Міністерство освіти і науки України  
Департамент освіти і науки Запорізької облдержадміністрації  
Запорізьке територіальне відділення Малої академії наук України

Відділення: комп’ютерні науки  
Секція: програмна інженерія

ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ АЛГОРИТМІВ СОРТУВАННЯ

Роботу виконала:  
Сольона Іларія Романівна, учениця 10-В класу Комунального закладу "Запорізька спеціалізована школа-інтернат II-III ступенів " Січовий колегіум" Запорізької міської ради Запорізької області

Наукові керівники:  
Москальов Павло Олександрович, вихователь КЗ «Запорізька спеціалізована школа-інтернат II-III ступенів «Січовий колегіум» Запорізької обласної ради

Циммерман Геннадій Анатолійович, старший викладач ЗНУ, керівник гуртка КЗ ЗОЦНТТУМ «Грані»

Запоріжжя – 2024

АНОТАЦІЯ

Запорізьке територіальне відділення Малої академії наук України

Сольона Іларія Романівна, учениця 10-В класу Комунального закладу "Запорізька спеціалізована школа-інтернат II-III ступенів " Січовий колегіум" Запорізької міської ради, місто Запоріжжя

Наукові керівники: Москальов Павло Олександрович, вихователь комунального закладу «Запорізька спеціалізована школа-інтернат II-III ступенів «Січовий колегіум» Запорізької обласної ради, Циммерман Геннадій Анатолійович, старший викладач ЗНУ, керівник гуртка КЗ ЗОЦНТТУМ «Грані»

ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ АЛГОРИТМІВ СОРТУВАННЯ

Мета: У комп’ютерному експерименті дослідити особливості алгоритмів сортування, виявити їх ефективність у залежності від вхідних даних.

Актуальність: Сучасні інформаційні системи поступово переорієнтовуються на використання алгоритмів та програм для швидкого опрацювання великих обсягів даних. Тому на поточний момент необхідно проаналізувати відповідний проблемі існуючий алгоритмічний інструментарій та виявити найбільш вдалі рішення залежно від особливостей вхідних даних.

Завдання:

* Ознайомитись з сучасним станом проблеми опрацювання великих обсягів даних,
* Визначити сутність задачі сортування,
* Ознайомитися із переліком та сутністю найбільш відомих алгоритмів сортування,
* Реалізувати обрані алгоритми сортування мовою C#,
* Створити програмну реалізацію генерації випадкових чисел мовою програмування Python,
* Написати програмну реалізацію для обчислення часу роботи кожного з обраних алгоритмів сортування.

Робота складалася з трьох етапів. Перший етап полягав в опрацюванні відповідних інтернет-джерел, стосовно обраних алгоритмів сортування. Другий кроком була програмна реалізація. Був створений скрипт для генерації масивів мовою Python (див. Додаток Т). Створена головна програма (див. Додаток У), для виміру часу роботи певного алгоритму сортування, яка була реалізована мовою C#. На основі отриманих даних було створено графік засобами табличного процесору (Excel), де показано залежність часу роботи кожного алгоритму від кількості даних у масиві.

Ключові слова: алгоритми сортування, програмна реалізація, час, генерація чисел, вхідні дані.

ЗМІСТ

[ВСТУП 6](#_Toc156677830)

[РОДІЛ 1 ТЕОРЕТИЧНА ЧАСТИНА 8](#_Toc156677831)

[1.1 Сортування бульбашкою (Bubble sort) 8](#_Toc156677832)

[1.2 Сортування вибором (Selection Sort) 8](#_Toc156677833)

[1.3 Сортування вставками (Insertion Sort) 9](#_Toc156677834)

[1.4 Сортування змішуванням (Cocktail sort) 9](#_Toc156677835)

[1.5 Сортування двійковим деревом (Tree Sort) 9](#_Toc156677836)

[1.6 Сортування гребінцем (Comb sort) 9](#_Toc156677837)

[1.7 Сортування злиттям (Merge sort) 10](#_Toc156677838)

[1.8 Пірамідальне сортування (Heapsort) 10](#_Toc156677839)

[1.9 Timsort 11](#_Toc156677840)

[1.10 Швидке сортування (Quick Sort) 11](#_Toc156677841)

[1.11 Сортування підрахунком (Counting Sort) 12](#_Toc156677842)

[1.12 Сортування за розрядами (Radix Sort) 12](#_Toc156677843)

[1.13 Сортування комірками (Bucket sort) 13](#_Toc156677844)

[1.14 Сортування Шелла (Shell Sort) 13](#_Toc156677845)

[Висновок до розділу 1 13](#_Toc156677846)

[РОЗДІЛ 2 ПРОГРАМНА РЕАЛІЗАЦІЯ 14](#_Toc156677847)

[Висновок до розділу 2 18](#_Toc156677848)

[ВИСНОВКИ 19](#_Toc156677849)

[ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ ТА ДЖЕРЕЛ 21](#_Toc156677850)

[ДОДАТКИ 24](#_Toc156677851)

[Додаток А 24](#_Toc156677852)

[Додаток Б 25](#_Toc156677853)

[Додаток В 26](#_Toc156677854)

[Додаток Г 27](#_Toc156677855)

[Додаток Д 29](#_Toc156677856)

[Додаток Е 32](#_Toc156677857)

[Додаток Ж 34](#_Toc156677858)

[Додаток К 37](#_Toc156677859)

[Додаток Л 39](#_Toc156677860)

[Додаток М 43](#_Toc156677861)

[Додаток Н 45](#_Toc156677862)

[Додаток П 47](#_Toc156677863)

[Додаток Р 50](#_Toc156677864)

[Додаток С 52](#_Toc156677865)

[Додаток Т 54](#_Toc156677866)

[Додаток У 55](#_Toc156677867)

## ВСТУП

На сьогоднішній момент у світі, який вже має сталу назву інформаційне суспільство, немає більш важливої проблеми, ніж пошук нових і вдосконалення вже існуючих способів і засобів обробки інформації.

На думку науковців, земна цивілізація проходить етап, коли за добу загальні обсяги інформації подвоюються. Як результат, було засновано новий напрямок ІТ - Великі дані (Big Data). Таким чином сортування даних, як різновид проблем з напрямку Big Data, є цікавим і актуальним напрямком дослідження. А ми хочемо розібратися з особливостями вирішення таких проблем і таким чином допомогти людству економити такий дорогоцінний час, який зазвичай не раціонально витрачається на повторний аналіз задачі та способів її вирішення, а потім вже – на вибір найбільш вдалих способів із розглянутих.

Актуальність: Сучасні інформаційні системи поступово переорієнтовуються на використання алгоритмів та програм для швидкого опрацювання великих обсягів даних. Тому на поточний момент необхідно проаналізувати відповідний проблемі існуючий алгоритмічний інструментарій та виявити найбільш вдалі рішення залежно від особливостей вхідних даних.

Мета: У комп’ютерному експерименті дослідити особливості алгоритмів сортування, виявити їх ефективність у залежності від вхідних даних.

Завдання:

* Ознайомитись з сучасним станом проблеми опрацювання великих обсягів даних,
* Визначити сутність задачі сортування,
* Ознайомитися із переліком найбільш відомих алгоритмів сортування,
* Реалізувати обрані алгоритми сортування мовою C#,
* Створити програмну реалізацію генерації випадкових чисел мовою програмування Python,
* Написати програмну реалізацію, для обчислення часу роботи кожного алгоритму сортування.

Об’єкт дослідження: програмні засоби обробки великих масивів даних.

Предмет дослідження: ефективність програмних реалізацій алгоритмів сортування.

Методи дослідження: опрацювання інтернет-джерел, спостереження, аналіз, синтез.

## РОДІЛ 1 ТЕОРЕТИЧНА ЧАСТИНА

Отже зупинимось спочатку на інформації про популярні алгоритми сортування з «першоджерел» - класичних публікаціях теоретиків та практиків програмування. Сортування у програмуванні – це перерозміщення (переставляння) елементів якоїсь початкової послідовності об’єктів відповідно до обраного відношення порядку між елементами (наприклад, для бінарних відношень порядку < та > кажуть відповідно про сортування за зростанням та спаданням значень елементів). А алгоритм сортування – процес, що здійснює впорядкування лінійного списку елементів. Такий список зазвичай в програмуванні називають масивом.

### Сортування бульбашкою (Bubble sort)

Бульбашкове сортування є простим алгоритмом. У публікаціях зазначається, що він не є оптимальним. Бульбашкове сортування - це сортуванням на основі порівнянь. Таку назву алгоритм отримав через те, що нагадує поведінку бульбашок повітря різного розміру у резервуарі з водою, на які впливає Архімедова сила.

У заданому наборі даних, порівнюються пари сусідніх елементів, при умові якщо у цій парі один з елементів є більшим або меншим за другий, то вони будуть обмінюватися місцями. Прохід по даним буде виконуватися доти, доки не буде відсортованим масив. [5] (див. Додаток А)

### Сортування вибором (Selection Sort)

Сортування вибором – простий алгоритм сортування, на основі вставок. Вважається неефективним при сортуванні великих масивів.

Алгоритм сортування вибором полягає в тому, що відбувається пошук найменшого елементу масиву, далі міняються місцями найменший елемент з першим, другим і т.д. Це повторюється доти, доки список не завершиться. [6] (див. Додаток Б)

### Сортування вставками (Insertion Sort)

Сортування вставками – простий алгоритм сортування на основі порівнянь. За матеріалами публікацій цей алгоритм більш ефективний на масивах маленького розміру ніж великого.

Алгоритм сортування вставками працює наступним чином: на початку роботи впорядкована частина масиву порожня, потім почергово кожний елемент невпорядкованої частини масиву, починаючи з першого, додається у впорядковану частину масиву, при цьому стаючи відразу на правильну позицію. Це повторюється доти, доки всі елементи масиву не опиняться у відсортованій частині. [7-8] (див. Додаток В)

### Сортування змішуванням (Cocktail sort)

Сортування змішуванням – це один із різновидів сортування бульбашкою. За теорією він є більш ефективним, ніж сортування бульбашкою.

Алгоритм сортування змішуванням схожий на Бульбашкове сортування. Відрізняється від сортування бульбашкою, тим що робить прохід одночасно в обидва напрямки – і зліва направо, і навпаки. [13] (див. Додаток Г)

### Сортування двійковим деревом (Tree Sort)

Сортування двійковим деревом – алгоритм сортування, що полягає в побудові двійкового дерева пошуку за ключами масиву.

Спочатку з вхідного масиву даних будується двійкове дерево пошуку (структура даних, де для кожного вузла дерева значення всіх вузлів у лівому піддереві менше за значення поточного вузла, а значення всіх вузлів у правому піддереві більше за значення поточного вузла). Далі виконується обхід двійкового дерева, щоб отримати масив впорядкованих елементів. [10] (див. Додаток Д)

### Сортування гребінцем (Comb sort)

Сортування гребінцем – поліпшений алгоритм сортування бульбашкою. Алгоритм сортування гребінцем розроблений Влодеком Добошєвічем у 1980 році, пізніше досліджений та популяризований Стефаном Лакеєм та Річардом Боксом у 1991 році.

Його ідея полягає в тому, щоб перемістити малі значення ближче до кінця списку (так як у сортуванні бульбашкою вони сильно уповільнюють процес сортування). Розрив (відстань між порівнювальними елементами) може бути більший за одиницю. Розрив починається зі значення, що дорівнює довжині списку, поділеного на фактор зменшення (зазвичай 1,3), а список сортується з урахуванням цього значення. Потім розрив знову ділиться на фактор розриву й масив продовжує сортуватися з новим значенням. Цей процес продовжується доти, доки розрив рівний 1. Далі масив сортується з розривом 1, поки масив не буде відсортованим. [9] (див. Додаток Е)

### Сортування злиттям (Merge sort)

Сортування злиттям – алгоритм сортування масиву, який реалізовано за принципом «розділяй і володарюй».

Робота алгоритму сортування злиттям починається з поділу вхідного масиву на дві частини однакового розміру. Далі рекурсивно (це коли функція викликає сама себе) поділяються ці дві частини на ще дві й так доти, доки у кожній підчастині не буде по одному елементу. Під час сортування попарно порівнюються частини та ставляться на відповідне місце доти, доки у відсортовану частину масиві не потраплять всі елементи. Далі виконується крок сортування та злиття двох відсортованих частин в один відсортований масив. [12] (див. Додаток Ж)

### Пірамідальне сортування (Heapsort)

Пірамідальне сортування – алгоритм сортування, який використовує бінарне сортувальне дерево (структура даних, де для кожного вузла дерева значення всіх вузлів у лівому піддереві менше за значення поточного вузла, а значення всіх вузлів у правому піддереві більше за значення поточного вузла).

Алгоритм роботи пірамідального сортування ґрунтується на побудові бінарного сортувального дерева. Далі найбільший елемент переміщається до кореня дерева (елемент що є напочатку коренем дереву міняється місцями з найбільшим з дочірніх елементів, допоки не стане більшим за дочірні елементи). Таким чином у масиві перший елемент стає найбільшим елементом і його міняють місцями з останнім. Далі знову будують бінарне дерево й повторюють вище описаний алгоритм знаходження найбільшого елемента, не враховуючи вже знайдені та відсортовані. Цей алгоритм буде виконуватися доти, доки не залишиться один елемент. Тоді масив буде відсортовано. [3] (див. Додаток К)

### Timsort

Timsort - гібридний стабільний алгоритм сортування. Він поєднує в собі сортування злиттям та сортування включенням, призначений для ефективної роботи з багатьма видами реальних даних. Розроблений Тімом Пітерсом (2002 р.)

Timsort був розроблений, щоб скористатися перевагами наборів упорядкованих елементів, які вже існують в більшості реальних даних. Він перебирає елементи, збираючи впорядковані частини і одночасно поміщає ці частини в стек (абстрактна структура даних, що визначається принципом "останнім прийшов - першим вийшов"). Щоразу, коли частини зверху стеку відповідають критеріям об'єднання (злиття), вони об'єднуються. Це триває поки всі дані не будуть пройдені; потім усі впорядковані частини об'єднуються по дві одночасно та сортуються алгоритмом сортування вставками, таким чином утворюється відсортований масив. [1] (див. Додаток Л)

### Швидке сортування (Quick Sort)

Швидке сортування – алгоритм сортування, який потребує додаткової пам’яті. Так як алгоритм використовує прості цикли і операції, він працює швидше за інші алгоритми. Швидке сортування є алгоритмом на основі порівнянь і не є стабільним. Алгоритм розроблено Тоні Гоаром у 1962 році.

Ідея алгоритму полягає в поділі масиву на дві частини ліву та праву, за принципом: щоб в лівій частині всі елементи мають бути менші за опорний елемент, а елементи правої частини більші за опорний елемент. Далі вибирається опорний елемент з лівої частини, й аналогічно ділить ліву частину на дві частини, аналогічне відбувається й з правою частиною. Цей алгоритм працює доти, доки масив не буде відсортованим. [17] (див. Додаток М)

### Сортування підрахунком (Counting Sort)

Сортування підрахунком – алгоритм сортування, що застосовується на масивах з малою кількістю елементів. Час його роботи лінійно залежить як від загальної кількості елементів у масиві так і від кількості різних елементів.

Ідея алгоритму полягає в наступному: спочатку підрахувати скільки разів кожен унікальний елемент зустрічається у вихідному масиві. Спираючись на ці дані можна одразу вирахувати на якому місці має стояти кожен елемент, а потім за один прохід поставити всі елементи на свої місця. Робота алгоритма починається з пошуку максимального елемента у масиві, щоб створити новий пустий масив, в який буде записуватися кількість кожного елемента. Таким чином стає відомо скільки разів кожний елемент зустрічається у масиві. Другим кроком треба обчислити й зберегти кумулятивну суму (накопичена сума послідовних значень). Створюємо пустий масив розміром вхідного масиву, й заповнюємо його елементами. [15] (див. Додаток Н)

### Сортування за розрядами (Radix Sort)

Сортування за розрядами – це швидкий та стабільний алгоритм сортування. Елементи вхідного масиву повинні бути додатними.

Робота алгоритму працює таким чином, що спочатку впорядковує всі елементи за першим (молодшим) розрядом, потім за другим, потім за третім й т.д., аж до найстаршого (кількість цифр у максимальному елементі). Таким чином дані з масиву впорядковуються. Оскільки, припускається, що кожен розряд приймає значення з невеликого діапазону, то кожен цикл впорядкування можна виконувати швидко і з малими витратами пам'яті. [11] (див. Додаток П)

### Сортування комірками (Bucket sort)

Сортування комірками – це стабільний алгоритм впорядкування, що доцільно використовувати, якщо вхідні дані розподілені рівномірно. Сортування комірками є узагальненням сортування підрахунком.

Ідея алгоритму полягає в розподілені елементів масиву по коміркам (спискам). Створюється скінчена кількість комірок, кожна з яких представляє з себе список. Кожна комірка впорядковується окремо іншим алгоритмом впорядкування або ж рекурсивно алгоритмом впорядкування комірками. [14] (див. Додаток Р)

### Сортування Шелла (Shell Sort)

Сортування Шелла – це алгоритм сортування, що є узагальненням сортування вставками. Сортування Шелла виконує декілька впорядкувань вставками, кожен раз порівнюючи і переставляючи елементи, що розташовані на різній відстані один від одного. Сортування Шелла не є стабільним. Сортування Шелла розробив Дональд Шелл у 1959 році.

Основна ідея алгоритма сортування Шелла полягає в сортуванні елементів, між якими є певний крок (відстань). Такий підмасив впорядковується сортуванням вставками. Спочатку обирається великий крок, який зменшується доти, доки не досягне мінімального значення. [16] (див. Додаток С)

### Висновок до розділу 1

Таким чином теорія та практика програмування показує, що проблема впорядкування даних була, є і буде актуальною. Багато людей цікавляться питанням, який з алгоритмів сортування та в якому випадку є більш результативним. І знову ж таки, досліджуючи існуючий опублікований досвід з вказаного науково-практичного напрямку ми з’ясували, що кількість даних для опрацювання є важливим питанням для вибору того чи іншого алгоритму сортування.

## РОЗДІЛ 2 ПРОГРАМНА РЕАЛІЗАЦІЯ

Основною мовою програмування, якою нами були реалізовані алгоритми сортування та власне головна програма, є C#. А для генерації масивів ми використали мову програмування Python.

Такі рішення були прийняті через особливості вже готових бібліотек вказаних мов. Бібліотеки для роботи із директоріями, файлами та часом найбільш зручно реалізовані мовою C#. Для генерації масивів створено скрипт мовою Python (див. Додаток Т). Програма отримує дані для роботи через аргументи командного рядка. На вхід потрібно подати 4 параметри: розмір згенерованого масиву, кількість масивів, ліву та праву границю діапазону елементів масиву. Після перевірки коректності даних скрипт створює папку з назвою, що відповідає розміру масиву. Далі у цій папці створюється певна кількість текстових файлів з числами по одному в рядку. Кількість створених файлів відповідає другому параметру при запуску програми. Таким чином цей скрипт можна використовувати для генерації псевдовипадкових послідовностей цілих чисел певного розміру. (див. Рис 2.1, див. Рис 2.2, див. Рис 2.3, див. Рис 2.4)

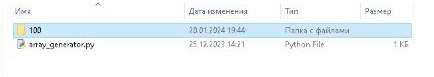


Рис. 2.2

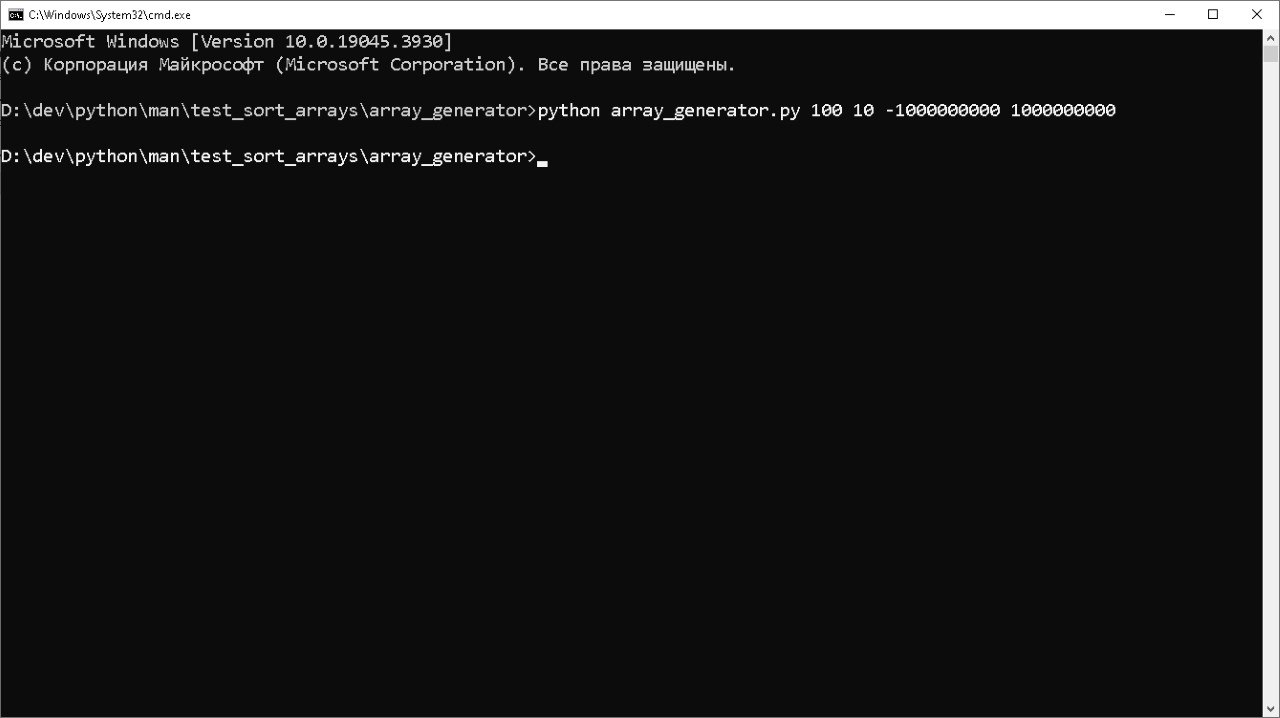


Рис. 2.1

Головна програма, що реалізована мовою C#, отримує на вхід 2 параметри: назву алгоритму сортування та кількість елементів у масиви, який ми хочемо перевірити. Задля коректної роботи програми, на початку перевіряється чи правильна кількість аргументів була надана, в іншому випадку програма закінчує свою роботу. Також перевіряється наявність названого алгоритму сортування, якщо назви алгоритму, яку запропонував користувач, не існує, програма закінчує свою роботу. Також задля коректної роботи програми, перевіряється чи існує директорія з назвою «data», в якій збережені масиви з даними. (див. Рис. 2.5)

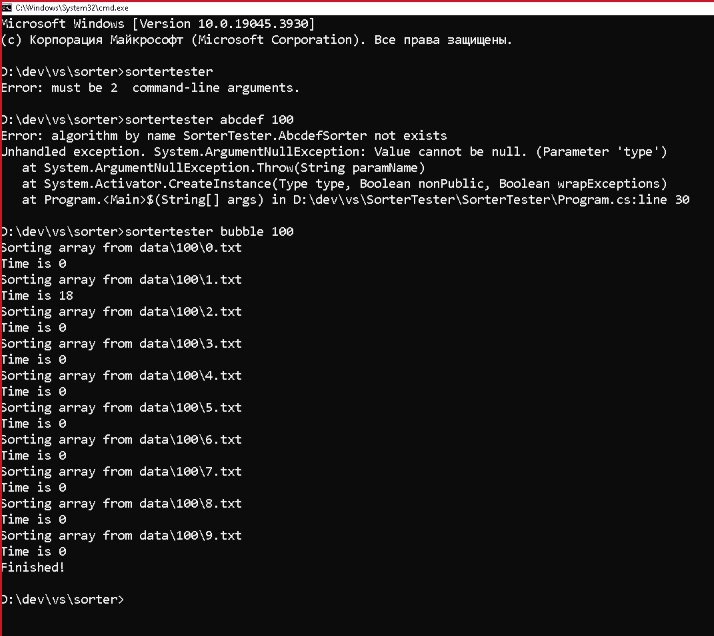


Рис. 2.5

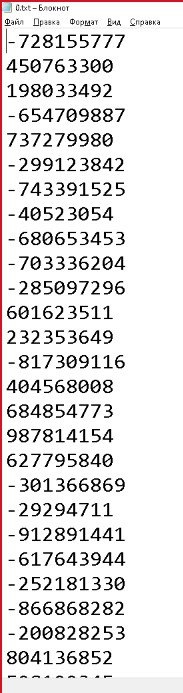


Рис. 2.4

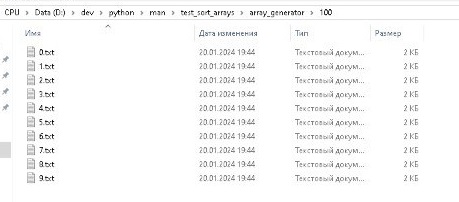


Рис. 2.3

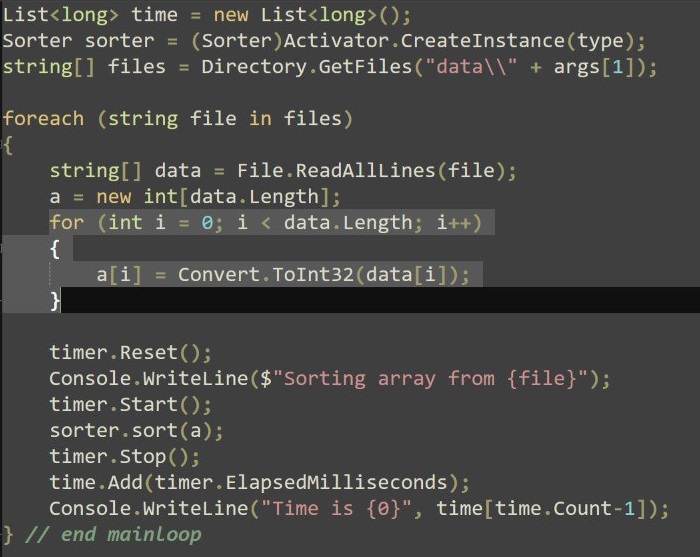


Рис. 2.6

Далі ми, скориставшись механізмом рефлексії (у програмуванні рефлексія вказує на здатність програми відображати свою структуру чи властивості в рантаймі, дозволяючи звертатися до цих елементів та взаємодіяти з ними динамічно. Рефлексія дозволяє програмі визначати та маніпулювати змінними, методами та іншими елементами коду під час виконання програми, а не під час компіляції), створюємо зміну sorter, що є екземпляром того класу, назву якого надав користувач у першому параметрі. Масив даних використовуємо із раніше створеної папки «data», що містить в собі усі масиви, попередньо згенеровані нами з певною кількістю елементів. Зчитуємо масив у вигляді колекції рядків та у циклі конвертуємо їх у цілочисельний тип int (див. Рис. 2.6). Новий отриманий масив і є безпосередньо тим, який ми будемо сортувати.

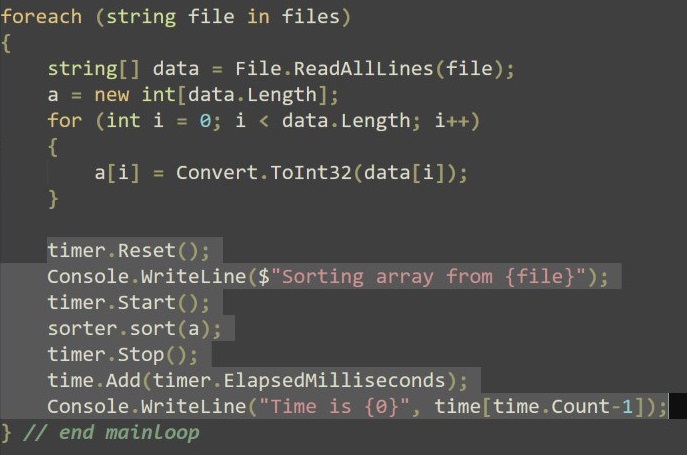


Рис. 2.7

Щоб визначити відносний час роботи того чи іншого алгоритму, витраченого на сортування кожного з масивів даних, використовується цикл, в якому обчислюється час роботи алгоритма, та виводиться в консоль. (див. Рис. 2.7)

Щоб зберегти дані для подальшого використання, створюється нова директорія «reports». У ній створюється файл з розширенням txt, з відповідним ім’ям: Назва алгоритму сортування\_кількість елементів. У файл txt записується усереднений час роботи алгоритму сортування (див. Рис. 2.8). Потім дані з усіх файлів з папки «reports» в ручному режимі переносяться до таблиці Excel та за цією таблицею будується діаграма. Ця діаграма дає достатньо повну інформацію про результати роботи алгоритму, залежно від кількості вхідних даних (див. Рис. 2.9). Отриманий час роботи алгоритмів записаний у мілісекундах. Результати не є абсолютно точні, адже вони залежать від характеристик комп’ютера на яких запускається програма, але дають чітке розуміння різниці між можливостями розглянутих алгоритмів.



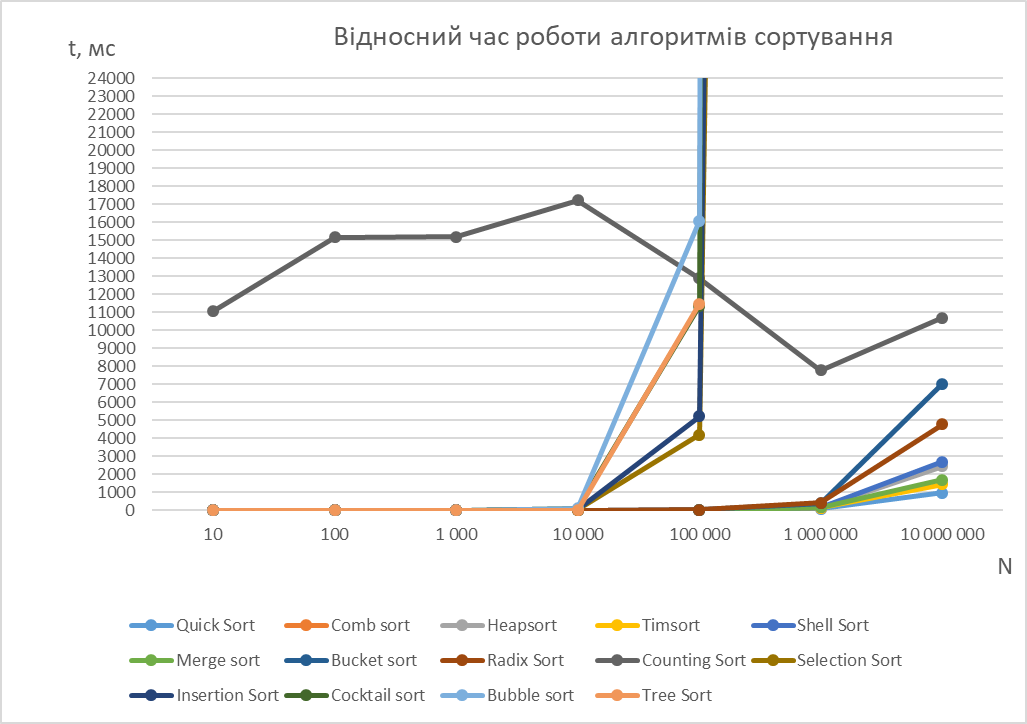
Рис. 2.8

Щоб програма працювала коректно перед створенням директорії «reports», перевіряється чи існує вже ця директорія в користувача, якщо ні, то вона створюється. (див. Рис. 2.8)

Щоб оповістити користувача, що програма закінчила свою роботу, виводиться в консоль «Finished!». (див. Рис. 2.8)

Повний текст програмного продукту розміщено у Додатку У.

Рис. 2.9



### Висновок до розділу 2

Таким чином ми на практиці реалізували традиційні етапи роботи сучасного інженера програмного забезпечення над проблемою – планування роботи, ознайомлення з проблемно-орієнтованою документацією, вибір інструментів реалізації (мови програмування з відповідними бібліотеками, середовище розробки програми), перевірка та інтерпретація результатів, створення власної технічної документації (пояснювальна записка до виконаної роботи з рекомендаціями щодо правил використання розробленого програмного продукту). Також ми дійшли до важливої думки щодо доцільності комбінування засобів розробки – бо інженер програмного забезпечення має розумітися на «сильних сторонах» різних існуючих програмних рішень і комбінувати їх, замість портування усього проєкту на одну платформу.

## ВИСНОВКИ

Провівши власні експерименти, зазначені у розділі 2, нами було з’ясовано особливості використання 14 популярних алгоритмів сортування (Bubble sort, Selection sort, Insertion sort, Cocktail sort, Tree sort, Comb sort, Merge sort, Heapsort, Timsort, Quick sort, Counting sort, Radix sort, Bucket sort, Shell sort).

Ми спланували нашу дослідницьку роботу так, що:

Спершу нашу увагу було сфокусовано на основних принципах, за якими працюють ці алгоритми сортування.

Потім було опрацьовано перевірені джерела інформації, стосовно технології розробки програм з використанням мови програмування C#, так як вона для мене є новою.

Як наслідок, було реалізовано програму (програма реалізована у середовищі розробки Visual Studio Community), яка обчислює відносний час роботи обраного користувачем алгоритму сортування масиву з певною кількістю елементів.

Щоб отримати масиви експериментальних даних з певною кількістю елементів була підготована програма генерації випадкових чисел мовою програмування Python (середовище розробки - Visual Studio Community).

Задля повного розуміння отриманих результатів, була створена діаграма засобами табличного процесора (Excel). (див. Рис. 2.9).

На Рисунку 2.9 видно, що із використаних алгоритмів сортування, найбільш ефективним є алгоритм Quick sort. В той час як самим повільним виявився Bubble sort.

За допомогою діаграми можна спостерігати швидкість роботи обраних нами алгоритмів сортування.

Найбільш важливо те, що діаграма дає орієнтир для вибору ефективного алгоритму сортування, залежно від кількості вхідних даних.

Результати виконаної роботи можуть бути корисними 1) сучасним розробникам інформаційних систем (для створення ефективного бекенду), 2) програмістам, які цікавляться вказаною тематикою, або лише навчаються професії ІПЗ, 3) учням та учителям шкіл, де програмування вивчається поглиблено.

## ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ ТА ДЖЕРЕЛ

1. Timsort // Вікіпедія. URL: <https://uk.wikipedia.org/wiki/Timsort>
2. Алгоритми сортування // Вікіпедія. URL: <https://uk.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F>
3. Пірамідальне сортування// Вікіпедія. URL: <https://uk.wikipedia.org/wiki/%D0%9F%D1%96%D1%80%D0%B0%D0%BC%D1%96%D0%B4%D0%B0%D0%BB%D1%8C%D0%BD%D0%B5_%D1%81%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F>
4. Повний гайд з алгоритмів сортування на Java для новачків ч. 2//r\_d media. URL: <https://robotdreams.cc/uk/blog/442-povniy-gayd-z-algoritmiv-sortuvannya-na-java>
5. Сортування бульбашкою // Вікіпедія. 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%B1%D1%83%D0%BB%D1%8C%D0%B1%D0%B0%D1%88%D0%BA%D0%BE%D1%8E>
6. [Сортування вибором // Вікіпедія. 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%B2%D0%B8%D0%B1%D0%BE%D1%80%D0%BE%D0%BC](file:///D:\my\Навчання\МАН\Роботи\10%20клас\програмування\Сортування%20вибором%20\%20Вікіпедія.%20URL:%20https:\uk.wikipedia.org\wiki\%25D0%25A1%25D0%25BE%25D1%2580%25D1%2582%25D1%2583%25D0%25B2%25D0%25B0%25D0%25BD%25D0%25BD%25D1%258F_%25D0%25B2%25D0%25B8%25D0%25B1%25D0%25BE%25D1%2580%25D0%25BE%25D0%25BC)
7. Сортування вставками // Вікіпедія. 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%B2%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%BD%D1%8F%D0%BC>
8. Сортування вставками //program.top. URL: <https://programm.top/uk/pascal/algorithm/array-sort/insertion-sort/>
9. Сортування гребінцем // Вікіпедія. 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%B3%D1%80%D0%B5%D0%B1%D1%96%D0%BD%D1%86%D0%B5%D0%BC>
10. Сортування двійковим деревом // Вікіпедія. 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%B4%D0%B2%D1%96%D0%B9%D0%BA%D0%BE%D0%B2%D0%B8%D0%BC_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE%D0%BC>
11. Сортування за розрядами // Вікіпедія. 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%B0_%D1%80%D0%BE%D0%B7%D1%80%D1%8F%D0%B4%D0%B0%D0%BC%D0%B8>
12. Сортування злиттям // Вікіпедія. 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#%D0%9F%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4_%D1%80%D0%B5%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%97_%D0%BD%D0%B0_%D0%A1>
13. Сортування змішуванням // Вікіпедія. 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%BC%D1%96%D1%88%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%D0%BC>
14. [Сортування комірками // Вікіпедія. 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%BA%D0%BE%D0%BC%D1%96%D1%80%D0%BA%D0%B0%D0%BC%D0%B8](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%BA%D0%BE%D0%BC%D1%96%D1%80%D0%BA%D0%B0%D0%BC%D0%B8)
15. Сортування підрахунком // Вікіпедія. 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%BF%D1%96%D0%B4%D1%80%D0%B0%D1%85%D1%83%D0%BD%D0%BA%D0%BE%D0%BC>
16. Сортування Шелла // Вікіпедія. 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%A8%D0%B5%D0%BB%D0%BB%D0%B0>
17. Швидке сортування // Вікіпедія. 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>

## ДОДАТКИ

### Додаток А

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class BubbleSorter : Sorter

{

public override void sort(int[] array)

{

int size = array.Length;

bool unsorted = true;

for (int j = 0; j < size && unsorted; j++)

{

unsorted = false;

for (int i = 0; i < size - 1 - j; i++)

{

if (array[i] > array[i + 1])

{

(array[i], array[i + 1]) = (array[i + 1], array[i]);

unsorted = true;

}

}

}

}

}

}

### Додаток Б

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class ChoiceSorter : Sorter

{

public override void sort(int[] array)

{

int i, pos;

int size = array.Length;

for (i = 0; i < size - 1; i++)

{

pos = i;

for (int j = i + 1; j < size; j++)

{

if (array[pos] > array[j]) { pos = j; }

}

if (pos != i)

{

(array[pos], array[i]) = (array[i], array[pos]);

}

}

}

}

}

### Додаток В

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class InsertSorter : Sorter

{

public override void sort(int[] array)

{

int size = array.Length;

for (int i = 1; i < size; i++)

{

for (int j = i; j > 0 && array[j] < array[j - 1]; j--)

{

(array[j], array[j - 1]) = (array[j - 1], array[j]);

}

}

}

}

}

### Додаток Г

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class CocktailSorter : Sorter

{

public override void sort(int[] array)

{

int size = array.Length;

bool swapFlag = true;

for (int i = 0; i < size / 2 && swapFlag; i++)

{

swapFlag = false;

for (int j = i; j < size - i - 1; j++)

{

if (array[j] > array[j + 1])

{

(array[j], array[j + 1]) = (array[j + 1], array[j]);

swapFlag = true;

}

}

for (int j = size - 2 - i; j > i; j--)

{

if (array[j - 1] > array[j])

{

(array[j - 1], array[j]) = (array[j], array[j - 1]);

swapFlag = true;

}

}

}

}

}

}

### Додаток Д

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

class TreeNode

{

public TreeNode(int data)

{

Data = data;

}

public int Data { get; set; }

public TreeNode Left { get; set; }

public TreeNode Right { get; set; }

public void Insert(TreeNode node)

{

if (node.Data < Data)

{

if (Left == null) { Left = node; }

else { Left.Insert(node); }

}

else

{

if (Right == null) { Right = node; }

else { Right.Insert(node); }

}

} // end insert

public int[] Transform(List<int> elements = null)

{

if (elements == null)

{

elements = new List<int>();

}

if (Left != null)

{

Left.Transform(elements);

}

elements.Add(Data);

if (Right != null)

{

Right.Transform(elements);

}

return elements.ToArray();

} // end transform

} // end class treenode

internal class TreeSorter : Sorter

{

public override void sort(int[] array)

{

TreeNode tree = new TreeNode(array[0]);

for (int i = 1; i < array.Length; i++)

{

tree.Insert(new TreeNode(array[i]));

}

int[] temp = tree.Transform();

for (int i = 0; i < temp.Length; i++)

{

array[i] = temp[i];

}

} // end sort

} // end class

} // end namespace

### Додаток Е

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class CombSorter : Sorter

{

int GetNextStep(int s)

{

s = s \* 1000 / 1247;

return s > 1 ? s : 1;

}

public override void sort(int[] array)

{

int size = array.Length;

int currentStep = size - 1;

while (currentStep > 1)

{

for (int i = 0; i + currentStep < size; i++)

{

if (array[i] > array[i + currentStep])

{

(array[i], array[i + currentStep]) = (array[i + currentStep], array[i]);

}

}

currentStep = GetNextStep(currentStep);

}

for (var i = 1; i < size; i++)

{

bool swapFlag = false;

for (var j = 0; j < size - i; j++)

{

if (array[j] > array[j + 1])

{

(array[j], array[j + 1])= (array[j+1], array[j]);

swapFlag = true;

}

}

if (!swapFlag)

{

break;

}

}

} // end sort

} // end class

} // end namespace

### Додаток Ж

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class MergeSorter : Sorter

{

void Merge(int[] array, int lowIndex, int middleIndex, int highIndex)

{

var left = lowIndex;

var right = middleIndex + 1;

var tempArray = new int[highIndex - lowIndex + 1];

var index = 0;

while ((left <= middleIndex) && (right <= highIndex))

{

if (array[left] < array[right])

{

tempArray[index] = array[left];

left++;

}

else

{

tempArray[index] = array[right];

right++;

}

index++;

}

for (var i = left; i <= middleIndex; i++)

{

tempArray[index] = array[i];

index++;

}

for (var i = right; i <= highIndex; i++)

{

tempArray[index] = array[i];

index++;

}

for (var i = 0; i < tempArray.Length; i++)

{

array[lowIndex + i] = tempArray[i];

}

} // end merge

public void MergeSort(int[] array, int lowIndex, int highIndex)

{

if (lowIndex < highIndex)

{

var middleIndex = (lowIndex + highIndex) / 2;

MergeSort(array, lowIndex, middleIndex);

MergeSort(array, middleIndex + 1, highIndex);

Merge(array, lowIndex, middleIndex, highIndex);

}

}

public override void sort(int[] array)

{

MergeSort(array, 0, array.Length - 1);

} // end sort

} // end class

} // end namespace

### Додаток К

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class HeapSorter : Sorter

{

public override void sort(int[] arr)

{

int size = arr.Length;

for (int i = size / 2 - 1; i >= 0; i--)

heapify(arr, size, i);

for (int i = size - 1; i >= 0; i--)

{

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapify(arr, i, 0);

}

} // end sort

void heapify(int[] arr, int n, int i)

{

int largest = i;

int l = 2 \* i + 1; // left = 2\*i + 1

int r = 2 \* i + 2; // right = 2\*i + 2

if (l < n && arr[l] > arr[largest])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i)

{

int swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

heapify(arr, n, largest);

}

}

} // end class

} // end namespace

### Додаток Л

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class TimSorter : Sorter

{

private const int MIN\_MERGE = 32;

public override void sort(int[] array)

{

int n = array.Length;

for (int i = 0; i < n; i += MIN\_MERGE)

{

int minRun = Math.Min(MIN\_MERGE, n - i);

InsertionSort(array, i, i + minRun - 1);

}

for (int size = MIN\_MERGE; size < n; size = 2 \* size)

{

for (int left = 0; left < n; left += 2 \* size)

{

int mid = left + size - 1;

int right = Math.Min((left + 2 \* size - 1), (n - 1));

if (mid < right)

{

Merge(array, left, mid, right);

}

}

}

}

private void InsertionSort(int[] array, int left, int right)

{

for (int i = left + 1; i <= right; i++)

{

int j = i;

while (j > left && array[j] < array[j - 1])

{

Swap(array, j, j - 1);

j--;

}

}

}

private void Merge(int[] array, int left, int mid, int right)

{

int len1 = mid - left + 1;

int len2 = right - mid;

int[] leftArray = new int[len1];

int[] rightArray = new int[len2];

Array.Copy(array, left, leftArray, 0, len1);

Array.Copy(array, mid + 1, rightArray, 0, len2);

int i = 0, j = 0;

int k = left;

while (i < len1 && j < len2)

{

if (leftArray[i] <= rightArray[j])

{

array[k++] = leftArray[i++];

}

else

{

array[k++] = rightArray[j++];

}

}

while (i < len1)

{

array[k++] = leftArray[i++];

}

while (j < len2)

{

array[k++] = rightArray[j++];

}

}

private void Swap(int[] array, int i, int j)

{

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

} // end class

} // end namespace

### Додаток М

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class QuickSorter: Sorter

{

public override void sort(int[] array)

{

if (array == null || array.Length <= 1)

return;

QuickSort(array, 0, array.Length - 1);

}

private void QuickSort(int[] array, int low, int high)

{

if (low < high)

{

int partitionIndex = Partition(array, low, high);

QuickSort(array, low, partitionIndex - 1);

QuickSort(array, partitionIndex + 1, high);

}

}

private int Partition(int[] array, int low, int high)

{

int pivot = array[high];

int i = low - 1;

for (int j = low; j < high; j++)

{

if (array[j] <= pivot)

{

i++;

Swap(array, i, j);

}

}

Swap(array, i + 1, high);

return i + 1;

}

private void Swap(int[] array, int i, int j)

{

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

} // end class

} // end namespace

### Додаток Н

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class CountingSorter: Sorter

{

public override void sort(int[] array)

{

if (array == null || array.Length <= 1)

return;

int max = FindMaxValue(array);

int[] countArray = new int[max + 1];

int[] sortedArray = new int[array.Length];

// Count occurrences of each element

foreach (int num in array)

{

countArray[num]++;

}

// Update countArray to store the position of each element in the sortedArray

for (int i = 1; i < countArray.Length; i++)

{

countArray[i] += countArray[i - 1];

}

// Build the sortedArray

for (int i = array.Length - 1; i >= 0; i--)

{

sortedArray[countArray[array[i]] - 1] = array[i];

countArray[array[i]]--;

}

// Copy the sortedArray back to the original array

Array.Copy(sortedArray, array, array.Length);

}

private int FindMaxValue(int[] array)

{

int max = array[0];

for (int i = 1; i < array.Length; i++)

{

if (array[i] > max)

{

max = array[i];

}

}

return max;

}

}

}

### Додаток П

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class RatingSorter: Sorter

{

public override void sort(int[] array)

{

if (array == null || array.Length <= 1)

return;

int max = FindMaxValue(array);

int exp = 1;

while (max / exp > 0)

{

CountingSort(array, exp);

exp \*= 10;

}

}

private void CountingSort(int[] array, int exp)

{

int n = array.Length;

int[] output = new int[n];

int[] count = new int[10];

// Count occurrences of each digit

for (int i = 0; i < n; i++)

{

count[(array[i] / exp) % 10]++;

}

// Update count array to store the position of each digit in the output array

for (int i = 1; i < 10; i++)

{

count[i] += count[i - 1];

}

// Build the output array

for (int i = n - 1; i >= 0; i--)

{

output[count[(array[i] / exp) % 10] - 1] = array[i];

count[(array[i] / exp) % 10]--;

}

// Copy the output array back to the original array

Array.Copy(output, array, n);

}

private int FindMaxValue(int[] array)

{

int max = array[0];

for (int i = 1; i < array.Length; i++)

{

if (array[i] > max)

{

max = array[i];

}

}

return max;

}

}

}

### Додаток Р

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class BucketSorter: Sorter

{

public override void sort(int[] array)

{

if (array == null || array.Length <= 1)

return;

int n = array.Length;

List<int>[] buckets = new List<int>[n];

// Имитация диапазона от -1 миллиарда до 1 миллиарда

int minValue = -1000000000;

int maxValue = 1000000000;

for (int i = 0; i < n; i++)

{

buckets[i] = new List<int>();

}

// Распределение элементов по ячейкам

for (int i = 0; i < n; i++)

{

int bucketIndex = (int)((long)(n - 1) \* (array[i] - minValue) / (maxValue - minValue));

buckets[bucketIndex].Add(array[i]);

}

// Сортировка каждой ячейки (bucket)

for (int i = 0; i < n; i++)

{

buckets[i].Sort();

}

// Объединение отсортированных ячеек

int index = 0;

for (int i = 0; i < n; i++)

{

for (int j = 0; j < buckets[i].Count; j++)

{

array[index++] = buckets[i][j];

}

}

}

}

}

### Додаток С

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SorterTester

{

internal class ShellSorter: Sorter

{

public override void sort(int[] array)

{

if (array == null || array.Length <= 1)

return;

int n = array.Length;

int gap = n / 2;

while (gap > 0)

{

for (int i = gap; i < n; i++)

{

int temp = array[i];

int j = i;

while (j >= gap && array[j - gap] > temp)

{

array[j] = array[j - gap];

j -= gap;

}

array[j] = temp;

}

gap /= 2;

}

}

}

}

### Додаток Т

import os

import sys

from random import randint as ri

if len(sys.argv)<5:

print("Error: too few arguments. Must be 5 parameters.")

exit(1)

args=[int(i) for i in sys.argv[1:]]

array\_size=args[0]

array\_count=args[1]

left\_limit=args[2]

right\_limit=args[3]

if os.path.exists(sys.argv[1]):

print("Folder by name", sys.argv[1], "already exists.")

os.mkdir(sys.argv[1])

for i in range(array\_count):

f=open(sys.argv[1]+"\\"+str(i)+".txt", "w")

for j in range(array\_size):

num=ri(left\_limit, right\_limit+1)

f.write(str(num)+"\n")

f.close()

### Додаток У

using SorterTester;

using System.Diagnostics;

if (args.Length < 2)

{

Console.WriteLine("Error: must be 2 command-line arguments.");

return;

}

if (!Directory.Exists("data"))

{

Console.WriteLine("Error: no data to produce");

return;

}

Stopwatch timer = new Stopwatch();

string sorterName = "SorterTester." + char.ToUpper(args[0][0]) + args[0].Substring(1) + "Sorter";

Type type = Type.GetType(sorterName);

if (type == null)

{

Console.WriteLine("Error: algorithm by name {0} not exists", sorterName);

//return;

}

int[] a = { 3, 2, 1 };

List<long> time = new List<long>();

Sorter sorter = (Sorter)Activator.CreateInstance(type);

string[] files = Directory.GetFiles("data\\" + args[1]);

foreach (string file in files)

{

string[] data = File.ReadAllLines(file);

a = new int[data.Length];

for (int i = 0; i < data.Length; i++)

{

a[i] = Convert.ToInt32(data[i]);

}

timer.Reset();

Console.WriteLine($"Sorting array from {file}");

timer.Start();

sorter.sort(a);

timer.Stop();

time.Add(timer.ElapsedMilliseconds);

Console.WriteLine("Time is {0}", time[time.Count-1]);

} // end mainloop

if (!Directory.Exists("reports"))

{

Directory.CreateDirectory("reports");

}

double ave = time.Average();

File.WriteAllText($"reports\\{sorterName}\_{args[1]}.txt", Convert.ToString(ave));

Console.WriteLine("Finished!");