КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій

(кафедра)

КВАЛІФІКАЦІЙНА РОБОТА ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

на тему: «Програмний додаток для визначення ефективності алгоритмів сортування для різних наборів даних»

Мойсеєнко Аліна Олегівна

(прізвище, ім’я та по батькові студента повністю)

Київ – 2025р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій

(кафедра)

ЗАТВЕРДЖУЮ
Завідувачка кафедри ІТ
д.т.н., проф. Гончаренко Т.А.

„___” _____2025 року

КВАЛІФІКАЦІЙНА РОБОТА ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

на тему: «Програмний додаток для визначення ефективності алгоритмів сортування для різних наборів даних»

Я як здобувач вищої освіти КНУБА розумію і підтримую політику закладу з академічної доброчесності. Я не надавав(-ла) і не одержував(-ла) недозволену допомогу під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Здобувач Мойсеєнко Аліна Олегівна
(прізвище, ім’я та по батькові повністю)
122 «Комп’ютерні науки»
(спеціальність)
Інформаційні управляючі системи і технології
(освітня програма)
Групи КН-21
Керівник Терентьєв О.О.
(прізвище та ініціали)
професор, доктор технічних наук
(вчене звання, науковий ступінь)
Рецензент к.т.н., доц Баліна О.І.
(Прізвище та ініціали)

Ідентичність підтверджую

Київ, 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет: автоматизації і інформаційних технологій
Випускова кафедра: інформаційних технологій
Ступінь вищої освіти: «бакалавр»
Спеціальність: 122 «Комп`ютерні науки»
Освітня програма: Інформаційні управляючі системи і технології

ЗАТВЕРДЖУЮ
Завідувачка кафедри ІТ
д.т.н., проф. Гончаренко Т.А.
_____
„___” _____2025 року

З А В Д А Н Н Я
ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

Мойсеєнко Аліна Олегівна

1. Тема роботи: Програмний додаток для визначення ефективності алгоритмів сортування для різних наборів даних
затверджена наказом ректора КНУБА № 2650/2 від 18.11.2025.
2. Керівник роботи: Терентьєв Олександр Олександрович д.т.н., проф. кафедри інформаційних технологій проектування та прикладної математики
3. Строк подання студентом роботи до захисту:
4. Зміст пояснювальної записки за розділами:
Р.1. Аналіз предметної області
Р.2. Специфікація вимог до інформаційної системи
Р.3. Опис прийнятих проєктних рішень
Р.4. Дослідна експлуатація можливих застосувань
5. Інформаційні слайди:
С1.
С2.
С3.
С4.
6. Консультанти розділів кваліфікаційної випускної роботи

Розділ	Прізвище, ініціали та посада консультанта, представника комісії	дата	підпис
Ергономіка інформаційних технологій	доц. Рябчун Ю.В.		
Прийом програмного продукту	ас. Мацієвський О.О.		

7. Календарний план виконання кваліфікаційної випускної роботи

Види робіт та їх зміст	Дата виконання
Р. 1. Аналіз предметної області	Січень 2025 р.
Р. 2. Специфікація вимог до інформаційної системи	Лютий 2025 р.
Р. 3. Опис прийнятих проєктних рішень	Травень 2025 р.
Р. 4. Дослідна експлуатація можливих застосувань	Травень 2025 р.
Остаточне оформлення роботи	Травень 2025 р.
Направлення роботи на рецензування	Червень 2025 р.
Попередній захист роботи на кафедрі	Червень 2025 р.

8. Дата видачі завдання: 20.01.2025 р.

Завідувачка _____ Гончаренко Т.А.
(підпис) (прізвище та ініціали)

Керівник _____ Терентьєв О.О.
(підпис) (прізвище та ініціали)

Студент _____ Мойсеєнко А.О.
(підпис) (прізвище та ініціали)

Зміст

ВСТУП 7
Розділ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 10
1.1. Загальна класифікація структур даних 10
1.2. Загальна характеристика алгоритмів сортування 12
1.3 Алгоритми внутрішнього сортування 15
1.5 Огляд і аналіз існуючих програмних продуктів для оцінки ефективності алгоритмів 24
1.6 Постановка задачі 27
Висновки до розділу 1 29
Розділ 2. СПЕЦИФІКАЦІЯ ВИМОГ ДО ІНФОРМАЦІЙНОЇ СИСТЕМИ 30
2.1 Глосарій 30
2.3 Специфікація функціональних та нефункціональних вимог 36
2.4 Технічне завдання 39
Висновки до розділу 2 44
Розділ 3. ОПИС ПРИЙНЯТИХ ПРОЄКТНИХ РІШЕНЬ 45
3.1 Опис вихідних і вхідних даних 45
3.1.1. Вхідні дані 45
3.1.2. Вихідні дані 46
3.4 Засоби розробки 59
3.5 Проєктування інтерфейсу програмної системи 60
3.6 Опис програмної реалізації 64
Висновки до розділу 3 76
Розділ 4. ДОСЛІДНА ЕКСПЛУАТАЦІЯ МОЖЛИВИХ ЗАСТОСУВАНЬ 78
4.1 Опис роботи програмного продукту 78
4.2 Методи тестування ефективності алгоритмів 81
4.2.1. Сценарії тестування 81
4.2.2. Аналіз результатів тестування 88
Висновки до розділу 4 89
ЗАГАЛЬНІ ВИСНОВКИ 90
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 91

ВСТУП

У цифрову епоху, коли інформація панує над усіма, а дані генеруються з безпрецедентною швидкістю, здатність обробка та управління цим потоком даних стала першорядною. Уявіть собі бібліотекаря, який постійно розширює колекцію книжок, кожна з яких займає своє унікальне місце на полиці. Завдання бібліотекаря не просто зберігати книжки, а також переконатися, що вони розташовані таким чином, щоб полегшити пошук. Так само і в області інформатики, алгоритми сортування відіграють роль цих старанних бібліотекарів, ретельно організовуючи дані для ефективного доступу та пошуку. Алгоритми сортування лежать в основі незліченних програм, від баз даних до пошуку двигунів, що забезпечує безперебійну роботу користувача та оптимізовану продуктивність [1].

Значення алгоритмів сортування стає очевидним у сценаріях, коли потрібні великі обсяги даних повинні швидко та точно оброблятися. Від організації списку імен за алфавітом до сортування база даних транзакцій клієнтів за датою, ці алгоритми є фундаментальними для безперебійної роботи різних технологічних систем. Алгоритми сортування є наріжним каменем інформатики, і розуміння їх тонкощів має вирішальне значення для програмістів і розробників. Вони служать шлюзом для більш складних алгоритмів і даних структур, що забезпечує фундаментальне розуміння алгоритмічного проектування та аналізу. Крім того, оволодіння алгоритмами сортування покращує навички критичного мислення.

Світ алгоритмів сортування надзвичайно різноманітний, кожен алгоритм використовує унікальний набір правила і методи сортування даних. Деякі алгоритми прості та інтуїтивно зрозумілі, що робить їх ідеальними для маленьких набори даних, тоді як інші є складними та витонченими, призначеними для обробки величезних масивів інформації.

В інформатиці алгоритм сортування - це алгоритм, який розміщує елементи списку в певному порядку. Найбільш вживаними є числовий порядок

і лексикографічний порядок. Ефективне сортування важливе для оптимізації використання інших алгоритмів (таких як алгоритми пошуку та злиття), які потребують відсортованих списків для правильної роботи; це також часто корисне для канонізації даних і для отримання зрозумілих для людини результатів. Більш формально, результат повинен задовольняти два умови:

1. Вихід у порядку неспадання (кожен елемент не менший за попередній відповідно до бажаний загальний порядок);

2. Вихід є перестановкою або перевпорядкуванням вхідних даних.

З самого початку обчислювальної техніки проблема сортування привернула багато досліджень, можливо, завдяки складності її ефективного вирішення, незважаючи на її просте, знайоме твердження. Хоча багато хто вважає цю проблему вирішеною, корисні нові алгоритми сортування все ще винаходяться (наприклад, бібліотечне сортування було вперше опубліковано в 2004 році).

Вибір алгоритму залежить від структури оброблюваних даних - це майже закон, але у разі сортування така залежність настільки глибока, що відповідні методи були навіть розбиті на два класи – сортування масивів та сортування файлів (також називається сортуванням послідовностей). Їх ще називають внутрішнім та зовнішнім сортуванням, тому що масиви зберігаються в оперативній, внутрішній пам'яті машини з прямим доступом, а файли розміщуються в повільнішій, але і більш ємній зовнішній пам'яті на пристроях, заснованих на механічних переміщеннях. Крім такої класифікації, алгоритми сортування поділяються на сортування на місці та сортування з перезаписом (всі елементи переписуються в іншу область пам'яті). Третя класифікація ділить алгоритми на сортування, засновану на порівняннях ключів, та сортування без порівняння ключів.

Актуальність розробки програмного додатку для визначення ефективності алгоритмів сортування для різних наборів даних обумовлена кількома факторами. Існує безліч алгоритмів сортування, кожен зі своїми перевагами та недоліками. Вибір оптимального алгоритму залежить від

багатьох факторів, таких як розмір набору даних, тип даних, вимоги до швидкодії та пам'яті. Дані, які потрібно сортувати, можуть бути різними за розміром, типом, розподілом та іншими характеристиками. Ефективність алгоритму сортування може суттєво відрізнятися залежно від цих характеристик. У багатьох задачах сортування є критично важливою операцією, тому важливо вибрати алгоритм, який забезпечить найкращу продуктивність для конкретного набору даних. Не існує універсального алгоритму сортування, який був би найкращим для всіх випадків. Тому розробка інструменту, який дозволяє порівнювати ефективність різних алгоритмів на різних наборах даних, є дуже актуальною. Результати дослідження ефективності алгоритмів сортування можуть бути використані для оптимізації роботи різних програмних систем, таких як бази даних, пошукові системи, системи обробки даних тощо.

Об’єкт розробки - програмний додаток для визначення ефективності алгоритмів сортування для різних наборів.

Предмет розробки - алгоритми сортування.

Метою даною роботи є розробка програмного додатку для визначення ефективності алгоритмів сортування для різних наборів даних

Для досягнення зазначеної мети, поставлені такі завдання:

а) аналіз предметної області;

б) здійснити опис основних алгоритмів сортування з врахуванням особливостей використання різних наборів даних;

в) сформулювати основні функціональні та нефункціональні вимоги для створення програми;

г) здійснити проектування програмного додатку для визначення ефективності алгоритмів сортування для різних наборів даних.

В якості мови програмування обирається C# й фреймворк Windows Forms.

Розділ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Загальна класифікація структур даних

Структура даних - це спеціалізований формат для організації, обробки, пошуку та зберігання даних. Існує кілька базових і розширених типів структур даних, усі вони призначені для впорядкування даних відповідно до потреб конкретної мети. Структури даних полегшують користувачам доступ до даних і роботу з ними потрібно відповідними способами. Найважливіше те, що структури даних обрамляють організацію інформації, щоб машини та люди могли її краще зрозуміти [2].

Структури даних об’єднують елементи даних у логічний спосіб і полегшують ефективне використання, збереження та обмін даними. Вони забезпечують формальну модель, яка описує спосіб організації елементів даних. Структури даних є будівельними блоками для більш складних програм. Вони розроблені для компонування елементів даних у логічний блок, що представляє абстрактний тип даних, який має релевантність до алгоритму або програми. Прикладом абстрактного типу даних є «ім’я клієнта». складається з рядків символів для «ім’я», «по батькові» та «прізвища». Існує наступна класифікація структур даних.

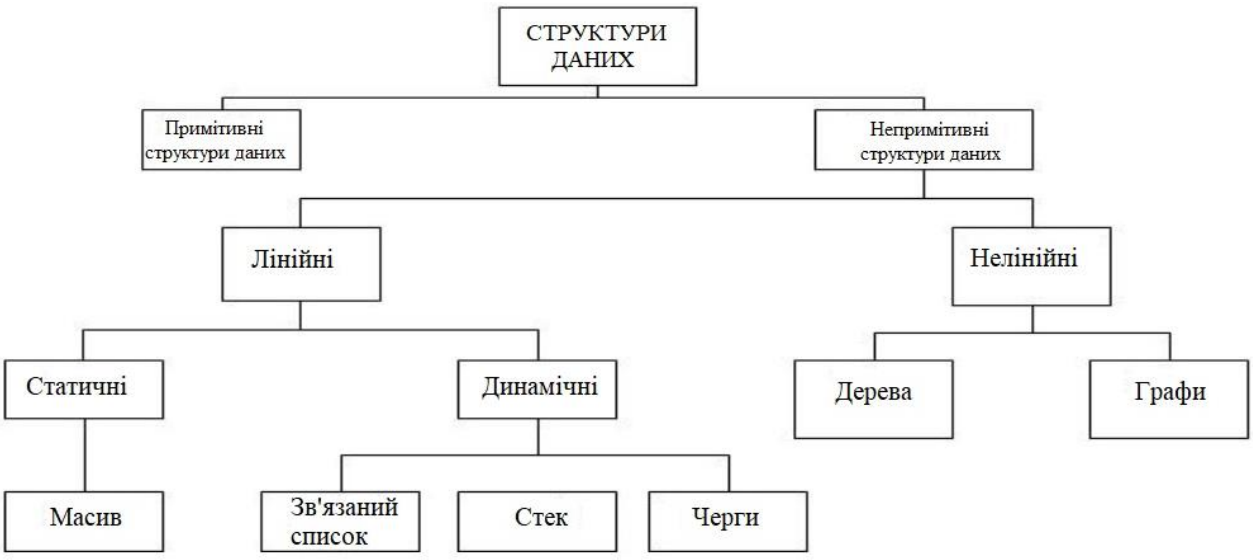

Рис. 1.1. Класифікація структури даних [2].

Далі описуються характеристики понять, наведених на схемі рис. 1.1.

Примітивні структури даних. Це найпростіші будівельні блоки для зберігання даних. Вони є базовими типами даних, які підтримуються мовою програмування.

Приклади:

Цілі числа (Integer). Використовуються для представлення цілих чисел (наприклад, -10, 0, 5).

Дійсні числа (Float). Використовуються для представлення чисел з плаваючою крапкою (наприклад, 3.14, -2.5).

Символи (Character). Використовуються для представлення окремих символів (наприклад, 'a', 'Z', '$').

Логічні значення (Boolean). Використовуються для представлення логічних значень true або false.

Непримітивні структури даних. Ці структури будуються на основі примітивних типів даних та використовуються для зберігання колекцій даних. Вони можуть бути лінійними або нелінійними.

Лінійні структури даних. Елементи в цих структурах розташовані в лінійній послідовності, один за одним.

Статичні. Розмір структури фіксований під час її створення і не може бути змінений.

Масив (Array). Колекція елементів одного типу, доступ до яких здійснюється за індексом [2].

Динамічні. Розмір структури може змінюватися під час виконання програми.

Зв'язаний список (Linked List). Колекція елементів (вузлів), кожен з яких містить дані та покажчик на наступний вузол.

Стек (Stack). LIFO (Last-In, First-Out) структура, де додавання та видалення елементів відбувається з одного кінця (вершини).

Черга (Queue). FIFO (First-In, First-Out) структура, де додавання елементів відбувається з одного кінця (хвоста), а видалення - з іншого (голови).

Нелінійні структури даних. Елементи в цих структурах не розташовані в лінійній послідовності.

Дерева (Trees). Ієрархічні структури, що складаються з вузлів (з даними) та ребер (зв'язків між вузлами). Кожен вузол має батьківський вузол (крім кореня) та може мати дочірні вузли.

Графи (Graphs). Колекція вузлів (вершин) та ребер, що з'єднують ці вузли. Графи можуть бути орієнтованими (ребра мають напрямок) та неорієнтованими.

1.2. Загальна характеристика алгоритмів сортування

Слово «сортування» походить від англійської "sorting", що, за сенсом, означає відбір за сортами. Програмісти використовують цей термін більш вузько, як процес вибудовування елементів у певному порядку, наприклад, за зростанням будь-якого значення елементів розглянутого списку. Як зазначено в [2], цей процес правильніше було б назвати не сортуванням, а впорядкуванням (англійською це було б ordering), але використання даного слова призвело б до плутанини через перевантаженості значеннями слова «порядок».

Під терміном сортування розуміється процедура перестановки елементів множини у певному порядку, тобто. якщо дані елементи

$$a_1 \ldots . a_n,$$

то сортування означає перестановку цих елементів у такому порядку

$$a_{k_1}, a_{k_2}, \ldots, a_{k_n}$$

при якому для заданої функції упорядкування f справедливо співвідношення:

$$f(a_{k_1}) \leq f(a_{k_2}) \leq \cdots \leq f(a_{k_1})$$

Наприклад, якщо розглянути список елементів 12, 4, 8, 3, 11, 5, 7, 4 та функцію впорядкування a<b, результатом сортування буде масив: 3, 4, 4, 5, 7, 8, 11, 12.

Хоча розроблено багато алгоритмів сортування, не існує єдиного алгоритму, який був би оптимальним у всіх випадках. Ефективність алгоритму сортування залежить від ряду факторів, серед яких:

- кількість відсортованих елементів;

- діапазон і розподіл значень відсортованих елементів;

- початковий ступінь сортування елементів;

- характеристики алгоритму (складність, вимоги до пам'яті тощо);

- місце розташування елементів (оперативна пам'ять (масив) або диск (файл)).

Методи сортування класифікуються на внутрішні та зовнішні. При внутрішньому сортуванню дані розміщуються в оперативній пам'яті, наприклад, у масиві. При зовнішньому сортуванні дані знаходяться у зовнішній пам'яті. До зовнішнього сортування вдаються у випадках, коли неможливо розмістити в оперативній пам'яті всі дані.

Елементи, що сортуються, часто є записами даних з певною структурою. Кожен запис має ключове поле, значення якого використовується для виконання сортування. При розгляді алгоритмів сортування дослідника цікавить тільки ключове поле, тому інші поля відсортованої структури не розглядаються, як у випадку зі списками і деревами.

Метод сортування називають стійким, якщо відносне положення елементів з однаковим (рівним) ключем не змінюється в процесі сортування. Стабільність сортування бажана при роботі з елементами, вже відсортованими за іншими критеріями (властивостями), які не впливають на ключ, за яким виконується сортування.

Основними вимогами до алгоритмів сортування, як і до інших алгоритмів, є вимоги до пам'яті та часу виконання. Це означає, що внутрішнє сортування (елементів масиву) виконується локально, без передачі в

результуючий масив. Доброю мірою часової ефективності алгоритму сортування є кількість необхідних порівнянь ключів C і кількість пересилань M, , які мають залежність від числа сортованих елементів n.

Записи даних можна сортувати на основі властивостей. Такі записи і поля властивостей називаються ключами сортування. Під час запуску реальних додатків часто виникає потреба сортувати масив записів за кількома ключами. Зазвичай це відбувається тоді, коли один ключ не може однозначно ідентифікувати запис. Наприклад, у великій організації може знадобитися відсортувати список співробітників за їхніми відділами, а потім відсортувати імена в алфавітному порядку в межах кожного відділу. Інші приклади сортування за кількома ключами:

- У телефонному довіднику імена сортуються за місцем розташування, категорією (ділові або цивільні), а потім за алфавітом.

- У бібліотеках інформація про книги сортується в алфавітному порядку за назвою, а потім за автором.

- Адреси клієнтів сортуються за назвою району, потім за вулицею.

Таким чином, повне сортування та його ключ можна визначити за допомогою двох або більше ключів часткового сортування. У цьому випадку перший ключ називається первинним ключем сортування, а другий і наступні ключі - вторинними ключами сортування.

Дуже важливим для алгоритмів сортування є критерій обчислювальної складності. Обчислювальна складність (у найгіршому, середньому та найкращому випадках) порівняння елементів залежно від розміру списку n. Для типових алгоритмів сортування хороша поведінка - це $O(nlogn)$, а погана - $O(n^2)$. Ідеальна поведінка для сортування - це $O(n)$, але це неможливо в середньому випадку. Алгоритми сортування на основі порівнянь, які оцінюють елементи списку за допомогою абстрактної операції порівняння ключів, вимагають щонайменше $O(nlogn)$ порівнянь для більшості вхідних даних.

Далі в роботі розглядаються виключно алгоритми внутрішнього сортування.

1.3 Алгоритми внутрішнього сортування

Розглянемо складність різних алгоритмів у конкретній ситуації. У таблиці 1.1. n - це кількість записів, які потрібно відсортувати. Стовпці "Середній" та "Найгірший" показують часову складність у кожному випадку, за умови, що довжина кожного ключа є сталою, а отже, всі порівняння, обміни та інші необхідні операції можуть виконуватись за сталий час. Пам'ять вказує на обсяг додаткової пам'яті, необхідної понад ту, яку використовує сам список, за тієї ж умови. Усі ці алгоритми є сортуваннями на основі порівнянь. Назви алгоритмів в табл. 1.1 наводяться англійською для уніфікації.

Таблиця 1.1. Порівняння алгоритмів сортування

Назва	Найкращий випадок	Середній випадок	Найгірший випадок	Пам'ять	Стабільність	Метод	Інші примітки
Spaghetti sort	n	n	n	n	Так	Вибірка	Лінійний за часом аналоговий алгоритм для сортування послідовності елементів, що потребує O(n) місця в стеку, сортування стабільне. Потребує паралельного процесора.
Quicksort	$n \log n$	$n \log n$я	n^2	$\log n$	Залежить	Розбиття	Quicksort можна виконати на місці з O(log(n)) місця в стеку, але сортування нестабільне. Наївні варіанти використовують масив розміром O(n) для зберігання розділу. Реалізація з O(n) місця може бути стабільною.
Merge sort	$n \log n$	$n \log n$	$n \log n$	Залежить	Так	Злиття	Використовується для сортування цієї таблиці у Firefox.
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	Ні	Вибір	
Insertion sort	n^2	n^2	n^2	1	Так	Вставка	Середній випадок також O(n+d), де d - кількість інверсій.
Introsort	—	$n \log n$	$n \log n$	$\log n$	Ні	Розбиття та Вибір	Використовується в реалізаціях SGI STL.
Selection sort	n^2	n^2	n^2	1	Ні	Вибір	Його стабільність залежить від реалізації. Використовується для сортування цієї таблиці в Safari або інших веб-браузерах Webkit.
Timsort	n	$n \log n$	$n \log n$	n	Так	Вставка та Злиття	O(n) порівнянь, коли дані вже відсортовані або відсортовані у зворотному порядку.

Продовження таблиця 1.1.

Назва	Найкращий випадок	Середній випадок	Найгірший випадок	Пам'ять	Стабільність	Метод	Примітки
Shell sort	n	$n(\log n)^2$	$O(n(\log n)^2)$	1	Ні	Вставка	Залежить від послідовності інтервалів. Найвідоміша:
Bubble sort	n	n^2	n^2	1	Так	Обмін	Дуже малий розмір коду.
Binary tree sort	n	$n \log n$	$n \log n$	n	Так	Вставка	При використанні самозбалансованого двійкового дерева пошуку.
Cycle sort	-	n^2	n^2	1	Ні	Вставка	На місці з теоретично оптимальною кількістю записів.
Library sort	-	-	$n \log n$	n	Так	Вставка	
Patience sorting	-	-	$n \log n$	1	Ні	Вставка та Вибір	Знаходить всі найдовші зростаючі підпослідовності за O(n log n).
Smoothsort	n	$n \log n$	$n \log n$	1	Ні	Вибір	Адаптивне сортування - O(n) порівнянь, коли дані вже відсортовані, і 0 обмінів.
Strand sort	n	n^2	n^2	n	Так	Вибір	
Tournament sort	-	$n \log n$	$n \log n$			Вибір	
Cocktail sort	n	n^2	n^2	1	Так	Обмін	
Comb sort	-	-	n^2	1	Ні	Обмін	Малий розмір коду.
Gnome sort	n	n^2	n^2	1	Так	Обмін	Дуже малий розмір коду.
In-place merge sort	-	-	$n(\log n)^2$	1	Так	Злиття	Реалізовано в Standard Template Library (STL); може бути реалізовано як стабільне сортування на основі стабільного злиття на місці.
Bogosort	n	$n!$	$n! - Infinity$	1	Ні	Випадкове перемішування	Випадково переставляє масив і перевіряє, чи він відсортований.

У наведеній нижче таблиці описано цілочисельні алгоритми сортування та інші алгоритми сортування, які не є сортуваннями порівняння. Як такі, вони не обмежені нижньою межею. Нижче наведено складності в термінах n, кількості елементи, які потрібно відсортувати, k - розмір кожного ключа, d - розмір цифри, що використовується реалізацією. Багато з них базуються за припущенням, що розмір ключа достатньо великий, щоб усі записи мали унікальні значення ключа, а отже, що $n << 2^k$, де << означає "набагато менше, ніж."

Таблиця 1.2. Алгоритми сортування без порівняння

Назва	Найкращий випадок	Середній випадок	Найгірший випадок	Пам'ять	Стабільність	$n \ll 2^k$	Примітки
Pigeonhole sort	-	$n + 2^k$	$n + 2^k$	2^k	Так	Так	
Bucket sort(uniform keys)	-	$n + k$	$n^2 \cdot k$	$n \cdot k$	Так	Ні	Припускає рівномірний розподіл елементів з домену в масиві.
Bucket sort (integerkeys)	-	$n + r$	$n + r$	$n + r$	Так	Так	r - діапазон чисел, які потрібно відсортувати. Якщо r = тоді Avg RT
Counting sort	-	$n + r$	$n + r$	$n + r$	Так	Так	r - діапазон чисел, які потрібно відсортувати. Якщо r = тоді Avg RT
LSD Radix Sort	-	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	n	Так	Ні	
MSD Radix Sort	-	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + \frac{k}{d} \cdot 2^d$	Так	Ні	Стабільна версія використовує зовнішній масив розміром n для зберігання всіх бункерів
Spreadsort	-	$n \cdot \frac{k}{d}$	$n \cdot \left(\frac{k}{d} + d\right)$	$n + \frac{k}{d} \cdot 2^d$	Ні	Ні	Асимптотика базується на припущенні, що $n << 2^k$, але алгоритм цього не вимагає

Далі наводиться огляд популярних алгоритмів сортування.

Bubble sort (бульбашкове сортування). Бульбашкове сортування - це алгоритм сортування, який безперервно проходить по списку, обмінюючи місцями елементи, доки вони не стануть у правильному порядку.

Бульбашкове сортування - простий алгоритм сортування. Алгоритм починається з початку набору даних. Він порівнює перші два елементи, і якщо перший більший за другий, то вони міняються місцями. Це продовжується для кожної пари сусідніх елементів до кінця набору даних. Потім алгоритм знову починає з перших двох елементів, повторюючи процес, доки під час проходу не відбудеться жодного обміну. Середня та найгірша продуктивність цього алгоритму – $O(n^2)$, тому його рідко використовують для сортування великих, невпорядкованих наборів даних.

Бульбашкове сортування може бути використане для сортування невеликої кількості елементів (де його низька ефективність не є критичною). Воно також може бути ефективним на майже впорядкованих списках. Наприклад, якщо лише один елемент не на своєму місці, бульбашкове сортування займе лише 2n часу. Якщо два елементи не на місці, воно потребуватиме щонайбільше 3n часу.

Середній та найгірший випадки для бульбашкового сортування – $O(n^2)$.

Selection sort (сортування вибором). Сортування вибором - це алгоритм сортування на місці (in-place comparison sort). Він має складність $O(n^2)$, що робить його неефективним для великих списків, і зазвичай він працює гірше, ніж подібне сортування вставками. Однак сортування вибором вирізняється своєю простотою і може мати переваги над складнішими алгоритмами в певних ситуаціях.

Алгоритм знаходить мінімальне значення, змінює його місцями зі значенням у першій позиції та повторює ці кроки для решти списку. Він виконує не більше n обмінів, тому може бути корисним там, де обмін елементів є дуже кошторисним.

Insertion sort (сортування вставками). Сортування вставками – це простий алгоритм сортування, який є відносно ефективним для невеликих списків і майже впорядкованих списків. Часто використовується як частина складніших алгоритмів.

Алгоритм працює, беручи елементи зі списку по одному та вставляючи їх на правильну позицію в новому відсортованому списку. У масивах новий відсортований список і залишкові елементи можуть розділяти ту саму область пам’яті, але вставка є дорогою операцією, оскільки вимагає зміщення всіх наступних елементів на одну позицію.

Shell sort (сортування Шелла). Сортування Шелла відрізняється від бульбашкового сортування тим, що воно переміщує елементи на кілька позицій за один обмін.

Сортування Шелла було винайдено Дональдом Шеллом у 1959 році. Воно покращує бульбашкове сортування та сортування вставками, переміщуючи невпорядковані елементи більш ніж на одну позицію за раз. Одна з реалізацій може бути описана як розташування послідовності даних у двовимірному масиві з подальшим сортуванням стовпців цього масиву за допомогою сортування вставками.

Comb sort (сортування гребінцем). Сортування гребінцем – це відносно простий алгоритм сортування, який спочатку розробив Владзімєж Добосевич у 1980 році. Пізніше він був переосмислений і популяризований Стівеном Лейсі та Річардом Боксом у статті для журналу Byte, опублікованій у квітні 1991 року.

Сортування гребінцем є покращенням бульбашкового сортування та може конкурувати з алгоритмами на кшталт Quicksort. Основна ідея полягає в усуненні «черепах» (малих значень біля кінця списку), оскільки в бульбашковому сортуванні вони значно уповільнюють процес сортування. («Кролики», тобто великі значення на початку списку, не створюють проблем у бульбашковому сортуванні.)

Merge sort (злиття). Сортування злиттям використовує простоту об’єднання вже відсортованих списків у новий відсортований список. Алгоритм починає з порівняння кожної пари елементів (тобто 1 з 2, потім 3 з 4 і так далі), змінюючи їх місцями, якщо перший елемент повинен йти після другого. Потім він об’єднує кожен із отриманих списків по два елементи в списки по чотири елементи, після цього ці списки по чотири об’єднуються в більші, і так далі, доки в кінці не будуть об’єднані два останні списки у фінальний відсортований список.

Сортування злиттям добре масштабується для дуже великих списків, оскільки в найгіршому випадку його час виконання складає O(n log n). Цей алгоритм набув популярності для практичних реалізацій і використовується як стандартна процедура сортування в мовах програмування Perl, Python (як timsort), Java (також використовує timsort з JDK7) та інших.

Heapsort (сортування купою) Heapsort – це значно ефективніша версія сортування вибором. Воно також працює шляхом визначення найбільшого (або найменшого) елемента у списку, розміщуючи його в кінці (або на початку) списку, а потім продовжує роботу з рештою списку. Це завдання виконується ефективно за допомогою структури даних, яка називається купа (heap) – спеціального типу бінарного дерева.

Після перетворення списку даних на купу, кореневий вузол стає найбільшим (або найменшим) елементом. Коли він видаляється та розміщується в кінці списку, купа перебудовується так, щоб найбільший елемент із тих, що залишилися, перемістився в корінь.

Завдяки купі знаходження наступного найбільшого елемента займає O(log n) часу, на відміну від O(n) для лінійного пошуку, як у простому сортуванні вибором. Це дозволяє Heapsort працювати за час O(n log n), що також є його найгіршим випадком.

Quicksort (швидке сортування). Quicksort - це алгоритм «розділяй і володарюй», який базується на операції розділення (partitioning). Для розділення масиву обирається елемент, який називається опорним елементом (pivot). Усі елементи, менші за опорний, переміщуються перед ним, а всі більші – після нього. Це можна зробити ефективно за лінійний час і на місці (in-place). Потім підсписки з меншими та більшими значеннями рекурсивно сортуються.

Ефективні реалізації Quicksort (з розділенням на місці) зазвичай є нестабільними та дещо складними, але на практиці вони належать до найшвидших алгоритмів сортування. Завдяки невеликому використанню пам’яті O(log n) Quicksort є одним із найпопулярніших алгоритмів сортування та доступний у багатьох стандартних бібліотеках програмування.

Найскладнішим аспектом у Quicksort є вибір гарного опорного елемента. Послідовно невдалі вибори опорних елементів можуть значно сповільнити роботу алгоритму до $O(n^2)$. Якщо кожного разу обирати медіану як опорний елемент, то алгоритм працюватиме за O(nlogn). Однак пошук медіани у невідсортованому списку є операцією O(n), що накладає власні витрати на сортування.

Counting Sort (підрахункове сортування). Counting sort застосовується, коли відомо, що кожен елемент вхідних даних належить до певного множини S можливих значень. Алгоритм працює за час O(|S| + n) та використовує пам'ять O(|S|), де n – це довжина вхідного масиву.

Він працює шляхом створення цілочисельного масиву розміру |S| і використання і-го елемента цього масиву для підрахунку кількості появ і-го елемента множини S у вхідних даних. Кожен вхідний елемент підраховується шляхом збільшення значення у відповідному елементі масиву.

Після цього масив підрахунків проходиться по черзі, і всі вхідні елементи розміщуються у впорядкованому вигляді.

Цей алгоритм не завжди можна використовувати, оскільки множина S має бути досить невеликою, щоб алгоритм був ефективним. Однак Counting Sort надзвичайно швидкий і демонструє чудову асимптотичну поведінку зі збільшенням n. Також його можна модифікувати для забезпечення стабільної поведінки.

Bucket Sort (сортування за кошиками). Bucket sort – це алгоритм сортування за принципом «розділяй і володарюй», який узагальнює Counting Sort шляхом розбиття масиву на скінченну кількість кошиків (buckets).

Кожен кошик потім сортується окремо або за допомогою іншого алгоритму сортування, або шляхом рекурсивного застосування алгоритму Bucket Sort.

Існує варіація цього методу під назвою однокомпонентне підрахункове сортування (single buffered count sort), яка швидша за Quicksort і вимагає приблизно однакового часу для виконання на будь-якому наборі даних.

Оскільки Bucket Sort вимагає використання обмеженої кількості кошиків, він найкраще підходить для сортування наборів даних з обмеженою варіативністю значень. Bucket Sort не підходить для даних з великим діапазоном значень, наприклад, для номерів соціального страхування, оскільки вони мають значну варіацію.

Radix Sort (поцифрове сортування). Radix Sort – це алгоритм, який сортує числа, обробляючи окремі цифри чисел. n чисел, які складаються з k цифр кожне, сортуються за час $O(n \cdot k)$.

Radix Sort може обробляти цифри кожного числа, починаючи або з найменш значущої цифри (LSD – Least Significant Digit), або з найбільш значущої цифри (MSD – Most Significant Digit).

LSD-алгоритм спочатку сортує список за найменш значущою цифрою, зберігаючи відносний порядок елементів за допомогою стабільного сортування. Потім він сортує за наступною цифрою і так далі, від найменш значущої до найбільш значущої, в результаті отримуючи відсортований список. LSD Radix Sort вимагає використання стабільного сортування.

MSD-алгоритм починає сортування з найбільш значущої цифри. In-place реалізація MSD Radix Sort не є стабільною (якщо не потрібна стабільність).

Зазвичай для внутрішнього сортування в Radix Sort використовується Counting Sort.

Гібридний підхід до сортування, наприклад, використання Insertion Sort для невеликих груп елементів (bins), значно покращує продуктивність Radix Sort.

Distribution Sort (розподільче сортування). Distribution Sort відноситься до будь-якого алгоритму сортування, в якому дані розподіляються з вхідного масиву до кількох проміжних структур, які потім об’єднуються та розміщуються на виході.

Bucket Sort є прикладом Distribution Sort.

Timsort (Тімсорт). Timsort шукає впорядковані підпослідовності (runs) у вхідних даних, створює такі підпослідовності за допомогою Insertion Sort (якщо це необхідно), а потім використовує Merge Sort для отримання остаточного відсортованого списку.

Timsort має таку ж складність O(nlogn) в середньому та найгіршому випадках, але для попередньо відсортованих даних складність знижується до O(n).

Далі наводяться важливі зауваження про особливості використання пам'яті та сортування за індексами.

Коли розмір масиву, який потрібно відсортувати, наближається або перевищує обсяг доступної основної пам'яті, і необхідно використовувати (значно повільніший) дисковий простір або підкачку (swap space), тоді шаблон використання пам'яті алгоритму сортування стає важливим. Алгоритм, який був досить ефективним, коли масив легко вміщувався в RAM, може стати непрактичним.

У такій ситуації загальна кількість порівнянь стає відносно менш важливою, а кількість разів, коли частини пам'яті потрібно копіювати або переміщувати на диск та з нього, може домінувати у характеристиках продуктивності алгоритму. Таким чином, кількість проходів (passes) і локалізація порівнянь можуть бути важливішими, ніж загальна кількість порівнянь. Це відбувається тому, що порівняння близьких елементів відбувається на швидкості системної шини (або, при кешуванні, навіть на швидкості процесора), що, порівняно зі швидкістю диска, є практично миттєвим.

Приклад з QuickSort. Наприклад, популярний рекурсивний алгоритм QuickSort забезпечує досить високу продуктивність за наявності достатньої кількості RAM. Однак через рекурсивний спосіб копіювання частин масиву він стає набагато менш практичним, коли масив не вміщується в RAM, оскільки може викликати велику кількість повільних операцій копіювання або переміщення на диск і з нього.

У такій ситуації може бути кращим інший алгоритм, навіть якщо він вимагає більше загальної кількості порівнянь.

Сортування за індексами (Tag Sort). Один зі способів обійти цю проблему, який добре працює, коли складні записи (наприклад, у реляційній базі даних) сортуються за відносно невеликим ключовим полем, — це створення індексу для масиву, а потім сортування індексу, а не всього масиву.

Відсортовану версію всього масиву можна отримати за один прохід, читаючи дані за індексом, але часто в цьому навіть немає потреби, оскільки відсортованого індексу зазвичай достатньо. Оскільки індекс значно менший,

ніж увесь масив, він може легко поміститися в пам'яті, де весь масив не помістився б, ефективно усуваючи проблему підкачки на диск. Ця процедура іноді називається "Tag Sort".

Комбінування алгоритмів. Інша техніка подолання проблеми з обсягом пам'яті – це поєднання двох алгоритмів таким чином, щоб скористатися сильними сторонами кожного з них для покращення загальної продуктивності.

Наприклад, масив можна розділити на частини (chunks) такого розміру, який легко поміщається в RAM (скажімо, кілька тисяч елементів).

Частини сортуються за допомогою ефективного алгоритму (наприклад, QuickSort або HeapSort). Результати об'єднуються за допомогою MergeSort. Це менш ефективно, ніж просто виконати MergeSort з самого початку, але вимагає менше фізичної RAM (для практичного використання), ніж повний QuickSort для всього масиву.

Комбінування технік для дуже великих наборів даних. Також можна поєднувати кілька технік. Для сортування дуже великих наборів даних, які значно перевищують обсяг системної пам'яті, навіть індекс може потребувати сортування за допомогою алгоритму або комбінації алгоритмів, які спроєктовані для ефективної роботи з віртуальною пам'яттю. Такі алгоритми спрямовані на зменшення кількості підкачок на диск.

1.5 Огляд і аналіз існуючих програмних продуктів для оцінки ефективності алгоритмів

Оцінка ефективності алгоритмів є критично важливою в сфері комп'ютерних наук та програмування, оскільки вона дозволяє визначити, наскільки швидко та з якими ресурсами алгоритм може вирішити задачу. Ось короткий огляд та аналіз деяких програмних продуктів, які використовуються для таких оцінок

1. Профілювання програмного забезпечення:

Профілювальники (наприклад, gprof для C, cProfile для Python) дозволяють вимірювати час виконання окремих частин коду. Вони корисні для визначення "вузьких місць" в алгоритмах, але не дають асимптотичних оцінок складності.

2. Бенчмаркингові інструменти:

Google Benchmark для C++ - дозволяє створювати мікробенчмарки для вимірювання продуктивності різних частин коду. Інструмент є особливо корисним для порівняння різних реалізацій одного алгоритму.

JMH (Java Microbenchmark Harness) для Java - надає точні вимірювання часу виконання коду, враховуючи JIT-компіляцію та інші особливості JVM.

3. Аналізатори складності:

Big O Calculator - онлайн-інструменти, які допомагають визначити асимптотичну складність алгоритму на основі введеного коду. Це корисно для теоретичного аналізу, але не для практичного тестування.

Code complexity analyzers (наприклад, SonarQube, CodeCoverage) більше зосереджені на загальній якості коду, але також можуть включати метрики, пов'язані з ефективністю, на кшталт цикломатичної складності.

4. Інструменти для порівняльного аналізу алгоритмів:

Algorithm Visualizer - веб-додатки або програми, що дозволяють візуалізувати роботу алгоритмів, що може допомогти в порівнянні їх ефективності на різних наборах даних.

LeetCode та HackerRank надають платформи, де можна тестувати алгоритми на реальних задачах, але не мають вбудованих інструментів для детального аналізу ефективності.

5. Спеціалізовані програмні бібліотеки:

NumPy для Python може бути використаний для швидкого виконання числових операцій та порівняння ефективності алгоритмів, спеціалізованих на обробці даних.

Аналіз:

Інструменти профілювання та бенчмаркингу корисні для реального життя, де потрібно оцінити продуктивність алгоритму на конкретних апаратних і програмних платформах. Для теоретичного аналізу складності алгоритмів більше підходять онлайн-калькулятори та аналізатори складності, які надають оцінки типу O(n), Ω(n), Θ(n). Важливо враховувати, що різні інструменти можуть дати різні результати залежно від тестових даних, платформи та методів вимірювання. Тому для повноцінного аналізу часто потрібно використовувати кілька інструментів. Програмне забезпечення оновлюється, тож важливо перевіряти актуальність інструментів для сучасних мов програмування та середовищ виконання-

Більш детально опишемо продукт Big(O) Calculator. Big(O) Calculator - це інструмент, який використовується для оцінки продуктивності та ефективності алгоритму. Це дозволяє нам оцінити, як змінюватиметься час роботи алгоритму або використання пам’яті зі збільшенням розміру вхідних даних.

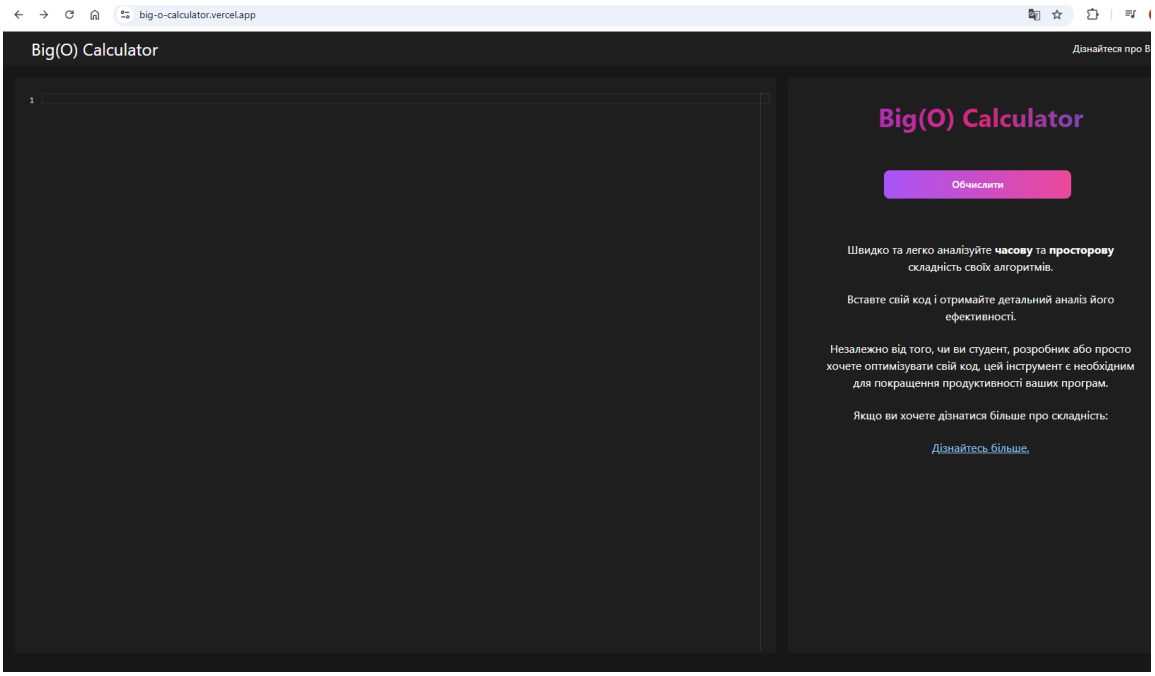

Рис. 1.2. Скріншот головної сторінки Big O Calculator

Простіше кажучи, Big(O) Calculator інформує нас про швидкість алгоритму, оскільки кількість даних, які йому необхідно обробити,

збільшується. Ці знання допомагають нам зрозуміти, як погіршуватиметься продуктивність алгоритму зі збільшенням розміру проблеми.

Big(O) Calculator зазвичай представлена функцією, яка описує найгіршу часову складність алгоритму. Функція виражається через розмір вхідних даних, позначений «n». Аналізуючи цю функцію, ми можемо визначити масштабованість алгоритму та його ефективність на більших вхідних даних. Нотація Big O забезпечує стандартизований спосіб опису продуктивності алгоритму в термінах розміру вхідних даних. Можна порівняти продуктивність різних алгоритмів і вибрати той, який найбільше підходить для даної проблеми. Визначте області, де можна оптимізувати алгоритм і покращити його продуктивність.

Складність. Розуміння часової та просторової складності має вирішальне значення для розробки та аналізу алгоритмів. У більшості випадків ми хочемо розробити алгоритми, які ефективно витрачають час і простір. Однак часто існує компроміс між часовою та просторовою складністю. Швидкий алгоритм може потребувати більше пам’яті, тоді як алгоритм, який використовує менше пам’яті, може бути повільнішим. Часова складність означає кількість часу, який потрібен для виконання алгоритму.

Наприклад, якщо алгоритм має часову складність O(n), це означає, що час роботи алгоритму зростатиме лінійно зі збільшенням розміру вхідних даних. Якщо розмір вхідних даних подвоїться, час роботи алгоритму також подвоїться. Складність простору стосується обсягу пам’яті, який повинен виконати алгоритм.

1.6 Постановка задачі

Мета. Розробити настільний програмний додаток для визначення ефективності алгоритмів сортування для різних наборів даних з графічним інтерфейсом користувача (GUI) типу Windows Forms Application, який дозволить користувачам досліджувати та порівнювати ефективність декількох

основних алгоритмів сортування. При цьому повинна бути присутня програмна можливість додати новий алгоритм сортування

Функціональні вимоги:

Генерація масиву:

Додаток повинен надавати можливість користувачеві обирати тип числових даних.

Додаток повинен надавати можливість користувачеві вводити розмірність масиву з обраним типом чисел.

Після введення розмірності, додаток повинен генерувати масив випадкових чисел заданої розмірності.

Генерація масиву повинна відбуватися в окремому потоці для кожного обраного алгоритму.

Додаток повинен візуалізувати процес сортування для кожного з алгоритмів.

Візуалізація повинна відображати зміни у масиві на кожному кроці сортування.

Візуалізація може бути представлена у вигляді графіків, діаграм або інших відповідних візуальних елементів.

Виведення інформації про тривалість сортування:

Додаток повинен вимірювати час виконання кожного алгоритму сортування.

Вивести на екран інформацію про тривалість сортування для кожного алгоритму.

Надати можливість порівняти тривалість сортування між різними алгоритмами.

Технічні вимоги:

Розробка додатку повинна здійснюватися на платформі .NET Framework з використанням мови програмування C#.

Для створення графічного інтерфейсу користувача використовувати Windows Forms Application.

Використання багатопоточності, для виконання задач генерації масиву та сортування.

Додаток повинен забезпечувати коректну обробку помилок та виключень.

Інтерфейс користувача повинен бути інтуїтивно зрозумілим та зручним у використанні.

Ця постановка задачі надає чітке розуміння цілей, вимог та критеріїв оцінювання для розробки програмного додатку, який дозволить користувачам досліджувати та порівнювати ефективність алгоритмів сортування.

Висновки до розділу 1

У першому розділі наданий огляд основних концепцій структур даних та алгоритмів сортування. Розглядаються класифікації структур даних (примітивні та непримітивні, лінійні та нелінійні) та їхні приклади. Описуються ключові аспекти алгоритмів сортування, такі як їхня ефективність, класифікація (внутрішнє та зовнішнє сортування), стійкість та обчислювальна складність. Наголошується на важливості вибору оптимального алгоритму сортування залежно від конкретних вимог та характеристик даних. Розглядаються фактори, що впливають на ефективність алгоритмів сортування, такі як розмірність даних, їх розподіл, початкова впорядкованість та характеристики самого алгоритму. Наведені характеристики та опис основних алгоритмів сортування.

Наданий огляд інструментів та методів для оцінки ефективності алгоритмів, поділяючи їх на профілювальники, бенчмаркингові інструменти, аналізатори складності, інструменти для порівняльного аналізу та спеціалізовані бібліотеки.

Зроблена постановка задачі, де розробляється додаток для визначення ефективності алгоритмів сортування для різних наборів даних з графічним інтерфейсом користувача (GUI) типу Windows Forms Application.

Розділ 2. СПЕЦИФІКАЦІЯ ВИМОГ ДО ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Глосарій

Створюється глосарій для задачі розробки настільного програмного додатку для аналізу ефективності алгоритмів сортування. Глосарій включає ключові терміни, які можуть бути використані в контексті розробки програмного забезпечення, з поясненнями їхнього значення.

Алгоритм сортування - набір інструкцій, який упорядковує елементи масиву в певному порядку (наприклад, за зростанням або спаданням). Приклади: сортування бульбашкою, швидке сортування, сортування злиттям.

Масив - структура даних, що складається з набору елементів одного типу, доступ до яких здійснюється за індексами.

Генерація масиву - процес створення масиву випадкових чисел із заданими параметрами (розмірність, тип даних).

Тип числових даних - категорія чисел, яку користувач може обрати для генерації масиву (наприклад, цілі числа (int), числа з плаваючою комою (float, double)).

Розмірність масиву - кількість елементів у масиві, яку задає користувач.

Графічний інтерфейс користувача (GUI) - візуальна оболонка програми, яка дозволяє користувачеві взаємодіяти з додатком через графічні елементи (кнопки, поля введення, графіки тощо).

Windows Forms Application - технологія в .NET Framework для створення настільних додатків із графічним інтерфейсом на базі форм Windows.

Багатопоточність - техніка програмування, яка дозволяє виконувати кілька задач одночасно в окремих потоках (наприклад, генерація масиву та сортування).

Потік (Thread) - одиниця виконання в програмі, яка дозволяє паралельно виконувати задачі.

Візуалізація - графічне відображення процесу сортування, яке показує зміни в масиві на кожному етапі (наприклад, через стовпчикові діаграми або графіки).

Тривалість сортування - час, витрачений алгоритмом на впорядкування масиву, який вимірюється в мілісекундах або секундах.

Порівняння ефективності - аналіз продуктивності алгоритмів сортування на основі їхньої тривалості виконання для однакових наборів даних.

Обробка помилок - механізм у програмі, який виявляє та коректно реагує на помилки або виключення (наприклад, введення некоректної розмірності масиву).

Випадкові числа - числа, згенеровані псевдовипадковим чином для заповнення масиву.

.NET Framework - платформа розробки від Microsoft, яка надає бібліотеки та інструменти для створення додатків, включаючи підтримку C# та Windows Forms.

C# (C Sharp) - об’єктно-орієнтована мова програмування, яка використовується для розробки додатку.

Інтуїтивно зрозумілий інтерфейс - дизайн GUI, який є простим і зручним для користувача без спеціальних знань.

Виключення (Exception) - непередбачена ситуація в програмі, яка може призвести до помилки (наприклад, переповнення пам’яті або некоректний ввід).

Програмна можливість - функціональність додатку, яка дозволяє розширювати його, наприклад, додавати нові алгоритми сортування.

Продуктивність алгоритму - характеристика, яка визначається швидкістю виконання та обчислювальною складністю алгоритму (наприклад, $O(n^2)$, $O(n \log n)$).

Концептуальна модель використання інформаційної системи (Use Case Model) для розроблюваного настільного програмного додатку з аналізу ефективності алгоритмів сортування описує основних акторів (користувачів) та сценарії їхньої взаємодії з системою (випадки використання або Use Cases). Вона допомагає зрозуміти, як система буде застосовуватися на практиці та які функції вона повинна виконувати. Нижче наведено графічний (рис. 1.1) та текстовий опис концептуальної моделі для виконуваної задачі.

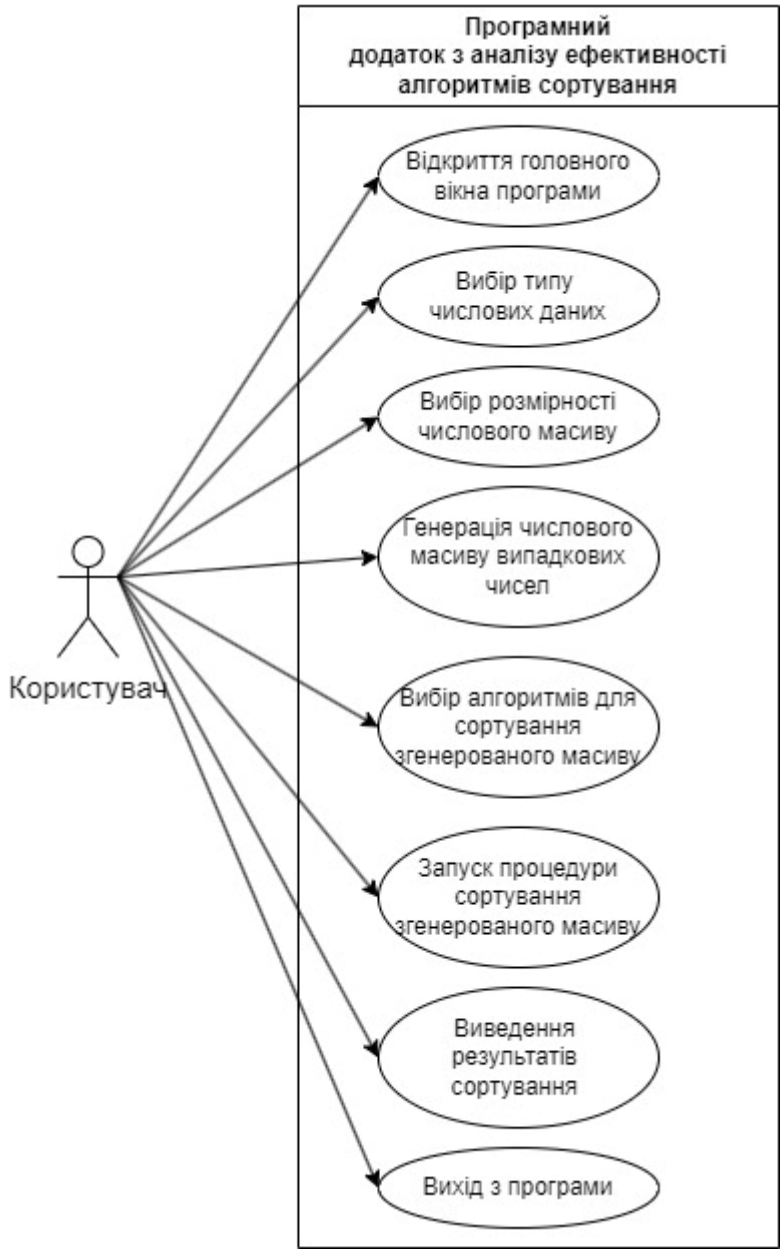

Рис. 1.2. Діаграма використання для програмного додатку з аналізу ефективності алгоритмів сортування

Актори:

Користувач - особа, яка взаємодіє з додатком для дослідження та порівняння алгоритмів сортування. Це може бути студент, викладач, розробник або будь-хто, хто цікавиться аналізом алгоритмів.

Випадки використання (Use Cases).

1. Вибір типу числових даних

Опис: Користувач обирає тип чисел (наприклад, цілі числа (int), числа з плаваючою комою (float)) для генерації масиву.

Попередні умови: Додаток запущено, головне вікно відкрито.

Основний потік:

- Користувач відкриває меню або панель налаштувань.

- Користувач обирає тип даних із запропонованого списку, що випадає.

- Система зберігає вибір для подальшого використання.

Результат: Тип даних для масиву обрано.

2. Введення розмірності масиву

Опис: Користувач задає розмір масиву (кількість елементів).

Попередні умови: Тип числових даних уже обрано.

Основний потік:

- Користувач вводить числове значення у відповідне текстове поле.

- Система перевіряє коректність введених даних (наприклад, чи є число позитивним).

- Якщо дані некоректні, система видає повідомлення про помилку.

- Якщо дані коректні, система зберігає розмірність.

Результат: Розмірність масиву визначено.

3. Генерація масиву

Опис: Система генерує масив випадкових чисел заданого типу та розмірності.

Попередні умови: Тип даних і розмірність масиву введено.

Основний потік:

- Користувач натискає кнопку «Генерувати».

- Система запускає генерацію масиву в окремому потоці.

- Система відображає згенерований масив (наприклад, у вигляді списку або графіка).

Результат: Масив випадкових чисел створено і готовий до сортування.

4. Вибір алгоритмів сортування

Опис: Користувач обирає один або кілька алгоритмів сортування для аналізу.

Попередні умови: Додаток запущено, доступний список алгоритмів.

Основний потік:

- Користувач переглядає список доступних алгоритмів (наприклад, бульбашкове сортування, швидке сортування).

- Користувач через прапорці обирає алгоритми.

- Система зберігає вибір для подальшого використання.

Результат: Обрано алгоритми для сортування.

5. Виконання сортування

Опис: Система сортує згенерований масив обраними алгоритмами.

Попередні умови: Масив згенеровано, алгоритми обрано.

Основний потік:

- Користувач натискає кнопку "Сортувати".

- Система запускає сортування для кожного алгоритму в окремому потоці.

- Система вимірює час виконання кожного алгоритму.

Результат: Масив відсортовано кожним алгоритмом, час виконання зафіксовано.

6. Візуалізація процесу сортування

Опис: Система відображає процес сортування для кожного алгоритму.

Попередні умови: Сортування запущено.

Основний потік:

Система показує зміни в масиві на кожному кроці сортування (наприклад, через стовпчикову діаграму або анімацію).

Користувач спостерігає за процесом у реальному часі або в прискореному режимі.

Результат: Користувач бачить, як алгоритми сортують масив.

7. Виведення результатів

Опис: Система показує тривалість сортування для кожного алгоритму та дозволяє їх порівняти.

Попередні умови: Сортування завершено.

Основний потік:

Система відображає час виконання кожного алгоритму (наприклад, у таблиці).

Користувач може порівняти результати (наприклад, через графік або виділення найшвидшого алгоритму).

Результат: Користувач отримує інформацію про ефективність алгоритмів.

8. Обробка помилок

Опис: Система реагує на помилки, що виникають під час роботи.

Попередні умови: Виникла помилка (наприклад, некоректний ввід).

Основний потік:

Система виявляє помилку.

Система відображає повідомлення з описом проблеми (наприклад, "Введіть додатне число").

Користувач виправляє помилку.

Результат: Помилка оброблена, робота додатку продовжується.

2.3 Специфікація функціональних та нефункціональних вимог

Специфікація функціональних та нефункціональних вимог для настільного програмного додатку з аналізу ефективності алгоритмів сортування є важливим документом, який деталізує, що саме система повинна робити (функціональні вимоги) і якими характеристиками вона повинна володіти (нефункціональні вимоги).

Функціональні вимоги описують конкретні функції, які додаток повинен виконувати, щоб відповідати цілям проекту. Функціональні вимоги базуються на описаних у постановці задачі можливостях додатку, таких як генерація масиву, візуалізація, порівняння тощо.

FR1. Генерація масиву:

- FR1.1. Додаток повинен дозволяти користувачеві обирати тип числових даних для масиву (int, float, double) через інтерфейс у вигляді списку, що випадає.

- FR1.2. Додаток повинен надавати поле для введення розмірності масиву (кількості елементів) у вигляді цілого позитивного числа.

- FR1.3. Після введення розмірності та натискання кнопки «Генерувати», додаток повинен створювати масив випадкових чисел заданого типу та розміру.

- FR1.4. Генерація масиву повинна виконуватися в окремому потоці для кожного алгоритму, щоб уникнути блокування інтерфейсу.

- FR1.5. Згенерований масив повинен відображатися у графічному інтерфейсі в вигляді списку.

FR2. Вибір алгоритмів сортування:

- FR2.1. Додаток повинен надавати список доступних алгоритмів сортування (наприклад, бульбашкове сортування, швидке сортування, сортування злиттям) для вибору користувачем.

- FR2.2. Користувач повинен мати можливість обрати один або кілька алгоритмів одночасно через прапорці.

FR3. Виконання сортування:

- FR3.1. Додаток повинен запускати сортування згенерованого масиву для кожного обраного алгоритму після натискання кнопки «Сортувати».

- FR3.2. Сортування для кожного алгоритму повинно виконуватися в окремому потоці, щоб забезпечити паралельну обробку.

- FR3.3. Додаток повинен вимірювати час виконання кожного алгоритму сортування з точністю до мілісекунд.

FR4. Візуалізація процесу сортування:

- FR4.1. Додаток повинен відображати процес сортування для кожного алгоритму в реальному часі або в прискореному режимі.

- FR4.2. Візуалізація повинна показувати зміни в масиві на кожному кроці сортування (наприклад, через стовпчикові діаграми, графіки або анімацію).

- FR4.3. Користувач повинен мати можливість призупиняти або регулювати швидкість візуалізації (опціонально, якщо це реалізовано).

FR5. Виведення результатів:

- FR5.1. Додаток повинен відображати тривалість сортування для кожного алгоритму у зрозумілому вигляді (наприклад, у таблиці або списку).

- FR5.2. Додаток повинен надавати можливість порівняння тривалості сортування між алгоритмами (наприклад, через графік або виділення найшвидшого алгоритму).

FR6. Додавання нового алгоритму:

- FR6.1. Додаток повинен надавати інтерфейс для введення користувачем нового алгоритму сортування (наприклад, через текстове поле для коду або завантаження файлу).

- FR6.2. Система повинна перевіряти коректність введеного алгоритму (синтаксис, можливість виконання).

- FR6.3. Після успішної перевірки новий алгоритм повинен додаватися до списку доступних для аналізу.

FR7. Обробка помилок:

- FR7.1. Додаток повинен виявляти помилки введення (наприклад, від’ємна розмірність масиву) і виводити відповідні повідомлення користувачеві.

- FR7.2. Система повинна коректно обробляти виключення (наприклад, переповнення пам’яті) і не допускати аварійного завершення роботи.

Нефункціональні вимоги визначають якісні характеристики системи, такі як продуктивність, зручність використання, надійність тощо. Нефункціональні вимоги враховують технічні аспекти (платформа, багатопоточність) та якісні характеристики (зручність, продуктивність), які забезпечують ефективну роботу системи.

NFR1. Платформа розробки:

- NFR1.1. Додаток повинен бути розроблений на платформі .NET Framework з використанням мови програмування C#.

- NFR1.2. Графічний інтерфейс користувача повинен бути створений за допомогою Windows Forms Application.

NFR2. Продуктивність:

- NFR2.1. Генерація масиву розміром до 10,000 елементів повинна виконуватися не довше ніж за 1 секунду.

- NFR2.2. Сортування масиву розміром до 10,000 елементів повинно завершуватися протягом 10 секунд для найповільнішого алгоритму (наприклад, бульбашкового сортування).

- NFR2.3. Використання багатопоточності не повинно призводити до перевантаження процесора (максимальне використання CPU не більше 80%).

NFR3. Зручність використання:

- NFR3.1. Інтерфейс користувача повинен бути інтуїтивно зрозумілим, з чіткими позначеннями кнопок і полів (наприклад, "Генерувати", "Сортувати").

- NFR3.2. Усі елементи GUI повинні бути доступними без необхідності прокручування на екрані з роздільною здатністю 1280x720.

- NFR3.3. Користувач повинен мати можливість виконати повний цикл аналізу (генерація → вибір алгоритмів → сортування → результати) за не більше ніж 5 кліків.

NFR4: Надійність:

- NFR4.1. Додаток не повинен аварійно завершувати роботу при коректному використанні.

- NFR4.2. У разі виникнення помилок система повинна зберігати стабільність і повертатися до початкового стану.

NFR5. Розширюваність:

- NFR5.1. Архітектура додатку повинна дозволяти додавати нові алгоритми сортування без значних змін у коді основної програми.

- NFR5.2. Код повинен бути модульним, з чітким розділенням логіки генерації, сортування та візуалізації.

NFR6. Сумісність:

- NFR6.1. Додаток повинен працювати на операційних системах Windows 10 та новіших версіях.

- NFR6.2. Додаток не повинен вимагати додаткового програмного забезпечення, крім .NET Framework.

NFR7. Безпека:

- NFR7.1. Додаток не повинен дозволяти виконання шкідливого коду через функцію додавання нового алгоритму (наприклад, обмеження доступу до системних ресурсів).

2.4 Технічне завдання

Технічне завдання (ТЗ) для розробки настільного програмного додатку з аналізу ефективності алгоритмів сортування є документом, який детально описує цілі, вимоги, технічні аспекти та етапи реалізації проєкту. Воно базується на попередньо розглянутих функціональних і нефункціональних

вимогах, а також на постановці задачі. ТЗ враховує всі аспекти постановки задачі, включаючи багатопоточність, візуалізацію та розширюваність.

Технічне завдання по розробці настільного додатку для аналізу ефективності алгоритмів сортування

1. Загальні положення

1.1. Назва проєкту «Аналізатор ефективності сортування».

1.2. Мета проєкту: Розробити настільний програмний додаток із графічним інтерфейсом користувача (GUI) типу Windows Forms Application для дослідження та порівняння ефективності алгоритмів сортування на різних наборах даних із можливістю додавання нових алгоритмів.

1.3. Замовник: вищий навчальний заклад (ВНЗ).

1.4. Виконавець: студент ВНЗ.

1.5. Термін виконання: до 10 квітня 2025 року.

2. Призначення та цілі

2.1. Призначення: Додаток призначений для користувачів (студентів, викладачів, розробників), які бажають досліджувати продуктивність алгоритмів сортування, візуалізувати їхню роботу та порівнювати результати.

2.2. Цілі:

- Забезпечити генерацію масивів випадкових чисел із заданими параметрами.

- Реалізувати виконання кількох алгоритмів сортування з вимірюванням часу.

- Надавати візуалізацію процесу сортування.

- Дозволити користувачам додавати власні алгоритми сортування.

- Забезпечити зручний та інтуїтивний інтерфейс.

3. Вимоги до програми

3.1. Функціональні вимоги

3.1.1. Генерація масиву:

- Користувач може обрати тип чисел (int, float, double) через випадаючий список.

- Користувач вводить розмірність масиву (ціле позитивне число) у текстове поле.

- Після натискання кнопки «Генерувати» додаток створює масив випадкових чисел у окремому потоці.

- Згенерований масив відображається у графічному інтерфейсі.

3.1.2. Вибір алгоритмів:

- Додаток надає список базових алгоритмів (наприклад, Bubble Sort, Quick Sort, Merge Sort) із можливістю вибору через прапорці.

3.1.3. Виконання сортування:

- Сортування запускається кнопкою "Сортувати" для всіх обраних алгоритмів.

- Кожен алгоритм виконується в окремому потоці.

- Час виконання кожного алгоритму вимірюється з точністю до мілісекунд.

3.1.4. Візуалізація:

- Процес сортування відображається у вигляді анімації для кожного алгоритму.

- Користувач бачить зміни в масиві на кожному кроці.

3.1.5. Виведення результатів:

- Тривалість сортування відображається у таблиці або списку.

- Надається графік для порівняння часу виконання алгоритмів.

- Система перевіряє коректність коду перед додаванням.

3.1.6. Обробка помилок:

- Некоректний ввід (наприклад, від’ємна розмірність) супроводжується повідомленням про помилку.

- Виключення обробляються без аварійного завершення роботи.

3.2. Нефункціональні вимоги

3.2.1. Технології:

- Платформа: .NET Framework.

- Мова програмування: C#.

- GUI: Windows Forms Application.

3.2.2. Продуктивність:

- Генерація масиву до 10,000 елементів – до 1 секунди.

- Сортування масиву до 10,000 елементів – до 10 секунд для найповільнішого алгоритму.

- Максимальне використання CPU – до 80%.

3.2.3. Зручність:

- Інтерфейс інтуїтивно зрозумілий, усі функції доступні за 5 кліків.

- Розмір вікна адаптований до роздільної здатності 1280x720.

3.2.4. Надійність:

- Відсутність аварійного завершення при коректному використанні.

- Стабільність при помилках.

3.2.5. Розширюваність:

- Модульна архітектура для легкого додавання нових алгоритмів.

3.2.6. Сумісність:

- Підтримка Windows 10 та Windows 11.

4. Склад і зміст робіт

4.1. Етапи розробки:

- Аналіз вимог. Збір і уточнення вимог (1 тиждень).

- Проєктування. Розробка архітектури та дизайну інтерфейсу (2 тижні).

- Реалізація. Кодування функціоналу (4 тижні).

- Тестування. Перевірка коректності роботи та продуктивності (2 тижні).

- Документація. Складання інструкції користувача (1 тиждень).

- Розгортання. Підготовка до запуску (1 тиждень).

4.2. Загальний термін: 11 тижнів.

5. Технічні характеристики

5.1. Мінімальні системні вимоги:

- ОС: Windows 10 або 11.

- Процесор: 1.5 ГГц, 2 ядра.

- Оперативна пам’ять: 2 ГБ.

- Вільне місце на диску: 100 МБ.

- .NET Framework 4.8 або новіший.

5.2. Вихідний продукт: Виконуваний файл (.exe) із графічним інтерфейсом.

6. Порядок контролю та приймання

6.1. Критерії приймання:

- Усі функціональні вимоги реалізовані.

- Продуктивність відповідає зазначеним параметрам.

- Інтерфейс зручний і стабільний.

- Документація та інструкція надані.

6.2. Тестування:

- Функціональні тести з перевіркою всіх функцій (генерація, сортування, візуалізація).

- Навантажувальні тести робота з масивами до 10,000 елементів.

- Тести на помилки у вигляді введення некоректних даних.

6.3. Приймання: Замовник перевіряє додаток і підписує акт приймання-здачі.

7. Додаткові вказівки

7.1. Документація в вигляді інструкції користувача у форматі PDF.

7.2. Мова інтерфейсу є українська.

7.3. Джерело коду зберігається у репозиторії GitHub із коментарями.

Висновки до розділу 2

Створений глосарій для задачі розробки настільного програмного додатку для аналізу ефективності алгоритмів сортування. Глосарій включає ключові терміни, які використовуються в контексті розробки програмного забезпечення, з поясненнями їхнього значення.

Створені специфікації функціональних та нефункціональних вимог для настільного програмного додатку з аналізу ефективності алгоритмів сортування є важливим документом, який деталізує, що саме система повинна робити (функціональні вимоги) і якими характеристиками вона повинна володіти (нефункціональні вимоги). Функціональні вимоги базуються на описаних у постановці задачі можливостях додатку, таких як генерація масиву, візуалізація, порівняння тощо. Нефункціональні вимоги враховують технічні аспекти (платформа, багатопоточність) та якісні характеристики (зручність, продуктивність), які забезпечують ефективну роботу системи.

Створено технічне завдання (ТЗ) для розробки настільного програмного додатку з аналізу ефективності алгоритмів сортування є документом, який детально описує цілі, вимоги, технічні аспекти та етапи реалізації проєкту. Воно базується на попередньо розглянутих функціональних і нефункціональних вимогах, а також на постановці задачі. ТЗ враховує всі аспекти постановки задачі, включаючи багатопоточність, візуалізацію та розширюваність.

Розділ 3. ОПИС ПРИЙНЯТИХ ПРОЄКТНИХ РІШЕНЬ

3.1 Опис вихідних і вхідних даних

Опис вхідних і вихідних даних для настільного програмного додатку з аналізу ефективності алгоритмів сортування є важливим для розуміння того, які дані користувач вводить у систему (вхідні дані) і які результати система повертає (вихідні дані). Нижче наводиться детальний опис, базований на постановці задачі та попередніх специфікаціях.

3.1.1. Вхідні дані

Вхідні дані - це інформація, яку користувач або система вводить у додаток для виконання його функцій. Вони визначають параметри роботи програми. Вхідні дані вводяться користувачем через графічний інтерфейс.

1. Тип числових даних

Тип даних, який визначає формат чисел у масиві. Вибір формату із заздалегідь визначеного списку. Джерело обирається користувачем через графічний інтерфейс у вигляді списку, що випадає. Лише один тип даних може бути обраний для одного масиву.

Можливі значення:

- int (цілі числа, наприклад, -2,147,483,648 до 2,147,483,647).

- float (числа з плаваючою комою одинарної точності, наприклад, 3.14).

- double (числа з плаваючою комою подвійної точності, наприклад, 3.14159265359).

2. Розмірність масиву

Кількість елементів у масиві, яку задає користувач. Форматом є ціле позитивне число. Діапазон значень обирається від 1 до 50, що є рекомендованим обмеженням для стабільної роботи; може бути змінено залежно від апаратних можливостей). Джерелом є введення користувачем до

текстового поля в інтерфейсі. Значення має бути більше 0. Некоректні значення (наприклад, від’ємні числа, літери) викликають повідомлення про помилку.

3. Обрані алгоритми сортування

Список алгоритмів, які користувач хоче проаналізувати, надаються у форматі множини назв алгоритмів (наприклад, "Bubble Sort", "Quick Sort", "Merge Sort"). Обирається користувачем через прапорці або список у графічному інтерфейсі. Має бути обрано принаймні один алгоритм. Кількість обраних алгоритмів обмежена лише ресурсами системи.

3.1.2. Вихідні дані

Вихідні дані - це результати, які додаток повертає користувачеві після обробки вхідних даних. Вони відображають роботу системи. Вихідні дані відображають результати обробки і представлені як у текстовому, так і в графічному вигляді для зручності аналізу.

1. Згенерований масив

Масив випадкових чисел, створений на основі обраного типу даних і розмірності у форматі одновимірного масиву чисел. Представлення у вигляді текстового списку у графічному інтерфейсі. Візуалізація у вигляді точок. Кількість елементів відповідає введеній розмірності.

Приклад:

- Для int, розмірність 5: [45, 12, 89, 3, 67].
- Для float, розмірність 5: [3.14, 0.25, 9.81, 1.41, 7.77].

2. Візуалізація процесу сортування

Графічне відображення змін у масиві під час сортування для кожного алгоритму у форматі динамічної візуалізації (анімація або покрокове оновлення). Представлення в GUI реалізується у вигляді окремого графічного елементу для кожного алгоритму (наприклад, кілька панелей на екрані). Частота оновлення залежить від продуктивності системи (наприклад, 10 кадрів/секунду).

3. Тривалість сортування

Час виконання кожного алгоритму сортування для заданого масиву у форматі числового значення в мілісекундах (ms). Представлення у вигляді таблиці з колонками «Алгоритм» і «Час виконання». Графік для порівняння (наприклад, стовпчикова діаграма). Точність вимірювання залежить від системного таймера (зазвичай до 1 ms).

Приклад:

- Bubble Sort: 245 ms.
- Quick Sort: 12 ms.
- Merge Sort: 18 ms.

4. Відсортований масив

Результат сортування масиву для кожного алгоритму у форматі одновимірного масиву чисел у впорядкованому вигляді (за зростанням). Представлення у вигляді текстового списку або графічного відображення в інтерфейсі. Кожен алгоритм повертає однаковий результат для одного масиву (якщо реалізація коректна).

Приклад:

- Вхідний масив: [45, 12, 89, 3, 67].
- Вихідний масив: [3, 12, 45, 67, 89].

3.2 Розробка об’єктної моделі

1. MainForm

Клас, що відповідає за графічний інтерфейс користувача (GUI) на базі Windows Forms. Координує взаємодію між компонентами системи та відображає результати.

Атрибути:

arrayGenerator: ArrayGenerator

Опис: Посилання на об'єкт класу ArrayGenerator для створення масивів.

Тип: ArrayGenerator

Призначення: Використовується для генерації масивів із заданими параметрами (розмір, тип даних).

sortingExecutor: SortingExecutor

Опис: Посилання на об'єкт класу SortingExecutor для виконання алгоритмів сортування.

Тип: SortingExecutor

Призначення: Координує запуск сортування в окремих потоках і збір результатів.

sortingVisualizer: SortingVisualizer

Опис: Посилання на об'єкт класу SortingVisualizer для відображення процесу сортування.

Тип: SortingVisualizer

Призначення: Забезпечує візуалізацію кроків сортування.

errorHandler: ErrorHandler

Опис: Посилання на об'єкт класу ErrorHandler для обробки помилок.

Тип: ErrorHandler

Призначення: Використовується для обробки некоректного введення та виключень.

results: List

Опис: Список результатів сортування для всіх виконаних алгоритмів.

Тип: List<SortingResult>

Призначення: Зберігає дані про час виконання алгоритмів для відображення в таблиці або на графіку.

Методи:

GenerateArray()

Опис: Ініціює генерацію масиву з параметрами, заданими користувачем (розмір, тип даних).

Повертає: Нічого (void)

Призначення: Викликає метод GenerateAsync класу ArrayGenerator і відображає згенерований масив у GUI.

SelectAlgorithms()

Опис: Дозволяє користувачу вибрати алгоритми сортування через прапорці.

Повертає: Нічого (void)

Призначення: Оновлює список обраних алгоритмів у SortingExecutor.

StartSorting()

Опис: Запускає процес сортування для всіх обраних алгоритмів.

Повертає: Нічого (void)

Призначення: Викликає ExecuteAsync класу SortingExecutor і координує візуалізацію.

DisplayResults()

Опис: Відображає результати сортування (час виконання) у таблиці або на графіку.

Повертає: Нічого (void)

Призначення: Використовує список results для побудови таблиці/графіка в GUI.

AddCustomAlgorithm()

Опис: Дозволяє користувачу додати власний алгоритм сортування.

Повертає: Нічого (void)

Призначення: Перевіряє коректність коду алгоритму та додає його до списку алгоритмів у SortingExecutor.

2. ArrayGenerator

Клас для генерації масивів випадкових чисел із заданими параметрами.

Атрибути:

size: int

Опис: Розмірність масиву, введена користувачем.

Тип: int

Призначення: Визначає кількість елементів у згенерованому масиві.

dataType: string

Опис: Тип даних елементів масиву (наприклад, "int", "float", "double").

Тип: string

Призначення: Визначає, які числа генерувати (цілі, з плаваючою комою тощо).

Методи:

GenerateAsync(): Task<object[]>

Опис: Генерує масив випадкових чисел у окремому потоці.

Повертає: Task<object[]> (асинхронно повертає масив об'єктів)

Призначення: Створює масив із випадковими числами відповідно до size і dataType.

3. ISortingAlgorithm

Інтерфейс, який визначає контракт для всіх алгоритмів сортування, забезпечуючи розширюваність.

Методи:

Sort(array: object[]): object[]

Опис: Виконує сортування заданого масиву.

Параметри: array: object[] – масив для сортування.

Повертає: object[] – відсортований масив.

Призначення: Реалізується кожним алгоритмом для виконання сортування.

GetName(): string

Опис: Повертає назву алгоритму.

Повертає: string – ім'я алгоритму (наприклад, "Bubble Sort").

Призначення: Використовується для ідентифікації алгоритму в GUI та результатах.

4. BubbleSort

Клас, що реалізує алгоритм бульбашкового сортування.

Методи:

Sort(array: object[]): object[]

Опис: Виконує бульбашкове сортування масиву.

Параметри: array: object[] – масив для сортування.

Повертає: object[] – відсортований масив.

Призначення: Реалізує логіку сортування шляхом порівняння сусідніх елементів.

GetName(): string

Опис: Повертає назву алгоритму.

Повертає: string – "Bubble Sort".

Призначення: Ідентифікує алгоритм у системі.

5. QuickSort

Клас, що реалізує алгоритм швидкого сортування.

Методи:

Sort(array: object[]): object[]

Опис: Виконує швидке сортування масиву.

Параметри: array: object[] – масив для сортування.

Повертає: object[] – відсортований масив.

Призначення: Використовує рекурсивний підхід із вибором опорного елемента.

GetName(): string

Опис: Повертає назву алгоритму.

Повертає: string – "Quick Sort".

Призначення: Ідентифікує алгоритм у системі.

6. MergeSort

Клас, що реалізує алгоритм сортування злиттям.

Методи:

Sort(array: object[]): object[]

Опис: Виконує сортування злиттям масиву.

Параметри: array: object[] – масив для сортування.

Повертає: object[] – відсортований масив.

Призначення: Розділяє масив на підмасиви, сортує їх і об'єднує.

GetName(): string

Опис: Повертає назву алгоритму.

Повертає: string – "Merge Sort".

Призначення: Ідентифікує алгоритм у системі.

7. SortingExecutor

Клас для асинхронного виконання алгоритмів сортування та вимірювання їхнього часу.

Атрибути:

algorithms: List

Опис: Список обраних алгоритмів сортування.

Тип: List<ISortingAlgorithm>

Призначення: Зберігає алгоритми, які будуть виконані.

Методи:

ExecuteAsync(array: object[]): Task<List>

Опис: Асинхронно виконує всі обрані алгоритми сортування.

Параметри: array: object[] – масив для сортування.

Повертає: Task<List<SortingResult>> – список результатів сортування.

Призначення: Запускає кожен алгоритм у окремому потоці і збирає результати.

MeasureTime(algorithm: ISortingAlgorithm, array: object[]): double

Опис: Вимірює час виконання одного алгоритму.

Параметри:

algorithm: ISortingAlgorithm – алгоритм для виконання.

array: object[] – масив для сортування.

Повертає: double – час виконання в мілісекундах.

Призначення: Використовує таймер для точного вимірювання продуктивності.

8. SortingVisualizer

Клас для візуалізації процесу сортування.

Методи:

VisualizeStep(array: object[], step: int)

Опис: Відображає один крок сортування.

Параметри:

array: object[] – поточний стан масиву.

step: int – номер кроку.

Повертає: Нічого (void)

Призначення: Оновлює графічне відображення масиву (наприклад, стовпчики чи точки).

RenderAnimation(algorithm: ISortingAlgorithm, array: object[])

Опис: Відображає повну анімацію сортування для одного алгоритму.

Параметри:

algorithm: ISortingAlgorithm – алгоритм, що виконується.

array: object[] – масив для сортування.

Повертає: Нічого (void)

Призначення: Координує послідовність викликів VisualizeStep для створення анімації.

9. SortingResult

Клас для зберігання результатів сортування одного алгоритму.

Атрибути:

algorithmName: string

Опис: Назва алгоритму, який виконав сортування.

Тип: string

Призначення: Ідентифікує алгоритм у результатах.

executionTime: double

Опис: Час виконання алгоритму в мілісекундах.

Тип: double

Призначення: Зберігає виміряний час для порівняння.

Методи:

GetAlgorithmName(): string

Опис: Повертає назву алгоритму.

Повертає: string – значення algorithmName.

Призначення: Використовується для відображення в GUI.

GetExecutionTime(): double

Опис: Повертає час виконання.

Повертає: double – значення executionTime.

Призначення: Використовується для побудови графіків і таблиць.

10. ErrorHandler

Клас для обробки помилок і виключень.

Методи:

HandleInvalidInput(message: string)

Опис: Обробляє некоректне введення користувача.

Параметри: message: string – повідомлення про помилку.

Повертає: Нічого (void)

Призначення: Відображає повідомлення про помилку в GUI (наприклад, від'ємна розмірність масиву).

HandleException(ex: Exception)

Опис: Обробляє виключення, що виникають під час виконання.

Параметри: ex: Exception – виключення.

Повертає: Нічого (void)

Призначення: Логує помилку та відображає користувачу дружнє повідомлення, запобігаючи аварійному завершенню.

3.3 Розробка архітектури

Для визначення програмної архітектури системи «Аналізатор ефективності сортування» на основі технічного завдання (ТЗ) та попередньо створеної діаграми класів, я проаналізую вимоги, зокрема функціональні (генерація масивів, виконання алгоритмів, візуалізація, обробка помилок) та нефункціональні (модульність, продуктивність, розширюваність, використання C# і Windows Forms). Архітектура буде описана як комбінація архітектурного шаблону, структури компонентів і їх взаємодії, з урахуванням модульної структури та вимог до асинхронного виконання. Для представлення

архітектури я використаю текстовий опис та діаграму компонентів у форматі PlantUML, що відповідає ТЗ.

Аналіз вимог для архітектури

Функціональні вимоги:

Генерація масивів у окремих потоках.

Виконання кількох алгоритмів сортування асинхронно з вимірюванням часу.

Візуалізація процесу сортування.

Можливість додавання нових алгоритмів (розширюваність).

Відображення результатів у таблиці або на графіку.

Обробка помилок без аварійного завершення.

Нефункціональні вимоги:

Модульна архітектура для легкого додавання алгоритмів.

Використання .NET Framework, C#, Windows Forms.

Продуктивність: генерація масивів до 10,000 елементів за 1 секунду, сортування за 10 секунд.

Інтуїтивний інтерфейс, адаптований до роздільної здатності 1280x720.

Стабільність і надійність.

Ключові аспекти:

Необхідність асинхронної обробки для генерації масивів і сортування.

Модульність для підтримки нових алгоритмів.

Чітке розділення логіки інтерфейсу, обробки даних і візуалізації.

Вибір архітектурного шаблону

На основі вимог найкраще підходить шарувата архітектура (Layered Architecture) з елементами MVC (Model-View-Controller) для організації взаємодії між інтерфейсом і логікою. Причини вибору:

Шарувата архітектура забезпечує чітке розділення відповідальностей (інтерфейс, бізнес-логіка, обробка даних), що сприяє модульності та легкому тестуванню.

MVC допомагає організувати Windows Forms додаток, де:

Model: Дані та логіка (генерація масивів, алгоритми сортування, результати).

View: Графічний інтерфейс (Windows Forms).

Controller: Координація між View і Model (обробка подій користувача, запуск сортування).

Асинхронне виконання підтримується через використання Task і async/await у C#.

Модульність досягається через інтерфейс ISortingAlgorithm для алгоритмів сортування.

Структура архітектури

Архітектура поділяється на три основні шари:

Шар представлення (Presentation Layer):

Відповідає за GUI (Windows Forms).

Включає MainForm, яка обробляє введення користувача, відображає масиви, результати та візуалізацію.

Взаємодіє з користувачем через кнопки, текстові поля, прапорці та графічні елементи.

Шар бізнес-логіки (Business Logic Layer):

Містить логіку обробки даних і координацію процесів.

Включає:

ArrayGenerator: Генерація масивів.

SortingExecutor: Асинхронне виконання алгоритмів і вимірювання часу.

SortingVisualizer: Координація візуалізації.

ErrorHandler: Обробка помилок.

Забезпечує модульність через інтерфейс ISortingAlgorithm, який дозволяє додавати нові алгоритми.

Шар даних (Data Layer):

Відповідає за зберігання даних і алгоритмів.

Включає:

SortingResult: Зберігання результатів сортування.

Конкретні реалізації ISortingAlgorithm (BubbleSort, QuickSort, MergeSort).

У цій системі немає бази даних, тому шар даних обмежується тимчасовим зберіганням масивів і результатів у пам’яті.

Компоненти та їх взаємодія

MainForm (Presentation Layer): Координує взаємодію користувача з системою, викликаючи методи інших компонентів.

ArrayGenerator (Business Logic Layer): Генерує масиви асинхронно та повертає їх у MainForm.

SortingExecutor (Business Logic Layer): Керує виконанням алгоритмів у окремих потоках, використовуючи ISortingAlgorithm.

Dot Net Framework (DNF) для асинхронності.

SortingVisualizer (Business Logic Layer): Отримує дані від SortingExecutor і відображає анімацію через MainForm.

ErrorHandler (Business Logic Layer): Обробляє помилки та повертає повідомлення у MainForm.

ISortingAlgorithm та реалізації (BubbleSort, QuickSort, MergeSort) (Data Layer): Надають алгоритми для SortingExecutor.

SortingResult (Data Layer): Зберігає результати для відображення у MainForm.

Діаграма компонентів

Для наочності архітектура представлена у вигляді діаграми компонентів у форматі PlantUML.

Component Diagram for Sorting Efficiency Analyzer

plantuml

Show inline

Опис взаємодії компонентів

Генерація масиву:

Користувач вводить розмір і тип даних у MainForm.

MainForm викликає ArrayGenerator.GenerateAsync().

ArrayGenerator асинхронно генерує масив і повертає його в MainForm для відображення.

У разі некоректного введення ErrorHandler.HandleInvalidInput() відображає повідомлення.

Вибір і виконання сортування:

Користувач обирає алгоритми через прапорці в MainForm.

MainForm передає список алгоритмів і масив у SortingExecutor.ExecuteAsync().

SortingExecutor запускає кожен алгоритм (ISortingAlgorithm.Sort()) у окремому потоці, вимірює час (MeasureTime) і формує список SortingResult.

У разі виключень ErrorHandler.HandleException() обробляє помилку.

Візуалізація:

SortingExecutor передає проміжні стани масиву в SortingVisualizer.

SortingVisualizer.RenderAnimation() викликає VisualizeStep() для відображення анімації через MainForm.

Відображення результатів:

MainForm отримує список SortingResult від SortingExecutor.

Результати відображаються в таблиці або на графіку через GUI.

Додавання алгоритмів:

Користувач додає код алгоритму через MainForm.AddCustomAlgorithm().

MainForm перевіряє код і додає новий клас, що реалізує ISortingAlgorithm, до списку алгоритмів у SortingExecutor.

Переваги архітектури

Модульність: Інтерфейс ISortingAlgorithm дозволяє легко додавати нові алгоритми.

Розділення відповідальностей: Шари чітко відокремлені, що полегшує тестування та підтримку.

Асинхронність: Використання Task і async/await забезпечує продуктивність і уникнення блокування GUI.

Обробка помилок: Централізована обробка через ErrorHandler гарантує стабільність.

Відповідність ТЗ: Архітектура підтримує всі функціональні та нефункціональні вимоги, включаючи продуктивність і зручність інтерфейсу.

Висновок

Запропонована шарувата архітектура з елементами MVC ідеально відповідає вимогам ТЗ, забезпечуючи модульність, розширюваність і продуктивність. Діаграма компонентів у PlantUML наочно ілюструє взаємодію між компонентами, що полегшить подальшу розробку та документування системи.

3.4 Засоби розробки

C# - це потужна, об'єктно-орієнтована, типобезпечна мова програмування, що використовується для створення надійних та безпечних додатків на платформі .NET Framework. Її обирають завдяки широкому спектру можливостей, які роблять її гнучким інструментом для розробників [11].

Windows Forms - це платформа програмування, що використовується для розробки графічних інтерфейсів користувача (GUI) для програм Windows. Вона є частиною .NET Framework і надає розробникам широкий спектр інструментів та компонентів для створення багатих та динамічних інтерфейсів.

Переваги використання Windows Forms:

- Windows Forms пропонує широкий спектр компонентів, які можна використовувати для створення різноманітних інтерфейсів. Розробники також можуть створювати власні компоненти, щоб розширити функціональні можливості своїх програм;

- завдяки візуальному дизайнеру, Windows Forms полегшує створення інтерфейсів користувача. Розробники можуть просто перетягувати та кидати компоненти на форму, а потім налаштовувати їхні властивості;

- Windows Forms підтримує широкий спектр функцій, таких як обробка подій, зв'язування даних та багатопотоковість. Це дозволяє створювати складні та масштабовані програми;

- Windows Forms тісно інтегрується з .NET Framework, надаючи доступ до широкого спектру класів та бібліотек, що робить можливим створення більш гнучких та потужних програм.

Для забезпечення роботи бази даних використовується Microsoft SQL Server, яка є потужною реляційною системою управління базами даних (СУБД).

Підтримує реляційну модель даних, що дозволяє зберігати та керувати даними в табличному форматі. Забезпечує можливість створення складних запитів за допомогою мови SQL. Оптимізована для обробки великих обсягів даних. Підтримує різні індекси та оптимізацію запитів для підвищення швидкості виконання операцій. Включає вбудовані механізми аутентифікації та авторизації користувачів. Підтримує шифрування даних як у стані спокою, так і при передачі. Включає вбудовані функції резервного копіювання та відновлення. Підтримує механізми реплікації та кластеризації для забезпечення високої доступності та відмовостійкості. Може працювати як на невеликих системах, так і на великих корпоративних серверах. Тісно інтегрується з Visual Studio.

3.5 Проєктування інтерфейсу програмної системи

Проектування графічного інтерфейсу користувача (GUI) для описаного програмного продукту, який є навчально-демонстраційною програмою для аналізу ефективності алгоритмів сортування (бульбашкою, вставками та вибором), виконано з використанням фреймворку Windows Forms у мові програмування C#. Інтерфейс спроектовано для забезпечення зручної взаємодії користувача з програмою, відображення процесу генерації масивів, виконання сортування та аналізу результатів. Нижче наведено детальний опис

процесу проектування графічного інтерфейсу, його компонентів, їх призначення та логіки розміщення, з урахуванням коду та функціональності, описаних у попередніх розділах.

Графічний інтерфейс створено за допомогою дизайнера форм Windows Forms у Visual Studio, що генерує метод InitializeComponent() у класі Form1. Цей метод ініціалізує всі елементи керування, їх властивості та розташування на формі.

Можна виділити ключові елементи керування, які складають інтерфейс:

1. Елементи введення даних

TextBox (textBox1):

Призначення: Дозволяє користувачеві ввести розмір масиву (ціле додатнє число).

Властивості:

Розташований у верхній частині форми для легкого доступу.

Властивість Enabled = true завжди, як зазначено в методі Active, щоб користувач міг вводити розмір навіть під час виконання операцій.

Перевірка введення здійснюється в обробнику Button1_Click через int.TryParse, що забезпечує коректність даних.

Логіка розміщення: Ймовірно, розташований поруч із міткою (Label), яка пояснює, що потрібно ввести розмір масиву (наприклад, "Розмір масиву:").

ComboBox (comboBoxType):

Призначення: Дозволяє вибрати тип даних масиву (int, double, float).

Властивості:

Заповнюється в конструкторі Form1 значеннями "int", "double", "float".

Властивість SelectedIndex = 0 встановлює "int" як тип за замовчуванням.

Завжди активний (Enabled = true), як зазначено в методі Active.

Логіка розміщення: Розташований поруч із textBox1, можливо, з міткою "Тип даних:", щоб користувач міг швидко вибрати тип перед генерацією масиву.

2. Кнопки керування

`Button (Button1) - "Генерувати масив":

Призначення: Запускає генерацію масиву заданого розміру та типу в фоновому потоці (BackgroundWorker1).

Властивості:

Текст змінюється під час генерації, відображаючи прогрес (наприклад, "Генерувати масив 50%") у методі BackgroundWorker1_ProgressChanged.

Деактивується під час генерації (Button1.Enabled = false) і активується після завершення (BackgroundWorker1_RunWorkerCompleted).

Логіка розміщення: Розташована під textBox1 і comboBoxType для логічного порядку дій: вибір типу → введення розміру → генерація.

`Button (Button2) - "Сортувати":

Призначення: Запускає три фонові потоки (BackgroundWorker2, BackgroundWorker3, BackgroundWorker4) для сортування масиву кожним алгоритмом.

Властивості:

Активна лише після успішної генерації масиву (Button2.Enabled = true у BackgroundWorker1_RunWorkerCompleted).

Деактивується під час сортування (Button2.Enabled = false у Button2_Click).

Логіка розміщення: Розташована поруч із Button1, ймовірно, в тій самій горизонтальній панелі керування.

`Button (Button3) - "Стоп":

Призначення: Скасовує виконання всіх фонових потоків сортування через CancelAsync.

Властивості:

Активна лише під час виконання сортування.

Обробляє винятки (InvalidOperationException) у Button3_Click, якщо потоки не запущені.

Логіка розміщення: Розташована поруч із Button2 для зручного доступу під час сортування.

`Button (Button4) - "Очистити" (згадана в описі, але не в коді):

Призначення: Очищає вміст textBox1, ListBox, ProgressBar, міток часу та скидає внутрішні масиви.

Властивості:

Активує Button1 і деактивує Button2 та Button3 після очищення.

Логіка розміщення: Ймовірно, розташована в тій самій панелі керування, що й інші кнопки.

3. Елементи відображення результатів

ListBox (listBox1, listBox2, listBox3):

Призначення:

listBox1: Відображає результати сортування бульбашкою (arrayBubInt, arrayBubDouble, arrayBubFloat).

listBox2: Відображає результати сортування вставками (arrayInsInt, arrayInsDouble, arrayInsFloat).

listBox3: Відображає результати сортування вибором (arraySelInt, arraySelDouble, arraySelFloat).

Перед сортуванням усі три ListBox відображають оригінальний масив (arrayInt, arrayDouble, arrayFloat).

Властивості:

Очищаються перед генерацією нового масиву (Button1_Click) і після скасування сортування.

Інтерфейс спроектовано за принципом логічного групування елементів:

Верхня панель (введення даних):

Містить textBox1, comboBoxType та Button1 для введення параметрів і запуску генерації.

Мітки для пояснення (наприклад, "Розмір масиву:", "Тип даних:").

Центральна панель (результати сортування):

Три блоки, кожен із ListBox (listBox1, listBox2, listBox3), ProgressBar (progressBar1, progressBar2, progressBar3) та Label (label5, label6, label7) для відображення прогресу та результатів кожного алгоритму.

Розташування: горизонтально (в ряд) або вертикально (у стовпець) для порівняння.

Нижня панель (керування та аналіз):

Кнопки Button2 ("Сортувати"), Button3 ("Стоп") і, можливо, Button4 ("Очистити").

DataGridView для зведених результатів і візуалізації.

3.6 Опис програмної реалізації

Програмний додаток для визначення ефективності алгоритмів сортування для різних наборів даних є навчально-демонстраційною програмою, розробленою для наочного представлення роботи трьох базових алгоритмів сортування: сортування бульбашкою (Bubble Sort), сортування вставками (Insertion Sort) та сортування вибором (Selection Sort). Програма надає користувачеві можливість генерувати масиви числових даних різних типів (цілі числа, числа з плаваючою комою одинарної та подвійної точності) та спостерігати за процесом їх сортування кожним з алгоритмів у реальному часі. Додатково програма вимірює та відображає час виконання кожного алгоритму сортування для порівняльного аналізу їхньої ефективності.

Основні Функціональні Можливості:

- Користувач має можливість обрати тип числових даних для масиву перед його генерацією за допомогою випадаючого списку (ComboBox).

Доступні типи:

- Цілі числа (int)
- Числа з плаваючою комою одинарної точності (float)
- Числа з плаваючою комою подвійної точності (double)

- Користувач вводить бажаний розмір масиву в текстове поле та натискає кнопку "Генерувати масив". Програма створює масив випадкових чисел обраного типу. Процес генерації відображається за допомогою індикатора

прогресу. Згенерований масив відображається у трьох окремих списках (ListBox) для кожного алгоритму сортування.

- Після генерації масиву користувач натискає кнопку "Сортувати". Програма запускає три окремі фонові процеси (потоки) для одночасного сортування копій згенерованого масиву кожним з трьох алгоритмів:

Сортування бульбашкою (Bubble sorting) результат відображається в першому ListBox.

- Сортування вставками (Insertion sorting) результат відображається в другому ListBox.

Сортування вибором (Selection sorting) результат відображається в третьому ListBox.

Процес сортування кожного алгоритму візуалізується за допомогою окремих індикаторів прогресу.

- Після завершення сортування кожним алгоритмом програма відображає час, витрачений на сортування, у відповідних мітках (Label) поруч з індикаторами прогресу. Час відображається у форматі "ГГ:ХХ:СС.МММ".

- Користувач має можливість у будь-який момент зупинити виконання всіх процесів сортування, натиснувши кнопку "Стоп". У цьому випадку поточні результати сортування можуть бути неповними, а у відповідних мітках відобразиться статус "Скасовано".

- Кнопка "Очистити" дозволяє користувачеві очистити всі відображені дані: вміст текстового поля для введення розміру масиву, вміст усіх трьох ListBox, значення індикаторів прогресу та мітки часу. Після очищення кнопка "Генерувати масив" знову стає активною, а кнопки "Сортувати" та "Стоп" деактивуються. Також скидаються внутрішні масиви даних.

Програма розроблена з використанням мови програмування C# та фреймворку Windows Forms (.NET Framework або .NET). Використання фонових потоків (BackgroundWorker) забезпечує виконання тривалих операцій (генерація та сортування) без блокування основного потоку інтерфейсу користувача, що робить програму більш чуйною та зручною у використанні.

Програма реалізована у вигляді класу Form1, який успадковує клас System.Windows.Forms.Form. Далі наводиться детальний опис кожної частини коду.

Оголошення Змінних Класу:

```
// Внутрішні змінні для масивів
private int[] arrayInt; // масив-оригінал int
private int[] arrayBubInt; // масив даних після сортування бульбашкою int
private int[] arrayInsInt; // масив після сортуванням вставкою int
private int[] arraySelInt; // масив після сортування вибором int
private double[] arrayDouble; // масив-оригінал double
private double[] arrayBubDouble; // масив даних після сортування бульбашкою double
private double[] arrayInsDouble; // масив після сортуванням вставкою double
private double[] arraySelDouble; // масив після сортування вибором double
private float[] arrayFloat; // масив-оригінал float
private float[] arrayBubFloat; // масив даних після сортування бульбашкою float
private float[] arrayInsFloat; // масив після сортуванням вставкою float
private float[] arraySelFloat; // масив після сортування вибором float
// Змінна, що зберігає обраний тип масиву
private string selectedArrayType;
// Змінні, що фіксують час початку виконання алгоритмів
TimeSpan tsBubble; // Алгоритм бульбашки
TimeSpan tsIns; // Вставка
TimeSpan tsSel; // Вибір
// Прапорці скасування
bool fCancelBub;
bool fCancelIns;
bool fCancelSel;
```

Оголошуються приватні змінні для зберігання оригінальних та відсортованих масивів для кожного підтримуваного типу даних (int, double, float). Для кожного типу є оригінальний масив (arrayInt, arrayDouble, arrayFloat) та три масиви для зберігання результатів сортування кожним з алгоритмів (arrayBubX, arrayInsX, arraySelX). selectedArrayType (string):

Зберігає обраний користувачем тип даних з ComboBox ("int", "double" або "float"). tsBubble, tsIns, tsSel (TimeSpan): Використовуються для фіксації часу початку виконання кожного алгоритму сортування.

fCancelBub, fCancelIns, fCancelSel (bool) є прапорцями, що вказують, чи була натиснута кнопка "Стоп" під час виконання відповідного алгоритму сортування.

Конструктор Form1() викликає метод InitializeComponent(), згенерований дизайнером форм, для ініціалізації візуальних елементів керування, очищує вміст textBox1 (для введення розміру масиву) та всіх трьох ListBox, скидає значення всіх трьох ProgressBar до 0, додає рядки "int", "double", "float" до елемента comboBoxType (випадаючий список вибору типу даних), встановлює початковий обраний елемент в comboBoxType на "int" (SelectedIndex = 0), викликає метод Active(false) для деактивації елементів керування, які не повинні бути активними до генерації масиву, ініціалізує прапорці скасування (fCancelBub, fCancelIns, fCancelSel) значенням false.

Перевантажений метод DisplayArray() є внутрішнім методом, який відображає масив int в елементі керування типу ListBox

```
private void DisplayArray(int[] A, ListBox LB)
{
    LB.Items.Clear();
    if (A != null) // Перевірка на null
    {
        for (int i = 0; i < A.Length; i++)
            LB.Items.Add(A[i]);
    }
}
```

Внутрішній метод, який відображає масив double в елементі керування типу ListBox

```
private void DisplayArray(double[] A, ListBox LB)
{
    LB.Items.Clear();
```

```
    if (A != null) // Перевірка на null
    {
        // Використовуємо форматування для дробів
        for (int i = 0; i < A.Length; i++)
            LB.Items.Add(A[i].ToString("F4")); // F4 - 4 знаки після коми
    }
}
```

Внутрішній метод, який відображає масив float в елементі керування типу ListBox

```
private void DisplayArray(float[] A, ListBox LB)
{
    LB.Items.Clear();
    if (A != null) // Перевірка на null
    {
        // Використовуємо форматування для дробів
        for (int i = 0; i < A.Length; i++)
            LB.Items.Add(A[i].ToString("F4")); // F4 - 4 знаки після коми
    }
}
```

Ці три перевантажені версії методу DisplayArray використовуються для відображення масивів різних числових типів (int, double, float) у заданому елементі керування ListBox. Кожна версія спочатку очищає вміст ListBox Потім перевіряє, чи переданий масив (A) не є null. Ітерується по елементах масиву та додає кожен елемент до ListBox. Для типів double та float використовується форматування .ToString("F4") для відображення чисел з чотирма знаками після коми.

Метод Active() є внутрішним методом активації елементів керування

```
private void Active(bool active)
{
    // Зробити активними/неактивними деякі елементи управління
    label2.Enabled = active;
    label3.Enabled = active;
    label4.Enabled = active;
```

```
    label5.Enabled = active;
    label6.Enabled = active;
    label7.Enabled = active;
    listBox1.Enabled = active;
    listBox2.Enabled = active;
    listBox3.Enabled = active;
    progressBar1.Enabled = active;
    progressBar2.Enabled = active;
    progressBar3.Enabled = active;
    //button1.Enabled = active; // Ця кнопка керується окремо
    Button2.Enabled = active;
    Button3.Enabled = active;

    // ComboBox та TextBox для розміру масиву завжди активні
    textBox1.Enabled = true;
    comboBoxType.Enabled = true;
}
```

Цей внутрішній метод використовується для одночасного ввімкнення або вимкнення групи елементів керування на формі (міток, ListBox, ProgressBar, кнопок "Сортувати" та "Стоп"). Стан (active - true для активації, false для деактивації) передається як аргумент. Кнопка "Генерувати масив" (Button1), TextBox для розміру масиву (textBox1) та ComboBox для вибору типу (comboBoxType) керуються окремо і завжди залишаються активними.

Обробник Події Button1_Click() (Генерувати масив):

```
// Кнопка "Генерувати масив"
private void Button1_Click(object sender, EventArgs e)
{
    // Зчитати обраний тип перед запуском фонового робітника
    if (comboBoxType.SelectedItem == null)
    {
        MessageBox.Show("Будь ласка, виберіть тип масиву.");
        return;
    }
```

```
    selectedArrayType = comboBoxType.SelectedItem.ToString();

    // Перевірка введення розміру масиву
    if (!int.TryParse(textBox1.Text, out int n) || n <= 0)
    {
        MessageBox.Show("Будь ласка, введіть коректний додатній розмір масиву.");
        return;
    }

    // Деактивувати деякі елементи керування
    Active(false);
    Button1.Enabled = false; // Деактивувати кнопку "Генерувати" під час генерації

    // Налаштувати мітки часу
    label5.Text = " ";
    label6.Text = " ";
    label7.Text = " ";

    // Очистити ListBox перед генерацією нового масиву
    listBox1.Items.Clear();
    listBox2.Items.Clear();
    listBox3.Items.Clear();

    // Запустити генерування масиву в потоці
    if (!BackgroundWorker1.IsBusy)
        BackgroundWorker1.RunWorkerAsync(n); // Передаємо розмір n як аргумент
}
```

Обробник події натискання на кнопку "Генерувати масив" (Button1), перевіряє, чи обрано тип масиву в comboBoxType. Якщо ні, виводить повідомлення, зчитує обраний тип та зберігає його у змінній selectedArrayType, намагається перетворити текст з textBox1 на ціле число (n), яке представляє розмір масиву. Якщо введене значення не є коректним додатнім цілим числом, виводить повідомлення про помилку. Деактивує елементи керування за допомогою методу Active(false) та саму кнопку "Генерувати" (Button1.Enabled

= false) під час виконання операції генерації. Очищає вміст міток часу (label5, label6, label7). Очищає вміст усіх трьох ListBox. Запускає фоновий процес генерації масиву (BackgroundWorker1.RunWorkerAsync()), передаючи розмір масиву (n) як аргумент.

Обробник Події Button2_Click() (Сортувати):

```
// Кнопка "Сортування" - запустити потоки виконання
private void Button2_Click(object sender, EventArgs e)
{
    // Перевірити, чи масив був згенерований для обраного типу
    bool isArrayGenerated = false;
    switch (selectedArrayType)
    {
        case "int": isArrayGenerated = (arrayInt != null); break;
        case "double": isArrayGenerated = (arrayDouble != null); break;
        case "float": isArrayGenerated = (arrayFloat != null); break;
    }
    if (!isArrayGenerated)
    {
        MessageBox.Show("Будь ласка, спочатку згенеруйте масив.");
        return;
    }
    // Деактивувати кнопку генерування масиву та сортування
    Button1.Enabled = false;
    Button2.Enabled = false;
    // Запуск методів сортування у потоках
    if (!BackgroundWorker2.IsBusy)
        BackgroundWorker2.RunWorkerAsync();
    if (!BackgroundWorker3.IsBusy)
        BackgroundWorker3.RunWorkerAsync();
    if (!BackgroundWorker4.IsBusy)
        BackgroundWorker4.RunWorkerAsync();
}
```

Обробник події натискання на кнопку "Сортувати" (Button2) перевіряє, чи був згенерований масив відповідного обраного типу (arrayInt, arrayDouble або arrayFloat не є null). Якщо масив не згенеровано, виводить повідомлення. Деактивує кнопки "Генерувати масив" (Button1) та "Сортувати" (Button2) під час виконання операцій сортування. Запускає три фонові процеси для виконання алгоритмів сортування:

- BackgroundWorker2.RunWorkerAsync(): Сортування бульбашкою.
- BackgroundWorker3.RunWorkerAsync(): Сортування вставками.
- BackgroundWorker4.RunWorkerAsync(): Сортування вибором.

Обробник Події Button3_Click() (Стоп):

```
// Кнопка Стоп – скасувати виконання всіх потоків
private void Button3_Click(object sender, EventArgs e)
{
    try
    {
        // Перевіряємо, чи потоки взагалі запущені, перш ніж скасовувати
        if (BackgroundWorker2.IsBusy) BackgroundWorker2.CancelAsync(); // зупинити сортування бульбашкою
        if (BackgroundWorker3.IsBusy) BackgroundWorker3.CancelAsync(); // Зупинити сортування вставками
        if (BackgroundWorker4.IsBusy) BackgroundWorker4.CancelAsync(); // зупинити сортування вибором
    }
    catch (InvalidOperationException ex)
    {
        // Ця помилка може виникнути, якщо спробувати скасувати незапущений worker
        MessageBox.Show("Помилка при скасуванні: " + ex.Message);
    }
    // Після натискання "Стоп", кнопки мають повернутись у відповідний стан
        }
```

Обробник події натискання на кнопку "Стоп" (Button3) намагається скасувати виконання всіх трьох фонових робітників, викликаючи метод

CancelAsync() для кожного з них. Блок try-catch обробляє можливу виняткову ситуацію InvalidOperationException, яка може виникнути при спробі скасувати робітника, який ще не був запущений.

Обробник Події BackgroundWorker1_DoWork() (Генерування масиву) здійснює виконання потоку, у якому генерується масив чисел

```
private void BackgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    int n = (int)e.Argument; // Отримуємо розмір масиву з аргументів
    Random rnd = new Random();

    switch (selectedArrayType)
    {
        case "int":
            arrayInt = new int[n];
            arrayBubInt = new int[n];
            arrayInsInt = new int[n];
            arraySelInt = new int[n];
            for (int i = 0; i < n; i++)
            {
                //Thread.Sleep(1); // Прибрав sleep для швидшої генерації, додайте якщо потрібна візуалізація прогресу
                int value = rnd.Next(1, n + 1); // випадкове число int
                arrayInt[i] = value;
                arrayBubInt[i] = arraySelInt[i] = arrayInsInt[i] = value; // скопіювати
                try
                {
                    BackgroundWorker1.ReportProgress((i * 100) / n);
                }
                catch { return; } // Уникаємо UI-операцій у DoWork
            }
            break;
        case "double":
            arrayDouble = new double[n];
            arrayBubDouble = new double[n];
```

```
            arrayInsDouble = new double[n];
            arraySelDouble = new double[n];
            for (int i = 0; i < n; i++)
            {
               //Thread.Sleep(1);
               double value = rnd.NextDouble() * n + 1; // випадкове число
               arrayDouble[i] = value;
               arrayBubDouble[i] = arraySelDouble[i] = arrayInsDouble[i] = value; // скопіювати
               try
               {
                  BackgroundWorker1.ReportProgress((i * 100) / n);
               }
               catch { return; }
            }
            break;
         case "float":
            arrayFloat = new float[n];
            arrayBubFloat = new float[n];
            arrayInsFloat = new float[n];
            arraySelFloat = new float[n];
            for (int i = 0; i < n; i++)
            {
               //Thread.Sleep(1);
               float value = (float)(rnd.NextDouble() * n + 1); // випадкове число
               arrayFloat[i] = value;
               arrayBubFloat[i] = arraySelFloat[i] = arrayInsFloat[i] = value; // скопіювати
               try
               {
                  BackgroundWorker1.ReportProgress((i * 100) / n);
               }
               catch { return; }
            }
            break;
      }
```

}

Цей метод виконується у фоновому потоці при виклику BackgroundWorker1.RunWorkerAsync(). Отримує розмір масиву (n), переданий як аргумент (e.Argument). Створює екземпляр класу Random для генерації випадкових чисел. Використовує оператор switch для обробки обраного типу даних (selectedArrayType).

Для кожного типу (int, double, float) створює масиви відповідного типу: оригінальний (arrayInt, arrayDouble, arrayFloat) та три копії для сортування різними алгоритмами (arrayBubX, arrayInsX, arraySelX). У циклі від 0 до n-1 генерує випадкове число відповідного типу (для int - ціле від 1 до n+1, для double та float - дійсне від 1 до n+1). Присвоює згенероване значення відповідному елементу оригінального масиву та його копіям. Викликає метод BackgroundWorker1.ReportProgress() для повідомлення про хід виконання (у відсотках) головному потоку для оновлення інтерфейсу (тексту кнопки "Генерувати масив"). Блок try-catch використовується для запобігання помилкам, якщо головне вікно буде закрито під час виконання фонового процесу.

Обробники Події BackgroundWorkerX_DoWork() (Сортування). Кожен з цих методів (BackgroundWorker2_DoWork, BackgroundWorker3_DoWork, BackgroundWorker4_DoWork) виконується у своєму окремому фоновому потоці при виклику відповідного RunWorkerAsync(). На початку кожного методу фіксується час початку сортування у відповідну змінну (tsBubble, tsIns, tsSel) та скидається відповідний прапорець скасування (fCancelBub, fCancelIns, fCancelSel). Використовується оператор switch для обробки обраного типу даних (selectedArrayType). Для кожного типу (int, double, float) виконується відповідний алгоритм сортування на копії згенерованого масиву (arrayBubX, arrayInsX, arraySelX). Логіка сортування (порівняння та обмін елементів) є стандартною для кожного з алгоритмів. Метод Thread.Sleep(1) вставляється у зовнішній цикл кожного алгоритму для того, щоб дати можливість іншим потокам (включаючи головний потік інтерфейсу) виконуватися, що необхідно

для оновлення прогрес-барів та обробки події натискання кнопки "Стоп". У кожній ітерації зовнішнього циклу викликається ReportProgress() для повідомлення про хід виконання сортування (у відсотках). Перевіряється властивість CancellationPending відповідного BackgroundWorker. Якщо вона true (була натиснута кнопка "Стоп"), встановлюється відповідний прапорець скасування (fCancelBub, fCancelIns, fCancelSel) і виконання потоку переривається за допомогою break.

Обробники Події BackgroundWorkerX_ProgressChanged() (Оновлення прогресу). Ці методи (BackgroundWorker1_ProgressChanged, BackgroundWorker2_ProgressChanged, BackgroundWorker3_ProgressChanged, BackgroundWorker4_ProgressChanged) виконуються в головному потоці інтерфейсу користувача кожного разу, коли відповідний фоновий робітник викликає ReportProgress(). Вони оновлюють візуальні елементи керування (текст кнопки "Генерувати масив", значення ProgressBar, текст міток прогресу) відповідно до отриманого відсотка виконання. В обробниках прогресу сортування додані перевірки на те, чи не були видалені елементи керування (наприклад, якщо користувач швидко закриває форму під час сортування).

Висновки до розділу 3

Розроблений програмний продукт є навчально-демонстраційною програмою, розробленою для аналізу ефективності трьох базових алгоритмів сортування: бульбашкою, вставками та вибором. Він дозволяє користувачам генерувати масиви чисел типів int, double і float, виконувати сортування в багатопотоковому режимі за допомогою багатопоточності і порівнювати продуктивність алгоритмів за часом виконання.

Програма забезпечує зручний інтерфейс для вибору типу даних (int, double, float) через вибору типу чисел, введення розміру масиву і запуску генерації та сортування за допомогою кнопок "Генерувати масив" і "Сортувати".

Використання багатопотоковості через BackgroundWorker дозволяє виконувати три алгоритми сортування паралельно, що покращує демонстраційну цінність програми, хоча додає накладні витрати на малих масивах.

Індикатори прогресу та мітки часу забезпечують візуалізацію процесу сортування та його результатів, а функція скасування через кнопку "Стоп" дозволяє переривати виконання потоків.

Розділ 4. ДОСЛІДНА ЕКСПЛУАТАЦІЯ МОЖЛИВИХ ЗАСТОСУВАНЬ

4.1 Опис роботи програмного продукту

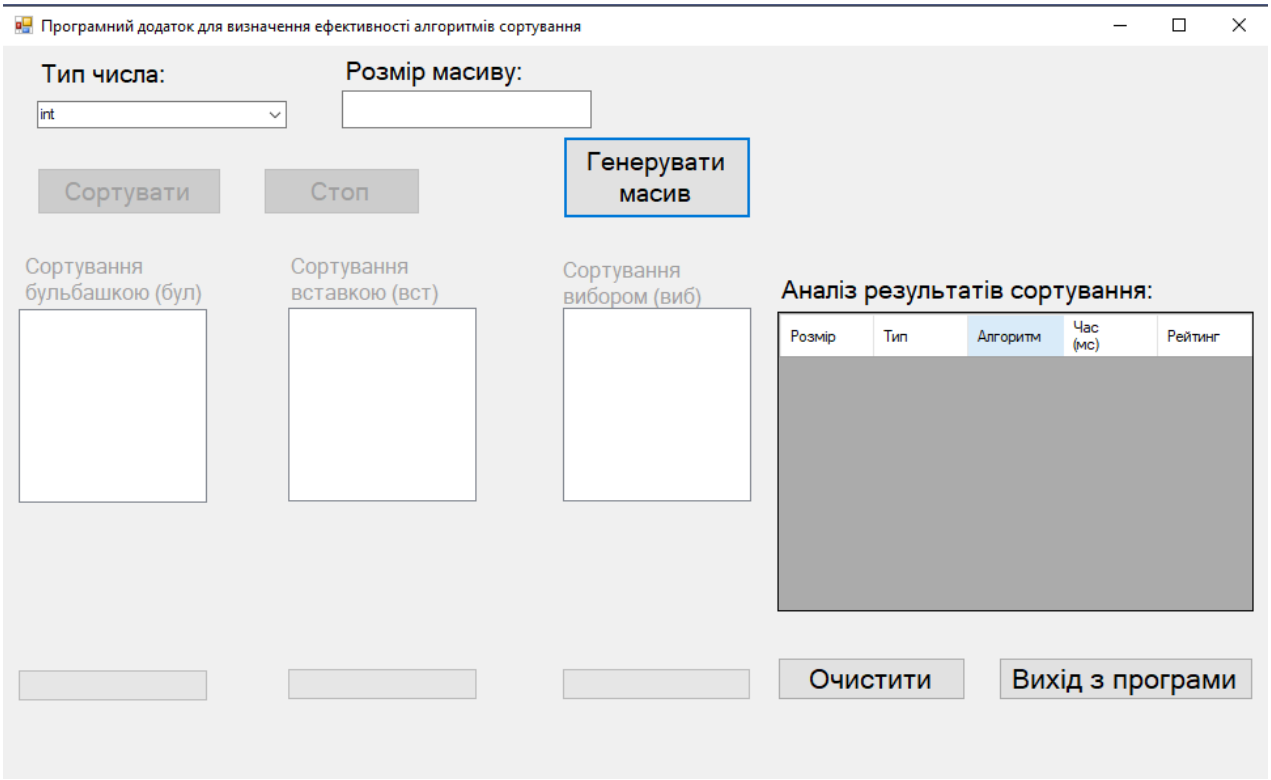

Рис. 4.1. Скріншот запуску програми

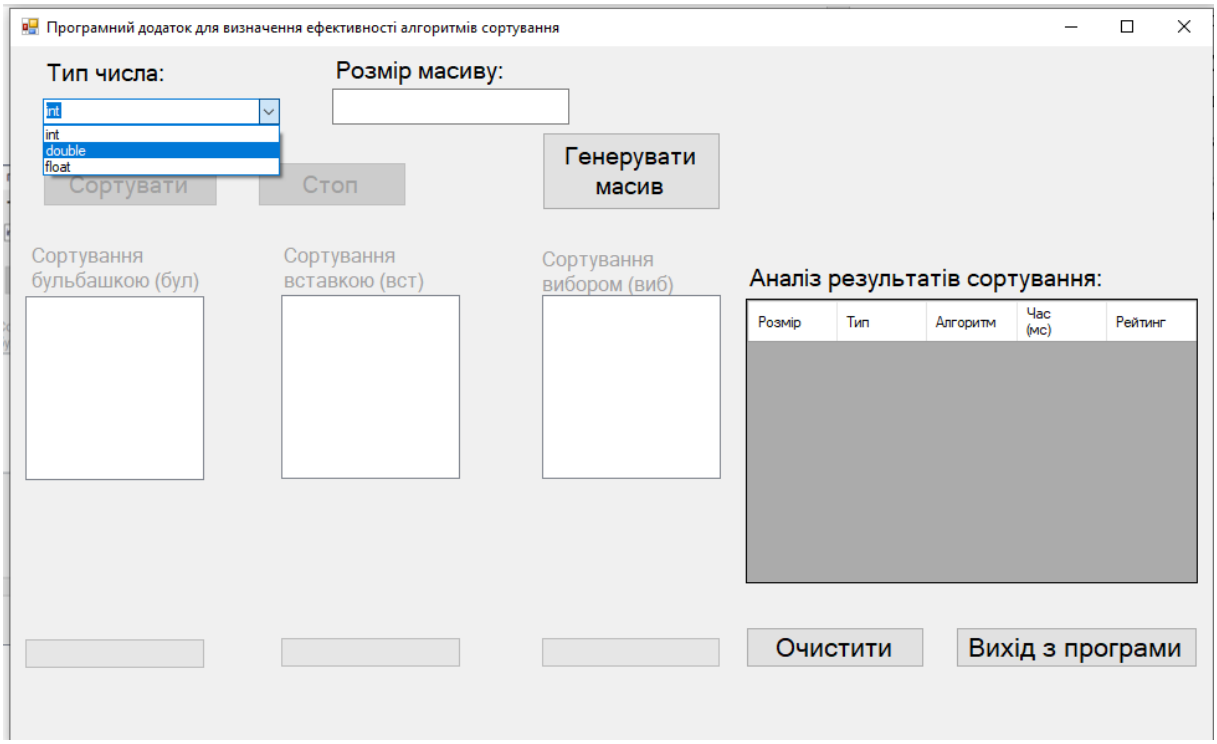

Рис. 4.2. Скріншот вибору типу числа

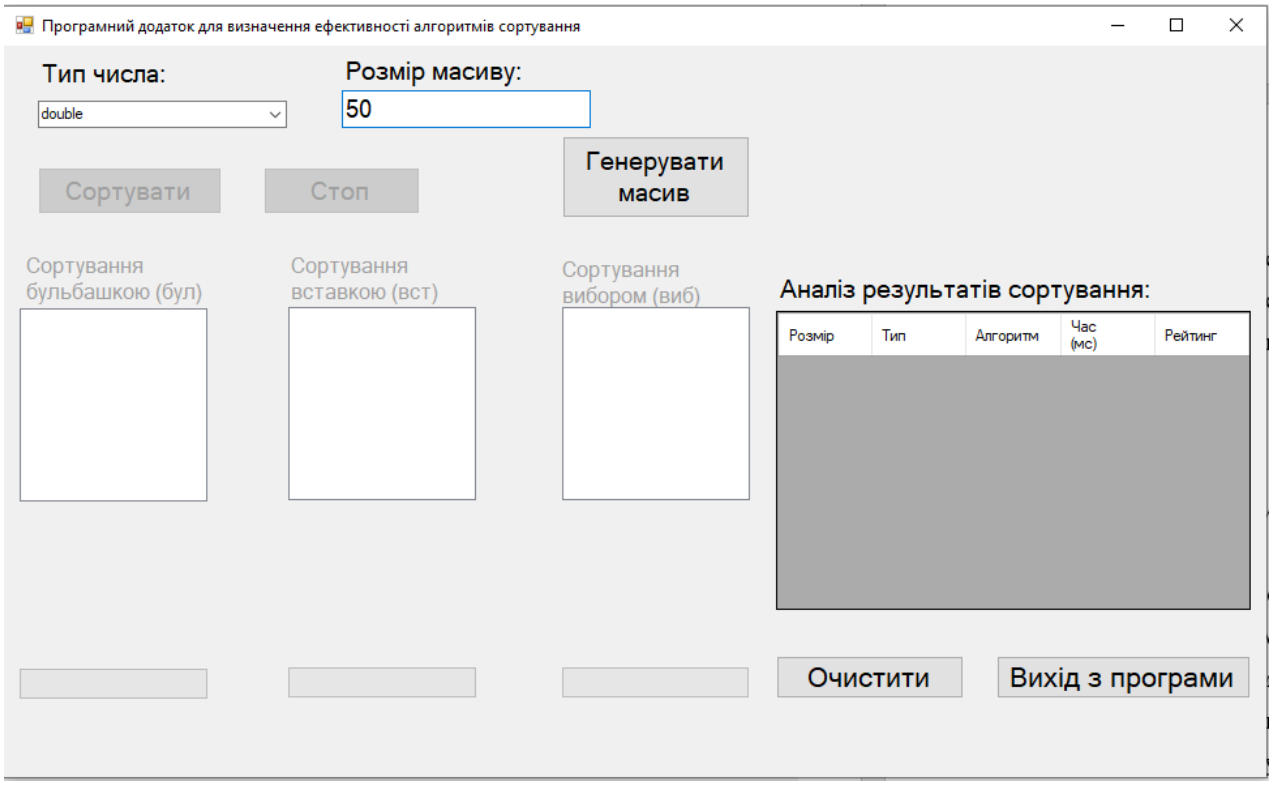

Рис. 4.3. Скріншот введення розміру масиву

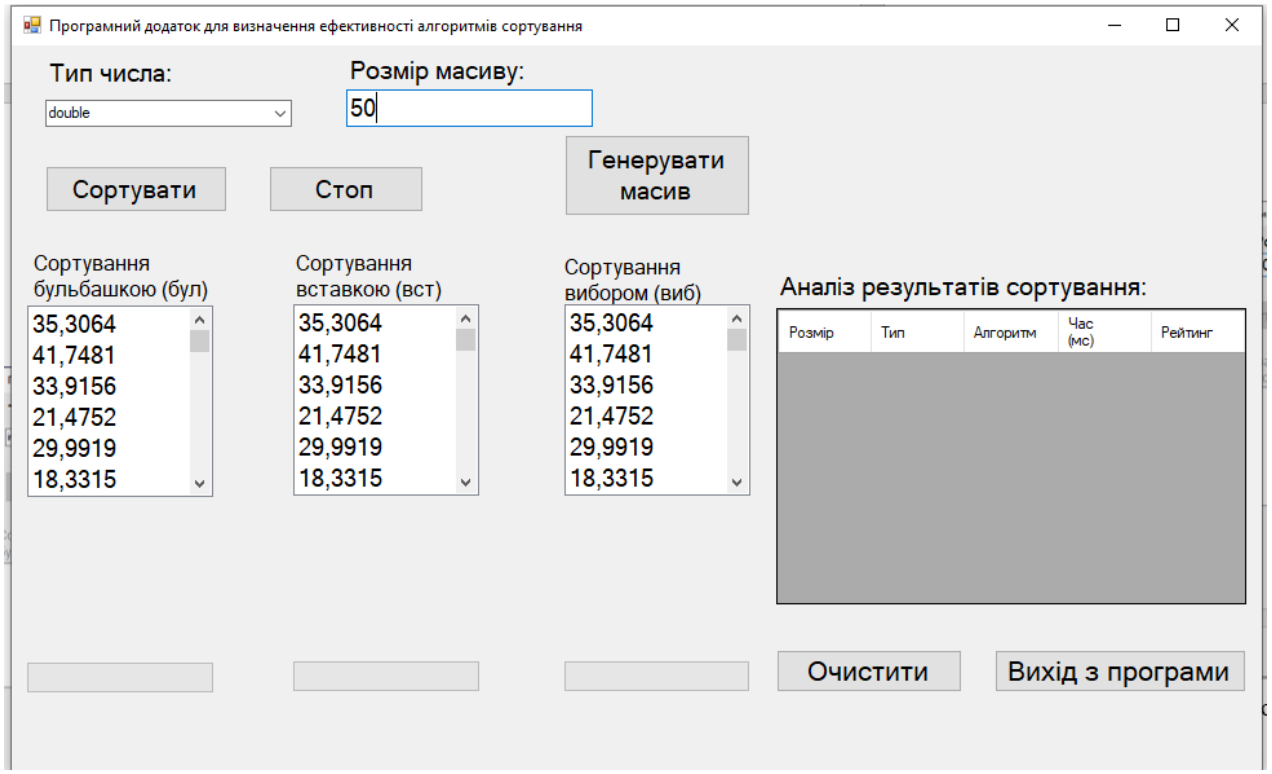

Рис. 4.4. Скріншот результатів генерації випадкового масиву чисел

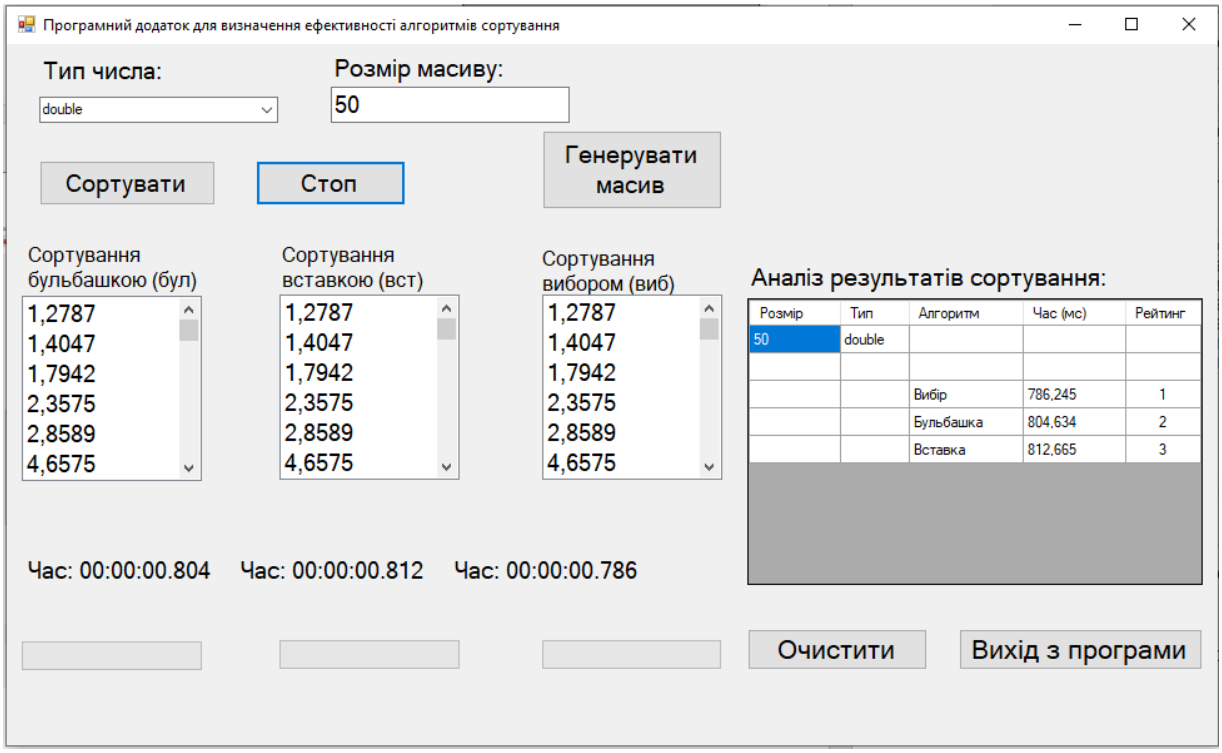

Рис. 4.5. Скріншот результату сортування масивів трьома різними методами

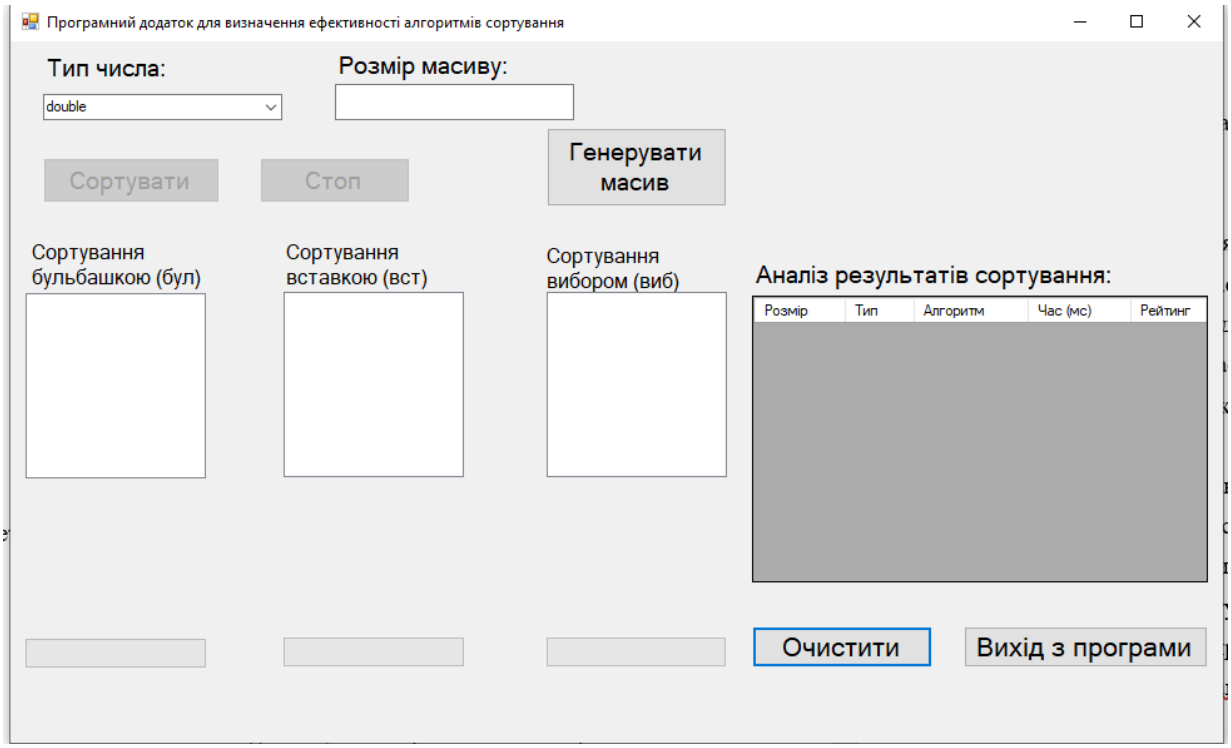

Рис. 4.6. Скріншот результату очищення полів для введення даних

4.2 Методи тестування ефективності алгоритмів

Для оцінки ефективності алгоритмів сортування (бульбашкою, вставками та вибором), реалізованих у наданому програмному продукті, потрібно розробити методи тестування, які дозволять кількісно та якісно оцінити їх продуктивність. Ці методи включають сценарії тестування, інструменти для збору даних, метрики ефективності та аналіз результатів. Нижче наведено детальний опис методів тестування ефективності алгоритмів, з урахуванням особливостей програми, яка працює з масивами типів int, double і float, використовує багатопотоковість через BackgroundWorker і відображає результати в ListBox, ProgressBar та DataGridView.

4.2.1. Сценарії тестування

Методи тестування ефективності алгоритмів будуть використані наступні.

Для оцінки продуктивності алгоритмів сортування використовуватимуться такі метрики для оцінки ефективності як час виконання та стабільність роботи в багатопотоковому середовищі

Порівняння продуктивності для масивів int, double і float. Поведінка залежно від розміру масиву з наданням оцінки масштабованості алгоритмів на малих, середніх і великих масивах. Перевірка коректності обробки скасування потоків.

Сценарії тестування розроблені для оцінки продуктивності алгоритмів у різних умовах. Кожен сценарій включає конкретні вхідні дані, дії користувача та очікувані результати.

Сценарій 1: Тестування на малих масивах

Мета: Оцінити продуктивність алгоритмів на невеликих наборах даних, де накладні витрати багатопотоковості можуть бути значними.

Вхідні дані:

Розмір масиву: 100 елементів.

Типи масивів: int, double, float.

Тип даних: Випадкові згенеровані числа.

Дії:

- Ввести розмір масиву: 100.

- Вибрати тип масиву.

- Натиснути кнопку "Генерувати масив".

- Натиснути кнопку "Сортування".

- Зафіксувати час виконання для кожного алгоритму

Повторити тест 5 разів для кожного типу даних, щоб отримати середнє значення часу.

Очікувані результати:

- Усі алгоритми завершують сортування без помилок.

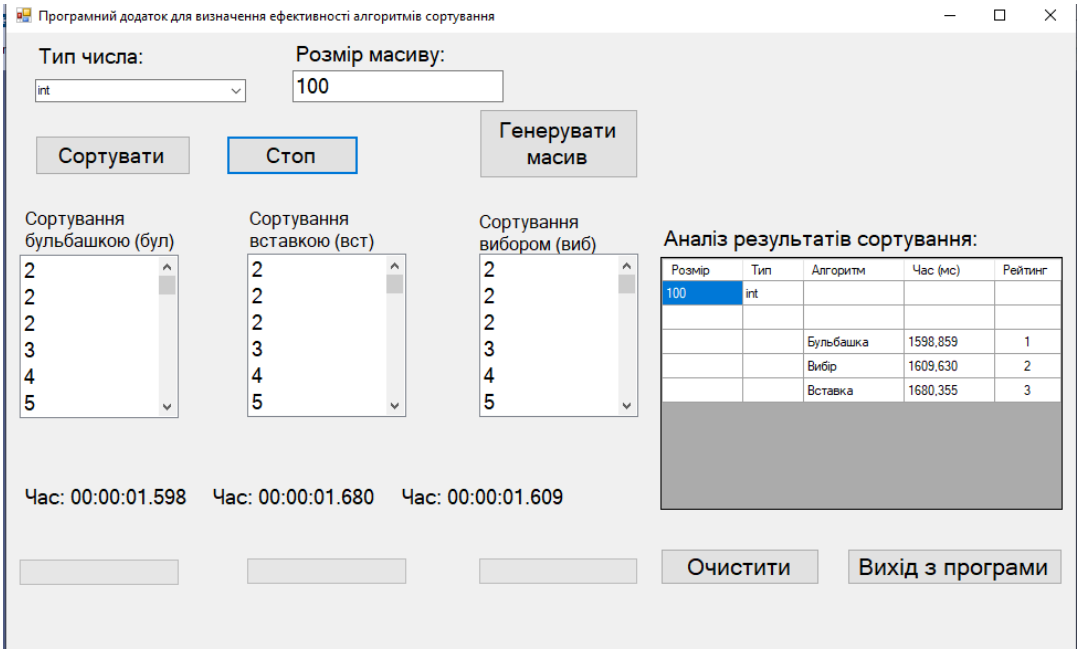

Рис. 4.7. Тестування на малих розміру масиву для int. Кращий метод бульбашки

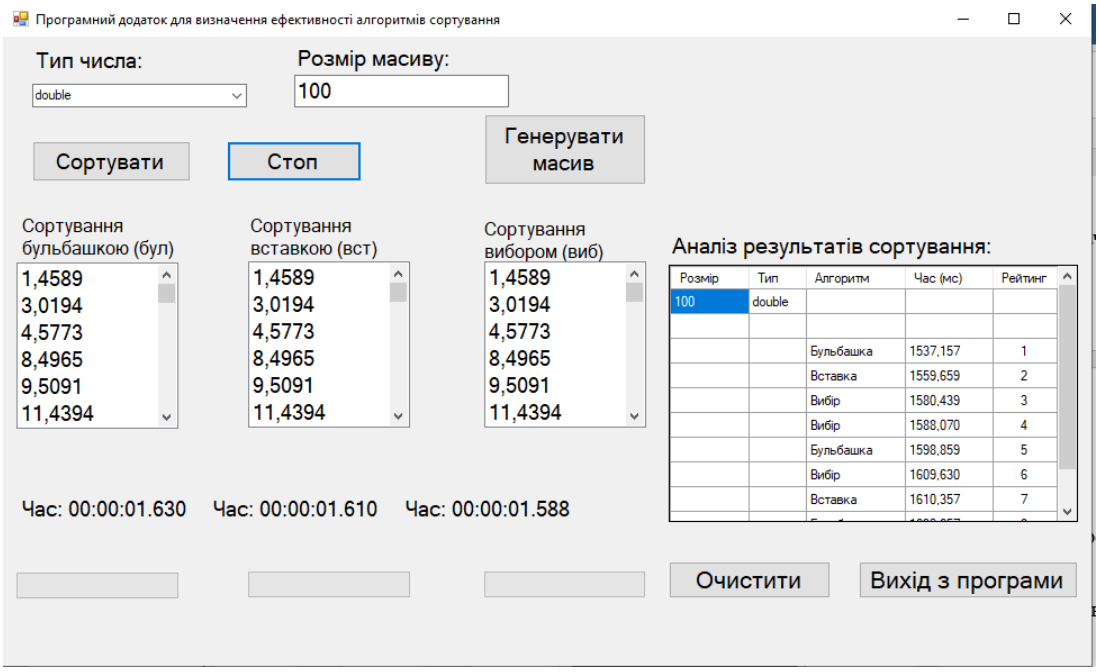

Рис. 4.8. Тестування на малих розміру масиву для double. Кращий метод вибором

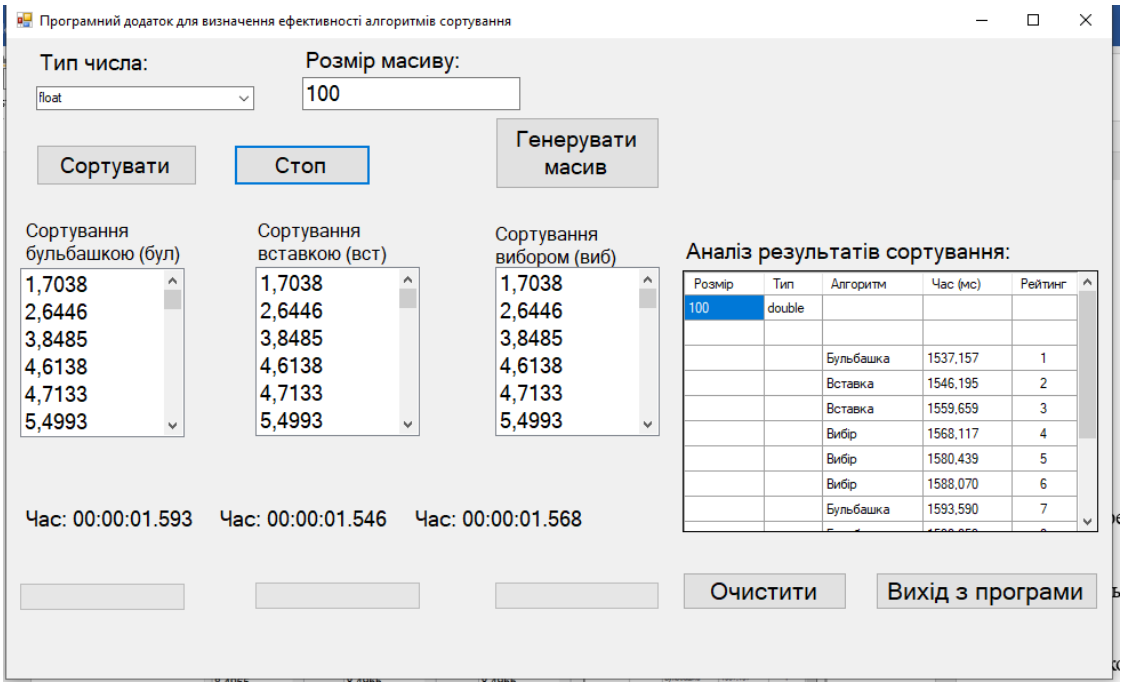

Рис. 4.9. Тестування на малих розміру масиву для float. Кращий метод вставкою.

Сценарій 2: Тестування на середніх масивах

Мета: Оцінити продуктивність алгоритмів на середніх наборах даних, де накладні витрати багатопотоковості можуть бути значними.

Вхідні дані:

Розмір масиву: 100 елементів.

Типи масивів: int, double, float.

Тип даних: Випадкові згенеровані числа.

Дії:

- Ввести розмір масиву: 100.

- Вибрати тип масиву.

- Натиснути кнопку "Генерувати масив".

- Натиснути кнопку "Сортування".

- Зафіксувати час виконання для кожного алгоритму

Повторити тест 5 разів для кожного типу даних, щоб отримати середнє значення часу.

Очікувані результати:

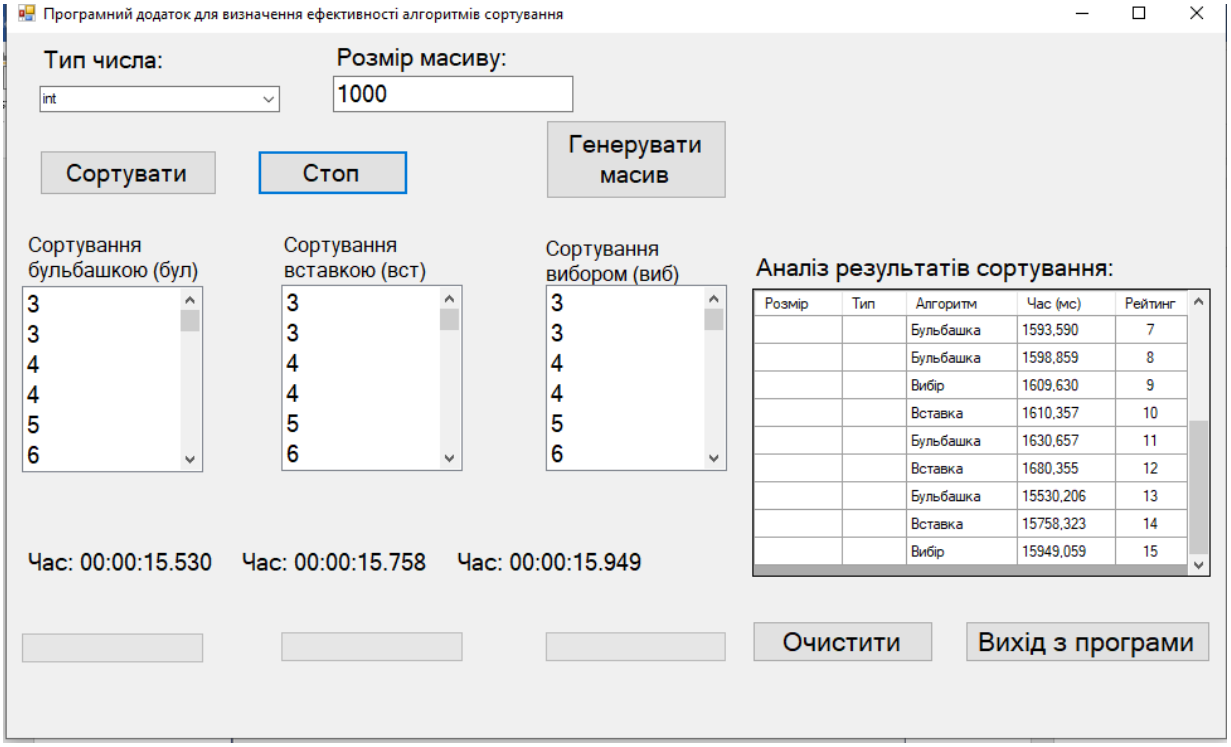

Рис. 4.10. Тестування на середньому розмірі масиву для int. Кращий метод бульбашки

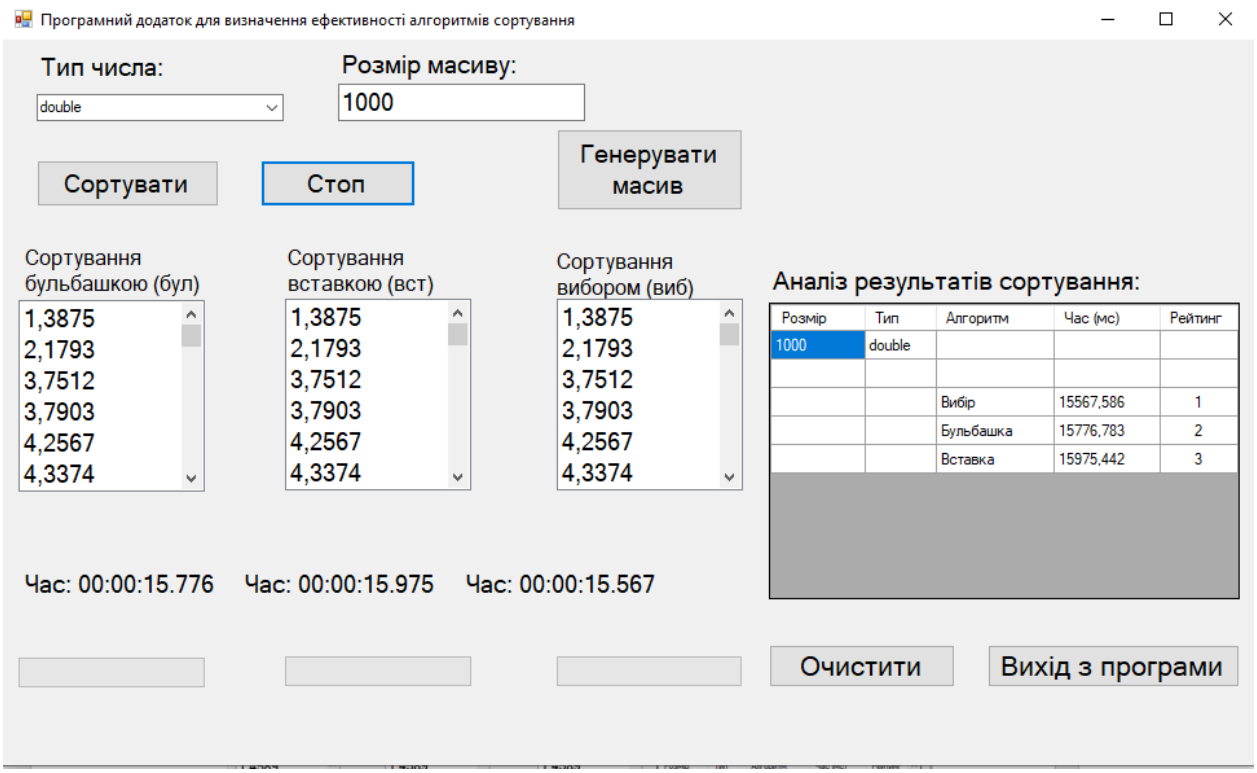

Рис. 4.11. Тестування на середньому розмірі масиву для double. Кращий метод вибором

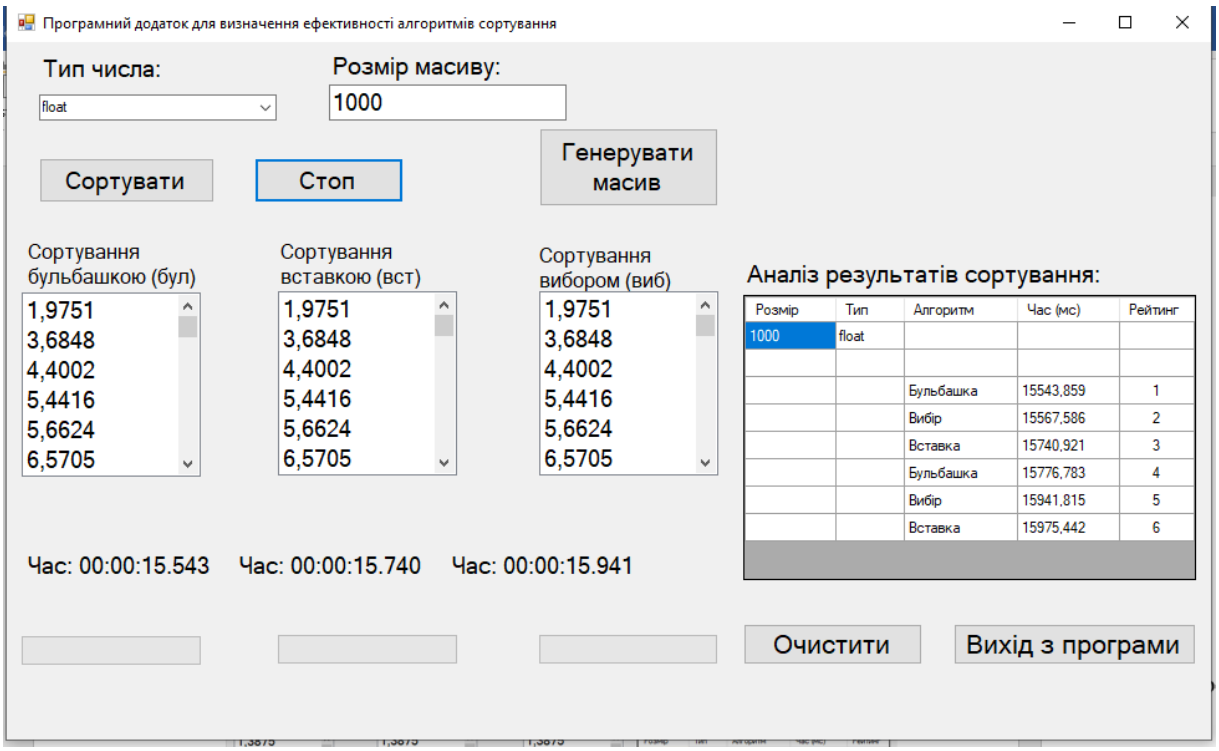

Рис. 4.12. Тестування на середньому розмірі масиву для float. Кращий метод бульбашка.

Сценарій 3: Тестування на великих масивах

Мета: Оцінити масштабування алгоритмів на великих наборах даних, де теоретична складність ($O(n^2)$ для всіх трьох алгоритмів) проявляється чіткіше.

Вхідні дані:

Розмір масиву: 10000 елементів.

Типи масивів: int, double, float.

Тип даних: Випадкові згенеровані числа.

Дії:

- Ввести розмір масиву: 100.
- Вибрати тип масиву.
- Натиснути кнопку "Генерувати масив".
- Натиснути кнопку "Сортування".
- Зафіксувати час виконання для кожного алгоритму

Повторити тест 3 рази для кожного типу даних.

Очікувані результати:

Алгоритми завершують сортування, але час виконання значно зростає через квадратичну складність.

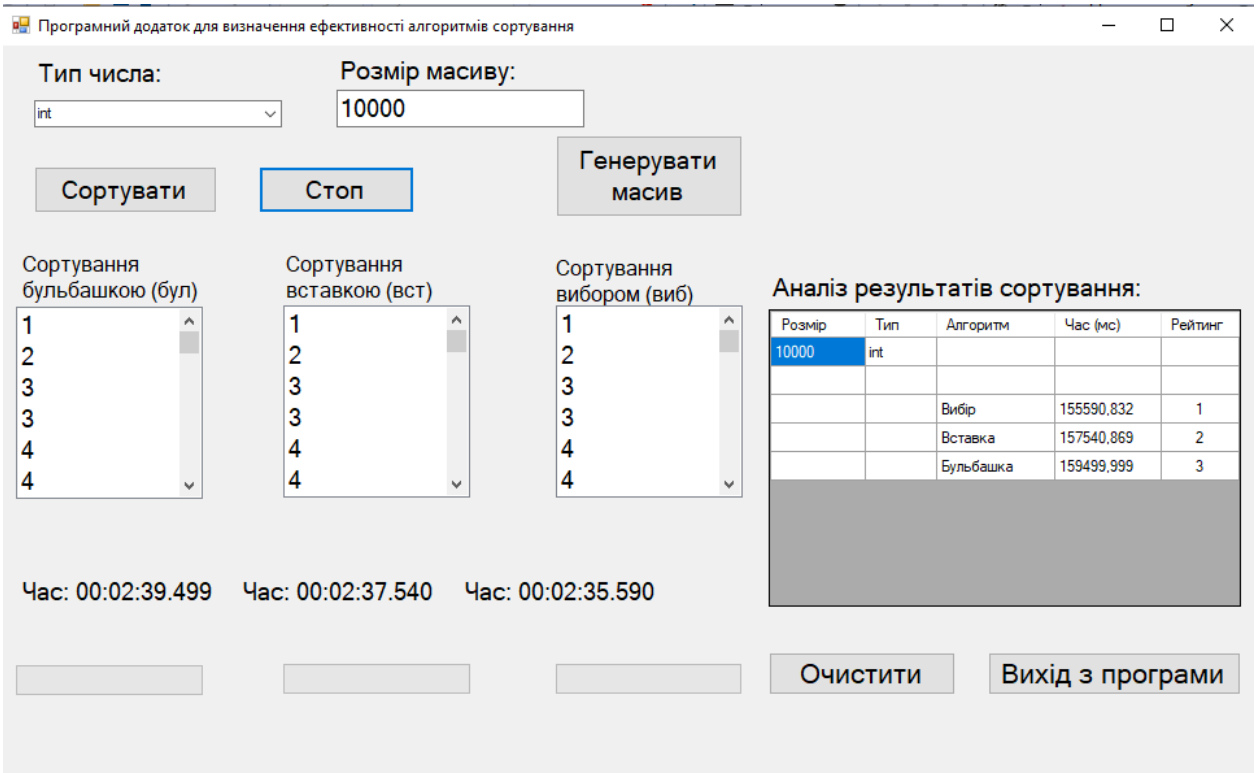

Рис. 4.13. Тестування на великому розмірі масиву для int. Кращий метод вибором.

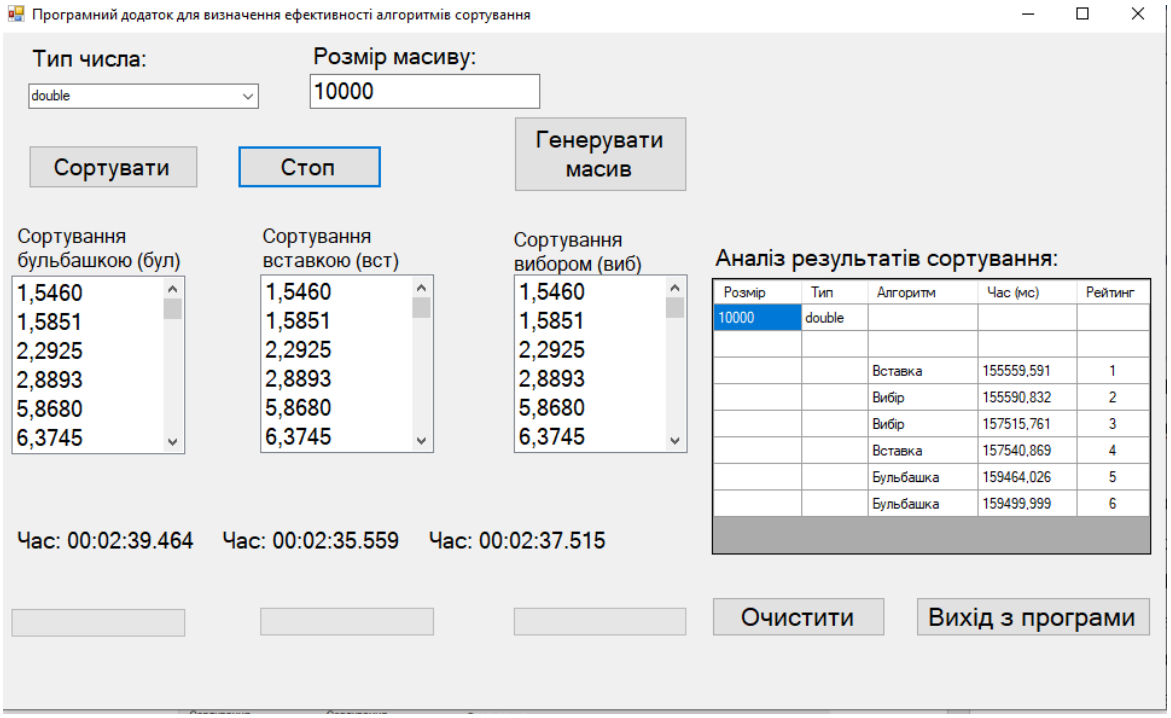

Рис. 4.14. Тестування на великому розмірі масиву для double. Кращий метод вставкою.

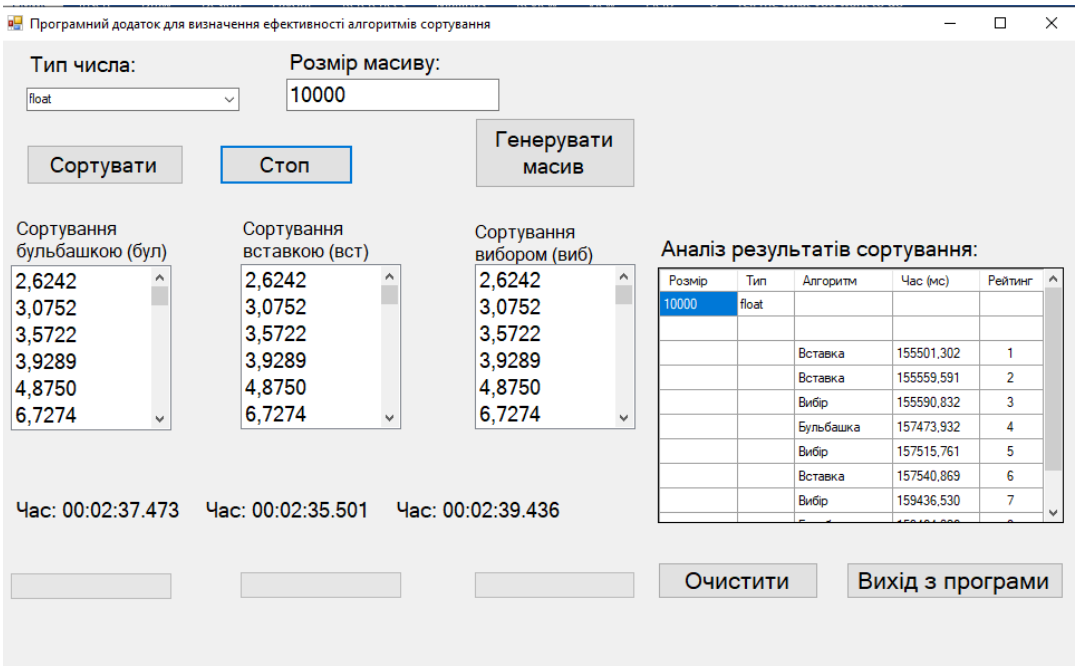

Рис. 4.15. Тестування на великому розмірі масиву для float. Кращий метод вставкою.

4.2.2. Аналіз результатів тестування

Важливий моментом виявляється, що метод сортування бульбашками добре працює з масивами типу int, тоді з даними типу double краще працює метод вибору. З великими розмірами масивів в середньому краще працює метод вставками

Сортування вставками є найефективнішим для великих розмірів масивів. Усі алгоритми мають квадратичну складність, що обмежує їх використання для великих масивів. Багатопотоковість ефективна для демонстрації, але додає накладні витрати на малих масивах.

Висновки до розділу 4

Тестування показало, що всі три алгоритми мають квадратичну складність ($O(n^2)$), що призводить до значного зростання часу виконання на великих масивах (10,000 елементів).

Сортування вставками виявилося найефективнішим для великих масивів, що може бути пов’язано з меншою кількістю обмінів на частково відсортованих даних.

Сортування бульбашкою показало кращі результати для масивів типу int на малих і середніх розмірах, тоді як сортування вибором було ефективнішим для типу double. Для типу float результати варіюються залежно від розміру масиву.

ЗАГАЛЬНІ ВИСНОВКИ

У першому розділі наданий огляд основних концепцій структур даних та алгоритмів сортування. Розглядаються класифікації структур даних (примітивні та непримітивні, лінійні та нелінійні) та їхні приклади. Описуються ключові аспекти алгоритмів сортування, такі як їхня ефективність, класифікація (внутрішнє та зовнішнє сортування), стійкість та обчислювальна складність.

Створено технічне завдання (ТЗ) для розробки настільного програмного додатку з аналізу ефективності алгоритмів сортування є документом, який детально описує цілі, вимоги, технічні аспекти та етапи реалізації проєкту. Воно базується на попередньо розглянутих функціональних і нефункціональних вимогах.

Розроблений програмний продукт є навчально-демонстраційною програмою, розробленою для аналізу ефективності трьох базових алгоритмів сортування: бульбашкою, вставками та вибором. Він дозволяє користувачам генерувати масиви чисел типів int, double і float, виконувати сортування в багатопотоковому режимі за допомогою багатопоточності і порівнювати продуктивність алгоритмів за часом виконання.

Тестування показало, що всі три алгоритми мають квадратичну складність ($O(n^2)$), що призводить до значного зростання часу виконання на великих масивах (10,000 елементів).

Сортування вставками виявилося найефективнішим для великих масивів, що може бути пов’язано з меншою кількістю обмінів на частково відсортованих даних.

Сортування бульбашкою показало кращі результати для масивів типу int на малих і середніх розмірах, тоді як сортування вибором було ефективнішим для типу double. Для типу float результати варіюються залежно від розміру

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Brown, Michael A. "Sorting Algorithms and Their Impact on User Experience." Journal of Information Processing and Management, vol. 30, no. 4, 2023, pp. 543-556.

2. Demuth, H. Electronic Data Sorting. PhD thesis, Stanford University, 1956.

3. Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03293-7.

4. Goodrich, Michael T.; Tamassia, Roberto (2002). "4.5 Bucket-Sort and Radix-Sort". *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons. pp. 241–243.

5. Y. Han. *Deterministic sorting in time and linear space*. Proceedings of the thirty-fourth annual ACM symposium on Theory of computing, Montreal, Quebec, Canada, 2002,p.602-608.

6. M. Thorup. *Randomized Sorting in Time and Linear Space Using Addition, Shift, and Bit-wise Boolean Operations*.

7. Journal of Algorithms, Volume 42, Number 2, February 2002, pp. 205-230(26)

8. https://big-o-calculator.vercel.app/big-o

9. Бублик В.В. Об’єктно-орієнтоване програмування: [Підручник] / В.В. Бублик. – К.: ІТкнига, 2015. – 624 с.: іл.

10. R. Stephens, Essential Algorithms. A Practical Approach to Computer Algorithms Using Python and C#, 2nd Edition ред., Indianapolis: John Wiley & Sons, Inc., 2019, p. 782.

11. Алгоритми та структури даних (комп'ютерний практикум): [Електронний ресурс] : навч. посіб. для студ. спеціальності 151 «Автоматизація та комп’ютерноінтегровані технології» / Укладач: Ю. Є. Грудзинський; КПІ ім. Ігоря Сікорського. - Електронні текстові дані (1 файл: 4,8 МБайт). – Київ : КПІ ім. Ігоря Сікорського,2022. – 100 с.

12. Пилипчук В.В., Іванюк П.О. Проектування алгоритмів та їх аналіз. Львів: ЛНУ ім. І. Франка, 2019. 312 с.

13. Тарасенко О.М. Чисельні методи та їх застосування в інформатиці. Дніпро: НМетАУ, 2021. 288 с.

14. Бородкіна І. Інженерія програмного забезпечення: навч. посібник. - К.: Центр учбової літератури, 2021. - 204 с.

15. Методичні вказівки до самостійної роботи студентів з проектування UML діаграм в ході виконання курсових робіт з дисципліни «Об'єктноорієнтоване програмування» для студентів спеціальності 121 − «Інженерія програмного забезпечення» [Електронний ресурс] / Уклад. Д. І. Катєльніков, О. О. Дудник, А. В. Денисюк – Вінниця : ВНТУ, 2021. – 28 с.

16. Івохін, Є.В. Махно, М.Ф. Піскунов. О.Г. Розробка додатків засобами мови програмування C#: Навч.-метод. посібник для проведення лабораторних робіт для студентів вищих навчальних закладів спеціальності «системний аналіз» /Є.В.Івохін, М.Ф.Махно, О.Г.Піскунов. – К.: Видавничо-поліграфічний центр "Київський університет", 2021. - 100 с.