Бродюк Михайло
Практичне завдання №1

Опис задачі: у чотирьох різних експериментах дослідити час роботи та кількість порівнянь алгоритму Selection sort, Insertion sort, Merge sort і Shellsort , з метою краще зрозуміти специфіку їх роботи і проаналізувати на яких масивах даних їх краще застосовувати

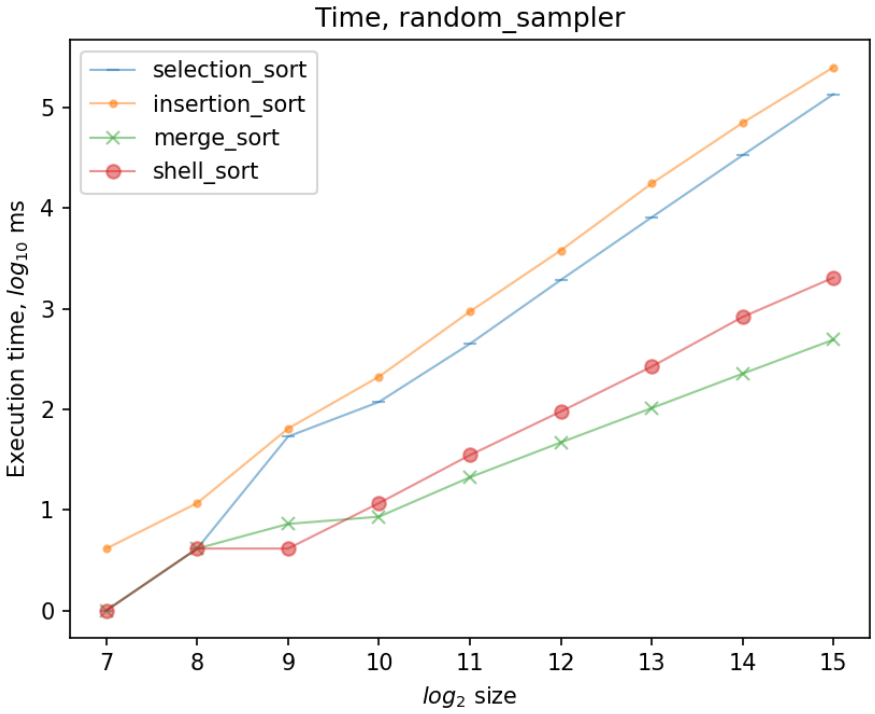
Було проведено 4 експерименти на масивах розміром від 2^7 до 2^15
1) На довільно згенерованому масиві 5 разів протестувати кожен з алгоритмів
2) На впорядкованому масиві протестувати кожен з алгоритмів
3) На масиві згенерованому у порядку спадання протестувати кожен з алгоритмів
4) На довільно згенерованому масиві ,лишень із числами 1, 2, 3, протестувати кожен з алгоритмів 3 рази

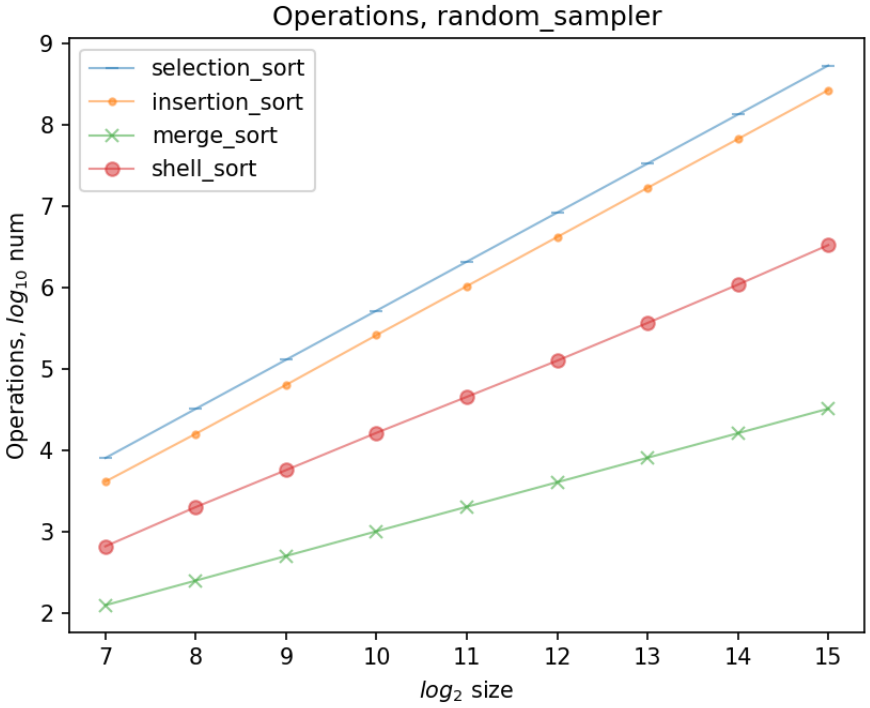
Виконана робота:
Експериметальним шляхом отримати дані про час та кількість порівнянь на різних типах масивів і з різними алгоритмами. Із отриманих даних побудувати графіки залежності часу виконання та кількості порівнянь від розміру масиву. (Загалом 8). Проаналізувати результати

Специфікація комп'ютера:

Кількість ядер:	2
Тактова частота:	2.9 ГГц
ОЗП:	8.00 Гб
ОС:	Windows 10

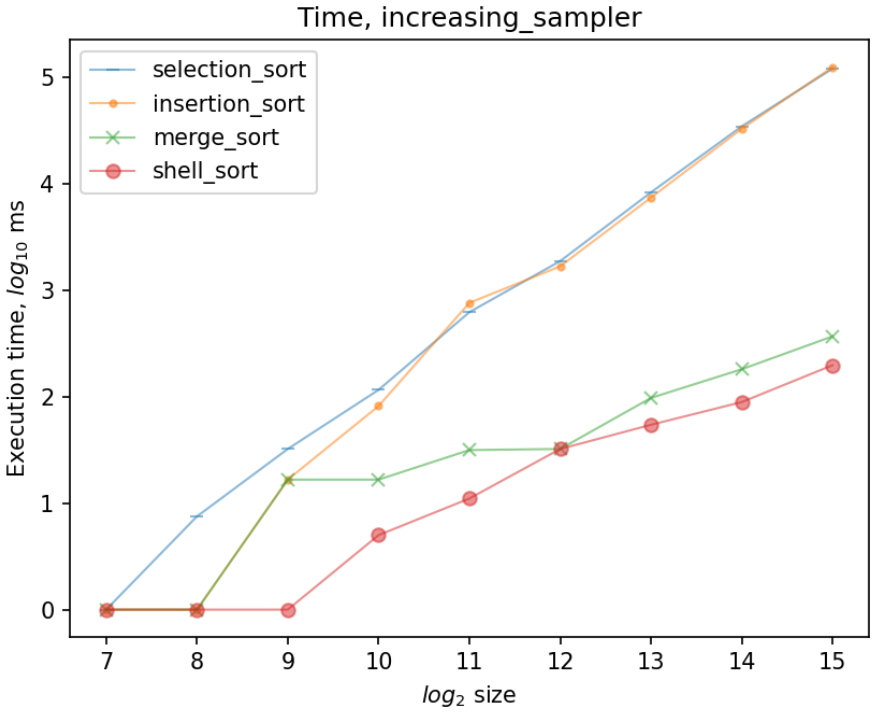
Task1

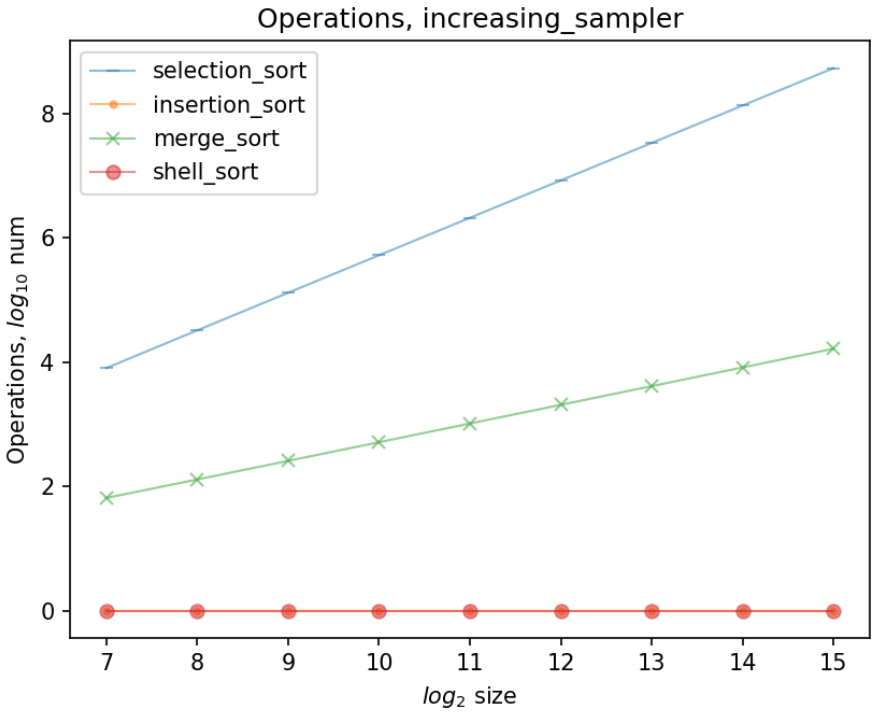

Time, random_sampler
selection_sort
insertion_sort
merge_sort
shell_sort
Execution time, log_{10} ms
log_2 size
0
1
2
3
4
5
7
8
9
10
11
12
13
14
15


Operations, random_sampler
selection_sort
insertion_sort
merge_sort
shell_sort
Operations, log_{10} num
log_2 size
2
3
4
5
6
7
8
9
7
8
9
10
11
12
13
14
15

Висновок:
У випадково згенерованому масиві, як видно, Selection sort та Insertion sort поводять себе доволі однаково – як за часом так і за кількістю операцій порівняння. Справді, у Insertion sort O(n^2) та Selection sort O(n^2) однакові. Однак Merge sort та Shell sort працюють у рази швидше – відрив на великих обсягах даних стає більш помітним, тобто саме їх слід використовувати при наявності невпорядкованого масиву. Але Shell sort є трохи повільнішим ніж Merge sort, тому краще використовувати саме той алгоритм, який є швидшим. Дані графіки також пояснюють час роботи алгоритмів, для Shell sort – O(n(log(n))^2), і для Merge sort – O(n*log(n)). Аналогічно щодо кількості операцій порівняння.

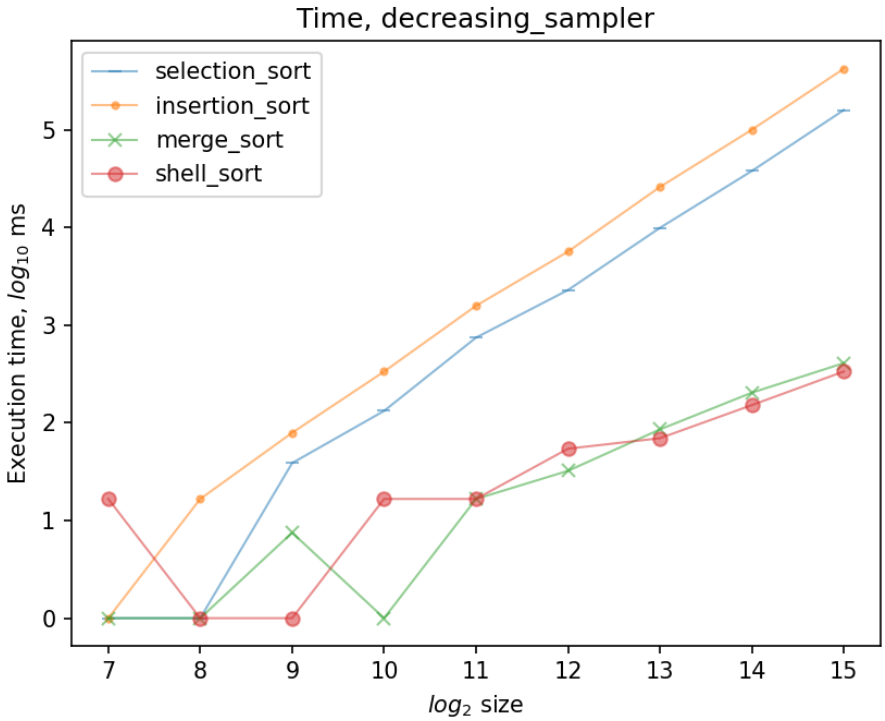
Task2

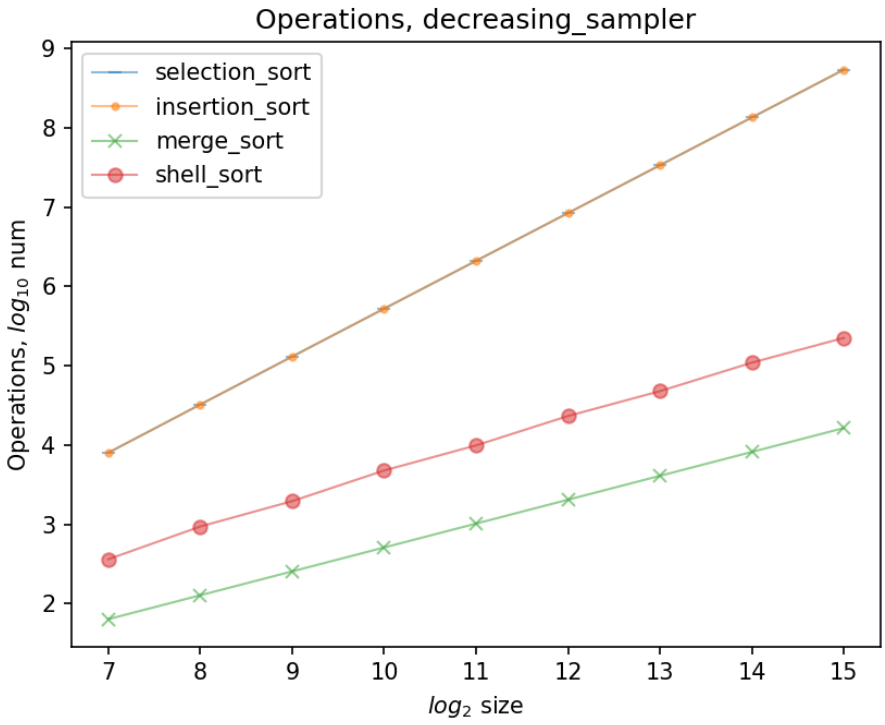




Висновок:
У сортуванні впорядкованого масиву найоптимальніший є Insertion sort – він використовує найменшу кількість порівнянь, і працює найшвидше. Оскільки це є найоптимальніший випадок то Ω(n). Якщо масив впорядкований і ми хочемо переконатися у цьому, або ж маємо майже впорядкований масив, то Insertion sort буде найшвидшим. Наступним іде Shell sort , оскільки це є також найкращий випадок то O(n*log(n)). Також Selection у цьому випадку поводиться найгірше – найбільша кількість порівнянь і найдовший час роботи. Merge sort – трохи кращий за Selection sort, але він програватиме Shell sort і Insertion sort.

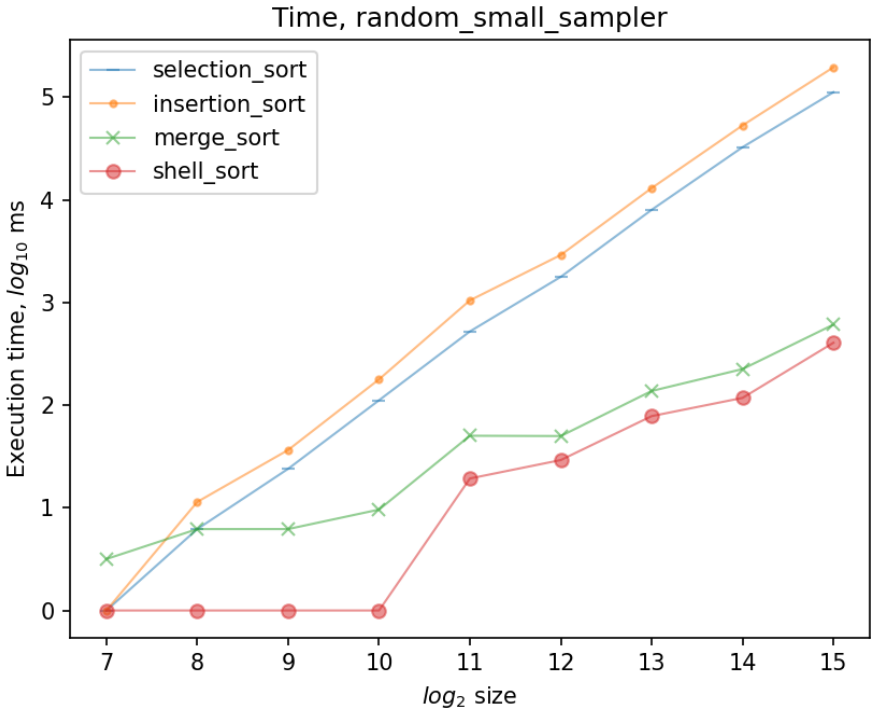
Task3

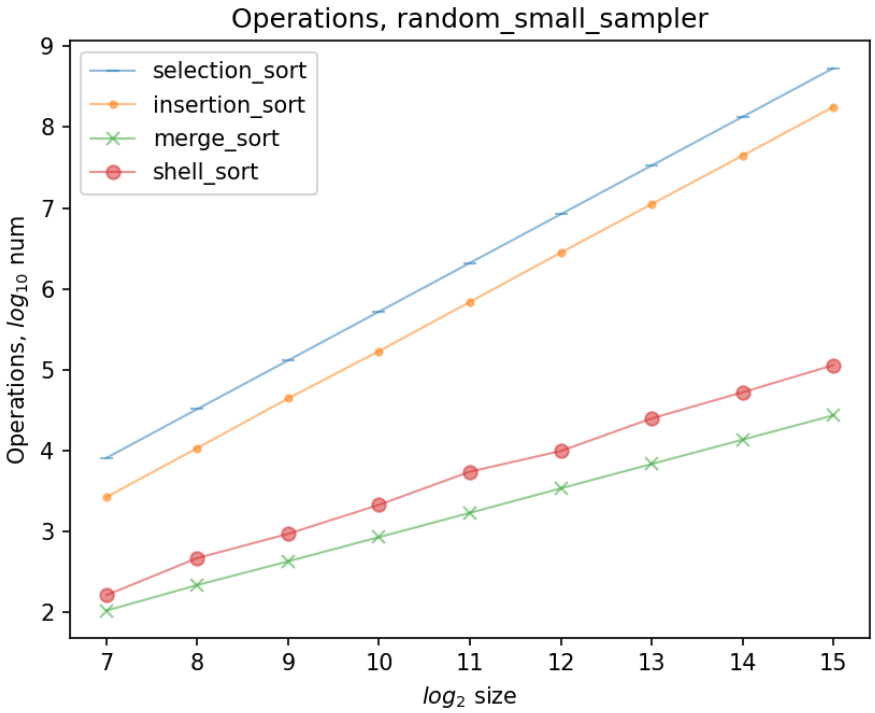

Time, decreasing_sampler
selection_sort
insertion_sort
merge_sort
shell_sort
Execution time, log_{10} ms
log_2 size
0
1
2
3
4
5
7
8
9
10
11
12
13
14
15


Operations, decreasing_sampler
selection_sort
insertion_sort
merge_sort
shell_sort
Operations, log_{10} num
log_2 size
2
3
4
5
6
7
8
9
7
8
9
10
11
12
13
14
15

Висновок:
Аналізуючи алгоритми Insertion sort та Selection sort можна помітити, що кількість порівнянь у найгіршому випадку у них буде співпадати. Однак цього разу Selection sort буде працювати швидше, і ця різниця стає все помітнішою із збільшенням розміру масиву. Однак обидва алгоритми програють Merge sort і Shell sort, які і потрібно застосовувати для масиву згенерованому у порядку спадання – і кількість порівнянь, і час роботи їх набагато меншим , ніж у двох інших. Різниця між Shell sort і Merge sort є незначна по часу, але Shell є трохи швидшим на великах масивах, але кількість операцій порівнянь відрізняється суттєво. Тому краще використовувати Shell sort.

Task4





Висновок:
При наявності масиву, випадково згенерованого тільки із чисел 1, 2 і 3, Shell sort, так як і у 1 випадку, є найоптимальнішим варіантом вибору - кількість його порівнянь і час роботи є найменшим. Insertion і selection sort поводять себе майже однаково (insertion працює трішки швидше) проте кількість порівнянь у них зростає із однаковою швидкістю. Merge sort працює швидше ніж Insertion і Selection sorts але повільніше ніж Shell sort. Також кількість порівнянь у Merge sort є найменшою.

Загальний висновок:
В залежності від якісного набору даних, чотири алгоритми показали різний ступінь ефективності, що дало розуміннння того, який алгоритм найкраще застосовувати, коли наявний відсортований, випадково згенерований або ж згенерований у порядку спадання масив. Експерименти також показали, що навіть якщо верхня межа у алгоритмів однакова (Selection I Insertion), то нижня межа може суттєво відрізнятися.

№ експ	Найоптимальніший	Найповільніший
1.	Shell sort	Selection sort
2.	Insertion sort	Selection sort
3.	Shell sort	Insertion sort
4.	Shell sort	Selection sort

Language: Python

Selection sort

```
#File: selection.py
# file represents algorithm of selection sort

def find(arr, idx, counter):
   min_index = idx
   min_value = arr[idx]
   for i in range(idx + 1, len(arr)):
      counter += 1
      if min_value > arr[i]:
         min_value = arr[i]
         min_index = i
   return min_index, counter

def selection_sort(arr):
   counter = 0
   for i in range(len(arr)):
      idx, counter = find(arr, i, counter)
      arr[idx], arr[i] = arr[i], arr[idx]
   return counter
```

Insertion sort

```
#File: insertion.py
# file represents algorithm of insertion sort

def put(arr, idx):
    counter = 0
    for i in range(idx - 1, -1, -1):
        if arr[i + 1] < arr[i]:
            arr[i + 1], arr[i] = arr[i], arr[i + 1]
            counter += 1
    return counter


def insertion_sort(arr):
    counter = 0
    for i in range(len(arr)):
        counter += put(arr, i)
    return counter
```

Merge sort

```
#File: merge.py
# file represents algorithm of merge sort

def combine(arr, left, mid, right):
    counter = 0
    idx1 = 0
    idx2 = 0
    arr_left = arr[left:mid + 1]
    arr_right = arr[mid + 1:right + 1]
    now = left
    while idx1 < len(arr_left) and idx2 < len(arr_right):
        counter += 1
        if arr_left[idx1] < arr_right[idx2]:
            arr[now] = arr_left[idx1]
            idx1 += 1
        else:
            arr[now] = arr_right[idx2]
            idx2 += 1
        now += 1
    while idx1 < len(arr_left):
        arr[now] = arr_left[idx1]
        now += 1
        idx1 += 1
```

```
    while idx2 < len(arr_right):
        arr[now] = arr_right[idx2]
        now += 1
        idx2 += 1
    return counter


def _merge_sort(arr, left, right):
    counter = 0
    if left == right:
        return
    mid = (left + right) // 2
    _merge_sort(arr, left, mid)
    _merge_sort(arr, mid + 1, right)

    counter += combine(arr, left, mid, right)
    return counter


def merge_sort(arr):
    return _merge_sort(arr, 0, len(arr) - 1)
```

Shell sort

```
#File: shell.py
# file represents algorithm of shell sort

def shell(arr, gap):
    counter = 0
    for i in range(gap, len(arr)):
        now = arr[i]
        j = i
        while j >= gap and arr[j - gap] > now:
            counter += 1
            arr[j] = arr[j - gap]
            j -= gap
        arr[j] = now
    return counter


def shell_sort(arr):
    counter = 0
    gap = len(arr) // 2
    while gap > 0:
        counter += shell(arr, gap)
        gap //= 2
    return counter
```