

# 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)

```
4 def selection_sort(array: list) -> (list, int):
5     """Sort the array in ascending order with selecton sort algorithm."""
6     comparisons = 0
7     ind_1 = 0
8
9     while ind_1 < len(array):
10        min_index = ind_1
11
12        ind_2 = ind_1 + 1
13        while ind_2 < len(array):
14            if array[min_index] > array[ind_2]:
15                min_index = ind_2
16            comparisons += 1
17            ind_2 += 1
18
19        array[ind_1], array[min_index] = array[min_index], array[ind_1]
20        ind_1 += 1
21
22    return array, comparisons
```

2. Insertion sort: (модуль insertion\_sort.py)

```
4 def insertion_sort(array: list) -> (list, int):
5     """Sort the array in ascneding order with insertion sort algorithm."""
6     comparisons = 0
7     index = 1
8     while index < len(array):
9
10        current_element = array[index]
11        predecessor = index - 1
12        comparisons += 1
13
14        while predecessor >= 0 and current_element < array[predecessor]:
15            # make space to insert the element
16            array[predecessor + 1] = array[predecessor]
17            predecessor -= 1
18            comparisons += 2
19
20        array[predecessor + 1] = current_element
21        index += 1
22
23    return array, comparisons
```

### 3. Merge sort: (модуль merge\_sort.py)

```
4 def merge_sort(array: list, comparisons: int = 0) -> (list, int):
5     """Sort list in ascending order using merge sort algorithm."""
6
7     if len(array) < 2:
8         return array, 1
9
10    middle_index = len(array) // 2
11    left_array, c1 = merge_sort(array[:middle_index], comparisons)
12    right_array, c2 = merge_sort(array[middle_index:], comparisons)
13
14    new_array, more_comparisons = merge_two_arrays(
15        left_array, right_array, comparisons)
16    comparisons += more_comparisons + c1 + c2
17
18    return new_array, comparisons
19
20
21 def merge_two_arrays(left_array: list, right_array: list, comparisons: int) -> (list, int):
22     """Merge two arrays."""
23     left_length = len(left_array)
24     right_length = len(right_array)
25
26     if not left_length or not right_length:
27         return left_array or right_array
28
29     merged = []
30     left_ind, right_ind = 0, 0
31
32     while (len(merged) < left_length + right_length):
33         comparisons += 2
34
35         if left_array[left_ind] < right_array[right_ind]:
36             merged.append(left_array[left_ind])
37             left_ind += 1
38         else:
39             merged.append(right_array[right_ind])
40             right_ind += 1
41
42         if left_ind == left_length or right_ind == right_length:
43             merged.extend(left_array[left_ind:] or right_array[right_ind:])
44             break
45
46     return merged, comparisons
```

#### 4. Shellsort: (модуль shellsort.py)

```
4 def shellsort(array: list) -> (list, int):
5     """Sort the array in ascending order with shellsort algorithm."""
6     comparisons = 0
7     interval = len(array) // 2
8
9     while interval > 0:
10         comparisons += 1
11
12         index = interval
13         while index < len(array):
14             temp_elem = array[index]
15             reverse_index = index
16
17             while reverse_index >= interval and array[reverse_index - interval] > temp_elem:
18                 comparisons += 2
19                 array[reverse_index] = array[reverse_index - interval]
20                 reverse_index -= interval
21
22             array[reverse_index] = temp_elem
23             index += 1
24
25         interval = interval // 2
26
27     return array, comparisons
```

Примітка:

Усі алгоритми були виконані без використання for-циклів в користь використання while-циклів для покращення швидкодії коду. Із подібних міркувань, усі алгоритми за винятку 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.

Програмний код виклику та тестування алгоритмів: (модуль calculations.py):

```
14 def generate_random_array(length: int, ascend=False, descend=False, subset: list = None) -> list:
15     """Return randomly generated list with possible options."""
16     if subset is None:
17         subset = [num for num in range(-(length // 2), length // 2 + 1)]
18
19     generated_list = [random.choice(subset) for _ in range(length)]
20
21     if ascend:
22         return sorted(generated_list)
23     elif descend:
24         return sorted(generated_list, reverse=True)
25     return generated_list
26
27
28 def test_one_task(task: str, algorithms: list, repetitions=5, ascend=False, descend=False, subset: list = None) -> dict:
29     """Test all the algorithms and save the results into json files."""
30     results_dict = {algorithm.__name__: {} for algorithm in algorithms}
31
32     for length_pow in range(7, 16):
33         array_size = 2 ** length_pow
34
35         for algorithm in algorithms:
36             average_time = 0
37             average_comparisons = 0
38
39             for _ in range(1, repetitions + 1):
40                 array = generate_random_array(
41                     array_size, ascend=ascend, descend=descend, subset=subset)
42
43                 # compute time taken and comparisons number
44                 start = time.time()
45                 _, comparisons = algorithm(array)
46                 end = time.time()
47
48                 average_time += end - start
49                 average_comparisons += comparisons
50
51             # find average time taken and comparisons number
52             average_time = average_time / repetitions
53             average_comparisons = average_comparisons / repetitions
54
55             results_dict[algorithm.__name__][length_pow] = [
56                 average_time, average_comparisons]
57
58     # save the results
59     with open(f'{task}.json', 'w') as json_file:
60         json.dump(results_dict, json_file)
61
62     return results_dict
63
64
65 if __name__ == '__main__':
66     algorithms_list = [selection_sort, insertion_sort, merge_sort, shellsort]
67
68     test_one_task('task_1', algorithms_list)
69     test_one_task('task_2', algorithms_list, repetitions=1, ascend=True)
70     test_one_task('task_3', algorithms_list, repetitions=1, descend=True)
71     test_one_task('task_4', algorithms_list, repetitions=3, subset=[1, 2, 3])
```

## Програмний код побудови графіків: (модуль visualisation.py)

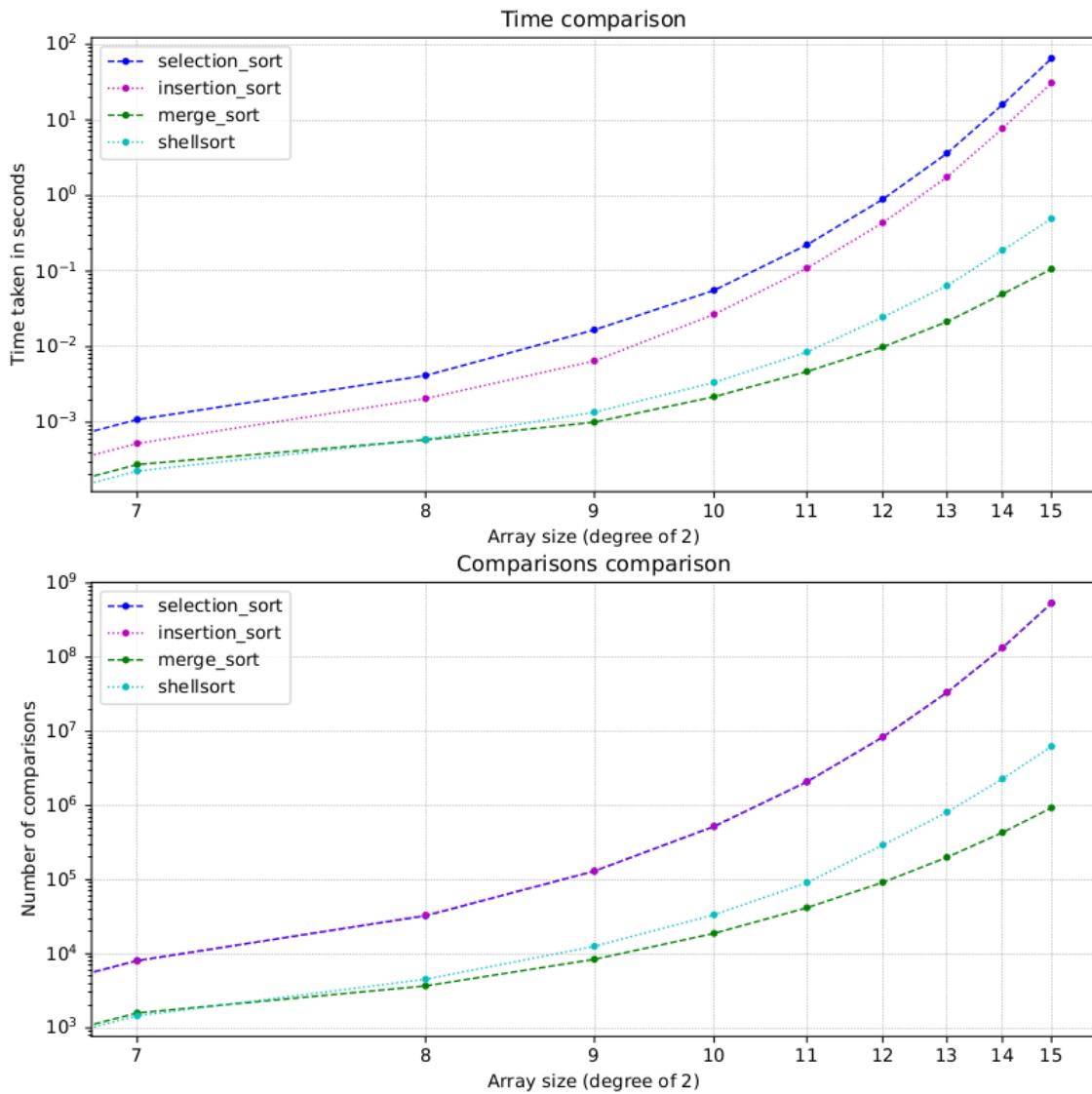
```
3 import json
4 from matplotlib import pyplot
5 from matplotlib.backends.backend_pdf import PdfPages
6
7
8 def draw_graphics(task: str) -> pyplot.figure:
9     """Create a graphic for one task and return it as a figure object."""
10
11     # read results data
12     with open(f'{task}.json') as json_file:
13         task_dict = json.load(json_file)
14
15     # add subplots for 2 graphics
16     fig, (time, comparisons) = pyplot.subplots(2, figsize=(10, 10))
17     fig.suptitle(f'Time and comparisons comparison for {task}')
18
19     time.set_title('Time comparison')
20     comparisons.set_title('Comparisons comparison')
21
22     line_style = ['b--', 'm:', 'g--', 'c:']
23     num = 0
24
25     for subplot in (time, comparisons):
26         # set logarithmic scaling for both x and y axis
27         subplot.set_yscale('log', base=10)
28         subplot.set_xscale('log', base=2)
29
30
31     for algo, algo_dict in task_dict.items():
32         time_points = [0]
33         comparisons_points = [0]
34
35         sizes_list = list(algo_dict.keys())
36         sizes_list.insert(0, '0')
37
38         for time_res, comparisons_res in algo_dict.values():
39             time_points.append(time_res)
40             comparisons_points.append(comparisons_res)
41
42         # draw lines for both time and comparisons graphics
43         time.plot(sizes_list, time_points,
44                    line_style[num], label=algo, lw=1, marker='.')
45         time.set_ylabel('Time taken in seconds')
46
47         comparisons.plot(sizes_list, comparisons_points,
48                           line_style[num], label=algo, lw=1, marker='.')
49         comparisons.set_ylabel('Number of comparisons')
50         num += 1
51
52     for subplot in (time, comparisons):
53         subplot.legend(loc='upper left')
54         subplot.grid(linestyle=':', linewidth=0.5)
55         subplot.set_xlabel("Array size (degree of 2)")
56
57     return fig
58
59
60 if __name__ == '__main__':
61     # driver to visualize all tasks and save them into one single pdf file
62     pdf_file = PdfPages('results.pdf')
63
64     for task_num in range(1, 5):
65         task_fig = draw_graphics(f'task_{task_num}')
66         pdf_file.savefig(task_fig)
67
68     pdf_file.close()
```

Результати проведення тестів із відповідними графіками:

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

### Завдання 1:

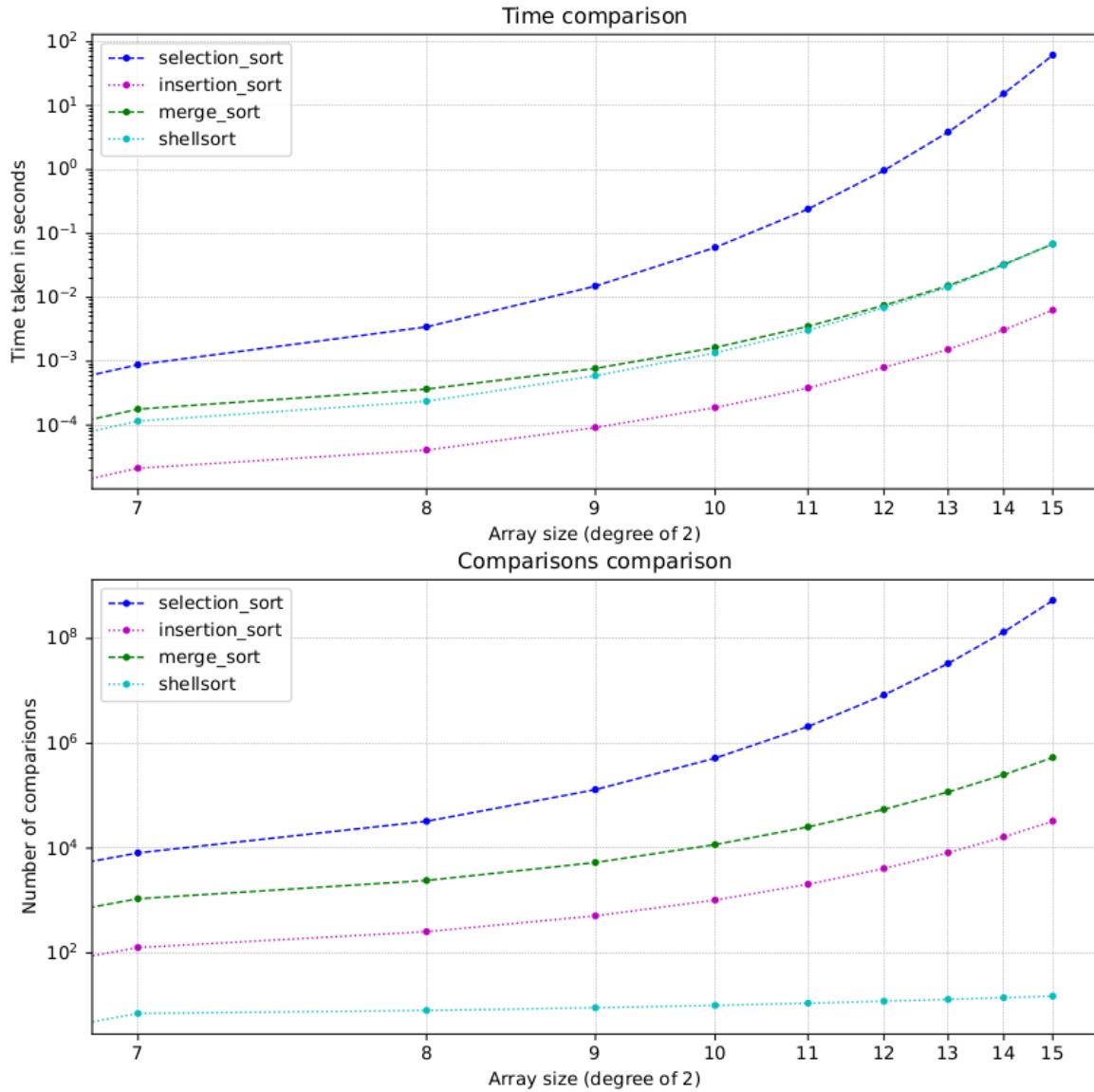
Перевірити усі алгоритми на рандомно згенерованому масиві 5 разів та знайти середні значення відповідних результатів.



У результаті видно, що selection sort та insertion sort мають подібні результати по затраченному 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 експерименті по усім параметрам, алгоритм **merge sort** та **shellsort** показали себе найкраще.

## Завдання 2:

Перевірти усі алгоритми на посортуваному в порядку зростання масиві. Отримані результати:



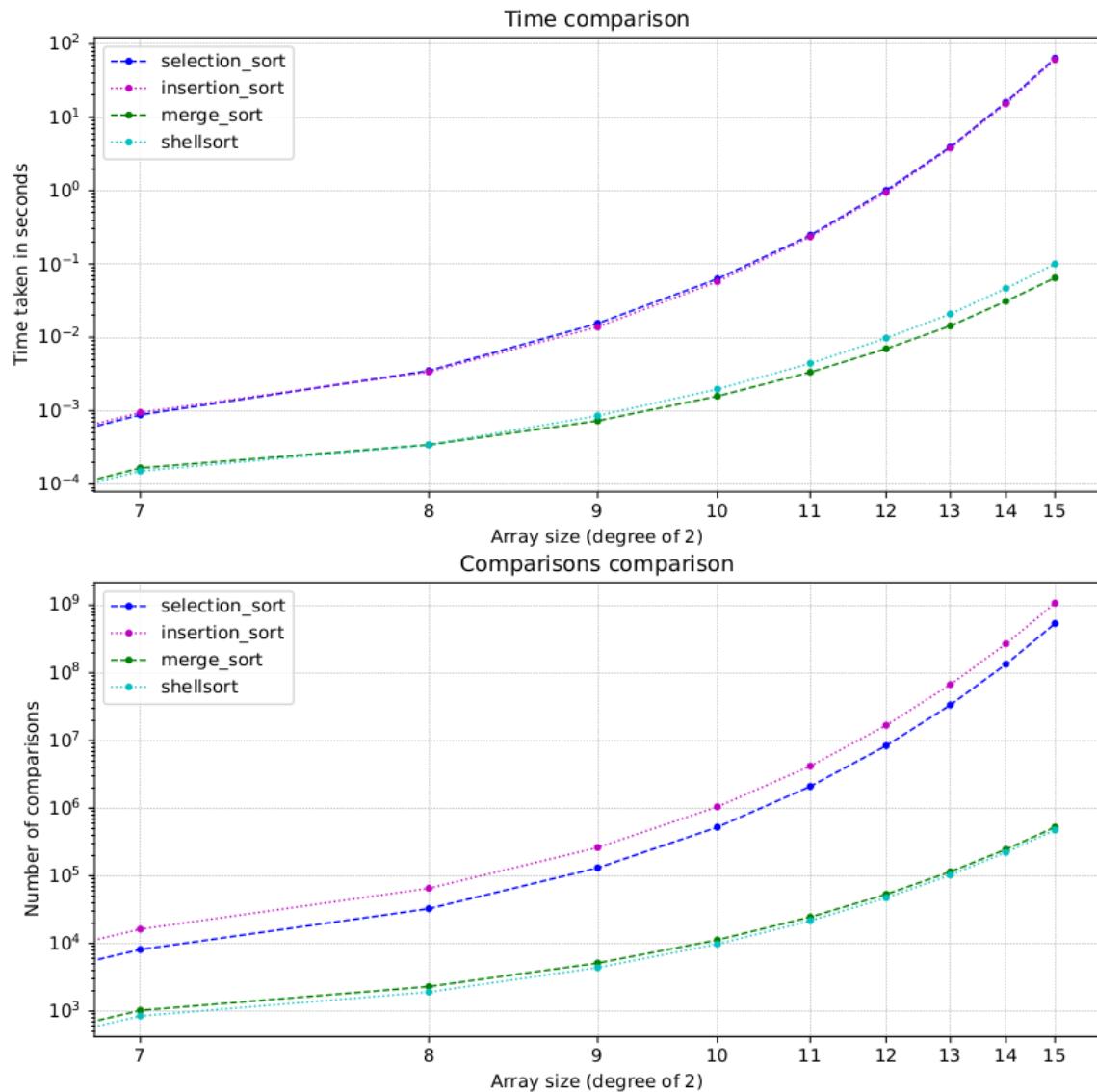
По часу виконання, merge sort та shellsort знову виявилися надзвичайно подібними по результатам, проте у випадку кількості операцій порівняння, shellsort є надзвичайно оптимальним та його графік представляється практично горизонтальною лінією навіть із збільшенням розміру масиву. На відміну від попереднього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

Загалом, по затраченному 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:

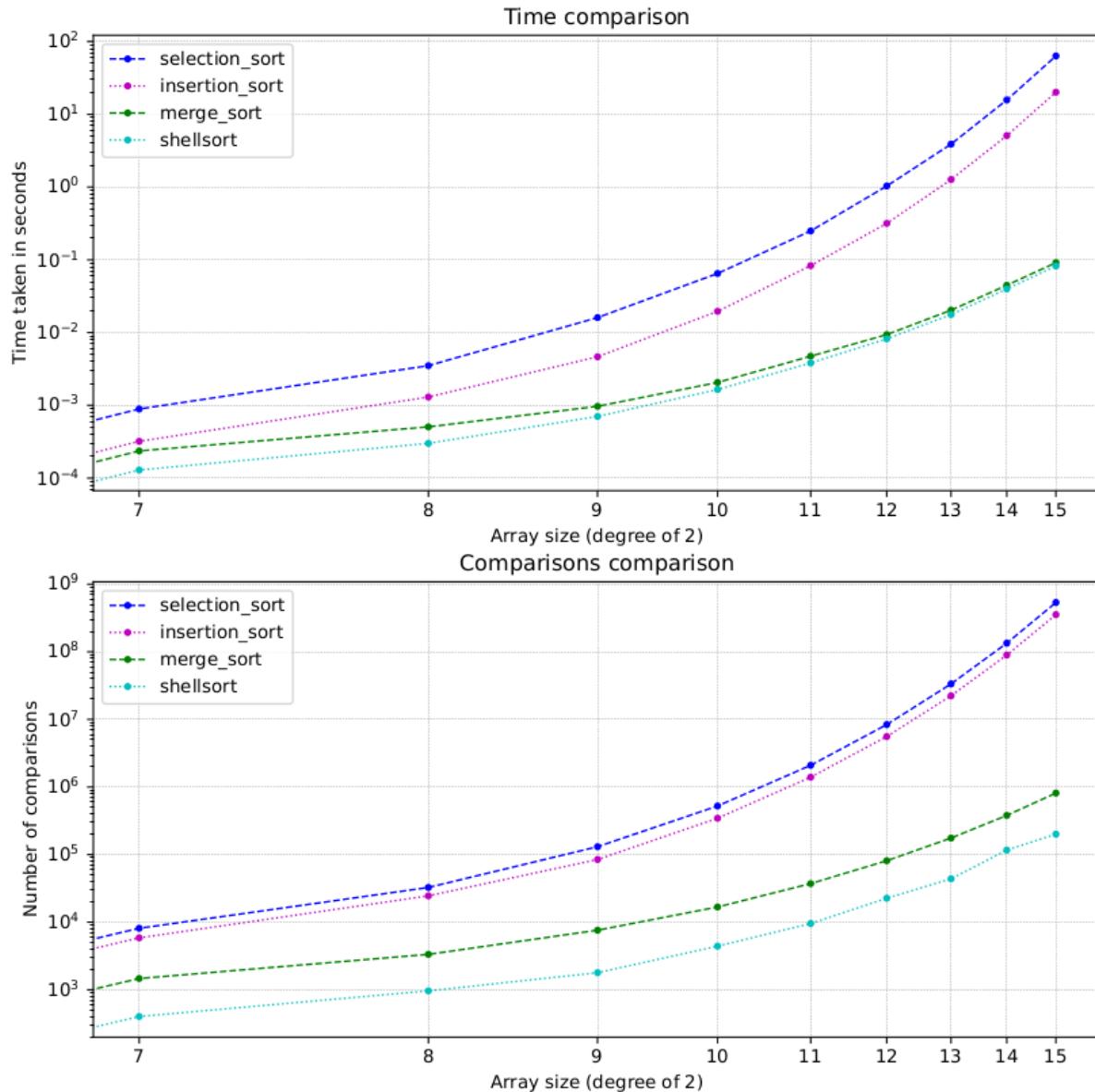


У цьому експерименті спостерігається подібна кореляція результатів, що й у першому: алгоритми selection sort та insertion sort, merge sort та shellsort попарно між собою навели дуже подібні результати. Підсумовуючи, першість у цьому експерименті належить двом алгоритмам **merge sort** та **shellsort**, 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 множини  $\{1, 2, 3\}$ . 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	Кінцевий рейтинг
	Час	Порівн.	Час	Порівн.	Час	Порівн.	Час	Порівн.		
Selection	4	3 *	4	4	3	3	4	4	3.625	12
Inserton	3	3 *	1	2	3	2	3	3	2.375	9.65
Merge	1	1	3	3	1	1	1 **	2	1.625	4
Shellsort	2	2	2	1	2	1	1 **	1	1.5	5.45

\* 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 ідентичний сер. рейтинг) → **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) або **shellsort** (по к-сті операцій порівнянь).

Саме тому, для більш “середньостатистичної” ситуації, я розрахувала кінцеву оцінку (стовбець “Кінцевий рейтинг”) по наступній формулі:

$$(T_1 + C_1) + (T_2 + C_2) \times 0.05 + (T_3 + C_3) \times 0.1 + (T_4 + C_4) \times 0.5,$$

де  $T_k$  та  $C_k$  – відповідно час та к-сть операцій порівняння для k-ого експерименту.

Таким чином, алгоритм **merge sort** виявився найефективнішим для середньостатистичного масиву при сортуванні.