МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

АНДРІЙЧЕНКО КАТЕРИНА АНДРІЇВНА

Допускається до захисту:
в. о. завідувача кафедри
Прикладної математики,
Трофименко О. Д.
«____» _____ 20__ р.

ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ДЕЯКИХ АЛГОРИТМІВ ПОШУКУ З ПОВЕРНЕННЯМ

Спеціальність 113 Прикладна математика

Кваліфікаційна (бакалаврська) робота

Керівник:
Вєтров О. С., старший викладач
кафедри Прикладної математики

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

Голова ЕК: _____
(підпис)

Вінниця – 2021

АНОТАЦІЯ

Андрійченко К. А. Дослідження ефективності деяких алгоритмів пошуку з поверненням. Спеціальність 113 «Прикладна математика». Донецький національний університет імені Василя Стуса, Вінниця, 2021.

У кваліфікаційній (бакалаврській) роботі досліджено ефективність деякого алгоритму пошуку з поверненням. Показано, які типи алгоритмів та методи їх вирішення існують. Реалізовано алгоритм пошуку з поверненням на прикладі задачі побудови греко-латинського квадрату. Встановлено, як складність алгоритму впливає на час його виконання.

Ключові слова: ефективність, алгоритм, пошук з поверненням, алгоритмізація, програмна реалізація, аналіз, греко-латинський квадрат.
59 с., 18 рисунків, 1 таблиця, 15 джерел

ABSTRACT

Andriichenko K. Study efficiency of some backtracking algorithms. Specialty 113 “Applied mathematics”. Vasyl’ Stus Donetsk National University, Vinnytsia, 2021.

In the qualification (bachelor's) work the efficiency of some backtracking algorithm is investigated. It is shown what types of algorithms and methods of their solution exist. A Graeco-Latin square as an example of the realization of the backtracking algorithm is implemented. It is established how the complexity of the algorithm affects on the execution time.

Keywords: efficiency, algorithm, backtracking, algorithmization, software implementation, analysis, Graeco-Latin square.

ЗМІСТ

ВСТУП

Завдяки розвитку науки інформатики та проникненню її в різні галузі людського життя слово "алгоритм" стало одним з найбільш вживаних понять у розмові для широкого кола фахівців. Більш того, з переходом до інформаційного суспільства алгоритми стають одним з найважливіших чинників цивілізації.

Відомо, що математична теорія алгоритмів склалася зовсім не в зв'язку з активним розвитком інформатики та обчислювальної техніки, а виникла ще в надрах математичної логіки для вирішення її ж власних проблем. Вона, перш за все, дуже вплинула на світогляд математиків та на їх науку.

Проте є безсумнівним взаємовплив теоретичних областей, пов'язаних з обчислювальною технікою та теорією алгоритмів. Теорія алгоритмів вплинула також і на теоретичне програмування. Зокрема, велику роль в теоретичному програмуванні грають моделі обчислювальних машин, які, по суті, є обмеженнями тих представницьких обчислювальних моделей, які були створені раніше в теорії алгоритмів. Трактування програм, як об'єктів обчислення, оператори, використовувані для складання структурованих програм (послідовне виконання, розгалуження, повторення) прийшли в програмування з теорії алгоритмів. Зворотний вплив виразився, наприклад, в тому, що виникла потреба в створенні і розвитку теорії обчислювальної складності алгоритмів. Таким чином, можна сказати, що теорія алгоритмів застосовується не тільки в інформатиці, а й в інших областях знань.

Актуальність обраної теми полягає в тому, що навіть на просторах мережі Інтернет дуже важко або навіть неможливо знайти гарно складені, працюючі алгоритми, котрі б давали зрозумілий та швидкий розв’язок. Особливо це стосується алгоритму з поверненням, котрий може бути застосований до таких потрібних задач, як, наприклад, побудова греко-латинського квадрату. Потрібність цієї задачі пояснюється просто:

застосування таких квадратів можливе для складання розкладів, вивчення впливу різних добрив на різні рослини на різних грунтах, планування експериментів у медицині та біології, розподілу ресурсів, черговості ініціювання подій, видачі завдань студентам тощо. А також і в суміжних областях математики, наприклад: в теорії груп та напівгруп, криптографії, комбінаториці, логіці, статистиці, теорії кодів тощо. Звичайно, що алгоритм з поверненням може бути застосований і для ряду інших задач, проте, чимала кількість таких розі'язків вже існує у відкритому доступі.

Мета дослідження: дослідити історію виникнення та застосування алгоритмів та ефективність алгоритму з поверненням для деяких задач; створити програму для складання греко-латинського квадрату; проаналізувати схожі алгоритми та їх застосування у розв'язку задач.

Об'єкт дослідження: ефективний алгоритм пошуку з поверненням.

Предмет дослідження:

- Написання алгоритму
- Дослідження та аналіз ефективності роботи алгоритму на прикладі задачі побудови греко-латинського квадрату.

Для досягнення поставленої мети необхідно виконати наступні **завдання**:

- Дослідити історію створення алгоритмів та їх теорію, необхідну для виконання роботи
- Проаналізувати існуючі алгоритмічні розв'язки до популярних задач з використанням методу перебору з поверненням
- Створити алгоритм з поверненням для обраної задачі
- Дослідити ефективність алгоритму з поверненням на прикладі обраної задачі

Кваліфікаційна (бакалаврська) робота складається із вступу, трьох розділів, висновків до кожного розділу, загального висновку, списку використаної літератури (15 літературних джерел).

Загальний обсяг роботи – 61 стр.

РОЗДІЛ 1
АЛГОРИТМИ ТА ЇХ ХАРАКТЕРИСТИКИ

1.1. Історія виникнення поняття «алгоритм»

Поняття алгоритму є одним з фундаментальних понять обчислювальної математики. Проте, виникло воно задовго до появи обчислювальних машин завдяки пошукам загальних методів розв'язку однотипних задач.

Так першим алгоритмом прийнято вважати правило обчислення найбільшого спільного дільника двох натуральних чисел створене ще у III сторіччі до нашої ери грецьким математиком Евклідом. Не завжди легка задача, але суттєва в багатьох ситуаціях. Суть алгоритму полягала у тому, щоб віднімати від більшого числа менше, підставляючи отриманий результат на місце більшого числа до того часу, поки числа не будуть однаковими. Отримане число і буде найбільшим спільним дільником як різниці, так і будь-якого з чисел. Цей набір дій і є алгоритмом, тому що виконуючи ці кроки людина отримає найбільший спільний дільник для певної пари чисел. Слово «алгоритм» не пішло від грецьких слів arithmos (ἀριθμός), що означає «число» та algos (ἄλγος), що означає «біль», як могли б подумати так звані «алгоритмофоби». Передбачається, що це слово пішло від імені перського математика Мухаммада ібн Мусса Аль-Хорезмі, що жив у IX сторіччі. З усіх його робіт до нашого часу дійшло, на жаль, лише дві – алгебраїчна та арифметична. Остання ж тривалий час вважалась втраченою, але у 1857 році у Кембриджській бібліотеці було знайдено переклад книги латиною. До речі, розповсюдженою була версія про грецьке походження книги. Так от англо-норманський рукопис XIII століття згадується: «Алгоризм був вигаданий у Греції. Це частина арифметики. Вигаданий він був завдяки майстру, котрого звали Алгоризм» [1]. Вона представляє собою опис чотирьох основних правил арифметичних дій, практично ж тих, що використовуються і сьогодні [2]. У своїх роботах Аль-Хорезмі намагався описувати правила таким чином, щоб

вони були зрозумілі усім грамотним людям. В той час, коли не існувало математичної символіки такої, як дужки, знаки операцій, буквенні позначення тощо – зробити це було складно. Але йому все ж вдалось розробити власний стиль чіткого словесного написання, який не залишав шансів читачеві щось не зрозуміти чи пропустити якісь дії.

За іншим джерелом [3], Аль-Хорезмі описав позиційну десяткову систему запису та дій над числами та знак «порожнє коло» - тобто нуль, для відображення відсутності позиції в записі числа. І те, і інше було винайдено в Індії декількома століттями раніше. Також методи, описані у його останньому трактаті, що використовували прописані фігури або лічильні камінчики, стали відомими пізніше як «algorism» або «augrym». Наразі, ці фігури всім відомі як цифри.

В першій половині XII століття книга Аль-Хорезмі в латинському перекладі (та сама, екземпляр якої згадувався раніше) потрапила до Європи - «Algoritmi de numero Indorum» або «Індійське мистецтво обрахунку, твір Аль-Хорезмі». Перші декілька стрічок тексту латиною звучали наступним чином: «Сказав Алгорізмі: «Воздасться ж хвала Богу, нашому вождю та захиснику»». Поступово люди почали забувати, що Алгоризмі це взагалі-то ім’я автора правил, тож почали просто називати їх алгорізмами. Таким чином, ім’я Аль-Хорезмі перейшло у Алгорізмі. Сам термін вживався для позначення чотирьох арифметичних операцій. Таким означення, свого часу, і ввійшло в деякі європейські мови.

Між іншим, обчислення за допомогою запису ваги розряду та роботи Аль-Хорезмі були вже добре знайомі завдяки деяким європейським вченим. Гінді-арабська система числення була популяризована в Європі середньовічним італійським математиком та торговцем Леонардо з Пізи, відомим як Фібоначчі. Дякуючи його книзі “Liber Abaci”, виданій у 1202 році, прописні цифри почали замінювати таблиці підрахунку та рахунок на пальцях, як переважний спосіб обрахунку у Європі XII століття. Причиною було навіть не те, що їх простіше запам’ятати, а те, що вони дали можливість вести журнали аудиту. Загалом ж,

цифри набули широкого розповсюдження у західній Європі з появою рухомого типу – технології друку, що використовує рухомі компоненти для відтворення елементів документа на папері [4]. Але справжня популярність з'явилась завдяки появі дешевого паперу у XIX столітті.

Врешті-решт, є підстави вважати, що слово algorism перетворилось вже у сучасне algorithm завдяки народній етимології - з грецького arithmos. До речі, деякі середньовічні джерела зазначають, що грецький префікс «algo» означає «мистецтво» або «вступ». Інші ж зазначають, що алгоритми були винайдені чи то грецьким філософом, чи то королем Індії або Іспанії, якого звали чи то Algus , чи то Algor, чи то Argus. Деякі джерела, включаючи Данте Аліг'єрі, навіть ідентифікували винахідника як грецького міфологічного кораблебудівника-аргонавта. Насправді не зрозуміло, чи хоч якась з цих версій була історично точною.

Таким чином, до останнього часу слово алгоритм посилалося виключно на механічні прийоми арифметики, що використовували арабське числення. З плином часу, значення слова розширювалось. Вчені застосовували його не тільки для обчислень, але і для інших математичних процедур. Близько 1360 року французький філософ Ніколай Орем написав трактат «Algorismus proportionum» - «Обчислення пропорцій», в якому було вперше використано степені з дробовими показниками. Вчений майже в лоб підійшов до ідеї логарифмів. Вже у 1684 році Г.В. Лейбніц у творі «Nova Methodvs pro maximis et minimis, itemque tangentibus…» вперше використав повноформатне слово «алгоритм» в ще більш широкому сенсі: як систематичний спосіб вирішення проблем диференціального обчислення. Користувався словом також і Л.Ейлер. Одна з його робіт має назву «Використання нового алгоритму для вирішення проблеми Пелля». Тобто стає помітним те, що розуміння Ейлером алгоритму як синоніму до способу рішення задачі стає близьким до сучасного розуміння.

Багато алгоритмів було створено та записано на папір. Але перший алгоритм, що був написаний для виконання на машині, належав Аді Лавлейс – 1843 рік. Свого часу вона познайомилась з Чарльзом Беббеджом, котрого часто

називають «батьком комп’ютерів» за його великий вклад у винахідництво останніх. Вона зацікавилась однією з його ідей – аналітичним двигуном [5]. Це був механічний комп’ютер – машина, що автоматизувала обрахунки, той самий аналітичний двигун, для якого вона написала алгоритм. Робота Лавлейс полягала у написанні формули, яка показувала необхідні налаштування двигуна для обчислення певної складності послідовності чисел, званих числами Бернуллі. Наразі ця формула визнана першим комп’ютерним алгоритмом в історії.

А. Лавлейс не обмежилась лише математичними розрахунками. У свій час вона була справжнім провидцем. Поки її сучасники бачили перші механічні комп’ютери лише як розбивачів чисел, вона запитувала: «А що ж стоїть за обрахунками?». Їй цікаво було розширити потенціал механічних комп’ютерів як інструментів для групових задач. Вона все ж сподівалася побачити комп’ютери, які могли б надати людям набагато більше, ніж просто автоматизовані розрахунки. На жаль, побудова аналітичного двигуна не була завершена до смерті Лавлейс, і тому вона ніколи не змогла побачити свій алгоритм у дії. Але він не побудований і на сьогоднішній день. Існують й інші часткові реалізації роботи Чарльза Беббеджа, але, на жаль, ми не можемо запустити алгоритм Ади Байрон на реальному аналітичному механізмі.

XIX століття стало епохою "алгоритмів, вбудованих у машини". Їх було багато, починала процвітати автоматизація людської діяльності. Якщо потрібен був незвичайний візерунок на шматку тканини, Жозеф Марі Жаккар, французький ткач і купець, мав рішення: жакардовий ткацький верстат. Це дозволило виробникам тканин виготовляти вишукані візерунки, використовуючи ряд перфорованих карток, які вказували ткацькому верстату, як саме потрібно ткати [5]. Подібним чином ранні телефонні станції використовували складні механічні пристрої для підключення телефонних дзвінків. Вони автоматично виконували покрокові інструкції, щоб зрештою змусити двох людей поговорити між собою. Ці машини, будь то ткацькі верстати чи станції, у свій час були новаторськими і досі вражають донині.

Однак усі ці пристрої все ще були суто механічними. Їх виготовляли з важелів, перемикачів, валів. Але вони все ж були дуже далекі від того, що ми сьогодні називаємо комп’ютерами.

Лише в 30-х роках ми побачили перші згадки про алгоритми в електронних (немеханічних) комп’ютерах. Алан Тьюрінг, англійський математик, був одним з перших вчених, які формально фіксували, як машини могли обчислювати. Метою Тьюрінга було зафіксувати загальний процес, а не той, який є специфічним для певного завдання, наприклад, для ідентифікації простих чисел або обчислення найбільшого спільного дільника. Науковець у 1936 році описав схему гіпотетичної машини та назвав алгоритмом те, що вміє робити така машину. Якщо щось не могло бути зроблено машиною Тюрінга, то це вже не був алгоритм. Таким чином, Тьюрінг формалізував правила виконання дій при допомозі опису роботи кількох конструкцій.

Розвиток так званої машини Тюрінга призвів до появи комп'ютерів загального призначення. На відміну від попередніх машин, нові комп'ютери мали б виконувати довільні набори інструкцій. Тобто, їх можна було б використовувати в цілях, навіть не передбачених їх творцями. Якщо іншими словами, то робота Тьюрінга призвела до розвитку комп’ютерів, на яких ми можемо встановлювати та запускати додатки.

Через десятиліття, алгоритми почали втрачати свою простоту. Настільки втрачати, що інколи люди не можуть пояснити, як вони працюють. Що ж стосовно людей, далеких від науки, то до початку 40-х років XX століття вони могли почути слово «алгоритм» лише під час навчання у школі, у купі з «алгоритмом Евкліда». Сам термін сприймався як «спеціальний» - це підтверджувала відсутність відповідних статей у невеликих енциклопедичних виданнях. Вже з середини XX століття стали розроблятися різні способи опису алгоритмів, наприклад, за допомогою спеціальних мов, які називаються алгоритмічними, і графових схем - графічного зображення алгоритму. Розвиток електронної обчислювальної техніки і методів програмування сприяло тому, що розробка алгоритмів стала необхідним етапом автоматизації. Багато хто в той

час хотів вважати, що комп’ютерні алгоритми – це чорні ящики. Не потрібно було розуміти, як саме вони працювали, адже все що турбувало – це входи та виходи, що потрапило до чорного ящика, та що отримали врешті-решт. Це спрощення роботи алгоритму було вибором, котрого ми не маємо наразі, адже люди не можуть пояснити, як саме алгоритми досягають певних результатів, то ж змушені думати про них, як про чорні, магічні ящики.

Яскравим підтвердженням цих слів є група алгоритмів штучного інтелекту. Ми можемо пояснити їх принципи: можна сказати, що алгоритм використовує штучну нейронну мережу. Ми можемо пояснити, як була створена мережа чи як вхідні дані результують у вихідні. Однак, не можемо пояснити, чому саме цей результат був результатом роботи алгоритму, за винятком чисто механічного пояснення. Ерік Л. Луміс, тривалість ув'язнення якого залежала від алгоритму, намагався зрозуміти, чому алгоритм «COMPAS» оцінював його як злочинця високого рівня ризику, але це було просто неможливо. Складність алгоритмів часто надзвичайна. І це лише початок. Адже ми живемо у світі, де алгоритми є скрізь - не лише на папері чи в наших думках, але й у керуванні машинами, комп’ютерами та роботами.

1.2. Основні характеристики алгоритмів

1.2.1. Властивості алгоритмів

Властивості алгоритму — це вимоги, яким повинен задовольняти алгоритм. Властивостями будуть наступні пункти:

- Послідовність (однозначність) - в кожен момент часу наступний крок роботи однозначно визначається системами. В цьому випадку видається один і той же результат для одних і тих самих вихідних даних.

- Зрозумілість - алгоритм для виконавця повинен включати тільки ті команди, які йому зрозумілі, тобто входять в його систему команд.
- Скінченність - алгоритм складається зі скінченної кількості кроків (дій), кожна з яких виконується за скінченний проміжок часу. При коректних заданих вихідних даних алгоритм повинен завершуватися видачою результату.
- Масовість - алгоритм повинен бути застосований до різних наборів вихідних даних, а також повинен передбачати можливість розв'язання групи типових задач та зміни вхідних даних в деяких визначених межах.
- Результативність - алгоритм повинен обов'язково видавати результат.
- Дискретність – можливість розбиття алгоритму на окремі елементарні дії , що можна виконати. Тільки закінчивши виконання однієї команди, виконавець переходить до виконання іншої [6].

Таким чином, маємо, що алгоритм — це однозначно визначена послідовність окремих зрозумілих команд, виконання яких за скінчене число кроків приводить до розв'язання поставленої задачі.

Основними характеристиками виконавця алгоритму, тобто живого чи не живого об’єкту, що буде виконувати покроково усі команди, можна назвати наступні пункти:

- Середовище — умови, у яких діятиме виконавець.
- Елементарні дії — найпростіші дії, які виконуватиме виконавець.
- Система команд — сукупність допустимих команд для виконавця.
- Допустимі команди — команди, які зрозумілі виконавцю і можуть бути ним виконані.
- Недопустимі команди — команди, які не можуть бути виконані виконавцем з тих чи інших причин [7].

При розробці алгоритму необхідно керуватись певними директивами, тобто вказівками, за допомогою яких визначається напрямок дій. Отже, наступні твердження є обов’язковими при написанні алгоритму:

- Алгоритм повинен починатись із «СТАРТ» (або «ПОЧАТИ») та закінчуватись на моменті «СТОП» (або «КІНЕЦЬ»).
- Щоб прийняти дані від користувача, потрібно використовувати оператори INPUT, READ, GET або OPTAIN.
- Для відображення результату або будь-якого повідомлення зазвичай потрібно використовувати оператори PRINT, DISPLAY або WRITE.
- Як правило, при описі математичних виразів використовується COMPUTE або CALCULATE, та за ситуацією можуть бути використані відповідні оператори.

Також для складання алгоритму та написання коду потрібні змінні. Змінні – це певні символічні імена, що даються даним, що використовуються у коді, де значення збережених даних може змінюватися під час виконання, власне, програми. Фактично, змінною можна назвати іменовану область пам'яті, що використовується для зберігання необхідних даних. Чимало програмування вимагають, щоб константи та змінні були оголошені ще на початку програми, де одразу ж визначається тип змінної (тобто, це текст, ціле число, логічна змінна тощо) та встановлюється значення будь-яких констант. Ім'я, що може використовуватись для позначення змінної чи константи, може бути й однією літерою, але багато мов дозволяють використовувати повноцінні, розгорнуті імена, які роблять програму набагато легшою для розуміння. Майже у всіх мовах імена змінних не можуть починатися з цифри (0–9) або спеціальних символів, та не можуть містити пробілів. Наприклад, щоб весь час не вводити значення числа Пі, можна завести на початку змінну з навою PI, це буде константа, та надати їй відповідне значення:

```
CONSTANT PI = 3.142

INPUT circle_radius

circle_area = PI * circle_radius * circle_radius
```

Тут константа PI використовується для обчислення площі кола. Змінна circle_radius використовується для зберігання радіуса кола, а змінна circle_area - для зберігання результату обчислення площі.

1.2.2. Форми представлення алгоритмів

Існує декілька форм подання алгоритмів. Коротко розглянемо їх.

- Словесна форма – тобто така, що може бути описана усно, за допомогою зрозумілої людям мови (наприклад, кулінарний рецепт).

Ця форма не надто поширена у науковій літературі через багатослівність та відсутність наглядності. Наприклад, розглянемо простий алгоритм порівняння двох простих чисел A та B, та збереження найбільшого з них у третій змінній C:

1. Отримати числа A та B
2. Порівняти їх
3. Якщо A більше за B, то запишемо значення A в C
4. Якщо не більше, то запишемо значення B в C
5. Вивід змінної C

Можна помітити, що такий запис має ряд недоліків: занадто багато слів, відсутня строга формалізація, деякі дії є неоднозначними. Через ці причини словесна форма не набула широкого поширення у якості запису алгоритму.

- Формульно-словесна форма – коли алгоритм подається у наукових чи навчальних цілях, з використанням різноманітних формул (фізика, математика, хімія тощо)
- Графічна форма – подання алгоритму у вигляді блок-схеми (структурної схеми).

Такий спосіб полегшує перехід від словесної форми до написання коду програми. Він є чи не найзручнішою формою запису алгоритму, тому зміг набути широкого поширення у науковій та навчальній літературі. Власне ж блок-схема алгоритму – це графічне зображення алгоритму у вигляді схеми, де

блоки (кроки алгоритму з відповідним описом) поєднані стрілками. Графічне зображення алгоритму використовується перед програмуванням завдання через його наочність, тому що зорове сприйняття зазвичай полегшує процес написання програми, її коригування при допущених помилках, осмислення процесу обробки інформації. Інколи можна побачити таке означення: «Ззовні алгоритм являє собою схему - набір прямокутників та інших символів, всередині яких записується, що обчислюється, що вводиться в машину і що видається на друк і інші засоби відображення інформації» [8]. Тут форма подання алгоритму вже змішується із самим алгоритмом. Для створення та редагування блок-схем можна користуватися графічним редактором – «Microsoft Office Visio».

- Програмна форма – подання алгоритму вже на мові програмування для подальшої його обробки машиною.
- Псевдокод – подання алгоритму змішаною формою – програмною та словесною.

Зазвичай у цьому способі використовуються скорочені назви команд, змінних, загальні математичні визначення тощо. Строгих синтаксичних правил тут немає, як і єдино точного визначення, що ж таке псевдокод, хоча існують певні службові слова, які виділяються як при написанні вручну, так і на комп'ютері (наприклад, *якщо…то…інакше*)

- Інші – специфічні способи запису алгоритмів. Наприклад, запис нот на нотному стані.

Не може здивувати те, що алгоритми теж мають переваги та недоліки. Серед переваг можна виділити те, що добре розроблений алгоритм полегшує налагодження, щоб ми могли виявити логічну помилку в програмі; алгоритм діє як проект, план програми та допомагає під час розробки програми; алгоритм не залежить від мов програмування і може бути легко кодований за допомогою будь-якої мови високого рівня. Щодо недоліків, то список буде коротшим:

розробка алгоритму для складних задач буде трудомісткою і складною для розуміння; розуміння складної логіки за допомогою алгоритмів може бути дуже складним.

1.3. Базові структури алгоритмів

Алгоритми, в залежності від початкових умов та мети задачі, способів її вирішення та визначення дій виконавця мають певну класифікацію, що собою відображає особливості реалізації того чи іншого алгоритму. Існує три основних типи:

- Лінійний алгоритм
- Розгалужений алгоритм
- Циклічний алгоритм

Рисунок 1.3.1 - Типи алгоритмічних процесів

Загалом, алгоритми можна представити як певні структури, що, в свою чергу, складаються з інших елементів – базових алгоритмічних структур. Характерна особливість цих структур полягає в тому, що у них є лише один вхід та один вихід. Логічна структура будь-якого алгоритму складається з комбінації

вищезгаданих базових алгоритмічних структур: розгалуження, слідування та циклу [9].

1.3.1. Лінійний алгоритм

Лінійний алгоритм – передбачає одержання розв’язку задачі одноразовим виконанням певної послідовності дій. Такі алгоритми застосовуються у найпростіших для процесу алгоритмізації задачах, в котрих перетворення інформації відбувається покроково за визначеними формулами.

Для представлення такого алгоритму використовується алгоритмічна конструкція послідовного виконання – «слідування», що передбачає собою послідовне виконання дій у визначеному порядку. У блок-схемі позначення такої конструкції здійснюється з допомогою блоків «процес» [9].

Простим прикладом лінійного алгоритму є процес обчислення за формулою (-

Рисунок 1.3.2 - Блок "процес"

ами). Сам процес обчислення є досить складним, він потребує час та пам’ять для збереження як кінцевих, так і проміжних результатів. Порядок виконання обчислень у мовах програмування визначається за пріоритетом операцій, в тому числі і за дужками. Складність таких алгоритмів може викликати збої у роботі програми, або ж дуже довгу її екзикуцію. Тому для вирішення цих проблем доцільно розбити складний вираз на декілька простіших враховуючи лінійність запису виразу.

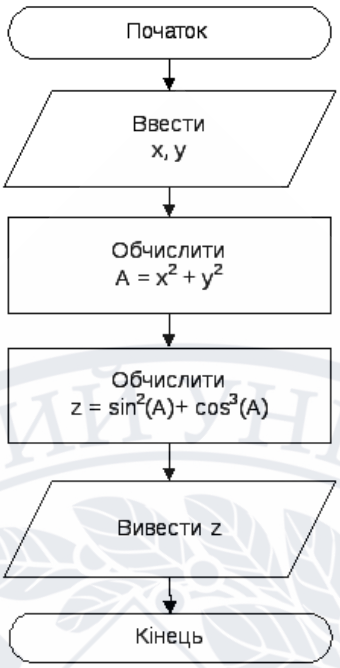


Рисунок 1.3.3 - Алгоритм обчислення виразу z = sin2(x2 + y2) + cos3(x2 + y2)

1.3.2. Розгалужений алгоритм

Розгалужений алгоритм – передбачає вибір однієї з кількох можливих послідовностей дій залежно від умови. Він використовується, коли перетворення інформації може відбуватись різними шляхами, залежно від властивостей вхідних даних та проміжних результатів. У розгалужених алгоритмах передбачаються всі можливі варіанти оброки інформації, кожен з яких розглядається як окрема гілка алгоритму, де вибір потрібної для виконання здійснюється за допомогою перевірки умови, що відображає усі властивості інформації, потрібної для процесу перетворення [9].

Для представлення розгалужених алгоритмів використовуються алгоритмічні конструкції розгалуження (вибору). Така структура обирає один з шляхів роботи алгоритму опираючись на результати перевірки необхідних умов. Алгоритм працюватиме незалежно від обраного шляху, адже усі вони ведуть до спільного виходу.

Структуру розгалуження можна поділити на чотири варіанти:

- «якщо – то – інакше» – повне розгалуження, виконання дій незалежно від виконання заданої умови. Остання ж формулюється таким чином, щоб відповідь перевірки була «так» або «ні».

- «якщо – то» – неповне розгалуження, виконання дій тільки у разі виконання або невиконання умови. Тобто є гілка, що не передбачає ніяких дій.

Для наступних двох варіантів необхідно ввести декілька нових означень:

1) багатоваріантним розгалуженням називається процес, коли всі логічні блоки алгоритму поєднуються в один блок аналізу змінної, що відповідає за гілку. Такий блок матиме стільки виходів, скільки існує варіантів обробки;
2) якщо для такого розгалуження існує варіант обробки, коли значення змінної не належить до перелічених значень, то таке розгалуження називається повним багатоваріантним вибором, а інакше – неповним [9].

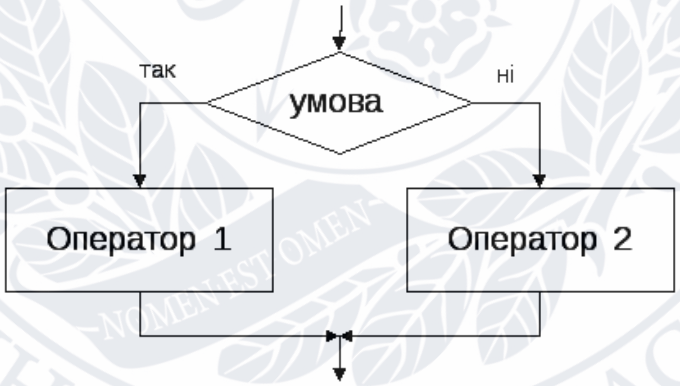


Рисунок 1.3.4 - Блок-схема повного розгалуження

- «вибір – інакше» – повний багатоваріантний вибір. При великій кількості умов для кожної гілки обирається змінна, що зберігає значення або проміжки значень даної гілки.

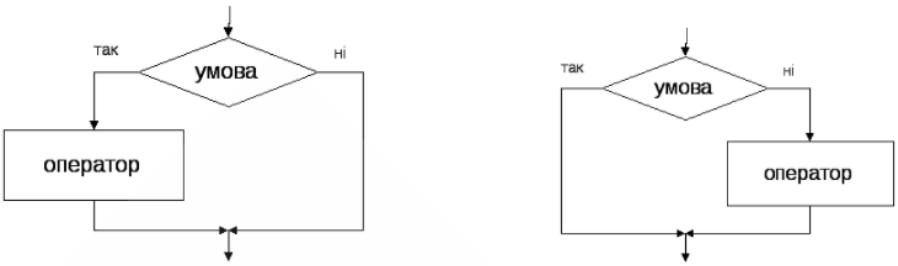


Рисунок 1.3.5 - Блок-схеми неповного розгалуження

- «вибір» – неповний багатоваріантний вибір.

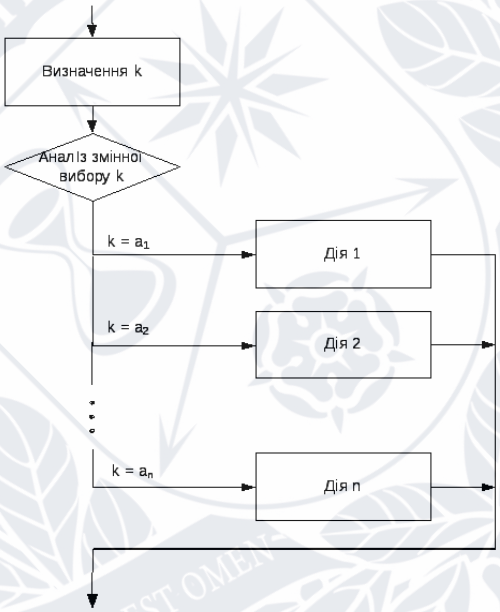


Рисунок 1.3.5 - Блок-схеми багатоваріантного вибору

1.3.3. Циклічний алгоритм

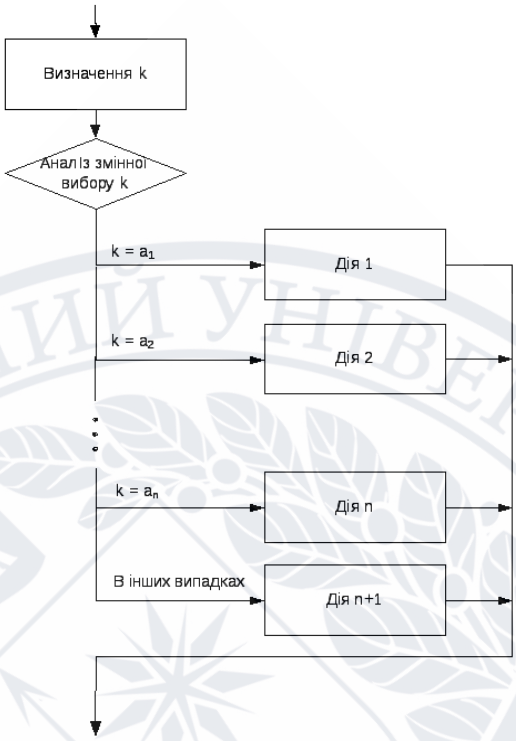


Рисунок 1.3.6 - Блок-схема циклу з параметром

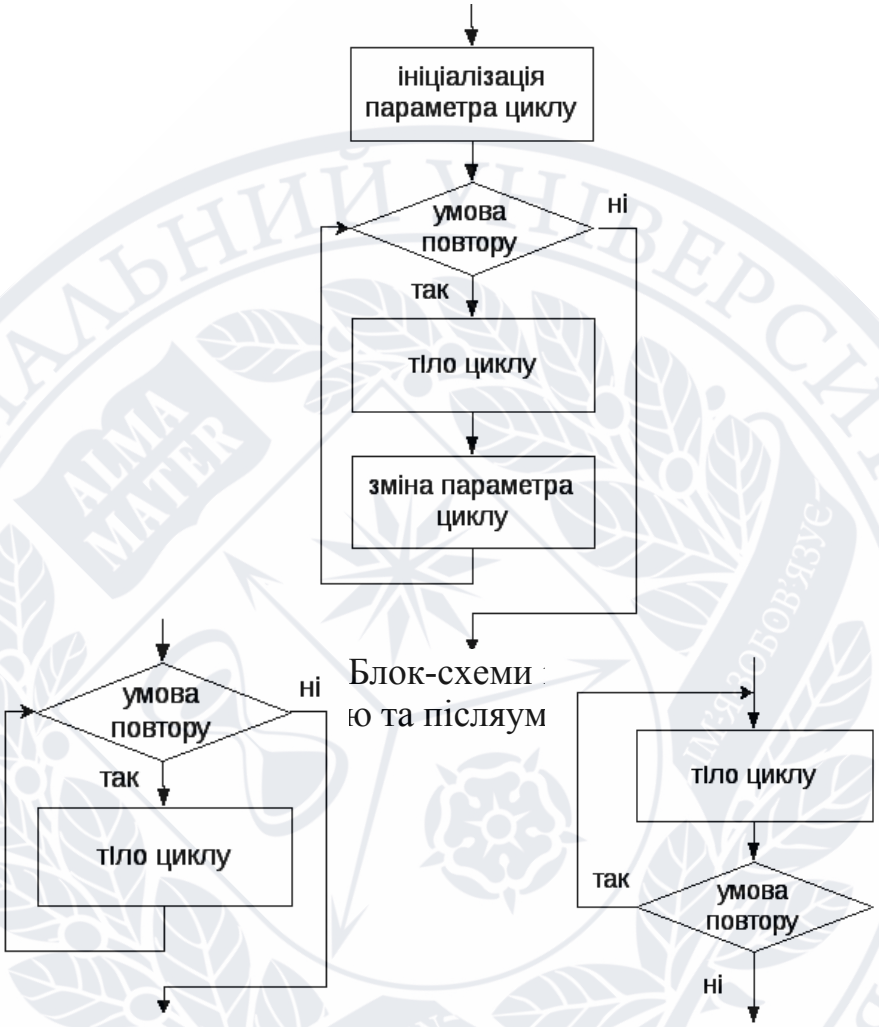
Циклічний алгоритм – передбачає одержання розв’язку задачі багаторазовим виконанням певної послідовності дій. Повторювану частину таких циклів називають тілом циклу. Для побудови циклічного алгоритму необхідно виконати наступні кроки [9]:

- визначити всі дії, що потрібні для входу до циклу, підготувати його;
- визначити всі операції у циклі;
- визначити умову виходу із циклу.

Для представлення таких алгоритмів використовуються алгоритмічні конструкції повторення, які реалізуються одним із трьох способів:

- Прості цикли з параметром.

Якщо під час перетворення інформації є змінна, що змінюється за певним визначеним правилом, то можна визначити кількість ітерацій циклу та організувати вихід з циклу.



Блок-схеми ю та післяум

- Ітераційні цикли.

«Розв’язання систем лінійних алгебраїчних рівнянь з десятками чи сотнями невідомих, пошук коренів алгебраїчних рівнянь високих степенів, розв’язання систем диференційних рівнянь, інтерполяція та екстраполяція функцій, обчислення значень функції за допомогою рядів, інтегрування тощо — усі ці задачі розв’язуються за допомогою циклічних алгоритмів, що реалізують циклічні ітераційні процеси, для яких заздалегідь неможливо визначити кількість повторень циклу» [9]. Тому умова виходу з циклу формулюється залежно від самої задачі. Також існує два види ітераційних циклів – з

передумовою та післяумовою. У циклі з передумовою, власне, умова знаходиться перед ітерацією, а з післяумовою – після ітерації циклу.

- Складні циклічні процеси.

З вищезгаданих структур можна будувати різні циклічні процеси із вкладеними циклами у них. Останні можуть бути внутрішні та зовнішні. Для зміни параметру у зовнішньому циклі відбувається багаторазове виконання визначених дій у внутрішньому (вкладеному) циклі. Кількість таких вкладених циклів не є обмеженою. Так, як особливістю базових алгоритмічних конструкцій є те, що вони мають лише один вхід та один вихід, то вони можуть вкладатися одна в одну довільним чином [9].

Так, комбінуючи різні алгоритмічні структури, можна отримати алгоритми будь-якої складності. За допомогою них можна створити навіть найскладніший алгоритм.

1.4. Етапи розробки алгоритму

Для вирішення будь-якої проблеми необхідно створити план – алгоритм дій. Є чимало шляхів, як це можна зробити. Деякі є занадто неформальними, інші навпаки – формальні та мають математичну природу, деякі ж можуть бути просто графічними. Розробка алгоритму - це ключовий крок у вирішенні задачі. То ж розробка алгоритму складається з п’яти основних кроків:

- Опис проблеми

Досить часто опис проблеми є найслабшою частиною усього процесу, адже він може страждати від декількох типів помилок: неоднозначність опису, його неповнота, внутрішні суперечності. Завданням розробника алгоритму є виявлення дефекту в описі та правильна обробка помилок.

- Аналіз проблеми

Метою цього кроку є визначення як початкової, так і кінцевої точок для вирішення проблеми. Цей процес аналогічний математику, який визначає, що дається і що має бути доведено. Хороший опис проблеми полегшує виконання цього кроку.

Визначаючи вихідну точку, слід почати з пошуку відповідей на такі запитання: які дані доступні, де ці дані, які формули стосуються проблеми, які взаємозв'язки між даними тощо. Визначаючи кінцеву точку, нам потрібно описати характеристики рішення. Іншими словами: «Як ми маємо дізнатись, коли закінчимо?»

- Розробка алгоритму високого рівня

Алгоритм - це план вирішення проблеми, але плани мають кілька рівнів деталізації. Зазвичай краще починати з високорівневого алгоритму, який включає основну частину рішення, але залишає деталі на наступний крок.

- Деталізація алгоритму

Алгоритм високого рівня показує основні кроки, яких потрібно дотримуватися для вирішення проблеми. Тепер потрібно додати деталей до цих кроків, але скільки деталей потрібно додати? Відповідь на це питання залежить від ситуації. Необхідно розглянути, хто (або що) збирається впровадити алгоритм і скільки ця людина (або річ) вже знає, як це зробити. Коли мета - розробити алгоритми, які ведуть до комп'ютерних програм, потрібно врахувати можливості комп'ютера та надати достатньо деталей, щоб хтось інший міг використати наш алгоритм для написання комп'ютерної програми, яка відповідає крокам нашого алгоритму. Для складних проблем, як правило, кілька разів проходять цей процес, розробляючи алгоритми середнього рівня. Кожного разу потрібно додавати більше деталей до попереднього алгоритму, зупиняючись, коли не видно користі для подальшого вдосконалення. Цю техніку поступового переходу від високого рівня до детального алгоритму часто називають поетапним вдосконаленням - процесом розробки детального алгоритму шляхом поступового додавання деталей до алгоритму високого рівня

- Перегляд алгоритму

Останній крок - перегляд алгоритму. Що шукається? По-перше, потрібно поетапно проробити алгоритм, щоб визначити, чи він вирішить вихідну проблему чи ні. Після того, як точно встановлено, що алгоритм надає рішення проблеми, потрібно почати шукати інші речі. Наступні запитання типові для тих, які слід задавати, коли ми переглядаємо алгоритм:

- Цей алгоритм вирішує дуже конкретну задачу чи вирішує більш загальну задачу?
- Якщо він вирішує цілком конкретну проблему, чи слід його узагальнювати?
- Чи можна спростити цей алгоритм?
- Чи подібне рішення до вирішення іншої проблеми?
- Чим вони схожі? Чим вони відрізняються?

Задавання цих питань та пошук їх відповідей - це хороший спосіб розвинути навички, які можна застосувати до будь-якої проблеми.

1.5. Складність алгоритмів та оцінка їх ефективності

Одними з найбільш значущих властивостей алгоритмів є їх складність та ефективність. Тому при різноманітному застосуванні алгоритмів дуже важливо вміти правильно оцінювати їх. У програмуванні ж, обчислювальну складність алгоритмів оцінюють за допомогою кількості дій, що виконує алгоритм та за обсягом при цьому задіяної пам’яті. Перед написанням алгоритму потрібно визначити необхідні йому ресурси, оцінити те, як складність буде залежати від кількості вхідних даних, щоб алгоритм не працював нескінченну кількість часу. Для досягнення максимальної ефективності рекомендовано зменшити використання ресурсів. Однак час та пам'ять не можна порівняти, тому який із двох алгоритмів вважати більш ефективним залежить від того, який фактор для

конкретної задачі важливіший – вимога високої швидкості, мінімального використання пам'яті чи інша міра ефективності.

Складність алгоритму — це «кількісна оцінка ресурсів, що витрачаються алгоритмом. В якості ресурсів можна розглядати людські (на створення і розуміння алгоритму) і обчислювальні (на виконання алгоритму) ресурси. Тому розрізняють описову (інтелектуальну) і обчислювальну складність алгоритму» [11].

Описова складність визначається з самого запису алгоритму та характеризується довжиною цього запису. Відомо, що єдиного критерію оцінки такої складності не існує.

Як обчислювальні ресурси для виконання алгоритму розглядають пам'ять і процесорний час. Тому основними мірами обчислювальної складності алгоритмів є:

- ємнісна (або просторова) складність – кількість пам'яті, необхідної для виконання алгоритму;
- часова складність – кількість часу, необхідного на виконання алгоритму (цей час, як правило, визначається кількістю елементарних операцій, які потрібно виконати для реалізації алгоритму) [11].

Комп'ютери мають обмежений обсяг пам'яті. «Якщо дві програми реалізують ідентичні функції, то та, яка використовує менший обсяг пам'яті, характеризується меншою ємнісною складністю. Іноді пам'ять є домінуючим чинником в оцінці ефективності програм. Однак в останні роки у зв'язку із швидким її здешевленням ця складова ефективності поступово втрачає своє значення» [11].

Для оцінки часової складності можна просто запустити кожен алгоритм на декількох задачах і порівняти час виконання. Інакший спосіб – це оцінити час виконання підрахунком операцій математично. У цьому випадку складність алгоритму описується функцією $f(n)$, де n - розмір вхідних даних (це може бути розмірність масиву, кількість слів в тексті, довжина послідовності в алгоритмі, число вершин оброблюваного графа тощо).

Знаходження точної залежності $f(n)$ для алгоритму - задача доволі складна. Часто буває, що така докладна оцінка не потрібна. З цієї причини зазвичай обмежуються лише асимптотичними оцінками швидкості росту функції, які описують граничну поведінку складності алгоритму при збільшенні n (тобто те, наскільки швидко або повільно зростає ця функція).

Загалом використовується декілька підходів, які виражають порядок складності алгоритму. В їх основі лежить порівняння функції $f(n)$ з якою-небудь функцією, поведінка якої є дослідженою.

Отже, розрізняють верхні (O), нижні (Ω) і ефективні (Θ) асимптотичні оцінки.

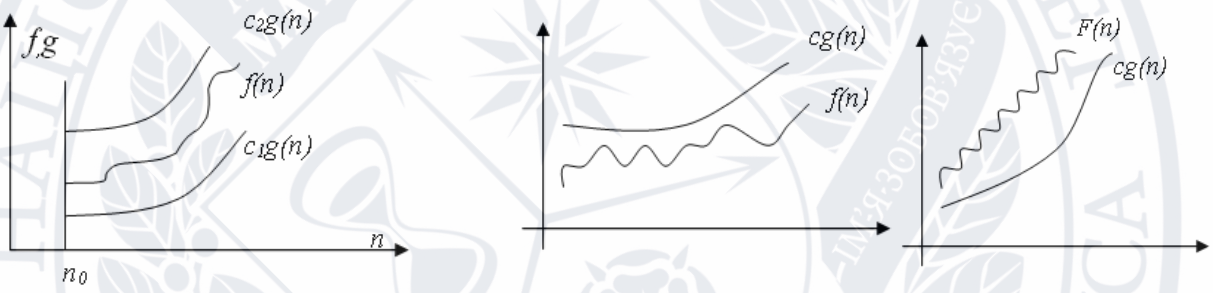


Рисунок 1.5.1 - Асимптотичні оцінки складності алгоритму, верхня, нижня та ефективна, відповідно

Найбільш популярною для оцінювання складності алгоритмів є верхня оцінка - O -нотація. Її позначення:

$$f(n) = O\big(g(n)\big), \tag{1.5.1}$$

f(n)= θ(g(n)), якщо ∃ c_1>0, c_2>0, n_0>0, такі, що
c_1*g(n) ≤ f(n) ≤ c_2*g(n), при n > n_0

де $O\big(g(n)\big)$ формально означає, що час роботи алгоритму (або обсягу пам'яті) зростає в залежності від обсягу вхідних даних не швидше, ніж деяка константа, помножена на $g(n)$. Ця нотація дозволяє враховувати у функції лише значущі елементи, відкидаючи ті, які менш важливі.

Розглянемо алгоритм обчислення значення багаточлена степеня n в заданій точці x [11]:

$$P_n(x) = a_n X^n + a_{n-1} X^{n-1} \ldots + a_i xi + \cdots + a_1 x^1 + a_0$$

- Алгоритм 1. Для кожного доданка, окрім a_0, піднести x в задану степінь послідовним множенням і домножити на коефіцієнт. Потім доданки скласти.

Обчислення і-го доданку $(i = 1 \ldots n)$ даним алгоритмом вимагає і множень. Значить, всього $1 + 2 + 3 + \cdots + n = n(n+1)/2$ множень. Додатково потрібно $n + 1$ додавання. Всього $n(n+1)/2 + n + 1 = n^2/2 + 3n/2 + 1$ операцій.

- Алгоритм 2. Винесемо x за дужки і перепишемо багаточлен у вигляді $P_n(x) = a_0 + x(a_1 + x(a_{2+} \ldots x(a_i + \cdots x(a_{n-1} + a_n x))))$

Наприклад ,

$$P_3(x) = a_3 x^3 + a_2 x^2 + a_1 x^1 + a_0 = a_0 + x\big(a_1 + x(a_2 + a_3 x)\big).$$

Будемо обчислювати вираз зсередини. Сама внутрішня дужка вимагає одне множення і одне додавання. Її значення використовується для наступної дужки. І так, одне множення і одне додавання на кожну дужку, яких $n - 1$ штука. І ще після обчислення самої зовнішньої дужки помножити на x і додати a_0. Всього n множень $+n$ додавань $= 2n$операцій.

У функції $f_{(n)} = n^2/2^{+3}n\,/2 + 1$при досить великих n компонента n^2 буде значно перевершувати інші складові, і тому характерна поведінка цієї функції визначається саме цією компонентою. Інші компоненти (порівняно повільно зростаючий доданок 3n/2 +1 і константний множник 1/2) можна відкинути і умовно записати, що дана функція має оцінку поведінки (в сенсі швидкості росту її значень) виду O(n^2) (читається як "О велике від ен квадрат"). Фраза «складність алгоритму є O(n^2)» означає, що при великих n час роботи алгоритму (або загальна кількість операцій) не більше ніж $c \cdot n^2$, де c - якась додатна константа [11].

Таким чином, O() - "урізана" оцінка часу роботи алгоритму, яку часто набагато простіше отримати, ніж точну формулу для кількості операцій.

Важливе теоретичне і практичне значення має класифікація алгоритмів, яка бере до увагу швидкість зростання цієї функції.

Отже, залежно від складності, виділяють такі основні класи алгоритмів:

- лінійні: O(n)

Лінійне зростання — подвоєння розміру задачі подвоїть і необхідний час. При $n = 1$, тобто O(1) - сталий час роботи незалежно від розміру задачі.

- логарифмічні: $O(\log n)$

Логарифмічне зростання — подвоєння розміру задачі збільшує час роботи на сталу величину

- поліноміальні: O(n^m), де m - натуральне число, більше від одиниці (при $m = 1$ алгоритм є лінійним)

Квадратичне зростання — подвоєння розміру задачі вчетверо збільшує необхідний час.

Кубічне зростання — подвоєння розміру задачі збільшує необхідний час у вісім разів.

- експоненціальні: O(c^n), де c - натуральне число, більше від одиниці

Експоненціальне зростання — збільшення розміру задачі на одиницю призводить до c-кратного збільшення необхідного часу; подвоєння розміру задачі підносить необхідний час у квадрат [11].

Для однієї й тієї ж задачі можуть існувати алгоритми різної складності. Наприклад, для попередньої задачі алгоритм 1 має складність O(n^2), алгоритм 2 - O(n).

Часто буває і так, що більш повільний алгоритм працює завжди, а більш швидкий - лише за певних умов.

Розглянемо правила формування оцінки O( ) [11]:

- При оцінці за функцію береться кількість операцій, що зростає найшвидше.

Тобто, якщо в програмі одна функція, наприклад, множення, виконується O(n) разів, а додавання - O(n^2) разів, то загальна складність програми - O(n^2), оскільки при збільшенні n додавання стануть виконуватися настільки часто, що будуть впливати на швидкодію куди більше, ніж множення.

- При оцінці O( ) константи не враховуються.

Нехай один алгоритм робить $2500n + 1000$ операцій, а інший - $2n + 1$ Обидва вони мають оцінку O(n), так як їхній час виконання зростає лінійно.

Інший наслідок опускання константи - алгоритм з часом O(n^2) може працювати значно швидше алгоритму O(n) при малих n. За рахунок того, що реальна кількість операцій першого алгоритму може бути $n^2 + 10n + 6$, а другого – $1000000n + 5$. Втім, другий алгоритм рано чи пізно обжене перший, оскільки n^2 росте куди швидше – $1000000n$.

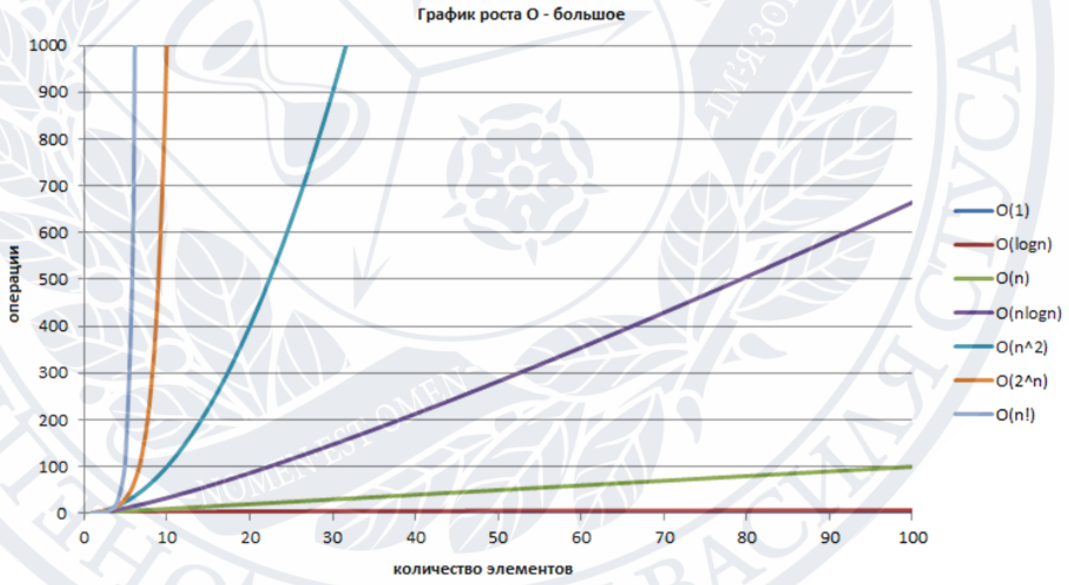


Рисунок 1.5.1 – Графіки росту O-велике в залежності від кількості операцій та елементів

1.6. Найпоширеніші види алгоритмів

1.6.1. Рекурсивний алгоритм

Цей тип алгоритму заснований на рекурсії. У рекурсії проблема розв'язується шляхом її розбиття на підзадачі того ж типу і повторне викликання власного себе, поки проблема не буде вирішена за допомогою базової умови. Це один з найцікавіших алгоритмів, оскільки він неодноразово викликає себе, доки проблема не буде вирішена. Деякі загальні проблеми, які вирішуються за допомогою рекурсивних алгоритмів, - це факторіал числа, серія Фібоначчі, вежа Ханоя, DFS графа тощо.

Взагалі, рекурсивні комп’ютерні програми вимагають більше пам’яті та обчислень порівняно з ітераційними алгоритмами, але вони простіші та у багатьох випадках є природним способом мислення про проблему. Тож розглянемо приклад рекурсивного алгоритму та його порівняння з ітераційним підходом:

- Алгоритм знаходження k-го парного натурального числа.

Тут необхідно зауважити, що це можна вирішити дуже просто, просто виводячи 2 * (k - 1) для даного k. Однак мета тут полягає в тому, щоб проілюструвати основну ідею рекурсії, а не вирішити проблему. Отже, маємо:

Алгоритм 1: функція Even(ціле додатне число k)

Вхідні дані: k, ціле додатне число

Вихід: k-те парне натуральне число (перше парне - 0)

Алгоритм:

якщо k = 1, то повернеться 0;

інакше повернеться Even(k-1) + 2.

Тут обчислення Even(k) зводиться до обчислення парних для меншого вхідного значення, тобто парного (k-1). Even(k) з часом стає Even(1), що дорівнює 0 у

першому проходженні. Наприклад, для обчислення Even(3) алгоритм Even(k) викликається з k = 2. При обчисленні Even(2) алгоритм Even(k) викликається з k = 1. Оскільки Even(1) = 0 , 0 повертається для обчислення Even(2), і отримується Even(2) = Even(1) + 2 = 2. Це значення 2 для Even(2) тепер повертається до обчислення Even(3), і отримується Even(3) = Even(2) + 2 = 4.

Для порівняння подивимося, як одну і ту ж проблему можна вирішити за допомогою ітераційного алгоритму:

Алгоритм 2: Even(ціле додатне число k)

Вхідні дані: k, ціле додатне число

Вихід: k-е парне натуральне число (перше парне - 0)

Алгоритм:

```
int i, Even;

i: = 1;

Even: = 0;

в той час як (i <k) {

        Even: = Even + 2;

        i: = i + 1;

}

повернути Even.
```

Як рекурсивна, так і ітераційна програми мають однакові можливості вирішення проблем, тобто кожна рекурсивна програма може бути написана ітеративно, і навпаки, це також відповідає дійсності. Рекурсивна програма має більші вимоги до простору, ніж ітераційна програма, оскільки всі функції залишатимуться в стеці до досягнення базового випадку. Він також має більші вимоги до часу через виклики функцій і повернення накладних витрат.

Натомість, рекурсія забезпечує чистий і простий спосіб написання коду. Деякі проблеми за своєю суттю є рекурсивними, як обхід дерев, задача про ханойську вежу тощо. Для таких проблем переважно писати рекурсивний код.

1.6.2. «Розділяй і володарюй»

Це ще один ефективний спосіб вирішення багатьох проблем. У алгоритмах типу «розділяй і володарюй» робота алгоритму поділяється на дві частини; у першій частині проблему поділяються на менші підзадачі того ж типу. Потім, у другій частині, ці менші проблеми вирішуються, а потім результати складаються (комбінуються), щоб отримати остаточне рішення проблеми. Тобто, цей прийом можна розділити на такі три частини:

- Розділяй: Це передбачає поділ загальної проблеми на підпроблеми.
- Володарюй: розв’язання проблеми за допомогою рекурсії, доки вона не буде розв’язана
- Поєднати: поєднання розв’язків з пункту «Володарюй».

Деякі загальні проблеми, які вирішуються за допомогою таких алгоритмів - це двійковий пошук, сортування об’єднань, швидке сортування, множення матриці Штрассена тощо.

1.6.3. Динамічні алгоритми

Динамічне програмування - це, головним чином, оптимізація над простою рекурсією. Скрізь, де є рекурсивне рішення, яке повторює виклики однакових входів, можна оптимізувати його за допомогою динамічного програмування.

Ці алгоритми працюють, запам'ятовуючи результати минулого циклу та використовуючи їх для пошуку нових результатів. Іншими словами, алгоритм динамічного програмування вирішує складні задачі, розбиваючи їх на кілька

простих підзадач, а потім вирішує кожну з них один раз, зберігаючи результати для подальшого використання.

Ця проста оптимізація зменшує складність у часі від експоненціального до поліноміального. Наприклад, якщо ми пишемо просте рекурсивне рішення для чисел Фібоначчі, ми отримуємо експоненціальну складність часу, а якщо оптимізуємо її, зберігаючи рішення підзадач, складність часу зменшується до лінійної.

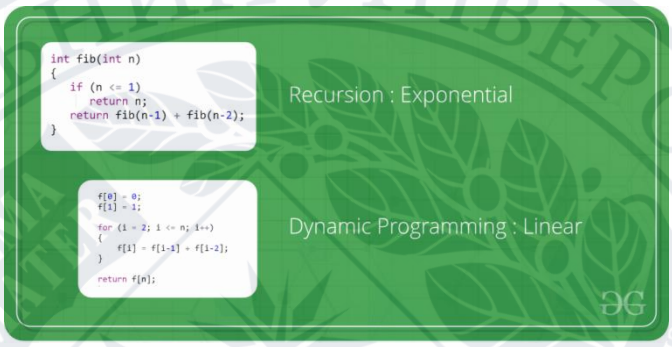


Рисунок 1.6.1 - Реалізація алгоритму рекурсивним та динамічним методами

1.6.4. Жадібні алгоритми

У жадібному алгоритмі рішення будується крок за кроком. Рішення про вибір наступної частини приймається на підставі того, що буде давати найкращий результат. Він ніколи не враховує вибір, зроблений раніше. Алгоритм складається з 5 компонентів:

- Набір кандидатів, з якого ми намагаємось знайти рішення.
- Функція відбору, яка допомагає вибрати найкращого з можливих кандидатів.
- Функціональна можливість, яка допомагає вирішити, чи можна використовувати кандидата для пошуку рішення.
- Цільова функція, яка присвоює значення можливому рішенню або частковому.
- Функція рішення, яка повідомляє, коли ми знайшли рішення проблеми.

Деякі загальні проблеми, які можна вирішити за допомогою жадібного алгоритму, - це алгоритм Прима («алгоритм побудови мінімального кістякового дерева зваженого зв'язного неорієнтованого графа. Побудова починається з дерева, що включає в себе одну вершину. Протягом роботи алгоритму дерево розростається, поки не охопить всі вершини вихідного графа» [10]), алгоритм Крускала (алгоритм побудови мінімального кістякового дерева зваженого неорієнтовного графа), кодування Хаффмана (стиснення даних без втрат), алгоритм Дейкстри (знаходження найкоротшого шляху від однієї вершини графа до всіх інших його вершин) тощо.

1.6.5. Алгоритм «грубої сили»

Це базовий та найпростіший тип алгоритмів. Алгоритм грубої сили (метод повного перебору, метод вирішення «в лоб») - це прямолінійний підхід до проблеми, тобто перший підхід, який приходить нам на думку при розгляді проблеми. Більш технічно це ітерація кожної можливості вирішення цієї проблеми.

Наприклад: є 4-значний PIN-код. Цифри, які слід вибрати - від 0 до 9. Тоді алгоритм грубої сили буде випробовувати всі можливі комбінації по одному, тобто: 0001, 0002, 0003, 0004, і так далі, поки ми не отримаємо правильний PIN-код. У гіршому випадку знадобиться 10 000 спроб, щоб знайти правильну комбінацію.

Як і усі алгоритми, він має свої плюси та мінуси. Розглянемо їх. Плюси:

- Метод грубої сили - це гарантований спосіб знайти правильне рішення шляхом перерахування всіх можливих варіантів вирішення проблеми.
- Це загальний метод, який не обмежується будь-якою конкретною областю проблем.
- Метод грубої сили ідеально підходить для вирішення невеликих і простих задач.

- Він відомий своєю простотою і може служити еталоном порівняння.

Мінуси:

- Підхід «грубої сили» неефективний. Що стосується проблем у режимі реального часу, складність алгоритмів часто перевищує O(n!).
- Цей метод покладається більше на компрометацію потужності комп'ютерної системи для вирішення проблеми, ніж на хороший дизайн алгоритму.
- Алгоритми «грубої сили» повільні.
- Алгоритми «грубої сили» не є конструктивними чи творчими в порівнянні з алгоритмами, які побудовані з використанням деяких інших парадигм проектування.

1.6.6. Алгоритм пошуку з поверненням

Отже, алгоритм «грубої сили» – це техніка, яка гарантує вирішення проблем будь-якої сфери, допомагає вирішувати більш прості задачі, а також пропонує рішення, яке може служити еталоном для оцінки інших методів проектування, але займає багато часу і є неефективним.Алгоритм пошуку з поверненням - це вдосконалений метод «грубої сили». Тут ми починаємо з одного з можливих варіантів із багатьох доступних та намагаємось вирішити проблему.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Рисунок 1.6.2 - Гра "Судоку"

Якщо зможемо вирішити проблему за допомогою вибраного ходу, тоді ми запишемо рішення, в іншому випадку – ми відступимо назад, видалимо результат невдалого ходу та оберемо інший хід, спробуємо його вирішити. Це форма рекурсії, тільки коли заданий варіант не може дати рішення, ми повертаємося до попереднього варіанту, який може дати рішення, і так продовжуємо далі з іншими варіантами. Іншими словами, алгоритм зворотного відстеження вирішує підзадачу, і якщо йому не вдається вирішити проблему, він скасовує останній крок і починає знову шукати рішення проблеми. Розглянемо проблему на прикладі гри «Судоку». Отже, ми намагаємось заповнити цифри одну за одною. Щоразу, коли ми виявляємо, що поточна цифра не може призвести до рішення, ми видаляємо її (зворотне відстеження) і пробуємо наступну цифру. Це краще, ніж підхід «в лоб» (генерувати всі можливі комбінації цифр, а потім пробувати кожну комбінацію по одному), оскільки він скидає набір перестановок, коли він відступає назад.

Алгоритм пошуку з поверненням застосовується до деяких конкретних типів проблем:

- Проблема прийняття рішення – пошук можливого рішення проблеми.
- Проблема оптимізації – пошук найкращого рішення, яке можна застосувати.
- Проблема перебору – пошук всіх можливих рішень задачі.

Наочним прикладом може бути наступна демонстрація роботи алгоритму:

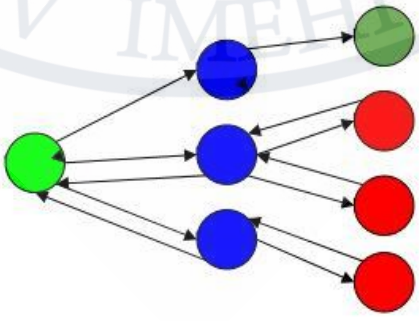


Рисунок 1.6.3 - Наочний приклад роботи алгоритму

Тут зелений круг - початкова точка, синій - проміжна точка, червоний - точки без реального рішення, темно-зелений - кінцевий. Як помітно, алгоритм поширюється до кінця, щоб перевірити, є це рішення чи ні, і якщо вони є, тоді повертає рішення, інакше – повертається до точки, що знаходиться на кроці позаду, щоб знайти доріжку до наступної точки на шляху до розв’язку.

Метод пошуку з поверненням є доволі універсальним. Достатньо легко проектувати та програмувати алгоритми розв’язку задач за допомогою цього методу. У той час як остаточне рішення може бути дуже великим, навіть при невеликих розмірах вхідних даних, та займати не просто години, а дні чи тижні для розв’язку, тому щодо практичного застосування не може бути і мови. Таким чином, при проектуванні цих алгоритмів, обов'язково теоретично оцінюється час їх роботи на конкретних даних. Існують також завдання вибору, для вирішення яких можна побудувати унікальні, «швидкі» алгоритми, що дозволяють швидко отримувати рішення навіть при більших розмірах завдань. Метод пошуку з поверненням у таких завданнях застосовувати неефективно.

Алгоритми пошуку з поверненням застосовуються у низці відомих задач: задача про 8 ферзів, задача про суму підмножини, побудова дерев ігор, задача про проблему рицарського турніру, судоку, побудови латинського та греко-латинського квадратів, гамільтонів цикл, у різних головоломках тощо. Деякі найпопулярніші задачі буде розглянуто у наступному розділі. Але акцент буде зроблено саме на задачу побудови греко-латинського квадрату, адже це задача, розв’язку якої присвячено найменше інформації. Знайти гарний працюючий алгоритм з поверненням є майже неможливим завданням. Тому написання алгоритму та його аналіз саме для цієї задачі є цікавим та кропітким процесом.

ВИСНОВОК ДО РОЗДІЛУ 1

Отже, можна зробити висновок, що поняття алгоритму пройшло складний шлях від моменту своєї появи до сьогодення. Кожна епоха додавала свої

поправки до його значення та застосування. Тому сьогодні ми маємо дуже багато характеристик, якими можна було б описати усе, що пов’язане з алгоритмом. Мабуть, найважчою для стислого опису є складність алгоритмів та оцінка їх ефективності, при цьому це одні з найважливіших характеристик, адже при різноманітному застосуванні алгоритмів дуже важливо вміти правильно оцінювати їх.

Серед усіх видів алгоритмів особливу увагу варто приділити алгоритму пошуку з поверненням, адже надалі, у другому розділі, буде детальніше розглянуто способи розв’язку задач саме цим способом. Уся описана інформація у першому розділі допоможе розкрити суть теми даної роботи через призму аналізу роботи алгоритму пошуку з поверненням для задачі побудови греко-латинського квадрату.

РОЗДІЛ 2
ЗАСТОСУВАННЯ АЛГОРИТМІВ ПОШУКУ З ПОВЕРНЕННЯМ У ВІДОМИХ ЗАДАЧАХ

2.1. Задача про вісім ферзів

Типовою проблемою, що розв’язується методом пошуку з поверненням є класична задача про вісім ферзів (англ. *n-Queens / королев, дам*), запропонована німецьким шаховим ентузіастом Максом Бецзелем у 1848 р. (під його псевдонімом “Schachfreund”) для стандартної дошки 8×8 та Франсуа-Йозефом Юсташем Ліонеттом в 1869 році для більш загальної дошки - $n \times n$. Проблема полягає в розміщенні n дам на шаховій дошці $n \times n$ так, що жодна дама не атакує одна одну. Це означає, що не існує двох дам, що знаходяться в одному рядку, одному стовпці або тій самій діагоналі.

У листі, написаному своєму товаришу Генріху Шумахеру в 1850 році, видатний математик Карл Фрідріх Гаус писав, що можна легко підтвердити

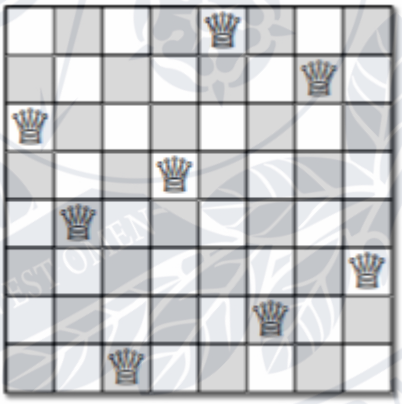


Рисунок 2.1.1 – Перший запропонований Гаусом розв'язок

твердження про те, що проблема восьми цариць має 92 рішення методом спроб і помилок за кілька годин. У листі Гауса описана наступна рекурсивна стратегія вирішення проблеми n -дам (та сама стратегія була описана в 1882 році французьким математиком-любителем Едуардом Лукасом, який приписував метод Еммануелю Лак'єру) : королеви розміщуємо на дошці по одному ряду,

починаючи зверху. Щоб розмістити r -ту королеву, ми методично пробуємо всі n квадратів у рядку r зліва направо в простій петлі for. Якщо на певний квадрат нападає попередня королева, ми ігноруємо цей квадрат; в іншому випадку ми попередньо розміщуємо королеву на цій площі і рекурсивно намацуємо послідовні розміщення королев у наступних рядах.

Наступний малюнок показує результуючий алгоритм, який рекурсивно перераховує всі повні розв'язки n -ферзів, які узгоджуються з даним частковим розв'язком. Слідом за Гаусом ми представляємо позиції маток за допомогою масиву $Q[1..n]$, де $Q[i]$ вказує, який квадрат у рядку i містить

```
PLACEQUEENS(Q[1..n], r):
  if r = n + 1
      print Q[1..n]
  else
      for j ← 1 to n
          legal ← TRUE
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
                  legal ← FALSE
          if legal
              Q[r] ← j
              PLACEQUEENS(Q[1..n], r + 1)      ⟨⟨Recursion!⟩⟩
```

Рисунок 2.1.2 - Алгоритм пошуку з поверненням Гауса та Лак'єра для задачі n-ферзів

ферзь. Коли викликається PlaceQueens, вхідним параметром r є індекс першого порожнього рядка, а префікс $Q[1..r-1]$ містить позиції перших $r-1$ маток. Зокрема, для обчислення всіх рішень n-queens без обмежень ми б називали PlaceQueens $(Q[1..n], 1)$. Зовнішня петля for розглядає всі можливі розміщення ферзя на ряд r; внутрішній цикл for перевіряє, чи відповідає розміщення кандидата рядка r дамам, які вже на перших $r-1$ рядках. Виконання PlaceQueens можна проілюструвати за допомогою дерева рекурсії. Кожному вузлу в цьому дереві відповідає рекурсивна підзадача, а отже, юридичне часткове рішення; зокрема, корінь відповідає порожній дошці (з $r = 0$). Грані у дереві рекурсії відповідають рекурсивним викликам. Листя відповідають

частковим рішенням, які неможливо продовжити далі, або тому, що на кожному рядку вже є дама, або тому, що кожна позиція в наступному порожньому рядку атакується наявною дамою. Пошук повних рішень для зворотного відстеження еквівалентний пошуку в глибину по цьому дереві.

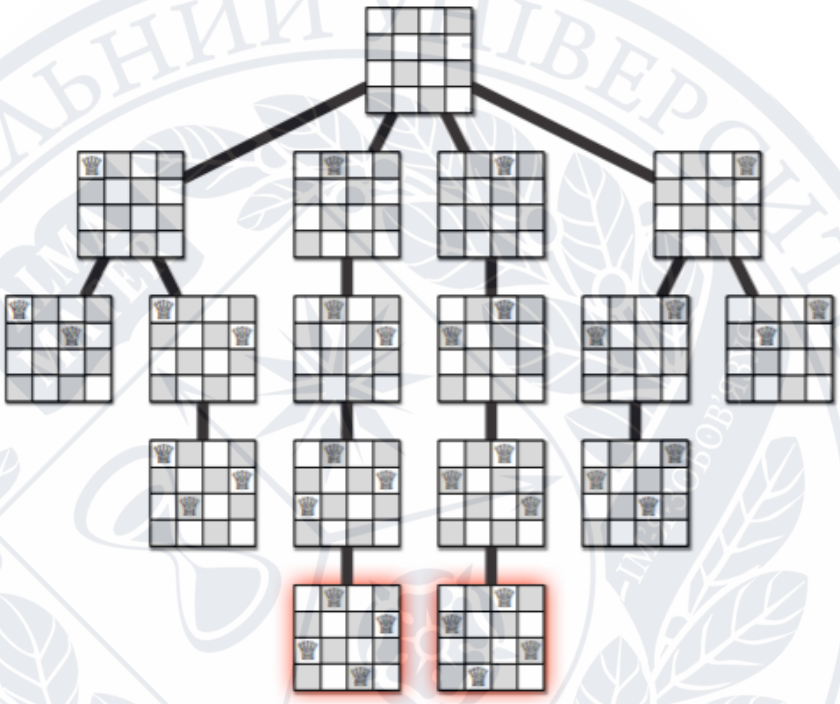


Рисунок 2.1.3 - Готове рекурсивне дерево розв'язків Гауса та Лак'єра для $n = 4$

2.2. Задача про лицарський турнір

Проблеми, які зазвичай вирішуються за допомогою пошуку з поверненням, мають наступну спільну властивість: ці проблеми можна вирішити лише спробувавши всі можливі конфігурації, і кожна конфігурація випробовується лише один раз. Простим рішенням цих проблем є випробування всіх конфігурацій та виведення конфігурації, яка відповідає заданим обмеженням проблеми. Зворотне відстеження працює поступово і є оптимізацією над простим рішенням, де генеруються та пробуються всі

можливі конфігурації. Наприклад, розглянемо наступну проблему лицарського турніру (англ. Knight's Tour.)

Постановка задачі: дається $n \times n$ дошка з лицарем, розміщеним на першому блоці порожньої дошки. Рухаючись за правилами шахів, лицар повинен відвідати кожен квадрат рівно один раз. Необхідно вивести номер (порядок) кожної комірки, в якій їх відвідують.

```
Input :
N = 8
Output:
0  59  38  33  30  17   8  63
37  34  31  60   9  62  29  16
58   1  36  39  32  27  18   7
35  48  41  26  61  10  15  28
42  57   2  49  40  23   6  19
47  50  45  54  25  20  11  14
56  43  52   3  22  13  24   5
51  46  55  44  53   4  21  12
```

Рисунок 2.2.1 - Приклад роботи алгоритму для задачі про лицарський турнір

Порівняємо алгоритм розв’язку «в лоб» та пошук з поверненням.

Алгоритм «в лоб» полягає в тому, щоб генерувати всі тури по одному і перевіряти, чи згенерований тур відповідає обмеженням:

```
Доки є непройдені квадрати
{
  створити наступний квадрат
  якщо шлях охоплює всі квадрати, то
  {
    надрукувати цей шлях;
  }
}
```

Алгоритм пошуку з поверненням працює поступово, щоб атакувати проблеми. Як правило, він починає з порожнього вектора розв’язання і по черзі додає елементи (значення елемента змінюється залежно від проблеми). Коли ми додаємо елемент, ми перевіряємо, чи додавання поточного елемента порушує обмеження проблеми, якщо це відбувається, тоді ми видаляємо елемент і пробуємо інші альтернативи. Якщо жодна з альтернатив не працює, ми переходимо до попереднього етапу та видаляємо доданий на попередньому етапі елемент. Якщо ми дійдемо до початкової стадії, тоді ми говоримо, що рішення не існує. Якщо додавання елемента не порушує обмежень, рекурсивно додаються елементи по одному. Якщо вектор рішення стає повним, відбувається вивід рішення. Отже, алгоритм у словесній формі має наступний вигляд:

- Якщо відвідано всі квадрати, то роздрукувати рішення.
- Інакше:
    1. Додайте один із наступних ходів до вектора розв’язку та рекурсивно перевірте, чи цей крок веде до рішення. (Лицар може зробити максимум вісім ходів. Ми обираємо один із 8 ходів на цьому кроці).
    2. Якщо перебіг, обраний на наведеному вище кроці, не призведе до рішення, то видаліть цей хід з вектора розв’язку та спробуйте альтернативні ходи.
    3. Якщо жодна з альтернатив не працює, поверніть false (Повертаючи false, програма видалить раніше доданий елемент у рекурсії, а якщо значення false повертається у початковому виклику рекурсії, тоді "рішення не існує".

2.4. Судоку

Суть завдання полягає в наступному: необхідно заповнити вільні клітинки цифрами від 1 до 9 так, щоб в кожному рядку, в кожному стовпці і в кожному малому квадраті 3×3, кожна цифра зустрічалася лише один раз. Вважається, що головоломка має одне рішення, проте зустрічаються спеціальні судоку з кількома варіантами розвитку подій.

Як і всі інші задача з пошуком з поверненням, судоку можна вирішити покроково, присвоюючи числа порожнім клітинкам. Перш ніж присвоювати номер, потрібно перевірити, чи безпечно його призначати. Необхідно точно переконатись, що однаковий номер відсутній у поточному рядку, стовпці та підквадраті 3×3. Перевіривши безпечність, треба призначити номер і рекурсивно перевірити, чи це призначення призводить до рішення чи ні. Якщо призначення не призводить до рішення, спробувати наступне число для поточної порожньої комірки. І якщо жодне з числа (від 1 до 9) не веде до рішення, повернути false і вивести повідомлення, що рішення не існує.

Алгоритм у словесній формі буде мати наступний вигляд:

- Створити функцію, яка перевіряє, чи після призначення поточного індексу сітка стає небезпечною чи ні. Зберегти хеш-таблицю для рядка, стовпця та квадратів. Якщо будь-яке число має частоту більше 1 у хеш-таблиці , то повернути false, а інакше – повернути true. Використання хеш-таблиці можна уникнути, використовуючи цикли.
- Створити рекурсивну функцію, що створює сітку значень.
- Перевірити наявність будь-якого порожнього місця. Якщо таке існує, то присвоїти число від 1 до 9 та перевірити, чи індексація поточного елементу робить сітку «безпечною» чи ні, якщо вона безпечна, то рекурсивно викликати функцію для всіх безпечних випадків від 0 до 9. Якщо будь-який рекурсивний виклик повертає true, то потрібно

закінчити цикл і повернути true. Якщо жоден рекурсивний виклик не повертає true, тоді повернути false.

- Якщо немає порожніх комірок, то повернути true.

.

2.5. Греко-латинський квадрат

Греко-латинський квадрат – це два ортогональні латинські квадрати. То ж почати потрібно з того, що таке латинський квадрат.

Латинський квадрат – це таблиця розмірністю $n \times n$ клітинок, складена з n елементів так, що жоден з них не повторюється ні в своєму стовпчику, ні в своєму рядку. «Інакше кажучи, латинський квадрат порядку n – це квадратна таблиця Ф, задана на елементах множини $\mathbb{Q}$, що містить n елементів, кожна стрічка і кожен стовпець якої містить всі n елементів множини $\mathbb{Q}$. Латинський квадрат А порядку n, складений з елементів множини $\mathbb{Q}$, коротко позначається як $A=[\alpha_{ij}]$, де $i, j = 1, 2,\ldots n$, а α_{ij} – елемент множини $\mathbb{Q}$, який знаходиться на перетині i-ї стрічки та j-го стовпчика латинського квадрату.

Два латинських квадрата $L=(l_{ij})$ и $K=(k_{ij})$ n-го порядку називаються ортогональними, якщо всі впорядковані пари (l_{ij},k_{ij}) відрізняються одна від одної» [14].

Для формування квадратів Л. Ейлер сформулював три основних випадки, що залежали від остачі від ділення показника n на 4. Також він запропонував способи побудування пар греко-латинських квадратів для n, що націло діляться на 4, та для непарних n. Найбільша проблема полягала в тому, що Ейлеру не вдалось побудувати такі пари для n = 6 та n = 10. То ж унаслідок цього він виказав гіпотезу про те, що пар ортогональних квадратів для $n = 4t + 2$ не існує, висвітливши це через призму такої поставленої задачі: необхідно було розмістити 36 офіцерів шести різних полків та воєнних звань в каре так, щоб у кожній колоні та ряду був лише один офіцер кожного полку та військового звання.

«Багато задач отримали практичний сенс, а, власне, латинські квадрати знайшли найрізноманітніше застосування у вирішенні доволі різних прикладних задач, таких, як: складання розкладів, вивчення впливу добрив на рослини, планування експериментів у медицині та біології, розподіл ресурсів, черговості ініціювання подій, видачі завдань студентам тощо. А також і в суміжних областях математики, наприклад: в теорії груп та напівгруп, криптографії, комбінаториці, логіці, статистиці, теорії кодів тощо» [14].

З плином часу здійснювалось чимало важливих відкриттів та з'являлось багато нових способів застосування ортогональних квадратів.

Так, наприклад, гіпотеза Л. Ейлера для $n = 6$ була доведена Г. Террі та його братом методом побудови усіх можливих варіантів нормалізованих латинських квадратів (9408 квадратів). Далі розклавши їх на 17 груп вони зрозуміли, що при будь-яких їхніх комбінаціях скласти пару ортогональних квадратів буде неможливо. Тому звідси очевидно, що задача Ейлера про військових розв'язку не мала і була вирішена в сенсі заперечення його наявності.

Серед доступних джерел інформації знайти зрозумілий алгоритм саме пошуку з поверненням для даної задачі виявилось важкою задачею. Тому саме цю проблему було взято для розробки потрібного алгоритму. Процес розробки зайняв чимало часу через постійне покращення алгоритму. То ж детальніше про створення розв'язку буде описано у розділі 3.

ВИСНОВОК ДО РОЗДІЛУ 2

Проблеми, які зазвичай вирішуються за допомогою пошуку з поверненням, мають наступну спільну властивість: ці проблеми можна вирішити лише спробувавши всі можливі конфігурації, і кожна конфігурація випробовується лише один раз.

Алгоритм пошуку з поверненням може застосовуватись до великої кількості різноманітних задач. У 2 розділі було наведено приклади

найпопулярніших з них: задача про вісім ферзів, задача про лицарський турнір, судоку тощо. Усі вони мають доступні програмні реалізації.

Проблемою, що не має зрозумілих та ефективних розв’язків на основі алгоритму з поверненням є задача про створення греко-латинського квадрату. Деякі історичні відомості дають зрозуміти, що ця проблематика вивчалась у всі часи та є досі актуальною. Тому програмна реалізація такого алгоритму необхідна через відсутність можливості спростити обрахунок, залишаючи його для комп’ютера.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ З ПОВЕРНЕННЯМ ТА АНАЛІЗ ЙОГО ЕФЕКТИВНОСТІ

3.1. Інструменти реалізації

Для програмної реалізації алгоритму з поверненням було обрано мову Python, адже вона є універсальною, та підходить для рішення великого кола задач. Це мова, що інтерпретується, а не компілюється. Тобто обробка здійснюється построково, написана програма одразу ж перетворюється у машинний код та виконується, на відміну від компілятора, що перетворює у машинний код усю програму без її виконання. Перевагою є те, що реакція є миттєвою, недоліком – те, що помилки у коді виявляються лише при спробі виконати команду чи рядок з допущеною помилкою. «Інтерпретація коду відбувається повільніше, ніж запуск скомпільованого коду, через те що інтерпретатор повинен аналізувати кожну інструкцію у програмі кожного разу, коли вона виконується, а потім виконувати потрібну дію, в той час як скомпільований код просто виконує фіксовані дії, визначені під час компіляції. Цей аналіз під час виконання відомий як «додаткові витрати інтерпретації». Доступ до змінних в інтерпретованих програм також повільніший, тому що операція зв'язування ідентифікаторів з місцями зберігання повторюється під час виконання, тоді як компілятором виконується один раз під час компіляції» [12].

Середовищем, де буде написана програма та де вона буде працювати було обрано PyCharm – інтегроване середовище для професійної розробки на мові Python від компанії JetBrains, що дає можливість писати, аналізувати та відлагоджувати код [13].

У середовища PyCharm є ряд наступних можливостей:

- Статичний аналіз коду, підсвічування синтаксису
- Навігація серед проектів та коду, наочна файлова структура
- Рефакторинг – можливість перейменування, введення змінної, введення константи тощо
- Вбудований відлагоджувач

- Підтримка систем контролю версій
- Тощо.

Завдяки функціоналу даного середовища можна побачити, які існують взаємозв’язки між методами та скільки часу потрібно на виконання кожного з них.

3.2. Реалізація алгоритму та аналіз ефективності його роботи

Лістинг програми та коментарі до коду можна переглянути у *Додатку А*.

Алгоритм було побудовано опираючись на алгоритм для побудови латинського квадрату розмірності n з курсової роботи за 3 курс. Було взято за основу розуміння, що задача побудови греко-латинського квадрату зводиться до задачі побудови двох латинських квадратів та їх коректного об’єднання. Новий побудований греко-латинський квадрат повинен відповідати додатковій умові – відсутність повтору значень.

Створений алгоритм поділяється на 3 частини.

Перша частина алгоритму – це побудова першого латинського квадрату розмірності $n \times n$, будемо називати його «первинний квадрат». Друга частина алгоритму – це побудова другого латинського квадрату, будемо називати його «вторинний квадрат», який в парі з першим буде утворювати греко-латинський квадрат. Та третя частина – це накладання первинного та вторинного квадратів один на одного для отримання «результуючого» греко-латинського квадрату.

В першій частині алгоритму будується випадковий існуючий (правильний) латинський квадрат розмірності $n \times n$. Ця частина реалізована в двох варіантах, де перший – побудова першого рядку випадковим чином, а інші рядки є зміщенням попереднього ряду на одне значення вліво; другий – це побудова первинного квадрату за допомогою перебору з поверненням.

Друга частина алгоритму реалізовує в собі перебір з поверненням за наступним принципом: вторинний квадрат представляється у вигляді двовимірного масиву $n \times n$, елементами якого є масив розміру n. Кожне

додатне значення цього масиву відповідає за те, на якому кроці та яке значення було записане у відповідну клітинку, та кожне від'ємне значення відповідає за значення, які не можливо записати у відповідну клітинку та на якому кроці додалось це обмеження; нульове значення показує, що відповідне значення можна записати у відповідну клітинку на відповідному кроці. На кожному кроці перевіряється можливість вписати у поточну клітинку будь-яке значення. Якщо це можливо, тоді це найменше значення записується, та додається обмеження на запис цього значення у всі клітинки нижче та справа від поточної, а також вводяться обмеження, які задовольняють означенню греко-латинського квадрату. Якщо ж вписати значення у поточну клітинку не є можливим, тоді відміняються усі обмеження, які вступили в силу на попередньому кроці, відбувається повернення на попередній крок та додається обмеження на запис у поточну клітинку значення, що було обране минулий раз на цьому кроці. Цей процес повторюється поки не буде закінчений $n \times n$ крок, або будуть додані обмеження на запис у першу клітинку усіх n можливих значень. Це буде означати, що для такого первинного квадрату не існує вторинного, який при накладанні утворюватиме з першим греко-латинський квадрат.

Якщо останній крок успішно завершився, тоді вторинний квадрат накладається на первинний та утворює результуючий квадрат розмірності $n \times n$.

Дослідимо *ефективність* алгоритму пошуку з поверненням для побудови греко-латинського квадрату у порівнянні з методом повного перебору.

Якщо використовується метод повного перебору, то задача побудови усіх латинських квадратів розмірності n є досить складною, адже кількість усіх квадратів, які потрібно пребрати та перевірити дорівнює n^{n^2}. Таким чином вже для досить малих n це число вражає.

Наприклад: при n = 9

n^{n^2} = 1.966270504755529136180759085269 1e+77

Звісно, що далеко не всі квадрати будуть латинськими, але їх кількість також є досить вагомою. Найкраще оцінити кількість латинських квадратів $L(n)$ можна за допомогою формули Ван Лінта – Вілсона [15] :

$$\prod_{k=1}^{n}(k!)^{n/k} \geq L(n) \geq \frac{(n!)^{2n}}{n^{n^2}} \qquad (3.3.1)$$

Також для n від 1 до 11 кількості латинських квадратів є порахованими. Відомо, що число $R(n)$ нормалізованих (тобто таких, що у першому рядку мають значення розташовані у порядку зростання) латинських квадратів n-го порядку в $n!\,(n-1)!$ раз менше, ніж кількість звичайних латинських квадратів. Точні значення цих величин наразі відомі для n *від* 1 *до* 11.

Очевидно, що для побудови навіть такої кількості латинських квадратів, потрібні великі обсяги пам’яті та часу.

Кількість латинських квадратів

n	*нормалізовані латинські квадрати*	*всі латинські квадрати n*
1	1	1
2	1	2
3	1	12
4	4	576
5	56	161280
6	9408	812851200
7	16942080	61479419904000
8	535281401856	108776032459082956800
9	377597570964258816	5524751496156892842531225600
10	7580721483160132811489280	9982437658213039871725064756920320000
11	536393777327737129811967354077184O	776966836171770144107444346734230682311065600000

Рисунок 3.3.1 Кількість нормалізованих та звичайних латинських квадратів для n від 1 до 11

Складність алгоритму, який використовує принцип повного перебору, є експоненціальною та її можна оцінити як $O(n^{n^2})$. Враховуючи таку складність, навіть при n = 9 алгоритм буде працювати надзвичайно довго.

В свою чергу алгоритм побудований з принципом перебору з поверненням є суттєво простішим та його складність можна оцінити як $O(n^{n\times\ln(n)})$. Таким чином можна зробити висновок, що застосування перебору з поверненням є дуже ефективним для зниження складності алгоритму.

В наступній таблиці наведено час роботи алгоритму для випадків n = 1, 3, 4, 5, 7, 9.

Таблиця 3.3.1 – Час роботи алгоритму в залежності від розмірності квадрату

Розмірність квадрату	Час виконання алгоритму в секундах
1	0.0(дуже швидко рахує)
3	0.0(дуже швидко рахує)
4	0.0(дуже швидко рахує)
5	0.0020360946655273438
7	1.1406219005584717
9	1.7344017028808594

Ці результати коливаються в залежності від латинського квадрату, що був взятий за основу греко-латинського квадрату. Часто трапляються випадки, коли для початкового квадрату не існує пари, яка утворює з ним греко-латинський квадрат. При таких обставинах, алгоритм працює максимально довго. Для випадку $n = 11$ алгоритм працював протягом усього дня, але не виконав свої обрахунки навіть на 10%. Тому можна зробити висновок, що вже при $n = 11$ алгоритм працює дуже довго. Звісно при використанні більш потужних ЕОМ цей час буе значно меншим, але при черговому збільшені розмірності квадрату, проблема знову стане актуальною.

ВИСНОВКИ ДО РОЗДІЛУ 3

Підсумовуючи, можна однозначно стверджувати, що алгоритми перебору з поверненням в рази скорочують ресурси необхідні для розв’язку задач, що мають подібну до задачі побудови греко-латинського квадрату складність.

Завдяки сучасним інструментам розробки, таких як PyCharm, можливо створити будь-який алгоритм будь-якої складності, переглянути характеристики процесу виконання, за необхідності – відредагувати код. Проте ще кращий результат можна отримати на потужніших ЕОМ в лабораторних умовах та за допомогою застосування спеціальних обчислювальних методів.

Метод пошуку з поверненням дозволяє зробити алгоритм ефективним, при цьому з нижчою складністю, ніж, наприклад, у методі повного перебору. Це робить його гарним інструментом для реалізації подібних алгоритмів.

ВИСНОВКИ

У бакалаврській роботі було досліджено алгоритми та їх характеристики, зокрема історичний шлях терміну «алгоритм», його основні характеристики, такі, як: властивості, форми представлення, базові структури, етапи вирішення, складність та ефективність. Найважчою для дослідження та опису є тема складності алгоритмів та оцінка їх ефективності, при цьому це одні з найважливіших характеристик, адже при різноманітному застосуванні алгоритмів дуже важливо вміти правильно оцінювати їх.

Додатково було продемонстровано на реальних прикладах застосування різних видів алгоритмів, їх особливостей. Проблеми, які зазвичай вирішуються за допомогою пошуку з поверненням, мають наступну спільну властивість: ці проблеми можна однозначно вирішити, спробувавши всі можливі конфігурації, і кожна конфігурація випробовується лише один раз. Задачею, що не має ефективних розв’язків на основі алгоритму з поверненням є задача про створення греко-латинського квадрату. Деякі історичні відомості дають зрозуміти, що ця проблематика вивчалась у всі часи та є досі актуальною.

Програмно реалізовано алгоритм пошуку з поверненням у середовищі PyCharm мовою програмування Python на прикладі задачі побудови греко-латинського квадрату. Також відбувся аналіз написаного алгоритму, його порівняння з методом повного перебору, коли перевіряються усі варіанти без винятку. Даний аналіз показав, що складність написаного алгоритму, який використовує принцип повного перебору, є експоненціальною та її можна оцінити як $O(n^{n^2})$. Тому, для зменшення часу який комп’ютер витрачає для виконання подібних задач, доцільним є перехід від алгоритмів повного перебору, до перебору з поверненням. В ході роботи було виявлено, що ефективність алгоритму та час його виконання сильно залежить від обраної реалізації; оптимальні методи та умови допомагають зберегти часові та ресурси комп‘ютера.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Пермякова О. Время вернуться домой, 2009. URL: https://letopisi.org/index.php/%D0%98%D1%81%D1%82%D0%BE%D1%80%D0%B8%D1%8F_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0 (Дата звернення: 28.05.2021)
2. Erickson J. Algorithms, paperback, 2019. 472p. URL: https://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf (Last accessed: 28.05.2021)
3. Movable type. Wikipedia, the free encyclopedia, 2021. URL: https://en.wikipedia.org/wiki/Movable_type (Last accessed: 28.05.2021)
4. Kowalkiewicz M. How did we get here? The story of algorithms. 2019. URL: https://towardsdatascience.com/how-did-we-get-here-the-story-of-algorithms-9ee186ba2a07 (Last accessed: 28.05.2021)
5. Шилов В. Удивительная история информатики и автоматики. Раздел: К вопросу об алгоритмах. Москва, 2013. 216 с.
6. Алгоритм. История возникновения термина. Формальные признаки алгоритма. Брестский Государственный Университет имени А.С. Пушкина. Брест, 2015. URL: https://studfile.net/preview/1725803/ (Дата звернення: 28.05.2021)
7. Малярчук С. М. Основи інформатики у визначеннях, таблицях і схемах: Довідково-навчальний посібник. «Ранок», 2007. 112 с.
8. Формы представления алгоритмов. Брестский Государственный Университет имени А.С. Пушкина. Брест, 2015. https://studfile.net/preview/1725803/page:2/ (Дата звернення: 1.06.2021)
9. Базові алгоритмічні структури. Національний технічний університет України «Київський політехнічний інститут». Київ, 2016. URL: https://studfile.net/preview/5994725/page:4/ (Дата звернення: 1.06.2021)

10. Prim's algorithm. Wikipedia, the free encyclopedia, 2021. URL: https://en.wikipedia.org/wiki/Prim%27s_algorithm (Last accessed: 02.06.2021)

11. Оцінка ефективності алгоритму. Національний технічний університет України «Київський політехнічний інститут». Київ, 2016. URL: https://studfile.net/preview/5994725/page:2/ (Дата звернення: 3.06.2021)

12. Інтерпретатор. Матеріал з Вікіпедії — вільної енциклопедії, 2020. URL:https://uk.wikipedia.org/wiki/%D0%86%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80 (Last accessed: 05.06.2021)

13. PyCharm. Wikipedia, the free encyclopedia, 2021. URL: https://en.wikipedia.org/wiki/PyCharm (Last accessed: 05.06.2021)

14. Андрійченко К. А., Вєтров О. С. Аналіз алгоритму «Перебір з поверненням» на прикладі задачі побудови латинського квадрату Матеріали всеукраїнської науково-практичної конференції для студентів, аспірантів та молодих вчених (29 квітня 2020 р.). Вінниця, 2020. С. 161-163.

15. van Lint J. H., Wilson R. M. A Course in Combinatorics. — Cambridge University Press, 1992.

ДОДАТКИ

ДОДАТОК А

Лістинг програмної реалізації алгоритму пошуку з поверненням на прикладі задачі побудови греко-латинського квадрату

```
1. import random
2. import time
3.
4. """Метод, який створює масив розмірності n*n"""
5. def createSquare(size):
6.     value = [[0 for i in range(size)] for i in range(size)]
7.     return value
8.
9.
10.     """Метод, який перетворює масив зроблених ходів у латинський
    квадрат"""
11.     def formatSquare(size, square):
12.         value = createSquare(size)
13.         for i in range(size):
14.             for j in range(size):
15.                 for k in range(size):
16.                     if square[i][j][k] > 0:
17.                         value[i][j] = k + 1
18.         return value
19.
20.
21.     """Метод, який відображає завершений латинський чи греко-
    латинський квадрат"""
22.     def printSquare(size, square):
23.         for i in range(size):
24.             print(square[i])
25.
26.
27.     """Метод, який відображає незавершений(проміжний) латинський
    квадрат"""
28.     def printUnReadySquare(size, square):
29.         value = formatSquare(size, square)
30.         printSquare(value, size)
```

```
31.         return value
32.
33.
34.     """Метод, який виключає повторення знаків у рядках та стовпцях
    після здійснення кроку"""
35.     def stepForward(size, square, turn, x, y, z):
36.         for i in range(size):
37.             if i != y and square[x][i][z] == 0:
38.                 square[x][i][z] = (turn * -1)
39.             if i != x and square[i][y][z] == 0:
40.                 square[i][y][z] = (turn * -1)
41.
42.
43.     """Метод, який повертає стан латинського квадрату на крок
    назад та виключає можливість здійснення невалідного кроку"""
44.     def stepBack(size, square, turn):
45.         for i in range(size):
46.             for j in range(size):
47.                 for k in range(size):
48.                     if turn != 1:
49.                         if square[i][j][k] <= turn * -1 or turn <
    square[i][j][k] < size ** 2 + 1:
50.                             square[i][j][k] = 0
51.                         if square[i][j][k] == size ** 2 + 1:
52.                             square[i][j][k] = size ** 2 + 1
53.                         if square[i][j][k] == turn:
54.                             square[i][j][k] = turn * -1 + 1
55.                     if turn == 1:
56.                         if square[i][j][k] == 1:
57.                             square[i][j][k] = size ** 2 + 1
58.                         if square[i][j][k] <= turn * -1:
59.                             square[i][j][k] = 0
60.
61.
62.     """Метод, який виключає можливість повторення пари значень у
    результуючому греко-латинському квадраті"""
63.     def stepForwardGraeco(size, square1, square2, turn, x, y, z):
64.         """
65.         print("stepForward")
66.         value = printUnReadySquare(size, square2)
```

```
67.         """
68.         stepForward(size, square2, turn, x, y, z)
69.         for i in range(size):
70.             if i != x:
71.                 for j in range(size):
72.                     if j != y:
73.                         if square1[i][j] == square1[x][y] and
    square2[i][j][z] == 0:
74.                             square2[i][j][z] = (turn * -1)
75.
76.
77.     """Метод, який повертає стан греко-латинського квадрату на
    крок назад та виключає можливість здійснення невалідного кроку"""
78.     def stepBackGraeco(size, square, turn):
79.         """
80.         print("stepBack")
81.         print("Before stepBack")
82.         valueBefore = printUnReadySquare(size, square2)
83.         square2Copy = [[[0 for i in range(size)] for i in
    range(size)] for i in range(size)]
84.         for i in range(size):
85.             for j in range(size):
86.                 for k in range(size):
87.                     square2Copy[i][j][k] = square[i][j][k]
88.         """
89.         stepBack(size, square, turn)
90.         """
91.         print("After stepBack")
92.         valueAfter = printUnReadySquare(size, square2)
93.         if valueAfter[0][0] < valueBefore[0][0]:
94.             print("before Error")
95.             printSquare(size, square2Copy)
96.             print("after Error")
97.             printSquare(size, square2)
98.         """
99.
100.
101.    """Метод, який генерує випадковий латинський квадрат,
    розмірності n*n, шляхом здвигу попереднього рядку на 1 символ"""
102.    def genLatinSquare(size):
```

```
103.         square = [[0] * size for i in range(size)]
104.         StartList = []
105.         for i in range(size):
106.             StartList.append(i + 1)
107.         for i in range(size):
108.             length = len(StartList)
109.             square[0][i] = StartList.pop(random.randint(0, length
    - 1))
110.         for i in range(1, size):
111.             for j in range(size):
112.                 square[i][j] = square[0][(j + i) % size]
113.         return square
114.
115.
116.     """Метод, який генерує випадковий латинський квадрат,
    розмірності n*n, методом перебору з поверненням"""
117.     def genLatinSquareWithBacktracking(size):
118.         StartList = []
119.         for i in range(size):
120.             StartList.append(i + 1)
121.         square = [[[0 for i in range(size)] for i in range(size)]
    for i in range(size)]
122.         turn = 0
123.         while turn < size:
124.             length = len(StartList)
125.             step = StartList.pop(random.randint(0, length - 1))
126.             square[0][turn][step - 1] = turn + 1
127.             stepForward(size, square, turn + 1, 0, turn, step - 1)
128.             turn = turn + 1
129.         while turn < size ** 2:
130.             """print(turn + 1)"""
131.             didTurn = False
132.             x = int(turn / size)
133.             y = turn % size
134.             for i in range(size):
135.                 if square[x][y][i] != 0:
136.                     didTurn = False
137.                 if square[x][y][i] == 0:
138.                     square[x][y][i] = turn + 1
139.                     stepForward(size, square, turn + 1, x, y, i)
```

```
140.                     didTurn = True
141.                     break
142.
143.             if didTurn:
144.                 turn = turn + 1
145.             if not didTurn:
146.                 turn = abs(min(square[x][y])) - 1
147.                 stepBack(size, square, turn + 1)
148.         value = formatSquare(size, square)
149.         return value
150.
151.
152.     """Метод, який генерує греко-латинський квадрат розмірності
    n*n"""
153.     def genGraecoLatinSquare(size):
154.         square1 = genLatinSquare(size)
155.         """square1 = genLatinSquareWithBacktracking(size)"""
156.         """printSquare(size, square1)"""
157.         square2 = [[[0 for i in range(size)] for i in range(size)]
    for i in range(size)]
158.         turn = 0
159.         while turn < size ** 2:
160.             """print(turn + 1)"""
161.             didTurn = False
162.             x = int(turn / size)
163.             y = turn % size
164.             for i in range(size):
165.                 if square2[x][y][i] != 0:
166.                     didTurn = False
167.                 if square2[x][y][i] == 0:
168.                     square2[x][y][i] = turn + 1
169.                     stepForwardGraeco(size, square1, square2, turn
    + 1, x, y, i)
170.                     didTurn = True
171.                     break
172.
173.             if didTurn:
174.                 turn = turn + 1
175.             if not didTurn:
176.                 turn = abs(min(square2[x][y])) - 1
```

```
177.                 stepBackGraeco(size, square2, turn + 1)
178.         """
179.         print("First Latin Square")
180.         printSquare(size, square1)
181.         print("Second Latin Square")
182.         printUnReadySquare(size, square2)
183.         """
184.         for i in range(size):
185.             for j in range(size):
186.                 for k in range(size):
187.                     if 0 < square2[i][j][k] < size ** 2 + 1:
188.                         square2[i][j] = [square1[i][j], k + 1]
189.                         break
190.         if square2[0][0].count(-(size*size)) == size:
191.             print("Неможливо побудувати Греко-Латинський квадрати
    з таким початковим латинським квадратом: ")
192.             return square1
193.         print("Побудовано такий Греко-Латинський квадрат: ")
194.         return square2
195.
196.
197.     """Генерується та виводиться Греко-Латинський квадрат
    розмірності n*n"""
198.
199.     n = 9
200.     start = time.time()
201.     graecoLatinSquare = genGraecoLatinSquare(n)
202.     end = time.time()
203.     printSquare(n, graecoLatinSquare)
204.     print("Було витрачено: " + str(end-start) + " секунд")
```