МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. Сікорського

Кафедра
інформатики та програмної інженерії
(повна назва кафедри, циклової комісії)

КУРСОВА РОБОТА

з Основ програмування
(назва дисципліни)
на тему: Упорядкування масивів

Студента 1 курсу, групи ІП-11
Головні Олександра Ростиславовича

Спеціальності 121 «Інженерія програмного забезпечення»

Керівник Головченко Максим Миколайович
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Кількість балів: _____
Національна оцінка _____

Члени комісії

_____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

_____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Київ- 2022_рік

КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. Сікорського

(назва вищого навчального закладу)

Кафедра інформатики та програмної інженерії

Дисципліна Основи програмування

Напрям "ІПЗ"

Курс 1 Група ІП-11 Семестр 2

ЗАВДАННЯ

на курсову роботу студента
Головні Олександра Ростиславовича

(прізвище, ім'я, по батькові)

1. Тема роботи Упорядкування масивів

2. Строк здачі студентом закінченої роботи 12.06.2022

3. Вихідні дані до роботи

4. Зміст розрахунково-пояснювальної записки (перелік питань, які підлягають розробці)

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Дата видачі завдання 10.02.2022

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів курсової роботи	Термін виконання етапів роботи	Підписи керівника, студента
1.	Отримання теми курсової роботи	10.02.2022	
2.	Підготовка ТЗ	02.05.2022	
3.	Пошук та вивчення літератури з питань курсової роботи	03.05.2022	
4.	Розробка сценарію роботи програми	04.05.2022	
6.	Узгодження сценарію роботи програми з керівником	04.05.2022	
5.	Розробка (вибір) алгоритму рішення задачі	04.05.2022	
6.	Узгодження алгоритму з керівником	04.05.2022	
7.	Узгодження з керівником інтерфейсу користувача	05.05.2022	
8.	Розробка програмного забезпечення	07.05.2022	
9.	Налагодження розрахункової частини програми	07.05.2022	
10.	Розробка та налагодження інтерфейсної частини програми	07.05.2022	
11.	Узгодження з керівником набору тестів для контрольного прикладу	25.05.2022	
12.	Тестування програми	26.05.2022	
13.	Підготовка пояснювальної записки	05.06.2022	
14.	Здача курсової роботи на перевірку	12.06.2022	
15.	Захист курсової роботи	15.06.2022	

Студент _____

(підпис)

Керівник _____ Головченко Максим Миколайович

(підпис) (прізвище, ім'я, по батькові)

"___" _____20__ р.

АНОТАЦІЯ

Пояснювальна записка до курсової роботи: 55 сторінок, 19 рисунків, 10 таблиць, 4 посилання.

Об’єкт дослідження: задача упорядкування масивів дійсних чисел.

Мета роботи: дослідження методів сортування масивів, , створення програмного забезпечення для сортування.

Вивчено метод розробки програмного забезпечення з використанням принципів ООП. Приведені змістовні постановки задач, їх індивідуальні математичні моделі, а також описано детальний процес розв’язання кожної з них.

Виконана програмна реалізація алгоритмів сортування масивів дійсних чисел.

1. ЗМІСТ

ВСТУП

Дана робота присвячена вивченню розробки програмного забезпечення з використанням парадигми ООП, і стосується написання програмного забезпечення для упорядкування масисів. Задача полягає у графічному представленні сортування.

1. ПОСТАНОВКА ЗАДАЧІ

Розробити програмне забезпечення, що генерує випадковим чином масив та упорядковує його наступними методами:

а) метод сортування злиттям;

б) метод швидкого сортування;

в) метод інтроспективного сортування;

Вхідними даними для даної роботи є згенерований масив чисел:

Програмне забезпечення повинно оброобробляти масив розмірність якого знаходиться в межах – не менше 100 елементів, максимально до 50000

Вихідними даними для даної роботи являється посортований масив, який виводяться на екран та записується в спеціальний файл. Програмне забезпечення повинно сортувати за умови: який обраний метод. Якщо це не так, то програма повинна вивести відповідне повідомлення. Якщо розмірність системи не знаходиться в межах, то програмне забезпечення повинно теж вивести відповідне повідомлення.

2. ТЕОРЕТИЧНІ ВІДОМОСТІ

а) метод сортування злиттям;

б) метод швидкого сортування;

в) метод інтроспективного сортування;

1. Метод сортування злиттям

Сортування злиттям – це один з базових алгоритмів інформатики, сформульований ще в 1945 році великим математиком Джоном фон Нейманом. Беручи участь у «Манхеттенському проекті», Нейман зіткнувся з необхідністю ефективної обробки величезних масивів даних. Розроблений ним метод використовував принцип «розділяй і володарюй», що дозволило істотно скоротити час, необхідне для роботи.

Метод сортування злиттям знаходить застосування в задачах сортування структур, що мають упорядкований доступ до елементів, наприклад, масивів, списків, потоків.

При обробці вихідний блок даних дробиться на маленькі складові, аж до одного елемента, який по суті вже є відсортованим списком. Потім відбувається зворотний збірка в правильному порядку.

В основі цього способу сортування лежить злиття двох упорядкованих ділянок масиву в одну впорядковану ділянку іншого масиву. Злиття двох упорядкованих послідовностей можна порівняти з перебудовою двох колон солдатів, вишикуваних за зростом, в одну, де вони також розташовуються за зростом. Якщо цим процесом керує офіцер, то він порівнює зріст солдатів, перших у своїх колонах і вказує, якому з них треба ставати останнім у нову колону, а кому залишатися першим у своїй. Так він вчиняє, поки одна з колон не вичерпається — тоді решта іншої колони додається до нової.

Під час сортування в дві допоміжні черги з основної поміщаються перші дві відсортовані підпослідовності, які потім зливаються в одну і результат

записується в тимчасову чергу. Потім з основної черги беруться наступні дві відсортовані підпослідовності і так доти, доки основна черга не стане порожньою. Після цього послідовність з тимчасової черги переміщається в основну чергу. І знову продовжується сортування злиттям двох відсортованих підпослідовностей. Сортування триватиме доти, доки довжина відсортованої підпослідовності не стане рівною довжині самої послідовності.

Часова складність алгоритму – $O(n\ log\ n)$.

Ємнісна складність алгоритму – $O(n)$

2. Метод швидкого сортування

Швидке сортування, **сортування Хоара** (quicksort) Один з найшвидших відомих універсальних алгоритмів сортування масивів: в середньому $O(nlog\ n)$ обмінів при упорядкуванні n елементів; через наявність ряду недоліків на практиці зазвичай використовується з деякими доробками.

QuickSort є істотно поліпшеним варіантом алгоритму сортування за допомогою прямого обміну (його варіанти відомі як «Бульбашкове сортування» та «шейкерне сортування»), відомого, в тому числі, своєю низькою ефективністю. Принципова відмінність полягає в тому, що в першу чергу проводяться перестановки на найбільш можливій відстані і після кожного проходу елементи діляться на дві незалежні групи. Цікавий факт: поліпшення самого неефективного прямого методу сортування дало в результаті один з найбільш ефективних поліпшених методів.

Загальна ідея алгоритму полягає в наступному:

Вибрати з масиву елемент, який називають опорним. Це може бути будь- який з елементів масиву. Від вибору опорного елемента не залежить коректність алгоритму, але в окремих випадках може сильно залежати його ефективність.

Порівняти всі інші елементи з опорним і переставити їх у масиві так,

щоб розбити масив на три безперервних підмасиви, наступні один за одним: «менші опорного», «рівні» і «більші».

Для підмасивів «менших» і «більших» значень виконати рекурсивно ту ж послідовність операцій, якщо довжина відрізка більше одиниці.

На практиці масив зазвичай ділять не на три, а на дві частини: наприклад, «менші опорного» та «рівні і більші»; такий підхід в загальному випадку ефективніше, так як спрощує алгоритм поділу.

Вибір опорного елемента

У ранніх реалізаціях, як правило, опорним вибирався перший елемент, що знижувало продуктивність на відсортованих масивах. Для поліпшення ефективності може вибиратися середній, випадковий елемент або (для великих масивів) медіана першого, середнього і останнього елементів. Медіана всієї послідовності є оптимальним опорним елементом, але її обчислення занадто трудомістке для використання в сортуванні.

Оцінка складності алгоритму

Ясно, що операція поділу масиву на дві частини відносно опорного елемента займає час $O(n)$. Оскільки всі операції поділу, що проробляються на одній глибині рекурсії, обробляють різні частини вихідного масиву, розмір якого постійний, сумарно на кожному рівні рекурсії буде потрібно також $O(n)$ операцій. Отже, загальна складність алгоритму визначається лише кількістю поділів, тобто глибиною рекурсії. Глибина рекурсії, в свою чергу, залежить від поєднання вхідних даних і способу визначення опорного елемента.

Найкращий випадок

У найбільш збалансованому варіанті при кожній операції поділу масив ділиться на дві однакові (плюс-мінус один елемент) частини, отже, максимальна глибина рекурсії, при якій розміри оброблюваних підмасивів досягнуть 1, складе $logn$. В результаті кількість порівнянь, що здійснюються швидким сортуванням, дорівнює значенню рекурентного виразу $T(n) = 2T(n/2) + cn$, що дає загальну складність алгоритму $O\ (nlogn)$.

Середній випадок

Середню складність при випадковому розподілі вхідних даних можна оцінити лише імовірнісно.

Перш за все слід зазначити, що в дійсності не обов'язково, щоб опорний елемент щоразу ділив масив на дві однакові частини. Наприклад, якщо на кожному етапі буде відбуватися поділ на масиви довжиною 75% і 25% від вхідного, глибина рекурсії дорівнюватиме $\log_{4/3}$n, а це як і раніше дає складність O(nlogn). Взагалі, при будь-якому фіксованому співвідношенні між лівою і правою частинами поділу складність алгоритму буде тією ж, тільки з різними константами.

Будемо вважати «вдалим» поділом такий, при якому опорний елемент виявиться серед центральних 50% елементів розділяється частини масиву; ясно, ймовірність успіху при випадковому розподілі елементів складає 0,5. При вдалому поділі розміри виділених підмасивів складуть не менше 25% і не більше 75% від початкового. Оскільки кожен виділений підмассів також матиме випадковий розподіл, всі ці міркування можна застосувати до будьякого етапу сортування і будь-якого вихідного фрагменту масиву.

Вдалий поділ дає глибину рекурсії не більше $log\ 4\ /\ 3n$. Оскільки ймовірність удачі дорівнює 0,5, для отримання k вдалих розділень в середньому потрібно 2k рекурсивних викликів, щоб опорний елемент k раз виявився серед центральних 50% масиву. Застосовуючи ці міркування, можна зробити висновок, що в середньому глибина рекурсії не перевищить

$2\ log4\ /\ 3n$, що дорівнює $O\ (nlogn)$. А оскільки на кожному рівні рекурсії як і раніше виконується не більше $O\ (n)$ операцій, середня складність складе $O\ (nlogn)$.

Найгірший випадок

У самому незбалансованому варіанті кожен поділ дає два подмасиви розмірами 1 і $n\ -\ 1$, тобто при кожному рекурсивному виклику більший масив буде на 1 коротше, ніж в попередній раз. Таке може статися, якщо в якості опорного на кожному етапі буде обраний елемент або найменший, або найбільший з усіх оброблюваних. При простому виборі опорного елемента - першого або останнього в масиві, - такий ефект дасть вже відсортований (в прямому або зворотному порядку) масив, для середнього або будь-якого іншого фіксованого елемента «масив гіршого випадку» також може бути спеціально підібраний. У цьому випадку буде потрібно n - 1 операцій поділу, а загальний час роботи складе $O(n2)$ операцій, тобто сортування буде виконуватися за квадратичний час.

Але кількість обмінів і, відповідно, час роботи - це не найбільший його недолік. Гірше те, що в такому випадку глибина рекурсії при виконанні алгоритму досягне n, що буде означати n-кратне збереження адреси повернення і локальних змінних процедури поділу масивів. Для великих значень n найгірший випадок може привести до вичерпання пам'яті (переповнення стека) під час роботи програми.

Часова складність алгоритму в усіх випадках – $O(n2)$ - найгірший, $O(nlogn)$ - середній.

Ємнісна складність алгоритму – $O(n)$ всього, $O(logn)$ додатково.

3. Метод інтроспективного сортування;

Introsort або **інтроспективне сортування** – алгоритм сортування, запропонований Девідом Мюссером в 1997 році. Він використовує швидке сортування і переключається на пірамідальне сортування, коли глибина рекурсії перевищить деякий заздалегідь встановлений рівень (наприклад, логарифм від числа елементів вхідного масиву). Цей підхід поєднує в собі переваги обох методів з гіршим випадком $O(n\ log\ n)$ і швидкодією, яку можна порівняти з швидким сортуванням. Так як обидва алгоритми використовують порівняння, цей алгоритм також належить класу сортувань на основі порівнянь.

У швидкому сортуванні одна з найважливіших операцій - вибір опорного елементу (елементу, щодо якого розбивається масив). Найпростіший алгоритм вибору опорного елементу - взяття першого або останнього елементу масиву за опорний елемент загрожує поганою ефективністю на відсортованих або майже відсортованих даних. Ніклаус Вірт запропонував використовувати серединний елемент для запобігання цьому випадку, який деградує до $O(n^2)$ при невдалих вхідних даних. Алгоритм вибору опорного елементу «медіана з трьох» вибирає опорний елемент середній з першого, середнього і останнього елементів масиву. Однак, незважаючи на те, що він працює добре на більшості вхідних даних, все ж можливо знайти такі вхідні дані, які сильно сповільнять цей алгоритм сортування. Такі дані потенційно можуть використовуватися зловмисниками. Наприклад, зловмисники можуть посилати такий масив Веб- серверу, домагаючись відмови в обслуговуванні.

Мюссера з'ясував, що на гіршому наборі даних для алгоритму швидкого сортування «медіана з трьох» (розглядався масив із 100 тисяч елементів) introsort працює приблизно в 200 разів швидше.

Він також оцінив ефект від кеша в разі сортування Роберта Седжвика, коли невеликі діапазони упорядковано в кінці одиночним проходом сортування

вставками. Він з'ясував, що такий підхід може подвоїти кількість промахів кеша, але його продуктивність при використанні двосторонньої черги значно краще, і його слід залишити для бібліотек шаблонів, хоча б тому, що виграш в інших випадках не великий.

У реалізації Стандартної бібліотеки шаблонів C++ для нестійкого сортування використовується підхід Мюссера з контролем глибини рекурсії, для перемикання на алгоритм пірамідального сортування, вибір опорного елемента як медіани з трьох і в кінці застосовується алгоритм сортування вставками Кнута для послідовностей, що містять менш ніж 16 елементів.

Часова складність алгоритму – $O(n\ log\ n)$.

Ємнісна складність алгоритму – $0(n)$ всього, $O(1)$ допоміжна змінна.

3. ОПИС АЛГОРИТМІВ

Перелік всіх основних змінних та їхнє призначення наведено в таблиці 3.1.

Таблиця 3.1 – Основні змінні та їхні призначення

Змінна	Призначення
a	Масив що передається у функції
start	Початок елементів масива
end	Кінець елементів масива
L	Ліва частина масива
R	Права частина масива
n,m,k, i,j, iRoot,iEnd,tmp	Допоміжні змінні
d	Рівень (для Introsort)

Загальний алгоритм

1. ПОЧАТОК
2. Зчитати розмірність системи.
3. Зчитати спосіб сортування

Якщо поточний розмір масиву – вірно записане число, ТО записати його в відповідну змінну, ІНАКШЕ видати повідомлення про можливу помилку

4. ЯКЩО обраний спосіб сортування злиттям, ТО обробити дані згідно алгоритму сортування злиттям (Пункт 1.2)
5. ЯКЩО обране швидке сортування, ТО обробити дані згідно алгоритму швидкого сортування(Пункт 1.3)
6. ЯКЩО обране інтроспективне сортування, ТО обробити дані згідно алгоритму інтроспективного сортування(Пункт 1.4)
7. ЯКЩО обраний метод сходиться для вхідних даних, ТО:
   7.1. Вивести рішення системи.
   7.2. Записати систему та її рішення у файл.
8. КІНЕЦЬ

Алгоритм сортування злиттям

Невідсортований список послідовно ділиться на N списків, де кожен включає один «невідсортований» елемент, а N — це число елементів в оригінальному масиві.

Списки послідовно зливаються групами по два, створюючи нові відсортовані списки до тих пір, поки не з'явиться один фінальний відсортований список.

ПОЧАТОК

1. Якщо довжина масиву == 1 або ==0, то:
   Повернути масив
2. Розділити масиви на дві частини(Ліву та праву)
   2.1 L = рекурсивна функція mergesort(Масив[:довжина//2]
   R = рекурсивна функція mergesort(Масив[довжина//2]
   (C – нулювий масив з довжиною (len(L)+len(R))
3. Поки n<довжини та m<довжини: (Де m, n, k =0)
   Якщо L[n] <=R[n]:
   C[k] = L[n]
   n +=1
   Інакше :
   C[k] = R[n]
   m +=1
   k += 1
4. Поки n<довжини масиву L:
   C[k] = L[n]
   n +=1
   k += 1
5. Поки m<довжини масиву R:
   C[k] = R[n]
   m +=1
   k +=1

6. Для і у діапазоні довжини масиву А:

   A[i] = C[i]

7. Повернути А(наш масив)

1.3 Алгоритм швидкого сортування

Вибирається один опорний елемент.

Всі елементи, які менші за опорний, переміщаються зліва від нього, інші — направо. Це називається операцією розбиття.

Рекурсивно 2 попередні кроки повторюються в кожному новому списку, де нові опорні елементи будуть менші та більші за опорний елемент.

ПОЧАТОК

1. Визначити опорний елемент x (середина)

   1.2 Здійснити перестановку

   1.3 Якщо у лівій частині більше 1 елемента, то:

   QuickSort(ліва частина)

   1.4 Якщо у правій частині більше 1 елемента, то:

   QuickSort(права частина)

2. Перестановка

   2.1 Повторювати

   2.1.1 Поки a[L]<x:

   L +=1;

   2.1.2 Поки a[R]>x:

   R -=1;

   2.1.3Якщо L<=R то:

   2.1.3.1 Переставити (a[l], a[r])

   L +=1

   R -=1

Доки (L>R)

1.4 Алгоритм інтроспективного сортування

Інтроспективне сортування — використовує швидке сортування та перемикається на пірамідальне сортування, коли глибина рекурсії перевищить деякий заздалегідь встановлений рівень

ПОЧАТОК

1. Для і у range(0,15): – наш рівень

    1.1 Introsort(a, d, start, end) – a -масив, d – рівень

    n = end – start

    1.2 Якщо n<=1:

    Повернути

    1.3 Інакше якшо d==0:

    IntroHS(a, start,end)

    1.4 Інакше:

    p = перестановка(a, start, end)

    Introsort(a, d-1, start, p)

    Introsort(a, d-1, p+1, end)

    1.5 Повернути a

2. IntroHS:

    b = a[start:end]

    2.1 heapSort(b)

    2.2 Для і у range(0, довжина b):

    a[start+i] = b[i]

3. heapSort:

    END = len(a)

    3.1 Для К у range(Округлення(END/2) -1, -1,-1):

        Heapify(a, END, k):

    3.2 Для К у range(END, -1, -1):

        Swap(a, 0, k-1)

        Heapify(a, k -1, 0)

4. Heapify (a, iEnd, iRoot):

iL = 2*iRoot + 1

iR = 2*iRoot + 2

4.1 Якщо iR < iEnd , то:

4.1.1 Якщо a[iRoot] >= a[iL] і a[iRoot] >= a[iR], то:

Повернути

4.1.2 Інакше:

Якщо , то:

j = iL

Інакше:

j = iR

swap(a, iRoot, j)

heapify(a, iEnd, j)

4.2 Інакше якщо iL < iEnd, то:

4.2.1 Якщо a[iRoot] >= a[iL], то:

Повернути

4.2.2 Інакше:

Swap(a, iRoot, iL)

Heapify(a, iEnd, iL)

4.3 Інакше:

Повернути

5. Звичайний swap(a, i, j):

tmp = a[i]

a[i] = a[j]

a[j] = tmp

4 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Діаграма класів програмного забезпечення

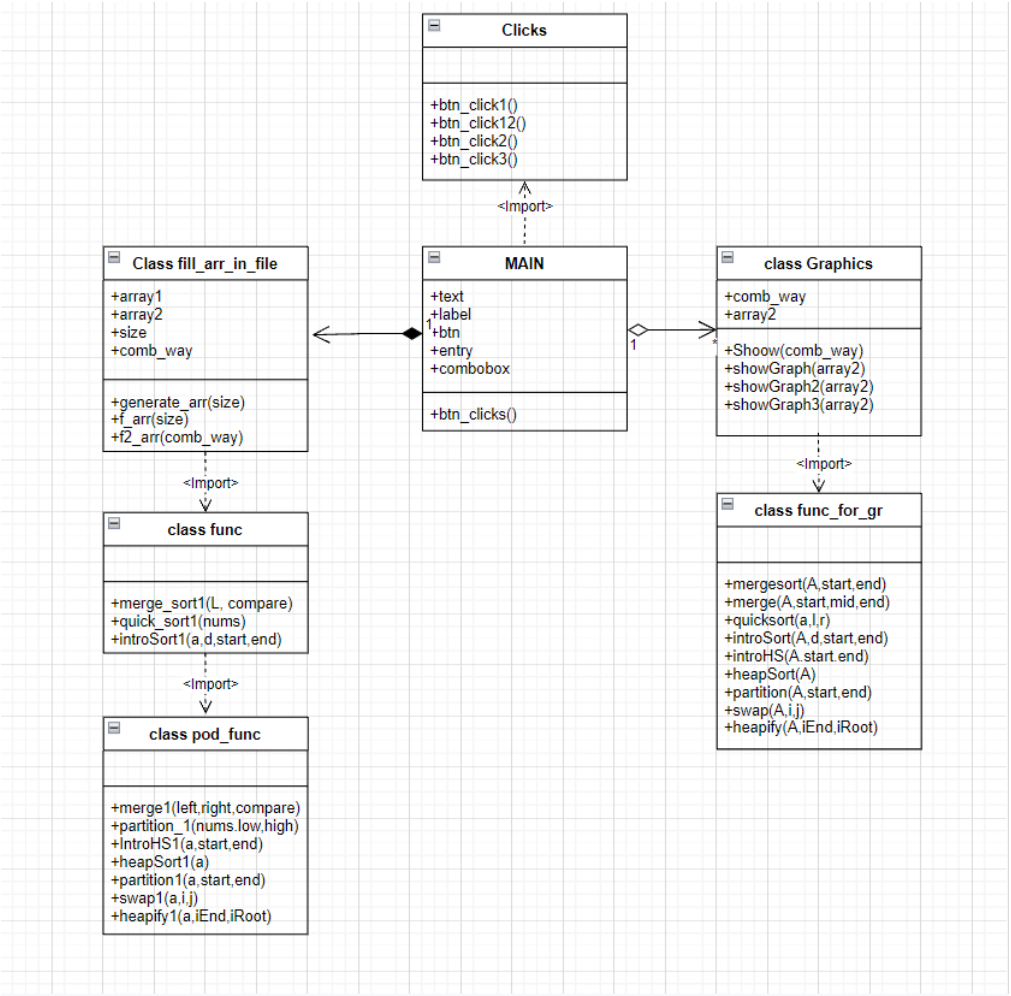


Рисунок 2.1 – Діаграма класів

4.2. Опис методів частин програмного забезпечення

4.2.1. Стандартні методи

Таблиця 1.1 – Стандартні методи

№ п/п	Назва классу	Назва функції	Призначення функції	Опис вхідних параметрів	Опис вихідних параметрів	Заголовний файл
1	math	floor	Округлення в меншу сторону	Дійсне число	Ціле число	math
2	-	len	Отримання довжини(кількість елементів)	Строка, кортеж, список, діапазон	Кількість елементів	-
3	-	range	Об’єкт ітерується, можна згенерувати посл. чисел з кроком частоти	Об’єкт	Ітерований об’єкт	-
4	-	abs	Знаходження значення по модулю	Дійсне число	Абсолютне значення	-

4.2.2. Користувацькі методи

Таблиця 1.2 – Користувацькі методи

№ п/п	Назва классу	Назва функції	Призначення функції	Опис вхідних параметрів	Опис вихідних параметрів	Заголовний файл
1	clicks	btn_click1	Підтвердження та перевірка розміру масиву	entry_n	-	main
2	clicks	btn_click12	Вивід першого файлу в комірку для текста	text1	-	main
3	clicks	btn_click2	Вивід другого файлу(посортований масив) в комірку для текста	text1	-	main
4	clicks	btn_click3	Підтвердження вибору способу сортування	entry_n, combobox	-	main
5	func	merge_sort1	Сортування злиттям	L, compare	Посортований масив	func_for_arr
6	func	quick_sort1	Швидке сортування	nums	Посортований масив	func_for_arr
7	func	introSort1	Інтроспективне сортування	a, d, start, end	Посортований масив	func_for_arr
8	pod_func	merge1	Допоміжна функція для сортування злиттям	left, right, compare	-	func_for_func

9	pod_func	partition_1	Допоміжна функція для швидкого сортування(Перестановка)	nums, low, high	-	func_for_func
10	pod_func	introHS1	Допоміжна функція для інтроспективного сортування	a, start, end	-	func_for_func
11	pod_func	heapSort1	Пірамідальне сортування	a	-	func_for_func
12	pod_func	swap1	Звичайний свап	a, i, j	-	func_for_func
13	pod_func	heapify1	Допоміжна функція для пірамідального сортування	a,iEnd, iRoot	-	func_for_func
14	Graphics	Shoow	Вибір графіку сортування	comb_way	-	Graphics
15	Graphics	showGraph	Візуалізація 1 способу	array2	-	Graphics
16	Graphics	showGraph2	Візуалізація 2 способу	array2	-	Graphics
17	Graphics	showGraph3	Візуалізація 3 способу	array2	-	Graphics
18	fill_arr_in_file	generate_arr	Генерація масиву	size	Згенерований масив	fill_arr_in_file
19	fill_arr_in_file	f_arr	Запис першого масиву у файл	size	-	fill_arr_in_file
20	fill_arr_in_file	f2_arr	Запис другого масиву у файл	comb_way	-	fill_arr_in_file
21	func_for_gr	mergesort	Модифіковане сортування злиттям для візуалізації	A, start, end	-	func_for_graphics

22	func_for_gr	merge	Допоміжна функція для mergesort	A, start, mid, end	-	func_for_graphics
23	func_for_gr	quicksort	Модифіковане швидке сортування для візуалізації	a, l, r	-	func_for_graphics
24	func_for_gr	introSort	Модифіковане інтроспективне сортування для візуалізації	A, d, start, end	-	func_for_graphics
25	func_for_gr	introHS	Допоміжна функція для інтроспективного сортування	A, start, end	-	func_for_graphics
26	func_for_gr	heapSort	Модифіковане пірамідальне сортування для візуалізації	A	-	func_for_graphics
27	func_for_gr	heapify	Допоміжна функція для пірамідального сортування	A, iEnd, iRoot	-	func_for_graphics

5. ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План тестування

Складемо план тестування програмного забезпечення, за допомогою якого протестуємо весь основний функціонал та реакцію на виключні ситуації

а) Тестування правильності введених значень.
   1) Тестування при введенні некоректних символів.
   2) Тестування при введенні замалих та завеликих значень.

б) Тестування коректності роботи методів 1,2,3.
   1) Перевірка коректності роботи методу 1.
   2) Перевірка коректності роботи методу 2.
   3) Перевірка коректності роботи методу 3.

Приклади тестування

1) Тестування при введенні некоректних символів.

Таблиця 5.1 - Приклад роботи програми при введенні некоректних символів

Мета тесту	Перевірити можливість введення некоректних даних
Початковий стан програми	Відкрите вікно програми
Вхідні дані	asd d46a xzzx s 234a
Схема проведення тесту	Заповнення поля для розмірності масиву
Очікуваний результат	Повідомлення про помилку формату даних
Стан програми після проведення випробувань	Видано помилку «Введіть число(від 100 до 50000»

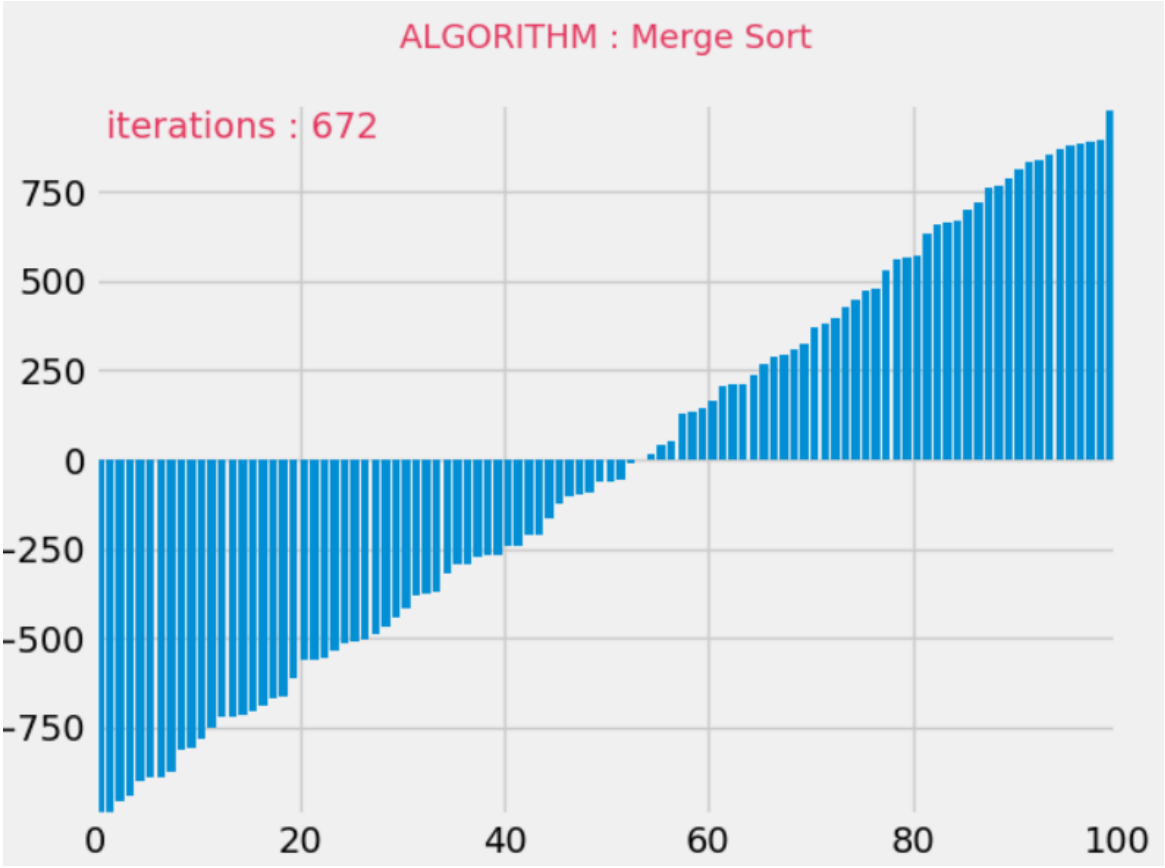
2) Тестування при введенні замалих та завеликих значень.

Таблиця 5.2 - Приклад роботи програми замалих та завеликих значень.

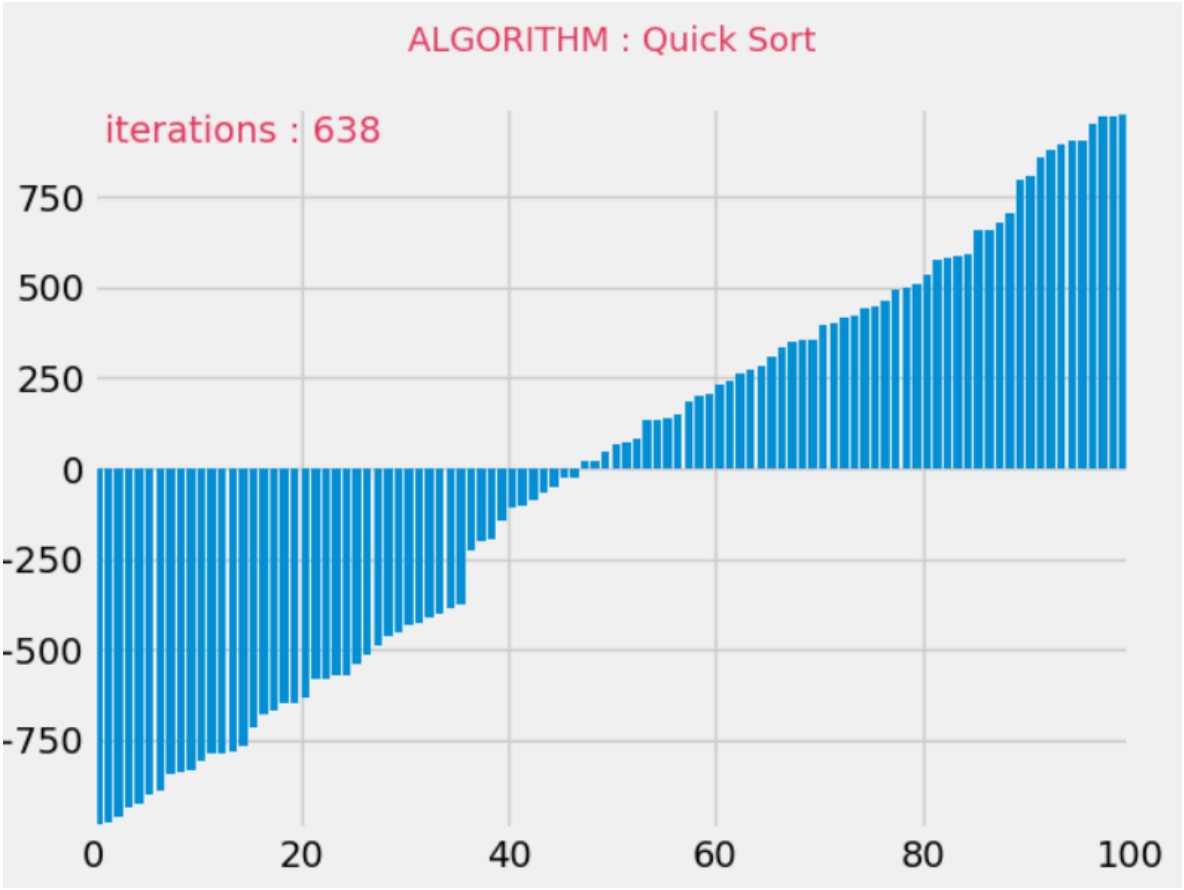
Мета тесту	Перевірити можливість введення замалих та завеликих значень.
Початковий стан програми	Відкрите вікно програми
Вхідні дані	33 50001
Схема проведення тесту	Заповнення поля для розмірності масиву
Очікуваний результат	Повідомлення про помилку формату даних
Стан програми після проведення випробувань	Видано помилку «Розмірність повинна бути від 100 до 50000»

Тестування коректності роботи методів 1,2,3.

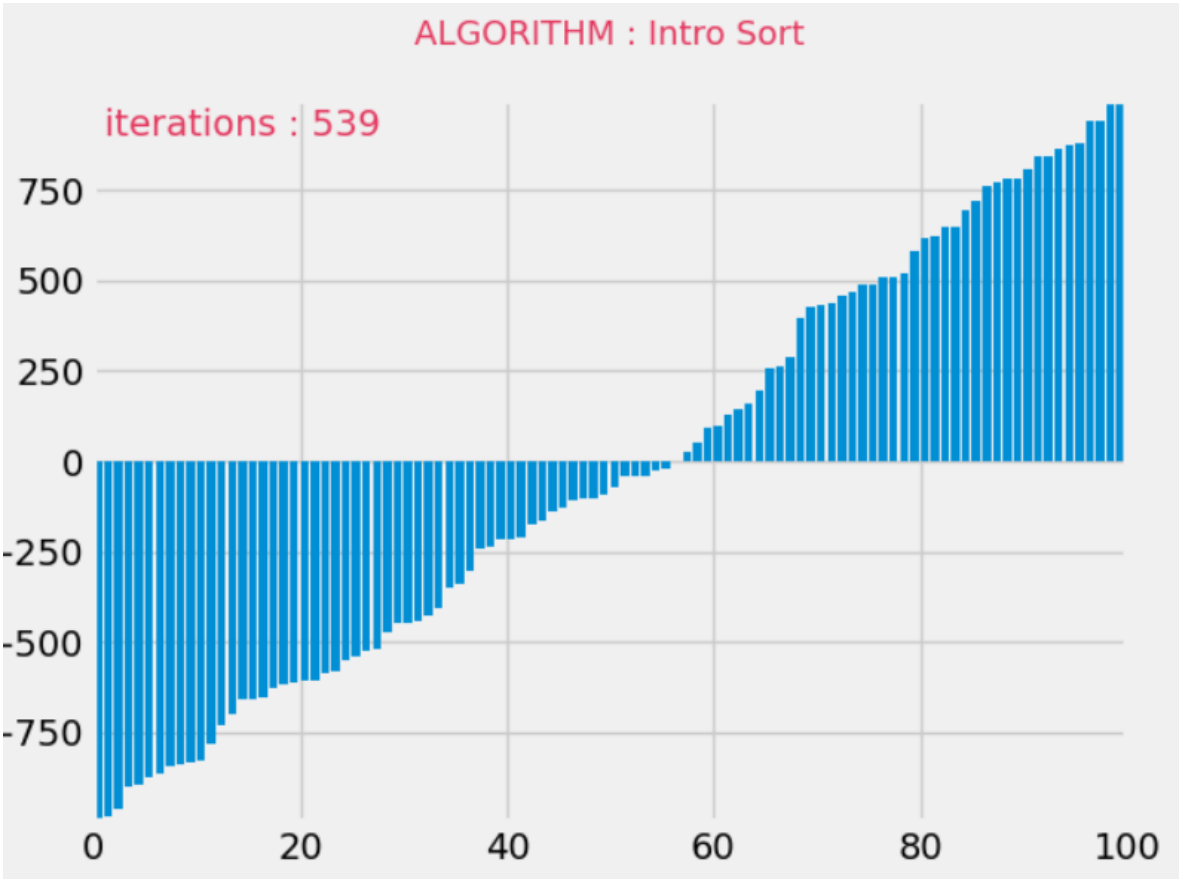
1) Перевірка коректності роботи методу 1.



2) Перевірка коректності роботи методу 2



3) Перевірка коректності роботи методу 3.



6. ІНСТРУКЦІЯ КОРИСТУВАЧА

Робота з програмою

Після запуску виконавчого файлу з розширенням *.exe, відкривається головне вікно програми (Рисунок 6.1).

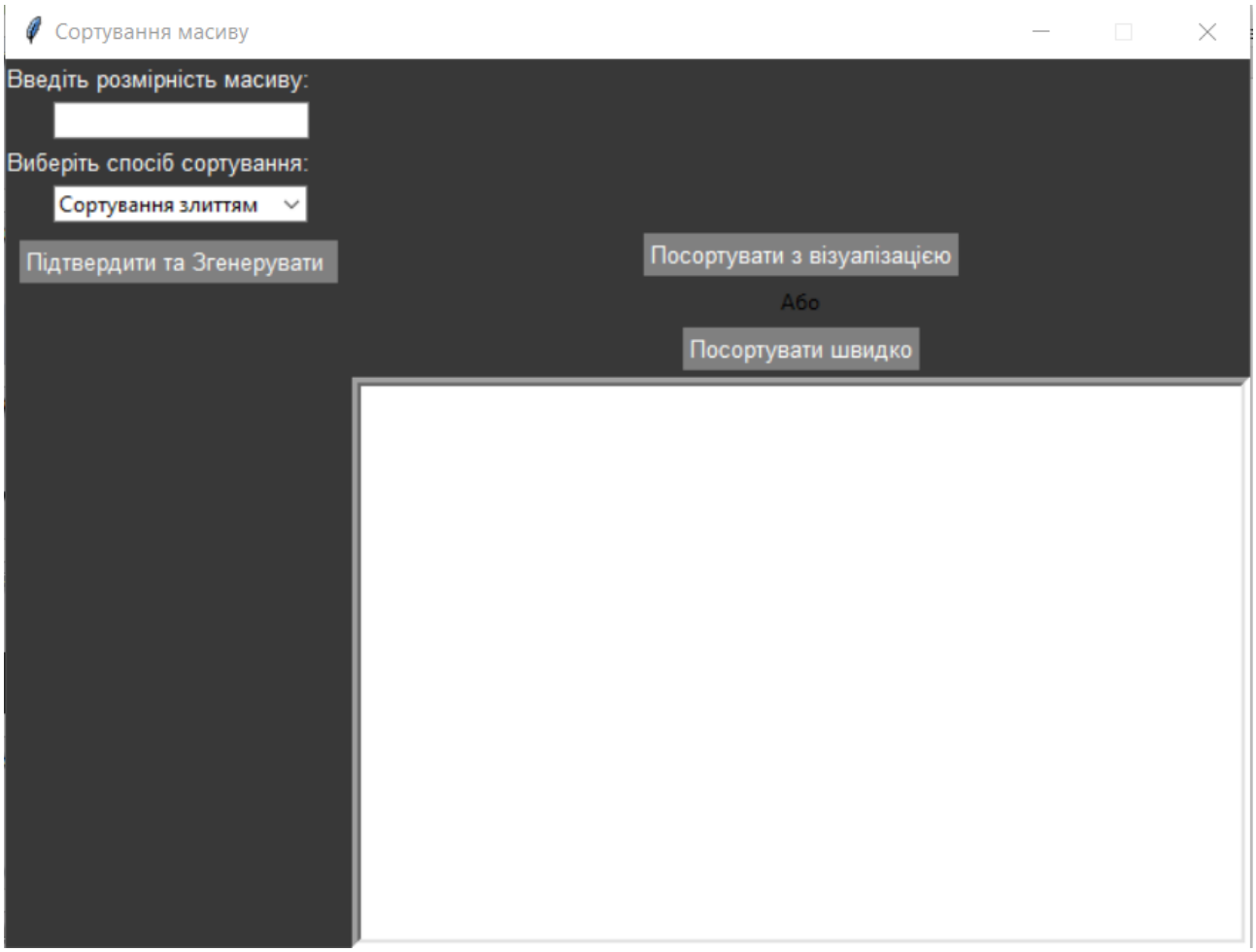


Рисунок 6.1 – Головне вікно програми

Далі за допомогою поля для введення тексту. з назвою «Розмірність масиву» шляхом введення числа з клавіатури необхідно виставити розмір масиву, що буде оброблятися програмою та вибрати(за допомогою комбобокса) відповідний спосіб сортування (рисунок 6.2):

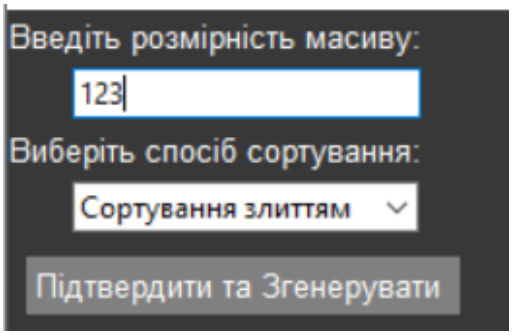


Рисунок 6.2 – Вибір необхідного розміру та способу сортування

Після підтвердження є вибір сортування з візуалізацією або швидко. В будь якому випадку при завершенні сортування масив буде посортовано, виведено на екран програми та збережений у відповідний файл(рисунок 3.3)

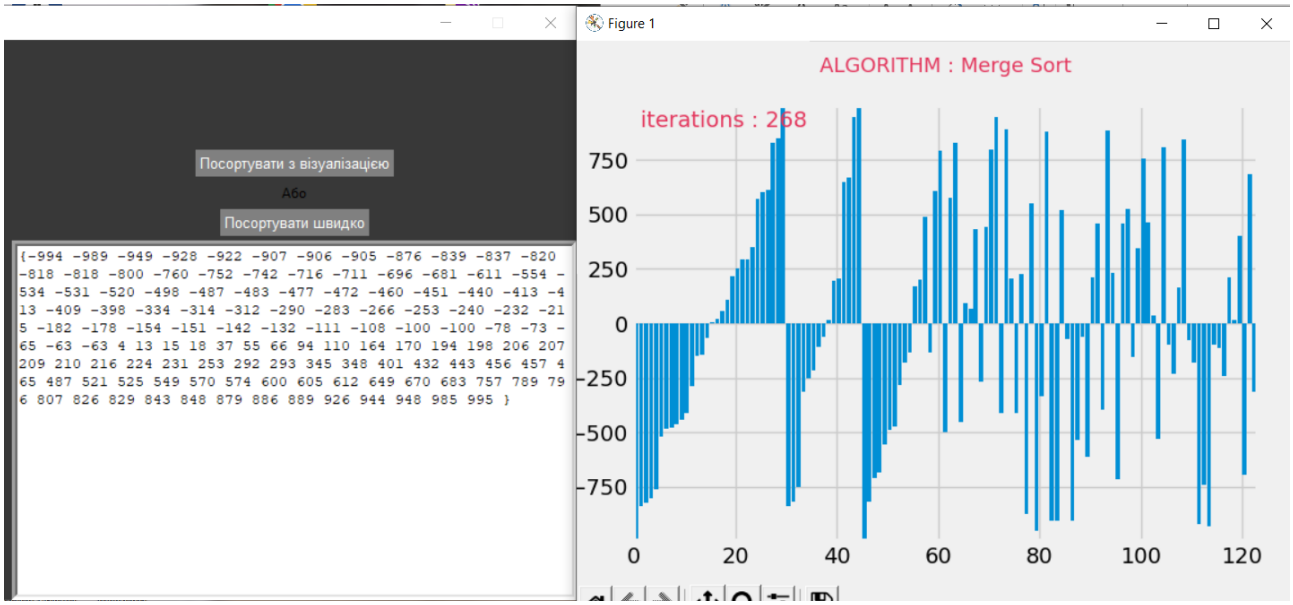


Рисунок 6.3 – Сортування масиву

Формат вхідних та вихідних даних

Користувачем на вхід програми подається розмірність масиву(натуральне число) та вибір способу сортування

Результатом виконання програми є посортований масив, що виводиться на екран та записується у файл

Системні вимоги

Системні вимоги до програмного забезпечення наведені в таблиці 6.1.

Таблиця 6.1 – Системні вимоги програмного забезпечення

	Мінімальні	Рекомендовані
Операційна система	Windows® XP/Windows Vista/Windows 7/ Windows 8/Windows 10 (з останніми обновленнями)	Windows 7/ Windows 8/Windows 10 (з останніми обновленнями)
Процесор	Intel® Pentium® III 1.0 GHz або AMD Athlon™ 1.0 GHz	Intel® Pentium® D або AMD Athlon™ 64 X2
Оперативна пам'ять	256 MB RAM (для Windows® XP) / 1 GB RAM (для Windows Vista/Windows 7/ Windows 8/Windows 10)	2 GB RAM
Відеоадаптер	Intel GMA 950 з відеопам'яттю об'ємом не менше 64 МБ (або сумісний аналог)	
Дисплей	800x600	1024x768 або краще
Приклади введення	Клавіатура, комп’ютерна миша	
Додаткове програмне забезпечення	Pip, tkinter, matplotlib	

7. АНАЛІЗ І УЗАГАЛЬНЕННЯ РЕЗУЛЬТАТІВ

Головною задачею курсової роботи була реалізація програми для сортування масиву наступними способами:

а) метод сортування злиттям;

б) метод швидкого сортування;

в) метод інтроспективного сортування;

Критичні ситуації у роботі програми виявлені не були. Під час тестування було виявлено, що більшість помилок виникало тоді, коли користувачем вводилися неправильні вхідні дані. Тому всі дані, які вводить користувач, ретельно провіряються на валідність і лише потім подаються на обробку програмі.

Для перевірки та доведення достовірності результатів виконання програмного забезпечення скористаюся достовірним сайтом https://www.coderstool.com/:

(Зрозуміло, що усі малюнки будуть показані у відповідний момент часу роботи алгоритму, щоб показати принцип способу, а результатом роботи завжди буде посортована послідовність чисел)

а) Метод сортування злиттям.

Результат виконання способу наведено на рисунку 7.1:

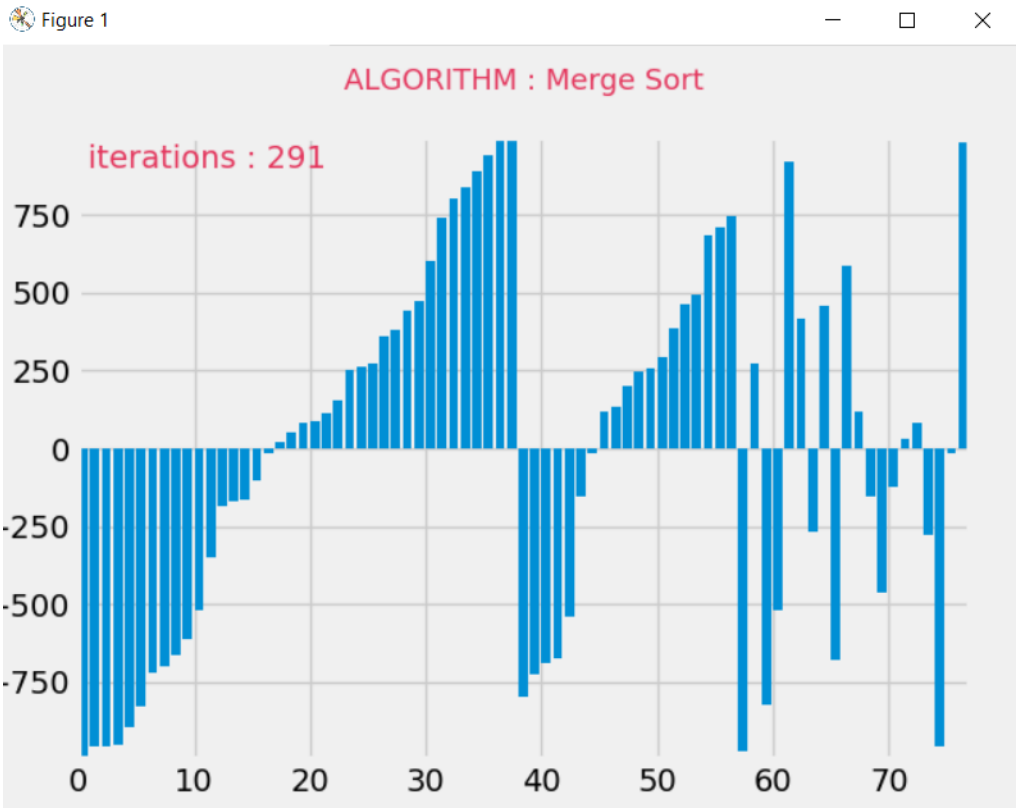


Рисунок 7.1 – Результат виконання сортування злиттям

Оскільки результати виконання збігається з результатом данного сайту(рисунок 7.2), то данне сортування працює вірно.

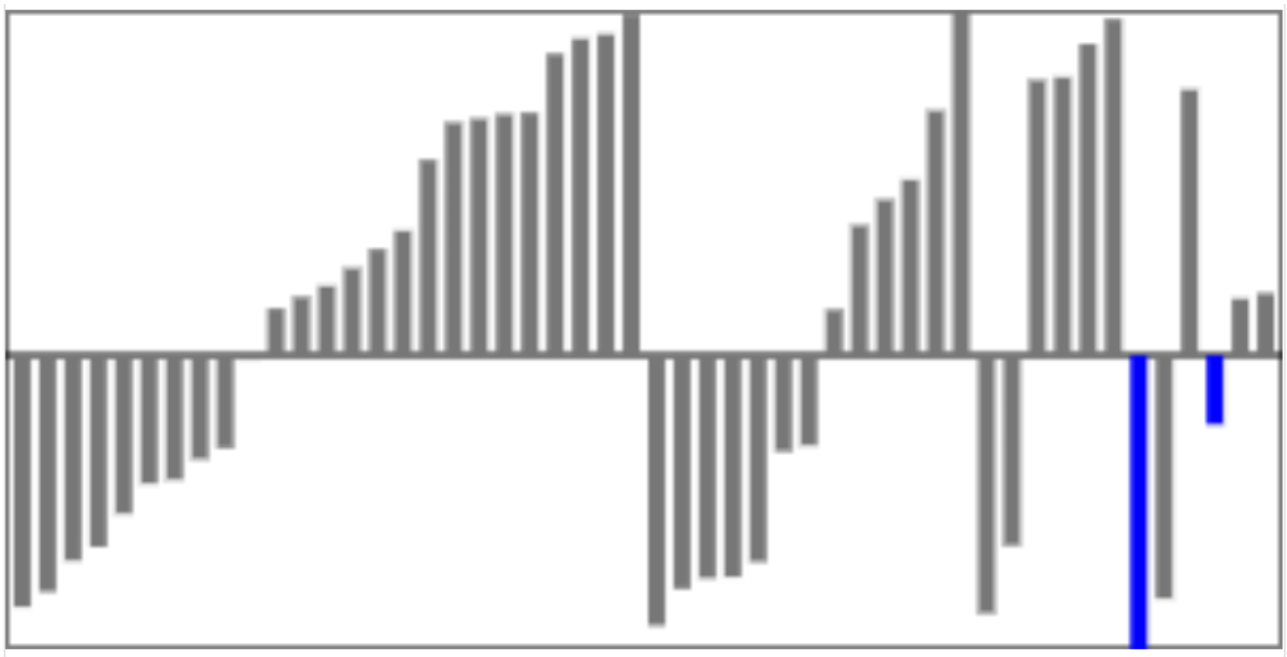


Рисунок 7.2 – Перевірка сортування злиттям(coderstool.com)

б) Метод швидкого сортування;

Результат виконання способу наведено на рисунку 7.3:

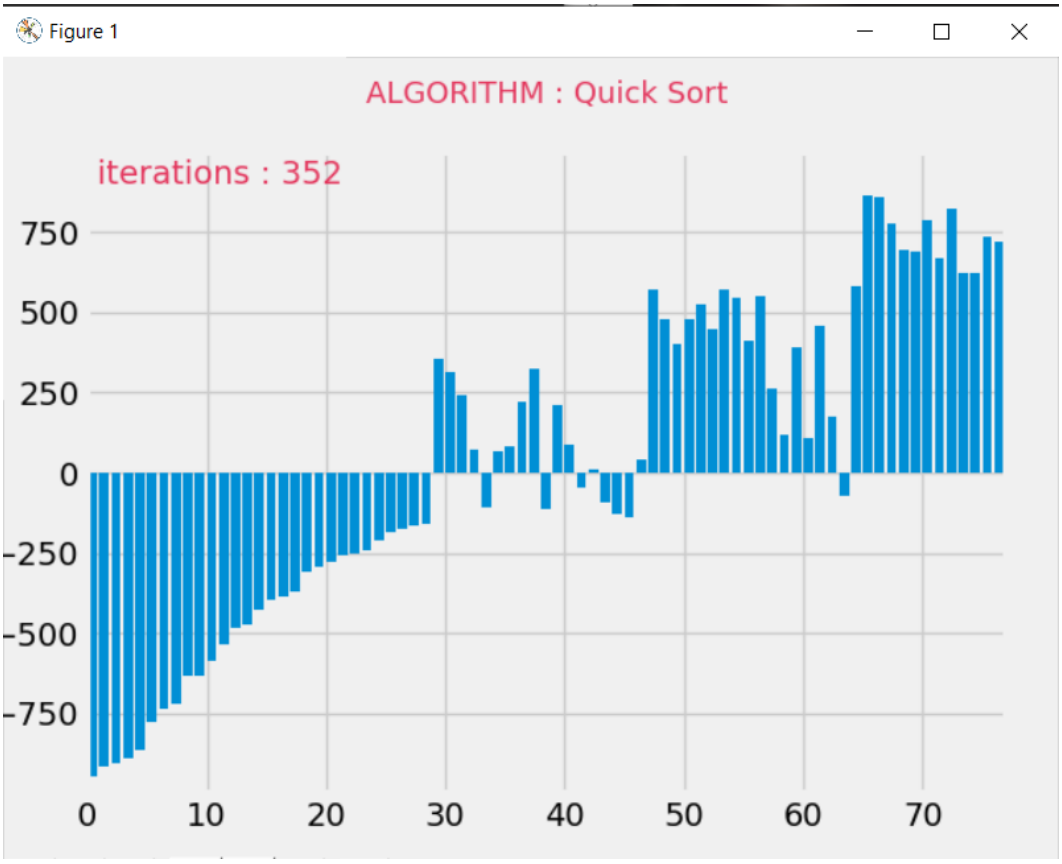


Рисунок 7.3 – Результат виконання швидкого сортування

Оскільки результати виконання збігається з результатом данного сайту(рисунок 7.4), то данне сортування працює вірно.

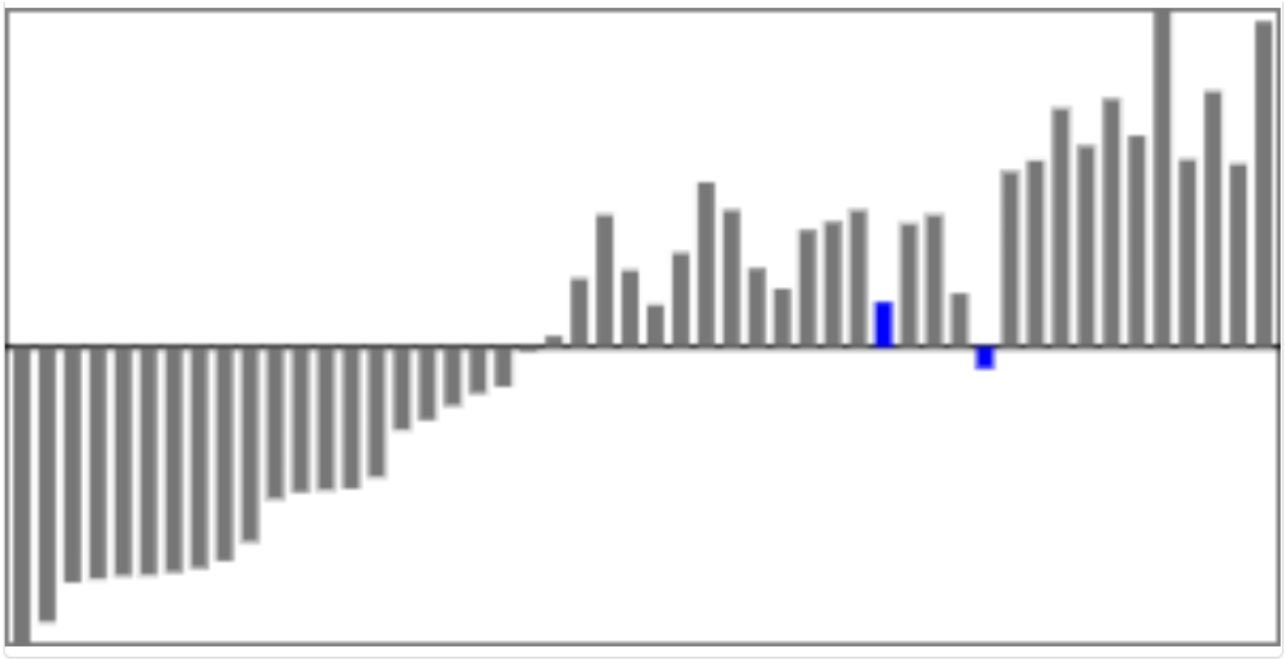


Рисунок 7.4 – Перевірка швидкого сортування (coderstool.com)

в) Метод інтроспективного сортування.

Результат виконання способу наведено на рисунку 7.5:

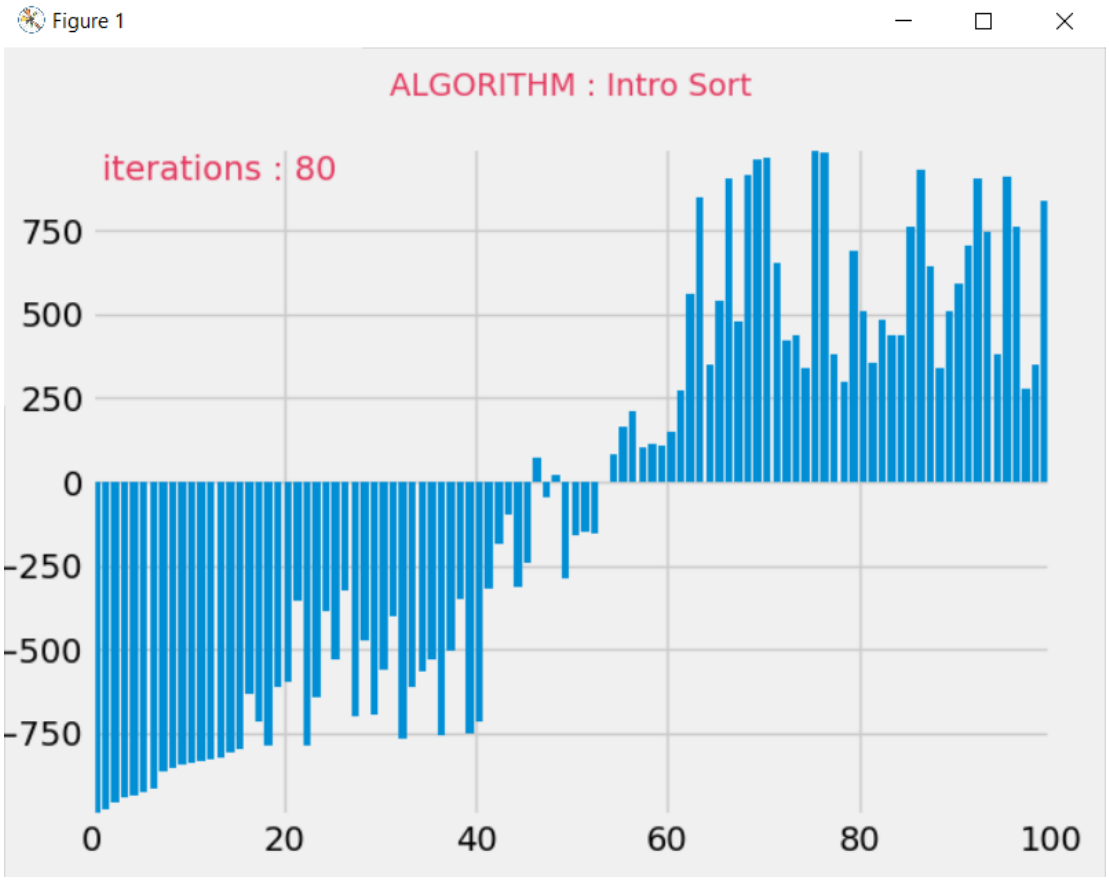


Рисунок 7.5 – Результат виконання інтроспективного сортування.

Оскільки результати виконання збігається з результатом данного сайту(рисунок 7.6), то данне сортування працює вірно.

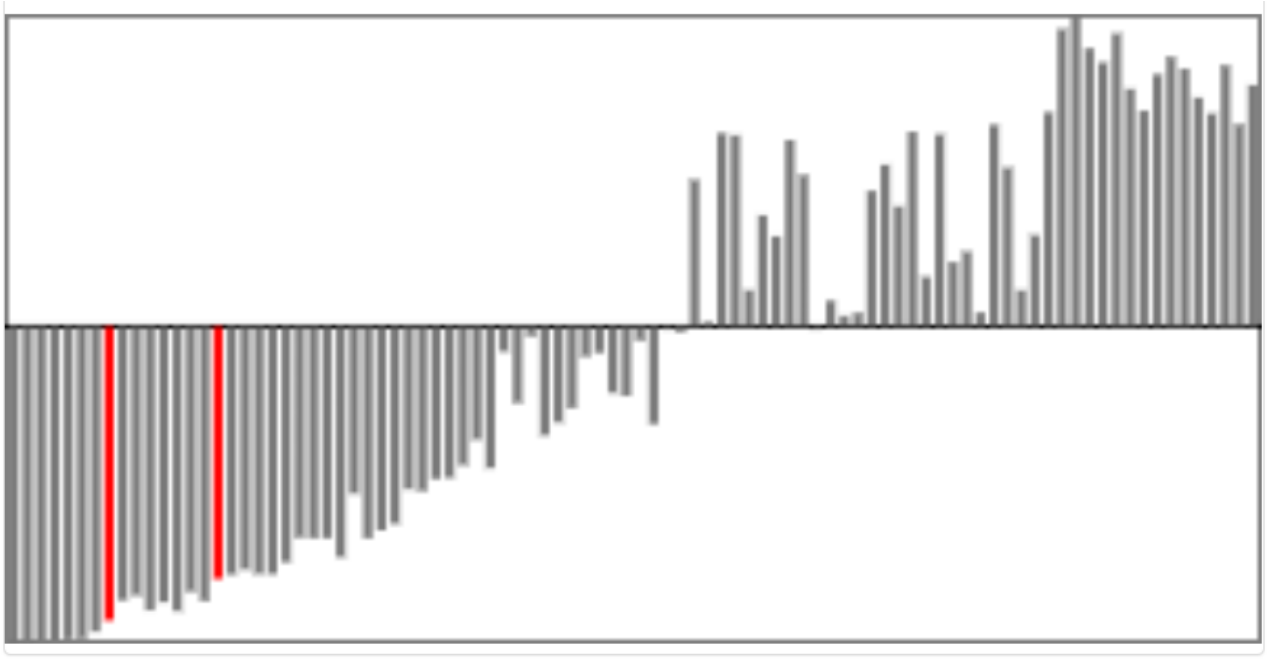


Рисунок 7.6 – Перевірка інтроспективного сортування(coderstool.com)

Результати тестування ефективності алгоритмів сортування наведено в таблиці 7.1:

Таблиця 7.1 – Тестування ефективності методів

Розмірність системи	Параметри тестування	Метод		
		Злиття	Швидке	Інтро
10	Кількість ітерацій	34	30	16
100	Кількість ітерацій	672	656	208
300	Кількість ітерацій	2488	2943	613
500	Кількість ітерацій	4488	5941	851
1000	Кількість ітерацій	10030	11783	1622

Алгоритм	Структура данних	Складність		
		Кращий	Середній	Гірший
Merge Sort	Масив	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Quick Sort	Масив	$O(nlogn)$	$O(nlogn)$	$O(n^2)$
Intro Sort	Масив	$O(n)$	$O(nlogn)$	$O(nlogn)$

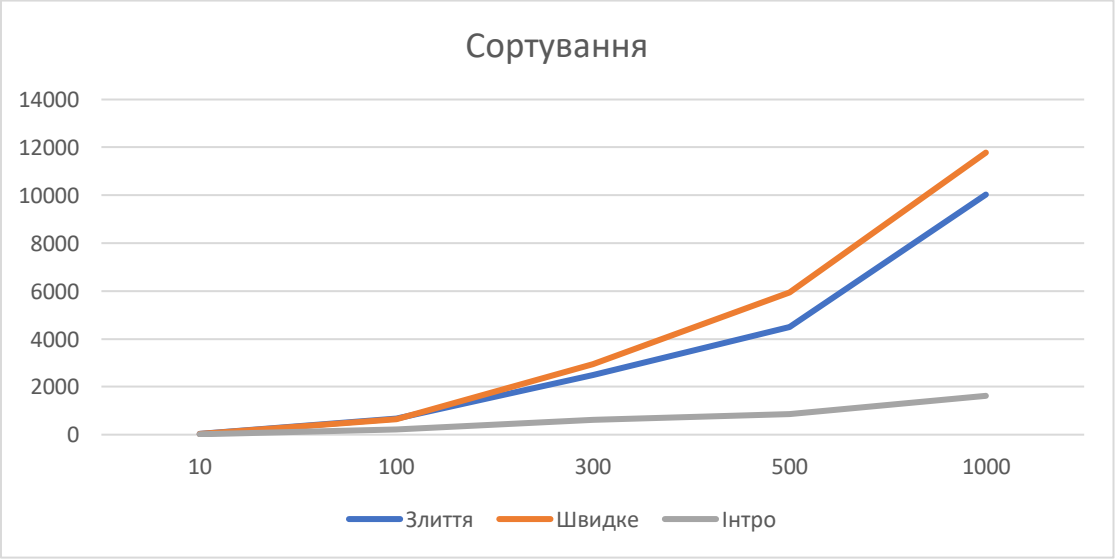


Рисунок 7.1 – Графік залежності кількості ітерацій методу від розміру вхідної системи

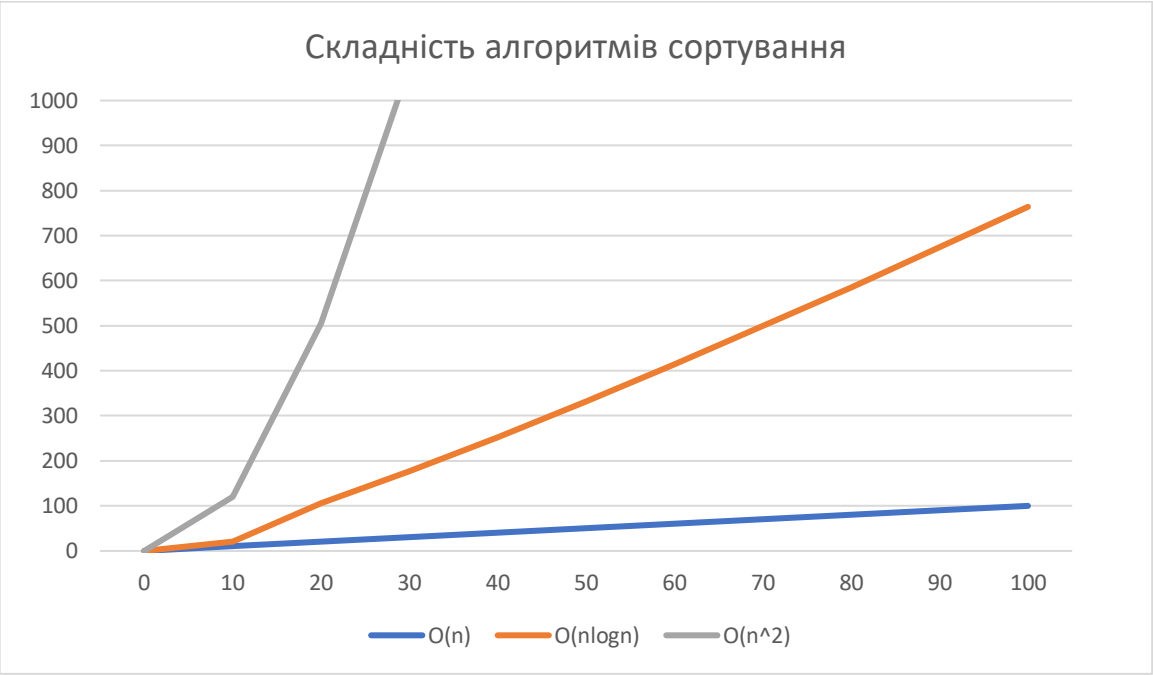


Рисунок 7.2 – Складність алгоритмів

За результатами тестування можна зробити такі висновки:

а) Всі розглянуті методи дозволяють посортувати масиви великих і надвеликих розмірів

б) Складність всіх розглянутих методів є різною(рисунок 5.2), а алгоритми показують різні результати в залежності від розмірності масиву

в) З розглянутих методів найоптимальнішим для практичного використання є метод Інтроспективного сортування, оскільки він виконується найшвидше за найменшу кількість ітерацій.

8. ПЕРЕЛІК ПОСИЛАНЬ

1. MergeSort URL:
   https://uk.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%B7%D0%BB%D0%B8%D1%82%D1%82%D1%8F%D0%BC
2. QuickSort URL:

   https://uk.wikipedia.org/wiki/%D0%A8%D0%B2%D0%B8%D0%B4%D0%BA%D0%B5_%D1%81%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F

3. IntroSort URL:
   https://ru.wikipedia.org/wiki/Introsort

ДОДАТОК А ТЕХНІЧНЕ ЗАВДАННЯ

КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. Сікорського

Кафедра
інформатики та програмної інженерії

Затвердив

Керівник Головченко М. М.

«___»_____201_ р.

Виконавець:

Студент Головня Олександр Ростиславович

«___» _____201_ р.

ТЕХНІЧНЕ ЗАВДАННЯ
на виконання курсової роботи
на тему: Упорядкування масивів
з дисципліни:
«Основи програмування»

Київ 2022

Мета: Метою курсової роботи є розробка програми, що генерує випадковим чином масив(розмір – не менше 100 елементів, максимально до 50000) та упорядковує його методами: метод сортування злиттям, метод швидкого сортування, метод інтроспективного сортування.

1. *Дата початку роботи*: «_____»_____202_ р.
2. *Дата закінчення роботи*: «___»_____ 202_ р.
3. *Вимоги до програмного забезпечення*.
   1) Функціональні вимоги:

      – Можливість задавати розмірність масиву для впорядкування(розмір – не менше 100 елементів, максимально до 50000).

      – Можливість генерувати масив випадкових чисел заданого розміру.

      – Можливість перевірки введених даних на коректність та виведення відповідних повідомлень у разі некоректних даних.

      – Можливість обрати метод сортування масиву.

      – Можливість сортування масиву обраним методом.

      – Можливість збереження результатів сортування масиву у текстовий файл.

      – Можливість відображення статичних та аналітичних даних для подальшого аналізу ефективності алгоритмів.

      Нефункціональні вимоги:

      – Можливість запуску програмного забезпечення на операційних системах Windows 10+

Все програмне забезпечення та супроводжуюча технічна документація повинні задовольняти наступним ДЕСТам:

ГОСТ 29.401 - 78 - Текст програми. Вимоги до змісту та оформлення.

ГОСТ 19.106 - 78 - Вимоги до програмної документації.

ГОСТ 7.1 - 84 та ДСТУ 3008 - 2015 - Розробка технічної документації.

4. *Стадії та етапи розробки*:

1) Об'єктно-орієнтований аналіз предметної області задачі (до__.__.202_ р.)
2) Об'єктно-орієнтоване проектування архітектури програмної системи (до __.__.202_р.)
3) Розробка програмного забезпечення (до __.__.202_р.)
4) Тестування розробленої програми (до __.__.202_р.)
5) Розробка пояснювальної записки (до __.__.202_ р.).
6) Захист курсової роботи (до __.__.202_ р.).

5. *Порядок контролю та приймання*. Поточні результати роботи над КР регулярно демонструються викладачу. Своєчасність виконання основних етапів графіку підготовки роботи впливає на оцінку за КР відповідно до критеріїв оцінювання.

ДОДАТОК Б ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду програмного забезпечення вирішення задачі упорядкування масивів

(Найменування програми (документа))

Github

(Вид носія даних)

182 арк, 779 Кб

(Обсяг програми (документа), арк., Кб)

студента групи ІП-11 І курсу
Головні О. Р.

FILL_ARR_IN_FILE.PY

```
from func_for_arr import *
import random
from random import randint

class fill_arr_in_file():
    def generate_arr(size):
        a,b = -999,999
        array = [randint(a,b) for _ in range(int(size))]
        return array
    def f_arr(size):
        global array1
        global array2
        file1 = open('file1.txt','w')
        array1 = fill_arr_in_file.generate_arr(size)
        array2=array1
        for i in array1:
            file1.write(str(i)+' ')
        file1.close()


    def f2_arr(comb_way):
        file2 = open('file2.txt','w')
        if comb_way == 1:
            array2 = func.merge_sort1(array1)
        elif comb_way == 2:
            array2 = func.quick_sort1(array1)
        elif comb_way == 3:
            for i in range(0,15):
                array2 = func.introSort1(array1,i,0,len(array1))
        for i in array2:
            file2.write(str(i)+' ')
        file2.close()
```

FUNC_FOR_ARR.PY

```
from func_for_func import *
import operator
import math

class func():
    def merge_sort1(L, compare=operator.lt):
        if len(L) < 2:
            return L[:]
        else:
            middle = int(len(L) / 2)
            left = func.merge_sort1(L[:middle], compare)
            right = func.merge_sort1(L[middle:], compare)
            return pod_func.merge1(left, right, compare)
    def quick_sort1(nums):
        def _quick_sort1(items, low, high):
            if low < high:
                split_index = pod_func.partition_1(items, low, high)
                _quick_sort1(items, low, split_index)
                _quick_sort1(items, split_index + 1, high)

        _quick_sort1(nums, 0, len(nums) - 1)
        return nums
    def introSort1(a, d, start, end):
        n = end - start
        if n <= 1:
            return
        elif d == 0:
            pod_func.introHS1(a, start, end)
        else:
            p = pod_func.partition1(a, start, end)
            func.introSort1(a, d-1, start, p)
            func.introSort1(a, d-1, p+1, end)
        return a
```

FUNC_FOR_FUNC.PY

```
import math
class pod_func():
    def merge1(left, right, compare):
        result = []
        i, j = 0, 0
        while i < len(left) and j < len(right):
            if compare(left[i], right[j]):
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        while i < len(left):
            result.append(left[i])
            i += 1
        while j < len(right):
            result.append(right[j])
            j += 1
        return result
    def partition_1(nums, low, high):
        pivot = nums[(low + high) // 2]
        i = low - 1
        j = high + 1
        while True:
            i += 1
            while nums[i] < pivot:
                i += 1

            j -= 1
            while nums[j] > pivot:
                j -= 1

            if i >= j:
                return j

            nums[i], nums[j] = nums[j], nums[i]
    def introHS1 (a, start, end):
        b = a[start:end]
        pod_func.heapSort1(b)
        for i in range(0,len(b)):
            a[start+i] = b[i]

    def heapSort1 (a):
        END = len(a)
```

```
    for k in range (math.floor(END/2) - 1, -1, -1):
        pod_func.heapify1(a, END, k)

    for k in range(END, 1, -1):
        pod_func.swap1(a, 0, k-1)
        pod_func.heapify1(a, k-1, 0)

def partition1(a, start, end):
    x = a[end-1]
    i = start-1
    for j in range(start, end-1):
        if a[j] <= x:
            i=i+1
            pod_func.swap1(a, i, j)
    pod_func.swap1(a, i+1, end-1)
    return i+1

def swap1(a, i, j):
    tmp = a[i]
    a[i] = a[j]
    a[j] = tmp

def heapify1(a,iEnd,iRoot):
    iL = 2*iRoot + 1
    iR = 2*iRoot + 2
    if iR < iEnd:
        if (a[iRoot] >= a[iL] and a[iRoot] >= a[iR]):
            return
        else:
            if(a[iL] > a[iR]):
                j = iL
            else:
                j = iR
            pod_func.swap1(a, iRoot, j)
            pod_func.heapify1(a, iEnd, j)

    elif iL < iEnd:
        if (a[iRoot] >= a[iL]):
            return
        else:
            pod_func.swap1(a, iRoot, iL)
            pod_func.heapify1(a,iEnd,iL)

    else:
        return
```

MAIN.PY

```
from fill_arr_in_file import *
from Graphics import *
import math
from tkinter import messagebox
from tkinter import *
import tkinter as tk
from tkinter import ttk

class clicks():
    def btn_click1(entry_n):
        global size
        size = entry_n.get()
        if not size.isdigit():
            messagebox.showinfo("Помилка", "Потрібно ввести число(Від 100 до
50000)")
        elif int(size)<100 or int(size)>50000:
            messagebox.showinfo("Помилка", "Розмірність масиву повинна бути від
100 до 50000")
        else:
            messagebox.showinfo("Повідомлення", "Масив з розмірністю "+str(size)+"
згенеровано у файл1")
    def btn_click12(text1):
        txt = open('file1.txt', encoding='utf-8').readlines()
        text1.delete(1.0,END)
        text1.insert(1.0,txt)
        text1.pack()
    def btn_click2(text1):
        txt2 = open('file2.txt', encoding='utf-8').readlines()
        text1.delete(1.0,END)
        text1.insert(1.0,txt2)
        text1.pack()
    def btn_click3(entry_n,combobox):
        global size
        global comb_way
        size = entry_n.get()
        str_comb_way = combobox.get()
        if str_comb_way == 'Сортування злиттям':
            comb_way=1
        elif str_comb_way == 'Швидке сортування':
            comb_way=2
        elif str_comb_way =='Інтроспективне сортування':
              comb_way=3
```

```
def main():
    root = Tk()
    root.title("Сортування масиву")
    root.geometry("700x500")
    root.resizable(False,False)
    root["bg"] = "gray22"

    label_1 =
tk.Label(root,bd=0,relief="groove",compound=tk.CENTER,bg="Gray22",fg="White
",activeforeground="black",activebackground="white",font="arial 10",text='Введіть
розмірність масиву:')
    label_1.pack(side=TOP, anchor=NW,pady=2)

    entry_n = ttk.Entry(root,width=23)
    entry_n.pack(side=TOP, anchor=NW,padx=27,pady=2)

    label_2 =
tk.Label(root,bd=0,relief="groove",compound=tk.CENTER,bg="Gray22",fg="White
",activeforeground="black",activebackground="white",font="arial
10",text='Виберіть спосіб сортування:')
    label_2.pack(side=TOP, anchor=NW,pady=2)

    combobox = ttk.Combobox(root,values=[u'Сортування злиттям',u'Швидке
сортування',u'Інтроспективне сортування'])
    combobox.current(0)
    combobox.pack(side=TOP, anchor=NW,padx =27,pady=2)

    btn_2 =
tk.Button(root,bd=0,relief="groove",compound=tk.CENTER,bg="Gray",fg="White",
activeforeground="black",activebackground="white",font="arial
10",text='Підтвердити та Згенерувати ',

command=lambda:[clicks.btn_click1(entry_n),fill_arr_in_file.f_arr(size),clicks.btn_c
lick12(text1)])
    btn_2.pack(side=LEFT, anchor=NW,padx=8,pady=8)

    btn_3 =
tk.Button(root,bd=0,relief="groove",compound=tk.CENTER,bg="Gray",fg="White",
activeforeground="black",activebackground="white",font="arial
10",text='Посортувати з візуалізацією',

command=lambda:[clicks.btn_click3(entry_n,combobox),Graphics.Shoow(comb_wa
y),clicks.btn_click2(text1)])
    btn_3.pack(side=TOP, anchor=CENTER,padx =40,pady=4)
```

```
    label_22 = tk.Label(root,bg="Gray22",text='Або')
    label_22.pack(side=TOP, anchor=CENTER,padx =0,pady=0)

    btn_4 =
tk.Button(root,bd=0,relief="groove",compound=tk.CENTER,bg="Gray",fg="White",
activeforeground="black",activebackground="white",font="arial
10",text='Посортувати швидко',

command=lambda:[clicks.btn_click3(entry_n,combobox),fill_arr_in_file.f2_arr(com
b_way),clicks.btn_click2(text1)])
    btn_4.pack(side=TOP, anchor=CENTER,padx =40,pady=4)

    text1=tk.Text(root,borderwidth = 5)
    text1.pack()
    root.mainloop()

main()
```

GRAPHICS.PY

```
from func_for_graphics import *
from fill_arr_in_file import *

import math
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from mpl_toolkits.mplot3d import axes3d
import matplotlib as mp
import numpy as np

class Graphics():
    def Shoow(comb_way):
        array2 = []
        with open("file1.txt") as f:
            for line in f:
                array2=([int(x) for x in line.split()])
        f.close()
        if comb_way==1:
            Graphics.showGraph(array2)
        if comb_way==2:
            Graphics.showGraph2(array2)
        if comb_way==3:
            Graphics.showGraph3(array2)
        fill_arr_in_file.f2_arr(comb_way)
    def showGraph(array2):
        generator = func_for_gr.mergesort(array2, 0, len(array2)-1)
        algoName='Merge Sort'

        plt.style.use('fivethirtyeight')

        data_normalizer = mp.colors.Normalize()

        fig, ax = plt.subplots()

        bar_rects = ax.bar(range(len(array2)), array2, align="edge")

        ax.set_xlim(0, len(array2))
        ax.set_ylim(-999, 999)
        ax.set_title("ALGORITHM : "+algoName+"\n",
                    fontdict={'fontsize': 13, 'fontweight': 'medium',
                              'color' : '#E4365D'})

        text = ax.text(0.01, 0.95, "", transform=ax.transAxes,
                       color="#E4365D")
```

```
        iteration = [0]

        def animate(A, rects, iteration):
            for rect, val in zip(rects, A):
                rect.set_height(val)
            iteration[0] += 1
            text.set_text("iterations : {}".format(iteration[0]))

        anim = FuncAnimation(fig, func=animate,
            fargs=(bar_rects, iteration), frames=generator, interval=0,
            repeat=False)
        plt.show()
    def showGraph2(array2):

        generator = func_for_gr.quicksort(array2, 0, len(array2)-1)
        algoName = 'Quick Sort'

        plt.style.use('fivethirtyeight')

        data_normalizer = mp.colors.Normalize()

        fig, ax = plt.subplots()

        bar_rects = ax.bar(range(len(array2)), array2, align ="edge")

        ax.set_xlim(0, len(array2))
        ax.set_ylim(-999, 999)
        ax.set_title("ALGORITHM : "+ algoName + "\n", fontdict = {'fontsize': 13,
'fontweight':
                                    'medium', 'color' : '#E4365D'})

        text = ax.text(0.01, 0.95, "", transform = ax.transAxes, color = "#E4365D")
        iteration = [0]

        def animate(A, rects, iteration):
            for rect, val in zip(rects, A):

                rect.set_height(val)
            iteration[0] += 1
            text.set_text("iterations : {}".format(iteration[0]))

        anim = FuncAnimation(fig, func = animate,
            fargs = (bar_rects, iteration), frames = generator, interval = 0,
            repeat = False)
        plt.show()
```

```
    def showGraph3(array2):
        for d in range(0,15):
            generator = func_for_gr.introSort(array2,d,0,len(array2))
        algoName = 'Intro Sort'

        plt.style.use('fivethirtyeight')

        data_normalizer = mp.colors.Normalize()

        fig, ax = plt.subplots()

        bar_rects = ax.bar(range(len(array2)), array2, align ="edge")

        ax.set_xlim(0, len(array2))
        ax.set_ylim(-999, 999)
        ax.set_title("ALGORITHM : "+ algoName + "\n", fontdict = {'fontsize': 13,
'fontweight':
                                    'medium', 'color' : '#E4365D'})

        text = ax.text(0.01, 0.95, "", transform = ax.transAxes, color = "#E4365D")
        iteration = [0]

        def animate(A, rects, iteration):
            for rect, val in zip(rects, A):

                rect.set_height(val)
            iteration[0] += 1
            text.set_text("iterations : {}".format(iteration[0]))

        anim = FuncAnimation(fig, func = animate,
            fargs = (bar_rects, iteration), frames = generator, interval = 0,
            repeat = False)
        plt.show()
```

FUNC_FOR_GRAPHICS.PY

```
import math
class func_for_gr():
    def mergesort(A, start, end):
        if end <= start:
            return
        mid = start + ((end - start + 1) // 2) - 1
        yield from func_for_gr.mergesort(A, start, mid)
        yield from func_for_gr.mergesort(A, mid + 1, end)
        yield from func_for_gr.merge(A, start, mid, end)

    def merge(A, start, mid, end):
        merged = []
        leftIdx = start
        rightIdx = mid + 1

        while leftIdx <= mid and rightIdx <= end:
            if A[leftIdx] < A[rightIdx]:
                merged.append(A[leftIdx])
                leftIdx += 1
            else:
                merged.append(A[rightIdx])
                rightIdx += 1

        while leftIdx <= mid:
            merged.append(A[leftIdx])
            leftIdx += 1
        while rightIdx <= end:
            merged.append(A[rightIdx])
            rightIdx += 1

        for i in range(len(merged)):
            A[start + i] = merged[i]
            yield A

    def quicksort(a, l, r):
        if l >= r:
            return
        x = a[l]
        j = l
        for i in range(l + 1, r + 1):
            if a[i] <= x:
                j += 1
                a[j], a[i] = a[i], a[j]
            yield a
```

```
        a[l], a[j]= a[j], a[l]
        yield a

        yield from func_for_gr.quicksort(a, l, j-1)
        yield from func_for_gr.quicksort(a, j + 1, r)

def introSort(A,d, start, end):
        n = end - start
        if n <= 1:
                yield A
                return
        elif d == 0:
                yield from func_for_gr.introHS(A, start, end)
                yield A
        else:
                p = func_for_gr.partition(A, start, end)
                yield A
                yield from func_for_gr.introSort(A, d-1, start, p)
                yield from func_for_gr.introSort(A, d-1, p+1, end)
        yield A

def introHS (A, start, end):
        b = A[start:end]
        func_for_gr.heapSort(b)
        for i in range(0,len(b)):
                A[start+i] = b[i]
        yield A
def heapSort (A):
        END = len(A)
        for k in range (math.floor(END/2) - 1, -1, -1):
                func_for_gr.heapify(A, END, k)

        for k in range(END, 1, -1):
                func_for_gr.swap(A, 0, k-1)
                func_for_gr.heapify(A, k-1, 0)


def partition(A, start, end):
        x = A[end-1]
        i = start-1
        for j in range(start, end-1):
                if A[j] <= x:
                        i=i+1
                        func_for_gr.swap(A, i, j)
        func_for_gr.swap(A, i+1, end-1)
```

```
        return i+1

def swap(A, i, j):
        tmp = A[i]
        A[i] = A[j]
        A[j] = tmp

def heapify(A,iEnd,iRoot):
        iL = 2*iRoot + 1
        iR = 2*iRoot + 2
        if iR < iEnd:
                if (A[iRoot] >= A[iL] and A[iRoot] >= A[iR]):
                        return

                else:
                        if(A[iL] > A[iR]):
                                j = iL
                        else:
                                j = iR
                        func_for_gr.swap(A, iRoot, j)
                        func_for_gr.heapify(A, iEnd, j)
        elif iL < iEnd:
                if (A[iRoot] >= A[iL]):
                        return
                else:
                        func_for_gr.swap(A, iRoot, iL)
                        func_for_gr.heapify(A,iEnd,iL)
        else:
                return
```