МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ ВІЗУАЛІЗАЦІЇ СТРУКТУР ДАНИХ ТА АЛГОРИТМІВ»**

Виконала: студентка 4 курсу, групи 6.1222
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)
освітньої програми Комп'ютерні науки
(назва освітньої програми)
А.О. Гостіщева
(ініціали та прізвище)
Керівник доцент кафедри комп'ютерних наук, доцент, к.т.н Матвіїшина Н.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент доцент кафедри програмної інженерії, к.ф.-м.н., Кривохата А. Г.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра комп'ютерних наук

Рівень вищої освіти бакалавр

Спеціальність 122 Комп'ютерні науки
(шифр і назва)

Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри комп'ютерних наук, к.пед.н., доцент

_____ Пшенична О.С.
(підпис)

" ____ " _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Гостіщевій Анастасії Олексіївні
(прізвище, ім'я та по-батькові)

1. Тема роботи Розробка вебзастосунку для візуалізації структур даних та алгоритмів

керівник роботи Матвіїшина Надія Вікторівна, к.т.н., доцент
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 04.06.2026

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка та реалізація вебзастосунку для візуалізації структур даних та алгоритмів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 05.12.2025

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	15.12.2025	
2.	Збір вихідних даних.	15.01.2026	
3.	Обробка методичних та теоретичних	15.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	15.03.2026	
5.	Розробка третього розділу.	15.05.2026	
6.	Оформлення та нормоконтроль	04.06.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	18.06.2026	

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

Керівник роботи _____ Н.В.Матвіїшина
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О.Г.Спиця
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота на тему: «Розробка вебзастосунку для візуалізації структур даних та алгоритмів»: 153 с., 34 рис., 19 джерел, 3 додатки.

АЛГОРИТМ, ВЕБЗАСТОСУНОК, ВІЗУАЛІЗАЦІЯ, ДИНАМІЧНЕ ПРОГРАМУВАННЯ, ІНТЕРАКТИВНИЙ ІНТЕРФЕЙС, СОРТУВАННЯ, СТРУКТУРА ДАНИХ, ТЕОРІЯ ГРАФІВ, ПОШУК, JAVASCRIPT.

Об’єктом дослідження є процеси візуального подання структур даних та алгоритмів у вебсередовищі. Предметом дослідження є методи, програмні засоби та принципи розробки вебзастосунку для інтерактивної демонстрації роботи алгоритмів сортування, пошуку, обходу графів і методів динамічного програмування.

Метою роботи є розробка вебзастосунку, який забезпечує наочну, інтерактивну та зручну візуалізацію структур даних і алгоритмів, дає змогу відстежувати послідовність виконання операцій, аналізувати проміжні стани та оцінювати базові характеристики роботи алгоритмів. Для досягнення мети поставленні такі задачі: проаналізувати предметну область, дослідити особливості алгоритмів і способів їх візуального подання, обґрунтувати вибір вебтехнологій, спроектувати структуру застосунку, реалізувати програмні модулі візуалізації та перевірити їх працездатність.

У роботі використано методи аналізу й узагальнення науково-технічних джерел, методи структурного та модульного проєктування ПЗ, методи алгоритмізації, а також методи порівняльного аналізу для оцінювання особливостей реалізованих алгоритмів. Практична частина виконана із застосуванням сучасних вебтехнологій HTML, CSS та JavaScript.

У результаті виконання кваліфікаційної роботи розроблено вебзастосунок для візуалізації алгоритмів сортування, пошуку, обходу графів і задач динамічного програмування. Реалізовано інтерактивний інтерфейс користувача, механізм покрокового відображення виконання алгоритмів, засоби керування

швидкістю анімації, генерацію вхідних даних та виведення статистичних показників. Новизна одержаних результатів полягає в поєднанні навчальної наочності, інтерактивності та програмної модульності в одному вебрішенні, що може бути адаптоване для подальшого розширення.

Результати роботи можуть бути використані в навчальному процесі під час вивчення дисциплін, пов’язаних з алгоритмами, структурами даних, програмуванням і веброзробкою. Розроблений програмний продукт є прикладом реалізації засобів візуалізації алгоритмів та структур даних і може застосовуватися для демонстрації принципів їх функціонування, а також для дослідження особливостей роботи відповідних алгоритмів. Отримані результати можуть бути корисними для ознайомлення з підходами до розробки інтерактивних вебзастосунків навчального призначення. Подальший розвиток роботи доцільно спрямувати на розширення набору алгоритмів, додавання нових структур даних, удосконалення інтерфейсу користувача та розширення функціональних можливостей системи.

SUMMARY

Bachelor's Qualifying Thesis: «Development of a web application for the visualization of data structures and algorithms»: 153 pages, 34 figures, 19 sources, 3 supplements.

ALGORITHM, WEB APPLICATION, VISUALIZATION, DYNAMIC PROGRAMMING, INTERACTIVE INTERFACE, SORTING, DATA STRUCTURE, GRAPH THEORY, SEARCH, JAVASCRIPT.

The object of the study is the processes of visual representation of data structures and algorithms in the web environment. The subject of the study is the methods, software tools and principles of developing a web application for interactive demonstration of the operation of sorting, search, graph traversal algorithms and dynamic programming methods.

The purpose of the work is to develop a web application that provides a clear, interactive and convenient visualization of data structures and algorithms, allows you to track the sequence of operations, analyze intermediate states and evaluate the basic characteristics of the algorithms. To achieve this goal, the following tasks were solved: the subject area was analyzed, the features of algorithms and methods of their visual representation were investigated, the choice of web technologies was justified, the structure of the application was designed, visualization software modules were implemented and their performance was tested.

The work used methods of analysis and generalization of scientific and technical sources, methods of structural and modular software design, algorithmization methods, as well as comparative analysis methods to evaluate the features of the implemented algorithms. The practical part was performed using modern web technologies HTML, CSS and JavaScript.

As a result of the course work, a web application was developed for the visualization of sorting algorithms, search, graph traversal and dynamic programming problems. An interactive user interface, a mechanism for step-by-step display of

algorithm execution, animation speed control tools, input data generation and statistical output have been implemented. The novelty of the results obtained lies in the combination of educational clarity, interactivity and software modularity in one web solution, which can be adapted for further expansion.

The results of the work can be used in the educational process when studying disciplines related to algorithms, data structures, programming and web development. The practical significance of the work lies in the creation of a software product that contributes to a better understanding of the principles of algorithm functioning and can be used as a demonstration or training tool. Further development of the work should be directed towards expanding the set of algorithms, adding new data structures and improving analytics.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 6
Перелік скорочень та умовних познак 9
Вступ 10
1 Теоретичні основи візуалізації структур даних та алгоритмів 12
1.1 Поняття алгоритмів та структур даних 12
1.2 Основні типи алгоритмів та їх класифікація 13
1.3 Методи візуалізації алгоритмів у програмному забезпеченні 15
1.4 Аналіз існуючих програмних рішень та вебсервісів 17
2 Проєктування вебзастосунку для візуалізації алгоритмів 22
2.2 Вибір технологій розробки вебзастосунку 23
2.3 Архітектура та структура вебзастосунку 25
2.4 Моделювання взаємодії компонентів системи 27
3 Розробка та реалізація вебзастосунку 31
3.1 Реалізація алгоритмів сортування 31
3.2 Реалізація алгоритмів пошуку 33
3.3 Реалізація алгоритмів теорії графів 35
3.4 Реалізація алгоритмів динамічного програмування 36
3.5 Реалізація розділу структур даних 38
3.6 Розробка модуля візуалізації та інтерфейсу користувача 42
3.7 Тестування та аналіз результатів роботи застосунку 45
Висновки 59
Перелік посилань 62
Додаток А Вихідний код HTML-сторінок 64
Додаток Б Основні алгоритми 84
Додаток В Програмні модулі інтерфейсу та візуалізації 92

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

ІТ – інформаційні технології;

ПЗ – програмне забезпечення;

ІС – інформаційна система;

HTML – HyperText Markup Language (мова розмітки гіпертексту);

CSS – Cascading Style Sheets (каскадні таблиці стилів);

ООП – об’єктно-орієнтоване програмування;

SVG – Scalable Vector Graphics (масштабована векторна графіка);

UML – Unified Modeling Language (уніфікована мова моделювання);

DFS – Depth-First Search (пошук у глибину);

BFS – Breadth-First Search (пошук у ширину);

LCS – Longest Common Subsequence (найдовша спільна підпослідовність).

ВСТУП

У сучасних умовах стрімкого розвитку ІТ особливого значення набуває ефективне засвоєння фундаментальних понять інформатики, зокрема структур даних та алгоритмів. Саме алгоритми є основою будь-якого ПЗ, оскільки визначають послідовність дій для розв’язання конкретних задач, тоді як структури даних забезпечують оптимальне зберігання та обробку інформації. Проте складність абстрактних алгоритмічних процесів значно ускладнює їх розуміння, особливо на початкових етапах навчання. У зв’язку з цим виникає потреба у створенні інструментів, які дозволяють наочно відобразити роботу алгоритмів у динаміці, що зумовлює актуальність теми даної кваліфікаційної роботи.

Аналіз сучасних програмних рішень показує, що існує значна кількість систем для візуалізації алгоритмів, однак багато з них або мають обмежений функціонал, або не забезпечують достатнього рівня інтерактивності та гнучкості. Частина таких систем орієнтована на вузьке коло задач або складна у використанні для початківців. У зв’язку з цим доцільним є розроблення вебзастосунку, який поєднує простоту використання, наочність та широкий набір алгоритмів, що дозволить підвищити ефективність навчання та аналізу алгоритмічних процесів.

Метою даної роботи є розробка вебзастосунку для візуалізації структур даних та алгоритмів, який забезпечує інтерактивне відображення процесу виконання алгоритмів, дозволяє користувачеві аналізувати їхню поведінку та оцінювати основні характеристики. Досягнення поставленої мети передбачає вирішення низки взаємопов’язаних завдань, зокрема аналіз предметної області та існуючих аналогів, дослідження методів візуалізації алгоритмів, обґрунтування вибору технологій розробки, проєктування структури вебзастосунку, реалізацію алгоритмів різних типів, та створення зручного користувацького інтерфейсу та проведення тестування розробленого

програмного продукту.

Об’єктом дослідження є процеси візуалізації алгоритмів та структур даних у програмних системах.

Предметом дослідження є методи, засоби та технології розробки вебзастосунків для інтерактивного відображення алгоритмічних процесів.

У процесі виконання роботи використано методи аналізу та узагальнення науково-технічної літератури, методи алгоритмізації, структурного та ООП, та методи проєктування ІС. Практична реалізація здійснювалась із використанням сучасних вебтехнологій, зокрема HTML, CSS та JavaScript.

Інтерактивний вебзастосунок поєднує візуалізацію різних типів алгоритмів у єдиному середовищі, забезпечує покрокове відображення процесів їх виконання та надає можливість гнучкого налаштування параметрів візуалізації. Запропонований підхід дозволяє підвищити наочність і доступність складних алгоритмічних процесів.

Практичне значення отриманих результатів полягає у створенні програмного засобу, який може бути використаний у навчальному процесі закладів освіти, під час самостійного вивчення алгоритмів, та у демонстраційних цілях. Розроблений вебзастосунок може слугувати основою для подальшого розширення функціоналу, зокрема додавання нових алгоритмів, удосконалення візуалізації та інтеграції з іншими навчальними платформами.

1 ТЕОРЕТИЧНІ ОСНОВИ ВІЗУАЛІЗАЦІЇ СТРУКТУР ДАНИХ ТА АЛГОРИТМІВ

1.1 Поняття алгоритмів та структур даних

У сучасній інформатиці алгоритми та структури даних є фундаментальними поняттями, що лежать в основі розробки програмного забезпечення та ІС. Алгоритм визначається як скінченна послідовність чітко сформульованих дій, спрямованих на розв’язання певної задачі або досягнення конкретного результату. Основними властивостями алгоритму є визначеність, дискретність, результативність, масовість та скінченність [1]. Визначеність означає однозначність виконання кожної інструкції, дискретність – поділ процесу на окремі кроки, результативність – отримання результату за скінченну кількість операцій, масовість – можливість застосування до широкого класу задач, а скінченність – завершення алгоритму за обмежену кількість кроків.

Алгоритми відіграють важливу роль у процесі обробки даних, оскільки саме вони визначають логіку виконання операцій над інформацією. Залежно від характеру задачі та способу її розв’язання алгоритми поділяються на різні типи, зокрема алгоритми сортування, пошуку, обробки графів, а також алгоритми динамічного програмування [2]. Кожен із цих типів має свої особливості реалізації та застосування, що безпосередньо впливає на ефективність роботи програмних систем.

Структура даних – це сукупність методів і засобів для впорядкування даних таким чином, щоб забезпечити їх ефективну обробку. Вибір оптимальної структури даних є важливим етапом розробки алгоритмів, оскільки від цього залежить швидкодія, використання пам’яті та загальна продуктивність програмного продукту.

Існує велика кількість структур даних, серед яких виділяють лінійні та нелінійні. До лінійних структур належать масиви, списки, стеки та черги, які характеризуються послідовним розташуванням елементів. Нелінійні структури

включають дерева та графи, що дозволяють представляти більш складні зв’язки між даними. Кожна структура має свої переваги та обмеження, що визначає доцільність її використання в конкретних умовах.

Ефективність алгоритмів тісно пов’язана з вибором структури даних, адже одна й та сама задача може мати різні реалізації залежно від способу організації даних. Наприклад, операції пошуку або вставки можуть виконуватися з різною швидкістю залежно від того, чи використовується масив, зв’язаний список або дерево. У такому разі при розробці програмних рішень важливим є комплексний підхід, який враховує як алгоритмічні аспекти, так і особливості структур даних.

У вебзастосунках, що призначені для візуалізації алгоритмів, важливу роль відіграє наочне відображення взаємозв’язку між алгоритмом і структурою даних. Візуалізація дозволяє користувачу спостерігати зміну стану даних у процесі виконання алгоритму, що значно полегшує розуміння складних процесів. Зокрема, при візуалізації алгоритмів сортування можна відстежувати переміщення елементів масиву, а при роботі з графами – процес обходу вершин і ребер.

1.2 Основні типи алгоритмів та їх класифікація

Алгоритми, як універсальний інструмент обробки даних, характеризуються значною різноманітністю, що зумовлює необхідність їх систематизації та класифікації за різними ознаками. Класифікація алгоритмів дозволяє структурувати знання про способи розв’язання задач, визначити їх особливості та обрати найбільш ефективний підхід для конкретних умов. У сучасній інформатиці алгоритми класифікують за призначенням, способом реалізації, структурою керування, складністю та іншими критеріями.

Однією з найпоширеніших є класифікація алгоритмів за їх функціональним призначенням. Згідно з цим підходом виділяють алгоритми сортування, пошуку, обробки графів, та алгоритми динамічного програмування. Алгоритми сортування призначені для впорядкування елементів у певній

послідовності та включають такі відомі реалізації, як сортування бульбашкою, вставками, злиттям, швидке сортування та сортування купою [3]. Алгоритми пошуку спрямовані на знаходження елементів у структурах даних і можуть бути як простими, наприклад лінійний пошук, так і більш ефективними, такими як бінарний пошук, що застосовується для відсортованих масивів [4]. Алгоритми теорії графів, зокрема пошук у ширину та пошук у глибину, використовуються для аналізу зв'язків між об'єктами, тоді як алгоритм Дейкстри дозволяє знаходити найкоротші шляхи в зважених графах [5]. Окрему групу становлять алгоритми динамічного програмування, які використовуються для розв'язання задач із перекривними підзадачами та оптимальною підструктурою, наприклад обчислення чисел Фібоначчі або пошук найдовшої спільної підпослідовності [6].

Іншою важливою ознакою класифікації є спосіб побудови алгоритму. За цим критерієм виділяють ітеративні та рекурсивні алгоритми. Ітеративні алгоритми реалізуються за допомогою циклів і передбачають повторення певних дій до досягнення заданої умови. Рекурсивні алгоритми, у свою чергу, базуються на принципі самовиклику функції, що дозволяє природно описувати задачі, які мають ієрархічну або розгалужену структуру. Кожен із цих підходів має свої переваги та обмеження, що впливає на вибір способу реалізації [7].

За структурою керування алгоритми поділяються на лінійні, розгалужені та циклічні. Лінійні алгоритми виконуються послідовно без відхилень, розгалужені передбачають вибір одного з кількох варіантів виконання залежно від умов, а циклічні містять повторювані фрагменти коду. У реальних програмних системах зазвичай використовуються комбіновані алгоритми, що поєднують усі зазначені типи структур керування.

Оцінка складності алгоритмів – важливий критерій класифікації, який дозволяє визначити ефективність їх виконання. Часова складність характеризує залежність кількості операцій від розміру вхідних даних, тоді як просторова складність відображає обсяг використовуваної пам'яті. Для формального опису складності використовують нотацію «велике O», яка дозволяє оцінити порядок зростання витрат ресурсів. Наприклад, алгоритми сортування можуть мати квадратичну складність $O(n^2)$, як у випадку сортування бульбашкою, або

логарифмічно-лінійну O(n log n), як у випадку швидкого сортування чи сортування злиттям [8].

Алгоритми, що базуються на певних підходах до розв’язання задач, таких як «розділяй і володарюй», жадібні алгоритми та динамічне програмування. Підхід «розділяй і володарюй» передбачає розбиття задачі на менші підзадачі з подальшим об’єднанням результатів, що реалізується, зокрема, у швидкому сортуванні. Жадібні алгоритми приймають локально оптимальні рішення на кожному кроці, що може приводити до глобально оптимального результату, тоді як динамічне програмування використовує збереження проміжних результатів для підвищення ефективності.

У контексті розробки вебзастосунку для візуалізації алгоритмів класифікація відіграє важливу роль, оскільки дозволяє структурувати функціональність системи та забезпечити зручну навігацію для користувача. Відображення різних типів алгоритмів у єдиному середовищі дає можливість порівнювати їх поведінку, аналізувати ефективність та краще розуміти принципи їх роботи. Зокрема, користувач може спостерігати відмінності між алгоритмами сортування, оцінювати швидкість виконання пошуку або аналізувати процес обходу графа.

1.3 Методи візуалізації алгоритмів у програмному забезпеченні

Візуалізація алгоритмів є важливим напрямом у галузі програмної інженерії та освітніх інформаційних технологій, що спрямований на підвищення наочності процесів обробки даних та полегшення їх сприйняття користувачами. Так як алгоритми часто оперують абстрактними структурами та складними логічними переходами, їх текстове або формальне представлення не завжди є достатнім для повного розуміння. Саме тому застосування методів візуалізації дозволяє відобразити динаміку виконання алгоритму, що значно підвищує ефективність навчання та аналізу.

Одним із базових методів візуалізації є графічне відображення структур

даних. У цьому випадку елементи даних представляються у вигляді геометричних об’єктів, таких як стовпчики, вузли або лінії, що дозволяє наочно відображати їхній стан у процесі виконання алгоритму. Масиви можуть візуалізуватися у вигляді набору стовпців різної висоти, що є особливо ефективним при демонстрації алгоритмів сортування. У процесі виконання алгоритму змінюється положення або властивості цих об’єктів, що дозволяє користувачеві спостерігати за змінами в реальному часі [9].

Покрокова анімація виконання алгоритму передбачає відображення кожної окремої операції або групи операцій у вигляді послідовних кадрів. Користувач отримує можливість контролювати швидкість виконання або переходити між кроками вручну, що значно підвищує інтерактивність системи [10].

За допомогою різних кольорів можна позначати поточні елементи, елементи, що порівнюються, відсортовані або вже оброблені частини структури даних. Наприклад, під час виконання алгоритму сортування один колір може використовуватися для позначення елементів, що порівнюються, інший – для вже відсортованих, а ще один – для поточного елемента.

Окрему групу методів становлять графові візуалізації, що використовуються для представлення алгоритмів, пов’язаних із теорією графів. У цьому випадку дані відображаються у вигляді вершин і ребер, що дозволяє наочно демонструвати процеси обходу графа, пошуку шляхів або аналізу зв’язків між елементами. Динамічне підсвічування вершин і ребер під час виконання алгоритму забезпечує розуміння послідовності відвідування та змін у структурі.

Важливим методом є таблична або матрична візуалізація, яка застосовується для алгоритмів динамічного програмування. У цьому випадку дані представляються у вигляді таблиць, де кожна комірка відповідає певному підрезультату. Заповнення таблиці відбувається поступово, що дозволяє відстежити процес формування остаточного результату.

У сучасному програмному забезпеченні значну роль відіграє інтерактивність візуалізації. Користувач може змінювати вхідні дані, запускати алгоритми з різними параметрами, керувати швидкістю виконання та

аналізувати результати. Інтерактивні елементи, такі як кнопки, повзунки та поля введення, дозволяють адаптувати візуалізацію до потреб користувача та забезпечують більш глибоке розуміння процесів.

З технічної точки зору, реалізація візуалізації алгоритмів у вебзастосунках здійснюється за допомогою сучасних технологій, таких як HTML для структурування інтерфейсу, CSS для стилізації та JavaScript для динамічного оновлення стану елементів. Використання canvas або SVG дозволяє створювати складні графічні сцени та забезпечувати плавну анімацію. Завдяки цьому вебзастосунки стають універсальними платформами для демонстрації алгоритмів без необхідності встановлення додаткового програмного забезпечення [11].

1.4 Аналіз існуючих програмних рішень та вебсервісів

На сучасному етапі розвитку інформаційних технологій існує значна кількість програмних засобів і вебсервісів, призначених для візуалізації алгоритмів і структур даних. Їх використання спрямоване на підвищення ефективності навчання, аналізу та розуміння алгоритмічних процесів шляхом наочного відображення змін у даних під час виконання алгоритмів. Аналіз таких рішень дозволяє визначити їхні переваги та недоліки, та обґрунтувати необхідність розробки власного вебзастосунку.

Одним із найбільш відомих вебресурсів є Algorithm Visualizer (див. рис. 1.1), який надає можливість візуалізувати алгоритми безпосередньо на основі програмного коду. Платформа підтримує різні мови програмування та дозволяє спостерігати виконання алгоритму у вигляді анімованих процесів, що значно полегшує розуміння його логіки. Основною перевагою цього рішення є інтеграція коду та візуалізації, однак для початківців використання може бути складним через необхідність розуміння синтаксису мов програмування [12].

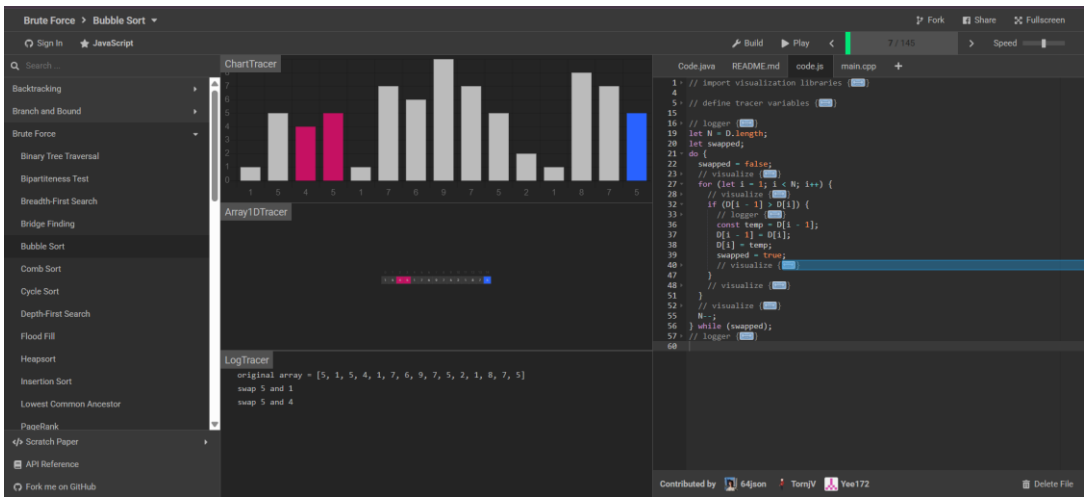


Рисунок 1.1 – Інтерфейс Algorithm Visualizer

Іншим популярним сервісом є VisuAlgo (див. рис. 1.2), який спеціалізується на інтерактивній візуалізації алгоритмів і структур даних. Цей вебресурс охоплює широкий спектр алгоритмів, включаючи сортування, пошук, роботу з графами та інші складні структури. Особливістю VisuAlgo є покрокова анімація, інтерактивність та наявність вбудованих тестових завдань, що сприяє закріпленню знань. Водночас інтерфейс системи може бути перевантаженим для нових користувачів, а також частково обмеженим для мобільних пристроїв [13].



Рисунок 1.2 – Інтерфейс VisuAlgo

Серед інших веборієнтованих інструментів варто відзначити AlgoVis (див. рис. 1.3), який пропонує простий та інтуїтивно зрозумілий інтерфейс для вивчення алгоритмів. Система дозволяє користувачам вводити власні дані та

спостерігати за виконанням алгоритмів у реальному часі, що підвищує рівень інтерактивності. Перевагою такого підходу є орієнтація на візуальне сприйняття та доступність, однак функціональні можливості можуть бути обмеженими порівняно з більш складними системами [14].



Рисунок 1.3 – Інтерфейс AlgoVis

Освітні платформи, такі як DSA Visualizer (див. рис. 1.4), поєднують візуалізацію алгоритмів із навчальними матеріалами, тестуванням та аналітикою. Такі системи забезпечують покрокове відображення алгоритмів, можливість зміни параметрів та інтерактивну взаємодію з користувачем. Вони орієнтовані як на початківців, так і на досвідчених користувачів, проте часто мають складну структуру та потребують значних ресурсів для повноцінного використання [15].

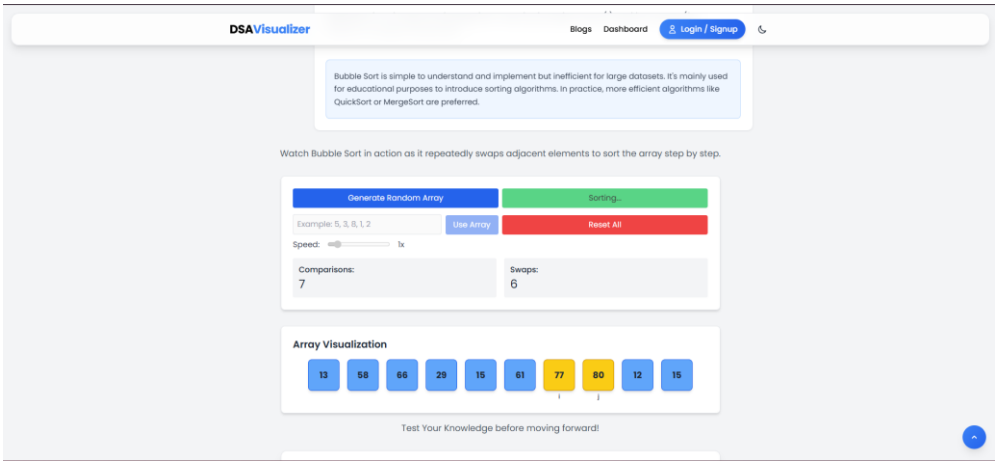


Рисунок 1.4 – Інтерфейс DSA Visualizer

Окрім вебсервісів, існують також десктопні програмні рішення для візуалізації даних і алгоритмів. Наприклад, такі системи, як Gephi (див. рис. 1.5) та Cytoscape (див. рис. 1.6), широко використовуються для аналізу та візуалізації складних мережевих структур. Вони забезпечують потужні інструменти для роботи з графами, включаючи алгоритми кластеризації та пошуку, проте орієнтовані переважно на наукові дослідження та професійне використання, що обмежує їх застосування в навчальному процесі [16].

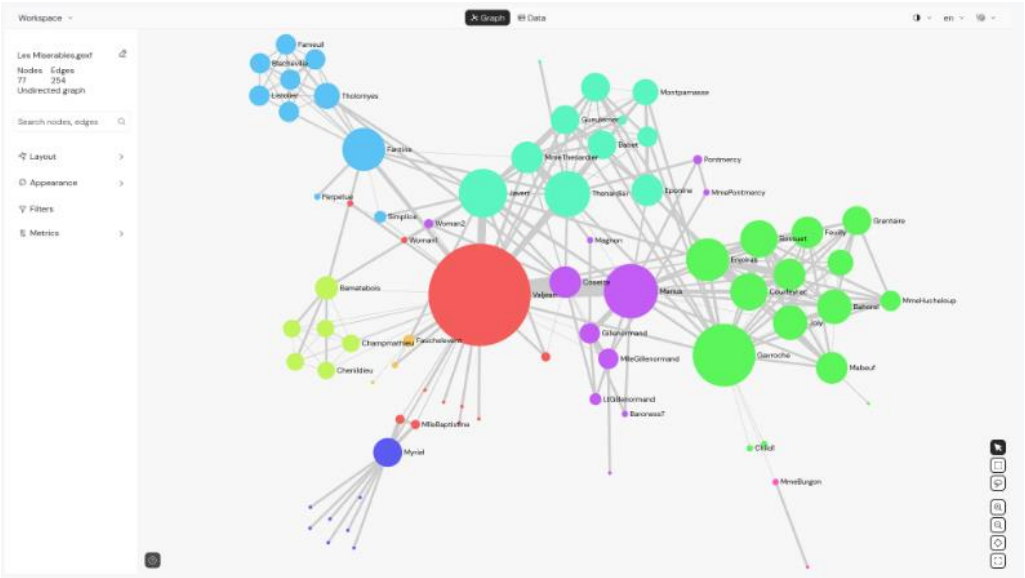


Рисунок 1.5 – Gephi

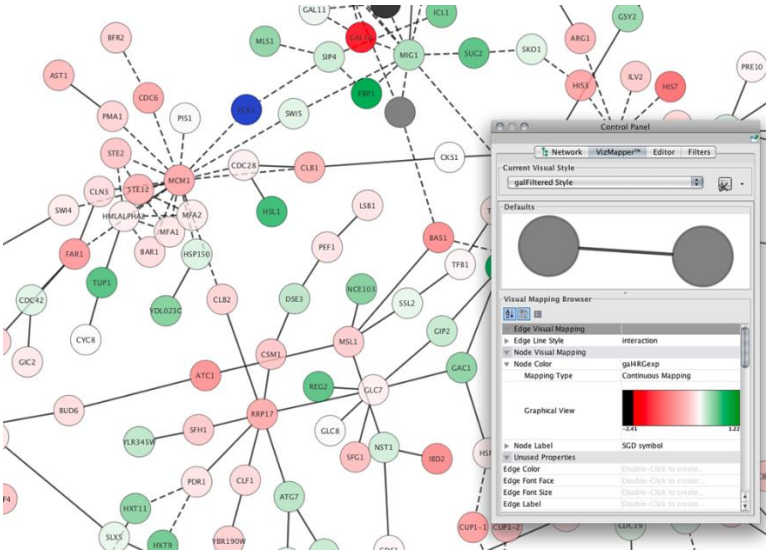


Рисунок 1.6 – Cytoscape

Також варто згадати бібліотеки та інструменти для програмної візуалізації, такі як NetworkX, які дозволяють створювати та аналізувати графи з подальшою їх візуалізацією. Подібні рішення є гнучкими та потужними, проте вимагають знань програмування та не забезпечують готового інтерактивного інтерфейсу для кінцевого користувача [17].

Аналіз розглянутих програмних засобів показує, що більшість із них мають ряд спільних переваг, зокрема наочність, інтерактивність та підтримку різних типів алгоритмів. Водночас вони мають і певні недоліки, серед яких складність інтерфейсу, обмеженість функціоналу або необхідність попередньої підготовки користувача. Це свідчить про доцільність розробки вебзастосунку, який поєднує простоту використання, універсальність та достатній рівень функціональності.

2 ПРОЄКТУВАННЯ ВЕБЗАСТОСУНКУ ДЛЯ ВІЗУАЛІЗАЦІЇ АЛГОРИТМІВ

2.1 Постановка задачі та вимоги до системи

Під час аналізу сучасних підходів до вивчення алгоритмів та структур даних було встановлено, що основною проблемою є складність їх сприйняття у текстовій або формалізованій формі. Традиційні методи навчання, які базуються на теоретичних описах та програмному коді, не завжди забезпечують достатній рівень розуміння, особливо для користувачів початкового рівня. У зв’язку з цим виникає необхідність створення програмного засобу, який дозволяє візуалізувати роботу алгоритмів у динамічному та інтерактивному режимі.

Постановка задачі полягає у розробці вебзастосунку, що забезпечує наочне відображення процесів виконання алгоритмів і змін стану структур даних у реальному часі. Система повинна надавати можливість користувачу обирати різні типи алгоритмів, запускати їх виконання, спостерігати за змінами даних на кожному кроці, та керувати параметрами візуалізації. Важливою складовою є забезпечення простоти використання та доступності застосунку без необхідності встановлення додаткового програмного забезпечення.

Розроблювана система повинна охоплювати кілька основних груп алгоритмів, зокрема алгоритми сортування, пошуку, обробки графів та алгоритми динамічного програмування. Для кожного з них необхідно забезпечити відображення ключових етапів виконання, таких як порівняння елементів, обмін значеннями, відвідування вузлів або заповнення таблиць.

Функціональні вимоги до системи передбачають наявність механізму генерації вхідних даних, можливість вибору алгоритму зі списку, запуску, зупинки та покрокового виконання процесу, а також зміни швидкості анімації. Система повинна відображати статистичні показники, такі як кількість операцій, порівнянь та обмінів, що дозволяє оцінити ефективність алгоритму. Важливою

функцією є можливість введення користувацьких даних для проведення експериментів і аналізу результатів.

Нефункціональні вимоги включають забезпечення високої швидкодії та плавності анімації, адаптивності інтерфейсу до різних типів пристроїв, зручності використання та зрозумілості інтерфейсу. Система повинна бути реалізована з використанням сучасних вебтехнологій, що забезпечують її кросплатформеність та доступність через веббраузер. Також важливими є вимоги до надійності та стабільності роботи, що передбачає коректну обробку помилок та забезпечення безперервності виконання алгоритмів.

Всі елементи управління повинні бути розміщені таким чином, щоб користувач міг легко взаємодіяти із системою без необхідності додаткового навчання. Візуалізація повинна бути наочною, із використанням кольорового кодування та анімації для відображення змін у даних.

2.2 Вибір технологій розробки вебзастосунку

Обґрунтований вибір технологій розробки є одним із ключових етапів створення програмного продукту, оскільки саме від цього залежить функціональність, продуктивність, масштабованість та зручність використання системи. У контексті розробки вебзастосунку для візуалізації структур даних та алгоритмів особливу увагу слід приділити технологіям, що забезпечують динамічне відображення даних, інтерактивність та високу швидкодію.

Для реалізації клієнтської частини застосунку було обрано стандартні вебтехнології, зокрема мову розмітки HTML, каскадні таблиці стилів CSS та мову програмування JavaScript. Використання HTML дозволяє створити структуровану основу інтерфейсу користувача, яка визначає розташування основних елементів, таких як панелі управління, область візуалізації та інформаційні блоки. CSS застосовується для оформлення інтерфейсу, забезпечення його привабливого вигляду та адаптивності до різних типів

пристроїв, що є важливою вимогою сучасних вебзастосунків [18].

Основна логіка роботи системи реалізується за допомогою мови JavaScript, яка забезпечує можливість обробки подій, управління станом даних та динамічного оновлення візуалізації без перезавантаження сторінки. Саме JavaScript використовується для реалізації алгоритмів сортування, пошуку, обходу графів та динамічного програмування, а також для створення механізму покрокового виконання алгоритмів і збору статистичних даних. Перевагою вибору цієї мови є її універсальність, широка підтримка у веббраузерах та можливість інтеграції з різними бібліотеками [19].

Для реалізації графічної візуалізації алгоритмів використовується HTML5 Canvas, який дозволяє малювати графічні об’єкти безпосередньо у браузері. Canvas забезпечує високу продуктивність та гнучкість при створенні анімацій, що є критично важливим для відображення динамічних процесів, таких як переміщення елементів масиву або зміна стану графа. Завдяки цьому забезпечується плавна анімація та можливість відображення великої кількості елементів без значних затримок.

Під час вибору технологій також розглядалися альтернативні підходи, зокрема використання бібліотек для роботи з графікою, таких як SVG або спеціалізовані фреймворки для візуалізації даних. Однак було прийнято рішення використовувати базові засоби Canvas у поєднанні з JavaScript, що дозволяє отримати більший контроль над процесом візуалізації та оптимізувати продуктивність системи.

Для організації структури коду використовується модульний підхід, що передбачає розподіл функціональності на окремі компоненти, зокрема модулі алгоритмів, модуль візуалізації та модуль управління станом.

Важливим аспектом є також забезпечення кросбраузерної сумісності та доступності вебзастосунку. Використання стандартних вебтехнологій гарантує коректну роботу системи у більшості сучасних браузерів без необхідності встановлення додаткового програмного забезпечення.

2.3 Архітектура та структура вебзастосунку

Архітектура розробленого вебзастосунку побудована за модульним принципом і орієнтована на забезпечення інтерактивної взаємодії користувача з системою, а також на ефективну візуалізацію алгоритмічних процесів. Загальна структура застосунку представлена на рисунку 2.1, де відображено основні функціональні компоненти та зв’язки між ними.



Рисунок 2.1 – Архітектура вебзастосунку

В основі архітектури лежить клієнтський підхід, відповідно до якого всі обчислення, обробка даних та візуалізація виконуються безпосередньо у браузері користувача. Це дозволяє досягти високої швидкодії, мінімізувати затримки та забезпечити автономність роботи системи без необхідності використання серверної частини.

Центральним елементом архітектури є модуль управління станом, який виконує роль координатора між іншими компонентами системи. Він відповідає за збереження поточного стану застосунку, обробку подій від користувача та передачу даних між модулями. Зокрема, цей модуль забезпечує контроль

параметрів виконання алгоритмів, таких як швидкість анімації, розмір вхідних даних та вибір конкретного алгоритму.

Користувацький інтерфейс є точкою входу в систему та забезпечує взаємодію користувача із застосунком. Через інтерфейс здійснюється вибір алгоритму, керування процесом візуалізації та налаштування параметрів. Усі дії користувача формують події, які передаються до модуля управління станом, що ініціює відповідні зміни у системі. Такий підхід дозволяє реалізувати подійно-орієнтовану модель взаємодії, що підвищує гнучкість і зручність використання застосунку.

Модуль алгоритмів реалізує логіку виконання різних типів алгоритмів, зокрема алгоритмів сортування, пошуку, роботи з графами та динамічного програмування. Він отримує параметри від модуля управління станом, виконує відповідні обчислення та формує послідовність кроків виконання алгоритму. Ці кроки передаються до модуля візуалізації, що дозволяє відобразити процес виконання у динамічному вигляді. Така організація забезпечує відокремлення логіки обчислень від їхнього відображення, що є важливою перевагою модульної архітектури.

Модуль візуалізації, реалізований із використанням технології Canvas, відповідає за графічне представлення даних та процесів. Він отримує дані про стан алгоритму у вигляді послідовності кроків і відображає їх у вигляді анімації. Залежно від типу алгоритму використовуються різні способи візуалізації, зокрема відображення масивів у вигляді стовпчиків або графів у вигляді вершин і ребер. Крім того, цей модуль забезпечує відображення статистичних показників виконання алгоритму, що дозволяє оцінити його ефективність.

Іншим компонентом архітектури є файлова структура проєкту, яка відображає організацію програмного коду. Вона включає каталоги алгоритмів, утиліт та конфігураційних файлів. Такий підхід дозволяє розділити логіку застосунку на незалежні частини, що спрощує розробку, тестування та подальше розширення системи. Зокрема, модуль алгоритмів містить реалізації окремих алгоритмів, модуль утиліт – допоміжні класи, а конфігураційні файли

забезпечують налаштування системи.

Взаємодія між компонентами системи здійснюється через передачу даних і подій. Користувацький інтерфейс генерує події, які обробляються модулем управління станом. Далі цей модуль передає необхідні параметри до модуля алгоритмів, який виконує обчислення та формує результати. Отримані дані передаються до модуля візуалізації, що забезпечує їх графічне відображення. Така схема взаємодії дозволяє забезпечити чітку логіку роботи системи та уникнути дублювання функціональності.

2.4 Моделювання взаємодії компонентів системи

Моделювання взаємодії компонентів системи є важливим етапом проєктування вебзастосунку, оскільки дозволяє формалізувати логіку функціонування системи, визначити ролі окремих компонентів та встановити зв’язки між ними. У межах даної роботи для опису поведінки та структури системи використано UML-діаграми, зокрема діаграму варіантів використання, діаграму класів та діаграму послідовності.

На рисунку 2.2 представлено діаграму варіантів використання, яка відображає основні сценарії взаємодії користувача із вебзастосунком. Головним актором системи є користувач, який взаємодіє із застосунком через інтерфейс. Основними варіантами використання є вибір алгоритму, налаштування параметрів, генерація або оновлення вхідних даних, керування процесом візуалізації, покроковий перегляд, перегляд статистики, перегляд інформації про алгоритм та порівняння алгоритмів. Дана діаграма демонструє, що система орієнтована на інтерактивну взаємодію та надання користувачу засобів для дослідження роботи алгоритмів.

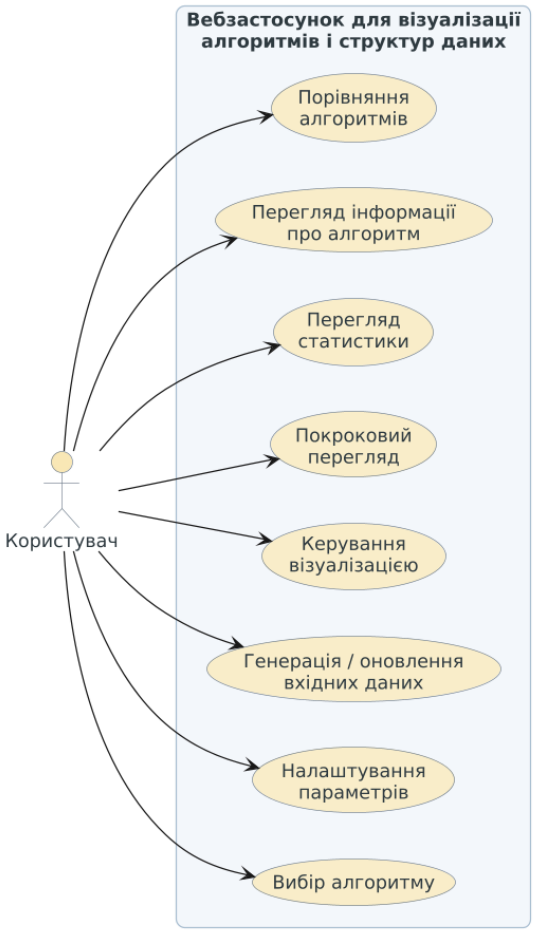


Рисунок 2.2 – Діаграма варіантів використання

На рисунку 2.3 наведено діаграму класів, яка відображає внутрішню структуру системи та взаємозв’язки між її основними компонентами. Центральним елементом логіки алгоритмів є базовий клас BaseAlgorithm, який визначає спільну структуру для всіх алгоритмів, зберігає послідовність кроків виконання та статистичні показники. Від нього успадковуються конкретні класи алгоритмів, зокрема алгоритми сортування, пошуку, роботи з графами та динамічного програмування.

Візуалізація результатів виконання алгоритмів здійснюється за допомогою компонента Visualizer, який відповідає за відображення даних на елементі Canvas. Для збирання показників роботи алгоритмів використовується компонент Statistics, який фіксує кількість операцій, порівнянь, обмінів або переміщень.

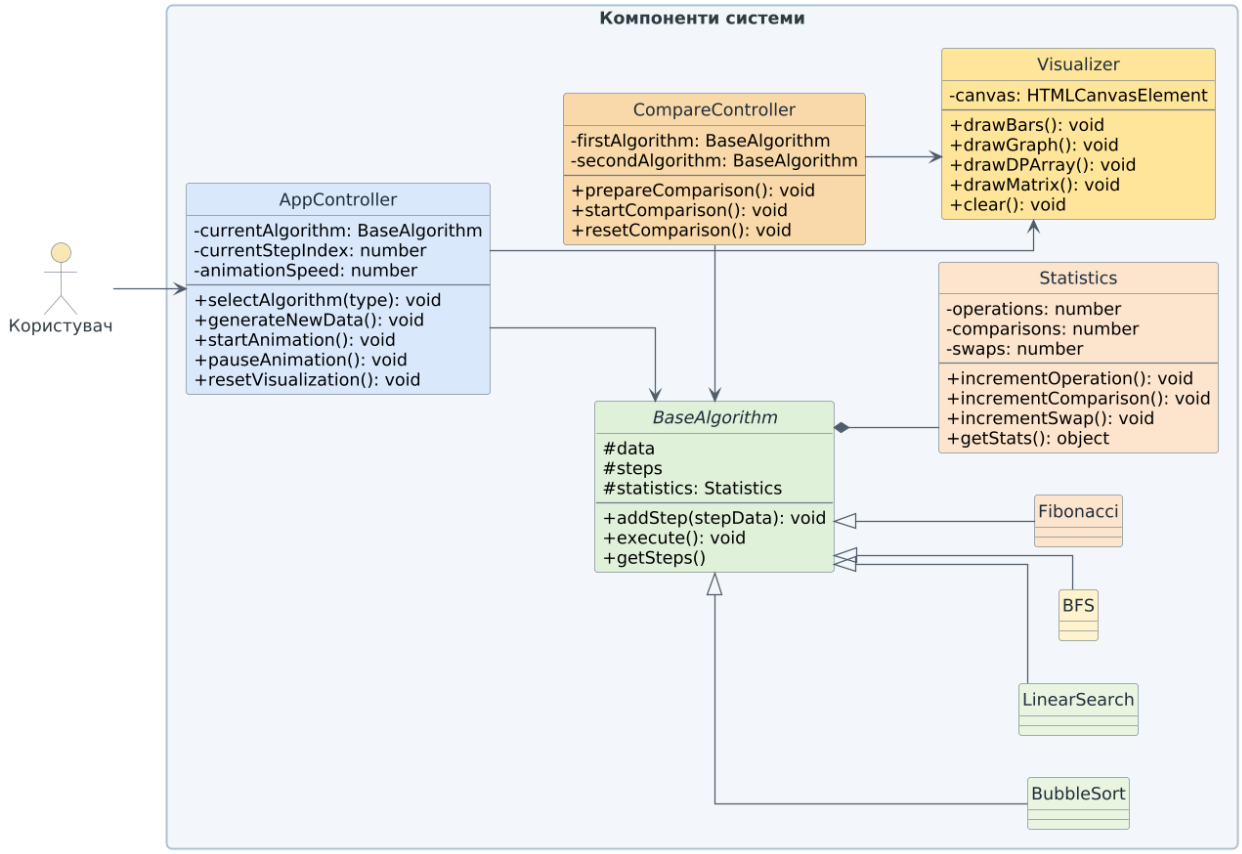


Рисунок 2.3 – Діаграма класів

Керування взаємодією користувача із застосунком виконує модуль керування інтерфейсом, який обробляє вибір алгоритму, зміну параметрів, запуск, паузу, скидання та покрокове виконання. Окремий модуль порівняння відповідає за одночасний запуск двох алгоритмів та відображення результатів їх роботи. Така структура забезпечує чіткий розподіл відповідальності між компонентами системи.

На рисунку 2.4 представлено діаграму послідовності, яка демонструє динамічний аспект взаємодії компонентів під час виконання алгоритму. Процес починається з дії користувача, який обирає алгоритм через інтерфейс. Після цього модуль керування інтерфейсом передає вибір до відповідного алгоритму, формує або отримує вхідні дані та ініціює виконання.

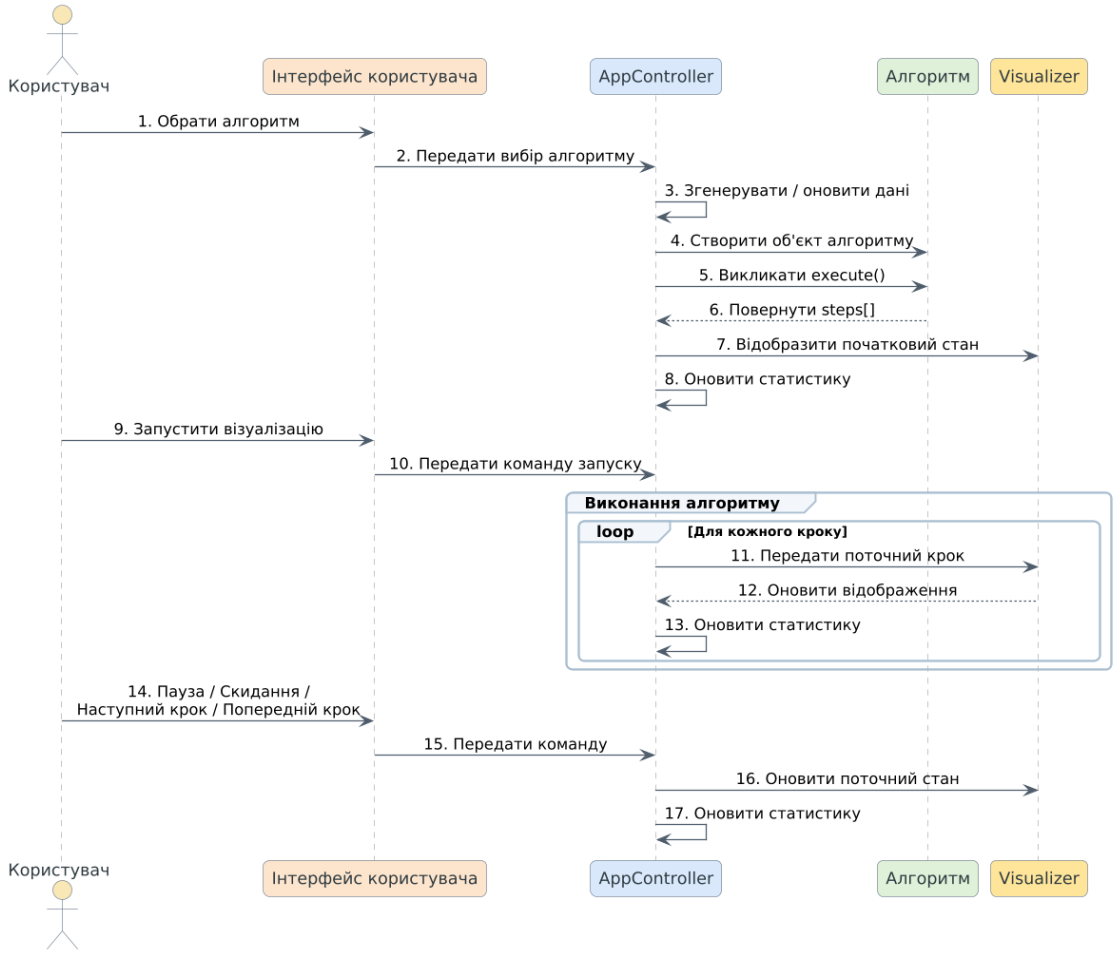


Рисунок 2.4 – Діаграма послідовності

Далі алгоритм формує послідовність кроків, які поступово передаються до візуалізатора. Компонент Visualizer відображає кожен крок на полотні Canvas, а компонент Statistics оновлює показники виконання. Цей процес повторюється до завершення роботи алгоритму або до зупинки з боку користувача. Використання циклу в діаграмі підкреслює ітеративний характер обробки даних та візуалізації.

Загалом, поєднання трьох типів UML-діаграм дозволяє отримати повне уявлення про систему: діаграма варіантів використання описує взаємодію із користувачем, діаграма класів – структуру програмного забезпечення, а діаграма послідовності – динаміку виконання процесів.

3 РОЗРОБКА ТА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКУ

3.1 Реалізація алгоритмів сортування

Реалізація алгоритмів сортування є однією з ключових складових розробленого вебзастосунку, оскільки саме ці алгоритми найчастіше використовуються для демонстрації принципів обробки даних та є наочними для візуалізації. У межах даної роботи реалізовано декілька базових алгоритмів сортування, зокрема сортування бульбашкою, вставками, злиттям, швидке сортування та сортування купою. Кожен із них відображає різні підходи до впорядкування даних та має свої особливості реалізації.

Усі алгоритми сортування реалізовані у вигляді окремих класів, які наслідують базовий клас алгоритму. Базовий клас відповідає за збереження початкових даних, послідовності кроків виконання та статистичних показників, таких як кількість операцій, порівнянь і переміщень або обмінів елементів.

Алгоритм сортування бульбашкою реалізовано як ітеративний процес, у якому відбувається послідовне порівняння сусідніх елементів і їх обмін у разі невпорядкованості. Після кожного проходу найбільший елемент поступово переміщується в кінець масиву. У процесі виконання алгоритму фіксуються кроки порівняння, що дозволяє відобразити поточний стан масиву та підсвітити елементи, які беруть участь у порівнянні. Даний алгоритм має квадратичну складність і використовується переважно для демонстраційних цілей.

Сортування вставками базується на поступовому формуванні впорядкованої частини масиву. Кожен новий елемент порівнюється з попередніми та вставляється у відповідну позицію. Реалізація алгоритму передбачає зсув елементів праворуч для звільнення місця під вставку. У візуалізації це відображається як зміна стану масиву та поступове формування відсортованої частини.

Алгоритм сортування злиттям реалізовано за принципом «розділяй і володарюй», відповідно до якого масив рекурсивно ділиться на менші

підмасиви, які потім об’єднуються у відсортований результат. У процесі реалізації особлива увага приділяється етапу злиття, де відбувається порівняння елементів із двох підмасивів та формування впорядкованої послідовності. Візуалізація цього алгоритму відображає проміжні стани масиву під час об’єднання його частин.

Швидке сортування також базується на рекурсивному підході та використовує опорний елемент для поділу масиву на дві частини. У реалізації як опорний елемент використовується останній елемент поточного фрагмента масиву. Елементи, менші за опорний, розміщуються ліворуч, а більші залишаються праворуч. Реалізація алгоритму включає процедуру розбиття масиву, яка визначає позицію опорного елемента. У процесі візуалізації відображаються порівняння елементів із опорним та зміни стану масиву.

Сортування купою використовує структуру даних «бінарна купа», яка дозволяє ефективно знаходити максимальний елемент. Спочатку з масиву будується max-heap, після чого найбільший елемент переміщується в кінець масиву, і структура купи перебудовується для решти елементів. Цей процес повторюється до повного сортування. У візуалізації купа відображається через поточний стан масиву у вигляді стовпчиків, де підсвічуються елементи, що порівнюються або вже належать до відсортованої частини.

Особливістю реалізації всіх алгоритмів є інтеграція механізму збору кроків виконання. Після основних операцій порівняння, переміщення або зміни стану масиву формується об’єкт, який містить інформацію про поточний стан масиву, індекси елементів, що порівнюються, а також статистичні показники на відповідному етапі виконання. Ці дані передаються до модуля візуалізації, який відображає їх у вигляді анімації. Завдяки цьому користувач може спостерігати за процесом сортування у покроковому режимі.

У процесі виконання алгоритмів ведеться підрахунок статистичних показників, що дозволяє оцінити ефективність кожного методу. Зокрема, фіксується кількість операцій, порівнянь і переміщень або обмінів елементів. Це дає можливість порівнювати алгоритми між собою та аналізувати їх поведінку залежно від розміру та структури вхідних дани

3.2 Реалізація алгоритмів пошуку

Алгоритми пошуку є невід’ємною складовою обробки даних, оскільки дозволяють знаходити необхідні елементи у структурах даних. У межах розробленого вебзастосунку реалізовано декілька основних алгоритмів пошуку, а саме лінійний пошук, бінарний пошук, пошук стрибками та інтерполяційний пошук. Вибір цих алгоритмів обумовлений їх фундаментальністю, відмінностями у підходах до пошуку та наочністю для візуалізації.

Реалізація алгоритмів пошуку виконана у вигляді окремих класів, що наслідують базовий клас алгоритму. Це дозволяє уніфікувати підхід до обробки даних, збереження кроків виконання та збору статистичних показників. Кожен алгоритм працює з масивом даних, який формується автоматично або завантажується користувачем із файлу, що забезпечує можливість тестування різних сценаріїв.

Лінійний пошук є найпростішим алгоритмом, який здійснює послідовний перегляд елементів масиву до знаходження шуканого значення або завершення перевірки всіх елементів. У процесі реалізації алгоритму кожен елемент масиву порівнюється з шуканим значенням, що дозволяє визначити його позицію. Для кожного кроку виконання формується відповідний стан, який включає індекс поточного елемента, значення цілі пошуку та інформацію про виконане порівняння. У разі знаходження елемента алгоритм завершує роботу, а у протилежному випадку повертає значення, що свідчить про відсутність елемента у масиві. Даний алгоритм має лінійну часову складність і є ефективним для невеликих або невпорядкованих масивів.

Бінарний пошук є більш ефективним алгоритмом, який застосовується до відсортованих масивів. Його реалізація базується на принципі поділу області пошуку навпіл на кожному кроці. Алгоритм створює відсортовану копію масиву, визначає середній елемент поточного діапазону та порівнює його із шуканим значенням. Якщо значення збігаються, пошук завершується; якщо шуканий елемент більший, пошук продовжується у правій частині масиву, а якщо менший – у лівій. У процесі виконання фіксуються межі пошуку та поточний середній

елемент, що дозволяє відобразити логіку алгоритму у візуальній формі.

Пошук стрибками також працює з відсортованим масивом. У реалізації алгоритму спочатку створюється відсортована копія вхідних даних, після чого масив переглядається блоками певного розміру. Розмір стрибка може визначатися автоматично як корінь із кількості елементів або задаватися користувачем. Алгоритм перевіряє кінцеві елементи блоків і пропускає ті частини масиву, у яких шукане значення не може знаходитися. Після визначення можливого блоку виконується послідовний пошук усередині нього. У візуалізації відображаються поточний блок, елементи, що перевіряються, та зміна діапазону пошуку.

Інтерполяційний пошук призначений для роботи з відсортованими числовими масивами. На відміну від бінарного пошуку, він не завжди перевіряє середину діапазону, а оцінює ймовірну позицію шуканого елемента на основі його значення та значень на межах поточного діапазону. У процесі виконання алгоритм визначає межі пошуку, обчислює позицію за формулою інтерполяції, порівнює відповідний елемент із цільовим значенням і звужує діапазон залежно від результату. Такий підхід є особливо ефективним для рівномірно розподілених числових даних.

Важливою особливістю реалізації алгоритмів пошуку є інтеграція механізму покрокової візуалізації. На кожному основному етапі виконання формується об’єкт стану, який містить поточний масив, індекси елементів, що перевіряються, межі активного діапазону пошуку, цільове значення та додаткове текстове пояснення. Ці дані передаються до модуля візуалізації, який відображає їх у вигляді графічних елементів із відповідним кольоровим виділенням.

Під час виконання алгоритмів також здійснюється підрахунок статистичних показників, зокрема кількості порівнянь та загальної кількості операцій. Це дозволяє оцінити ефективність кожного алгоритму та продемонструвати переваги більш оптимізованих підходів залежно від структури вхідних даних.

3.3 Реалізація алгоритмів теорії графів

Алгоритми теорії графів займають важливе місце в сучасній інформатиці, оскільки дозволяють моделювати та аналізувати складні системи, що складаються з взаємопов’язаних елементів. У межах розробленого вебзастосунку реалізовано базові алгоритми обходу графів та пошуку найкоротших шляхів, зокрема пошук у ширину (BFS), пошук у глибину (DFS) та алгоритм Дейкстри. Вибір саме цих алгоритмів обумовлений їх широким застосуванням та наочністю при візуалізації.

Усі алгоритми реалізовано у вигляді окремих класів, які наслідують базовий клас алгоритму, що забезпечує уніфікований підхід до обробки даних, збереження кроків виконання та збору статистики. Граф у системі представлено у вигляді множини вершин і списку ребер, що дозволяє зручно визначати зв’язки між елементами та здійснювати їх обхід. У програмі граф формується автоматично відповідно до заданої кількості вершин, а для алгоритму Дейкстри додатково генеруються ваги ребер.

Алгоритм пошуку у ширину (Breadth-First Search) реалізовано з використанням черги та множини відвіданих вершин. Процес починається з вибраної початкової вершини, яка додається до черги, після чого алгоритм послідовно відвідує суміжні вершини. Для кожної вершини визначаються її сусіди, які ще не були відвідані, і додаються до черги. У процесі виконання алгоритму фіксуються відвідані вершини, поточна вершина та стан черги, що дозволяє відобразити послідовність обходу у візуальній формі.

Алгоритм пошуку у глибину (Depth-First Search) реалізовано за допомогою рекурсивного підходу, який дозволяє заглиблюватися в граф до максимально можливого рівня перед поверненням до попередніх вершин. Кожна відвідана вершина додається до множини відвіданих, після чого алгоритм переходить до одного з її невідвіданих сусідів. Якщо всі сусіди вже відвідані, відбувається повернення до попереднього рівня. У візуалізації відображається поточна вершина, множина вже відвіданих вершин і поточний шлях обходу, що дозволяє простежити роботу рекурсивного алгоритму.

Алгоритм Дейкстри використовується для знаходження найкоротших шляхів у зваженому графі з невід’ємними вагами ребер. Його реалізація базується на поступовому виборі невідвіданої вершини з мінімальною відстанню від початкової та оновленні відстаней до її сусідів. У даній реалізації вибір вершини з мінімальною відстанню здійснюється звичайним лінійним пошуком, без використання пріоритетної черги. Для кожної вершини зберігається поточна оцінка відстані, яка оновлюється у разі знаходження коротшого шляху. Алгоритм може виконувати пошук від початкової вершини до всіх інших вершин або завершувати роботу після досягнення вибраної кінцевої вершини. Візуалізація відображає поточну вершину, множину відвіданих вершин, ваги ребер та значення відстаней, що дозволяє наочно простежити процес пошуку оптимального маршруту.

Особливістю реалізації алгоритмів теорії графів є використання допоміжних методів для побудови списку суміжності на основі списку ребер. У BFS і DFS формується список суміжності для неорієнтованого графа, а в алгоритмі Дейкстри використовується зважений список суміжності. Це забезпечує універсальність підходу та дозволяє зручно отримувати сусідні вершини під час виконання алгоритмів. Кожен значущий крок алгоритму супроводжується формуванням об’єкта стану, який передається до модуля візуалізації.

У процесі візуалізації графи відображаються у вигляді вершин, розташованих на площині, та ребер, що їх з’єднують. Для підвищення наочності використовуються кольорові позначення, які дозволяють виділити поточну вершину, відвідані вершини, активні зв’язки, чергу або шлях обходу, а також значення відстаней для алгоритму Дейкстри.

3.4 Реалізація алгоритмів динамічного програмування

Алгоритми динамічного програмування є ефективним підходом до розв’язання задач, що характеризуються наявністю перекривних підзадач та

оптимальної підструктури. Їх основна ідея полягає у збереженні результатів проміжних обчислень з метою уникнення повторних розрахунків, що дозволяє суттєво зменшити часову складність. У межах розробленого вебзастосунку реалізовано два базові алгоритми динамічного програмування, а саме обчислення чисел Фібоначчі та знаходження найдовшої спільної підпослідовності (LCS).

Реалізація алгоритмів виконана у вигляді окремих класів, які наслідують базовий клас алгоритму, що забезпечує уніфікований підхід до обробки даних, збереження кроків виконання та збору статистичних показників. Кожен алгоритм формує послідовність станів, що відображають проміжні результати обчислень і передаються до модуля візуалізації.

Алгоритм обчислення чисел Фібоначчі реалізовано із використанням табличного підходу (bottom-up), який передбачає послідовне заповнення масиву значень dp. Початкові значення визначаються як базові випадки: dp[0] = 0 та dp[1] = 1. Після цього кожен наступний елемент обчислюється як сума двох попередніх значень. У процесі виконання алгоритму формується масив, який поступово заповнюється новими результатами. Кожен крок супроводжується фіксацією поточного індексу, використаних попередніх елементів, формули обчислення та поточного результату, що дозволяє відобразити процес побудови послідовності у вигляді анімації. У вебзастосунку значення n обмежується діапазоном від 0 до 50, що забезпечує зручність візуалізації.

Алгоритм знаходження найдовшої спільної підпослідовності (Longest Common Subsequence) реалізовано з використанням двовимірної таблиці, яка містить результати порівняння префіксів двох рядків. У процесі заповнення таблиці кожна комірка визначається на основі попередніх значень залежно від того, чи збігаються відповідні символи рядків. Якщо символи рівні, значення комірки збільшується на одиницю відносно попередньої діагональної комірки. Якщо символи різні, обирається максимальне значення з верхньої або лівої комірки.

При реалізації алгоритму LCS особлива увага приділяється формуванню візуалізації, оскільки двовимірна структура даних потребує більш складного

відображення. Кожен крок алгоритму супроводжується оновленням відповідної комірки таблиці, підсвічуванням поточної комірки та комірок, значення яких використовуються для обчислення. Крім того, під час роботи алгоритму формується поточний варіант спільної підпослідовності, а після заповнення таблиці відновлюється фінальний результат рухом назад від правого нижнього кута матриці. Візуалізація представлена у вигляді матриці, де кожна комірка містить числове значення, що відповідає довжині спільної підпослідовності для відповідних префіксів. Для збереження читабельності візуалізації вхідні рядки обмежуються довжиною до 14 символів.

Важливою складовою реалізації є механізм збереження кроків виконання, який дозволяє відобразити процес обчислень у покроковому режимі. Для кожного алгоритму формується об’єкт стану, що містить поточні дані, індекси оброблюваних елементів, використані попередні значення, пояснення поточного кроку та результати обчислень. Ці дані передаються до модуля візуалізації, який забезпечує їх графічне відображення.

Під час виконання алгоритмів здійснюється підрахунок статистичних показників, зокрема кількості операцій і порівнянь. Для обчислення чисел Фібоначчі фіксуються операції заповнення таблиці, а для LCS додатково підраховуються порівняння символів двох рядків. Це дозволяє оцінити ефективність алгоритмів та продемонструвати переваги динамічного програмування порівняно з наївними підходами, що використовують рекурсію без запам’ятовування проміжних результатів.

3.5 Реалізація розділу структур даних

Розділ структур даних у розробленому вебзастосунку реалізовано як окремий теоретичний модуль, який доповнює візуалізацію алгоритмів та пояснює принципи організації даних у пам’яті. Його призначення полягає в тому, щоб показати користувачу зв’язок між алгоритмами та структурами, на основі яких ці алгоритми працюють. Якщо алгоритм визначає послідовність дій, то

структура даних визначає спосіб зберігання елементів, доступу до них та виконання основних операцій.

У вебзастосунку розділ структур даних інтегровано в головну сторінку та відкривається через відповідний пункт навігації. При виборі цього розділу основні блоки візуалізації алгоритмів приховуються, а користувачу відображається навчальний матеріал, поділений на окремі тематичні підрозділи. Такий підхід дозволяє не перевантажувати інтерфейс і водночас зберегти зв’язок між теоретичним поясненням та практичною частиною застосунку.

У межах розділу розглянуто поняття структури даних, абстрактного типу даних та конкретної реалізації. Це дозволяє пояснити різницю між логічною поведінкою структури та способом її фактичного зберігання у пам’яті. Наприклад, стек як абстрактний тип даних визначається операціями додавання, вилучення та перегляду верхнього елемента, але може бути реалізований на основі масиву, динамічного масиву або зв’язного списку.

У розділі описано лінійні структури, до яких належать масив, динамічний масив, зв’язний список, стек, черга та дек, а також нелінійні структури, зокрема дерева, купи та графи. Крім цього, пояснюється поділ структур на статичні й динамічні, а також різниця між фізичним і логічним поданням даних. Це важливо, оскільки одна й та сама структура може логічно мати один вигляд, але в пам’яті зберігатися зовсім інакше. Наприклад, купа логічно розглядається як дерево, але часто реалізується через масив.

Масив описано як базову структуру даних, елементи якої зберігаються послідовно та мають індекси. Саме масиви активно використовуються у вебзастосунку для демонстрації алгоритмів сортування та пошуку. У візуалізації елементи масиву подаються у вигляді стовпчиків, де висота відповідає числовому значенню, а колір відображає поточний стан елемента: порівняння, активну позицію або відсортовану частину. Це дає змогу наочно показати, як алгоритми змінюють порядок елементів або перевіряють окремі позиції.

Динамічний масив розглядається як розвиток звичайного масиву, який може змінювати свою місткість під час виконання програми. У розділі

пояснюється принцип розширення пам’яті, копіювання елементів у новий блок та амортизована складність додавання в кінець. Такий матеріал допомагає зрозуміти, чому окремі операції можуть бути дорогими, але в середньому структура залишається ефективною для послідовного накопичення даних.

Зв’язний список подано як структуру, що складається з вузлів, кожен із яких містить значення та посилання на інші вузли, пояснюється різниця між однозв’язним і двозв’язним списком, а також переваги й недоліки такої організації даних. Список є зручним для вставок і видалень за відомою позицією, але менш ефективний для доступу за індексом, оскільки потребує послідовного проходу елементів.

Стек описано як структуру, що працює за принципом LIFO, тобто останній доданий елемент обробляється першим. У межах проєкту стек пов’язаний насамперед із рекурсією та алгоритмом пошуку у глибину. Хоча в реалізації DFS використовується рекурсивний підхід, логічно він відповідає роботі стека викликів, що дозволяє пояснити механізм повернення до попередніх вершин після завершення обходу поточного шляху.

Черга розглядається як структура, що працює за принципом FIFO, коли перший доданий елемент обробляється першим. Ця структура безпосередньо пов’язана з алгоритмом BFS, у якому черга зберігає вершини, що вже були виявлені, але ще не оброблені. Завдяки цьому пошук у ширину проходить граф рівнями, спочатку обробляючи сусідів початкової вершини, а потім сусідів наступного рівня.

Дек описано як двосторонню чергу, у якій елементи можна додавати й видаляти з обох кінців. Він поєднує властивості стека та черги, оскільки залежно від способу використання може підтримувати як поведінку LIFO, так і FIFO. Такий підхід допомагає показати, що структури даних можуть бути гнучкими інструментами, які адаптуються до різних задач.

Хеш-таблиця представлена як структура для зберігання пар «ключ – значення». У розділі пояснюється роль хеш-функції, яка перетворює ключ на індекс у внутрішньому масиві, а також розглядається проблема колізій. Описано

основні підходи до їх розв’язання, зокрема метод ланцюжків і відкриту адресацію. Це дозволяє показати, чому хеш-таблиці забезпечують швидкий пошук у середньому випадку, але можуть втрачати ефективність за невдалого розподілу ключів.

Дерева описано як ієрархічні структури даних, що складаються з вузлів і зв’язків між ними. Основні поняття, які пояснюються в розділі: корінь, лист, висота, піддерево та обхід. Також розглянуто бінарне дерево пошуку, у якому значення лівого піддерева менші за значення вузла, а значення правого піддерева більші. Така структура дозволяє виконувати пошук, вставку та видалення ефективно за умови, що дерево залишається збалансованим.

Купа розглядається як деревоподібна структура, у якій батьківський вузол має пріоритет відносно дочірніх. У розділі пояснюється різниця між max-heap і min-heap, а також спосіб подання купи через масив. Ця структура безпосередньо пов’язана з реалізованим алгоритмом Heap Sort, де спочатку будується max-heap, а потім найбільші елементи послідовно переміщуються в кінець масиву.

Граф описано як структуру, що складається з вершин і ребер та використовується для моделювання зв’язків між об’єктами. У розділі наведено пояснення орієнтованих і неорієнтованих, зважених і незважених графів, а також способів їх подання у пам’яті. Особлива увага приділяється списку суміжності та матриці суміжності. У межах проєкту графи використовуються для візуалізації BFS, DFS та алгоритму Дейкстри.

Завершальна частина розділу присвячена порівнянню складності основних операцій і практичному вибору структури даних. Користувачу пояснюється, що вибір структури залежить від характеру задачі: частоти доступу за індексом, необхідності швидкого пошуку за ключем, кількості вставок і видалень, наявності ієрархічних або мережевих зв’язків. Завдяки цьому розділ структур даних виконує не лише довідкову функцію, а й допомагає краще зрозуміти логіку роботи візуалізованих алгоритмів.

3.6 Розробка модуля візуалізації та інтерфейсу користувача

Однією з ключових складових розробленого вебзастосунку є модуль візуалізації, який забезпечує наочне відображення процесів виконання алгоритмів, а також інтерфейс користувача, що дозволяє здійснювати взаємодію із системою. Саме поєднання цих компонентів формує інтерактивне середовище, у якому користувач може не лише запускати алгоритми, але й аналізувати їхню поведінку в покроковому режимі.

Модуль візуалізації реалізовано з використанням технології HTML5 Canvas, яка дозволяє здійснювати динамічне відображення графічних елементів безпосередньо у браузері. Основною функцією цього модуля є відтворення стану даних на кожному етапі виконання алгоритму. Для цього використовується клас Visualizer, який відповідає за ініціалізацію графічного полотна, його очищення, масштабування відповідно до розміру контейнера та малювання елементів залежно від типу алгоритму.

У випадку алгоритмів, що працюють із масивами, використовується метод відображення у вигляді стовпчикової діаграми. Кожен елемент масиву представлений вертикальним прямокутником, висота якого пропорційна значенню елемента. У процесі виконання алгоритму змінюється стан і колір цих елементів, що дозволяє відобразити операції порівняння, переміщення, пошуку та впорядкування. Для підвищення наочності застосовується кольорове кодування: елементи, що порівнюються, виділяються одним кольором, поточний елемент – іншим, а відсортовані або знайдені елементи – третім. Також для пошукових алгоритмів може відображатися активний діапазон пошуку та текстове пояснення поточного кроку.

Для алгоритмів теорії графів реалізовано окремий метод візуалізації, який відображає вершини у вигляді точок, розташованих на площині, та ребра у вигляді ліній, що їх з’єднують. Позиції вершин розраховуються автоматично з урахуванням кількості вузлів і розміру полотна. Під час виконання алгоритму відвідані вершини змінюють свій колір, поточна вершина додатково виділяється,

а активні зв’язки підсвічуються. Для алгоритму Дейкстри також відображаються ваги ребер і поточні значення відстаней від початкової вершини. Для BFS і DFS додатково може відображатися стан черги або поточного шляху обходу.

Окремі методи візуалізації реалізовано для алгоритмів динамічного програмування. Для обчислення чисел Фібоначчі використовується відображення масиву dp, у якому підсвічується поточна комірка та значення, що використовуються для обчислення. Для алгоритму LCS застосовується матрична візуалізація, де кожна комірка таблиці містить числове значення, а поточна комірка та джерела обчислення виділяються кольором. Крім того, на полотні відображається поточний результат LCS і пояснення виконуваного кроку.

Модуль візуалізації працює у тісній взаємодії з модулями алгоритмів, отримуючи від них послідовність кроків виконання. Кожен крок містить інформацію про поточний стан даних, індекси активних елементів, відвідані вершини, діапазони пошуку, матриці або масиви динамічного програмування, а також додаткові параметри для пояснення дії. Основна логіка визначає тип кроку і передає його до відповідного методу візуалізації: для масивів, графів, DP-масивів або матриць.

Інтерфейс користувача реалізовано із використанням HTML, CSS та JavaScript. Він містить основні елементи управління, зокрема вибір алгоритму, налаштування розміру масиву, регулювання швидкості анімації, завантаження вхідних даних із TXT-файлу, генерацію нових даних, запуск, паузу, скидання та збереження звіту у форматі PDF. Для деяких алгоритмів передбачено додаткові налаштування: цільове значення для пошуку, розмір стрибка для Jump Search, початкова і кінцева вершини графа, кількість вершин, номер числа Фібоначчі або рядки для LCS.

Навігація між алгоритмами реалізована через бічну панель, у якій алгоритми згруповано за категоріями: сортування, пошук, теорія графів, динамічне програмування та структури даних. Формування цієї панелі винесено в окремий модуль, що дозволяє централізовано описувати доступні розділи та їхні характеристики. Для алгоритмів також передбачено перехід до сторінки з

теоретичним описом, що доповнює практичну візуалізацію.

Взаємодія користувача із системою реалізується через обробку подій за допомогою JavaScript. Основна логіка управління зосереджена у файлі app.js, який відповідає за вибір алгоритму, створення відповідного об’єкта, запуск обчислень, збереження кроків, керування анімацією та оновлення статистики. Окремий файл index-events.js відповідає за прив’язку кнопок інтерфейсу до функцій запуску, паузи, скидання, генерації нових даних і експорту звіту. Такий поділ спрощує підтримку коду та робить структуру застосунку зрозумілішою.

Для забезпечення плавності анімації використовується поєднання таймерів і requestAnimationFrame. Таймери контролюють перехід між кроками алгоритму відповідно до обраної користувачем швидкості, а requestAnimationFrame використовується для плавного перемальовування графічних елементів на полотні. Реалізовано також можливість покрокового керування, зокрема перехід до наступного або попереднього кроку для окремих алгоритмів.

Для адаптивності інтерфейсу реалізовано автоматичне масштабування полотна відповідно до розміру його контейнера та щільності пікселів екрана. При зміні розміру вікна або орієнтації пристрою полотно оновлюється, а поточний крок перемальовується. Це забезпечує коректне відображення візуалізації на різних екранах і пристроях.

Важливою складовою інтерфейсу є відображення статистичних показників виконання алгоритмів, таких як кількість операцій, порівнянь, обмінів або переміщень, а також умовний час виконання алгоритму. Ці показники оновлюються під час відтворення кроків і дозволяють користувачу не лише спостерігати за процесом виконання, але й аналізувати ефективність різних алгоритмів.

Додатковою функціональністю є можливість формування звіту за результатами роботи алгоритму. Під час експорту система зберігає початкові дані, налаштування, поточний або фінальний стан виконання, статистичні показники та візуальні знімки полотна. Це дає змогу використовувати застосунок не лише як інтерактивний навчальний інструмент, а й як засіб для фіксації

результатів аналізу алгоритмів.

3.7 Тестування та аналіз результатів роботи застосунку

Завершальним етапом розробки вебзастосунку для візуалізації структур даних та алгоритмів є проведення тестування та аналіз отриманих результатів. Метою цього етапу є перевірка коректності функціонування програмного продукту, оцінка правильності реалізованих алгоритмів, а також визначення якості візуалізації та зручності користувацького інтерфейсу.

Тестування застосунку здійснювалося у браузерному середовищі з використанням різних наборів вхідних даних. Дані формувалися автоматично засобами програми або завантажувалися користувачем із текстового файлу. Основна увага приділялася перевірці правильності виконання алгоритмів сортування, пошуку, обробки графів і динамічного програмування, а також відповідності візуалізації кожному етапу їх виконання.

При тестуванні алгоритмів сортування, зокрема Bubble Sort (див. рис. 3.1), Insertion Sort (див. рис. 3.2), Merge Sort (див. рис. 3.3), Quick Sort (див. рис. 3.4) та Heap Sort (див. рис. 3.5), було перевірено коректність впорядкування масивів чисел різного розміру. У результаті тестування встановлено, що всі реалізовані алгоритми формують відсортований масив у зростаючому порядку.

На представлених рисунках видно, що під час виконання алгоритмів відбувається поетапна зміна стану елементів масиву, яка супроводжується відповідною кольоровою індикацією. Елементи, які порівнюються, виділяються червоним кольором, поточний або активний елемент в окремих алгоритмах – жовтим, а елементи, які вже належать до відсортованої частини масиву, – зеленим. Такий підхід дозволяє користувачу простежити логіку роботи кожного алгоритму та зрозуміти, які операції виконуються на конкретному етапі.



Рисунок 3.1 – Сортування Bubble Sort



Рисунок 3.2 – Сортування Insertion Sort



Рисунок 3.3 – Сортування Merge Sort



Рисунок 3.4 – Сортування Quick Sort



Рисунок 3.5 – Сортування Heap Sort

Аналіз результатів виконання алгоритмів сортування показав, що їхня поведінка відповідає теоретичним оцінкам складності. Наприклад, для алгоритму Bubble Sort спостерігається значна кількість операцій та порівнянь, що підтверджує квадратичну часову складність (див. рис. 3.6). У той же час алгоритми Quick Sort (див. рис. 3.7) та Merge Sort (див. рис. 3.8) демонструють значно кращі результати, що підтверджується меншою кількістю виконаних операцій та швидшим завершенням процесу.

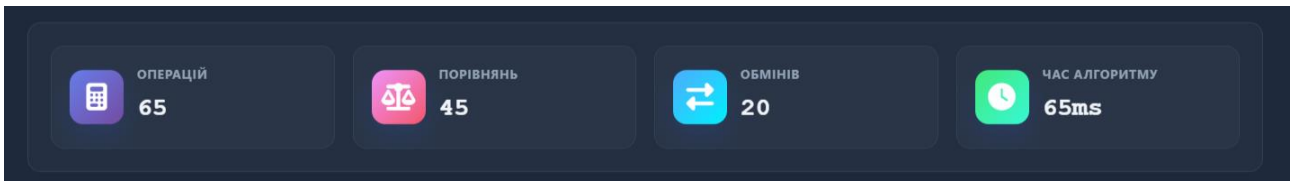


Рисунок 3.6 – Аналіз результатів Bubble Sort

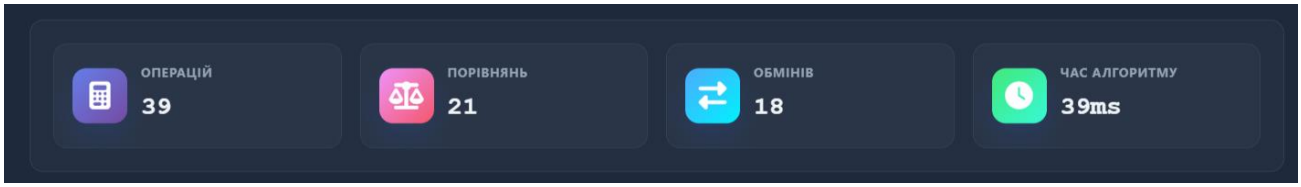


Рисунок 3.7 – Аналіз результатів Quick Sort

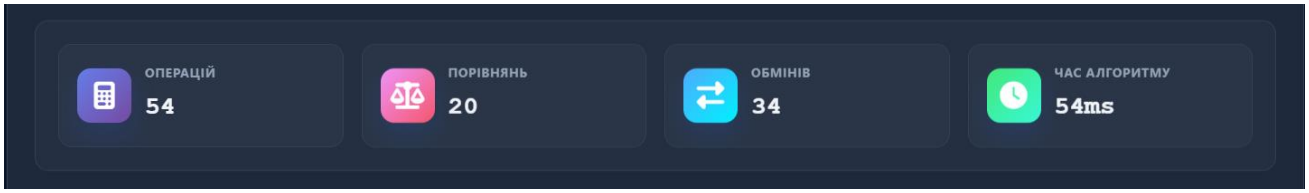


Рисунок 3.8 – Аналіз результатів Merge Sort

Тестування алгоритмів пошуку показало коректність визначення позиції шуканого елемента в масиві. У вебзастосунку реалізовано Linear Search, Binary Search, Jump Search та Interpolation Search. Для лінійного пошуку характерною є послідовна перевірка кожного елемента, що відображається поступовим переміщенням активного маркера по масиву (див. рис. 3.9). У випадку Binary Search пошук виконується у відсортованому масиві шляхом поділу діапазону навпіл, що дозволяє значно зменшити кількість перевірок (див. рис. 3.10). Jump Search демонструє пошук блоками з подальшою перевіркою всередині знайденого діапазону (див. рис. 3.11), а Interpolation Search оцінює можливу позицію шуканого елемента на основі його значення (див. рис. 3.12).



Рисунок 3.9 – Візуалізація Linear Search



Рисунок 3.10 – Візуалізація Binary Search



Рисунок 3.11 – Візуалізація Jump Search



Рисунок 3.12 – Візуалізація Interpolation Search

Окрему увагу приділено аналізу статистичних показників, які автоматично обчислюються під час виконання алгоритмів (див. рис. 3.13). До таких показників належать кількість операцій, кількість порівнянь, кількість обмінів або переміщень, а також час алгоритму, що відображається у панелі аналітики. На основі цих даних можна зробити висновок про ефективність кожного алгоритму та провести їх порівняльний аналіз. Наприклад, алгоритми з квадратичною складністю виконують більше операцій при збільшенні розміру масиву, тоді як більш оптимізовані методи завершують роботу швидше.



Рисунок 3.13 – Панель аналітики виконання алгоритму

Додатковою можливістю застосунку є порівняння двох алгоритмів між собою. На окремій сторінці порівняння користувач може вибрати два алгоритми однієї категорії, зокрема алгоритми сортування або пошуку, і запустити їх на спільному наборі даних (див. рис. 3.14). Під час порівняння система одночасно відображає дві візуалізації, фіксує кількість операцій, порівнянь, обмінів або переміщень, кількість кроків і час виконання для кожного методу. Після завершення порівняння система визначає алгоритм із меншою кількістю операцій та подає результати у вигляді таблиці й діаграм показників (див. рис. 3.15).

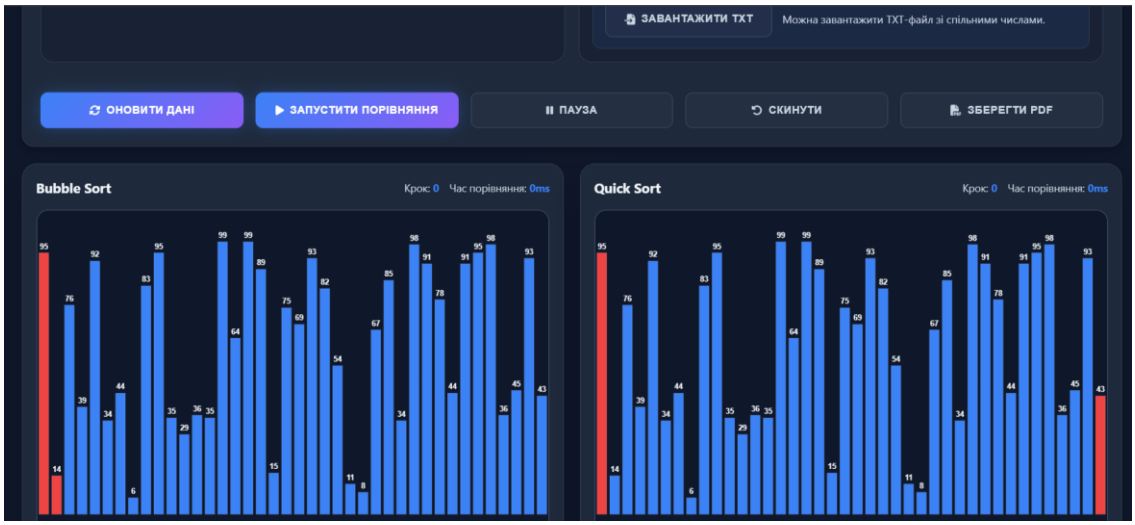


Рисунок 3.14 – Сторінка порівняння двох алгоритмів



Рисунок 3.15 – Результати порівняння алгоритмів

Окрім автоматичної генерації даних, у застосунку реалізовано можливість завантаження вхідного набору з текстового файлу. TXT-файл може містити числові значення для масиву, а також цільове значення для алгоритмів пошуку. Це дозволяє користувачу тестувати алгоритми не лише на випадкових даних, а й на власних наборах (див. рис. 3.16). Після завантаження дані використовуються для побудови візуалізації, а розмір масиву автоматично синхронізується з кількістю прочитаних чисел.

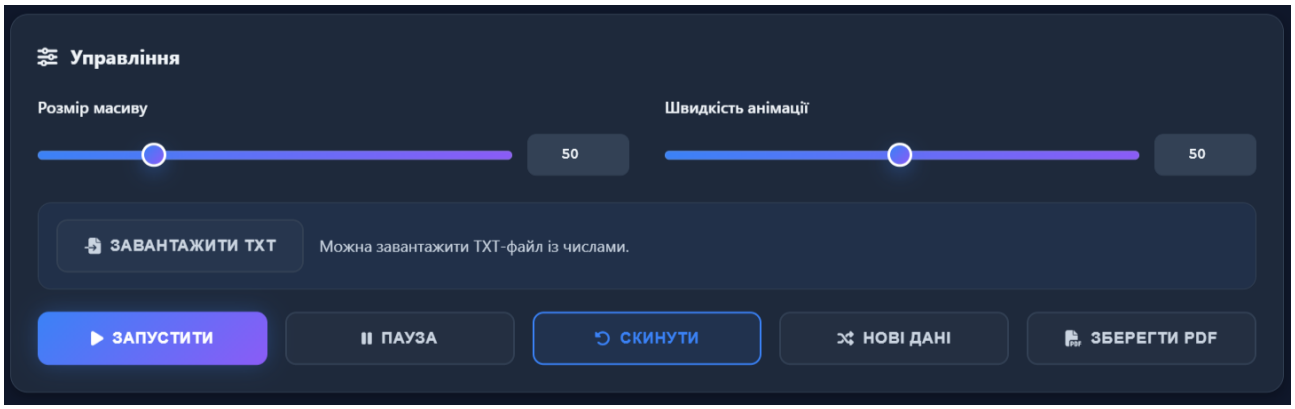


Рисунок 3.16 – Завантаження вхідних даних із TXT-файлу

Окрім перевірки алгоритмічної частини, було проведено тестування модуля візуалізації. Встановлено, що відображення даних на Canvas відбувається коректно, а анімація виконання алгоритмів є плавною. Можливість регулювання швидкості дозволяє адаптувати процес під потреби користувача: сповільнити виконання для детального аналізу окремих кроків або пришвидшити його для перегляду загального результату. Також перевірено коректність роботи елементів управління, зокрема кнопок запуску, паузи, скидання, переходу між кроками, генерації нових даних і збереження звіту.

Також проаналізовано роботу алгоритмів теорії графів, зокрема алгоритмів обходу в ширину (BFS, див. рис. 3.17), обходу в глибину (DFS, див. рис. 3.18) та алгоритму Дейкстри (див. рис. 3.19). Візуалізація цих алгоритмів представлена у вигляді графічної моделі, де вершини відображаються у вигляді вузлів, а зв’язки між ними – у вигляді ребер. Для алгоритму Дейкстри додатково відображаються ваги ребер і поточні відстані від початкової вершини.

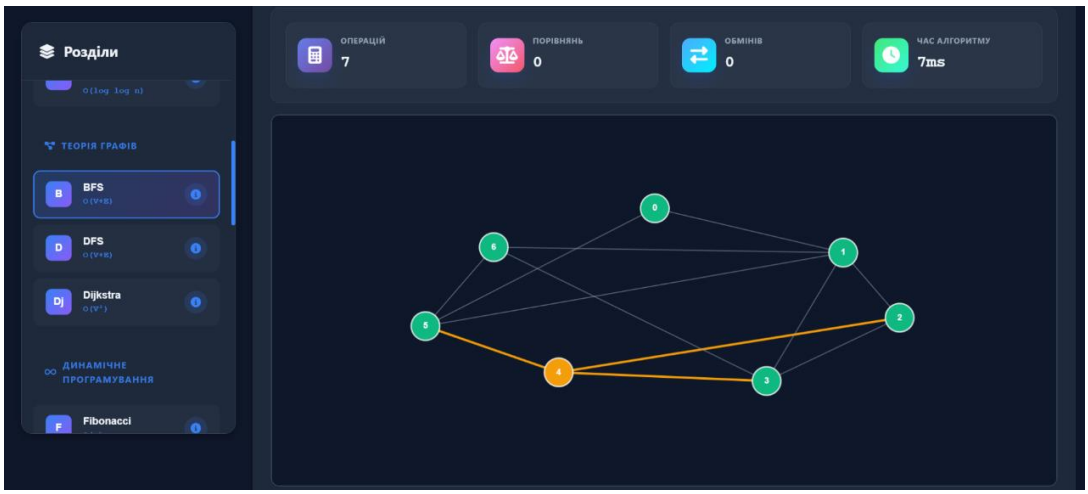


Рисунок 3.15 – Візуалізація BFS

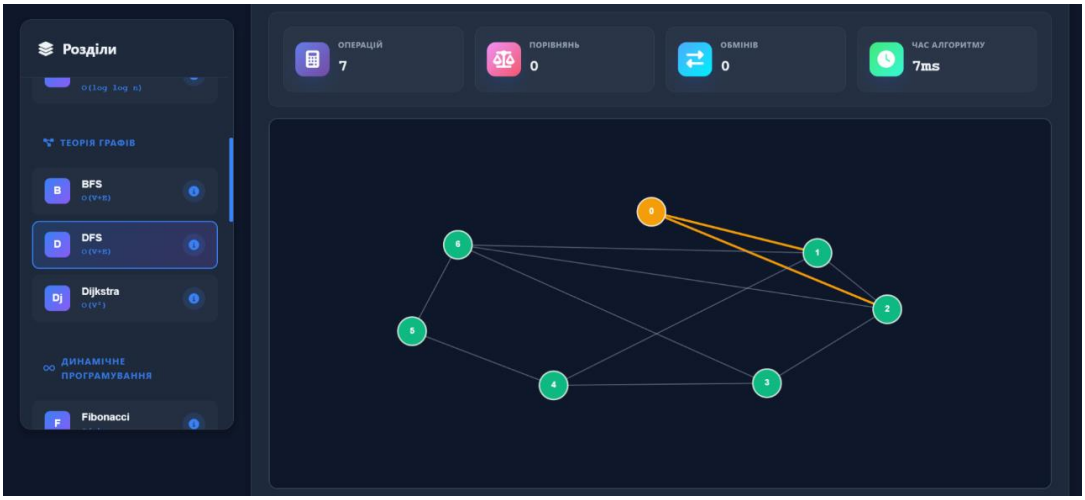


Рисунок 3.15 – Візуалізація DFS

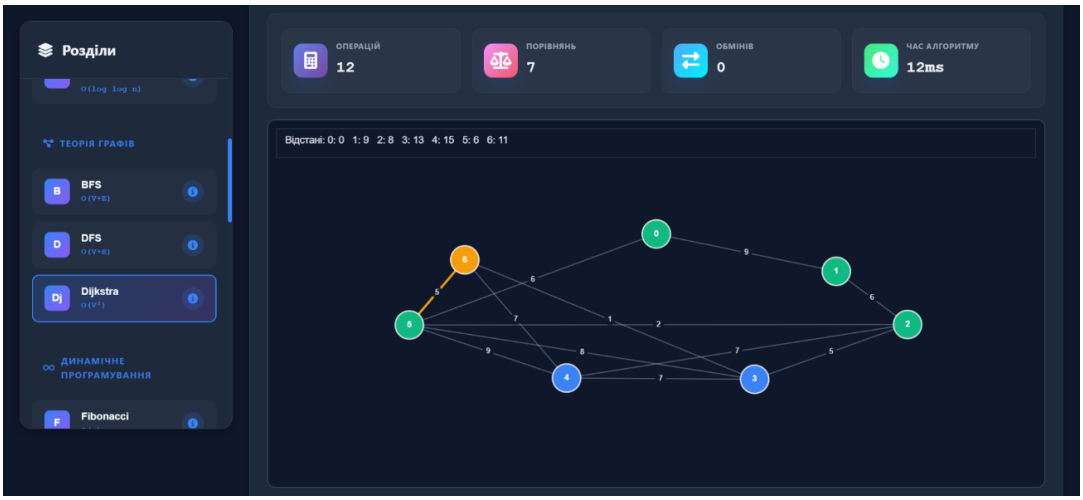


Рисунок 3.15 – Візуалізація алгоритму Дейкстри

Як видно з результатів тестування, під час виконання BFS вершини

відвідуються за рівнями, що відповідає логіці обходу в ширину. Поточна вершина та вже відвідані вузли виділяються різними кольорами, а стан черги відображається у верхній частині області візуалізації. У випадку DFS відбувається глибокий рекурсивний обхід графа, що відображається через поточну вершину, відвідані вузли та шлях обходу. Алгоритм Дейкстри демонструє поступове оновлення найкоротших відстаней і вибір наступної вершини з мінімальною поточною оцінкою.

На наступному етапі тестування було досліджено реалізацію алгоритмів динамічного програмування. Для алгоритму Фібоначчі система коректно відображає процес поступового заповнення масиву dp[] (див. рис. 3.20). Поточна комірка виділяється окремим кольором, а значення, які беруть участь у формуванні нового елемента, також підсвічуються. Наприклад, для n = 20 система відображає обчислення dp[20] = dp[19] + dp[18] = 4181 + 2584 = 6765, що відповідає теоретичному значенню двадцятого числа Фібоначчі.

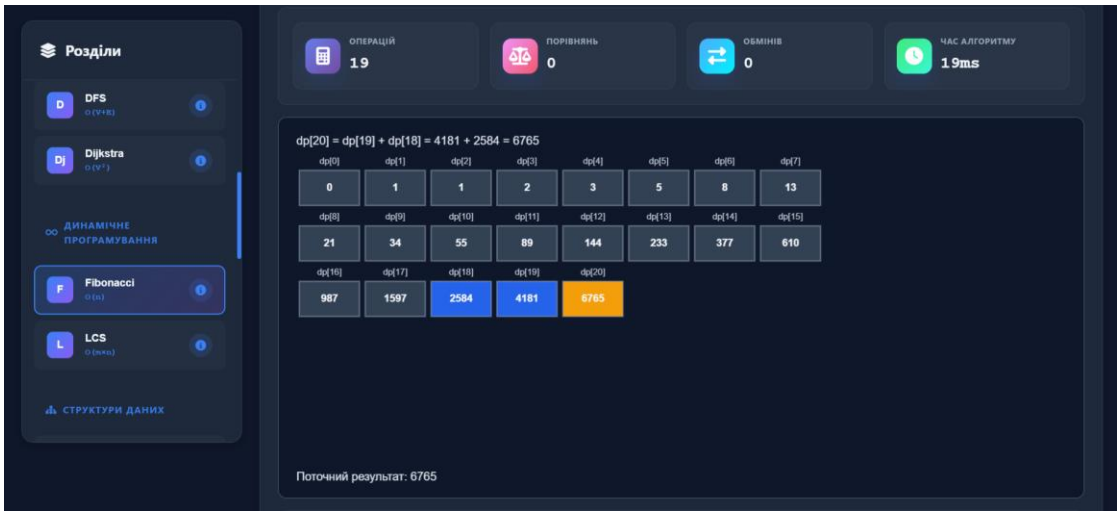


Рисунок 3.20 – Візуалізація обчислення чисел Фібоначчі

Також було перевірено роботу алгоритму пошуку найдовшої спільної підпослідовності (LCS). На візуалізації відображається матриця динамічного програмування, у якій кожна комірка містить значення для відповідних префіксів двох рядків (див. рис. 3.21). Під час виконання алгоритму система підсвічує поточну комірку та комірки, значення яких використовуються для обчислення. У фінальному стані відображається знайдена найдовша спільна

підпослідовність, що підтверджує правильність реалізації алгоритму.

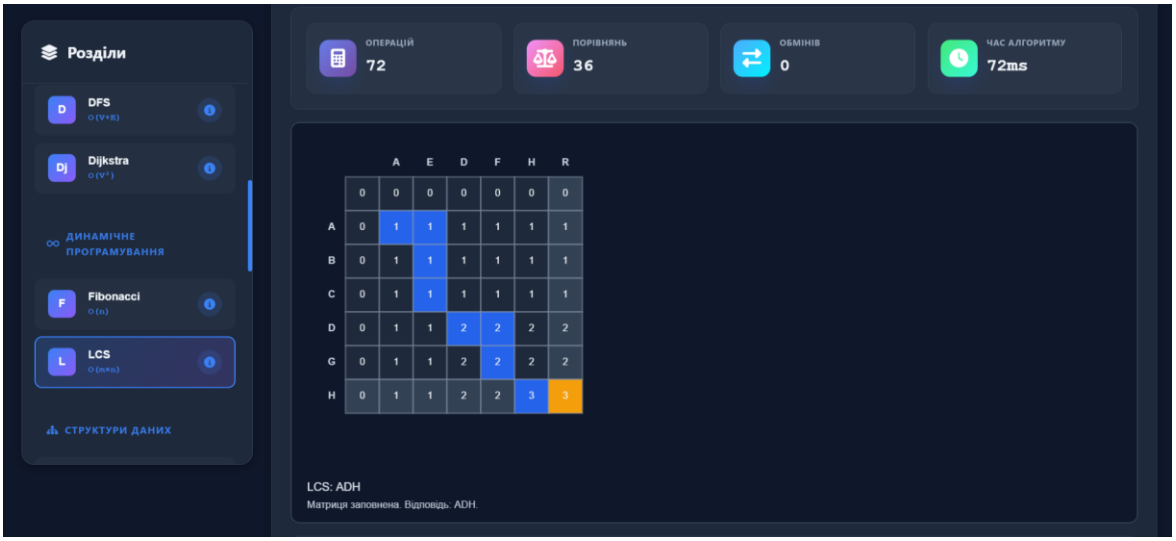


Рисунок 3.21 – Візуалізація алгоритму LCS

У процесі тестування також було перевірено розділ структур даних (див. рис. 3.22). Він реалізований як теоретичний навчальний модуль, що пояснює основні структури даних, їх класифікацію, особливості зберігання в пам’яті, складність операцій та зв’язок із візуалізованими алгоритмами. Було встановлено, що навігація між підрозділами працює коректно, активний пункт меню підсвічується, а матеріал відображається без порушень структури сторінки.

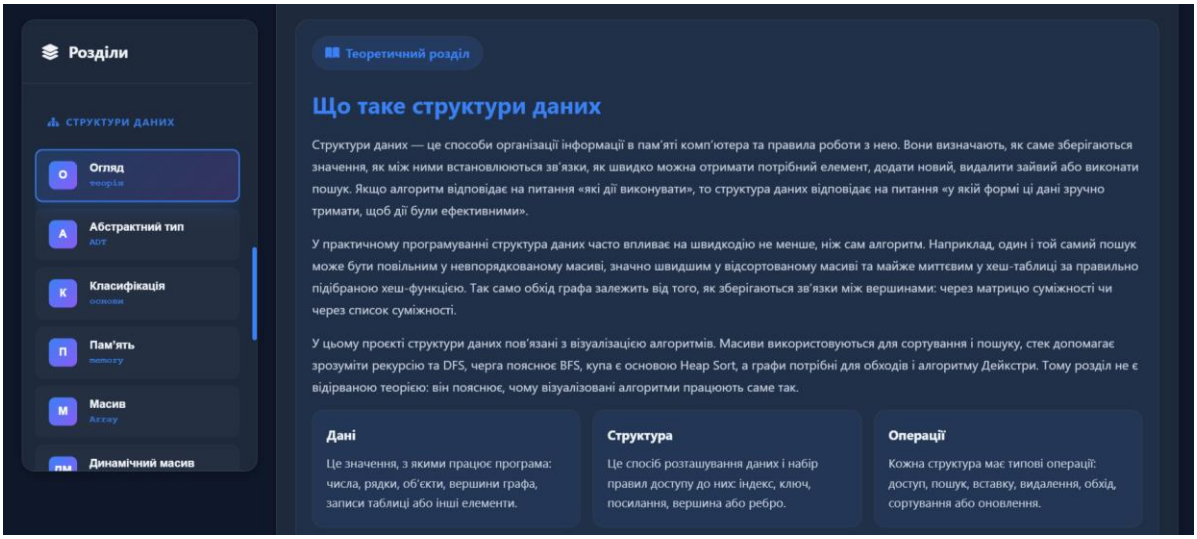


Рисунок 3.22 – Теоретичний розділ структур даних

Важливою функцією застосунку є можливість формування звітів за результатами виконання алгоритмів. Після запуску або завершення алгоритму користувач може зберегти звіт у форматі PDF (див. рис. 3.23). У звіт додаються назва алгоритму, його опис і складність, налаштування, початкові дані, візуальні знімки початкового та поточного або фінального стану, а також статистичні показники. Окремо реалізовано збереження звіту для сторінки порівняння двох алгоритмів, де фіксуються результати обох методів (див. рис. 3.24).

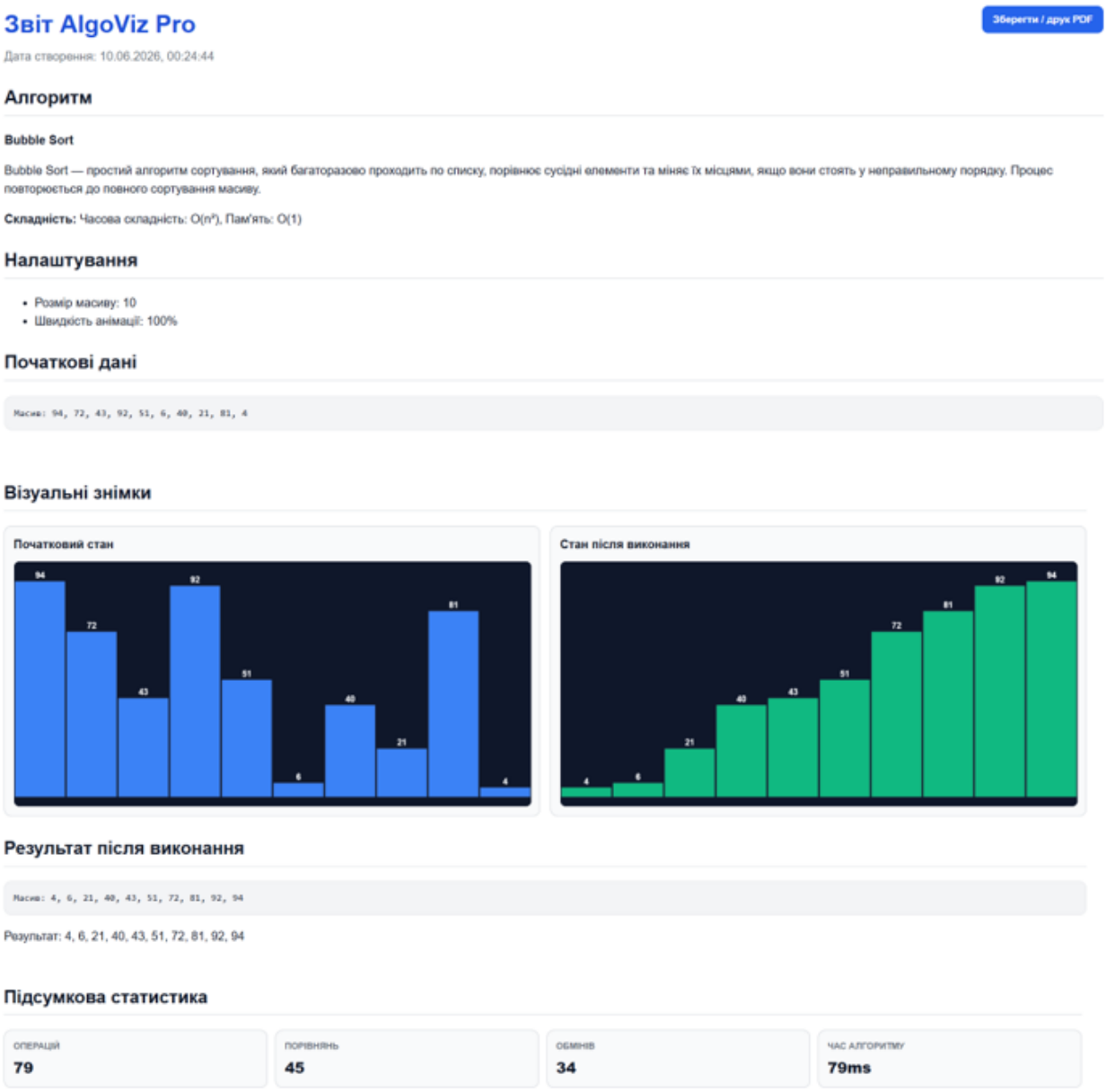


Рисунок 3.23 – Формування PDF-звіту для одного алгоритму

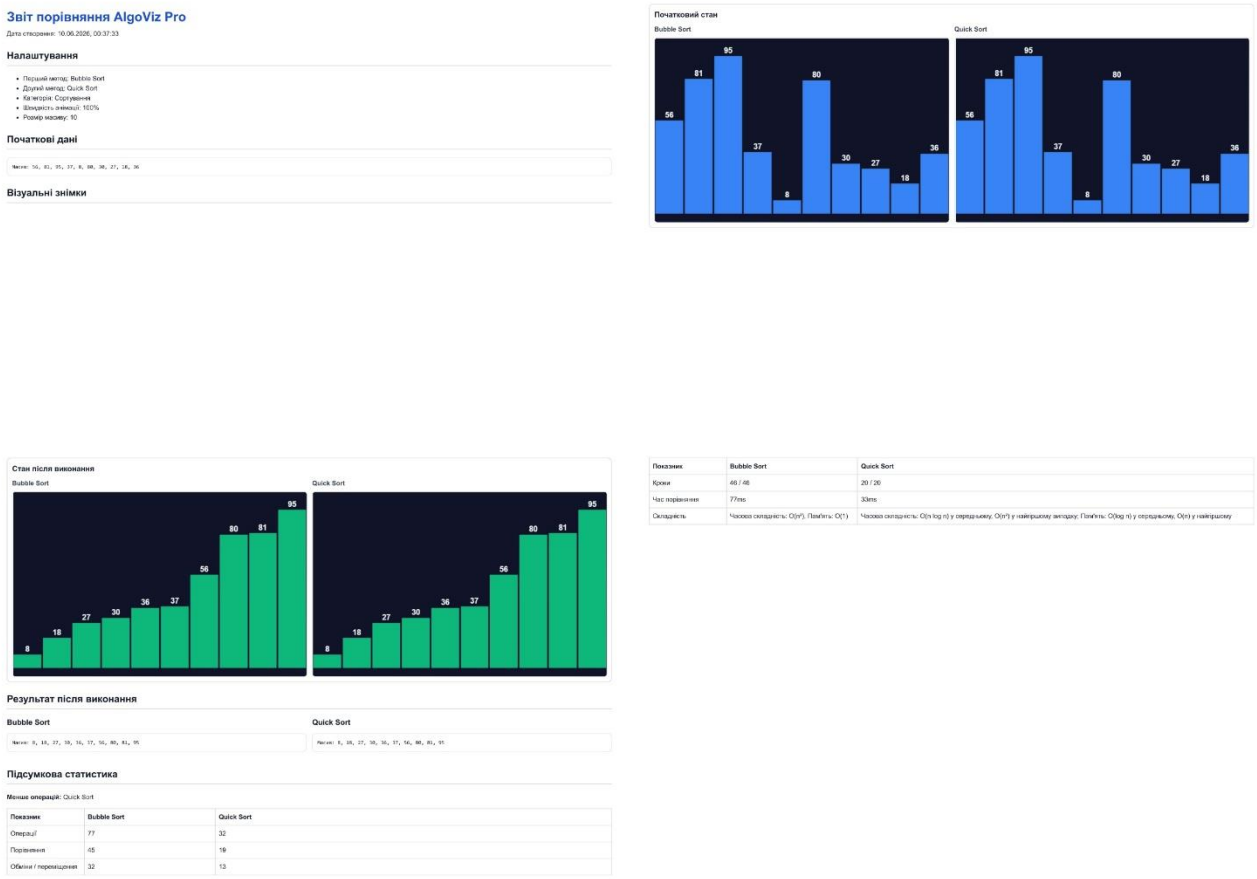


Рисунок 3.24 – Формування PDF-звіту порівняння алгоритмів

Система коректно відображає статистичні показники виконання алгоритмів, зокрема кількість операцій, порівнянь, обмінів або переміщень та час алгоритму. Це дозволяє користувачу не лише візуально спостерігати процес, але й проводити кількісний аналіз ефективності алгоритмів.

Легенда кольорів пояснює значення кожного візуального маркера. Завдяки цьому користувач може легко інтерпретувати стан елементів у процесі виконання алгоритму. Проведене тестування показало, що кольорові позначення працюють коректно та відповідають заданій логіці: необроблені елементи, елементи порівняння, поточний елемент, готові або відвідані елементи мають окреме візуальне представлення.

Було перевірено адаптивність інтерфейсу при зміні параметрів, зокрема розміру масиву, швидкості анімації та кількості вершин графа. Збільшення розміру вхідних даних не призводить до критичних затримок у роботі системи, що свідчить про достатню оптимізацію реалізації для навчального

вебзастосунку. При зміні розміру вікна браузера полотно автоматично масштабується, а поточний стан візуалізації перемальовується.

Отримані результати свідчать про коректну роботу реалізованого вебзастосунку. Система правильно виконує алгоритми сортування, пошуку, теорії графів і динамічного програмування, забезпечує наочну покрокову візуалізацію, підтримує завантаження користувацьких даних, порівняння алгоритмів, збереження звітів і перегляд теоретичного розділу структур даних. Це підтверджує відповідність розробленого продукту поставленій меті та його придатність для навчального використання.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було досягнуто поставленої мети, яка полягала у розробці вебзастосунку для візуалізації структур даних та алгоритмів. Створений програмний продукт забезпечує інтерактивне, наочне та зрозуміле відображення алгоритмічних процесів, дозволяє користувачу спостерігати за покроковим виконанням алгоритмів, аналізувати проміжні стани даних і порівнювати ефективність різних методів.

У ході дослідження було опрацьовано теоретичні основи алгоритмів і структур даних, визначено їх сутність, класифікацію та роль у сучасних програмних системах. Це дозволило сформувати цілісне уявлення про предметну область та обґрунтувати необхідність застосування засобів візуалізації для кращого розуміння принципів роботи алгоритмів. Окрему увагу було приділено структурам даних, оскільки саме спосіб організації даних значною мірою впливає на ефективність алгоритмічних рішень.

Проведений аналіз існуючих програмних рішень і вебсервісів показав, що хоча існує значна кількість інструментів для візуалізації алгоритмів, частина з них має обмежену функціональність, недостатній рівень інтерактивності або складний інтерфейс для початківців. Це підтвердило доцільність створення власного вебзастосунку, який поєднує зручність, наочність, можливість налаштування параметрів і підтримку різних груп алгоритмів.

На етапі проєктування було сформульовано вимоги до системи, обґрунтовано вибір вебтехнологій HTML, CSS та JavaScript, а також розроблено архітектуру застосунку. Структура проєкту передбачає поділ на окремі модулі: базовий клас алгоритму, класи конкретних алгоритмів, модуль статистики, модуль візуалізації, модуль завантаження даних, сторінку порівняння та компоненти інтерфейсу. Такий підхід забезпечив модульність, зрозумілість коду та можливість подальшого розширення функціональності.

У процесі реалізації було створено алгоритми сортування, пошуку, теорії

графів і динамічного програмування. Зокрема, реалізовано Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Linear Search, Binary Search, Jump Search, Interpolation Search, BFS, DFS, алгоритм Дейкстри, обчислення чисел Фібоначчі та алгоритм LCS. Кожен алгоритм формує послідовність кроків виконання, що дозволяє відображати не лише фінальний результат, а й проміжні етапи роботи.

Особливу увагу було приділено розробці модуля візуалізації, реалізованого з використанням HTML5 Canvas. Для масивів застосовано подання у вигляді стовпчикової діаграми, для графів – відображення вершин і ребер, для динамічного програмування – візуалізацію масиву dp[] та матриці. Використання кольорового виділення дозволяє позначати поточні елементи, порівняння, відвідані вершини, активні діапазони та готові результати.

Розроблений інтерфейс користувача забезпечує вибір алгоритму, запуск, паузу, скидання, покрокове виконання, зміну швидкості анімації, налаштування розміру масиву, параметрів пошуку, кількості вершин графа та вхідних значень для алгоритмів динамічного програмування. Додатково реалізовано можливість завантаження користувацьких даних із TXT-файлу, що розширює можливості тестування алгоритмів на власних наборах даних.

Важливою функціональністю застосунку стала сторінка порівняння двох алгоритмів. Вона дозволяє запускати два алгоритми однієї категорії на спільному наборі даних і порівнювати їх за кількістю операцій, порівнянь, обмінів або переміщень, кількістю кроків і часом виконання. Це робить застосунок корисним не лише для демонстрації алгоритмів, а й для аналізу їх ефективності.

Окремо було реалізовано теоретичний розділ структур даних, у якому розглянуто масиви, динамічні масиви, списки, стек, чергу, дек, хеш-таблицю, дерева, бінарне дерево пошуку, купу та граф. Цей розділ доповнює практичну частину застосунку та пояснює, чому ті чи інші структури використовуються в реалізованих алгоритмах.

Результати тестування підтвердили коректність реалізації алгоритмів і стабільність роботи вебзастосунку. Було перевірено виконання алгоритмів на різних наборах вхідних даних, зокрема автоматично згенерованих і

завантажених із текстового файлу. Аналіз статистичних показників, таких як кількість операцій, порівнянь, обмінів або переміщень та час виконання, показав відповідність отриманих результатів теоретичним оцінкам складності алгоритмів.

Також було перевірено роботу модуля візуалізації, адаптивність інтерфейсу, коректність кольорових позначень, функціонування панелі аналітики, сторінки порівняння та механізму формування PDF-звітів. Можливість збереження звітів дозволяє фіксувати результати виконання алгоритмів, початкові дані, налаштування, візуальні знімки та статистичні показники.

Отже, розроблений вебзастосунок відповідає поставленим вимогам і може бути використаний як навчальний інструмент для вивчення алгоритмів і структур даних. Він поєднує теоретичний матеріал, інтерактивну візуалізацію, статистичний аналіз, порівняння алгоритмів і можливість збереження результатів, що робить його зручним засобом для кращого розуміння алгоритмічних процесів.

ПЕРЕЛІК ПОСИЛАНЬ

1. З чого складається алгоритм? URL : https://dovidka.biz.ua/z-chogo-skladayetsya-algoritm/ (дата звернення : 10.04.2026).

2. Що таке алгоритми : кроки, приклади, конструкції, мислення. URL : https://dan-it.com.ua/uk/blog/shho-take-algorytmy-kroky-pryklady-konstrukcziyi-myslennya/ (дата звернення : 10.04.2026).

3. Про алгоритми сортування. URL : https://foxminded.ua/alhorytmy-sortuvannia/ (дата звернення : 10.04.2026).

4. Аушева Н. М., Баган Т. Г. Алгоритми та структури даних : навч. посіб. Київ : НТУУ «КПІ імені Ігоря Сікорського», 2022. 215 с.

5. Федорін І. В. Проектування та аналіз обчислювальних алгоритмів: Структури даних : навч. посіб. для студ. спеціальності 122 «Комп'ютерні науки». Київ : КПІ ім. Ігоря Сікорського, 2022. 173 с.

6. Метод динамічного програмування: ключові аспекти та застосування. URL : https://foxminded.ua/metod-dynamichnoho-prohramuvannia/ (дата звернення : 10.04.2026).

7. Алгоритми та методи обчислень : навч. посіб. для виконання лабораторних робіт за освітньо-професійною програмою «Системне програмування та спеціалізовані комп'ютерні системи» / уклад. Ю. Є. Боярінова, О. О. Кучмій, О. П. Мартинова. Київ : КПІ ім. Ігоря Сікорського, 2025. 123 с.

8. Складність алгоритмів. URL : https://javarush.com/ua/groups/posts/uk.2325.skladnstjh-algoritmv (дата звернення: 10.04.2026).

9. Сафронова О. О., Донець К. В. Основи двовимірної комп'ютерної графіки : навч. посіб. Київ : КНУТД, 2016. 175 с.

10. Актуальні питання сучасної інформатики : матеріали доповідей IV Всеукр. наук.-практ. конф. з міжнар. участю «Сучасні інформаційні технології в освіті та науці» (Житомир, 07–08 листопада 2019 р.) / за заг. ред. Я. Б. Сікори. Житомир : Вид-во ЖДУ, 2019. Вип. 7. 207 с.

11. Ільченко Д. С. Вебзастосунок для зміни стилю зображень із

застосуванням нейронних мереж : дипломний проєкт ... бакалавра : 121 Інженерія програмного забезпечення. Київ, 2025. 108 с.

12. Algorithm Visualizer. URL : https://algorithm-visualizer.org/ (дата звернення : 10.04.2026).

13. VisuAlgo. URL : https://visualgo.net/en (дата звернення : 10.04.2026).

14. AlgoVis. URL : https://algovis.website/ (дата звернення : 10.04.2026).

15. DSA Visualizer. URL : https://www.dsavisualizer.in/ (дата звернення : 10.04.2026).

16. Gephi та Cytoscape. URL : https://prf1.org/docs/gephi_cytoscape_cafephe.pdf (дата звернення : 10.04.2026).

17. NetworkX. URL : https://networkx.org/en/ (дата звернення : 10.04.2026).

18. Готинчан Т. І. Основи веброзробки: HTML і CSS. Частина 1 : навч. посіб. Чернівці : Чернівецький національний університет, 2023. 208 с.

19. Основи JavaScript. URL : https://uk.javascript.info/first-steps (дата звернення : 10.04.2026).

ДОДАТОК А

Вихідний код HTML-сторінок

А.1 Головна сторінка (index.html)

```
<!doctype html>
<html lang="uk">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>AlgoViz Pro – Візуалізація алгоритмів</title>
    <link
      rel="stylesheet"
      href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/css/all.min.css" />
    <script id="algoviz-theme-boot">
      (function () {
        try {
          var saved = localStorage.getItem("algoviz-theme") || "dark";
          document.documentElement.classList.add(saved === "light" ? "light-theme" : "dark-theme");
        } catch (e) {
          document.documentElement.classList.add("dark-theme");
        }
      })();
    </script>
    <link rel="stylesheet" href="styles.css" />
    <link rel="stylesheet" href="assets/css/accessibility-fixes.css" />
  </head>
  <body>
    <div class="app-wrapper">
      <nav class="navbar">
        <div class="navbar-container">
          <div class="navbar-brand">
            <i class="fas fa-code"></i>
            <span>AlgoViz Pro</span>
          </div>
          <div class="navbar-info">
            <div class="top-tabs">
              <a class="nav-link active" href="index.html"><i class="fas fa-chart-bar"></i>
Візуалізація</a>
              <a class="nav-link" href="compare.html"><i class="fas fa-code-compare"></i>
Порівняння</a>
            </div>
            <button type="button" class="theme-toggle" id="themeToggle" aria-label="Перемкнути
тему" title="Перемкнути тему"><i class="fas fa-moon"></i></button>
          </div>
        </div>
```

```
</nav>

<div class="main-scroll">
 <div class="main-grid">
  <aside class="sidebar-panel">
   <div class="sidebar-header">
    <h2><i class="fas fa-layer-group"></i> Розділи</h2>
   </div>
   <div id="algorithmMenu"></div>
  </aside>
  <main class="content-panel">
   <section class="control-section">
    <div class="control-header">
     <h3><i class="fas fa-sliders-h"></i> Управління</h3>
    </div>
    <div class="control-grid">
     <div class="control-item" id="arraySizeControl">
      <label class="control-label" for="sizeInput">
       <span class="label-text">Розмір масиву</span>
      </label>
      <div class="range-input-row">
       <input
        type="range"
        id="sizeInput"
        min="5"
        max="200"
        value="50"
        class="slider" />
       <input
        type="number"
        id="sizeNumber"
        min="5"
        max="200"
        value="50"
        class="manual-value-input"
        aria-label="Ввести розмір масиву вручну" />
      </div>
     </div>
     <div class="control-item">
      <label class="control-label" for="speedInput">
       <span class="label-text">Швидкість анімації</span>
      </label>

      <div class="range-input-row">
       <input
        type="range"
        id="speedInput"
        min="1"
        max="100"
        value="50"
        class="slider" />
       <input
```

```
                type="number"
                id="speedNumber"
                min="1"
                max="100"
                value="50"
                class="manual-value-input"
                aria-label="Ввести швидкість анімації вручну" />
              </div>
            </div>
          </div>
          <div
            class="settings-section control-settings"
            id="algorithmSettings"
            style="display: none">
            <h4>Налаштування алгоритму</h4>
            <div id="settingsContent"></div>
          </div>

          <div class="data-import-control" id="dataImportControl">
            <input type="file" id="dataFileInput" accept=".txt,text/plain" hidden />
            <button type="button" id="importDataButton" class="btn btn-secondary data-import-
button">
              <i class="fas fa-file-import"></i> Завантажити TXT
            </button>
            <span id="dataImportStatus" class="data-import-status">Можна завантажити TXT-
файл із числами.</span>
          </div>
          <div class="button-group">
            <button
              type="button"
              id="startAnimationButton"
              class="btn btn-primary">
              <i class="fas fa-play"></i> Запустити
            </button>
            <button
              type="button"
              id="pauseAnimationButton"
              class="btn btn-secondary">
              <i class="fas fa-pause"></i> Пауза
            </button>
            <button
              type="button"
              id="resetVisualizationButton"
              class="btn btn-secondary">
              <i class="fas fa-rotate-left"></i> Скинути
            </button>
            <button
              type="button"
              id="newDataButton"
              class="btn btn-secondary">
              <i class="fas fa-random"></i> Нові дані
            </button>
```

```
    <button
      type="button"
      id="exportReportButton"
      class="btn btn-secondary">
      <i class="fas fa-file-pdf"></i> Зберегти PDF
    </button>
  </div>
</section>
<section class="visualization-section">
  <div class="visualization-header">
    <h3><i class="fas fa-chart-bar"></i> Візуалізація</h3>
  </div>
  <div class="visualization-stats-section">
    <div class="statistics-dashboard">
                <div class="stat-item">
                  <div class="stat-icon operations">
                    <i class="fas fa-calculator"></i>
                  </div>
                  <div class="stat-content">
                    <span class="stat-title">Операцій</span>
                    <span class="stat-number" id="operationCount">0</span>
                  </div>
                </div>
                <div class="stat-item">
                  <div class="stat-icon comparisons">
                    <i class="fas fa-scale-balanced"></i>
                  </div>
                  <div class="stat-content">
                    <span class="stat-title">Порівнянь</span>
                    <span class="stat-number" id="comparisonCount">0</span>
                  </div>
                </div>

                <div class="stat-item">
                  <div class="stat-icon swaps">
                    <i class="fas fa-right-left"></i>
                  </div>
                  <div class="stat-content">
                    <span class="stat-title">Обмінів</span>
                    <span class="stat-number" id="swapCount">0</span>
                  </div>
                </div>

                <div class="stat-item">
                  <div class="stat-icon time">
                    <i class="fas fa-clock"></i>
                  </div>
                  <div class="stat-content">
                    <span class="stat-title">Час алгоритму</span>
                    <span class="stat-number" id="timeElapsed">0ms</span>
                  </div>
                </div>
```

```
            </div>
  </div>
  <div class="canvas-wrapper">
    <canvas id="canvas"></canvas>
  </div>
  <div class="legend-section">
    <h4>Легенда кольорів</h4>
    <div class="legend-items">
      <div class="legend-item">
        <div class="legend-color legend-color-default"></div>
        <span>Не оброблені</span>
      </div>
      <div class="legend-item">
        <div class="legend-color legend-color-compare"></div>
        <span>Порівняння</span>
      </div>
      <div class="legend-item">
        <div class="legend-color legend-color-current"></div>
        <span>Поточний</span>
      </div>
      <div class="legend-item">
        <div class="legend-color legend-color-complete"></div>
        <span>Готові / відвідані</span>
      </div>
    </div>
  </div>
</section>

<section class="info-section">
  <div class="info-header">
    <h3><i class="fas fa-info-circle"></i> Аналіз</h3>
  </div>
  <div class="algorithm-details">
    <h4 id="algorithmName">Виберіть алгоритм</h4>
    <p id="algorithmDesc" class="algo-description"></p>
    <div class="complexity-box">
      <span class="complexity-label">Складність:</span>
      <span class="complexity-value" id="algorithmComplexity">
        —
      </span>
    </div>
  </div>
  <div
    class="instruction-section"
    id="algorithmInstruction"
    style="display: none">
    <h4>Інструкція</h4>
    <ul id="algorithmSteps"></ul>
  </div>
</section>
<section class="data-structures-section is-hidden" id="dataStructuresSection">
  <div class="info-header data-structures-title">
```

```
        <h3><i class="fas fa-sitemap"></i> Структури даних</h3>
      </div>
      <div class="data-structures-intro" id="ds-overview">
        <div class="page-badge">
          <i class="fas fa-book-open"></i>
          Теоретичний розділ
        </div>
        <h2>Що таке структури даних</h2>
        <p>
          Структури даних – це способи організації інформації в пам’яті комп’ютера та правила роботи з нею. Вони визначають, як саме зберігаються значення, як між ними встановлюються зв’язки, як швидко можна отримати потрібний елемент, додати новий, видалити зайвий або виконати пошук. Якщо алгоритм відповідає на питання «які дії виконувати», то структура даних відповідає на питання «у якій формі ці дані зручно тримати, щоб дії були ефективними».
        </p>
        <p>
          У практичному програмуванні структура даних часто впливає на швидкодію не менше, ніж сам алгоритм. Наприклад, один і той самий пошук може бути повільним у невпорядкованому масиві, значно швидшим у відсортованому масиві та майже миттєвим у хеш-таблиці за правильно підібраною хеш-функцією. Так само обхід графа залежить від того, як зберігаються зв’язки між вершинами: через матрицю суміжності чи через список суміжності.
        </p>
        <p>
          У цьому проєкті структури даних пов’язані з візуалізацією алгоритмів. Масиви використовуються для сортування і пошуку, стек допомагає зрозуміти рекурсію та DFS, черга пояснює BFS, купа є основою Heap Sort, а графи потрібні для обходів і алгоритму Дейкстри. Тому розділ не є відірваною теорією: він пояснює, чому візуалізовані алгоритми працюють саме так.
        </p>
        <div class="theory-grid compact-theory-grid">
          <article class="theory-card">
            <h3>Дані</h3>
            <p>Це значення, з якими працює програма: числа, рядки, об’єкти, вершини графа, записи таблиці або інші елементи.</p>
          </article>
          <article class="theory-card">
            <h3>Структура</h3>
            <p>Це спосіб розташування даних і набір правил доступу до них: індекс, ключ, посилання, вершина або ребро.</p>
          </article>
          <article class="theory-card">
            <h3>Операції</h3>
            <p>Кожна структура має типові операції: доступ, пошук, вставку, видалення, обхід, сортування або оновлення.</p>
          </article>
          <article class="theory-card">
            <h3>Складність</h3>
            <p>Ефективність оцінюють за часом і пам’яттю, найчастіше через нотацію O(1), O(log n), O(n), O(n log n), O(n²).</p>
          </article>
```

```
      </div>
    </div>
    <article class="data-topic" id="ds-adt">
      <h2>Абстрактний тип даних і реалізація</h2>
      <p>
        Важливо розрізняти абстрактний тип даних і конкретну реалізацію. Абстрактний тип даних описує, які операції доступні користувачу структури, але не зобов’язує зберігати дані певним способом. Наприклад, стек як абстрактний тип має операції push, pop і peek. Але реалізувати його можна на масиві, динамічному масиві або зв’язному списку.
      </p>
      <p>
        Реалізація визначає реальну швидкість операцій, витрати пам’яті та зручність підтримки коду. Дві програми можуть використовувати однакову логічну структуру, але мати різну продуктивність через різні способи зберігання. Саме тому під час проєктування треба спочатку зрозуміти поведінку структури, а потім обрати реалізацію відповідно до задачі.
      </p>
      <div class="theory-table-wrap">
        <table class="theory-table">
          <thead>
            <tr><th>Рівень</th><th>Що описує</th><th>Приклад</th></tr>
          </thead>
          <tbody>
            <tr><td>Абстрактний тип</td><td>Набір операцій і логіку використання</td><td>Стек: додати вгору, забрати зверху</td></tr>
            <tr><td>Структура даних</td><td>Організацію елементів і зв’язків</td><td>Масив, список, дерево, граф</td></tr>
            <tr><td>Реалізація</td><td>Конкретний код і спосіб розміщення в пам’яті</td><td>Стек на масиві або стек на списку</td></tr>
          </tbody>
        </table>
      </div>
    </article>
    <article class="data-topic" id="ds-classification">
      <h2>Класифікація структур даних</h2>
      <p>
        Структури даних класифікують за кількома ознаками. Лінійні структури зберігають елементи у вигляді послідовності: масив, список, стек, черга, дек. Нелінійні структури мають складніші зв’язки: дерева, купи, графи. За способом зміни розміру структури поділяють на статичні та динамічні. Статична структура має фіксовані межі, а динамічна може збільшуватися або зменшуватися під час виконання програми.
      </p>
      <p>
        Ще один важливий поділ – фізичне та логічне подання. Фізично купа часто зберігається у масиві, але логічно її розглядають як дерево. Граф логічно складається з вершин і ребер, але в пам’яті може бути поданий матрицею або списками суміжності. Через це в навчанні важливо не тільки знати назву структури, а й розуміти, як вона поводиться у пам’яті та які операції робить швидкими.
      </p>
      <div class="theory-table-wrap">
        <table class="theory-table">
          <thead>
```

```
            <tr><th>Ознака</th><th>Види</th><th>Пояснення</th></tr>
          </thead>
          <tbody>
            <tr><td>За організацією</td><td>Лінійні, нелінійні</td><td>Лінійні мають послідовний порядок, нелінійні – ієрархічні або мережеві зв’язки.</td></tr>
            <tr><td>За розміром</td><td>Статичні, динамічні</td><td>Статичні мають обмежений розмір, динамічні можуть змінюватися під час роботи.</td></tr>
            <tr><td>За способом доступу</td><td>Індекс, ключ, посилання, вершина</td><td>Доступ може виконуватися за позицією, ключем, адресою вузла або зв’язками графа.</td></tr>
            <tr><td>За впорядкованістю</td><td>Впорядковані, невпорядковані</td><td>Впорядкованість дозволяє застосовувати ефективніші алгоритми пошуку.</td></tr>
            <tr><td>За однорідністю</td><td>Однорідні, неоднорідні</td><td>Масив чисел є однорідним, а запис студента з полями ПІБ, група, оцінки – неоднорідним.</td></tr>
          </tbody>
        </table>
      </div>
    </article>

    <article class="data-topic" id="ds-memory">
      <h2>Пам’ять, посилання та ціна операцій</h2>
      <p>
        Дані в програмі займають пам’ять. Частина структур зберігає тільки значення, а частина додатково зберігає службову інформацію: посилання на наступний вузол, індекс батька, вагу ребра, ключ, хеш або прапорець стану. Через це структура, яка швидко виконує одну операцію, може потребувати більше пам’яті або мати складнішу реалізацію.
      </p>
      <p>
        Масив ефективний завдяки компактному розміщенню елементів поруч. Процесор швидко проходить такі дані послідовно. Зв’язний список гнучкіший під час вставок, але кожен вузол має додаткове посилання, а елементи можуть лежати в різних місцях пам’яті. Хеш-таблиця швидко знаходить значення за ключем, але потребує резервного простору, щоб не заповнювати таблицю повністю.
      </p>
      <div class="example-box">
        <strong>Головна ідея:</strong> не існує універсальної структури, яка найкраща для всіх задач. Правильний вибір залежить від того, що програма робить частіше: читає за індексом, шукає за ключем, вставляє в середину, видаляє з країв, обходить граф або підтримує пріоритети.
      </div>
    </article>
    <article class="data-topic" id="ds-array">
      <h2>Масив</h2>
      <p>
        Масив – це структура даних, у якій елементи зберігаються послідовно та мають індекси. Індекс дозволяє швидко звертатися до конкретної позиції, тому масив зручний у задачах, де потрібно часто читати елементи, проходити їх у циклі, сортувати або виконувати пошук. У цьому проєкті масиви показані у вигляді стовпчиків: число визначає висоту стовпчика, а колір показує поточну дію алгоритму.
      </p>
      <p>
```

Головна перевага масиву – доступ за індексом за O(1). Якщо відомо, що потрібний елемент стоїть на позиції i, програма може одразу перейти до цієї комірки. Недолік полягає в тому, що вставка або видалення всередині масиву часто потребує зсуву інших елементів. Для невідсортованого масиву пошук зазвичай виконується за O(n), але для відсортованого можна застосувати Binary Search, Jump Search або Interpolation Search.

```
</p>
<div class="theory-grid">
 <article class="theory-card">
  <h3>Переваги</h3>
  <ul>
   <li>просте подання даних;</li>
   <li>швидкий доступ за індексом;</li>
   <li>зручний послідовний прохід;</li>
   <li>добре підходить для сортування;</li>
   <li>ефективне використання кешу процесора.</li>
  </ul>
 </article>
 <article class="theory-card">
  <h3>Недоліки</h3>
  <ul>
   <li>вставка в середину потребує зсуву;</li>
   <li>видалення з середини також може бути дорогим;</li>
   <li>без сортування пошук часто лінійний;</li>
   <li>при фіксованому розмірі потрібно заздалегідь знати місткість.</li>
  </ul>
 </article>
</div>
<div class="example-box">
 <strong>Зв’язок з проєктом:</strong> Bubble Sort, Insertion Sort, Merge Sort, Quick Sort і Heap Sort працюють з масивом, але по-різному змінюють порядок його елементів. Пошукові алгоритми також використовують масив, однак деякі з них вимагають попереднього сортування.
</div>
</article>
<article class="data-topic" id="ds-dynamic-array">
 <h2>Динамічний масив</h2>
 <p>
  Динамічний масив схожий на звичайний масив, але може збільшувати свою місткість. Коли місця стає недостатньо, створюється новий більший блок пам’яті, старі елементи копіюються в нього, а новий елемент додається в кінець. Такий підхід використовується у багатьох мовах програмування: наприклад, у JavaScript масиви поводяться як динамічні структури.
 </p>
 <p>
  Додавання в кінець у середньому виконується швидко, але іноді відбувається дорогий момент розширення. Тому говорять про амортизовану складність O(1): більшість додавань дешеві, а рідкісні дорогі копіювання розподіляються на багато операцій. У навчальних задачах це допомагає пояснити, чому одна операція може бути дорогою, але серія операцій залишається ефективною.
 </p>
 <div class="theory-table-wrap">
  <table class="theory-table">
```

```
            <thead>
              <tr><th>Операція</th><th>Звичайна оцінка</th><th>Пояснення</th></tr>
            </thead>
            <tbody>
              <tr><td>Доступ за індексом</td><td>O(1)</td><td>Позиція елемента обчислюється напряму.</td></tr>
              <tr><td>Додавання в кінець</td><td>O(1) амортизовано</td><td>Іноді потрібне розширення і копіювання.</td></tr>
              <tr><td>Вставка в середину</td><td>O(n)</td><td>Елементи праворуч треба зсунути.</td></tr>
              <tr><td>Видалення з середини</td><td>O(n)</td><td>Після видалення треба закрити проміжок.</td></tr>
            </tbody>
          </table>
        </div>
      </article>

      <article class="data-topic" id="ds-list">
        <h2>Зв’язний список</h2>
        <p>
          Зв’язний список складається з вузлів. Кожен вузол містить значення та посилання на наступний вузол. У двозв’язному списку вузол додатково має посилання на попередній елемент. На відміну від масиву, елементи списку не зобов’язані лежати поруч у пам’яті. Це дає гнучкість під час додавання і видалення, але ускладнює доступ за індексом.
        </p>
        <p>
          Якщо потрібно часто вставляти або видаляти елементи в різних місцях, список може бути зручнішим за масив, особливо коли вже відоме місце вставки. Але якщо потрібно часто звертатися до елемента за номером, список програє масиву, бо для знаходження позиції потрібно пройти вузли послідовно. Саме тому список добре підходить для динамічних наборів даних, але не завжди підходить для задач швидкого випадкового доступу.
        </p>
        <div class="theory-grid">
          <article class="theory-card">
            <h3>Однозв’язний список</h3>
            <p>Кожен вузол знає тільки наступний елемент. Структура проста, але рух назад неможливий без повторного проходу з початку.</p>
          </article>
          <article class="theory-card">
            <h3>Двозв’язний список</h3>
            <p>Кожен вузол знає і наступний, і попередній елемент. Це зручніше для видалення, але потребує більше пам’яті.</p>
          </article>
        </div>
        <div class="example-box">
          <strong>Приклад логіки:</strong> елемент 18 зберігає не тільки своє значення, а й адресу наступного вузла. Якщо потрібно видалити вузол, достатньо змінити посилання сусідніх елементів, не зсуваючи весь набір даних.
        </div>
      </article>
      <article class="data-topic" id="ds-stack">
```

```
<h2>Стек</h2>
<p>
  Стек – це структура даних, яка працює за принципом LIFO: останній доданий елемент обробляється першим. Його можна уявити як стопку тарілок: нову тарілку кладуть зверху і забирають теж зверху. Основні операції стека – push, pop і peek. Push додає елемент, pop забирає верхній елемент, peek лише переглядає його без видалення.
</p>
<p>
  Стек часто використовується там, де потрібно повернутися до попереднього стану. Приклади: стек викликів функцій, рекурсія, скасування дій у редакторі, перевірка правильності дужок, обхід графа в глибину. У DFS стек може бути явним або прихованим через рекурсивні виклики.
</p>
<div class="theory-table-wrap">
  <table class="theory-table">
    <thead>
      <tr><th>Операція</th><th>Сенс</th><th>Складність</th></tr>
    </thead>
    <tbody>
      <tr><td>push</td><td>додати елемент на вершину</td><td>O(1)</td></tr>
      <tr><td>pop</td><td>зняти елемент з вершини</td><td>O(1)</td></tr>
      <tr><td>peek</td><td>подивитися верхній елемент</td><td>O(1)</td></tr>
    </tbody>
  </table>
</div>
</article>
<article class="data-topic" id="ds-queue">
  <h2>Черга</h2>
  <p>
    Черга працює за принципом FIFO: перший доданий елемент обробляється першим. Це схоже на звичайну чергу людей: хто прийшов раніше, той обслуговується раніше. Основні операції – enqueue для додавання в кінець і dequeue для вилучення з початку.
  </p>
  <p>
    Черги використовуються в планувальниках задач, обробці запитів, буферах, системах друку, мережевих пакетах і алгоритмах обходу графів. У BFS черга зберігає вершини, які вже виявлені, але ще не оброблені. Завдяки цьому алгоритм проходить граф шарами: спочатку сусіди початкової вершини, потім сусіди цих сусідів і так далі.
  </p>
  <div class="example-box">
    <strong>Зв’язок з BFS:</strong> якщо вершина потрапила в чергу раніше, вона буде оброблена раніше. Саме це гарантує, що обхід у ширину знаходить найкоротшу кількість ребер у незваженому графі.
  </div>
</article>
<article class="data-topic" id="ds-deque">
  <h2>Дек</h2>
  <p>
    Дек – це двостороння черга, у якій елементи можна додавати і видаляти з обох кінців. Він поєднує властивості стека та черги. Якщо працювати тільки з одним кінцем, дек поводиться як стек. Якщо додавати в кінець, а забирати з початку, він поводиться як
```

звичайна черга.

Дек застосовують у задачах з ковзним вікном, у деяких алгоритмах пошуку, в історії дій, у системах буферизації та в ситуаціях, де важливо швидко працювати з обома краями послідовності. Порівняно зі звичайною чергою, дек гнучкіший, але реалізація має уважно контролювати межі, порожній стан і порядок додавання.

<article class="theory-card"><h3>Як стек</h3> Додавання і видалення з одного кінця дає поведінку LIFO. </article>
<article class="theory-card"><h3>Як черга</h3> Додавання в кінець і видалення з початку дає поведінку FIFO. </article>

</article>

<article class="data-topic" id="ds-hash">
<h2>Хеш-таблиця</h2>

Хеш-таблиця зберігає дані у вигляді пар «ключ – значення». Ключ обробляється хеш-функцією, яка перетворює його на індекс або позицію у внутрішньому масиві. Завдяки цьому пошук, додавання та видалення в середньому можуть виконуватися дуже швидко. Якщо кілька ключів потрапляють в одну позицію, виникає колізія, яку потрібно обробити.

Основні способи обробки колізій – ланцюжки та відкрита адресація. У методі ланцюжків в одній позиції зберігається список елементів. У відкритій адресації таблиця шукає іншу вільну позицію за певним правилом. Якість хеш-таблиці залежить від хеш-функції, коефіцієнта заповнення та стратегії розв’язання колізій.

<article class="theory-card">
<h3>Де використовується</h3>

словники та мапи;
кешування результатів;
підрахунок частот;
перевірка наявності елемента;
зберігання налаштувань за ключем.

</article>
<article class="theory-card">
<h3>Що важливо</h3>

хеш-функція має рівномірно розподіляти ключі;
потрібно враховувати колізії;
при сильному заповненні таблицю часто збільшують;
у найгіршому випадку швидкість може погіршитися до O(n).

</article>

</article>

```
<article class="data-topic" id="ds-tree">
  <h2>Дерева</h2>
  <p>
    Дерево – це ієрархічна структура даних, яка складається з вузлів і зв’язків між ними. Верхній вузол називається коренем. Вузли нижче називаються нащадками, а вузол, від якого вони відходять, – батьківським. Дерева добре підходять для подання ієрархій: файлових систем, меню, категорій, синтаксичних дерев, організаційних структур.
  </p>
  <p>
    У дереві важливі поняття висоти, рівня, листа, піддерева та ступеня вузла. Висота впливає на швидкість пошуку: якщо дерево збалансоване, шлях від кореня до потрібного елемента короткий. Якщо дерево вироджується в ланцюжок, його переваги зникають і пошук може стати лінійним.
  </p>
  <div class="theory-table-wrap">
    <table class="theory-table">
      <thead><tr><th>Поняття</th><th>Пояснення</th></tr></thead>
      <tbody>
        <tr><td>Корінь</td><td>Початковий вузол дерева.</td></tr>
        <tr><td>Лист</td><td>Вузол без нащадків.</td></tr>
        <tr><td>Висота</td><td>Максимальна довжина шляху від кореня до листа.</td></tr>
        <tr><td>Піддерево</td><td>Частина дерева, яка сама може розглядатися як дерево.</td></tr>
        <tr><td>Обхід</td><td>Порядок відвідування вузлів: прямий, симетричний, зворотний або рівнями.</td></tr>
      </tbody>
    </table>
  </div>
</article>
<article class="data-topic" id="ds-bst">
  <h2>Бінарне дерево пошуку</h2>
  <p>
    Бінарне дерево пошуку – це дерево, у якому для кожного вузла значення лівого піддерева менші, а значення правого піддерева більші. Така властивість дозволяє виконувати пошук подібно до Binary Search: на кожному кроці відкидається частина дерева. Якщо дерево збалансоване, пошук, вставка і видалення можуть виконуватися за O(log n).
  </p>
  <p>
    Проблема звичайного бінарного дерева пошуку полягає в тому, що порядок додавання елементів може зробити його незбалансованим. Наприклад, якщо додавати вже відсортовані значення, дерево може перетворитися на ланцюжок. Для уникнення цього використовують збалансовані дерева: AVL-дерева, червоно-чорні дерева та інші варіанти.
  </p>
  <div class="example-box">
    <strong>Навчальний висновок:</strong> сама ідея дерева пошуку ефективна лише тоді, коли висота дерева залишається малою. Тому при аналізі складності треба завжди враховувати не лише кількість елементів, а й форму дерева.
  </div>
</article>
<article class="data-topic" id="ds-heap">
  <h2>Купа</h2>
```

Купа – це спеціальна деревоподібна структура, у якій батьківський вузол має пріоритет відносно дочірніх. У max-heap найбільший елемент знаходиться у корені, а в min-heap – найменший. Хоча купу зручно уявляти як дерево, часто вона реалізується через масив.

Для елемента з індексом i у масивній реалізації лівий нащадок має індекс 2i + 1, правий – 2i + 2, а батьківський вузол – ⌊(i - 1) / 2⌋. Така формула дозволяє не зберігати явні посилання між вузлами. У проєкті купа пов’язана з Heap Sort: алгоритм будує max-heap, а потім поступово переносить найбільші елементи в кінець масиву.

<article class="theory-card"><h3>Черга з пріоритетами</h3> Купа дозволяє швидко отримати елемент із найбільшим або найменшим пріоритетом. </article>
<article class="theory-card"><h3>Heap Sort</h3> Алгоритм використовує властивість max-heap, щоб послідовно виносити найбільші елементи. </article>

</article>
<article class="data-topic" id="ds-graph">
<h2>Граф</h2>

Граф складається з вершин і ребер. Вершини позначають об’єкти, а ребра – зв’язки між ними. Графи можуть бути орієнтованими або неорієнтованими, зваженими або незваженими, зв’язними або незв’язними. У зваженому графі кожне ребро має вагу, наприклад відстань, вартість або час переходу.

Графи використовуються для моделювання доріг, соціальних мереж, комп’ютерних мереж, залежностей між задачами, маршрутів і зв’язків між об’єктами. У проєкті графи застосовуються для BFS, DFS та алгоритму Дейкстри. BFS демонструє обхід у ширину, DFS – обхід у глибину, а Dijkstra – пошук найкоротших шляхів у графі з невід’ємними вагами.

<article class="theory-card">
<h3>Матриця суміжності</h3>
Зручна, коли граф щільний. Перевірка наявності ребра між двома вершинами швидка, але пам’яті потрібно O(V²).
</article>
<article class="theory-card">
<h3>Список суміжності</h3>
Зручний для розріджених графів. Зберігає тільки наявні ребра, економить пам’ять і добре підходить для обходів.
</article>

<table class="theory-table">
<thead><tr><th>Алгоритм</th><th>Яка структура допомагає</th><th>Сенс</th></tr></thead>
<tbody>
<tr><td>BFS</td><td>Черга</td><td>Вершини обробляються у порядку виявлення.</td></tr>

```
            <tr><td>DFS</td><td>Стек або рекурсія</td><td>Алгоритм заглиблюється по одному шляху, потім повертається.</td></tr>
            <tr><td>Dijkstra</td><td>Граф і структура для вибору мінімальної відстані</td><td>Поступово уточнюються найкоротші шляхи від стартової вершини.</td></tr>
          </tbody>
        </table>
      </div>
    </article>
    <article class="data-topic" id="ds-complexity">
      <h2>Складність операцій і вибір структури</h2>
      <p>
        Вибір структури даних залежить від того, які операції виконуються найчастіше. Якщо програма багато разів читає елементи за індексом, масив буде зручним. Якщо часто потрібні вставки та видалення, може підійти список або дек. Якщо потрібно швидко перевіряти наявність елементів, часто використовують хеш-таблицю. Якщо дані мають ієрархію, обирають дерево. Якщо потрібно моделювати зв’язки між об’єктами, застосовують граф.
      </p>
      <div class="theory-table-wrap">
        <table class="theory-table">
          <thead>
            <tr><th>Структура</th><th>Доступ</th><th>Пошук</th><th>Вставка</th><th>Типові задачі</th></tr>
          </thead>
          <tbody>
            <tr><td>Масив</td><td>O(1) за індексом</td><td>O(n), або O(log n) для відсортованого</td><td>O(n) у середину</td><td>сортування, таблиці, послідовності</td></tr>
            <tr><td>Динамічний масив</td><td>O(1)</td><td>O(n)</td><td>O(1) амортизовано в кінець</td><td>списки елементів змінного розміру</td></tr>
            <tr><td>Список</td><td>O(n)</td><td>O(n)</td><td>O(1), якщо відомий вузол</td><td>динамічні послідовності</td></tr>
            <tr><td>Стек</td><td>тільки вершина</td><td>O(n)</td><td>O(1)</td><td>DFS, рекурсія, історія дій</td></tr>
            <tr><td>Черга</td><td>початок і кінець</td><td>O(n)</td><td>O(1)</td><td>BFS, планування, буфери</td></tr>
            <tr><td>Хеш-таблиця</td><td>за ключем</td><td>O(1) у середньому</td><td>O(1) у середньому</td><td>словники, кеш, частоти</td></tr>
            <tr><td>Дерево пошуку</td><td>через шлях</td><td>O(log n) у збалансованому</td><td>O(log n) у збалансованому</td><td>ієрархії, впорядковані множини</td></tr>
            <tr><td>Купа</td><td>корінь O(1)</td><td>O(n)</td><td>O(log n)</td><td>пріоритети, Heap Sort</td></tr>
            <tr><td>Граф</td><td>через вершини й ребра</td><td>залежить від алгоритму</td><td>залежить від подання</td><td>мережі, маршрути, залежності</td></tr>
          </tbody>
        </table>
      </div>
    </article>
    <article class="data-topic" id="ds-practice">
      <h2>Як обирати структуру даних на практиці</h2>
```

```
            <p>
              Для практичного вибору структури треба починати не з назви, а з вимог задачі. Потрібно визначити, які операції виконуються найчастіше, чи важливий порядок елементів, чи потрібен доступ за індексом, чи будуть часті вставки, чи потрібно швидко шукати за ключем, чи існують зв’язки між об’єктами. Після цього можна порівнювати структури за складністю операцій і пам’яттю.
            </p>
            <p>
              Типова помилка студентів – вивчати структури як набір визначень без зв’язку з алгоритмами. Насправді структура даних і алгоритм працюють разом. Невдалий вибір структури може зробити хороший алгоритм повільним, а правильний вибір може суттєво спростити розв’язання задачі. Наприклад, BFS майже неможливо зрозуміти без черги, Heap Sort – без купи, а швидкий пошук за ключем – без ідеї хешування.
            </p>
            <div class="theory-grid">
              <article class="theory-card">
                <h3>Питання перед вибором</h3>
                <ul>
                  <li>дані мають порядок чи ні;</li>
                  <li>потрібен доступ за індексом чи за ключем;</li>
                  <li>частіше відбувається пошук чи вставка;</li>
                  <li>дані мають ієрархію або мережеві зв’язки;</li>
                  <li>скільки пам’яті можна витратити.</li>
                </ul>
              </article>
              <article class="theory-card">
                <h3>Висновок для проєкту</h3>
                <p>Розділ структур даних пояснює основу візуалізованих алгоритмів і допомагає користувачу зрозуміти не лише результат, а й причину кожного кроку.</p>
              </article>
            </div>
          </article>
        </section>
      </main>
    </div>
  </div>
  <footer class="app-footer">
    <div class="footer-content">
      <p>
        <strong>AlgoViz Pro</strong> – платформа для інтерактивної
        візуалізації алгоритмів та структур даних
      </p>
    </div>
  </footer>
</div>
<script src="assets/js/utils/statistics.js"></script>
<script src="assets/js/utils/visualizer.js"></script>
<script src="assets/js/utils/data-file-loader.js"></script>
<script src="algorithms/base-algorithm.js"></script>
<script src="algorithms/sorting/bubble-sort.js"></script>
<script src="algorithms/sorting/insertion-sort.js"></script>
<script src="algorithms/sorting/merge-sort.js"></script>
```

```
    <script src="algorithms/sorting/quick-sort.js"></script>
    <script src="algorithms/sorting/heap-sort.js"></script>
    <script src="algorithms/searching/linear-search.js"></script>
    <script src="algorithms/searching/binary-search.js"></script>
    <script src="algorithms/searching/jump-search.js"></script>
    <script src="algorithms/searching/interpolation-search.js"></script>
    <script src="algorithms/graphs/bfs.js"></script>
    <script src="algorithms/graphs/dfs.js"></script>
    <script src="algorithms/graphs/dijkstra.js"></script>
    <script src="algorithms/dp/fibonacci.js"></script>
    <script src="algorithms/dp/lcs.js"></script>
    <script src="assets/js/pages/sidebar-render.js"></script>
    <script src="assets/js/pages/app.js"></script>
    <script src="assets/js/core/theme.js"></script>
    <script src="assets/js/core/nav-transition.js"></script>
    <script src="assets/js/pages/index-events.js"></script>
  </body>
</html>
```

A.2 Сторінка порівняння алгоритмів (compare.html)

```
<!doctype html>
<html lang="uk">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>AlgoViz Pro – Порівняння методів</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.4.0/css/all.min.css" />
    <script id="algoviz-theme-boot">
      (function () {
        try {
          var saved = localStorage.getItem("algoviz-theme") || "dark";
          document.documentElement.classList.add(saved === "light" ? "light-theme" : "dark-theme");
        } catch (e) {
          document.documentElement.classList.add("dark-theme");
        }
      })();
    </script>
    <link rel="stylesheet" href="styles.css" />
  </head>
  <body>
    <div class="app-wrapper">
      <nav class="navbar">
        <div class="navbar-container">
          <div class="navbar-brand">
            <i class="fas fa-code"></i>
            <span>AlgoViz Pro</span>
          </div>
          <div class="navbar-info">
```

```
        <div class="top-tabs">
          <a class="nav-link" href="index.html"><i class="fas fa-chart-bar"></i> Візуалізація</a>
          <a class="nav-link active" href="compare.html"><i class="fas fa-code-compare"></i>
Порівняння</a>
        </div>
        <button type="button" class="theme-toggle" id="themeToggle" aria-label="Перемкнути
тему" title="Перемкнути тему"><i class="fas fa-moon"></i></button>
       </div>
      </div>
    </nav>
    <div class="main-scroll compare-page-scroll">
      <main class="compare-page">
        <section class="compare-section compare-page-section">
          <div class="info-header compare-page-title">
            <h3><i class="fas fa-code-compare"></i> Порівняння методів</h3>
          </div>
          <div class="comparison-layout">
            <div class="compare-card">
              <h4>Методи</h4>
              <div class="compare-controls compare-controls-grid">
                <div class="compare-field">
                  <label for="compareAlgo1">Перший метод</label>
                  <select id="compareAlgo1"></select>
                </div>
                <div class="compare-field">
                  <label for="compareAlgo2">Другий метод</label>
                  <select id="compareAlgo2"></select>
                </div>
              </div>
            </div>
            <div class="compare-card">
              <h4>Загальні налаштування</h4>
              <div class="compare-controls compare-controls-grid">
                <div class="control-item compare-speed-control">
                  <label class="control-label" for="compareSpeedInput">
                    <span class="label-text">Швидкість анімації</span>
                  </label>
                  <div class="range-input-row">
                    <input type="range" id="compareSpeedInput" min="1" max="100" value="50"
class="slider" />
                    <input type="number" id="compareSpeedNumber" min="1" max="100" value="50"
class="manual-value-input" aria-label="Ввести швидкість анімації вручну" />
                  </div>
                </div>
              </div>
              <div id="compareSettings" class="compare-method-settings"></div>
              <div class="data-import-control compare-file-import" id="compareDataImportControl">
                <input type="file" id="compareDataFileInput" accept=".txt,text/plain" hidden />
                <button type="button" id="compareImportDataButton" class="btn btn-secondary data-
import-button">
                  <i class="fas fa-file-import"></i> Завантажити TXT
                </button>
```

```
            <span id="compareDataImportStatus" class="data-import-status">Можна завантажити
TXT-файл зі спільними числами.</span>
          </div>
         </div>
        </div>
        <div class="button-group compare-button-row">
         <button type="button" class="btn btn-primary" id="prepareCompareBtn">
          <i class="fas fa-rotate"></i> Оновити дані
         </button>
         <button type="button" class="btn btn-primary" id="startCompareBtn">
          <i class="fas fa-play"></i> Запустити порівняння
         </button>
         <button type="button" class="btn btn-secondary" id="pauseCompareBtn">
          <i class="fas fa-pause"></i> Пауза
         </button>
         <button type="button" class="btn btn-secondary" id="resetCompareBtn">
          <i class="fas fa-rotate-left"></i> Скинути
         </button>
         <button type="button" class="btn btn-secondary" id="exportCompareReportBtn">
          <i class="fas fa-file-pdf"></i> Зберегти PDF
         </button>
        </div>
       </section>
       <section class="compare-visual-section">
        <div class="compare-visual-card">
         <div class="compare-visual-header">
          <h4 id="compareTitle1">Метод 1</h4>
          <div class="compare-mini-stats">
           <span>Крок: <strong id="compareStep1">0</strong></span>
           <span>Час порівняння: <strong id="compareTime1">0ms</strong></span>
          </div>
         </div>
         <div class="compare-canvas-wrapper"><canvas id="compareCanvas1"></canvas></div>
        </div>
        <div class="compare-visual-card">
         <div class="compare-visual-header">
          <h4 id="compareTitle2">Метод 2</h4>
          <div class="compare-mini-stats">
           <span>Крок: <strong id="compareStep2">0</strong></span>
           <span>Час порівняння: <strong id="compareTime2">0ms</strong></span>
          </div>
         </div>
         <div class="compare-canvas-wrapper"><canvas id="compareCanvas2"></canvas></div>
        </div>
       </section>

       <section class="compare-section compare-results-section">
        <div class="info-header"><h3><i class="fas fa-chart-column"></i>
Результати</h3></div>
        <div id="compareResult" class="compare-result">
         <div class="empty-compare">
          <i class="fas fa-chart-column"></i>
```

```
            <span>Оберіть методи й натисніть «Запустити порівняння».</span>
          </div>
        </div>
      </section>
    </main>
  </div>
</div>
<script src="assets/js/utils/statistics.js"></script>
<script src="assets/js/utils/visualizer.js"></script>
<script src="assets/js/utils/data-file-loader.js"></script>
<script src="algorithms/base-algorithm.js"></script>
<script src="algorithms/sorting/bubble-sort.js"></script>
<script src="algorithms/sorting/insertion-sort.js"></script>
<script src="algorithms/sorting/merge-sort.js"></script>
<script src="algorithms/sorting/quick-sort.js"></script>
<script src="algorithms/sorting/heap-sort.js"></script>
<script src="algorithms/searching/linear-search.js"></script>
<script src="algorithms/searching/binary-search.js"></script>
<script src="algorithms/searching/jump-search.js"></script>
<script src="algorithms/searching/interpolation-search.js"></script>
<script src="assets/js/pages/compare.js"></script>
  <script src="assets/js/core/theme.js"></script>
<script src="assets/js/core/nav-transition.js"></script>
</body>
</html>
```

ДОДАТОК Б

Основні алгоритми

Б.1 Швидке сортування (quick-sort.js)

```
class QuickSort extends BaseAlgorithm {
   static info = {
      name: "Quick Sort",
      description: "Quick Sort – швидкий алгоритм сортування, який вибирає опорний елемент (pivot), розділяє масив на дві частини (менші та більші за pivot), і рекурсивно сортує ці частини.",
      steps: [
         "Вибирає опорний елемент (pivot).",
         "Розділяє масив на менші/більші за pivot.",
         "Рекурсивно сортує підмасиви.",
         "Об'єднує результат."
      ],
      complexity: "Часова складність: O(n log n) у середньому, O(n²) у найгіршому випадку; Пам'ять: O(log n) у середньому, O(n) у найгіршому"
   };

   execute() {
      const arr = [...this.data];
      this._quickSort(arr, 0, arr.length - 1);
      this.addStep({
         array: arr,
         sorted: Array.from({ length: arr.length }, (_, i) => i)
      });
      return arr;
   }
   _quickSort(arr, low, high) {
      if (low < high) {
         const pi = this._partition(arr, low, high);
         this._quickSort(arr, low, pi - 1);
         this._quickSort(arr, pi + 1, high);
      }
   }
   _partition(arr, low, high) {
      const pivot = arr[high];
      let i = low - 1;
      for (let j = low; j < high; j++) {
         this.statistics.incrementComparison();
         this.addStep({
```

```
            array: [...arr],
            comparing: [j, high]
        });
        if (arr[j] < pivot) {
            i++;
            [arr[i], arr[j]] = [arr[j], arr[i]];
            this.statistics.incrementSwap();
        }
    }
    [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]];
    this.statistics.incrementSwap();
    return i + 1;
  }
}
```

Б.2 Бінарний пошук (binary-search.js)

```
class BinarySearch extends BaseAlgorithm {
  static info = {
    name: "Binary Search",
    description: "Binary Search – ефективний алгоритм пошуку у відсортованому масиві. На кожному кроці порівнює шуканий елемент із середнім, відкидаючи половину масиву.",
    steps: [
      "Створює відсортовану копію масиву.",
      "Визначає межі поточного діапазону пошуку: left і right.",
      "Обчислює середній індекс mid.",
      "Порівнює array[mid] із шуканим значенням.",
      "Якщо значення рівні – елемент знайдено.",
      "Якщо array[mid] менше за ціль – пошук переходить у праву половину.",
      "Якщо array[mid] більше за ціль – пошук переходить у ліву половину.",
      "Якщо діапазон порожній – точного значення в масиві немає."
    ],
    complexity: "Разом із сортуванням: O(n log n); сам пошук: O(log n), Пам'ять: O(n)"
  };
  execute(target) {
    const arr = [...this.data].sort((a, b) => a - b);
    const searchTarget = Number(target);
    let left = 0;
    let right = arr.length - 1;
    this.addStep({
      array: arr,
      current: null,
      comparing: [],
      range: [left, right],
      target: searchTarget,
      message: `Шукаємо точне значення ${searchTarget} у відсортованому масиві.`
    });
    while (left <= right) {
      const mid = Math.floor((left + right) / 2);
```

```
            this.statistics.incrementComparison();
            this.addStep({
                array: arr,
                current: mid,
                comparing: [mid],
                range: [left, right],
                target: searchTarget,
                message: `Перевіряємо середній елемент: array[${mid}] = ${arr[mid]}. Діапазон:
${left}–${right}.`
            });
            if (arr[mid] === searchTarget) {
                this.addStep({
                    array: arr,
                    sorted: [mid],
                    current: mid,
                    comparing: [],
                    range: [mid, mid],
                    target: searchTarget,
                    foundIndex: mid,
                    message: `Значення ${searchTarget} знайдено на позиції ${mid}.`
                });
                return mid;
            }
            if (arr[mid] < searchTarget) {
                left = mid + 1;
                this.addStep({
                    array: arr,
                    current: null,
                    comparing: [],
                    range: left <= right ? [left, right] : [],
                    target: searchTarget,
                    message: `${arr[mid]} < ${searchTarget}, тому ліва частина відкидається. Новий
діапазон: ${left <= right ? `${left}–${right}` : "порожній"}.`
                });
            } else {
                right = mid - 1;
                this.addStep({
                    array: arr,
                    current: null,
                    comparing: [],
                    range: left <= right ? [left, right] : [],
                    target: searchTarget,
                    message: `${arr[mid]} > ${searchTarget}, тому права частина відкидається. Новий
діапазон: ${left <= right ? `${left}–${right}` : "порожній"}.`
                });
            }
        }
        this.addStep({
            array: arr,
            current: null,
            comparing: [],
            range: [],
```

```
            target: searchTarget,
            notFound: true,
            message: `Значення ${searchTarget} не знайдено.`
        });
        return -1;
    }
}
```

Б.3 Алгоритм Дейкстри (dijkstra.js)

```
class Dijkstra extends BaseAlgorithm {
    static info = {
        name: "Dijkstra's Algorithm",
        description: "Dijkstra – алгоритм пошуку найкоротших шляхів у графі з невід'ємними вагами ребер. Знаходить найкоротший шлях від стартової вершини до всіх інших або до вибраної кінцевої вершини.",
        steps: [
            "Будує зважений список суміжності графа.",
            "Ініціалізує відстані до всіх вершин як нескінченність, стартової – 0.",
            "Вибирає невідвідану вершину з мінімальною відстанню звичайним лінійним пошуком.",
            "Оновлює відстані до сусідів, якщо знайдено коротший шлях.",
            "Позначає вершину як відвідану.",
            "Повторює, поки всі потрібні вершини не відвідані."
        ],
        complexity: "Часова складність: O(V² + E), Пам'ять: O(V+E)"
    };
    constructor(nodes, edges, weights = {}) {
        super({ nodes, edges, weights });
        this.nodes = nodes;
        this.edges = edges;
        this.weights = weights;
        this.adjacency = this._buildWeightedAdjacency();
    }
    execute(startNode = 0, endNode = null) {
        const maxNode = this.nodes.length - 1;
        startNode = Math.min(maxNode, Math.max(0, parseInt(startNode, 10) || 0));
        endNode = endNode === null || Number.isNaN(endNode)
            ? null
            : Math.min(maxNode, Math.max(0, parseInt(endNode, 10) || 0));
        const distances = Array(this.nodes.length).fill(Infinity);
        const visited = new Set();
        distances[startNode] = 0;
        this.addStep({
            nodes: this.nodes,
            edges: this.edges,
            current: startNode,
            visited: [],
            distances: [...distances],
            endNode,
```

```
            weights: this.weights
        });
        for (let i = 0; i < this.nodes.length; i++) {
            let minNode = -1;
            let minDist = Infinity;
            for (let j = 0; j < this.nodes.length; j++) {
                if (!visited.has(j) && distances[j] < minDist) {
                    minNode = j;
                    minDist = distances[j];
                }
            }
            if (minNode === -1) break;
            visited.add(minNode);
            this.statistics.incrementOperation();
            this.addStep({
                nodes: this.nodes,
                edges: this.edges,
                current: minNode,
                visited: Array.from(visited),
                distances: [...distances],
                endNode,
                weights: this.weights
            });
            if (endNode !== null && minNode === endNode) {
                break;
            }
            const neighbors = this._getNeighbors(minNode);
            neighbors.forEach(({ node: neighbor, weight }) => {
                if (!visited.has(neighbor) && distances[minNode] + weight < distances[neighbor]) {
                    distances[neighbor] = distances[minNode] + weight;
                    this.statistics.incrementComparison();
                    this.addStep({
                        nodes: this.nodes,
                        edges: this.edges,
                        current: neighbor,
                        visited: Array.from(visited),
                        distances: [...distances],
                        endNode,
                        weights: this.weights
                    });
                }
            });
        }
        return distances;
    }
    _buildWeightedAdjacency() {
        const adjacency = Array.from({ length: this.nodes.length }, () => []);
        this.edges.forEach(edge => {
            const [from, to] = edge;
            if (!adjacency[from] || !adjacency[to]) return;
            const weight = this._getEdgeWeight(from, to);
            adjacency[from].push({ node: to, weight });
```

```
        adjacency[to].push({ node: from, weight });
      });
      return adjacency;
    }
    _getEdgeWeight(from, to) {
      const directWeight = this.weights[`${from}-${to}`];
      const reverseWeight = this.weights[`${to}-${from}`];
      const weight = directWeight ?? reverseWeight ?? 1;
      const numericWeight = Number(weight);
      return Number.isFinite(numericWeight) && numericWeight >= 0 ? numericWeight : 1;
    }
    _getNeighbors(node) {
      return this.adjacency[node] || [];
    }
}
```

Б.4 LCS (lcs.js)

```
class LCS extends BaseAlgorithm {
  static info = {
    name: "Longest Common Subsequence (LCS)",
    description:
      "LCS знаходить найдовшу спільну підпослідовність двох рядків за допомогою DP-
матриці.",
    steps: [
      "Створює матрицю dp розміром (m + 1) × (n + 1).",
      "Якщо символи однакові, бере значення з діагоналі й додає 1.",
      "Якщо символи різні, бере максимум із верхньої або лівої клітинки.",
      "Після заповнення матриці відновлює відповідь рухом назад від правого нижнього кута.",
    ],
    complexity: "DP: O(m×n); у візуалізації з відновленням поточного LCS: до O(m×n×(m+n)),
Пам'ять: O(m×n)",
  };
  execute(str1 = "ABCDGH", str2 = "AEDFHR") {
    this.steps = [];
    this.statistics.reset();
    str1 = String(str1 || "").slice(0, 14);
    str2 = String(str2 || "").slice(0, 14);
    const m = str1.length;
    const n = str2.length;
    const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));
    this.addStep({
      matrix: dp.map((row) => [...row]),
      currentCell: [0, 0],
      sourceCells: [],
      str1,
      str2,
      lcs: "",
```

```
      explanation: "Початкова матриця заповнена нулями.",
    });
    for (let i = 1; i <= m; i++) {
      for (let j = 1; j <= n; j++) {
        this.statistics.incrementComparison();
        const char1 = str1[i - 1];
        const char2 = str2[j - 1];
        let explanation;
        let sourceCells;
        if (char1 === char2) {
          dp[i][j] = dp[i - 1][j - 1] + 1;
          sourceCells = [[i - 1, j - 1]];
          explanation = `${char1} = ${char2}: беремо діагональ ${dp[i - 1][j - 1]} + 1 = ${dp[i][j]}`;
        } else {
          const top = dp[i - 1][j];
          const left = dp[i][j - 1];
          dp[i][j] = Math.max(top, left);
          sourceCells = top >= left ? [[i - 1, j]] : [[i, j - 1]];
          explanation = `${char1} ≠ ${char2}: max(зверху ${top}, зліва ${left}) = ${dp[i][j]}`;
        }
        this.statistics.incrementOperation();
        this.addStep({
          matrix: dp.map((row) => [...row]),
          currentCell: [i, j],
          sourceCells,
          str1,
          str2,
          lcs: this._buildCurrentLCS(dp, str1, str2, i, j),
          explanation,
        });
      }
    }
    const finalLCS = this._buildCurrentLCS(dp, str1, str2, m, n);
    this.addStep({
      matrix: dp.map((row) => [...row]),
      currentCell: [m, n],
      sourceCells: this._tracebackPath(dp, str1, str2),
      str1,
      str2,
      lcs: finalLCS,
      explanation: `Матриця заповнена. Відповідь: ${finalLCS || "порожній рядок"}.`,
    });
    return finalLCS;
  }
  _buildCurrentLCS(dp, str1, str2, i, j) {
    const result = [];
    while (i > 0 && j > 0) {
      if (str1[i - 1] === str2[j - 1]) {
        result.unshift(str1[i - 1]);
        i--;
        j--;
      } else if (dp[i - 1][j] >= dp[i][j - 1]) {
```

```
        i--;
      } else {
        j--;
      }
    }
    return result.join("");
  }
  _tracebackPath(dp, str1, str2) {
    const path = [];
    let i = str1.length;
    let j = str2.length;
    while (i > 0 && j > 0) {
      path.push([i, j]);
      if (str1[i - 1] === str2[j - 1]) {
        i--;
        j--;
      } else if (dp[i - 1][j] >= dp[i][j - 1]) {
        i--;
      } else {
        j--;
      }
    }
    return path;
  }
}
```

ДОДАТОК В

Програмні модулі інтерфейсу та візуалізації

B.1 visualizer.js

```
class Visualizer {
  constructor(canvasId) {
    this.canvas = document.getElementById(canvasId);
    if (!this.canvas) {
      throw new Error("Canvas element not found");
    }
    this.ctx = this.canvas.getContext("2d");
    this.dpr = window.devicePixelRatio || 1;
    this.width = 0;
    this.height = 0;
    this.animationFrame = null;
    this.lastBars = [];
    this._resizeRaf = null;
    this._lastCanvasState = { width: 0, height: 0, dpr: window.devicePixelRatio || 1 };
    this._forceInstantBars = false;
    this.resizeCanvas(true);
    const scheduleResize = () => this.scheduleResize();
    window.addEventListener("resize", scheduleResize);
    if (window.visualViewport) {
      window.visualViewport.addEventListener("resize", scheduleResize);
    }
    if (window.ResizeObserver && this.canvas.parentElement) {
      this.resizeObserver = new ResizeObserver(scheduleResize);
      this.resizeObserver.observe(this.canvas.parentElement);
    }

  }
  scheduleResize() {
    if (this._resizeRaf) {
      cancelAnimationFrame(this._resizeRaf);
    }
    this._resizeRaf = requestAnimationFrame(() => {
      this._resizeRaf = null;
      const changed = this.resizeCanvas();
      if (changed) {
        window.dispatchEvent(new CustomEvent("algoviz-canvas-resized"));
      }
    });
  }
  resizeCanvas(force = false) {
```

```
  if (!this.canvas || !this.canvas.parentElement) return false;

  const dpr = window.devicePixelRatio || 1;
  const rect = this.canvas.parentElement.getBoundingClientRect();
  const width = Math.max(300, Math.round(rect.width));
  const height = Math.max(250, Math.round(rect.height));
  const sameSize =
    !force &&
    this._lastCanvasState &&
    this._lastCanvasState.width === width &&
    this._lastCanvasState.height === height &&
    Math.abs(this._lastCanvasState.dpr - dpr) < 0.001;
  if (sameSize) {
    return false;
  }
  this.dpr = dpr;
  this.canvas.width = Math.round(width * this.dpr);
  this.canvas.height = Math.round(height * this.dpr);
  this.canvas.style.width = "100%";
  this.canvas.style.height = "100%";
  this.ctx.setTransform(1, 0, 0, 1, 0, 0);
  this.ctx.scale(this.dpr, this.dpr);
  this.width = width;
  this.height = height;
  this._lastCanvasState = { width, height, dpr };
  this.lastBars = [];
  this._forceInstantBars = true;
  this.clear();
  return true;
}
stopAnimation() {
  if (this.animationFrame) {
    cancelAnimationFrame(this.animationFrame);
    this.animationFrame = null;
  }
}
isLightTheme() {
  return document.body && document.body.classList.contains("light-theme");
}

getThemeColors() {
  const light = this.isLightTheme();
  return {
    canvasBg: light ? "#f8fafc" : "#0f172a",
    text: light ? "#0f172a" : "#e2e8f0",
    textMuted: light ? "#475569" : "#cbd5e1",
    border: light ? "rgba(37, 99, 235, 0.22)" : "rgba(148, 163, 184, 0.22)",
    bar: "#3b82f6",
    range: light ? "#bfdbfe" : "#60a5fa",
    stroke: light ? "rgba(15, 23, 42, 0.12)" : "rgba(255,255,255,0.12)",
    panelBg: light ? "rgba(239, 246, 255, 0.96)" : "rgba(15, 23, 42, 0.92)",
    cellEmpty: light ? "#eaf2ff" : "#1e293b",
```

```
    cellFilled: light ? "#dbeafe" : "#334155",
    cellText: light ? "#0f172a" : "#f8fafc",
    edge: light ? "rgba(71, 85, 105, 0.5)" : "rgba(148, 163, 184, 0.55)",
    labelBg: light ? "rgba(255, 255, 255, 0.92)" : "rgba(15, 23, 42, 0.9)",
    labelText: light ? "#0f172a" : "#f8fafc",
  };
}
clear() {
  if (!this.ctx) return;
  const c = this.getThemeColors();
  this.ctx.clearRect(0, 0, this.width, this.height);
  this.ctx.fillStyle = c.canvasBg;
  this.ctx.fillRect(0, 0, this.width, this.height);
}
drawBars(array, highlights = {}) {
  if (!Array.isArray(array) || array.length === 0) {
    this.clear();
    return;
  }
  this.stopAnimation();
  const fromArr =
    this.lastBars && this.lastBars.length === array.length
      ? this.lastBars.slice()
      : Array(array.length).fill(0);
  const toArr = array.slice();
  this.lastBars = toArr.slice();
  const animate = (progress) => {
    this.clear();
    const t = Math.min(progress, 1);
    const ease = t < 0.5 ? 4 * t * t * t : 1 - Math.pow(-2 * t + 2, 3) / 2;
    const barWidth = this.width / toArr.length;
    const maxValue = Math.max(...toArr, 1);
    const usableHeight = this.height - 62;
    if (highlights.message) {
      const c = this.getThemeColors();
      this.ctx.fillStyle = highlights.notFound ? "#ef4444" : c.text;
      this.ctx.font = "15px Arial";
      this.ctx.textAlign = "left";
      this.ctx.textBaseline = "top";
      this.ctx.fillText(highlights.message, 14, 12);
    }
    toArr.forEach((targetValue, index) => {
      const startValue = fromArr[index] ?? 0;
      const value = startValue + (targetValue - startValue) * ease;
      const height = (value / maxValue) * usableHeight;
      const x = index * barWidth + 2;
      const y = this.height - height - 20;
      const width = Math.max(2, barWidth - 4);
      const c = this.getThemeColors();
      let color = c.bar;
      const isInSearchRange = Array.isArray(highlights.range) &&
        highlights.range.length === 2 &&
```

```
      index >= highlights.range[0] &&
      index <= highlights.range[1];
    if (isInSearchRange) {
      color = c.range;
    }
    if (highlights.sorted && highlights.sorted.includes(index)) {
      color = "#10b981";
    } else if (
      highlights.comparing &&
      highlights.comparing.includes(index)
    ) {
      color = "#ef4444";
    } else if (highlights.current === index) {
      color = "#f59e0b";
    }
    this.ctx.fillStyle = color;
    this.ctx.fillRect(x, y, width, height);
    this.ctx.strokeStyle = c.stroke;
    this.ctx.strokeRect(x, y, width, height);
    const labelY = Math.max(12, y - 3);
    let fontSize = 18;
    if (toArr.length > 160) fontSize = 5;
    else if (toArr.length > 120) fontSize = 6;
    else if (toArr.length > 90) fontSize = 7;
    else if (toArr.length > 65) fontSize = 8;
    else if (toArr.length > 45) fontSize = 10;
    else if (toArr.length > 30) fontSize = 12;
    else if (toArr.length > 18) fontSize = 14;
    else if (toArr.length > 10) fontSize = 16;
    fontSize = Math.max(5, Math.min(fontSize, Math.floor(width * 0.82)));
    this.ctx.save();
    this.ctx.font = `700 ${fontSize}px Arial`;
    this.ctx.textAlign = "center";
    this.ctx.textBaseline = "bottom";
    this.ctx.lineWidth = this.isLightTheme() ? 2.4 : 1.8;
    this.ctx.strokeStyle = this.isLightTheme() ? "rgba(255,255,255,0.98)" : "rgba(15,23,42,0.96)";
    this.ctx.fillStyle = this.isLightTheme() ? "#0b1220" : "#f8fafc";
    this.ctx.strokeText(Math.round(value), x + width / 2, labelY);
    this.ctx.fillText(Math.round(value), x + width / 2, labelY);
    this.ctx.restore();
  });
};
const requestedDuration = Number(highlights.animationDuration);
const barAnimationDuration = Number.isFinite(requestedDuration)
  ? Math.min(700, Math.max(420, requestedDuration * 0.55))
  : 420;
const duration = this._forceInstantBars ? 0 : barAnimationDuration;
this._forceInstantBars = false;
if (duration === 0) {
  animate(1);
  this.animationFrame = null;
  return;
```

```
    }
    let start = null;
    const frame = (timestamp) => {
      if (!start) start = timestamp;
      const progress = (timestamp - start) / duration;
      animate(progress);
      if (progress < 1) {
        this.animationFrame = requestAnimationFrame(frame);
      } else {
        this.animationFrame = null;
      }
    };
    this.animationFrame = requestAnimationFrame(frame);
  }

  drawGraph(nodes, edges, highlights = {}) {
    this.stopAnimation();
    this.clear();
    if (!Array.isArray(nodes) || nodes.length === 0) return;
    const stateLines = this.getGraphStateLines(highlights);
    const stateHeight = this.drawGraphStateBox(stateLines);
    const topReserved = stateHeight + 38;
    const positions = this.calculateGraphPositions(nodes.length, topReserved, highlights.layoutSeed
|| 1);
    const radius = nodes.length > 35 ? 11 : nodes.length > 20 ? 14 : 20;
    const activeEdges = new Set();
    if (Array.isArray(highlights.visited) && highlights.current !== undefined && highlights.current
!== null) {
      edges.forEach(([from, to]) => {
        const connectsCurrent = from === highlights.current || to === highlights.current;
        const other = from === highlights.current ? to : from;
        if (connectsCurrent && highlights.visited.includes(other)) {
          activeEdges.add(`${from}-${to}`);
          activeEdges.add(`${to}-${from}`);
        }
      });
    }
    edges.forEach(([from, to]) => {
      const p1 = positions[from];
      const p2 = positions[to];
      if (!p1 || !p2) return;
      const isActive = activeEdges.has(`${from}-${to}`) || activeEdges.has(`${to}-${from}`);
      const c = this.getThemeColors();
      this.ctx.strokeStyle = isActive ? "#f59e0b" : c.edge;
      this.ctx.lineWidth = isActive ? 3.2 : 1.45;
      this.ctx.beginPath();
      this.ctx.moveTo(p1.x, p1.y);
      this.ctx.lineTo(p2.x, p2.y);
      this.ctx.stroke();
      const weight = highlights.weights && (highlights.weights[`${from}-${to}`] ??
highlights.weights[`${to}-${from}`]);
      if (weight !== undefined && weight !== null && weight !== "") {
```

```
      const mx = (p1.x + p2.x) / 2;
      const my = (p1.y + p2.y) / 2;
      this.ctx.fillStyle = c.labelBg;
      this.ctx.beginPath();
      this.ctx.arc(mx, my, 8, 0, Math.PI * 2);
      this.ctx.fill();
      this.ctx.fillStyle = c.labelText;
      this.ctx.font = "10px Arial";
      this.ctx.textAlign = "center";
      this.ctx.textBaseline = "middle";
      this.ctx.fillText(String(weight), mx, my);
    }
  });
  nodes.forEach((node, index) => {
    const pos = positions[index];
    if (!pos) return;
    let color = "#3b82f6";
    if (highlights.current === index) {
      color = "#f59e0b";
    } else if (highlights.endNode === index) {
      color = "#ec4899";
    } else if (highlights.visited && highlights.visited.includes(index)) {
      color = "#10b981";
    }
    this.ctx.fillStyle = color;
    this.ctx.beginPath();
    this.ctx.arc(pos.x, pos.y, radius, 0, Math.PI * 2);
    this.ctx.fill();
    this.ctx.strokeStyle = this.isLightTheme() ? "rgba(15,23,42,0.35)" : "rgba(255,255,255,0.82)";
    this.ctx.lineWidth = 2;
    this.ctx.stroke();
    this.ctx.fillStyle = "white";
    this.ctx.font = `bold ${nodes.length > 35 ? 8 : 12}px Arial`;
    this.ctx.textAlign = "center";
    this.ctx.textBaseline = "middle";
    this.ctx.fillText(String(node), pos.x, pos.y);
  });
}
seededRandom(seed) {
  let value = Math.sin(seed) * 10000;
  return value - Math.floor(value);
}
calculateGraphPositions(count, topReserved = 95, seed = 1) {
  const padding = 46;
  const availableWidth = Math.max(260, this.width - padding * 2);
  const availableHeight = Math.max(220, this.height - topReserved - padding);
  if (count <= 12) {
    const centerX = this.width / 2;
    const centerY = topReserved + availableHeight / 2;
    const rx = availableWidth * 0.42;
    const ry = availableHeight * 0.36;
```

```
    return Array.from({ length: count }, (_, index) => {
      const angle = -Math.PI / 2 + (index * 2 * Math.PI) / count;
      const wave = 0.7 + this.seededRandom(seed + index * 13) * 0.22;
      const shiftX = (this.seededRandom(seed + index * 17) - 0.5) * 34;
      const shiftY = (this.seededRandom(seed + index * 23) - 0.5) * 30;

      return {
        x: Math.min(this.width - padding, Math.max(padding, centerX + Math.cos(angle) * rx * wave
+ shiftX)),
        y: Math.min(this.height - padding, Math.max(topReserved, centerY + Math.sin(angle) * ry *
wave + shiftY)),
      };
    });
  }
  const columns = Math.ceil(Math.sqrt(count * (availableWidth / availableHeight)));
  const rows = Math.ceil(count / columns);
  const cellW = availableWidth / Math.max(columns - 1, 1);
  const cellH = availableHeight / Math.max(rows - 1, 1);

  return Array.from({ length: count }, (_, index) => {
    const row = Math.floor(index / columns);
    const col = index % columns;
    const baseX = columns === 1 ? this.width / 2 : padding + col * cellW;
    const baseY = rows === 1 ? topReserved + availableHeight / 2 : topReserved + row * cellH;
    const jitterLimit = Math.min(18, cellW * 0.16, cellH * 0.16);
    const jitterX = (this.seededRandom(seed + index * 31) - 0.5) * 2 * jitterLimit;
    const jitterY = (this.seededRandom(seed + index * 47) - 0.5) * 2 * jitterLimit;
    return {
      x: Math.min(this.width - padding, Math.max(padding, baseX + jitterX)),
      y: Math.min(this.height - padding, Math.max(topReserved, baseY + jitterY)),
    };
  });
 }
 drawDPArray(values, highlights = {}) {
  this.stopAnimation();
  this.clear();
  if (!Array.isArray(values) || values.length === 0) return;
  const padding = 28;
  const gap = 8;
  const maxColumns = values.length > 25 ? 10 : 8;
  const columns = Math.min(maxColumns, values.length);
  const cellW = Math.min(82, (this.width - padding * 2 - gap * (columns - 1)) / columns);
  const cellH = 48;
  const startY = 72;
  this.ctx.textAlign = "center";
  this.ctx.textBaseline = "middle";
  values.forEach((value, index) => {
    const row = Math.floor(index / columns);
    const col = index % columns;
    const x = padding + col * (cellW + gap);
    const y = startY + row * (cellH + 28);
    const c = this.getThemeColors();
```

```
      let fill = c.cellEmpty;
      if (index === highlights.current) fill = "#f59e0b";
      else if (Array.isArray(highlights.used) && highlights.used.includes(index)) fill = "#2563eb";
      else if (value !== null && value !== undefined) fill = c.cellFilled;
      this.ctx.fillStyle = fill;
      this.ctx.fillRect(x, y, cellW, cellH);
      this.ctx.strokeStyle = "rgba(148, 163, 184, 0.55)";
      this.ctx.strokeRect(x, y, cellW, cellH);
      this.ctx.fillStyle = c.textMuted;
      this.ctx.font = "12px Arial";
      this.ctx.fillText(`dp[${index}]`, x + cellW / 2, y - 12);
      this.ctx.fillStyle = c.cellText;
      this.ctx.font = "bold 14px Arial";
      this.ctx.fillText(value === null || value === undefined ? "" : String(value), x + cellW / 2, y +
cellH / 2);
    });
    this.ctx.fillStyle = this.getThemeColors().text;
    this.ctx.font = "16px Arial";
    this.ctx.textAlign = "left";
    this.ctx.fillText(highlights.formula || "Побудова таблиці dp[]", 24, 32);
    if (highlights.result !== undefined) {
      this.ctx.fillText(`Поточний результат: ${highlights.result}`, 24, this.height - 28);
    }
  }
  getGraphStateLines(highlights = {}) {
    const lines = [];
    if (Array.isArray(highlights.queue) && highlights.queue.length > 0) {
      lines.push(`Черга: ${highlights.queue.join(" → ")}`);
    }
    if (Array.isArray(highlights.stack) && highlights.stack.length > 0) {
      lines.push(`Стек/шлях: ${highlights.stack.join(" → ")}`);
    }
    if (Array.isArray(highlights.distances) && highlights.distances.length > 0) {
      const distanceItems = highlights.distances
        .map((value, index) => `${index}: ${value === Infinity ? "∞" : value}`);
      const maxItemsPerLine = Math.max(6, Math.floor((this.width - 90) / 82));
      for (let i = 0; i < distanceItems.length; i += maxItemsPerLine) {
        const part = distanceItems.slice(i, i + maxItemsPerLine).join("   ");
        lines.push(`${i === 0 ? "Відстані:" : ""} ${part}`.trim());
      }
    }

    return lines;
  }
  drawGraphStateBox(lines = []) {
    if (!Array.isArray(lines) || lines.length === 0) return 0;
    const lineHeight = 21;
    const boxHeight = 20 + lines.length * lineHeight;
    const c = this.getThemeColors();
    this.ctx.fillStyle = c.panelBg;
    this.ctx.fillRect(12, 12, this.width - 24, boxHeight);
    this.ctx.strokeStyle = "rgba(148, 163, 184, 0.22)";
```

```
  this.ctx.lineWidth = 1;
  this.ctx.strokeRect(12, 12, this.width - 24, boxHeight);
  this.ctx.fillStyle = c.text;
  this.ctx.font = "14px Arial";
  this.ctx.textAlign = "left";
  this.ctx.textBaseline = "top";
  lines.forEach((line, index) => {
    this.ctx.fillText(line, 24, 22 + index * lineHeight);
  });
  return boxHeight;
}
drawGraphState(highlights = {}) {
  return this.drawGraphStateBox(this.getGraphStateLines(highlights));
}
drawMatrix(matrix, highlights = {}) {
  this.stopAnimation();
  this.clear();
  if (!Array.isArray(matrix) || matrix.length === 0) return;
  const rows = matrix.length;
  const cols = matrix[0].length;
  const topOffset = 70;
  const leftOffset = 70;
  const maxCellWidth = (this.width - leftOffset - 20) / Math.max(cols, 1);
  const maxCellHeight = (this.height - topOffset - 55) / Math.max(rows, 1);
  const cellSize = Math.max(22, Math.min(44, maxCellWidth, maxCellHeight));
  const [currentRow, currentCol] = highlights.currentCell || [-1, -1];
  this.ctx.font = "13px Arial";
  this.ctx.textAlign = "center";
  this.ctx.textBaseline = "middle";

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      const x = leftOffset + j * cellSize;
      const y = topOffset + i * cellSize;
      const isSource = Array.isArray(highlights.sourceCells) &&
        highlights.sourceCells.some(([r, c]) => r === i && c === j);
      const c = this.getThemeColors();
      let fill = c.cellEmpty;
      if (i === currentRow && j === currentCol) {
        fill = "#f59e0b";
      } else if (isSource) {
        fill = "#2563eb";
      } else if (i === currentRow || j === currentCol) {
        fill = c.cellFilled;
      }
      this.ctx.fillStyle = fill;
      this.ctx.fillRect(x, y, cellSize, cellSize);
      this.ctx.strokeStyle = "rgba(148, 163, 184, 0.45)";
      this.ctx.strokeRect(x, y, cellSize, cellSize);
      this.ctx.fillStyle = c.cellText;
      this.ctx.fillText(String(matrix[i][j]), x + cellSize / 2, y + cellSize / 2);
    }
```

```
    }
    this.ctx.fillStyle = this.getThemeColors().textMuted;
    this.ctx.font = "bold 13px Arial";
    if (highlights.str2) {
      for (let j = 1; j < cols; j++) {
        const char = highlights.str2[j - 1] || "";
        this.ctx.fillText(char, leftOffset + j * cellSize + cellSize / 2, topOffset - 18);
      }
    }
    if (highlights.str1) {
      for (let i = 1; i < rows; i++) {
        const char = highlights.str1[i - 1] || "";
        this.ctx.fillText(char, leftOffset - 18, topOffset + i * cellSize + cellSize / 2);
      }
    }
    this.ctx.fillStyle = this.getThemeColors().text;
    this.ctx.font = "15px Arial";
    this.ctx.textAlign = "left";
    this.ctx.fillText(`LCS: ${highlights.lcs || "—"}`, 20, this.height - 45);

    if (highlights.explanation) {
      this.ctx.fillStyle = this.getThemeColors().textMuted;
      this.ctx.font = "13px Arial";
      this.ctx.fillText(highlights.explanation, 20, this.height - 22);
    }
  }
}
```

B.2 statistics.js

```
class Statistics {
  constructor() {
    this.reset();
  }
  static now() {
    if (typeof performance !== "undefined" && typeof performance.now === "function") {
      return performance.now();
    }
    return Date.now();
  }
  now() {
    return Statistics.now();
  }
  reset() {
    this.operations = 0;
    this.comparisons = 0;
    this.swaps = 0;
    this.startTime = this.now();
```

```
  }
  incrementComparison() {
    this.comparisons++;
    this.operations++;
  }
  incrementSwap() {
    this.swaps++;
    this.operations++;
  }
  incrementOperation() {
    this.operations++;
  }
  getElapsedTime() {
    return Math.max(0, Math.round(this.now() - this.startTime));
  }
  getStats() {
    return {
      operations: this.operations,
      comparisons: this.comparisons,
      swaps: this.swaps,
      timeElapsed: this.getElapsedTime(),
    };
  }
}
```

B.3 app.js

```
let visualizer;
let currentAlgorithm = null;
let currentAlgorithmType = null;
let isAnimating = false;
let animationSpeed = 50;
let currentStepIndex = 0;
let animationTimeout = null;
let currentGraphLayoutSeed = 1;
let visualizationStartedAt = 0;
let visualizationElapsedBeforePause = 0;
let visualizationDisplayedElapsed = 0;
let visualizationElapsedTarget = 0;
let visualizationTimerFrame = null;
let currentVisualizationStepDuration = 0;
let visualizationStepStartedAt = 0;
let currentAlgorithmResult = null;
let currentAlgorithmHasStarted = false;
let currentAlgorithmIsFinished = false;
let customArrayData = null;
let customArrayFileName = "";
let customSearchTarget = null;
```

```
let activeDataStructureScrollId = "";
function getAnimationDelay(speedValue) {
  const value = Math.max(1, Math.min(100, parseInt(speedValue, 10) || 50));
  return Math.round(460 + (100 - value) * 12);
}
function getAlgorithmDisplayTimeFromStats(stats) {
  if (!stats) return 0;
  const operations = Math.max(0, Number(stats.operations) || 0);
  const comparisons = Math.max(0, Number(stats.comparisons) || 0);
  const swaps = Math.max(0, Number(stats.swaps) || 0);
  const otherOperations = Math.max(0, operations - comparisons - swaps);
  return Math.round(comparisons + swaps + otherOperations);
}
function getAlgorithmDisplayTimeFromStep(step) {
  return getAlgorithmDisplayTimeFromStats(step && step.stats ? step.stats : null);
}

function getAnimationNow() {
  if (typeof Statistics !== "undefined" && typeof Statistics.now === "function") {
    return Statistics.now();
  }
  if (typeof performance !== "undefined" && typeof performance.now === "function") {
    return performance.now();
  }
  return Date.now();
}
function getCurrentStepProgress() {
  if (!animationTimeout || currentVisualizationStepDuration <= 0 || visualizationStepStartedAt <= 0)
{
    return 0;
  }
  const elapsed = Math.max(0, getAnimationNow() - visualizationStepStartedAt);
  return Math.min(1, elapsed / currentVisualizationStepDuration);
}
function getRemainingVisualizationDuration() {
  if (!currentAlgorithm || !Array.isArray(currentAlgorithm.steps)) {
    return animationSpeed;
  }
  const remainingSteps = Math.max(0, currentAlgorithm.steps.length - currentStepIndex);
  if (remainingSteps === 0) return animationSpeed;

  const currentStepRemaining = animationTimeout
    ? Math.max(0, animationSpeed * (1 - getCurrentStepProgress()))
    : animationSpeed;
  return Math.max(animationSpeed, currentStepRemaining + Math.max(0, remainingSteps - 1) *
animationSpeed);
}
function scheduleNextAnimationStep(delay = animationSpeed) {
  if (animationTimeout) {
    clearTimeout(animationTimeout);
  }
  currentVisualizationStepDuration = Math.max(1, delay);
```

```
  visualizationStepStartedAt = getAnimationNow();
  animationTimeout = setTimeout(animateSteps, currentVisualizationStepDuration);
}
function rescheduleAnimationAfterSpeedChange() {
  if (!isAnimating || !animationTimeout) return;

  const progress = getCurrentStepProgress();
  const remainingDelay = Math.max(1, Math.round(animationSpeed * (1 - progress)));

  clearTimeout(animationTimeout);
  currentVisualizationStepDuration = animationSpeed;
  visualizationStepStartedAt = getAnimationNow() - progress * currentVisualizationStepDuration;
  animationTimeout = setTimeout(animateSteps, remainingDelay);
}
document.addEventListener("DOMContentLoaded", () => {
  visualizer = new Visualizer("canvas");
  setupEventListeners();
  setupCollapsibleCategories();
  initializeUI();
});
function startVisualizationTimer() {
  stopVisualizationTimer();
  if (!currentAlgorithm || !Array.isArray(currentAlgorithm.steps) || currentAlgorithm.steps.length
=== 0) {
    updateVisualizationTime();
    return;
  }
  const finalStep = currentAlgorithm.steps[currentAlgorithm.steps.length - 1];
  const startValue = Math.max(0, Number(visualizationDisplayedElapsed) || 0);
  const targetValue = getAlgorithmDisplayTimeFromStep(finalStep);
  const duration = getRemainingVisualizationDuration();
  visualizationElapsedTarget = targetValue;
  if (!isAnimating || startValue >= targetValue) {
    visualizationDisplayedElapsed = targetValue;
    visualizationElapsedBeforePause = targetValue;
    updateVisualizationTime();
    return;
  }
  let startedAt = null;
  const tick = (timestamp) => {
    if (startedAt === null) startedAt = timestamp;
    const progress = Math.min(1, (timestamp - startedAt) / duration);
    visualizationDisplayedElapsed = startValue + (targetValue - startValue) * progress;
    visualizationElapsedBeforePause = visualizationDisplayedElapsed;
    updateVisualizationTime();
    if (progress < 1 && isAnimating) {
      visualizationTimerFrame = requestAnimationFrame(tick);
      return;
    }
    visualizationDisplayedElapsed = targetValue;
    visualizationElapsedBeforePause = targetValue;
    visualizationTimerFrame = null;
```

```
    updateVisualizationTime();
  };
  visualizationTimerFrame = requestAnimationFrame(tick);
}
function stopVisualizationTimer() {
  if (visualizationTimerFrame) {
    cancelAnimationFrame(visualizationTimerFrame);
    visualizationTimerFrame = null;
  }
}
function setVisualizationElapsedInstant(value) {
  stopVisualizationTimer();
  const safeValue = Math.max(0, Number(value) || 0);
  visualizationElapsedBeforePause = safeValue;
  visualizationElapsedTarget = safeValue;
  visualizationDisplayedElapsed = safeValue;
  updateVisualizationTime();
}
function completeVisualizationTimeAnimation() {
  stopVisualizationTimer();
  visualizationDisplayedElapsed = visualizationElapsedTarget;
  visualizationElapsedBeforePause = visualizationElapsedTarget;
  updateVisualizationTime();
}
function setupCollapsibleCategories() {
  document.querySelectorAll(".algorithm-category .category-header").forEach((header) => {
    if (header.dataset.collapseReady === "true") return;
    header.dataset.collapseReady = "true";
    header.setAttribute("role", "button");
    header.setAttribute("tabindex", "0");
    const toggle = () => {
      const category = header.closest(".algorithm-category");
      if (!category) return;
      const collapsed = category.classList.toggle("collapsed");
      header.setAttribute("aria-expanded", collapsed ? "false" : "true");
    };
    header.addEventListener("click", toggle);
    header.addEventListener("keydown", (event) => {
      if (event.key === "Enter" || event.key === " ") {
        event.preventDefault();
        toggle();
      }
    });
  });
}
function clampNumber(value, min, max, fallback) {
  const parsed = parseInt(value, 10);
  const safeValue = Number.isFinite(parsed) ? parsed : fallback;
  return Math.min(max, Math.max(min, safeValue));
}
function syncSizeControls(value, shouldGenerateNewData) {
  const sizeInput = document.getElementById("sizeInput");
```

```
  const sizeNumber = document.getElementById("sizeNumber");
  const display = document.getElementById("sizeDisplay");
  const size = clampNumber(value, 5, 200, 50);
  if (sizeInput) sizeInput.value = String(size);
  if (sizeNumber) sizeNumber.value = String(size);
  if (display) display.textContent = String(size);
  if (shouldGenerateNewData && currentAlgorithmType) {
    customArrayData = null;
    customArrayFileName = "";
    customSearchTarget = null;
    updateDataImportStatus();
    generateNewData();
  }
}
function syncSpeedControls(value) {
  const speedInput = document.getElementById("speedInput");
  const speedNumber = document.getElementById("speedNumber");
  const display = document.getElementById("speedDisplay");
  const speed = clampNumber(value, 1, 100, 50);
  if (speedInput) speedInput.value = String(speed);
  if (speedNumber) speedNumber.value = String(speed);
  if (display) display.textContent = speed + "%";
  animationSpeed = getAnimationDelay(speed);
  if (isAnimating) {
    rescheduleAnimationAfterSpeedChange();
    startVisualizationTimer();
  } else {
    updateVisualizationTime();
  }
}
function setupEventListeners() {
  const sizeInput = document.getElementById("sizeInput");
  const sizeNumber = document.getElementById("sizeNumber");
  const speedInput = document.getElementById("speedInput");
  const speedNumber = document.getElementById("speedNumber");
  const dataFileInput = document.getElementById("dataFileInput");
  const importDataButton = document.getElementById("importDataButton");
  if (sizeInput) {
    sizeInput.addEventListener("input", (e) => syncSizeControls(e.target.value, false));
    sizeInput.addEventListener("change", (e) => syncSizeControls(e.target.value, true));
  }
  if (sizeNumber) {
    sizeNumber.addEventListener("change", (e) => syncSizeControls(e.target.value, true));
    sizeNumber.addEventListener("keydown", (e) => {
      if (e.key === "Enter") e.currentTarget.blur();
    });
  }
  if (speedInput) {
    speedInput.addEventListener("input", (e) => syncSpeedControls(e.target.value));
  }
  if (speedNumber) {
    speedNumber.addEventListener("change", (e) => syncSpeedControls(e.target.value));
```

```
    speedNumber.addEventListener("keydown", (e) => {
      if (e.key === "Enter") e.currentTarget.blur();
    });
  }
  if (importDataButton && dataFileInput) {
    importDataButton.addEventListener("click", () => dataFileInput.click());
    dataFileInput.addEventListener("change", handleDataFileUpload);
  }
}
function initializeUI() {
  const speedInput = document.getElementById("speedInput");
  const sizeInput = document.getElementById("sizeInput");
  syncSpeedControls(speedInput ? speedInput.value : 50);
  syncSizeControls(sizeInput ? sizeInput.value : 50, false);
  updateImportControlsForAlgorithm(currentAlgorithmType);

  if (visualizer) {
    visualizer.clear();
  }
}
function openAlgorithmPage(type) {
  const href = `algorithm-info.html?algorithm=${encodeURIComponent(type)}`;
  if (window.navigateWithTransition) {
    window.navigateWithTransition(href);
  } else {
    window.location.href = href;
  }
}
function getMainVisualizationSections() {
  return [
    document.querySelector(".control-section"),
    document.querySelector(".visualization-section"),
    document.querySelector(".info-section"),
  ].filter(Boolean);
}
function showVisualizationMode() {
  getMainVisualizationSections().forEach((section) => {
    section.classList.remove("is-hidden");
  });
  const theorySection = document.getElementById("dataStructuresSection");
  if (theorySection) {
    theorySection.classList.add("is-hidden");
  }
  document.querySelectorAll(".data-structure-item").forEach((button) => {
    button.classList.remove("active");
  });
  activeDataStructureScrollId = "";
}
function selectDataStructureSection(sectionId) {
  pauseAnimation();
  getMainVisualizationSections().forEach((section) => {
    section.classList.add("is-hidden");
```

```
  });
  document.querySelectorAll(".algo-item").forEach((button) => {
    button.classList.remove("active");
  });
  const theorySection = document.getElementById("dataStructuresSection");
  if (theorySection) {
    theorySection.classList.remove("is-hidden");
  }
  const activeButton = document.querySelector(`.data-structure-item[data-structure-
section="${sectionId}"]`);
  if (activeButton) {
    activeButton.classList.add("active");
  }
  const target = document.getElementById(sectionId || "ds-overview");
  if (target) {
    window.setTimeout(() => {
      target.scrollIntoView({ behavior: "smooth", block: "start" });
      window.setTimeout(updateActiveDataStructureByScroll, 180);
    }, 40);
  } else {
    updateActiveDataStructureByScroll();
  }
}
window.selectDataStructureSection = selectDataStructureSection;
function updateActiveDataStructureByScroll() {
  const theorySection = document.getElementById("dataStructuresSection");
  if (!theorySection || theorySection.classList.contains("is-hidden")) return;
  const sections = Array.from(theorySection.querySelectorAll(".data-structures-intro, .data-topic"));
  if (sections.length === 0) return;
  const checkpoint = Math.min(window.innerHeight * 0.35, 260);
  let activeSection = sections[0];
  sections.forEach((section) => {
    const rect = section.getBoundingClientRect();
    if (rect.top <= checkpoint && rect.bottom > 120) {
      activeSection = section;
    }
  });
  document.querySelectorAll(".data-structure-item").forEach((button) => {
    button.classList.toggle("active", button.dataset.structureSection === activeSection.id);
  });
  if (activeDataStructureScrollId !== activeSection.id) {
    activeDataStructureScrollId = activeSection.id;
    const activeButton = document.querySelector(`.data-structure-item[data-structure-
section="${activeSection.id}"]`);
    if (activeButton) {
      activeButton.scrollIntoView({ block: "nearest", inline: "nearest" });
    }
  }
}
window.addEventListener("scroll", updateActiveDataStructureByScroll, { passive: true });
window.addEventListener("resize", updateActiveDataStructureByScroll);
function usesArrayData(type) {
```

```
  return ["bubble", "insertion", "merge", "quick", "heap", "linear", "binary", "jump",
"interpolation"].includes(type);
}
function updateDataImportStatus() {
  const status = document.getElementById("dataImportStatus");
  if (!status) return;
  if (!usesArrayData(currentAlgorithmType)) {
    status.textContent = "Завантаження TXT доступне для алгоритмів сортування та пошуку.";
    return;
  }
  if (Array.isArray(customArrayData) && customArrayData.length > 0) {
    status.textContent = `Завантажено: ${customArrayFileName} (${customArrayData.length}
чисел).`;
  } else {
    status.textContent = "Можна завантажити TXT-файл із числами.";
  }
}

function updateImportControlsForAlgorithm(type) {
  const control = document.getElementById("dataImportControl");
  if (control) {
    control.style.display = usesArrayData(type) ? "" : "none";
  }
  updateDataImportStatus();
}

async function handleDataFileUpload(event) {
  const input = event.currentTarget;
  const file = input.files && input.files[0];
  if (!file) return;
  if (!usesArrayData(currentAlgorithmType)) {
    alert("TXT-файл можна завантажувати для алгоритмів сортування та пошуку.");
    input.value = "";
    return;
  }

  try {
    const text = await window.AlgoVizDataFile.readTextFile(file);
    const parsed = window.AlgoVizDataFile.parseTextData(text, { minLength: 2, maxLength: 200
});
    customArrayData = parsed.array;
    customArrayFileName = file.name || "дані.txt";
    customSearchTarget = parsed.target;
    syncSizeControls(customArrayData.length, false);

    const targetInput = document.getElementById("searchTarget");
    if (targetInput && customSearchTarget !== null) {
      targetInput.value = String(customSearchTarget);
    }

    updateDataImportStatus();
    generateNewData();
```

```
  } catch (error) {
    alert(error.message || "Не вдалося завантажити дані з файлу.");
  } finally {
    input.value = "";
  }
}

function generateRandomData() {
  customArrayData = null;
  customArrayFileName = "";
  customSearchTarget = null;
  updateDataImportStatus();
  generateNewData();
}

function getArrayDataForCurrentRun(size) {
  if (usesArrayData(currentAlgorithmType) && Array.isArray(customArrayData) &&
customArrayData.length > 0) {
    return customArrayData.slice();
  }

  return generateRandomArray(size);
}

function getAlgorithmClassMap() {
  return {
    bubble: window.BubbleSort || BubbleSort,
    insertion: window.InsertionSort || InsertionSort,
    merge: window.MergeSort || MergeSort,
    quick: window.QuickSort || QuickSort,
    heap: window.HeapSort || HeapSort,
    linear: window.LinearSearch || LinearSearch,
    binary: window.BinarySearch || BinarySearch,
    jump: window.JumpSearch || JumpSearch,
    interpolation: window.InterpolationSearch || InterpolationSearch,
    bfs: window.BFS || BFS,
    dfs: window.DFS || DFS,
    dijkstra: window.Dijkstra || Dijkstra,
    fib: window.Fibonacci || Fibonacci,
    lcs: window.LCS || LCS,
  };
}

function getAlgorithmInfo(type) {
  const classMap = getAlgorithmClassMap();
  const cls = classMap[type];

  return cls && cls.info ? cls.info : null;
}

function selectAlgorithm(type) {
  showVisualizationMode();
```

```
  currentAlgorithmType = type;
  isAnimating = false;
  stopVisualizationTimer();
  currentStepIndex = 0;
  currentAlgorithmResult = null;
  currentAlgorithmHasStarted = false;
  currentAlgorithmIsFinished = false;

  if (animationTimeout) {
    clearTimeout(animationTimeout);
    animationTimeout = null;
  }

  const info = getAlgorithmInfo(type);

  updateAlgorithmInfo(info);
  showSettings(type);
  activateSelectedButton(type);
  generateNewData();
}

function updateAlgorithmInfo(info) {
  const nameEl = document.getElementById("algorithmName");
  const descEl = document.getElementById("algorithmDesc");
  const complexityEl = document.getElementById("algorithmComplexity");
  const instructionEl = document.getElementById("algorithmInstruction");
  const stepsList = document.getElementById("algorithmSteps");

  if (!nameEl || !descEl || !complexityEl || !instructionEl || !stepsList) {
    return;
  }

  if (!info) {
    nameEl.textContent = "Виберіть алгоритм";
    descEl.textContent = "";
    complexityEl.textContent = "—";
    instructionEl.style.display = "none";
    stepsList.innerHTML = "";
    return;
  }

  nameEl.textContent = info.name || "Алгоритм";
  descEl.textContent = info.description || "";
  complexityEl.textContent = info.complexity || "—";

  stepsList.innerHTML = "";

  if (Array.isArray(info.steps) && info.steps.length > 0) {
    info.steps.forEach((step) => {
      const li = document.createElement("li");
      li.textContent = step;
      stepsList.appendChild(li);
```

```
    });

    instructionEl.style.display = "block";
  } else {
    instructionEl.style.display = "none";
  }
}

function activateSelectedButton(type) {
  document.querySelectorAll(".algo-item").forEach((btn) => {
    btn.classList.remove("active");
  });

  const targetButton = document.querySelector(`.algo-item[data-algorithm="${type}"]`);

  if (targetButton) {
    targetButton.classList.add("active");
  }
}

function showSettings(type) {
  const settingsSection = document.getElementById("algorithmSettings");
  const settingsContent = document.getElementById("settingsContent");
  const arraySizeControl = document.getElementById("arraySizeControl");
  const newDataButton = document.getElementById("newDataButton");

  if (!settingsSection || !settingsContent) {
    return;
  }

  const usesArraySize = usesArrayData(type);
  if (arraySizeControl) {
    arraySizeControl.style.display = usesArraySize ? "" : "none";
  }
  updateImportControlsForAlgorithm(type);

  if (newDataButton) {
    newDataButton.innerHTML =
      type === "bfs" || type === "dfs" || type === "dijkstra"
        ? '<i class="fas fa-random"></i> Оновити граф'
        : '<i class="fas fa-random"></i> Нові дані';
  }

  settingsContent.innerHTML = "";
  settingsSection.style.display = "none";

  if (["linear", "binary", "jump", "interpolation"].includes(type)) {
    settingsSection.style.display = "block";

    const jumpSettings =
      type === "jump"
        ? `
```

```
      <div>
        <label for="jumpStepMode">Розмір стрибка:</label>
        <select id="jumpStepMode">
          <option value="auto" selected>Автоматично (√n)</option>
          <option value="manual">Вручну</option>
        </select>
      </div>

      <div id="jumpStepValueWrap" style="display: none;">
        <label for="jumpStepValue">Значення стрибка:</label>
        <input type="number" id="jumpStepValue" min="1" max="120" value="7">
      </div>
    `
    : "";

  settingsContent.innerHTML = `
    <div class="method-settings-grid">
      <div>
        <label for="searchTarget">Цільове значення для пошуку:</label>
        <input type="number" id="searchTarget" min="1" max="100" value="50">
      </div>

      ${jumpSettings}
    </div>
  `;

  const searchTargetInput = document.getElementById("searchTarget");
  if (searchTargetInput && customSearchTarget !== null) {
    searchTargetInput.value = String(customSearchTarget);
  }
  searchTargetInput.addEventListener("change", generateNewData);

  const jumpStepMode = document.getElementById("jumpStepMode");
  const jumpStepValue = document.getElementById("jumpStepValue");
  const jumpStepValueWrap = document.getElementById("jumpStepValueWrap");

  if (jumpStepMode) {
    jumpStepMode.addEventListener("change", () => {
      if (jumpStepValueWrap) {
        jumpStepValueWrap.style.display = jumpStepMode.value === "manual" ? "" : "none";
      }
      generateNewData();
    });
  }

  if (jumpStepValue) {
    jumpStepValue.addEventListener("change", generateNewData);
  }
} else if (type === "lcs") {
  settingsSection.style.display = "block";

  settingsContent.innerHTML = `
```

```
    <div class="method-settings-grid">
      <div>
        <label for="lcsStr1">Рядок 1:</label>
        <input type="text" id="lcsStr1" value="ABCDGH">
      </div>

      <div>
        <label for="lcsStr2">Рядок 2:</label>
        <input type="text" id="lcsStr2" value="AEDFHR">
      </div>
    </div>

    <p class="settings-note">
      Візуалізація показує заповнення DP-матриці. Жовта клітинка – поточний крок.
    </p>

    <div class="step-controls">
      <button type="button" class="btn btn-secondary" onclick="previousStep()">Попередній
крок</button>
      <button type="button" class="btn btn-secondary" onclick="nextStep()">Наступний
крок</button>
    </div>
  `;

  document
    .getElementById("lcsStr1")
    .addEventListener("input", generateNewData);
  document
    .getElementById("lcsStr2")
    .addEventListener("input", generateNewData);
 } else if (type === "fib") {
  settingsSection.style.display = "block";

  settingsContent.innerHTML = `
    <label for="fibN">n – номер числа Фібоначчі:</label>
    <input type="number" id="fibN" min="0" max="50" value="20">

    <p class="settings-note">
      Масив dp[] будується поступово: кожне нове значення дорівнює сумі двох попередніх.
    </p>

    <div class="step-controls">
      <button type="button" class="btn btn-secondary" onclick="previousStep()">Попередній
крок</button>
      <button type="button" class="btn btn-secondary" onclick="nextStep()">Наступний
крок</button>
    </div>
  `;

  document.getElementById("fibN").addEventListener("change", generateNewData);
 } else if (type === "bfs" || type === "dfs" || type === "dijkstra") {
  settingsSection.style.display = "block";
```

```
    const endNodeField =
      type === "dijkstra"
        ? `
          <div>
            <label for="graphEndNode">Кінцева вершина:</label>
            <input type="number" id="graphEndNode" min="0" max="49" value="6">
          </div>
        `
        : "";

    settingsContent.innerHTML = `
      <div class="method-settings-grid">
        <div>
          <label for="graphStartNode">Початкова вершина:</label>
          <input type="number" id="graphStartNode" min="0" max="49" value="0">
        </div>

        ${endNodeField}

        <div>
          <label for="graphNodeCount">Кількість вершин:</label>
          <input type="number" id="graphNodeCount" min="3" max="50" value="7">
        </div>
      </div>

      <p class="settings-note">
        Для BFS показується черга обходу, для DFS – рух у глибину, для Dijkstra – поточні найкоротші відстані.
      </p>

      <div class="step-controls">
        <button type="button" class="btn btn-secondary" onclick="previousStep()">Попередній крок</button>
        <button type="button" class="btn btn-secondary" onclick="nextStep()">Наступний крок</button>
      </div>
    `;

    ["graphStartNode", "graphEndNode", "graphNodeCount"].forEach((id) => {
      const input = document.getElementById(id);
      if (input) {
        input.addEventListener("change", generateNewData);
      }
    });
  }
}

function generateRandomArray(size) {
  return Array.from(
    { length: size },
    () => Math.floor(Math.random() * 100) + 1,
```

```
  );
}

function getDemoGraphData() {
  currentGraphLayoutSeed = Math.floor(Math.random() * 1000000) + 1;
  const countInput = document.getElementById("graphNodeCount");
  const nodeCount = Math.min(50, Math.max(3, countInput ? parseInt(countInput.value, 10) || 7 :
7));
  const nodes = Array.from({ length: nodeCount }, (_, i) => i);

  const edgeSet = new Set();
  const addEdge = (from, to) => {
    if (from === to || from < 0 || to < 0 || from >= nodeCount || to >= nodeCount) return;
    const a = Math.min(from, to);
    const b = Math.max(from, to);
    edgeSet.add(`${a}-${b}`);
  };

  for (let i = 0; i < nodeCount - 1; i++) {
    addEdge(i, i + 1);
  }

  const extraEdges = Math.max(nodeCount, Math.floor(nodeCount * 1.15));
  for (let i = 0; i < extraEdges; i++) {
    const from = Math.floor(Math.random() * nodeCount);
    let to = Math.floor(Math.random() * nodeCount);

    if (from === to) {
      to = (to + 1 + Math.floor(Math.random() * Math.max(1, nodeCount - 1))) % nodeCount;
    }

    addEdge(from, to);
  }

  const edges = Array.from(edgeSet).map((edge) => edge.split("-").map(Number));

  const weights = {};
  edges.forEach(([from, to]) => {
    const weight = Math.floor(Math.random() * 9) + 1;
    weights[`${from}-${to}`] = weight;
    weights[`${to}-${from}`] = weight;
  });

  return {
    nodes,
    edges,
    weights,
  };
}

function generateNewData() {
  if (!currentAlgorithmType) {
```

```
    return;
  }

  const sizeInput = document.getElementById("sizeInput");
  const size = sizeInput ? parseInt(sizeInput.value, 10) : 50;
  try {
    const array = getArrayDataForCurrentRun(size);
    currentAlgorithm = createAlgorithmInstance(currentAlgorithmType, array);
    currentAlgorithmResult = runAlgorithm(currentAlgorithmType, currentAlgorithm, array);
    resetVisualization();
  } catch (error) {
    console.error(error);
    alert("Помилка при ініціалізації алгоритму: " + error.message);
  }
}
function createAlgorithmInstance(type, commonArray) {
  switch (type) {
    case "bubble":
      return new BubbleSort(commonArray);
    case "insertion":
      return new InsertionSort(commonArray);
    case "merge":
      return new MergeSort(commonArray);
    case "quick":
      return new QuickSort(commonArray);
    case "heap":
      return new HeapSort(commonArray);
    case "linear":
      return new LinearSearch(commonArray);
    case "binary":
      return new BinarySearch([...commonArray].sort((a, b) => a - b));
    case "jump":
      return new JumpSearch([...commonArray].sort((a, b) => a - b));
    case "interpolation":
      return new InterpolationSearch([...commonArray].sort((a, b) => a - b));
    case "bfs": {
      const { nodes, edges } = getDemoGraphData();
      return new BFS(nodes, edges);
    }
    case "dfs": {
      const { nodes, edges } = getDemoGraphData();
      return new DFS(nodes, edges);
    }
    case "dijkstra": {
      const { nodes, edges, weights } = getDemoGraphData();
      return new Dijkstra(nodes, edges, weights);
    }
    case "fib":
      return new Fibonacci();
    case "lcs":
      return new LCS();
    default:
```

```
    throw new Error("Невідомий алгоритм: " + type);
  }
}
function runAlgorithm(type, algorithm, commonArray) {
  if (!algorithm || typeof algorithm.execute !== "function") {
    throw new Error("Алгоритм не має методу execute()");
  }
  if (["bubble", "insertion", "merge", "quick", "heap"].includes(type)) {
    return algorithm.execute();
  }
  if (type === "linear") {
    const input = document.getElementById("searchTarget");
    const target = input ? parseInt(input.value, 10) : commonArray[0];
    return algorithm.execute(target);
  }
  if (["binary", "jump", "interpolation"].includes(type)) {
    const input = document.getElementById("searchTarget");
    const target = input ? parseInt(input.value, 10) : commonArray[0];
    if (type === "jump") {
      const modeInput = document.getElementById("jumpStepMode");
      const stepInput = document.getElementById("jumpStepValue");
      const autoJump = !modeInput || modeInput.value !== "manual";
      const jumpStep = stepInput ? parseInt(stepInput.value, 10) : undefined;
      return algorithm.execute(target, { autoJump, jumpStep });
    }
    return algorithm.execute(target);
  }
  if (type === "bfs" || type === "dfs" || type === "dijkstra") {
    const startInput = document.getElementById("graphStartNode");
    const startNode = startInput ? parseInt(startInput.value, 10) || 0 : 0;
    if (type === "dijkstra") {
      const endInput = document.getElementById("graphEndNode");
      const endNode = endInput ? parseInt(endInput.value, 10) : null;
      return algorithm.execute(startNode, endNode);
    }
    return algorithm.execute(startNode);
  }
  if (type === "fib") {
    const input = document.getElementById("fibN");
    const n = input ? parseInt(input.value, 10) : 20;
    return algorithm.execute(n);
  }
  if (type === "lcs") {
    const input1 = document.getElementById("lcsStr1");
    const input2 = document.getElementById("lcsStr2");
    const str1 = input1 ? input1.value : "ABCDGH";
    const str2 = input2 ? input2.value : "AEDFHR";
    return algorithm.execute(str1, str2);
  }
  throw new Error("Невідомий алгоритм: " + type);
}
function startAnimation() {
```

```
  if (
    !currentAlgorithm ||
    !currentAlgorithm.steps ||
    currentAlgorithm.steps.length === 0
  ) {
    alert("Виберіть алгоритм і зачекайте його ініціалізації.");
    return;
  }
  if (isAnimating) {
    return;
  }
  if (animationTimeout) {
    clearTimeout(animationTimeout);
    animationTimeout = null;
  }
  if (currentStepIndex >= currentAlgorithm.steps.length) {
    currentStepIndex = 0;
    setVisualizationElapsedInstant(0);
    currentVisualizationStepDuration = 0;
    visualizationStepStartedAt = 0;
    currentAlgorithmIsFinished = false;
  }
  currentAlgorithmHasStarted = true;
  isAnimating = true;
  visualizationStartedAt = 0;
  startVisualizationTimer();
  animateSteps();
}
function animateSteps() {
  if (
    !isAnimating ||
    !currentAlgorithm ||
    currentStepIndex >= currentAlgorithm.steps.length
  ) {
    isAnimating = false;
    animationTimeout = null;
    visualizationStepStartedAt = 0;
    if (currentAlgorithm && currentAlgorithm.steps && currentStepIndex >=
currentAlgorithm.steps.length) {
      currentAlgorithmIsFinished = currentAlgorithmHasStarted;
    }
    completeVisualizationTimeAnimation();
    updateVisualizationTime();
    return;
  }
  const step = currentAlgorithm.steps[currentStepIndex];
  currentVisualizationStepDuration = animationSpeed;
  renderStep(step);
  updateStatistics(step, currentStepIndex + 1);
  currentStepIndex++;
  scheduleNextAnimationStep(animationSpeed);
}
```

```
function renderStep(step) {
  if (!step || !visualizer) {
    return;
  }
  if (step.matrix && Array.isArray(step.matrix)) {
    visualizer.drawMatrix(step.matrix, {
      currentCell: step.currentCell,
      sourceCells: step.sourceCells || [],
      str1: step.str1,
      str2: step.str2,
      lcs: step.lcs || "",
      explanation: step.explanation || "",
    });
    return;
  }
  if (step.dp && Array.isArray(step.dp)) {
    visualizer.drawDPArray(step.dp, {
      current: step.current,
      used: step.used || [],
      formula: step.formula || "",
      result: step.result,
    });
    return;
  }
  if (step.array && Array.isArray(step.array)) {
    visualizer.drawBars(step.array, {
      sorted: step.sorted || [],
      comparing: step.comparing || [],
      current: step.current,
      range: step.range || [],
      target: step.target,
      message: step.message || "",
      foundIndex: step.foundIndex,
      notFound: step.notFound || false,
      animationDuration: currentVisualizationStepDuration || animationSpeed,
    });
    return;
  }
  if (step.nodes && step.edges) {
    visualizer.drawGraph(step.nodes, step.edges, {
      visited: step.visited || [],
      current: step.current,
      queue: step.queue || [],
      stack: step.stack || [],
      distances: step.distances || [],
      endNode: step.endNode,
      weights: currentAlgorithmType === "dijkstra" ? step.weights : undefined,
      layoutSeed: currentGraphLayoutSeed,
    });
  }
}
function getVisualizationElapsed() {
```

```
  return visualizationDisplayedElapsed;
}
function formatDuration(ms) {
  const value = Math.max(0, Math.round(Number(ms) || 0));
  if (value < 1000) return `${value}ms`;
  return `${(value / 1000).toFixed(value < 10000 ? 2 : 1)}s`;
}
function updateVisualizationTime() {
  const value = formatDuration(getVisualizationElapsed());
  setText("timeElapsed", value);
}
function updateStatistics(step, stepNumber) {
  if (!step || !step.stats) {
    return;
  }
  setText("operationCount", step.stats.operations);
  setText("comparisonCount", step.stats.comparisons);
  setText("swapCount", step.stats.swaps);
  setText("stepCount", stepNumber);
  updateVisualizationTime();
}
function setText(id, value) {
  const element = document.getElementById(id);
  if (element) {
    element.textContent = value;
  }
}
function pauseAnimation() {
  visualizationStartedAt = 0;
  isAnimating = false;
  stopVisualizationTimer();
  if (animationTimeout) {
    clearTimeout(animationTimeout);
    animationTimeout = null;
  }
  visualizationStepStartedAt = 0;
  updateVisualizationTime();
}
function resetVisualization() {
  pauseAnimation();
  currentStepIndex = 0;
  visualizationStartedAt = 0;
  setVisualizationElapsedInstant(0);
  currentVisualizationStepDuration = 0;
  visualizationStepStartedAt = 0;
  currentAlgorithmHasStarted = false;
  currentAlgorithmIsFinished = false;
  setText("operationCount", "0");
  setText("comparisonCount", "0");
  setText("swapCount", "0");
  setText("stepCount", "0");
  if (
```

```
    currentAlgorithm &&
    currentAlgorithm.steps &&
    currentAlgorithm.steps.length > 0
  ) {
    renderStep(currentAlgorithm.steps[0]);
  } else if (visualizer) {
    visualizer.clear();
  }
}
function nextStep() {
  pauseAnimation();
  if (!currentAlgorithm || !currentAlgorithm.steps || currentAlgorithm.steps.length === 0) {
    return;
  }
  currentAlgorithmHasStarted = true;
  currentStepIndex = Math.min(currentStepIndex + 1, currentAlgorithm.steps.length - 1);
  currentAlgorithmIsFinished = currentStepIndex >= currentAlgorithm.steps.length - 1;
  const step = currentAlgorithm.steps[currentStepIndex];
  setVisualizationElapsedInstant(getAlgorithmDisplayTimeFromStep(step));
  renderStep(step);
  updateStatistics(step, currentStepIndex + 1);
}
function previousStep() {
  pauseAnimation();
  if (!currentAlgorithm || !currentAlgorithm.steps || currentAlgorithm.steps.length === 0) {
    return;
  }
  currentStepIndex = Math.max(currentStepIndex - 1, 0);
  currentAlgorithmHasStarted = currentStepIndex > 0;
  currentAlgorithmIsFinished = false;
  const step = currentAlgorithm.steps[currentStepIndex];
  setVisualizationElapsedInstant(getAlgorithmDisplayTimeFromStep(step));
  renderStep(step);
  updateStatistics(step, currentStepIndex + 1);
}
function escapeHtml(value) {
  return String(value ?? "")
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/"/g, "&quot;")
    .replace(/'/g, "&#039;");
}
function formatReportValue(value, limit = 160) {
  if (Array.isArray(value)) {
    const items = value.slice(0, limit).map((item) => Array.isArray(item) ? `[${item.join(", ")}]` :
String(item));
    const suffix = value.length > limit ? ` … (+${value.length - limit})` : "";
    return items.join(", ") + suffix;
  }
  if (value && typeof value === "object") {
    return JSON.stringify(value, null, 2);
```

```
  }
  return String(value ?? "—");
}
function getStepSnapshot(step) {
  if (!step) return "—";
  if (Array.isArray(step.array)) {
    return `Масив: ${formatReportValue(step.array)}`;
  }
  if (Array.isArray(step.dp)) {
    return `DP-масив: ${formatReportValue(step.dp)}`;
  }
  if (Array.isArray(step.matrix)) {
    const rows = step.matrix.slice(0, 12).map((row) => `[${formatReportValue(row, 30)}]`);
    const suffix = step.matrix.length > 12 ? `\n… (+${step.matrix.length - 12} ряд.)` : "";
    return `Матриця:\n${rows.join("\n")}${suffix}`;
  }
  if (Array.isArray(step.nodes) && Array.isArray(step.edges)) {
    const parts = [
      `Вершини: ${formatReportValue(step.nodes)}`,
      `Ребра: ${formatReportValue(step.edges, 80)}`,
    ];
    if (Array.isArray(step.visited)) parts.push(`Відвідано: ${formatReportValue(step.visited)}`);
    if (Array.isArray(step.distances)) parts.push(`Відстані: ${formatReportValue(step.distances)}`);
    if (step.current !== undefined && step.current !== null) parts.push(`Поточна вершина:
${step.current}`);
    return parts.join("\n");
  }
  return formatReportValue(step);
}
function getInitialDataSnapshot() {
  if (!currentAlgorithm) return "—";

  if (Array.isArray(currentAlgorithm.data)) {
    return `Масив: ${formatReportValue(currentAlgorithm.data)}`;
  }
  if (currentAlgorithm.data && Array.isArray(currentAlgorithm.data.nodes) &&
Array.isArray(currentAlgorithm.data.edges)) {
    return [
      `Вершини: ${formatReportValue(currentAlgorithm.data.nodes)}`,
      `Ребра: ${formatReportValue(currentAlgorithm.data.edges, 80)}`,
    ].join("\n");
  }
  if (currentAlgorithmType === "fib") {
    const fibN = document.getElementById("fibN");
    return `Обчислення числа Фібоначчі для n = ${fibN ? fibN.value : "—"}.`;
  }
  if (currentAlgorithmType === "lcs") {
    const lcsStr1 = document.getElementById("lcsStr1");
    const lcsStr2 = document.getElementById("lcsStr2");
    return `Рядок 1: ${lcsStr1 ? lcsStr1.value : "—"}\nРядок 2: ${lcsStr2 ? lcsStr2.value : "—"}`;
  }
  return formatReportValue(currentAlgorithm.data);
```

```
}
function getFinalResultSummary() {
  const type = currentAlgorithmType;
  const result = currentAlgorithmResult;
  if (result === null || result === undefined) return "—";
  if (["linear", "binary", "jump", "interpolation"].includes(type)) {
    const targetInput = document.getElementById("searchTarget");
    const target = targetInput ? targetInput.value : "—";
    return Number(result) >= 0
      ? `Шукане значення ${target} знайдено на позиції ${result}.`
      : `Шукане значення ${target} не знайдено.`;
  }
  if (type === "fib") return `Результат: ${formatReportValue(result)}`;
  if (type === "lcs") return `Найдовша спільна підпослідовність: ${result || "—"}`;
  if (type === "bfs" || type === "dfs") return `Порядок відвідування:
${formatReportValue(result)}`;
  if (type === "dijkstra") return `Відстані від початкової вершини:
${formatReportValue(result)}`;
  if (Array.isArray(result)) return `Результат: ${formatReportValue(result)}`;
  return formatReportValue(result);
}
function getSettingsSummary() {
  const sizeInput = document.getElementById("sizeInput");
  const speedInput = document.getElementById("speedInput");
  const arraySizeControl = document.getElementById("arraySizeControl");
  const settings = [`Швидкість анімації: ${speedInput ? speedInput.value : "—"}%`];
  if (sizeInput && (!arraySizeControl || arraySizeControl.style.display !== "none")) {
    settings.unshift(`Розмір масиву: ${sizeInput.value}`);
  }
  const searchTarget = document.getElementById("searchTarget");
  if (searchTarget) settings.push(`Ціль пошуку: ${searchTarget.value}`);
  const graphStartNode = document.getElementById("graphStartNode");
  if (graphStartNode) settings.push(`Початкова вершина: ${graphStartNode.value}`);
  const graphEndNode = document.getElementById("graphEndNode");
  if (graphEndNode) settings.push(`Кінцева вершина: ${graphEndNode.value}`);
  const graphNodeCount = document.getElementById("graphNodeCount");
  if (graphNodeCount) settings.push(`Кількість вершин: ${graphNodeCount.value}`);
  const fibN = document.getElementById("fibN");
  if (fibN) settings.push(`n для Фібоначчі: ${fibN.value}`);
  const lcsStr1 = document.getElementById("lcsStr1");
  const lcsStr2 = document.getElementById("lcsStr2");
  if (lcsStr1 && lcsStr2) settings.push(`Рядки LCS: "${lcsStr1.value}" та "${lcsStr2.value}"`);
  return settings;
}
function getInitialStepForReport() {
  if (!currentAlgorithm) return null;
  if (Array.isArray(currentAlgorithm.data)) {
    return { array: [...currentAlgorithm.data] };
  }
  if (currentAlgorithm.data && Array.isArray(currentAlgorithm.data.nodes) &&
Array.isArray(currentAlgorithm.data.edges)) {
    return {
```

```
      nodes: [...currentAlgorithm.data.nodes],
      edges: currentAlgorithm.data.edges.map((edge) => Array.isArray(edge) ? [...edge] : edge),
      weights: currentAlgorithm.data.weights || undefined,
    };
  }
  if (Array.isArray(currentAlgorithm.steps) && currentAlgorithm.steps.length > 0) {
    return currentAlgorithm.steps[0];
  }
  return null;
}
function getVisibleStepForReport() {
  if (!currentAlgorithm || !Array.isArray(currentAlgorithm.steps) || currentAlgorithm.steps.length
=== 0) {
    return null;
  }
  const steps = currentAlgorithm.steps;
  const index = Math.min(Math.max(0, currentStepIndex > 0 ? currentStepIndex - 1 : 0),
steps.length - 1);
  return steps[index] || steps[0];
}
function captureRenderedStep(step) {
  if (!visualizer || !visualizer.canvas || !step) return "";
  try {
    if (typeof visualizer.stopAnimation === "function") {
      visualizer.stopAnimation();
    }
    if ("_forceInstantBars" in visualizer) {
      visualizer._forceInstantBars = true;
    }
    renderStep(step);
    return visualizer.canvas.toDataURL("image/png");
  } catch (error) {
    console.error(error);
    return "";
  }
}
function captureReportImages(initialStep, secondStep) {
  const result = { initial: "", second: "" };
  if (!visualizer || !visualizer.canvas) {
    return result;
  }

  const visibleStep = getVisibleStepForReport();
  const previousBars = Array.isArray(visualizer.lastBars) ? visualizer.lastBars.slice() : [];
  const previousForce = visualizer._forceInstantBars;
  result.initial = captureRenderedStep(initialStep);
  if (secondStep) {
    result.second = captureRenderedStep(secondStep);
  }
  visualizer.lastBars = previousBars;
  if ("_forceInstantBars" in visualizer) {
    visualizer._forceInstantBars = true;
```

```
  }
  if (visibleStep) {
    renderStep(visibleStep);
  }
  if ("_forceInstantBars" in visualizer) {
    visualizer._forceInstantBars = previousForce;
  }
  return result;
}
function getReportImageMarkup(title, src) {
  if (!src) {
    return `<div class="image-card"><h3>${escapeHtml(title)}</h3><p>Знімок
недоступний.</p></div>`;
  }
  return `<div class="image-card"><h3>${escapeHtml(title)}</h3><img src="${src}"
alt="${escapeHtml(title)}"></div>`;
}
function exportResultsToPdf() {
  if (!currentAlgorithm || !currentAlgorithm.steps || currentAlgorithm.steps.length === 0) {
    alert("Спочатку виберіть алгоритм та створіть дані для візуалізації.");
    return;
  }
  pauseAnimation();
  const info = getAlgorithmInfo(currentAlgorithmType) || {};
  const steps = currentAlgorithm.steps;
  const initialStep = getInitialStepForReport() || steps[0];
  const hasStarted = currentAlgorithmHasStarted || currentStepIndex > 0;
  const hasCompleted = currentAlgorithmIsFinished || currentStepIndex >= steps.length;
  const visibleIndex = Math.min(Math.max(0, currentStepIndex > 0 ? currentStepIndex - 1 : 0),
steps.length - 1);
  const visibleStep = steps[visibleIndex];
  const reportStep = hasCompleted ? steps[steps.length - 1] : hasStarted ? visibleStep : null;
  const reportStats = reportStep && reportStep.stats ? reportStep.stats : { operations: 0,
comparisons: 0, swaps: 0 };
  const currentDate = new Date().toLocaleString("uk-UA");
  const settingsItems = getSettingsSummary().map((item) =>
`<li>${escapeHtml(item)}</li>`).join("");
  const reportTime = hasCompleted
    ? formatDuration(getAlgorithmDisplayTimeFromStep(steps[steps.length - 1]))
    : formatDuration(visualizationElapsedBeforePause);
  const reportImages = captureReportImages(initialStep, reportStep);
  const imageCards = [getReportImageMarkup("Початковий стан", reportImages.initial)];
  if (hasCompleted) {
    imageCards.push(getReportImageMarkup("Стан після виконання", reportImages.second));
  } else if (hasStarted) {
    imageCards.push(getReportImageMarkup("Поточний стан", reportImages.second));
  }
  const reportImagesHtml = imageCards.join("");
  const executionHtml = hasCompleted
    ? `
  <h2>Результат після виконання</h2>
  <pre>${escapeHtml(getStepSnapshot(reportStep))}</pre>
```

```
  <p>${escapeHtml(getFinalResultSummary())}</p>
  <h2>Підсумкова статистика</h2>
  <div class="grid">
    <div class="card"><div class="label">Операцій</div><div
class="value">${escapeHtml(reportStats.operations ?? 0)}</div></div>
    <div class="card"><div class="label">Порівнянь</div><div
class="value">${escapeHtml(reportStats.comparisons ?? 0)}</div></div>
    <div class="card"><div class="label">Обмінів</div><div
class="value">${escapeHtml(reportStats.swaps ?? 0)}</div></div>
    <div class="card"><div class="label">Час алгоритму</div><div
class="value">${escapeHtml(reportTime)}</div></div>
  </div>`
    : hasStarted
      ? `
  <h2>Поточний стан</h2>
  <pre>${escapeHtml(getStepSnapshot(reportStep))}</pre>
  <p>Алгоритм ще не завершено, тому фінальний результат у звіт не додано.</p>
  <h2>Поточна статистика</h2>
  <div class="grid">
    <div class="card"><div class="label">Операцій</div><div
class="value">${escapeHtml(reportStats.operations ?? 0)}</div></div>
    <div class="card"><div class="label">Порівнянь</div><div
class="value">${escapeHtml(reportStats.comparisons ?? 0)}</div></div>
    <div class="card"><div class="label">Обмінів</div><div
class="value">${escapeHtml(reportStats.swaps ?? 0)}</div></div>
    <div class="card"><div class="label">Час алгоритму</div><div
class="value">${escapeHtml(reportTime)}</div></div>
  </div>`
      : `
  <h2>Результат</h2>
  <p>Алгоритм ще не запускався, тому у звіт додано лише початкові дані.</p>`;
  const reportHtml = `<!doctype html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <title>Звіт AlgoViz Pro</title>
  <style>
    body { font-family: Arial, sans-serif; color: #111827; line-height: 1.5; padding: 28px; }
    h1 { margin: 0 0 6px; color: #1d4ed8; }
    h2 { margin-top: 24px; color: #111827; border-bottom: 1px solid #d1d5db; padding-bottom: 6px;
}
    .meta { color: #6b7280; margin-bottom: 18px; }
    .grid { display: grid; grid-template-columns: repeat(4, 1fr); gap: 10px; margin: 14px 0; }
    .image-grid { display: grid; grid-template-columns: repeat(${imageCards.length > 1 ? 2 : 1},
minmax(0, 1fr)); gap: 14px; margin: 14px 0; }
    .card, .image-card { border: 1px solid #d1d5db; border-radius: 10px; padding: 12px; background:
#f9fafb; }
    .image-card h3 { margin: 0 0 10px; font-size: 16px; }
    .image-card img { width: 100%; height: auto; display: block; border: 1px solid #e5e7eb; border-
radius: 8px; background: #0f172a; }
    .label { color: #6b7280; font-size: 12px; text-transform: uppercase; font-weight: 700; }
    .value { font-size: 20px; font-weight: 800; margin-top: 4px; }
```

```
    pre { white-space: pre-wrap; word-break: break-word; background: #f3f4f6; border: 1px solid
#e5e7eb; border-radius: 10px; padding: 12px; }
    ul { margin-top: 8px; }
    @media print { body { padding: 0; } .no-print { display: none; } .image-grid { break-inside:
avoid; } }
    @media (max-width: 800px) { .grid, .image-grid { grid-template-columns: 1fr; } }
  </style>
</head>
<body>
  <button class="no-print" onclick="window.print()" style="float:right;padding:10px
14px;border:0;border-radius:8px;background:#2563eb;color:#fff;font-
weight:700;cursor:pointer;">Зберегти / друк PDF</button>
  <h1>Звіт AlgoViz Pro</h1>
  <div class="meta">Дата створення: ${escapeHtml(currentDate)}</div>
  <h2>Алгоритм</h2>
  <p><strong>${escapeHtml(info.name || "Алгоритм")}</strong></p>
  <p>${escapeHtml(info.description || "")}</p>
  <p><strong>Складність:</strong> ${escapeHtml(info.complexity || "—")}</p>
  <h2>Налаштування</h2>
  <ul>${settingsItems}</ul>
  <h2>Початкові дані</h2>
  <pre>${escapeHtml(getInitialDataSnapshot())}</pre>
  <h2>Візуальні знімки</h2>
  <div class="image-grid">${reportImagesHtml}</div>
  ${executionHtml}
  <script>
    window.addEventListener('load', function () {
      setTimeout(function () { window.print(); }, 250);
    });
  <\/script>
</body>
</html>`;
  const reportWindow = window.open("", "_blank");
  if (!reportWindow) {
    alert("Браузер заблокував відкриття звіту. Дозвольте спливаюче вікно для цієї сторінки.");
    return;
  }
  reportWindow.document.open();
  reportWindow.document.write(reportHtml);
  reportWindow.document.close();
}
function rerenderCurrentStepAfterResize() {
  if (!visualizer) return;
  if (currentAlgorithm && currentAlgorithm.steps && currentAlgorithm.steps.length > 0) {
    const stepIndex = Math.min(
      Math.max(0, currentStepIndex > 0 ? currentStepIndex - 1 : 0),
      currentAlgorithm.steps.length - 1,
    );
    renderStep(currentAlgorithm.steps[stepIndex] || currentAlgorithm.steps[0]);
  } else if (visualizer) {
    visualizer.clear();
  }
```

```
}
let resizeRerenderTimeout = null;
function scheduleCurrentStepRerender() {
  if (resizeRerenderTimeout) {
    clearTimeout(resizeRerenderTimeout);
  }
  resizeRerenderTimeout = setTimeout(() => {
    resizeRerenderTimeout = null;
    rerenderCurrentStepAfterResize();
  }, 120);
}
window.addEventListener("resize", scheduleCurrentStepRerender);
window.addEventListener("orientationchange", () => setTimeout(scheduleCurrentStepRerender,
250));
window.addEventListener("algoviz-canvas-resized", scheduleCurrentStepRerender);

window.addEventListener("algoviz-theme-changed", () => {
  if (!visualizer) return;
  if (currentAlgorithm && currentAlgorithm.steps && currentAlgorithm.steps.length > 0) {
    const stepIndex = Math.max(0, currentStepIndex - 1);
    renderStep(currentAlgorithm.steps[stepIndex] || currentAlgorithm.steps[0]);
  } else {
    visualizer.clear();
  }
});
```

B.4 compare.js

```
const COMPARE_GROUPS = {
  sorting: {
    title: "Сортування",
    algorithms: ["bubble", "insertion", "merge", "quick", "heap"],
  },
  searching: {
    title: "Пошук",
    algorithms: ["linear", "binary", "jump", "interpolation"],
  },
};

const COMPARE_ALGORITHM_CLASSES = {
  bubble: () => BubbleSort,
  insertion: () => InsertionSort,
  merge: () => MergeSort,
  quick: () => QuickSort,
  heap: () => HeapSort,
  linear: () => LinearSearch,
  binary: () => BinarySearch,
  jump: () => JumpSearch,
```

```
  interpolation: () => InterpolationSearch,
};

function getCompareAlgorithmClass(type) {
  return COMPARE_ALGORITHM_CLASSES[type] ?
COMPARE_ALGORITHM_CLASSES[type]() : null;
}

function getCompareAlgorithmInfo(type) {
  const cls = getCompareAlgorithmClass(type);
  return cls && cls.info ? cls.info : { name: type, complexity: "—" };
}

function getCompareAlgorithmName(type) {
  return getCompareAlgorithmInfo(type).name || type;
}

function getCompareAlgorithmGroup(type) {
  const entry = Object.entries(COMPARE_GROUPS).find(([, group]) =>
group.algorithms.includes(type));
  return entry ? entry[0] : null;
}

let compareVisualizers = [];
let compareState = null;
let compareAnimationTimeout = null;
let compareAnimationSpeed = 50;
let compareCustomArrayData = null;
let compareCustomArrayFileName = "";
let compareCustomSearchTarget = null;
function getCompareAnimationDelay(speedValue) {
  const value = Math.max(1, Math.min(100, parseInt(speedValue, 10) || 50));
  return Math.round(460 + (100 - value) * 12);
}

function getCompareAlgorithmDisplayTimeFromStats(stats) {
  if (!stats) return 0;
  const operations = Math.max(0, Number(stats.operations) || 0);
  const comparisons = Math.max(0, Number(stats.comparisons) || 0);
  const swaps = Math.max(0, Number(stats.swaps) || 0);
  const otherOperations = Math.max(0, operations - comparisons - swaps);
  return Math.round(comparisons + swaps + otherOperations);
}

function getCompareAlgorithmDisplayTimeFromStep(step) {
  return getCompareAlgorithmDisplayTimeFromStats(step && step.stats ? step.stats : null);
}

function getCompareAlgorithmDisplayTimeAtIndex(result, index) {
  if (!result || !result.steps || result.steps.length === 0 || index < 0) return 0;
  const safeIndex = Math.min(index, result.steps.length - 1);
  return getCompareAlgorithmDisplayTimeFromStep(result.steps[safeIndex]);
```

```
}

function getCompareSharedDisplayTimeAtIndex(index) {
  if (!compareState || index < 0) return 0;
  const firstTime = getCompareAlgorithmDisplayTimeAtIndex(compareState.first, index);
  const secondTime = getCompareAlgorithmDisplayTimeAtIndex(compareState.second, index);
  return Math.max(firstTime, secondTime, index + 1);
}

function getCompareElapsedTargetsAtIndex(index) {
  const sharedTime = getCompareSharedDisplayTimeAtIndex(index);
  const sharedElapsed = Math.max(compareLastDisplayedElapsed, sharedTime);
  return [
    getCompareSideElapsedTarget(0, compareState.first, index, sharedElapsed),
    getCompareSideElapsedTarget(1, compareState.second, index, sharedElapsed),
  ];
}

function updateCompareSharedElapsed(index, shouldAnimate = false, duration =
compareAnimationSpeed) {
  const sharedTime = getCompareSharedDisplayTimeAtIndex(index);
  compareLastDisplayedElapsed = Math.max(compareLastDisplayedElapsed, sharedTime);
  const nextTimes = getCompareElapsedTargetsAtIndex(index);

  if (shouldAnimate) {
    animateCompareElapsedTo(nextTimes, duration);
  } else {
    setCompareElapsedInstant(nextTimes);
  }
}

function getCompareSideElapsedTarget(sideIndex, result, index, sharedElapsed =
compareLastDisplayedElapsed) {
  if (!result || !Array.isArray(result.steps) || result.steps.length === 0) {
    return sharedElapsed;
  }

  const lastIndex = result.steps.length - 1;

  if (index >= lastIndex) {
    if (!compareFinishedTimes[sideIndex]) {
      compareFinishedTimes[sideIndex] = getCompareSharedDisplayTimeAtIndex(lastIndex);
    }
    return compareFinishedTimes[sideIndex];
  }

  return sharedElapsed;
}
let compareStepIndex = 0;
let compareIsAnimating = false;
let compareLayoutSeed = 1;
let compareRunStartedAt = 0;
```

```
let compareElapsedBeforePause = 0;
let compareTimerFrame = null;
let compareFinishedTimes = [0, 0];
let compareElapsedTimes = [0, 0];
let compareTargetElapsedTimes = [0, 0];
let compareHasStarted = false;
let compareIsFinished = false;
let compareLastDisplayedElapsed = 0;
let currentCompareStepDuration = 0;
let compareStepStartedAt = 0;

function getCompareNow() {
  if (typeof Statistics !== "undefined" && typeof Statistics.now === "function") {
    return Statistics.now();
  }

  if (typeof performance !== "undefined" && typeof performance.now === "function") {
    return performance.now();
  }

  return Date.now();
}

function getCurrentCompareStepProgress() {
  if (!compareAnimationTimeout || currentCompareStepDuration <= 0 || compareStepStartedAt <=
0) {
    return 0;
  }

  const elapsed = Math.max(0, getCompareNow() - compareStepStartedAt);
  return Math.min(1, elapsed / currentCompareStepDuration);
}

function getRemainingCompareDuration() {
  if (!compareState) return compareAnimationSpeed;

  const maxSteps = Math.max(compareState.first.steps.length, compareState.second.steps.length);
  const remainingSteps = Math.max(0, maxSteps - compareStepIndex);
  if (remainingSteps === 0) return compareAnimationSpeed;

  const currentStepRemaining = compareAnimationTimeout
    ? Math.max(0, compareAnimationSpeed * (1 - getCurrentCompareStepProgress()))
    : compareAnimationSpeed;

  return Math.max(compareAnimationSpeed, currentStepRemaining + Math.max(0, remainingSteps
- 1) * compareAnimationSpeed);
}

function scheduleNextCompareStep(delay = compareAnimationSpeed) {
  if (compareAnimationTimeout) {
    clearTimeout(compareAnimationTimeout);
  }
```

```
  currentCompareStepDuration = Math.max(1, delay);
  compareStepStartedAt = getCompareNow();
  compareAnimationTimeout = setTimeout(animateComparison, currentCompareStepDuration);
}

function rescheduleComparisonAfterSpeedChange() {
  if (!compareIsAnimating || !compareAnimationTimeout) return;

  const progress = getCurrentCompareStepProgress();
  const remainingDelay = Math.max(1, Math.round(compareAnimationSpeed * (1 - progress)));

  clearTimeout(compareAnimationTimeout);
  currentCompareStepDuration = compareAnimationSpeed;
  compareStepStartedAt = getCompareNow() - progress * currentCompareStepDuration;
  compareAnimationTimeout = setTimeout(animateComparison, remainingDelay);
}

function getCompareFinalElapsedTargets() {
  if (!compareState) return [0, 0];

  const firstSteps = Math.max(1, compareState.first.steps.length);
  const secondSteps = Math.max(1, compareState.second.steps.length);
  const maxSteps = Math.max(firstSteps, secondSteps);
  const finalSharedTime = getCompareSharedDisplayTimeAtIndex(maxSteps - 1);

  return [
    Math.round(finalSharedTime * (firstSteps / maxSteps)),
    Math.round(finalSharedTime * (secondSteps / maxSteps)),
  ];
}

function setCompareElapsedInstant(times) {
  stopCompareTimer();
  compareTargetElapsedTimes = [
    Math.max(0, Number(times && times[0]) || 0),
    Math.max(0, Number(times && times[1]) || 0),
  ];
  compareElapsedTimes = [...compareTargetElapsedTimes];
  updateCompareTimeLabels();
}

function completeCompareTimeAnimation() {
  if (!compareTimerFrame) return;
  stopCompareTimer();
  compareElapsedTimes = [...compareTargetElapsedTimes];
  updateCompareTimeLabels();
}

function animateCompareElapsedTo(times, duration = compareAnimationSpeed) {
  stopCompareTimer();
```

```
  const targets = [
    Math.max(0, Number(times && times[0]) || 0),
    Math.max(0, Number(times && times[1]) || 0),
  ];
  const starts = [
    Math.max(0, Number(compareElapsedTimes[0]) || 0),
    Math.max(0, Number(compareElapsedTimes[1]) || 0),
  ];
  const sharedStart = Math.max(starts[0], starts[1]);
  const sharedTarget = Math.max(targets[0], targets[1]);
  const animationDuration = Math.max(0, Number(duration) || 0);

  compareTargetElapsedTimes = targets;

  if (
    !compareIsAnimating ||
    animationDuration <= 40 ||
    sharedStart >= sharedTarget
  ) {
    compareElapsedTimes = [...targets];
    updateCompareTimeLabels();
    return;
  }

  let startedAt = null;

  const tick = (timestamp) => {
    if (startedAt === null) startedAt = timestamp;

    const progress = Math.min(1, (timestamp - startedAt) / animationDuration);
    const sharedElapsed = sharedStart + (sharedTarget - sharedStart) * progress;
    compareElapsedTimes = [
      Math.min(sharedElapsed, targets[0]),
      Math.min(sharedElapsed, targets[1]),
    ];
    updateCompareTimeLabels();

    if (progress < 1 && compareIsAnimating) {
      compareTimerFrame = requestAnimationFrame(tick);
      return;
    }

    compareElapsedTimes = [...targets];
    compareTimerFrame = null;
    updateCompareTimeLabels();
  };

  compareTimerFrame = requestAnimationFrame(tick);
}

function syncCompareSpeedControls(value) {
  const speed = Math.max(1, Math.min(100, parseInt(value, 10) || 50));
```

```
  const speedInput = document.getElementById("compareSpeedInput");
  const speedNumber = document.getElementById("compareSpeedNumber");
  const display = document.getElementById("compareSpeedDisplay");

  if (speedInput) speedInput.value = String(speed);
  if (speedNumber) speedNumber.value = String(speed);
  if (display) display.textContent = speed + "%";

  compareAnimationSpeed = getCompareAnimationDelay(speed);
  if (compareIsAnimating) {
    rescheduleComparisonAfterSpeedChange();
    startCompareTimer();
  } else {
    updateCompareTimeLabels();
  }

  if (compareState && (compareHasStarted || compareIsFinished)) {
    updateCompareResults(compareIsFinished);
  }
}

window.addEventListener("DOMContentLoaded", () => {
  compareVisualizers = [new Visualizer("compareCanvas1"), new Visualizer("compareCanvas2")];
  initCompareControls();
  prepareComparison();
});

function bindCompareButtons() {
  const buttonHandlers = {
    prepareCompareBtn: prepareComparisonWithRandomData,
    startCompareBtn: startComparison,
    pauseCompareBtn: pauseComparison,
    resetCompareBtn: resetComparison,
    exportCompareReportBtn: exportCompareToPdf,
  };

  Object.entries(buttonHandlers).forEach(([id, handler]) => {
    const button = document.getElementById(id);
    if (button && button.dataset.compareBound !== "true") {
      button.dataset.compareBound = "true";
      button.addEventListener("click", handler);
    }
  });
}

function initCompareControls() {
  const algo1 = document.getElementById("compareAlgo1");
  const algo2 = document.getElementById("compareAlgo2");
  const speed = document.getElementById("compareSpeedInput");
  const speedNumber = document.getElementById("compareSpeedNumber");

  bindCompareButtons();
```

```
  initCompareFileImport();

  fillFirstAlgorithmOptions();

  algo1.value = "bubble";
  fillSecondAlgorithmOptions();
  algo2.value = "quick";

  algo1.addEventListener("change", () => {
    fillSecondAlgorithmOptions();
    renderCompareSettings();
    prepareComparison();
  });

  algo2.addEventListener("change", () => {
    renderCompareSettings();
    prepareComparison();
  });

  if (speed) {
    speed.addEventListener("input", (event) => syncCompareSpeedControls(event.target.value));
  }

  if (speedNumber) {
    speedNumber.addEventListener("change", (event) =>
syncCompareSpeedControls(event.target.value));
    speedNumber.addEventListener("keydown", (event) => {
      if (event.key === "Enter") event.currentTarget.blur();
    });
  }

  syncCompareSpeedControls(speed ? speed.value : 50);
  renderCompareSettings();
}

function fillFirstAlgorithmOptions() {
  const algo1 = document.getElementById("compareAlgo1");
  algo1.innerHTML = "";

  Object.entries(COMPARE_GROUPS).forEach(([groupKey, group]) => {
    const optgroup = document.createElement("optgroup");
    optgroup.label = group.title;

    group.algorithms.forEach((type) => {
      const option = document.createElement("option");
      option.value = type;
      option.textContent = getCompareAlgorithmName(type);
      optgroup.appendChild(option);
    });

    algo1.appendChild(optgroup);
  });
```

```
}

function fillSecondAlgorithmOptions() {
  const algo1 = document.getElementById("compareAlgo1");
  const algo2 = document.getElementById("compareAlgo2");
  const firstType = algo1.value || "bubble";
  const group = getCompareAlgorithmGroup(firstType);
  const allowed = COMPARE_GROUPS[group].algorithms;
  const previous = algo2.value;

  algo2.innerHTML = "";
  const optgroup = document.createElement("optgroup");
  optgroup.label = COMPARE_GROUPS[group].title;
  allowed.forEach((type) => {
    if (type === firstType && allowed.length > 1) return;
    const option = document.createElement("option");
    option.value = type;
    option.textContent = getCompareAlgorithmName(type);
    optgroup.appendChild(option);
  });
  algo2.appendChild(optgroup);

  if (allowed.includes(previous) && previous !== firstType) {
    algo2.value = previous;
  } else {
    algo2.value = algo2.options[0]?.value || firstType;
  }
}

function renderCompareSettings() {
  const settings = document.getElementById("compareSettings");
  const type1 = document.getElementById("compareAlgo1").value;
  const type2 = document.getElementById("compareAlgo2").value;
  const group = getCompareAlgorithmGroup(type1);
  const title = COMPARE_GROUPS[group].title;

  let html = `<div class="settings-title">Категорія: ${title}</div>`;

  if (group === "sorting") {
    html += `
      <div class="method-settings-grid">
        <div>
          <label for="compareArraySize">Розмір масиву:</label>
          <input type="number" id="compareArraySize" min="5" max="120" value="40">
        </div>
      </div>
    `;
  }

  if (group === "searching") {
    html += `
      <div class="method-settings-grid">
```

```
      <div>
        <label for="compareArraySize">Розмір масиву:</label>
        <input type="number" id="compareArraySize" min="5" max="120" value="40">
      </div>
      <div>
        <label for="compareSearchTarget">Шукане значення:</label>
        <input type="number" id="compareSearchTarget" min="1" max="100" value="50">
      </div>
    </div>
  `;
 }


 settings.innerHTML = html;
 const targetInput = document.getElementById("compareSearchTarget");
 if (targetInput && compareCustomSearchTarget !== null) {
  targetInput.value = String(compareCustomSearchTarget);
 }
 settings.querySelectorAll("input").forEach((input) => {
  input.addEventListener("change", () => {
   if (input.id === "compareArraySize") {
    compareCustomArrayData = null;
    compareCustomArrayFileName = "";
    compareCustomSearchTarget = null;
    updateCompareImportStatus();
   }
   prepareComparison();
  });
 });
 updateCompareImportStatus();
}

function getCompareInputNumber(id, fallback, min, max) {
 const input = document.getElementById(id);
 const raw = input ? parseInt(input.value, 10) : fallback;
 const value = Number.isFinite(raw) ? raw : fallback;
 return Math.min(max, Math.max(min, value));
}

function generateCompareArray(size) {
 return Array.from({ length: size }, () => Math.floor(Math.random() * 100) + 1);
}

function getCompareArray(size) {
 if (Array.isArray(compareCustomArrayData) && compareCustomArrayData.length > 0) {
  return compareCustomArrayData.slice();
 }
 return generateCompareArray(size);
}

function updateCompareImportStatus() {
 const status = document.getElementById("compareDataImportStatus");
```

```
  if (!status) return;
  if (Array.isArray(compareCustomArrayData) && compareCustomArrayData.length > 0) {
    status.textContent = `Завантажено: ${compareCustomArrayFileName}
(${compareCustomArrayData.length} чисел).`;
  } else {
    status.textContent = "Можна завантажити TXT-файл зі спільними числами.";
  }
}

function initCompareFileImport() {
  const input = document.getElementById("compareDataFileInput");
  const button = document.getElementById("compareImportDataButton");
  if (!input || !button) return;
  button.addEventListener("click", () => input.click());
  input.addEventListener("change", handleCompareDataFileUpload);
}

async function handleCompareDataFileUpload(event) {
  const input = event.currentTarget;
  const file = input.files && input.files[0];
  if (!file) return;

  try {
    const text = await window.AlgoVizDataFile.readTextFile(file);
    const parsed = window.AlgoVizDataFile.parseTextData(text, { minLength: 2, maxLength: 120
});
    compareCustomArrayData = parsed.array;
    compareCustomArrayFileName = file.name || "дані.txt";
    compareCustomSearchTarget = parsed.target;
    const sizeInput = document.getElementById("compareArraySize");
    const targetInput = document.getElementById("compareSearchTarget");
    if (sizeInput) sizeInput.value = String(compareCustomArrayData.length);
    if (targetInput && compareCustomSearchTarget !== null) targetInput.value =
String(compareCustomSearchTarget);
    updateCompareImportStatus();
    prepareComparison();
  } catch (error) {
    alert(error.message || "Не вдалося завантажити дані з файлу.");
  } finally {
    input.value = "";
  }
}

function prepareComparisonWithRandomData() {
  compareCustomArrayData = null;
  compareCustomArrayFileName = "";
  compareCustomSearchTarget = null;
  updateCompareImportStatus();
  prepareComparison();
}

function prepareComparison() {
```

```
  pauseComparison();
  stopCompareTimer();
  compareStepIndex = 0;
  compareRunStartedAt = 0;
  compareElapsedBeforePause = 0;
  compareFinishedTimes = [0, 0];
  compareElapsedTimes = [0, 0];
  compareTargetElapsedTimes = [0, 0];
  compareLastDisplayedElapsed = 0;
  currentCompareStepDuration = 0;
  compareHasStarted = false;
  compareIsFinished = false;

  const type1 = document.getElementById("compareAlgo1").value;
  const type2 = document.getElementById("compareAlgo2").value;
  const group = getCompareAlgorithmGroup(type1);
  let commonData = null;

  if (group === "sorting" || group === "searching") {
    const size = getCompareInputNumber("compareArraySize", 40, 5, 120);
    commonData = { array: getCompareArray(size) };
  }

  const first = buildCompareAlgorithm(type1, commonData);
  const second = buildCompareAlgorithm(type2, commonData);

  compareState = { type1, type2, group, commonData, first, second };
  document.getElementById("compareTitle1").textContent = getCompareAlgorithmName(type1);
  document.getElementById("compareTitle2").textContent = getCompareAlgorithmName(type2);

  renderCompareStep(first, compareVisualizers[0], 0, type1, 1);
  renderCompareStep(second, compareVisualizers[1], 0, type2, 2);
  updateCompareTimeLabels();
  showComparePlaceholder();
}

function showComparePlaceholder() {
  const resultBox = document.getElementById("compareResult");
  if (!resultBox) return;
  resultBox.innerHTML = `
    <div class="empty-compare">
      <i class="fas fa-chart-column"></i>
      <span>Натисніть «Запустити порівняння», щоб побачити результати.</span>
    </div>
  `;
}

function buildCompareAlgorithm(type, commonData) {
  const cls = getCompareAlgorithmClass(type);
  const group = getCompareAlgorithmGroup(type);
  if (!cls || !group) {
    throw new Error(`Невідомий алгоритм для порівняння: ${type}`);
```

```
  }

  const startTime = Statistics.now();
  let algorithm;

  if (group === "sorting") {
    algorithm = new cls([...commonData.array]);
    algorithm.execute();
  }

  if (group === "searching") {
    const target = getCompareInputNumber("compareSearchTarget", 50, 1, 100);
    const data = ["binary", "jump", "interpolation"].includes(type) ? [...commonData.array].sort((a, b)
=> a - b) : [...commonData.array];
    algorithm = new cls(data);
    algorithm.execute(target);
  }


  const time = Statistics.now() - startTime;
  const stats = algorithm.statistics.getStats();

  return {
    algorithm,
    stats,
    time,
    steps: algorithm.steps || [],
  };
}

function startComparison() {
  if (!compareState) prepareComparison();
  if (!compareState) return;
  if (compareIsAnimating) return;

  const maxSteps = Math.max(compareState.first.steps.length, compareState.second.steps.length);

  if (!compareHasStarted || compareIsFinished || compareStepIndex >= maxSteps) {
    compareStepIndex = 0;
    compareRunStartedAt = 0;
    compareElapsedBeforePause = 0;
    compareFinishedTimes = [0, 0];
    compareElapsedTimes = [0, 0];
    compareTargetElapsedTimes = [0, 0];
    compareLastDisplayedElapsed = 0;
    currentCompareStepDuration = 0;
    compareStepStartedAt = 0;
    compareIsFinished = false;
  }

  compareHasStarted = true;
  compareIsAnimating = true;
```

```
  startCompareTimer();
  updateCompareResults(false);
  animateComparison();
}

function pauseComparison() {
  compareIsAnimating = false;
  stopCompareTimer();
  if (compareAnimationTimeout) {
    clearTimeout(compareAnimationTimeout);
    compareAnimationTimeout = null;
  }
  compareStepStartedAt = 0;
}

function resetComparison() {
  pauseComparison();
  stopCompareTimer();
  compareStepIndex = 0;
  compareRunStartedAt = 0;
  compareElapsedBeforePause = 0;
  compareFinishedTimes = [0, 0];
  compareElapsedTimes = [0, 0];
  compareTargetElapsedTimes = [0, 0];
  compareLastDisplayedElapsed = 0;
  currentCompareStepDuration = 0;
  compareStepStartedAt = 0;
  compareHasStarted = false;
  compareIsFinished = false;
  if (!compareState) return;
  renderCompareStep(compareState.first, compareVisualizers[0], 0, compareState.type1, 1);
  renderCompareStep(compareState.second, compareVisualizers[1], 0, compareState.type2, 2);
  updateCompareTimeLabels();
  showComparePlaceholder();
}

function animateComparison() {
  if (!compareIsAnimating || !compareState) return;

  const maxSteps = Math.max(compareState.first.steps.length, compareState.second.steps.length);
  if (compareStepIndex >= maxSteps) {
    completeCompareTimeAnimation();
    setCompareElapsedInstant(getCompareFinalElapsedTargets());
    compareIsAnimating = false;
    compareStepStartedAt = 0;
    compareIsFinished = true;
    updateCompareTimeLabels();
    updateCompareResults(true);
    return;
  }

  currentCompareStepDuration = compareAnimationSpeed;
```

```
  renderCompareStep(compareState.first, compareVisualizers[0], compareStepIndex,
compareState.type1, 1);
  renderCompareStep(compareState.second, compareVisualizers[1], compareStepIndex,
compareState.type2, 2);

  compareStepIndex++;
  updateCompareResults(false);
  scheduleNextCompareStep(compareAnimationSpeed);
}

function renderCompareStep(result, visualizer, index, type, side) {
  if (!result || !visualizer) return;
  const steps = result.steps;
  const safeIndex = Math.min(Math.max(0, index), Math.max(0, steps.length - 1));
  const step = steps[safeIndex];
  if (!step) {
    visualizer.clear();
    setCompareText(`compareStep${side}`, 0);
    updateCompareTimeLabels();
    return;
  }

  if (step.array) {
    visualizer.drawBars(step.array, {
      sorted: step.sorted || [],
      comparing: step.comparing || [],
      current: step.current,
      range: step.range || [],
      target: step.target,
      message: step.message || "",
      foundIndex: step.foundIndex,
      notFound: step.notFound || false,
      animationDuration: currentCompareStepDuration || compareAnimationSpeed,
    });
  }

  const stepNumber = compareHasStarted ? Math.min(safeIndex + 1, steps.length) : 0;
  setCompareText(`compareStep${side}`, stepNumber);
}

function formatCompareDuration(ms) {
  const value = Math.max(0, Math.round(Number(ms) || 0));
  if (value < 1000) return `${value}ms`;
  return `${(value / 1000).toFixed(value < 10000 ? 2 : 1)}s`;
}

function getCompareElapsed() {
  if (!compareState || !compareHasStarted) return 0;
  return Math.max(compareElapsedTimes[0] || 0, compareElapsedTimes[1] || 0);
}
```

```
function getCompareDisplayTimeForSide(sideIndex) {
  if (!compareState || !compareHasStarted) return 0;
  return compareElapsedTimes[sideIndex] || 0;
}

function updateCompareTimeLabels() {
  const firstTime = formatCompareDuration(getCompareDisplayTimeForSide(0));
  const secondTime = formatCompareDuration(getCompareDisplayTimeForSide(1));
  setCompareText("compareTime1", firstTime);
  setCompareText("compareTime2", secondTime);
  setCompareText("compareResultTime1", firstTime);
  setCompareText("compareResultTime2", secondTime);
}

function startCompareTimer() {
  if (!compareState) {
    updateCompareTimeLabels();
    return;
  }

  const maxSteps = Math.max(compareState.first.steps.length, compareState.second.steps.length);
  const duration = getRemainingCompareDuration();
  animateCompareElapsedTo(getCompareFinalElapsedTargets(), duration);
}

function pauseCompareTimer() {
  stopCompareTimer();
  updateCompareTimeLabels();
}

function stopCompareTimer() {
  if (compareTimerFrame) {
    cancelAnimationFrame(compareTimerFrame);
    compareTimerFrame = null;
  }
}

function finalizeCompareTimes(maxSteps) {
  setCompareElapsedInstant(getCompareFinalElapsedTargets());
  compareFinishedTimes = [compareElapsedTimes[0] || 0, compareElapsedTimes[1] || 0];
}

function setCompareText(id, value) {
  const element = document.getElementById(id);
  if (element) element.textContent = value;
}

function updateCompareResults(final = false) {
  if (!compareState) return;
  const resultBox = document.getElementById("compareResult");
  const a = summarizeCompare(compareState.type1, compareState.first, 0);
  const b = summarizeCompare(compareState.type2, compareState.second, 1);
```

```
  const winner = final
    ? (a.operations < b.operations ? a.name : b.operations < a.operations ? b.name : "однаково")
    : "—";
  const maxOps = Math.max(a.operations, b.operations, 1);
  const maxComparisons = Math.max(a.comparisons, b.comparisons, 1);
  const maxSteps = Math.max(a.visibleSteps, b.visibleSteps, 1);

  resultBox.innerHTML = `
    <div class="compare-summary">

<div><strong>Категорія:</strong><span>${COMPARE_GROUPS[compareState.group].title}</s
pan></div>
      <div><strong>Менше операцій:</strong><span>${winner}</span></div>
      <div><strong>Статус:</strong><span>${final ? "порівняння завершено" : "порівняння
виконується"}</span></div>
    </div>

    <div class="compare-metric-bars">
      ${renderMetricBar("Операції", a, b, "operations", maxOps)}
      ${renderMetricBar("Порівняння", a, b, "comparisons", maxComparisons)}
      ${renderMetricBar("Кроки", a, b, "visibleSteps", maxSteps)}
    </div>

    <table class="compare-table">
      <thead><tr><th>Показник</th><th>${a.name}</th><th>${b.name}</th></tr></thead>
      <tbody>
        <tr><td>Операції</td><td>${a.operations}</td><td>${b.operations}</td></tr>
        <tr><td>Порівняння</td><td>${a.comparisons}</td><td>${b.comparisons}</td></tr>
        <tr><td>Обміни / переміщення</td><td>${a.swaps}</td><td>${b.swaps}</td></tr>
        <tr><td>Поточний крок</td><td>${a.visibleSteps} / ${a.steps}</td><td>${b.visibleSteps} /
${b.steps}</td></tr>
        <tr><td>Час порівняння</td><td
id="compareResultTime1">${formatCompareDuration(a.displayTime)}</td><td
id="compareResultTime2">${formatCompareDuration(b.displayTime)}</td></tr>
        <tr><td>Складність</td><td>${a.complexity}</td><td>${b.complexity}</td></tr>
      </tbody>
    </table>
  `;
}

function renderMetricBar(label, a, b, key, max) {
  const widthA = Math.max(4, (a[key] / max) * 100);
  const widthB = Math.max(4, (b[key] / max) * 100);
  return `
    <div class="metric-row">
      <div class="metric-label">${label}</div>
      <div class="metric-line"><span>${a.name}</span><div class="metric-track"><div
class="metric-fill" style="width:${widthA}%"></div></div><strong>${a[key]}</strong></div>
      <div class="metric-line"><span>${b.name}</span><div class="metric-track"><div
class="metric-fill alt" style="width:${widthB}%"></div></div><strong>${b[key]}</strong></div>
    </div>
  `;
```

```
}

function summarizeCompare(type, result, sideIndex) {
  const cls = getCompareAlgorithmClass(type);
  const classInfo = getCompareAlgorithmInfo(type);
  const visibleSteps = Math.min(compareStepIndex, result.steps.length);
  const step = result.steps[Math.max(0, visibleSteps - 1)];
  const stats = step?.stats || { operations: 0, comparisons: 0, swaps: 0 };
  const displayTime = getCompareDisplayTimeForSide(sideIndex);

  return {
    name: classInfo.name || type,
    complexity: classInfo.complexity || "—",
    operations: stats.operations || 0,
    comparisons: stats.comparisons || 0,
    swaps: stats.swaps || 0,
    visibleSteps,
    steps: result.steps.length,
    displayTime,
  };
}


function escapeCompareHtml(value) {
  return String(value ?? "")
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/"/g, "&quot;")
    .replace(/'/g, "&#039;");
}

function formatCompareReportValue(value, limit = 160) {
  if (Array.isArray(value)) {
    const items = value.slice(0, limit).map((item) => Array.isArray(item) ? `[${item.join(", ")}]` :
String(item));
    const suffix = value.length > limit ? ` … (+${value.length - limit})` : "";
    return items.join(", ") + suffix;
  }

  if (value && typeof value === "object") {
    return JSON.stringify(value, null, 2);
  }

  return String(value ?? "—");
}

function getCompareSettingsSummary() {
  if (!compareState) return [];
  const speedInput = document.getElementById("compareSpeedInput");
  const sizeInput = document.getElementById("compareArraySize");
  const targetInput = document.getElementById("compareSearchTarget");
```

```
  const items = [
    `Перший метод: ${getCompareAlgorithmName(compareState.type1)}`,
    `Другий метод: ${getCompareAlgorithmName(compareState.type2)}`,
    `Категорія: ${COMPARE_GROUPS[compareState.group].title}`,
    `Швидкість анімації: ${speedInput ? speedInput.value : "—"}%`,
  ];

  if (sizeInput) items.push(`Розмір масиву: ${sizeInput.value}`);
  if (targetInput) items.push(`Шукане значення: ${targetInput.value}`);

  return items;
}

function getCompareInitialDataSnapshot() {
  if (!compareState || !compareState.commonData) return "—";
  if (Array.isArray(compareState.commonData.array)) {
    return `Масив: ${formatCompareReportValue(compareState.commonData.array)}`;
  }
  return formatCompareReportValue(compareState.commonData);
}

function getCompareStepSnapshot(result, index) {
  if (!result || !Array.isArray(result.steps) || result.steps.length === 0) return "—";
  const safeIndex = Math.min(Math.max(0, index), result.steps.length - 1);
  const step = result.steps[safeIndex];
  if (!step) return "—";
  if (Array.isArray(step.array)) return `Масив: ${formatCompareReportValue(step.array)}`;
  return formatCompareReportValue(step);
}

function drawCompareSnapshot(step, visualizer) {
  if (!step || !visualizer) return "";

  try {
    if (typeof visualizer.stopAnimation === "function") {
      visualizer.stopAnimation();
    }
    if ("_forceInstantBars" in visualizer) {
      visualizer._forceInstantBars = true;
    }
    if (step.array) {
      visualizer.drawBars(step.array, {
        sorted: step.sorted || [],
        comparing: step.comparing || [],
        current: step.current,
        range: step.range || [],
        target: step.target,
        message: step.message || "",
        foundIndex: step.foundIndex,
        notFound: step.notFound || false,
      });
    }
```

```
    return visualizer.canvas.toDataURL("image/png");
  } catch (error) {
    console.error(error);
    return "";
  }
}

function getCompareInitialStepForReport(result) {
  if (result && result.algorithm && Array.isArray(result.algorithm.data)) {
    return { array: [...result.algorithm.data] };
  }

  if (result && Array.isArray(result.steps) && result.steps.length > 0) {
    return result.steps[0];
  }

  return null;
}

function drawCompareStepOnly(result, visualizer, index) {
  if (!result || !Array.isArray(result.steps) || result.steps.length === 0) return "";
  const safeIndex = Math.min(Math.max(0, index), result.steps.length - 1);
  return drawCompareSnapshot(result.steps[safeIndex], visualizer);
}

function restoreCompareCanvasesAfterCapture() {
  if (!compareState) return;
  const visibleIndex = compareHasStarted ? Math.max(0, compareStepIndex - 1) : 0;
  renderCompareStep(compareState.first, compareVisualizers[0], visibleIndex, compareState.type1,
1);
  renderCompareStep(compareState.second, compareVisualizers[1], visibleIndex,
compareState.type2, 2);
  updateCompareTimeLabels();
}

function getCompareReportImageMarkup(title, firstSrc, secondSrc) {
  const image = (src, alt) => src
    ? `<img src="${src}" alt="${escapeCompareHtml(alt)}">`
    : `<p>Знімок недоступний.</p>`;

  return `
    <div class="image-card">
      <h3>${escapeCompareHtml(title)}</h3>
      <div class="two-images">

<div><h4>${escapeCompareHtml(getCompareAlgorithmName(compareState.type1))}</h4>${ima
ge(firstSrc, title + " 1")}</div>

<div><h4>${escapeCompareHtml(getCompareAlgorithmName(compareState.type2))}</h4>${ima
ge(secondSrc, title + " 2")}</div>
      </div>
    </div>`;
```

```
}

function buildCompareReportTable(final) {
  const a = summarizeCompare(compareState.type1, compareState.first, 0);
  const b = summarizeCompare(compareState.type2, compareState.second, 1);
  const winner = final
    ? (a.operations < b.operations ? a.name : b.operations < a.operations ? b.name : "однаково")
    : "—";

  return `
    <p><strong>Менше операцій:</strong> ${escapeCompareHtml(winner)}</p>
    <table>

<thead><tr><th>Показник</th><th>${escapeCompareHtml(a.name)}</th><th>${escapeCompare
Html(b.name)}</th></tr></thead>
      <tbody>

<tr><td>Операції</td><td>${escapeCompareHtml(a.operations)}</td><td>${escapeCompareHtm
l(b.operations)}</td></tr>

<tr><td>Порівняння</td><td>${escapeCompareHtml(a.comparisons)}</td><td>${escapeCompar
eHtml(b.comparisons)}</td></tr>
        <tr><td>Обміни /
переміщення</td><td>${escapeCompareHtml(a.swaps)}</td><td>${escapeCompareHtml(b.swap
s)}</td></tr>
        <tr><td>Кроки</td><td>${escapeCompareHtml(a.visibleSteps)} /
${escapeCompareHtml(a.steps)}</td><td>${escapeCompareHtml(b.visibleSteps)} /
${escapeCompareHtml(b.steps)}</td></tr>
        <tr><td>Час
порівняння</td><td>${escapeCompareHtml(formatCompareDuration(a.displayTime))}</td><td>
${escapeCompareHtml(formatCompareDuration(b.displayTime))}</td></tr>

<tr><td>Складність</td><td>${escapeCompareHtml(a.complexity)}</td><td>${escapeCompare
Html(b.complexity)}</td></tr>
      </tbody>
    </table>`;
}

function exportCompareToPdf() {
  if (!compareState) {
    alert("Спочатку підготуйте дані для порівняння.");
    return;
  }

  pauseComparison();

  const maxSteps = Math.max(compareState.first.steps.length, compareState.second.steps.length);
  const hasStarted = compareHasStarted || compareStepIndex > 0;
  const hasCompleted = compareIsFinished || (hasStarted && compareStepIndex >= maxSteps);
  const currentDate = new Date().toLocaleString("uk-UA");
  const settingsItems = getCompareSettingsSummary().map((item) =>
`<li>${escapeCompareHtml(item)}</li>`).join("");
```

```
  const firstInitialImage =
drawCompareSnapshot(getCompareInitialStepForReport(compareState.first),
compareVisualizers[0]);
  const secondInitialImage =
drawCompareSnapshot(getCompareInitialStepForReport(compareState.second),
compareVisualizers[1]);
  const firstStateIndex = hasCompleted
    ? compareState.first.steps.length - 1
    : hasStarted
      ? Math.min(Math.max(0, compareStepIndex - 1), compareState.first.steps.length - 1)
      : null;
  const secondStateIndex = hasCompleted
    ? compareState.second.steps.length - 1
    : hasStarted
      ? Math.min(Math.max(0, compareStepIndex - 1), compareState.second.steps.length - 1)
      : null;
  const firstStateImage = firstStateIndex === null ? "" : drawCompareStepOnly(compareState.first,
compareVisualizers[0], firstStateIndex);
  const secondStateImage = secondStateIndex === null ? "" :
drawCompareStepOnly(compareState.second, compareVisualizers[1], secondStateIndex);

  restoreCompareCanvasesAfterCapture();

  const imageBlocks = [getCompareReportImageMarkup("Початковий стан", firstInitialImage,
secondInitialImage)];
  if (hasCompleted) {
    imageBlocks.push(getCompareReportImageMarkup("Стан після виконання", firstStateImage,
secondStateImage));
  } else if (hasStarted) {
    imageBlocks.push(getCompareReportImageMarkup("Поточний стан", firstStateImage,
secondStateImage));
  }

  const resultHtml = hasCompleted
    ? `
      <h2>Результат після виконання</h2>
      <div class="snapshots">

<div><h3>${escapeCompareHtml(getCompareAlgorithmName(compareState.type1))}</h3><pre>
${escapeCompareHtml(getCompareStepSnapshot(compareState.first,
firstStateIndex))}</pre></div>

<div><h3>${escapeCompareHtml(getCompareAlgorithmName(compareState.type2))}</h3><pre>
${escapeCompareHtml(getCompareStepSnapshot(compareState.second,
secondStateIndex))}</pre></div>
      </div>
      <h2>Підсумкова статистика</h2>
      ${buildCompareReportTable(true)}`
    : hasStarted
      ? `
      <h2>Поточний стан</h2>
      <div class="snapshots">
```

```
<div><h3>${escapeCompareHtml(getCompareAlgorithmName(compareState.type1))}</h3><pre>
${escapeCompareHtml(getCompareStepSnapshot(compareState.first,
firstStateIndex))}</pre></div>

<div><h3>${escapeCompareHtml(getCompareAlgorithmName(compareState.type2))}</h3><pre>
${escapeCompareHtml(getCompareStepSnapshot(compareState.second,
secondStateIndex))}</pre></div>
    </div>
    <p>Порівняння ще не завершено, тому фінальні результати у звіт не додано.</p>
    <h2>Поточна статистика</h2>
    ${buildCompareReportTable(false)}`
    : `
    <h2>Результат</h2>
    <p>Порівняння ще не запускалося, тому у звіт додано лише початкові дані.</p>`;

  const reportHtml = `<!doctype html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <title>Звіт порівняння AlgoViz Pro</title>
  <style>
    body { font-family: Arial, sans-serif; color: #111827; line-height: 1.5; padding: 28px; }
    h1 { margin: 0 0 6px; color: #1d4ed8; }
    h2 { margin-top: 24px; color: #111827; border-bottom: 1px solid #d1d5db; padding-bottom: 6px;
}
    h3 { margin: 0 0 10px; }
    h4 { margin: 0 0 8px; color: #374151; }
    .meta { color: #6b7280; margin-bottom: 18px; }
    .image-grid { display: grid; grid-template-columns: 1fr; gap: 14px; margin: 14px 0; }
    .image-card { border: 1px solid #d1d5db; border-radius: 10px; padding: 12px; background:
#f9fafb; }
    .two-images, .snapshots { display: grid; grid-template-columns: repeat(2, minmax(0, 1fr)); gap:
14px; }
    img { width: 100%; height: auto; display: block; border: 1px solid #e5e7eb; border-radius: 8px;
background: #0f172a; }
    pre { white-space: pre-wrap; word-break: break-word; background: #f3f4f6; border: 1px solid
#e5e7eb; border-radius: 10px; padding: 12px; }
    table { width: 100%; border-collapse: collapse; margin-top: 12px; }
    th, td { border: 1px solid #d1d5db; padding: 8px; text-align: left; }
    th { background: #eff6ff; }
    ul { margin-top: 8px; }
    @media print { body { padding: 0; } .no-print { display: none; } .image-card { break-inside:
avoid; } }
    @media (max-width: 800px) { .two-images, .snapshots { grid-template-columns: 1fr; } }
  </style>
</head>
<body>
  <button class="no-print" onclick="window.print()" style="float:right;padding:10px
14px;border:0;border-radius:8px;background:#2563eb;color:#fff;font-
weight:700;cursor:pointer;">Зберегти / друк PDF</button>
  <h1>Звіт порівняння AlgoViz Pro</h1>
```

```
  <div class="meta">Дата створення: ${escapeCompareHtml(currentDate)}</div>

  <h2>Налаштування</h2>
  <ul>${settingsItems}</ul>

  <h2>Початкові дані</h2>
  <pre>${escapeCompareHtml(getCompareInitialDataSnapshot())}</pre>

  <h2>Візуальні знімки</h2>
  <div class="image-grid">${imageBlocks.join("")}</div>
  ${resultHtml}

  <script>
    window.addEventListener('load', function () {
      setTimeout(function () { window.print(); }, 250);
    });
  <\/script>
</body>
</html>`;

  const reportWindow = window.open("", "_blank");

  if (!reportWindow) {
    alert("Браузер заблокував відкриття звіту. Дозвольте спливаюче вікно для цієї сторінки.");
    return;
  }

  reportWindow.document.open();
  reportWindow.document.write(reportHtml);
  reportWindow.document.close();
}

let compareResizeRerenderTimeout = null;
function scheduleCompareRerender() {
  if (compareResizeRerenderTimeout) {
    clearTimeout(compareResizeRerenderTimeout);
  }

  compareResizeRerenderTimeout = setTimeout(() => {
    compareResizeRerenderTimeout = null;
    if (!compareState) return;
    const visibleIndex = compareHasStarted ? Math.max(0, compareStepIndex - 1) : 0;
    renderCompareStep(compareState.first, compareVisualizers[0], visibleIndex,
compareState.type1, 1);
    renderCompareStep(compareState.second, compareVisualizers[1], visibleIndex,
compareState.type2, 2);
  }, 120);
}

window.addEventListener("resize", scheduleCompareRerender);
window.addEventListener("orientationchange", () => setTimeout(scheduleCompareRerender,
250));
```

```
window.addEventListener("algoviz-canvas-resized", scheduleCompareRerender);

window.addEventListener("algoviz-theme-changed", () => {
  if (!compareVisualizers || compareVisualizers.length === 0) return;
  if (compareState && compareState.first && compareState.second) {
    const visibleIndex = compareHasStarted ? Math.max(0, compareStepIndex - 1) : 0;
    renderCompareStep(compareState.first, compareVisualizers[0], visibleIndex,
compareState.type1, 1);
    renderCompareStep(compareState.second, compareVisualizers[1], visibleIndex,
compareState.type2, 2);
  } else {
    compareVisualizers.forEach((v) => v && v.clear && v.clear());
  }
});
```