МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ТІТОВ ІВАН СЕМЕНОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
канд. техн. наук, доцент

_____О. В. Зелінська
«_ __ »_____2024 р.

ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ КРОСПЛАТФОРМОВИХ МОВ ПРОГРАМУВАННЯ В ЗАДАЧАХ ОБРОБКИ ДАНИХ

Спеціальність 122 «Комп'ютерні науки»

Кваліфікаційна (магістерська) робота

Науковий керівник:
Р.М. Бабаков , професор кафедри
інформаційних технологій,
д.т.н., доцент
_____

Оцінка: ___/___/_____
(бали за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця 2024

АНОТАЦІЯ

Тітов І.С. Визначення ефективності кросплатформових мов програмування в задачах обробки даних. Спеціальність 122 “Комп’ютерні науки”, Освітня програма “Data science”. Донецький національний університет імені Василя Стуса, Вінниця, 2024.

Магістерська робота присвячена визначенню ефективності кросплатформових мов програмування у контексті задач обробки даних. Актуальність дослідження зумовлена необхідністю оптимізації процесів обробки великих обсягів даних в умовах швидкого розвитку інформаційних технологій та вимог до високої продуктивності програмного забезпечення, яке повинно працювати на різних операційних системах. В роботі проведено порівняння популярних кросплатформових мов програмування, таких як Python, JavaScript, Java та інших, з метою оцінки їх здатності до ефективної обробки великих даних, виконання паралельних обчислень та інтеграції з різноманітними платформами.

Методологія дослідження включає експериментальне вимірювання часу виконання задач, споживаної пам’яті, масштабованості рішень та зручності інтеграції з різними інструментами обробки даних. В рамках роботи розглянуті різні підходи до паралелізації обробки даних, а також обмеження та переваги кожної з мов програмування в контексті реальних умов використання.

Результати дослідження можуть бути корисними для розробників програмного забезпечення, які займаються обробкою великих обсягів даних на різних платформах, а також для науковців, які шукають ефективні інструменти для вирішення складних обчислювальних задач.

Ключові слова: кросплатформові мови програмування, ефективність, обробка даних, паралельні обчислення, масштабованість, оптимізація.

ABSTRACT

Titov I.S. Determining the Effectiveness of Cross-Platform Programming Languages in Data Processing Tasks. Specialty 122 "Computer Science," Educational Program "Data Science." Donetsk National University named after Vasyl Stus, Vinnytsia, 2024.

The master's thesis is dedicated to determining the effectiveness of cross-platform programming languages in the context of data processing tasks. The relevance of the research is due to the need for optimizing the processing of large data volumes in the context of the rapid development of information technologies and the demand for high-performance software that must operate across different operating systems. The thesis compares popular cross-platform programming languages, such as Python, JavaScript, Java, and others, with the aim of evaluating their ability to efficiently process large data sets, perform parallel computations, and integrate with various platforms.

The research methodology includes experimental measurement of task execution time, memory consumption, scalability of solutions, and ease of integration with various data processing tools. The work also explores different approaches to parallelizing data processing, as well as the limitations and advantages of each programming language in the context of real-world usage scenarios.

The results of the research may be useful for software developers working with large data sets across different platforms, as well as for researchers seeking effective tools for solving complex computational problems.

Keywords: cross-platform programming languages, efficiency, data processing, parallel computing, scalability, optimization.

ЗМІСТ

ВСТУП 5
РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ 8
1.1 Аналіз предметної області 8
1.2 Постановка задачі 10
РОЗДІЛ 2. АНАЛІЗ МОВ ПРОГРАМУВАННЯ ТА АЛГОРИТМІВ 12
2.1 Мови програмування 12
2.2 Огляд сильних та слабких сторін обраних мов програмування 13
2.3 Алгоритми для роботи з даними та структури 21
2.4 Інструментарій 42
2.5 Тести 50
РОЗДІЛ 3. РОЗРОБКА ТА АНАЛІЗ РЕЗУЛЬТАТІВ 54
3.1 Реалізація алгоритмів на різних мовах програмування 54
3.2 Аналіз отриманих даних 65
3.3 Висновок щодо переваг Java в обробці даних 83
ВИСНОВКИ 85
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 86

ВСТУП

Актуальність теми:

У сучасному світі обсяги даних зростають експоненціально, що створює нові виклики для їх обробки, зберігання та аналізу. Водночас розвиток технологій веде до потреби в універсальних інструментах, які можуть працювати на різних платформах і в різних середовищах. У зв'язку з цим, вибір кросплатформових мов програмування набуває особливого значення, оскільки вони дозволяють розробникам створювати продуктивні та адаптивні рішення для різних операційних систем та апаратних архітектур.

Кросплатформові мови програмування, такі як Python, Java, JavaScript, Kotlin та інші, пропонують не лише можливість виконання коду на різних платформах, але й часто забезпечують потужні інструменти для обробки великих обсягів даних. Їх використання дозволяє оптимізувати процеси розробки, знижуючи витрати часу на адаптацію програмного забезпечення до різних платформ. Це особливо важливо для компаній та організацій, що працюють з великими даними та вимагають гнучких, масштабованих рішень.

Більше того, зростання попиту на кросплатформові рішення у сфері обробки даних, зокрема у хмарних обчисленнях, інтернеті речей (IoT) та штучному інтелекті, підкреслює необхідність у глибокому аналізі ефективності різних мов програмування. Вибір оптимальної мови для обробки даних може вплинути не лише на продуктивність системи, але й на витрати ресурсів, енергоспоживання та час розробки, що робить цю тему надзвичайно актуальною для сучасних технологічних тенденцій.

Отже, дослідження ефективності кросплатформових мов програмування у задачах обробки даних має велике практичне значення, оскільки дозволяє знайти оптимальні рішення для роботи з великими даними в умовах різних операційних середовищ та інфраструктур.

Мета дослідження:

Мета дослідження полягає у визначенні та порівнянні ефективності різних кросплатформових мов програмування в задачах обробки даних, щоб на основі отриманих результатів зробити висновки про оптимальний вибір мови для виконання таких задач.

Для досягнення цієї мети необхідно вирішити наступні завдання:

Проаналізувати сучасні тенденції розвитку кросплатформових мов програмування та їх застосування в обробці даних.

Вибрати критерії оцінки ефективності мов програмування, які є важливими для обробки даних (продуктивність, швидкість виконання, використання пам'яті, масштабованість тощо).

Розробити та реалізувати тестові задачі обробки даних на кількох популярних кросплатформових мовах програмування.

Провести експериментальні дослідження для порівняння ефективності виконання тестових задач на різних мовах програмування.

Здійснити аналіз отриманих результатів і зробити висновки щодо продуктивності та переваг кожної мови програмування у конкретних типах задач обробки даних.

Розробити рекомендації щодо вибору мови програмування для різних типів задач обробки даних з урахуванням отриманих результатів.

Завдання дослідження:

Для досягнення поставленої мети було визначено наступні завдання:

1. Визначити критерії оцінки ефективності мов програмування.
2. Вибрати мови програмування для аналізу та описати тестові задачі.
3. Провести експериментальне тестування вибраних мов програмування для вирішення задач обробки даних.
4. Здійснити порівняльний аналіз ефективності у отриманих результатах.

Предмет дослідження:

Предметом дослідження є ефективність кросплатформових мов програмування в контексті використання різних алгоритмів для вирішення задач обробки даних, яка оцінюється за такими параметрами, як швидкість виконання, використання ресурсів пристрою, зручність розробки та підтримки коду, а також наявність допоміжних бібліотек та фреймворків.

В результаті даного дослідження очікується отримати практичні висновки та рекомендації, які допоможуть вибирати оптимальні інструменти для своїх проектів, а також сприятимуть підвищенню загальної ефективності процесів обробки даних.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної області

У сучасній технологічній екосистемі обробка даних є однією з ключових задач для багатьох галузей, таких як фінанси, медицина, маркетинг, наука, і навіть соціальні мережі. Великі обсяги даних, відомі як "Big Data", постійно генеруються та вимагають ефективних засобів для їх зберігання, аналізу та використання. Це зумовлює зростання попиту на мови програмування, здатні швидко і ефективно вирішувати задачі обробки даних у різних умовах [1].

Кросплатформові мови програмування

Кросплатформові мови програмування дозволяють розробникам створювати код, який може виконуватися на різних операційних системах і платформах без необхідності його значної адаптації. Це надзвичайно важливо в сучасному світі, де програмне забезпечення повинно працювати як на мобільних пристроях, так і на серверних інфраструктурах.

До найбільш поширених кросплатформових мов програмування відносяться:

- Python — одна з найпопулярніших мов для обробки даних завдяки великій кількості бібліотек (NumPy, Pandas, Scikit-learn) і відносно простій синтаксисній структурі.
- Java — мова з високим рівнем продуктивності, яка забезпечує стабільну роботу на різних платформах. Завдяки віртуальній машині Java (JVM) вона широко використовується в задачах Big Data та розподілених системах.
- JavaScript — має велике поширення у веб-технологіях, але також активно використовується для обробки даних завдяки платформі Node.js та можливості роботи на серверній стороні.
- Kotlin — сучасна кросплатформова мова, яка використовується переважно для розробки Android-додатків, але також отримує поширення в обробці даних.

- Rust — мова з високою продуктивністю та контролем пам'яті, яка завойовує популярність серед системних і наукових розробників, завдяки своїй швидкості та безпеці в роботі з даними.
- C# — мова програмування, яка спочатку була орієнтована на роботу в середовищі .NET. Проте останніми роками, завдяки розвитку .NET Core (зараз .NET), C# став однією з потужних кросплатформових мов програмування, яка може працювати на різних операційних системах, включаючи Windows, Linux і macOS. [1]

Завдання обробки даних

Обробка даних може включати різноманітні завдання:

- ETL (Extract, Transform, Load) — процес вилучення, перетворення і завантаження даних, який є критичним для більшості аналітичних та бізнес-додатків.

- Обробка великих даних (Big Data) — аналіз величезних обсягів даних з використанням таких платформ, як Hadoop або Spark, де швидкість і ефективність обробки відіграють ключову роль.

- Аналіз потокових даних (Streaming) — реальний час обробки даних, таких як фінансові транзакції або події в соціальних мережах.

- Машинне навчання — тренування моделей на великих обсягах даних, що потребує значних обчислювальних ресурсів і оптимальної швидкості виконання алгоритмів. [1]

Критерії ефективності мов програмування

Для оцінки ефективності кросплатформових мов програмування в задачах обробки даних можна використовувати такі ключові критерії:

- Швидкість виконання — час, необхідний для виконання завдання з обробки даних.

- Використання пам'яті — обсяг оперативної пам'яті, яка використовується програмою під час роботи.
- Масштабованість — здатність системи розширюватися без втрати продуктивності під час обробки великих обсягів даних або при збільшенні числа користувачів.
- Паралельність і багатопоточність — можливість обробки великих обсягів даних паралельно на кількох ядрах або серверах.
- Простота розробки і підтримки — важливий аспект, який впливає на швидкість розробки програм і підтримку коду в довгостроковій перспективі. [1]

Актуальність вибору ефективної мови програмування

У контексті сучасних вимог до обробки даних вибір кросплатформової мови програмування може значно вплинути на продуктивність системи, споживання ресурсів та загальні витрати на розробку. Для компаній, які працюють з великими даними або займаються машинним навчанням, це може стати конкурентною перевагою.

Аналіз предметної області демонструє необхідність глибокого дослідження ефективності мов програмування для різних типів задач обробки даних, що дозволить знайти оптимальні рішення для конкретних умов і вимог.[2]

1.2 Постановка задачі

Основна **задача дослідження** полягає у визначенні та порівнянні ефективності кількох кросплатформових мов програмування (таких як Python, Java, C#) і в противіс обробка цих даних за допомогою SQL, в типових задачах обробки даних, оскільки ми будемо працювати з базами даних:

1. Обробка великих обсягів структурованих і неструктурованих даних.
2. Реалізація ETL (Extract, Transform, Load) процесів [2].

Конкретні аспекти задачі включають:

- Оцінку продуктивності (швидкість виконання програм) на різних етапах обробки даних.
- Аналіз використання ресурсів (пам’яті, процесорного часу) програмами на кожній мові.
- Порівняння простоти реалізації та підтримки кодової бази для кожної мови в контексті різних типів задач обробки даних. [2]

Ці завдання мають на меті дати відповідь на ключове питання: яка з кросплатформових мов програмування є найбільш ефективною для вирішення задач обробки даних і чи можуть конкурувати звичайні алгоритми обробки даних з інтегрованими в SQL[33]?

РОЗДІЛ 2. АНАЛІЗ МОВ ПРОГРАМУВАННЯ ТА АЛГОРИТМІВ

Попередній огляд мов програмування та алгоритмів є важливим етапом, оскільки дасть можливість приблизно передбачити результати, які ми можемо отримати у процесі дослідження.

2.1 Мови програмування

Java — об'єктно-орієнтована мова програмування, розроблена компанією Sun Microsystems (тепер належить Oracle). Вона відома своєю кросплатформеністю завдяки використанню Java Virtual Machine (JVM), яка дозволяє виконувати байт-код на будь-якій платформі, що підтримує JVM. Java широко використовується в корпоративних додатках, веб-розробці, а також для створення великих масштабованих систем. Мова має багатий набір стандартних бібліотек і сильні сторони у продуктивності та безпеці [3].

Python — високорівнева мова програмування з динамічною типізацією, відома своєю простотою і читабельністю коду. Вона підтримує кілька парадигм програмування, включаючи об'єктно-орієнтоване, процедурне та функціональне програмування. Python має багату екосистему бібліотек, таких як NumPy, pandas, і scikit-learn, що робить її ідеальною для обробки даних, наукових розрахунків і машинного навчання. Однак, через інтерпретовану природу, Python може бути менш продуктивною у порівнянні з компільованими мовами. [3]

C# (C-sharp) — мова програмування, розроблена компанією Microsoft, яка працює на платформі .NET. Це об'єктно-орієнтована мова, яка також підтримує функціональне та процедурне програмування. C# відома своєю продуктивністю та можливістю створення широкого спектра додатків, включаючи веб-додатки, десктопні додатки та ігри.

2.2 Огляд сильних та слабких сторін обраних мов програмування

Використання Java, Python та C# для обробки великих обсягів даних має свої переваги та недоліки. Ці мови програмування широко використовуються завдяки своїм можливостям, але вони по-різному підходять до задач обробки даних. Розглянемо кожну з них детальніше за різними критеріями ефективності:

Java

Java є однією з найпоширеніших мов програмування для обробки даних і має значні переваги в цій сфері завдяки своїй продуктивності, універсальності та підтримці великих фреймворків для обробки даних. Її ефективність можна оцінити за кількома ключовими аспектами: [3]

1. Платформонезалежність і масштабованість

Java є справжньою кросплатформовою мовою завдяки використанню **Java Virtual Machine (JVM)**, яка дозволяє виконувати код на будь-якій операційній системі, де доступна JVM. Це особливо важливо для великих систем обробки даних, де платформа може варіюватися від локальних серверів до хмарних рішень. [3]

Java також відома своєю масштабованістю. Її можна використовувати як для обробки невеликих наборів даних на окремих машинах, так і для великих розподілених обчислень у хмарних середовищах. Саме тому Java є основою багатьох фреймворків для великих даних, таких як **Apache Hadoop** і **Apache Spark**, які можуть працювати в розподілених системах і забезпечувати обробку терабайтів або петабайтів даних [4].

2. Висока продуктивність

Java є компільованою мовою, що дозволяє досягати високої продуктивності під час обробки даних. Хоча Java трохи поступається C++ за швидкістю, вона забезпечує кращу продуктивність, ніж інтерпретовані мови, такі

як Python або Ruby, особливо при роботі з великими наборами даних і багатопоточними обчисленнями. [4]

Підтримка **JVM** також дозволяє використовувати Java разом з іншими мовами, такими як **Scala** або **Kotlin**, які можуть запропонувати додаткову гнучкість для вирішення спеціалізованих завдань.

3. Інструменти та фреймворки для обробки даних

Java має багату екосистему бібліотек і фреймворків для роботи з великими даними. Основні з них:

- Apache Hadoop — одна з найпоширеніших платформ для обробки великих даних, що підтримує розподілене зберігання та обробку даних через модель MapReduce. Hadoop використовує Java як основну мову програмування для своєї розробки.
- Apache Spark — фреймворк для обробки великих даних, який працює в пам'яті і забезпечує високу швидкість обробки даних. Хоча Spark підтримує кілька мов (Scala, Python, R), Java є однією з основних мов, що дозволяє отримати максимальну продуктивність.
- Apache Kafka — популярна платформа для обробки потокових даних у реальному часі. Kafka широко використовується в системах, де необхідно обробляти величезні обсяги даних у режимі реального часу [5].

4. Підтримка багатопоточності та паралельної обробки

Java має вбудовану підтримку багатопоточності, що дозволяє ефективно використовувати ресурси багатоядерних процесорів для обробки даних паралельно. Це критично важливо для обробки великих обсягів даних, коли завдання можуть бути розподілені між кількома потоками або серверами [6].

5. Надійність і безпека

Java забезпечує високий рівень надійності і безпеки завдяки своїй віртуальній машині, яка контролює доступ до ресурсів та забезпечує захист від багатьох типів вразливостей. Це важливо для обробки конфіденційних даних або при роботі в хмарних середовищах [7].

6. Недоліки Java в обробці даних

- Зайве споживання пам'яті: Java може використовувати більше оперативної пам'яті порівняно з іншими мовами через JVM і збирач сміття (Garbage Collector), що може бути проблематичним у ресурсно обмежених системах.
- Кодова надмірність: Порівняно з іншими мовами, такими як Python чи Scala, Java вимагає більше коду для реалізації тієї ж функціональності, що може уповільнити розробку і ускладнити підтримку [8].

Python

Python є однією з найпопулярніших мов програмування для обробки даних завдяки своїй простоті, універсальності та розвиненій екосистемі бібліотек. Його ефективність у роботі з даними можна пояснити низкою ключових факторів [31]:

1. Простота та зрозумілість синтаксису

Python має простий і читабельний синтаксис, що дозволяє розробникам швидко писати код і легко підтримувати його. Це особливо важливо для обробки даних, де часто потрібно експериментувати з різними підходами та алгоритмами [32]. Завдяки зрозумілому коду Python широко використовується як для початкового аналізу даних, так і для складних моделей машинного навчання [9].

2. Багата екосистема бібліотек для обробки даних

Однією з головних причин популярності Python у сфері обробки даних є велика кількість спеціалізованих бібліотек і фреймворків, які значно спрощують роботу з різними типами даних:

- **NumPy** — бібліотека для роботи з багатовимірними масивами і матрицями, яка забезпечує швидкі операції з числовими даними.
- **Pandas** — потужна бібліотека для роботи з табличними даними (наприклад, з електронними таблицями або базами даних), що надає зручний інструментарій для фільтрації, сортування та агрегації даних.
- **Matplotlib та Seaborn** — бібліотеки для візуалізації даних, які дозволяють створювати графіки, діаграми і візуалізувати результати аналізу.
- **Scikit-learn** — один з найпопулярніших фреймворків для машинного навчання, який надає велику кількість алгоритмів для класифікації, регресії, кластеризації та інших типів аналізу даних.
- **TensorFlow і PyTorch** — фреймворки для створення та тренування моделей глибокого навчання, які активно використовуються у нейромережах та штучному інтелекті.

Ці бібліотеки роблять Python універсальним інструментом для вирішення широкого спектра задач обробки даних, від початкового очищення і підготовки даних до реалізації складних моделей машинного навчання [10].

3. Широке застосування у науковому середовищі

Python активно використовується в наукових дослідженнях завдяки простоті, підтримці бібліотек для наукових обчислень (як NumPy та SciPy) та можливості інтеграції з інструментами для візуалізації даних. Наукова спільнота вибирає Python за його гнучкість та можливість швидкої побудови прототипів моделей.

4. Підтримка роботи з великими даними

Хоча Python не є найшвидшою мовою програмування, він добре інтегрується з платформами для обробки великих даних. Завдяки бібліотекам і фреймворкам, таким як Dask і PySpark (інтерфейс для роботи з Apache Spark), Python може працювати з великими обсягами даних у розподілених системах. Це дозволяє обробляти дані, які не вміщуються в пам'ять на одному комп'ютері, і виконувати паралельні обчислення [11].

5. Інструменти для роботи з потоковими даними

Python підтримує роботу з потоковими даними в режимі реального часу за допомогою таких інструментів, як Apache Kafka або Flask, що дозволяє інтегрувати Python в системи потокової обробки даних. Це стає важливим для завдань, що потребують обробки в реальному часі, таких як аналіз транзакцій або подій у соціальних мережах.

6. Велика кількість інструментів для візуалізації

Python надає багатий набір інструментів для візуалізації даних, що є критично важливим для ефективного представлення результатів аналізу. Крім вже згаданих Matplotlib та Seaborn, існують більш спеціалізовані бібліотеки, як Plotly та Bokeh, що дозволяють створювати інтерактивні графіки для презентацій та аналітики [12].

7. Недоліки Python у роботі з даними

Швидкодія: Python є інтерпретованою мовою, що робить його повільнішим порівняно з компільованими мовами, такими як C++ або Java. Це може стати проблемою для обробки дуже великих обсягів даних або для завдань, де продуктивність є критично важливою.

Обмеження з використанням пам'яті: Python використовує багато пам'яті, і це може бути обмеженням при роботі з великими наборами даних, особливо у випадках, коли дані не можуть бути розбиті на частини [13].

C#

C# є потужною мовою програмування, яка останніми роками набула популярності в обробці даних, особливо після появи кросплатформової платформи .NET Core (нині .NET). Ефективність C# у задачах обробки даних обумовлена кількома ключовими аспектами:

1. Висока продуктивність і компіляція в машинний код

C# є компільованою мовою, що забезпечує вищу продуктивність у порівнянні з інтерпретованими мовами, такими як Python. Програми на C# компілюються у проміжний код (IL), який виконується в середовищі .NET Runtime (CLR), що оптимізує виконання коду. Це дозволяє C# бути швидким у обробці даних, особливо в задачах, що потребують інтенсивних обчислень, таких як алгоритми машинного навчання або аналіз великих масивів даних [34].

2. Підтримка багатопоточності та асинхронної обробки

C# надає потужні засоби для багатопотокової та асинхронної обробки, що дозволяє ефективно розподіляти задачі обробки даних між потоками, особливо коли потрібно обробляти великі обсяги даних або виконувати паралельні операції. Використання Task Parallel Library (TPL) та ключового слова async/await робить асинхронне програмування інтуїтивно зрозумілим, що важливо при обробці даних у реальному часі або в розподілених системах [14].

3. Інструменти та фреймворки для роботи з даними

C# має доступ до кількох важливих бібліотек і фреймворків, які дозволяють ефективно працювати з даними:

- **ML.NET** — це бібліотека для машинного навчання, що дозволяє працювати з даними та створювати моделі прямо на C#. Це забезпечує інтеграцію з існуючими додатками, написаними на C#, і дозволяє уникати необхідності використовувати окремі інструменти, такі як Python або R.
- **Entity Framework (EF)** — ORM-фреймворк для взаємодії з базами даних, який спрощує роботу з реляційними базами даних. Він дозволяє писати ефективний код для запитів до бази даних і управління даними, що робить його зручним для розробки рішень з обробки великих обсягів структурованих даних.
- **.NET для Apache Spark** — це бібліотека, яка дозволяє використовувати C# для роботи з Apache Spark, платформою для обробки великих даних у розподілених системах. Ця інтеграція робить C# конкурентоспроможним у сфері обробки великих даних [15].

4. Підтримка роботи з хмарними сервісами

C# має чудову інтеграцію з хмарними платформами, такими як Microsoft Azure. Це дозволяє легко масштабувати рішення з обробки даних, використовуючи сервіси, які підтримують масове зберігання та аналіз даних у хмарі, наприклад, Azure Data Lake, Azure Synapse або Azure Machine Learning. Це робить C# ефективним вибором для побудови розподілених і масштабованих рішень.

5. Масштабованість і розподілена обробка

Завдяки інтеграції з Apache Spark та підтримці масштабованих хмарних сервісів, C# може використовуватися для обробки великих обсягів даних у розподілених системах. Крім того, підтримка багатопотоковості дозволяє створювати рішення, які ефективно використовують ресурси для обробки великих масивів даних паралельно [16].

6. Інструменти для потокової обробки даних

C# також може бути використаний для обробки потокових даних у реальному часі. За допомогою таких технологій, як Azure Stream Analytics або інтеграції з Apache Kafka, можна створювати системи, які аналізують дані на льоту, що важливо для моніторингу подій у реальному часі, таких як фінансові транзакції або обробка даних із сенсорів [17].

7. Сильні сторони C# у роботі з даними

Стабільність і надійність: Завдяки підтримці Microsoft C# постійно оновлюється та розширюється, що робить його надійним вибором для критично важливих бізнес-додатків.

Інтеграція з корпоративними рішеннями: C# добре інтегрується з корпоративними системами та платформами, такими як SQL Server, що робить його зручним для побудови комплексних рішень із обробки даних у великих організаціях.

Типізація та безпека: Статична типізація в C# дозволяє виявляти помилки на етапі компіляції, що знижує ризик появи багів у продуктивних системах, особливо при роботі з великими масивами даних [18].

8. Недоліки C# у роботі з даними

Споживання пам’яті: Через використання збирача сміття (Garbage Collector) C# може мати проблеми з ефективним використанням пам'яті при обробці великих обсягів даних у порівнянні з мовами на низькому рівні, такими як C++.

Меньша кількість бібліотек для аналізу даних: Хоча C# має ML.NET і інші бібліотеки для роботи з даними, його екосистема значно поступається Python за кількістю спеціалізованих бібліотек для обробки даних і машинного навчання.

2.3 Алгоритми для роботи з даними та структури

Перш за все розглянемо що таке алгоритм.

Алгоритм — це чітка, послідовна інструкція або набір правил для виконання певного завдання або розв'язання проблеми. Він визначає, як повинні бути виконані різні кроки для досягнення конкретного результату, і може бути представленим у різних формах, таких як:

- Псевдокод: Набір інструкцій, написаних на зрозумілій людині мові, без синтаксису конкретної мови програмування. Це дозволяє зосередитися на логіці алгоритму, а не на деталях реалізації [35].
- Діаграми: Використання графічних представлень, таких як блок-схеми, для візуалізації процесу виконання алгоритму. Це може допомогти зрозуміти зв'язки між різними частинами алгоритму.
- Код: Алгоритм може бути реалізований на конкретній мові програмування, такій як Python, Java, C++ тощо.

Основні властивості алгоритму:

- Визначеність: Кожен крок алгоритму повинен бути чітко визначеним і однозначним.
- Кінцевість: Алгоритм повинен завершуватися через обмежену кількість кроків, даючи результат.
- Вхідні дані: Алгоритм може мати нульові або більше вхідних даних, які він використовує для виконання обчислень.
- Вихідні дані: Алгоритм повинен повертати результати (вихідні дані) на основі виконаних дій.
- Універсальність: Алгоритм повинен бути здатним виконувати свою задачу для широкого класу вхідних даних.

Big O — це нотація, яка використовується для опису асимптотичної складності алгоритмів, зокрема для оцінки їхньої швидкості виконання або обсягу пам'яті, що використовується в залежності від розміру вхідних даних. Ця

нотація дозволяє аналізувати, як змінюється продуктивність алгоритму при збільшенні обсягу даних [19].

Основні аспекти Big O нотації:

- **Асимптотичний аналіз**: Big O фокусується на поведінці алгоритму, коли розмір вхідних даних прямує до безкінечності. Це означає, що ми розглядаємо найбільш значущі частини часу виконання, ігноруючи константи і менш значущі терміни [36].
- **Оцінка найгіршого випадку**: Big O зазвичай використовується для опису найгіршого сценарію виконання алгоритму, що означає максимальний час або ресурси, які можуть бути витрачені на виконання.
- **Різні класи складності**: Ось кілька загальних класів складності, описуваних за допомогою Big O:
    - **O(1)**: Константна складність. Час виконання не залежить від розміру вхідних даних.
    - **O(log n)**: Логарифмічна складність. Час виконання зростає логарифмічно при збільшенні вхідних даних (наприклад, бінарний пошук).
    - **O(n)**: Лінійна складність. Час виконання пропорційний розміру вхідних даних.
    - **O(n log n)**: Лінійно-логарифмічна складність. Зазвичай спостерігається в ефективних алгоритмах сортування (наприклад, швидке або злиткове сортування) [37].
    - **O(n²)**: Квадратична складність. Час виконання пропорційний квадрату розміру вхідних даних (наприклад, алгоритм сортування бульбашкою).
    - **O(2^n)**: Експоненційна складність. Час виконання різко зростає при збільшенні вхідних даних (наприклад, рішення задачі про рюкзак).

- **O(n!)**: Факторіальна складність. Використовується в алгоритмах, що вирішують задачі комбінаційного характеру (наприклад, перебір усіх перестановок).

Розглянемо візуальний приклад кількості операцій в залежності від складності алгоритму з однаковою кількістю вхідних даних (Рисунок 2.3.1)

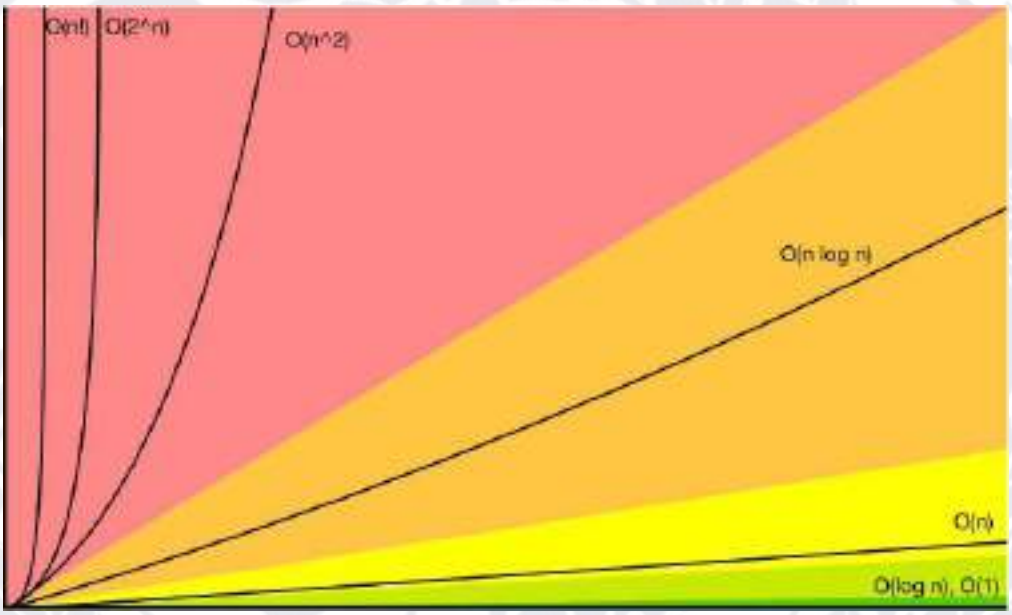


Рисунок 2.3.1- Графік залежності кількості операцій з однією кількістю даних

Структури даних — це спеціалізовані формати організації, зберігання та обробки даних, що дозволяють ефективно виконувати різноманітні операції над цими даними. Вибір правильної структури даних має критичне значення для продуктивності програми, оскільки різні структури даних оптимізують виконання певних операцій (зберігання, доступ, модифікація тощо).[19]

Основні типи структур даних

- **Масиви (Arrays):**
    - Опис: Послідовний набір елементів одного типу, доступних за індексом.
    - Переваги:
        - Швидкий доступ до елементів за індексом (O(1)).
        - Простота в реалізації.
    - Недоліки:
        - Фіксована довжина (необхідно заздалегідь знати розмір).
        - Витрати на переміщення елементів при вставці або видаленні (O(n)).
- **Списки (Lists):**
    - Опис: Динамічні структури, що складаються з елементів, які можуть зростати або зменшуватися в розмірі.
    - Типи:
        - Зв'язані списки (Linked Lists): Кожен елемент (вузол) містить дані та посилання на наступний елемент.
        - Двосторонні зв'язані списки (Doubly Linked Lists): Вузли мають посилання на попередній та наступний елементи.
    - Переваги:
        - Динамічна пам'ять (немає фіксованого розміру).
        - Легко вставляти та видаляти елементи (O(1)).
    - Недоліки:
        - Повільніший доступ до елементів (O(n)).
        - Потреба в додатковій пам'яті для зберігання вказівників.
- **Стек (Stack):**
    - Опис: Структура даних, що реалізує принцип "останній прийшов — перший вийшов" (LIFO).
    - Операції:
        - Push: Додати елемент на верх стека.
        - Pop: Видалити верхній елемент.

- Переваги:
  - Швидкий доступ до верхнього елемента (O(1)).
  - Простота реалізації.
- Недоліки:
  - Обмежений доступ (можна отримати доступ тільки до верхнього елемента).

- **Черга (Queue):**
  - Опис: Структура даних, що реалізує принцип "перший прийшов — перший вийшов" (FIFO).
  - Операції:
    - Enqueue: Додати елемент в кінець черги.
    - Dequeue: Видалити перший елемент.
  - Переваги:
    - Швидкий доступ до переднього елемента (O(1)).
    - Застосування в задачах планування (наприклад, в обробці запитів).
  - Недоліки:
    - Обмежений доступ (можна отримати доступ тільки до переднього елемента).
- **Хеш-таблиці (Hash Tables):**
  - Опис: Структура даних, що забезпечує асоціативний доступ до даних за ключем.
  - Операції:
    - Додавання, видалення та пошук елемента за ключем.
  - Переваги:
    - Швидкий доступ (середнє O(1) при хорошій хеш-функції).
  - Недоліки:
    - Витрати пам'яті.
    - Вирішення колізій (деколи два ключі можуть віддавати один і той же хеш).

- **Деревоподібні структури (Trees):**
    - Опис: Ієрархічна структура даних, де кожен елемент (вузол) може мати нуль або більше нащадків.
    - Типи:
        - Бінарні дерева: Кожен вузол має не більше ніж двох нащадків.
        - Бінарні дерева пошуку (BST): Ліва піддерево містить менші значення, а права — більші.
        - Сбалансовані дерева: Наприклад, AVL-дерев’я та червоні-чорні дерева, що підтримують збалансованість для зменшення висоти дерева.
    - Переваги:
        - Швидкий доступ до елементів (O(log n) для збалансованих дерев).
        - Ієрархічне зберігання даних.
    - Недоліки:
        - Складність реалізації.
        - Потреба у підтримці балансування (для BST).
- **Графи (Graphs):**
    - Опис: Структура даних, що складається з вузлів (вершин) і ребер (з'єднань між вершинами).
    - Представлення:
        - Списки суміжності: Вершини зберігаються в масивах або списках.
        - Матриці суміжності: Двовимірний масив, що показує, чи існує ребро між парами вершин.
    - Переваги:
        - Можливість моделювання складних взаємозв'язків (мережі, соціальні зв'язки).
    - Недоліки:
        - Складність реалізації.

- Витрати пам'яті.

Вибір структури даних

Вибір відповідної структури даних залежить від специфіки задачі, яку потрібно вирішити. Наприклад:

- Для швидкого доступу до елементів — масиви або хеш-таблиці.
- Для частих вставок та видалень — списки або стеки.
- Для організації даних у ієрархічному вигляді — дерева.
- Для моделювання складних зв'язків — графи.

Приклади алгоритмів

Алгоритми сортування

Розглянемо основні алгоритми для сортування даних:

- Сортування бульбашкою;
- Сортування вибором;
- Циклічне сортування;
- Швидке сортування.

Розглянемо, чим вони відрізняються один від одного, для чого їх використовують і де застосовують. Також ми проаналізуємо їх реалізацію та візуалізацію, і розберемося, як реалізувати їх покроково.

Перед тим як перейти до вивчення алгоритмів сортування, потрібно уточнити, що алгоритми сортування поділяються на стабільні та нестабільні [38].

Алгоритм вважається стабільним лише в тому випадку, якщо він не змінює порядок елементів з однаковими значеннями відносно один одного. Відповідно, нестабільний алгоритм — це, навпаки, той, який може змінити цей порядок [20].

Сортування бульбашкою — це найпримітивніший і базовий алгоритм сортування. Він є основою для деяких інших алгоритмів. Цей алгоритм є стабільним. Сортування бульбашкою перебирає весь масив елементів, порівнюючи два сусідні елементи між собою та змінюючи їх місцями відповідно до умов [39]. Елементи з більшими значеннями опускаються вниз масиву, а

елементи з найменшими значеннями піднімаються вгору, подібно до бульбашок газу.

Складність алгоритму: $O(n^2)$, де n — кількість елементів масиву. Оскільки ми запускаємо вкладений цикл, складність алгоритму дорівнює $O(n^2)$.

Кроки реалізації:

- Запускаємо цикл і по масиву.
- Запускаємо внутрішній цикл j, який іде від 0 до arr.length - i. Це прискорює алгоритм, оскільки він не проходить по вже відсортованим елементам.
- У внутрішньому циклі перевіряємо сусідні елементи та змінюємо їх місцями, якщо сусід зліва більший за сусіда справа.

Простий приклад реалізації бульбашкового алгоритму на мові Python(Рисунок 2.3.2)

```python
def bubble_sort(arr):
    n = len(arr)
    # Проходимо по всім елементам масиву
    for i in range(n):
        # Останні і елементів вже відсортовані, тому їх можна пропустити
        for j in range(0, n - i - 1):
            # Якщо елемент більше сусіднього, то обмінюємо їх
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

# Приклад використання
numbers = [64, 34, 25, 12, 22, 11, 90]
sorted_numbers = bubble_sort(numbers)
print("Відсортований масив:", sorted_numbers)
```

Рисунок 2.3.2- Приклад бульбашкового алгоритму

Алгоритм сортування бульбашкою слід використовувати в таких випадках:

1. **Простота реалізації**: Алгоритм бульбашкового сортування дуже простий у розумінні і реалізації, що робить його хорошим вибором для навчальних цілей або для швидкого написання коду, коли не потрібна оптимізація продуктивності.

2. **Малий обсяг даних**: Сортування бульбашкою має квадратичну складність $O(n^2)$, тому воно підходить для невеликих масивів. Для наборів даних з менше ніж 20-30 елементів його продуктивності може бути достатньо.
3. **Сталість**: Алгоритм є стабільним, що означає, що порядок однакових елементів залишається незмінним. Це може бути важливим у випадках, коли потрібно зберігати первинний порядок однакових значень.
4. **Для детального навчання алгоритмів**: Якщо ви вивчаєте основи алгоритмів і структури даних, бульбашкове сортування може бути корисним прикладом для ілюстрації принципів сортування та обміну елементів.
5. **Коли дані майже відсортовані**: Алгоритм може працювати швидше, якщо масив вже майже відсортований, оскільки в таких випадках кількість обміну елементів буде значно зменшеною. Додавання оптимізації, що перевіряє, чи відбулися зміни під час ітерацій, може підвищити його ефективність [21].

Сортування вибором- це алгоритм сортування який на кожній ітерації проходить по несортованій частині масиву, знаходить мінімальний елемент і переносить його на початок масиву.

Він може бути як стабільним, так і нестабільним алгоритмом, в залежності від реалізації. Складність алгоритму: $O(n^2)$.

Кроки реалізації:

1. Запускаємо цикл і по масиву.
2. Усередині циклу створюємо змінну, рівну ітерації циклу: min = i.
3. Запускаємо внутрішній цикл j по залишковим елементам масиву до його кінця.
4. Якщо елемент arr[min] > arr[j], то присвоюємо min значення j.
5. По завершенні внутрішнього циклу j змінна min зберігає індекс найменшого несортованого елемента масиву.

6. Міняємо місцями елементи з індексами i і min.

Простий приклад реалізації алгоритму сортування вибором на мові Python(Рисунок 2.3.3)

```python
def selection_sort(arr):
    n = len(arr)
    # Проходимо по всіх елементах масиву
    for i in range(n):
        # Зберігаємо індекс мінімального елемента
        min_index = i
        # Проходимо по залишкових елементах масиву
        for j in range(i + 1, n):
            # Якщо знайдено новий мінімальний елемент, оновлюємо min_index
            if arr[j] < arr[min_index]:
                min_index = j
        # Міняємо місцями мінімальний елемент з елементом на позиції i
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

# Приклад використання
numbers = [64, 25, 12, 22, 11]
sorted_numbers = selection_sort(numbers)
print("Відсортований масив:", sorted_numbers)
```

Рисунок 2.3.3- Приклад алгоритму сортування вибором

Алгоритм сортування вибором слід використовувати в наступних випадках:

1. **Простота реалізації**: Алгоритм простий у розумінні та реалізації, що робить його хорошим вибором для навчальних цілей або для швидкого написання коду в проектах, де не потрібно оптимізувати продуктивність.
2. **Малий обсяг даних**: Сортування вибором має квадратичну складність $O(n^2)$, тому воно працює ефективніше для невеликих масивів. Для масивів з менше ніж 100-200 елементів його продуктивності може бути достатньо.
3. **Нестабільні вимоги**: Якщо не важливо, чи зберігається порядок однакових елементів, сортування вибором може бути хорошим вибором, оскільки цей алгоритм є нестабільним.
4. **Обмеження пам'яті**: Алгоритм сортування вибором не вимагає додаткової пам'яті для створення нових масивів, оскільки він виконує перестановки на місці. Це робить його корисним, коли пам'ять обмежена.

5. **Коли потрібно зменшити кількість обміну елементів**: Сортування вибором робить менше обміну елементів, ніж, наприклад, бульбашкове сортування. Якщо обмін є дорогою операцією (наприклад, в контексті роботи з великими об'єктами), це може бути перевагою.[21]

Циклічне сортування

Основною ідеєю алгоритму циклічного сортування є розкладання масиву на цикли. Потім всередині цих циклів відбуваються перестановки елементів доти, поки всі цикли не завершаться і масив не буде відсортований.

Алгоритм циклічного сортування є нестабільним.

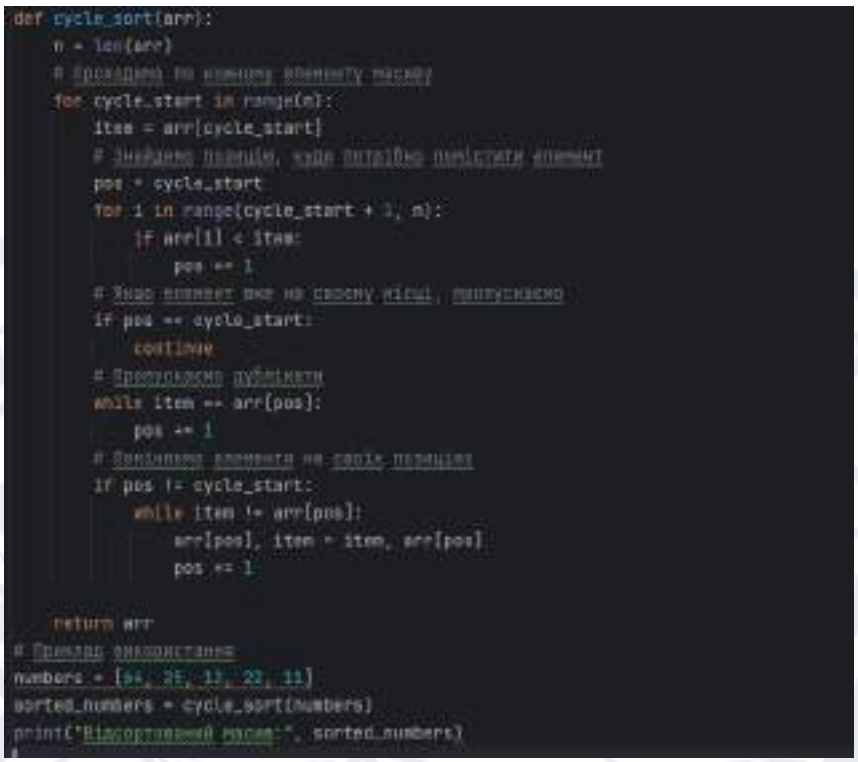
Хоча алгоритм циклічного сортування не є простим для розуміння, він цінується за те, що зміни елементів масиву відбуваються тільки тоді, коли елемент ставиться на своє місце. Це особливо важливо, якщо зміна елементів масиву є затратною.

Складність алгоритму: $O(n^2)$.

Кроки реалізації:

- Запускаємо цикл i, який має пройти по всьому масиву.
- Створюємо тимчасову змінну position, яка буде дорівнювати i.
- Запускаємо внутрішній цикл j, який перебирає всіх сусідів справа.
- Коли всередині циклу j знаходимо елемент, який менший за arr[i], збільшуємо position на одиницю.
- Якщо значення position дорівнює i, переходимо до наступної ітерації зовнішнього циклу i.
- Пропускаємо дублікати, порівнюючи значення елементів під індексами position та i за допомогою циклу.
- Міняємо місцями елементи під індексами i та position.
- Запускаємо цикл, поки position не буде вказувати на i.
- Повторюємо всі операції, описані всередині циклу j.

Простий приклад реалізації циклічного алгоритму сортування вибором на мові Python(Рисунок 2.3.4)



```
def cycle_sort(arr):
    n = len(arr)
    # Проходимо по кожному елементу масиву
    for cycle_start in range(n):
        item = arr[cycle_start]
        # Знайдемо позицію, куди потрібно помістити елемент
        pos = cycle_start
        for i in range(cycle_start + 1, n):
            if arr[i] < item:
                pos += 1
        # Якщо елемент вже на своєму місці, пропускаємо
        if pos == cycle_start:
            continue
        # Пропускаємо дублікати
        while item == arr[pos]:
            pos += 1
        # Розміщуємо елемент на своїй позиції
        if pos != cycle_start:
            while item != arr[pos]:
                arr[pos], item = item, arr[pos]
                pos += 1

    return arr
# Приклад використання
numbers = [64, 25, 12, 22, 11]
sorted_numbers = cycle_sort(numbers)
print("Відсортований масив:", sorted_numbers)
```

Рисунок 2.3.4- Приклад циклічного алгоритму сортування

Алгоритм циклічного сортування слід використовувати в таких випадках:

- Обмежений обсяг пам'яті: Цей алгоритм не використовує додаткову пам'ять для створення нових масивів, оскільки виконує перестановки на місці. Це робить його корисним у ситуаціях з обмеженими ресурсами пам'яті.
- Малий розмір масиву: Циклічне сортування є ефективним для невеликих масивів. Хоча його складність становить $O(n^2)$, для малих обсягів даних це може бути прийнятним, оскільки кількість операцій буде незначною.
- Коли важливі витрати на зміну елементів: Якщо зміна елементів є витратною операцією (наприклад, при обробці великих об'єктів), цикл-сортування може бути корисним, оскільки він мінімізує кількість перестановок, виконуючи їх тільки тоді, коли елемент ставиться на своє місце.
- Відомі значення діапазону: Якщо діапазон значень, що сортуються, відомий заздалегідь, циклічне сортування може бути ефективним, оскільки дозволяє безпосередньо маніпулювати елементами в масиві.

- Відсутність дублікатів: Алгоритм краще працює в умовах, коли в масиві немає дублікатів, оскільки це спрощує логіку перестановок.

Швидке сортування

Алгоритм швидкого сортування працює наступним чином: він визначає так званий «стержень» і розбиває масив на підмасиви відносно цього «стержня», які потім сортуються.

Цей алгоритм сортування є нестабільним.

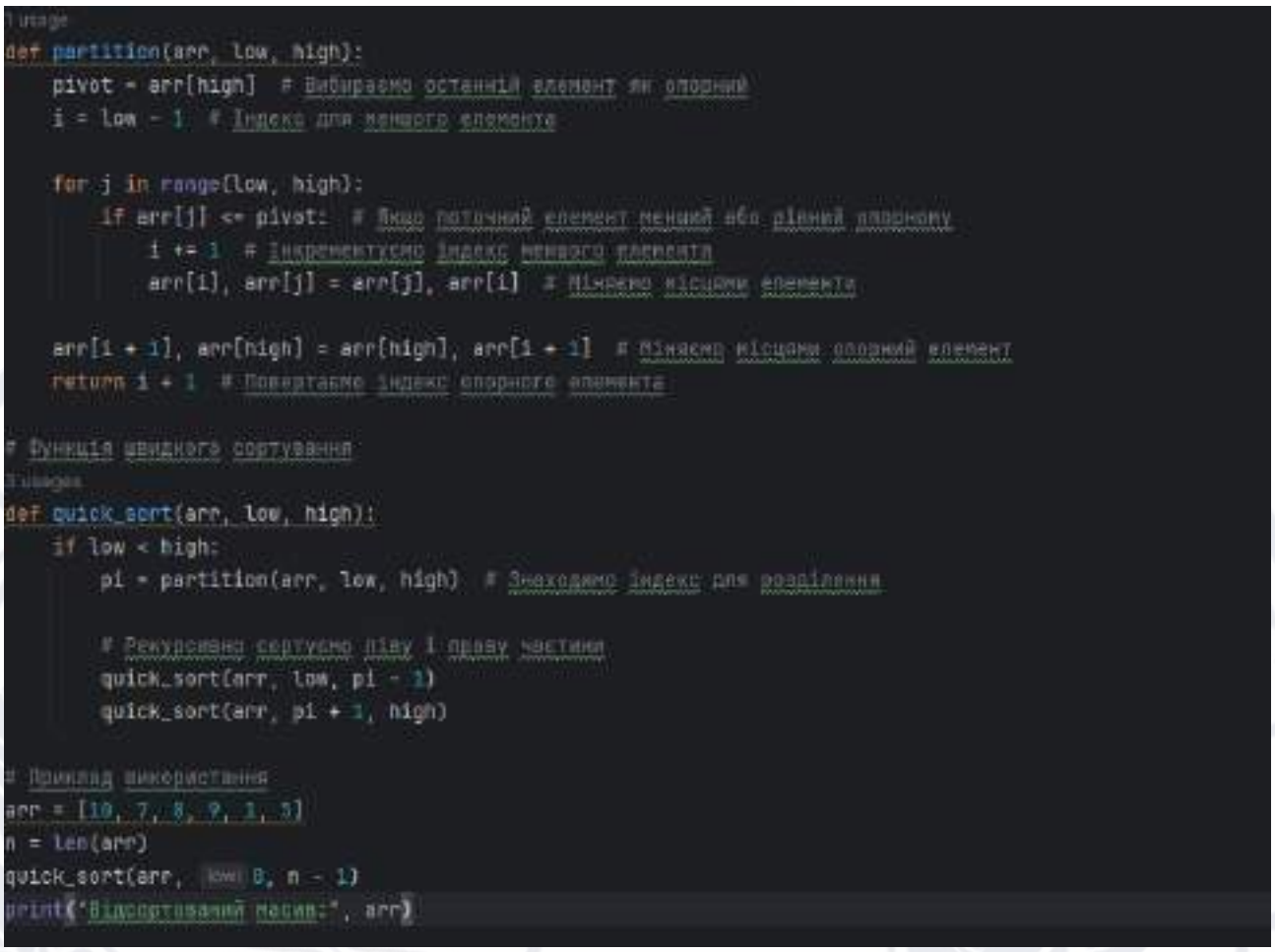
Середня складність алгоритму: O(n * log n).

Складність у найгіршому випадку: $O(n^2)$. Найгірший випадок для алгоритму швидкого сортування виникає тоді, коли опорний елемент (pivot) має максимальне або мінімальне значення у всьому масиві.

Кроки реалізації:

- Перевіряємо, що вказівник на початок масиву (start) не збігається з вказівником на кінець масиву (end).
- Якщо умова виконується, знаходимо індекс елемента, який розділяє масив на дві частини (pi).
- Коли pi знайдено, рекурсивно запускаємо функцію quickSort для кожної з отриманих частин (від start до pi - 1) і (від pi + 1 до end).
- При знаходженні pi визначаємо pivot — опорний елемент і i — індекс, за яким будемо ділити масив.
- Запускаємо цикл по масиву j до передостаннього елемента. При знаходженні елемента, який менший або рівний pivot, міняємо місцями arr[i] і arr[j] та інкрементуємо i.
- Після завершення циклу міняємо місцями arr[i] і arr[end] та повертаємо значення i.

Простий приклад реалізації алгоритму швидкого сортування вибором на мові Python(Рисунок 2.3.5)



```
1 usage
def partition(arr, low, high):
    pivot = arr[high]  # Вибираємо останній елемент як опорний
    i = low - 1  # Індекс для меншого елемента

    for j in range(low, high):
        if arr[j] <= pivot:  # Якщо поточний елемент менший або рівний опорному
            i += 1  # Інкрементуємо індекс меншого елемента
            arr[i], arr[j] = arr[j], arr[i]  # Міняємо місцями елементи

    arr[i + 1], arr[high] = arr[high], arr[i + 1]  # Міняємо місцями опорний елемент
    return i + 1  # Повертаємо індекс опорного елемента

# Функція швидкого сортування
1 usage
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)  # Знаходимо індекс для розділення

        # Рекурсивно сортуємо ліву і праву частини
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

# Приклад використання
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quick_sort(arr, low: 0, n - 1)
print("Відсортований масив:", arr)
```

Рисунок 2.3.5- Приклад алгоритму швидкого сортування на Python

Швидке сортування слід використовувати в таких випадках:

1. Великий обсяг даних: Алгоритм працює швидше для великих масивів у порівнянні з іншими алгоритмами сортування, такими як сортування вибором чи бульбашкове сортування, завдяки середній складності O(n * log n). Це робить його ефективним для масивів з тисячами або навіть мільйонами елементів.
2. Збалансованість даних: Швидке сортування є найбільш ефективним, коли дані масиву збалансовані або розподілені рівномірно. У такому випадку його середня складність O(n * log n) забезпечує швидку роботу.
3. Обмежена пам'ять: Quick Sort є алгоритмом **сортування на місці**, тобто він не потребує додаткової пам'яті для створення копій масиву, на відміну від сортування злиттям (Merge Sort). Він змінює елементи масиву без створення додаткових масивів.

4. Випадковий розподіл елементів: Алгоритм швидкого сортування зазвичай працює добре на масивах з випадковим порядком елементів, оскільки це зменшує ймовірність найгіршого сценарію (коли опорний елемент є найбільшим або найменшим).
5. Швидке виконання в середньому: У багатьох практичних випадках швидке сортування є найшвидшим методом завдяки своєму ефективному розбиттю масиву і мінімальній кількості порівнянь та переміщень елементів [22].

Коли не слід використовувати швидке сортування:

1. Майже відсортовані дані: У випадку, якщо масив уже майже відсортований або має структуру з великими групами однакових елементів, алгоритм може впасти до своєї найгіршої складності $O(n^2)$. У таких випадках інші алгоритми, як-от сортування злиттям або сортування вставками, можуть бути кращими.
2. Нестабільність: Quick Sort є нестабільним алгоритмом, тобто він не зберігає відносний порядок елементів з однаковими значеннями. Якщо вам важливо, щоб однакові елементи залишилися в початковому порядку, слід розглянути інші алгоритми, такі як сортування злиттям.
3. Найгірший сценарій: Найгірший сценарій для швидкого сортування трапляється, коли опорний елемент є найменшим або найбільшим у масиві, що може звести ефективність до $O(n^2)$. Щоб уникнути цього, часто використовують модифіковані версії алгоритму з рандомізацією або вибором середнього елемента як опорного.

Алгоритми пошуку

Тепер розглянемо основні алгоритми для пошуку у наявних даних :

- Лінійний пошук
- Бінарний пошук
- Пошук у ширину та глибину

- Алгоритм Дейкстри

Подібно до алгоритмів сортування, ми розглянемо їхню імплементацію та візуалізацію, а також розберемо, для чого вони були створені, як і де застосовуються.[22]

Лінійний пошук

Лінійний пошук — це найпростіший алгоритм пошуку, який працює як з відсортованими, так і з невідсортованими масивами [23].

Складність алгоритму: O(n).

Кроки реалізації:

1. Запускаємо цикл по масиву і.
2. Перевіряємо поточний елемент на відповідність шуканому елементу; якщо він не відповідає, йдемо далі. Якщо елемент відповідає шуканому, повертаємо його індекс.
3. Якщо елемента немає в масиві, повертаємо -1.

Простий приклад реалізації алгоритму лінійного пошуку на Python (Рисунок 2.3.6)

```python
def linear_search(arr, target):
    # Проходимо по кожному елементу масиву
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Повертаємо індекс, якщо знайдено
    return -1  # Повертаємо -1, якщо елемент не знайдено

# Приклад використання
arr = [10, 23, 45, 70, 11, 15]
target = 70

result = linear_search(arr, target)

if result != -1:
    print(f"Елемент знайдено на індексі: {result}")
else:
    print("Елемент не знайдено")
```

Рисунок 2.3.6- Приклад алгоритму лінійного пошуку на Python

Лінійний пошук слід використовувати в таких випадках:

1. Малий обсяг даних: Лінійний пошук має часову складність O(n), що означає, що він не є дуже ефективним для великих масивів. Однак, для невеликих наборів даних (до кількох десятків або сотень елементів), він може бути достатньо швидким і простим у реалізації.
2. Невідсортовані дані: Лінійний пошук працює однаково добре для відсортованих і невідсортованих масивів. Якщо дані не впорядковані, і вам потрібно знайти елемент, лінійний пошук буде хорошим варіантом, оскільки він не вимагає попереднього сортування.
3. Пошук першого входження: Якщо вам важливо знайти перше входження елемента у масиві, лінійний пошук може бути корисним, оскільки він послідовно перевіряє кожен елемент.
4. Складні або нечислові структури даних: Якщо дані представлені нечисловими елементами або складними структурами, які не можуть бути

легко відсортовані, лінійний пошук є єдиним методом, який можна застосувати для знаходження конкретного елемента.

5. Окремі випадки або рідкісний доступ до даних: Якщо вам потрібно зробити лише кілька запитів на пошук в масиві, то простота лінійного пошуку може переважати над необхідністю застосовувати більш складні алгоритми, як-от бінарний пошук.[23]

Коли не слід використовувати лінійний пошук:

- Великі обсяги даних: Для великих масивів пошук елемента за допомогою лінійного пошуку може зайняти занадто багато часу через його лінійну складність O(n). Для великих наборів даних краще використовувати більш ефективні алгоритми, як-от бінарний пошук (але тільки для відсортованих масивів).
- Відсортовані дані: Якщо дані вже відсортовані, лінійний пошук не використовує цього факту для прискорення процесу. У такому випадку ефективніше буде застосувати бінарний пошук з складністю O(log n).

Бінарний пошук

Основна ідея бінарного пошуку полягає у поділі масиву навпіл і відсіченні непідходящої частини. Алгоритм має сенс використовувати лише для відсортованих масивів.

Складність алгоритму:

- У найкращому випадку: O(1).
- У середньому випадку: O(log n).

Кроки реалізації:

- Створюємо два вказівники: на початок масиву і на кінець масиву.
- Запускаємо цикл, який ітерується доти, доки вказівник на початок не збігається з вказівником на кінець масиву.

- Усередині циклу створюємо вказівник на середину масиву.
- Перевіряємо, чи середній елемент дорівнює шуканому. Якщо так, повертаємо його індекс.
- Якщо середній елемент менший за шуканий, зміщуємо вказівник на початок на mid + 1. Інакше зміщуємо вказівник на кінець на mid - 1.
- Якщо шуканий елемент не знайдено в масиві, повертаємо -1.[23]

Простий приклад реалізації алгоритму бінарного пошуку на Python (Рисунок 2.3.7)

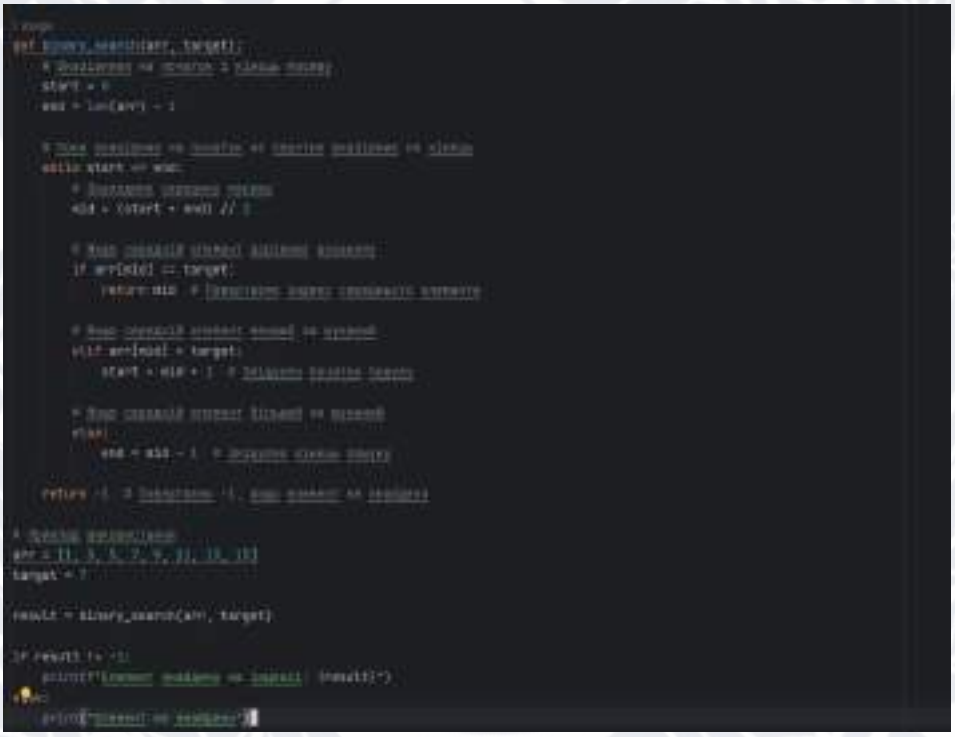


Рисунок 2.3.7- Приклад алгоритму бінарного пошуку на Python

Бінарний пошук слід використовувати в таких випадках:

1. Дані відсортовані: Бінарний пошук працює тільки з відсортованими масивами. Якщо ваші дані вже впорядковані за зростанням або спаданням, бінарний пошук може значно прискорити пошук потрібного елемента.
2. Великі набори даних: Завдяки своїй логарифмічній складності O(log n), бінарний пошук є набагато ефективнішим за лінійний пошук O(n) для

великих масивів даних. Він швидко звужує область пошуку, поділяючи масив навпіл на кожному кроці.

3. Пошук елемента за точним значенням: Якщо потрібно знайти конкретне значення в масиві, бінарний пошук є ідеальним вибором для цього завдання.
4. Статичні набори даних: Якщо набір даних не змінюється часто, а запити на пошук відбуваються регулярно, бінарний пошук є ідеальним вибором, оскільки він не потребує постійного оновлення структури даних.

Коли не слід використовувати бінарний пошук:

- Невідсортовані дані: Якщо масив не відсортований, бінарний пошук не може бути застосований. У такому випадку вам потрібно або відсортувати дані перед пошуком, або використовувати лінійний пошук.
- Часті зміни в масиві: Якщо масив часто змінюється (вставка, видалення елементів), постійне сортування для забезпечення коректної роботи бінарного пошуку може бути неефективним.[23]

Пошук в глибину

Пошук в глибину — це алгоритм обходу або пошуку в таких структурах даних, як дерева або графи. Він базується на структурі даних, відомій як стек. Алгоритм починає роботу з кореневого вузла (у випадку графа кореневим вузлом вибирається будь-який довільний вузол) і перш ніж повернутися назад, намагається пройти якомога далі по кожній гілці.

Складність алгоритму: O(V + E), де V — кількість вершин, E — кількість ребер.

Кроки реалізації:

- У випадку обходу графа необхідно проініціалізувати об'єкт visited, який буде зберігати вершини, пройдені алгоритмом, щоб уникнути зациклення. Для дерев цей крок можна пропустити.

- Створюємо масив stack, який виконуватиме роль стека, та одразу додаємо в нього вершину, з якої почнемо обхід.
- Запускаємо цикл, який ітеруватиметься доти, поки стек не стане порожнім.
- Витягуємо перше значення зі стека та присвоюємо його змінній vert.
- Перевіряємо, чи має ця вершина дочірні вершини.
- Якщо дочірні вершини знайдені, додаємо їх на початок стека.
- У випадку обходу графа запускаємо цикл, який проходить по дочірніх вершинах, і якщо вони не були пройдені раніше, додаємо їх у стек.

Простий приклад реалізації алгоритму пошуку в глибину на Python (Рисунок 2.3.8)

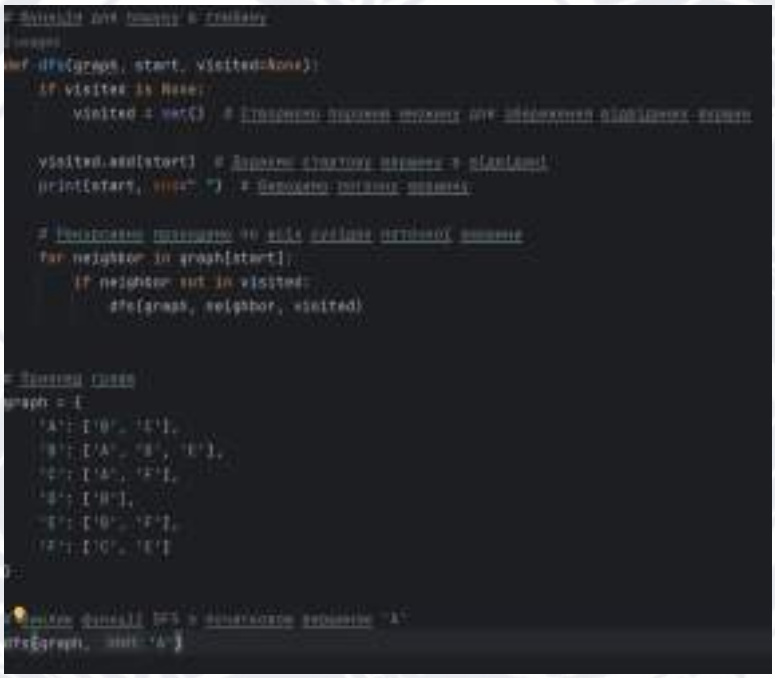


Рисунок 2.3.8- Приклад алгоритму пошуку в глибину на Python

Пошук в глибину (DFS) слід використовувати в таких випадках:

1. Обхід глибоких структур даних: Якщо структура даних, з якою працюєте (дерево або граф), має велику глибину або складні ієрархічні зв'язки, DFS дозволяє спускатися якомога глибше по кожній гілці, перш ніж повернутися назад і перейти до наступної. Наприклад, DFS добре підходить для дерев, де важливо обійти всі нащадки кожної вершини.

2. Вивчення всіх шляхів: DFS використовується, коли важливо дослідити всі можливі шляхи в графі. Це може бути корисно для задач, де потрібно знайти всі можливі рішення або маршрути.
3. Пошук у лабіринтах або подібних структурах: DFS добре підходить для задач, де потрібно знайти шлях через лабіринт або подібну систему, оскільки він глибоко досліджує кожен маршрут до його кінця, перш ніж переходити до іншого.
4. Пошук компонент зв'язності: DFS застосовується для визначення компонент зв'язності в графах, тобто підмножин вершин, які з'єднані між собою.
5. Відстеження циклів у графах: DFS використовується для виявлення циклів у графах. Якщо під час обходу зустрічається вершина, яка вже була відвідана, можна зробити висновок про наявність циклу.
6. Задачі на перевірку двочастковості графа: Алгоритм DFS допомагає перевірити, чи є граф двочастковим, оскільки під час обходу можна маркувати вершини та перевіряти, чи існує конфлікт у розфарбуванні.

Коли не слід використовувати DFS:

- Коли важлива мінімальна довжина шляху: DFS не гарантує знаходження найкоротшого шляху між двома вершинами в графі. Для цього краще використовувати пошук в ширину (BFS) або алгоритм Дейкстри.
- Коли граф дуже широкий: Якщо граф або дерево мають велику ширину, DFS може бути неефективним, оскільки занадто глибоке занурення в одне з піддерев може відвести далеко від рішення.[23]

2.4 Інструментарій

PyCharm — це потужне інтегроване середовище розробки (IDE) для Python, розроблене компанією **JetBrains**. Воно надає широкий набір інструментів для ефективного написання, тестування та налагодження програмного коду, а також підтримує багато популярних технологій та бібліотек.

Основні функції та переваги PyCharm:

- Підтримка Python:
  - PyCharm надає потужну підтримку для написання коду на Python, включаючи автодоповнення, підсвічування синтаксису та швидку навігацію по коду.
  - PyCharm також підтримує Django, Flask та інші веб-фреймворки для розробки веб-додатків на Python.
- Інструменти для налагодження та тестування:
  - PyCharm пропонує вбудований дебагер, який дозволяє запускати програму покроково, встановлювати точки зупинки та відслідковувати значення змінних під час виконання програми.
  - Також є підтримка для написання і запуску тестів за допомогою бібліотек, таких як unittest, pytest і doctest.
- Інтеграція з системами контролю версій:
  - PyCharm інтегрується з системами контролю версій, такими як Git, Mercurial та Subversion, що дозволяє легко керувати версіями коду, створювати та зливати гілки, переглядати історію змін тощо.
- Рефакторинг та аналіз коду:
  - PyCharm допомагає покращувати якість коду за допомогою функцій рефакторингу: перейменування змінних, методів або класів, зміна сигнатури функцій, автоматичне виправлення помилок та попереджень.
  - PyCharm пропонує аналіз коду для пошуку помилок, дублікатів коду та інших проблем.
- Вбудована підтримка Jupyter Notebook:
  - PyCharm дозволяє використовувати Jupyter Notebook безпосередньо у IDE, що особливо корисно для наукових досліджень і аналізу даних.
- Робота з базами даних:
  - PyCharm має вбудовані інструменти для підключення до баз даних, таких як MySQL, PostgreSQL, SQLite та інших. Це дозволяє переглядати структуру баз даних, виконувати SQL-запити та керувати даними.
- Розширюваність:
  - PyCharm має безліч плагінів, які розширюють його функціональність. Серед популярних плагінів — підтримка інших мов програмування, інструменти для роботи з Docker, Kubernetes, та інтеграція з різними фреймворками.

- Підтримка веб-технологій:
    - PyCharm також підтримує HTML, CSS, JavaScript, TypeScript, що робить його зручним для розробки веб-додатків [24].

IntelliJ IDEA — це потужне інтегроване середовище розробки (IDE), створене компанією **JetBrains**, яке широко використовується для розробки програмного забезпечення на мовах Java, Kotlin та інших. IntelliJ IDEA пропонує великий набір інструментів для ефективної роботи з кодом, включаючи можливості для рефакторингу, аналізу коду, інтеграції з системами контролю версій та підтримки численних фреймворків.

Основні функції та переваги IntelliJ IDEA:

- Підтримка Java та інших мов програмування:
    - IntelliJ IDEA спочатку створювалась для розробки на Java, тому має дуже глибоку інтеграцію з мовою. IDE пропонує інтелектуальне автодоповнення, потужний аналіз коду, швидку навігацію, та підсвічування помилок у реальному часі.
    - Крім Java, IntelliJ IDEA підтримує багато інших мов програмування, таких як Kotlin, Scala, Groovy, JavaScript, Python, Ruby, та інші через плагіни.
- Інструменти для рефакторингу та навігації по коду:
    - IntelliJ IDEA забезпечує безліч інструментів для рефакторингу коду, таких як перейменування класів, методів і змінних, зміна сигнатури методів та автоматичне виправлення проблем. Це дозволяє безпечно і швидко змінювати структуру коду без побоювань за його коректність.
    - Швидка навігація по проекту дозволяє розробнику швидко переміщатися між класами, методами та іншими елементами проекту.
- Підтримка фреймворків:

- IntelliJ IDEA підтримує багато популярних фреймворків для веб-розробки, таких як Spring, Spring Boot, Hibernate, Grails, Vaadin, JavaServer Faces (JSF) та інші.
- Також є підтримка для мобільної розробки через Android SDK, інтеграція з Java EE, Jakarta EE, MicroProfile для створення корпоративних додатків.

- Інструменти для налагодження та тестування:
  - IntelliJ IDEA містить вбудовані інструменти для налагодження (debugging), що дозволяють відслідковувати виконання коду, аналізувати стек викликів, перевіряти змінні на різних етапах виконання програми та ставити точки зупинки (breakpoints).
  - IDE підтримує основні бібліотеки для тестування, такі як JUnit, TestNG, Mockito, що полегшує процес тестування та написання юніт-тестів.
- Інтеграція з системами контролю версій:
  - IntelliJ IDEA інтегрується з Git, Mercurial, Subversion та іншими системами контролю версій, що дозволяє легко керувати змінами в коді, виконувати merge, переглядати історію комітів та вирішувати конфлікти.
- Інструменти для роботи з базами даних:
  - IDE має вбудовані засоби для роботи з базами даних. Це дозволяє виконувати SQL-запити, переглядати структуру баз даних, редагувати дані та керувати підключеннями до баз даних, таких як MySQL, PostgreSQL, Oracle тощо.
- Інструменти для побудови та керування проектами:
  - IntelliJ IDEA підтримує популярні інструменти для управління збірками проектів, такі як Maven, Gradle, Ant. Це дозволяє автоматизувати збірку, тестування і розгортання проектів.
- Підтримка контейнеризації та DevOps:

- IntelliJ IDEA інтегрується з такими технологіями, як Docker, Kubernetes, що дозволяє легко створювати, розгортати та управляти контейнеризованими додатками.
- IDE також підтримує інструменти CI/CD, такі як Jenkins, для автоматизації процесу розробки.

- Плагіни та розширюваність:
  - IntelliJ IDEA має величезну кількість плагінів, які можна додати для розширення її функціональності. Наприклад, є плагіни для підтримки додаткових мов програмування, інструменти для дизайну інтерфейсів, інтеграції з різними API та іншими інструментами розробки.[24]

Visual Studio — це інтегроване середовище розробки (IDE), розроблене компанією Microsoft, яке використовується для створення різних типів програмного забезпечення, включаючи настільні, мобільні, веб-додатки, а також хмарні рішення. Visual Studio підтримує широкий спектр мов програмування і фреймворків, що робить його одним із найпопулярніших середовищ для розробки.

Основні функції та можливості Visual Studio:

- Підтримка багатьох мов програмування:
  - Visual Studio надає інструменти для розробки на таких мовах, як C#, C++, Visual Basic, Python, JavaScript, TypeScript, F#, PHP, а також підтримує мови через плагіни, наприклад, Go, Rust, Swift.
  - IDE забезпечує інтелектуальне автодоповнення коду (IntelliSense), що полегшує роботу з різними мовами і пришвидшує процес написання коду.
- Інструменти для розробки додатків на .NET:

    - Visual Studio широко використовується для розробки на платформі .NET, що включає мови C#, F#, Visual Basic, а також підтримує розробку під ASP.NET для створення веб-додатків.
    - IDE дозволяє легко розробляти і тестувати хмарні додатки з використанням Azure, що є корисним для створення хмарних рішень.
- Інструменти для розробки мобільних додатків:
    - Завдяки інтеграції з Xamarin, Visual Studio дозволяє створювати кросплатформні мобільні додатки для Android та iOS, використовуючи C# та загальну базу коду.
    - Xamarin.Forms надає інструменти для створення нативних інтерфейсів користувача для мобільних додатків.
- Інструменти для веб-розробки:
    - Visual Studio підтримує створення веб-додатків з використанням фреймворків, таких як ASP.NET, Node.js, Angular, React, Vue.js.
    - IDE містить засоби для інтеграції з хмарними сервісами, такими як Azure, що дозволяє легко розгортати веб-додатки безпосередньо з середовища розробки.
- Інструменти для роботи з базами даних:
    - Visual Studio підтримує SQL Server, MySQL, SQLite та інші бази даних. Вбудовані інструменти дозволяють легко підключатися до баз даних, писати SQL-запити, аналізувати і модифікувати дані.
- Налагодження та тестування:
    - IDE забезпечує потужні інструменти для налагодження (debugging), які дозволяють відслідковувати виконання коду, використовувати точки зупинки, аналізувати стан змінних і стек викликів.
    - Visual Studio підтримує основні бібліотеки для тестування, такі як MSTest, NUnit, xUnit, що дозволяє ефективно організовувати процес тестування додатків.
- Інтеграція з системами контролю версій:

- Visual Studio інтегрується з Git, GitHub, Azure DevOps, Subversion, дозволяючи розробникам легко керувати змінами в коді, виконувати коміти, вирішувати конфлікти та переглядати історію проекту.

- Розробка ігор:
  - За допомогою Visual Studio можна розробляти ігри для платформ Unity, Unreal Engine, Cocos, а також ігри на базі DirectX або OpenGL. IDE забезпечує всі необхідні інструменти для роботи з графікою, фізикою та логікою ігор.
- DevOps і CI/CD:
  - Visual Studio інтегрується з Azure DevOps і іншими інструментами для безперервної інтеграції та доставки (CI/CD). Це дозволяє налаштовувати процеси автоматизованої збірки, тестування і розгортання додатків.
- Підтримка контейнеризації:
  - IDE підтримує роботу з контейнерами Docker і дозволяє розробляти та тестувати контейнеризовані додатки. Це спрощує процес налаштування середовищ розробки та розгортання програм у хмарі або на серверах.

MySQL — це одна з найпопулярніших і широко використовуваних систем управління реляційними базами даних (РСУБД). Вона була розроблена компанією MySQL AB і зараз підтримується Oracle Corporation після її придбання. MySQL застосовується для зберігання, організації та управління великими обсягами даних, часто використовується в веб-розробці та хмарних додатках.[24]

Основні характеристики MySQL:

- Реляційна база даних:
  - MySQL є реляційною базою даних, що означає, що дані зберігаються в таблицях, які можуть бути пов'язані між собою за допомогою

ключів (первинних та зовнішніх). Це дозволяє організовувати дані в структурованому вигляді і забезпечує цілісність інформації.

- SQL (Structured Query Language):
    - MySQL використовує мову запитів SQL, що є стандартом для управління даними в реляційних базах даних. За допомогою SQL можна виконувати операції вибірки даних, вставки, оновлення, видалення, створення і модифікації таблиць, а також керувати правами доступу користувачів.
- Підтримка транзакцій:
    - MySQL підтримує ACID-транзакції (атомарність, узгодженість, ізольованість, надійність), що дозволяє виконувати складні операції з даними з гарантією їхньої цілісності та узгодженості.
    - Транзакції підтримуються за допомогою механізму InnoDB, який є основним рушієм для транзакційних операцій в MySQL.
- Масштабованість та продуктивність:
    - MySQL добре підходить як для малих, так і для великих проектів. Вона може обробляти значні обсяги даних і забезпечувати високу продуктивність завдяки підтримці індексів, оптимізованих запитів і кешування.
    - MySQL також підтримує шардінг та кластеризацію, що дозволяє розподіляти навантаження на кілька серверів.
- Безпека:
    - MySQL забезпечує високий рівень безпеки завдяки підтримці шифрування з'єднань, аутентифікації користувачів і управління правами доступу. Користувачам можна призначати різні ролі та дозволи для роботи з базами даних.
- Реплікація:
    - MySQL підтримує реплікацію даних, що дозволяє створювати копії баз даних на різних серверах для забезпечення надійності та доступності. Є можливість налаштування як однонаправленої

реплікації (Master-Slave), так і двонаправленої реплікації (Master-Master).

- Платформи і сумісність:
  - MySQL є кросплатформною системою і може працювати на різних операційних системах, таких як Windows, Linux, macOS.
  - Підтримує різні програмні мови, включаючи PHP, Java, Python, C#, C++, Perl та інші.
- Гнучкість у використанні:
  - MySQL є легкою у встановленні та налаштуванні, що робить її ідеальною для веб-розробки та інших додатків. Вона широко використовується з такими веб-фреймворками та CMS, як WordPress, Drupal, Joomla, Magento, а також із платформами на основі LAMP (Linux, Apache, MySQL, PHP).
- Open Source:
  - MySQL є відкритим програмним забезпеченням, що дозволяє використовувати її безкоштовно, вивчати її код і вносити зміни відповідно до своїх потреб. Однак Oracle також пропонує платні версії з додатковими можливостями для великих підприємств [25].

2.5 Тести

Для визначення ефективності мови програмування та алгоритмів можна провести кілька видів тестів, які охоплюють різні аспекти продуктивності та ефективності. Ось деякі з них:

1. Тести продуктивності (Performance Tests)

- Час виконання (Execution Time): Вимірювання часу, необхідного для виконання певного завдання. Наприклад, обробка великого набору даних, сортування масиву або пошук в базі даних.

- Продуктивність під навантаженням (Load Testing): Оцінка продуктивності при різних рівнях навантаження. Наприклад, виконання алгоритму обробки даних на наборах даних різного розміру.
- Час відгуку (Response Time): Вимірювання часу відгуку системи або алгоритму на запити. Важливо для реальних додатків з інтерактивними користувачами.

2. Тести використання ресурсів (Resource Utilization Tests)

- Використання пам'яті (Memory Usage): Вимірювання обсягу пам'яті, необхідного для виконання завдання. Це може включати як оперативну пам'ять (RAM), так і постійну пам'ять (дисковий простір).
- Процесорне навантаження (CPU Usage): Моніторинг рівня завантаженості процесора під час виконання алгоритму.
- Ефективність збору сміття (Garbage Collection Efficiency): Аналіз впливу механізмів автоматичного управління пам'яттю на продуктивність.

3. Тести масштабованості (Scalability Tests)

- Горизонтальна масштабованість (Horizontal Scalability): Оцінка здатності системи ефективно обробляти збільшення обсягу даних при додаванні нових вузлів або серверів.
- Вертикальна масштабованість (Vertical Scalability): Вимірювання продуктивності при збільшенні ресурсів одного вузла (наприклад, додавання процесорів або оперативної пам'яті).

4. Стрес-тести (Stress Tests)

- Екстремальні навантаження (Extreme Load Testing): Перевірка стійкості системи або алгоритму при значно перевищеному рівні навантаження, ніж очікується в реальних умовах.

- Тестування на збій (Failure Testing): Перевірка поведінки системи при виході за межі допустимих параметрів (наприклад, переповнення пам'яті або перевантаження процесора)

5. Тести зручності розробки та підтримки коду (Code Maintenance Tests)

- Читабельність коду (Code Readability): Оцінка простоти та зрозумілості коду для інших розробників. Це може включати аналіз складності коду, використання коментарів та дотримання стилістичних рекомендацій.
- Легкість тестування (Ease of Testing): Оцінка зручності написання та проведення автоматизованих тестів для коду.
- Модульність та повторне використання (Modularity and Reusability): Вимірювання здатності коду бути розбитим на модулі та використаним повторно в інших проектах.

6. Тести надійності (Reliability Tests)

- Виявлення та виправлення помилок (Bug Detection and Fixing): Оцінка часу та зусиль, необхідних для виявлення та виправлення помилок у коді.
- Стійкість до помилок (Fault Tolerance): Аналіз здатності системи або алгоритму продовжувати роботу у разі виникнення помилок [26].

Приклади тестів:

1. Виконання алгоритмів сортування:

- Виконати алгоритми сортування у датасетах різних розмірів (від 1 000 до 1 000 000 елементів) і виміряйте час виконання та використання пам'яті.

2. Обробка великих файлів:

- Написати програму, яка читає великий текстовий файл, аналізує вміст (наприклад, підраховує частоту слів) і виміряйте час виконання та використання ресурсів.

3. Виконання запитів до бази даних:

- Реалізувати різні типи запитів до великої бази даних (вибірка, вставка, оновлення) і оцінити продуктивність кожного запиту.

4. Паралельна обробка даних:

- Реалізувати алгоритми таким чином, щоб вони використовували багатопоточність або асинхронні обчислення для обробки даних, і порівняти продуктивність з однопоточними варіаціями [27].

Проведення таких тестів дозволить отримати об'єктивну оцінку ефективності мов програмування та алгоритмів для обробки даних, що допоможе зробити обґрунтований вибір при розробці систем [28].

РОЗДІЛ 3. РОЗРОБКА ТА АНАЛІЗ РЕЗУЛЬТАТІВ

Тепер варто поговорити про реалізацію та отримані результати у ході роботи. Під час розробки для алгоритмів пошуку був використаний датасет на 200 тисяч рядків, а для алгоритмів сортування датасет на 18 тисяч рядків. Датасети представлені у вигляді CSV файлів. Датасети були взяті з ресурсу https://catalog.data.gov/dataset.

3.1 Реалізація алгоритмів на різних мовах програмування

Під час реалізації ми використовуємо бібліотеки для читання та взаємодії із CSV файлами, які необхідні для мови програмування.

Спочатку розглянемо реалізацію алгоритмів пошуку:

Алгоритм лінійного пошуку:

Реалізація на мові Python Рисунок 3.1.1

Рисунок 3.1.1. Реалізація на мові Python алгоритму лінійного пошуку

Реалізація на мові Java Рисунок 3.1.2

Рисунок 3.1.2. Реалізація на мові Java алгоритму лінійного пошуку

Реалізація на мові C# Рисунок 3.1.3

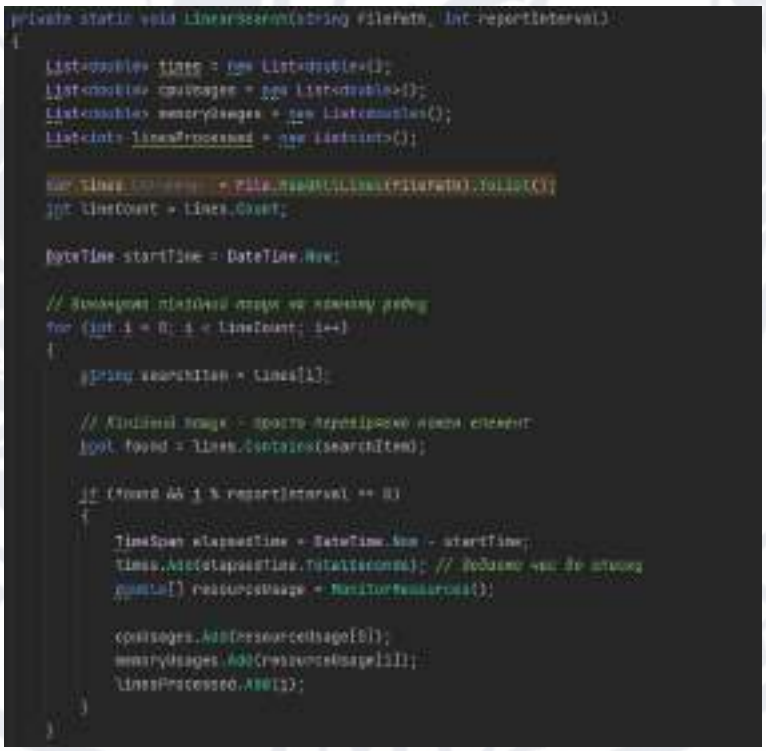


Рисунок 3.1.3. Реалізація на мові C# алгоритму лінійного пошуку

Алгоритм бінарного пошуку:

Реалізація на мові Python Рисунок 3.1.4

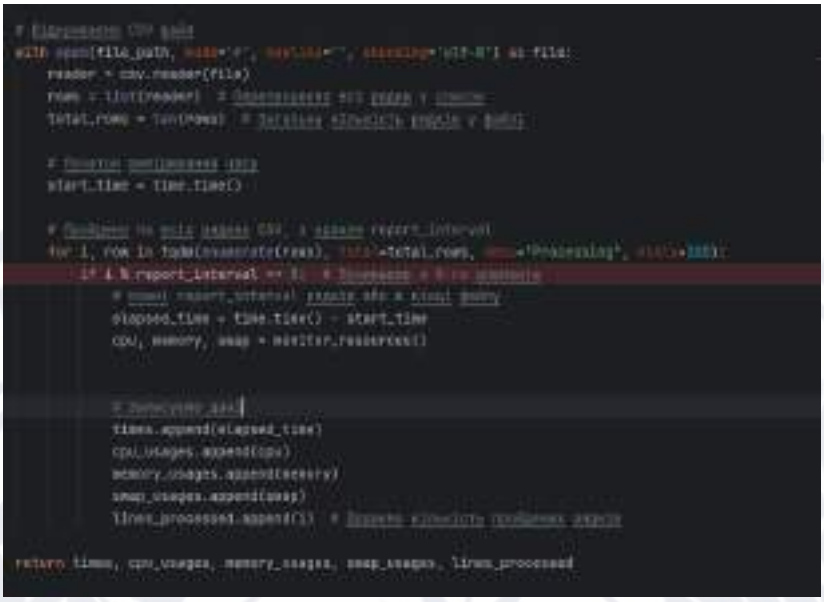


Рисунок 3.1.4. Реалізація на мові Python алгоритму бінарного пошуку

Реалізація на мові Java Рисунок 3.1.5

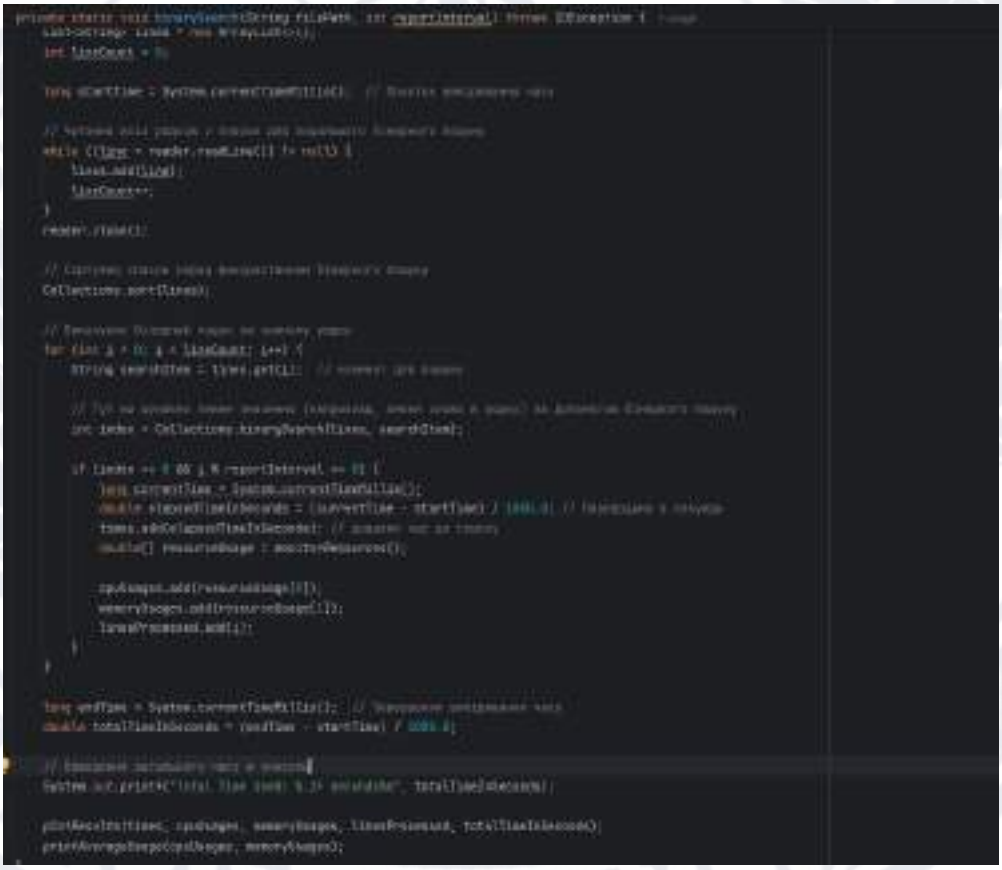


Рисунок 3.1.5 Реалізація на мові Java алгоритму бінарного пошуку

Реалізація на мові C# Рисунок 3.1.6

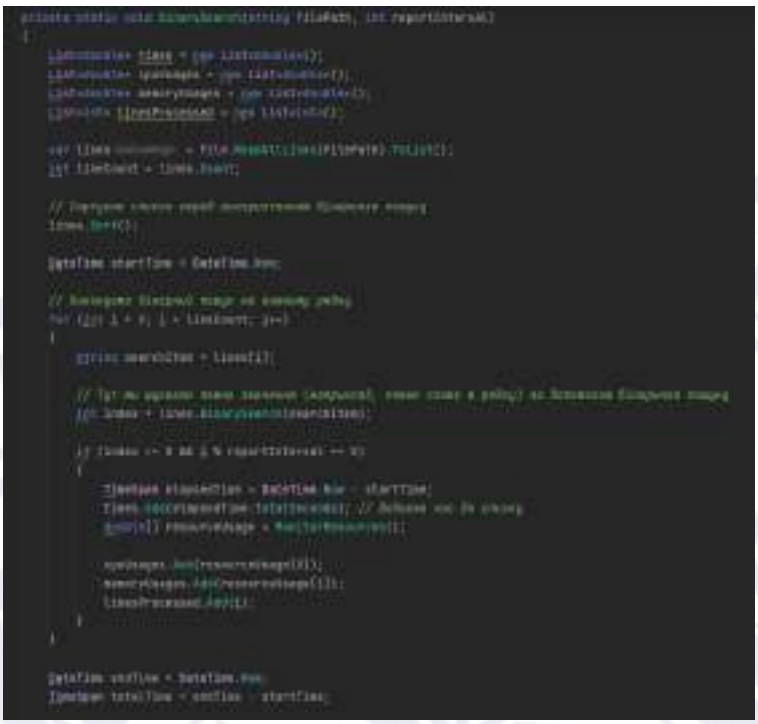


Рисунок 3.1.6 Реалізація на мові C# алгоритму бінарного пошуку

Алгоритм пошуку в глибину:

Реалізація на мові Python Рисунок 3.1.7

Рисунок 3.1.7 Реалізація на мові Python алгоритму пошуку в глибину

Реалізація на мові Java Рисунок 3.1.8

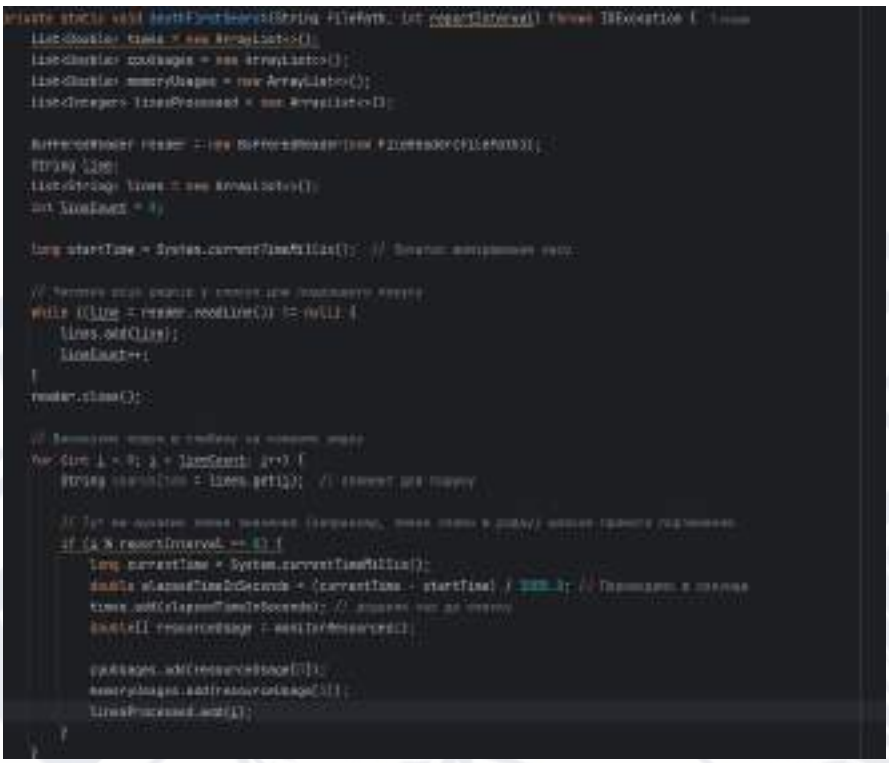


Рисунок 3.1.8 Реалізація на мові Java алгоритму пошуку в глибину

Реалізація на мові C# Рисунок 3.1.9

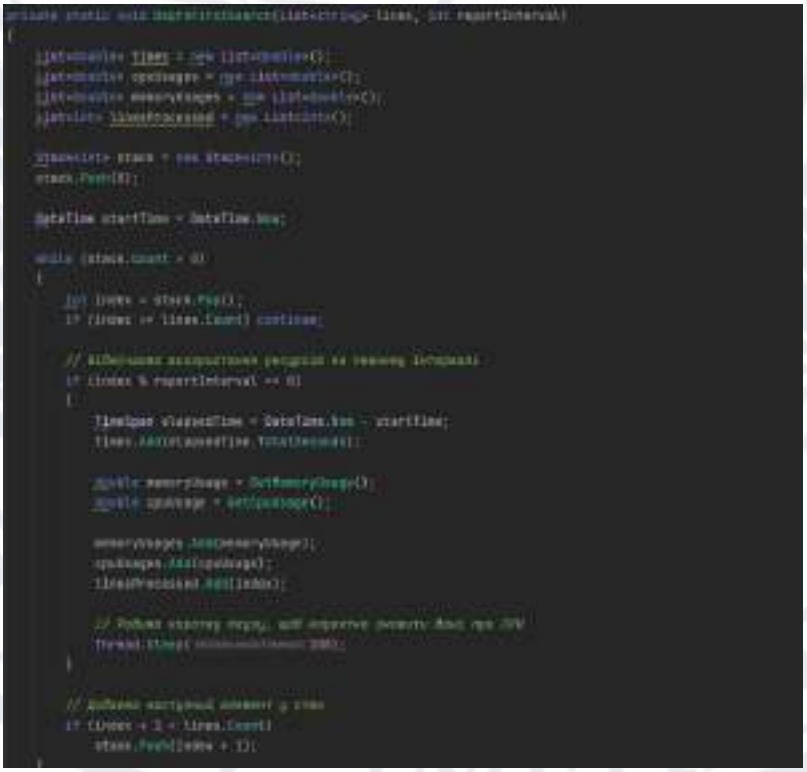


Рисунок 3.1.9 Реалізація на мові C# алгоритму пошуку в глибину

Алгоритм пошуку в ширину:

Реалізація на мові Python Рисунок 3.1.10

Рис. 3.1.10 Реалізація на мові Python алгоритму пошуку в ширину

Реалізація на мові Java Рисунок 3.1.11

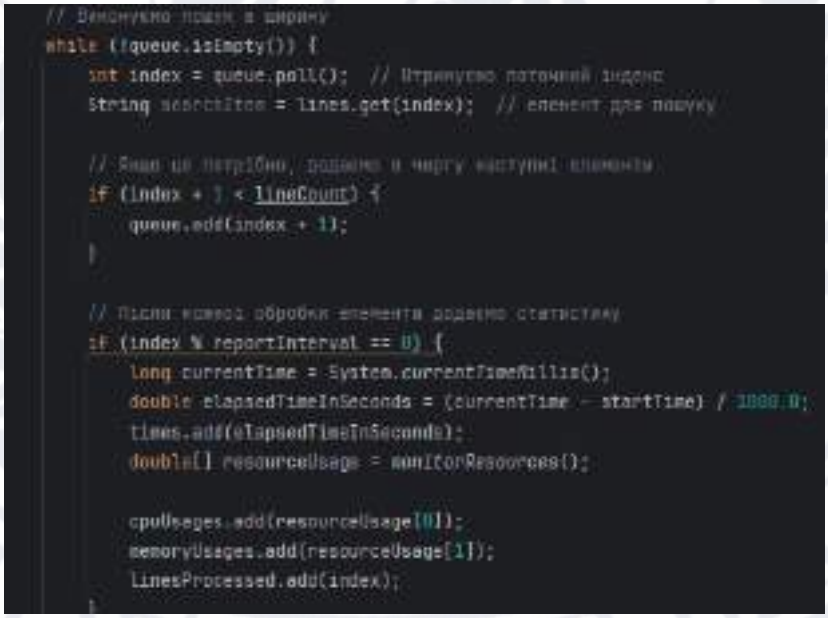


Рис. 3.1.11 Реалізація на мові Java алгоритму пошуку в ширину

Реалізація на мові C# Рисунок 3.1.12

Рисунок 3.1.12 Реалізація на мові C# алгоритму пошуку в ширину

Тепер розглянемо реалізацію алгоритмів сортування. Під час реалізації ми спочатку створюємо новий текстовий файл, щоб скопіювати дані які у нас є у датасеті, а потім вже сортуємо у цьому новому файлі дані, після того як отримали результати, видаляємо цей файл, аби не забивати пам’ять [30].

Бульбашковий алгоритм:

Реалізація бульбашкового алгоритму на мові Python Рисунок 3.1.13

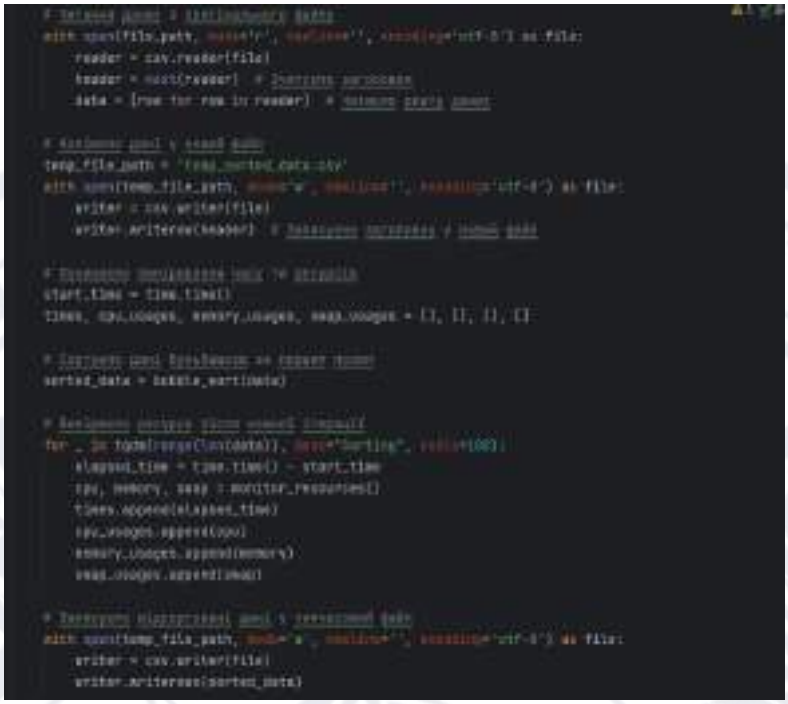


Рисунок 3.1.13 Реалізація на мові Python бульбашкового алгоритму

Реалізація бульбашкового алгоритму на мові Java Рисунок 3.1.14

Рисунок 3.1.14 Реалізація на мові Java бульбашкового алгоритму

Реалізація бульбашкового алгоритму на мові C# Рисунок 3.1.15

```
private static void BubbleSort(List<string> lines)
{
    for (int i = 0; i < lines.Count - 1; i++)
    {
        for (int j = 0; j < lines.Count - 1 - i; j++)
        {
            string[] line1 = lines[j].Split(',');
            string[] line2 = lines[j + 1].Split(',');

            // Порівнюємо перше поле (порівнюємо за юнікодом)
            string firstField1 = line1[0].Trim();
            string firstField2 = line2[0].Trim();

            // Порівнюємо рядки за їхнім юнікодом
            if (firstField1.CompareTo(firstField2) > 0)
            {
                // Якщо рядок1 більший за рядок2, міняємо місцями
                var temp string = lines[j];
                lines[j] = lines[j + 1];
                lines[j + 1] = temp;
            }
        }
    }
}
```

Рисунок 3.1.15 Реалізація на мові C# бульбашкового алгоритму

Через майже однакову ефективність із бульбашковим алгоритмом, було вирішено не реалізовувати алгоритм сортування вибором. А також через недостатню кількість оперативної пам’яті було вирішено не реалізовувати на мові Python алгоритм циклічного сортування.

Циклічний алгоритм:

Реалізація циклічного алгоритму на мові Java Рисунок 3.1.16

Рис 3.1.16 Реалізація циклічного алгоритму на мові Java

Реалізація циклічного алгоритму на мові C# Рисунок 3.1.17

```
private static void CyclicSort(List<string> lines)
{
    int n = lines.Count;

    // Проходимо через всі елементи
    for (int cycleStart = 0; cycleStart < n - 1; cycleStart++)
    {
        string[] line1 = lines[cycleStart].Split(',');
        string firstField1 = line1[0].Trim();

        // Перевіряємо, чи елемент вже на своєму місці
        if (firstField1.CompareTo(firstField1) == 0)
            continue;

        // Потрібно виконати циклічний обмін
        string temp = lines[cycleStart];
        lines[cycleStart] = lines[cycleStart + 1];
        lines[cycleStart + 1] = temp;
    }
}
```

Рис 3.1.17 Реалізація циклічного алгоритму на мові C#

Алгоритм швидкого сортування:

Реалізація алгоритму швидкого сортування на мові Python Рисунок 3.1.18

Рисунок 3.1.18 Реалізація алгоритму швидкого сортування на мові Python

Реалізація алгоритму швидкого сортування на мові Java Рисунок 3.1.19

Рисунок 3.1.19 Реалізація алгоритму швидкого сортування на мові Java

Реалізація алгоритму швидкого сортування на мові C# Рисунок 3.1.20

```
private static void QuickSort(List<string> lines, int low, int high)
{
    if (low < high)
    {
        int pivotIndex = Partition(lines, low, high);

        // Рекурсивно сортуємо дві частини
        QuickSort(lines, low, pivotIndex - 1);  // Ліва частина
        QuickSort(lines, pivotIndex + 1, high); // Права частина
    }
}

// Функція поділу для швидкого сортування
private static int Partition(List<string> lines, int low, int high)
{
    // Вибираємо останній елемент як опорний
    string pivot = lines[high];
    string[] pivotLine = pivot.Split(',');
    string pivotField = pivotLine[0].Trim();

    int i = low - 1;  // Індекс менших елементів

    // Переміщаємо елементи менші за опорний елемент вліво, більші — вправо
    for (int j = low; j < high; j++)
    {
        string[] line = lines[j].Split(',');
        string field = line[0].Trim();

        if (field.CompareTo(pivotField) <= 0)
        {
            i++;
            // Міняємо місцями елементи
            Swap(lines, i, j);
        }
    }

    // Міняємо місцями опорний елемент з елементом після останнього меншого елемента
    Swap(lines, i + 1, high);
    return i + 1;
```

Рисунок 3.1.20 Реалізація алгоритму швидкого сортування на мові C#

3.2 Аналіз отриманих даних

Отже, ми реалізували програмний код, прийшов час проаналізувати отримані дані в ході роботи [29]. Так як у Java та C# немає адекватних для реалізації бібліотек із графіками, було вирішено показати отримані результати у консолі, з Python же результати в обидвох варіантах. Також були зроблені діаграми як швидкодії, так і використання ресурсів комп’ютера. Також на мові Java, через ядро актуальні дані з процесора показувались в 0 значеннях, тому в діаграмах були використані приблизні значення з системи моніторингу комп’ютера.

Також комплектуючі які були використані під час розробки та аналізу даних:

- Процесор - AMD Ryzen 5 5600
- RAM - 16 ГБ DDR4
- GPU - Nvidia RTX 3060 TI
- Операційна система - Windows 10 Pro

Алгоритми пошуку:

Лінійний пошук:

Результати лінійного пошуку на мові Python рис 3.2.1, 3.2.2

Рисунок 3.2.1 Результати лінійного пошуку на мові Python

Рисунок 3.2.2 Результати лінійного пошуку на мові Python

Результати лінійного пошуку на мові Java рис 3.2.3

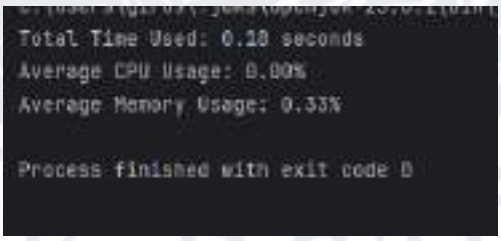


Рисунок 3.2.3 Результати лінійного пошуку на мові Java

Результати лінійного пошуку на мові C# рис 3.2.4

```
Total Time Used: 64.6389615 seconds
Average CPU Usage: 2.09%
Average Memory Usage: 63.58%

Process finished with exit code 0.
```

Рисунок 3.2.4 Результати лінійного пошуку на мові C#

Порівняльна діаграма швидкодії лінійного пошуку Рисунок 3.2.5

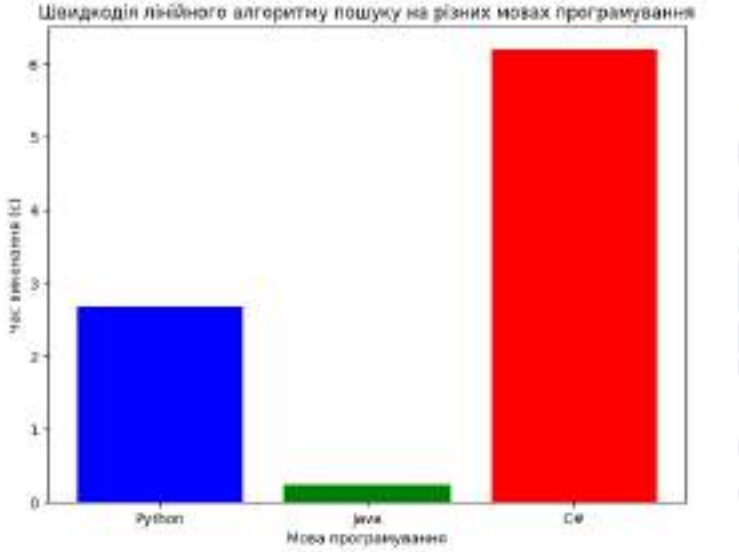


Рисунок 3.2.5 Порівняльна діаграма швидкодії лінійного пошуку

Порівняльна діаграма використання процесору під час лінійного пошуку Рисунок 3.2.6

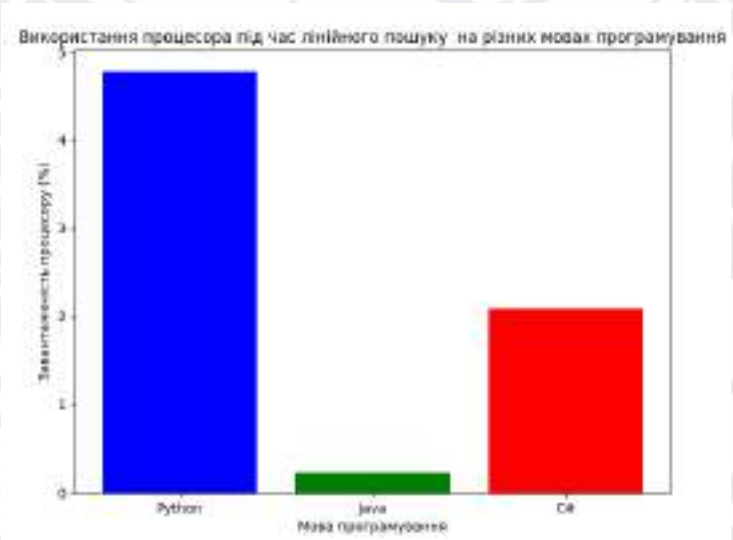


Рисунок 3.2.6 Порівняльна діаграма використання процесору під час лінійного пошуку

Порівняльна діаграма використання оперативної пам’яті під час обробки лінійним пошуком Рисунок 3.2.7

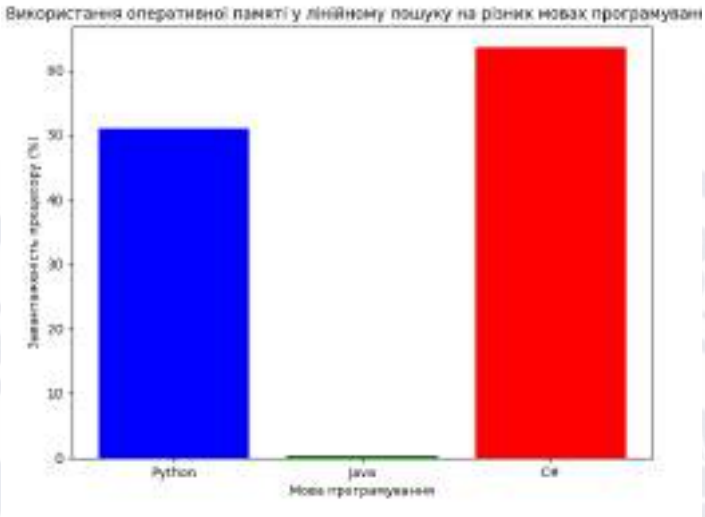


Рисунок 3.2.7 Порівняльна діаграма використання оперативної пам’яті під час обробки лінійним пошуком

Бінарний пошук:

Результати лінійного пошуку на мові Python рис 3.2.8, 3.2.9

Рисунок 3.2.8 Результати бінарного пошуку на мові Python

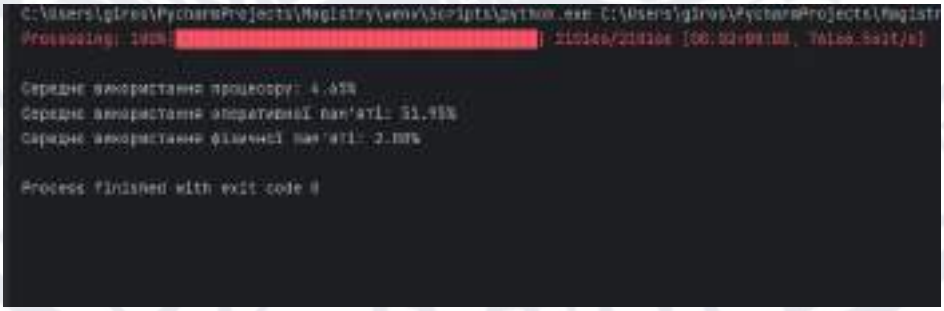


Рисунок 3.2.9 Результати бінарного пошуку на мові Python

Результати бінарного пошуку на мові Java рис 3.2.10

```
C:\Users\giros\.jdks\openjdk-23.0.1
Total Time Used: 0.36 seconds
Average CPU Usage: 0.00%
Average Memory Usage: 1.64%
```

Рисунок 3.2.10 Результати бінарного пошуку на мові Java

Результати бінарного пошуку на мові C# рис 3.2.11

```
Total Time Used: 23.146608 seconds
Average CPU Usage: 5.27%
Average Memory Usage: 62.15%
```

Рисунок 3.2.11 Результати бінарного пошуку на мові C#

Порівняльна діаграма швидкодії бінарного пошуку Рисунок 3.2.12

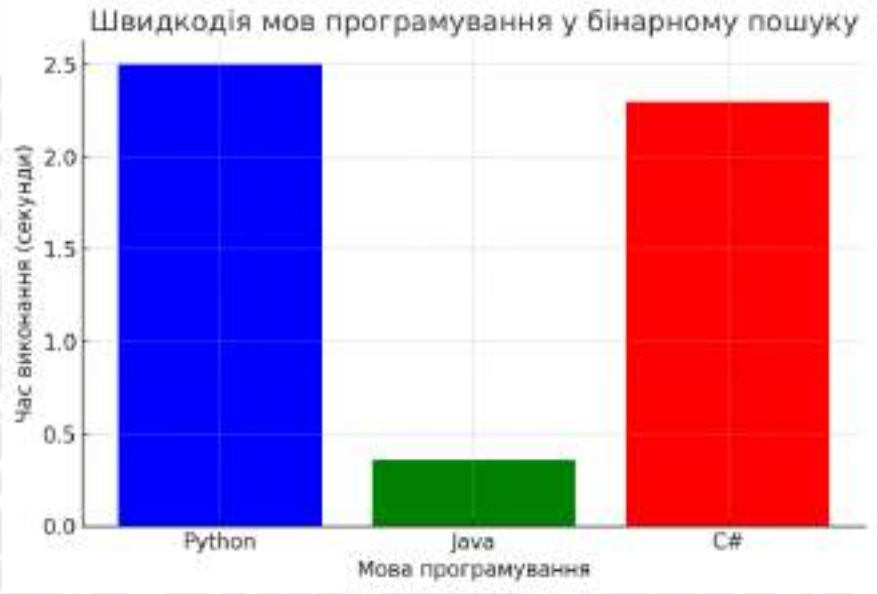


Рисунок 3.2.12 Порівняльна діаграма швидкодії бінарного пошуку

Порівняльна діаграма використання процесору під час бінарного пошуку Рисунок 3.2.13

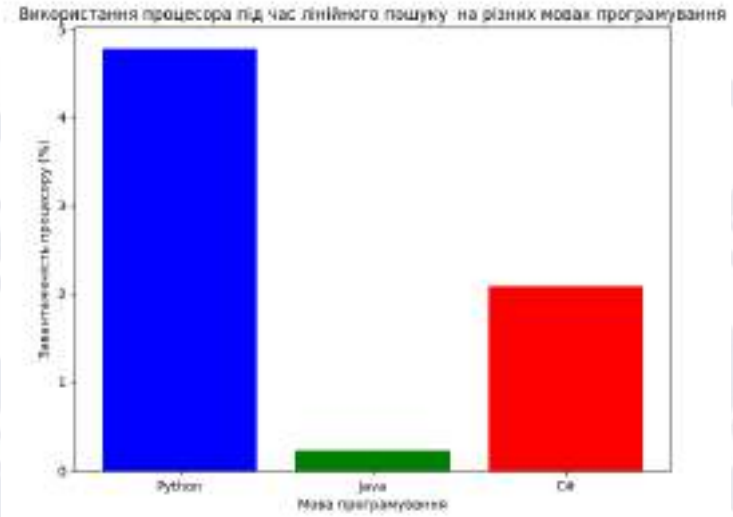


Рисунок 3.2.13 Порівняльна діаграма використання процесору під час бінарного пошуку

Порівняльна діаграма використання оперативної пам’яті під час обробки бінарним пошуком Рисунок 3.2.14

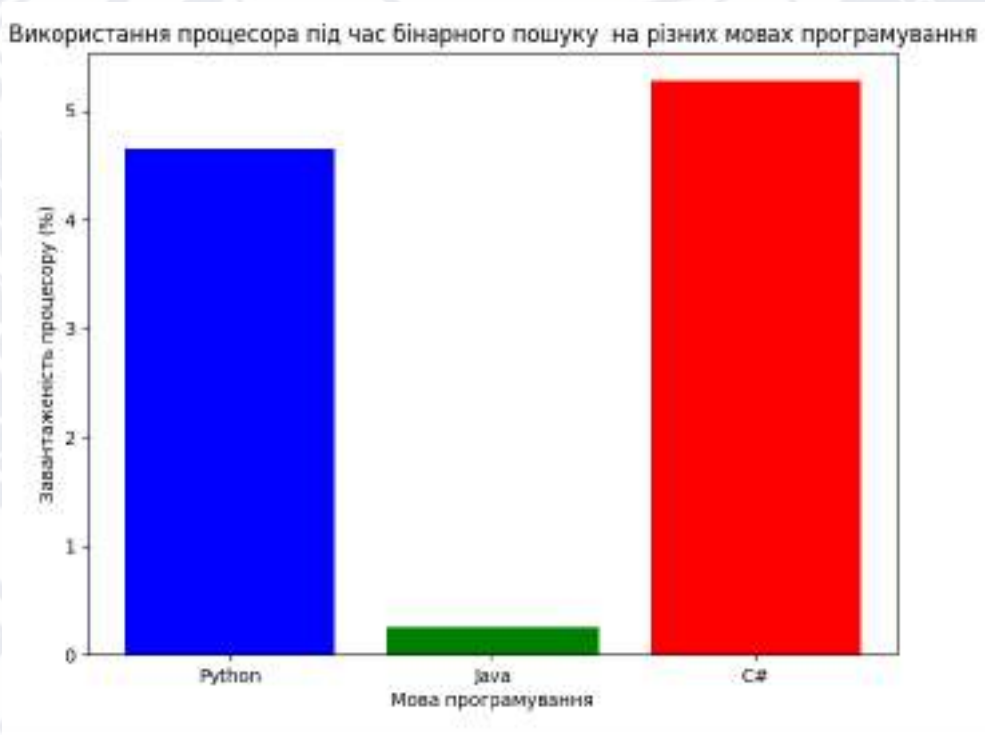


Рисунок 3.2.14 Порівняльна діаграма використання оперативної пам’яті під час обробки бінарним пошуком

Пошук в глибину

Результати алгоритму пошуку в глибину на мові Python рис 3.2.15, 3.2.16

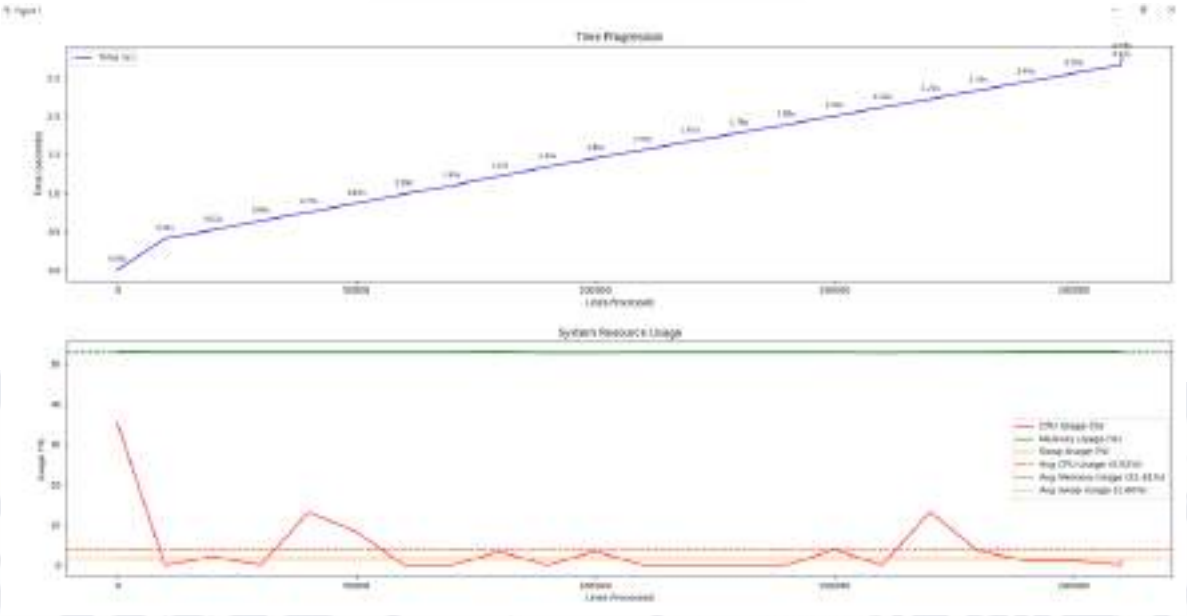


Рисунок 3.2.15 Результати алгоритму пошуку в глибину на мові Python

```
Average CPU Usage: 3.92%
Average Memory Usage: 52.81%
Average Swap Usage: 1.60%

Process finished with exit code 0
```

Рисунок 3.2.16 Результати алгоритму пошуку в глибину на мові Python

Результати алгоритму пошуку в глибину на мові Java рис 3.2.17

```
Total Time Used: 0.19 seconds
Average CPU Usage: 0.00%
Average Memory Usage: 1.62%

Process finished with exit code 0
```

Рисунок 3.2.17 Результати алгоритму пошуку в глибину на мові Java

Результати алгоритму пошуку в глибину на мові C# рис 3.2.18

```
Total Time Used: 2.4123961 seconds
Average CPU Usage: 2.57%
Average Memory Usage: 108.15 MB

Process finished with exit code 0.
```

Рисунок 3.2.18 Результати алгоритму пошуку в глибину на мові C#

Порівняльна діаграма швидкодії алгоритму пошуку в глибину Рисунок 3.2.19

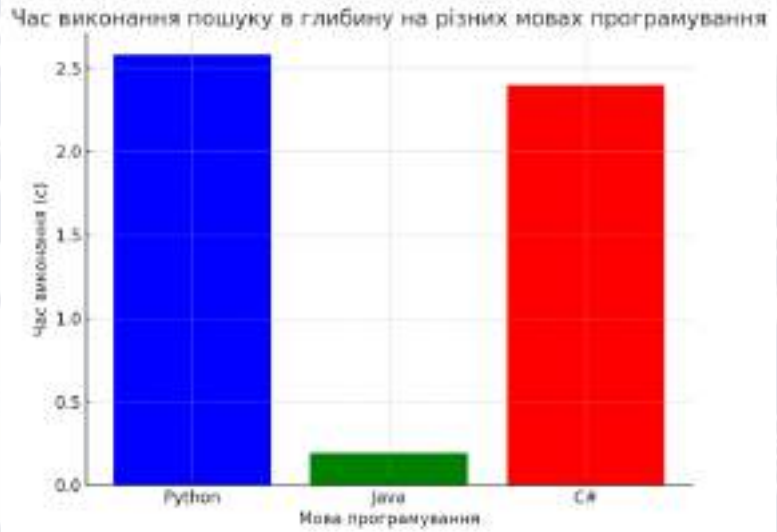


Рисунок 3.2.19 Порівняльна діаграма швидкодії алгоритму пошуку в глибину

Порівняльна діаграма використання процесору під час алгоритму пошуку в глибину Рисунок 3.2.20

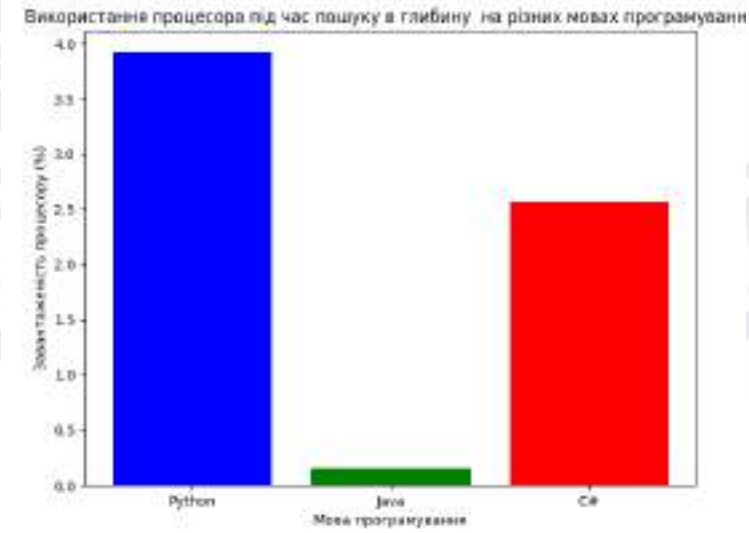


Рисунок 3.2.20 Порівняльна діаграма використання процесору під час алгоритму пошуку в глибину

Порівняльна діаграма використання оперативної пам’яті під час обробки алгоритмом пошуку в глибину Рисунок 3.2.21

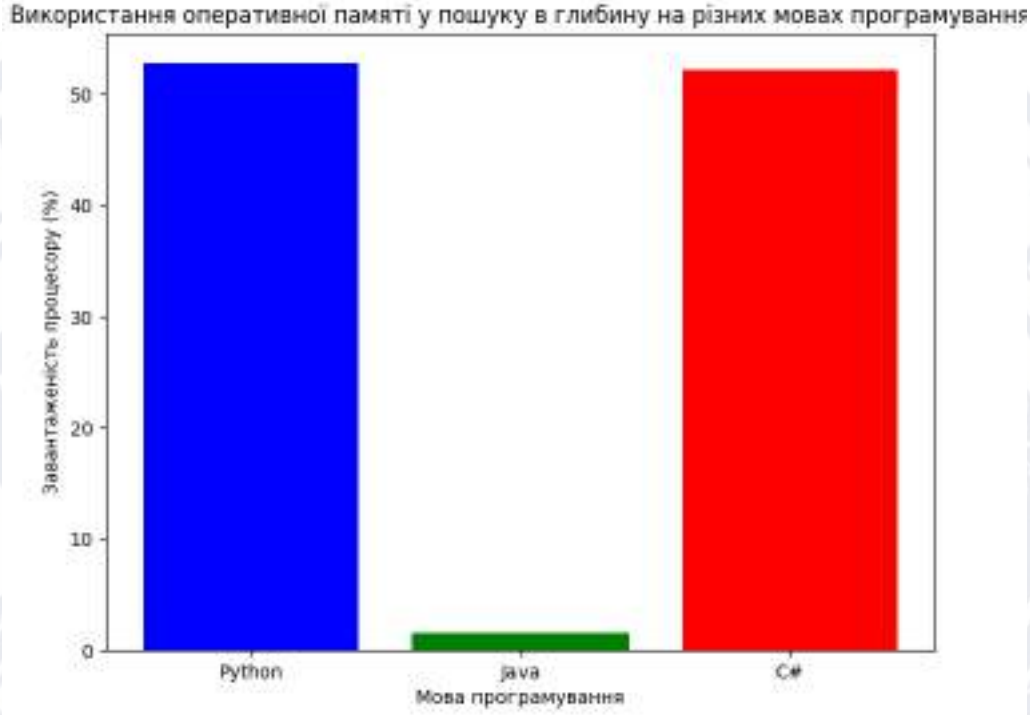


Рисунок 3.2.21 Порівняльна діаграма використання оперативної пам’яті під час обробки алгоритмом пошуку в глибину

Пошук в ширину

Результати алгоритму пошуку в ширину на мові Python рис 3.2.22, 3.2.23

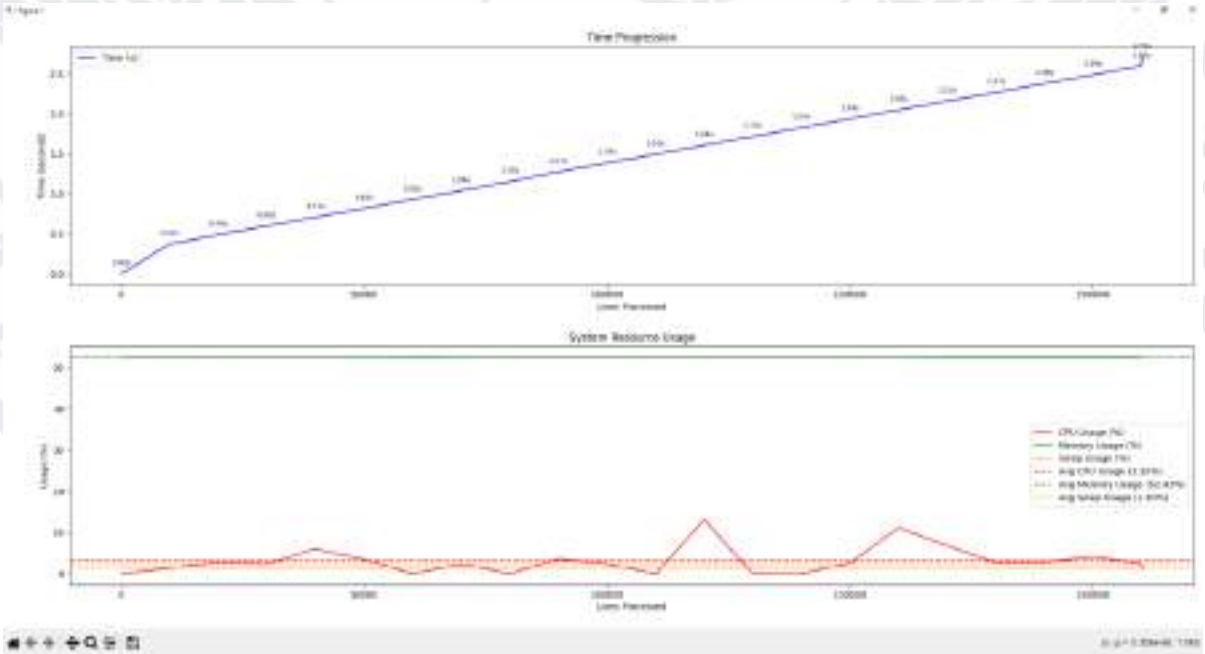


Рисунок 3.2.22 Результати алгоритму пошуку в ширину на мові Python

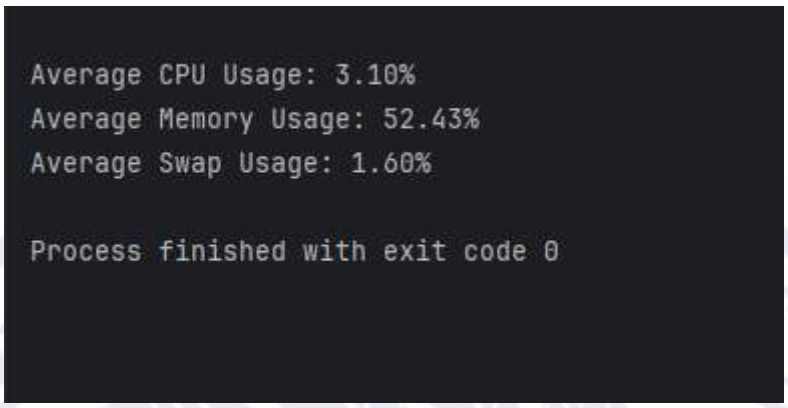


Рисунок 3.2.23 Результати алгоритму пошуку в ширину на мові Python

Результати алгоритму пошуку в ширину на мові Java рис 3.2.24

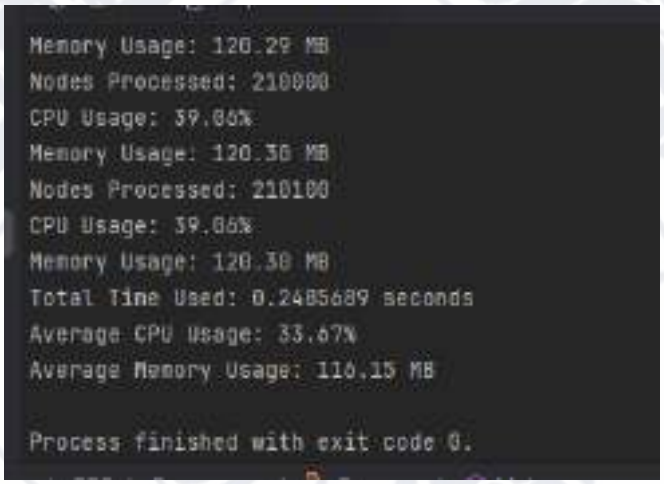


Рисунок 3.2.24 Результати алгоритму пошуку в ширину на мові Java

Результати алгоритму пошуку в ширину на мові C# рис 3.2.25

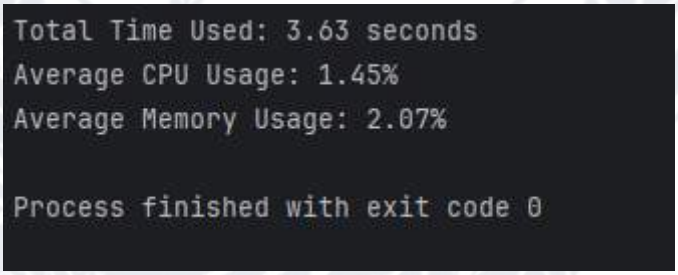


Рисунок 3.2.25 Результати алгоритму пошуку в ширину на мові C#

Порівняльна діаграма швидкодії алгоритму пошуку в ширину Рисунок 3.2.26

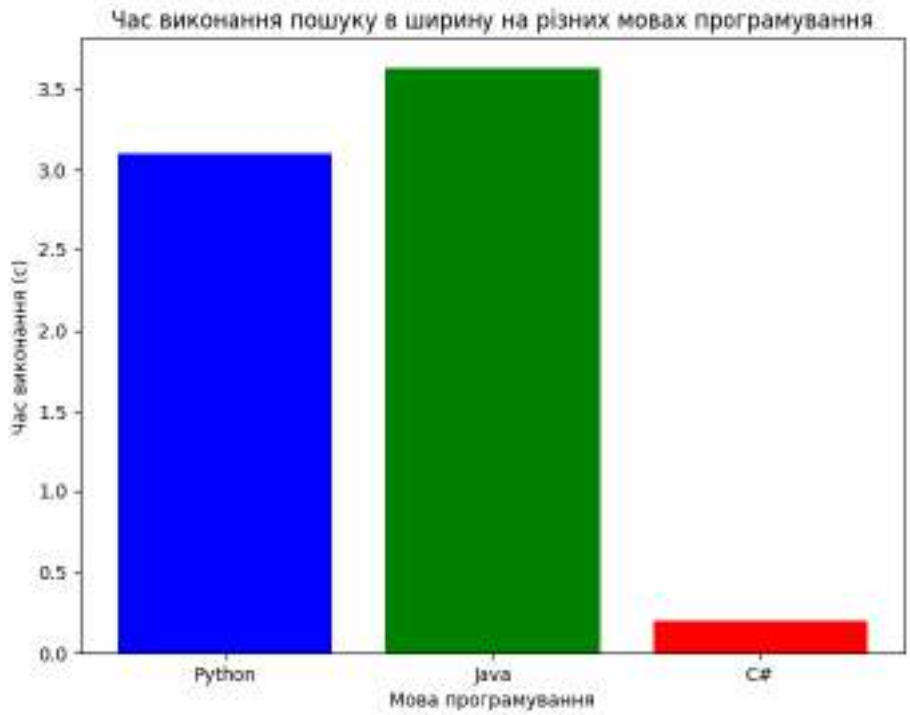


Рисунок 3.2.26 Порівняльна діаграма швидкодії алгоритму пошуку в ширину

Порівняльна діаграма використання процесору під час алгоритму пошуку в ширину Рисунок 3.2.27

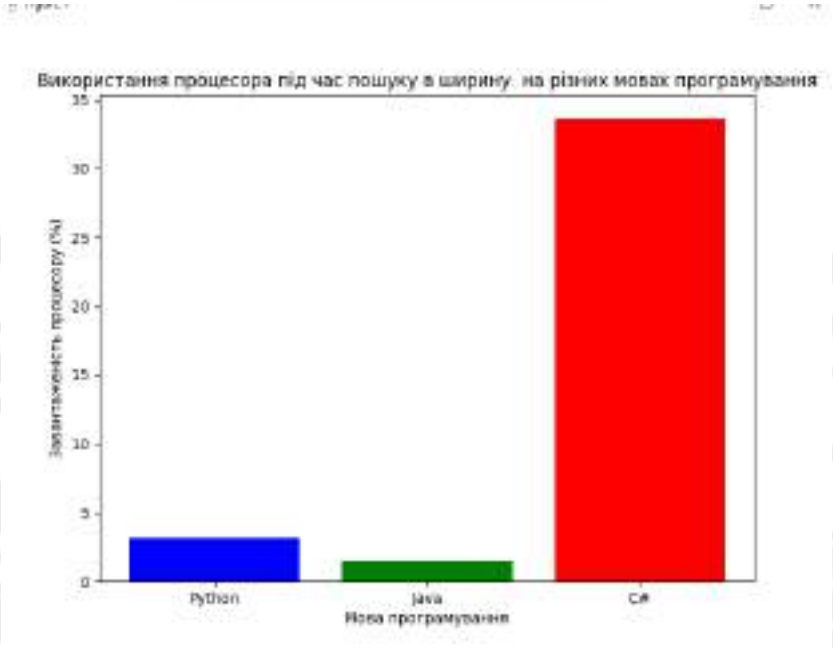


Рисунок 3.2.27 Порівняльна діаграма використання процесору під час алгоритму пошуку в ширину

Порівняльна діаграма використання оперативної пам’яті під час обробки алгоритмом пошуку в ширину Рисунок 3.2.28

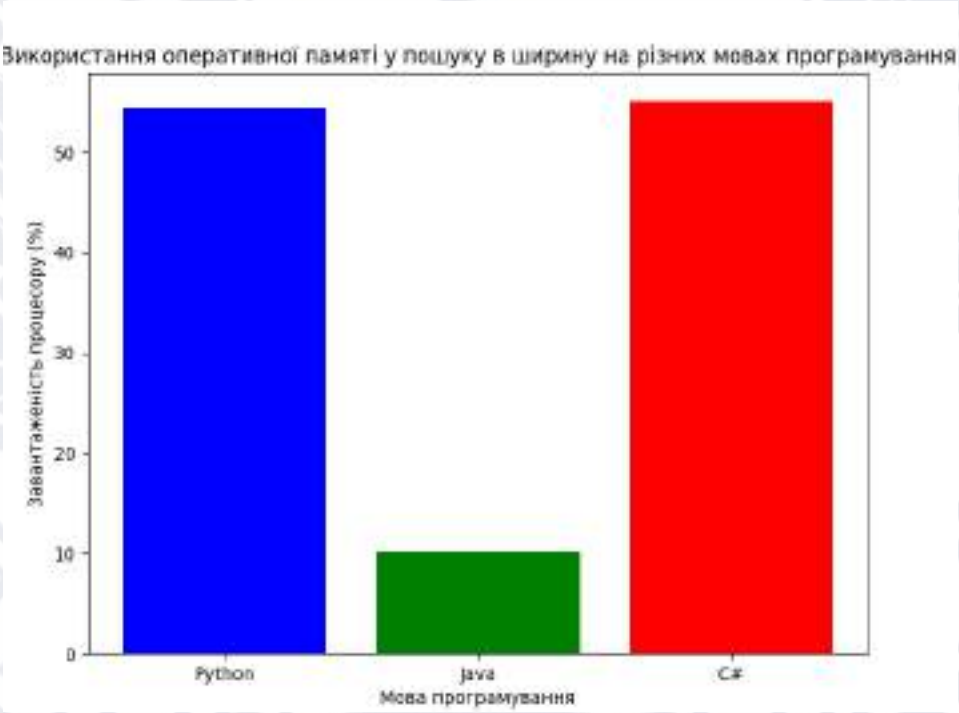


Рисунок 3.2.28 Порівняльна діаграма використання оперативної пам’яті під час обробки алгоритмом пошуку в ширину

Алгоритми сортування:

Результати алгоритму сортування бульбашкою на мові Python рис 3.2.29, 3.2.30

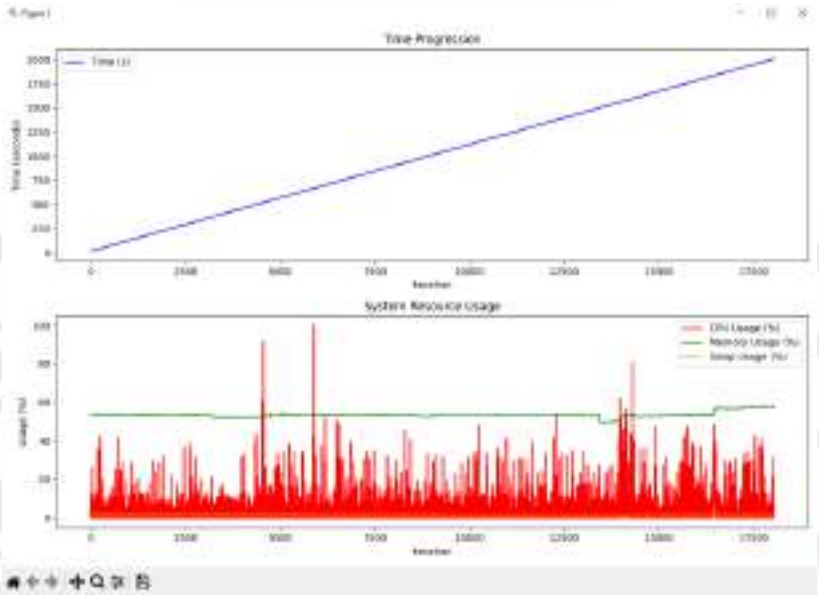


Рисунок 3.2.29 Результати алгоритмусортування бульбашкою на мові Python

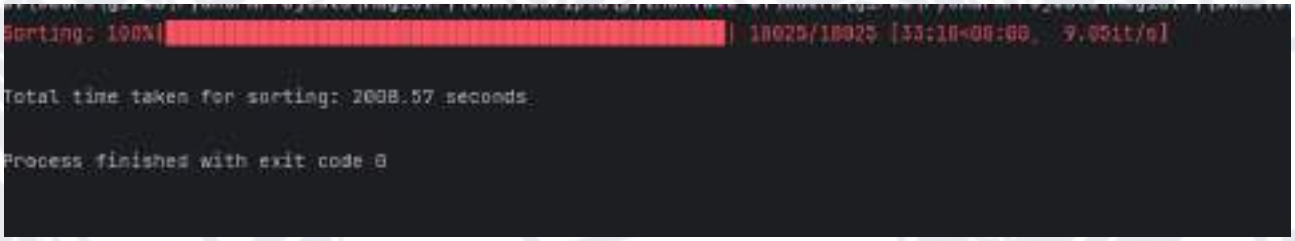


Рисунок 3.2.30 Результати алгоритму сортування бульбашкою на мові Python

Результати алгоритму сортування бульбашкою на мові Java рис 3.2.31

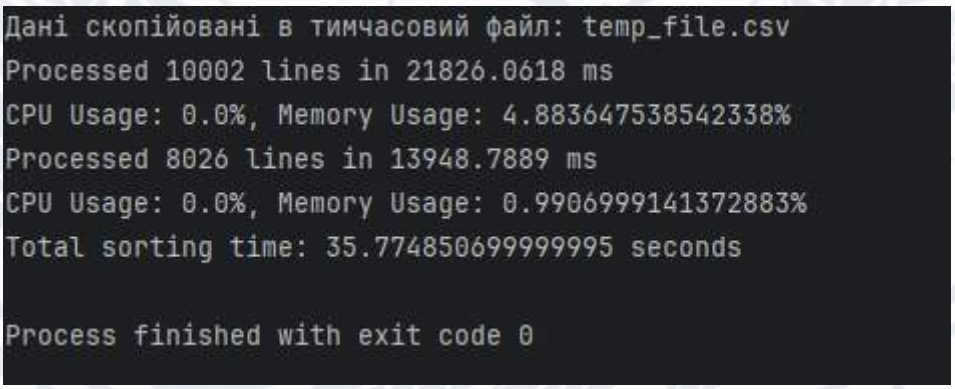


Рисунок 3.2.31 Результати алгоритму сортування бульбашкою на мові Java

Результати алгоритму сортування бульбашкою на мові C# рис 3.2.32

```
Data copied to temporary file: temp_file.csv
Processed 18025 lines in 81327.68 ms
CPU Usage: 0.55%, Memory Usage: 68.14%
Data sorted successfully.
Total sorting time: 81.3276808 seconds

Process finished with exit code 0.
```

Рисунок 3.2.32 Результати алгоритму сортування бульбашкою на мові C#

Порівняльна діаграма швидкодії алгоритму сортування бульбашкою Рисунок 3.2.33

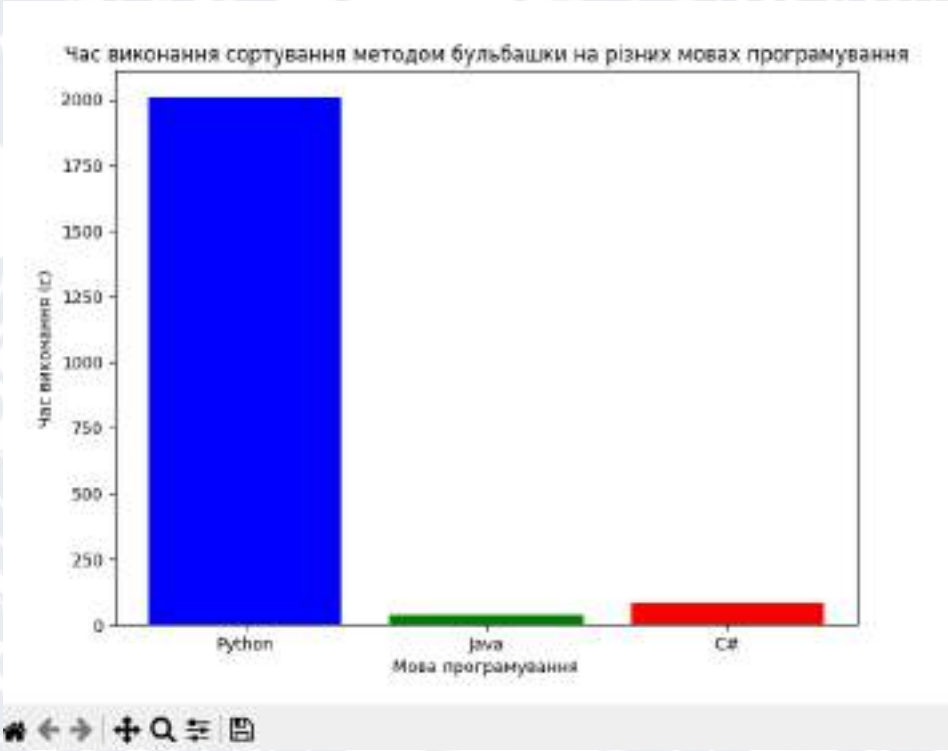


Рисунок 3.2.33 Порівняльна діаграма швидкодії алгоритму пошуку в ширину сортування бульбашкою

Порівняльна діаграма використання процесору під час алгоритму сортування бульбашкою Рисунок 3.2.34

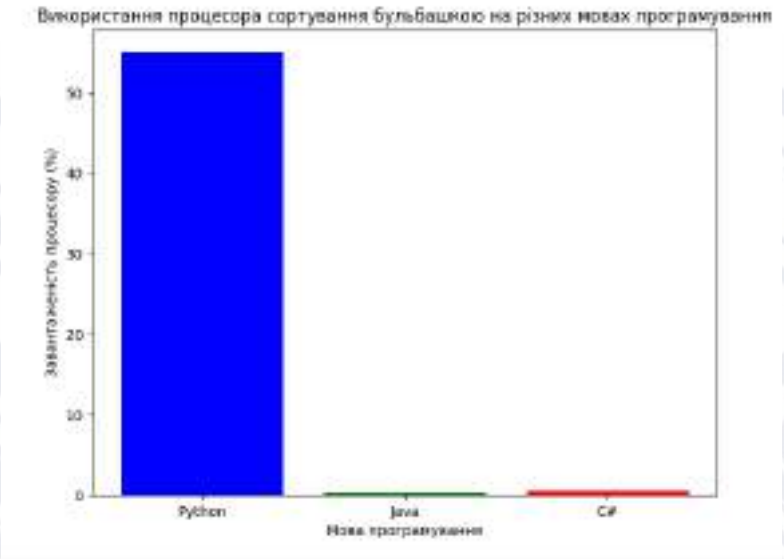


Рисунок 3.2.34 Порівняльна діаграма використання процесору під час алгоритму сортування бульбашкою

Порівняльна діаграма використання оперативної пам’яті під час обробки алгоритмом сортування бульбашкою Рисунок 3.2.35

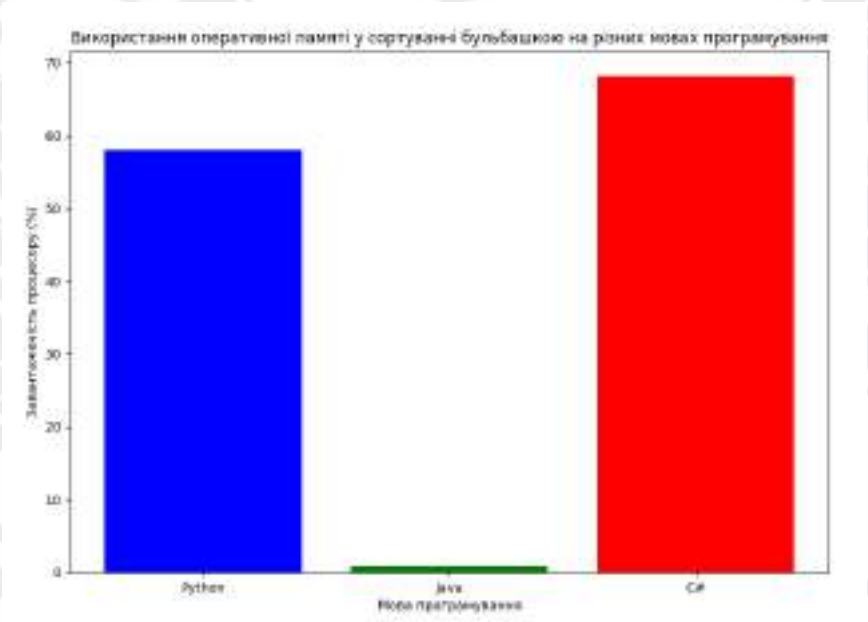


Рисунок 3.2.35 Порівняльна діаграма використання оперативної пам’яті під час обробки алгоритмом сортування бульбашкою

Швидке сортування

Результати алгоритму швидкого сортування на мові Python рис 3.2.36, 3.2.37

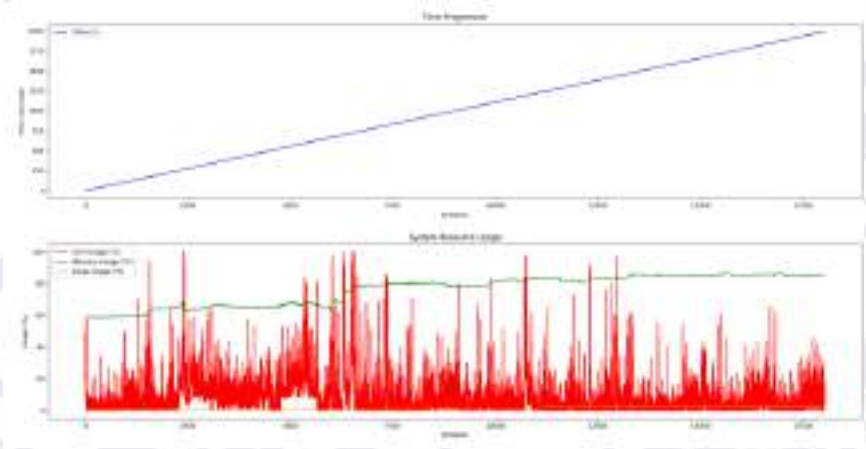


Рисунок 3.2.36 Результати алгоритмусортування бульбашкою на мові Python

Рисунок 3.2.38 Результати алгоритму швидкого сортування на мові Python

Результати алгоритму швидкого сортування на мові Java рис 3.2.39

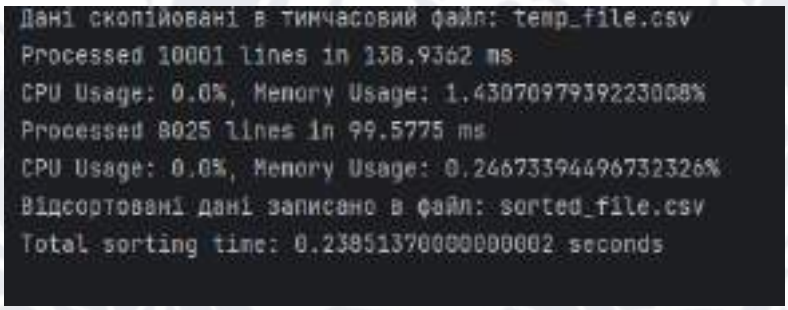


Рисунок 3.2.39 Результати алгоритму швидкого сортування на мові Java

Результати алгоритму швидкого сортування на мові C# рис 3.2.40

```
Data copied to temporary file: temp_file.csv
Processed 18025 lines in 142.07 ms
CPU Usage: 2.57%, Memory Usage: 69.15%
Data sorted successfully.
Total sorting time: 0.1420722 seconds

Process finished with exit code 0.
```

Рисунок 3.2.40 Результати алгоритму швидкого сортування на мові C#

Порівняльна діаграма швидкодії алгоритму швидкого сортування Рисунок 3.2.41

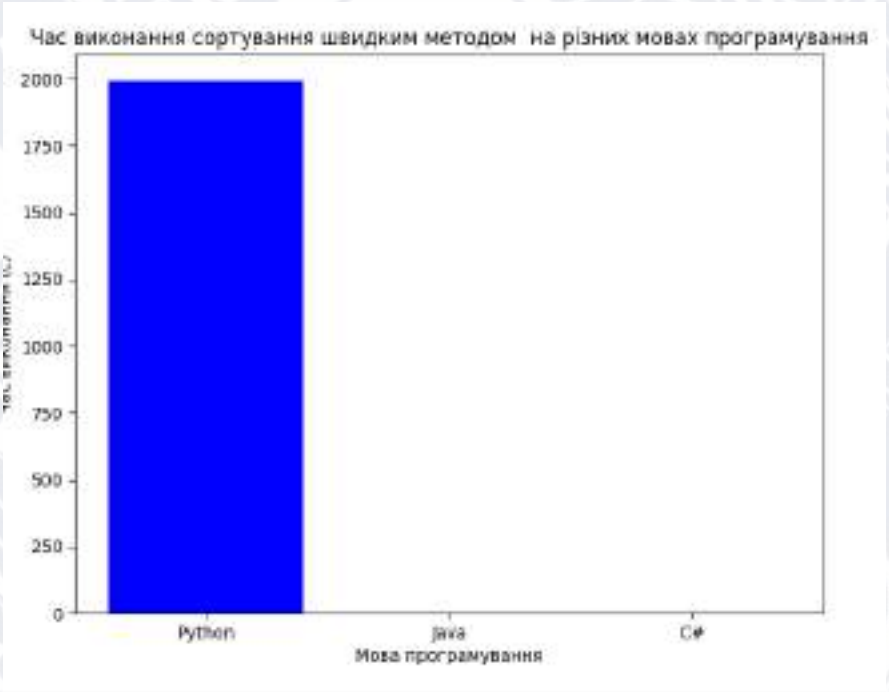


Рисунок 3.2.41 Порівняльна діаграма швидкодії алгоритму швидкого сортування

Порівняльна діаграма використання процесору під час алгоритму швидкого сортування Рисунок 3.2.42

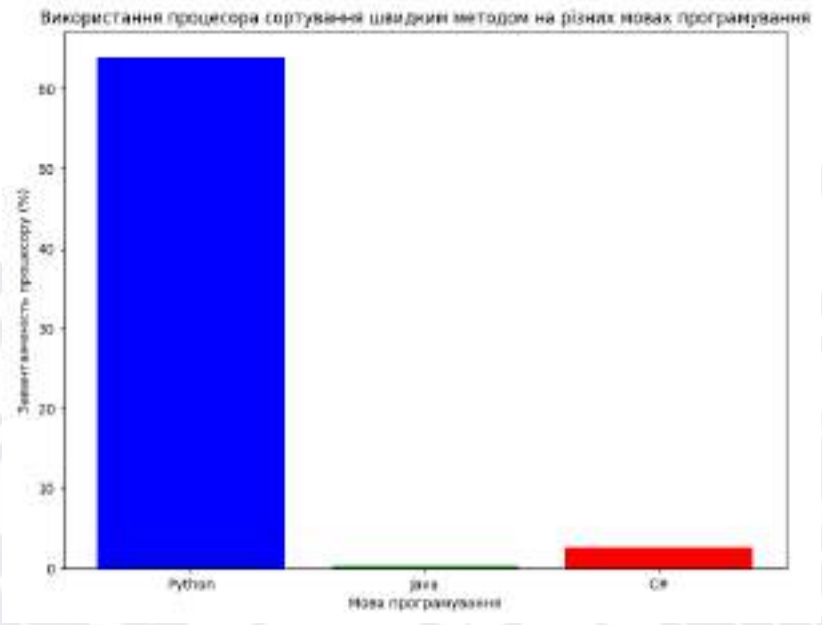


Рисунок 3.2.42 Порівняльна діаграма використання процесору під час алгоритму швидкого сортування

Порівняльна діаграма використання оперативної пам’яті під час обробки алгоритмом швидкого сортування Рисунок 3.2.43

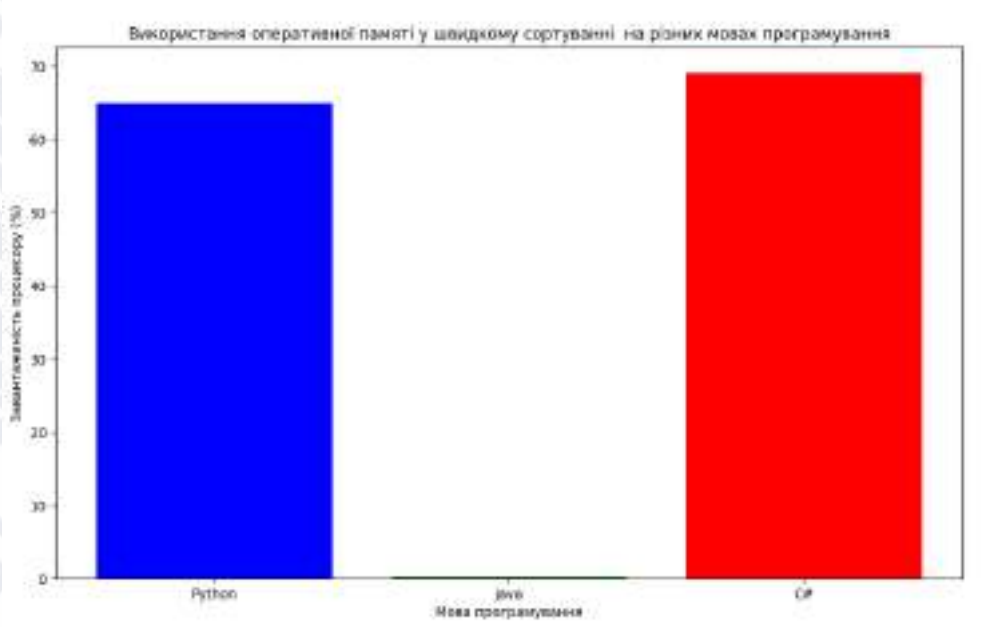


Рисунок 3.2.43 Порівняльна діаграма використання оперативної пам’яті під час обробки алгоритмом швидкого сортування

3.3 Висновок щодо переваг Java в обробці даних

Java часто демонструє вищу швидкість обробки даних порівняно з Python і C# завдяки наступним факторам:

1. Продуктивність завдяки JVM. Java працює на базі високопродуктивної Java Virtual Machine (JVM), яка застосовує сучасні оптимізації, зокрема Just-In-Time (JIT) компіляцію. Це дозволяє перетворювати байт-код у високоефективний машинний код безпосередньо під час виконання програми, що забезпечує швидке виконання часто використовуваних операцій.
2. Оптимізована робота з пам'яттю. Java використовує Garbage Collection для управління пам’яттю, що мінімізує ручне втручання і знижує ризик витоків пам’яті. Це дозволяє ефективно обробляти великі обсяги даних, не витрачаючи ресурси на помилки, властиві, наприклад, C#.
3. Статична типізація та компіляція. Статична типізація Java дозволяє виявляти помилки на етапі компіляції, що скорочує накладні витрати на час виконання. Це відрізняє Java від Python, де типи перевіряються динамічно, що уповільнює виконання задач.
4. Ефективність багатопотоковості. Java має вбудовану підтримку багатопотоковості з ефективними засобами синхронізації потоків. Це особливо важливо для паралельної обробки даних, яка часто використовується в великих і складних системах. У Python багатопоточність обмежена GIL (Global Interpreter Lock), а в C# підтримка потоків хоч і потужна, але працює дещо інакше через прив’язку до Windows-середовища.
5. Кросплатформова сумісність. Java розроблена як кросплатформова мова "write once, run anywhere". Це дає можливість запускати високопродуктивний код на будь-якій платформі без значних модифікацій, що знижує залежність від специфіки середовища.
6. Розвинуті бібліотеки для роботи з даними. Java пропонує численні високоефективні бібліотеки та фреймворки для обробки даних, такі як Apache Spark, Hadoop, і Flink. Ці інструменти написані на Java і оптимізовані для високопродуктивної роботи з великими обсягами даних.
7. Чому Python повільніший? Python інтерпретується, а не компілюється, що значно знижує його продуктивність у складних обчисленнях. Навіть із використанням бібліотек, таких як NumPy, частина роботи залежить від інтерпретатора Python. Python більше орієнтований на зручність розробки, ніж на максимальну швидкість.

8. Чому C# може поступатися Java? Хоча C# має продуктивну реалізацію на платформі .NET, її продуктивність може бути нижчою в сценаріях, де критичною є абсолютна незалежність від середовища. Більша

орієнтованість C# на Windows-середовище може створювати складнощі в налаштуванні багатоплатформових рішень.

ВИСНОВКИ

У ході дослідження ефективності кросплатформових мов програмування для задач обробки даних було виконано теоретичний та експериментальний аналіз. На основі отриманих результатів сформульовано такі висновки:

Кросплатформовість є важливим критерієм при виборі мови програмування для обробки даних. Python, Java та C# забезпечують високий рівень кросплатформової підтримки завдяки своїм екосистемам (JVM для Java, .NET Core для C#, віртуальне середовище Python).

Python є найзручнішою мовою для швидкої реалізації алгоритмів та аналізу даних. Його велика бібліотечна екосистема (наприклад, Pandas, NumPy, TensorFlow) робить Python найкращим вибором для наукових досліджень, аналізу великих даних та роботи з машинним навчанням. Проте, через інтерпретований характер, Python поступається Java та C# у продуктивності під час обробки великих обсягів даних.

Java продемонструвала високу продуктивність, особливо в багатопотокових задачах. Завдяки оптимізації JVM та розвиненій підтримці великих даних (Hadoop, Apache Spark), Java є доцільним вибором для завдань, що вимагають обробки великих обсягів даних або роботи з розподіленими системами. Проте її складніший синтаксис може уповільнити розробку.

C# добре підходить для задач, що вимагають роботи з структурованими даними, завдяки можливостям LINQ та інтеграції з SQL Server. Завдяки оптимізації на платформі .NET Core C# забезпечує високу продуктивність. Проте його поширеність обмежується переважним використанням в екосистемі Microsoft.

Продуктивність мов програмування залежить від специфіки задачі. Для швидкої розробки прототипів і досліджень найкраще підходить Python. Java є оптимальною для високонавантажених систем і багатопоточних обчислень. C# демонструє ефективність у бізнес-застосунках із високими вимогами до продуктивності роботи з базами даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. "Python for Data Analysis" автора Wes McKinney. (2017) 327 с.
2. "Java Concurrency in Practice" авторів Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. (2006) 282 с.
3. "C# 9.0 in a Nutshell" авторів Joseph Albahari, Ben Albahari. (2021) 128 с.
4. "Introduction to Algorithms" авторів Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. (2022) 915-970 с.
5. "Algorithms" автора Robert Sedgewick та Kevin Wayne. (2020) 625 с.
6. "The Algorithm Design Manual" автора Steven S. Skiena. (2020) 378 с.
7. "Algorithm Design" автора Jon Kleinberg та Éva Tardos. (2006) 138 с.
8. "Algorithms Unlocked" автора Thomas H. Cormen. (2013) 54 с.
9. "Algorithms, 4th Edition" by Robert Sedgewick and Kevin Wayne. (2011) 673 с.
10. "Data Structures and Algorithms in Python" by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. (2013) 256 с.
11. "Competitive Programming" by Steven Halim, Felix Halim, and Suhendry Effendy. (2020) 127 с.
12. "Data Structures and Algorithms in Java" by Robert Lafore. (2003) 568 с.
13. "Java: The Complete Reference" by Herbert Schildt. (2022)
14. "Java Performance: The Definitive Guide" by Scott Oaks. (2020) 135 с.
15. "Effective Java" by Joshua Bloch. (2008) 126 с.
16. "Algorithms in Java" (Parts 1-4) by Robert Sedgewick. (2002) 456 с.
17. "Python Cookbook" by David Beazley and Brian K. Jones. (2013) 325 с.
18. "Python Tricks: A Buffet of Awesome Python Features" by Dan Bader. (2023) 215 с.
19. "Mastering Python: Master the Art of Writing Beautiful and Powerful Python by Learning the Best Practices and Advanced Programming Concepts" by Rick van Hattem. (2016) 37 с.
20. "Python Data Science Handbook" by Jake VanderPlas. (2023) 216 с.

21."C# 9.0 and .NET 5 – Modern Cross-Platform Development" by Mark J. Price. (2022) 87 c.
22."The C# Player's Guide" by R.B. Whitaker. (2022) 168 c.
23."C# Programming Yellow Book" by Rob Miles. (2023) 763 c.
24."Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming" by Stephen Cleary. (2014) 168 c.
25."Design Patterns in C#" by Dmitri Nesteruk (1995) 178 c.
26."Learning SQL: Generate, Modify, and Query Data" by Alan Beaulieu. (2020) 121 c.
27."SQL Queries for Mere Mortals: A Hands-On Guide to Data Manipulation in SQL" by John L. Viescas and Michael J. Hernandez. (2014) 165 c.
28."SQL Server Execution Plans" by Grant Fritchey (2018) 243 c.
29."Comparative Study on Programming Languages for Data Analysis" by Seungwoo Kang, Keonsoo Ha, and Dongil Shin. https://scholar.google.co.kr/citations?user=klsjaRwAAAAJ&hl=en
30."Efficient Data Processing for Large-scale Scientific Data Analysis" by Dorian Arnold, Kshitij Mehta, David Boehme, Daniel Quinlan, and Karthik Murthy. https://commencement.umich.edu/wp-content/uploads/2022/05/SC-Program-2015-FINAL.pdf
31."A Comparative Study on Data Analysis with Python" by Anshul Mittal and Gaurav Aggarwal. https://www.researchgate.net/publication/340180778_DDBJ_Data_Analysis_Challenge_a_machine_learning_competition_to_predict_Arabidopsis_chromatin_feature_annotations_from_DNA_sequences
32.Coursera – "Algorithms Specialization" by Stanford University (by Tim Roughgarden) https://www.coursera.org/learn/algorithms-graphs-data-structures
33."The Algorithm Design Manual" by Steven S. Skiena (Java code examples available online)

https://mimoza.marmara.edu.tr/~msakalli/cse706_12/SkienaTheAlgorithmDesignManual.pdf

34. "Python Crash Course: A Hands-On, Project-Based Introduction to Programming" by Eric Matthes https://www.coursesidekick.com/computer-science/2722887
35. "SQL for Data Analysis: A Beginner's Guide" by Upendra Kumar"Data Warehousing in the Age of Big Data" by Krish Krishnan. http://ppdi.stmik-banjarbaru.ac.id/index.php?dir=8.%20Data%20Warehouse/&file=2013%20Data%20Warehousing%20in%20the%20Age%20of%20Big%20Data.pdf
36. "SQL Server 2019 Administration Inside Out" by Randolph West, Brian McCurdy, and Alesha M. K. McGee https://netinfo.click/books/prog/SQL%20Server%202022%20Administration%20Inside%20Out.pdf
37. Codecademy Python Course - URL https://try.codecademy.com/learn-python
38. Codecademy Java Course – URL https://www.codecademy.com/learn/learn-java
39. Codecademy C# Course – URL https://www.codecademy.com/learn/learn-C#

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

_____

_____

_____

Повністю вказується ПІБ та статус (освітня (освітньо-наукова) програма – для здобувачів вищої освіти, назва кваліфікаційної роботи)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)