Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Дослідження продуктивності методів та алгоритмів сортування за різних умов використання
(тема)

Виконав:

студент II курсу, групи СПм-23-1

Бєлицький Д.М.
(прізвище, ініціали)

Спеціальність

123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма

Системне програмування
(повна назва освітньої програми)

Керівник: ст. викл. Єрьоміна Н.С.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. Кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Бєлицькому Данилу Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження продуктивності методів та алгоритмів сортування за різних умов використання

затверджена наказом по університету від " 22 " листопада 2024 р. № 1236 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 20 січня 2025 р.

3. Вхідні дані до роботи
NodeJS
WebStorm
Visual Studio
Windows 10
Комп'ютер
Масив з даними

4. Перелік питань, що потрібно опрацювати у роботі
Вибір алгоритмів та методів для тестування
Розробка алгоритму створення випадкових масивів
Розробка структури та алгоритмів додатку
Аналіз отриманих результатів
Налаштування системи для відтворювання результатів тестів
Критерії та способи виміру продуктивності алгоритмів та методів

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів)

16 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Ознайомлення з літературними джерелами	23.07.2024 - 26.08.2024	
2	Аналіз та вибір методів вирішення задачі	27.08.2024 - 10.09.2024	
3	Розробка алгоритмів для вирішення задач	11.09.2024 - 29.09.2024	
4	Тестування алгоритмів та методів сортування	29.09.2024 - 02.11.2024	
5	Оформлення кваліфікаційної роботи	02.11.2024 - 18.01.2025	
6	Подання кваліфікаційної роботи	08.01.2025 - 11.01.2025	
	та її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	13.01.2025 – 20.01.2025	
	вання		

Дата видачі завдання 25 листопада 2024 р.

Студент _____
(підпис)

Керівник роботи _____ ст. викл. Єрьоміна Н.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 103 с., 23 р., 8 табл., 1 дод., 15 джерел.

СОРТУВАННЯ, ШВИДКІСТЬ ПАМ’ЯТІ, ПРОДУКТИВНІСТЬ АЛГОРИТМІВ, ДОСЛІДЖЕННЯ, ПОРІВНЯННЯ ПРОДУКТИВНОСТІ, АЛГОРИТМИ СОРТУВАННЯ, МЕТОДИ СОРТУВАННЯ.

Мета дослідження полягає у визначенні продуктивності методів та алгоритмів сортування при різних умовах використання.

Об'єктом дослідження є методи комплексного тестування алгоритмів сортування.

Предметом дослідження виступає розробка сценарію комплексного тестування для дослідження продуктивності алгоритмів сортування.

Гіпотеза дослідження передбачає можливість створення на основі існуючих методів сортування сценарію комплексного тестування алгоритмів сортування для дослідження продуктивності та ефективності.

Положення дослідження включають розробку теоретичних концепцій аналізу та порівняння методів сортування, що дозволяють здійснювати вибір найбільш відповідного методу для практичного використання.

В рамках дослідження буде проведено всебічний аналіз продуктивності різних алгоритмів сортування, таких як сортування бульбашкою, швидке сортування, сортування злиттям тощо, на різних мовах програмування. Окрему увагу буде приділено впливу частоти пам'яті та інших умов, таких як розбиття сортування на різні потоки. Проведені експерименти дозволять не тільки визначити найефективніші методи для конкретних умов, але й виявити можливі слабкі місця та шляхи оптимізації.

ABSTRACT

Master’s thesis: 103 pages, 23 figures, 8 tables, 1 appendices, 15 sources.

SORTING, MEMORY SPEED, PERFORMANCE OF ALGORITHMS, RESEARCH, COMPARISON OF PERFORMANCE, SORTING ALGORITHMS, SORTING METHODS.

The purpose of the study is to determine the performance of sorting methods and algorithms under different conditions of use.

The object of research is methods of complex testing of sorting algorithms.

The subject of the research is the development of a complex testing scenario for researching the performance and reliability of sorting algorithms.

The research hypothesis provides for the possibility of creating, based on existing sorting methods, a comprehensive testing scenario of sorting algorithms for performance and efficiency research.

The provisions of the study include the development of theoretical concepts of analysis and comparison of sorting methods, which allow the selection of the most suitable method for practical use.

The research will comprehensively analyze the performance of various sorting algorithms such as bubble sort, quick sort, merge sort, etc., in different programming languages. Particular attention will be paid to the effect of memory frequency and other conditions, such as splitting the sort into different streams. The conducted experiments will allow not only to determine the most effective methods for specific conditions, but also to reveal possible weak points and ways of optimization.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ОЗП – оперативний запам'ятовуючий пристрій

ЦП – центральний процесор

ЗП – зовнішня пам'ять

JS - JavaScript

ООП - об'єктно-орієнтоване програмування

MHz - це одиниця вимірювання частоти, яка дорівнює одному мільйону герців (Hz)

R – масив з випадковими елементами

S – масив з випадковими елементами та сортованими послідовностями

ВСТУП

Швидкий розвиток інформаційних технологій ставить перед проектувальниками та розробниками програмного забезпечення завдання створення ефективних та надійних алгоритмів обробки даних. Однією з важливих задач у цій сфері є сортування, яке використовується в багатьох інформаційних системах і додатках. Вибір найефективнішого алгоритму сортування може суттєво вплинути на продуктивність програмного забезпечення. Сортування є ключовою операцією в багатьох галузях, включаючи бази даних, пошукові системи та алгоритмічну торгівлю.

На даний момент немає алгоритму сортування який може задовольняти всім поставленим задачам, інакше кажучи бути ідеальним алгоритмом [1].

Тому вибір оптимального алгоритму сортування при різних умовах використання є важливою проблемою через велику кількість алгоритмів та різноманітні умови їх виконання на різних мовах програмування і апаратних платформах. Аналіз літератури показує, що більшість досліджень зосереджується на теоретичних аспектах алгоритмів сортування, тоді як практичні аспекти їх тестування в різних середовищах залишаються менш дослідженими. Зокрема, більшість робіт аналізують алгоритми на одній мові програмування або в одному середовищі, що не дає повної картини їх продуктивності в реальних умовах.

Сучасні інформаційні системи потребують високої швидкодії та ефективності обробки даних, що робить дослідження алгоритмів сортування особливо актуальним. Важливо враховувати, що різні алгоритми можуть показувати різну продуктивність залежно від мови програмування та апаратних умов. Наприклад, алгоритми сортування можуть працювати по-різному на процесорах з різною архітектурою, різною частотою пам'яті або кількістю доступних обчислювальних потоків.

Для проведення всебічного аналізу ефективності алгоритмів сортування вони також будуть тестуватися на частково відсортованих масивах даних. Це

дасть змогу визначити продуктивність алгоритмів у випадках, коли дані вже мають певну структуру впорядкованості, що часто зустрічається в реальних інформаційних системах. Частково відсортовані масиви можуть виникати, наприклад, у ситуаціях, коли дані оновлюються або додаються нові елементи до вже впорядкованих списків.

Це дослідження має на меті визначення продуктивності різних алгоритмів сортування при змінних умовах використання. У ході роботи буде реалізовано велика кількість алгоритмів сортування, таких як, наприклад, сортування бульбашкою, швидке сортування та сортування злиттям, на декількох мовах програмування.

Для тестування будуть використовуватися масиви даних різного розміру: малий (до 100 елементів), середній (до 10000 елементів) і великий (до 1000000 елементів). Додатково будуть проведені тести з використанням ОЗП (штучного зниження частоти та таймінгів) для оцінки впливу цього фактора на продуктивність алгоритмів.

Основні етапи дослідження включають:

- теоретичний аналіз алгоритмів сортування та вибір мов програмування для їх реалізації;

- розробка та проведення тестів на масивах даних різного розміру;

- аналіз впливу частоти ОЗП та інших апаратних умов на продуктивність алгоритмів;

- узагальнення результатів дослідження та формулювання висновків.

Теоретичний аналіз алгоритмів сортування включає детальне вивчення таких алгоритмів, як сортування бульбашкою, швидке сортування, сортування злиттям, сортування вибором, сортування вставками, сортування купою та інші. Кожен з цих алгоритмів має свої переваги та недоліки, які будуть враховані при проведенні тестів.

Розробка та проведення тестів включає написання програм на кількох мовах програмування, таких як C++, Java, Python та JavaScript. Кожна мова має свої особливості, що впливають на продуктивність алгоритмів. Наприклад,

C++ відома своєю високою швидкодією завдяки компіляції коду, тоді як Python має зручний синтаксис, але повільніший через інтерпретацію коду.

Аналіз впливу частоти пам'яті включає проведення тестів з використанням пам'яті різної швидкості. Це дозволить визначити, як зміна частоти ОЗП впливає на продуктивність алгоритмів сортування. Дослідження також включатиме оцінку впливу багатопоточності на ефективність сортування. Для цього будуть реалізовані алгоритми сортування, що можуть працювати в паралельних потоках, і проведені відповідні тести.

Узагальнення результатів дослідження включатиме аналіз отриманих даних та формулювання рекомендацій щодо вибору оптимальних алгоритмів сортування для різних умов. Це дозволить визначити найефективніші методи для конкретних задач і виявити можливі шляхи їх оптимізації.

Очікується, що результати цього дослідження будуть корисними для розробників програмного забезпечення, що працюють з великими обсягами даних, а також для фахівців, що займаються оптимізацією продуктивності інформаційних систем та хочуть обрати алгоритм на основі існуючих тестів для того щоб буду впевненими в те що алгоритм не тільки на бумазі має гарну продуктивність, а також переконую їх в реальному тестуванні. Це дозволить більш ефективно використовувати апаратні ресурси, забезпечити високу швидкодію обробки даних в різних умовах та прискорити виконання поставлених задач через економію часу на тестування.

Таким чином, дане дослідження спрямоване на визначення найбільш ефективних алгоритмів сортування при різних умовах використання, що включає реалізацію цих алгоритмів на декількох мовах програмування, тестування їх продуктивності на масивах даних різного розміру, а також аналіз впливу апаратних умов, таких як частота ОЗП та багатопоточність ЦП. Узагальнення отриманих результатів дозволить зробити висновки щодо вибору оптимальних методів сортування для різних типів задач та умов, що сприятиме підвищенню продуктивності та ефективності інформаційних систем.

1 АНАЛІЗ ДОСЛІДЖУВАНИХ АЛГОРИТМІВ ТА МЕТОДІВ СОРТУВАННЯ

1.1 Теоретична основа сортування

Алгоритми та методи сортування є основою багатьох обчислювальних задач та застосунків. Розуміння ключових концепцій та термінології допомагає ефективно вибирати та використовувати ці алгоритми. Далі ми розглянемо основні поняття, які є важливими для розуміння процесу сортування.

Стабільний алгоритм сортування зберігає відносний порядок елементів з однаковими значеннями. Це важливо, коли порядок елементів має значення для подальшої обробки даних. Наприклад, при сортуванні списку студентів за оцінками стабільний алгоритм забезпечить, що студенти з однаковими оцінками залишаться в тому ж порядку, в якому вони були спочатку.

Алгоритми та методи сортування можна розділити на порівняльні та непорівняльні. Порівняльні алгоритми, такі як швидке сортування і сортування злиттям, використовують порівняння елементів для впорядкування. Непорівняльні алгоритми, такі як сортування підрахунком та сортування за розрядами, використовують інші методи для сортування елементів, не залучаючи порівнянь [3].

Часова складність алгоритму вимірюється кількістю операцій, які він виконує відносно розміру вхідних даних, і зазвичай виражається в великому О-нотації. Наприклад, $O(n^2)$ означає, що кількість операцій пропорційна квадрату кількості елементів. Просторова складність визначає обсяг додаткової пам'яті, необхідної алгоритму для роботи.

Оцінка продуктивності алгоритму включає аналіз у трьох сценаріях а саме найкращому, середньому та найгіршому випадках. Найкращий випадок трапляється, коли вхідні дані вже відсортовані або майже відсортовані. Середній випадок представляє типові умови, а найгірший випадок — це

сценарій з найбільш несприятливими вхідними даними.

Адаптивні алгоритми сортування можуть підлаштовувати свою стратегію залежно від структури вхідних даних. Наприклад, сортування вставками може бути дуже ефективним для майже відсортованих даних, оскільки воно зменшує кількість необхідних переміщень.

Алгоритми та методи сортування можуть бути внутрішніми або зовнішніми. Внутрішні алгоритми, такі як сортування бульбашкою, працюють повністю в оперативній пам'яті. Зовнішні алгоритми, такі як зовнішнє сортування злиттям, призначені для роботи з великими обсягами даних, що не вміщуються в оперативній пам'яті, і виконують сортування завдяки використанню зовнішньої пам’яті.

Алгоритми сортування можуть бути реалізовані як рекурсивні або ітеративні процедури. Рекурсивні алгоритми, такі як швидке сортування і сортування злиттям, використовують підхід «розділяй і володарюй», тоді як ітеративні алгоритми, такі як сортування бульбашкою і сортування вибором, застосовують цикли для послідовного сортування елементів.

Алгоритми сортування знаходять широке застосування у різних галузях, таких як бази даних, пошукові системи, машинне навчання та обробка великих даних. Наприклад, швидке сортування є основою багатьох сучасних методів пошуку та індексації даних у базах даних.

Продуктивність алгоритмів сортування може значно змінюватися в залежності від апаратних умов. Наприклад, такі фактори, як частота процесора, швидкість оперативної пам'яті та кількість ядер, можуть впливати на швидкість виконання алгоритмів. Також слід враховувати використання багатопоточності та паралельних обчислень, які можуть суттєво підвищити ефективність сортування великих обсягів даних. Ці теоретичні основи є важливими для глибокого розуміння алгоритмів та методів сортування. Вони дозволяють визначити, який алгоритм або метод є найбільш придатним для поставленої задачі та умов в яких він буде використовуватися.

1.2 Методи сортування

1.2.1 Сортування Бульбашкою

Сортування бульбашкою (Bubble Sort) є одним із найпростіших методів сортування, який працює за принципом повторного проходження через список. Під час кожного проходу алгоритм порівнює кожну пару сусідніх елементів і обмінює їх місцями, якщо вони розташовані у неправильному порядку. Цей процес повторюється доти, доки жодних обмінів більше не потрібно, що означає, що список відсортовано. Даний алгоритм є стабільним.

Часова складність методу сортування бульбашкою залежить від кількості елементів у списку (n). У найгіршому і середньому випадках складність становить $O(n^2)$, оскільки кожен елемент може бути порівняний з кожним іншим елементом. У найкращому випадку, коли список вже відсортований, складність буде O(n), оскільки алгоритм виконує лише один прохід по списку. Просторова складність алгоритму сортування бульбашкою є O(1)[4].

Метод сортування бульбашкою має як переваги, так і недоліки. Його головними перевагами є простота реалізації та зрозумілість, що робить його зручним для навчання та початкового застосування. Він також показує гарну ефективність при сортуванні невеликих списків або коли дані вже майже відсортовані. Додатково, алгоритм можна оптимізувати якщо виявляється, що список уже впорядкований, що зменшує кількість необхідних проходів.

Однак, сортування бульбашкою має і значні недоліки. Воно не підходить для великих списків через високу часову складність яка досягає $O(n^2)$. Це призводить до великої кількості порівнянь та обмінів, що робить алгоритм значно повільнішим у порівнянні з більш ефективними методами, такими як швидке сортування або сортування злиттям. Через свою реалізацію алгоритм не доцільно робити багатопоточним хоча це і можливо зробити.

На рисунку 1.1 наведена блок-схема методу сортування бульбашкою.

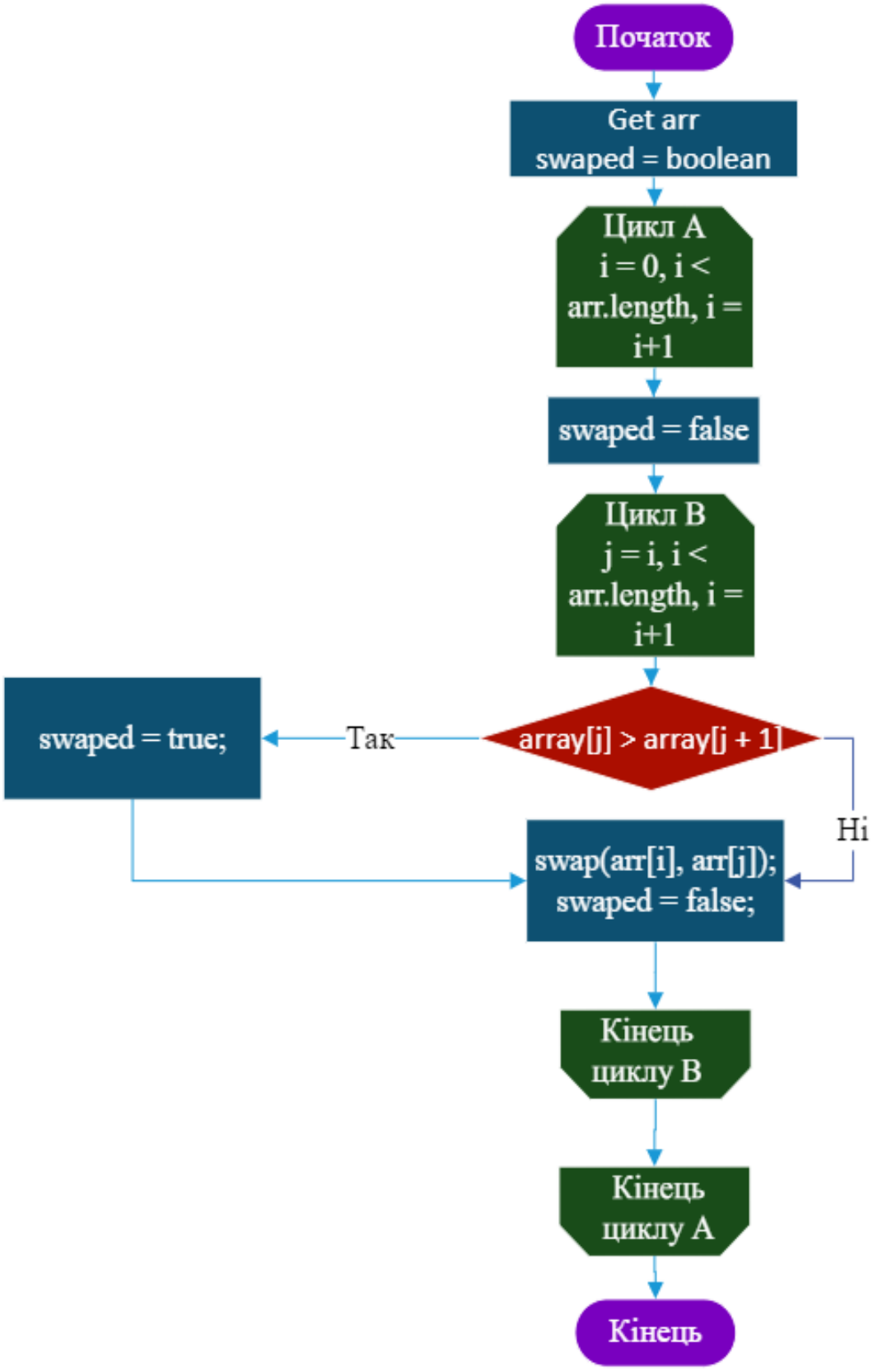

Рисунок 1.1 – Блок-схема методу сортування бульбашкою

1.2.2 Сортування Шейкером

Метод сортування шейкером (Cocktail Sort), відомий також як двонаправлене бульбашкове сортування. Він є не принципово новим методом сортування а лише модифікацією для сортування бульбашкою. Він зберігає простоту і зрозумілість бульбашкового сортування, але суттєво підвищує його ефективність завдяки проходженню масиву в обох напрямках.

Існують також інші варіанти та модифікації методу сортування, які намагаються покращити ефективність бульбашкового сортування, але шейкерне сортування залишається одним із найпоширеніших і найбільш зрозумілих. Часова складність методу сортування шейкером у найгіршому та середньому випадках складність становить $O(n^2)$, оскільки кожен елемент може бути порівняний з кожним іншим елементом [5].

У найкращому випадку, коли список вже відсортований або майже відсортований, складність буде O(n), оскільки метод виконує лише один чи кілька проходів по списку. Просторова складність методу сортування шейкером є O(1). Даний метод сортування є стабільним.

Метод сортування шейкером має кілька переваг і недоліків. Його основними перевагами є простота реалізації та зрозумілість. Також, він ефективніший за класичне бульбашкове сортування завдяки подвійним проходам, що зменшують кількість необхідних ітерацій для сортування списку.

Однак, як і бульбашкове сортування, метод алгоритм шейкером не підходить для великих списків через високу часову складність $O(n^2)$ у найгіршому випадку. Це призводить до великої кількості порівнянь та обмінів, що робить його менш ефективним у порівнянні з іншими алгоритмами та методами сортування, такими як швидке сортування або сортування злиттям.

Метод дуже схожий на той що наведено на рисунку 1.1, однак він складається з 1 циклу while який містить практично однакові 2 цикли for. На рисунку 1.2 наведено блок-схему алгоритму сортування шейкером.

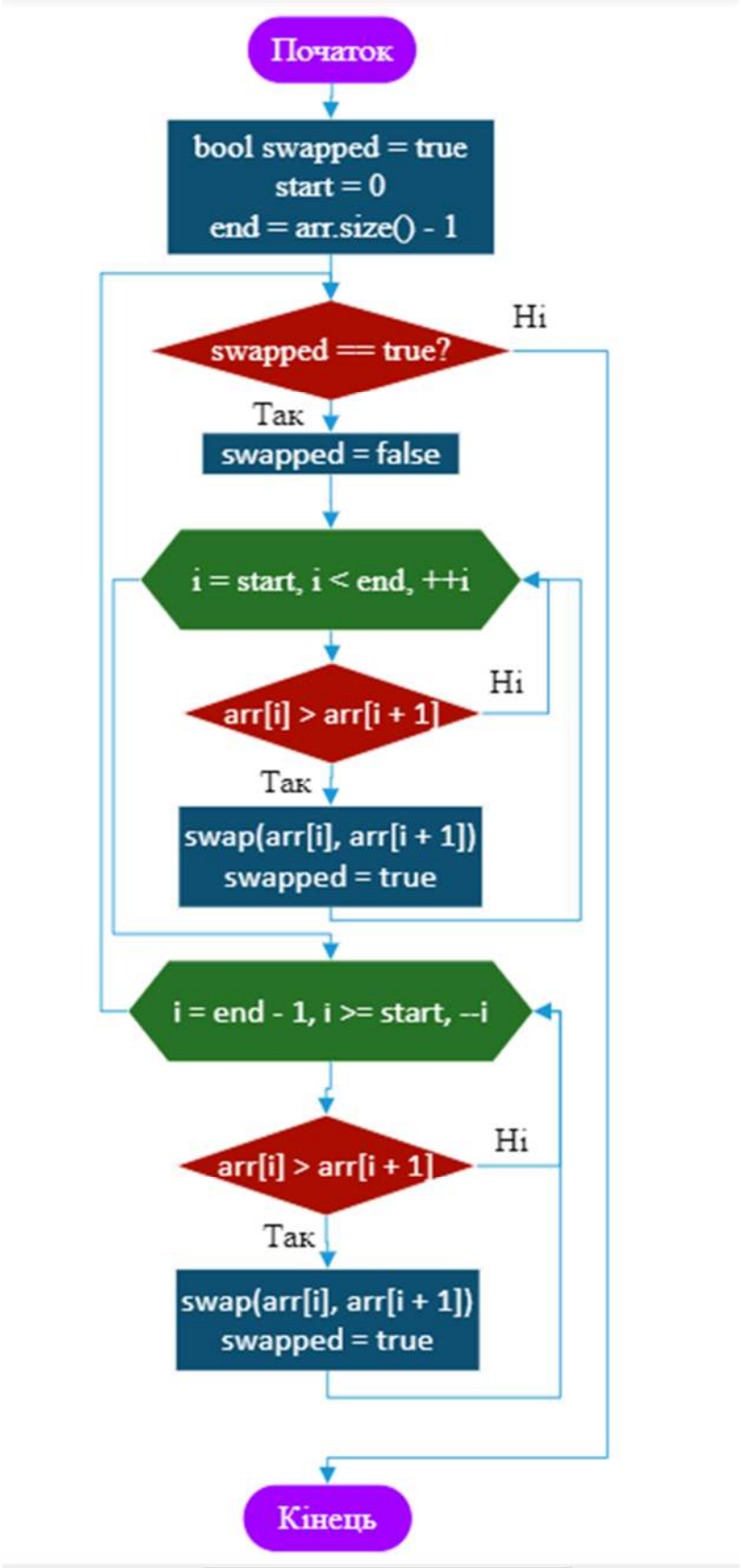

Рисунок 1.2 – Блок-схема алгоритму сортування шейкером

1.2.3 Сортування вставками

Сортування вставками (Insertion Sort) — це простий та інтуїтивно зрозумілий метод сортування, який добре підходить для невеликих масивів або масивів, що майже відсортовані. Суть методу полягає у поступовому «вставленні» кожного елемента у його правильне місце у вже відсортованій частині масиву.

Сортування вставками працює наступним чином: масив ділиться на дві частини — відсортовану і невідсортовану. Спочатку відсортована частина складається лише з першого елемента. Потім кожен наступний елемент з невідсортованої частини вставляється у відповідне місце у відсортованій частині, доки весь масив не буде відсортовано.

Метод сортування вставками має кілька переваг. Він простий у реалізації, добре працює на невеликих масивах і з майже відсортованими даними. Крім того, він є стабільним, тобто зберігає відносний порядок елементів з однаковими значеннями. Однак, через високу часову складність, він не підходить для сортування великих масивів даних.

Алгоритм сортування вставками є ефективним для невеликих масивів і даних, які майже відсортовані, завдяки простій реалізації та стабільності. Однак, для великих обсягів даних варто розглянути більш ефективні алгоритми сортування. Даний алгоритм має модифікацію а саме бінарне сортування вставками [6].

Він використовує алгоритм бінарного пошуку, алгоритм пошуку наведено як частину блок схеми на рисунку 1.3 у блоках виконання двох циклів, для визначення позиції куди повинен бути вставлений черговий елемент із невідсортованої частини масиву.

Після визначення позиції, всі елементи у відсортованій частині, які більші за новий елемент, зсуваються на одну позицію вправо для створення місця для вставки.

На рисунку 1.3 наведено блок схему методу з бінарним пошуком.

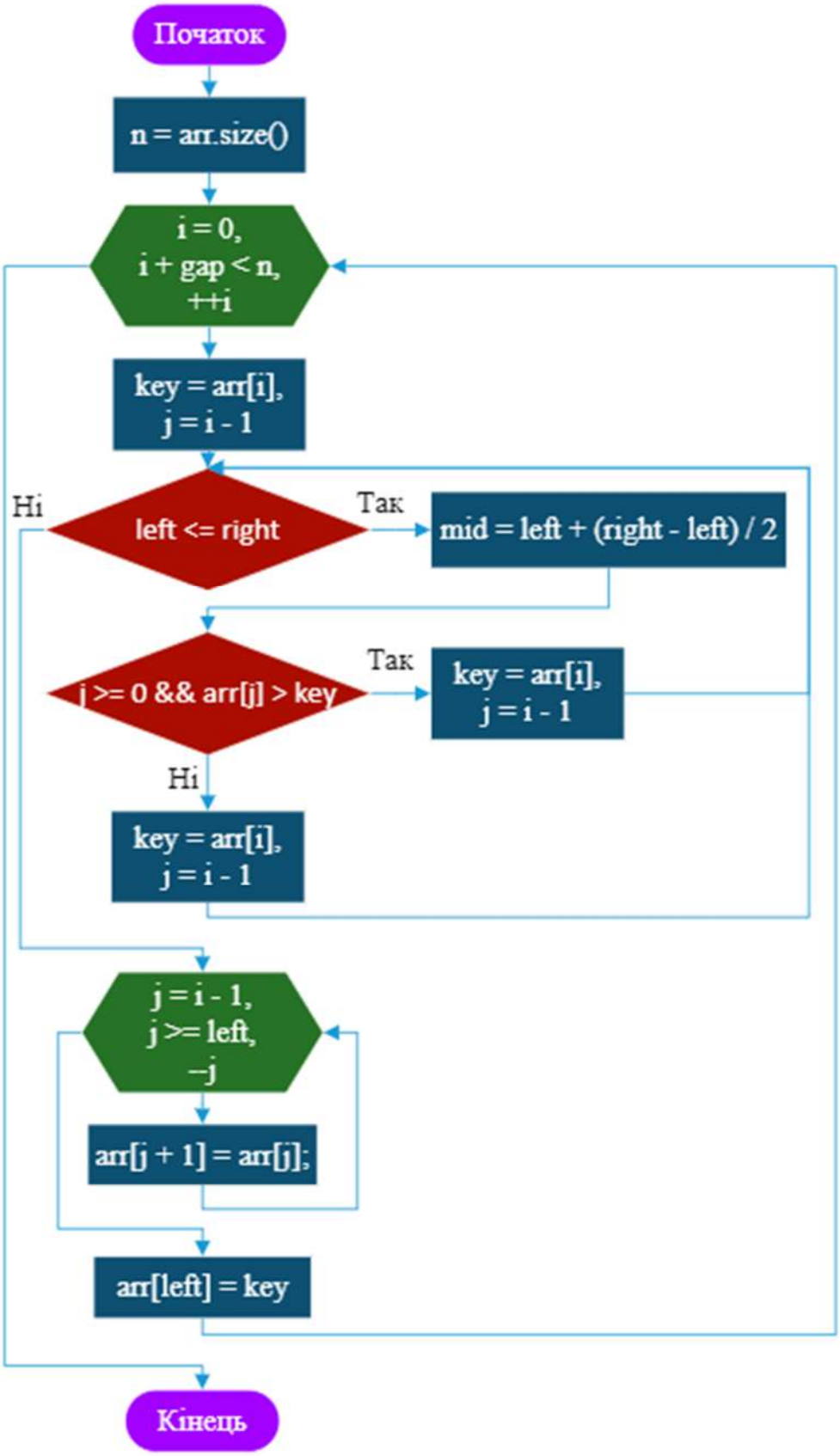

Рисунок 1.3 – Блок схема методу сортування вставкою з бінарним пошуком

1.2.4 Сортування Шелла

Сортування Шелла (Shell Sort) є вдосконаленою версією методу сортування вставками, що дозволяє зменшити кількість необхідних переміщень елементів за рахунок використання «щілин» (gaps). Це значно підвищує ефективність методу в порівнянні з класичним сортуванням вставками.

Часова складність методу сортування Шелла залежить від вибору послідовності щілин (gap sequence). У найгіршому випадку складність може бути $O(n^2)$, але зазвичай вона значно краща. Для оптимальних послідовностей щілин (наприклад, послідовність Кнута або Седжвіка) складність може наближатися до $O(n^{3/2})$. Просторова складність є O(1), оскільки метод не потребує додаткової пам'яті [7].

Сортування Шелла має декілька важливих переваг. Воно значно ефективніше за класичне сортування вставками для великих масивів завдяки використанню щілин, що зменшують кількість переміщень елементів. Метод також простий у реалізації та адаптації для різних послідовностей щілин. Недоліком є те, що вибір неправильної послідовності щілин може призвести до менш ефективного сортування.

Метод сортування Шелла працює шляхом розподілу масиву на підмасиви, визначені певною «щілиною» (gap). На кожному етапі щілина зменшується, і підмасиви сортуються за допомогою сортування вставками. Процес продовжується, доки щілина не зменшиться до 1, після чого виконується остаточне сортування вставками для всього масиву.

В тестах ми будемо використовувати модифіковану версію а саме сортування Шелла з використанням послідовності Циура так як вона має перевагу в продуктивності та має кращі показники порівнюючи з іншими послідовностями. На рисунку 1.4 наведено блок-схему методу сортування Шелла з використанням послідовності Циура.

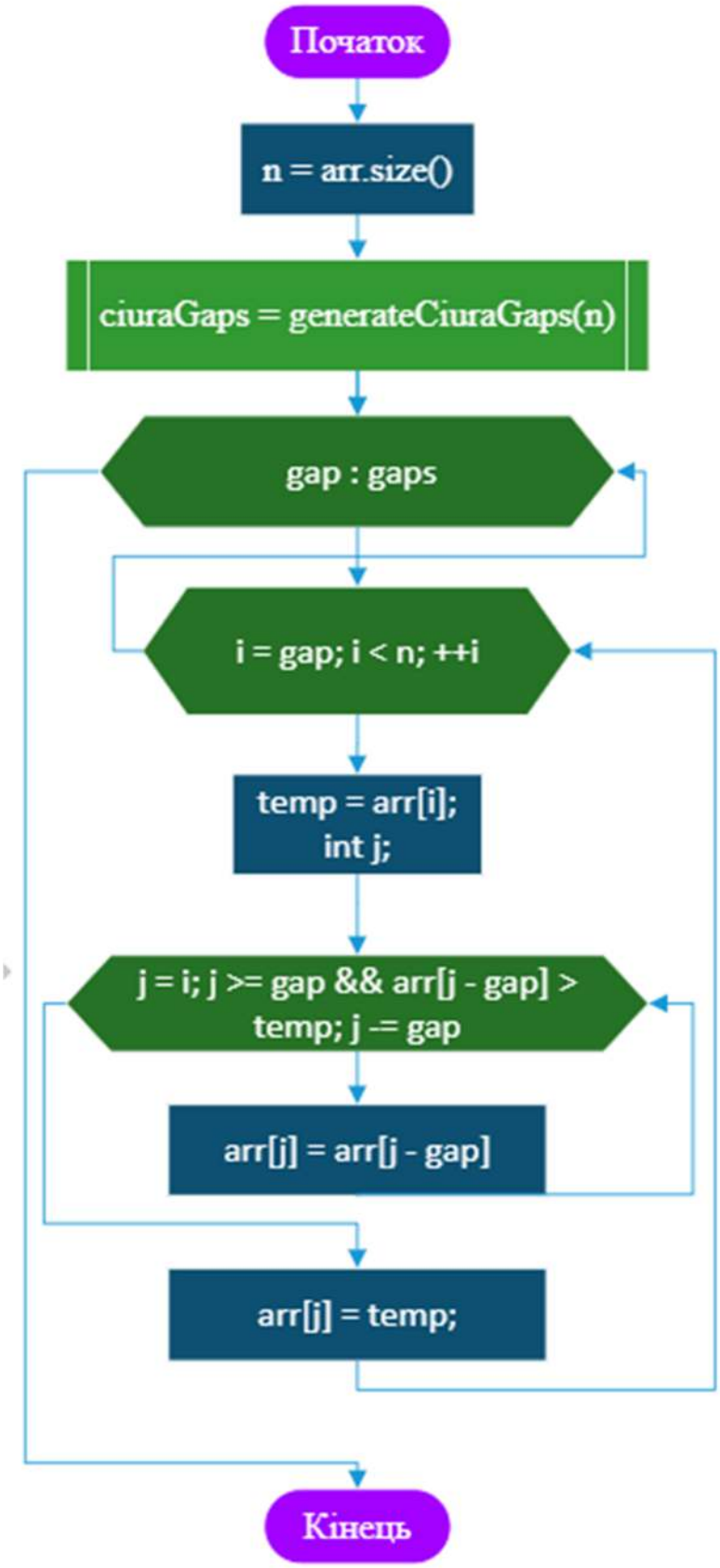

Рисунок 1.4 – Блок-схема методу сортування Шелла с послідовністю Циура

1.3 Алгоритми сортування

1.3.1 Сортування Гребінцем

Алгоритм сортування гребінцем (Comb Sort) є ще однією модифікацією класичного бульбашкового сортування, спрямованою на підвищення ефективності. Його основна ідея полягає у порівнянні та обміні елементів, розташованих на великій відстані один від одного, що дозволяє швидше переміщати великі елементи до їх кінцевих позицій. На рисунку 1.3 наведено блок-схему алгоритму сортування гребінцем.

Сортування гребінцем використовує поняття «розриву» (gap), який спочатку дорівнює розміру масиву і зменшується на кожному етапі сортування. Порівнюються та обмінюються елементи, які знаходяться на цій відстані. Процес триває, доки розрив не стане рівним 1 і масив не буде повністю відсортованим.

Середня часова складність алгоритму сортування гребінцем становить O(n log n), що значно краще, ніж $O(n^2)$ для класичного бульбашкового сортування. У найгіршому випадку складність може наближатися до $O(n^2)$, але такі випадки трапляються рідко. Просторова складність становить O(1), оскільки алгоритм сортує «на місці» і не потребує додаткової пам'яті.

Алгоритм сортування гребінцем має свої переваги. Він простий у реалізації та забезпечує кращу продуктивність у порівнянні з бульбашковим сортуванням завдяки зменшенню кількості малих переміщень. Проте він менш ефективним у порівнянні з іншими алгоритмами сортування, такими як швидке сортування або сортування злиттям, особливо при роботі з дуже великими масивами даних.

Алгоритм сортування гребінцем може бути налаштований шляхом зміни коефіцієнта «усадки» (shrink factor). Значення 1.3 було визначено як оптимальне експериментально, але інші значення можуть бути використані для досягнення кращої продуктивності в конкретних умовах. На рисунку 1.5

наведено блок схему алгоритму сортування гребінцем.

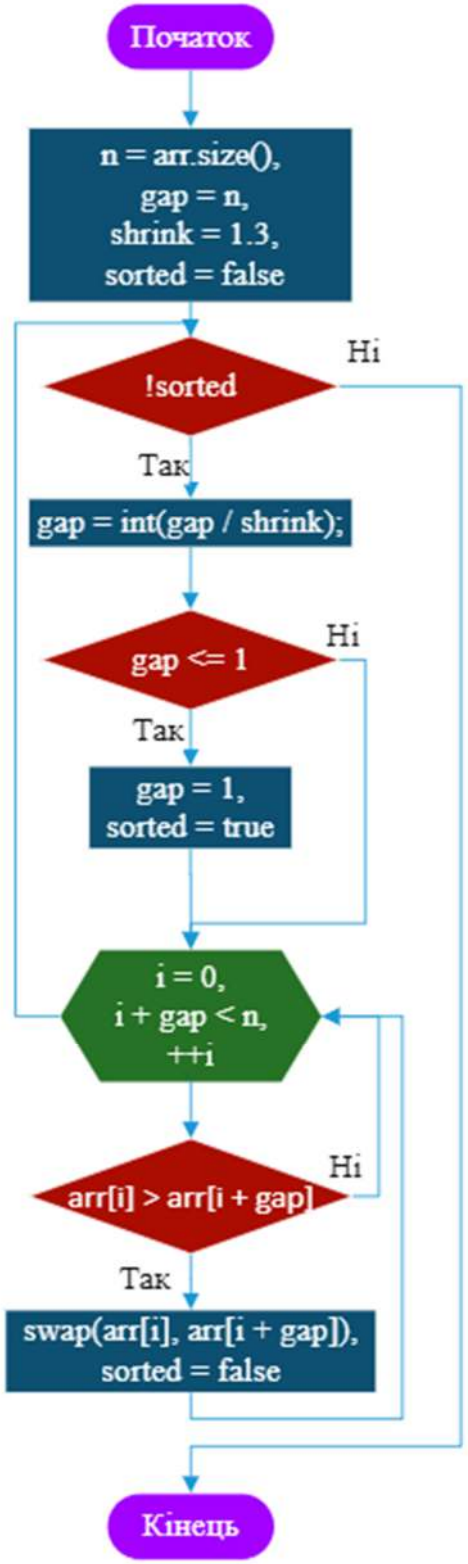

Рисунок 1.5 – Блок-схема алгоритму сортування гребінцем

1.3.2 Алгоритм сортування за допомогою двійкового дерева

Алгоритм сортування за допомогою двійкового дерева використовує бінарне дерево пошуку для впорядкування елементів. Кожен елемент масиву вставляється в дерево, після чого дерево обходиться симетричним методом, що дозволяє отримати відсортовані значення. Основна ідея полягає в тому, що при вставці кожен елемент порівнюється з вузлом дерева та розміщується у відповідній позиції: ліворуч — якщо менший, праворуч — якщо більший.

Алгоритм працює так: спочатку створюється порожнє дерево, потім послідовно вставляються елементи з масиву. Після того, як усі елементи додані, відбувається обхід дерева. Під час обходу значення виводяться в порядку зростання, забезпечуючи впорядкований результат.

Часова складність алгоритму у середньому випадку є логарифмічною O(n log n), що робить його ефективним для великих наборів даних. Але в найгіршому випадку, коли дерево стає незбалансованим, складність може зрости до $O(n^2)$. Це відбувається тоді, коли всі елементи додаються в уже відсортованому або зворотньо відсортованому порядку, що перетворює дерево в лінійну структуру. З точки зору пам’яті, алгоритм потребує додаткових ресурсів для зберігання бінарного дерева, що відповідає O(n) [8].

Серед основних переваг цього алгоритму виділяють можливість ефективної роботи з випадковими даними, де структура дерева залишається збалансованою. Однак значним недоліком є ризик виродження дерева, що негативно впливає на продуктивність. Використання дерев, таких як AVL-дерева або червоно-чорні дерева, дозволяє вирішити цю проблему і забезпечити стабільний час виконання.

Цей алгоритм є нестабільним, оскільки при його виконанні порядок однакових елементів може бути змінений. Модифікації з використання самобалансуючих дерев, можуть покращити ефективність у найгірших випадках. На рисунку 1.6 наведено блок схему алгоритму сортування за допомогою двійкового дерева.

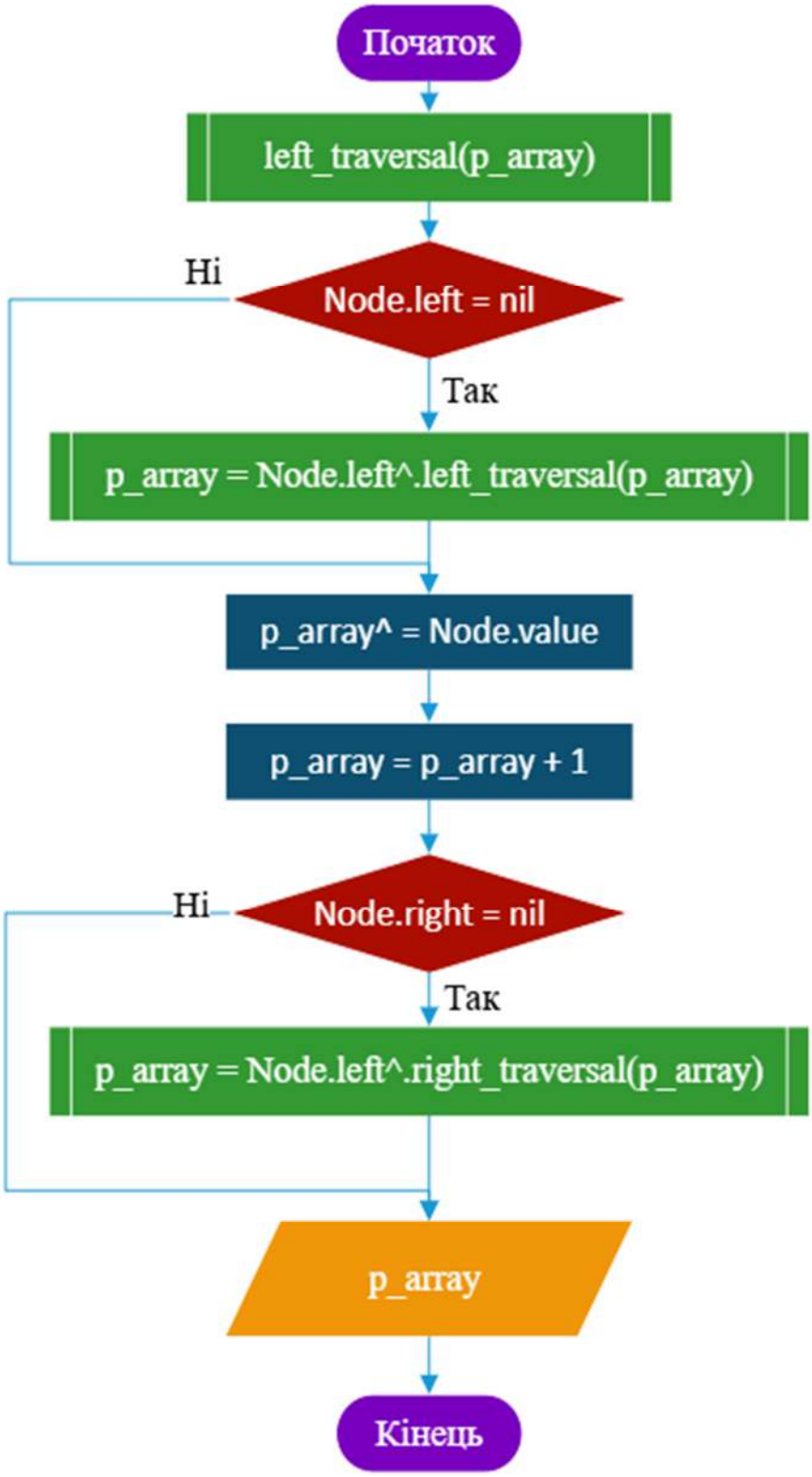

Рисунок 1.6 – Блок-схема алгоритму сортування за допомогою двійкового дерева.

1.3.3 Сортування злиттям

Сортування (Merge Sort) є потужним алгоритмом, що використовує метод розділяй і володарюй. Алгоритму спочатку потрібно розділити масив на дві половини після чого він сортує кожну з них окремо за допомогою рекурсивних викликів, а потім зливає відсортовані половини в один цілісний відсортований масив.

Часова складність сортування злиттям у найгіршому, середньому та найкращому випадках становить O(n log n), де n — кількість елементів у масиві. Просторова складність становить O(n), оскільки потрібна додаткова пам'ять для тимчасового зберігання підмасивів під час злиття.

Сортування злиттям має низку переваг. Це стабільний алгоритм, який зберігає відносний порядок елементів з однаковими значеннями. Він ефективно обробляє великі масиви даних і гарантує часову складність O(n log n) незалежно від випадку. Основний недолік сортування злиттям полягає у вимозі додаткової пам'яті O(n) для зберігання тимчасових підмасивів, що може бути проблематичним при роботі з дуже великими масивами даних.

Існують варіації сортування злиттям, такі як сортування природним злиттям, де підмасиви визначаються природним чином як уже відсортовані послідовності у початковому масиві.

Це може зменшити кількість злиттів і покращити продуктивність у певних випадках але якщо масив має погану комбінацію елементів то він буде програвати в продуктивності. Тому дану модифікацію не буде розглядатися [9].

Сортування злиттям підходить для зовнішнього сортування, оскільки його структура дозволяє ефективно працювати з даними, що не вміщуються в оперативну пам'ять. Однак дана реалізація підходить не для кожного випадку так як для неї потрібен доступ до роботи з файлами.

На рисунку 1.7 наведено блок-схему алгоритму сортування злиттям логіка якого загалом пов’язана з рекурсією.

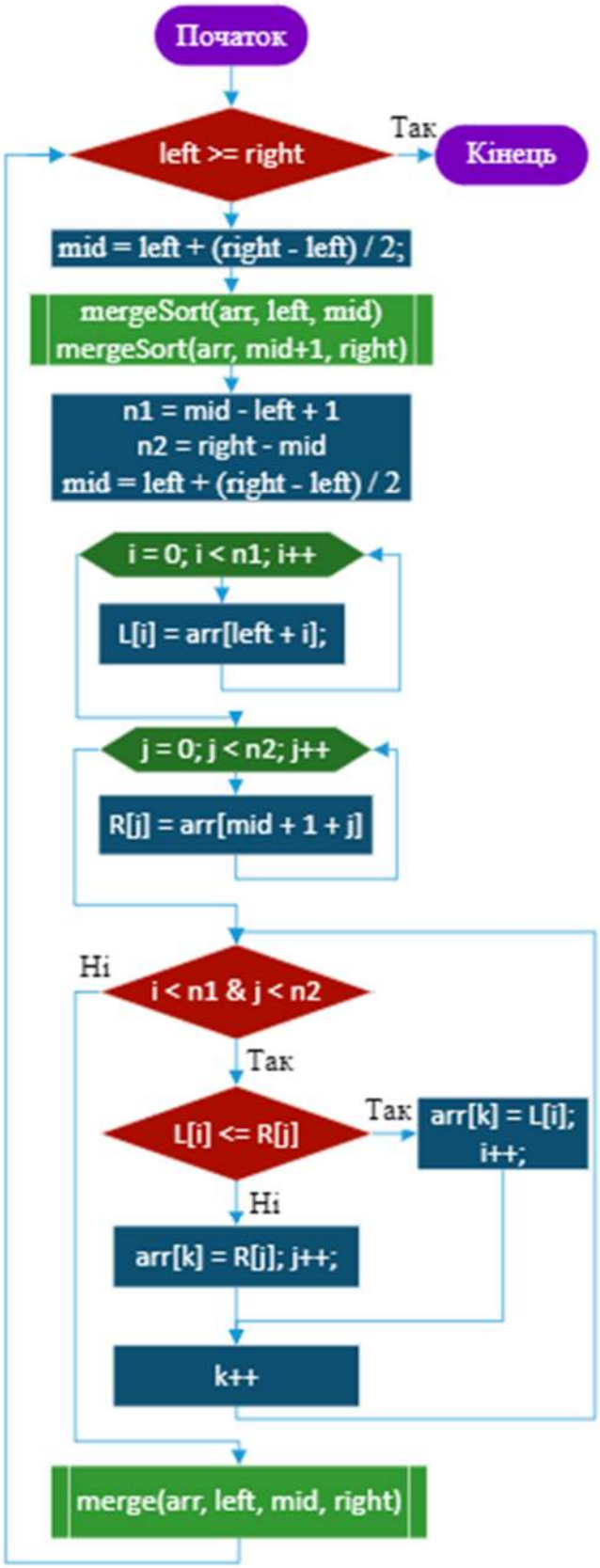

Рисунок 1.7 – Блок-схема алгоритму сортування злиттям

1.3.4 Сортування Timsort

Timsort — це адаптивний алгоритм сортування, який поєднує сортування вставками і сортування злиттям. Він розроблений для практичного використання і є стандартним алгоритмом сортування в багатьох мовах програмування, таких як Python і Java.

Часова складність Timsort у найгіршому, середньому та найкращому випадках становить O(n log n), де n — кількість елементів у масиві. Просторова складність становить O(n), оскільки потрібна додаткова пам'ять для зберігання підмасивів під час злиття. Адаптивність алгоритму дозволяє йому ефективно працювати на частково відсортованих даних.

Timsort має кілька значних переваг. Оскільки він адаптивний, алгоритм добре працює на реальних даних, які часто мають певний ступінь впорядкованості. Висока ефективність на практиці робить Timsort одним з найкращих алгоритмів для великих масивів даних. Крім того, цей алгоритм є стабільним, тобто зберігає відносний порядок однакових елементів, що є важливим для багатьох застосувань. Timsort також підходить для зовнішнього сортування, оскільки він базується на сортуванні злиттям, яке добре працює з великими обсягами даних, що не вміщуються в оперативну пам'ять.

Однак, є й недоліки. Реалізація Timsort є складнішою порівняно з іншими алгоритмами сортування. Це може ускладнювати розуміння і підтримку коду. Крім того, алгоритм потребує додаткової пам'яті для зберігання підмасивів під час злиття, що може бути проблемою при обробці дуже великих наборів даних.

Алгоритм Timsort починається з розділення масиву на невеликі підмасиви, відомі як «run». Кожен «run» сортується окремо за допомогою сортування вставками, після чого відсортовані «run» зливаються за допомогою сортування злиттям.

На рисунку 1.8 наведено блок-схему алгоритму де в блоках підпрограм insertionSort це сортування вставками та merge це сортування злиттям.

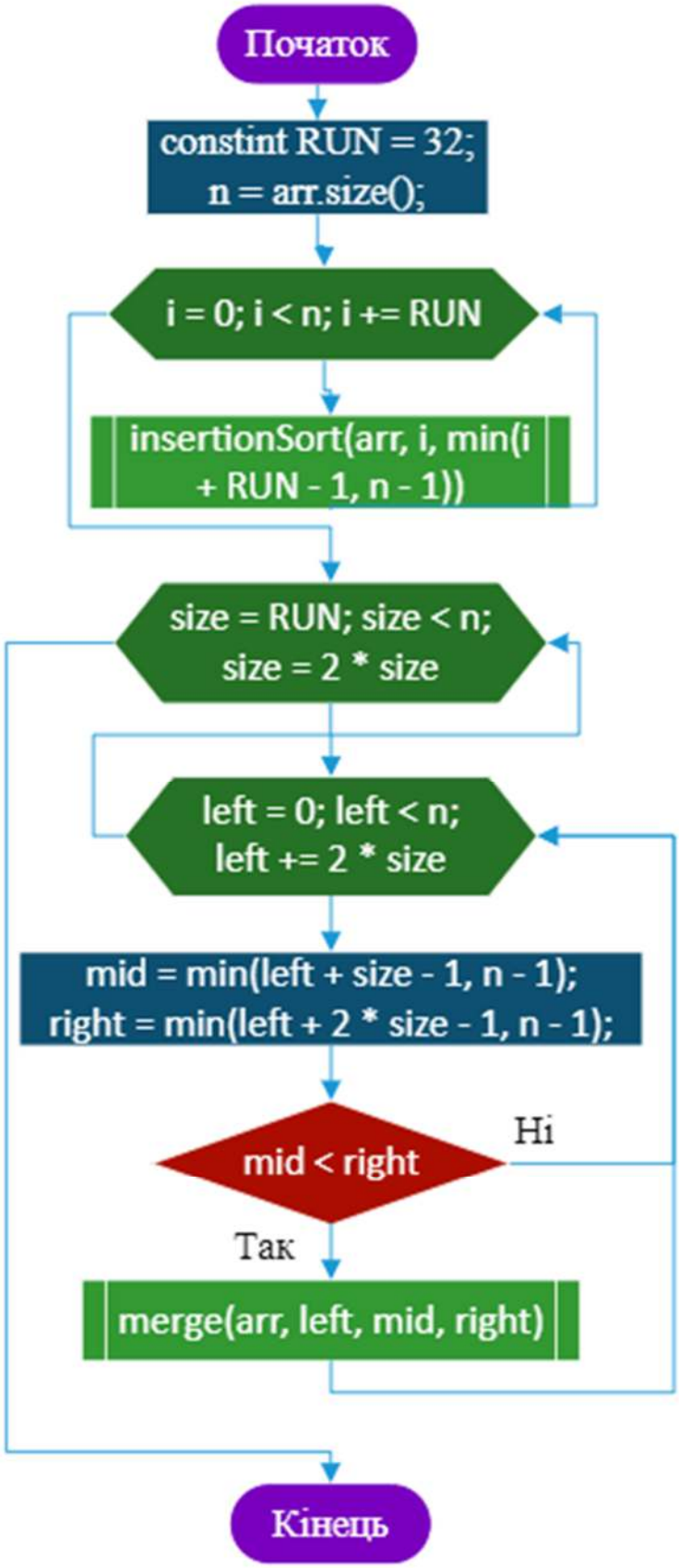

Рисунок 1.8 – Блок-схема сортування Timsort

1.3.5 Пірамідальне сортування

Пірамідальне сортування (Heapsort) є порівняльним алгоритмом сортування, що базується на використанні структури даних, відомої як двійкова купа (heap). Алгоритм починається з побудови максимальної купи з масиву, після чого найбільший елемент витягується з купи і розміщується в кінці масиву. Цей процес повторюється, зменшуючи розмір купи на 1 щоразу, доки весь масив не буде відсортовано.

Часова складність пірамідального сортування в найгіршому випадку становить O(n log n). Це досягається завдяки побудові купи, яка має часову складність O(n), та видаленню максимального елемента з подальшим відновленням купи, що вимагає O(log n) операцій для кожного з n елементів. Просторова складність алгоритму становить O(1), оскільки він виконується «на місці» без використання додаткової пам'яті.

Пірамідальне сортування забезпечує високу продуктивність на великих наборах даних завдяки своїй часовій складності O(n log n), яка гарантована в будь-якому випадку.

Проте, цей алгоритм не є стабільним, тобто порядок однакових елементів може змінюватися під час сортування. Просторова ефективність алгоритму забезпечується використанням наявного масиву без додаткового використання пам'яті.

Варто зазначити, що реалізація пірамідального сортування є складнішою порівняно з іншими алгоритмами. На частково відсортованих масивах пірамідальне сортування може мати гіршу продуктивність у порівнянні з деякими іншими алгоритмами.

Пірамідальне сортування може бути застосовано для зовнішнього сортування завдяки його просторовій ефективності, що дозволяє працювати з великими наборами даних, які не вміщуються в оперативну пам'ять.

На рисунку 1.9 наведено блок-схему алгоритму пірамідального сортування.

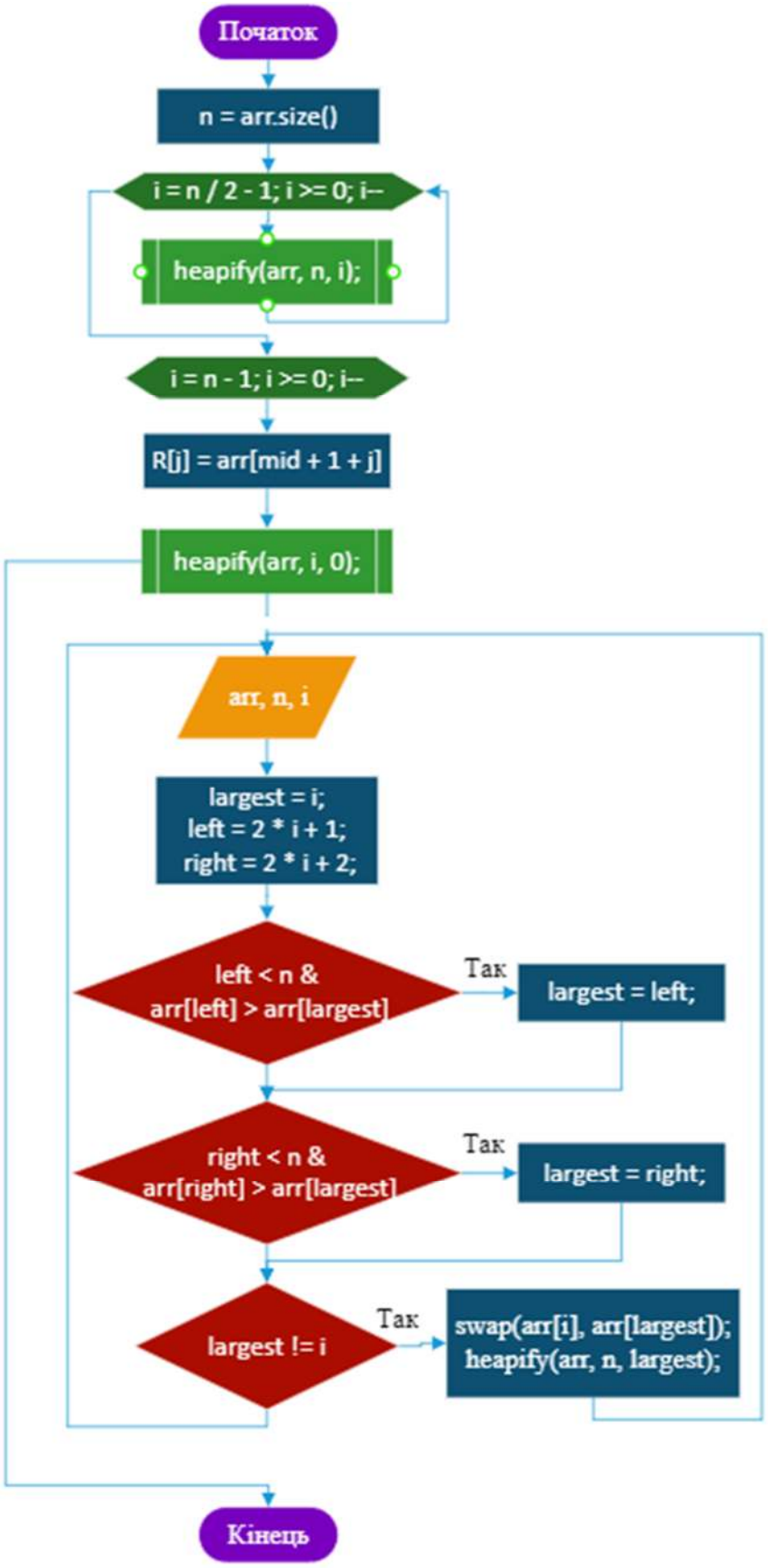

Рисунок 1.9 – Блок-схеми алгоритму пірамідального сортування

1.3.6 Плавне сортування

Плавне сортування (Smoothsort) — це адаптивний алгоритм сортування який є модифікацією пірамідального алгоритму сортування. Він є варіантом пірамідального сортування (Heapsort), але він більш оптимізований для сортування частково відсортованих масивах. Алгоритм використовує спеціальну структуру даних, відому як Леонардові купи (Leonardo heaps), для оптимізації процесу сортування.

Часова складність плавного сортування у найгіршому випадку становить O(n log n). У середньому випадку, коли масив частково відсортований, часова складність може бути ближчою до O(n). Просторова складність алгоритму становить O(1), оскільки він виконується «на місці» без використання додаткової пам'яті. Плавне сортування не є ефективним для зовнішнього сортування через складність управління Леонардовими купами при роботі з даними, що не вміщуються в оперативну пам'ять.

Даний алгоритм забезпечує високу ефективність на частково відсортованих масивах завдяки тому що він своїй здатності адаптуватися до вже впорядкованих даних. Просторова ефективність алгоритму також є важливою перевагою, оскільки він використовує мінімум додаткової оперативної пам'яті.

Однак, плавне сортування є складнішим для реалізації та розуміння порівняно з іншими алгоритмами сортування, такими як пірамідальне сортування або сортування вставками.

Крім того, продуктивність у найгіршому випадку аналогічна продуктивності пірамідального сортування, що може бути недоліком для деяких застосувань.

На рисунку 1.10 наведено блок-схему алгоритм плавного сортування. Слід зазначити що генерація чисел Леонарда не обов’язкова коли розмір масиву даних відомий до початку виконання сортування. Тому блок-схема містить деякі модифікації.

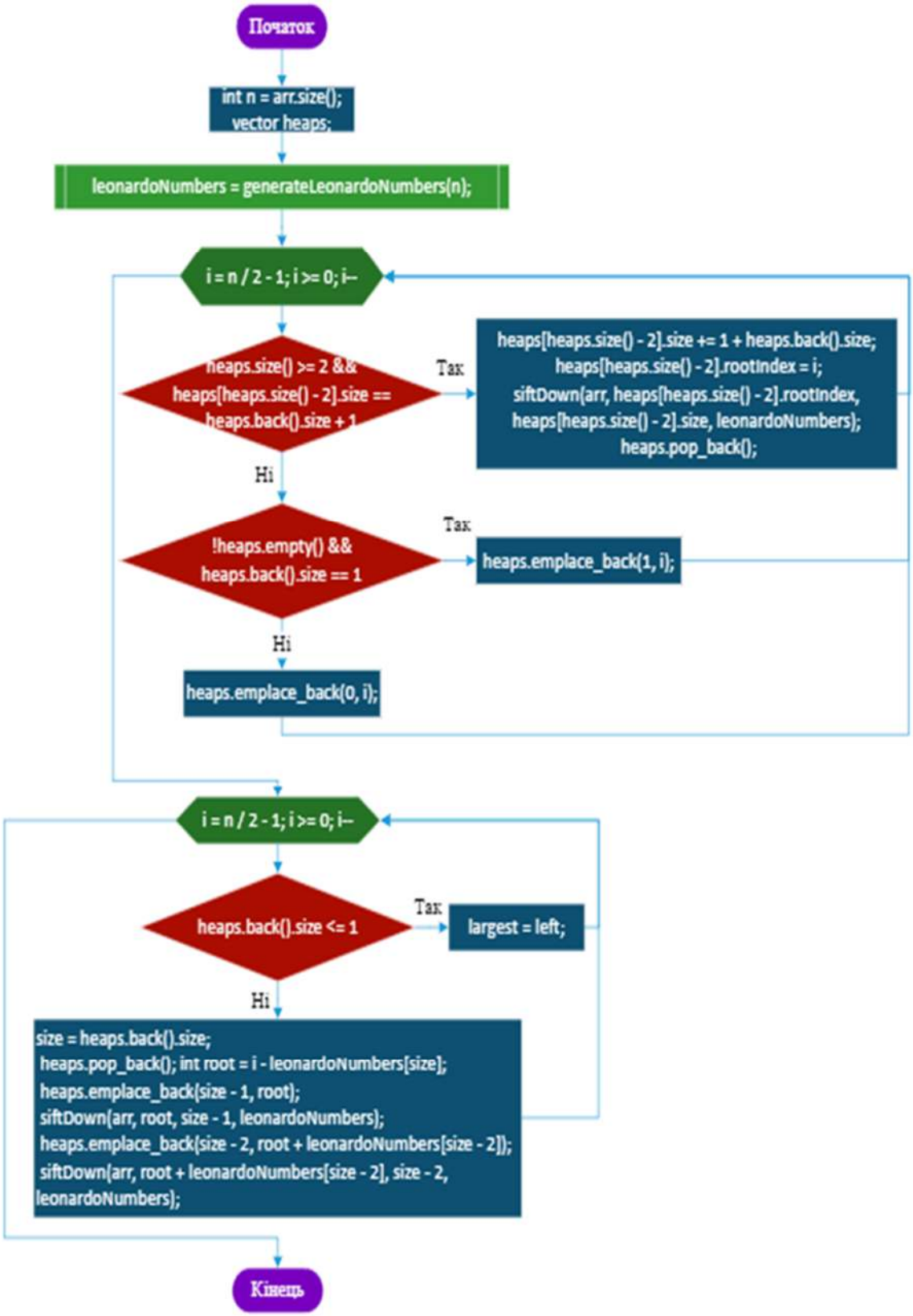

Рисунок 1.10 – Псевдокод для алгоритму Плавного сортування

1.3.7 Швидке сортування

Швидке сортування (Quicksort) — це один з найпопулярніших алгоритмів сортування, який базується на принципі «розділяй і володарюй». Алгоритм працює шляхом вибору опорного елемента, який загалом називають pivot, і розділення масиву на дві частини: елементи, менші за опорний, розташовуються перед ним, а елементи, більші або рівні опорному, розташовуються після нього. Потім алгоритм рекурсивно застосовується до кожної з цих частин.

Швидке сортування демонструє середню складність O(n log n), що робить його дуже ефективним для великих масивів даних. У найгіршому випадку, коли вибір опорного елемента є невдалим, алгоритм може мати складність $O(n^2)$. Проте, завдяки різним методам вибору опорного елемента, таких як median-of-three, ця ймовірність знижується, і алгоритм забезпечує кращу продуктивність в середньому випадку.

У найкращому випадку, коли масив вже частково відсортований, складність також становить O(n log n). Це робить швидке сортування універсальним вибором для багатьох завдань сортування, хоча його нестабільність і чутливість до вибору опорного елемента можуть бути обмежуючими факторами в певних випадках [11]. Популярної модифікацією є Dual-Pivot яка обирає декілька опорних елементів [12].

Швидке сортування має кілька переваг, включаючи ефективність та простоту реалізації. Він також ефективно використовує пам'ять, оскільки не потребує додаткового простору для сортування (окрім стека рекурсії). Проте, швидке сортування не є стабільним, тобто порядок однакових елементів може змінитися.

Крім того, алгоритм чутливий до вибору опорного елемента, і найгірший випадок може трапитися, якщо вибір опорного елемента невдалий.

На рисунку 1.11 наведено блок-схему алгоритму Dual-Pivot швидкого сортування.

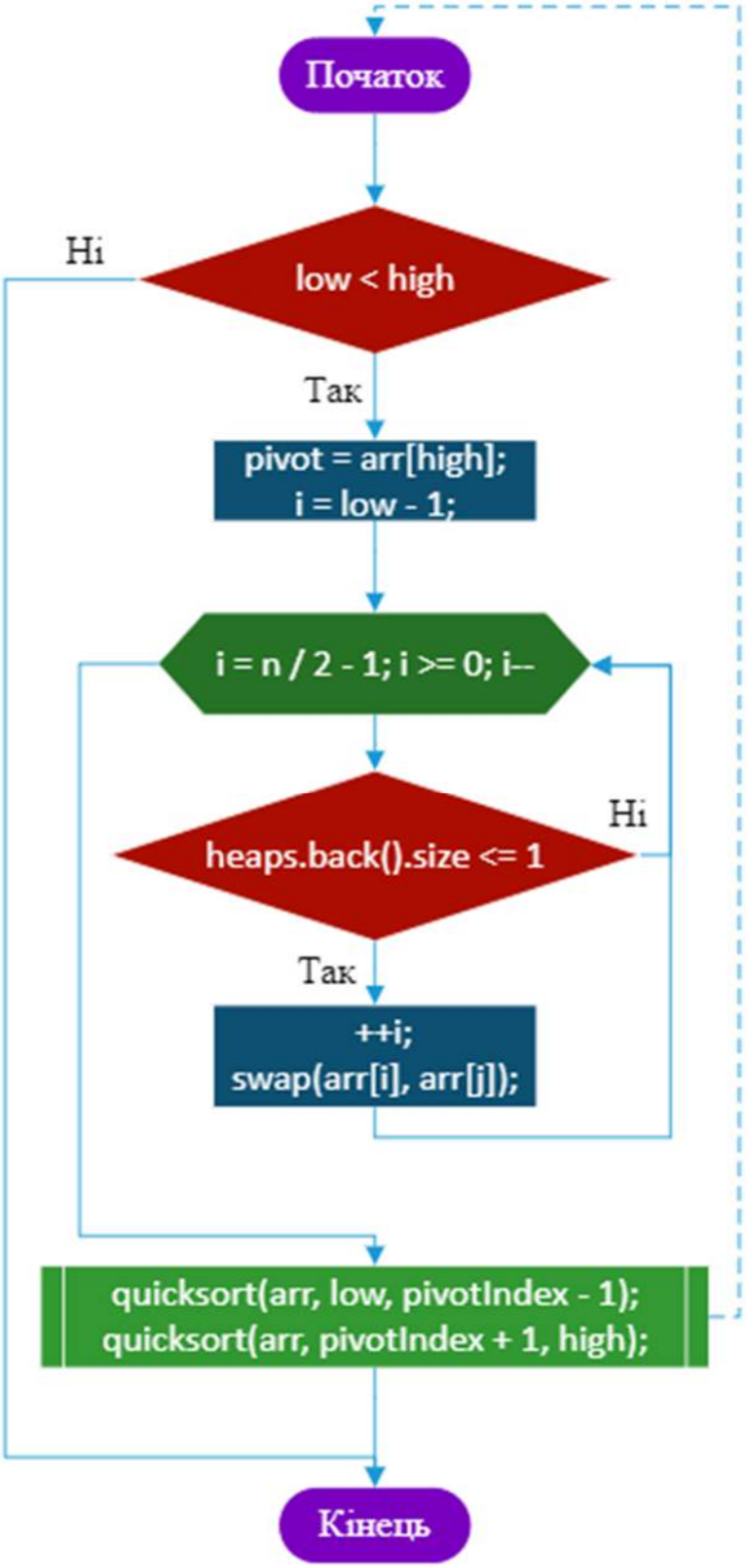

Рисунок 1.11 – Псевдокод для алгоритму швидкого сортування

2 ТЕХНІЧНІ ВІДОМОСТІ МОВ ПРОГРАМУВАННЯ ТА СИСТЕМИ

2.1 Вибір мови програмування

Розробка програмного забезпечення вимагає врахування багатьох факторів, одним з яких є ефективність управління пам'яттю та підтримка багатопоточності. Різні мови програмування реалізують ці аспекти по-різному, що впливає на продуктивність, надійність та масштабованість додатків. Тому вибір мови програмування повинен залежати від специфічних вимог проекту та особливостей мови щодо управління пам'яттю та багатопоточністю.

Управління пам'яттю є одним з найважливіших аспектів, які слід враховувати при виборі мови програмування. Деякі мови використовують автоматичне управління пам'яттю через збирач сміття, що звільняє розробників від необхідності вручну видаляти невикористовувані об'єкти. Збирач сміття автоматично виявляє і видаляє непотрібні об'єкти, значно спрощуючи розробку та зменшуючи кількість помилок, пов'язаних з управлінням пам'яттю.

Інші мови надають програмістам повний контроль над управлінням пам'яттю. Використання ручного управління пам'яттю дозволяє створювати високопродуктивні програми, але вимагає уважного підходу для уникнення витоків пам'яті та інших помилок. Ручне управління пам'яттю підходить для розробки системного програмного забезпечення та додатків.

Багатопоточність є ключовим аспектом для підвищення продуктивності додатків. Різні мови програмування мають різні підходи до реалізації багатопоточності. Деякі мови мають вбудовану підтримку багатопоточності, що дозволяє створювати потужні багатопоточні додатки. Вони надають широкий спектр інструментів для роботи з потоками, синхронізації та управління конкурентністю.

Інші мови можуть мати обмеження щодо багатопоточності, наприклад, наявність глобального блокування інтерпретатора, що не дозволяє виконувати кілька потоків одночасно в одному процесі. Проте, такі мови можуть надавати можливість обійти це обмеження через використання багатопроцесорності або сторонніх бібліотек.

Деякі мови забезпечують багатопоточність через модель асинхронного виконання та подійно-орієнтовану архітектуру, що дозволяє ефективно обробляти події без блокування виконання коду. Така модель підходить для додатків, які потребують високої продуктивності при обробці подій у реальному часі. Інші мови використовують стандартні бібліотеки для створення та управління потоками, надаючи простий і потужний інтерфейс для розробки багатопоточних додатків, що ефективно використовують багатоядерні процесори.

Вибір мови програмування для конкретного проекту повинен враховувати особливості управління пам'яттю та підтримки багатопоточності. Якщо важлива автоматизація управління пам'яттю, варто обирати мови з автоматичним управлінням пам'яттю. Якщо проект вимагає високої продуктивності та низького рівня контролю, краще обрати мови з ручним управлінням пам'яттю.

Щодо багатопоточності, слід обирати мови, які надають потужні інструменти для створення багатопоточних додатків, якщо це є важливим аспектом проекту. У випадках, де багатопоточність може бути обмеженою, слід розглянути можливість використання багатопроцесорності або асинхронного виконання. Враховуючи ці аспекти, розробники можуть обрати мову програмування, яка найкраще відповідає вимогам їхнього проекту, забезпечуючи оптимальне управління пам'яттю та ефективну підтримку багатопоточності. Такий підхід дозволяє досягти високої продуктивності, надійності та масштабованості додатку що є ключовими факторами успіху в сучасному світі технологій.

2.2 Python

Python була створена Гвідо ван Россумом і вперше випущена у 1991 році. Ця мова програмування розроблялася з акцентом на читабельність та простоту синтаксису, що полегшує навчання та використання. З моменту свого створення Python постійно оновлюється, що значно розширило її функціональні можливості. Сьогодні Python є однією з найпопулярніших мов програмування в світі.

Однією з ключових переваг Python є її простота та зрозумілість коду. Синтаксис мови розроблений таким чином, щоб бути інтуїтивно зрозумілим, що дозволяє швидко писати та підтримувати код. У Python використовуються відступи для позначення блоків коду, що зменшує кількість синтаксичних помилок та робить код більш структурованим. Крім того, Python підтримує ООП, що сприяє модульності та повторному використанню коду [13].

Python використовує збирач сміття для автоматичного управління пам'яттю. Це означає, що розробники не повинні вручну звільняти пам'ять, яка більше не використовується, оскільки збирач сміття автоматично виявляє і видаляє невикористовувані об'єкти. Це значно спрощує розробку та знижує кількість помилок з управлінням пам'яттю. Збирач сміття працює у фоновому режимі та використовує різні алгоритми для ефективного управління пам'яттю, такі як підрахунок посилань та виявлення циклічних посилань.

Одним із суттєвих обмежень Python є наявність Global Interpreter Lock (GIL), який не дозволяє виконувати кілька потоків одночасно в одному процесі. GIL дозволяє виконання лише одного потоку за раз, що може обмежувати продуктивність багатопотокових програм. Це обмеження особливо помітне в програмах з великою кількістю обчислень. Однак, Python надає можливість обійти це обмеження через використання багатопроцесорності або за допомогою сторонніх бібліотек, таких як multiprocessing, що дозволяють створювати та керувати окремими процесами, які можуть виконуватись паралельно.

2.3 JavaScript

JavaScript була створена Бренданом Айком у 1995 році і швидко стала ключовою мовою для розробки веб-сторінок. Спочатку призначена для використання в браузерах, JavaScript дозволяє створювати динамічні та інтерактивні веб-інтерфейси. З появою таких технологій, як Node.js, JavaScript також стала популярною для серверної розробки. На сьогодні JavaScript є однією з найбільш розповсюджених мов програмування у світі, активно використовується як у фронтенд, так і в бекенд розробці [14].

JavaScript відома своєю асинхронною природою, що дозволяє ефективно обробляти події без блокування виконання коду. Завдяки подійно-орієнтованій моделі, JavaScript може реагувати на дії користувачів, серверні запити та інші події в реальному часі. Це досягається через використання функцій зворотного виклику (callbacks), промісів (promises) та новітнього синтаксису async/await, що значно спрощує роботу з асинхронним кодом [15].

JavaScript використовує збирач сміття для автоматичного управління пам'яттю, що звільняє розробників від необхідності вручну видаляти невикористовувані об'єкти. Збирач сміття працює у фоновому режимі і видаляє об'єкти, які більше не доступні через посилання. Це дозволяє оптимізувати використання пам'яті, хоча може викликати короткочасні затримки в роботі додатку під час виконання збирача сміття та затримки вивільнення пам’яті яка була використана, так як потрібен час для аналізу через посилання чи може об’єкт бути досягнутий [14].

Хоча JavaScript в браузері виконується в одному потоці, Node.js дозволяє використовувати багатопоточність через Worker Threads. Worker Threads забезпечують можливість виконання фонових задач у окремих потоках, що дозволяє значно підвищити продуктивність і ефективність серверних додатків. Ця функціональність особливо корисна для обробки обчислювально-інтенсивних задач, які можуть заблокувати основний потік але не підходить для алгоритмів.

2.4 Java

Java була створена Джеймсом Гослінгом і його командою в Sun Microsystems у 1995 році. Вона була розроблена для створення портативних, високопродуктивних програм, здатних працювати на будь-якому пристрої, що підтримує платформу Java. Завдяки своїй платформо-незалежності та надійності, Java швидко здобула популярність у корпоративному секторі. Сьогодні Java є однією з найбільш використовуваних мов програмування у світі, активно застосовується для розробки корпоративних, веб та мобільних додатків.

Java є об'єктно-орієнтованою мовою програмування, що означає, що все в Java розглядається як об'єкт. Це забезпечує модульність, легкість підтримки та повторне використання коду. Java також має простий і зрозумілий синтаксис, що сприяє легкому навчанню та швидкому розробленню програм. Крім того, Java включає в себе велику стандартну бібліотеку класів, яка надає розробникам широкий спектр готових до використання функцій.

Java використовує збирач сміття (Garbage Collector) для автоматичного управління пам'яттю. Це означає, що розробникам не потрібно вручну звільняти пам'ять, яка більше не використовується, оскільки збирач сміття автоматично виявляє і видаляє невикористовувані об'єкти. Це значно спрощує розробку та знижує ризик витоків пам'яті. Збирач сміття працює у фоновому режимі.

Java має вбудовану підтримку багатопоточності, що дозволяє створювати програми, які можуть виконувати кілька завдань одночасно. Це досягається через використання класів і методів з бібліотеки java.util.concurrent, яка надає широкий спектр інструментів для роботи з потоками, синхронізації та управління конкурентністю.

Завдяки цьому мова програмування Java ідеально підходить для розробки високопродуктивних додатків, які вимагають ефективного використання багатоядерних процесорів.

2.5 C++

C++ була створена Б'ярне Страуструпом у 1985 році як розширення мови C. Спочатку ця мова програмування розроблялася для додавання об'єктно-орієнтованих можливостей до C, що дозволило програмістам створювати більш структуровані та легко підтримувані програми. Завдяки своїй ефективності та здатності працювати на низькому рівні, C++ швидко здобула популярність серед розробників. Вона стала стандартом для розробки системного програмного забезпечення, драйверів, ігор, вбудованих систем та інших додатків, де важливими є висока продуктивність і контроль над апаратними ресурсами.

C++ відома своєю високою ефективністю та продуктивністю. Вона дозволяє програмістам працювати на низькому рівні, що сприяє оптимізації використання ресурсів системи. Мова підтримує об'єктно-орієнтоване програмування, а також включає елементи процедурного та функціонального програмування, що робить її дуже гнучкою. C++ також забезпечує прямий доступ до пам'яті через вказівники, що дозволяє більш точно контролювати управління пам'яттю.

C++ надає програмістам повний контроль над управлінням пам'яттю. Це досягається за допомогою операторів new і delete для виділення та звільнення пам'яті відповідно. Також можна використовувати функції malloc і free з мови C. Хоча це дозволяє створювати дуже оптимізовані програми, таке управління пам'яттю вимагає уважного підходу, щоб уникнути витоків пам'яті.

C++ має вбудовану підтримку багатопоточності через стандартну бібліотеку <thread>, яка надає простий і потужний інтерфейс для створення та управління потоками. Крім того, існують численні сторонні бібліотеки, такі як Boost.Thread, які забезпечують додаткові можливості для роботи з багатопоточністю. Завдяки цьому C++ підходить для розробки високопродуктивних багатопоточних додатків, які можуть ефективно використовувати багатоядерні процесори.

2.6 C#

C# була створена компанією Microsoft у 2000 році як частина платформи .NET. Дана мова програмування була розроблена для поєднання потужності C++ з простотою Visual Basic. Завдяки інтеграції з платформою .NET, C# швидко стала популярною для розробки програм, веб та мобільних додатків. На сьогоднішній день C# є однією з провідних мов програмування, широко використовуваною у багатьох сферах, включаючи корпоративне програмування та ігрову індустрію.

C# є об'єктно-орієнтованою мовою програмування, що забезпечує високу модульність, зручність підтримки та повторне використання коду. Мова включає сучасні мовні конструкції, такі як властивості, події, делегати та анонімні методи, що полегшують розробку. Крім того, C# підтримує багато парадигм програмування, включаючи імперативне, декларативне, функціональне та подійно-орієнтоване програмування.

C# використовує збирач сміття (Garbage Collector) для автоматичного управління пам'яттю. Це означає, що розробникам не потрібно вручну видаляти об'єкти, які більше не використовуються, оскільки збирач сміття автоматично виявляє і видаляє невикористовувані об'єкти. Це значно спрощує розробку і знижує ризик витоків пам'яті. Збирач сміття працює у фоновому режимі, забезпечуючи оптимальне використання пам'яті системи.

Крім того, сучасні версії збирача сміття в C# використовують складні алгоритми для мінімізації затримок та підвищення продуктивності, що робить його ефективним навіть у додатках з високими вимогами до продуктивності додатку.

C# має вбудовану підтримку багатопоточності через простий і зручний інтерфейс для створення та управління потоками. Мова надає розширені можливості для роботи з багатопоточністю через бібліотеки System.Threading та System.Threading.Tasks, що забезпечують високий рівень абстракції та ефективне управління потоками.

2.7 Тестовий стенд

В тестовому стенді який використовується для тестування продуктивність алгоритмів сортування установлені такі комплектуючі як:

- тип ЦП: HexaCore AMD Ryzen 5 5600X, 4642 MHz (46.5 x 100);
- системна плата: Gigabyte B550 Aorus Pro AC (2 PCI-E x1, 3 PCI-E x16, 2 M.2, 4 DDR4 DIMM, Audio, Video, 2.5GbE LAN, WiFi);
- чипсет системної плати: AMD B550, AMD K19.2 FCH, AMD K19.2 IMC
- системна пам'ять: 32672 МБ;
- DIMM(1-4): Kingston Fury KF3600C17D4/8GX 8 ГБ DDR4-3000 DDR4 SDRAM (17-17-17-36 @ 1500 МГц) (16-17-17-36 @ 1500 МГц);
- тип BIOS: AMI (07/20/2022);
- відеоадаптер: NVIDIA GeForce RTX 2060 SUPER (8 ГБ);
- дисковий накопичувач: Samsung SSD 980 500GB (500 ГБ, PCI-E 3.0 x4).

На рисунку 2.1 наведено кількість віртуальної пам’яті яка виставлена у системі. Слід бути уважним тому що тестовий стенд має 3 диски на яких використовуються віртуальна пам’ять тому вона буде працювати швидше. Також слід зазначити що в системі встановлено SSD у форматі підключення M2 що має набагато більшу швидкість читання за запису а ніж класичні жорсткі диски.

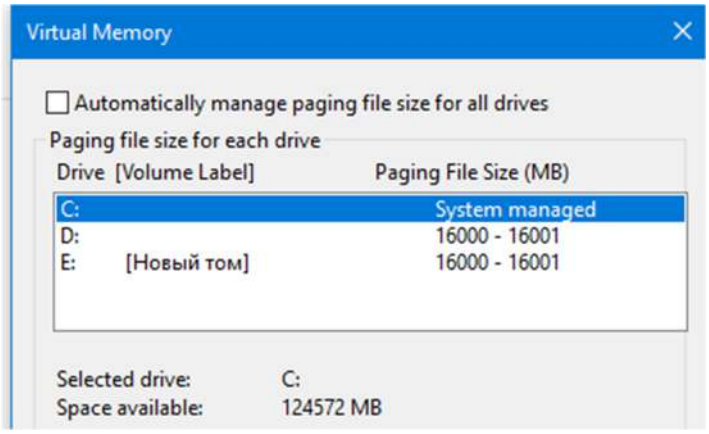

Рисунок 2.1 – Обсяг віртуальної пам’яті системи

2.8 Налаштування системи

Для забезпечення максимальної продуктивності та точності результатів тестування важливо налаштувати систему на стабільну та високу частоту роботи. Одним із критичних аспектів цього налаштування є зміна частоти процесора. Нижче наведено детальний опис процесу налаштування частоти процесора за допомогою AMD Ryzen Master.

Перед початком налаштування необхідно завантажити та встановити програму AMD Ryzen Master, яка забезпечує інтуїтивно зрозумілий інтерфейс для налаштування та моніторингу параметрів процесора. Завантажити AMD Ryzen Master можна з офіційного сайту AMD.

Після того як додаток був завантажений потрібно його інсталювати. Після завершення встановлення перезавантажити комп'ютер.

Після пророблених кроків отримуємо працюючий програмний додаток інтерфейс якого наведено на рисунку 2.2.

Рисунок 2.2 – Інтерфейс встановленого додатку AMD Ryzen Master

Після вибору профілю в AMD Ryzen Master можна приступити до налаштування частоти процесора для досягнення бажаного рівня продуктивності. У даному випадку частота процесора встановлюється на 4600 MHz (4.6 GHz). Спочатку необхідно відобразити всі ядра процесора, натиснувши кнопку «Show Details» або подібну. Потім знайти параметр «CPU Core Ratio» або подібний для кожного ядра та встановити значення 46 для кожного ядра, що відповідатиме частоті 4600 MHz (при базовій частоті 100 MHz).

Для стабільної роботи процесора на підвищеній частоті може знадобитися зміна напруги. Потрібно знайти параметр «CPU Core Voltage» або подібний та встановити значення напруги, яке забезпечить стабільну роботу процесора на частоті 4600 MHz. Це значення може варіюватися в залежності від конкретного процесора, але зазвичай знаходиться в діапазоні 1.35-1.45 V.

Після внесення всіх необхідних змін слід зберегти та застосувати налаштування. Для цього натиснути кнопку «Apply» або «Apply & Test». Програма може запропонувати провести тест для перевірки стабільності нових налаштувань. Погоджуючись, необхідно дочекатися завершення перевірки, щоб виявити можливі проблеми зі стабільністю при нових налаштуваннях.

Після застосування налаштувань важливо стежити за станом системи та параметрами процесора, щоб переконатися у стабільності роботи. Використовуйте вкладку «Monitoring» у програмі AMD Ryzen Master для відстеження температури, частоти, напруги та інших параметрів процесора під час роботи. За необхідності треба винести додаткові зміни до налаштувань, щоб забезпечити оптимальну продуктивність і стабільність.

Налаштування частоти процесора за допомогою AMD Ryzen Master є важливим кроком для досягнення стабільної та високопродуктивної роботи системи. Встановлення частоти процесора на 4600 MHz дозволяє забезпечити необхідний рівень продуктивності для проведення тестів і отримання точних результатів. Детальне налаштування параметрів і перевірка стабільності системи забезпечують надійну роботу комп'ютера під час проведення

експериментів і тестування алгоритмів сортування.

Для підвищення продуктивності системи використано XMP-профілі для налаштування пам'яті за допомогою AMD Ryzen Master. XMP (Extreme Memory Profile) – це технологія, розроблена для спрощення налаштування оперативної пам'яті, що дозволяє автоматично застосовувати оптимальні параметри для досягнення максимальної продуктивності.

У програмі AMD Ryzen Master активовано XMP-профіль для оперативної пам'яті. Вибрано найшвидший доступний профіль, який забезпечує максимальну частоту і мінімальні затримки пам'яті, що дозволяє системі працювати з максимальною ефективністю. Це забезпечує більш швидку обробку даних та покращує загальну продуктивність системи. На рисунку 2.3 наведено характеристики пам’яті після налаштувань.

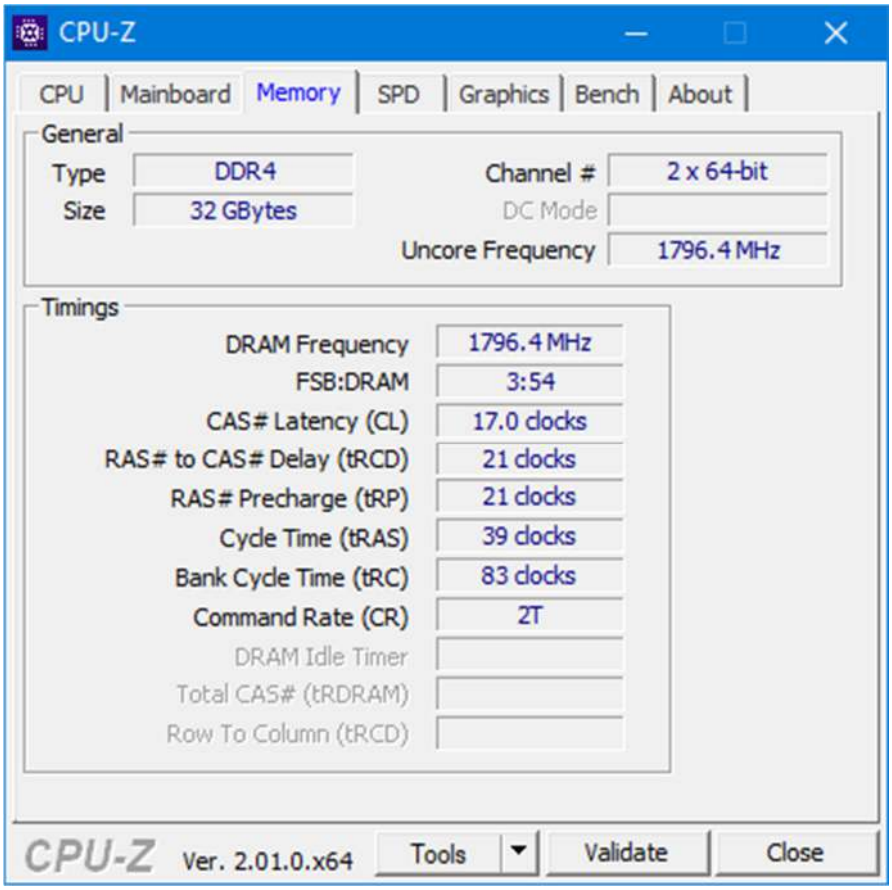

Рисунок 2.3 – Інтерфейс встановленого додатку AMD Ryzen Master

Для тестування алгоритмів сортування також створено повільний профіль пам'яті. Це дозволяє оцінити вплив швидкості пам'яті на продуктивність алгоритмів. У налаштуваннях AMD Ryzen Master створено профіль з зниженою частотою пам'яті та збільшеними затримками. Це дозволяє моделювати умови роботи на системах з повільнішою ОП і отримати більш повне уявлення про продуктивність алгоритмів у різних умовах.

Процес налаштування XMP-профілів пам'яті включав наступні кроки:

- відкриття програми AMD Ryzen Master;

- перехід до налаштувань пам'яті;

- вибір XMP-профілю з найвищою частотою для максимального підвищення продуктивності.

Створення повільного профілю пам'яті зі зниженою частотою і збільшеними затримками для тестування в умовах меншої продуктивності.

Таким чином, налаштування XMP-профілів пам'яті за допомогою AMD Ryzen Master дозволило забезпечити оптимальну продуктивність системи та створити умови для всебічного тестування алгоритмів сортування.

Вибір найшвидшого профілю забезпечив максимальну продуктивність для основних тестів, тоді як створення повільного профілю дозволило оцінити вплив швидкості пам'яті на продуктивність алгоритмів в умовах повільної швидкості та таймінгів оперативної пам’яті.

Також було вимкнено всі енергозберігаючі режими. У Windows було виставлено план живлення «Максимальна продуктивність» для того щоб в перервах між часом простою процесору та навантаженням не витрачався додатковий час на перемикання системи на більш або менш продуктивний план.

Налаштування частоти процесора на 4600 MHz та використання XMP-профілів для оперативної пам'яті забезпечили максимальну продуктивність та стабільність системи. Відключення енергозберігаючих режимів запобігло затримкам і зниженню частоти, що гарантує точність і повторюваність результатів тестування.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Вибір середовищ розробки для тестування алгоритмів

Для реалізації та тестування алгоритмів сортування на різних мовах програмування важливо вибрати відповідні середовища розробки. Правильний вибір середовища забезпечить зручність розробки, налагодження та тестування коду, а також оптимальну продуктивність.

Для реалізації алгоритмів на C++ було обрано обрали Visual Studio. Це середовище розробки надає широкий спектр інструментів для написання, налагодження та оптимізації коду.

Visual Studio підтримує інтеграцію з Git, має зручний інтерфейс для роботи з проектами різного масштабу та надає потужні можливості для роботи з багатопоточністю та управління пам'яттю.

Переваги Visual Studio для C++;

- підтримка різних версій стандарту C++;
- потужні інструменти для налагодження та профілювання коду;
- інтеграція з Git та іншими системами контролю версій;
- широкий набір розширень для додаткової функціональності.

Для розробки алгоритмів на JavaScript було обрано WebStorm. Це середовище розробки від компанії JetBrains спеціалізується на JavaScript і веб-розробці. WebStorm забезпечує інтелектуальне автодоповнення коду, потужні інструменти для налагодження та підтримку сучасних веб-технологій.

Переваги WebStorm для JavaScript:

- підтримка всіх сучасних фреймворків і бібліотек JavaScript;
- інтелектуальне автодоповнення та навігація по коду;
- вбудовані інструменти для налагодження та тестування;
- інтеграція з системами контролю версій;
- підтримка різних веб-технологій, таких як HTML і CSS.

3.2 Генерація тестових масивів

Генерація масивів для тестування алгоритмів сортування є важливою складовою дослідження ефективності різних методів сортування. У реальних додатках дані можуть мати різні властивості, і важливо перевірити, як алгоритми сортування працюють у різних умовах. Використання згенерованих масивів дозволяє створити контрольоване середовище для тестування, де можна точно оцінити поведінку кожного алгоритму в різних ситуаціях. Це забезпечує надійність і репрезентативність результатів тестування, що є критичним для оцінки продуктивності алгоритмів.

Використання згенерованих масивів також стандартизує процес тестування, що важливо для порівняння результатів різних методів сортування.

При тестуванні будуть згенеровані масиви з розмірами 100, 10000, 1000000 Така кількість елементів була обрана для тестування з кількох причин. Вибір таких розмірів масивів забезпечує всебічний аналіз продуктивності алгоритмів сортування на малих, середніх і великих наборах даних.

Масиви з 100 елементів дозволяють швидко отримати початкову оцінку продуктивності алгоритму та виявити основні тенденції. Вони ідеально підходять для швидкого тестування та визначення базових характеристик алгоритмів.

Масиви з 10000 елементів представляють більш реалістичні сценарії використання, які часто зустрічаються в реальних додатках. Вони дозволяють оцінити, як алгоритми працюють з середніми обсягами даних, що є типовим для багатьох прикладних програм, таких як обробка транзакцій, аналіз даних та управління контентом. Дані масиви також допомагають виявити, як алгоритми масштабуються при збільшенні розміру даних.

Масиви з 1000000 елементів дають можливість оцінити масштабованість і продуктивність алгоритмів у великих системах з великими обсягами даних.

Великі масиви є типовими для додатків, що обробляють великі дані, таких як системи керування базами даних, пошукові системи, системи рекомендацій та аналітичні платформи. Тестування на великих масивах дозволяє виявити потенційні проблеми з продуктивністю та оптимізацією, які можуть виникнути при роботі з великими наборами даних.

Генерація масивів, у яких зустрічаються відсортовані послідовності, додає важливий аспект до тестування. Реальні дані часто містять структуровані або частково впорядковані сегменти, і важливо зрозуміти, як алгоритми сортування справляються з такими випадками. Алгоритми, які ефективно працюють з частково впорядкованими даними, можуть мати значну перевагу в продуктивності у реальних умовах, де дані не завжди бувають повністю випадковими.

Генерація таких масивів дозволяє оцінити, як алгоритми адаптуються до наявності впорядкованих послідовностей і чи можуть вони ефективно використовувати цю інформацію для покращення продуктивності. Це також допомагає виявити алгоритми, які можуть використовувати наявну впорядкованість для прискорення процесу сортування.

В реальних сценаріях дані часто мають певний рівень впорядкованості через природу їх генерування або попередньої обробки, тому тестування алгоритмів у таких умовах є важливим для отримання достовірних результатів.

Для генерації масивів використовуються стандартні інструменти C++ для створення випадкових чисел. Процес генерації включає кілька етапів:

- використовується генератор псевдовипадкових чисел Mersenne Twister (std::mt19937), який ініціалізується значенням від апаратного генератора випадкових чисел (std::random_device). Це забезпечує отримання високоякісних випадкових значень у діапазоні від 1 до розміру масиву. Такий підхід гарантує, що кожен масив матиме унікальний набір значень, що дозволяє уникнути повторюваності і забезпечує різноманітність тестових даних;

- кожен масив заповнюється випадковими значеннями, що гарантує рівномірний розподіл чисел у масиві. Це дозволяє оцінити, як алгоритми справляються з повністю випадковими даними, що є типовим для багатьох реальних додатків, де дані генеруються або отримуються з різних джерел без попереднього сортування;

- для масивів, що містять відсортовані послідовності, додатково вставляються кілька впорядкованих сегментів. Це досягається шляхом генерації послідовностей довжиною 10 елементів, які розміщуються у випадкових позиціях масиву. Такий підхід дозволяє створити умови, що імітують частково впорядковані дані, що часто зустрічаються в реальних системах, таких як бази даних, журнали транзакцій та інші джерела даних.

Для тестування алгоритмів була розроблена універсальна функція generateRandomArray для того щоб створювати масиви, які використовуються при тестуванні алгоритмів сортування. Вона дозволяє генерувати масиви випадкових чисел, а також додавати до них частково відсортовані послідовності, що імітують реальні умови, коли дані можуть бути як хаотичними, так і мати вже частково впорядковані послідовності. Функцію було розроблено завдяки мові програмування C++ так як вона із обраних є найшвидшою.

Розробка цієї функції починається з ініціалізації пустого масиву заданого розміру, що задається параметром size. Далі необхідно забезпечити генерацію випадкових чисел. Для цього використовується спеціальний об'єкт (random_device), який дозволяє отримувати справжні випадкові числа, оскільки він базується на апаратних характеристиках системи. Ці числа служать основою для ініціалізації генератора (mt19937), який є високопродуктивним і надійним способом створення псевдовипадкових чисел.

Щоб заповнити масив, використовується цикл, що проходить по кожному його елементу і присвоює йому випадкове число з діапазону від 1 до значення розміру масиву. Це дозволяє створити повністю випадковий набір

чисел, який стане базою для подальших експериментів із алгоритмами сортування.

Окрім випадкових чисел, у масив можна вставити відсортовані фрагменти, якщо це необхідно. Відсоток упорядкованості масиву задається параметром sortedPercentage, який визначає, скільки відсортованих послідовностей має бути додано. Розмір кожної такої послідовності фіксований і складається з 10 елементів — це класичний приклад, коли відсортований фрагмент є коротким та легко впізнаваним.

Кількість відсортованих фрагментів, які потрібно вставити, розраховується відповідно до розміру масиву та заданого відсотка. Наприклад, якщо потрібно додати 10% відсортованих даних то функція розраховує кількість таких фрагментів і вставляє їх у випадкові місця масиву. Для кожного фрагменту генерується початкова позиція, яка визначається випадковим чином, щоб уникнути регулярності в розташуванні відсортованих частин.

Під час вставки відсортованого фрагмента, масив заповнюється числами від 1 до 10, що відповідає класичній відсортованій послідовності. Таким чином, навіть у великому масиві випадкових чисел з’являються впорядковані елементи, що дозволяє тестувати сортування в умовах змішаних даних.

Цей підхід корисний для оцінки ефективності алгоритмів у різних сценаріях, коли масиви можуть містити як випадкові, так і частково впорядковані дані, відображаючи реальні випадки використання сортування в комп'ютерних системах.

Генерація масивів відбувається в шість етапів, щоб створити окремі файли для різних типів масивів. Отже маємо наступні файли з даними як:

- масиви розміром 100, 10000 та 1000000 елементів, мають повністю випадкові послідовності;

- масиви розміром 100, 10000 та 1000000 елементів з впорядкованими послідовностями.

Цей підхід дозволяє створити різноманітний набір тестових даних для

комплексного аналізу продуктивності алгоритмів сортування. Кожен тип масиву зберігається в окремому файлі, що дозволяє легко організувати та виконувати тестування. Це також забезпечує зручність у повторенні експериментів і порівнянні результатів.

Після виконання коду було отриманно шість текстових файлів з масивами які наведено на рисунку 3.1.

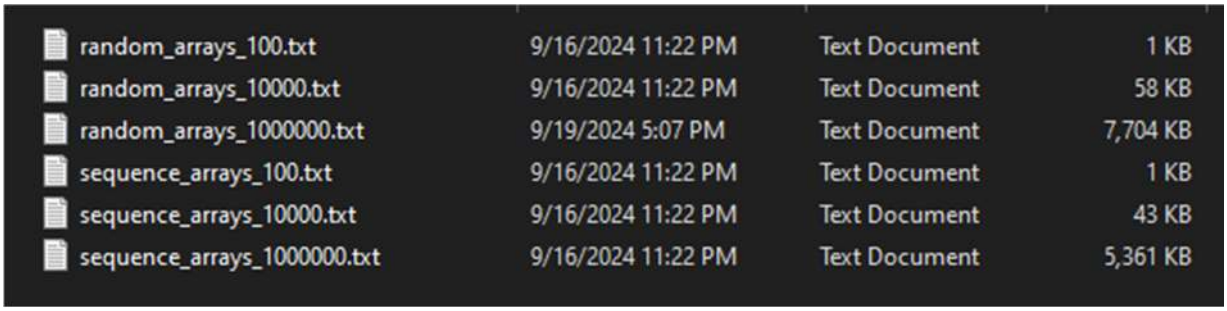

Рисунок 3.1 – Згенеровані файли з масивами для тестування

Цей підхід дозволяє створити різноманітний набір тестових даних для комплексного аналізу продуктивності алгоритмів сортування. Кожен тип масиву зберігається в окремому файлі, що дозволяє легко організувати та виконувати тестування. Це також забезпечує зручність у повторенні експериментів і порівнянні результатів.

Варто також зазначити, що процес генерації масивів був спроектований таким чином, щоб мінімізувати вплив зовнішніх факторів і забезпечити відтворюваність результатів. Використання псевдовипадкових чисел дозволяє повторити процес генерації масивів за необхідності, що є важливим для подальшого аналізу та порівняння результатів. Таким чином, генерація масивів різних розмірів та з різними характеристиками є критично важливим етапом у дослідженні, що дозволяє оцінити продуктивність алгоритмів сортування в різних умовах і забезпечити всебічний аналіз їх ефективності.

Створення стандартизованих тестових наборів даних дозволяє не лише отримати точні та репрезентативні результати, але й забезпечує можливість

повторного тестування та верифікації результатів у майбутньому.

Для того щоб перевірити чи добре було згенеровано масиви з відсортованими послідовностями було додатково розроблено функцію яка перевіряє відсоток послідовностей. Це потрібно тому що послідовності вставляються у масив в випадковому місці, тому потрібно бути впевненим що файли містять приблизно той відсоток який нам потрібен щоб забезпечити відтворюваність тестів на інших системах.

Для тестування алгоритмів сортування важливо використовувати масиви, що містять частково відсортовані послідовності, зокрема близько 40%. Це значення обране з певних причин, оскільки воно дозволяє краще моделювати реальні умови, в яких дані часто бувають частково впорядкованими.

Масиви з 40% відсортованих елементів відображають ситуації, коли дані мають певну структуру, але також містять невпорядковані ділянки, що забезпечує більш реалістичну оцінку роботи алгоритмів.

Часткова впорядкованість, близька до 40%, дозволяє протестувати, як алгоритм поводиться в умовах, де початкова структура масиву допомагає у сортуванні, але не настільки впорядкована, щоб повністю полегшити задачу. Це значення імітує проміжні сценарії, коли дані є частково впорядкованими, наприклад, через минулі маніпуляції або природний порядок, що часто трапляється в реальному житті.

Така частка відсортованості корисна для виявлення алгоритмів, здатних адаптуватися до початкової структури даних, показуючи кращу продуктивність порівняно з алгоритмами, які не враховують часткову впорядкованість. Вона також дає змогу оцінити стабільність алгоритму: якщо продуктивність значно знижується навіть при 40% впорядкованості, це може свідчити про його непридатність для задач із частково впорядкованими даними.

Таким чином, тестування на масивах із 40% відсортованих елементів надає всебічну оцінку алгоритмів сортування, дозволяючи виявити їхні сильні

і слабкі сторони. Це значення забезпечує баланс між випадковістю і впорядкованістю, що важливо для реалістичного аналізу ефективності алгоритмів у різноманітних умовах.

Функція calculatePartialSequencePercentage призначена для визначення процентного вмісту частково впорядкованих послідовностей у числовому масиві. Вона аналізує масив, виявляючи ряди чисел, що знаходяться в межах від 1 до 10, навіть якщо ці ряди не є суцільними або можуть мати повторювані значення. Цей підхід дозволяє оцінити наявність і довжину послідовностей у масиві, включаючи неповні або рвані ряди.

Робота функції починається з по елементного перебору масиву. Коли функція знаходить число, що входить до діапазону від 1 до 10, це число вважається початком нової частково впорядкованої послідовності. Після цього відбувається перевірка наступних елементів масиву, щоб визначити, чи продовжується послідовність. Кожен наступний елемент порівнюється з очікуваним значенням: якщо число дорівнює або менше очікуваного, то довжина послідовності збільшується. Це дозволяє включати до розрахунку навіть ті ряди, що мають пропуски чи повторення.

Якщо ж число в ряді перевищує очікуване значення на одиницю або більше, це вказує на кінець послідовності, і функція переходить до пошуку наступного можливого ряду. Усі знайдені послідовності накопичуються у змінній sequenceLength, яка відображає загальну довжину всіх частково впорядкованих рядів у масиві. Якщо жодної послідовності не знайдено, функція повертає нуль, що свідчить про повну відсутність упорядкованих числових рядів у масиві.

Результуюче значення обчислюється як процентне співвідношення загальної довжини знайдених послідовностей до загальної кількості елементів у масиві. Цей підхід надає можливість не лише виявити частково впорядковані послідовності, але й визначити їх вагу у загальному контексті даних, що є корисним для аналізу структури даних.

Оцінка таких показників важлива для досліджень, де часткова

впорядкованість може впливати на ефективність алгоритмів обробки чи аналізу даних, підкреслюючи, наскільки значущими є упорядковані частини у загальній масі числових даних.

На рисунку 3.2 наведено результат виконання даної функції для масивів з відсортованими послідовностями які будуть в подальшому використовуватися для тестування продуктивності алгоритмів сортування.

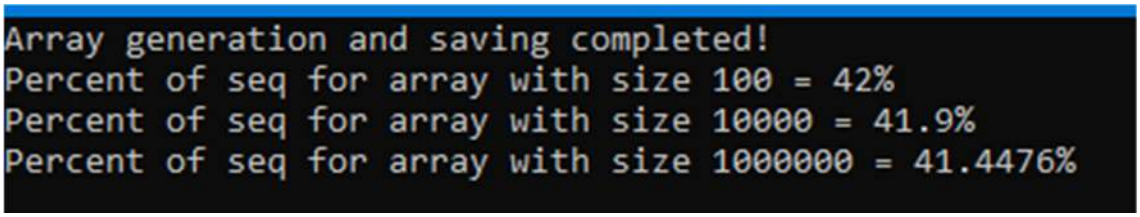

Рисунок 3.2 – Результат тестування масивів на процент кількості відсортованих послідовностей

Додатково було створено декілька функції для зчитування та запису масивів у файл. В результаті маємо програмний додаток котрий вміє не тільки генерувати файли а й тестувати масиви на кількість відсортованих послідовностей що дозволить легко створювати нові файли.

3.3 Вибір алгоритмів та методів сортування для тестування

Коли тестові масиви були зроблені наступним етапом було вибір алгоритмів та методів сортування для дослідження їх продуктивності.

Першим методом сортуванням було обрано сортування бульбашкою. Хоча даний алгоритм сортування показує себе досить погано але він є найвідомішим. Кожен починає своє знайомство з алгоритмами сортування з нього тому кожен повинен не тільки на теорії розуміти що він є досить поганим у швидкодії, але і бачити реальні результати які він показує.

Другий алгоритм сортування це сортування це сортування злиттям. Воно буде використовуватися в різних реалізаціях для заміру продуктивності.

Буде протестована як одно поточна, багато поточна реалізація та реалізація для зовнішнього сортування де даний алгоритм повинен буде себе показати дуже ефективно.

Пірамідальне сортування обране за його стабільну ефективність, незалежно від порядку масиву. Він надійний для великих даних, працює на місці без додаткової пам'яті та ефективний як для випадкових, так і частково впорядкованих масивів.

Останні два алгоритми такі як Timsort та швидке сортування є лідерами по швидкодії в теоретичному описі та використовуються досить часто та є стандартними алгоритмами сортування у деяких мов програмування. Тому вони обов’язково повинні бути порівняні між собою щоб було зрозуміли чи настільки вони ефективні.

Отже в таблиці 3.1 наведено алгоритми та методи сортування з короткою інформацію щодо подальших їх реалізацій.

Таблиця 3.1 Реалізації обраних алгоритмів та методів сортування.

Назва \ Характеристики	Багато поточна	Зовнішнє сортування	Модифікована
Сортування бульбашкою	-	-	+
Сортування злиттям	+	+	-
Пірамідальне сортування	-	-	-
Сортування Шелла	-	-	+
Timsort	+	+	-
Швидке сортування	+	-	+

3.4 Вибір інструментів для замірів продуктивності

Для реалізації поставленої задачі треба для кожного з трьох мов програмування дослідити внутрішні оптимізації які використовуються та

інструменти якими можливо дослідити як швидкодію алгоритму так і замір максимального розміру пам’яті коли алгоритм працює.

Починаючи з продуктивності слід зазначити як саме ми будемо заміряти час від запуску тесту та після виконання алгоритмів та методів сортування. Обраним варіантом є початок дослідження часу безпосередньо перед викликом функції алгоритму або методу та одразу після того як функція повернула результат. Результатом же буде різниця між початковим значенням таймеру який був записаний у змінну та останній таймер який ми можемо як записати в змінну для підвищення читабельності коду так і відразу відняти від початкового таймера. Обраний варіант був другий де ми записуємо в другу змінну значення таймеру після завершення функції.

На рисунку 3.3 наведено порядок виконання інструкцій для заміру часу виконання алгоритмів та методів сортування.

```
const startTime = performance.now();

await method(arr);

const endTime = performance.now();

return endTime - startTime;
```

Рисунок 3.3 – Порядок виконання інструкцій для заміру часу виконання коду

Починаючи реалізацію заміру часу в різних мовах програмування буде використано різні інструменти з різними інтерфейсами взаємодії для отримання часу.

В C++ було обрано бібліотеку <chrono> яка є важливим інструментом для точного вимірювання часу виконання програм.

Вона надає різноманітні можливості для роботи з часом, включаючи точки часу, тривалість та годинники.

Під час виконання програми бібліотека дозволяє отримати точний момент часу перед початком виконання певного коду та після його завершення. Це корисно для аналізу продуктивності, оскільки розробники можуть визначити, скільки часу витрачається на виконання конкретних частин, що допомагає виявити затримки і проблеми.

Вимірювання часу починається з отримання початкової точки часу за допомогою методу high_resolution_clock::now, який повертає час в момент виклику з дуже високою точністю. Після завершення виконання коду отримується фінальна точка часу тим же методом, що дозволяє визначити, коли код закінчив своє виконання.

Далі обчислюється тривалість, яка визначається як різниця між початковою та фінальною точками часу. Цю тривалість можна перетворити у зручний формат, наприклад, у мілісекунди або секунди, за допомогою методу duration_cast. Це дозволяє чітко зрозуміти, скільки часу було витрачено на виконання фрагмента коду.

Застосування бібліотеки chrono надає переваги в точності та зручності, оскільки вона підтримує різноманітні одиниці виміру — секунди, мілісекунди, мікросекунди і навіть наносекунди. Це робить бібліотеку універсальним інструментом для оптимізації програм у C++, допомагаючи розробникам виявляти затримки, підвищувати продуктивність і ефективність коду, а також проводити детальний аналіз часу виконання різних функцій або алгоритмів.

На рисунку 3.4 наведено послідовність інструкцій для отримання часу на мові C++.

```
auto startTime = chrono::high_resolution_clock::now();
methodName = method(arr);
auto endTime   = chrono::high_resolution_clock::now();

chrono::duration<double, milli> duration = endTime - startTime;
```

Рисунок 3.4 – Порядок виконання інструкцій для заміру часу в C++

В JavaScript для заміру часу було використано performance. Метод performance.now у JavaScript призначений для точного вимірювання часу, що минув з моменту початку завантаження веб-сторінки. Він надає значно вищу точність – до мікросекунд, на відміну від методів на кшталт Date.now, які обмежуються точністю до мілісекунд. Важливою особливістю цього методу є те, що він не залежить від змін системного часу, що забезпечує стабільні та надійні вимірювання.

При його використанні спочатку фіксується початковий час перед виконанням операції, після чого, по завершенню процесу, повторно викликається метод для отримання кінцевого часу. Різниця між цими двома показниками дозволяє точно визначити тривалість виконання певної операції, що є важливим інструментом для оцінки продуктивності.

Завдяки високій точності та незалежності від зовнішніх факторів, метод performance.now є оптимальним для вимірювання короткотривалих процесів, таких як виконання алгоритмів або асинхронних операцій у сучасних веб-додатках. На рисунку 3.3 було вже наведено приклад коду для реалізації заміру часу на JS.

Наступним кроком буде написання тестів для того щоб отримувати точні дані щодо використання оперативної пам’яті. Ця задача досить складна якщо ми не будемо використовувати програми які можуть спостерігати скільки пам’яті було використано максимально.

Для виконання було розглянуто декілька стандартних інструментів для моніторингу споживання пам'яті в Windows, таких як Диспетчер завдань та Performance Monitor (Perfmon).

Диспетчер завдань Windows надає можливість отримувати базову інформацію про використання пам'яті окремими процесами в реальному часі. Однак, його функціональність обмежується загальними показниками, і це не завжди достатньо для детального аналізу, особливо коли потрібне більш глибоке розуміння витрат пам'яті на рівні окремих операцій програми.

Обидва інструменти підходять для моніторингу тривалих процесів та

великих систем, однак для більш детального аналізу споживання пам'яті в невеликих або короткотривалих завданнях вони виявилися не такими зручними та недостатньо гнучкими.

На рисунку 3.5 наведено інтерфейс додатку Диспетчеру завдань у розгорнутому режимі.

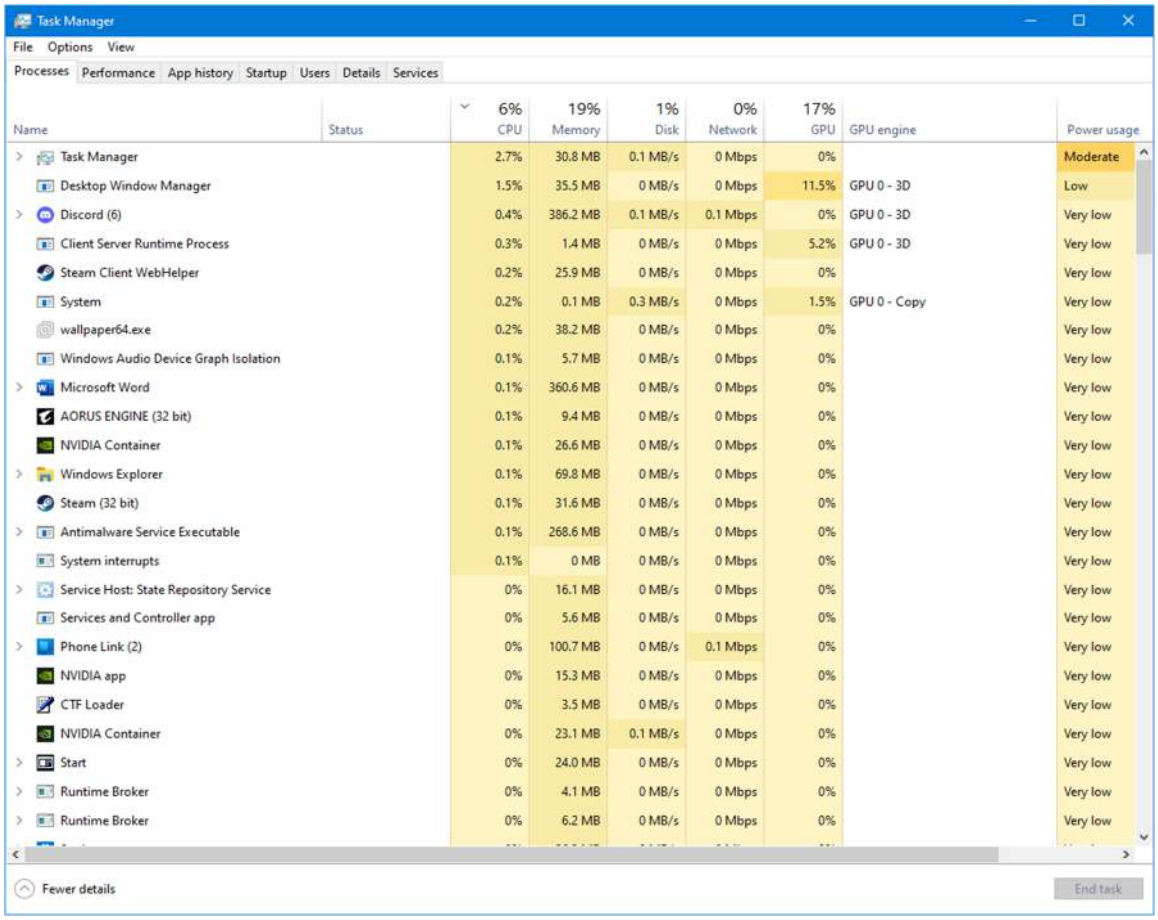

Рисунок 3.5 – Диспетчер завдань

Performance Monitor пропонує розширені можливості моніторингу, даючи змогу налаштовувати лічильники для більш детального відстеження ресурсів. Проте, цей інструмент виявився досить складним у використанні, що ускладнює його застосування для швидкого аналізу короткочасних процесів і для зручного контролю використання пам'яті.

Розглянуті інструменти, такі як Диспетчер завдань Windows та Performance Monitor, не підходять для наших цілей з кількох причин.

По-перше, процес виконується дуже швидко, і ми фізично не зможемо встигнути зафіксувати точне споживання пам'яті процесом у реальному часі. Швидкість виконання настільки висока, що ці інструменти не здатні зловити момент пікового споживання пам'яті під час виконання алгоритму або методу.

По-друге, навіть якщо ми і побачимо якесь споживання пам'яті, це не гарантує, що зафіксований показник буде максимальним. Можливо, пікове використання пам'яті вже минуло або ще не настало на момент фіксації інструментами, оскільки вони оновлюють дані з певною періодичністю. Можна встановити паузу під час виконання алгоритму але при великій кількості елементів таких пауз може бути тисячі а також додавання коду може тільки збільшити розмір процесу а також для JS дозволити збиральнику сміття видалити непотрібні масиви даних що призведе до зменшення споживання процесу та не відповідати вимогам що до тестування алгоритмів та методів.

Крім того, ці інструменти відображають загальне споживання пам'яті всією програмою, що є ще одним недоліком. Для нашого завдання важливо визначити максимальне споживання пам'яті саме окремими алгоритмами та методами, без урахування витрат інших частин програми або системних процесів. Тобто, потрібно мати можливість відстежувати пам'ять на більш деталізованому рівні, що ці інструменти забезпечити не можуть.

Отже, через вищезазначені обмеження стандартних інструментів моніторингу, було прийнято рішення використовувати інший підхід до вимірювання споживання пам'яті. Для того, щоб отримати точні дані про використання пам'яті конкретними алгоритмами або методами, вирішено інтегрувати код для фіксації споживання пам'яті безпосередньо в сам алгоритм чи метод сортування.

Цей підхід дозволяє отримати більш точні результати, оскільки вимірювання буде проводитися в конкретних точках виконання коду. Це означає, що споживання пам'яті можна буде зафіксувати не лише в будь-який момент часу, але й у критичні моменти роботи алгоритму або методу, наприклад, під час пікового навантаження або використання пам'яті.

Вбудовування коду для фіксації дозволить контролювати споживання на більш глибокому рівні, а не просто бачити загальний результат виконання програми.

Також цей метод дасть змогу побачити динаміку змін споживання пам'яті протягом усього циклу роботи алгоритму, а не лише його кінцевий результат. Це дозволить визначити моменти пікового споживання пам'яті та оцінити ефективність кожного окремого етапу алгоритму. Відстежуючи пам'ять безпосередньо всередині алгоритму, можна також виключити інші джерела споживання пам'яті, такі як робота системи чи інші частини програми, що може заважати точному аналізу.

На рисунку 3.6 наведено приклад вбудовування контрольних точок для заміру розміру пам’яті.

```
static void _mergeSort(vector<int>& arr, int left, int right) {
    trackMaxMemoryUsage();

    if (left >= right)
        return;
    int mid = left + (right - left) / 2;
```

Рисунок 3.6 – Контрольна точка для заміру продуктивності

Таким чином, цей підхід дозволить більш точно й детально проаналізувати споживання пам'яті саме на рівні алгоритмів, що є необхідним для повноцінної оцінки їхньої продуктивності й ефективності.

Проте такий підхід із вбудованим тестуванням споживання пам'яті суттєво вплине на швидкість виконання алгоритму. Це пов'язано з необхідністю додавання контрольних точок у критичні моменти, зокрема під час найбільших ітерацій циклів. Для того щоб виміряти обсяг використаної пам'яті в цих точках, спочатку потрібно зробити відповідний запит через інструменти або API, після чого результат зберігається в змінній. Лише після цього алгоритм продовжує виконання.

Така додаткова операція кожного разу уповільнює процес, оскільки програма зупиняється для збору інформації про пам'ять. Це робить загальне виконання алгоритму значно повільнішим. Таким чином, вбудовані механізми контролю споживання пам'яті можуть негативно вплинути на точність оцінки швидкості самого алгоритму.

З огляду на це, було вирішено розробити і тестувати два окремі варіанти алгоритмів. Перший варіант буде «чистим» і служитиме виключно для тестування продуктивності, без втручання у вимірювання пам'яті під час його виконання. Це дозволить оцінити швидкість алгоритму в умовах максимального навантаження без стороннього впливу.

Другий варіант буде включати вбудовані контрольні точки для фіксації споживання пам'яті на різних етапах. Цей алгоритм, хоча й працюватиме повільніше через необхідність постійних звернень для вимірювання пам'яті, забезпечить точні дані про пікові показники використання пам'яті під час виконання. Такий підхід дозволить зафіксувати максимальні значення споживання пам'яті і детально проаналізувати її використання на різних етапах виконання.

Таким чином, тестування двох алгоритмів – одного для продуктивності, іншого для споживання пам'яті – дасть змогу отримати комплексне уявлення про ефективність роботи алгоритму як із точки зору швидкості, так і з точки зору використання системних ресурсів.

Код для відстеження споживання пам'яті в C++ використовує функції Windows API, що дозволяють отримати дані про оперативну пам'ять поточного процесу. Підключаються бібліотеки windows.h і psapi.h, які надають функції для роботи з пам'яттю.

Функція getCurrentMemoryUsage використовує GetProcessMemoryInfo для отримання розміру пам'яті (WorkingSetSize), яку на цей момент використовує процес. Функція trackMaxMemoryUsage перевіряє, чи перевищує поточне використання пам'яті раніше зафіксоване максимальне значення, і якщо так, оновлює його. Це дає можливість динамічно

відстежувати пікові значення споживання пам'яті на різних етапах роботи алгоритму.

Для того щоб виміряти пам'ять в JS буде використано обьект process зокрема його метод memoryUsage. Він містить різноманітну інформацію про поточний процес, зокрема про використання пам'яті. За допомогою методу process.memoryUsage, який повертає об'єкт з кількома полями, ми можемо отримати інформацію про використання різних сегментів пам'яті програмою.

Основне поле, яке використовується в цьому коді, – це heapUsed. Воно показує обсяг пам'яті на купі (heap), що використовується поточним процесом для зберігання об'єктів і змінних. Купа – це область пам'яті, яку JavaScript використовує для динамічного виділення пам'яті під час виконання програми.

На початку виконується збереження початкового значення використання пам'яті, яке отримується через process.memoryUsage().heapUsed. Це дає змогу дізнатися, скільки пам'яті використовувалося перед тим, як запустився алгоритм. Потім викликається метод methodMemo(arr, maxMemoUsage), який виконує певні операції над масивом і відстежує максимальне використання пам'яті під час виконання. У міру того, як метод працює, він порівнює поточне використання пам'яті з уже зафіксованим максимальним значенням і оновлює його, якщо новий показник перевищує попередній. Після завершення роботи методу знову вимірюється використання пам'яті за допомогою process.memoryUsage().heapUsed. Кінцеве значення порівнюється з максимальним, яке було зафіксоване під час виконання, щоб отримати найбільший показник споживання пам'яті.

Після описання всіх додаткових функція для оптимізації процесу розробки та по забезпеченню достовірного запису результатів по файлах наступник кроком була розробка програмного додатка для зчитування результатів з файлу та підрахунок середнього значення часу та пам’яті.

Даний підхід захищає від помилок які можуть виникнути під час ручного вимірювання продуктивності що може привести до недостовірних тестів та помилкових суджень.

4 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ ДОСЛІДЖУВАННИХ МЕТОДІВ ТА АЛГОРИТМІВ СОРТУВАННЯ

4.1 Сортування злиттям

При тестуванні сортування злиттям маємо такі середні показники які наведені у таблиці 4.1.

Таблиця 4.1 Результати тестування сортування злиттям

Мова	R-100	R-10000	R-1000000	S-100	S-10000	S-1000000
C++ 2400	0.0219мс 0б	1.8320мс 38912б	184.61мс 5008663б	0.01984мс 0б	1.7112мс 39321б	167.93мс 5044224б
JS 2400	0.159мс 330125б	4.55мс 2260180б	187.33мс 242898360б	0.165мс 322854b	4.42мс 2191390б	177.12мс 24260895
C++ 3600	0.0222мс 0б	2.1393мс 32768б	182.509мс 4855808б	0.0204мс 0б	1.8965мс 32768б	173.88мс 4857856б
JS 3600	0.15909мс 339385б	4.57684мс 2256301б	182.1918мс 24253400б	0.15249мс 339363б	4.30402мс 2210145б	173.09мс 24257472

При швидкості 2400 C++ відсортовані масиви показують поліпшення продуктивності на 6-9% в порівнянні з випадковими. Наприклад, для масиву на 100 елементів відсортовані дані обробляються на 9.4% швидше (0.0219 мс проти 0.01984 мс). Для 10,000 елементів різниця становить 6.6%, а 1,000,000 – близько 9%. По пам'яті різниця мінімальна – для 10,000 елементів відсортований масив споживає на 1.05% більше пам'яті, для 1,000,000 – на 0.71%. Таким чином, сортування даних призводить до прискорення операцій із незначним збільшенням пам'яті.

При швидкості 3600 C++ відсортовані масиви також демонструють

помітне поліпшення продуктивності в порівнянні з випадковими. Для масиву зі 100 елементів відсортовані дані обробляються на 7.9% швидше (0.02215 мс проти 0.0204 мс). У разі масиву на 10,000 елементів відсортовані дані виконуються на 11.3% швидше, а 1,000,000 елементів – на 4.7% швидше (182.50856 мс проти 173.878 мс).

За обсягом пам'яті різниця також мінімальна: для 10,000 елементів відсортований масив споживає на 0.92% більше пам'яті, для 1,000,000 елементів – на 0.71% більше. Таким чином, на швидкості 3600 сортування даних C++ дає перевагу в швидкості виконання операцій з незначним зростанням споживання пам'яті.

При порівнянні продуктивності C++ на швидкостях 2400 і 3600 спостерігається збільшення ефективності зі зростанням швидкості, особливо великих масивів. Так, для випадковими масиву зі 100 елементів різниця мінімальна – виконання на 3600 трохи довше (0.02215 мс проти 0.0219 мс), що дає приріст всього 1.1%. Однак для масиву на 10,000 елементів час виконання на 3600 зростає на 16.7%.

Для відсортованих масивів приріст продуктивності менш виражений, але стабільно зберігається. Наприклад, відсортований масив на 1,000,000 елементів виконується на 3600 на 5.3% швидше, ніж 2400. Різниця у споживанні пам'яті між швидкостями мінімальна і перевищує 1% всім розмірів масивів. В цілому, перехід на швидкість 3600 призводить до поліпшення продуктивності для великих масивів C++, проте для невеликих масивів ефект майже непомітний.

На швидкості 2400 JS відсортовані масиви показують поліпшення продуктивності на великих масивах, але трохи поступаються на менших. Наприклад, для масиву зі 100 елементів відсортований масив працює на 3.8% повільніше (0.165 мс проти 0.159 мс). У той самий час, для масиву на 10,000 елементів відсортований варіант виконується на 2.9% швидше, а масиву на 1,000,000 елементів – на 5.5% швидше.

У плані використання пам'яті відсортовані масиви також мають невеликі

переваги: для 100 елементів вони вимагають на 2.2% менше, для 10,000 елементів – на 3% менше, а для 1,000,000 елементів – на 0.1% менше. Ці незначні зміни свідчать про мінімальний вплив сортування обсяг пам'яті.

На швидкості 3600 JS демонструє ще помітніше поліпшення продуктивності відсортованих масивів. Для 100 елементів відсортовані дані виконуються на 4.1% швидше (0.15249 мс проти 0.15909 мс), для 10,000 елементів різниця сягає 6%, а 1,000,000 елементів – 5%. Це підтверджує, що зі збільшенням обсягу даних ефективність сортування зростає.

Щодо пам'яті, відмінності мінімальні дуже мінімальні. Для масиву на 100 елементів відсортовані дані використовують на 0.006% менше пам'яті, для 10,000 елементів – на 2.05% менше, а для 1,000,000 елементів різниця становить -0.02%, що говорить про незначне збільшення пам'яті для відсортованих даних.

При швидкості 2400 відсортовані масиви в середньому забезпечують приріст продуктивності на 4.93% у порівнянні з випадковими масивами, з незначною різницею в пам'яті на рівні 1.18%. Це свідчить про те, що навіть при базовій швидкості обробки використання відсортованих масивів надає помітну перевагу в продуктивності з мінімальними додатковими затратами пам'яті.

При підвищенні швидкості до 3600 ефективність відсортованих масивів зростає: середній приріст продуктивності досягає 6.5%, а споживання пам'яті майже не змінюється, із різницею лише в 0.61%. Це показує, що при більш високих швидкостях обробки використання відсортованих масивів стає ще більш виправданим, оскільки вони дозволяють досягти більшого приросту продуктивності, практично не збільшуючи обсяг пам'яті, що використовується.

Відсортовані масиви в середньому працюють на 5.72% швидше за випадкові, при цьому різниця в споживанні пам'яті становить лише 0.89%. Це вказує, що сортування дає значне підвищення продуктивності без помітного збільшення використання пам'яті.

4.2 Сортування Шелла

При тестуванні сортування Шелла маємо такі середні показники які наведені у таблиці 4.2.

Таблиця 4.2 Результати тестування сортування Шелла

Мова	R-100	R-10000	R-1000000	S-100	S-10000	S-1000000
C++ 2400	0.00403мс 0б	0.85мс 0б	125.29мс 0б	0.0045мс 0б	0.63мс 0б	90.73мс 0б
JS 2400	0.15мс 504529б	2.36мс 93756б	131.17мс 233872б	0.13мс 323503б	2.29мс 110849б	96.69мс 142041б
C++ 3600	0.004564мс 0б	0.7802мс 0б	123.76мс 0б	0.0043мс 0б	0.593мс 0б	89.07мс 0б
JS 3600	0.15011мс 378187б	2.3451мс 112400б	129.14мс 161923б	0.1266мс 301186б	2.304мс 54957б	94.612мс 128859б

На швидкості 2400 для C++ відсортовані масиви демонструють значне зростання продуктивності зі збільшенням обсягу даних. Так, для масиву на 10,000 елементів продуктивність підвищується на 25.88%, а для масиву на 1,000,000 елементів – на 27.58%.

Водночас, для невеликого масиву з 100 елементів відсортований масив обробляється на 12.66% повільніше, ніж випадкові. Це свідчить про те, що сортування не завжди оптимальне для малих обсягів даних, проте для великих масивів воно суттєво підвищує швидкість обробки.

На високій швидкості відсортовані масиви стають ефективнішими зі збільшенням розміру даних. Для малих масивів відсортовані дані обробляються на 6.44% швидше, для масиву на 10,000 елементів продуктивність підвищується на 24.07%, а для масиву на 1,000,000 елементів досягає 28.03%. Це свідчить про те, що при великих обсягах даних

відсортовані масиви значно скорочують час обробки, забезпечуючи більш високу ефективність.

Загалом C++ при обробці відсортованих масивів показує, що впорядкування даних стає значною перевагою при роботі з великими обсягами, особливо на швидкостях 2400 і 3600.

На швидкості 2400 відсортовані масиви забезпечують приріст продуктивності на 25.88% для масивів розміром 10,000 елементів і на 27.58% для масивів з 1,000,000 елементів.

Для невеликих масивів з 100 елементів, однак, відсортовані масиви працюють на 12.66% повільніше, що вказує на обмежену ефективність сортування при малих обсягах даних.

На швидкості 3600 тенденція зберігається, і відсортовані масиви ще більше підвищують продуктивність для великих даних. Приріст швидкості для масивів на 10,000 елементів складає 24.07%, а для масивів на 1,000,000 елементів – 28.03%, що свідчить про значне скорочення часу обробки. Для масивів з 100 елементів відсортовані дані обробляються на 6.44% швидше, але ефект менш виражений.

Загалом, при великих обсягах даних і високих швидкостях обробки сортування масивів в C++ є ефективним підходом, який дозволяє досягти суттєвого підвищення продуктивності, тоді як для невеликих масивів ефективність відсортованих даних виявляється менш значною.

Що до JavaScript то на швидкості 2400 показує, що відсортовані масиви стають ефективнішими з ростом розміру даних.

Для масиву з 100 елементів обробка відсортованих даних проходить на 13.33% швидше, для масиву з 10,000 елементів продуктивність покращується на 2.97%, а для масиву на 1,000,000 елементів приріст досягає 26.29%. Це свідчить про те, що сортування даних особливо вигідне при роботі з великими масивами, оскільки значно скорочує час обробки.

При збільшені швидкості до 3600 показує, що відсортовані масиви значно підвищують ефективність обробки, особливо на великих обсягах

даних. Для масиву зі 100 елементів швидкість зростає на 15.64%, для масиву з 10,000 елементів – на 1.76%, а для масиву з 1,000,000 елементів приріст продуктивності досягає 26.73%. Це вказує на те, що відсортовані масиви значно скорочують час обробки при великих розмірах даних, підвищуючи загальну ефективність.

Загальний аналіз продуктивності JS при використанні відсортованих масивів показує, що вони забезпечують відчутне покращення швидкості виконання, особливо для великих обсягів даних.

У середньому, відсортовані масиви обробляються швидше на 13-15% для масивів розміром 100 елементів, приблизно на 2-3% швидше для масивів на 10,000 елементів, і демонструють приріст продуктивності на 26-27% для масивів з 1,000,000 елементів.

Ці результати свідчать про те, що відсортовані дані особливо ефективні при роботі з великими масивами, що дозволяє значно скоротити час обробки і підвищити загальну продуктивність JS у таких умовах.

Загальний аналіз продуктивності C++ та JS при обробці відсортованих масивів показує середній приріст швидкості на 15.51% порівняно з випадковими даними, особливо при великих обсягах даних.

На масивах розміром 10,000 і 1,000,000 елементів приріст продуктивності становить від 24% до 28%, що особливо помітно на швидкості 3600. Для малих масивів (100 елементів) результати менш однозначні, відсортовані масиви можуть працювати повільніше в деяких випадках.

Різниця в споживанні пам'яті мінімальна, лише 0.73%, що вказує на невеликий вплив впорядкування на пам'ять. У цілому, для великих масивів відсортовані послідовності є оптимальним рішенням для підвищення ефективності обробки. При тестуванні використовувалась як раніше було зазначено сортування Шелла з модифікацією, а саме використання для інтервалу між елементами послідовність Циуара яка дозволяє оптимізувати процес розміщення дальніх елементів.

4.3 Пірамідальне сортування

При тестуванні пірамідального сортування маємо такі середні показники які наведені у таблиці 4.3.

Таблиця 4.3 Результати тестування пірамідального сортування

Мова	R-100	R-10000	R-1000000	S-100	S-10000	S-1000000
C++ 2400	0.0034мс 0б	0.66мс 0б	127.33мс 0б	0.0034мс 0б	0.66мс 0б	115.74мс 0б
JS 2400	0.40мс 636906	3.93мс 1925733б	188.32мс 13690946	0.26мс 627682б	3.87мс 1937532	164.24мс 1286810б
C++ 3600	0.0034мс 0б	0.63032мс 0б	123.475мс 4096б	0.00332мс 0б	0.6061мс 0б	112.976мс 4096б
JS 3600	0.2597мс 6093786	3.9924мс 19695666	183.758мс 13043896	0.2857мс 604859б	3.7331мс 1923508	162.473мс 12847386

При аналізі продуктивності на C++ при швидкості 2400 показує, що для невеликих масивів (100 і 10,000 елементів) відсортовані та рандомні масиви обробляються з однаковою швидкістю. Проте для великих масивів обсягом 1,000,000 елементів відсортовані дані забезпечують приріст продуктивності на 9.1%. Це вказує на те, що переваги сортування стають помітними лише при обробці великих обсягів.

При швидкості 3600 показує, що відсортовані масиви забезпечують деяке підвищення ефективності для малих і середніх розмірів даних.
Так, обробка масиву з 100 елементів стає швидшою на 2.64%, масиву з 10,000 елементів – на 3.84%, а для великого масиву на 1,000,000 елементів швидкість зростає на 8.5%. Це свідчить про те, що пірамідальне сортування значно виграє від впорядкування на великих масивах, тоді як на менших масивах покращення є не таким вираженим.

Отже нескладно побачити що переваги відсортованих масивів найбільш

виражені при обробці великих обсягів даних. На швидкості 2400 відсортовані та випадковими даними мають однакову продуктивність для масивів розміром 100 і 10,000 елементів, але для масиву з 1,000,000 елементів відсортовані дані забезпечують приріст продуктивності на 9.1%. Це вказує на помітний ефект від впорядкування при великих обсягах.

Пірамідальне сортування на JS при швидкості 2400 показує значне підвищення ефективності відсортованих масивів, особливо для малих і великих обсягів даних. Для масиву з 100 елементів швидкість зростає на 35%, для масиву з 10,000 елементів – на 1.53%, а для масиву з 1,000,000 елементів – на 12.79%. Це демонструє, що впорядкування масивів є особливо вигідним для малих і великих розмірів, тоді як для середніх обсягів різниця в продуктивності є менш помітною.

При швидкості 3600 показує цікаві результати, а саме для маленьких масивів обсягом 100 елементів відсортовані дані навіть трохи уповільнюють процес – продуктивність падає на 10.02%. Однак ситуація кардинально змінюється для більших масивів.

При обробці масивів з 10,000 елементів швидкість зростає на 6.49%, а для великих масивів на 1,000,000 елементів приріст продуктивності сягає 11.58%. Це доводить, що пірамідальне сортування справді розкриває свій потенціал на великих обсягах даних, тоді як на малих масивах ефект може бути непередбачуваним.

Загальний огляд продуктивності пірамідального сортування на JS показує цікаву картин. На великих обсягах даних, зокрема для масивів на 1,000,000 елементів, відсортовані дані дають суттєвий приріст швидкості – в середньому на 12%. Для середніх масивів обсягом 10,000 елементів також спостерігається виграш у продуктивності на 5-7%, хоча ефект не такий яскравий.

А от для малих масивів, лише на 100 елементів, результат несподіваний – відсортовані дані можуть навіть дещо сповільнювати обробку, призводячи до втрати продуктивності на 10%. Цей контраст підкреслює, що пірамідальне

сортування справді розкриває свої переваги на великих масивах, залишаючи дрібні обсяги в зоні непередбачуваних результатів.

Загальний аналіз продуктивності пірамідального сортування на C++ та JS при різних швидкостях та розмірах масивів демонструє цікаві закономірності, які варто враховувати при виборі цього методу для оптимізації.

Переваги відсортованих масивів найчіткіше проявляються на великих обсягах даних, особливо для масивів розміром 1,000,000 елементів, де приріст продуктивності досягає 9-12% для обох мов програмування. Це вказує на те, що пірамідальне сортування дійсно розкриває свій потенціал при обробці великих масивів, дозволяючи значно скоротити час обробки без значного зростання ресурсів.

Проте для малих масивів розміром 100 елементів ситуація стає менш передбачуваною. Для C++ різниця в продуктивності між випадковими і відсортованими масивами майже відсутня, що означає, що для таких малих обсягів перевага від впорядкування даних не виявляється.

У випадку з JS спостерігається навіть зниження продуктивності для відсортованих масивів – у середньому на 10%. Це може бути пов’язано з додатковими витратами на підтримку структури піраміди для вже впорядкованих даних, які виявляються менш ефективними для обробки малих масивів.

Для середніх масивів, таких як 10,000 елементів, відсортовані масиви також демонструють виграш у продуктивності, але цей ефект менш виражений. JS показує приріст продуктивності від 5% до 7%, тоді як C++ забезпечує покращення на 3-4%.

Хоча цей приріст не є таким суттєвим, як можна побачити на великих масивах, він все ж підкреслює що впорядкування даних може прискорювати виконання сортування даних і для середніх обсягів, особливо у випадках коли потрібна стабільність в водночас зі швидкістю обробки даних в масивах різних величин.

4.4 Швидке сортування

При тестуванні швидкого сортування маємо такі середні показники які наведені у таблиці 4.4.

Таблиця 4.4 Результати тестування швидкого сортування

Мова	R-100	R-10000	R-1000000	S-100	S-10000	S-1000000
C++ 2400	0.00373мс 0б	0.51мс 0б	62.53мс 0б	0.00424мс 0б	0.66мс 0б	2834.1мс 0б
JS 2400	0.39мс 328493б	3.08см 865150б	86.39мс 16663558б	0.28мс 136923б	5.29мс 804573б	3864.1мс 1665208
C++ 3600	0.00423мс 0б	0.4739мс 0б	61.9864мс 0б	0.00427мс 0б	0.74641мс 0б	2901.8мс 0б
JS 3600	0.27551мс 259092б	3.0671мс 865366б	82.4994мс 16643550	0.256мс 142690б	5.2947мс 789766б	3783.5мс 16673000

Швидке сортування на C++ при пониженій швидкості ОП виявляє суттєве зниження продуктивності при обробці великих масивів, у яких значна частина даних вже відсортована.

Для масивів на 100 елементів відсортовані дані обробляються трохи повільніше а саме на 13.67% порівняно з випадковими масивами. Це уповільнення стає ще помітнішим для масивів на 10,000 елементів, де швидкість знижується на 29.41% для відсортованих послідовностей. Найбільша різниця спостерігається для масивів на 1,000,000 елементів, де відсортовані масиви обробляються на 4432.43% повільніше, що вказує на значне падіння продуктивності.

Такі результати показують, що швидке сортування не підходить для обробки великих масивів з частково відсортованими даними, особливо коли

такі послідовності займають значну частину масиву.

Аналіз швидкого сортування на C++ при швидкості 3600 демонструє дивний парадокс, чим більше відсортований масив, тим більше проблем виникає у швидкого сортування. На малих масивах з 100 елементів різниця майже непомітна – відсортовані дані обробляються лише на 0.97% повільніше. Однак на масивах з 10,000 елементів починаються помітні сповільнення де відсортовані масиви втрачають у швидкості вже 57.5%. Для великих масивів результати погані. Масиви з відсортованими послідовностями обробляються на 4581.4% повільніше.

Швидке сортування на JavaScript показує також погані результати у деяких випадках. Для невеликих масивів на 100 елементів відсортовані послідовності забезпечують перевагу, скорочуючи час обробки на 28.2% порівняно з випадковими даними. Це вказує на здатність алгоритму ефективно працювати з малими обсягами навіть за умов часткового впорядкування. Однак із зростанням розміру масиву ситуація кардинально змінюється. На масивах з 10,000 елементів відсортовані дані викликають значне сповільнення, знижуючи швидкість на 71.75%. Найбільше це помітно як і у випадку C++ для масивів на 1,000,000 елементів, де відсортовані послідовності обробляються на 4372.89% повільніше.

При зміні швидкості ОП тенденція зниження продуктивності при масивах з відсортованими послідовностями як у випадку з C++ зберігається. Тому не складно обачити знову різкі контрасти в продуктивності залежно від розміру масиву та ступеня впорядкованості даних.

Для масивів малих масивів алгоритм демонструє помірне покращення а саме на 7.08% швидше порівняно з випадковими даними. Однак із зростанням обсягу даних продуктивність швидкого сортування починає стрімко падати: для масивів на 10,000 елементів швидкість обробки знижується вже на 72.63%, коли дані впорядковані.

Особливо помітне зниження спостерігається для великих масивів розміром 1,000,000 елементів, де відсортовані послідовності призводять до

уповільнення обробки на колосальні 4486.04%. Така тенденція свідчить про те, що швидке сортування, хоч і може забезпечити швидку обробку малих масивів, втрачає ефективність на великих обсягах із частково впорядкованими даними. Ці результати підкреслюють важливість врахування структури даних при виборі алгоритму, адже в певних умовах швидке сортування може суттєво втрачати свою перевагу.

Загальний аналіз продуктивності швидкого сортування на C++ та JavaScript в умовах різних розмірів масивів і ступеня впорядкованості даних чітко демонструє його переваги та обмеження. При обробці малих масивів відсортовані дані мають незначний вплив на швидкість роботи алгоритму. В середньому відсортовані масиви обробляються лише на 12.48% повільніше або швидше, залежно від платформи, що вказує на прийнятну ефективність швидкого сортування в умовах малого обсягу. Проте зі збільшенням обсягу даних продуктивність швидкого сортування починає помітно знижуватися.

Для середніх масивів відсортовані послідовності вже призводять до зниження швидкодії на 57.82% у середньому. Це означає, що навіть часткова впорядкованість значно ускладнює роботу алгоритму, що стає очевидним на середніх обсягах даних.

Найбільші втрати продуктивності спостерігаються при обробці великих масивів. У таких умовах впорядкованість даних середнє уповільнення для відсортованих послідовностей досягає 4468.19%. Це вказує на критичне зниження ефективності алгоритму де часткова впорядкованість створює надмірні витрати обчислювальних ресурсів і значно сповільнює процес сортування.

Загалом, ці результати підкреслюють важливість врахування розміру масиву та ступеня впорядкованості даних при виборі алгоритму сортування. Швидке сортування ефективне для масивів що не містять багато впорядкованих послідовностей. Це показує, що жоден алгоритм не є універсальним, і вибір повинен враховувати конкретні характеристики даних, щоб уникнути значного зниження швидкості обробки та оптимізувати

використання ресурсів.

4.5 Timsort

При тестуванні Timsort маємо такі середні показники які наведені у таблиці 4.4.

Таблиця 4.5 Результати тестування Timsort

Мова	R-100	R-10000	R-1000000	S-100	S-10000	S-1000000
C++ 2400	0.00377мс 0б	0.54мс 37274б	75.04мс 37552136	0.00339мс 0б	0.47мс 38502б	61.32мс 382853 1б
JS 2400	0.17мс 166991б	4.75мс 822177б	141.97мс 28491227	0.17мс 166164б	4.67мс 820509б	116.85мс 2851590 9б
C++ 3600	0.0042мс 0б	0.55595мс 48742б	74.0651мс 39432196	0.00349мс 0б	0.4752мс 33587б	61.6561мс 425113 6б
JS 3600	0.17593мс 222542б	4.63586мс 830584б	131.402мс 28473300	0.16741мс 257575б	4.6019мс 847613б	113.003мс 28492300

Продуктивності Timsort на C++ демонструє стабільну перевагу при обробці відсортованих масивів. На малих масивах обсягом 100 елементів відсортовані дані забезпечують приріст продуктивності на 10.08% порівняно з випадковими даними.

Зі збільшенням обсягу масиву ефективність Timsort стає ще помітнішою а саме для масивів на 10,000 елементів швидкість обробки відсортованих даних зростає на 12.96%. Найбільш виражена різниця спостерігається на великих масивах розміром 1,000,000 елементів, де відсортовані дані обробляються швидше на 18.28%.

Ці результати підкреслюють, що Timsort, завдяки своїй оптимізації для впорядкованих послідовностей, демонструє високу ефективність при роботі з

великими масивами, де упорядкованість даних може значно покращити швидкодію.

При збільшені частоти ОП до 3600 алгоритм показує чітку тенденцію до підвищення ефективності при роботі з відсортованими масивами різного обсягу. На малих масивах обсягом 100 елементів приріст швидкості обробки відсортованих даних становить 16.9%, що підкреслює перевагу алгоритму в умовах невеликих обсягів. Зі збільшенням розміру масиву Timsort продовжує демонструвати ефективність.

Для середніх масивів продуктивність покращується на 14.52%, а для великих масивів із 1,000,000 елементів різниця досягає 16.75% що демонструє тенденцію що при збільшенні даних продуктивність збільшується так як до того виділяєтеся час для формування відсортованих частин завдяки сортуванню вставками.

Отже Timsort на C++ стабільно демонструє високу ефективність при обробці відсортованих масивів на різних частотах ОП і розмірах даних. На низькій частоті 2400 він забезпечує приріст швидкості обробки відсортованих даних на 10.08% для малих масивів, на 12.96% для середніх і на 18.28% для великих.

При збільшенні частоти до 3600 Timsort покращує продуктивність ще більше: 16.9% для малих масивів, 14.52% для середніх та 16.75% для великих. Це робить Timsort оптимальним вибором для великих впорядкованих масивів, де він стабільно забезпечує високу швидкодію.

Щодо реалізації на JS то при швидкості 2400 демонструє стабільну продуктивність з помітними перевагами для великих масивів, особливо при обробці відсортованих даних. Для малих масивів обсягом 100 елементів відсортовані та випадкові дані обробляються з однаковою швидкістю, що вказує на відсутність значущого впливу структури даних на продуктивність.

Проте, на масивах середнього розміру (10,000 елементів) Timsort починає демонструвати переваги – обробка відсортованих послідовностей забезпечує приріст продуктивності на 1.68%. Найбільше зростання

ефективності спостерігається для великих масивів на 1,000,000 елементів, де впорядковані дані скорочують час обробки на 17.69%.

При швидкості 3600 демонструє помітні переваги при роботі з відсортованими даними, особливо на великих масивах. Для невеликих масивів обсягом 100 елементів відсортовані дані забезпечують покращення продуктивності на 4.84% – невелике, але показове підвищення, яке свідчить про здатність алгоритму оптимізувати роботу навіть на невеликих обсягах. На масивах середнього розміру (10,000 елементів) різниця у швидкодії майже непомітна, становлячи лише 0.73%, що вказує на стабільну роботу Timsort без значної залежності від впорядкованості даних.

Timsort на C++ і JS демонструє вражаючу стабільність і помітне підвищення продуктивності, особливо на великих масивах, де упорядкованість даних стає вирішальною. На C++ при частоті 2400 приріст швидкості обробки відсортованих масивів становить 10.08% для малих масивів, 12.96% для середніх і досягає 18.28% на великих. Коли частота збільшується до 3600, приріст стає ще виразнішим: 16.9% для малих, 14.52% для середніх і 16.75% для великих масивів, що підкреслює гнучкість алгоритму при різних умовах навантаження.

JS також показує себе з найкращого боку при обробці великих масивів. На частоті 2400 Timsort забезпечує значне скорочення часу обробки – 17.69% для великих масивів, тоді як для середніх приріст складає 1.68%.

При збільшенні частоти до 3600 відсортовані дані на великих масивах обробляються швидше на 14.0%, а для малих масивів ефективність підвищується на 4.84%, демонструючи плавність і адаптивність алгоритму навіть при невеликих масивах.

Загалом, Timsort показує себе як надійний інструмент для великих і частково впорядкованих масивів, де упорядкованість дозволяє значно прискорити обробку. Його стабільність і адаптивність до різних розмірів масивів і швидкостей процесора роблять Timsort оптимальним вибором для тих, хто шукає баланс між швидкодією і надійністю хоча сам алгоритм

потребує більшу структуру коду що робить його менш легким для модифікацій.

4.6 Сортування бульбашкою

При тестуванні сортування бульбашкою маємо такі середні показники які наведені у таблиці 4.6.

Таблиця 4.6 Результати тестування Сортування бульбашкою

Мова	R-100	R-10000	R-1000000	S-100	S-10000	S-1000000
C++ 2400	0.0062мс 0б	30.7445мс 0б	997423мс 0б	0.00539мс 0б	28.471мс 0б	624907мс 0б
JS 2400	0.5806мс 803617б	86.6339мс 1957448б	1540075мс 644296	0.5332мс 674186б	83.841мс 873458б	1115227мс 1918881б
C++ 3600	0.0075мс 0б	35.3695мс 0б	929714мс 0б	0.00607мс 0б	34.328мс 0б	600404мс 0б
JS 3600	0.5995мс 691410б	86.001мс 1777045б	1583749мс 1087240б	0.55018мс 629558б	84.698мс 84.69834	1097864мс 371632б

Так як сортування бульбашкою має модифікацію яка перевіряє чи було зроблено переміщення за ітерацію то це дозволяє оптимізувати випадки коли маємо відсортований масив.

Для невеликих масивів обсягом 100 елементів відсортовані дані дозволяють скоротити час обробки на 13.2%, що вказує на певну адаптивність алгоритму навіть при обробці малих обсягів. При переході до середніх масивів (10,000 елементів) продуктивність покращується на 7.37%, що також свідчить про стабільний ефект від впорядкованості даних.

Найвиразніші результати спостерігаються при роботі з великими масивами розміром 1,000,000 елементів. Тут відсортовані дані забезпечують приріст швидкості обробки на 37.35%. Така динаміка підкреслює, що

сортування бульбашкою, незважаючи на свою простоту, здатне ефективно обробляти великі масиви, коли дані вже частково впорядковані, демонструючи стабільне підвищення продуктивності завдяки впорядкованості.

При швидкості 3600 демонструє цікаві результати, особливо коли дані вже частково впорядковані. На малих масивах обсягом 100 елементів відсортовані дані дозволяють підвищити продуктивність на 19.28%, що підкреслює здатність алгоритму швидко адаптуватися навіть до невеликих упорядкованих послідовностей. У випадку середніх масивів (10,000 елементів) приріст продуктивності більш помірний – 2.94%, що свідчить про стабільну роботу алгоритму без значних залежностей від упорядкованості.

Як і у випадку при швидкості 2400 на великих масивах обсягом 1,000,000 елементів відсортовані дані забезпечують скорочення часу обробки на 35.42%. Така динаміка свідчить про те, що сортування бульбашкою, хоча й відоме своєю базовою реалізацією, демонструє вражаючу адаптивність до великих обсягів даних, особливо коли ці дані вже частково впорядковані. Це робить його доцільним вибором у ситуаціях, де структура даних сприяє швидкій обробці, навіть у межах простого алгоритму.

JavaScript при швидкості ОП 2400 демонструє результати які схожі з тенденціями які ми отримали при тестуванні на C++. Для малих масивів відсортовані дані забезпечують приріст продуктивності на 8.16%, що вказує на невелику, але стабільну оптимізацію навіть на незначних обсягах.

На середніх масивах приріст продуктивності складає 3.22%, підтверджуючи адаптивність алгоритму, що реагує на упорядкованість без значних коливань у швидкості.

Найпомітніші покращення спостерігаються при роботі з великими масивами обсягом 1,000,000 елементів, де відсортовані дані скорочують час обробки на 27.59%. Це підкреслює, що сортування бульбашкою, попри свою простоту, може ефективно працювати з великими обсягами частково впорядкованих даних, демонструючи значний приріст у продуктивності за рахунок структури масиву.

Сортування бульбашкою на JavaScript при швидкості 3600 демонструє вражаючі покращення продуктивності, особливо на великих масивах із частково впорядкованими даними. Для невеликих масивів обсягом 100 елементів відсортовані дані забезпечують зростання швидкості на 8.22%, що свідчить про помірну, але стабільну оптимізацію навіть для невеликих обсягів. На масивах середнього розміру (10,000 елементів) приріст продуктивності становить 1.51% – незначний, але показовий результат, який підтверджує стабільність роботи алгоритму незалежно від структури даних.

Найбільше зростання швидкості спостерігається на масивах із 1,000,000 елементів, де відсортованість дозволяє прискорити обробку на 30.68%. Це свідчить про те, що сортування бульбашкою, хоча й простий алгоритм, здатний показати відчутну ефективність на великих масивах із частково впорядкованими даними, роблячи його надійним інструментом у відповідних умовах.

Аналізуючи результати продуктивності сортування бульбашкою на C++ та JavaScript при різних швидкостях обробки даних, можна побачити загальні тенденції, що підкреслюють адаптивність цього алгоритму до частково впорядкованих даних.

На невеликих масивах загальний приріст продуктивності в середньому становить близько 12% для обох мов, що підкреслює базову оптимізацію навіть при невеликих обсягах. На середніх масивах розміром 10,000 елементів загальна ефективність зростає приблизно на 4-5%, свідчачи про стабільність роботи алгоритму без значних коливань у швидкості залежно від мови програмування.

Однак найбільше зростання продуктивності спостерігається при роботі з масивами обсягом 1,000,000 елементів. Для таких великих масивів відсортованість дозволяє скоротити час обробки в середньому на 30-35% між двома мовами, що робить сортування бульбашкою ефективним вибором для таких ситуацій.

Отже попри свою простоту, алгоритм сортування бульбашкою здатний

значно покращити швидкість обробки великих масивів із частково впорядкованими даними, роблячи його актуальним і надійним рішенням для завдань, де впорядкованість масивів відіграє ключову роль.

4.7 Аналіз продуктивності в багатопоточній реалізації

При тестуванні алгоритмів загалом при реалізації багатопоточних реалізацій використовувався підхід з розподілом частин масиву на кількість потоків процесору. Даний підхід є найбільш оптимальним так як зменшує кількість створених потоків що не тільки можуть уповільнити виконання алгоритму а й збільшити витрату оперативної пам'яті на внутрішньої структури для реалізації потоків і очікування кінця дочірніх процесів.

Загалом якщо додавати кожне ядро для обчислення частини масиву то процес сортування зростає в 20-40%. Але це тільки у випадку коли ми використовуємо тільки 1 ядро без використання його іншого потоку. Якщо додається інший потік ядра то додаткове пришвидшення падає до 5-15% тому що компоненти ядра вже зайняті та так як задача однотипна то загалом в більшості випадків потрібні однакові блоки.

Але ця статистика справедлива тільки у випадку C++. Так як спочатку JS не був створений з метою виконання у багатопоточному режимі його функціональність на дуже низькому рівні. Основна проблема виникаю коли ми передаємо частину масиву у поток.

У випадку C++ ми передаємо частину по посиланню без створення копії що забезпечує швидкодію та мінімальне використання ОП. У випадку JS ми не можемо передавати масив по посиланню у воркері що викликає колосальні навантаження. По перше копіювання частини масиву не тільки досить важка операцію, а в додаток до цього вона створює в ОП ще один масив що в результаті може збільшити використання ОП.

Тому не складно зробити такі висновки щодо використання багатопотоних технологій для реалізаціях алгоритмів та методів сортування.

Їх використання зменшує час виконання алгоритмів та методів, але у випадку алгоритмів які створюю додаткові масиви як сортування злиттям, вони підвищують пікове використання пам’яті. Також збільшується кількість змінних які використовуються що злегка збільшує використання стеку.

4.8 Аналіз продуктивності при зовнішньому сортуванні

В зовнішньому сортуванні головну роль виконує диск на якому виконується читання та запис відсортованих частин. Як було вже зазначено в системі встановлено SSD у форматі M2 який має дуже високу частоту читання та запису порівняно з стандартним жорстким диском.

Отже маємо такі результати для сортування злиттям та timsort:

- продуктивність знизилась так як потрібно кожен відсортований масив записувати у файла що досить довго так як ми прямцюємо не з ОП а записуємо на диск. У випадку сортування злиттям продуктивність знизилась у 9.3 рази а саме 1773 мілі секунд проти 171 у не зовнішньому варіанті;

- для timsort знизилась у 25 разів а саме 1655 проти 61. Хоча на перший погляд здається що проблема в алгоритмі але якщо порівняти с злиттям то маємо покращення продуктивності але в більшості випадків слабим місцем буде диск;

- споживання пам'яті у зовнішнього сортування очікувано знизилося. Оскільки велика частина масиву зберігається у файлі, а сортування відбувається з маленькими масивами або вже відсортованими це дозволяє значно скорочувати споживання оперативної пам'яті. Для сортування злиттям використання ОП зменшилось в 8.8 разів а для timsort у 8.6 разів.

Отже нескладно побачити що для систем в яких є обмеження в оперативної та віртуальної пам’яті даний вид сортування може вирішити їх проблеми.

4.9 Загальний аналіз результатів

Після аналізу отриманих результатів для кожного розібраного алгоритму або методу сортування ми отримало багато даних що до їх продуктивності при використанні при різних умовах.

В таблиці 4.7 наведено загальний аналіз методів та алгоритмів сортування а саме середню продуктивність для різних за розміром масивів, різницю між випадковими даними та впорядкованими послідовностями, пришвидшення при зміні частоти та таймінгів оперативної пам’яті та різниця між продуктивність на різних мовах.

Таблиця 4.7 – Загальний аналіз методів та алгоритмів сортування

	100ел.	10000ел.	1000000ел.	R-S	Пам’ять	JS-C++
Сортування злиттям	0.0899 мс	3.18мс	178.58мс	5.72%	0.34%	3.02%
Сортування Шелла	0.0718 мс	1.519мс	110.06мс	15.51%	1.35%	6.89%
Пірамідальне сортування	0.152 мс	2.26мс	147.29мс	6.79%	4.23%	48.42%
Швидке сортування	0.152 мс	2.39мс	1709.62мс	-1507%	1.94%	33.63%
Timsort	0.0873 мс	2.59мс	96.91мс	10.71%	0.42%	90.62%
Сортування бульбашкою	0.286 мс	58.76мс	1061170мс	16.25%	5.55%	69.30%

Загальна вплив наявністю відсортованих послідовностей сильно впливало для деяких алгоритмів та методів. У середньому зміна була в 240% але якщо не брати у розрахунок найгірший випадок у швидкому сортуванні то 10.196%. Найбільший вплив був для алгоритму швидкого сортування яка

сповільнилося на 1506.95%, а у випадку відсортованих послідовностей сортування бульбашкою стало виконатися на 16.25% швидше.

У випадку з швидким сортування проблемою такого розриву було наявність відсортованих послідовностей які робили проблеми для алгоритму для вибіру опорних елементів.

Розглядаючи продуктивність між мовами програмування то можна побачити що деякі алгоритми або методи майже не мають розбіжності у продуктивності між мовами. Це обумовлено оптимізацією just-in-time компіляцій в JS. Коли код виконується багато раз то тип для змінних буде с кожним разом все швидше і швидше встановлювати тип. Тому для різних алгоритмів та методів оптимізація виконується досить не рівномірно тому й різниця між мовами навіть при однакових реалізаціях алгоритму відрізняється на 41.98%.

Що до змінення частоти оперативної памяті то не для всіх алгоритмів або методів це принесло суттєве прискорення. Сучасні компілятори гарно оптимізовані та можуть звичайні змінні або іншу інформацію записувати в регістри що суттєво прискорює програму тому як треба робити менше звернень до оперативної пам’яті. В середньому прискорення дорівнює 2.31%. Для алгоритму сортування бульбашкою було найбільше прискорення так як даний алгоритм дуже часто використовує оперативну пам'ять так як дуже багато звернень до неї.

На рисунку 4.1 наведено продуктивність методів та алгоритмів при сортуванні малих масивів.

Не складно побачити що для малих масивів найшвидшим був алгоритм сортування Шелла с послідовністю Циуара. Даний алгоритм дуже простий та має мало змінних що забезпечує швидкий початок сортування без розподілу частин масиву. Сортування бульбашкою має найгіршу продуктивність що ще раз доказує що даний алгоритм завжди програє на любих за розміром масивах.

Рисунок 4.1 – Продуктивність методів та алгоритмів на малих масивах

На рисунку 4.2 наведено продуктивність алгоритмів та методів на середніх за розміром масивах.

Рисунок 4.2 – Продуктивність методів та алгоритмів на середніх масивах

Не складно побачити що навіть для масивів в 10000 елементів сортування алгоритмом Шелла з послідовністю Циуара є найшвидшою але це у випадку якщо timsort виконується на JS тому як на C++ тим сорт має перевагу. Найгіршу продуктивність має сортування бульбашкою що вже демонструє велику прорву продуктивності між собою та останнім за продуктивність алгоритмом сортування злиттям. Інші алгоритми сортування демонструють практично однакову швидкість виконання.

Останнім буде розглянуто продуктивність для великих масивів на 1000000 елементів. На рисунку 4.3 наведено продуктивність для таких масивів та алгоритмів та методів що сортують.

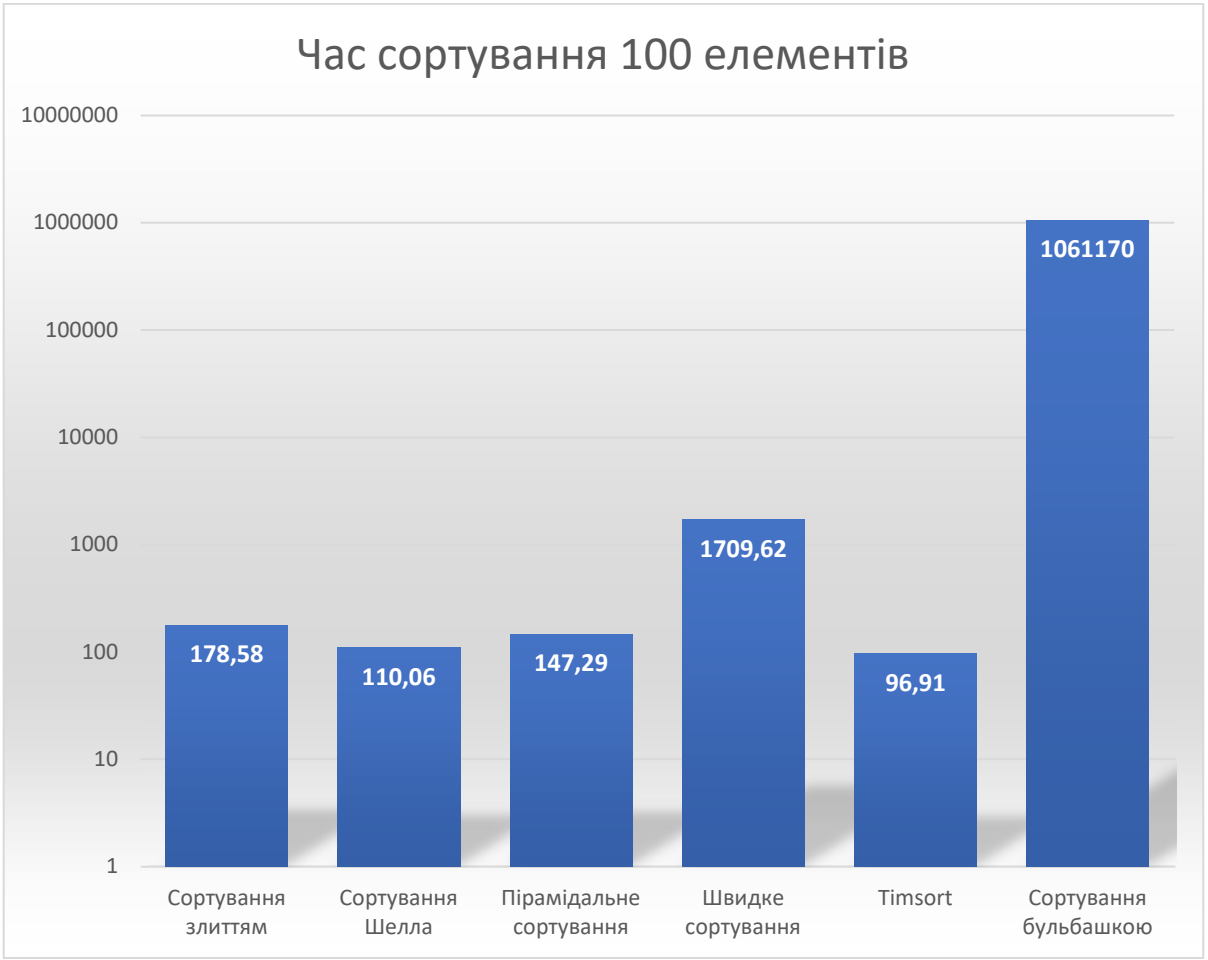

Рисунок 4.3 – Продуктивність методів та алгоритмів на великих масивах

Коли масиви набувають великої кількості елементів то сортування

timsort починає переважати у продуктивності над сортуванням Шелла. Якщо розглядати продуктивність то тем більше елементів тим більше timsort починає переважати над іншими алгоритмами. Також на рисунку можна побачити наскільки сортування бульбашкою має великі проблеми з часом сортування масиву.

4.10 Рекомендації для алгоритмів та методів сортування

Отже після того як було розглянуто всі дані які були отримані під час тестування алгоритмів та методів можна зробити загальні рекомендації щодо кожного алгоритми сортування.

Сортування злиттям надає гарну продуктивність але потребую додаткову пам'ять що може призвести до проблем якщо масив занадто великий. Алгоритм досить простий на теорії але потребує багато коду для реалізації що робить його не досить зручним для модифікації. Цей алгоритм є стабільним та має реалізацію для роботи зовні.

Сортування Шелла виявилось дуже ефектним для сортування малих та середніх масивів при тому що має досить легкий для розуміння алгоритм, а саме сортування вставками але з проміжками. Він практично не потребую додаткової ОП та має гарні показники приросту ефективності якщо масив має відсортовані послідовності. Він також дає стабільні результати на різних мовах програмування.

Пірамідальне сортування має непогані показники за різних за розмірами масивах. На малих вона досить повільна так як її алгоритм потребує додаткового часу для розбору на бінарне дерево тому її продуктивність для них досить низька але на середніх масивах вона дуже висока та середня для великих масивів.

Сам алгоритм потребує додаткову ОП але це досить низькі потреби щоб зациклюватися на цьому мінусі. Алгоритм непогано пришвидшився після підвищення частоти оперативної пам’яті. Реалізація для JS не така

продуктивна що даний алгоритм може досить сильно змінювати продуктивність на різних мовах що робить його не універсальним вибором с точки зору інструментів які надає мова програмування.

Швидке сортування хоча і дуже ефективна на масивах з випадковими елементами, у випадку відсортованих послідовностей вона показує дуже погану продуктивність. Ще одна проблема виникає під час сортування великого масиву з відсортованими послідовностями а саме те що виникає помилка переповнення стеку як у C++ так і в JS. Через це потрібно переписати з рекурсивної реалізації на стек та додатковий цикл щоб уникнути цю проблему. Пам'ять практично не використовується. Отже якщо використовувати алгоритм на випадкових масивах то він показує гарну продуктивність.

Наступний алгоритм timsort э дуже популярним за свою ефективність та стабільність. В результатах не складно побачити що він є практично лідером і завжди випереджає інші алгоритми чи методи. Для великих масивів він є найшвидшим і дивлячись на результати можна зробити висновок що зі збільшенням масиву він буде все більше випереджати в продуктивності конкурентів. Алгоритм потребує невелику кількість додаткової пам’яті тому як він використовує метод злиття.

Останнім алгоритмом було сортування бульбашкою. Даний алгоритм в більшості випадків краще не використовувати. Алгоритм є найлегшим та інтуїтивно зрозумілим що робить його гарним алгоритмом сортування для початківців щоб познайомитися з концепцією алгоритмів у програмуванні. З його переваг є те що він не використовує додаткову ОП так весь код займає не більше 8 строк коду.

ВИСНОВКИ

В даній роботі було досліджено продуктивність алгоритмів сортування за різних умов використання але для цього потрібно було зробити комплексну роботу для того щоб достигнути поставленої мети. Було розглянуто багато методів та алгоритмів сортування та проаналізовано їх алгоритм, побудовано блок схему а також розглянуто слабкі та сильні сторони кожного.

Перед початком тестування було налаштовано систему для стабільних результатів під час тестування щоб уникнути випадків коли виникає задача з більшим пріоритетом під час виконання сортування.

Для самого тестування було обрано шість алгоритмів за різними критеріями. Для декількох алгоритмів та методів сортування було розроблено декілька спеціальних реалізацій таких як зовнішнє та паралельне сортування.

Для пришвидшення сортування було розроблено багато допоміжних функцій щоб пришвидшити процес тестування та уникнути помилок при обчисленні результатів.

В результаті було проведено на основі отриманих результатів дослідження продуктивності для кожного обраного методу і алгоритму та написано рекомендації що до використання. Не складно побачити що деякі алгоритми в результаті демонстрували іншу продуктивність порівняно з тим що описано в теоретичних матеріалах.

Отже дослідження повністю виконало поставлено мету а саме продемонструвати та проаналізувати яку продуктивність алгоритми можуть дати якщо виконувати їх на одному тестовому стенді та змінювати тільки обставини за якими вони виконуються. Хоча під час процесу виникали проблеми з реалізацією на різних мовах програмування чи з кодом алгоритмів та методів, вони були вирішені оптимальними шляхами хоча і потребували додаткового часу для аналізу.

ПЕРЕЛІК ПОСИЛАНЬ

1. Bielytskyi D.M., Yeromina N.S. Ponomarenko O.M. IN SEARCH OF THE PERFECT ALGORITHM: THE INSOLUBLE RIDDLE OF COMPUTER TECHNOLOGY // Проблеми інформатизації : XII міжнародна науково-технічна конференція. - 21-22 листопада 2024. –с.78. doi: https://doi.org/10.32620/PI.24.t2

2. Bielytskyi, D., & Yeromina, N. (2024). Research on the Performance of Sorting Methods and Algorithms under Various Usage Conditions. *COMPUTER AND INFORMATION SYSTEMS AND TECHNOLOGIES*, *2024*, 17–18.

3. Kwardem D. Algorithm. WestBow Press, 2022. 262 p.

4. Astrachan O. Bubble sort. *ACM SIGCSE Bulletin*. 2003. Vol. 35, no. 1. P. 1–5. URL: https://doi.org/10.1145/792548.611918.

5. Weiss M. A., Sedgewick R. Bad cases for shaker-sort. *Information Processing Letters*. 1988. Vol. 28, no. 3. P. 133–136. URL: https://doi.org/10.1016/0020-0190(88)90158-5.

6. Insertion Sort. *Sorting*. Hoboken, NJ, USA, 2011. P. 83–102. URL: https://doi.org/10.1002/9781118032886.ch2.

7. Shell Sort. *Sorting*. Hoboken, NJ, USA, 2011. P. 103–128. URL: https://doi.org/10.1002/9781118032886.ch3.

8. McLuckie K., Barber A. Binary Tree Sort. *Sorting Routines for Microcomputers*. London, 1986. P. 58–67. URL: https://doi.org/10.1007/978-1-349-08147-9_4.

9. Levcopoulos C., Petersson O. Adaptive Heapsort. *Journal of Algorithms*. 1993. Vol. 14, no. 3. P. 395–413. URL: https://doi.org/10.1006/jagm.1993.1021.

10. McCool M., Robison A. D., Reinders J. Merge Sort. *Structured Parallel Programming*. 2012. P. 299–305. URL: https://doi.org/10.1016/b978-0-12-415993-8.00013-x.

11. Steier D. M., Anderson A. P. Quicksort. *Algorithm Synthesis: A*

Comparative Study. New York, NY, 1989. P. 24–38. URL: https://doi.org/10.1007/978-1-4613-8877-7_3.

12. Aumüller M., Dietzfelbinger M. Optimal Partitioning for Dual Pivot Quicksort. *Automata, Languages, and Programming*. Berlin, Heidelberg, 2013. P. 33–44. URL: https://doi.org/10.1007/978-3-642-39206-1_4.

13. Python Software Foundation. (2024, November 2). *Python 3.14 Tutorial*. Python.org. https://docs.python.org/3.14/tutorial/index.html

14. Queinnec, C. (2017). Javascript. *Technologies logicielles Architectures des systèmes*. https://doi.org/10.51257/a-v2-h3120

15. Ficarra, M., Gibbons, K., & Guo, S.-Y. (2023, June 27). *ECMAScript® 2023 Language Specification*. Ecma International. https://262.ecma-international.org/14.0/