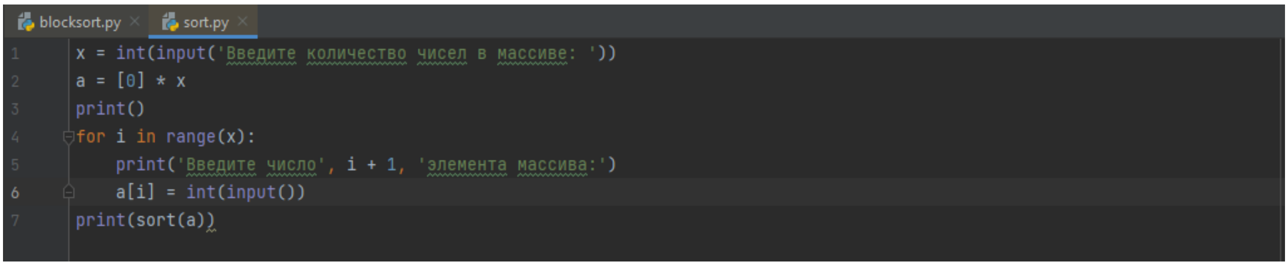


```
x=int(input('Введите количество чисел для сортировки: '))
a=[0]*x
print()
for i in range(x):
    a[i]=int(input('Введите число: '))
print()
print('Метод Sorted: ',sorted(a))
print()
def fun (a):
    for x in range(0,len(a)-1):
        for j in range(0,len(a)-1):
            if (a[j]>a[j+1]):
                a[j],a[j+1]=a[j+1],a[j]
    return a
print('Пузырьковый метод: ',fun(a))
```

рис.1

Данная программа сортирует заданный массив по возрастанию методом “пузырька”

В языке программирования Python существует метод “sort()”. Этот метод также позволяет выполнить сортировку чисел в массиве. Пример программы с использованием метода “sort()” приведен ниже (рис. 2).



```
x = int(input('Введите количество чисел в массиве: '))
a = [0] * x
print()
for i in range(x):
    print('Введите число', i + 1, 'элемента массива:')
    a[i] = int(input())
print(sort(a))
```

рис.2

Мы имеем две программы, которые выполняют сортировку массива, но выполняют они эту операцию разными способами. Нам необходимо сравнить их.

Сложность алгоритма сортировки методом “пузырька” можно оценить с помощью концепции **Big O**. Данный метод сортировки содержит в себе три цикла **for**, один из которых вложен в другой. Итого мы имеем сложность **O(n + n2)**. Однако **n** можно пренебречь, потому что это не что иное, как, так называемая, “ненужная сложность”. Соответственно, окончательно сложность нашей программы можно охарактеризовать как **O(n2)**. Тогда как метод sort имеет сложность **O(nlog(n))**.

Задание 2

“Придумать и реализовать алгоритмы, имеющие сложность O(3n), O(n log n), O(n!), O(n^3), O(3log(n))”

Выполнение:

Мы написали программы, имеющие сложность **O(3n)** (рис.3), **O(n log n)** (рис.4,5), **O(n!)** (рис.6), **O(n^3)** (рис.7), **O(3log(n))** (рис.8).

Пузырьки + sort.py | 3n.py | n!.py | n^3.py | 3logn.py | nlogn.py

```python
n = int(input('Введите количество строк первой матрицы: '))
a = []
for i in range(n):
    a.append(list(map(int, input('Введите элементы строки: ').split())))
print()
print('Первая матрица: ', a)
print()
n = int(input('Введите количество строк второй матрицы: '))
b = []
for i in range(n):
    a.append(list(map(int, input('Введите элементы строки: ').split())))
print()
print('Вторая матрица: ', b)
print()
n = int(input('Введите количество строк третьей матрицы: '))
c = []
for i in range(n):
    a.append(list(map(int, input('Введите элементы строки: ').split())))
print()
print('Третья матрица: ', c)
print()
```

рис.3

Данная программа создает **n** количество элементов для каждого из трех массивов.

```python
def m(l, r):
    res = []
    i = j = 0
    llen, rlen = len(l), len(r)
    for K in range(llen + rlen):
        if i < llen and j < rlen:
            if l[i] <= r[j]:
                res.append(l[i])
                i += 1
            else:
                res.append(r[j])
                j += 1
        elif i == llen:
            res.append(r[j])
            j += 1
        elif j == rlen:
            res.append(l[i])
            i += 1
    return res
def ms(nums):
    if len(nums) <= 1:
        return nums
    mid = len(nums) // 2
    l = ms(nums[:mid])
    r = ms(nums[mid:])
    return m(l, r)
```

рис.4

```python
n=int(input('Введите кол-во чисел в массиве: '))
a=[]
f=1
print()
for i in range (n):
    print('Введите элемент', f ,'массива:')
    a.append(int(input()))
    f+=1
print(ms(a))
```

рис.5

Merge sort (сортировка слиянием)

```python
def fac(n):
    if n == 0:
        return 1
    return fac(n - 1) * n
f = int(input('Введите число: '))
print('Факториал введенного числа: ',fac(f))
```

рис.6

Программа вычисляет факториал для каждого числа в заданном массиве с помощью рекурсии.

Пузырьки + sort.py 3n.py n!.py n^3.py 3logn.py nlogn.py

```python
n = int(input('Введите положительное нечетное число делящееся на 5 без остатка: '))
if n > 0:
    if n % 2 != 0:
        if n % 5 == 0:
            print('Число удовлетворяет условиям')
        else:
            print('Число не удовлетворяет условиям')
    else:
        print('Число не удовлетворяет условиям')
else:
    print('Число не удовлетворяет условиям')
```

рис.7

Эта программа проверяет цифру на удовлетворение заданным условиям.

Пузырьки + sort.py 3n.py n!.py n^3.py 3logn.py nlogn.py

```python
n=int(input('Введите кол-во чисел в массиве: '))
a=[]
f=1
print()
for i in range (n):
    print('Введите элемент', f,'массива:')
    a.append(int(input()))
    f+=1
print('Задан массив:')
print(sorted(a)); print()
g=int(input('Введите число для нахождения его в массиве: '))
print()
m = len(b) // 2
l = 0
h = len(b) - 1
q = 0
while b[m] != g and l <= h:
    if g > b[m]:
        l = m + 1
    else:
        h = m - 1
    m = (l + h) // 2
    q += 1
print()
if l > h:
    print('Числа', g,'нет в массиве')
    print('Количество шагов для нахождния числа', g, ':', q)
else:
    print('Число', g,'есть в массиве')
```

рис.8

Программа трижды выполняет бинарный поиск

Задание 3

“Построить зависимость между количеством элементом и количеством шагов для алгоритмов со сложностью O(1), O(log n), O(n^2), O(2^n). Сравнить сложность данных алгоритмов.”

Выполнение:

Для наглядного понимания зависимости между количеством элементов и количеством шагов для алгоритмов со сложностью **O(1)** (рис.9), **O(log n)** (рис.10), **O(n^2)** (рис.11), **O(2^n)** (рис.12) мы построили графики. Все графики приведены ниже.

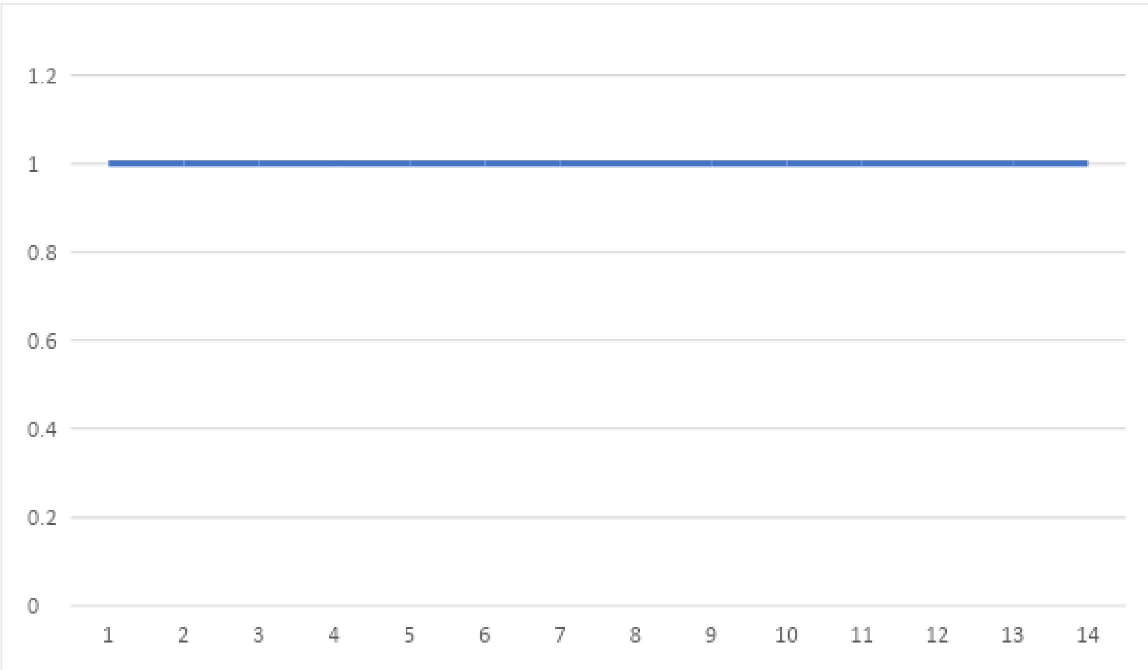


рис.9

В сложности **O(1)** количество шагов не зависит от количества переменных, это означает то, что при изменении количества переменных время выполнения программы никак не изменится.

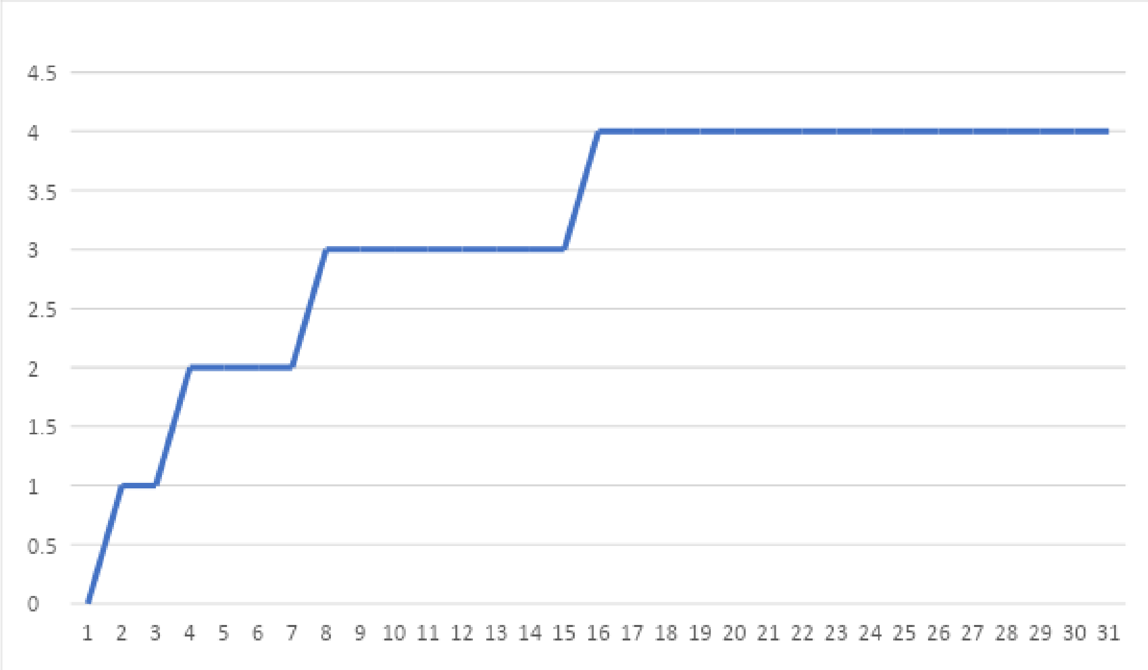


рис.10

В сложности **O(log n)** происходит отрицательное ускорение функции, следовательно данная сложность выполнения алгоритма самая оптимизированная, особенно, когда речь идет о большом количестве n.

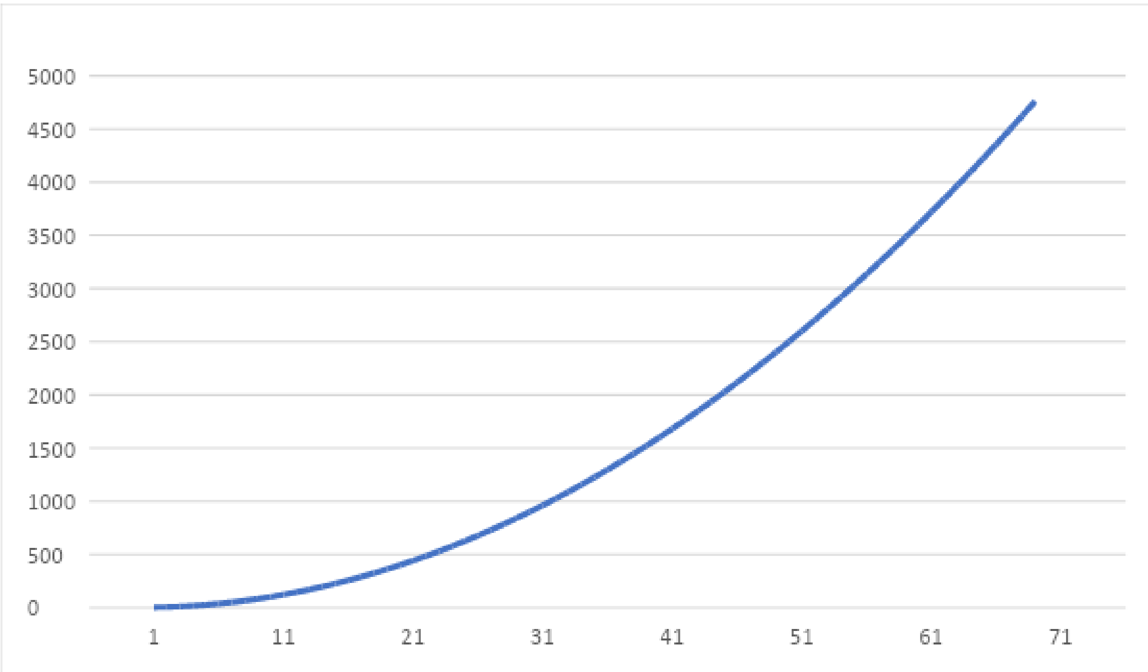


рис.11

В сложности **O(n^2)** количество шагов увеличивается полиномиально.

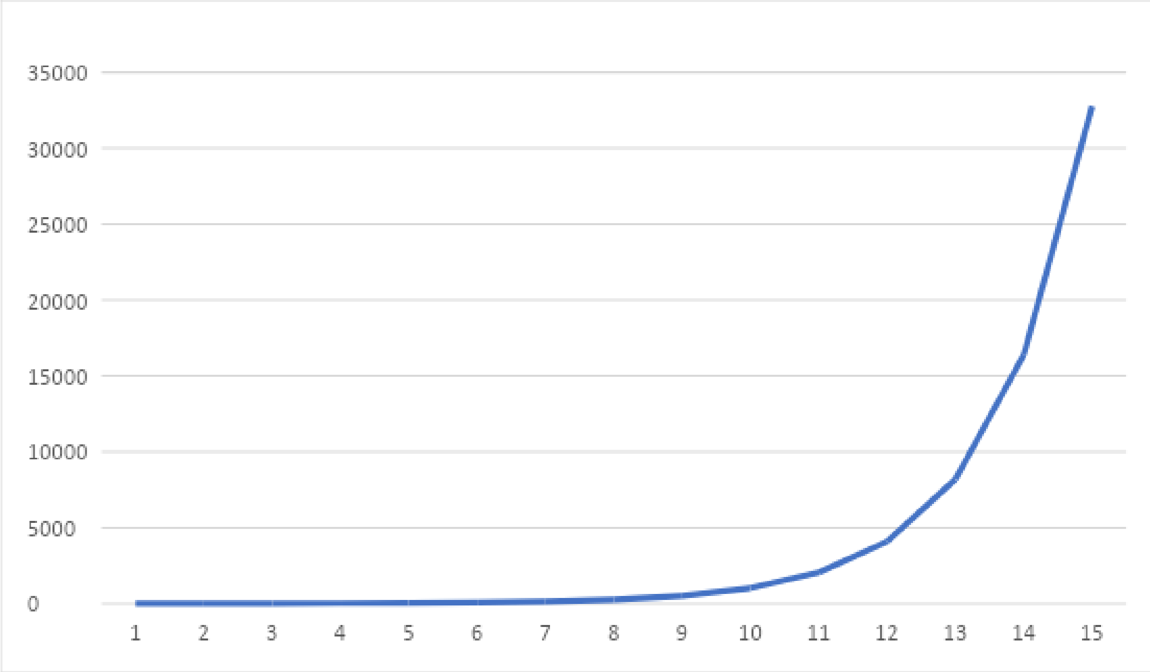


рис.12

В сложности **O(2^n)** количество шагов увеличивается экспоненциально.

Сравнение: Отталкиваясь от оценки всех вышеперечисленных сложностей, можно сказать, что **худшей** сложностью из приведенных является **O(2^n)**, а **лучшей** сложностью **O(log n)**.