

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ  
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ  
ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ  
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе № 3  
по курсу «Алгоритмы и структуры данных»

Тема: Быстрая сортировка, сортировки за  
линейное время.  
Вариант 23

Выполнил:  
Хабиби Ясер  
Группа К3140

Проверил:  
Афанасьев А. В.

Санкт-Петербург  
2024 г.

## Введение

Лабораторная работа посвящена быстрой сортировке и различным вариантам её применения. В ходе работы были изучены различные способы использования алгоритма быстрой сортировки и алгоритмов сортировки за линейное время.

Задачи по варианту: 1, 3 и 6, также были взяты и решены задания 2 и 5.

### Задача 1: Улучшение Quick sort

В данном задании был проанализирован алгоритм быстрой сортировки, после чего он был неоднократно улучшен путём добавления выбора случайного pivot-элемента и деления исходного массива не на две, а на три части.

### Код программы

```
from Lab3.utils import *
import random


def partition(array, left, right):  2 usages
    pivot_value = array[left]
    left_pointer = left
    for current_pointer in range(left + 1, right):
        if array[current_pointer] < pivot_value:
            left_pointer += 1
            array[left_pointer], array[current_pointer] = array[current_pointer], array[left_pointer]
    array[left], array[left_pointer] = array[left_pointer], array[left]
    return left_pointer


def quick_sort(array, left=0, right=None):
    if len(array) <= 1:
        return array
    if right is None:
        right = len(array)

    if left < right:
        pivot = (left + right) // 2
        array[pivot], array[left] = array[left], array[pivot]
        pivot = partition(array, left, right)
        random_quick_sort(array, left, pivot)
        random_quick_sort(array, pivot + 1, right)
    return array
```

```

def random_quick_sort(array, left=0, right=None):  5 usages
    if len(array) <= 1:
        return array
    if right is None:
        right = len(array)

    if left < right:
        pivot = random.randint(left, right - 1)
        array[pivot], array[left] = array[left], array[pivot]
        pivot_index = partition(array, left, right)
        random_quick_sort(array, left, pivot_index)
        random_quick_sort(array, pivot_index + 1, right)
    return array


def partition3(array, left, right):  1 usage
    pivot_value = array[left]
    left_pointer = left
    right_pointer = right - 1
    current_pointer = left + 1
    while current_pointer <= right_pointer:
        if array[current_pointer] < pivot_value:
            array[left_pointer], array[current_pointer] = array[current_pointer], array[left_pointer]
            left_pointer += 1
            current_pointer += 1
        elif array[current_pointer] > pivot_value:
            array[right_pointer], array[current_pointer] = array[current_pointer], array[right_pointer]
            right_pointer -= 1
        else:
            current_pointer += 1
    return left_pointer, right_pointer

```

```

def random_quick_sort3(array, left=0, right=None):  4 usages
    if len(array) <= 1:
        return array
    if right is None:
        right = len(array)

    if left < right:
        pivot = random.randint(left, right - 1)
        array[pivot], array[left] = array[left], array[pivot]
        middle_left, middle_right = partition3(array, left, right)
        random_quick_sort3(array, left, middle_left)
        random_quick_sort3(array, middle_right + 1, right)
    return array

qsort = random_quick_sort3

def qsort_plus_main():  1 usage
    n, arr = file_read_size_int_array()
    sorted_array = qsort(arr)
    file_write(sorted_array)


if __name__ == '__main__':
    measure_performance(qsort_plus_main)

```

## Тесты и анализ

Алгоритм улучшенной быстрой сортировки был протестирован на пустом массиве, малом массиве, большом массиве случайных элементов и большом массиве, состоящем из множества копий немногочисленных различных элементов, также было произведено сравнение с исходным алгоритмом. После анализа выяснилось, что улучшенный алгоритм действительно работает в разы быстрей исходного и решает все задачи без ошибок.

```
import unittest
import random
from Lab3.task1.src.qsort_plus import *
from Lab3.utils import *

class TestScarecrowSort(unittest.TestCase):
    testlist_empty = []
    testlist_basic = [2, 3, 9, 2, 2]
    testlist_random = [random.randint(-10 ** 9, 10 ** 9) for _ in range(10000)]
    testlist_few_different = [random.randint(-10 ** 9, 10 ** 9) for _ in range(20)] * 500

    def check_sorting(self, testlist): 4 usages
        result, elapsed_time, peak_memory_megabytes = measure_performance(qsort, *args: testlist)
        self.assertEqual(result, sorted(testlist))

    def test_should_sort_empty(self):
        self.check_sorting(self.testlist_empty)

    def test_should_sort_basic(self):
        self.check_sorting(self.testlist_basic)

    def test_should_sort_random(self):
        self.check_sorting(self.testlist_random)

    def test_should_sort_few_different(self):
        self.check_sorting(self.testlist_few_different)

    def test_should_compare(self):
        result, elapsed_time, peak_memory_megabytes = (
            measure_performance(random_quick_sort, *args: self.testlist_few_different))
        result3, elapsed_time3, peak_memory_megabytes3 = (
            measure_performance(random_quick_sort3, *args: self.testlist_few_different))
        self.assertGreater(elapsed_time, elapsed_time3)

if __name__ == '__main__':
    unittest.main()
```

## Задача 2: Анти-quick sort

Хотя QuickSort является очень быстрой сортировкой в среднем, существуют тесты, на которых она работает очень долго. Время работы алгоритма было оценено числом сравнений с элементами массива (то есть, суммарным числом сравнений в первом и втором while). Была написана программа, генерирующая тест, на котором быстрая сортировка сделает наибольшее число таких сравнений, после чего та была успешно проверена на сайте acmp.

### Код программы

```
from Lab3.utils import *
from Lab3.task2.src.anti_qsort import anti_qsor_main

def anti_qsor(numbers):
    res = list(range(1, numbers + 1))
    for i in range(2, numbers):
        res[i // 2], res[i] = res[i], res[i // 2]
    return res

def anti_qsor_main():
    n = file_read_size_int_single()
    answer = anti_qsor(n)
    file_write(answer)

if __name__ == '__main__':
    measure_performance(anti_qsor_main)
```

### Тесты и анализ

Алгоритм был испытан на различных сетах данных, и на всех он показал себя отлично. Это подтверждает, что программа для генерации наихудшего теста была написана верно.

```
import unittest
from Lab3.task2.src.anti_qsor import *
from Lab3.utils import *

class TestAntiQSort(unittest.TestCase):
    def check_antiqsoring(self, number, testlist):
        result, elapsed_time, peak_memory_megabytes = measure_performance(anti_qsor, *args: number)
        self.assertEqual(result, testlist)

    def test_should_antiqsort_small(self):
        number = 3
        testlist = [1, 3, 2]
        self.check_antiqsoring(number, testlist)

    def test_should_antiqsort_big(self):
        number = 10
        testlist = [1, 4, 6, 8, 10, 5, 3, 7, 2, 9]
        self.check_antiqsoring(number, testlist)

if __name__ == '__main__':
    unittest.main()
```

### Задача 3: Сортировка пугалом

«Сортировка пугалом» — это давно забытая народная потешка. Участнику под верхнюю одежду продевают деревянную палку, так что у него оказываются растопырены руки, как у огородного пугала. Перед ним ставятся  $n$  матрёшек в ряд. Из-за палки единственное, что он может сделать — это взять в руки две матрешки на расстоянии  $k$  друг от друга (то есть  $i$ -ую и  $i + k$ -ую), развернуться и поставить их обратно в ряд, таким образом поменяв их местами.

Задача участника — расположить матрёшки по неубыванию размера. Было проверено, может ли он это сделать, применяя алгоритм быстрой сортировки, описанный в первом задании.

#### Код программы

```
from Lab3.task1.src.qsort_plus import qsort
from Lab3.utils import *

def can_sort_dolls(dolls_number, hand, dolls):  2 usages
    groups = [[] for _ in range(hand)]

    for i in range(dolls_number):
        groups[i % hand].append(dolls[i])

    for group in groups:
        qsort(group)

    sorted_dolls = []
    index = [0] * hand

    for i in range(dolls_number):
        group_index = i % hand
        sorted_dolls.append(groups[group_index][index[group_index]])
        index[group_index] += 1

    if sorted_dolls == qsort(dolls):
        return "YES"
    return "NO"

def scarecrow_sort_main():  1 usage
    n, k, array = file_read_int_int_array()
    result = can_sort_dolls(n, k, array)
    file_write([result])

if __name__ == '__main__':
    measure_performance(scarecrow_sort_main)
```

## Тесты и анализ

Алгоритм был протестируирован на различных сетах данных, в каждом своё значение числа матрёшек и длины перекладины пугала. Во всех случаях алгоритм отработал быстро и безошибочно, что доказывает, что тот был написан верно.

```
import unittest
from Lab3.task3.src.scarecrow_sort import *
from Lab3.utils import *

class TestScarecrowSort(unittest.TestCase):
    def check_sorting(self, filename, answer): 3 usages
        n, k, array = file_read_int_int_array(filename)
        result, elapsed_time, peak_memory_megabytes = measure_performance(can_sort_dolls, *args: n, k, array)
        self.assertEqual(result, answer)

    def test_example_a(self):
        self.check_sorting(filename: '../txtf/test_input1.txt', answer: "NO")

    def test_example_b(self):
        self.check_sorting(filename: '../txtf/test_input2.txt', answer: "YES")

    def test_example_c(self):
        self.check_sorting(filename: '../txtf/test_input3.txt', answer: "NO")

if __name__ == '__main__':
    unittest.main()
```

## Задача 5: Индекс Хирша

Для заданного массива целых чисел citations, где каждое из этих чисел - число цитирований i-ой статьи ученого-исследователя, был высчитан индекс Хирша этого ученого.

Код программы

```
from Lab3.utils import *
from Lab3.task1.src.qsort_plus import qsort

def hirsh_index(citations): 2 usages
    qsort(citations)
    citations.reverse()

    h = 0
    for i in range(len(citations)):
        if citations[i] > i:
            h += 1
        else:
            break
    return h

def hirsh_index_main(): 1 usage
    line = file_read_array()
    result = hirsh_index(line)
    file_write([result])

if __name__ == '__main__':
    measure_performance(hirsh_index_main)
```

Тесты и анализ

Для испытания данной программы были загружены различные сеты данных. Во всех случаях программа отработала точно и быстро. Это доказывает, что написанный алгоритм расчёта индекса Хирша верен и безошибочен.

Были выяснены следующие данные:

Поскольку используется линейный проход за  $O(n)$  и быстрая сортировка за  $O(n \log n)$ , то асимптотика алгоритма —  $O(n \log n)$ .

Если бы массив был уже отсортирован по возрастанию, мы бы сразу перешли к линейному проходу с конца его, улучшив асимптотику до  $O(n)$ .

```
import unittest
from Lab3.task5.src.hirsh_index import *
from Lab3.utils import *

class TestHirshIndex(unittest.TestCase):
    def check_hirsh_index(self, filename, expected): 3 usages
        line = file_read_array(filename)
        result, elapsed_time, peak_memory_megabytes = measure_performance(hirsh_index, *args: line)
        self.assertEqual(result, expected)

    def test_example_a(self):
        self.check_hirsh_index(filename: '../txtf/test_input1.txt', expected: 0)

    def test_example_b(self):
        self.check_hirsh_index(filename: '../txtf/test_input2.txt', expected: 1)

    def test_example_c(self):
        self.check_hirsh_index(filename: '../txtf/test_input3.txt', expected: 3)

if __name__ == '__main__':
    unittest.main()
```

## Задача 6: Сортировка целых чисел

В этой задаче нужно было отсортировать много неотрицательных целых чисел.

Были даны два массива, A и B, содержащие соответственно n и m элементов. Числа, которые нужно было отсортировать, имеют вид  $A_i \cdot B_j$ , где  $1 \leq i \leq n$  и  $1 \leq j \leq m$ . Иными словами, каждый элемент первого массива нужно было умножить на каждый элемент второго массива.

Из этих чисел получилась отсортированная последовательность C длиной  $n \cdot m$ . Была выведена сумма каждого десятого элемента этой последовательности (то есть,  $C_1 + C_{11} + C_{21} + \dots$ ).

Код программы

```
from Lab3.utils import *
from Lab3.task1.src.qsort_plus import qsort

def sum_tenth(combinations):  2 usages
    qsort(combinations)
    return sum(combinations[i] for i in range(0, len(combinations), 10))

def multiply(n, m, array1, array2):  2 usages
    combinations = []
    for i in range(n):
        for j in range(m):
            combinations.append(array1[i] * array2[j])
    return combinations

def solve(n, m, array1, array2):  1 usage
    combinations = multiply(n, m, array1, array2)
    result, elapsed_time, peak_memory_megabytes = measure_performance(sum_tenth, *args: combinations)
    return result

def zahlen_sort_main():  1 usage
    n, m, array_a, array_b = file_read_int_int_array_array()
    result = solve(n, m, array_a, array_b)
    file_write([result])

if __name__ == '__main__':
    measure_performance(zahlen_sort_main)
```

## Тесты и анализ

Для испытания данной программы были загружены различные сеты данных. Во всех случаях программа отработала точно и быстро. Это доказывает, что алгоритм быстрой сортировки удобен и позволяет быстро и эффективно сортировать как крупные, так и небольшие объёмы данных.

```
from Lab3.task6.src.zahlen_sort import *

class TestZahlenSort(unittest.TestCase):
    def check_zahlen_sort(self, filename, expected):  3 usages
        n, m, array_a, array_b = file_read_int_int_array_array(filename)
        combinations = multiply(n, m, array_a, array_b)
        result, elapsed_time, peak_memory_megabytes = measure_performance(sum_tenth, *args: combinations)
        self.assertEqual(result, expected)

    def test_should_zahlen_sort_example_a(self):
        filename = '../txtf/test_input1.txt'
        self.check_zahlen_sort(filename, expected: 51)

    def test_should_zahlen_sort_example_b(self):
        filename = '../txtf/test_input2.txt'
        self.check_zahlen_sort(filename, expected: 3)

    def test_should_zahlen_sort_example_c(self):
        filename = '../txtf/test_input3.txt'
        self.check_zahlen_sort(filename, expected: 0)

if __name__ == '__main__':
    unittest.main()
```

## Заключение

В процессе выполнения лабораторной работы, посвящённой быстрой сортировке и её различным вариантам, были разработаны и протестираны несколько алгоритмов сортировки и алгоритмов сортировки за линейное время. Каждый из методов, включая различные способы использования алгоритма быстрой сортировки и алгоритмов сортировки за линейное время, был подробно изучен, реализован на Python и проверен с помощью модуля unittest. Были проанализированы принципы их работы, выявлены преимущества и недостатки, а также выполнена оценка их временной сложности.

Выполнение этой лабораторной работы позволило углубить знания об основных алгоритмах сортировки и алгоритмов сортировки за линейное время, улучшить навыки программирования на Python, а также развить умение проводить модульное тестирование программ.