# Практичне завдання №1. Порівняння роботи Selection sort, Insertion sort, Merge sort, Shell Sort

Звіт

Косик Дарини
2-й курс ІТ та БА

# Мета:

Порівняти ефективність роботи алгоритмів Selection sort, Insertion sort, Merge sort, та Shellsort для різних масивів розмірами від 2^7 і до 2^15, з кроком *x2*. Серед масивів були: випадковим чином згенерований масив, масив, відсортований у порядку зростання; масив, відсортований у порядку спадання; масив, який містить лише елементи з множини {1, 2, 3}.

# Специфікація комп’ютера:

Кількість ядер: 8
Тактова частота: 2 ГГц
Оперативна пам’ять: 16 ГБ
ОС: Windows 10

## Selection sort:

```
def selection_sort(num_lst):
    for i in range(len(num_lst)):
        min_idx = i
        for j in range(i+1, len(num_lst)):
            if num_lst[min_idx] > num_lst[j]:
                min_idx = j
            global comparisons
            comparisons += 1
        num_lst[i], num_lst[min_idx] = num_lst[min_idx], num_lst[i]

    return num_lst
```

Insertion sort:

```
def insertion_sort(num_lst):

    for i in range(1, len(num_lst)):

        minimal_idx = num_lst[i]
        j = i-1
        while j >= 0 and minimal_idx < num_lst[j]:
            num_lst[j + 1] = num_lst[j]
            j -= 1
            global comparisons
            comparisons += 1
        num_lst[j + 1] = minimal_idx

        comparisons += 1

    return num_lst
```

# Merge sort:

```python
def merge_sort(lst):

    lst_length = len(lst)

    if lst_length == 1:
        return lst

    middle = lst_length // 2

    left_lst = merge_sort(lst[:middle])
    right_lst = merge_sort(lst[middle:])

    return merge(left_lst, right_lst)


def merge(left_lst, right_lst):

    final_lst = []
    i = j = 0

    while i < len(left_lst) and j < len(right_lst):

        if left_lst[i] < right_lst[j]:
            final_lst.append(left_lst[i])
            i += 1
        else:
            final_lst.append(right_lst[j])
            j += 1

        global comparisons
        comparisons += 1

    final_lst.extend(left_lst[i:])
    final_lst.extend(right_lst[j:])

    return final_lst
```

Shell sort:

```
def shell_sort(num_list):
    n = len(num_list)
    gap = n//2

    while gap > 0:

        for i in range(gap, n):

            current = num_list[i]
            j = i
            while j >= gap and num_list[j-gap] > current:
                num_list[j] = num_list[j-gap]
                j -= gap
                global comparisons
                comparisons += 1

            num_list[j] = current
            comparisons += 1
        gap //= 2

    return num_list
```

Програмний код проведення експериментів:

```
def call_funcs(func, size, lst):

    before = time.perf_counter()
    lst_sorted = func(lst)
    time_alg = time.perf_counter() - before
    global comparisons
    comparisons_now = comparisons
    comparisons = 0

    return [time_alg, comparisons_now, func.__name__, size]


def create_lists():
    total_random_lst = []
    average_random_lst = []
    total_asc_lst = []
    total_desc_lst = []
    total_123_lst = []
    algorithms = [selection_sort, insertion_sort, merge_sort, shell_sort]
    sizes = [2**7, 2**8, 2**9, 2**10, 2**11, 2**12, 2**13, 2**14, 2**15]

    for size in sizes:
        time_sel_random = 0
        time_ins_random = 0
        time_merge_random = 0
        time_shell_random = 0
        comparisons_sel_random = 0
        comparisons_ins_random = 0
        comparisons_merge_random = 0
        comparisons_shell_random = 0

        for i in range(5):
            lst_random = [random.randrange(-10000, 10000) for x in range(size)]

            before = time.perf_counter()
            lst_sorted_sel = selection_sort(lst_random)
            time_alg = time.perf_counter() - before
            time_sel_random += time_alg
            global comparisons
            comparisons_sel_random += comparisons
            comparisons = 0

            before = time.perf_counter()
            lst_sorted_sel = insertion_sort(lst_random)
            time_alg = time.perf_counter() - before
            time_ins_random += time_alg
            comparisons_ins_random += comparisons
            comparisons = 0
```

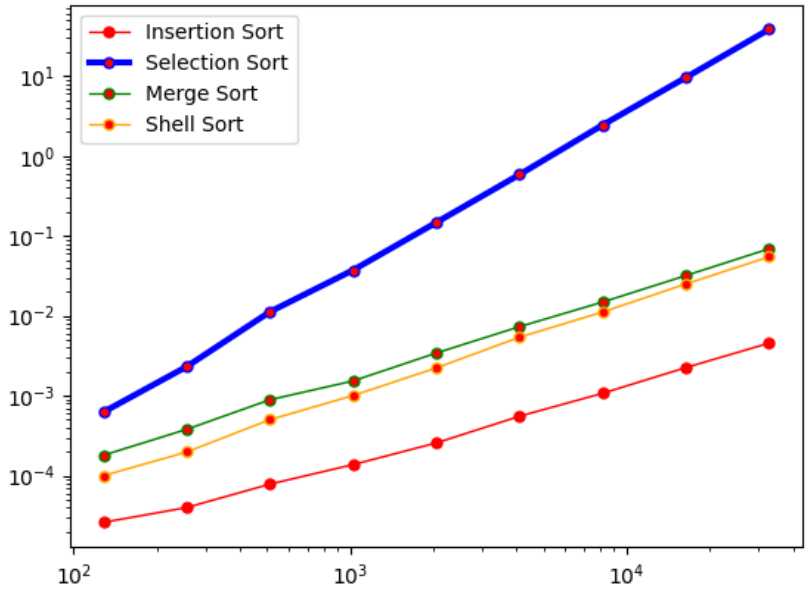
```
            before = time.perf_counter()
            lst_sorted_sel = merge_sort(lst_random)
            time_alg = time.perf_counter() - before
            time_merge_random += time_alg
            comparisons_merge_random += comparisons
            comparisons = 0

            before = time.perf_counter()
            lst_sorted_sel = shell_sort(lst_random)
            time_alg = time.perf_counter() - before
            time_shell_random += time_alg
            comparisons_shell_random += comparisons
            comparisons = 0

        total_random_lst.append(
            [time_sel_random/5, comparisons_sel_random/5, "selection_sort", size])
        total_random_lst.append(
            [time_ins_random/5, comparisons_ins_random/5, "insertion_sort", size])
        total_random_lst.append(
            [time_merge_random/5, comparisons_merge_random/5, "merge_sort", size])
        total_random_lst.append(
            [time_shell_random/5, comparisons_shell_random/5, "shell_sort", size])

    for size in sizes:
        lst_asc = [10000 * z + (random.randint(-10000, 1000))
                   for z in range(size)]
        lst_desc = lst_asc[::-1]

        for algorithm in algorithms:
            total_asc_lst.append(call_funcs(algorithm, size, lst_asc))
            total_desc_lst.append(call_funcs(algorithm, size, lst_desc))

    for size in sizes:
        time_sel_123, time_ins_123, time_merge_123, time_shell_123 = [
            0, 0, 0, 0]
        comparisons_sel_123, comparisons_ins_123, comparisons_merge_123, comparisons_shell_123 = [
            0, 0, 0, 0]

        for i in range(3):
            lst_random123 = [random.randrange(1, 4) for i in range(size)]

            before = time.perf_counter()
            lst_sorted_sel = selection_sort(lst_random123)
            time_alg = time.perf_counter() - before
            time_sel_123 += time_alg
            comparisons_sel_123 += comparisons
            comparisons = 0
```

```
            before = time.perf_counter()
            lst_sorted_ins = insertion_sort(lst_random123)
            time_alg = time.perf_counter() - before
            time_ins_123 += time_alg
            comparisons_ins_123 += comparisons
            comparisons = 0

            before = time.perf_counter()
            lst_sorted_merge = merge_sort(lst_random123)
            time_alg = time.perf_counter() - before
            time_merge_123 += time_alg
            comparisons_merge_123 += comparisons
            comparisons = 0

            before = time.perf_counter()
            lst_sorted_shell = shell_sort(lst_random123)
            time_alg = time.perf_counter() - before
            time_shell_123 += time_alg
            comparisons_shell_123 += comparisons
            comparisons = 0

        total_123_lst.append(
            [time_sel_123/3, comparisons_sel_123/3, "selection_sort", size])
        total_123_lst.append(
            [time_ins_123/3, comparisons_ins_123/3, "insertion_sort", size])
        total_123_lst.append(
            [time_merge_123/3, comparisons_merge_123/3, "merge_sort", size])
        total_123_lst.append(
            [time_shell_123/3, comparisons_shell_123/3, "shell_sort", size])

    return [total_random_lst, total_asc_lst, total_desc_lst, total_123_lst]
```

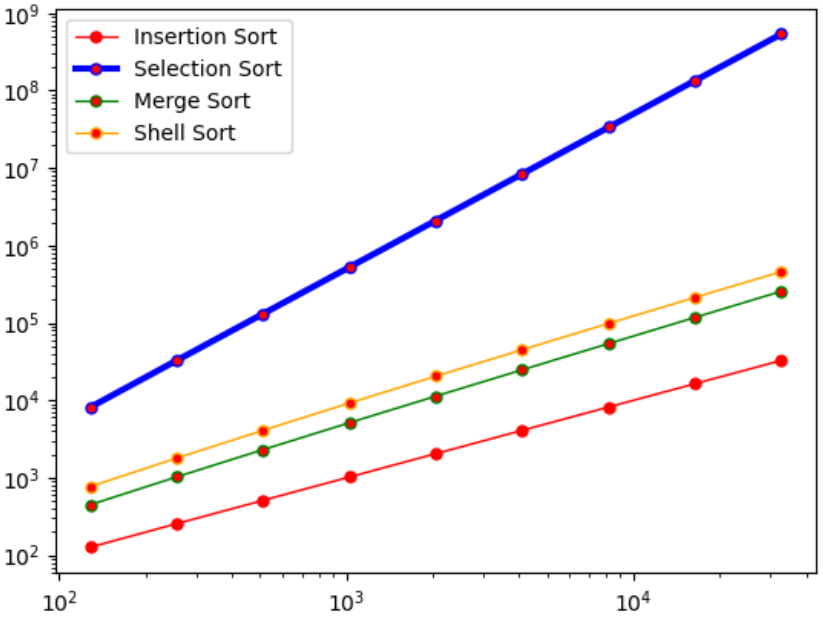
# Графіки

1. Масив, згенерований випадковим чином:

## Час



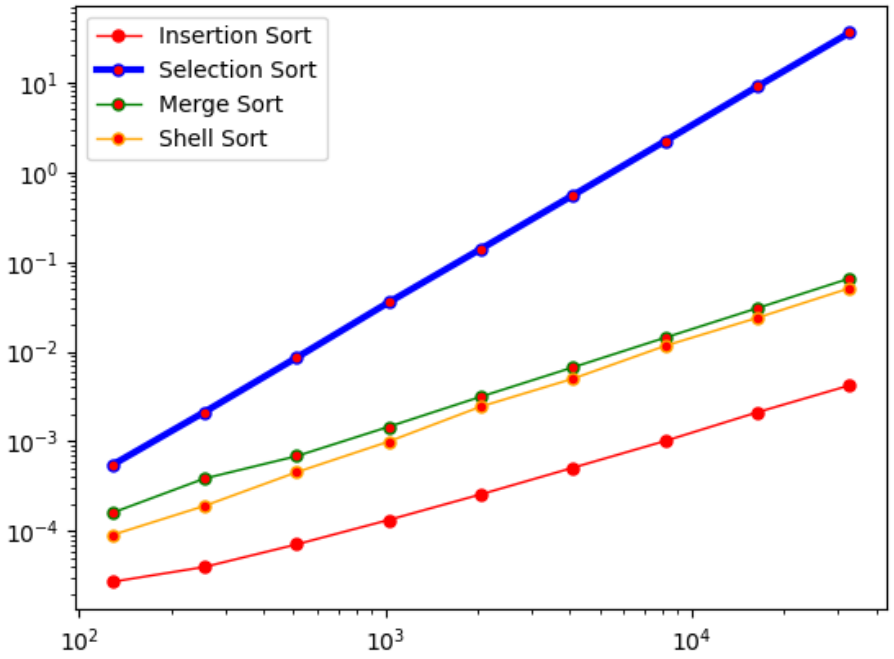

## Порівняння



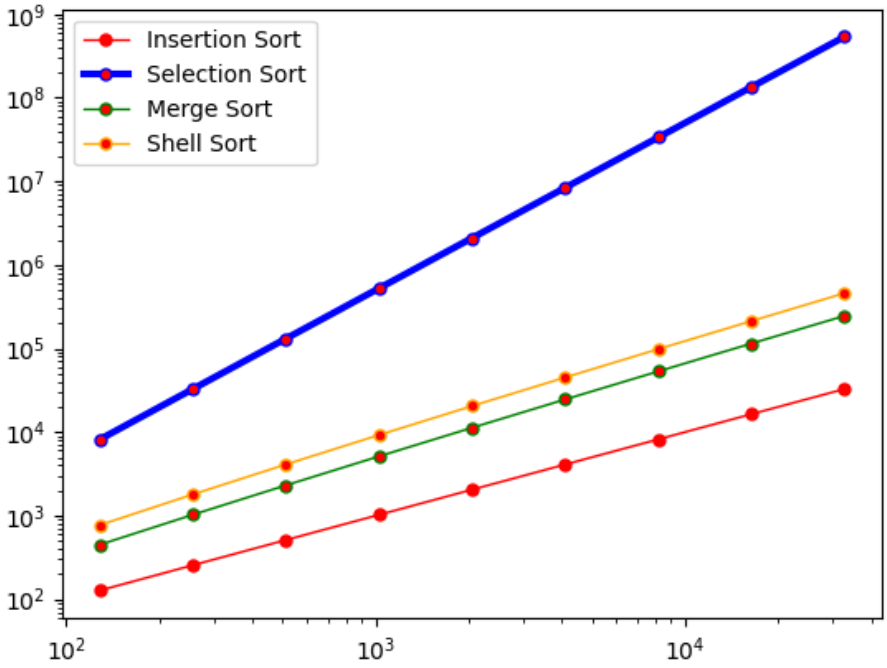

Після проведення експерименту на масиві, згенерованому випадковим чином, можна зробити висновок, що алгоритм Selection Sort працює найменш ефективно. Insertion Sort під час цього експерименту працює найефективніше як стосовно часу роботи, так і стосовно кількості порівнянь. Merge Sort та Shell Sort працюють приблизно однаково.

2. Масив, посортований у порядку зростання:

Час



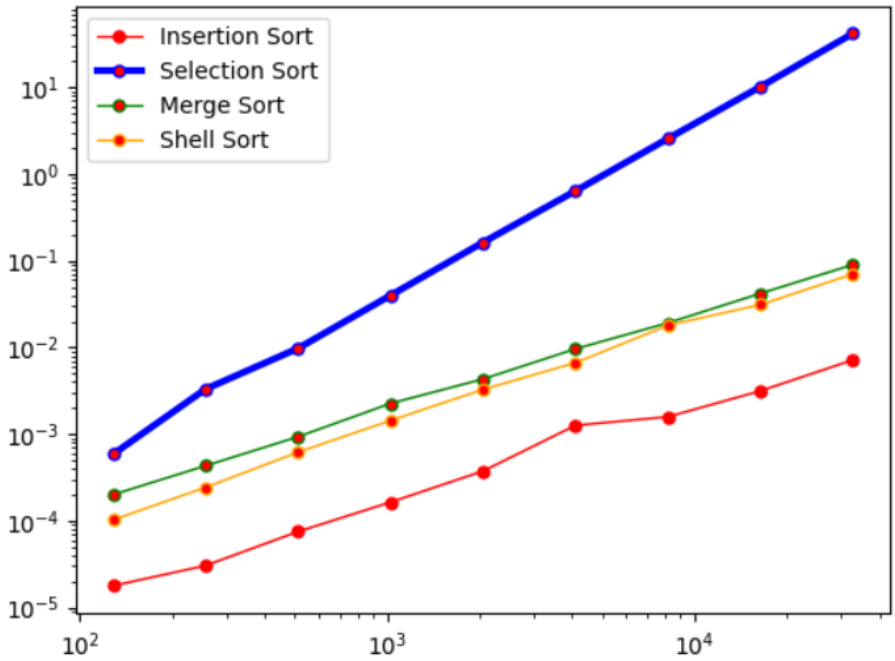

Порівняння



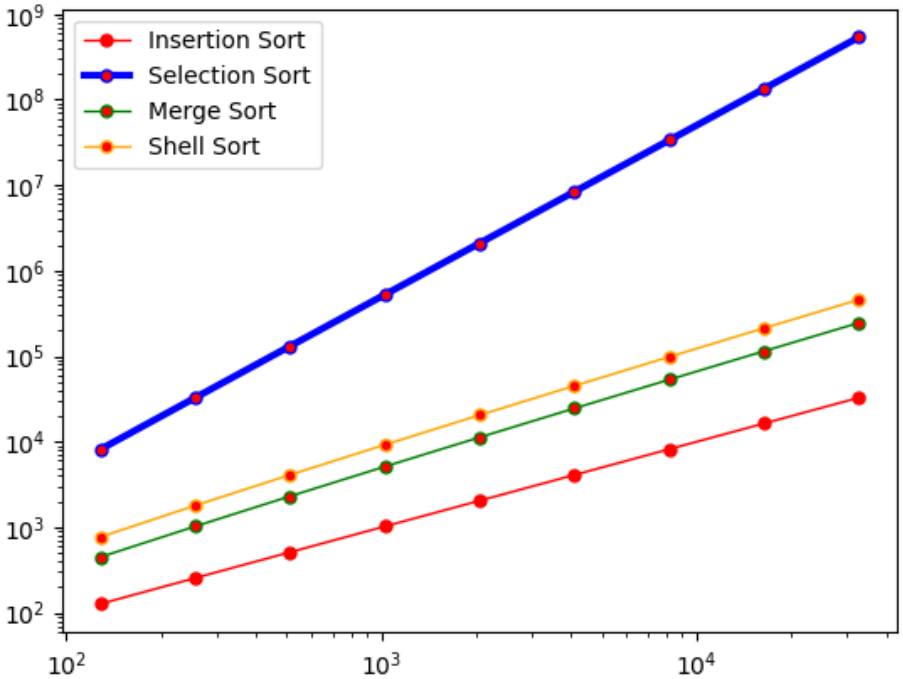

Динаміка роботи алгоритмів практично не змінилась, усі вони зростають майже лінійно.

3. Масив, посортований у порядку спадання:

Час



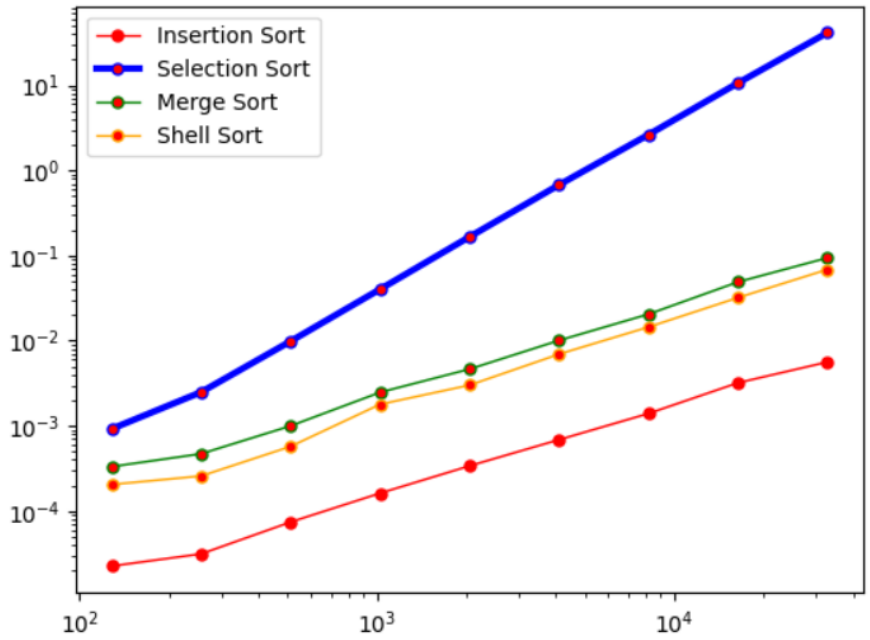

Порівняння



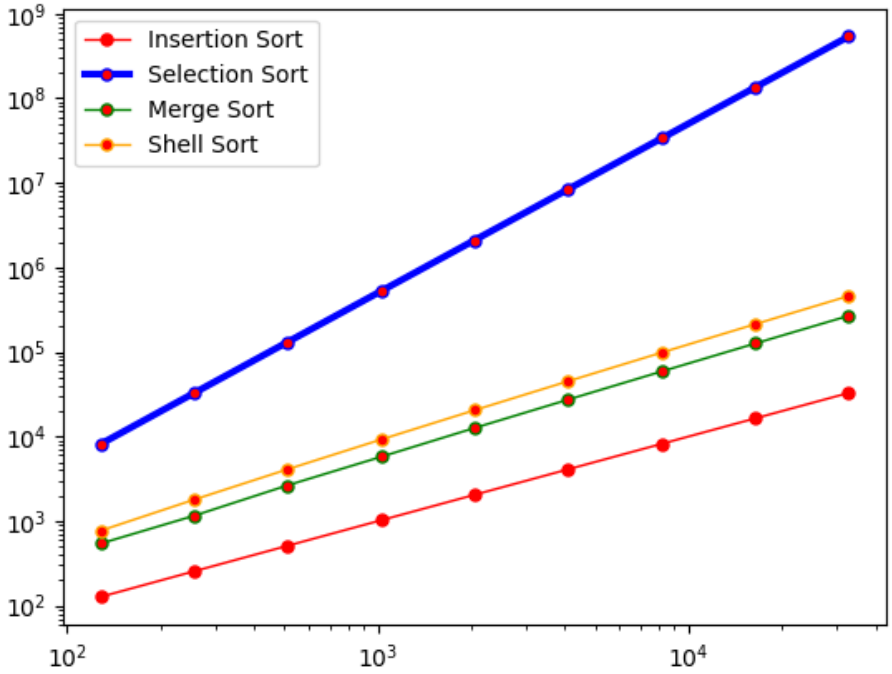

Робота алгоритмів майже нічим не відрізняється, найефективнішим залишається Insertion Sort.

4, Масив, який складається з елементів з множини {1, 2, 3}

Час




Порівняння




Робота алгоритмів залишається майже незмінною, Merge Sort та Shell Sort працюють майже за однаковий час та мають приблизно ту саму кількість порівнять, проте Shell Sort переважає за часом, а Merge Sort має меншу кількість порівнянь.

# Висновок

Під час усіх експериментів алгоритми Selection sort, Insertion sort, Merge sort та Shell Sort показували схожу динаміку роботи. Алгоритм Selection sort залишається найменш ефективним з усіх алгоритмів, адже він має найбільші час роботи та кількість порівнянь. Найшвидшим та з найменшою кількістю порівнянь є алгоритм Insertion Sort. Алгоритми Merge sort та Shell Sort працюють приблизно зі схожою ефективністю, проте Merge sort має меншу кількість порівнянь, а Shell Sort - коротший час роботи. Усі графіки зростають майже лінійно, їх ефективність мало залежить від особливостей вхідного масиву.