МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

Спеціальність: 121 «Інженерія програмного забезпечення»

Освітня програма: «Розробка програмного забезпечення»

Група: 4РП-06

ДИПЛОМНИЙ ПРОЕКТ

здобувача освіти денної форми навчання

РП.06.08.000.ДП

ГРІЧИНА ВЛАДИСЛАВА ОЛЕКСАНДРОВИЧА

м. Одеса
2023 р

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

Спеціальність: **121 «Інженерія програмного забезпечення»**

Освітня програма: «**Розробка програмного забезпечення»**

Група: **4РП-06**

ПОЯСНЮВАЛЬНА ЗАПИСКА

до дипломного проекту (роботи) на тему:

Розробка та порівняння роботи алгоритмів у різних структурах даних

Проектний матеріал складається з пояснювальної записки на 79 сторінках та графічного (презентаційного) матеріалу на 19 аркушах (слайдах).

Дипломник ______________________ (Гричін В.О.)

Керівник ______________________ (Кунуп Т.В.)

Консультанти:

з економічної частини ______________________ (Копайгородська Т.Г.)

з охорони праці ______________________ (Чорновол Н.І.)

з дотримання вимог ЄСКД ______________________ (Петрашова В.І.)

старший консультант ______________________ (Кунуп Т.В.)

До захисту допущений

Голова циклової комісії ______________________ (Кривченко Ю.В.)

Завідувач відділення ______________________ (Скорнякова О.В.)

Захист «22» 06 2023 р. Протокол ДКК № 1

Оцінка ДКК 4 (добре)

Секретар ДКК ______________________

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

Відділення **комп'ютерних систем** Комісія **КТ та ПІ**

Спеціальність **121 «Інженерія програмного забезпечення»**

Освітня програма **«Розробка програмного забезпечення»**

ЗАТВЕРДЖУЮ:

Заст. дир. з НВР Беркань І.В.

"____" ________ 2023 р.

ЗАВДАННЯ

на дипломний проект (роботу)

Здобувачеві (здобувачці) освіти Грічіну Владиславу Олександровичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Розробка та порівняння роботи алгоритмів у різноманітних структурах даних

затверджена наказом по коледжу від "17" 10 2022 р. № 235-А2-ОД

2. Термін здачі закінченого проекту (роботи) 12.06.2023 р.

3. Вихідні данні до проекту (роботи)

1. Алгоритми пошуку та сортування в структурах даних;
2. Алгоритми пошуку даних на прикладах різних структур даних;
3. Алгоритми сортування даних на прикладах різних структур даних;
4. Виконання програмної реалізації алгоритму пошуку елементів в даних статичної структур;
5. Здійснення програмної реалізації алгоритмів на прикладі статичної структури масиву.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які необхідно розробити)

Аналіз алгоритмів пошуку та сортування в різних структурах даних ; Розробка алгоритмів пошуку у структурі даних масив; Розробка алгоритму сортування у структурах даних масив; Результати експерементальної частини; Економічна частина; Охорона праці

5. Перелік графічного (презентаційного) матеріалу (з точним зазначенням обов'язкових креслень, кількості слайдів)

1.Схема алгоритму даних;

2.Скриншот «Програмна реалізація алгоритму пошуку»;

3.Скриншот «Програмна реалізація алгоритму сортування»;

4.Скриншот «Аналіз ефективності різних алгоритмів сортування у структурах даних».

6. Консультанти по проекту (роботі), із зазначенням розділів проекту, що їх стосується

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Технологічний	Кунуп Т.В.		
Економічна частина	Копайгородська Т.Г.		
Охорона праці	Чорновол Н.І.		
Нормоконтроль	Петрашова В.І.		

7. Дата видачі завдання ___________________________

Керівник Кунуп Т.В. ___________________
(підпис)

Завдання прийняв до виконання ___________________
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/р	Назва етапів дипломного проекту (роботи)	Термін виконання етапів дипломного проекту (роботи)	Відмітка про виконання
1	Вступ. Постановка мети та задач проектування	5.05.2023	Виконано
2	Методи пошуку та сортування в різних структурах даних	7.05.2023	Виконано
3	Розробка алгоритмів пошуку даних на прикладах різних структур даних	9.05.2023	Виконано
4	Загальний опис реалізації алгоритмів	11.05.2023	Виконано
5	Розробк алгоритми сортування даних на прикладах різних структур даних	13.05.2023	Виконано
6	Аналіз алгоритмів сортування у різних сруктурах даних	16.05.2023	Виконано
7	Аналіз результатів, підготовка слайдів презентації	18.05.2023	Виконано
8	Економічні розрахунки та питання з охорони праці	20.05.2023	Виконано
9	Підготовка графічної частини проекту	23.05.2023	Виконано
10	Підготовка проекту до захисту та тестування алгоритмів	25.05.2023	Виконано
11	Аналіз результатів, підготовка слайдів презентації	3.06.2023	Виконано
12	Економічні розрахунки та питання з охорони праці	6.06.2023	Виконано
13	Підготовка до захисту	8.06.2023	Виконано

Дипломник ___________________
(підпис)

Керівник ___________________
(підпис)

Формат	Зона	Поз.	Позначення	Назва	Кіл.	Примітка
				<u>Документація</u>		
			РП 06. 08 000. 00 ДП	Дипломний проект		
А4			РП 06. 08 000. 00 ДП ПЗ	Пояснювальна записка	1	

Зм.	Арк.	№ докум.	Підпис	Дата	РП 06. 08 000. 00 ДП		
Розробив		Грічин В.О.			Розробка та порівняння роботи алгоритмів у різноманітних структурах даних	Літ. Н Д П	Аркуш 4 / Аркушів 72
Перевірив		Кунуп Т.В					
						ВСП “ОТФК ОНТУ” гр.4РП-06	
Н. Контр.		Петрашова В.І.					
Затверд.		Кривченко Ю.В.					

ЗМІСТ

ВСТУП **6**

1 ТЕХНОЛОГІЧНИЙ РОЗДІЛ **7**

1.1 Аналіз існуючих алгоритмів 7

1.2 Математичні основи для вивчення алгоритмів і їх властивостей 8

1.3 Алгоритм побудови мінімального кістякового дерева 10

1.4 Алгоритм оптимізації мурашиної колонії 14

1.5 Оптимізація природнього еволюційного процесу 21

1.6 Впорядковання елементів у лінійних списках та масивах за певними критеріями 25

1.7 Розробка та порівняння робои алгоритмів сортування для різних структур даних 26

1.7.1 Характеристики алгоритмів сортування 26

1.7.2 Використання алгоритмів сортування 29

1.7.3 Порівняння роботи алгоритмів сортування 43

1.7.4 Опис роботи програми 44

2. ЕКОНОМІЧНА ЧАСТИНА **52**

2.1 Резюме 52

2.2 Вивчення трудоиісткості розробки програмного забезпечення 54

2.3 Розрахунок ціни програмного продукту 54

3. ОХОРОНА ПРАЦІ **56**

3.1 Аналіз та безпека умов праці працівника на робочому місці 56

3.2 Розробка заходів з охорони праці 57

3.3 Організація робочого місця користувача ПК 58

3.4 Пожежна безпека 58

ВИСНОВКИ **59**

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ **60**

Додаток 1

ВСТУП

Одними з найважливіших процедур обробки структурованої інформації є пошук та сортування. Задачі пошуку та сортування привертали увагу вчених (програмістів) ще на початку комп’ютерної ери. З 50-х років почалося вирішення проблеми пошуку елементів, які володіють певними властивостями в заданій множині. Дослідження алгоритмів пошуку та сортування тривають і до наших днів.

У кожного алгоритму є свої переваги і недоліки. Тому важливо вибрати той алгоритм, який найкраще підходить для вирішення конкретної задачі. Існує кілька способів оцінки складності алгоритмів. Програмісти, звичайно, зосереджують увагу на швидкості алгоритму, але важливі також інші вимоги, наприклад, до розмірів пам'яті, вільного місця на диску або інших ресурсів. Швидкий алгоритм може бути мало ефективним, якщо його виконання вимагатиме більше пам'яті, ніж доступно на комп'ютері. Важливо розрізняти практичну складність, яка є точною мірою часу обчислення і обсягу пам'яті для конкретної моделі обчислювальної машини, і теоретичну складність, яка є більш незалежною від практичних умов виконання алгоритму і визначає порядок величини його вартості.

Продовжувати дослідження алгоритмів пошуку та сортування в структурах даних; аналізувати їх переваги і недоліки з точки зору швидкості виконання, використання ресурсів і практичної складності; розробляти нові алгоритми або вдосконалювати існуючі для досягнення оптимальних результатів; порівнювати різні алгоритми і визначати їх ефективність у різних сценаріях; досліджувати вплив структури даних на процеси пошуку та сортування; визначати теоретичну складність алгоритмів і їх практичну застосовність; розробляти методики тестування алгоритмів і проводити експериментальне порівняння їх продуктивності; висувати висновки та рекомендації щодо вибору оптимальних алгоритмів для конкретних завдань обробки структурованої інформації.

1 ТЕХНОЛОГІЧНИЙ РОЗДІЛ

1.1 Аналіз існуючих алгоритмів

Алгоритм є набором інструкцій, що описують послідовність дій для досягнення результату вирішення задачі за скінченну кількість дій. Він використовується для виконання дискретних процесів з метою досягнення поставленої мети за обмежений час. У випадку комп'ютерних програм, алгоритм представляє собою деталізований список інструкцій, які реалізують процес обчислення і приводять систему від початкового стану до кінцевого стану через послідовність логічних станів. Перехід між станами не обов'язково є детермінованим і може містити елементи випадковості у деяких алгоритмах.

Поняття алгоритму має глибокі коріння в математиці. Людство вже з давніх-давен використовувало обчислювальні процеси алгоритмічного характеру, такі як арифметичні операції з цілими числами або пошук найбільшого спільного дільника двох чисел. Проте чітке формалізоване поняття алгоритму сформувалося лише на початку XX століття.

Для візуалізації алгоритмів часто використовують блок-схеми, які допомагають відображати послідовність дій та зв'язки між ними. Алгоритми відіграють важливу роль у розробці програмного забезпечення, а їх аналіз та дослідження дозволяють обрати найбільш ефективні методи пошуку та сортування для різних структур даних.

Розвиток поняття алгоритму мав значний вплив на математику та комп'ютерну науку. В 1928 році Давід Гільберт поставив задачу розв'язності, яка викликала потребу в формалізації поняття алгоритму. Подальші формалізації були спрямовані на визначення поняття ефективної обчислювальності або "ефективного методу". Зокрема, в цей час були розроблені рекурсивні функції Геделя-Ербрана-Кліна, λ-числення Алонзо Черча, "Формулювання 1" Еміля Поста та машина Тюрінга, запропонована Аланом Тюрінгом.

У методології алгоритм вважається базовим поняттям і є основою для опису

методів. Виходячи з методології, поняття алгоритму стало новим якісно, ближчим до прогнозованого абсолюту. Шляхом послідовного застосування алгоритму при граничних умовах задачі можна досягти ідеального розв'язку важливих науково-практичних проблем. В сучасному світі алгоритм будь-якої діяльності в формалізованому вигляді є основою для навчання за прикладами та визначення подібності. Концепція експертних систем базується на аналізі подібності алгоритмів у різних сферах діяльності.

Алгоритми відіграють ключову роль у розв'язанні завдань і їх дослідження допомагає вибрати найбільш оптимальні методи пошуку та сортування для різних структур даних. Історичний розвиток поняття алгоритму дав поштовх до розуміння ефективності обчислювальних процесів та створення комп'ютерних систем алгоритму, включаючи рекурсивні функції Геделя-Ербрана-Кліна та машину Тюрінга.

1.2 Математичні основи для вивчення алгоритмів і їх властивостей

Завдяки розробці цих формалізацій, стало можливим встановити математичні основи для вивчення алгоритмів і їх властивостей. Це дало змогу визначити обмеження та можливості обчислювальних процесів. Концепція алгоритму стала ключовою в комп'ютерній науці та інформатиці, де вивчаються проблеми обчислювальної складності, оптимізації алгоритмів та розробки нових методів розв'язання задач.

Алгоритми широко застосовуються в сучасному світі у різних сферах діяльності. Вони використовуються для розв'язання завдань в інформаційних технологіях, науці, бізнесі, фінансах, медицині, транспорті та багатьох інших галузях. Важливим аспектом роботи з алгоритмами є їх аналіз, тестування та вдосконалення з метою досягнення кращої ефективності і точності результатів.

В цілому, розуміння та розвиток алгоритмів відіграють важливу роль у розвитку сучасної науки та технологій. Це допомагає здійснювати складні

обчислення, розробляти нові програми та системи, а також знаходити оптимальні рішення в різних галузях людської діяльності.

Рисунок 1.1. Баронеса Ада Лавлейс, яку вважають першим програмістом.

У 1990-х роках з'явилися мови програмування, які базувалися на об'єктно-орієнтованому підході, такі як C++, Java та Python. Ці мови набули великої популярності та стали широко використовуватися для розробки різноманітних програмних систем.

Одночасно з розвитком мов програмування, з'явилися й розширення в області алгоритмічного підходу. Наприклад, виникла ідея рекурсивних алгоритмів, де алгоритм може викликати сам себе. Це дало змогу елегантно вирішувати деякі складні завдання, такі як обробка дерев або графів.

Залежно від потреб та конкретних вимог, сьогодні існує велика кількість мов програмування з різними парадигмами та призначеннями. Деякі з них спеціалізуються на швидкості обчислень (наприклад, C), інші - на простоті та широкому застосуванні (наприклад, Python), а деякі-на паралельному обчисленні (наприклад, MPI або CUDA).

Розвиток мов програмування та алгоритмів є невід'ємною частиною інформатики і комп'ютерних наук. Він продовжується й досі, і нові інновації у цій

галузі сприяють створенню більш потужних та ефективних програмних систем, які знаходять застосування в різних сферах життя.

Логічне програмування, представлене мовою Пролог, базується на використанні формальної логіки для опису задач та їх розв'язання. Замість прямого опису послідовності кроків для виконання завдання, в логічному програмуванні розробник описує відношення, які повинні бути задоволені для вирішення проблеми.

У логічному програмуванні використовується інференційний механізм, що дозволяє автоматично знаходити відповіді на запити, використовуючи логічні правила та факти, які були визначені розробником. Це дає можливість вирішувати складні задачі, включаючи логічне розміщення об'єктів, планування, штучний інтелект та інші.

Парадигма логічного програмування виявилася корисною у багатьох областях, включаючи природну мову обробки, експертні системи, аналіз баз знань та розв'язання логічних задач.

Поміж інших розповсюджених мов логічного програмування можна відзначити Datalog, Answer Set Programming (ASP) та Prolog-подібні мови, які застосовуються в багатьох наукових дослідженнях та практичних застосуваннях.

Логічне програмування продовжує розвиватися, включаючи інтеграцію з іншими парадигмами програмування, такими як об'єктно-орієнтоване програмування та функціональне програмування. Це дає можливість розробникам використовувати найкращі підходи для вирішення конкретних завдань та побудови складних програмних систем.

1.3 Алгоритм побудови мінімального кістякового дерева

Алгоритм побудови мінімального кістякового дерева - це алгоритм Прима

Загальна схема алгоритму:

MST-PRIM(G,w, r)

1: Ω < V[G]

2: foreach (для кожної) вершини u ∈ Ω

3: do key[u] < ∞

4: key[r] < 0

5: p[r] = NIL

6: while (пока) Ω ≠ 0

7: do u < EXTRACT-MIN(Ω)

8: foreach (для кожної) вершини v ∈Adj(u)

9: do if v ∈Ω и w(u,v) < key[v] then

10: p[v] < u

11: key[v] < w(u,v)

12: DECREASE-KEY(Ω, v, w(u,v)) 13: return MST

14: DECREASE-KEY(Ω, v, w(u,v))

Для продовження алгоритму Прима, в якому ми будуємо мінімальне кістякове дерево, продовжимо з кроку 12:

Крок 12 виконує зміну пріоритету вершини v в черзі Ω на нове значення w(u,v). Це необхідно, оскільки ми знайшли безпечне ребро, яке з'єднує вершину v з поточним мінімальним кістяком. Зміна пріоритету дозволяє нам оновити значення key[v] та зберегти властивість черги з пріоритетами.

Після цього ми повертаємося до кроку 6, де перевіряємо, чи є ще вершини, які ще не потрапили в мінімальне кістякове дерево. Якщо Ω (черга вершин) не порожня, ми повторюємо процес знаходження безпечного ребра та оновлення пріоритетів для наступної вершини.

Це триває до тих пір, поки у черзі Ω є вершини для обробки. По завершенні алгоритму ми повертаємо побудоване мінімальне кістякове дерево як результат.

Отже, алгоритм Прима здатний знаходити мінімальне кістякове дерево шляхом додавання безпечних ребер з найменшою вагою на кожному кроці. Використання черги з пріоритетами дозволяє швидко вибирати безпечне ребро, забезпечуючи ефективність алгоритму.

Примітка: В кроці 14 ми повторно виконуємо крок 12 для можливих інших ребер, що з'єднують вершину v з іншими вершинами. Це можливо, якщо є кілька ребер з однаковою вагою.

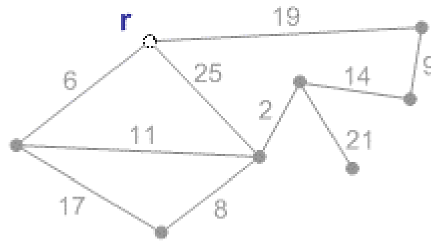


Рисунок 1.2. Схема роботи алгоритму Прима

Час роботи алгоритму Прима залежить від реалізації черги з пріоритетами. Зазначена оцінка часу роботи відповідає використанню двійкової купи. Така реалізація вимагає ініціалізації у рядках 1-4 за час O(V) і виконання операції EXTRACT-MIN (EXTRACT-MI) у рядках 7-8, яка забирає час O(VlogV). Цикл for у рядках 8-11 виконується в сумі O(E) разів, а перевірка приналежності в рядку 9 може бути виконана за час O(1) за умови збереження стану черги у вигляді бітового вектора розміру |V|. Операція зменшення ключа (DECREASE-KEY) у рядку 11 для двійкової купи забирає час O(logV).

Отже, загальний час роботи алгоритму Прима з двійковою купою становить O(VlogV + ElogV) = O(ElogV).

Проте, застосування фібоначчієвих куп дозволяє отримати кращу оцінку часу роботи. Використання фібоначчієвих куп дозволяє виконувати операцію EXTRACT-MIN за розрахунковий час O(logV) і операцію DECREASE-KEY за розрахунковий час O(1). У такому випадку загальний час роботи алгоритму Прима.

Алгоритм Крускала є ще одним алгоритмом для побудови мінімального кістякового дерева в зваженому зв'язному ненаправленому графі. Принцип дії алгоритму Крускала полягає в послідовному об'єднанні компонентів графу шляхом додавання ребер з найменшою вагою, доки не буде сформоване мінімальне кістякове дерево.

Основні кроки алгоритму Крускала наступні:

1. Сортування всіх ребер графу за зростанням їх ваги.

2. Створення порожньої множини MST, яка представлятиме мінімальне кістякове дерево.

3. Прохід по всіх ребрах, відсортованих на попередньому кроці.

4. Якщо поточне ребро не утворює циклу в MST, додаємо його до MST. Інакше ігноруємо ребро і переходимо до наступного. Повторюємо крок 3, доки не пройдемо всі ребра.

Алгоритм Крускала використовує структуру даних "розбиття-злиття" (disjoint-set), щоб визначити, чи утворює поточне ребро цикл. Це забезпечує ефективну перевірку циклів і дозволяє алгоритму працювати з часовою складністю O(E log E), де E - кількість ребер в графі.

Алгоритм Крускала є гарним варіантом для використання у графах з великою кількістю ребер або коли необхідно знайти мінімальне кістякове дерево в графі, де ребра мають різні ваги. Він також може бути застосований для знаходження максимального кістякового дерева, змінивши порядок сортування ребер на спадний.

Загальна схема алгоритму Крускала:

1: Відсортувати ребра графу за зростанням ваги.

2:Створити порожню множину MST для збереження мінімального кістякового дерева.

3: Пройти по відсортованих ребрах. a. Якщо додавання поточного ребра не створює цикл в MST, додати його до MST. b. Інакше ігнорувати ребро.

4: Повернути MST як мінімальне кістякове дерево.

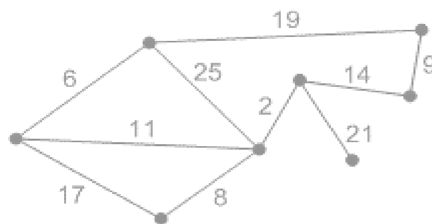


Рисунок 1.3. Приклад виконування алгоритму

Так, для зберігання непересічних множин у методі Крускала

використовується метод з об'єднанням за рангом і стиском шляхів. Цей метод дозволяє швидко виконувати операції об'єднання двох множин і пошуку кореня елемента.

При ініціалізації, кожен елемент графу розглядається як окрема множина. Ініціалізація займає час O(V), де V - кількість вершин графу.

Ребра графу сортуються за вагою, що займає час O(E log E), де E - кількість ребер графу.

Після сортування ребер, проходить цикл по відсортованим ребрам, що займає O(E) операцій. У цьому циклі виконується перевірка, чи додання поточного ребра утворює цикл в MST. Якщо цикл не утворюється, ребро додається до MST. Ця перевірка виконується за допомогою методу об'єднання за рангом і стискання шляхів, що забезпечує швидку перевірку і об'єднання множин.

Загальний час роботи алгоритму Крускала складає O(E log E), оскільки найбільше часу витрачається на сортування ребер.

Застосування методу з об'єднанням за рангом і стисканням шляхів дозволяє досягти ефективності алгоритму Крускала та забезпечити швидке об'єднання множин без утворення циклів.

1.4 Алгоритм оптимізації мурашиної колонії

Алгоритм оптимізації мурашиної колонії (Ant Colony Optimization, ACO) є метаевристичним алгоритмом, який базується на природному поведінці мурах для розв'язання оптимізаційних задач. Цей алгоритм надихнутий спостереженнями поведінки мурашок, особливо їх здатності знаходити найкоротший шлях між місцями з їжею та мурашником.

Основна ідея алгоритму полягає в тому, що мурашки залишають феромонні сліди на своєму шляху, а інші мурашки сприймають ці сліди і використовують їх для прийняття рішень щодо вибору шляху. Чим більша кількість феромонів на шляху тим більше ймовірностей, що інша мурашка обере цей шлях.

Основні кроки алгоритму оптимізації мурашиної колонії:

1. Ініціалізація феромонів: Кожне ребро графу, що представляє оптимізаційну задачу, ініціалізується початковими значеннями феромонів.
2. Переміщення мурашок: Кожна мурашка стартує зі стартової точки і рухається по графу, вибираючи наступний вузол на основі ймовірностей, що визначаються феромонами та іншими факторами, такими як відстань до вузла та евристична інформація.
3. Оновлення феромонів: Після того, як всі мурашки завершать свій шлях, феромони оновлюються на основі результатів. Феромони на кращих шляхах посилюються, що збільшує ймовірність вибору цих шляхів в майбутньому. Феромони на менш ефективні.

Робота починається з розміщення мурашок у вершинах графа (містах), потім починається рух мурашок – напрям визначається імовірнісним методом, на підставі формули:

$$P_i = \frac{{l_i}^q \cdot {f_i}^p}{\sum_{k=0}^{N} {l_k}^q \cdot {f_k}^p} \quad (1.1)$$

де:

Pi – вірогідність переходу по дорозі,

li – довжина i-ого переходу,

fi – кількість феромонів на i-ому переході,

q – величина, яка визначає «жадібність» алгоритму,

p – величина, яка визначає «стадність» алгоритму і q + p = 1.

Результат не є точним і навіть може бути одним з гірших, проте, в силу імовірності рішення, повторення алгоритму може видавати (досить) точний результат. Алгоритм оптимізації мурашиної колонії можна описати наступним чином:

1. Ініціалізація:
2. Визначити граф задачі, де вузли представляють рішення, а ребра - залежності між рішеннями.
3. Ініціалізувати феромони на всіх ребрах графу початковими значеннями.

4. Генерація мурашок:

Створити популяцію мурашок, кожна з яких розміщується у випадковому вузлі графу.

5. Переміщення мурашок:

Кожна мураха вибирає наступне рішення на основі ймовірностей, які залежать від феромонів на ребрах та інших факторів, таких як відстань та евристична інформація.

Мурахи продовжують переходити з вузла в вузол до досягнення кінцевого вузла.

6. Оновлення феромонів:

Після завершення всіх мурахами свого шляху, оновити феромони на ребрах графу.

7. Феромони на кращих шляхах підсилюються, а на менш ефективних шляхах випаровуються або зменшуються.

8. Повторення:

Повторити кроки 3-4 до досягнення заданої умови зупинки, наприклад, заданої кількості ітерацій або зміни значення об'єктивної функції.

9. Вибір оптимального рішення:

Вибрати найкраще рішення, знайдене протягом всіх ітерацій.

Алгоритм оптимізації мурашиної колонії може бути додатково покращений за допомогою різних підходів, таких як впровадження елітних мурах, локального посилення феромонів та інших варіацій.

Так, використання феромонів як засобу спілкування між мурахами в системі має велике значення. Феромони є хімічними речовинами, які мурахи виділяють і залишають на своєму шляху. Вони служать якісною ознакою середовища, що допомагає мурахам спілкуватися та обмінюватися інформацією.

Коли мураха знаходить джерело їжі, вона повертається до гнізда, залишаючи за собою слід з феромонів. Інші мурахи, які перебувають у безпосередній близькості до цього сліду, сприймають цю інформацію і використовують її для визначення свого маршруту. Чим більша концентрація феромонів на певному шляху, тим більш ймовірно, що інші мурахи оберуть саме цей шлях.

З часом, якщо деякий шлях виявляється більш корисним, тобто мурахи успішно досягають їжі, то кількість феромонів на цьому шляху збільшується. З іншого боку, феромони поступово випаровуються, що призводить до зменшення їх концентрації на менш використовуваних шляхах. Цей зворотний зв'язок дозволяє системі самоорганізовуватися і зосереджувати увагу мурах на найкоротших шляхах до джерела їжі.

Важливо відзначити, що сам алгоритм мурашиної колонії базується на багатоагентному підході, де кожна мураха діє локально, приймаючи рішення на основі доступної інформації. Принцип взаємодії мурах, посилення шляхів з більшими феромонами і і випаровування феромонів з менш використовуваних шляхів сприяє пошуку оптимального шляху у системі. Механізм роботи данного алгориму зазданачено на рисунку 3.

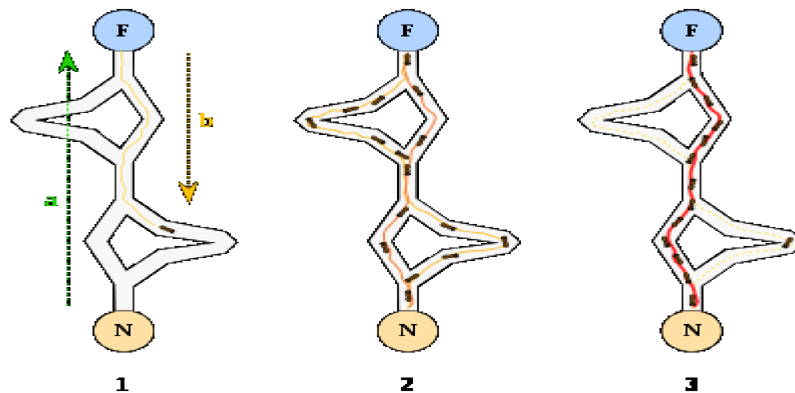


Рисунок 1.4. Механізм роботи алгоритма оптимізації мурашиної колонії

$$p_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})}{\sum(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})} \quad (1.4)$$

де:

$\tau_{i,j}$ – це кількість феромонів на краю i,j;

α – параметр впливу на $\tau_{i,j}$;

$\eta_{i,j}$ – бажаний край i,j (апріорного знання, майже завжди, $1 / d_{i,j}$, де d відстань);

β – параметр впливу на $\eta_{i,j}$.

Оновлення феромонів

$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \Delta\tau_{i,j}$

де:

$\tau_{i,j}$ – кількість феромону на дузі i,j

ρ – швидкість випарення феромону

$\Delta\tau_{i,j}$ – кількість відкладеного феромону, зазвичай визначається як

$$\Delta\tau_{i,j}^{k} = \begin{cases} 1/L_k & \text{if ant } k \text{ travels on edge } i,j \\ 0 & \text{otherwise} \end{cases} \quad (1.3)$$

де:

L_k – вартість k-го шляху мурахи (зазвичай довжина).

Мурашиний алгоритм знаходить застосування в багатьох областях, особливо там, де необхідно здійснювати оптимізацію на основі графових моделей. Основні області застосування мурашиного алгоритму включають:

Задача комівояжера: Мурашиний алгоритм може знайти ефективний маршрут для обходу набору місць, мінімізуючи загальну відстань подорожі.

Транспортне завдання: Мурашиний алгоритм може допомогти вирішити задачу найбільш ефективного розподілу вантажів між джерелами та призначеннями з урахуванням обмежень і вартості перевезення.

Календарне планування: Мурашиний алгоритм може допомогти оптимізувати розподіл завдань та ресурсів в часових графіках, забезпечуючи ефективне використання часу та мінімізуючи конфлікти.

Розфарбування графа: Мурашиний алгоритм може вирішувати задачу розфарбування графа, де кожен вузол графа повинен бути призначений одному з кольорів, з урахуванням обмежень на кількість суміжних вузлів з однаковим кольором.

Завдання оптимізації мережних трафіків: Мурашиний алгоритм може бути використаний для оптимізації маршрутизації трафіку в мережах з метою мінімізації затримок, витрати пропускної здатності або інших метрик.

Ці області застосування є лише деякими прикладами, і мурашиний алгоритм може бути адаптований до вирішення багатьох інших оптимізаційних задач. Модифікації алгоритму можуть включати динамічне адаптаційне настроювання параметрів, покращення правил вибору шляху на основі феромонів, використання інших методів випаровування феромонів та удосконалення початкового розподілу феромону для покращення якості отримуваних рішень.

Алгоритм мурашиної колонії має кілька достоїнств, які роблять його привабливим для використання в різних завданнях оптимізації:

Пам'ять про всю колонію: У мурашиної колонії інформація про здійснені вибори та результати їх виконання зберігається в феромонному сліді на графі. Це дозволяє враховувати колективний досвід всієї колонії при виборі оптимального шляху. У генетичних алгоритмах, зазвичай, зберігається інформація тільки про попереднє покоління.

Менша вразливість до початкових рішень: Мурашиний алгоритм має властивість декомпозиції проблеми, тобто вибір кращих рішень залежить від активності мурах під час пересування. Це дозволяє уникнути застрягання в локальних оптимумах, які можуть бути викликані неоптимальним початковим рішенням.

Адаптивність до змін: Мурашиний алгоритм може працювати в динамічних середовищах, де умови змінюються з часом. Завдяки феромонному сліду, мурахи можуть швидко адаптуватися до нових умов і змінювати свої вибори шляхів для досягнення оптимального розв'язку.

Незважаючи на свої переваги, алгоритм мурашиної колонії також має певні недоліки:

Висока обчислювальна складність: Залежно від розмірності проблеми та кількості мурах, алгоритм може вимагати значних обчислювальних ресурсів. При великих масштабах задачі виконання алгоритму може бути часово витратним.

Залежність від налаштування параметрів: Як я згадував раніше, налаштування параметрів, таких як інтенсивність випаровування феромону і ймовірність вибору шляху, можуть суттєво впливати на якість отримуваних

рішень. Неправильне налаштування може призвести до погіршення ефективності алгоритму.

Обмежена глобальна інформація: Хоча алгоритм зберігає колективний досвід колонії, він може бути обмежений доступом до глобальної інформації. Мурахи вибирають шляхи на основі локальної інформації, яка може бути недостатньою для знаходження глобально оптимального розв'язку.

У цілому, алгоритм мурашиної колонії є потужним інструментом оптимізації, але його ефективність залежить від конкретного завдання, налаштування параметрів і обмежень, які в ньому присутні.

Ви правильно вказали на деякі недоліки алгоритму мурашиної колонії. Додам до них наступне:

Висока часова складність: При роботі зі складними задачами, особливо з великою кількістю вузлів або шляхів, алгоритм мурашиної колонії може вимагати значних обчислювальних ресурсів і бути часово витратним.

Підходить для задач з дискретним простором розв'язків: Мурашиний алгоритм зазвичай використовується для задач з дискретним простором розв'язків, таких як комівояжер, розфарбування графа тощо. Для задач з неперервними просторами розв'язків, наприклад, у задачах оптимізації функцій, потрібні спеціальні модифікації або інші алгоритми.

Ризик застрягання в локальних оптимумах: Хоча алгоритм має механізм для уникнення локальних оптимумів за рахунок стохастичних виборів та феромонного сліду, існує ризик застрягання в підоптимальних рішеннях. Тому може бути потрібне використання додаткових методів, таких як локальний пошук, для поліпшення отриманих розв'язків.

Обмежена глобальна інформація: Мурашиний алгоритм базується на локальній взаємодії мурах і обміні локальною інформацією. Це може призвести до обмеженості глобальної інформації, особливо якщо задача має складну структуру або вимагає глибокого аналізу всього простору розв'язків.

Незважаючи на ці недоліки, алгоритм мурашиної колонії залишається ефективним інструментом.

1.5 Оптимізація природнього еволюційного процесу

Генетичний алгоритм є потужним методом оптимізації, який імітує природний еволюційний процес. Основні компоненти генетичного алгоритму включають:

Популяція: Сукупність потенційних розв'язків (індивідів), які представлені у вигляді генетичного коду. Популяція може бути ініціалізована випадковими розв'язками або за допомогою певних евристичних методів.

Оцінка пристосованості: Кожному індивіду в популяції надається оцінка пристосованості, що відображає якість його розв'язку відповідно до задачі оптимізації. Це може бути значення функції цілі, яку потрібно мінімізувати або максимізувати.

Відбір: Індивіди з більшою пристосованістю мають більшу ймовірність бути вибраними для формування нової популяції. Цей процес сприяє збільшенню якості розв'язків у популяції з покоління в покоління.

Схрещування: Вибрані індивіди поєднуються з метою створення нових потомків. Це може включати обмін генетичним матеріалом між батьківськими індивідами, що може включати різні методи схрещування, такі як одноточкове схрещування або багатоточкове схрещування.

Мутація: З малим ймовірністю генетичні оператори, відомі як мутація, виконуються для внесення випадкових змін у генетичний код індивіда. Це дозволяє розвивати нові варіації розв'язків і уникати застрягання в локальних оптимумах.

Заміна: Сформовані потомки замінюють частину популяції, що дозволяє розвивати популяцію з покоління в покоління. Цей процес може включати заміну за принципом елітарності, де найкращі індивіди зберігаються у новій популяції.

Генетичні алгоритми мають кілька переваг, таких як можливість знаходження оптимальних або близьких до оптимальних розв'язків для складних задач оптимізації, здатність працювати зі змінними, дискретними або

неперервними просторами розв'язків, а також відносну простоту реалізації та гнучкість.

Проте, генетичні алгоритми також мають свої обмеження. Наприклад, вони можуть бути вимогливими до обчислювальних ресурсів, особливо при оптимізації складних задач. Крім того, ефективність генетичного алгоритму сильно залежить від налаштування його параметрів, які можуть бути нетривіальними для визначення.

Є кілька можливих критеріїв зупинки генетичного алгоритму:
Досягнення максимальної кількості ітерацій або поколінь.
Досягнення заданого значення функції пристосованості або метрики якості.
Збіжність популяції, тобто стабілізація значень функції пристосованості на протязі кількох поколінь.

Вичерпання обчислювальних ресурсів (часу, пам'яті, обчислювальної потужності) для подальшого виконання алгоритму.

Відсутність значимого поліпшення протягом певної кількості поколінь.

Вибір конкретного критерію залежить від задачі та вимог до оптимізації. Іноді можуть використовуватися комбінації кількох критеріїв для зупинки алгоритму.

Загальна схема генетичного алгоритму має такі кроки:

Ініціалізація: Створення початкової популяції із випадково згенерованих хромосом.

Оцінка пристосованості: Обчислення значення функції пристосованості для кожної хромосоми в популяції.

Вибір: Вибір батьківських хромосом для схрещування на основі їхньої пристосованості. Частіше обираються хромосоми з більшим значенням пристосованості.

Схрещування: Застосування генетичного оператора схрещування для створення нащадків шляхом комбінування генетичного матеріалу батьківських хромосом.

Мутація: Застосування генетичного оператора мутації для випадкової зміни деяких генів у створених нащадках.

Оцінка пристосованості нащадків: Обчислення значення функції пристосованості для створених нащадків.

Заміна: Заміна частини початкової популяції нащадками на основі їхньої пристосованості.

Повторення: Повторення кроків 3-7 до досягнення критерію зупинки.
Таким чином, генетичні алгоритми здатні здійснювати ефективний пошук рішень у просторі параметрів, використовуючи елементи еволюції та взаємодіючи зі значеннями пристосованості для вдосконалення популяції покоління в покоління.

Розглядається також розрізнення етапів генетичного алгоритму. Пропоную розглянути детальніше кожен з них:
Створення початкової популяції:

Випадковим чином генерується початкова множина хромосом (осіб) для представлення потенційних рішень. Кількість осіб і їхні хромосоми залежать від розміру задачі та вимог до оптимізації.

Обчислення функції пристосованості:
Кожна особа в популяції оцінюється на основі функції пристосованості, яка визначає якість її рішення.

Фітнес-функція може бути задана відповідно до постановки задачі і може оцінювати критерії якості, ефективності чи вартості рішення.

Повторювання до досягнення критерію зупинки:
Вибір індивідів (батьків) для схрещення або мутації. Індивіди вибираються з більшою ймовірністю в залежності від їхньої пристосованості, що підсилює перенесення корисних хромосом у наступні покоління.

Схрещення (кросовер) здійснює комбінування генетичного матеріалу (хромосом) двох батьків з метою створення нащадка (дитини) з новими характеристиками.

Мутація застосовується до окремих генів хромосоми для випадкової зміни значень з метою розширення простору пошуку та збереження різноманітності популяції.

Обчислення функції пристосованості для нащадків з метою оцінювання якості їхніх рішень.

Формування нового покоління шляхом заміни частини початкової популяції нащадками, вибраними залежно від їхньої пристосованості.

Після цих етапів алгоритм повторюється до досягнення критерію зупинки, який може бути заданий на основі кількості ітерацій, досягнення певного рівня пристосованості або інших критеріїв, залежно від задачі. Кінцевий результат генетичного алгоритму є набір хромосом, які представляють оптимальні або прийнятні рішення для вхідної задачі оптимізації.

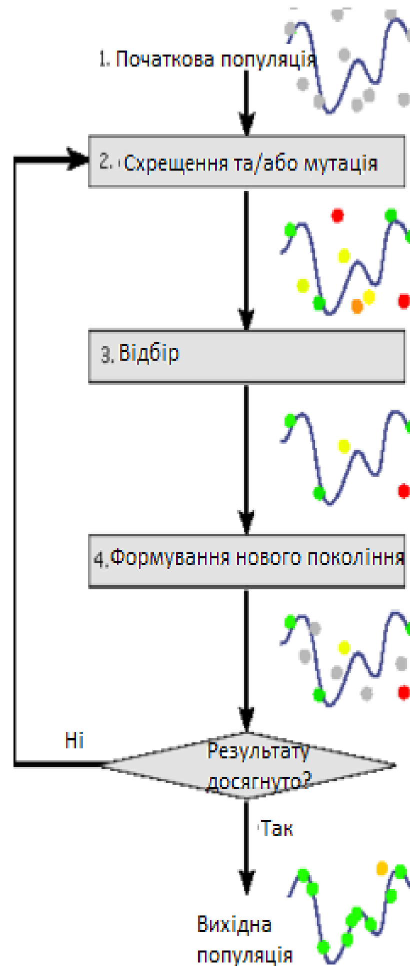


Рисунок 1. 5. Схема роботи генетичного алгоритму

1.6 Впорядкування елементів у лінійних списках та масивах за певними критеріями

Справді, алгоритм сортування є процедурою, що впорядковує елементи у лінійному списку чи масиві за певними критеріями. Задача сортування полягає в тому, щоб переставити елементи у такому порядку, який задовольняє вимогам посортованості, наприклад, від мінімального до максимального значення або у зростаючому порядку.

Сортування є важливим алгоритмічним завданням, оскільки ефективне впорядкування даних дає змогу полегшити пошук, злиття, фільтрацію та інші операції з даними. Також, впорядковані дані сприяють зрозумілості та зручності роботи з ними.

Існує багато різних алгоритмів сортування, кожен з яких має свої переваги та обмеження щодо швидкості, ефективності та використання пам'яті. Деякі з найвідоміших алгоритмів сортування включають:

Сортування бульбашкою (Bubble Sort)

Сортування вибором (Selection Sort)

Сортування вставками (Insertion Sort)

Сортування швидкими операціями (Quicksort)

Сортування злиттям (Merge Sort)

Сортування купою (Heapsort)

Сортування злиттям з використанням програвачів (Tim Sort)

Сортування підрахунком (Counting Sort)

Сортування відбором з використанням програвачів (Radix Sort)

Кожен з цих алгоритмів має свою складність, ефективність та відмінності у роботі. Вибір конкретного алгоритму сортування залежить від характеристик вхідних даних, розміру списку та вимог до швидкості та пам'яті.

1.8 Розробка та порівняння роботи алгоритмів сортування для різних структур даних

1.7.1 Характеристики алгоритмів сортування

Характеристики алгоритмів сортування включають різні аспекти їхньої поведінки та ефективності. Основні характеристики алгоритмів сортування включають:

Ефективність (часова складність): Часова складність визначає, як швидко алгоритм впорядковує дані в залежності від розміру вхідного набору даних. Вона вимірюється у вигляді кількості операцій або порівнянь, які потрібні для завершення сортування. Ефективність алгоритмів сортування може варіюватися від квадратичного (наприклад, вставкове сортування) до лінійного (наприклад, швидке сортування) і навіть до логарифмічного (наприклад, сортування злиттям) часу.

Додатковий простір (пам'ять): Додатковий простір вказує на кількість додаткової пам'яті, яка потрібна для виконання алгоритму сортування, окрім самого вхідного набору даних. Деякі алгоритми сортування можуть вимагати значно більше додаткової пам'яті, особливо при великих розмірах вхідних даних, що може бути обмеженням у деяких обмежених середовищах.

Стабільність: Стабільність алгоритму сортування означає, що він зберігає відносний порядок елементів з однаковими значеннями. Іншими словами, якщо два елементи мають однакове значення, і один із них з'являється перед іншим у вихідному наборі даних, то після сортування перший елемент все ще має з'являтися перед другим. Не всі алгоритми сортування є стабільними і в деяких випадках стабільність може бути важливою властивістю.

Рекурсивність: Деякі алгоритми сортування можуть бути рекурсивними, тобто вони використовують самі себе для сортування менших піднаборів даних. Це може спростити реалізацію алгоритму, але також може мати вплив на швидкодію та споживання пам'яті.

Адаптивність: Адаптивні алгоритми сортування можуть використовувати існуючий впорядок даних для покращення ефективності. Наприклад, якщо деякі частини вхідного набору даних вже впорядковані, адаптивний алгоритм може використовувати цю інформацію для швидкого сортування.

Вплив на початковий набір даних: Деякі алгоритми сортування змінюють початковий набір даних безпосередньо, тоді як інші створюють новий відсортований набір. Це може бути важливою характеристикою, якщо початкові дані мають бути збережені в незмінному стані.

Кожен алгоритм сортування має свої переваги та недоліки в залежності від конкретних ситуацій та вимог до ефективності, тому вибір певного алгоритму залежить від конкретного випадку використання.

Обчислювальна складність алгоритму визначається кількістю обчислювальних операцій, необхідних для виконання алгоритму, в залежності від розміру вхідних даних. Вона дає загальну оцінку того, наскільки алгоритм ефективний з точки зору використання ресурсів, таких як час та пам'ять.

Обчислювальна складність може бути виражена у різних вимірах, таких як:

Часова складність: Визначає кількість обчислювальних операцій, які потрібні для завершення алгоритму. Вимірюється у кількості операцій або кількості кроків, які алгоритм потребує для обробки вхідних даних. Часова складність може бути виражена у вигляді точної кількості операцій (наприклад, O(n^2)) або у вигляді асимптотичної оцінки (наприклад, O(n log n)), яка вказує на зростання складності алгоритму при збільшенні розміру вхідних даних.

Пам'ятівідчутливість: Визначає кількість пам'яті, необхідної для виконання алгоритму. Вимірюється у кількості пам'яті, яка використовується для зберігання даних та проміжних результатів обчислень. Пам'ятівідчутлівість може бути виражена у вигляді точної кількості пам'яті (наприклад, O(n)) або у вигляді асимптотичної оцінки.

Просторова складність: Визначає загальний обсяг пам'яті, необхідний для виконання алгоритму, включаючи як використану пам'ять для даних, так і додаткову пам'ять для проміжних результатів.

Просторова складність може бути виражена у вигляді точної кількості пам'яті або у вигляді асимптотичної оцінки.

Враховуючи обчислювальну складність алгоритму, можна оцінити його ефективність та придатність для використання у конкретних ситуаціях. Зазвичай, намагаються знайти компроміс між часовою складністю та просторовою складністю алгоритму, залежно від конкретних вимог та обмежень задачі.

Основні приклади складностей алгоритмів включають наступні категорії:

Константна складність (O(1)): Алгоритм має постійну кількість операцій незалежно від розміру вхідних даних. Прикладом може бути отримання значення з певного індексу у масиві, обчислення простої формули або виконання фіксованої кількості кроків.

Лінійна складність (O(n)): Кількість операцій залежить від розміру вхідних даних. Прикладами можуть бути ітерація по всім елементам масиву або списку, пошук певного значення у несортованому списку, сумування всіх елементів тощо.

Квадратична складність (O(n^2)): Кількість операцій залежить від квадрата розміру вхідних даних. Це типовий приклад вкладених циклів. Наприклад, вкладений цикл, що проходиться по кожній парі елементів у масиві, або сортування вибором.

Логарифмічна складність (O(log n)): Кількість операцій зростає логарифмічно з розміром вхідних даних. Це часто спостерігається у бінарних алгоритмах, таких як бінарний пошук або деякі поділ-і-перевиношування алгоритмів.

Експоненціальна складність (O(2^n)): Кількість операцій зростає експоненційно з розміром вхідних даних. Це дуже невпорядкована складність, і алгоритми з такою складністю зазвичай використовуються лише для дуже малих розмірів вхідних даних.

Це лише декілька прикладів складностей алгоритмів, існують інші категорії складностей, такі як логлінійна (O(n log n)), кубічна (O(n^3)), факторіальна (O(n!)) і т.д. Вибір оптимального алгоритму залежить від розміру вхідних даних, обмежень часу та пам'яті, а також від конкретних вимог задачі.

1.7.2 Використання алгоритмів сортування

Існує багато різних алгоритмів сортування з різними характеристиками та складностями. Ось декілька найпоширеніших алгоритмів сортування:

Сортування бульбашкою (Bubble Sort):

Цей алгоритм порівнює пари сусідніх елементів і міняє їх місцями, якщо вони не впорядковані. Він продовжує проходитися по масиву, поки не буде досягнуто повного впорядкування. Це один з найпростіших алгоритмів сортування, але має квадратичну складність.

Сортування вставками (Insertion Sort): Цей алгоритм розглядає елементи послідовно та вставляє кожен елемент у відповідне місце в відсортованій частині масиву. Він повторює цю операцію для кожного елемента, поки весь масив не буде впорядкований. Вставкове сортування також має квадратичну складність, але виконує менше порівнянь, ніж сортування бульбашкою.

Сортування вибором (Selection Sort): Цей алгоритм обирає найменший елемент з несортованої частини масиву та розміщує його у відповідному місці впорядкованої частини. Він продовжує цей процес для кожного елемента до повного впорядкування масиву. Сортування вибором також має квадратичну складність.

Сортування злиттям (Merge Sort): Цей алгоритм використовує стратегію "розділяй і володарюй". Він рекурсивно розбиває масив на половини, сортує їх окремо, а потім злитає впорядковані підмасиви, формуючи відсортований масив. Сортування злиттям має складність O(n log n) і є стабільним алгоритмом сортування.

Швидке сортування (Quick Sort): Цей алгоритм також використовує стратегію "розділяй і володарюй". Він обирає підмасив і розташовує його елементи таким чином, що всі елементи, менші за обраний елемент (півподіл), розташовуються перед ним, а всі елементи, більші за нього, розташовуються після нього. Потім цей процес повторюється рекурсивно для підмасивів з меншими та

більшими елементами. Швидке сортування також має складність O(n log n) і є одним з найшвидших алгоритмів сортування в середньому.

Це лише кілька прикладів алгоритмів сортування, існує ще багато інших алгоритмів з різними властивостями та складностями, які можуть бути використані в залежності від конкретних вимог задачі.

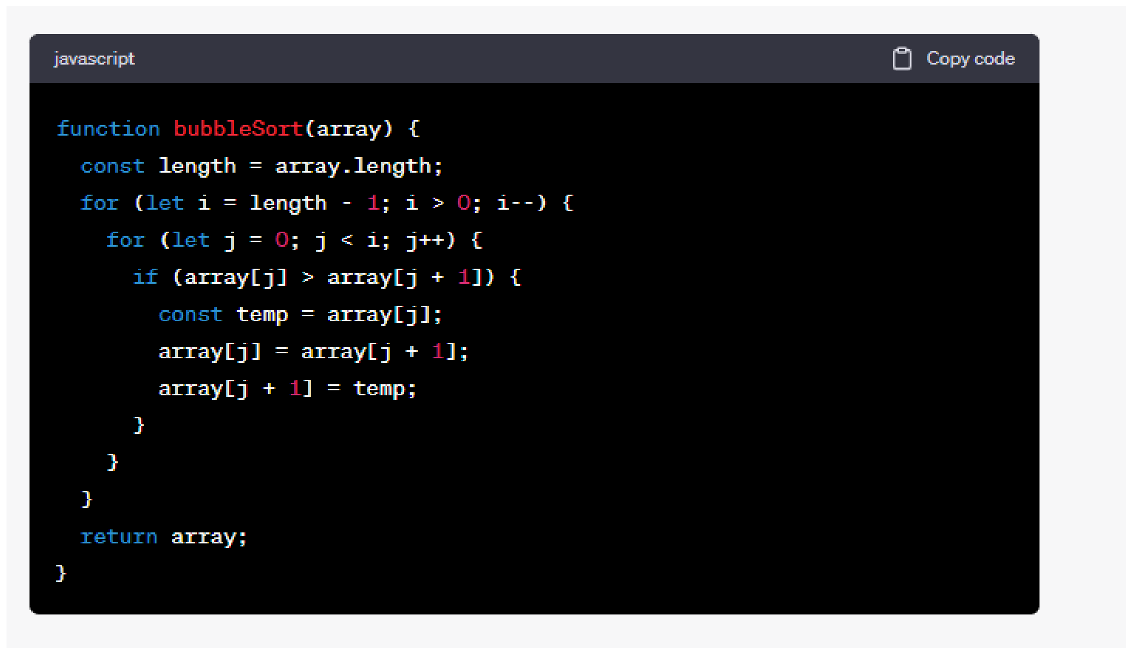


Рисунок 1.6. Скриншот кода алгоритмів сортування

В цьому випадку було виправлено умову внутрішнього циклу for, щоб j не досягало значення i-1, але лише i. Також замість деструктивного присвоювання було використано тимчасову змінну temp для обміну значень місцями.

Так, ви правильно описали суть алгоритму бульбашкового сортування. Для кращого розуміння давайте розглянемо його кроки більш детально:

Починаємо з невідсортованого масиву елементів.

Проходимося по масиву, порівнюючи кожну пару сусідніх елементів.

Якщо елементи знаходяться у неправильному порядку (менший елемент після більшого), то ми їх обмінюємо місцями, встановлюючи правильний порядок.

Продовжуємо цей процес для кожної пари сусідніх елементів по всьому масиву.

Після першої ітерації найбільший елемент "спливає" до кінця масиву (встановлюється на своє правильне місце).

Повторюємо кроки 2-5 для підмасиву, що складається з усіх елементів, за винятком останнього, оскільки останній елемент залишається найбільшим після попередньої ітерації.

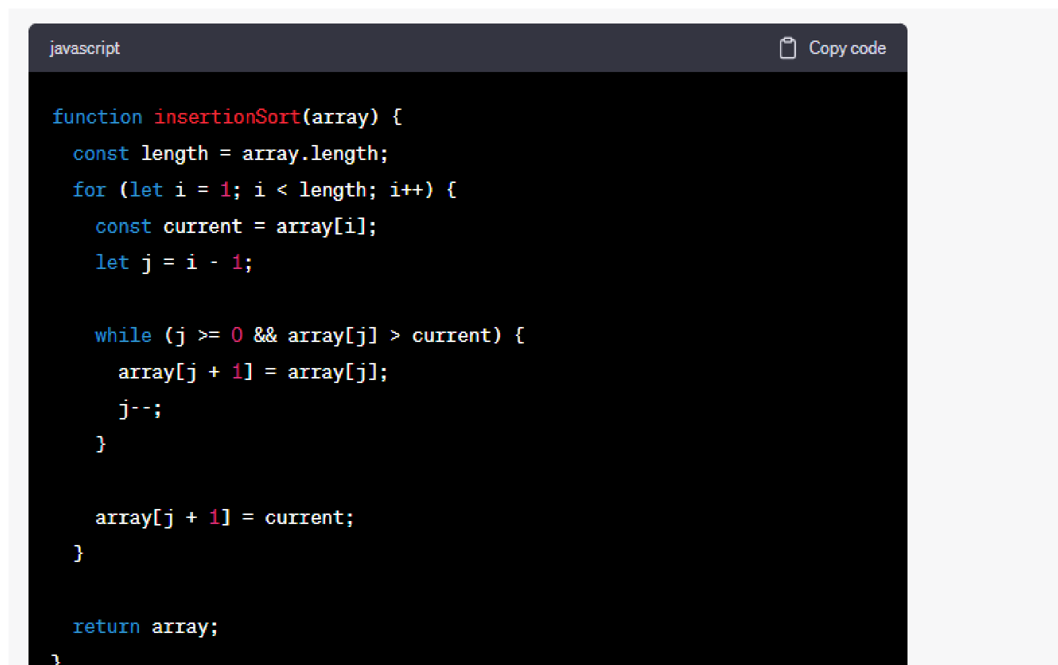
Продовжуємо виконувати ітерації, скорочуючи довжину підмасиву на 1 після кожної ітерації, оскільки найбільший елемент знаходиться на правильному місці.

Алгоритм завершується, коли масив стає повністю відсортованим, тобто після останньої ітерації, коли довжина підмасиву стає рівною 1.

Алгоритм бульбашкового сортування простий і легкий для реалізації, але має квадратичну обчислювальну складність у найгіршому випадку. Тому, для великих масивів, його ефективність може бути обмеженою.

Описаний мною алгоритм сортування називається сортуванням вставкою (Insertion Sort). Він базується на послідовному виборі елементів з вхідного масиву і вставці їх на правильну позицію у відсортованій частині масиву.

Нижче наведений переписаний код алгоритму сортування вставкою:




```javascript
function insertionSort(array) {
  const length = array.length;
  for (let i = 1; i < length; i++) {
    const current = array[i];
    let j = i - 1;

    while (j >= 0 && array[j] > current) {
      array[j + 1] = array[j];
      j--;
    }

    array[j + 1] = current;
  }

  return array;
}
```

Рисунок 1.7 Скриншот коду алгоритму сортування вставкою

У цьому алгоритмі ми починаємо з другого елемента масиву і вибираємо його як поточний елемент. Далі, ми порівнюємо його з елементами у відсортованій частині масиву, починаючи з останнього елемента і рухаючись вліво. Якщо знайдений елемент більший за поточний, ми зсуваємо його вправо на одну позицію. Після закінчення цього циклу ми вставляємо поточний елемент на знайдену позицію.

Алгоритм сортування вставкою є простим і ефективним для невеликих масивів або вже частково відсортованих масивів. Його обчислювальна складність також може бути квадратичною, але він може бути оптимізований для деяких випадків, наприклад, використовуючи бінарний пошук для знаходження правильної позиції відсортованої частини масиву.

Описаний нами алгоритм сортування називається сортуванням змішуванням (Cocktail Sort), і він є модифікацією алгоритму сортування бульбашкою. Суть цього алгоритму полягає в тому, що сортування відбувається у обох напрямках: від початку масиву до його кінця і від кінця до початку. При кожному проході алгоритм порівнює сусідні елементи і, якщо вони знаходяться в неправильному порядку, вони обмінюються місцями.

Нижче наведений переписаний код алгоритму сортування змішуванням:

```
  let left = 0;
  let right = length - 1;

  while (left < right) {
    for (let i = left; i < right; i++) {
      if (array[i] > array[i + 1]) {
        [array[i], array[i + 1]] = [array[i + 1], array[i]];
      }
    }
    right--;

    for (let i = right; i > left; i--) {
      if (array[i] < array[i - 1]) {
        [array[i], array[i - 1]] = [array[i - 1], array[i]];
      }
    }
    left++;
  }

  return array;
}
```

Рисунок 1.8. Скриншот коду алгоритму сортування сортування змішуванням

У цьому алгоритмі ми починаємо з двох країв масиву: лівого (left) і правого (right). На кожному проході ми спочатку проходимо зліва направо і порівнюємо сусідні елементи. Якщо вони знаходяться в неправильному порядку, ми їх обмінюємо. Після цього зменшуємо значення правого краю (right) на 1.

Далі ми проходимо зправа наліво і порівнюємо сусідні елементи. Якщо вони знаходяться в неправильному порядку, ми їх обмінюємо. Після цього збільшуємо значення лівого краю (left) на 1.

Цей процес повторюється до тих пір, поки лівий край не дорівнюватиме правому. На кожному проході найбільші елементи "спливають" до кінця масиву, а найменші елементи "тонуть" до початку масиву.

Алгоритм сортування змішуванням має обчислювальну складність O(n^2) у середньому і найгіршому випадку, але в окремих випадках може досягати ефективності O(n), особливо якщо масив уже відсортований або майже відсортований.

Ви маєте рацію. Сортування змішуванням може бути ефективнішим, ніж сортування бульбашкою, особливо в тих випадках, коли список майже відсортований. Оскільки сортування змішуванням виконується у обох напрямках, це дозволяє найбільшим елементам швидко "спливати" до кінця списку, а найменшим елементам швидко "тонути" до початку списку.

Також запам'ятовування попередніх перестановок може допомогти зменшити кількість проходів по списку. Після кожного проходу, алгоритм може зберігати індекс останньої перестановки і використовувати його як межу для наступного проходу. Це дозволяє уникнути повторних перестановок і скоротити кількість порівнянь.

Загалом, сортування змішуванням може бути швидшим за сортування бульбашкою у багатьох випадках, особливо при майже відсортованих списках. Однак, обидва алгоритми мають квадратичну складність у найгіршому випадку, тому для великих списків рекомендується використовувати більш ефективні алгоритми сортування, такі як швидке сортування (quicksort) чи сортування злиттям (merge sort).

Функція ***cocktailSort*** реалізує алгоритм сортування змішуванням (cocktail sort). Цей алгоритм працює шляхом порівняння сусідніх елементів і обміну їх, якщо вони знаходяться не в правильному порядку. Після кожного проходу в одному напрямку, границі сортування зменшуються, що дозволяє "випливати" найбільшим елементам до кінця списку. Потім алгоритм змінює напрямок і виконує проход в іншому напрямку, "тонучи" найменші елементи до початку списку.

Після цього границя right зменшується, оскільки найбільший елемент вже знаходиться на правильній позиції. Другий цикл for проходить від right до *left* і порівнює сусідні елементи в зворотному порядку. Якщо поточний елемент менший за попередній, вони обмінюються місцями.

Після закінчення циклу while ітерація повторюється знову, але з новими зменшеними границями. Процес продовжується, доки left не дорівнює right, що означає, що весь масив відсортований.

На кінці функція повертає відсортований масив. Код програми наведений нижче:

```
function cocktailSort(array) {
  let left = 0;
  let right = array.length - 1;
  while (left < right) {
    let isSwapped = false;

    for (let i = left; i < right; i++) {
      if (array[i] > array[i + 1]) {
        [array[i], array[i + 1]] = [array[i + 1], array[i]];
        isSwapped = true;
      }
    }
    if (!isSwapped) {
      // Якщо в проході не відбулося обміну, то масив вже відсортований
      break;
    }
    right--;
    for (let i = right; i > left; i--) {
      if (array[i] < array[i - 1]) {
        [array[i], array[i - 1]] = [array[i - 1], array[i]];
        isSwapped = true;
      }
```

```
    }
    if (!isSwapped) {
      // Якщо в проході не відбулося обміну, то масив вже відсортований
      break;
    }
    left++;
  }
  return array;
}
```

Цей код включає додаткову перевірку isSwapped, яка допомагає перевірити, чи відбулися обміни елементів в кожному проході. Якщо жодного обміну не відбулося, це означає, що масив вже відсортований, і процес сортування може бути припинений. Це допомагає покращити ефективність алгоритму в деяких випадках, особливо коли масив вже майже відсортований.

```
function quickSort(array) {
  if (array.length <= 1) {
    return array;
  }
  const pivot = array[Math.floor(Math.random() * array.length)];
  const lesser = [];
  const equal = [];
  const greater = [];

  for (let element of array) {
    if (element < pivot) {
      lesser.push(element);
    } else if (element === pivot) {
      equal.push(element);
    } else {
      greater.push(element);
    }
  }
  return quickSort(lesser).concat(equal, quickSort(greater));
}
```

У цьому коді ми вибираємо опорний елемент (pivot) з масиву (у цьому випадку вибирається випадковий елемент). Потім ми розділяємо масив на три частини: менші елементи, рівні елементи та більші елементи порівняно з опорним елементом. Після цього ми рекурсивно застосовуємо швидке сортування до

менших та більших частин, а потім об'єднуємо відсортовані частини разом з опорними елементами.

Швидке сортування (англ. Quick Sort) - це алгоритм сортування, розроблений Тоні Гоаром. Він не потребує додаткової пам'яті і в середньому виконує O(n log n) операцій. Однак, у найгіршому випадку може здійснювати O(n^2) порівнянь. Швидке сортування використовує прості цикли та операції і працює швидше за інші алгоритми з такою самою асимптотичною складністю. Наприклад, воно зазвичай більш ніж удвічі швидше в порівнянні з сортуванням злиттям.

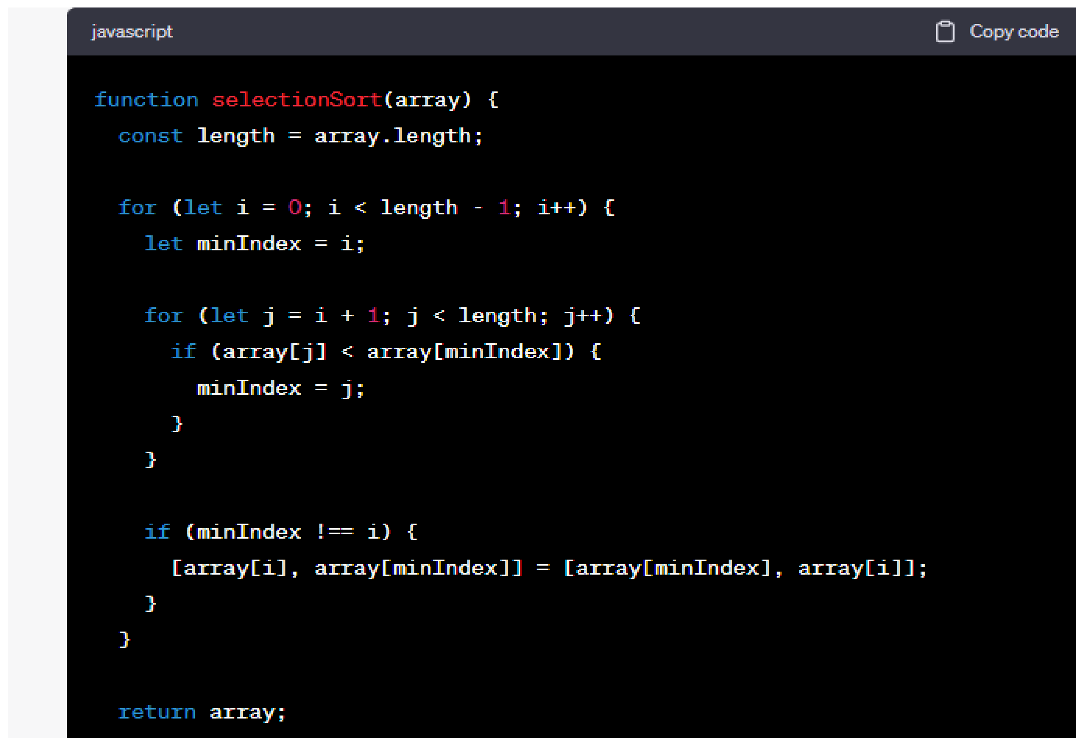


Рисунок 1.9. Скриншот коду алгоритму сортування

У цьому коді ми проходимося по масиву та для кожної позиції вибираємо найменший елемент в підмасиві, починаючи з поточної позиції. Після знаходження найменшого елемента ми обмінюємо його з елементом на поточній позиції. Цей процес повторюється до тих пір, поки масив не буде повністю відсортований.

Пірамідальне сортування (англ. Heapsort) - це алгоритм сортування, який використовує бінарне сортувальне дерево, відоме як піраміда або двійкова купа. Алгоритм складається з двох основних кроків:

Побудова піраміди: Спочатку масив елементів перетворюється на піраміду. Піраміда - це таке дерево, де кожен батьківський вузол має значення, не менше за значення його дочірніх вузлів. Цей крок здійснюється шляхом поетапного перебудовування масиву, починаючи з його середини і рухаючись до початку. Після цього найбільший елемент масиву знаходиться в кореневому вузлі піраміди.

Сортування: На цьому етапі ми послідовно видаляємо найбільший елемент з кореня піраміди (початку масиву) і переставляємо його в кінець масиву. Після цього зменшуємо розмір піраміди на одиницю і відновлюємо її структуру шляхом перебудови. Цей процес повторюється, доки весь масив не буде відсортований в зростаючому порядку.

Пірамідальне сортування виконується за час O(n log n), де n - кількість елементів у масиві. Воно є ефективним алгоритмом для сортування великих наборів даних і має стабільну асимптотичну складність. Приклад коду:

```
function heapify(array, length, i) {
  let largest = i;
  const left = 2 * i + 1;
  const right = 2 * i + 2;

  if (left < length && array[left] > array[largest]) {
    largest = left;
  }

  if (right < length && array[right] > array[largest]) {
    largest = right;
  }

  if (largest !== i) {
    [array[i], array[largest]] = [array[largest], array[i]];
    heapify(array, length, largest);
  }
}

function heapSort(array) {
  const length = array.length;
```

```
 // Побудова піраміди (max heap)
 for (let i = Math.floor(length / 2) - 1; i >= 0; i--) {
   heapify(array, length, i);
 }

 // Витягування максимальних елементів з піраміди
 for (let i = length - 1; i >= 0; i--) {
   [array[0], array[i]] = [array[i], array[0]];
   heapify(array, i, 0);
 }

 return array;
}

// Приклад використання:
const arr = [6, 5, 3, 1, 8, 7, 2, 4];
const sortedArr = heapSort(arr);
console.log(sortedArr);
```

Цей код використовує дві функції - heapify для побудови піраміди і heapSort для виконання сортування пірамідою. Початковий масив arr буде сортований за допомогою пірамідального сортування, і результат буде виведений в консоль.

Плавне сортування (англ. Smoothsort) - це алгоритм сортування, який є варіацією пірамідального сортування та був розроблений Е. Дейкстрою у 1981 році. Як і пірамідальне сортування, плавне сортування має швидкодію O(n log n) у найгіршому випадку. Однак, перевагою плавного сортування є те, що його швидкодія наближається до O(n), якщо вхідні дані частково відсортовані. У той час, як швидкодія пірамідального сортування залишається незмінною і не залежить від стану вхідних даних.

Основна ідея плавного сортування полягає у використанні "контейнерів" або "кубів" для групування елементів за кожним розрядом. Кожен елемент розподіляється відповідно до значення його поточного розряду. Після кожного проходу по всіх розрядах, елементи збираються в правильному порядку, утворюючи відсортований масив.

Ось приклад реалізації плавного сортування на JavaScript:

```
function countingSort(array, exp) {
  const n = array.length;
  const output = new Array(n).fill(0);
  const count = new Array(10).fill(0);
  for (let i = 0; i < n; i++) {
    const index = Math.floor(array[i] / exp) % 10;
    count[index]++;
  }
  for (let i = 1; i < 10; i++) {
    count[i] += count[i - 1];
  }
  for (let i = n - 1; i >= 0; i--) {
    const index = Math.floor(array[i] / exp) % 10;
    output[count[index] - 1] = array[i];
    count[index]--;
  }
  for (let i = 0; i < n; i++) {
    array[i] = output[i];
  }
}
function radixSort(array) {
  const max = Math.max(...array);
  for (let exp = 1; Math.floor(max / exp) > 0; exp *= 10) {
    countingSort(array, exp);
  }
  return array;
}
// Приклад використання:
const arr = [170, 45, 75, 90, 802, 24, 2, 66];
const sortedArr = radixSort(arr);
console.log(sortedArr);
```

У цьому прикладі ми використовуємо функцію countingSort, яка виконує сортування для одного розряду, та функцію radixSort, яка викликає countingSort для кожного розряду від найменш значущого до найбільш значущого. Початковий масив arr буде сортований за допомогою плавного сортування, і результат буде виведений в консоль.

Зазвичай базується на ефективних алгоритмах сортування, таких як швидке сортування чи сортування злиттям. Швидкодія залежить від конкретної реалізації. Вибір алгоритму сортування залежить від розміру масиву, типу даних, ступеня

відсортованості та інших факторів. Кращий алгоритм для конкретної задачі може бути визначений експериментально або враховуючи особливості даних.

1. Сортування обміном (Bubble Sort):

```
python                                                    Copy code

def bubble_sort(arr):
    n = len(arr)

    for i in range(n-1):
        # Здійснюємо i проходів через масив
        for j in range(n-i-1):
            # Порівнюємо сусідні елементи і обмінюємо їх, якщо потрібно
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

    return arr

# Приклад використання:
my_array = [5, 2, 8, 1, 6]
sorted_array = bubble_sort(my_array)
print(sorted_array)
```

Рисунок 1. 10 Скриншот прикладу коду для сортування обміном (Bubble Sort) на мові програмування Python

У цьому коді ми використовуємо два вкладені цикли для проходу через масив і порівняння сусідніх елементів. Якщо поточний елемент більший за наступний, ми їх обмінюємо. Після кожного проходу найбільший невідсортований елемент "спливає" до кінця масиву, тому наступний прохід може обробити на 1 елемент менше.

На виході ми отримаємо відсортований масив [1, 2, 5, 6, 8].

На мови програмування JavaScript:

javascript Copy code

```javascript
function bubbleSort(arr) {
  var len = arr.length;

  for (var i = 0; i < len - 1; i++) {
    // Здійснюємо і проходи через масив
    for (var j = 0; j < len - i - 1; j++) {
      // Порівнюємо сусідні елементи і обмінюємо їх, якщо потрібно
      if (arr[j] > arr[j + 1]) {
        var temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }

  return arr;
}
```

```javascript
// Приклад використання:
var myArray = [5, 2, 8, 1, 6];
var sortedArray = bubbleSort(myArray);
console.log(sortedArray);
```

Рисунок 1. 11. Скриншот виконання прикладу коду

У цьому коді ми використовуємо два вкладені цикли для проходу через масив і порівняння сусідніх елементів. Якщо поточний елемент більший за наступний, ми їх обмінюємо за допомогою тимчасової змінної temp. Після кожного проходу найбільший невідсортований елемент "спливає" до кінця масиву, тому наступний прохід може обробити на 1 елемент менше.

На виході ми отримаємо відсортований масив [1, 2, 5, 6, 8].

Сортування вставлення (Insertion Sort)

Ось приклад коду для сортування вставлення (Insertion Sort) на мові програмування JavaScript:

javascript Copy code

```javascript
function insertionSort(arr) {
  var len = arr.length;

  for (var i = 1; i < len; i++) {
    var current = arr[i];
    var j = i - 1;

    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j];
      j--;
    }

    arr[j + 1] = current;
  }

  return arr;
}

// Приклад використання:
var myArray = [5, 2, 8, 1, 6];
var sortedArray = insertionSort(myArray);
console.log(sortedArray);
```

Рисунок 12 - Скриншот прикладу коду сортування вставленням

У цьому коді ми починаємо з другого елемента структури даних «масиву» і перебираємо його від ліва до права. На кожному кроці ми зберігаємо поточний елемент в змінній current, а потім порівнюємо його з попередніми елементами (від j до 0) у внутрішньому циклі while. Якщо поточний елемент менший за елементи, з якими порівнюємо, ми зсуваємо ці елементи вправо, щоб зробити місце для вставки поточного елемента. На кожній ітерації внутрішнього циклу while ми зменшуємо значення j, щоб перейти до наступного попереднього елемента.

На виході ми отримаємо відсортований масив [1, 2, 5, 6, 8].

3. Сортування змішуванням (Cocktail Sort)

Ось приклад коду для сортування змішанням (Cocktail Sort) на мові програмування JavaScript:

```
    // Прохід справа наліво
    for (var i = right; i > left; i--) {
      if (array[i] < array[i - 1]) {
        var temp = array[i];
        array[i] = array[i - 1];
        array[i - 1] = temp;
      }
    }
    left++;
  }

  return array;
}

// Приклад використання:
var myArray = [5, 2, 8, 1, 6];
var sortedArray = cocktailSort(myArray);
console.log(sortedArray);
```

Рисунок 1. 13 Скриншот прикладу коду сортування змішанням

У цьому коді ми використовуємо два цикли for для проходу по масиву. Перший цикл проходить зліва направо і порівнює кожен елемент з його наступником. Якщо поточний елемент більший за наступний, ми обмінюємо їх місцями. Після першого циклу зменшуємо значення right, оскільки найбільший елемент вже знаходиться в кінці масиву.

Другий цикл проходить справа наліво і порівнює кожен елемент з його попередником. Якщо поточний елемент менший за попередній, ми обмінюємо їх місцями. Після другого циклу збільшуємо значення left, оскільки найменший елемент вже знаходиться в початку масиву.

Ці проходи повторюються до тих пір, поки left не стане більшим або рівним right, що вказує на завершення сортування.

На виході ми отримаємо відсортований масив [1, 2, 5, 6, 8].

Розглянемо сортування пірамідою (Heapsort) для структури даних кучі, то ось приклад коду на JavaScript:

javascript Copy code

```javascript
// Функція для обміну двох елементів масиву
function swap(arr, i, j) {
  var temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}

// Функція для перебудови кучі з коренем у вузлі `root`
function heapify(arr, heapSize, root) {
  var largest = root; // Початково вважаємо, що корінь є найбільшим
  var leftChild = 2 * root + 1;
  var rightChild = 2 * root + 2;

  // Порівняння з лівим дочірнім елементом
  if (leftChild < heapSize && arr[leftChild] > arr[largest]) {
    largest = leftChild;
  }
```

```javascript
  // Порівняння з правим дочірнім елементом
  if (rightChild < heapSize && arr[rightChild] > arr[largest]) {
    largest = rightChild;
  }

  // Якщо найбільший елемент не корінь, то проводимо обмін і перебудовуємо кучу
  if (largest !== root) {
    swap(arr, root, largest);
    heapify(arr, heapSize, largest);
  }
}

// Основна функція пірамідального сортування (Heapsort)
function heapSort(arr) {
  var n = arr.length;

  // Побудова початкової кучі (max-heap)
  for (var i = Math.floor(n / 2) - 1; i >= 0; i--) {
    heapify(arr, n, i);
  }
```

```
  // Послідовне видалення найбільшого елементу з кучі та перебудова кучі
  for (var i = n - 1; i > 0; i--) {
    swap(arr, 0, i);
    heapify(arr, i, 0);
  }

  return arr;
}

// Приклад використання:
var myArray = [5, 2, 8, 1, 6];
var sortedArray = heapSort(myArray);
console.log(sortedArray);
```

Рисунок 1.14. Скриншот прикладу коду структури даних «купа»

У цьому коді ми використовуємо функцію **heapify** для перебудови кучі з коренем у вузлі root. Функція heapify порівнює значення кореня з його лівим і правим дочірніми елементами і обмінює їх.

Сортування змішанням (Cocktail Sort) можна застосувати і до структур даних рядок. Ось приклад коду на JavaScript для сортування рядків за допомогою сортування змішанням:

У цьому коді ми використовуємо два цикли: прямий прохід і зворотній прохід. У кожному проході порівнюємо сусідні елементи і, якщо вони не впорядковані, обмінюємо їх місцями. Процес продовжується до тих пір, поки не буде пройдений весь масив без жодного обміну.

Сортування змішанням для структур даних «граф» на Jawa-script**.**

Сортування змішанням (Cocktail Sort) не є прямим вибором для сортування графів. Графи є складними структурами даних, і їх сортування зазвичай вимагає використання спеціальних алгоритмів, таких як алгоритми топологічного сортування або алгоритми пошуку шляхів.

Основна причина полягає в тому, що графи можуть містити цикли та залежності між вершинами, що робить їх сортування складним процесом.

Сортування змішанням підходить для сортування лінійних структур даних, таких як масиви або списки, де елементи можуть бути порівняні та переміщені.

Якщо потрібно сортувати граф, рекомендую розглянути спеціалізовані алгоритми сортування графів, які враховують їх особливості. Наприклад, алгоритм топологічного сортування може бути використаний для сортування напрямлених ациклічних графів (DAG), де вершини впорядковані залежно від їх залежностей.

1.7.3 Порівняння роботи алгоритмів сортування

Порівняємо роботу алгоритмів сортування: Сортуваннях обміном (сортуваннях Бульбашка (англ. Bubble sort)), Сортування вставлянням (англ. Insertion sort), Сортування змішуванням (англ. Cocktail sort), Сортування вибором (англ. Selection sort), Швидке сортування (англ. Quick Sort), Пірамідальне сортування (англ. Heapsort) і функцію sort() мови програмування Javascript.

1.7.4 Опис платформи порівняння швидкості роботи алгоритмів

Для порівняння швидкості роботи алгоритмів використовуємо комп'ютер з такими параметрами:

Processor: Intel Core i7-10610U, 4 Core (s)

RAM: 32.0 GB

OS: Microsoft Windows 10 Pro

Програму порівняння напишемо в середовищі виконання мови JavaScript Node.js (версія v14.15.1).

Node.js — це JavaScript–оточення побудоване на JavaScript–рушієві Chrome V8.

1.7.5 Опис роботи програми

Програми алгоритмів винесклі в окремі модулі (файли) і підключаємо їх використовуючи синтаксис платформи Node.js require:

```
const {bubbleSort} = require ( './ bubble-sort.js');
const {insertSort} = require ( './ insert-sort');
const {cocktailSort} = require ( './ coctail-sort');
const {selectionSort} = require ( './ selection-sort');
const {quickSort} = require ( './ quick-sort');
const {heapSort} = require ( './ heap-sort');
```

Для об'єктивного порівняння швидкості роботи вище зазначених алгоритмів запускаємо кілька разів на одному масиві, кількість циклів будемо вказувати в константі LOOP_COUNT.

Довжину масиву вказуємо в константі LENGTH.

Формуємо два невідсортованих масива:

1. Випадкові числа від 0 до вказаної довжини LENGTH, числа можуть повторюватися, в цьому випадку використовуємо метод Math.random() мови JavaScript:

```
array = Array.from ({length: LENGTH}, () => Math.floor (Math.random () * LENGTH));
```

2. Випадкові унікальні числа від 0 до LENGTH. Для формування такого масиву використовуємо функцію:

```
function uniqueArray (length) {
  const array = [];
  for (let i = 0; i <length; ++ i) {
    array [i] = i;
  }
  let temp,
    current,
    top = array.length;
  if (top)
    while (--top) {
      current = Math.floor (Math.random () * (top + 1));
      temp = array [current];
```

```
    array [current] = array [top];
    array [top] = temp;
   }
  return array;
}
```

Цикл буде виглядати так:

for (let i = 1; i <= LOOP_COUNT; i ++) {

на кожен непарний **i**, формується масив типу 2 (з унікальними значеннями), на непарний **i** формується масив типу 1 (значення можуть повторюватися):

```
let array = [];
if (i% 2) {
  array = uniqueArray (LENGTH);
} else {
  array = Array.from ({length: LENGTH}, () => Math.floor (Math.random
() * LENGTH));
}
```

Программи алгоритмів виконується LOOP_COUNT разів, час виконання алгоритму треба зберігати, для цього був сформованості об'єкт **algorithms**, ключами якого є назви алгоритмів, а значення це масив, в який будемо зберігати час його виконання після кожної ітерації:

```
const algorithms = {
  bubbleSort: [],
  insertSort: [],
  cocktailSort: [],
  selectionSort: [],
  quickSort: [],
  heapSort: [],
  sort: [],
};
```

У властивість sort будемо записувати час виконання функції мови JavaScript **Array.prototype.sort ().**

Кроки які виконуються для кожного алгоритму:

1. Для виміру часу виконання використовуємо можливості статичного методу getTime() об'єкта new Date ().

start = new Date (). getTime ();

2. Виклик функції сортування, передаємо в нього копію масиву, використовуючи синтаксис десеріалізації

sortArray = insertSort ([... array]);

3. Виводимо в консоль значення першого і останнього елемента масива. Це необов'язково, але якщо з функції ми не будемо нічого повертати, то оптимізуючий компілятор Node.js може проігнорувати роботу функції і наші дані будуть некоректними.

4. Обчислюємо час виконання функції в милисекундах

new Date(). getTime() - start

5. Зберігаємо це значення в масиві, відповідному властивості об'єкта algorithms.

```
start = new Date(). getTime();
sortArray = insertSort ([... array]);

console.log ( 'insertSort:', sortArray [0], sortArray [LAST_INDEX]);
algorithms.insertSort.push(new Date(). getTime() - start);
```

Після завершення всіх циклів обчислюємо середнє значення по кожному алгоритму сортування і виводимо в консоль:

```
for (const name in algorithms) {
  const times = algorithms [name];
  const average = times.reduce ((acc, time) => acc + time);
  console.log (name, average / LOOP_COUNT);
}
```

Код програми

```
const { uniqueArray } = require('./unique-array');
const { bubbleSort } = require('./bubble-sort.js');
```

```
const { insertSort } = require('./insert-sort');
const { cocktailSort } = require('./coctail-sort');
const { selectionSort } = require('./selection-sort');
const { quickSort } = require('./quick-sort');
const { heapSort } = require('./heap-sort');

const LENGTH = 20000;
const LAST_INDEX = LENGTH - 1;
const LOOP_COUNT = 6;

let start = 0;
const algorithms = {
  bubbleSort: [],
  insertSort: [],
  cocktailSort: [],
  selectionSort: [],
  quickSort: [],
  heapSort: [],
  sort: [],
};
let sortArray;
for (let i = 1; i <= LOOP_COUNT; i++) {
  let array = [];
  if (i % 2) {
    array = uniqueArray(LENGTH);
  } else {
    array = Array.from({ length: LENGTH }, () => Math.floor(Math.random() *
  LENGTH));
  }

  start = new Date().getTime();
  sortArray = insertSort([...array]);
  console.log('insertSort: ', sortArray[0], sortArray[LAST_INDEX]);
  algorithms.insertSort.push(new Date().getTime() - start);

  start = new Date().getTime();
  sortArray = bubbleSort([...array]);
  console.log('bubbleSort: ', sortArray[0], sortArray[LAST_INDEX]);
  algorithms.bubbleSort.push(new Date().getTime() - start);
```

```
  start = new Date().getTime();
  sortArray = cocktailSort([...array]);
  console.log('cocktailSort: ', sortArray[0], sortArray[LAST_INDEX]);
  algorithms.cocktailSort.push(new Date().getTime() - start);

  start = new Date().getTime();
  sortArray = selectionSort([...array]);
  console.log('selectionSort: ', sortArray[0], sortArray[LAST_INDEX]);
  algorithms.selectionSort.push(new Date().getTime() - start);

  start = new Date().getTime();
  sortArray = quickSort([...array]);
  console.log('quickSort: ', sortArray[0], sortArray[LAST_INDEX]);
  algorithms.quickSort.push(new Date().getTime() - start);

  start = new Date().getTime();
  sortArray = heapSort([...array]);
  console.log('heapSort: ', sortArray[0], sortArray[LAST_INDEX]);
  algorithms.heapSort.push(new Date().getTime() - start);

  start = new Date().getTime();
  const arrayForSort = [...array];
  sortArray = arrayForSort.sort((a, b) => a - b);
  console.log('sort: ', sortArray[0], sortArray[LAST_INDEX]);
  algorithms.sort.push(new Date().getTime() - start);
}
console.log('Результати роботи алгоритвів:');
for (const name in algorithms) {
  const times = algorithms[name];
  const average = times.reduce((acc, time) => acc + time);
  console.log(name, algorithms[name], ' Середнє значення:',
  Math.floor(average / LOOP_COUNT), 'млс.');

}
```

Запускаємо наші програми порівняння роботи алгоритмів

1. **Експеримент 1**

Довжина масиву 10000 значень.

Кількість циклів 6.

```
Дліна масива:  10000
Кількість ціклів:  6
Результати роботи алгоритвів:
bubbleSort [ 171, 216, 213, 235, 229, 231 ]  Середне значення: 215 млс.
insertSort [ 38, 45, 44, 38, 37, 36 ]  Середне значення: 39 млс.
cocktailSort [ 167, 226, 212, 197, 196, 203 ]  Середне значення: 200 млс.
selectionSort [ 96, 100, 96, 84, 85, 89 ]  Середне значення: 91 млс.
quickSort [ 7, 6, 8, 1, 2, 2 ]  Середне значення: 4 млс.
heapSort [ 12, 13, 16, 11, 2, 2 ]  Середне значення: 9 млс.
sort [ 3, 4, 3, 2, 3, 3 ]  Середне значення: 3 млс.
```

Рисунок 1.17 Скріншот виконання

Кращий результат результат показала функція Javascript sort().

Часова складність алгоритмів Quick Sort, Heapsort и Javascript sort() дорівнює O(n log n) і це добре видно по результатам отриманим работою программи, вони на декілька порядків швидше за інших алгоритмів часова складність яких дорівнює $O(n^2)$. І це ми бачимо по часу виконання. Самий довший алгорим **Сортування обміном** (сортування бульбашкою (англ. Bubble sort)) 215 мілісекунд

2. Експеримент 2

Довжина масиву 100000 значень.

Кількість циклів 6.

Результат роботи:

```
Дліна масива:  100000
Кількість ціклів:  6
Результати роботи алгоритвів:
bubbleSort [ 18467, 24114, 25519, 29294, 28456, 25716 ]  Середне значення: 25261 млс.
insertSort [ 2776, 4421, 3798, 4179, 3726, 3725 ]  Середне значення: 3770 млс.
cocktailSort [ 15674, 23105, 24573, 23782, 21762, 20979 ]  Середне значення: 21645 млс.
selectionSort [ 5906, 9800, 8996, 8655, 8713, 8622 ]  Середне значення: 8448 млс.
quickSort [ 25, 22, 24, 15, 15, 14 ]  Середне значення: 19 млс.
heapSort [ 30, 37, 34, 29, 21, 21 ]  Середне значення: 28 млс.
sort [ 32, 32, 33, 32, 36, 30 ]  Середне значення: 32 млс.
```

Рисунок 1.18 Скріншот виконання

Алгоритм **швидке сортування** (англ. Quick Sort) виявився швидше за всіх алгоритмів, відсортував масив у 100 000 елементів у середньому за 19 мілісекунд та виявився швидше за вбудовану функцію Javascript sort().

Як ми бачимо по виконанню алгоритмів самий довший алгорим **Сортування обміном** (сортування бульбашкою (англ. Bubble sort)) 25,261 секунд

Эксперимент 3

Довжина массива 100000 значень.
Кількість циклів 6.
Перевіряемо тільки функції

Часова складність алгоритмів дорівнює O(n log n) Quick Sort, Heapsort и Javascript sort(). Алгоритми, часова складність яких дорівнює $O(n^2)$ дуже довго обробляють данні.

```
Дліна масива:  1000000
Кількість циклів:  6
Результати роботи алгоритвів:
quickSort [ 137, 147, 145, 140, 140, 141 ]  Середне значення: 141 млс.
heapSort [ 242, 277, 282, 255, 258, 251 ]  Середне значення: 260 млс.
sort [ 323, 339, 328, 331, 335, 337 ]  Середне значення: 332 млс.
```

Рисунок 1.18 Скріншот виконання

Алгоритм **швидке сортування** (англ. Quick Sort) швидше за усі алгоритми, відсортував масив в 1000 000 елемент у середньому за 141 мілісекунду та це у 3 рази швидше за встроену функцію Javascript sort().

Експеримент 4

Довжина масиву 10 000 000 значень.
Кількість циклів 6.

Перевіряємо тільки функції часова, складність алгоритмів дорівнює O(n log n) Quick Sort, Heapsort и Javascript sort(). Алгоритми, часова складність которых равна $O(n^2)$ дуже довго обробляють данні.

```
Дліна масива:  10000000
Кількість циклів:  6
Результати роботи алгоритвів:
quickSort [ 1445, 1594, 1585, 1591, 1611, 1685 ]  Середне значення: 1585 млс.
heapSort [ 3859, 4195, 4422, 4358, 4346, 4316 ]  Середне значення: 4249 млс.
sort [ 3923, 3851, 3847, 3880, 3929, 4078 ]  Середне значення: 3918 млс.
```

Рисунок 1.20 Скріншот виконання

Алгоритм швидке сортування (англ. Quick Sort) швидше за усі, відсортував масив в 10 000 000 елемент в середньому за 1585 мілісекунди, це майже у 3 раза швидше вбудованої функции Javascript sort().

Отже можемо зробити такі висновки по роботі алгоритмів сортування**:**

Можно зробити висновки, що серед цих алгоритмів функцій для цифрових массивів довжиною меньш ніж 100000 краще використовати функцію Javascript sort(). Якщо масив більше, то краще використовати функцію алгоритма швидке сортування (англ. Quick Sort).

2. ЕКОНОМІЧНА ЧАСТИНА

2.1 Резюме

Темою даного дипломного проекту є «Розробка та порівняння роботи алгоритмів у різних структурних даних». У ДП проведено розгляд таких важливих понять в програмуванні як пошук та сортування елементів структур даних. У кожного алгоритма є свої переваги і недоліки. Тому були вибрані алгоритми, які краще всього підходили для рішення поставленної задачі. Існує декілька способів оцінки складності алгоритмів. Програмісти, звичайно, повинні зосереджувати увагу на швидкості алгоритму, але важливі й інші вимоги, наприклад, до розмірів пам'яті, вільного місця на диску або інших ресурсах

Ефективність кожного програмного продукту визначається його якістю та ефективністю процесу розробки. Якість ПП визначається наступними складовими: з точки зору користувача; з позиції використання ресурсів; виконання вимог до програмного забезпечення. Оцінка якості програмного продукту включає визначення трудомісткості і вартості його створення.

2.2. Визначення трудомісткості розробки програмного забезпечення.

Тривалість розробки програмного продукту залежить від його обсягу, трудомісткості розробки, кваліфікації виконавців, а також планових термінів, визначених умовами ринку. Методом структурної аналогії по відповідних каталогах аналогів програмного забезпечення визначаємо обсяг програмних засобів, у тисячах умовних машинних команд програми аналога

У таблиці 2.1 представлені аналоги програмного забезпечення, функції яких, у більшому або меншому ступені, виконує розроблений програмний продукт.

Таблиця 2.1 Каталог аналогів

Найменування ПП	Обсяг функції ПП – V_o, усл. машинних командах.
1. ПП СУБД	2500 – 9800
2. Комплексні системи ведення БД	950 – 7430
3. ПП організації обчислювального процесу	13000 – 10200

Для нашого варіанта виділено сірим кольором.

Вибравши аналог ПП, що містить Vo в умовних машинних командах, трудомісткості визначати на основі табл.2.2

Таблиця.2.2 Аналог ПП

Обсяг ПП, тис.умов.машинних команд	Норма часу, люд/год
1.00	229
2.00	244
3.00	262

На підставі отриманого значення, по довіднику, визначається укрупнена норма часу на розробку аналога програмного забезпечення (коректується поправочним коефіцієнтом враховуючої умови розробки ПП, тобто в умовах комп'ютера, Кк=0,7÷0,8): T^a = 229 х 0,8 = 183,2 (люд/годин).

Трудомісткість програмного продукту визначається по кожному етапу розробки окремо на підставі трудомісткості аналога з урахуванням складності розробки, ступеня новизни і ступеня використання в розробці стандартних модулів на підставі формул:

$$T_{ТЗ} = T^a p \times L_1 \times K_H \qquad (2.1)$$

$$T_{ТП} = T^a p \times L_2 \times K_H \qquad (2.2)$$

$$T_{РП} = T^a p \times L_3 \times K_H \times K_T \qquad (2.3)$$

Для розрахунку необхідні наступні коефіцієнти:

Li – питома вага і-го етапу розробки (див. табл. 2.2.);

Кн – поправочний коефіцієнт, що враховує ступінь новизни (див. табл. 2.3.);

Кт – поправочний коефіцієнт, що враховує ступінь використання в розробці типових програм (див. табл. 2.4.).

Таблиця 2.2. Значення питомих коефіцієнтів трудомісткості стадії в загальній трудомісткості розробки ПП.

Код стадії	**Ступінь новизни**		
	А	**Б**	**В**
ТЗ (L_1)	0,15	0,12	0,12
ТП (L_2)	0,16	0,15	0,11
РП (L_3)	0,55	0,58	0,61

Для нашого варіанта виділено сірим кольором.

Таблиця 2.3. Значення поправочного коефіцієнта, що враховує ступінь новизни

Кодступеня новизни	**Ступінь новизни**	**Значення $К_н$**
А	Принципово нові ПО	1,75 – 1,2
Б	ПО – розвиток визначеного параметричного ряду	1,0 – 0,8
В	ПО маючий аналог	0,7

Для нашого варіанта виділено сірим кольором.

Таблиця 2.4. Значення коефіцієнта ступеня використання в розробці типових програм

Ступінь охоплення реалізованих функцій розроблювального ПО типовими програмами, %	**Значення $К_т$**
60 і вище	0,6
40-60	0,7
20-40	0,8
До 20	0,9

Для нашого варіанта виділено сірим кольором.

Тепер розраховуємо трудомісткість по кожному етапу окремо:

Трудомісткість технічного завдання

$$T^a * L_1 * К_н \quad (2.1)$$

$Т_{тз}$=183,2 *0,12*0,7= 15,39 (люд/годин)

Трудомісткість розробки технічного проекту

$$Т_{тп} = T^a * L_2 * К_н \quad (2.2)$$

$Т_{тп}$ =183.2 *0,11*0,7 = 17,42 (люд/годин

Трудомісткість розробки робочого проекту

$$Т_{рп} = T^a * L_3 * К_н * К_т \quad (2.3)$$

$Т_{рп}$ =183,2 *0,61*0,7*0,7=54,76 (люд/годин)

Для подальших розрахунків визначили кількість папера, витраченого на кожен етап: технічне завдання $N_{тз}$= 2 (стор), розробка ТП $N_{тп}$=18 (стор), розробка

робочого проекту $N_{рп}$=25(стор), пояснювальна записка відповідно $N_{пз}$= 10 (стор)

Розрахунок зведений у таблицю 2.5

Таблиця 2.5. Розрахунок трудомісткості ПП

Найменування етапів	**Розрахунок, годин.**		
1	2	3	4
1.ТЗ	$T_{Ртз}$=15,39	$T_{кк}=0,7*N_{тз}$= 0,7*2=1,4	$T_{нк}=0,15*N_{тз}$=0,15*2=0,30
2.Розробка ТП	$T_{Ртп}$=14,12	$T_{кк}=0,7*N_{тп}$=0,7*18=12,6	$T_{нк}=0,15*N_{тп}$=0,15*18=2,7
3.Розробка РП	$T_{Ррп}$=54,76	$T_{кк}=0,7*N_{рп}$=0,7*25=17,5	$T_{нк}=0,15*N_{рп}$=0,15*25=3.8
4.Розробка ПЗ	$T_{пз}=1,5*$ $*N_{пз}$=1,5*10= 15	$T_{кк}=0,7*N_{тз}$=0,7*10=7	$T_{нк}=0,15*N_{пз}$=0,15*10 =1,5
Усього, в т.ч.:	ΣT=146,1		
- на розробку	$ΣT_{р}$=99,3		
- контроль керівника		$ΣT_{кк}$=38,5	
- нормоконтроль			$ΣT_{нк}$=8,3

2.3 Розрахунок ціни програмного продукту.

У цьому розділі для визначення ціни розраховуємо основну заробітну плату виконавців, матеріальні витрати, вартість машино – години і витрати на розробку ПО. Розрахунок основної заробітної плати виконавців приведений у таблиці 2.6. Відповідно до статті 8 «Закону про Державний бюджет України на 2023» встановлено мінімальну заробітну плату у місячному розмірі з 1 січня 2023 року - 6700 гривень; мінімальну погодинну тарифну ставку – 40.46 грн.

Таблиця 2.6 Розрахунок основної заробітної плати виконавців.

Найменування робіт	**Трудомісткість робіт, години**	**Погодинна тарифна ставка, грн.**	**Розрахунок, грн.**
1.Розробка ПП	99,3	40.46	4017,68
2.Контроль керівника	38,5	100.00	3850
3.Нормоконт-роль	8,3	100.00	830
Усього	-	-	**ΣЗо= 8697,68**

Зробимо розрахунок матеріальних витрат на розробку ПП. Розрахунок зведемо в таблицю 2.7

Таблиця 2.7 Розрахунок матеріальних витрат на розробку ПО

Найменування матеріальних витрат	**Тип, модель**	**Кількість**	**Ціна одиниці, грн.**	**Вартість, грн.**
Папір	Лист А4	55	3.0	165
Разом	-	-	-	$В_{м\ i}$ =
Транспортно– заготівельні витрати (10%)				$Втр_з = 0,1 \times Вм1$ = 16,5
Усього				$В_м = В_{мі} + В_{тр_з}$ = 181,5

На підставі отриманих даних по окремих статтях витрат складена калькуляція планової собівартості в цілому ПП за формою, приведеною в таблиці 2.8.

Таблиця 2.8. Розрахунок статей витрат планової собівартості

Стаття витрат	**Значення, грн.**	**Формула розрахунку**
1. Матеріали	181,5	$В_м$ (див. табл. 2.7)
2. Основна заробітна плата	8697,68	$З_о$ (див. табл. 2.6)
3.Додаткова заробітна плата	1304,65	$Зд = 0,15 \times Зо$ =0,15*8697,68
4.Відрахування до єдиного фонду соціального внеску	2200,51	$Вє.с.в. = 0,22 \times (Зо + Зд)$ =0,22* (8697,68+1304,65)
5. Накладні витрати	2609,30	$Внак. = 0,3 \times Зо$ =0,3* 8697,68
6. Повна собівартість	14993,64	$С_{пов} = В_м + З_о + З_д$ +Вє.с.в. + Внак.= 181,5+8697,68+1304,65+2200,51+2609,30

Розмір прибутку, що включається в ціну, визначаємо по наступній формулі:

$$П=(С_п*Р)/100 \qquad (2.4)$$

де, р – плановий рівень рентабельності (10-15%).

П= (14993,64*10)/100=1499,36 грн

Оптова ціна (кошторисна вартість) визначається по формулі:

$$Ц_о = С_{пов} + П \qquad (2.5)$$

$Ц_о$ =14993,64+ 1499,36 грн =16493,00 грн

Податок на додану вартість визначаємо по наступній формулі:

$$ПДВ = 0.2 * Ц_о = 0,2*16493,00 = 3298,60 \text{ грн} \qquad (2.6)$$

Виходячи з отриманих даних, ціна реалізації розробленого програмного продукту на основі наступної формули, становитиме:

$$Ц_р = Ц_о + ПДВ \qquad (2.7)$$

$Ц_р$ = 16493,00 + 3298,60 = 19791,60 грн

3 ОХОРОНА ПРАЦІ

Охорона праці на виробництві завжди була дуже важлива, отже саме завдяки рекомендаціям з охорони праці, персонал, який працює на підприємстві створює алгоритм виконання робочих завдань з чітким дотриманням рекомендацій. Основне завдання охорони праці – це створення та проведення заходів, спрямованих на захист життя, працездатності та здоров’я людини у процесі трудової діяльності.

При роботі з комп’ютером, як і в багатьох інших галузях, повинні враховуватись нормативи освітлення, температура, відносна вологість і сили вібрації. Але при роботі у приміщенні з комп’ютером найважливішим є дотримання правил пожежної безпеки, це вогнестійкість приміщення, також рівень звукового шуму, характеристики електромагнітних, ультрафіолетових та інфрачервоних полів.

Для аналізу охорони праці у дипломному проєкті досліджується безпека праці розробника веб-сторінок у офісному приміщені.

3.1 Аналіз та безпека умов праці працівника на робочому місці

Під час будь-якого виду роботи за комп’ютером, працівник наражає себе на небезпечні фактори виробничого середовища, а саме: фізичні та психофізіологічні небезпечні й шкідливі виробничі фактори. Це підвищена температура повітря робочої зони, підвищений рівень шуму, знижена вологість повітря. У нервово-психічних перевантаженнях програміст зазнає перенапругу аналізаторів та монотонність праці, інколи, ще й розмовну перенапругу, коли розробнику потрібно складати технічне завдання разом з клієнтом.

Це виробничі фактори, які виникають при роботі у приміщеннях з комп’ютерами,

Окрім цього, комп’ютер випромінює електростатичні та електромагнітні поля у діапазоні від 5 Гц до 2 кГц та від 2 до 400 кГц, тож робота за комп’ютером

включає ще підвищений рівень електромагнітний випромінювання та підвищений рівень статичної електрики.

У офісних приміщеннях не завжди є достатня кількість природного освітлення у такому разі присутня велика кількість штучного освітлення, яке у свою чергу не завжди правильно налаштоване, з цього виникає, що світло може бути недостатньо яскравим або дуже яскравим.

3.2 Розробка заходів з охорони праці

Виробниче освітлення

Штучне освітлення в приміщеннях з робочими місцями, обладнаними ВДТ має здійснюватися системою загального рівномірного освітлення. У виробничих та адміністративно-громадських приміщеннях, у разі переважної роботи з документами, допускається застосування системи комбінованого освітлення (крім системи загального освітлення, додатково встановлюються світильники місцевого освітлення).

Зазначення освітленості на поверхні робочого столу в зоні розміщення документів має становити 300-500лк, а освітленість екрана має не перевищувати 300лк

Мікроклімат

При роботі у приміщеннях з великою кількістю комп'ютерів, приміщення класифікуються як приміщення з підвищеною небезпекою електротравм, температура повітря влітку може становити більше 35 С, що дуже погано впливає на здоров'я людини, тож у таких приміщеннях повітря повинне охолоджуватись та понижена вологість повітря повинна регулюватись спеціальним обладнанням.

Відповідно до норм ДСН 3.3.6.042-99 температура повітря в офісі повинна становити 22-25 С, вологість повітря 40-60%, швидкість руху повітря не більше 0,1 м/с. Якщо ці норми перевищені, робочій день працівника повинен бути скорочений на 10%.

3.3 Організація робочого місця користувача ПК

Конструкція робочого місця користувача ПК й взаємне розташування всіх його елементів (сидіння, органи керування, засобу відображення інформації) відповідають антропометричним, фізіологічним і психологічним вимогам, а також характеру роботи. Конструкція робочих меблів повинна забезпечувати можливість індивідуального регулювання відповідно росту працюючих для підтримки зручної пози. Робочий стіл повинен бути пофарбований матовою фарбою. Дисплей розташований так, що його верхній край перебуває на рівні очей на відстані близько 70 см, що укладається в у припустимі рамки від 60 до 90 см. Частота мерехтіння екрана fмер=100 Гц, що відповідає умові fмер>70 Гц.

Робоче місце розташоване перпендикулярно віконним прорізам, це зроблено з тією метою, щоб виключити пряму й відбиту мерехтливість екрана від вікон і приладів штучного освітлення.

Для офісів та приміщень, обладнаних персональними комп'ютерами або технікою для бізнесу допустимий рівень шуму цілодобово - 50 дБА, а максимальний- 65 дБА

Згідно темі дипломного проекту робоче місце програміста укомплектовано пристроями з електромагнітним випромінюванням.

3.4 Пожежна безпека

Забезпечення пожежної безпеки на об’єкті праці є важливою частиною роботи по створенню безпечних та здорових умов праці.

Прохід до аварійних виходів повинен бути вільний, шириною не менше 1 метру, у разі великої кількості горючих відходів потрібно використовувати відведені сміттєзбірники. Електроприлади повинні використовуватися тільки для їхнього прямого призначення, а у разі пошкодження приладів, слід вимкнути їх живлення та привести до пожежобезпечного стану.

Первинні засоби пожежогасіння застосовуються для боротьби з пожежами

на початковій стадії. До них належать: пожежні кран-комплекти, вогнегасники, пожежний інвентар (резервуари з водою, ящики з піском, пожежні відра, лопати), а також різний переносний пожежний інструмент (кирки, сокири, багри, ломи і т. ін.).

Для гасіння пожеж промисловість випускає різні вогнегасники. Найбільшого поширення набули водопінні, водяні, газові (вуглекислотні) і порошкові. За ефективністю пожежогасіння гасіння, економічністю та іншими показниками більш перспективними вважаються порошкові вогнегасники.

Первинні засоби пожежогасіння розміщують на пожежних щитах, які встановлюють на виробничій території з розрахунку один щит на 5000 м2. Вони фарбуються у червоний колір.

Згідно Правил, на кожному поверсі будинку адміністративного призначення повинно знаходитися не менше двох вогнегасників з масою заряду вогнегасної речовини 5 кг і більше. Експлуатація вогнегасників без призначення відповідального за організацію цієї роботи не допускається. Забороняється палити на підприємстві, крім спеціально виведених для цього місцях, забороняється зберігати легкозаймисті матеріали, такі як папір ближче ніж 1 метр від електрощитів, 0,15 м від приладів центрального водяного опалення та 0,6 м від сповіщувачів автоматичної пожежної сигналізації, також документація повинна зберігатися у спеціально відведених для цього шафах.

Для запобігання розповсюдження пожежі встановлюють протипожежні системи, які складаються з датчиків, звукових сповіщувачів, аварійних кнопок, приймально-контрольної панелі, яка виступає як аналізатор інформації, яку отримали датчики і відправляє ці данні на пульт пожежної охорони. Протипожежна сигналізація призначення для виявлення пожежі на початковому етапі.

Підприємство крім установки пожежної сигналізації на своєму об'єкті, має укласти договір на обслуговування даної системи з фірмою, що має на це ліцензію. В обслуговування входить проведення встановлених нормами регламентних робіт, а так само усунення несправностей в роботі системи. Періодичність

перевірки узгоджується з замовником, але повинна бути не рідше ніж один раз на місяць.

У разі, якщо пожежі не вдалось уникнути, необхідно:

1. терміново повідомити пожежну охорону по телефону 101, вказати при цьому адресу, кількість поверхів, місце виникнення пожежі, наявність людей, своє прізвище;
2. організувати евакуацію людей та матеріальних цінностей;
3. повідомити про виникнення пожежі адміністрацію та чергового (за його наявності);
4. вимкнути, у разі необхідності, струмоприймачі та вентиляцію;
5. розпочати гасіння пожежі наявними первинними засобами пожежогасіння;
6. організувати зустріч підрозділів пожежної охорони й надати їм консультаційну та іншу допомогу в процесі гасіння пожежі.

ВИСНОВКИ

У данному дипломному проекті було проведено розгляд таких важливих понять в програмуванні як пошук та сортування елементів структур даних. У кожного алгоритма є свої переваги і недоліки. Тому були вибрані алгоритми, які краще всього підходили для рішення поставленної задачі. Існує декілька способів оцінки складності алгоритмів. Програмісти, звичайно, повинні зосереджувати увагу на швидкості алгоритму, але важливі й інші вимоги, наприклад, до розмірів пам'яті, вільного місця на диску або інших ресурсах.

Складність алгоритмів пошуку і сортування визначається кількістю порівнянь і кількістю перестановок у програмі. На практиці існує два види оцінки ефективності алгоритму: часова і просторова. Часова ефективність є індикатором швидкості роботи алгоритму, а просторова ефективність показує кількість додаткової оперативної пам'яті, необхідної для роботи алгоритму.

У ДП Було розроблені алгоритми, для рішення однієї і тієї ж задачі, часто дуже сильно відрізняються по ефективності. Ці відмінності можуть бути набагато суттєвіші, ніж ті, що викликані застосуванням неоднакового апаратного та програмного забезпеченням. При вимірі складності алгоритмів і структур даних ми, зазвичай, говоримо про дві речі: кількість операцій, необхідних для завершення роботи (обчислювальна складність), і обсяг ресурсів, зокрема, пам'яті, який необхідний алгоритму (просторова складність).

Алгоритм, який виконується в десять разів швидше, але використовує в десять разів більше місця, може цілком підходити для серверної машини з великим об'ємом пам'яті. Але на вбудованих системах, де кількість пам'яті обмежена, такий алгоритм використовувати не можна.

У дипломному проекті також було досліджено використання різних алгоритмів пошуку і те, що їх використання може бути мінливим навіть не в залежності від швидкості дій, а в залежності від поставленої мети. Так, наприклад, швидкість виконання алгоритму лінійним пошуком в масиві з невеликою кількістю елементів більш прийнятна, ніж застосування бінарного

пошуку, оскільки затрати на реалізацію даного методу не виправдовують його застосування.

Серед розглянутих алгоритмів сортування більшість входить в три групи, які умовно можна назвати «швидке стійке сортування» (сортування злиттям і алгоритм сортування Тіма Петерса), «швидке нестійке сортування» (сортування Шелла, швидке сортування, пірамідальне сортування), а також «повільне стійке сортування» (сортування бульбашковим методом, сортування вставками, сортування вибором, сортування перемішуванням, сортування гнома). Таким чином, існуючі алгоритми сортування масивів значно різняться за рівнем складності, швидкості, стійкості, вимогам до пам'яті та іншими параметрами. Однак практично кожен алгоритм виявляється найбільш зручним в будь-якій конкретної ситуації. Затребуваними є навіть дуже повільні алгоритми, які через свою простоту знаходять застосування в освітніх цілях

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Кормен Т. Аглоритми: побудова і аналіз, 2-ге видання. Пер. с англ. / Т.Кормен, Ч.Лейзерсон, Р.Ривест. – М. : Видавничий дім «Вільямс», 2005. – 1296 с.

2. Кристофидес Н. Теорія графів. Алгоритмичний підхід [Електронний ресурс]. – Режим доступу: http://mirknig.com/knigi/nauka_ucheba/1529-teorija_grafov.html

3. Ахо Альфред В. Структури данних і алгоритми / Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман – М. : Видавничий дім «Вільямс», 2003. – 384 с.

4. Лекції по курсу «Комп'ютерна дискретна математика» [Електронний ресурс]. – Режим доступу: http://itlcvs.wordpress.com/2021/06/06/лекції-по-курсу-компьютерна-дискрет/

5. Бондаренко М. Ф., Білоус Н. В., Руткас А. Г. Комп'ютерна дискретна математика [Електронний ресурс]. – Режим доступу:

6. http://pikkalo.com/2700-kompyuterna-diskretna-matematika.html

7. Трамбле Ж., Соренсон П. Введення в структури данних [Електронний ресурс]. – Режим доступу: http://mirknig.com/knigi/os_bd/1181478197-vvedenie-v-struktury-dannyh.html

8. Руденко В.Д. Курс інформатики / В.Д. Руденко, А.М. Макарчук, М.А. Патланжоглу. – К. : Фенікс, 2000. – С. 230-235.

9. Субботін С.О. Неітеративні, еволюційні та мультиагентні методи синтезу нечіткологічних і нейромережних моделей: Монографія / С.О. Субботін, А.О. Олійник, О.О. Олійник. – Запоріжжя: ЗНТУ, 2009. – 375 с.

10. Tree data structure [Електронний ресурс]. – Режим доступу: http://ideainfo.8m.com/

11. https://uk.wikipedia.org/wiki

ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

РЕЦЕНЗІЯ

на дипломний проект (роботу) здобувача (здобувачки) освіти
відділення комп'ютерних систем

Грічина Владислава Олександровича

(прізвище, ім`я та по батькові)

Спеціальність **121 «Інженерія програмного забезпечення»**

Освітня програма ______ **«Розробка програмного забезпечення»**

Керівник дипломного проекту (роботи) к.т.н., Кунуп Т.В.

(прізвище, ім`я та по батькові)

Тема дипломного проекту (роботи)

Розробка та порівняння роботи алгоритмів у різноманітних структурах даних

Обсяг розрахунково-пояснювальної записки 64 сторінок

Обсяг графічної (презентаційної) частини 19 аркушів (слайдів)

ХАРАКТЕРИСТИКА ДИПЛОМНОГО ПРОЕКТУ (РОБОТИ)

а) заключення про ступінь відповідності виконаного дипломного проекту (роботи) завданню

Представлений на рецензію робота відповідає затверджений темі та виконаний відповідно технічному завданню.

б) характеристика виконання кожного розділу дипломного проекту (роботи)

Пояснювальна записка роботу виконана якісно, з дотриманням усіх норм та стандартів. В дипломному проекті проаналізовано існуючі рішення, проведено аналіз та проведено аналіз алгоритмів у різноманітних структурах даних. Проведено аналіз та побудовано відповідні алгоритмів пошуку та сортування для структр даних.

в) оцінка якості виконання пояснювальної записки та графічної частини дипломного проекту (роботи) Графічна частина складається з 19 слайдів мультимедійної презентації, виконаної у програмному продукті MS PowerPoint, які містять ілюстративні схеми та блок-схеми алгоритмів, передбачені технічним завданням. Пояснювальна записка виконана акуратно та у відповідності до норм. Якість виконання графічної частини проекту та пояснювальної записки висока, розробку виконано у повному обсязі

г) перелік позитивних якостей дипломного проекту (роботи)

Застосування сучасних комп'ютерних програм для розробки алгоритмів пошуку та сортування є актуальним завданням. Також інтерес представляє висновки та рекомендації щодо подальшого їх застосування адаптації під відповідні завдання.

д) основні недоліки дипломного проекту (роботи)

1. У пояснювальній записці не зазначено для якої конкретної схеми та варіанта запропоновані сучасні програмні продукти та платформи.

2. У тексті пояснювальної записки зустрічаються друкарські помилки та неточності.

3. Відсутнє посилання літературу.

Оцінка розрахункової частини добре

Оцінка графічної частини добре

Загальна оцінка добре

Прізвище, ім'я, по батькові рецензента Царьов Роман Юрійович

Місце роботи і посада рецензента Державний університет інтелектуальних технологій і зв'язку, старший викладач кафедри комп'ютерної інженерії та інформаційних систем

Підпис:

«16» 06 2023 р.

ДОЗВІЛ
НА РОЗМІЩЕННЯ
ВИПУСКНГО ДИПЛОМНОГО ПРОЕКТА
В ЕЛЕКТРОННОМУ РЕПОЗИТАРІЇ ВСП «ОТФК ОНТУ»

Ми, що нижче підписалися,

Грічин Владислав Олександрович,
здобувач освіти гр. 4РП-06, та

Кунуп Тетяна Василівна,
керівник дипломного проекту,

не заперечуємо щодо розміщення електронного варіанту пояснювальної записки до випускного дипломного проекту молодшого спеціаліста на тему:

«Розробка та порівняння роботи алгоритмів у різних структурах даних»
(автор роботи – Грічин В.О., керівник роботи – Кунуп Т.В.)

виконаного у ВСП «Одеський технічний фаховий коледж Одеського національного технологічного університету» в 2023 році, у повному обсязі в електронному репозитарії ВСП «ОТФК ОНТУ» для вільного доступу через мережу Інтернет.

Несемо відповідальність за ідентичність електронного та друкованого варіантів випускної кваліфікаційної роботи, і даємо згоду на обробку персональних даних.

Виконавець ______________ / Грічин В.О. /

Керівник ______________ / Кунуп Т.В. /

«12» 06 2023 р.

UNICHECK

Ім'я користувача:
Наталія Вікторівна Копусь

ID перевірки:
1015218614

Дата перевірки:
24.05.2023 07:22:56 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
24.05.2023 07:25:28 EEST

ID користувача:
100011688

Назва документа: 4РП-06 Грічин В.О.

Кількість сторінок: 49 **Кількість слів:** 8804 **Кількість символів:** 66252 **Розмір файлу:** 543.61 KB **ID файлу:** 1014896151

9.43% Схожість

Найбільша схожість: 2.52% з Інтернет-джерелом (http://masters.donntu.org/2012/iii/lunov/diss/indexu.htm)

9.43% Джерела з Інтернету | 426 .... Сторінка 51

Не знайдено джерел з Бібліотеки

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0% Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | 19

ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

ВІДГУК

керівника на дипломний проект здобувача (здобувачки) освіти

відділення комп'ютерних систем

Грічина Владислава Олександровича

(прізвище, ім`я та по батькові)

Спеціальність: 121 «Інженерія програмного забезпечення»"

Освітня програма: «Розробка програмного забезпечення»

Тема дипломного проекту: Розробка та порівняння роботи алгоритмів у різноманітних структурах даних

ХАРАКТЕРИСТИКА ДИПЛОМНОГО ПРОЕКТУ

а) обсяг і якість виконання проекту (графічного матеріалу і розрахунково-пояснювальної записки) Дипломний проект виконано відповідно технічному завданню. Пояснювальна записка містить 79 сторінок. У пояснювальній записці виконано опис етапів розробки алгоритмів пошуку та сортування в структурах даних. Графічна частина складається з 19 слайдів мультимедійної презентації, які також містять креслення, передбачені технічним завданням. Якість виконання пояснювальної записки та графічної частини добра, розробку виконано в повному обсязі.

б) самостійність роботи над проектом:

Протягом всього строку дипломного проектування та переддипломної практики здобувач освіти Грічін В.О. поступово та послідовно виконував всі етапи розробки. Всі роботи здобувач освіти виконував самостійно, з оглядом на рекомендації керівника

в) теоретична підготовка випускника (випускниці): Здобувач освіти Грічин В.О. під час роботи над диплом проектом вивчив достатню кількість літературних джерел та матеріалів за даною тематикою.

Вважаю, що теоретична підготовка дипломника добра і він готовий до захисту дипломного проекту

г) вміння розв'язувати виробничі та конструкторські питання

Під час дипломного проектування здобувач освіти Грічин В.О мав змогу самостійно приймати окремі рішення з реалізації алгоритмів пошуку та алгоритмів сортування у різноманітних структурах даних, працювати над поставленим завданням, розробити програмний алгоритми за допомогою мови програмування JawaScript.

Оцінка розрахункової частини Відмінно

Оцінка графічної частини Відмінно

Загальна оцінка Відмінно

Прізвище, ім'я, по батькові керівника дипломного проекту

Кунуп Тетяна Вісилівна

Місце роботи і посада керівника дипломного проекту

ВСП "Одеський технічний фаховий коледж ОНТУ", викладач спецдисциплін комісії комп'ютерних технологій та програмної інженерії, голова циклової комісії КТ та ПІ

Підпис

«09» 06 2023 р.