Звіт до практичного завдання №1

на тему:

“Порівняння ефективності роботи алгоритмів сортування: Selection sort, Insertion sort, Merge sort та Shellsort”

Роботу виконала:

Бондарець Анна-Аліна Вікторівна

Група ПКН-20Б

Завдання:

Порівняти швидкодію та кількість операцій у роботі алгоритмів Selection sort, Insertion sort, Merge sort та Shellsort у реалізації на python. Для кожного із алгоритмів потрібно провести 4 типи експериметрів із розміром вхідного масиву від 2^7 і до 2^{15} елементів, кожного разу збільшуючи попередній розмір у 2 рази.

Експерименти залежать від типу вхідного масиву (впорядкованості елементів) та кількості повторень. Алгоритми перевіряються на одинакових між собою в межах одного тесту масивах, а результати зберігаються у відповідних json-файлах та python словниках.

Вхідні дані для різних типів експериментів:

1. Випадком чином згенерований масив; провести 5 повторень та знайти середнє значення;

2. Значення масиву відсортовані у порядку зростання; 1 повторення;

3. Значення масиву відсортовані у порядку спадання; 1 повторення;

4. Масив містить лише елементи із множини {1, 2, 3}, розставлені у довільному порядку; провести 3 повторення та знайти середнє значення.

Отримані результати у відповідних json-файлах візуалізовуються за допомогою python бібліотеки та зберігаються у pdf-файлі. **Усі файли можна знайти на GitHub репозиторії за посиланням: https://github.com/alorthius/algorithms-lab-1**

Специфікація комп'ютера, на якому виконувалась лабораторна:

CPU: Intel® Core™ i7-10510U

Number of Cores / Threads: 4 / 8

Clock Rate: 1800 – 4800 MHz

System Memory size: 16GiB

OS: Manjaro Linux

Програмний код алгоритмів:

1. Selection sort: (модуль selection_sort.py)

```python
def selection_sort(array: list) -> (list, int):
    """Sort the array in ascending order with selecton sort algorithm."""
    comparisons = 0
    ind_1 = 0

    while ind_1 < len(array):
        min_index = ind_1

        ind_2 = ind_1 + 1
        while ind_2 < len(array):
            if array[min_index] > array[ind_2]:
                min_index = ind_2
            comparisons += 1
            ind_2 += 1

        array[ind_1], array[min_index] = array[min_index], array[ind_1]
        ind_1 += 1

    return array, comparisons
```

2. Insertion sort: (модуль insertion_sort.py)

```python
def insertion_sort(array: list) -> (list, int):
    """Sort the array in ascneding order with insertion sort algorithm."""
    comparisons = 0
    index = 1
    while index < len(array):

        current_element = array[index]
        predicessor = index - 1
        comparisons += 1

        while predicessor >= 0 and current_element < array[predicessor]:
            # make space to insert the element
            array[predicessor + 1] = array[predicessor]
            predicessor -= 1
            comparisons += 2

        array[predicessor + 1] = current_element
        index += 1

    return array, comparisons
```

3. Merge sort: (модуль merge_sort.py)

```python
def merge_sort(array: list, comparisons: int = 0) -> (list, int):
    """Sort list in ascneding order using merge sort algorithm."""

    if len(array) < 2:
        return array, 1

    middle_index = len(array) // 2
    left_array, c1 = merge_sort(array[:middle_index], comparisons)
    right_array, c2 = merge_sort(array[middle_index:], comparisons)

    new_array, more_comparisons = merge_two_arrays(
        left_array, right_array, comparisons)
    comparisons += more_comparisons + c1 + c2

    return new_array, comparisons


def merge_two_arrays(left_array: list, right_array: list, comparisons: int) -> (list, int):
    """Merge two arrays."""
    left_length = len(left_array)
    right_length = len(right_array)

    if not left_length or not right_length:
        return left_array or right_array

    merged = []
    left_ind, right_ind = 0, 0

    while (len(merged) < left_length + right_length):
        comparisons += 2

        if left_array[left_ind] < right_array[right_ind]:
            merged.append(left_array[left_ind])
            left_ind += 1
        else:
            merged.append(right_array[right_ind])
            right_ind += 1

        if left_ind == left_length or right_ind == right_length:
            merged.extend(left_array[left_ind:] or right_array[right_ind:])
            break

    return merged, comparisons
```

4. Shellsort: (модуль shellsort.py)

```python
def shellsort(array: list) -> (list, int):
    """Sort the array in ascending order with shellsort algorithm."""
    comparisons = 0
    interval = len(array) // 2

    while interval > 0:
        comparisons += 1

        index = interval
        while index < len(array):
            temp_elem = array[index]
            reversive_index = index

            while reversive_index >= interval and array[reversive_index - interval] > temp_elem:
                comparisons += 2
                array[reversive_index] = array[reversive_index - interval]
                reversive_index -= interval

            array[reversive_index] = temp_elem
            index += 1

        interval = interval // 2

    return array, comparisons
```

Примітка:

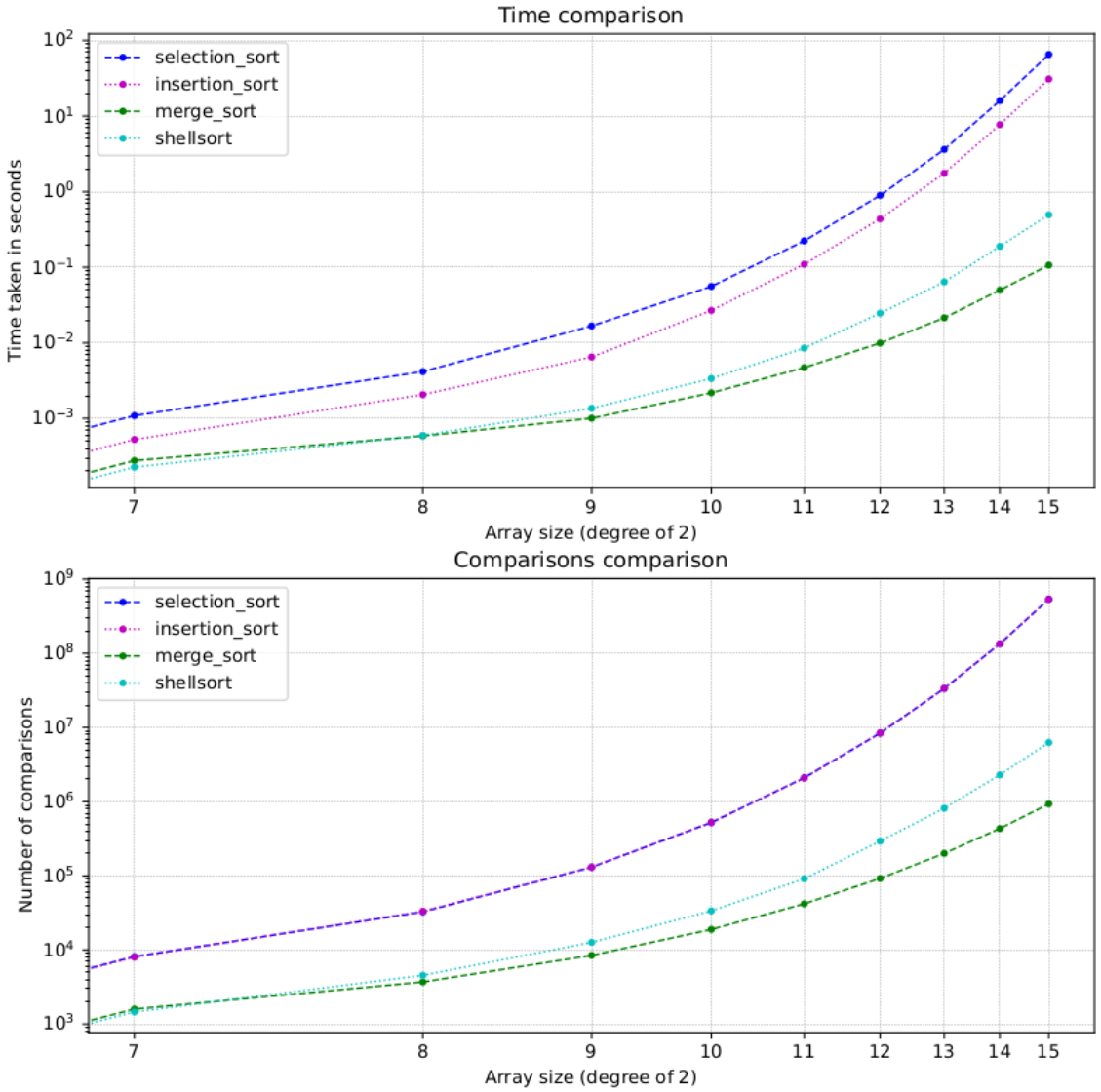
Усі алгоритми були виконанні без використання for-циклів в користь використання while-циклів для покращення швидкодії коду. Із подібних міркувань, усі алгоритми за вийнятку Merge sort були написані без використання рекурсії, надаючи перевагу ітераційному методу. Проте, Merge sort є виключенням, адже у його випадку, ітерація лише сповільнювала б час виконання програми.

Результати проведення тестів із відповідними графіками:

(графіки виконані за допомогою python бібліотеки matplotlib, див. модуль “visualisation.py”)

Завдання 1:

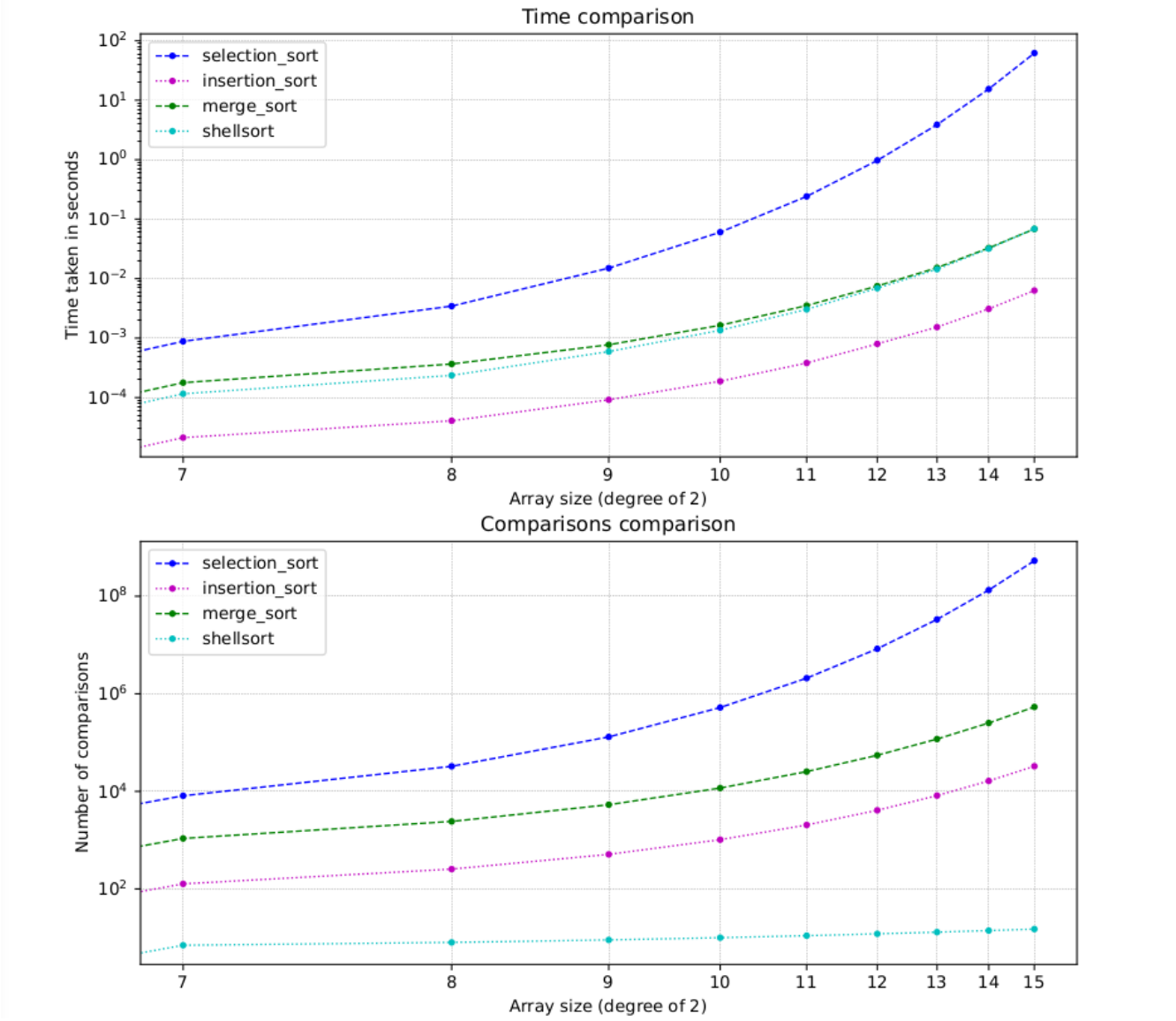
Перевірити усі алгоритми на рандомно згенерованому масиві 5 разів та знайти середні значення відповідних результатів.



У результаті видно, що selection sort та insertion sort мають подібні результати по затраченому часу, а округлена кількість операцій порівняння є практично ідентичною (по логарифмічній шкалі із базою 10). Merge та shellsort, в свою чергу, також дали подібні між собою результати, проте вони виявилися більш оптимальними, і різниця між цими та двома попередніми алгоритмами із збільшенням розміру масиву лише збільшується. Таким чином, у цьому екперименті по усім параметрам, алгоритм **merge sort** та **shellsort** показали себе найкраще.

Завдання 2:

Перевірити усі лагоритми на посортованому в порядку зростання масиві. Отримані результати:



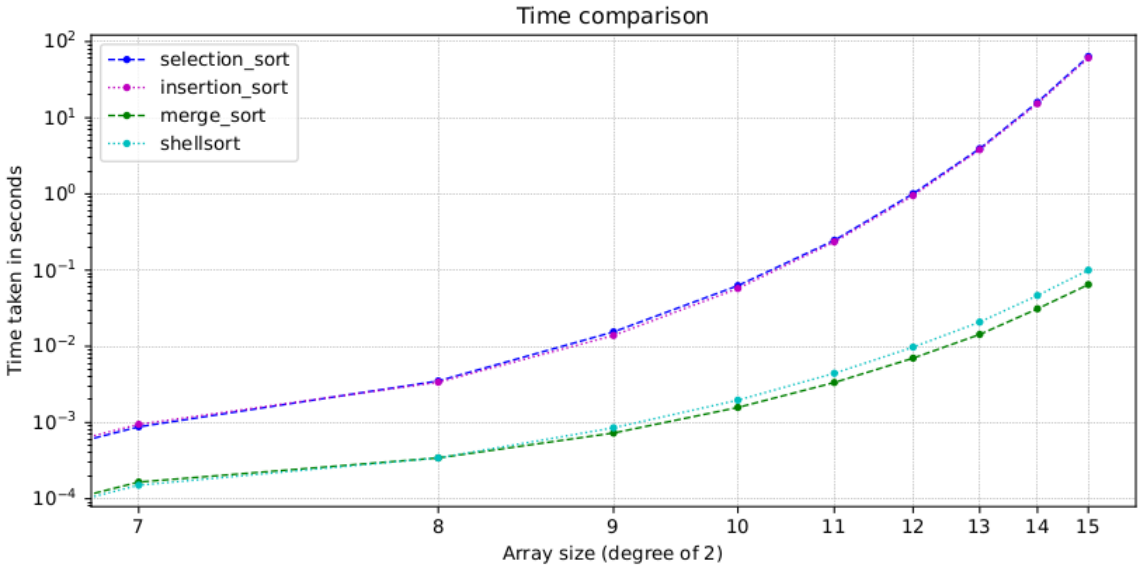
По часу виконання, merge sort та shellsort знову виявились надзвичайно подібними по результатам, проте у випадку кількості операцій порівнянь, shellsort є надзвичайно оптимальним та його графік представляється практично горизонтальною лінією навіть із збільшенням розміру масиву. На відміну від попереднього експерименту, selection sort та insertion sort на цей раз показали разюче відмінні результати.

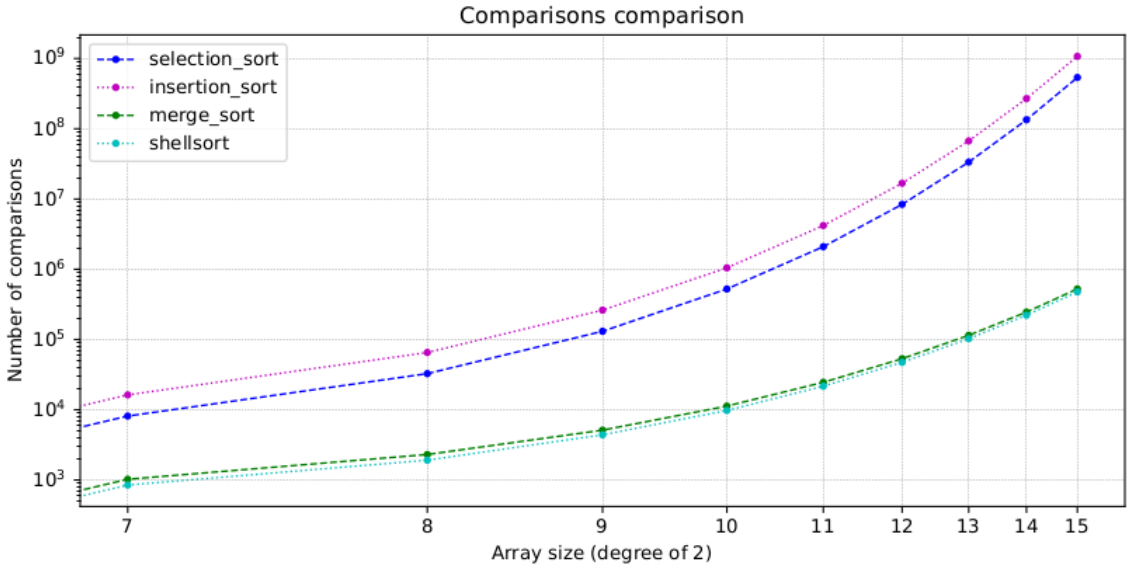
Загалом, по затраченому часу, алгоритм **insertion sort** виявився найоптимальнішим, проте при підрахунку кількості операцій порівняння, він уступає місце **shellsort**'у, який отримує явну першість по цьому параметру. Selection sort, в свою чергу, знову показав себе як самий неоптимальний алгоритм.

Завдання 3:

Перевірити усі лагоритми на посортованому в порядку спадання масиві.

Отримані результати:



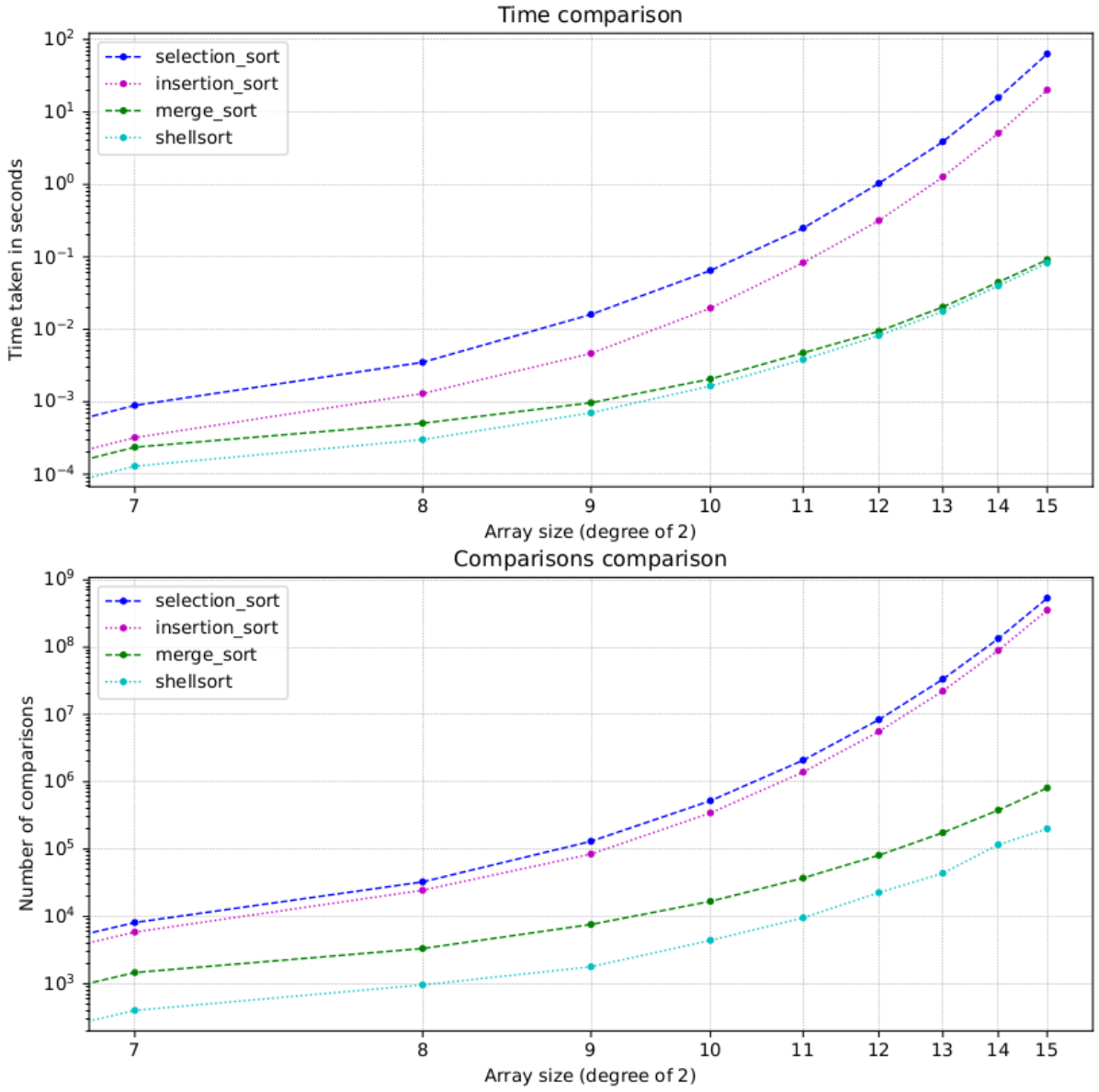


У цьому експерименті спостерігається подібна кореляція результатів, що й у першому: алгоритми selection sort та insertion sort, merge sort та shellsort попарно між собою навели дуже подібні результати. Підсумовуючи, першість у цьому експерименті належить двом алгоритмам **merge sort** та **shellsort**, у той час як інші два навели гірші параметри у 10 і більше (в залежності від розміру вхідного масиву) разів.

Завдання 4:

Перевірити усі алгоритми на масиві, що містить лише елементи множити {1, 2, 3}. Повторити 3 рази та обчислити середні значення результатів.

Отримані результати:



У цьому експерименті вже в третій раз алгоритми merge sort та shellsort показують найоптимальніші результати, а selection та insertion sort поступаються у 10 та більше (в залежності від розміру вхідного масиву) разів. Таким чином, найоптимальнішим по усім параметрам, знову виявився **shellsort**, а найгіршим selection sort.

Підсумки:

Аби підвести конструктивний підсумок, я хочу проілюструвати найважливіші отриманні результати у наступній таблиці:

	Експ. 1 (average)		Експ. 2 (already sorted)		Експ. 3 (reversed)		Експ. 4 ({1, 2, 3})		Сер. оцінка
	Час	Порівн.	Час	Порівн.	Час	Порівн.	Час	Порівн.	
Selection	4	3 *	4	4	3	3	4	4	3.625
Inserton	3	3 *	1	2	3	2	3	3	2.375
Merge	1	1	3	3	1	1	1 **	2	1.625
Shellsort	2	2	2	1	2	1	1 **	1	1.5

* та ** означають, що результати виконання різних алгоритмів дуже подібні між собою, і вони розділяють одинакові місця.

У таблиці, числа під експериментами означають “рейтинг” отриманих результатів алгоритмів на кожному експерименті по параметрам затраченого часу та виконаних операцій порівнянь. Чим менше число, тим оптимальніші результати проілюстрував певний алгоритм. У колонці “Сер. оцінка” взяте середнє арифметичне значення цих чисел.

Таким чином, можна скласти *загальний* рейтинг по *усім* параметрам:
Shellsort та Merge sort (практично ідентична сер. оцінка) → Insertion sort → Selection sort.

Проте, у різних ситуаціях, алгоритми показують різні результати, тому, наприклад, у ситуації, коли нам потрібно перевірити чи вхідний масив є відсортованим – оптимальніше буде використовувати insertion sort (по часу).

Проте, у інших випадках, найоптимальнішими алгоритмами сортування виявились саме Shellsort та Merge sort.

Усі файли можна знайти на GitHub репозиторії за посиланням:
https://github.com/alorthius/algorithms-lab-1