Дніпровський ліцей інформаційних технологій

при Дніпропетровському національному університеті

імені Олеся Гончара

Кафедра інформатики

**КУРСОВА РОБОТА**

**Тема:** Порівняльний аналіз ефективності різних алгоритмів сортування масивів в залежності від початкового стану масиву

**Виконав:**

Ліцеїст 10-Г класу

*Каманцев Артем Сергійович*

**Керівники:**

*Ентін Йосиф Абрамович, вчитель-методист, к. ф.-м. н.*

Дніпро

2017

ЗМІСТ

**ВСТУП** ................................................................................................ 3

Тема роботи ……………………………….…………………... 3

Актуальність роботи ………………………….………………. 3

Мета роботи ……………………….…………………………... 4

Задачі роботи …………………….……………………………. 4

**ОСНОВНА ЧАСТИНА** .................................................................... 5

1. Оцінка ефективності алгоритмів сортування….................. 5
2. Опис роботи.............................................................................6
3. Блок-схема .............................................................................19
4. Напрямки, де можна використати роботу……………...... 19
5. Програмні та апаратні вимоги…………............................. 20
6. Використані програми……….............................................. 20
7. Комплектація програм………….…...…...………………... 21

**ВИСНОВКИ** ..................................................................................... 22

**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ** .................................... 23

**ДОДАТОК 1**……………………………………………………….. 24

**ДОДАТОК 2**……………………………………………………….. 26

Вступ

***Тема роботи***

Сортування – проблема, яка до появи комп’ютерів забирала чимало людино-годин продуктивної праці. Перші сортувальні машини були винайдені ще в XIX сторіччі, а з появою ЕОМ розроблені нові алгоритми сортування. Нині сортування, як правило, виконується машинами, бо обсяги даних, які потребують сортування, ростуть і навіть потужні комп’ютери не справляються з їх обробкою. В роботі програміста часто постає питання сортування величезних обсягів даних за відносно невеликий відрізок часу, яке перетікає у питання вибору найбільш придатного до конкретних умов алгоритму сортування. Я спробував розглянуте вказане питання.

***Актуальність роботи***

Актуальність роботи в наступному. Існує широкий вибір алгоритмів сортування але питання про вибір найбільш ефективного за певних початкових даних не завжди просте. Моя робота дозволяє наочно порівняти ефективність алгоритмів при різних початкових станах масиву, чим допомагає у вирішенні цього питання. Окрему увагу приділено сортуванню підрахунком, яке, на перший погляд, здається найефективнішим через лінійний час виконування сортування. По-друге, у роботі відображені тенденції впорядкованості масивів при випадковій генерації. Це дозволяє приблизно дослідити впорядкованість даних, якщо правила їх формування достеменно не відомі, що дозволить скорегувати вибір алгоритму сортування.

***Мета роботи***

Головна мета роботи - дослідження та порівняння ефективності відомих алгоритмів сортування і впливу на їх ефективність початкової впорядкованості даних, а також вияв закономірності в отриманих результатах. Результати відображуються за допомогою графіків, які дозволяють виявити певні закономірності у висвітлених питаннях.

***Задачі роботи***

У роботі розглянуто декілька задач, пов’язаних із ефективністю алгоритмів сортування в певних умовах.

1. Дослідити зв’язок ефективності роботи алгоритмів сортування зі впорядкованістю початкового масиву, яка обумовлена СНП (сусідніми невпорядкованими парами) елементів в ньому.

2. Дослідити зв’язок невпорядкованості масиву, обумовленої такими парами елементів, із кількістю елементів у ньому при випадковому генеруванні даних.

3. Проаналізувати роботу сортування підрахунком на основі даних комп’ютерного експерименту і виявити його слабкі сторони та визначити межі його застосування.

Основна частина

1. **Оцінка ефективності алгоритмів сортування**

Оцінка ефективності роботи алгоритмів сортування проводиться за такими факторами: 1) кількість виконаних порівнянь, 2) кількість зроблених обмінів між елементами в масиві, 3) необхідний об’єм пам’яті. Кількість порівнянь, виконаних алгоритмом, позначають O(f(n)), де n – кількість елементів. Ця залежність може змінюватися в залежності від вхідних даних. Розглянемо ефективність наведених алгоритмів сортування в трьох випадках.

*Сортування вставками*.

* Найгірший час –
* Середній час –
* Найкращий час –
* Додаткова пам’ять –

*Сортування бінарними вставками*.

* Найгірший час –
* Середній час –
* Найкращий час –
* Додаткова пам’ять –

*QuickSort*.

* Найгірший час –
* Середній час -
* Найкращий час -
* Додаткова пам’ять –

1. ***Опис роботи***
   1. **Елементи програми**

Робота виконана у програмному середовищі Microsoft Visual Studio 2015 Community мовою C# .NET. Для розробки дизайну використовувалась технологія WPF (Windows Presentation Foundation) через її переваги у створенні динамічного інтерфейсу користувача, порівняно з Windows Forms. Використані такі елементи управління.

* Grid, StackPanel,TreeView – для компонування елементів інтерфейсу.
* WindowsFormsHost – для імпорту елементів управління, які не входять до базового набору компонентів WPF(зокрема Chart).
* Label, TextBox, Button, ComboBox, WebBrowser – для взаємодії з користувачем і відображення текстової інформації та різного походження(статичної та змінюваної програмно)
* Chart, Canvas – для відображення графічно отриманих даних та візуалізації
* MessageBox – для виведення повідомлень про помилку

Проект складається з п’яти розділів, три з яких відповідають задачам, висвітленим у роботі («Порівняння сортувань», «Впорядкованість масиву», «Ефективність сортування підрахунком») та двох допоміжних розділів, що допомагають краще зрозуміти поданий матеріал («Теорія» та «Візуалізація»).

Основною частиною у вихідній структурі програми є клас Sort (файл Sort.cs), який реалізує алгоритми сортування і об’єкти якого забезпечують вхідні дані. Ці алгоритми добре відомі, не було суттєвих складнощів у їх реалізації. Значно складніше виглядає і займає більше обсягу службовий та керуючий код. Трохи складнощів у реалізації програми додало використання технології WPF. Зокрема постали такі питання.

1. Необхідності підключення елементу Chart, якого немає серед стандартних контролерів WPF.
2. Візуалізація алгоритмів сортування шляхом програмного створення анімації, в залежності від вхідних даних.
   1. **Розділи програми**
      1. *Головне меню*

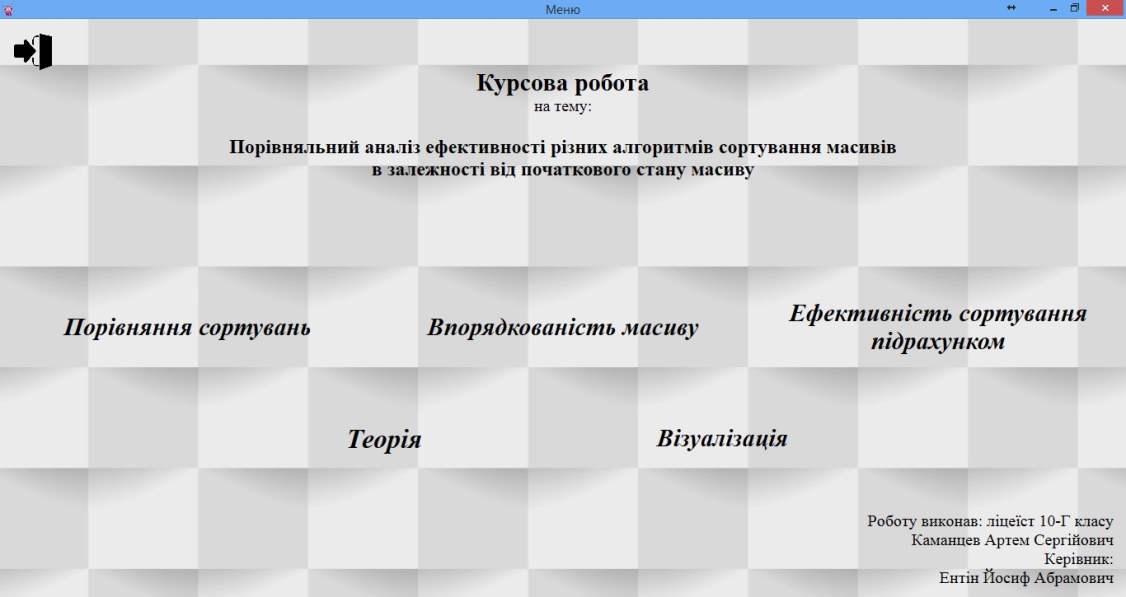


Рис. 2.1. Головне меню

При запуску програми відкривається головне меню (рис. 2.1.), з якого можна перейти до одного з п’яти розділів або вийти з програми. Повернення до головного меню доступне з будь-якого розділу і відбувається при натисканні на таку кнопку (рис. 2.2.).



Рис. 2.2. Кнопка повернення до головного меню

Якщо введено невірні дані до текстових полів, програма виведе повідомлення про помилку у MessegeBox, де буде вказана причина помилки (рис. 2.3.). В деяких випадках користувачу буде запропоновано зчитати дані з файлу (зокрема, у розділах «Порівняння сортувань» та «Візуалізація»).

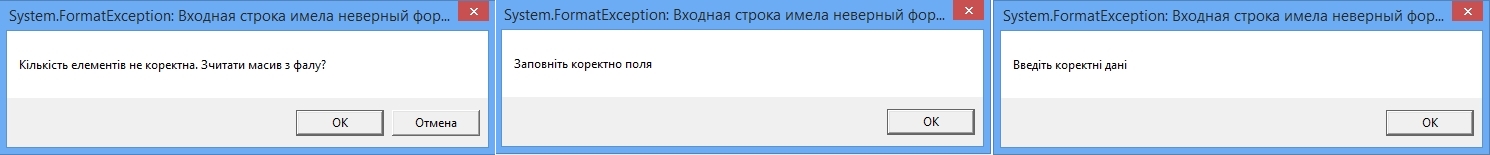
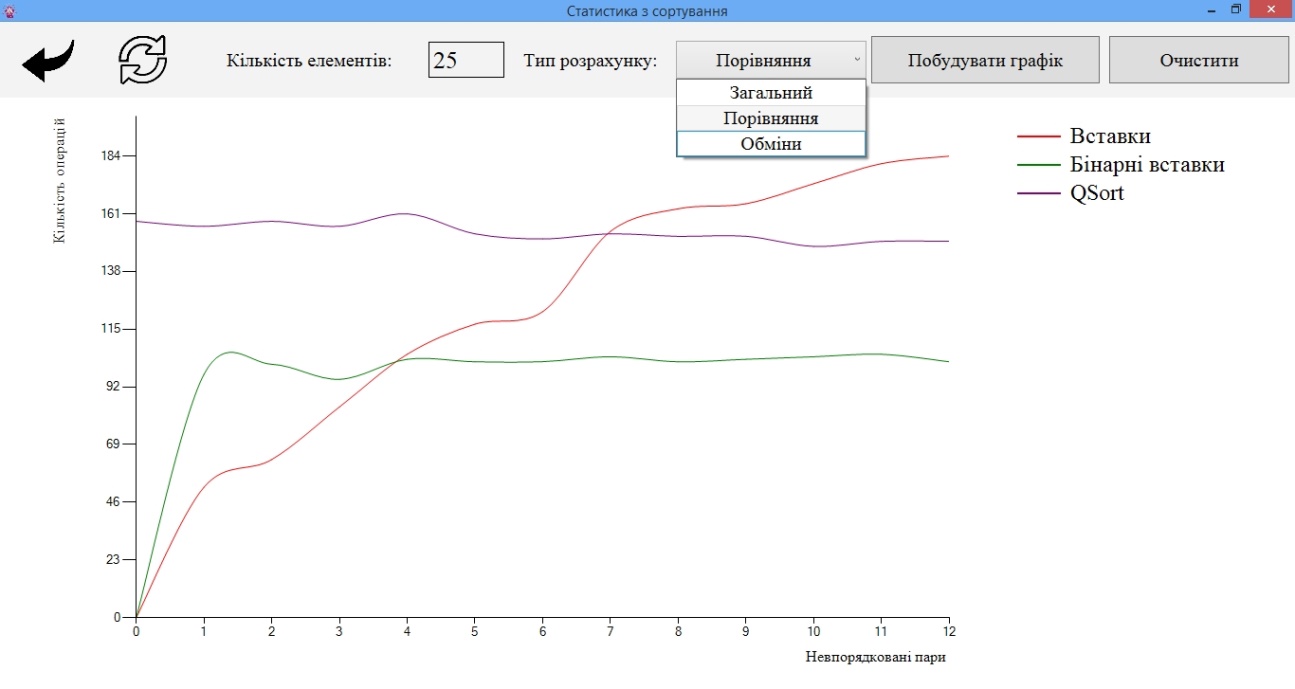


Рис. 2.3. Повідомлення про помилку

Розглянемо детальніше особливості інтерфейсу кожного з трьох основних розділів.

* + 1. *Порівняння сортувань*

Рис. 2.4. Інтерфейс розділу «Статистика з сортування»



У даному розділі представлено залежність кількості операцій, які виконуються при кожному з трьох сортувань, від початкової впорядкованості масиву. Загальна кількість операцій при сортуванні складається з порівнянь ( як правило, їх кількість і визначає ефективність алгоритму) та обмінів значеннями елементів. ComboBox надає можливість обрати, за яким типом операції будуть порівнюватись алгоритми. Якщо обрано пункт «Загальна», в результатах будуть враховані як порівняння, так і обміни. Досягти бажаної кількості СНП можна шляхом зміни кількості елементів відповідно до графіку, представленого в наступному розділі. Цей розділ дає можливість побачити, скільки виконано операцій порівнянь/обмінів для поточної генерації, оскільки зміна даних у програмі відбувається лише після натискання на кнопку **. Далі наведено приклади графіків отриманих у цьому розділі.З цих даних можна зробити наступні висновки.

Якщо порівнювати ефективність алгоритмів сортування за кількістю порівнянь, отримаємо графік, близький до наступного (рис. 2.5).

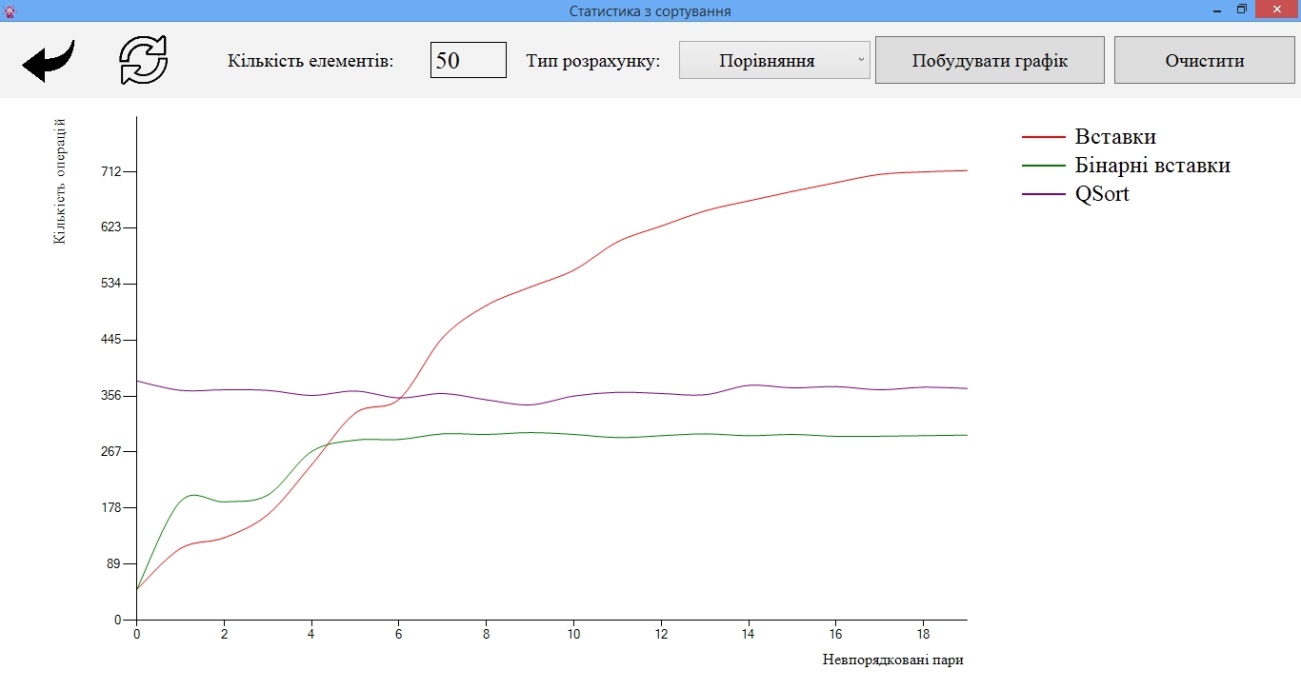


Рис 2.5. Приклад порівняння сортувань за кількістю зроблених порівнянь

З графіку видно наступне.

* Кількість операцій порівняння при сортуванні вставками зростає пропорційно зменшенню впорядкованості вхідних даних (до зростання кількості СНП).
* Кількість операцій порівняння при сортуванні бінарними вставками не залежить від впорядкованості даних**,**  адже бінарний пошук на кожному кроці робить перевірок.
* Кількість операцій порівняння для QuickSort не залежить від кількості СНП, падіння ефективності даного сортування до відбувається при формуванні даних невигідним чином відносно вибору опорного елементу. Це стається через те, що в якості опорного елементу вибирається максимальне або мінімальне значення, що на кожному кроці спричиняє незбалансоване розділення масиву на частини: 1 та n-1. Така ситуація призводить не тільки до зростання кількості операцій до , а й до досягнення рекурсією глибини рівної n, що може призвести до переповнення стеку. Оскільки значення опорного елементу розраховується як середнє значення між максимальним і мінімальним значеннями вхідних даних, то така ситуація маловірогідна і важко прогнозована
* Кількість операцій порівняння при сортуванні бінарними вставками завжди залишається трохи меншою від кількості операцій порівняння при сортуванні QuickSort.

Якщо порівняння проводити за кількістю обмінів, отримаємо графік, близький до наступного(рис. 2.6).

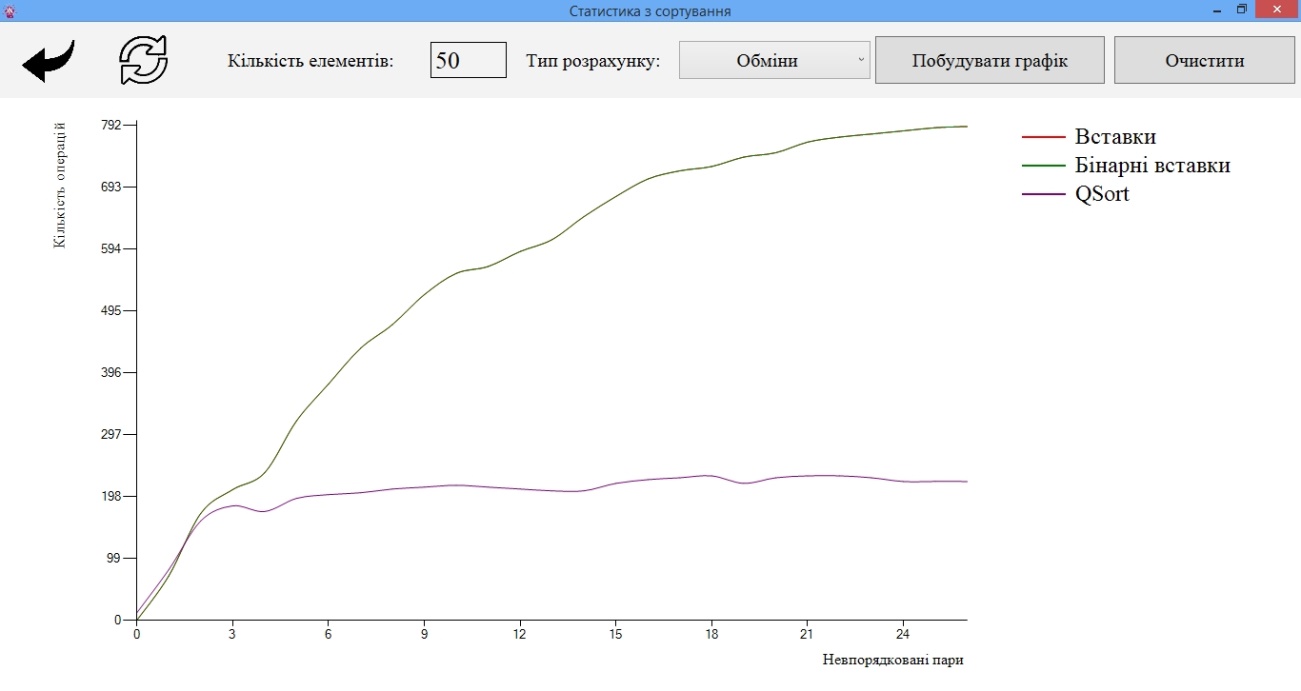


Рис. 2.6. Приклад порівняння сортувань за кількістю зроблених обмінів

З графіку видно наступне.

* Кількості операцій обміну при сортуванні вставками та бінарними вставками повністю збігаються і зростають пропорційно зменшенню впорядкованості вхідних даних.
* Кількість операцій обміну при сортуванні QuickSort не залежить від початкової впорядкованості даних. Тільки при повній впорядкованості даних кількість операцій обміну значно знижується, але не дорівнює 0 через те, що QuickSort не є стійким алгоритмом(змінює взаємне розташування елементів з однаковими ключами).

Якщо об’єднати попередньо отримані результати і проводити порівняння алгоритмів за загальною кількістю операцій, отримаємо графік, близький до наступного (рис. 2.7).

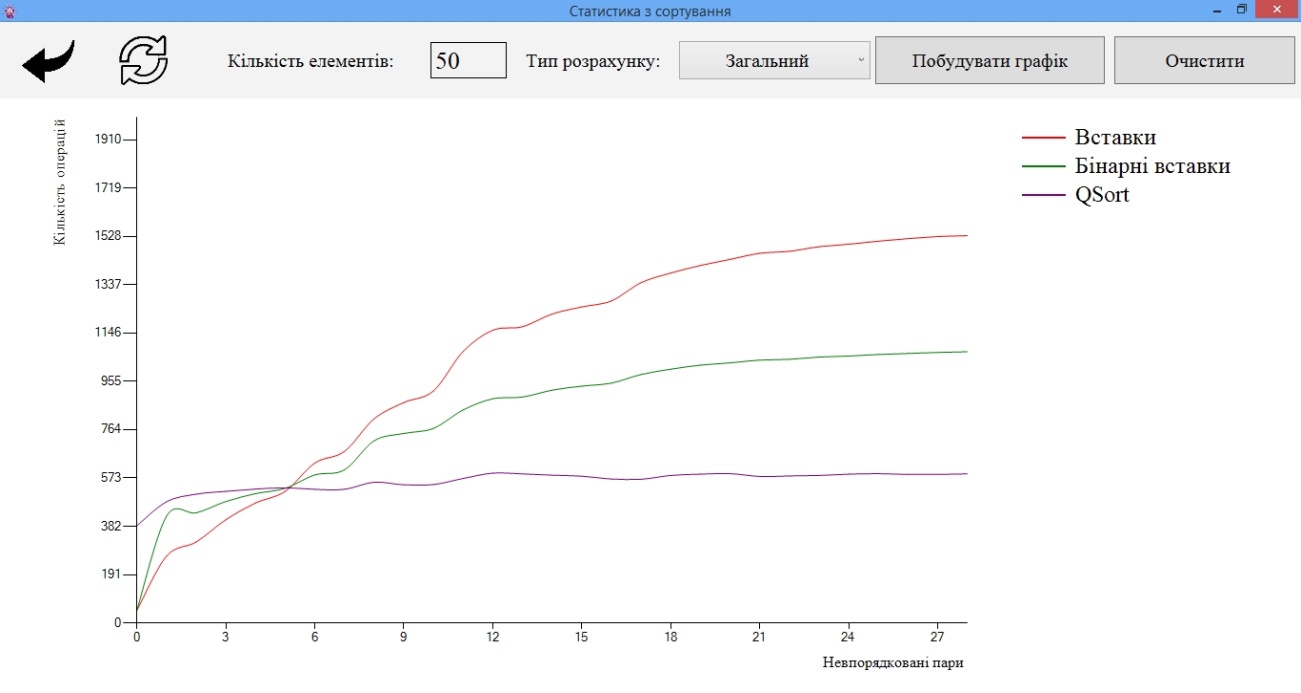


Рис. 2.7. Приклад порівняння сортувань за загальною кількістю зроблених операцій

З графіку видно таке.

* Якщо кількість СНП у вхідних даних менше або приблизно дорівнює 5, найефективнішим виявляється сортування вставками, менш ефективне сортування бінарними вставками і найгірший результат дає QuickSort.
* Якщо кількість СНП у вхідних даних більша від 5 (стабільно такий результат при кількості СНП, більшої від 10), найефективнішим є QuickSort. Кількість його операцій, у загальному випадку, не залежить від початкової впорядкованості масиву, бо на ефективність його роботи впливає не кількість СНП, а принцип вибору опорного елементу. Кількість операцій при сортуванні вставками та бінарними вставками зростає пропорційно кількості СНП (зменшенню впорядкованості вхідних даних), причому кількість операцій, виконаних звичайними вставками, зростає значно швидше порівняно з бінарними.

Експериментально можна встановити, що при зростанні кількості елементів у вхідних даних загальна кількість операцій, виконуваних при сортуванні, зростає для всіх трьох алгоритмів сортування.

*Підсумок.* Кількість виконаних операцій, при зменшенні впорядкованості масиву, швидше за все зростає у звичайних вставок, трохи повільніше в бінарних, а найкращі результати демонструє QuickSort. Це твердження справедливе для вхідних даних, кількість невпорядкованих пар у яких перевищує 10. Ситуація змінюється на кардинально протилежну якщо треба відсортувати дані, що містять мало СНП, або якщо потрібно знайти місце для вставлення одного елементу у вже відсортований масив. Отже, QuickSort є ефективним при сортуванні великих обсягів погано впорядкованих вхідних даних. Звичайні вставки виявляються ефективнішими при сортуванні майже впорядкованих даних, а оптимізований варіант - бінарні вставки краще використовувати коли початкова впорядкованість стоїть під сумнівом (якщо масив виявиться добре впорядкованим, то ми отримаємо відносно невеликі втрати у виконаних операціях порівняно зі звичайними вставками, але якщо масив буде погано впорядкований, то ми отримаємо не таке стрімке зростання кількості необхідних операцій як при використані звичайних вставок).

* + 1. *Впорядкованість масиву*

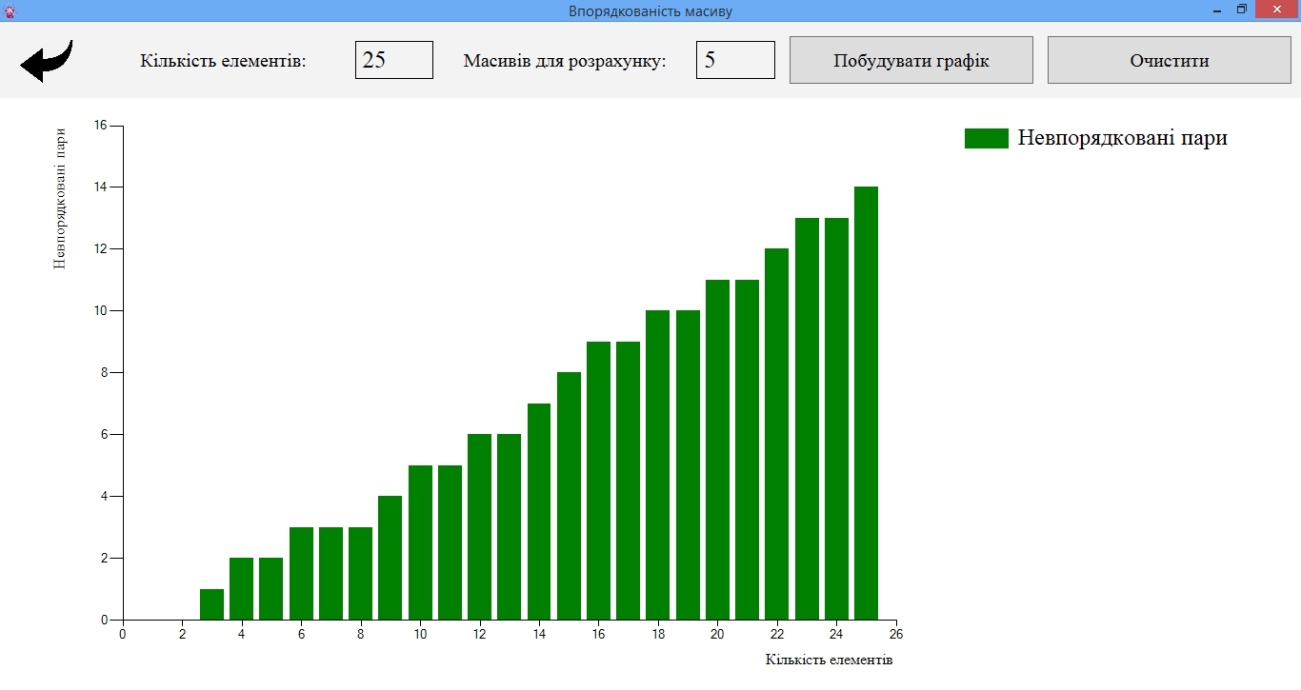


Рис. 2.8. Інтерфейс розділу «Впорядкованість масиву»

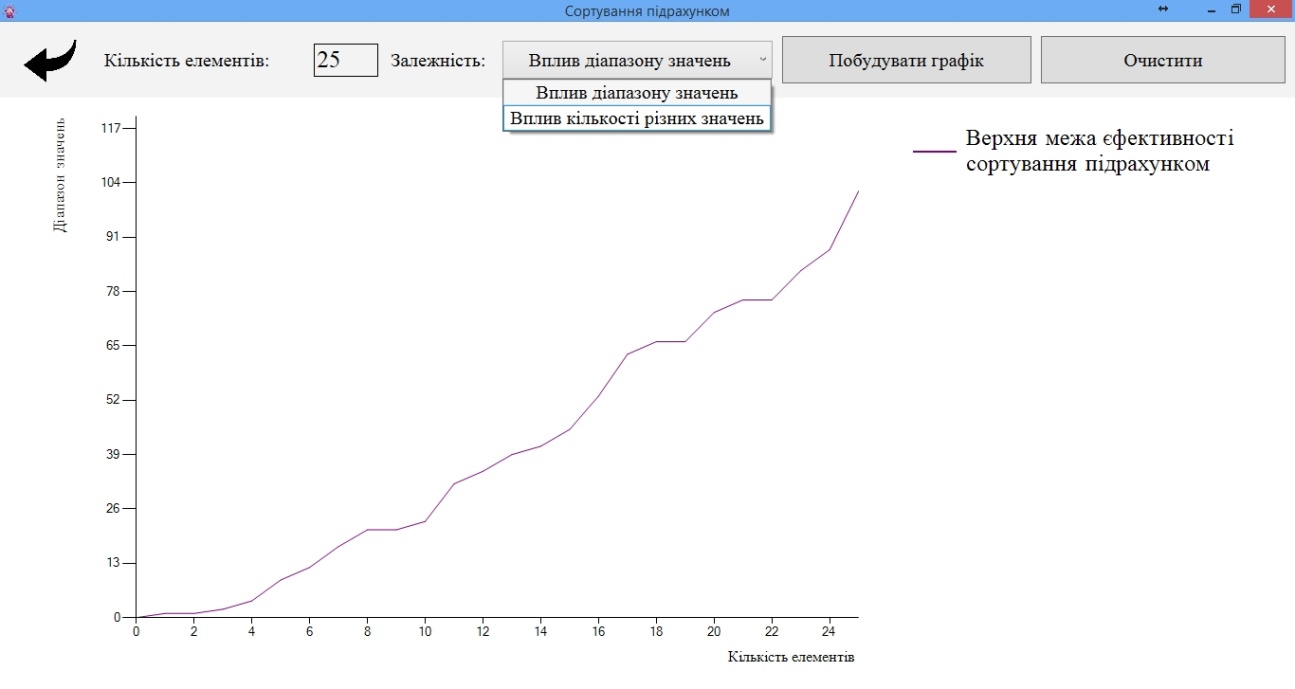
Рис. 2.7

Цей розділ надає можливість побудувати графік залежності кількості СНП від кількості елементів у масиві при випадковій генерації даних. Користувач може вводити максимальну кількість елементів, для яких будуть проведені розрахунки. Параметр «Масивів для розрахунку» визначає, скільки разів буде змодельована дана ситуація шляхом повторної генерацій масиву для розрахунку середніх даних і виведення їх на графіку.

Впорядкованість масиву має обернено пропорційну залежність від кількості елементів у ньому, тобто зі збільшенням елементів в масиві зростає кількість СНП. Головним чином це залежить від способу отримання даних. У моєму випадку, при генерації елементів за допомогою об’єкта Random, рівняння графіку можна приблизно описати формулою: , де y – СНП, x- кількість елементів.

* + 1. *Ефективність сортування підрахунком*

Рис. 2.9. Інтерфейс розділу «Сортування підрахунком»



У цьому розділі представлено розрахунки, які відображують верхню межу ефективності сортування підрахунком. ComboBox визначає, який вплив буде досліджуватись і, відповідно, яка реалізація даного алгоритму буде використана.

При аналізі впливу діапазону значень на ефективність сортування підрахунком, досліджується його класична реалізація: ми створюємо допоміжний масив C [0..k], що складається з нулів, потім послідовно читаємо елементи вхідного масиву A, для кожного A [i] збільшуємо C [A [i]] на одиницю. Тепер досить пройти по масиву C, для кожного i, що належить {0,…k}, і записати в масив A послідовно число i C [i] раз. Ця реалізація має складність 2n+k, де n – кількість елементів, k – діапазон значень. Класичний алгоритм сортування підрахунком залишається ефективним доти, доки виконується нерівність ( – складність QuickSort). Розглянемо приклад. Нехай є початкова послідовність:

[10, 5, 9, 4, 8, 3, 7, 2, 6, 1] – k буде дорівнювати 10, а допоміжний масив C буде мати вигляд: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]. За формулою 2n+k отримуємо кількість операцій, що здійснить сортування підрахунком – 30. При тій самій кількості елементів QuickSort виконує приблизно 33 операції порівняння(, а отже , що підтверджується експериментально). Як ми бачимо, рівняння виконується (30<33). Але якщо ми збільшимо діапазон значень:

[20, 5, 9, 4, 8, 3, 7, 2, 6, 1], то k буде дорівнювати 20, а допоміжний масив C буде мати вигляд: [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]. Кількість операцій зросте до 40, що вже перевищує кількість операцій виконуваних QuickSort. Зі збільшенням діапазону значень кількість пустих позицій у допоміжному масиві буде збільшуватись, що виявляє слабку сторону цього алгоритму і змушує шукати способи його вдосконалення.

Класичний алгоритм залишається ефективним доки виконується нерівність: , що і відображується на наступному графіку (рис 2.10).

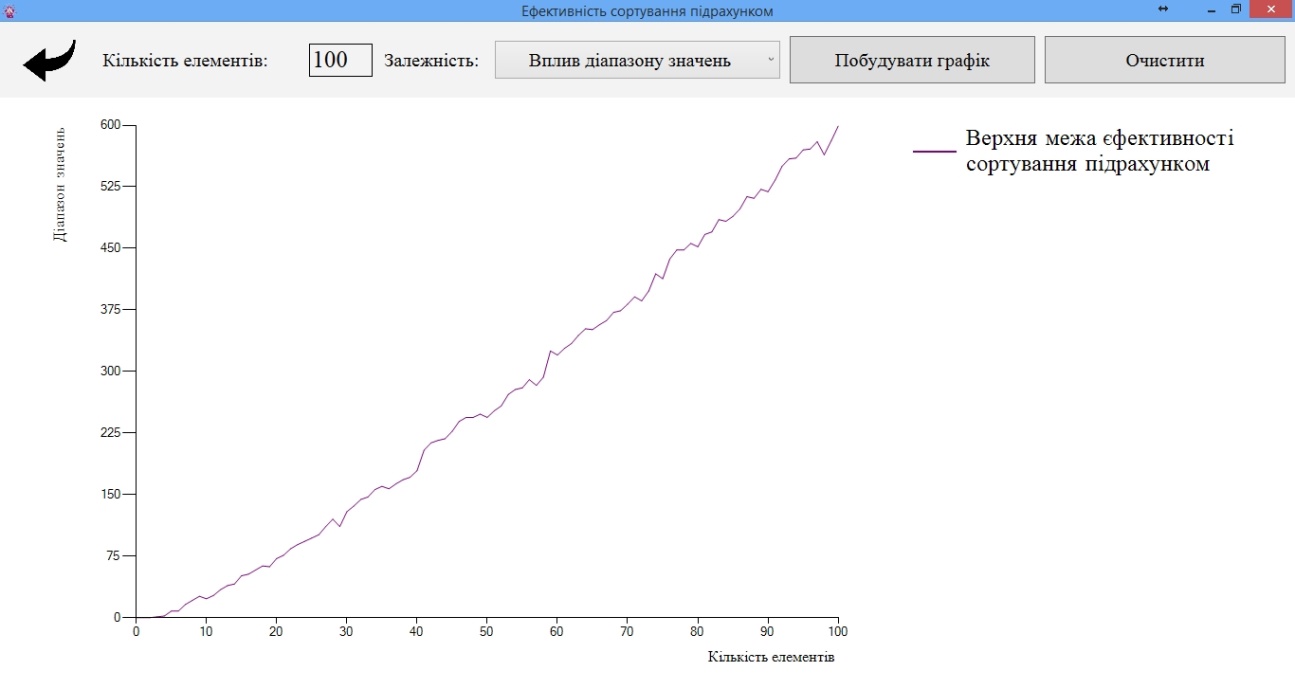


Рис. 2.10. Залежність ефективного діапазону значень від кількості елементів

При аналізі впливу кількості різних значень на ефективність сортування підрахунком, ми досліджуємо його видозмінену реалізацію, що є спробою позбавитись залежності від діапазону значень при використанні даного алгоритму, зберігши інші його переваги. Це необхідно, бо вірогідно, що дані будуть мати великий діапазон, проте кількість різних значень буде незначною відносно цього діапазону(наприклад при діапазоні 1000, буде лише 10 різних значень). При даній реалізації ми замість масиву C [0..k] створюємо об’єкт колекції Dictionary - D, у якому ключами будуть різні значення елементів масиву А, а значеннями буде кількість їх повторень у цьому масиві. Отже тепер ми послідовно читаємо елементи вхідного масиву A, і якщо A[i] зустрівся вперше, то ми додаємо новий запис у D: D.Add(A[i],1). Якщо ж елемент вже зустрічався, то ми знаходимо відповідний запис у D по ключу A[i] і збільшуємо значення у цього запису на 1. Але ми не можемо тепер просто пройти по D і записати в A число **i** D[i] разів, бо записи у D не впорядковані. Отже перед нами постає необхідність додаткового сортування колекції D за ключами, і тільки після цього ми зможемо послідовно перебрати значення D.Keys(вже впорядковані) і записати кожне з них до A D[i] разів.

Проаналізуємо ефективність вдосконаленого алгоритму: на етапі підрахунку, при заповненні Dictionary ми отримуємо n операцій. Оскільки додаткове сортування колекції D ми проводимо за допомогою QuickSort, ця дія дає операцій, де p – кількість різних значень у вхідних даних. При відновленні даних з D, виконується ще p+n операцій(n-кількість записів, які необхідно відновити, p- кількість елементів у D, які ми мусимо перебрати). Отже сумарно ми отримуємо складність .

Вдосконалений алгоритм сортування підрахунком буде залишатись ефективним доти, доки буде виконувати менше операцій за QuickSort, тобто буде виконуватись нерівність: , де p - кількість різних значень, а n – кількість елементів у вхідних даних. З рівняння бачимо, що коли кількість різних значень (p) близька до кількості елементів(n), такий алгоритм менш ефективний ніж QuickSort.

Розглянемо приклад. Нехай є послідовність елементів для сортування:

[1, 2, 3, 1, 2, 3, 1, 2, 3, 1] – p буде дорівнювати 3, а у Dictionary будуть зберігатись наступні пари: 1-4; 2-3; 3-3. Відсортувавши три ключа (1,2,3) за допомогою QuickSort, отримаємо приблизно 5 операцій(, а отже ). Додамо до цих операцій 2n та p, згідно до формули, і отримаємо кількість операцій, що виконає сортування підрахунком при такій реалізації – 28. При тій самій кількості елементів QuickSort виконує приблизно 33 операції порівняння(, а отже , що підтверджується експериментально). Як ми бачимо, рівняння

виконується(28<33), а отже сортування є ефективним. Проте збільшивши кількість різних значень до 5, наприклад сформувавши такі дані: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5] - p буде дорівнювати 5, а у Dictionary будуть зберігатись наступні пари: 1-2; 2-2; 3-2; 4-2; 5-2. Провівши необхідні розрахунки, ми отримаємо приблизно 37 операцій(, отже , а ), що більше за операції виконані QuickSort при тій самій кількості елементів(33<37), а отже при кількості різних значень рівній 5 і кількості елементів рівній 10, ця реалізація сортування підрахунком перестає бути ефективною.

Варто звернути увагу на наступну ситуацію: нехай у масиві усі елементи різні, тобто p=n, тоді впорядкованість елементів у початковому масиві може виявитись значно гіршою від впорядкованості елементів у Dictionary (таке трапляється через те, що алгоритм для формування даних з заданою кількістю значень формує масив таким чином, що у Dictionary дані повністю впорядковані). Тоді даний алгоритм буде ефективним за будь-якої кількості різних значень, але ймовірність формування даних з таким співвідношенням впорядкованості у реальних умовах мінімальна. Тому для гарантованого визначення кількості різних значень, за яких даний алгоритм не буде втрачати ефективності, порівняння виконується з найкращою ситуацією для QuickSort (дані повністю впорядковані, а отже кількість операцій мінімальна). Залежність ефективної кількості різних значень від кількості елементів у вхідних даних відображена на наступному графіку (рис. 2.11).

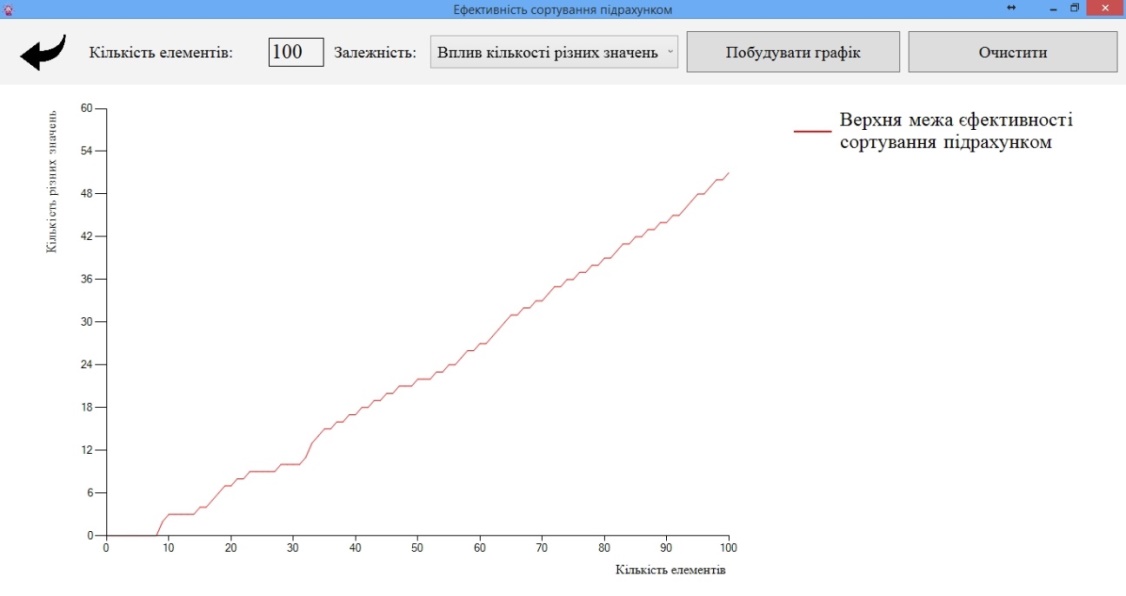


Рис. 2.11. Залежність ефективної кількості значень від кількості елементів

Впевнитись у справедливості вищенаведених фактів можна при детальному аналізі блок-схеми класичного алгоритму сортування підрахунком (див. додаток 2) а також розглянувши розділи програми «Теорія» та «Візуалізація».

* + 1. *Допоміжні розділи програми*

Окрім трьох основних розділі, що відповідають задачам висвітленим у роботі, у програмі присутні два допоміжних розділи: «Теорія» та «Візуалізація», які слугують для пояснення принципів роботи висвітлених алгоритмів. Розділ теорія, окрім опису принципу роботи чотирьох алгоритмів сортування, містить висновки до розглянутих у роботі задач. У розділі «Візуалізація» наочно демонструються принципи роботи таких алгоритмів сортування як: вставки, QuickSort та сортування підрахунком. Цей розділ містить кнопку «Легенда», що відкриває вікно пояснень до візуальних ефектів.

1. ***Блок-схема***
2. ***Напрямки, де можна використати роботу***

Метою створення цієї роботи, як вже зазначалось, є порівняння ефективності алгоритмів сортування. Отже, вона придатна для наочної демонстрації при вивченні теорії алгоритмів, зокрема алгоритмів сортування, і можливо буде цікава учням середнього шкільного віку. Також можливе використання даної роботи в якості початкових дослідницьких даних при поглибленому вивченні, аналізі та порівнянні алгоритмів сортування.

1. ***Програмні та апаратні вимоги***

Апаратні вимоги.

* Частота процесора 1.5GHz або вище
* 1024 MB RAM
* 5 MB вільного місця на жорсткому диску
* Роздільна здатність монітору 1024 x 768 або вище

Підтримувані операційні системи.

* Windows 10
* Windows 7, Windows 7 with SP1
* Windows 8, Windows 8 Pro, Windows 8.1

1. ***Використані програми:***

* Visual Studio 2015 Community
* Microsoft Word 2010/2016
* Google Chrome
* paint.net
* Adobe Acrobat Pro
* Notepad++

1. ***Комплектація програми***

Для встановлення програми необхідно скопіювати папку «Sorts» на жорсткий диск. ЇЇ зміст такий.

1. Sorts.exe – файл запуску програми для подальшої роботи
2. mas.txt – файл який містить єдиний рядок – масив, що буде використаний при помилці введення даних у програмі (елементи розташовані через пропуск, зайві рядки не призводять до помилки).
3. Папка Information з файлами: BinInsert.html, Conclusion.html, Counting.html, Insert.html, QSot.html та відповідні їм папки, що мають назви формату name.files і містять потрібні ресурси для файлів .html.

**Висновки**

В ході виконання роботи я зіткнувся з такою проблемою як перевірка достовірності експериментально отриманих даних. Під час її вирішення я навчився аналізувати отримані результати: виявляти в них закономірності та пояснювати їх походження.

В ході роботи було встановлено наступне:

* Підтвердження ефективності алгоритму сортування QuickSort у загальному випадку, та виявлення його не ефективності при майже впорядкованих даних.
* При генерації даних за допомогою об’єкту Random, графік залежності невпорядкованості даних від кількості елементів можна приблизно описати рівнянням : , де y – СНП, x- кількість елементів.
* Сортування підрахунком, порівняно з QuickSort, є ефективним якщо вхідні дані знаходяться у невеликому діапазоні відносно кількості сортованих елементів або якщо кількість різних значень у вхідних даних значно менша за кількість елементів.

Список використаних джерел

***Літературні джерела:***

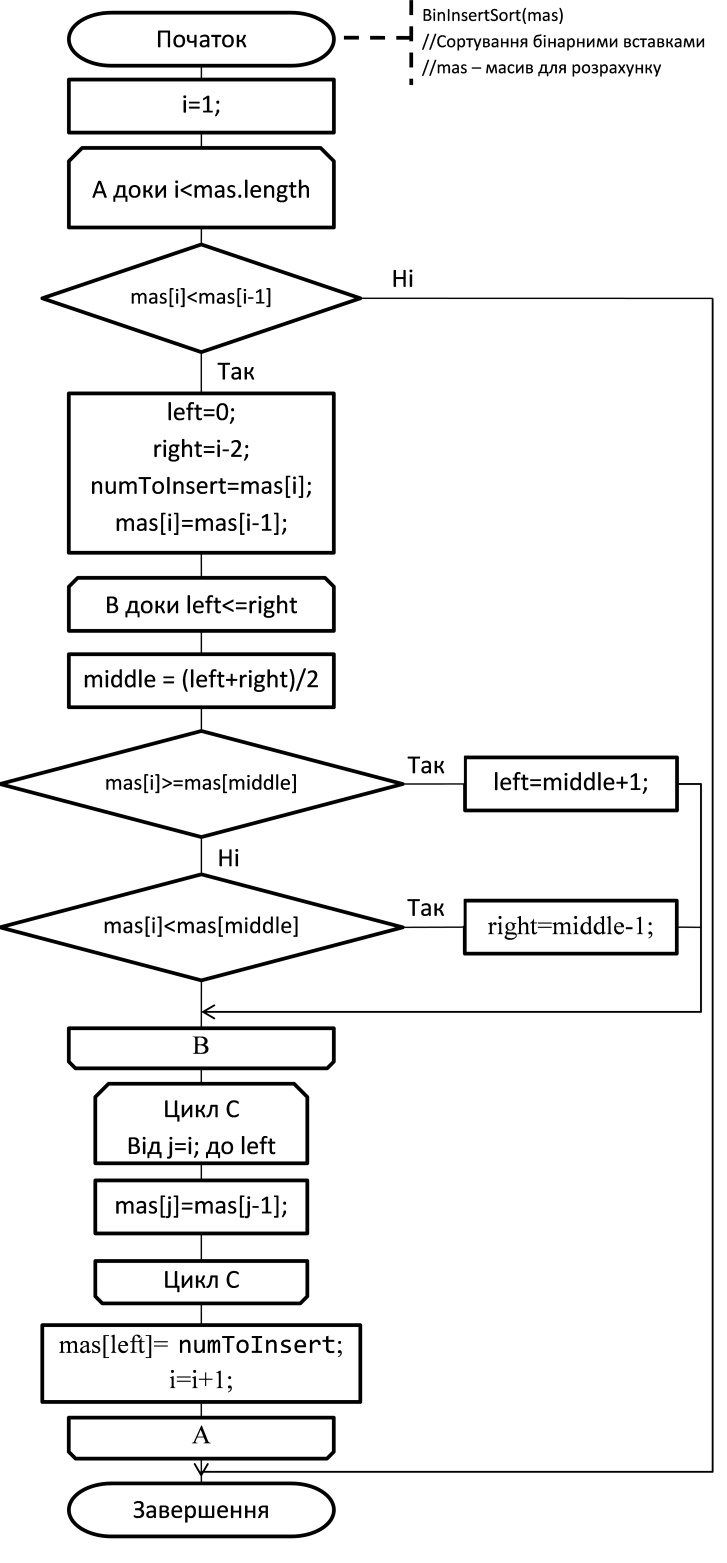
1. Т.П.Караванова «Інформатика. Методи побудови алгоритмів та їх аналіз» 2007р.

***Інтернет-джерела:***

1. Інтернет-енциклопедія «Вікіпедія» <https://ru.wikipedia.org/>, <https://uk.wikipedia.org/>.
2. «Хабрахабр» <https://habrahabr.ru/>.
3. MSDN <https://msdn.microsoft.com/>.
4. StackOverflow <http://ru.stackoverflow.com/>.
5. CodeProject <https://www.codeproject.com/>.
6. Professorweb <https://professorweb.ru>

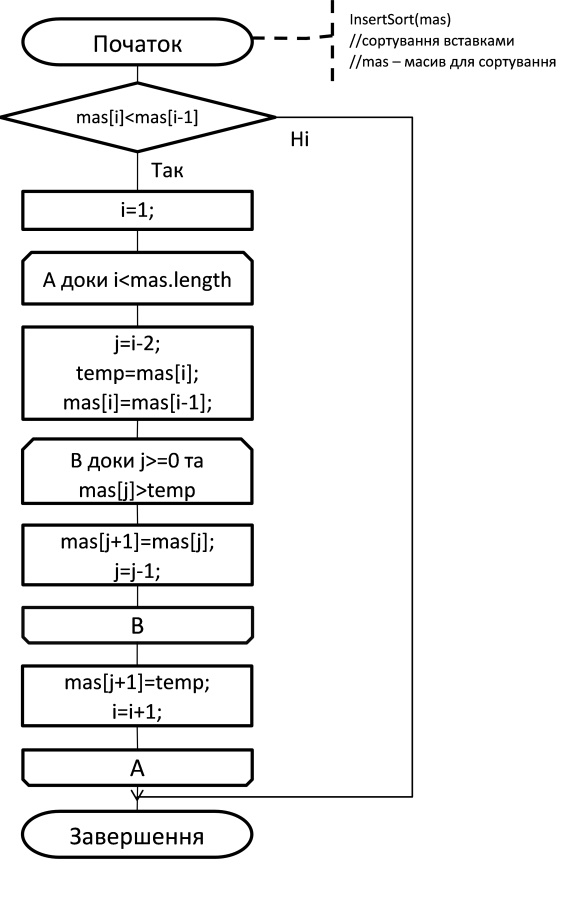
ДОДАТКИ

**Додаток 1**

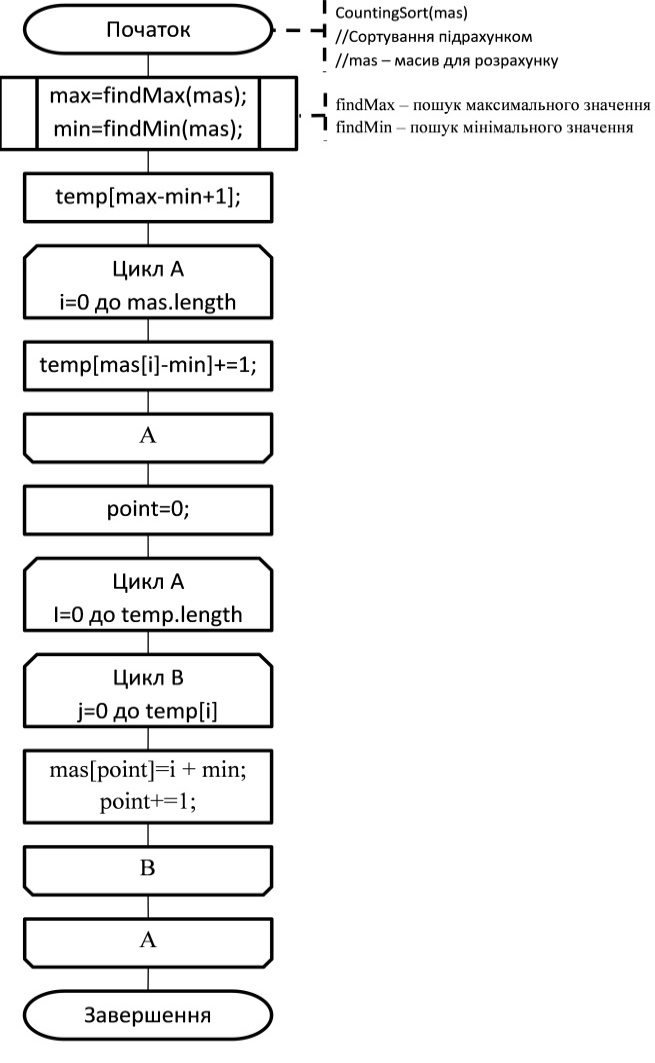
***Блок-схеми алгоритмів сортування:***

Блок-схема 2. Сортування

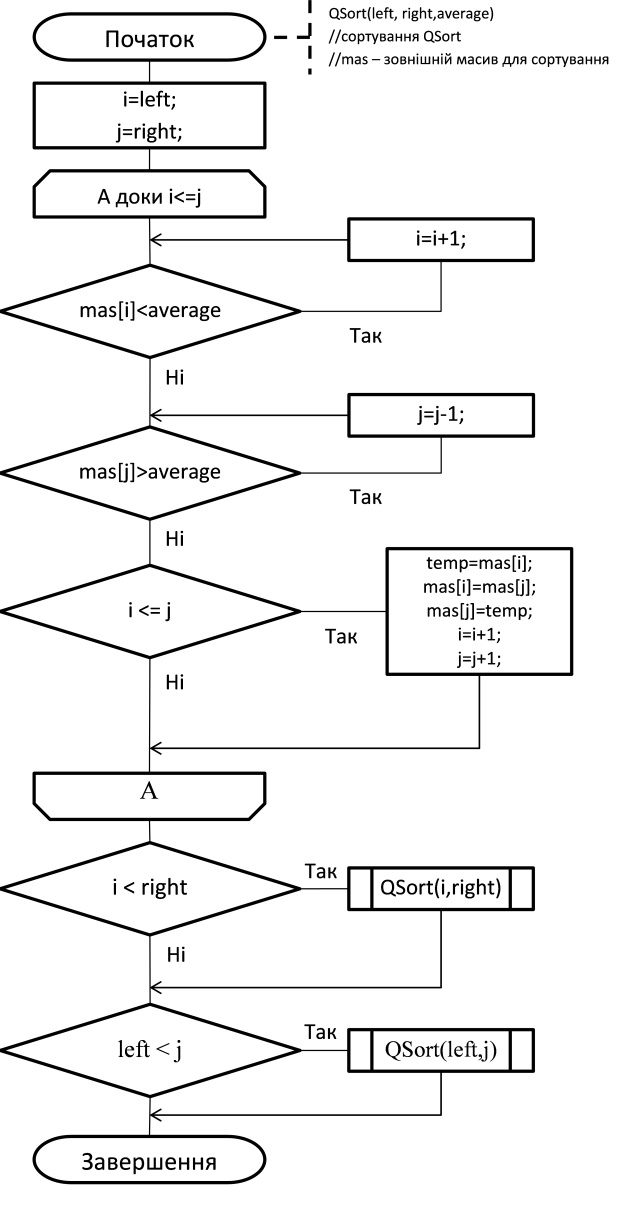
бінарними вставками



Блок-схема 1. Сортування вставками



Блок-схема 4. Сортування підрахунком (класичне)



Блок-схема 3. QuickSort

**Додаток 2**

*Код сортування підрахунком:*

//n- кількість елементів у масиві, k- діапазон значень

public int CountingSort(int[] mas)

{

int max = mas[0];

int min = mas[0];

int operation = 0;

for (int i = 1; i < mas.Length; i++)//шукаємо мінімальне та максимальне значення

{

if (mas[i] > max)

max = mas[i];

if (mas[i] < min)

min = mas[i];

}//цей пошук дає додаткові n операцій, проте ці дані можуть бути відомі, ці операції тож ці операції не є обов’язковими у роботі алгоритму, тому при загальному підрахунку операцій вони не враховуються

int[] temp = new int[max - min + 1];//створюємо службовий масив довжиною рівною діапазону значень

//При початковому масиві [1,1,3,3,5,5] службовий масив заповнюється так: [0,2,0,2,0,2]

for (int i = 0; i < mas.Length; i++)//підраховуємо кількість кожного з елементів в масиві. ЦЕ ДАЄ НАМ N ОПЕРАЦІЙ.

{

temp[mas[i] - min]++;//Збільшуємо відповідний елемент службового масиву

operation++;//враховуємо виконану операцію

}

int pos = 0;//маркує позицію, на яку потрібно ставити поточне число

for (int i = 0; i < temp.Length; i++)//цикл по службовому масиву. ЦЕ ДАЄ ЩЕ K ОПЕРАЦІЙ.

{

operation++;

for (int j = 0; j < temp[i]; j++)//вставляє до початкового масиву поточне число відповідну кількість разів. ЦЕ ДАЄ ЩЕ N ОПЕРАЦІЙ.

{

mas[pos] = i + min;//відновлюємо справжнє значення

pos++;//зсуваємо позицію для наступного вставлення на 1

operation++;

}

}

return operation;

}//Загалом маємо 2n+k операцій:n операцій при заповненні службового масиву temp, k операцій при проході по службовому масиві temp, та n операцій при поверненні даних у потрібному порядку до початкового масиву.