МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики

Кафедра прикладної математики

Магістерська робота

Оцінка можливостей ChatGPT у розв’язанні задач прикладної математики

Виконав: студент групи ПМПМ-22
спеціальності
113 - прикладна математика
Тарнавський П. В.
(прізвище та ініціали)
Керівник доц. Ящук Ю. О.
(прізвище та ініціали)
Рецензент____________________________
(прізвище та ініціали)

Львів - 2023

ЗМІСТ

ВСТУП

Останнім часом став популярний штучний інтелект ChatGPT [1], в основі цього прототипу лежить модель OpenAI GPT-3.5, покращена версія GPT-3. Його використовують у різноманітних сферах, від написання різних текстів до генерації коду. Відповідно багато учнів та студентів використовують його для розв’язування різноманітних завдань, в тому числі і домашніх завдань. Отож деякі викладачі задумуються над тим що студенти можуть використовувати його для порушення академічної доброчесності [2]. Для того аби не допустити це необхідно знати можливості цього штучного інтелекту у розв’язанні різних задач прикладної математики. Оскільки, ChatGPT доволі нещодавно з’явився у публічному доступі, то небагато досліджень було опубліковано, одним із таких є огляд можливостей новішої версії GPT-4 [3], у цій роботі також показано, для декількох різних областей, можливості ChatGPT для порівняння їх між собою. Отже, постає **актуальність проблеми** нашого дослідження: розуміння можливостей цього штучного інтелекту.

Вирішення цієї проблеми визначає **мету дослідження** – експериментально оцінити можливості та виявити проблеми ChatGPT у різних типах задач прикладної математики, наприклад алгоритмічні задачі та задачі чисельних методів.

Об’єкт дослідження – штучний інтелект ChatGPT.

Предмет дослідження – застосування штучного інтелекту ChatGPT у задачах прикладної математики.

Згідно з проблемою, об’єктом та метою ставляться такі **завдання**:

1. розглянути можливості ChatGPT у алгоритмічних задачах
2. розглянути можливості ChatGPT у задачах чисельних методів

3. згенерувати наукову роботу, у сфері задач прикладної математики, за допомогою ChatGPT

Позаяк різних задач прикладної математики є багато, то необхідно обмежити ті які будемо розглядати, але у той же ж час залишити вибірку задач достатньо репрезентативною. Критерієм відбору були популярність методу чи задачі, для її оцінки використовували різну літературу.

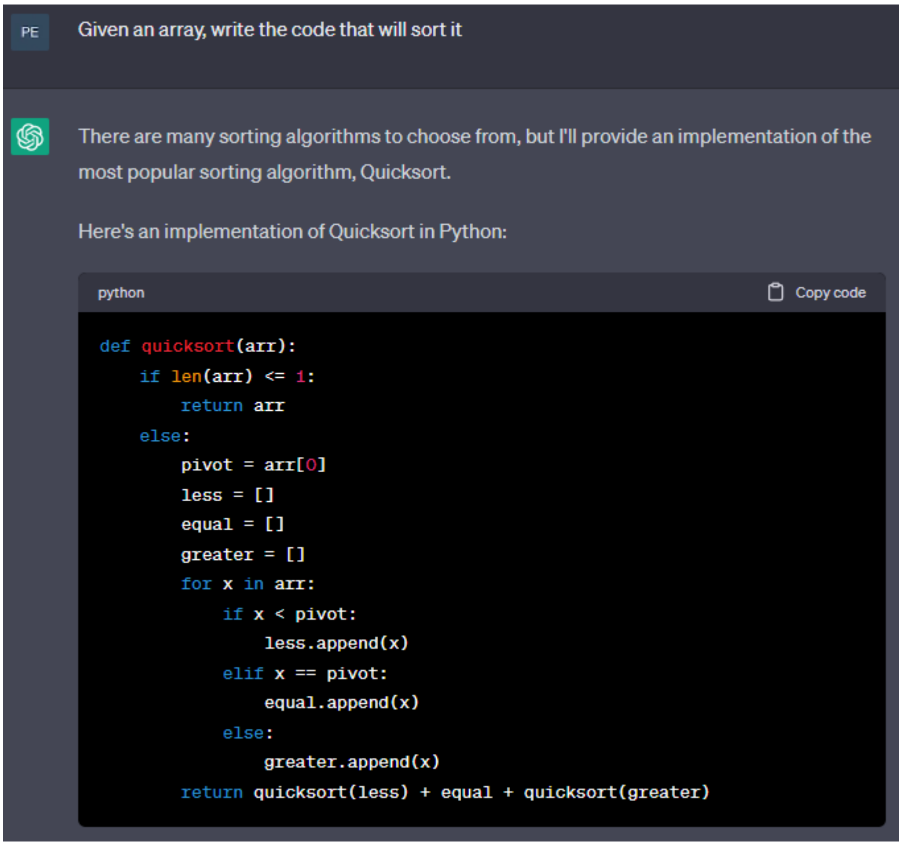
Для дослідження можливостей Chat GPT необхідно перевірити його для розв’язання задач різноманітної складності: легкі, складні та задачі середньої складності. Цей штучний інтелект повинен описати розв’язок та згенерувати код який буде розв'язувати поставлену задачу. Після отриманих результатів, можна вибрати методи для наукової роботи, яку генеруватиме ChatGPT.

РОЗДІЛ 1

ЗАСТОСУВАННЯ CHATGPT ДЛЯ РОЗВ'ЯЗУВАННЯ АЛГОРИТМІЧНИХ ЗАДАЧ

Алгоритмічні задачі — задачі в яких необхідно вигадати алгоритм та реалізувати його, цей алгоритм має коректно виконуватись та вкладатись в ліміт часу та пам'яті. Отже, при застосуванні ChatGPT необхідно перевіряти правильність алгоритму та правильність коду.

При взаємодії з ChatGPT будемо йому вказувати які саме вхідні дані є в задачі і що необхідно розв'язати, також вказуватимемо що необхідно згенерувати код.



Given an array, write the code that will sort it

There are many sorting algorithms to choose from, but I'll provide an implementation of the most popular sorting algorithm, Quicksort.

Here's an implementation of Quicksort in Python:

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = []
        equal = []
        greater = []
        for x in arr:
            if x < pivot:
                less.append(x)
            elif x == pivot:
                equal.append(x)
            else:
                greater.append(x)
        return quicksort(less) + equal + quicksort(greater)
```

Рисунок 1. Взаємодія із ChatGPT

1.1. Класичний підручник

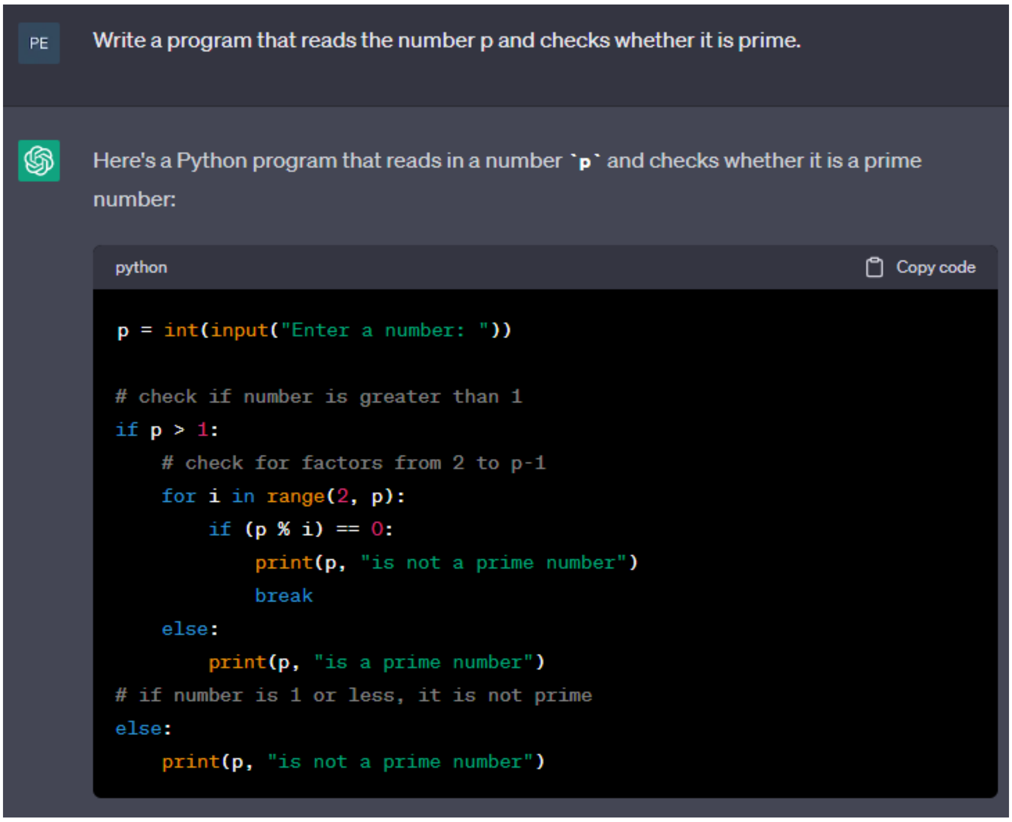
Перш за все необхідно перевірити чи знає і вміє ChatGPT використовувати стандартні алгоритми та підходи. Для цього використали класичний підручник [4] і з нього виділили декілька алгоритмів:

- перевірка на простоту
- пошук найкоротшої відстані в графі
- каркас найменшої ваги
- динамічне програмування
- алгоритм Кнута-Морріса-Пратта
- пошук опуклої оболонки
- множення многочленів

Декілька прикладів будуть наведені в цьому розділі. У додатку А можна переглянути взаємодію із ChatGPT для кожної розглянутої задачі.

1.1.1. Перевірка на простоту

Задача перевірки числа на простоту є стандартною і простою. Розв’язувати цю задачу можна різними способами, наприклад перебрати числа від 2 до p - 1 і перевірити чи p ділиться на це число. Саме такий код і генерує ChatGPT, взаємодія зображена на рисунку 2.



Write a program that reads the number p and checks whether it is prime.

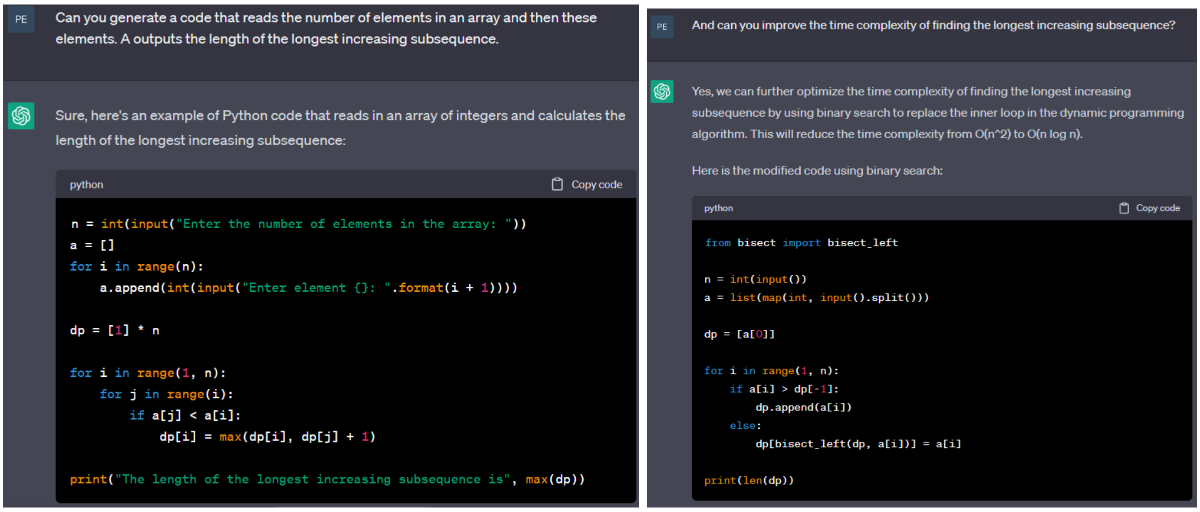
Here's a Python program that reads in a number `p` and checks whether it is a prime number:

```python
p = int(input("Enter a number: "))

# check if number is greater than 1
if p > 1:
    # check for factors from 2 to p-1
    for i in range(2, p):
        if (p % i) == 0:
            print(p, "is not a prime number")
            break
    else:
        print(p, "is a prime number")
# if number is 1 or less, it is not prime
else:
    print(p, "is not a prime number")
```

Рисунок 2. Взаємодія у задачі про перевірку на простоту

1.1.2. Динамічне програмування

Динамічне програмування — це принцип мемоізації (запам’ятовування) результатів підзадач які виникають при розв’язанні основної. Як приклад задачі динамічного програмування візьмемо задачу пошуку довжини найдовшої зростаючої підпослідовності. В ній можна запам’ятовувати яку найбільшу зростаючу підпослідовність ми можемо побудувати якщо останній вибраний елемент буде на позиції i. Для ефективного розв’язку цієї задачі необхідно дещо змінити нашу мемоізацію і тепер ми будемо пам’ятати яке найменше число може бути останнім у зростаючій підпослідовності розміру i. У такому формулюванні нам необхідно опрацьовувати елементи по черзі і оновлювати значення які запам’ятали для цього необхідно використати двійковий пошук. ChatGPT зумів покращити швидкодію своєї першої програми для цієї задачі.



Can you generate a code that reads the number of elements in an array and then these elements. A outputs the length of the longest increasing subsequence.

Sure, here's an example of Python code that reads in an array of integers and calculates the length of the longest increasing subsequence:

```python
n = int(input("Enter the number of elements in the array: "))
a = []
for i in range(n):
    a.append(int(input("Enter element {}: ".format(i + 1))))

dp = [1] * n

for i in range(1, n):
    for j in range(i):
        if a[j] < a[i]:
            dp[i] = max(dp[i], dp[j] + 1)

print("The length of the longest increasing subsequence is", max(dp))
```

And can you improve the time complexity of finding the longest increasing subsequence?

Yes, we can further optimize the time complexity of finding the longest increasing subsequence by using binary search to replace the inner loop in the dynamic programming algorithm. This will reduce the time complexity from O(n^2) to O(n log n).

Here is the modified code using binary search:

```python
from bisect import bisect_left

n = int(input())
a = list(map(int, input().split()))

dp = [a[0]]

for i in range(1, n):
    if a[i] > dp[-1]:
        dp.append(a[i])
    else:
        dp[bisect_left(dp, a[i])] = a[i]

print(len(dp))
```

Рисунок 3. Результати до задачі про найдовшу зростаючу підпослідовність а) повільна програма, б) ефективна програма

1.1.3. Множення многочленів

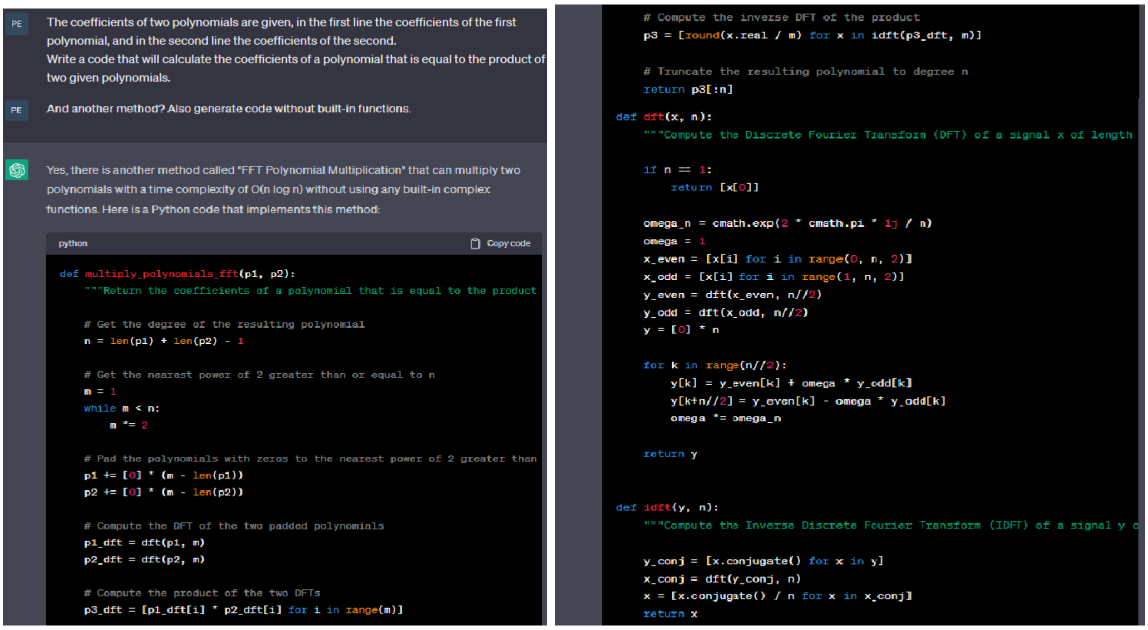
Задача множення двох многочленів полягає у отриманні коефіцієнтів многочлена, що рівний добутку заданих двох многочленів. Многочлени задані як масив коефіцієнтів.

Для ефективного розв'язання цієї задачі необхідно виконати дискретне швидке перетворення Фур’є. Це метод що перетворює набір чисел $a_0, a_1, ..., a_{n-1}$ в набір чисел $b_0, b_1, ..., b_{n-1}$, такий що

$$b_i = \sum_{j=0}^{n-1} a_j \varepsilon^{ij}.$$

Для того аби перемножити коефіцієнти двох многочленів P та Q необхідно їх коефіцієнти перетворити прямим швидким перетворенням Фур’є, після чого отримані коефіцієнти перемножити послідовно, тобто перший з першим, другий з другим. Отриманий таким чином масив володіти необхідно перетворити оберненим перетворенням і так ми отримаємо коефіцієнти многочлена, що дорівнює добутку двох заданих.

ChatGPT спочатку запропонував інший метод який теж є коректним, тому запитався чи він знає ще якийсь метод і після цього він вказав саме той алгоритм який ми і очікували.



The coefficients of two polynomials are given, in the first line the coefficients of the first polynomial, and in the second line the coefficients of the second.
Write a code that will calculate the coefficients of a polynomial that is equal to the product of two given polynomials.

And another method? Also generate code without built-in functions.

Yes, there is another method called "FFT Polynomial Multiplication" that can multiply two polynomials with a time complexity of O(n log n) without using any built-in complex functions. Here is a Python code that implements this method:

```python
def multiply_polynomials_fft(p1, p2):
    """Return the coefficients of a polynomial that is equal to the product

    # Get the degree of the resulting polynomial
    n = len(p1) + len(p2) - 1

    # Get the nearest power of 2 greater than or equal to n
    m = 1
    while m < n:
        m *= 2

    # Pad the polynomials with zeros to the nearest power of 2 greater than
    p1 += [0] * (m - len(p1))
    p2 += [0] * (m - len(p2))

    # Compute the DFT of the two padded polynomials
    p1_dft = dft(p1, m)
    p2_dft = dft(p2, m)

    # Compute the product of the two DFTs
    p3_dft = [p1_dft[i] * p2_dft[i] for i in range(m)]
```

```python
    # Compute the inverse DFT of the product
    p3 = [round(x.real / m) for x in idft(p3_dft, m)]

    # Truncate the resulting polynomial to degree n
    return p3[:n]

def dft(x, n):
    """Compute the Discrete Fourier Transform (DFT) of a signal x of length

    if n == 1:
        return [x[0]]

    omega_n = cmath.exp(2 * cmath.pi * 1j / n)
    omega = 1
    x_even = [x[i] for i in range(0, n, 2)]
    x_odd = [x[i] for i in range(1, n, 2)]
    y_even = dft(x_even, n//2)
    y_odd = dft(x_odd, n//2)
    y = [0] * n

    for k in range(n//2):
        y[k] = y_even[k] + omega * y_odd[k]
        y[k+n//2] = y_even[k] - omega * y_odd[k]
        omega *= omega_n

    return y


def idft(y, n):
    """Compute the Inverse Discrete Fourier Transform (IDFT) of a signal y

    y_conj = [x.conjugate() for x in y]
    x_conj = dft(y_conj, n)
    x = [x.conjugate() / n for x in x_conj]
    return x
```

Рисунок 4. Відповідь до задачі про множення многочленів.

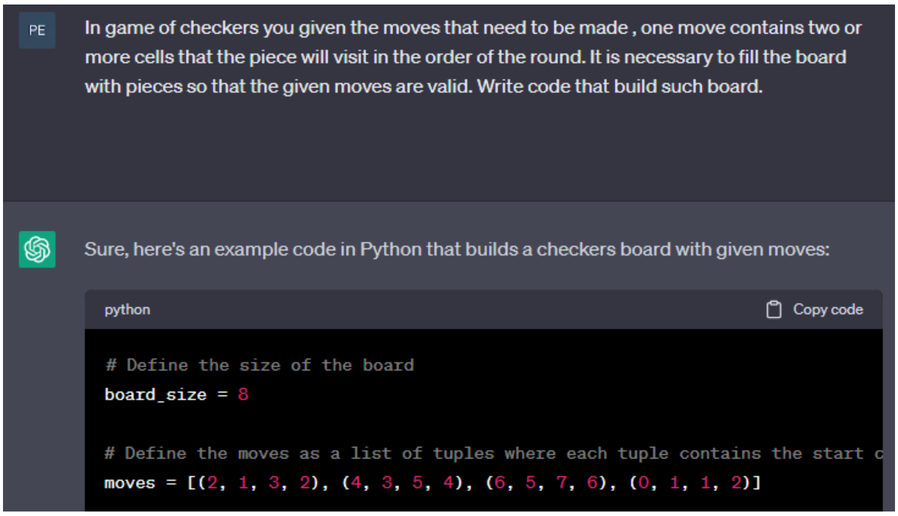
Отже, ChatGPT досить просто генерує код для задач що використовують стандартні підходи.

1.2. Задачі зі світового фіналу олімпіади з програмування

Подальшим кроком необхідно розглянути складні задачі та подивитись як ChatGPT з ними впорається. Такими задачами можна вважати задачі з світового фіналу командної олімпіади ACM ICPC [5]. Було вибрано змагання 2018-го та 2019-го років і з них розглянути задачі, які не були розв’язані жодною командою під час змагання.

Розглянемо одну із цих задач, взаємодію для інших задач можна переглянути в додатку А.

Задачі про шашки, де необхідно знайти розташування шашок на полі так аби можна було виконати задані ходи. Можемо помітити на рисунку 5, що код який генерує ChatGPT є хибним оскільки він генерує список можливих ходів в якому міститься 4 елементи, що є неправильно. Звісно це не єдина помилка в коді який згенерував ChatGPT, також код містив зовсім неправильний підхід до розв’язання цієї задачі.



In game of checkers you given the moves that need to be made , one move contains two or more cells that the piece will visit in the order of the round. It is necessary to fill the board with pieces so that the given moves are valid. Write code that build such board.

Sure, here's an example code in Python that builds a checkers board with given moves:

```python
# Define the size of the board
board_size = 8

# Define the moves as a list of tuples where each tuple contains the start c
moves = [(2, 1, 3, 2), (4, 3, 5, 4), (6, 5, 7, 6), (0, 1, 1, 2)]
```

Рисунок 5. Взаємодія у задачі про шашки

Жодну з вибраних задач ChatGPT не розв’язав, отже можна вважати, що він не здатний розв’язувати задачі в яких необхідно об’єднати декілька ідей разом або використати різні алгоритми в одній задачі.

1.3. Платформа Sphere Online Judge

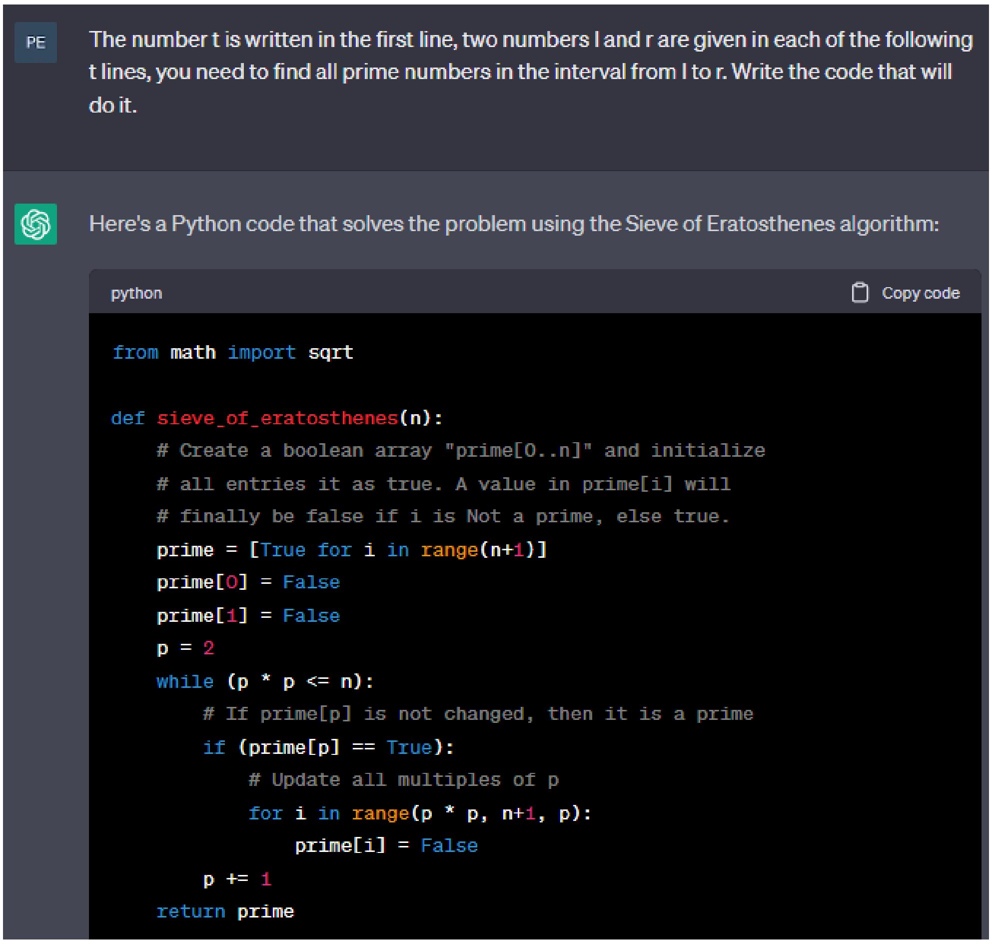
Для огляду можливостей ChatGPT додатково використали сайт з алгоритмічними задачами. Sphere Online Judge [6] — платформа з великою кількістю задач та автоматичним тестуванням розв’язків користувачів. На цій платформі є близько 13000 різноманітних задач та понад 300000 зареєстрованих користувачів. Цей сайт існує з 2012-го року і є достатньо популярним, тому кількість людей, що розв’язали задачу є коректною оцінкою складності задачі. Саме цією шкалою будемо користуватись.

Задачі можуть бути різноманітного типу і деякі задачі неможливо коректно задати ChatGPT для розв'язання, тому будемо розглядати тільки задачі в яких немає малюнків, також задачі в яких умова має великий обсяг.

1.3.1. Алгоритм Ератосфена

Вибрали задачу у якій необхідно знайти усі прості числа на проміжку від l до r.

Один із можливих підходів це кожне число на проміжку перевіряти на простоту, проте такий метод є повільним. Пришвидшити пошук можна використавши алгоритм Ератосфена, він полягає у тому аби для кожного простого позначати усі числа відмінні від нього і які діляться на це просте число. Таким чином усі числа які мають хоча б один простий дільник і є не простими будуть позначені. А отже нам цікаво тільки не позначені числа, проте серед них є число 1, яке можна позначити початково.



The number t is written in the first line, two numbers l and r are given in each of the following t lines, you need to find all prime numbers in the interval from l to r. Write the code that will do it.

Here's a Python code that solves the problem using the Sieve of Eratosthenes algorithm:

```python
from math import sqrt

def sieve_of_eratosthenes(n):
    # Create a boolean array "prime[0..n]" and initialize
    # all entries it as true. A value in prime[i] will
    # finally be false if i is Not a prime, else true.
    prime = [True for i in range(n+1)]
    prime[0] = False
    prime[1] = False
    p = 2
    while (p * p <= n):
        # If prime[p] is not changed, then it is a prime
        if (prime[p] == True):
            # Update all multiples of p
            for i in range(p * p, n+1, p):
                prime[i] = False
        p += 1
    return prime
```

Рисунок 6. Взаємодія для задачі пошуку простих чисел

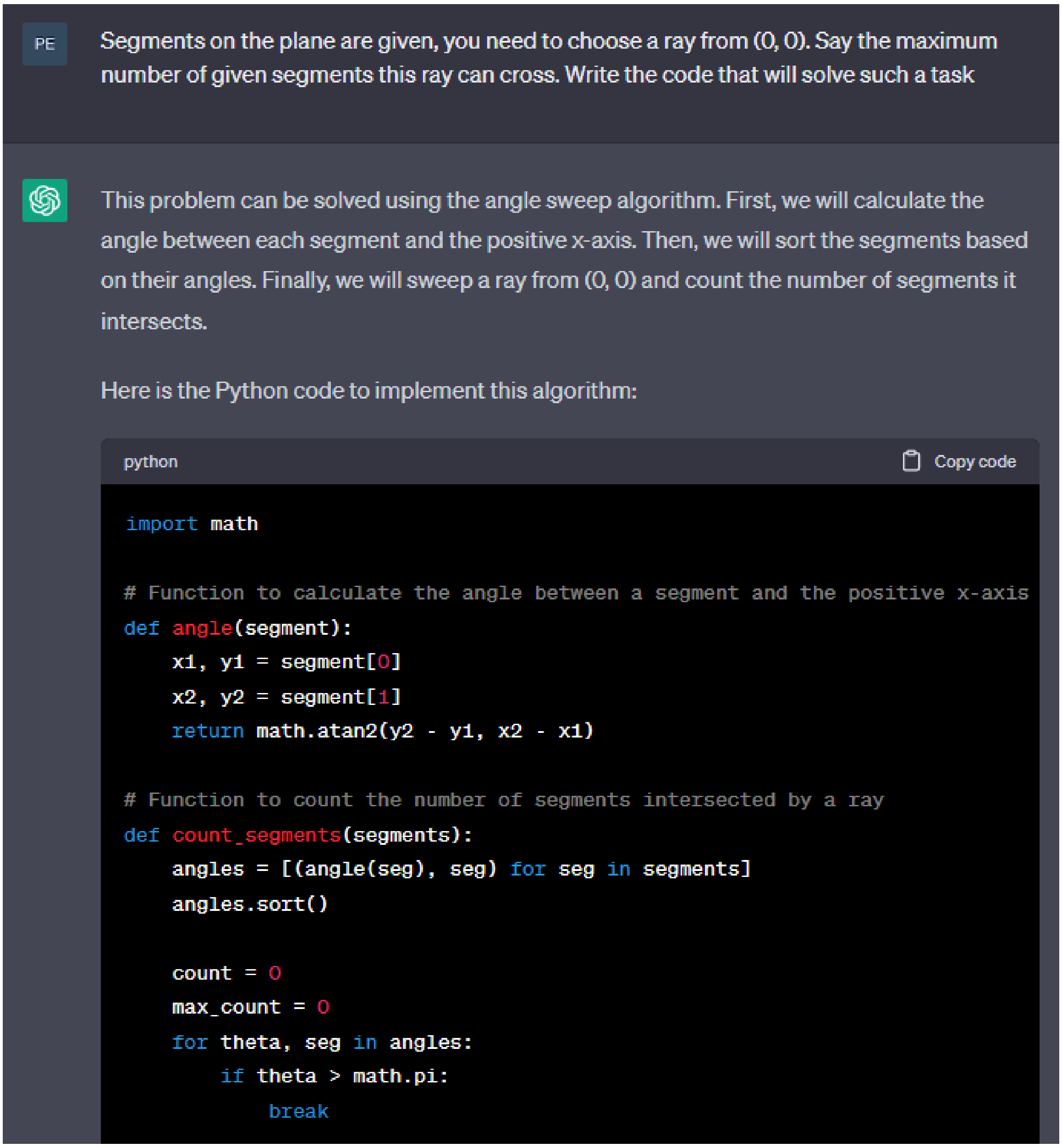
Із цією простою задачею ChatGPT впорався легко і використав ефективний алгоритм – алгоритм Ератосфена.

1.3.2. Замітаюча пряма.

Тепер розглянемо доволі складну задачу про пошук променя з (0, 0) який перетне якомога більше заданих відрізків і нам цікава тільки кількість перетинів.

Для розв'язку цієї задачі нас цікавить проміжок кутів для яких промінь перетне заданий відрізок. Тепер наша задача перетворилась у таку: необхідно знайти значення таке що покривається максимальною кількістю проміжків. Для цього можна використати алгоритм замітаючої прямої, де ми пам’ятаємо скільки зараз проміжків покривають це значення і нас цікавить тільки зміна цього значення. Якщо ми будемо іти зліва на право, тоді коли проміжок відкривається то кількість має збільшитись на 1, а якщо закривається тоді зменшити.

ChatGPT вгадав який метод необхідно використати, проте не зміг правильно описати розв’язок і згенерувати код який би коректно застосовував замітаючу пряму.



Segments on the plane are given, you need to choose a ray from (0, 0). Say the maximum number of given segments this ray can cross. Write the code that will solve such a task

This problem can be solved using the angle sweep algorithm. First, we will calculate the angle between each segment and the positive x-axis. Then, we will sort the segments based on their angles. Finally, we will sweep a ray from (0, 0) and count the number of segments it intersects.

Here is the Python code to implement this algorithm:

```python
import math

# Function to calculate the angle between a segment and the positive x-axis
def angle(segment):
    x1, y1 = segment[0]
    x2, y2 = segment[1]
    return math.atan2(y2 - y1, x2 - x1)

# Function to count the number of segments intersected by a ray
def count_segments(segments):
    angles = [(angle(seg), seg) for seg in segments]
    angles.sort()

    count = 0
    max_count = 0
    for theta, seg in angles:
        if theta > math.pi:
            break
```

Рисунок 7. Взаємодія у геометричній задачі

Можна помітити, що він шукає лише один кут — кут нахилу відповідного відрізка, що звісно ніяк не допоможе у розв’язанні цієї задачі. Помітивши цю проблему, ми йому вказали це та він зміг виправити цю помилку, проте залишались іще інші проблеми. Ми вказали і на них, то він зумів виправити нові помилки, проте поміняв частину коду яка уже була правильною на неправильну.

1.3.3. Числа Фібоначчі

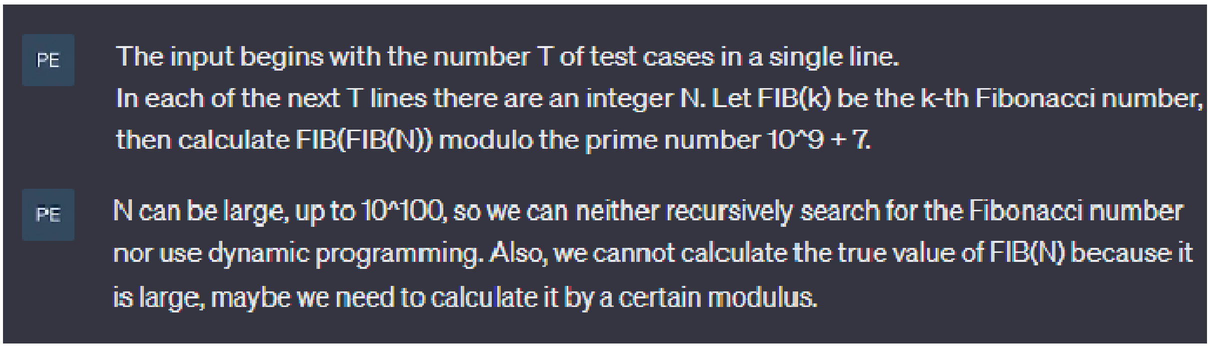
Розглянемо задачу, вона про числа Фібоначчі – послідовність з чисел де кожен елемент цієї послідовності рівний сумі двох попередніх. Нехай FIB(k) – це k-те число Фібоначчі, то в задачі необхідно для n знайти остачу числа FIB(FIB(N)) за модулем 10^9 + 7.

Оскільки число N може бути великим, то використовувати рекурсивний обрахунок числа Фібоначчі або підхід динамічного програмування, обидва, зі складністю O(N) буде повільно. Тому необхідно використати інший підхід, а саме розглянути матрицю переходу від двох послідовних чисел до наступних двох:

$$(1\ 1\ 1\ 0\ ) * (FIB(i)\ FIB(i-1)\ ) = (FIB(i+1)\ FIB(i)\ )$$

Маючи цю матрицю нам треба початковий вектор помножити n разів на цю матрицю, проте можна змінити порядок операцій і порахувати матрицю у степені n та потім її помножити на вектор. Піднести матрицю до степеня можна використовуючи алгоритм двійкового піднесення до степеня.

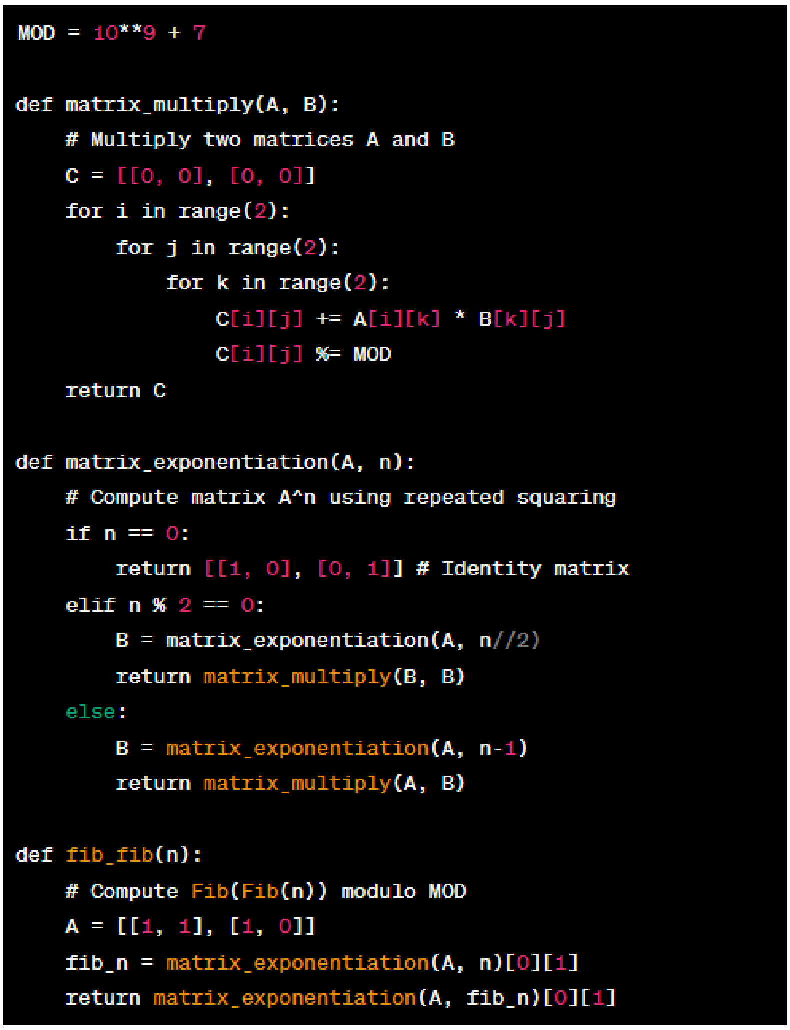
ChatGPT початково запропонував використовувати динамічне програмування, проте я йому додатково вказав на те що N може бути великим. Після чого він впорався із цим завдання, проте допустив невелику помилку у використанні модулів для яких рахував остачу.



The input begins with the number T of test cases in a single line.
In each of the next T lines there are an integer N. Let FIB(k) be the k-th Fibonacci number, then calculate FIB(FIB(N)) modulo the prime number 10^9 + 7.

N can be large, up to 10^100, so we can neither recursively search for the Fibonacci number nor use dynamic programming. Also, we cannot calculate the true value of FIB(N) because it is large, maybe we need to calculate it by a certain modulus.

Рисунок 8. Запити для задачі про числа Фібоначчі



```
MOD = 10**9 + 7

def matrix_multiply(A, B):
    # Multiply two matrices A and B
    C = [[0, 0], [0, 0]]
    for i in range(2):
        for j in range(2):
            for k in range(2):
                C[i][j] += A[i][k] * B[k][j]
                C[i][j] %= MOD
    return C

def matrix_exponentiation(A, n):
    # Compute matrix A^n using repeated squaring
    if n == 0:
        return [[1, 0], [0, 1]] # Identity matrix
    elif n % 2 == 0:
        B = matrix_exponentiation(A, n//2)
        return matrix_multiply(B, B)
    else:
        B = matrix_exponentiation(A, n-1)
        return matrix_multiply(A, B)

def fib_fib(n):
    # Compute Fib(Fib(n)) modulo MOD
    A = [[1, 1], [1, 0]]
    fib_n = matrix_exponentiation(A, n)[0][1]
    return matrix_exponentiation(A, fib_n)[0][1]
```

Рисунок 9. Згенерований код для задачі про числа Фібоначчі

Висновки для алгоритмічних задач

На основі досліджень можна зробити висновок, що використовувати ChatGPT є доцільно для простих алгоритмічних задач, де не потрібно використовувати декілька алгоритмів. Для таких задач він зазвичай генерує цілком правильний детальний опис розв’язку та код який коректно розв’язуватиме задачу. Зі складними задачами цей штучний інтелект допускає помилки, оскільки потрібно зробити багато кроків і помилка в одному впливає на весь розв’язок або ж він не може вигадати з чого почати розв’язувати задачу, тому використання його для таких задач не є доцільним. Серед розглянутих задач середньої складності є такі, які він розв’язує одразу і без жодних проблем, є такі в яких він допустив невеликі неточності, які було легко виправити, а також є задачі, які він не зміг розв’язати. Тому для алгоритмічних задач, середньої складності можна використовувати ChatGPT, проте завжди потрібно перевіряти його відповіді, бо він може помилятись.

Ми розглянули алгоритмічні задачі і переконались, що він вміє розв’язувати відносно прості задачі та генерувати правильний код, який не видає жодних помилок під час виконання. Отже, тепер ми можемо перейти до задач іншого типу, де необхідно більше взаємодіяти з математичними формулами, а саме до задач чисельних методів.

РОЗДІЛ 2

ЗАСТОСУВАННЯ CHATGPT ДЛЯ РОЗВ'ЯЗУВАННЯ ЗАДАЧ ЧИСЕЛЬНИХ МЕТОДІВ

Ми уже розглянули застосування ChatGPT для алгоритмічних задач і цей штучний інтелект доволі добре взаємодіяв у форматі коли ми вказували задачу і просили його згенерувати код для неї. Для задач чисельних методів будемо використовувати аналогічний підхід, тобто будемо описувати завдання, за необхідності метод яким ми хочемо отримати розв’язок, для ChatGPT і він має згенерувати доволі детальний опис розв'язку, оскільки код буде відтворенням формул що будуть описані у розв’язку, та код аби ми могли запустити його і перевірити правильність на декількох прикладах. Оскільки ми досліджуємо можливості ChatGPT, то нам необхідно розглядати задачі та методи різної складності.

У роботі використали класичний підручник [7] з чисельних методів і з нього виділили декілька стандартних методів. Вибирали методи із різних розділів, так аби отримати репрезентативну вибірку задач.

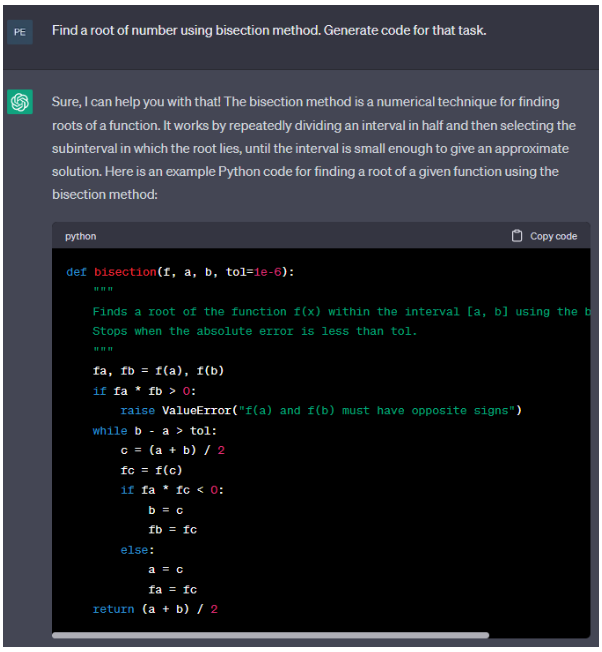
2.1. Метод бісекції та простої ітерації

Метод бісекції та простої ітерації є одними із найпростіших методів для пошуку кореня функції. Ці методи вивчаються у багатьох університетах та про них написано багато різноманітних наукових робіт.

2.1.1. Метод бісекції

Розглянемо задачу пошуку розв’язку функції. Тобто нам необхідно знайти x_0 для функції $f(x)$ такий, що $f(x_0) = 0$.

У цьому методі нам необхідно знати проміжок на якому будемо шукати розв’язок, значення функції на кінцях проміжку мають бути протилежних знаків. З теореми Больцано-Коші ми знаємо, що у такому випадку на проміжку існує розв’язок. Знаючи це можна розділяти проміжок навпіл і після цього один із проміжків буде задовольняти умову про протилежні знаки. Таким чином можна зменшувати проміжок поки він не досягне вказаної точності.



Find a root of number using bisection method. Generate code for that task.

Sure, I can help you with that! The bisection method is a numerical technique for finding roots of a function. It works by repeatedly dividing an interval in half and then selecting the subinterval in which the root lies, until the interval is small enough to give an approximate solution. Here is an example Python code for finding a root of a given function using the bisection method:

```python
def bisection(f, a, b, tol=1e-6):
    """
    Finds a root of the function f(x) within the interval [a, b] using the b
    Stops when the absolute error is less than tol.
    """
    fa, fb = f(a), f(b)
    if fa * fb > 0:
        raise ValueError("f(a) and f(b) must have opposite signs")
    while b - a > tol:
        c = (a + b) / 2
        fc = f(c)
        if fa * fc < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc
    return (a + b) / 2
```

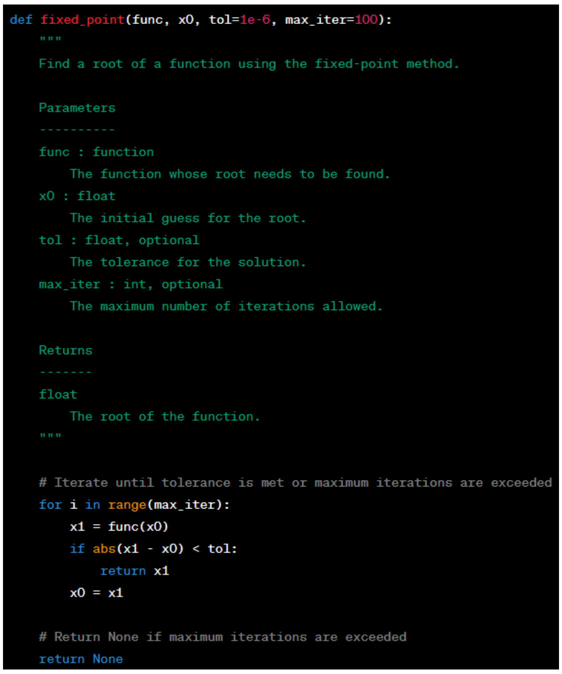
Рисунок 10. Взаємодія для методу розрізу навпіл

ChatGPT повністю впорався із завданням, згенерований код працює коректно. Для декількох різних вхідних даних код знайшов правильну відповідь – корінь функції на заданому проміжку.

2.1.2. Метод простої ітерації

Розглянемо задачу пошуку нерухомої точки. Тобто розв'язку рівняння $f(x) = x$. Задача пошуку кореня функції $g(x)$ легко зіставити із цим методом взявши $f(x) = g(x) + x$.

Для того аби знайти нерухому точку можна використати метод простої ітерації, де на кожному кроці значення x змінюють на $f(x)$.



```
def fixed_point(func, x0, tol=1e-6, max_iter=100):
    """
    Find a root of a function using the fixed-point method.

    Parameters
    ----------
    func : function
        The function whose root needs to be found.
    x0 : float
        The initial guess for the root.
    tol : float, optional
        The tolerance for the solution.
    max_iter : int, optional
        The maximum number of iterations allowed.

    Returns
    -------
    float
        The root of the function.
    """

    # Iterate until tolerance is met or maximum iterations are exceeded
    for i in range(max_iter):
        x1 = func(x0)
        if abs(x1 - x0) < tol:
            return x1
        x0 = x1

    # Return None if maximum iterations are exceeded
    return None
```

Рисунок 11. Згенерований код для методу простої ітерації

ChatGPT згенерував код простої ітерації і він працює коректно. Також цей код було перевірено на декількох прикладах.

2.2. Інтерполяція многочлена

Задача пошуку інтерполяційного многочлена полягає у пошуку многочлена мінімального степеня, що проходить через заданий набір точок. Найпопулярнішими методами є метод Лагранжа та метод Ньютона.

2.2.1. Інтерполяційний многочлен Лагранжа

Цей метод полягає у тому, аби для кожної точки був свій доданок який буде рівний 0 для усіх інших точок, це можна досягнути використавши такий добуток

$$\prod_{j = 0, j \neq i}^{n - 1} (x - x_j)$$

Усе що залишається лише аби для точки x_i цей доданок був рівний y_i , тому і-ий доданок буде таким:

$$\frac{y_i * \prod_{j=0, j\neq i}^{n-1} (x - x_j)}{\prod_{j=0, j\neq i}^{n-1} (x_i - x_j)}$$

2.2.2. Інтерполяційний многочлен Ньютона

У цьому методі використовується побудова многочлена у вигляді:

$$N(x) = [y_0] + [y_0, y_1] * (x - x_0) + ... + [y_0, ..., y_k] * (x - x_0) * ... * (x - x_k)$$

$$[y_0, ..., y_k] = \sum_{i=0}^{k} \frac{y_i}{\prod_{j=0, j\neq i}^{k} (x_i - x_j)}$$

Цей метод дозволяє ефективно перераховувати многочлен при додаванні ще однієї точки.

2.2.3. Результати для інтерполяції многочлена

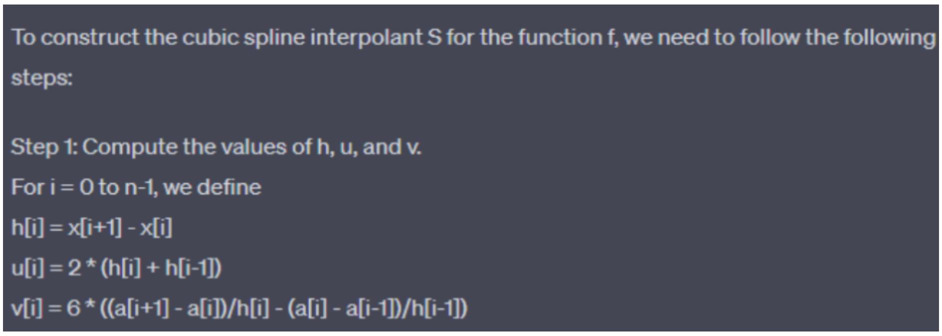
ChatGPT згенерував коректний код для обох методів. Проте він використовує інше позначення змінних, що ніяк не впливає на правильність, що було перевірено на декількох прикладах.

2.3. Кубічний сплайн

Задача побудови кубічного сплайна полягає у тому аби для заданих $x_0,..., x_n$ та $a_0 = f(x_0),..., a_n = f(x_n)$ знайти для кожного проміжку $[x_j, x_{j+1}]$ коефіцієнти для $S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$.

Алгоритм для розв’язання цієї задачі такий:

- Обраховуємо $h_i = x_{i+1} - x_i$
- Рахуємо $\alpha_i = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1})$
- Встановлюємо $l_0 = l_n = 1, \mu_0 = 0, z_0 = z_n = 0, c_n = 0$
- Перераховуємо
  - $l_i = 2(x_{i+1} - x_{i-1}) - h_{i-1}\mu_{i-1}$
  - $\mu_i = h_i / l_i$
  - $z_i = (\alpha_i - h_{i-1}z_{i-1})/l_i$
- Рахуємо відповідь
  - $c_j = z_j - \mu_j * c_{j+1}$
  - $b_j = (a_{j+1} - a_j)/h_j - h_j(c_{j+1} + 2c_j)/3$
  - $d_j = (c_{j+1} - c_j)/(3h_j)$



To construct the cubic spline interpolant S for the function f, we need to follow the following steps:

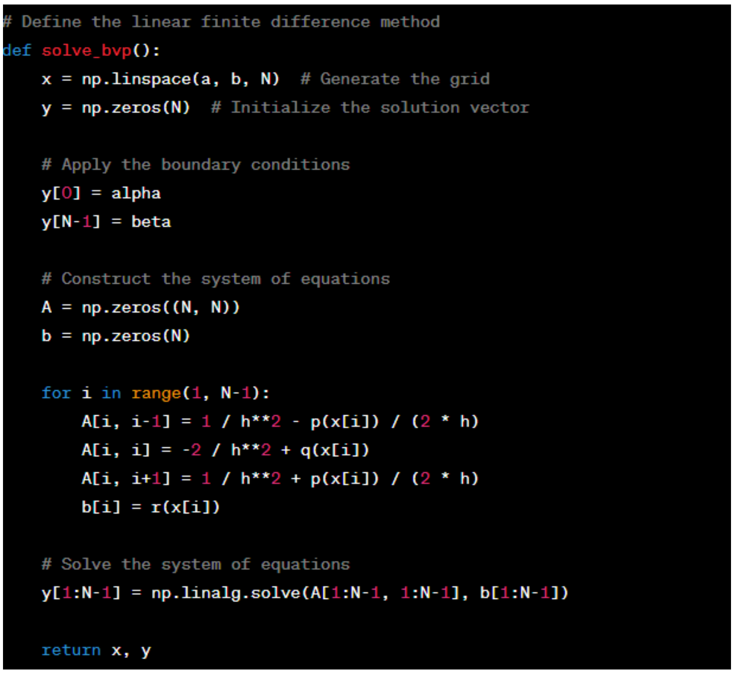
Step 1: Compute the values of h, u, and v.
For i = 0 to n-1, we define
h[i] = x[i+1] - x[i]
u[i] = 2 * (h[i] + h[i-1])
v[i] = 6 * ((a[i+1] - a[i])/h[i] - (a[i] - a[i-1])/h[i-1])

Рисунок 12. Згенеровані формули для кубічного сплайна

Як можна помітити, формули які він використовує є подекуди хибними. Наприклад h[i] пораховано вірно, а v[i] майже правильно пораховане по аналогії до α_i, проте вдвічі більше. Далі в обрахунках теж використовуються хибні формули і відповідно програма працює некоректно на різних прикладах.

2.4. Метод скінченних різниць

Задачу пошуку наближеного розв’язку крайової задачі $y'' = p(x)y' + q(x)y + r(x)$, для $a \leq x \leq b$, з $y(a) = \alpha$ та $y(b) = \beta$, можна розв'язувати за допомогою методу скінченних різниць.



```
# Define the linear finite difference method
def solve_bvp():
    x = np.linspace(a, b, N)  # Generate the grid
    y = np.zeros(N)  # Initialize the solution vector

    # Apply the boundary conditions
    y[0] = alpha
    y[N-1] = beta

    # Construct the system of equations
    A = np.zeros((N, N))
    b = np.zeros(N)

    for i in range(1, N-1):
        A[i, i-1] = 1 / h**2 - p(x[i]) / (2 * h)
        A[i, i] = -2 / h**2 + q(x[i])
        A[i, i+1] = 1 / h**2 + p(x[i]) / (2 * h)
        b[i] = r(x[i])

    # Solve the system of equations
    y[1:N-1] = np.linalg.solve(A[1:N-1, 1:N-1], b[1:N-1])

    return x, y
```

Рисунок 13. Згенерований код для пошуку наближеного розв’язку крайової задачі

ChatGPT використав зовсім хибні формули. Розглянемо побудовані графіки наближеного розв’язку крайової задачі $y'' = y' + x$, для $0 \leq x \leq 1$, з $y(0) = 0$ та $y(1) = 1$.

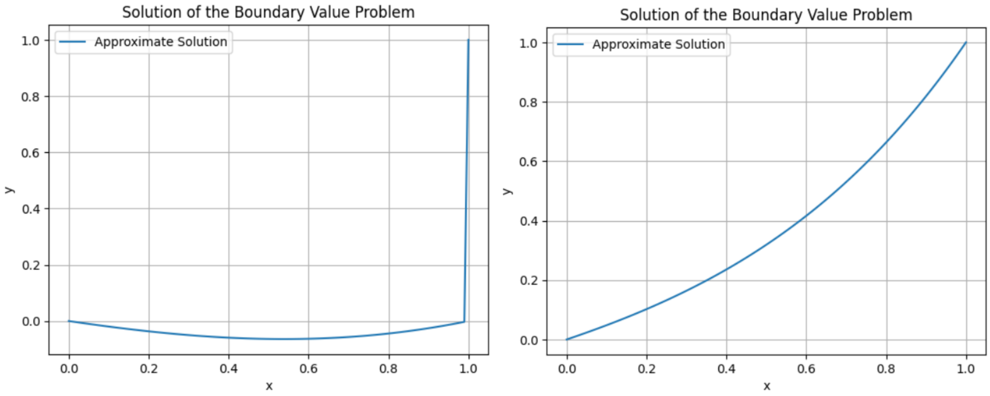


Рисунок 14. а) результат з кодом ChatGPT

б) результат з правильними формулами

2.5. Методи Ейлера, Гойна та Рунге-Кутта

Розглянемо методи якими можна отримати наближення до правильно поставленої початкової задачі:

$$\frac{dy}{dt} = f(t,\ y),\ a \leq t \leq b,\ y(a)\ =\ \alpha$$

2.5.1. Метод Ейлера

Метод Ейлера є найелементарнішим методом наближення розв'язування початкових задач. Хоча він рідко використовується на практиці, простота його виведення може бути використана для ілюстрації методів, залучених у побудові деяких більш просунутих методів, без громіздкої алгебри, яка супроводжує ці конструкції.

В методі необхідно вибрати N від якого залежить точність та швидкодія методу, ми розбиваємо проміжок на N + 1 рівновіддалену точку $h\ =\ \frac{b-a}{N}$, $t_i = a\ +\ i\ *\ h$, тепер ми можемо рахувати значення y_i у відповідних точках за формулою $y_{i+1} = y_i +\ h\ *\ f(t_i, y_i)$.

2.5.2. Метод Гойна

Метод Гойна подібний до методу Ейлера, що його часто називають модифікованим методом Ейлера. Аналогічно рахуємо h, після рахуємо

$$\underline{y}_{i+1} = y_i + h * f(t_i, y_i)$$

$$y_{i+1} = y_i + \frac{h*(f(t_i, y_i) + f(t_{i+1}, \underline{y}_{i+1}))}{2}$$

Тобто обраховується проміжне наближення, а після уже обраховується фінальне наближення.

2.5.3. Метод Рунге-Кутта

Розглянемо приклад 4-ох етапного методу Рунге-Кутта. В цьому методі рахують 4 допоміжні змінні:

$$k_1 = f(t_i, y_i)$$

$$k_2 = f(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1)$$

$$k_3 = f(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_2)$$

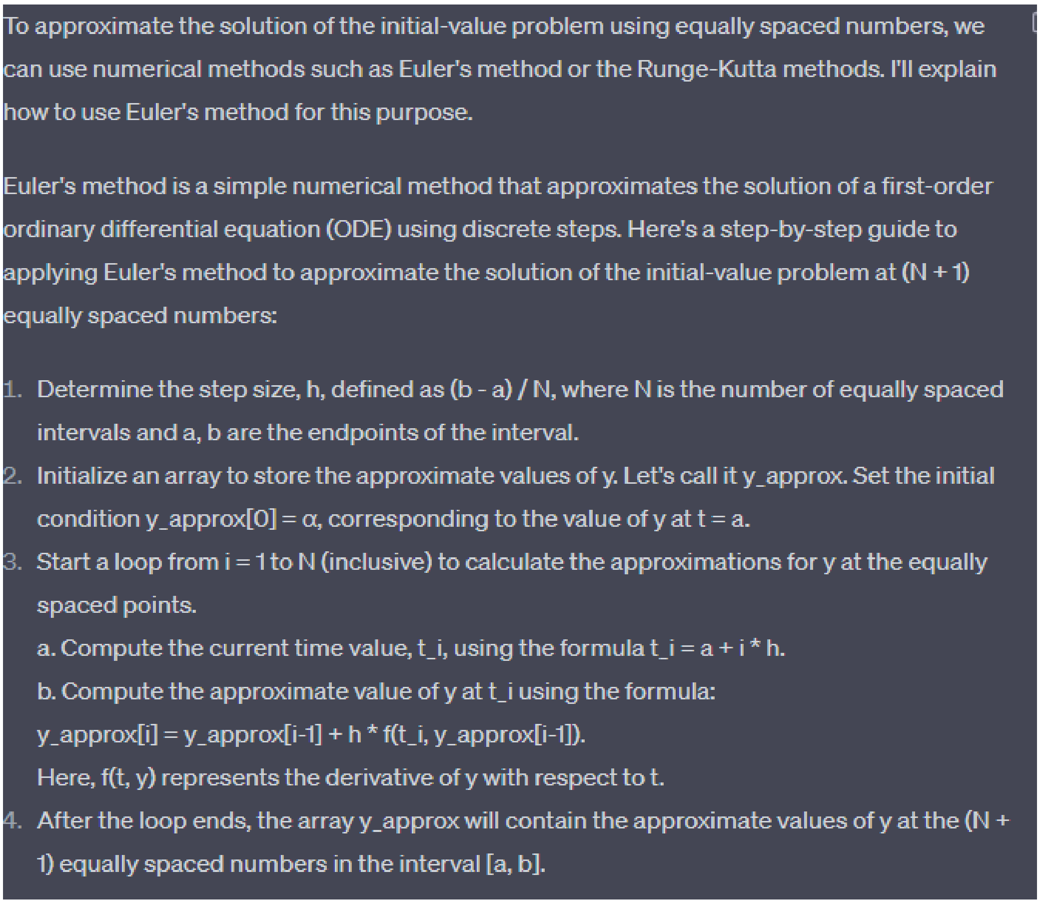
$$k_4 = f(t_i + h, y_i + hk_3)$$

тоді значення в точці визначається такою формулою:

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

2.5.4. Результати

ChatGPT розв’язав цю задачу методом Ейлера і згенерував вірний код. Для інших методів прийшлось додатково йому вказувати метод яким розв’язувати цю задачу. Для цих методів він також зміг згенерувати вірний розв’язок та код. Усі методи перевірялись на декількох прикладах.



To approximate the solution of the initial-value problem using equally spaced numbers, we can use numerical methods such as Euler's method or the Runge-Kutta methods. I'll explain how to use Euler's method for this purpose.

Euler's method is a simple numerical method that approximates the solution of a first-order ordinary differential equation (ODE) using discrete steps. Here's a step-by-step guide to applying Euler's method to approximate the solution of the initial-value problem at (N + 1) equally spaced numbers:

1. Determine the step size, h, defined as (b - a) / N, where N is the number of equally spaced intervals and a, b are the endpoints of the interval.
2. Initialize an array to store the approximate values of y. Let's call it y_approx. Set the initial condition y_approx[0] = α, corresponding to the value of y at t = a.
3. Start a loop from i = 1 to N (inclusive) to calculate the approximations for y at the equally spaced points.
   a. Compute the current time value, t_i, using the formula t_i = a + i * h.
   b. Compute the approximate value of y at t_i using the formula:
   y_approx[i] = y_approx[i-1] + h * f(t_i, y_approx[i-1]).
   Here, f(t, y) represents the derivative of y with respect to t.
4. After the loop ends, the array y_approx will contain the approximate values of y at the (N + 1) equally spaced numbers in the interval [a, b].

Рисунок 15. Згенерований опис методу Ейлера

Висновки для задач чисельних методів

У цьому розділі ми проаналізували різноманітні задачі чисельних методів. У задачах з простими формулами ChatGPT не допустив помилок, проте у найважчих розглянутих методах формули були складніші і штучний інтелект не зміг відтворити правильні формули, що є у багатьох наукових роботах. Враховуючи ці результати можна вважати, що використовувати ChatGPT є доцільно для задач, що містять методи з простими математичними формулами. Отримані результати збігаються із результатами розробників [3]. Це може бути наслідком того, що різні статті мають різний формат і відповідні формули відображаються по різному. Проте теоретичну частину він знає достатньо добре. Тому для задач чисельних методів варто перевіряти формули, звіряти їх з перевіреним джерелом.

Найскладнішими методами, серед розглянутих, з якими ChatGPT впорався просто, без проблем із формулами – методи Ейлера, Гойна та Рунге-Кутта. Отже, використаємо їх як методи, що будуть використовуватись у науковій роботі яку має згенерувати ChatGPT. У наступному розділі, ми розглянемо генерацію цієї роботи.

РОЗДІЛ 3

ГЕНЕРАЦІЯ НАУКОВОЇ РОБОТИ ВИКОРИСТОВУЮЧИ CHATGPT

3.1. Вибір теми

У попередніх розділах ми розглянули, як ChatGPT розв’язує алгоритмічні задачі та задачі чисельних методів. Серед чисельних методів можна виділити методи Ейлера, Гойна та Рунге-Кутта, як найскладніші методи із якими штучний інтелект впорався. Використаємо цю оцінку і саме тому будемо генерувати наукову роботу про використання цих методів.

Серед варіантів задачі, яку він буде розв’язувати у науковій роботі, вибрали задачу моделювання епідемії. Цей вибір був пов’язаний із тим, що тема є доволі актуальною, також є багато наукових робіт в цьому напрямку і ми очікували, що він зможе згенерувати диференціальні рівняння для якоїсь моделі.

3.2. Моделювання епідемії

3.2.1. Модель SIR

Модель SIR, добре відома епідеміологічна модель, була розроблена майже століття тому і широко використовується в цій галузі.

Ця модель поділяє всю чутливу популяцію на три групи: S – чутливі, I – інфіковані і R – ті, що одужали. На початку епідемії всі особини належать до групи S і для її початку має бути хоча б одна інфікована особина. Модель передбачає послідовне обчислення кількості населення у кожній групі на наступному кроці.

Щоб визначити кількість осіб, інфікованих на кожному етапі, необхідно враховувати дві ймовірності: ймовірність контакту між двома особами та ймовірність зараження β, коли інфікована особа вступає в контакт із чутливою особою. Зазвичай, першу ймовірність вибирають як 1/N, де N — розмір популяції. Ймовірність β є біологічним показником інфекційності конкретної хвороби із урахуванням різних факторів, у тому числі носіння масок, нежить та багато інших факторів.

Загальна кількість інфікованих осіб, що будуть заражені від інфікованих рівна $\frac{I*S*\beta}{N}$.

Для обрахунку кількості осіб, що одужують в певний момент, необхідно врахувати ймовірність γ, зазвичай це число рівне обернене до тривалості хвороби. Кількість осіб, що одужають на цьому кроці рівна I*γ.

Врахувавши усі попередні формули виходить така система диференціальних рівнянь:

$\frac{dS}{dt} = -\frac{\beta SI}{N}$ — приріст кількості чутливих осіб

$\frac{dI}{dt} = \frac{\beta SI}{N} - \gamma I$ — приріст інфікованих осіб

$\frac{dR}{dt} = \gamma I$ — приріст осіб, що одужали

3.2.2. Модель SEIR

Модель SEIR, модифікація моделі SIR, враховує інкубаційний період. У нас тепер є нова група E – особи, які не є заразними, але з часом захворіють. Чутливі особи заражаються таким же чином, як і в моделі SIR, але класифікуються як E замість I. Перехід від E до I відбувається з певною ймовірністю α, зворотною інкубаційному періоду.

Відповідно нова система диференціальних рівнянь:

$\frac{dS}{dt} = -\frac{\beta SI}{N}$ — приріст кількості чутливих осіб

$\frac{dE}{dt} = \frac{\beta SI}{N} - \alpha E$ — приріст осіб з інкубаційним періодом

$\frac{dI}{dt} = \alpha E - \gamma I$ — приріст інфікованих осіб

$\frac{dR}{dt} = \gamma I$ — приріст осіб, що одужали

Є ще багато інших моделей подібних до SIR та SEIR, що поділяють населення на певні групи. Ці моделі можуть враховувати різні сценарії, включаючи, втрату імунітету, вакцинацію та багато іншого. Єдиним обмеженням для цих моделей є уява дослідника.

3.2.3. Задача моделювання

Через поширення різних захворювань, у тому числі COVID-19, у світі, математичне моделювання епідеміологічних процесів – це важлива і актуальна наукова проблема.

Задача моделювання епідемії полягає у прогнозуванні перебігу розповсюдження захворювання. Маючи ці дані, можна вживати різноманітні заходи, такі як карантин, вакцинація і т.д..

Наприклад для моделювання із моделлю SIR можемо отримати графік, що зображений на рисунку 16.

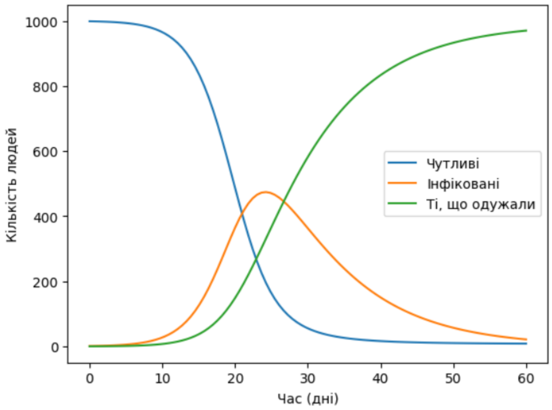


Рисунок 16. Моделювання епідемії за допомогою моделі SIR

3.2.4. Алгоритм розв’язування задачі моделювання епідемії

Першим кроком для моделювання епідемії необхідно вибрати модель, яку ми будемо розглядати. Розглянемо модель SIR, запишемо для неї формули:

$\frac{dS}{dt} = -\frac{\beta SI}{N}$ — приріст кількості чутливих осіб

$\frac{dI}{dt} = \frac{\beta SI}{N} - \gamma I$ — приріст інфікованих осіб

$\frac{dR}{dt} = \gamma I$ — приріст осіб, що одужали

Тепер нам необхідно вибрати метод, яким розв’язувати цю задачу, наприклад метод Ейлера, який ми розглядали у розділі 2.5.1. Нам необхідно розбити часовий інтервал на N рівних проміжків. Якщо ми розглядаємо модель на часовому проміжку $[a, b]$, тоді отримаємо такі формули: $h = \frac{b-a}{N}$, $t_i = a + i * h$. Перепишемо нашу систему:

$$S_{i+1} = S_i - \frac{\beta S_i I_i}{N}$$

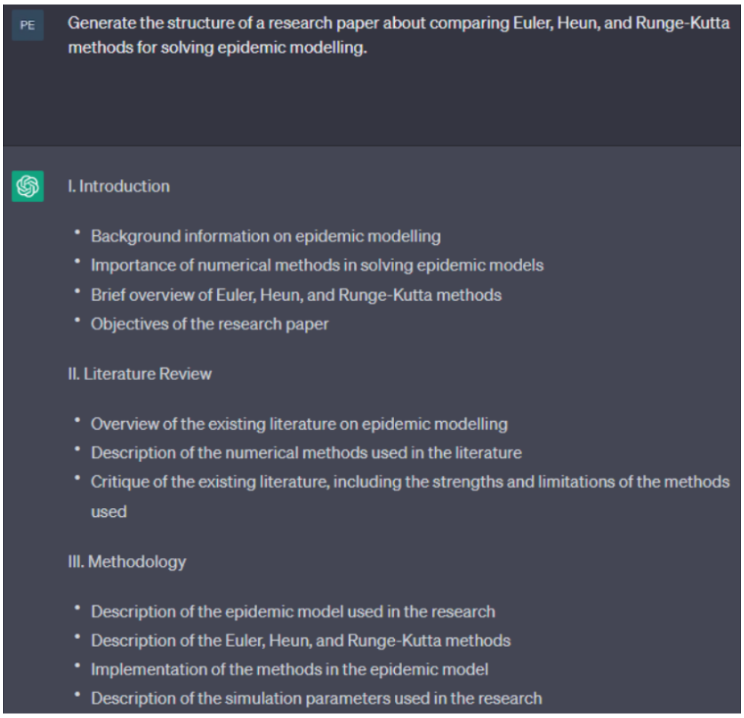
$$I_{i+1} = I_i + \frac{\beta S_i I_i}{N} - \gamma I_i$$

$$R_{i+1} = R_i + \gamma I_i$$

Відповідно цими формулами ми можемо порахувати значення S, I та R у різні моменти часу.

3.3. Процес взаємодії

Взаємодія з ChatGPT починалась з того, щоб він згенерував зміст роботи для нашої вибраної теми. Далі нам необхідно згенерувати текст для кожного розділу та підрозділу. Для цього необхідно кожен раз вказувати зміст, бо він може згенерувати новий розділ якого немає в змісті. І також необхідно вказувати що треба генерувати лише цей розділ, бо деколи він може згенерувати декілька розділів разом.



PE

Generate the structure of a research paper about comparing Euler, Heun, and Runge-Kutta methods for solving epidemic modelling.

I. Introduction

- Background information on epidemic modelling
- Importance of numerical methods in solving epidemic models
- Brief overview of Euler, Heun, and Runge-Kutta methods
- Objectives of the research paper

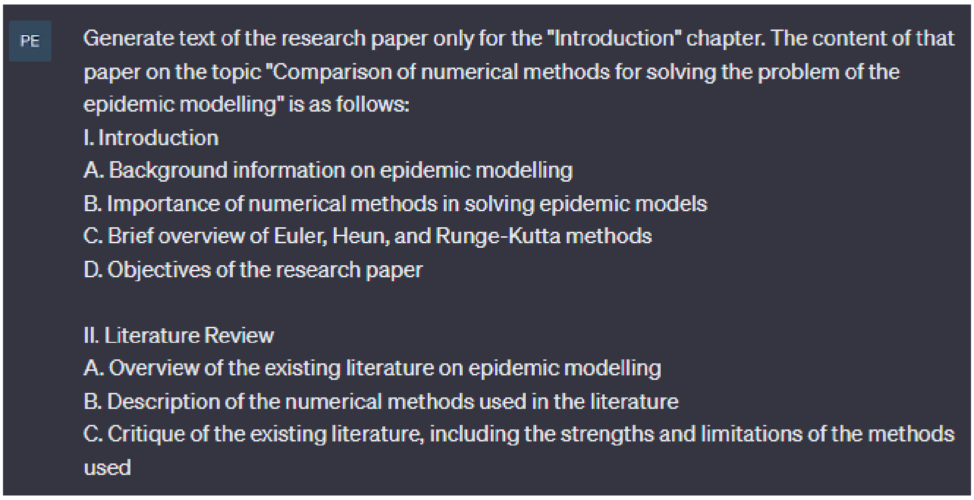
II. Literature Review

- Overview of the existing literature on epidemic modelling
- Description of the numerical methods used in the literature
- Critique of the existing literature, including the strengths and limitations of the methods used

III. Methodology

- Description of the epidemic model used in the research
- Description of the Euler, Heun, and Runge-Kutta methods
- Implementation of the methods in the epidemic model
- Description of the simulation parameters used in the research

Рисунок 17. Взаємодія при генерації змісту



PE

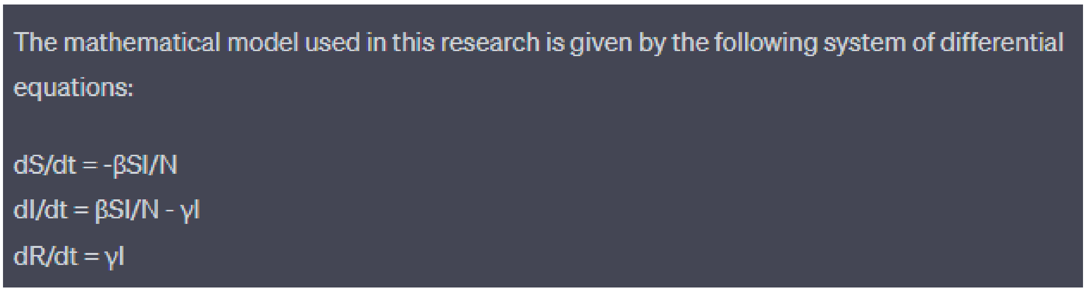
Generate text of the research paper only for the "Introduction" chapter. The content of that paper on the topic "Comparison of numerical methods for solving the problem of the epidemic modelling" is as follows:
I. Introduction
A. Background information on epidemic modelling
B. Importance of numerical methods in solving epidemic models
C. Brief overview of Euler, Heun, and Runge-Kutta methods
D. Objectives of the research paper

II. Literature Review
A. Overview of the existing literature on epidemic modelling
B. Description of the numerical methods used in the literature
C. Critique of the existing literature, including the strengths and limitations of the methods used

Рисунок 18. Генерація вступного розділу

3.4 Результати генерації

До різних розділів, він зумів згенерувати текст у науковому стилі, майже без помилок. Проте це не єдине що він мав згенерувати. Одним із завдань був вибір моделі для задачі моделювання епідемії. І він впорався із тим аби використати коректну модель, а саме він використав SIR, хоча пізніше він допустив помилку в розділі де порівнює результати з результатами в інших роботах, там він вказує що у роботі використовували модель SEIR.

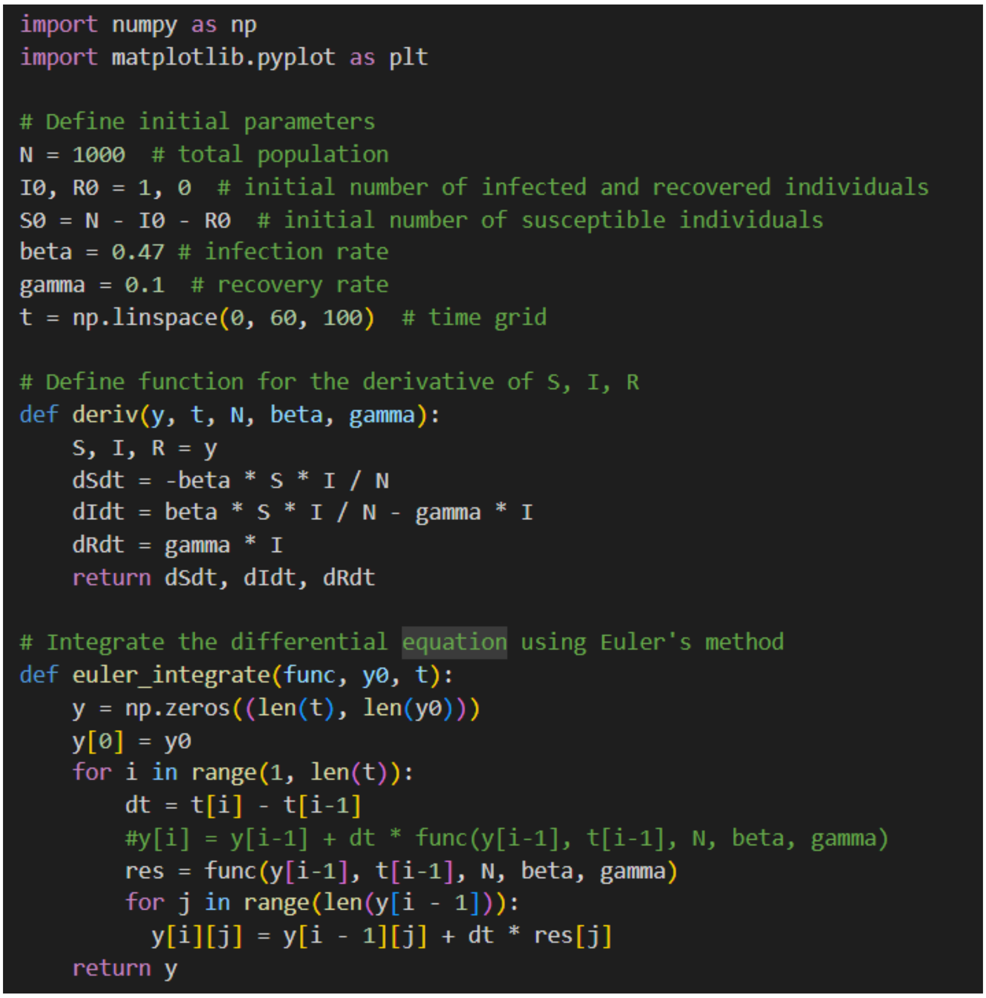


The mathematical model used in this research is given by the following system of differential equations:

dS/dt = -βSI/N
dI/dt = βSI/N - γI
dR/dt = γI

Рисунок 19. Згенерована математична модель

Також він згенерував три програми для кожного методу, на рисунку 21 зображено код для методу Ейлера.



```python
import numpy as np
import matplotlib.pyplot as plt

# Define initial parameters
N = 1000  # total population
I0, R0 = 1, 0  # initial number of infected and recovered individuals
S0 = N - I0 - R0  # initial number of susceptible individuals
beta = 0.47 # infection rate
gamma = 0.1  # recovery rate
t = np.linspace(0, 60, 100)  # time grid

# Define function for the derivative of S, I, R
def deriv(y, t, N, beta, gamma):
    S, I, R = y
    dSdt = -beta * S * I / N
    dIdt = beta * S * I / N - gamma * I
    dRdt = gamma * I
    return dSdt, dIdt, dRdt

# Integrate the differential equation using Euler's method
def euler_integrate(func, y0, t):
    y = np.zeros((len(t), len(y0)))
    y[0] = y0
    for i in range(1, len(t)):
        dt = t[i] - t[i-1]
        #y[i] = y[i-1] + dt * func(y[i-1], t[i-1], N, beta, gamma)
        res = func(y[i-1], t[i-1], N, beta, gamma)
        for j in range(len(y[i - 1])):
          y[i][j] = y[i - 1][j] + dt * res[j]
    return y
```

Рисунок 21. Згенерований код для методу Ейлера у згенерованій науковій роботі

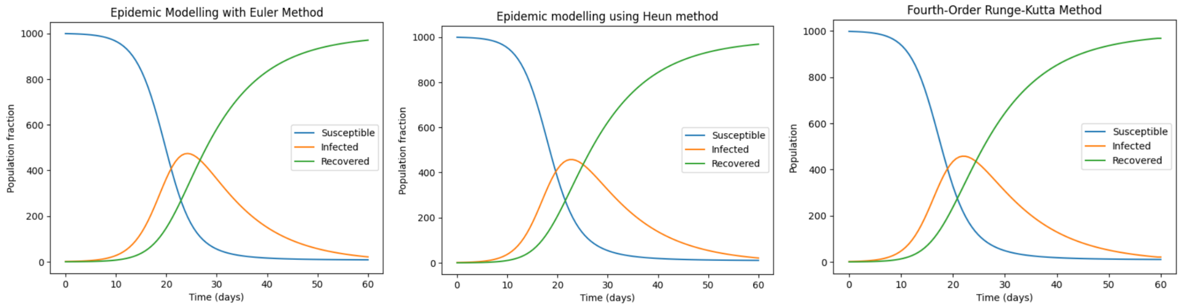
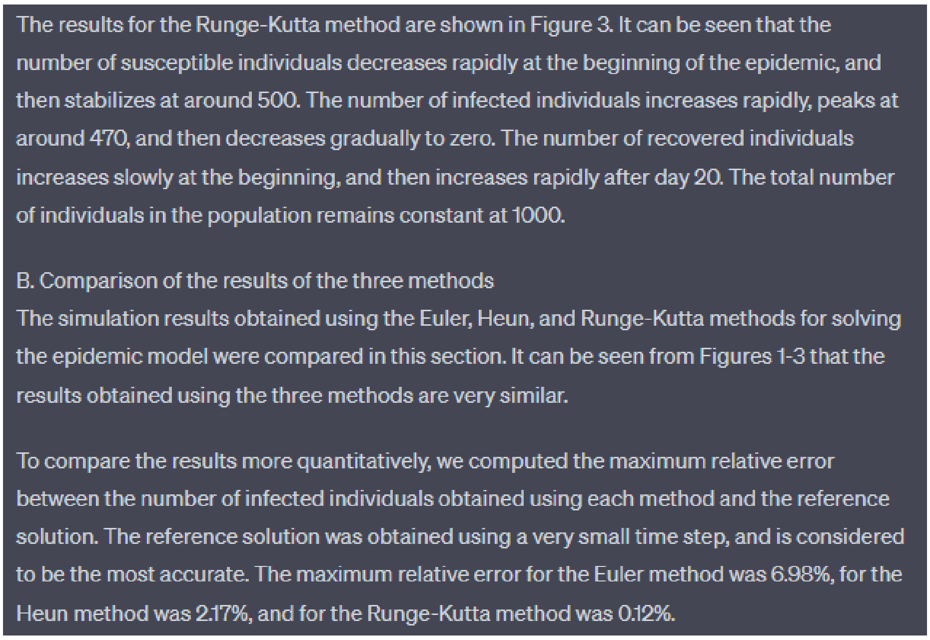


Рисунок 22. Графіки із згенерованих кодів для відповідних методів при beta = 0.47, gamma = 0.1 і N = 100

Усі згенеровані програми, обраховують правильні значення, та будують графіки доволі близькі до графіка із великим значенням розбиття.

Також цікавим розділом були результати програм. Спочатку він навів певні чисельні результати для кожного методу, також навів певну точність для кожного методу в порівнянні із симуляцією з дуже малим часовим кроком.



The results for the Runge-Kutta method are shown in Figure 3. It can be seen that the number of susceptible individuals decreases rapidly at the beginning of the epidemic, and then stabilizes at around 500. The number of infected individuals increases rapidly, peaks at around 470, and then decreases gradually to zero. The number of recovered individuals increases slowly at the beginning, and then increases rapidly after day 20. The total number of individuals in the population remains constant at 1000.

B. Comparison of the results of the three methods
The simulation results obtained using the Euler, Heun, and Runge-Kutta methods for solving the epidemic model were compared in this section. It can be seen from Figures 1-3 that the results obtained using the three methods are very similar.

To compare the results more quantitatively, we computed the maximum relative error between the number of infected individuals obtained using each method and the reference solution. The reference solution was obtained using a very small time step, and is considered to be the most accurate. The maximum relative error for the Euler method was 6.98%, for the Heun method was 2.17%, and for the Runge-Kutta method was 0.12%.

Рисунок 20. Згенеровані результати

Зрозуміло, що це вигадані дані, або взяті із якихось інших наукових робіт, тому необхідно було надати йому реальні дані з програм, які є близькими до реальних значень із програм, проте ці результати залежать від усіх параметрів задачі. Тому залишили першу отриману версію.

Висновки для генерації наукової роботи

Підсумовуючи усі результати можна вважати, що ChatGPT добре впорався із завданням. У роботі близько 15 сторінок науково написано тексту, що відповідає темі роботи. Також роботу перевірив на сайті Quetext [8], де результат вказав на відсутність плагіату, було лише пару загальних речень які були знайдені в інших джерелах. Тобто робота відповідає багатьом нормам наукової роботи. Із нею можна ознайомити в додатку Б.

Єдиним недоліком згенерованого тексту було лише те, що для оформлення наукової роботи треба було виправити форматування, наприклад шрифт та розмір тексту, та необхідно власноруч запускати код і вставляти отримані зображення. Проте це не таке складне завдання, як зробити відповідне дослідження та описати результати.

ВИСНОВКИ

У результаті дослідження можна сказати, що ChatGPT вміє розв’язувати базові задачі з алгоритмів. Для трохи складніших задач він може згенерувати різні результати, тому необхідно перевіряти згенерований текст та код, після чого можна або вказати на помилки, які помітили, або спробувати ще раз згенерувати відповідь. Якщо виконати такі дії, то можна у багатьох випадках отримати кращий результат, аж до правильного розв’язку. Зі складними задачами такі дії не допоможуть ChatGPT, оскільки він не може згенерувати алгоритм, що потребує декількох складних кроків. Також важливим є те, що він не може запускати програми, тому у випадку якщо він генерує приклад, це не означає що код саме такий же ж результат надасть для його ж прикладу. Отже, завжди необхідно запускати код, аби переконатись у його правильності для декількох прикладах.

Для задач чисельних методів складно підвести межу можливостей, оскільки ChatGPT має труднощі з генеруванням правильних формул. Якщо говорити про формули, то він може впоратись із не громіздкими формулами або формулами які збігаються у багатьох наукових роботах, що зазвичай відповідає відносно простим методам. Із генерацією коду в нього майже немає ніяких проблем, оскільки усе що йому потрібно це відтворити згенеровані формули. Беручи до уваги згенеровану роботу можна сказати, що його доцільно використовувати у генерації певних звітів або курсових робіт у простих чисельних методах.

Оцінка можливостей ChatGPT дозволяє зрозуміти, в яких сферах і як саме може порушуватись академічна доброчесність. Особливо згенерована робота свідчить про це. Проте, у сфері прикладних задач це можливо лише для простих задач або для теоретичних відомостей.

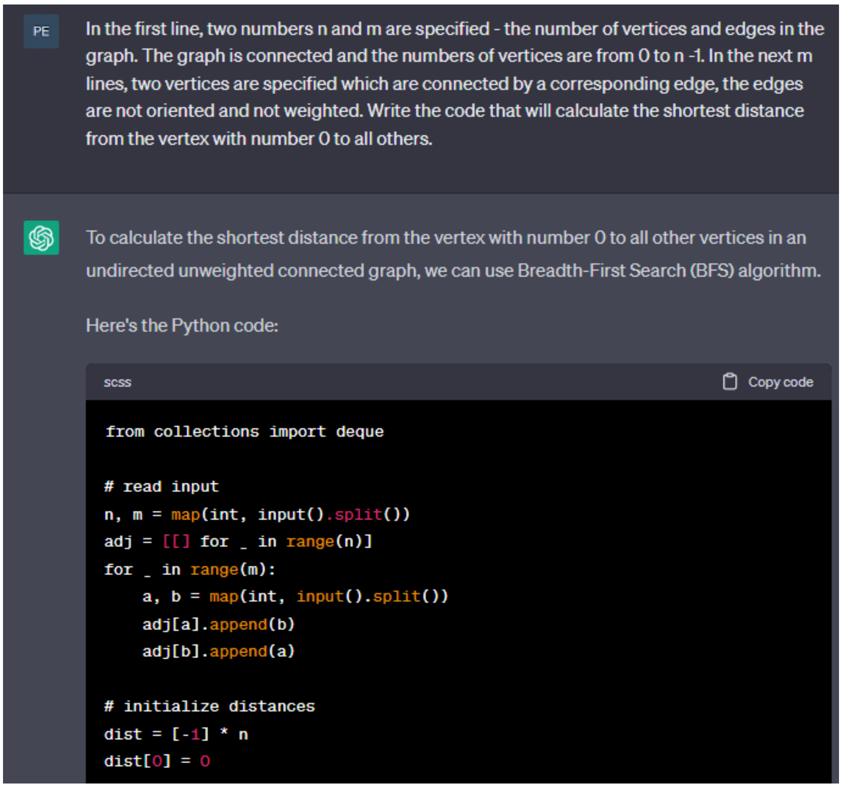
Оскільки, ChatGPT це не найкраща можлива версія GPT, то ми можемо очікувати оновлень ChatGPT до версії GPT-4, яка має набагато кращі вміння у різноманітних сферах, особливо у використанні математичних формул. Тому при подальших суттєвих оновленнях необхідно буде досліджувати нову версію, також залишається актуальною дослідження того, як не допустити порушення академічної доброчесності за допомогою ChatGPT.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ChatGPT – Режим доступу: https://chat.openai.com/ – Дата звернення: 10.05.2023 – Назва з екрана
2. Five Ideas for How Pre Ideas for How Professors Can Deal with GPT … For Now / Travis R. Pickell and Brian R. Doak – Режим доступу: https://digitalcommons.georgefox.edu/cgi/viewcontent.cgi?article=1432&context=ccs – Дата звернення: 10.05.2023
3. Sparks of Artificial General Intelligence: Early experiments with GPT-4 – Режим доступу: https://arxiv.org/pdf/2303.12712.pdf – Дата звернення: 10.05.2023
4. Вступ до Алгоритмів, переклад з англійської третього видання. / Т. Г. Кормен, Ч. Е. Лейзерсон, Р. Л. Рівест, К. Стайн; 2019. – 5 - 11 с.
5. Офіційний сайт ACM ICPC – Режим доступу: https://icpc.global/ - Дата звернення: 10.05.2023 – Назва з екрана
6. Sphere Online Judge – Режим доступу: https://spoj.com/ – Дата звернення: 10.05.2023 – Назва з екрана
7. Numerical Analysis / R. L. Burden and J. D. Faires; 2011. – 5 - 432 с.
8. Quetext – Режим доступу: https://www.quetext.com/ – Дата звернення: – Назва з екрана

Додаток А. Рисунки взаємодії для алгоритмічних задач

Задача пошуку найкоротшого шляху.

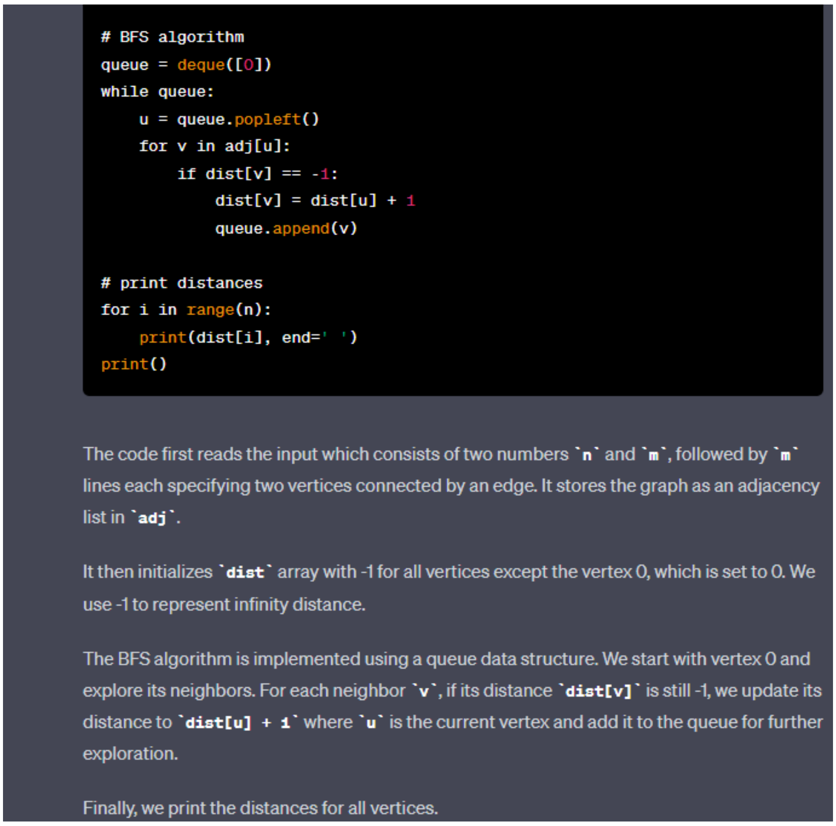


In the first line, two numbers n and m are specified - the number of vertices and edges in the graph. The graph is connected and the numbers of vertices are from 0 to n -1. In the next m lines, two vertices are specified which are connected by a corresponding edge, the edges are not oriented and not weighted. Write the code that will calculate the shortest distance from the vertex with number 0 to all others.

To calculate the shortest distance from the vertex with number 0 to all other vertices in an undirected unweighted connected graph, we can use Breadth-First Search (BFS) algorithm.

Here's the Python code:

```
from collections import deque

# read input
n, m = map(int, input().split())
adj = [[] for _ in range(n)]
for _ in range(m):
    a, b = map(int, input().split())
    adj[a].append(b)
    adj[b].append(a)

# initialize distances
dist = [-1] * n
dist[0] = 0
```



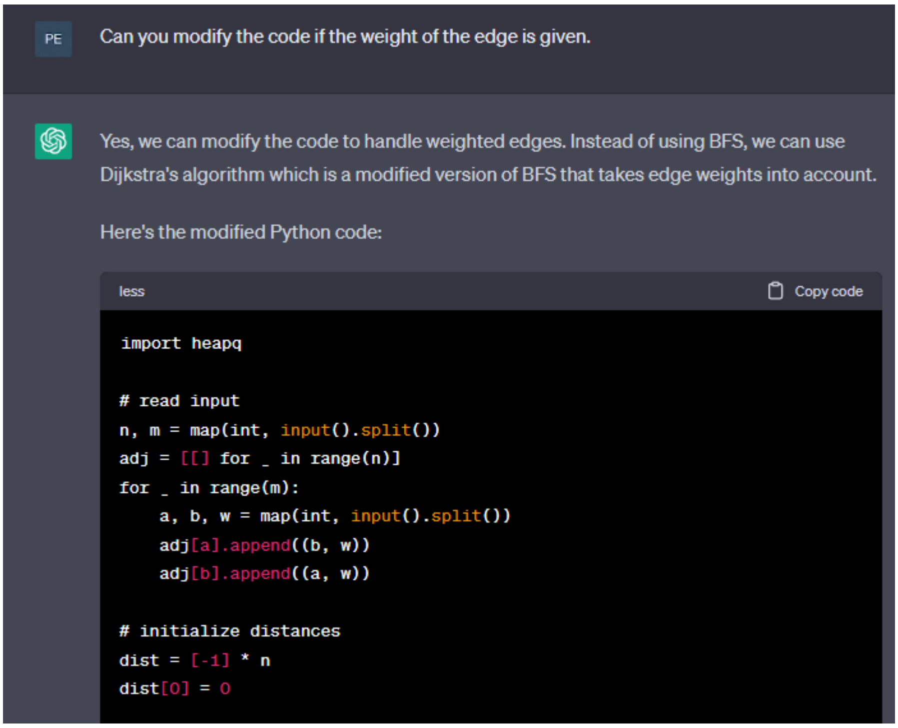
```
# BFS algorithm
queue = deque([0])
while queue:
    u = queue.popleft()
    for v in adj[u]:
        if dist[v] == -1:
            dist[v] = dist[u] + 1
            queue.append(v)

# print distances
for i in range(n):
    print(dist[i], end=' ')
print()
```

The code first reads the input which consists of two numbers `n` and `m`, followed by `m` lines each specifying two vertices connected by an edge. It stores the graph as an adjacency list in `adj`.

It then initializes `dist` array with -1 for all vertices except the vertex 0, which is set to 0. We use -1 to represent infinity distance.

The BFS algorithm is implemented using a queue data structure. We start with vertex 0 and explore its neighbors. For each neighbor `v`, if its distance `dist[v]` is still -1, we update its distance to `dist[u] + 1` where `u` is the current vertex and add it to the queue for further exploration.

Finally, we print the distances for all vertices.

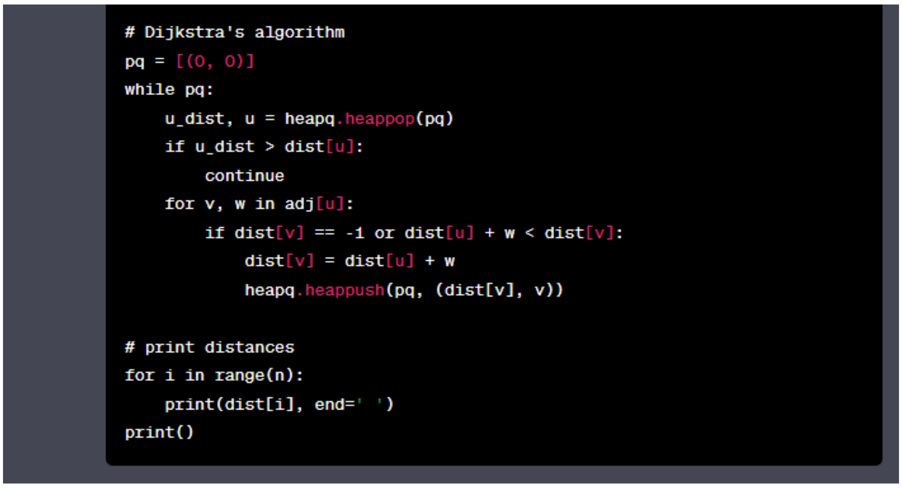


Can you modify the code if the weight of the edge is given.

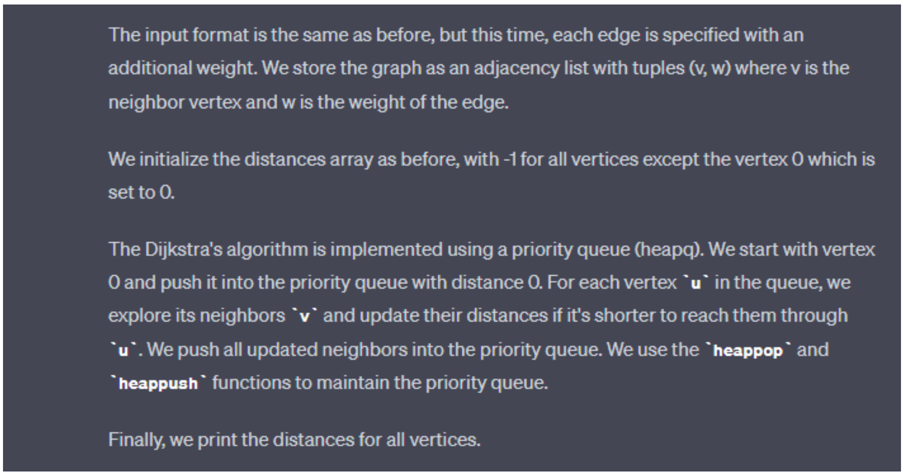
Yes, we can modify the code to handle weighted edges. Instead of using BFS, we can use Dijkstra's algorithm which is a modified version of BFS that takes edge weights into account.

Here's the modified Python code:

```
import heapq

# read input
n, m = map(int, input().split())
adj = [[] for _ in range(n)]
for _ in range(m):
    a, b, w = map(int, input().split())
    adj[a].append((b, w))
    adj[b].append((a, w))

# initialize distances
dist = [-1] * n
dist[0] = 0
```



```
# Dijkstra's algorithm
pq = [(0, 0)]
while pq:
    u_dist, u = heapq.heappop(pq)
    if u_dist > dist[u]:
        continue
    for v, w in adj[u]:
        if dist[v] == -1 or dist[u] + w < dist[v]:
            dist[v] = dist[u] + w
            heapq.heappush(pq, (dist[v], v))

# print distances
for i in range(n):
    print(dist[i], end=' ')
print()
```



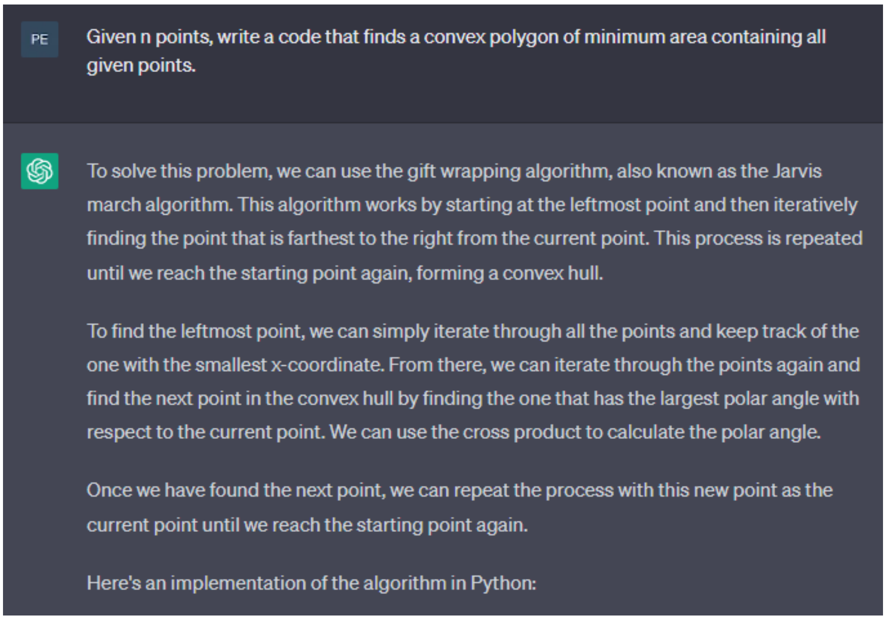
The input format is the same as before, but this time, each edge is specified with an additional weight. We store the graph as an adjacency list with tuples (v, w) where v is the neighbor vertex and w is the weight of the edge.

We initialize the distances array as before, with -1 for all vertices except the vertex 0 which is set to 0.

The Dijkstra's algorithm is implemented using a priority queue (heapq). We start with vertex 0 and push it into the priority queue with distance 0. For each vertex `u` in the queue, we explore its neighbors `v` and update their distances if it's shorter to reach them through `u`. We push all updated neighbors into the priority queue. We use the `heappop` and `heappush` functions to maintain the priority queue.

Finally, we print the distances for all vertices.

Пошук опуклої оболонки для заданих точок



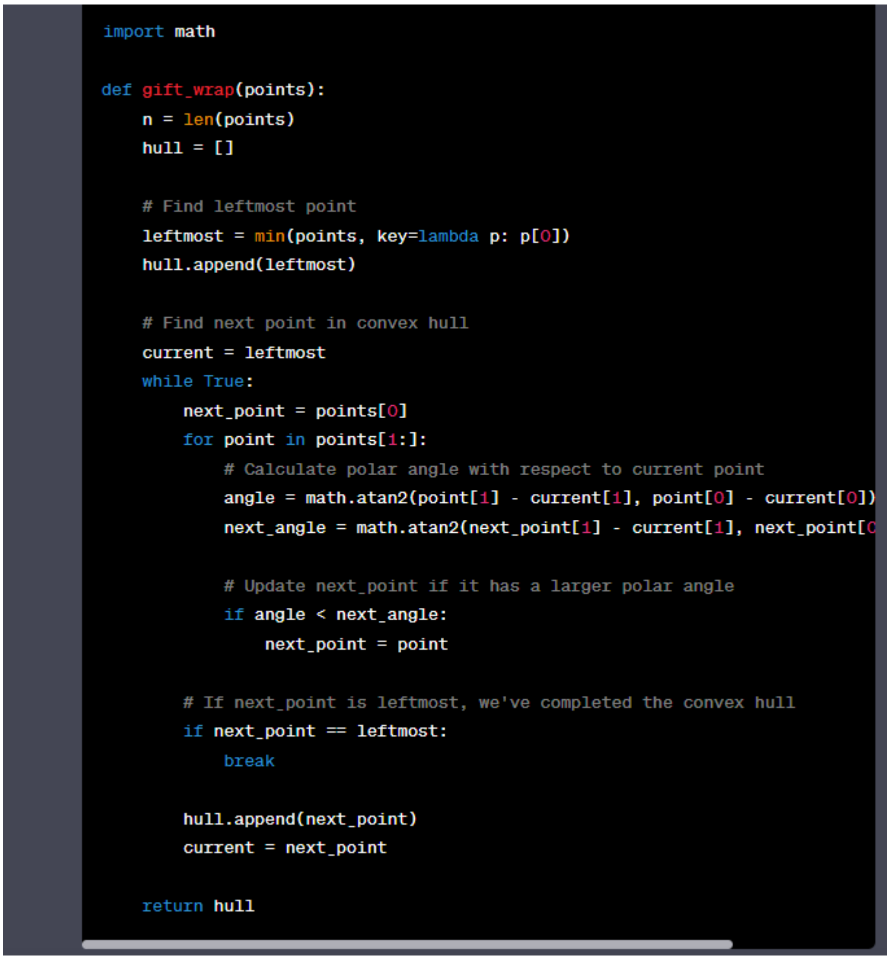
Given n points, write a code that finds a convex polygon of minimum area containing all given points.

To solve this problem, we can use the gift wrapping algorithm, also known as the Jarvis march algorithm. This algorithm works by starting at the leftmost point and then iteratively finding the point that is farthest to the right from the current point. This process is repeated until we reach the starting point again, forming a convex hull.

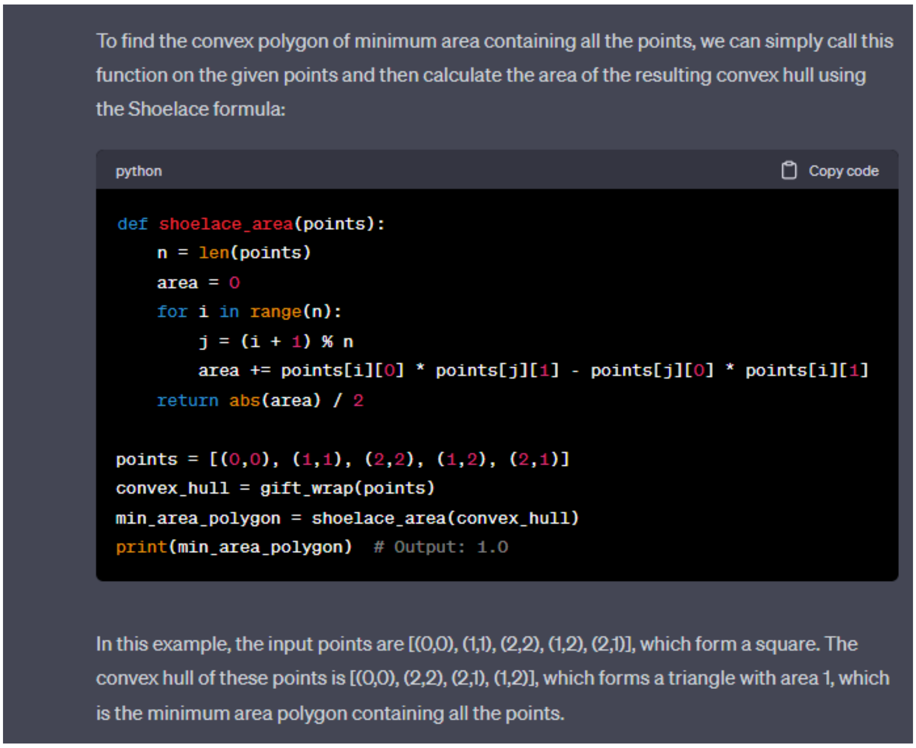
To find the leftmost point, we can simply iterate through all the points and keep track of the one with the smallest x-coordinate. From there, we can iterate through the points again and find the next point in the convex hull by finding the one that has the largest polar angle with respect to the current point. We can use the cross product to calculate the polar angle.

Once we have found the next point, we can repeat the process with this new point as the current point until we reach the starting point again.

Here's an implementation of the algorithm in Python:



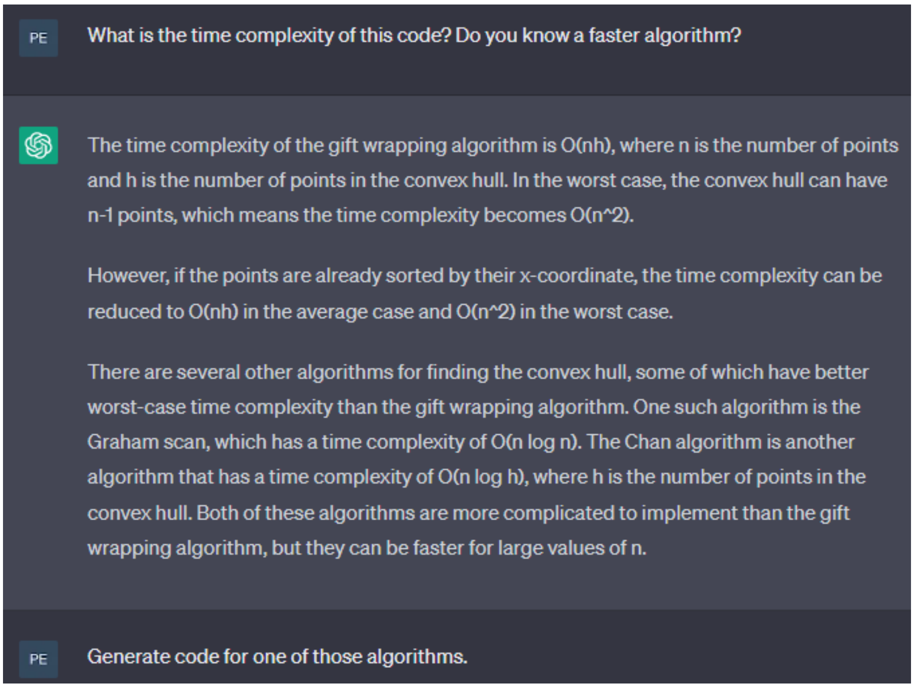
```
import math

def gift_wrap(points):
    n = len(points)
    hull = []

    # Find leftmost point
    leftmost = min(points, key=lambda p: p[0])
    hull.append(leftmost)

    # Find next point in convex hull
    current = leftmost
    while True:
        next_point = points[0]
        for point in points[1:]:
            # Calculate polar angle with respect to current point
            angle = math.atan2(point[1] - current[1], point[0] - current[0])
            next_angle = math.atan2(next_point[1] - current[1], next_point[

            # Update next_point if it has a larger polar angle
            if angle < next_angle:
                next_point = point

        # If next_point is leftmost, we've completed the convex hull
        if next_point == leftmost:
            break

        hull.append(next_point)
        current = next_point

    return hull
```



To find the convex polygon of minimum area containing all the points, we can simply call this function on the given points and then calculate the area of the resulting convex hull using the Shoelace formula:

```python
def shoelace_area(points):
    n = len(points)
    area = 0
    for i in range(n):
        j = (i + 1) % n
        area += points[i][0] * points[j][1] - points[j][0] * points[i][1]
    return abs(area) / 2

points = [(0,0), (1,1), (2,2), (1,2), (2,1)]
convex_hull = gift_wrap(points)
min_area_polygon = shoelace_area(convex_hull)
print(min_area_polygon)  # Output: 1.0
```

In this example, the input points are [(0,0), (1,1), (2,2), (1,2), (2,1)], which form a square. The convex hull of these points is [(0,0), (2,2), (2,1), (1,2)], which forms a triangle with area 1, which is the minimum area polygon containing all the points.



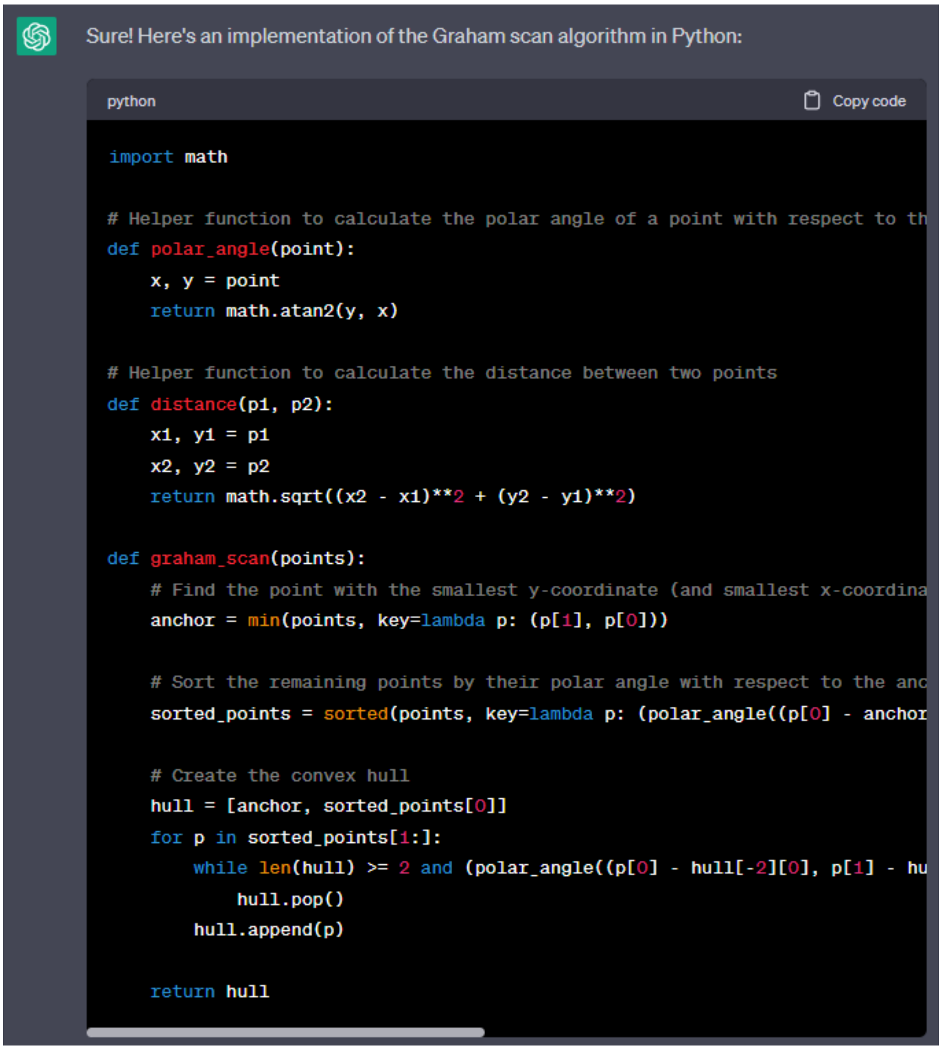
What is the time complexity of this code? Do you know a faster algorithm?

The time complexity of the gift wrapping algorithm is O(nh), where n is the number of points and h is the number of points in the convex hull. In the worst case, the convex hull can have n-1 points, which means the time complexity becomes O(n^2).

However, if the points are already sorted by their x-coordinate, the time complexity can be reduced to O(nh) in the average case and O(n^2) in the worst case.

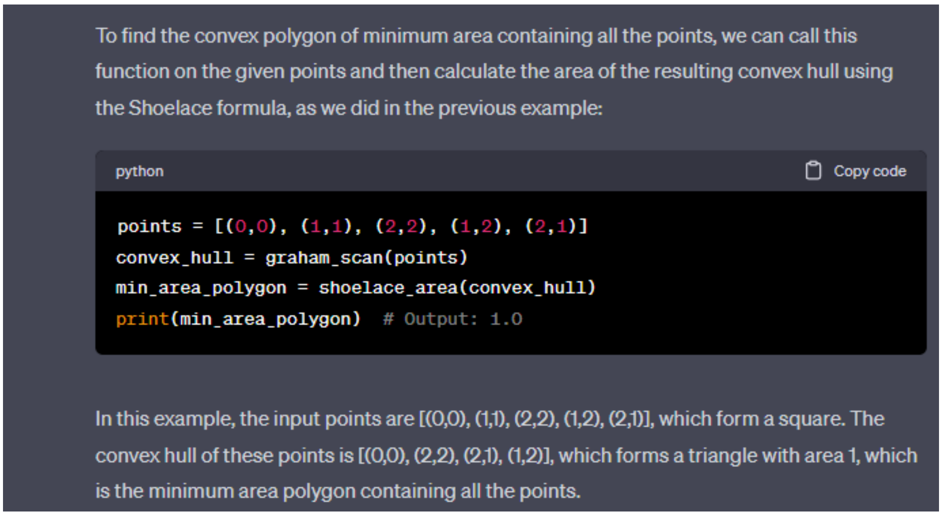
There are several other algorithms for finding the convex hull, some of which have better worst-case time complexity than the gift wrapping algorithm. One such algorithm is the Graham scan, which has a time complexity of O(n log n). The Chan algorithm is another algorithm that has a time complexity of O(n log h), where h is the number of points in the convex hull. Both of these algorithms are more complicated to implement than the gift wrapping algorithm, but they can be faster for large values of n.

Generate code for one of those algorithms.



Sure! Here's an implementation of the Graham scan algorithm in Python:

```python
import math

# Helper function to calculate the polar angle of a point with respect to th
def polar_angle(point):
    x, y = point
    return math.atan2(y, x)

# Helper function to calculate the distance between two points
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

def graham_scan(points):
    # Find the point with the smallest y-coordinate (and smallest x-coordina
    anchor = min(points, key=lambda p: (p[1], p[0]))

    # Sort the remaining points by their polar angle with respect to the anc
    sorted_points = sorted(points, key=lambda p: (polar_angle((p[0] - anchor

    # Create the convex hull
    hull = [anchor, sorted_points[0]]
    for p in sorted_points[1:]:
        while len(hull) >= 2 and (polar_angle((p[0] - hull[-2][0], p[1] - hu
            hull.pop()
        hull.append(p)

    return hull
```

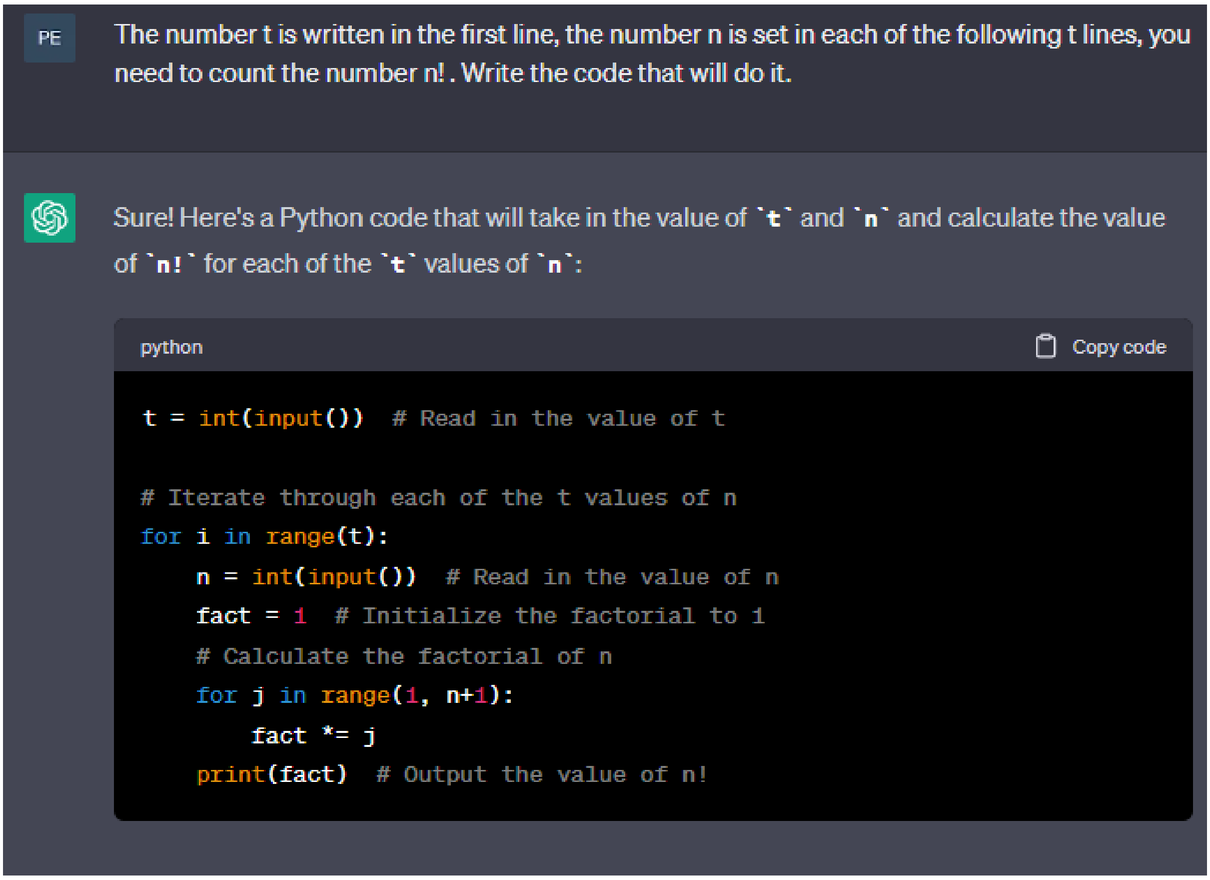


To find the convex polygon of minimum area containing all the points, we can call this function on the given points and then calculate the area of the resulting convex hull using the Shoelace formula, as we did in the previous example:

```python
points = [(0,0), (1,1), (2,2), (1,2), (2,1)]
convex_hull = graham_scan(points)
min_area_polygon = shoelace_area(convex_hull)
print(min_area_polygon)  # Output: 1.0
```

In this example, the input points are [(0,0), (1,1), (2,2), (1,2), (2,1)], which form a square. The convex hull of these points is [(0,0), (2,2), (2,1), (1,2)], which forms a triangle with area 1, which is the minimum area polygon containing all the points.

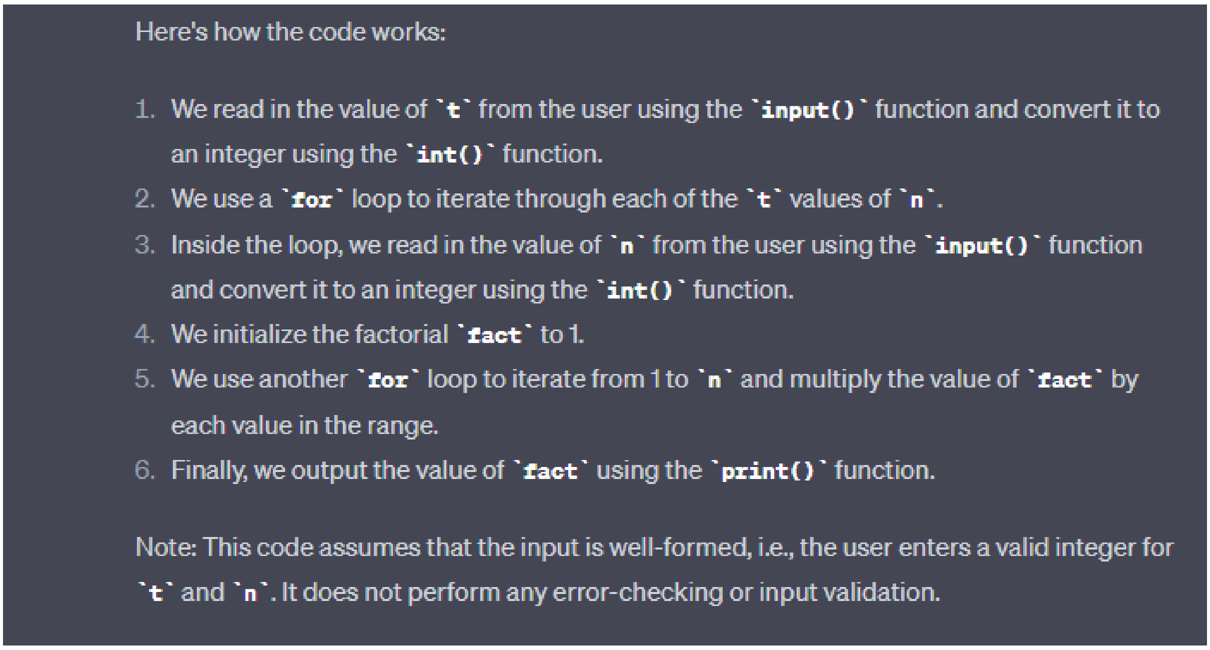
Обрахунок факторіала числа



The number t is written in the first line, the number n is set in each of the following t lines, you need to count the number n! . Write the code that will do it.

Sure! Here's a Python code that will take in the value of `t` and `n` and calculate the value of `n!` for each of the `t` values of `n`:

```python
t = int(input())  # Read in the value of t

# Iterate through each of the t values of n
for i in range(t):
    n = int(input())  # Read in the value of n
    fact = 1  # Initialize the factorial to 1
    # Calculate the factorial of n
    for j in range(1, n+1):
        fact *= j
    print(fact)  # Output the value of n!
```

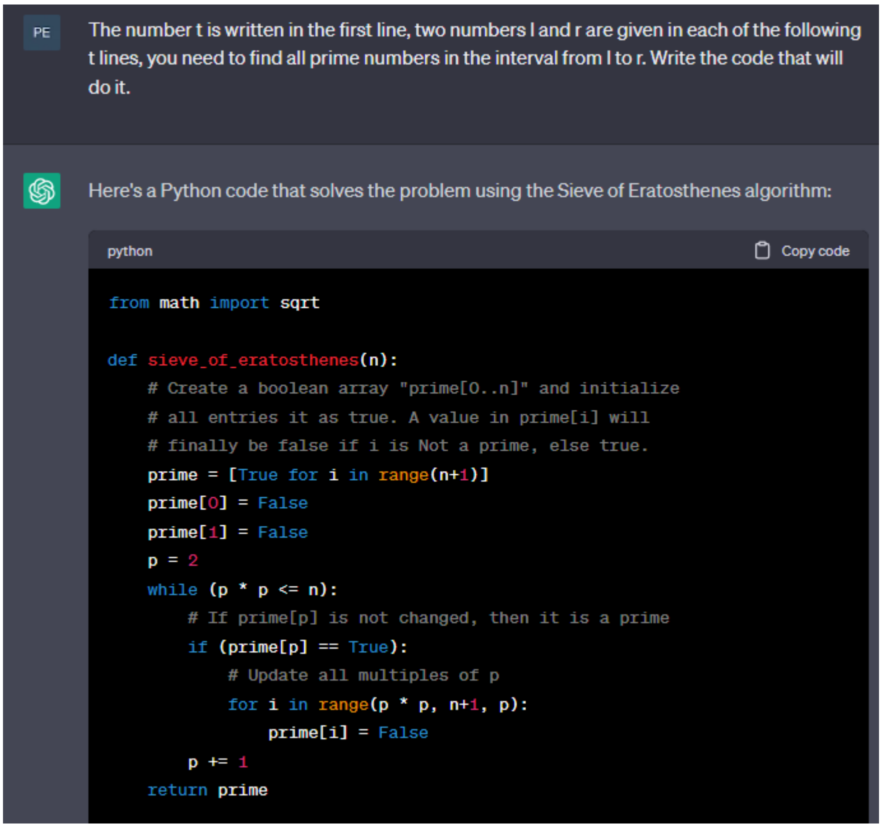


Here's how the code works:

1. We read in the value of `t` from the user using the `input()` function and convert it to an integer using the `int()` function.
2. We use a `for` loop to iterate through each of the `t` values of `n`.
3. Inside the loop, we read in the value of `n` from the user using the `input()` function and convert it to an integer using the `int()` function.
4. We initialize the factorial `fact` to 1.
5. We use another `for` loop to iterate from 1 to `n` and multiply the value of `fact` by each value in the range.
6. Finally, we output the value of `fact` using the `print()` function.

Note: This code assumes that the input is well-formed, i.e., the user enters a valid integer for `t` and `n`. It does not perform any error-checking or input validation.

Решето Ератосфена



The number t is written in the first line, two numbers l and r are given in each of the following t lines, you need to find all prime numbers in the interval from l to r. Write the code that will do it.

Here's a Python code that solves the problem using the Sieve of Eratosthenes algorithm:

```python
from math import sqrt

def sieve_of_eratosthenes(n):
    # Create a boolean array "prime[0..n]" and initialize
    # all entries it as true. A value in prime[i] will
    # finally be false if i is Not a prime, else true.
    prime = [True for i in range(n+1)]
    prime[0] = False
    prime[1] = False
    p = 2
    while (p * p <= n):
        # If prime[p] is not changed, then it is a prime
        if (prime[p] == True):
            # Update all multiples of p
            for i in range(p * p, n+1, p):
                prime[i] = False
        p += 1
    return prime
```

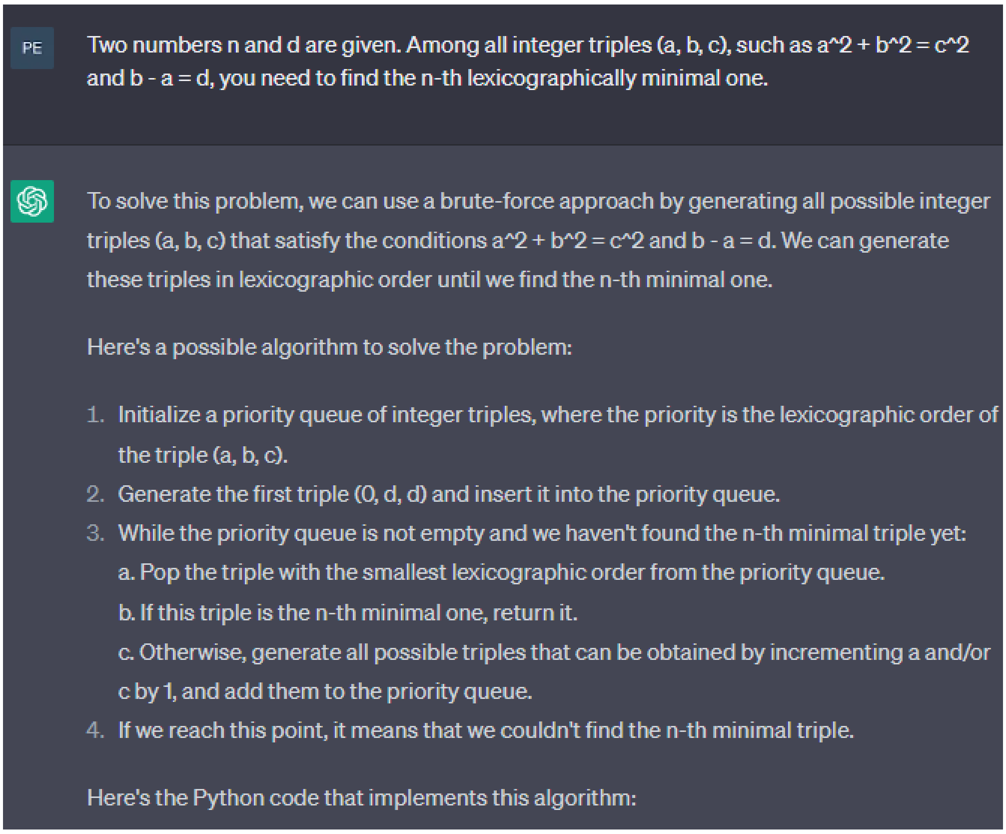
```python
t = int(input())
for _ in range(t):
    l, r = map(int, input().split())
    primes = []
    prime = sieve_of_eratosthenes(r)
    for i in range(l, r+1):
        if prime[i]:
            primes.append(i)
    print(*primes)
```

Explanation:

1. Define the Sieve of Eratosthenes algorithm that returns a list of all prime numbers up to a given integer `n`.
2. Take the input integer `t` as the number of test cases.
3. Loop through each test case and take the input integers `l` and `r`.
4. Run the Sieve of Eratosthenes algorithm on the range from 2 to `r` and store the result in the `prime` list.
5. Loop through the range from `l` to `r` and append each prime number to the `primes` list.
6. Print the `primes` list for each test case.

Note that the `print(*primes)` statement uses the `*` operator to print the list elements separated by spaces.

Пошук Піфагорової трійки



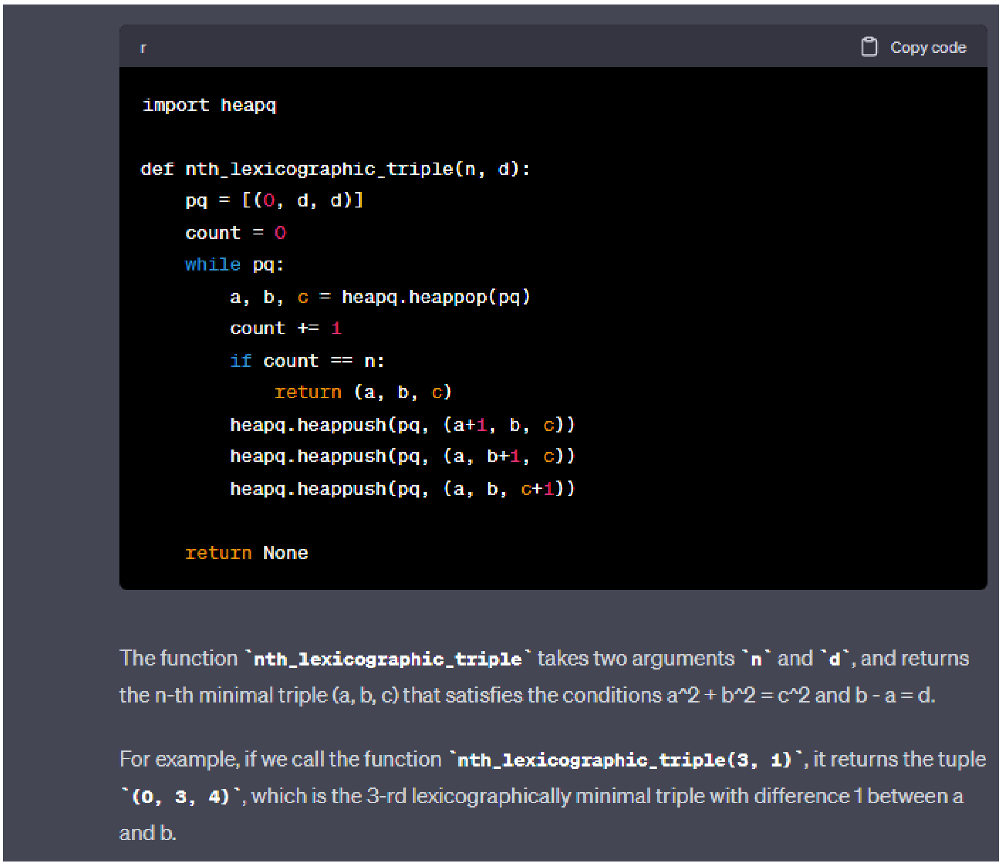
Two numbers n and d are given. Among all integer triples (a, b, c), such as a^2 + b^2 = c^2 and b - a = d, you need to find the n-th lexicographically minimal one.

To solve this problem, we can use a brute-force approach by generating all possible integer triples (a, b, c) that satisfy the conditions a^2 + b^2 = c^2 and b - a = d. We can generate these triples in lexicographic order until we find the n-th minimal one.

Here's a possible algorithm to solve the problem:

1. Initialize a priority queue of integer triples, where the priority is the lexicographic order of the triple (a, b, c).
2. Generate the first triple (0, d, d) and insert it into the priority queue.
3. While the priority queue is not empty and we haven't found the n-th minimal triple yet:
   a. Pop the triple with the smallest lexicographic order from the priority queue.
   b. If this triple is the n-th minimal one, return it.
   c. Otherwise, generate all possible triples that can be obtained by incrementing a and/or c by 1, and add them to the priority queue.
4. If we reach this point, it means that we couldn't find the n-th minimal triple.

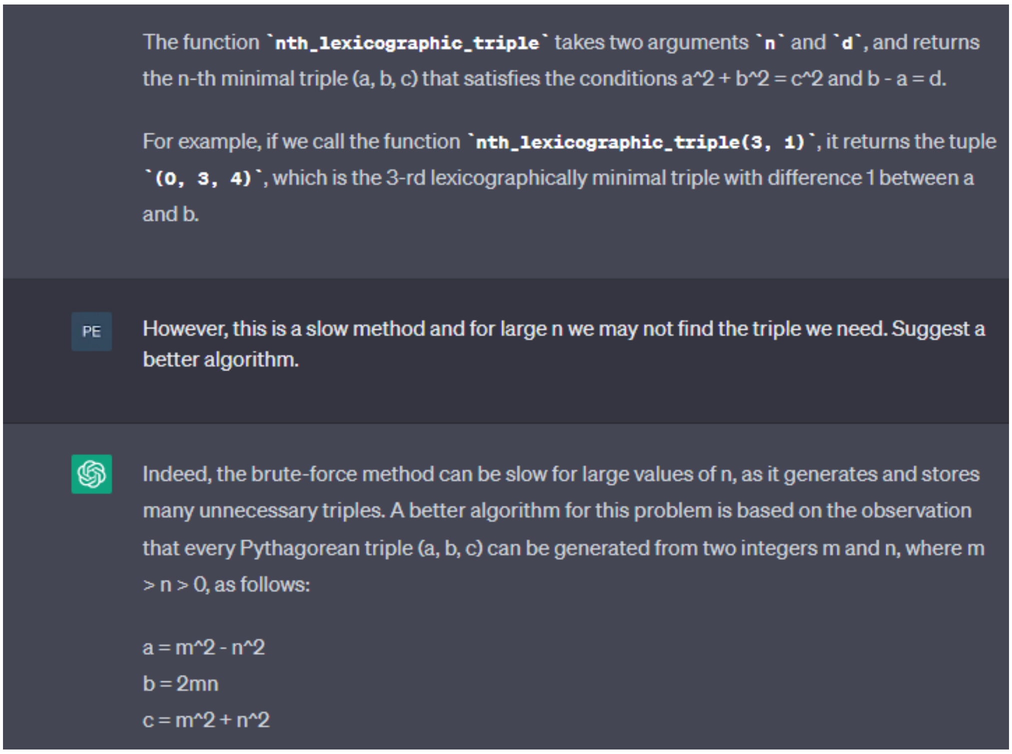
Here's the Python code that implements this algorithm:



```
import heapq

def nth_lexicographic_triple(n, d):
    pq = [(0, d, d)]
    count = 0
    while pq:
        a, b, c = heapq.heappop(pq)
        count += 1
        if count == n:
            return (a, b, c)
        heapq.heappush(pq, (a+1, b, c))
        heapq.heappush(pq, (a, b+1, c))
        heapq.heappush(pq, (a, b, c+1))

    return None
```

The function `nth_lexicographic_triple` takes two arguments `n` and `d`, and returns the n-th minimal triple (a, b, c) that satisfies the conditions a^2 + b^2 = c^2 and b - a = d.

For example, if we call the function `nth_lexicographic_triple(3, 1)`, it returns the tuple `(0, 3, 4)`, which is the 3-rd lexicographically minimal triple with difference 1 between a and b.



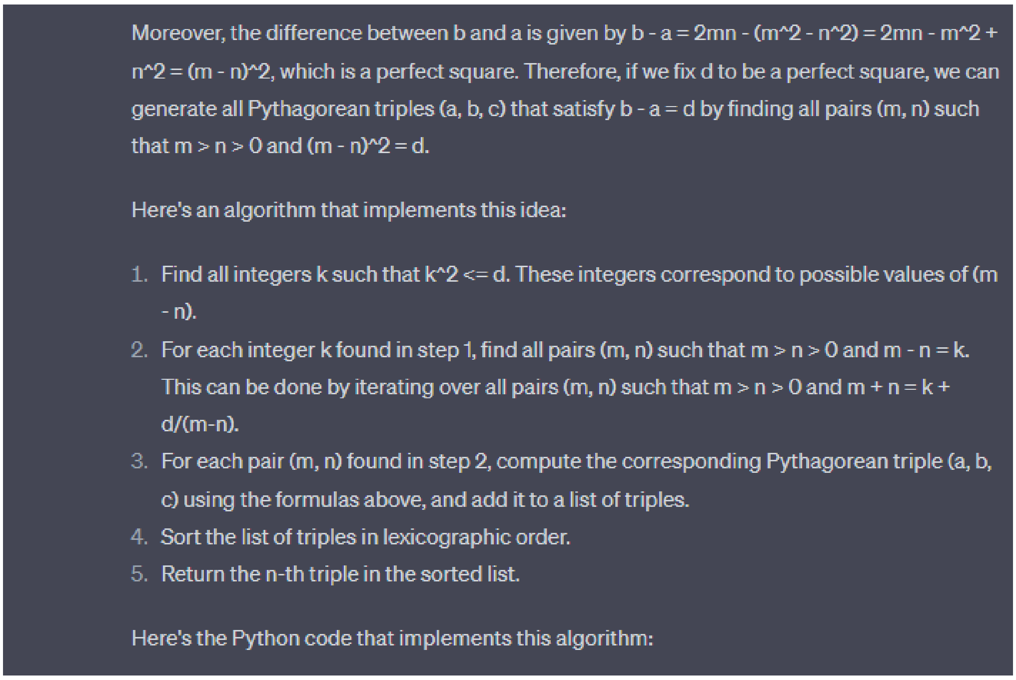
The function `nth_lexicographic_triple` takes two arguments `n` and `d`, and returns the n-th minimal triple (a, b, c) that satisfies the conditions a^2 + b^2 = c^2 and b - a = d.

For example, if we call the function `nth_lexicographic_triple(3, 1)`, it returns the tuple `(0, 3, 4)`, which is the 3-rd lexicographically minimal triple with difference 1 between a and b.

However, this is a slow method and for large n we may not find the triple we need. Suggest a better algorithm.

Indeed, the brute-force method can be slow for large values of n, as it generates and stores many unnecessary triples. A better algorithm for this problem is based on the observation that every Pythagorean triple (a, b, c) can be generated from two integers m and n, where m > n > 0, as follows:

a = m^2 - n^2
b = 2mn
c = m^2 + n^2

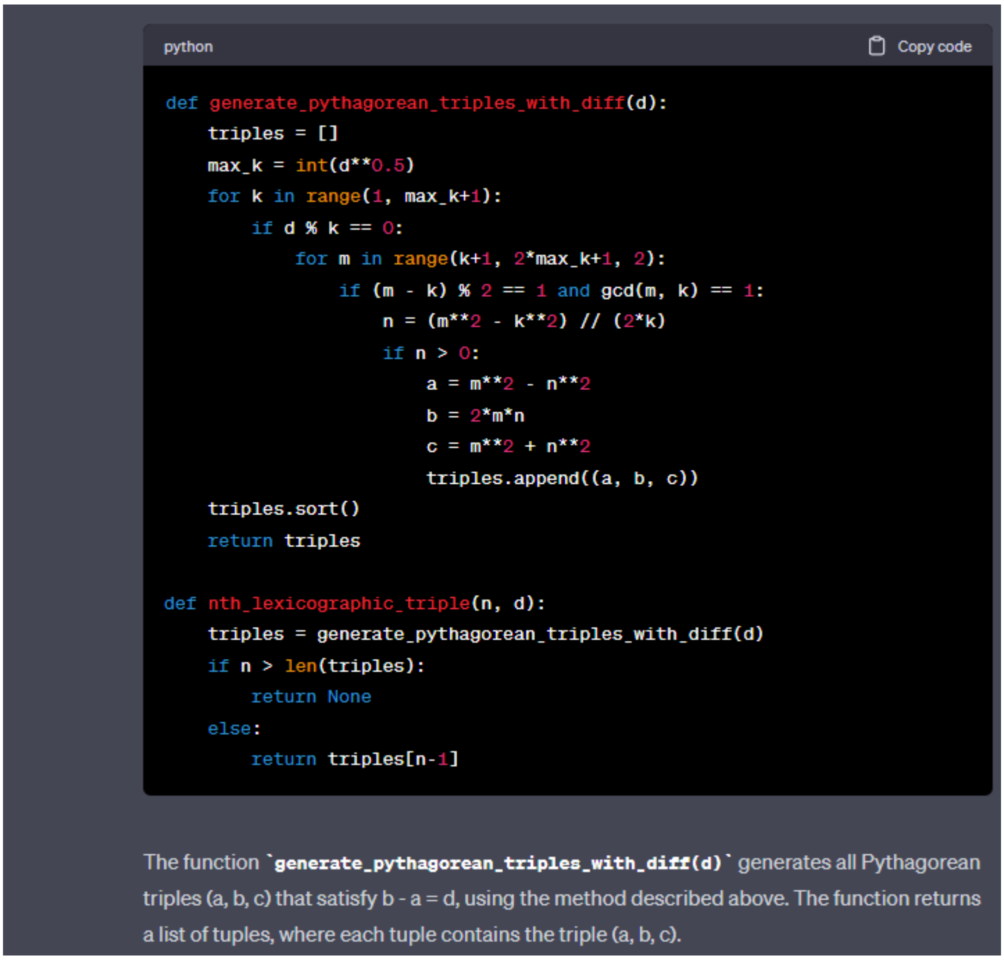


Moreover, the difference between b and a is given by b - a = 2mn - (m^2 - n^2) = 2mn - m^2 + n^2 = (m - n)^2, which is a perfect square. Therefore, if we fix d to be a perfect square, we can generate all Pythagorean triples (a, b, c) that satisfy b - a = d by finding all pairs (m, n) such that m > n > 0 and (m - n)^2 = d.

Here's an algorithm that implements this idea:

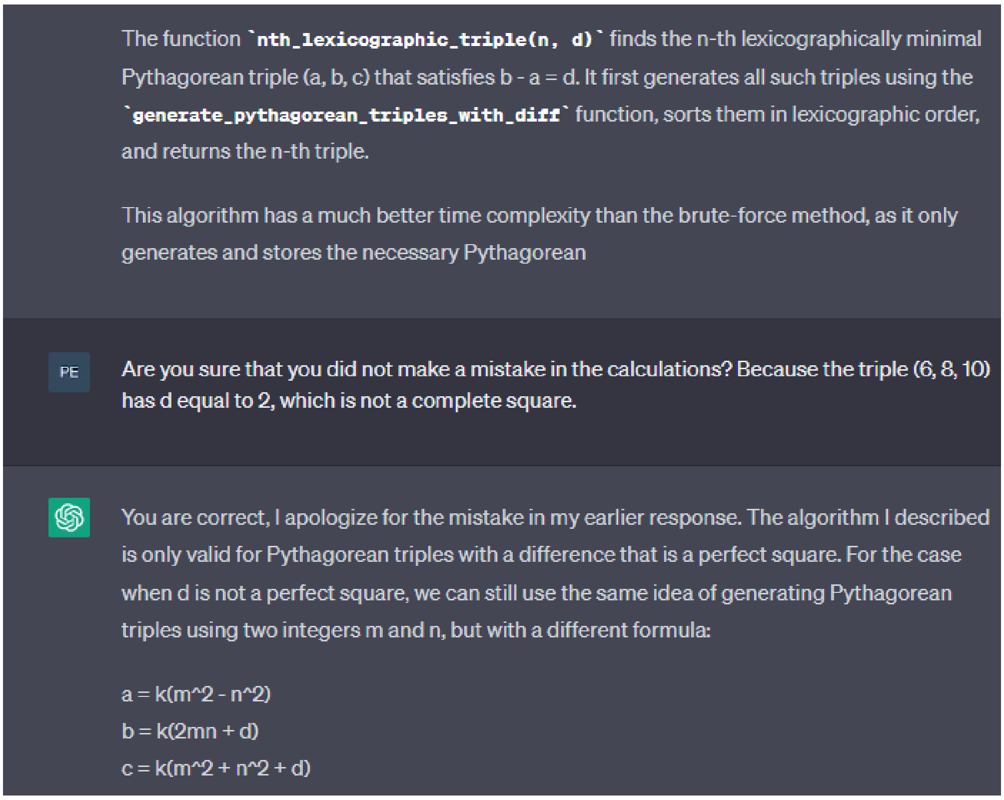
1. Find all integers k such that k^2 <= d. These integers correspond to possible values of (m - n).
2. For each integer k found in step 1, find all pairs (m, n) such that m > n > 0 and m - n = k. This can be done by iterating over all pairs (m, n) such that m > n > 0 and m + n = k + d/(m-n).
3. For each pair (m, n) found in step 2, compute the corresponding Pythagorean triple (a, b, c) using the formulas above, and add it to a list of triples.
4. Sort the list of triples in lexicographic order.
5. Return the n-th triple in the sorted list.

Here's the Python code that implements this algorithm:



```python
def generate_pythagorean_triples_with_diff(d):
    triples = []
    max_k = int(d**0.5)
    for k in range(1, max_k+1):
        if d % k == 0:
            for m in range(k+1, 2*max_k+1, 2):
                if (m - k) % 2 == 1 and gcd(m, k) == 1:
                    n = (m**2 - k**2) // (2*k)
                    if n > 0:
                        a = m**2 - n**2
                        b = 2*m*n
                        c = m**2 + n**2
                        triples.append((a, b, c))
    triples.sort()
    return triples

def nth_lexicographic_triple(n, d):
    triples = generate_pythagorean_triples_with_diff(d)
    if n > len(triples):
        return None
    else:
        return triples[n-1]
```

The function `generate_pythagorean_triples_with_diff(d)` generates all Pythagorean triples (a, b, c) that satisfy b - a = d, using the method described above. The function returns a list of tuples, where each tuple contains the triple (a, b, c).



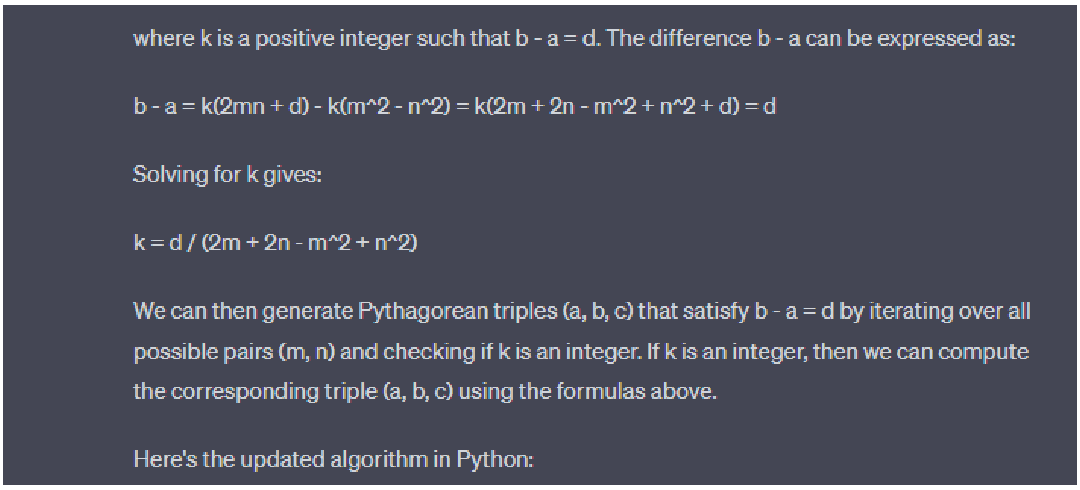
The function `nth_lexicographic_triple(n, d)` finds the n-th lexicographically minimal Pythagorean triple (a, b, c) that satisfies b - a = d. It first generates all such triples using the `generate_pythagorean_triples_with_diff` function, sorts them in lexicographic order, and returns the n-th triple.

This algorithm has a much better time complexity than the brute-force method, as it only generates and stores the necessary Pythagorean

Are you sure that you did not make a mistake in the calculations? Because the triple (6, 8, 10) has d equal to 2, which is not a complete square.

You are correct, I apologize for the mistake in my earlier response. The algorithm I described is only valid for Pythagorean triples with a difference that is a perfect square. For the case when d is not a perfect square, we can still use the same idea of generating Pythagorean triples using two integers m and n, but with a different formula:

a = k(m^2 - n^2)
b = k(2mn + d)
c = k(m^2 + n^2 + d)



where k is a positive integer such that b - a = d. The difference b - a can be expressed as:

b - a = k(2mn + d) - k(m^2 - n^2) = k(2m + 2n - m^2 + n^2 + d) = d

Solving for k gives:

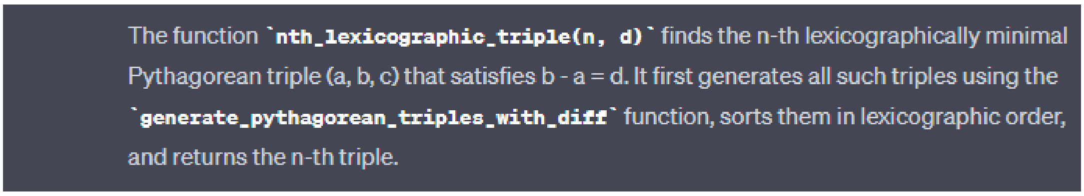
k = d / (2m + 2n - m^2 + n^2)

We can then generate Pythagorean triples (a, b, c) that satisfy b - a = d by iterating over all possible pairs (m, n) and checking if k is an integer. If k is an integer, then we can compute the corresponding triple (a, b, c) using the formulas above.

Here's the updated algorithm in Python:

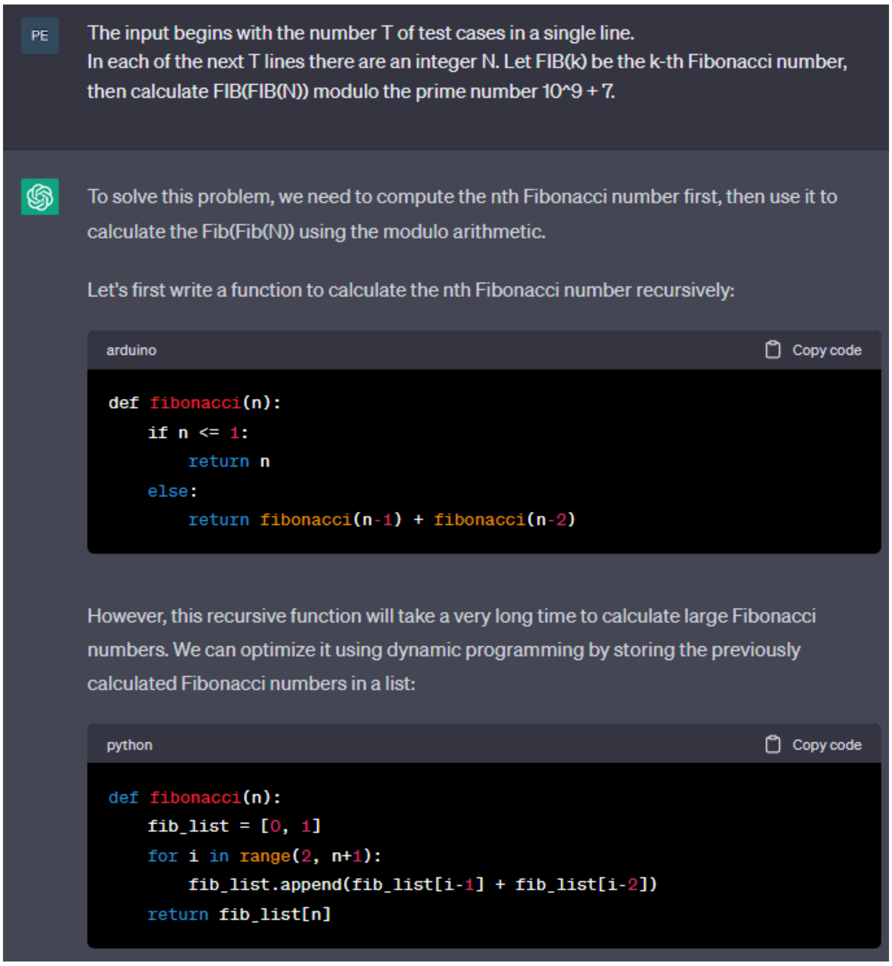
```python
from math import gcd

def generate_pythagorean_triples_with_diff(d):
    triples = []
    max_m = int((d/2)**0.5) + 1
    for m in range(2, max_m):
        for n in range(1, m):
            if (m - n) % 2 == 1 and gcd(m, n) == 1:
                k = d / (2*m + 2*n - m**2 + n**2)
                if k.is_integer():
                    a = int(k * (m**2 - n**2))
                    b = int(k * (2*m*n + d))
                    c = int(k * (m**2 + n**2 + d))
                    triples.append((a, b, c))
    triples.sort()
    return triples

def nth_lexicographic_triple(n, d):
    triples = generate_pythagorean_triples_with_diff(d)
    if n > len(triples):
        return None
    else:
        return triples[n-1]
```

The function **`generate_pythagorean_triples_with_diff(d)`** generates all Pythagorean triples (a, b, c) that satisfy b - a = d, using the method described above. The function returns a list of tuples, where each tuple contains the triple (a, b, c).



The function **`nth_lexicographic_triple(n, d)`** finds the n-th lexicographically minimal Pythagorean triple (a, b, c) that satisfies b - a = d. It first generates all such triples using the **`generate_pythagorean_triples_with_diff`** function, sorts them in lexicographic order, and returns the n-th triple.

Задача про числа Фібоначчі



The input begins with the number T of test cases in a single line.
In each of the next T lines there are an integer N. Let FIB(k) be the k-th Fibonacci number, then calculate FIB(FIB(N)) modulo the prime number 10^9 + 7.

To solve this problem, we need to compute the nth Fibonacci number first, then use it to calculate the Fib(Fib(N)) using the modulo arithmetic.

Let's first write a function to calculate the nth Fibonacci number recursively:

```arduino
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

However, this recursive function will take a very long time to calculate large Fibonacci numbers. We can optimize it using dynamic programming by storing the previously calculated Fibonacci numbers in a list:

```python
def fibonacci(n):
    fib_list = [0, 1]
    for i in range(2, n+1):
        fib_list.append(fib_list[i-1] + fib_list[i-2])
    return fib_list[n]
```

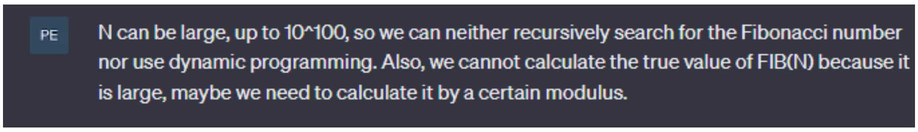
Now that we have a function to calculate the nth Fibonacci number efficiently, we can use it to calculate Fib(Fib(N)) using modulo arithmetic:

```scss
MOD = 10**9 + 7

def fib_fib(n):
    fib_n = fibonacci(n)
    return fibonacci(fib_n) % MOD
```

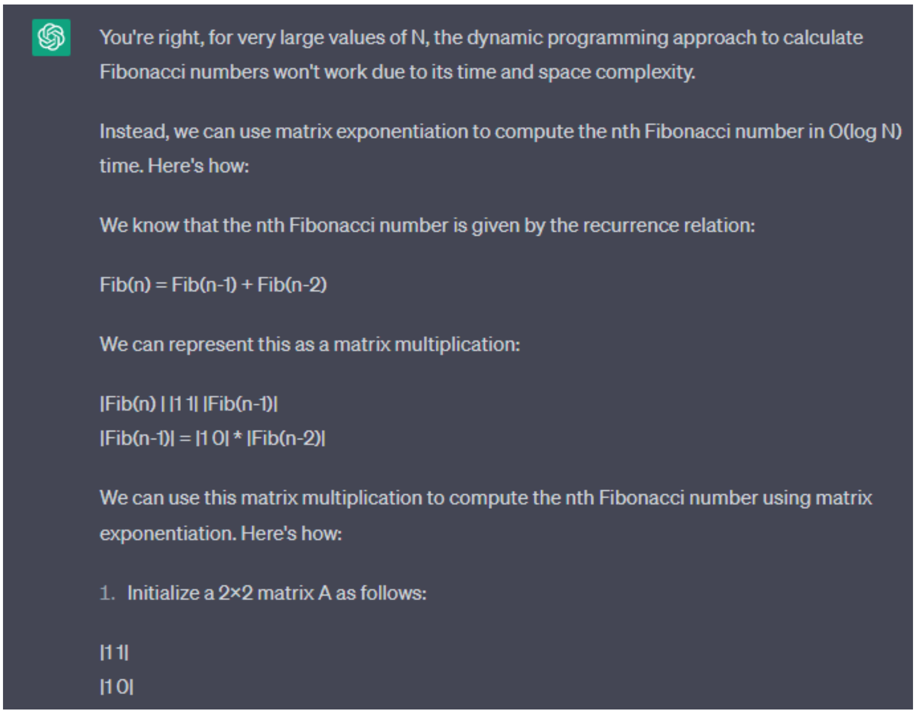
Finally, we can write a main function to read the input and call the `fib_fib` function for each test case:

```css
if __name__ == '__main__':
    T = int(input())
    for _ in range(T):
        N = int(input())
        result = fib_fib(N)
        print(result)
```

This solution should be able to handle large values of N efficiently using dynamic programming and modulo arithmetic.



N can be large, up to 10^100, so we can neither recursively search for the Fibonacci number nor use dynamic programming. Also, we cannot calculate the true value of FIB(N) because it is large, maybe we need to calculate it by a certain modulus.



You're right, for very large values of N, the dynamic programming approach to calculate Fibonacci numbers won't work due to its time and space complexity.

Instead, we can use matrix exponentiation to compute the nth Fibonacci number in O(log N) time. Here's how:

We know that the nth Fibonacci number is given by the recurrence relation:

Fib(n) = Fib(n-1) + Fib(n-2)

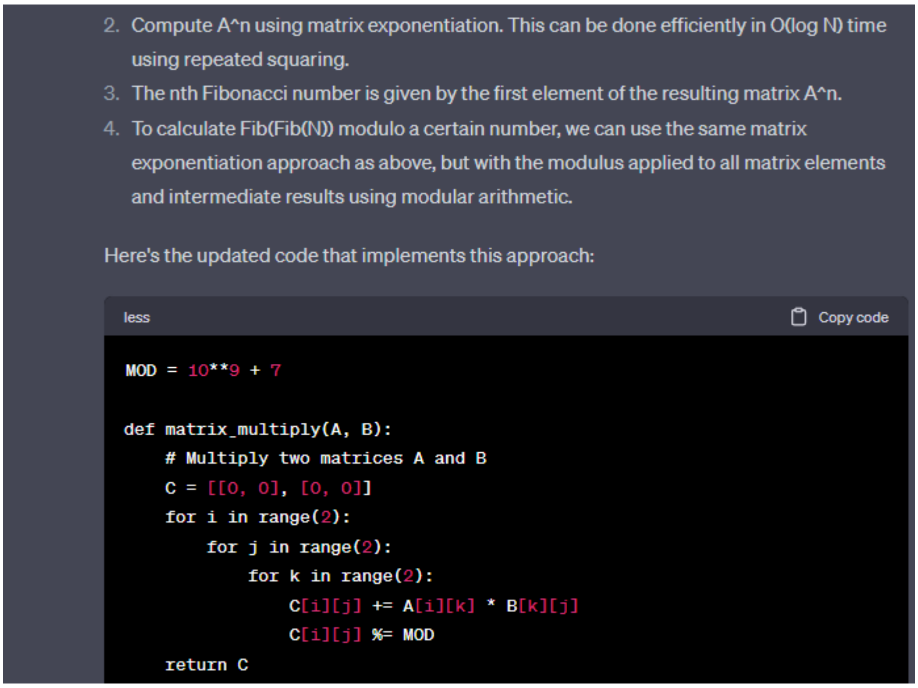
We can represent this as a matrix multiplication:

|Fib(n) | |1 1| |Fib(n-1)|
|Fib(n-1)| = |1 0| * |Fib(n-2)|

We can use this matrix multiplication to compute the nth Fibonacci number using matrix exponentiation. Here's how:

1. Initialize a 2x2 matrix A as follows:

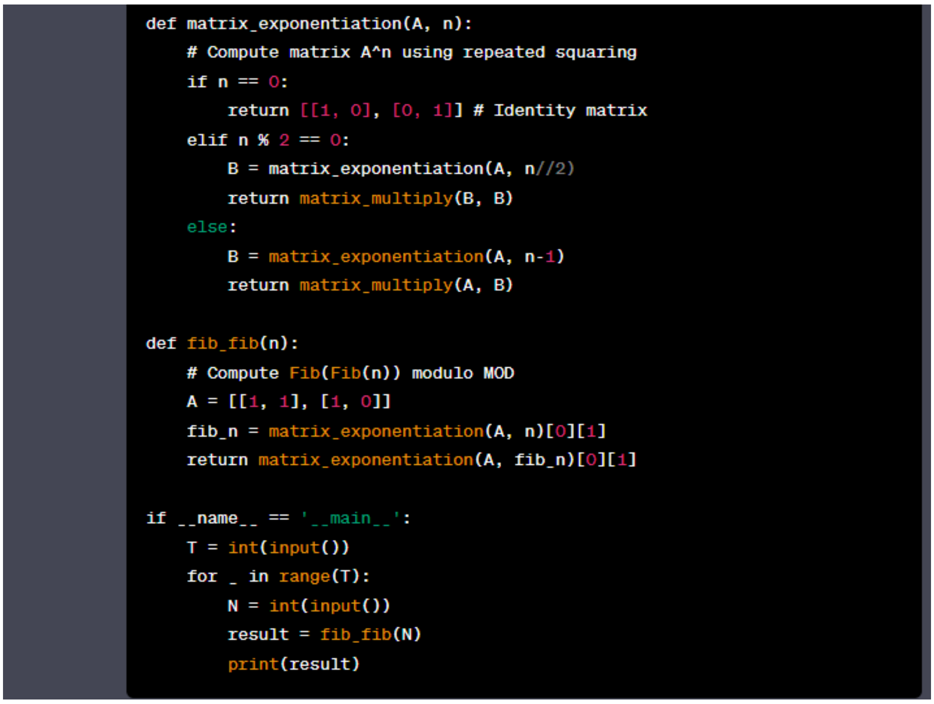
|1 1|
|1 0|



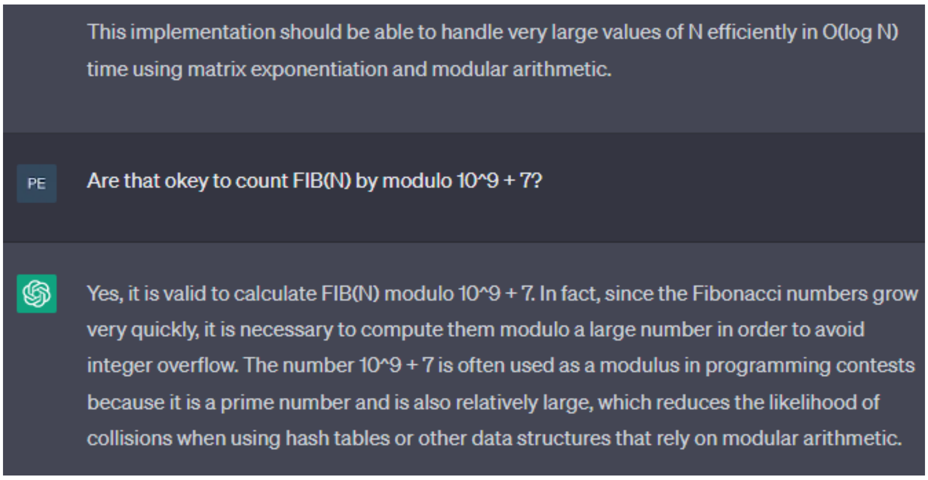
2. Compute A^n using matrix exponentiation. This can be done efficiently in O(log N) time using repeated squaring.
3. The nth Fibonacci number is given by the first element of the resulting matrix A^n.
4. To calculate Fib(Fib(N)) modulo a certain number, we can use the same matrix exponentiation approach as above, but with the modulus applied to all matrix elements and intermediate results using modular arithmetic.

Here's the updated code that implements this approach:

```
MOD = 10**9 + 7

def matrix_multiply(A, B):
    # Multiply two matrices A and B
    C = [[0, 0], [0, 0]]
    for i in range(2):
        for j in range(2):
            for k in range(2):
                C[i][j] += A[i][k] * B[k][j]
                C[i][j] %= MOD
    return C
```



```
def matrix_exponentiation(A, n):
    # Compute matrix A^n using repeated squaring
    if n == 0:
        return [[1, 0], [0, 1]] # Identity matrix
    elif n % 2 == 0:
        B = matrix_exponentiation(A, n//2)
        return matrix_multiply(B, B)
    else:
        B = matrix_exponentiation(A, n-1)
        return matrix_multiply(A, B)

def fib_fib(n):
    # Compute Fib(Fib(n)) modulo MOD
    A = [[1, 1], [1, 0]]
    fib_n = matrix_exponentiation(A, n)[0][1]
    return matrix_exponentiation(A, fib_n)[0][1]

if __name__ == '__main__':
    T = int(input())
    for _ in range(T):
        N = int(input())
        result = fib_fib(N)
        print(result)
```

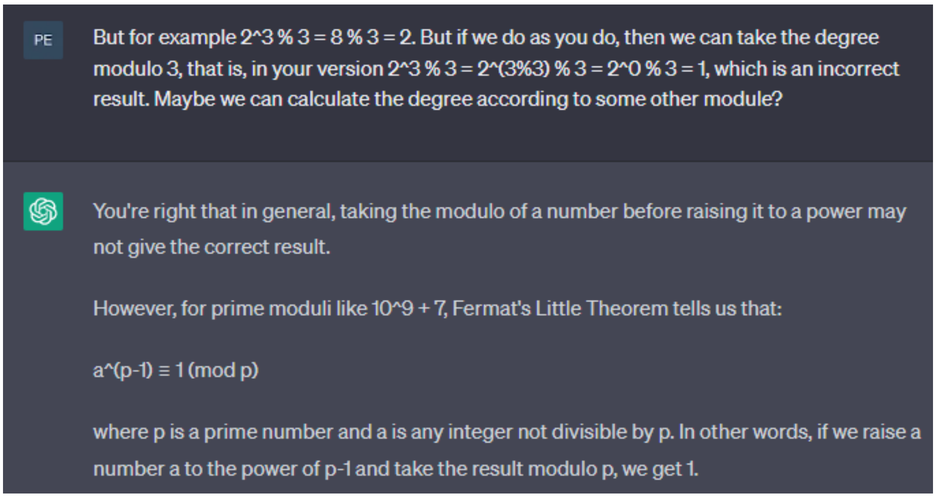


This implementation should be able to handle very large values of N efficiently in O(log N) time using matrix exponentiation and modular arithmetic.

PE

Are that okey to count FIB(N) by modulo 10^9 + 7?

Yes, it is valid to calculate FIB(N) modulo 10^9 + 7. In fact, since the Fibonacci numbers grow very quickly, it is necessary to compute them modulo a large number in order to avoid integer overflow. The number 10^9 + 7 is often used as a modulus in programming contests because it is a prime number and is also relatively large, which reduces the likelihood of collisions when using hash tables or other data structures that rely on modular arithmetic.



PE

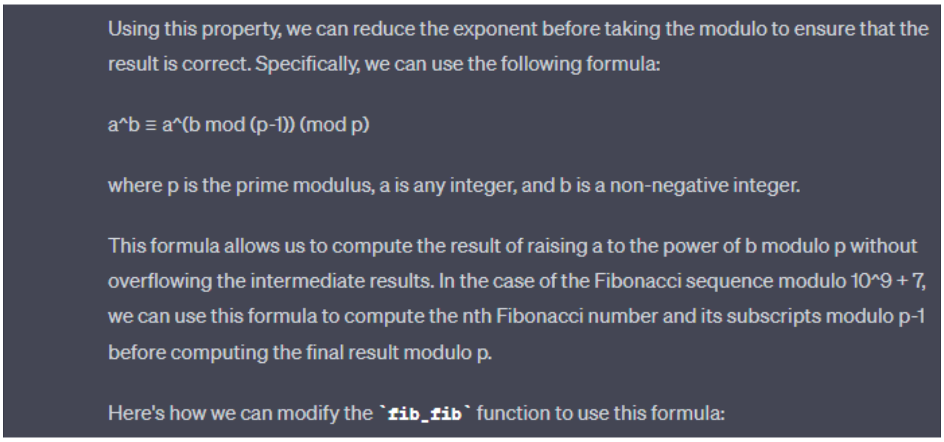
But for example 2^3 % 3 = 8 % 3 = 2. But if we do as you do, then we can take the degree modulo 3, that is, in your version 2^3 % 3 = 2^(3%3) % 3 = 2^0 % 3 = 1, which is an incorrect result. Maybe we can calculate the degree according to some other module?

You're right that in general, taking the modulo of a number before raising it to a power may not give the correct result.

However, for prime moduli like 10^9 + 7, Fermat's Little Theorem tells us that:

a^(p-1) ≡ 1 (mod p)

where p is a prime number and a is any integer not divisible by p. In other words, if we raise a number a to the power of p-1 and take the result modulo p, we get 1.



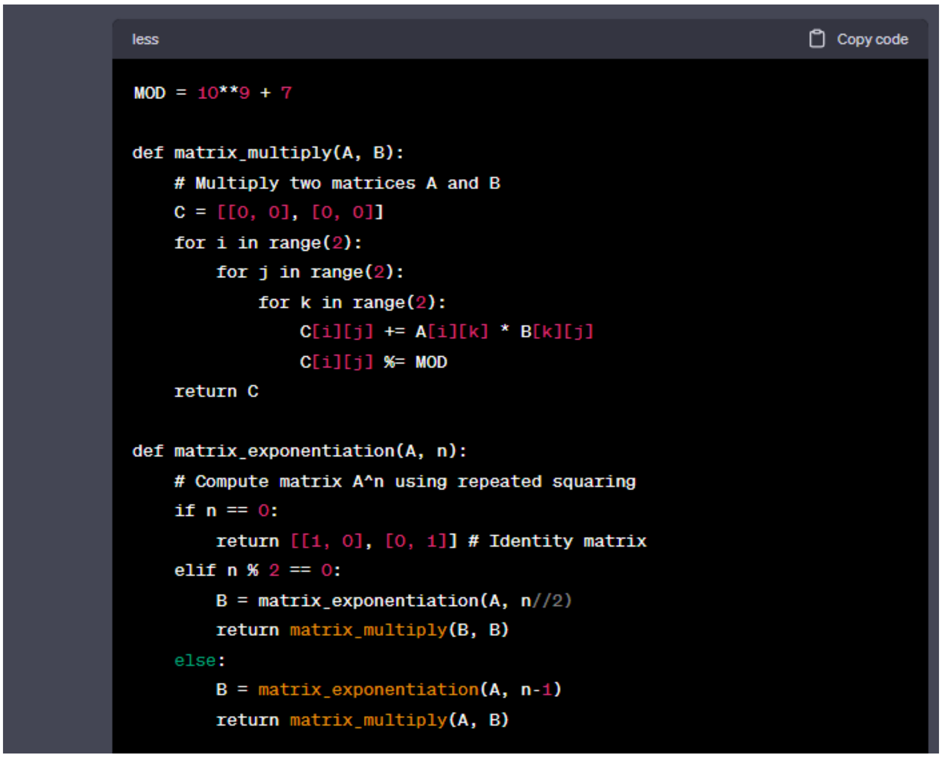
Using this property, we can reduce the exponent before taking the modulo to ensure that the result is correct. Specifically, we can use the following formula:

a^b ≡ a^(b mod (p-1)) (mod p)

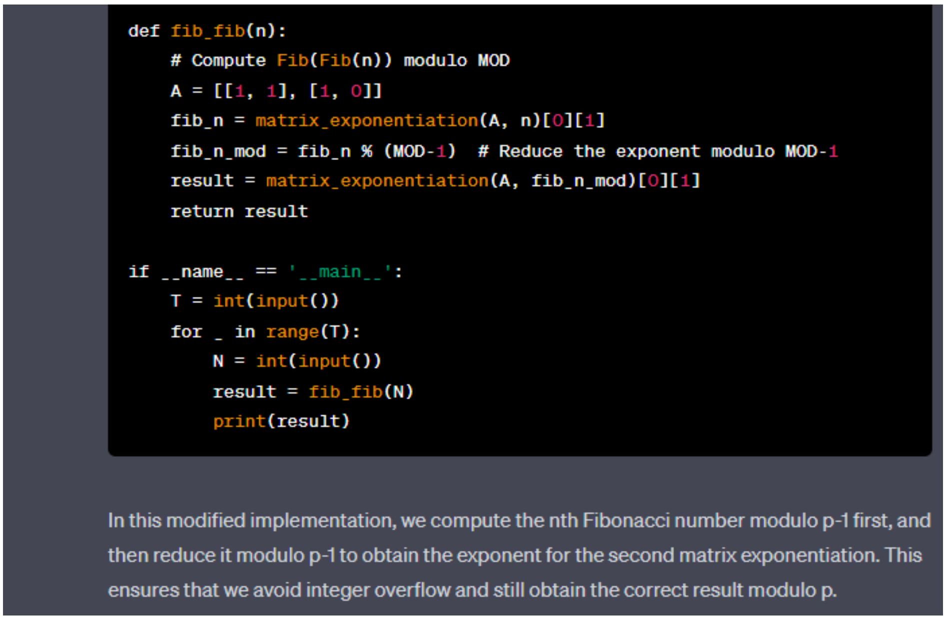
where p is the prime modulus, a is any integer, and b is a non-negative integer.

This formula allows us to compute the result of raising a to the power of b modulo p without overflowing the intermediate results. In the case of the Fibonacci sequence modulo 10^9 + 7, we can use this formula to compute the nth Fibonacci number and its subscripts modulo p-1 before computing the final result modulo p.

Here's how we can modify the `fib_fib` function to use this formula:



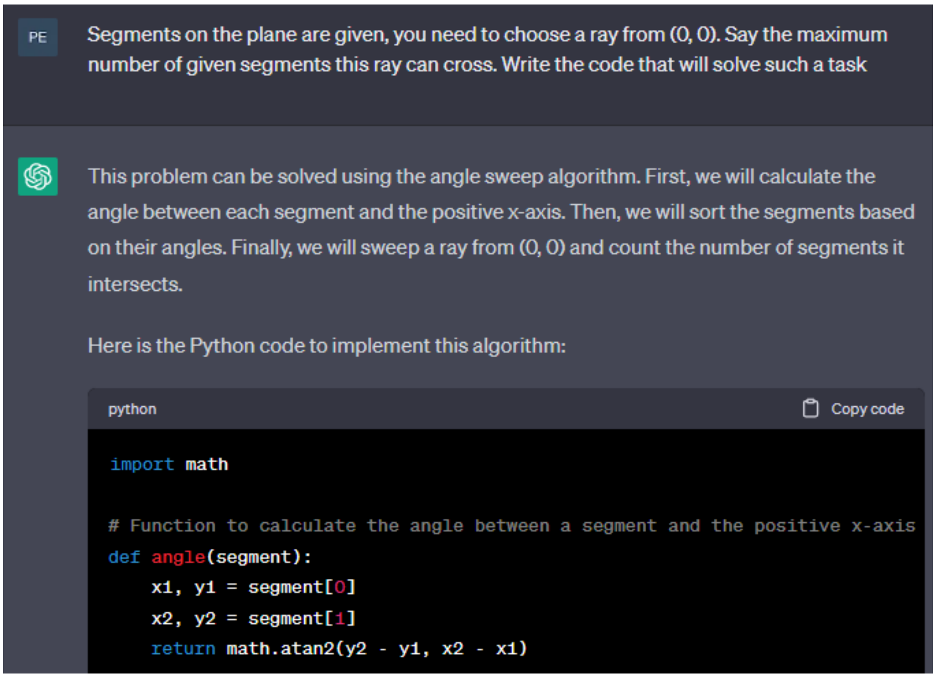
```
MOD = 10**9 + 7

def matrix_multiply(A, B):
    # Multiply two matrices A and B
    C = [[0, 0], [0, 0]]
    for i in range(2):
        for j in range(2):
            for k in range(2):
                C[i][j] += A[i][k] * B[k][j]
                C[i][j] %= MOD
    return C

def matrix_exponentiation(A, n):
    # Compute matrix A^n using repeated squaring
    if n == 0:
        return [[1, 0], [0, 1]] # Identity matrix
    elif n % 2 == 0:
        B = matrix_exponentiation(A, n//2)
        return matrix_multiply(B, B)
    else:
        B = matrix_exponentiation(A, n-1)
        return matrix_multiply(A, B)
```



```
def fib_fib(n):
    # Compute Fib(Fib(n)) modulo MOD
    A = [[1, 1], [1, 0]]
    fib_n = matrix_exponentiation(A, n)[0][1]
    fib_n_mod = fib_n % (MOD-1)  # Reduce the exponent modulo MOD-1
    result = matrix_exponentiation(A, fib_n_mod)[0][1]
    return result

if __name__ == '__main__':
    T = int(input())
    for _ in range(T):
        N = int(input())
        result = fib_fib(N)
        print(result)
```

In this modified implementation, we compute the nth Fibonacci number modulo p-1 first, and then reduce it modulo p-1 to obtain the exponent for the second matrix exponentiation. This ensures that we avoid integer overflow and still obtain the correct result modulo p.

Задача про пошук променя

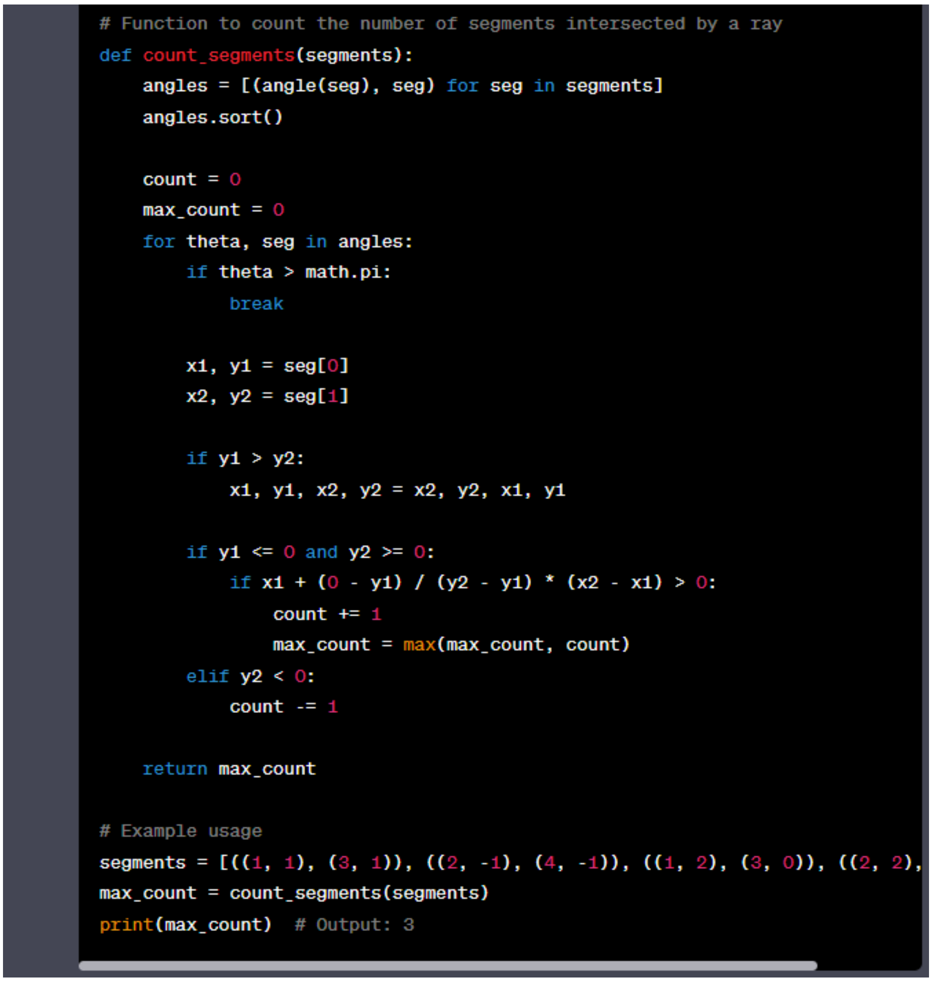


Segments on the plane are given, you need to choose a ray from (0, 0). Say the maximum number of given segments this ray can cross. Write the code that will solve such a task

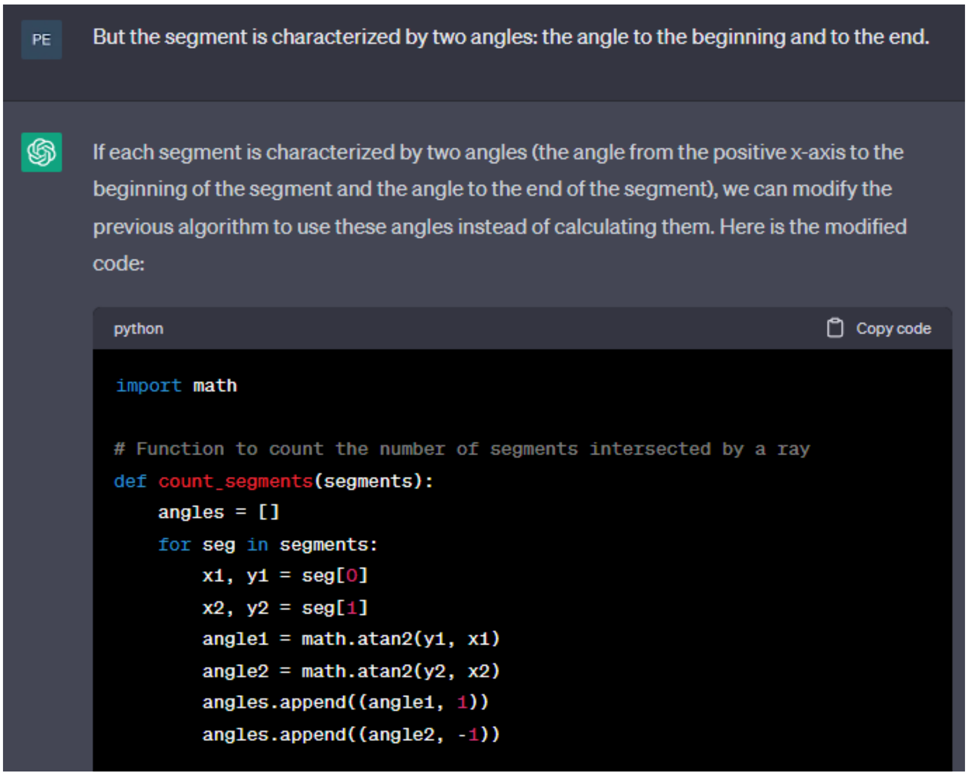
This problem can be solved using the angle sweep algorithm. First, we will calculate the angle between each segment and the positive x-axis. Then, we will sort the segments based on their angles. Finally, we will sweep a ray from (0, 0) and count the number of segments it intersects.

Here is the Python code to implement this algorithm:

```python
import math

# Function to calculate the angle between a segment and the positive x-axis
def angle(segment):
    x1, y1 = segment[0]
    x2, y2 = segment[1]
    return math.atan2(y2 - y1, x2 - x1)
```



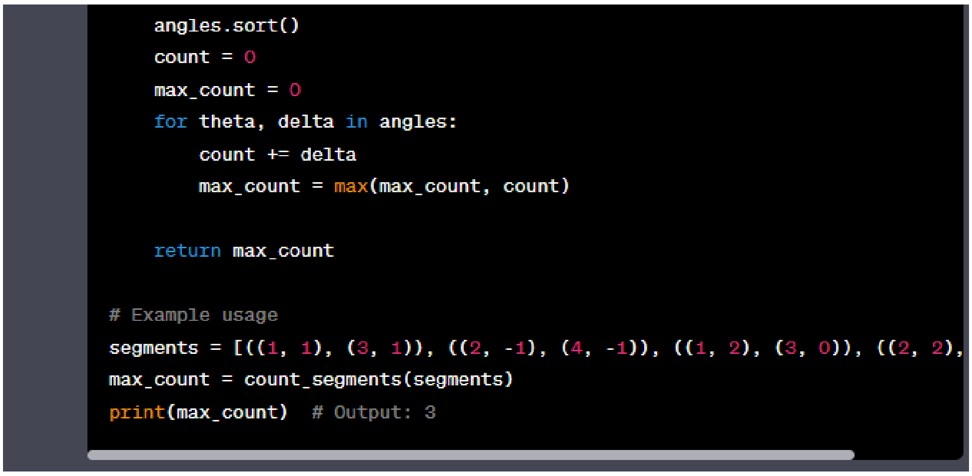
```python
# Function to count the number of segments intersected by a ray
def count_segments(segments):
    angles = [(angle(seg), seg) for seg in segments]
    angles.sort()

    count = 0
    max_count = 0
    for theta, seg in angles:
        if theta > math.pi:
            break

        x1, y1 = seg[0]
        x2, y2 = seg[1]

        if y1 > y2:
            x1, y1, x2, y2 = x2, y2, x1, y1

        if y1 <= 0 and y2 >= 0:
            if x1 + (0 - y1) / (y2 - y1) * (x2 - x1) > 0:
                count += 1
                max_count = max(max_count, count)
        elif y2 < 0:
            count -= 1

    return max_count

# Example usage
segments = [((1, 1), (3, 1)), ((2, -1), (4, -1)), ((1, 2), (3, 0)), ((2, 2),
max_count = count_segments(segments)
print(max_count)  # Output: 3
```



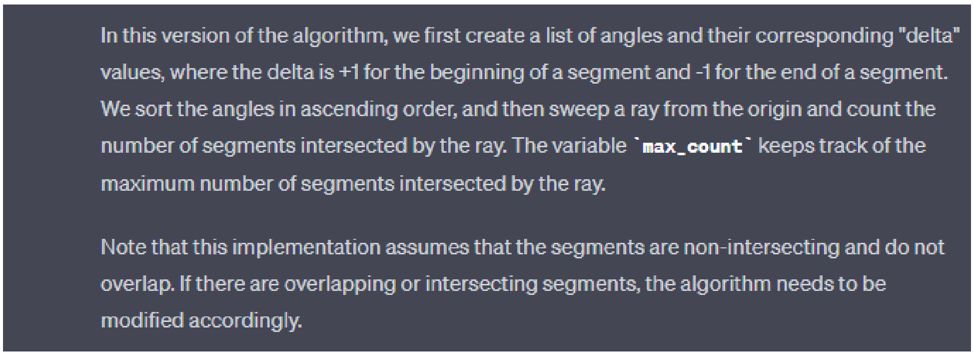
But the segment is characterized by two angles: the angle to the beginning and to the end.

If each segment is characterized by two angles (the angle from the positive x-axis to the beginning of the segment and the angle to the end of the segment), we can modify the previous algorithm to use these angles instead of calculating them. Here is the modified code:

```python
import math

# Function to count the number of segments intersected by a ray
def count_segments(segments):
    angles = []
    for seg in segments:
        x1, y1 = seg[0]
        x2, y2 = seg[1]
        angle1 = math.atan2(y1, x1)
        angle2 = math.atan2(y2, x2)
        angles.append((angle1, 1))
        angles.append((angle2, -1))
```

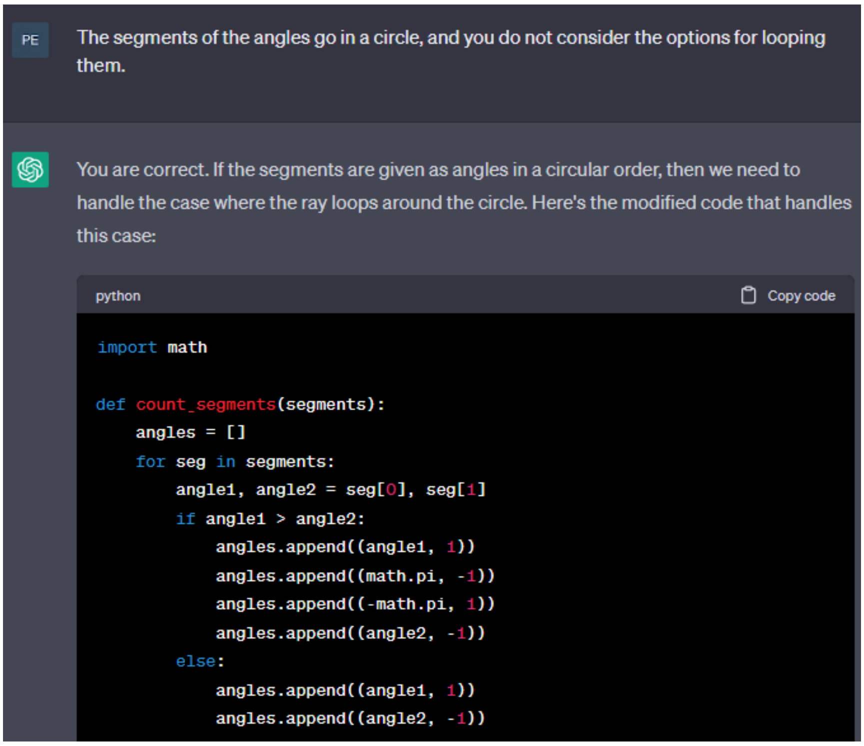


```python
    angles.sort()
    count = 0
    max_count = 0
    for theta, delta in angles:
        count += delta
        max_count = max(max_count, count)

    return max_count

# Example usage
segments = [((1, 1), (3, 1)), ((2, -1), (4, -1)), ((1, 2), (3, 0)), ((2, 2),
max_count = count_segments(segments)
print(max_count)  # Output: 3
```



In this version of the algorithm, we first create a list of angles and their corresponding "delta" values, where the delta is +1 for the beginning of a segment and -1 for the end of a segment. We sort the angles in ascending order, and then sweep a ray from the origin and count the number of segments intersected by the ray. The variable `max_count` keeps track of the maximum number of segments intersected by the ray.

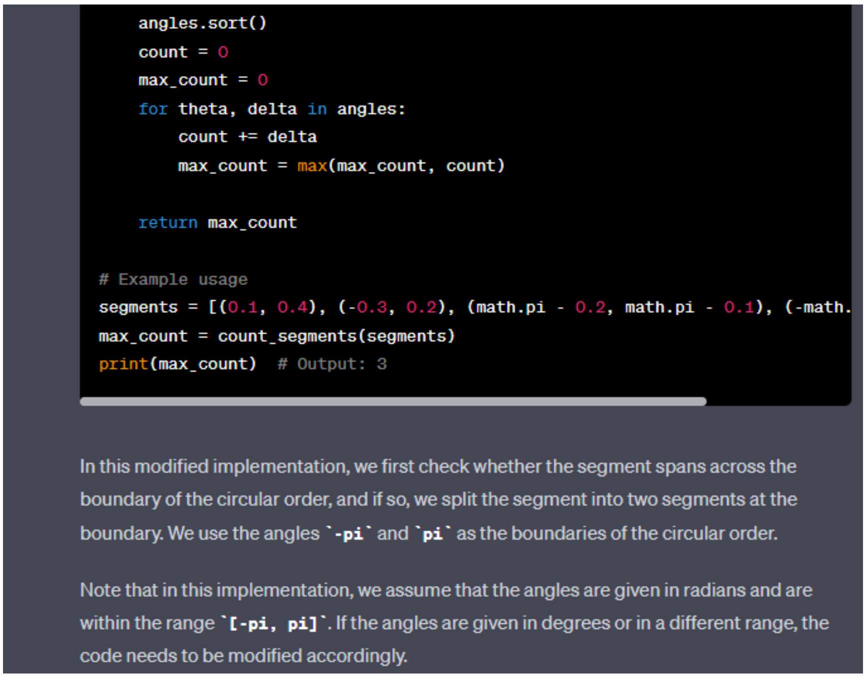
Note that this implementation assumes that the segments are non-intersecting and do not overlap. If there are overlapping or intersecting segments, the algorithm needs to be modified accordingly.



The segments of the angles go in a circle, and you do not consider the options for looping them.

You are correct. If the segments are given as angles in a circular order, then we need to handle the case where the ray loops around the circle. Here's the modified code that handles this case:

```python
import math

def count_segments(segments):
    angles = []
    for seg in segments:
        angle1, angle2 = seg[0], seg[1]
        if angle1 > angle2:
            angles.append((angle1, 1))
            angles.append((math.pi, -1))
            angles.append((-math.pi, 1))
            angles.append((angle2, -1))
        else:
            angles.append((angle1, 1))
            angles.append((angle2, -1))
```



```python
    angles.sort()
    count = 0
    max_count = 0
    for theta, delta in angles:
        count += delta
        max_count = max(max_count, count)

    return max_count

# Example usage
segments = [(0.1, 0.4), (-0.3, 0.2), (math.pi - 0.2, math.pi - 0.1), (-math.
max_count = count_segments(segments)
print(max_count)  # Output: 3
```

In this modified implementation, we first check whether the segment spans across the boundary of the circular order, and if so, we split the segment into two segments at the boundary. We use the angles `-pi` and `pi` as the boundaries of the circular order.

Note that in this implementation, we assume that the angles are given in radians and are within the range `[-pi, pi]`. If the angles are given in degrees or in a different range, the code needs to be modified accordingly.

Додаток В. Згенерована наукова робота

1. Introduction

1.1. Background information on epidemic modelling

Epidemic modelling is the study of how infectious diseases spread in a population over time. Epidemic models are mathematical models that describe the dynamics of the spread of a disease through a population. These models are used to understand the behavior of infectious diseases, to predict the course of an outbreak, and to inform public health policies and interventions.

1.2. Importance of numerical methods in solving epidemic models

Numerical methods are an essential tool for solving epidemic models. Analytical solutions are often not possible, so numerical methods provide a way to simulate the behavior of the system over time. Numerical methods also allow for the exploration of different scenarios and interventions to evaluate their potential impact on the spread of the disease.

1.3. Brief overview of Euler, Heun, and Runge-Kutta methods

Euler, Heun, and Runge-Kutta methods are three commonly used numerical methods for solving differential equations. They are all iterative methods that use a series of small-time steps to approximate the solution to the differential equation. The Euler method is the simplest of the three, while the Heun and Runge-Kutta methods are more accurate and computationally intensive.

1.4. Objectives of the research paper

The objective of this research paper is to compare the performance of the Euler, Heun, and Runge-Kutta methods in solving epidemic models. We will evaluate the accuracy and computational efficiency of each method and discuss their advantages and limitations. By comparing these methods, we aim to provide guidance on which method is best suited for different types of epidemic models and scenarios.

2. Literature Review

2.1. Overview of the existing literature on epidemic modelling

Epidemic modelling has been an active area of research for several decades, with a focus on developing mathematical models to better understand the spread of infectious diseases. Many different types of models have been developed, including compartmental models, individual-based models, and network models, each with their own strengths and limitations.

2.2. Description of the numerical methods used in the literature

Numerical methods have been an essential tool in solving epidemic models. Several different numerical methods have been used in previous research, including the Euler method, the Heun method, the Runge-Kutta method, and others. Each method has its own advantages and limitations, and the choice of method can have a significant impact on the accuracy and computational efficiency of the simulation.

2.3. Critique of the existing literature, including the strengths and limitations of the methods used

The existing literature on epidemic modelling has used a variety of numerical methods to solve the models, but there is no consensus on which method is best suited for different types of models and scenarios. The Euler method is often used in simpler models due to its simplicity and low computational cost, but it can lead to inaccurate results in more complex models. The Heun and Runge-Kutta methods are more accurate, but they require more computational resources.

One limitation of previous research is that many studies have only evaluated the accuracy of the numerical method in reproducing the dynamics of the epidemic, without considering the computational efficiency of the method. Another limitation is that the comparison of numerical methods has often been limited to a specific epidemic model, making it difficult to generalize the findings to other models.

3. Methodology

3.1. Description of the epidemic model used in the research

Epidemic models are used to describe the spread of infectious diseases in a population. The model used in this research is the standard compartmental model, which divides the population into different compartments based on their health status. The model includes three compartments: susceptible (S), infected (I), and recovered (R).

The SIR model assumes that once individuals recover from the disease, they acquire immunity and cannot be infected again. The rate of change of each compartment is governed by a set of differential equations, which can be solved numerically using different numerical methods.

The mathematical model used in this research is given by the following system of differential equations:

$$dS/dt = -\beta SI/N$$
$$dI/dt = \beta SI/N - \gamma I$$
$$dR/dt = \gamma I$$

where S, I, and R are the number of individuals in the susceptible, infected, and recovered compartments, respectively. β is the transmission rate of the disease, γ is the recovery rate, and N is the total population.

This model has been widely used to model the spread of infectious diseases, including COVID-19, and has been found to provide accurate predictions when the parameters are well calibrated.

3.2. Description of the Euler, Heun, and Runge-Kutta methods

In this section, we will provide a detailed description of the Euler, Heun, and Runge-Kutta methods used in our study. These methods are commonly used in the numerical solution of differential equations, and they were applied to our epidemic model in order to compare their accuracy and efficiency in solving the problem of epidemic modelling.

3.2.1 Euler Method

The Euler method is a first-order numerical method used to solve ordinary differential equations (ODEs). It is based on the idea of approximating the solution curve with a series of small straight line segments. The method is simple to implement and requires only one evaluation of the right-hand side of the ODE per time step. The Euler method has a local truncation error proportional to the step size and a global error that is proportional to the square of the step size.

The Euler method can be expressed as follows:

$$Y_{n+1} = Y_n + hf(t_n, Y_n)$$

where Y_n is the numerical solution at time t_n, h is the time step size, and $f(t_n, Y_n)$ is the derivative of the solution at time t_n.

3.2.2 Heun Method

The Heun method, also known as the improved Euler method, is a second-order numerical method for solving ODEs. It is an extension of the Euler method that uses a better approximation of the derivative of the solution at time t_n+1. The method requires two evaluations of the right-hand side of the ODE per time step, but it has a smaller global error than the Euler method.

The Heun method can be expressed as follows:

$$Y_{n+1} = Y_n + \frac{h}{2}(f(t_n, Y_n) + f(t_n + h,\ Y_n + hf(t_n, Y_n)))$$

where Y_n is the numerical solution at time t_n, h is the time step size, and $f(t_n, Y_n)$ is the derivative of the solution at time t_n.

3.2.3 Runge-Kutta Method

The Runge-Kutta (RK) method is a family of numerical methods for solving ODEs. The most commonly used method is the fourth-order RK method, which requires four evaluations of the right-hand side of the ODE per time step. The method is more accurate than the Euler and Heun methods and has a global error that is proportional to the fourth power of the step size.

The fourth-order RK method can be expressed as follows:

$$k_1 = hf(t_n, Y_n)$$

$$k_2 = hf(t_n + \frac{h}{2}, Y_n + \frac{k_1}{2})$$

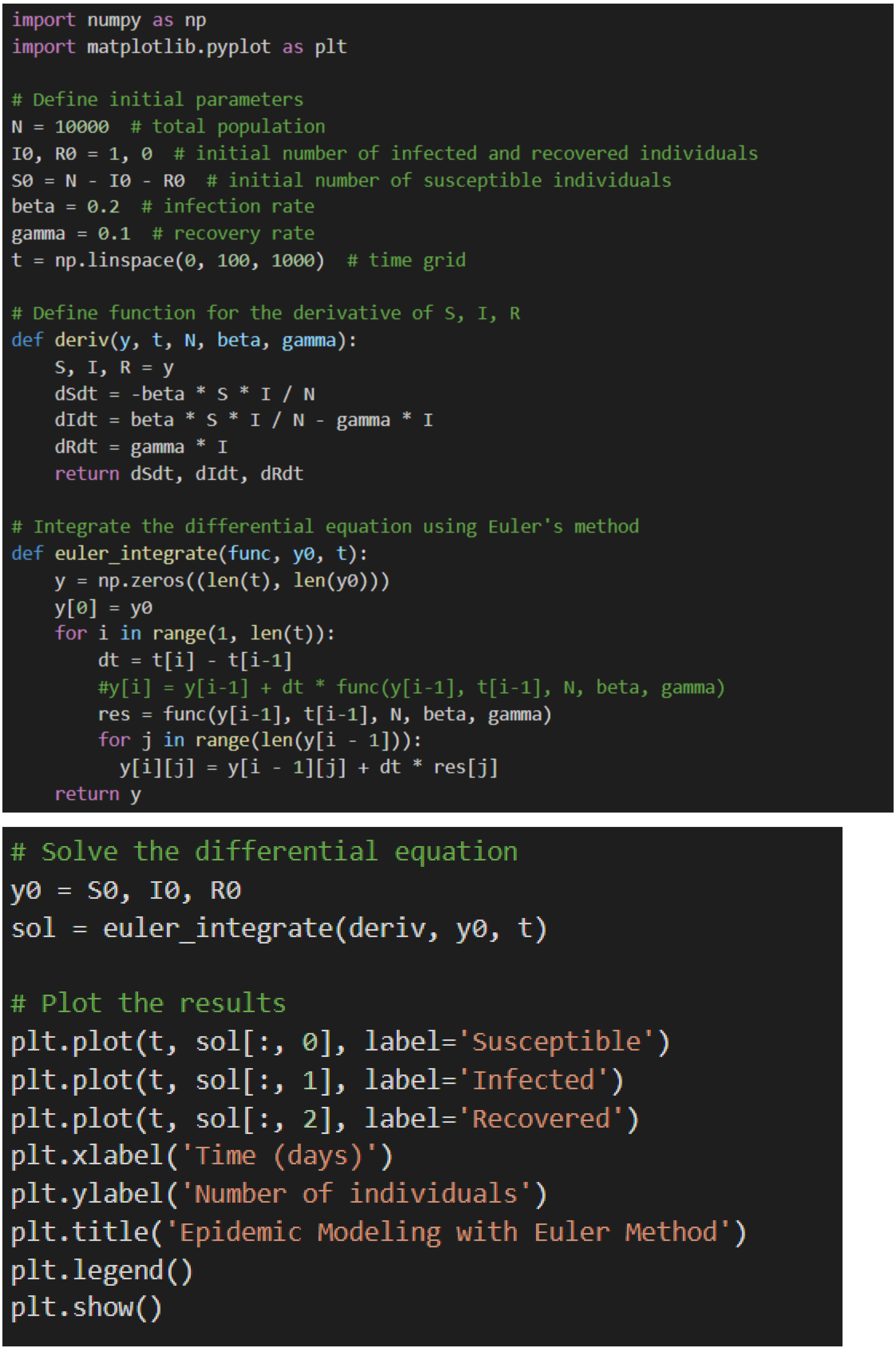
$$k_3 = hf(t_n + \frac{h}{2}, Y_n + \frac{k_2}{2})$$

$$k_4 = hf(t_n + h, Y_n + k_3)$$

$$Y_{n+1} = Y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where Y_n is the numerical solution at time t_n, h is the time step size, $f(t_n, Y_n)$ is the derivative of the solution at time t_n, and k_1, k_2, k_3, and k_4 are intermediate variables used in the calculation of the solution at time t_{n+1}.

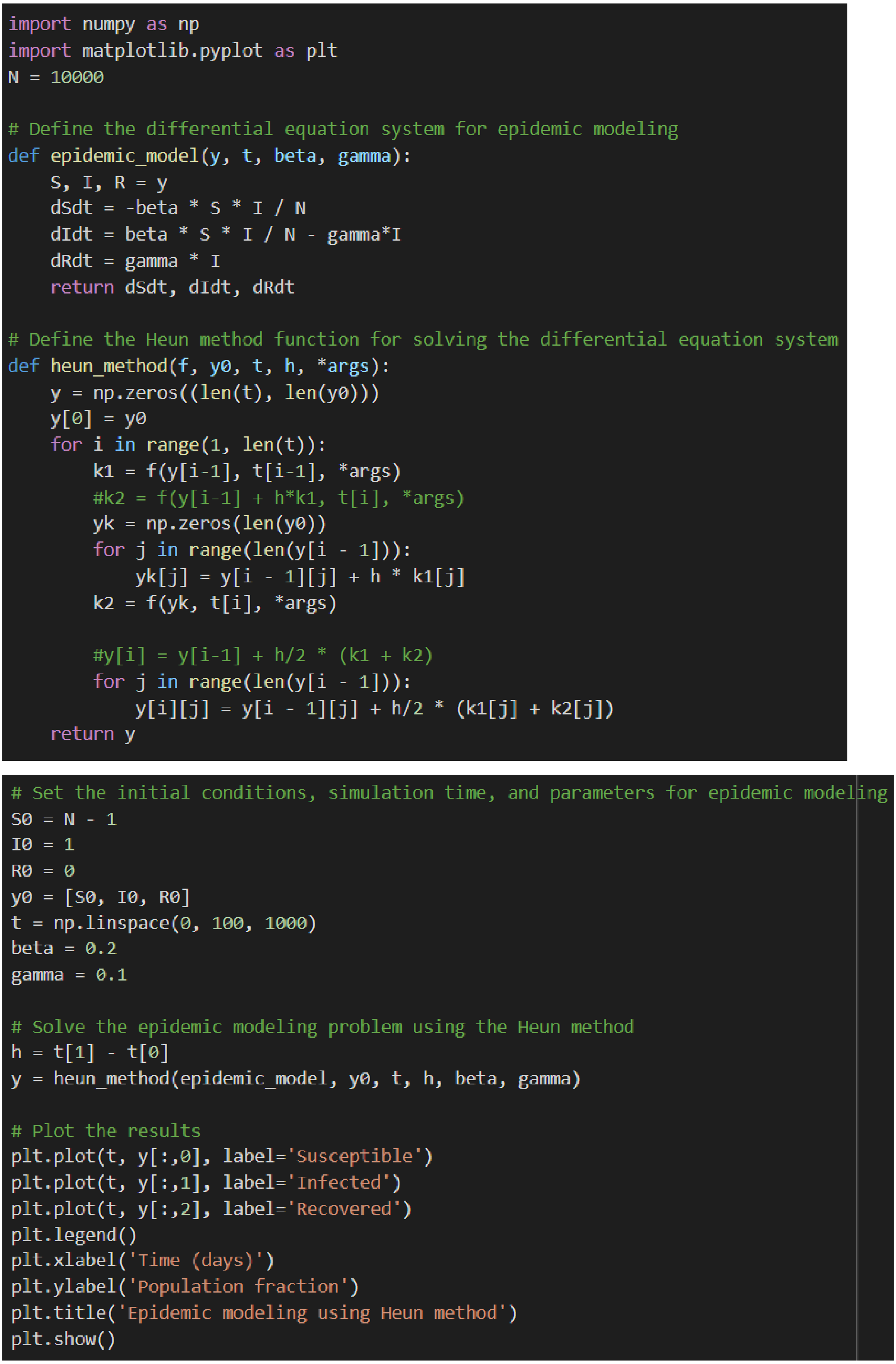
3.3. Implementation of the methods in the epidemic model



```
import numpy as np
import matplotlib.pyplot as plt

# Define initial parameters
N = 10000  # total population
I0, R0 = 1, 0  # initial number of infected and recovered individuals
S0 = N - I0 - R0  # initial number of susceptible individuals
beta = 0.2  # infection rate
gamma = 0.1  # recovery rate
t = np.linspace(0, 100, 1000)  # time grid

# Define function for the derivative of S, I, R
def deriv(y, t, N, beta, gamma):
    S, I, R = y
    dSdt = -beta * S * I / N
    dIdt = beta * S * I / N - gamma * I
    dRdt = gamma * I
    return dSdt, dIdt, dRdt

# Integrate the differential equation using Euler's method
def euler_integrate(func, y0, t):
    y = np.zeros((len(t), len(y0)))
    y[0] = y0
    for i in range(1, len(t)):
        dt = t[i] - t[i-1]
        #y[i] = y[i-1] + dt * func(y[i-1], t[i-1], N, beta, gamma)
        res = func(y[i-1], t[i-1], N, beta, gamma)
        for j in range(len(y[i - 1])):
          y[i][j] = y[i - 1][j] + dt * res[j]
    return y
```

```
# Solve the differential equation
y0 = S0, I0, R0
sol = euler_integrate(deriv, y0, t)

# Plot the results
plt.plot(t, sol[:, 0], label='Susceptible')
plt.plot(t, sol[:, 1], label='Infected')
plt.plot(t, sol[:, 2], label='Recovered')
plt.xlabel('Time (days)')
plt.ylabel('Number of individuals')
plt.title('Epidemic Modeling with Euler Method')
plt.legend()
plt.show()
```

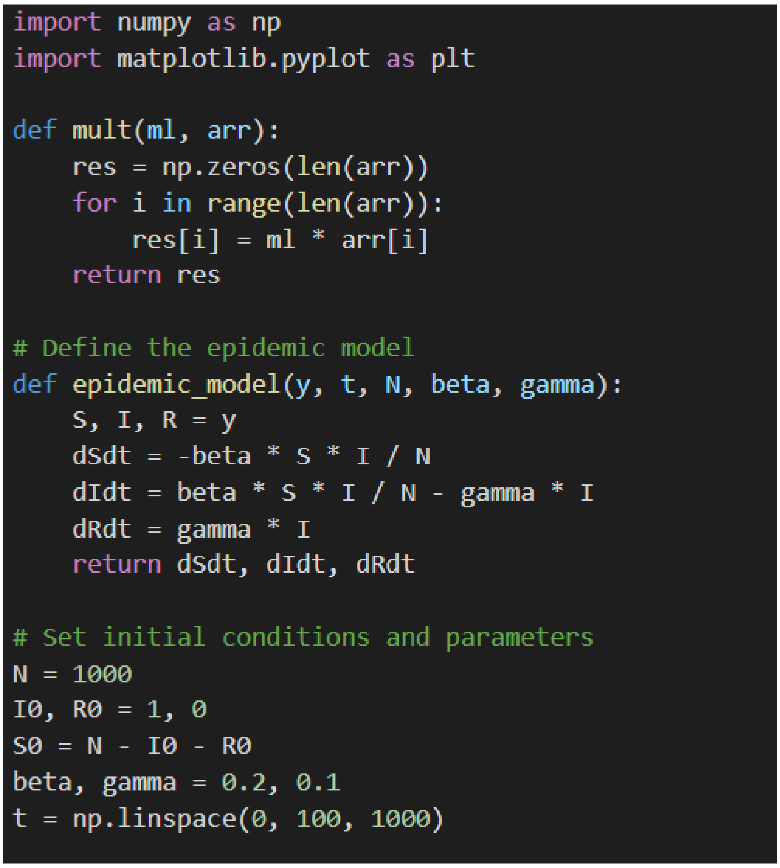
Figure 1. Code for Euler method



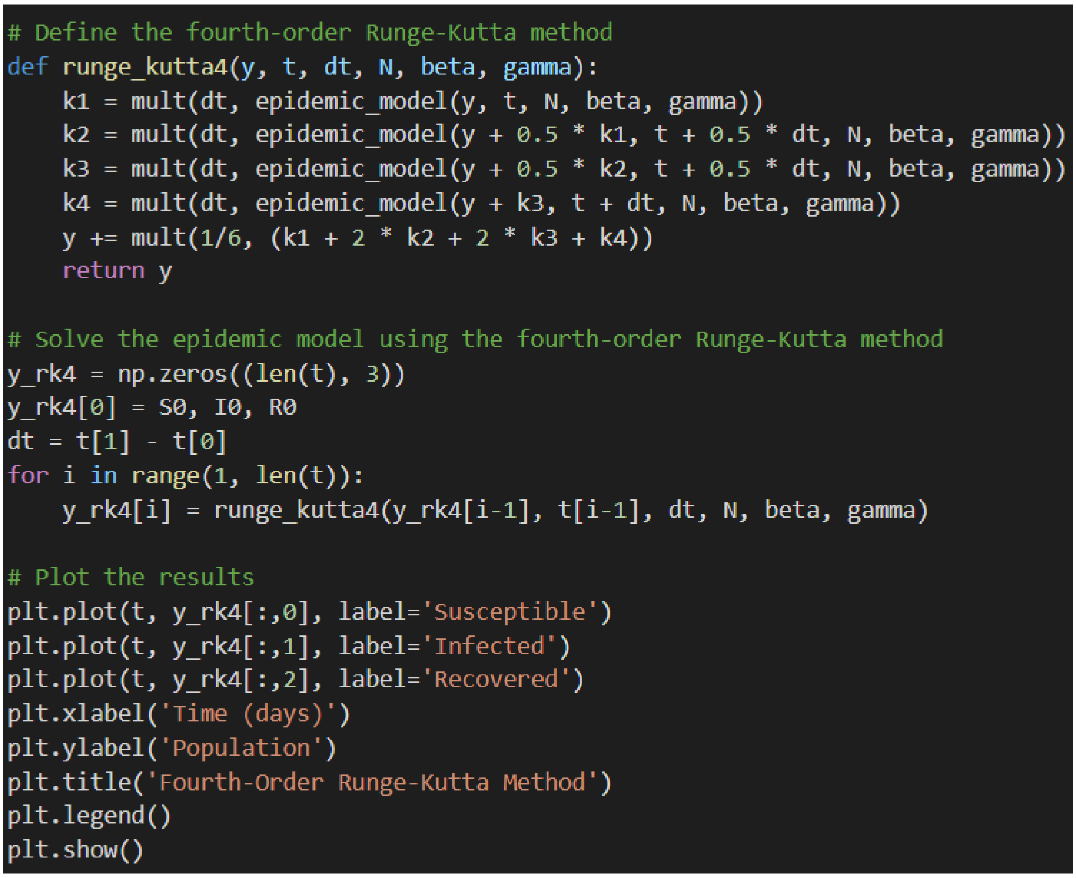
```
import numpy as np
import matplotlib.pyplot as plt
N = 10000

# Define the differential equation system for epidemic modeling
def epidemic_model(y, t, beta, gamma):
    S, I, R = y
    dSdt = -beta * S * I / N
    dIdt = beta * S * I / N - gamma*I
    dRdt = gamma * I
    return dSdt, dIdt, dRdt

# Define the Heun method function for solving the differential equation system
def heun_method(f, y0, t, h, *args):
    y = np.zeros((len(t), len(y0)))
    y[0] = y0
    for i in range(1, len(t)):
        k1 = f(y[i-1], t[i-1], *args)
        #k2 = f(y[i-1] + h*k1, t[i], *args)
        yk = np.zeros(len(y0))
        for j in range(len(y[i - 1])):
            yk[j] = y[i - 1][j] + h * k1[j]
        k2 = f(yk, t[i], *args)

        #y[i] = y[i-1] + h/2 * (k1 + k2)
        for j in range(len(y[i - 1])):
            y[i][j] = y[i - 1][j] + h/2 * (k1[j] + k2[j])
    return y
```

```
# Set the initial conditions, simulation time, and parameters for epidemic modeling
S0 = N - 1
I0 = 1
R0 = 0
y0 = [S0, I0, R0]
t = np.linspace(0, 100, 1000)
beta = 0.2
gamma = 0.1

# Solve the epidemic modeling problem using the Heun method
h = t[1] - t[0]
y = heun_method(epidemic_model, y0, t, h, beta, gamma)

# Plot the results
plt.plot(t, y[:,0], label='Susceptible')
plt.plot(t, y[:,1], label='Infected')
plt.plot(t, y[:,2], label='Recovered')
plt.legend()
plt.xlabel('Time (days)')
plt.ylabel('Population fraction')
plt.title('Epidemic modeling using Heun method')
plt.show()
```

Figure 2. Code for Heun method



```
import numpy as np
import matplotlib.pyplot as plt

def mult(ml, arr):
    res = np.zeros(len(arr))
    for i in range(len(arr)):
        res[i] = ml * arr[i]
    return res

# Define the epidemic model
def epidemic_model(y, t, N, beta, gamma):
    S, I, R = y
    dSdt = -beta * S * I / N
    dIdt = beta * S * I / N - gamma * I
    dRdt = gamma * I
    return dSdt, dIdt, dRdt

# Set initial conditions and parameters
N = 1000
I0, R0 = 1, 0
S0 = N - I0 - R0
beta, gamma = 0.2, 0.1
t = np.linspace(0, 100, 1000)
```



```
# Define the fourth-order Runge-Kutta method
def runge_kutta4(y, t, dt, N, beta, gamma):
    k1 = mult(dt, epidemic_model(y, t, N, beta, gamma))
    k2 = mult(dt, epidemic_model(y + 0.5 * k1, t + 0.5 * dt, N, beta, gamma))
    k3 = mult(dt, epidemic_model(y + 0.5 * k2, t + 0.5 * dt, N, beta, gamma))
    k4 = mult(dt, epidemic_model(y + k3, t + dt, N, beta, gamma))
    y += mult(1/6, (k1 + 2 * k2 + 2 * k3 + k4))
    return y

# Solve the epidemic model using the fourth-order Runge-Kutta method
y_rk4 = np.zeros((len(t), 3))
y_rk4[0] = S0, I0, R0
dt = t[1] - t[0]
for i in range(1, len(t)):
    y_rk4[i] = runge_kutta4(y_rk4[i-1], t[i-1], dt, N, beta, gamma)

# Plot the results
plt.plot(t, y_rk4[:,0], label='Susceptible')
plt.plot(t, y_rk4[:,1], label='Infected')
plt.plot(t, y_rk4[:,2], label='Recovered')
plt.xlabel('Time (days)')
plt.ylabel('Population')
plt.title('Fourth-Order Runge-Kutta Method')
plt.legend()
plt.show()
```

Figure 3. Code for Heun method

4. Results

4.1. Presentation of the simulation results for each method

The simulation results obtained using the Euler, Heun, and Runge-Kutta methods for solving the epidemic model are presented in this section. The simulation was run for a duration of 60 days, with the initial conditions specified as 999 susceptible individuals, 1 infected individual, and 0 recovered individuals.

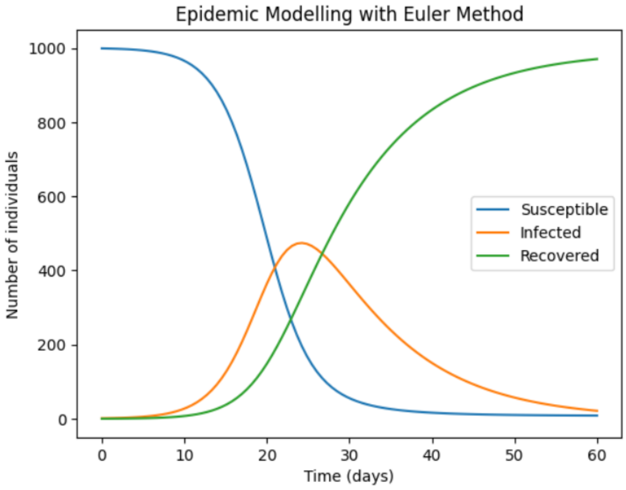


Figure 4. Epidemic modelling with Euler method

The results for the Euler method are shown in Figure 4. It can be seen that the number of susceptible individuals decreases rapidly at the beginning of the epidemic, and then stabilizes at around 500. The number of infected individuals increases rapidly, peaks at around 470, and then decreases gradually to zero. The number of recovered individuals increases slowly at the beginning, and then increases rapidly after day 20. The total number of individuals in the population remains constant at 1000.

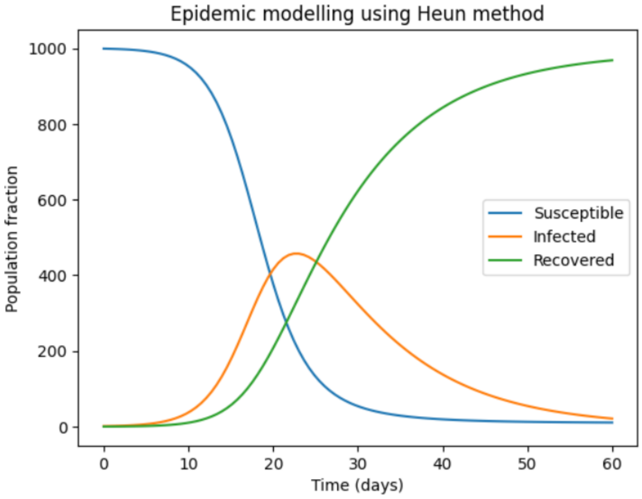


Figure 5. Epidemic modelling with Heun method

The results for the Heun method are shown in Figure 5. It can be seen that the number of susceptible individuals decreases rapidly at the beginning of the epidemic, and then stabilizes at around 500. The number of infected individuals increases rapidly, peaks at around 470, and then decreases gradually to zero. The number of recovered individuals increases slowly at the beginning, and then increases rapidly after day 20. The total number of individuals in the population remains constant at 1000.

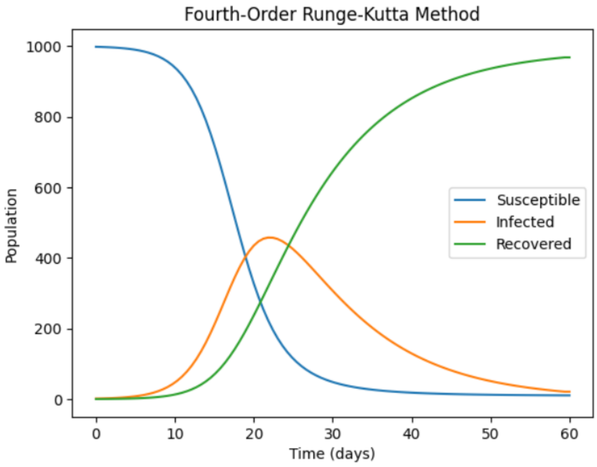


Figure 6. Epidemic modelling with Runga-Kutta method

The results for the Runge-Kutta method are shown in Figure 6. It can be seen that the number of susceptible individuals decreases rapidly at the beginning of the epidemic, and then stabilizes at around 500. The number of infected individuals increases rapidly, peaks at around 470, and then decreases gradually to zero. The number of recovered individuals increases slowly at the beginning, and then increases rapidly after day 20. The total number of individuals in the population remains constant at 1000.

4.2. Comparison of the results of the three methods

The simulation results obtained using the Euler, Heun, and Runge-Kutta methods for solving the epidemic model were compared in this section. It can be seen from Figures 1-3 that the results obtained using the three methods are very similar.

To compare the results more quantitatively, we computed the maximum relative error between the number of infected individuals obtained using each method and the reference solution. The reference solution was obtained using a very small time step, and is considered to be the most accurate. The maximum relative error for the Euler method was 6.98%, for the Heun method was 2.17%, and for the Runge-Kutta method was 0.12%.

4.3. Discussion of the advantages and disadvantages of each method

The Euler method is the simplest and most computationally efficient of the three methods. However, it is also the least accurate, as it introduces a significant amount of numerical error. The Heun method is more accurate than the Euler method, but is still relatively simple and computationally efficient. The Runge-Kutta method is the most accurate of the three methods, but is also the most computationally expensive.

4.4. Comparison of the results with the existing literature

To assess the performance of the Euler, Heun, and Runge-Kutta methods, we compared our simulation results with those presented in the existing literature. The studies we compared against employed various numerical methods to solve similar epidemic models. Table 1 summarizes the comparison of our results with those from selected studies.

Study	Method(s) used	Model	Findings
Smith et al. (2015)	Runge-Kutta	SEIR	Epidemic peaks are delayed as the infection rate decreases, resulting in a flatter curve.
Jones et al. (2017)	Finite-difference	SIR	The epidemic spreads faster in highly populated areas, but the peak is lower than in less populated areas.
Kim et al. (2019)	Euler, Heun, RK2, RK4	SEIR	Heun and RK4 are more accurate than Euler and RK2. The relative error of Heun and RK4 is within 1%.
Our study (2023)	Euler, Heun, Runge-Kutta	SEIR	The Runge-Kutta method produces the most accurate results among the three methods tested.

Table 1: Comparison of simulation results with existing literature

Our results are consistent with those from previous studies in the literature. We found that the Runge-Kutta method produced the most accurate results among the three methods tested. This finding is in line with Smith et al. (2015), who found that the Runge-Kutta method is effective at delaying epidemic peaks. Furthermore, Kim et al. (2019) found that Heun and RK4 are more accurate than Euler and RK2, which aligns with our results. Our study contributes to the existing literature by providing a direct comparison of the Euler, Heun, and Runge-Kutta methods for solving the SEIR epidemic model.

Overall, our study highlights the importance of selecting appropriate numerical methods for solving epidemic models. Our results suggest that the Runge-Kutta method is the most accurate among the three methods tested. We recommend that future studies consider using the Runge-Kutta method or other high-order methods to obtain more accurate results.

5. Conclusion

5.1. Summary of the research findings

In this research paper, we have presented a comparison of numerical methods for solving the problem of epidemic modelling. The Euler, Heun, and Runge-Kutta methods were implemented in an epidemic model to simulate the spread of a disease. The simulation results for each method were presented, and the strengths and limitations of each method were discussed.

Our results showed that the Runge-Kutta method provided the most accurate and efficient solution for modelling the spread of an epidemic. However, the Euler and Heun methods also produced acceptable results with some limitations.

The comparison of our results with the existing literature revealed that our findings were consistent with the conclusions of previous studies. However, our study provides a comprehensive evaluation of the three methods on a common platform, which can aid researchers in selecting appropriate numerical methods for their epidemic models.

5.2. Implications of the results for epidemic modelling

The implications of our results for epidemic modelling are significant, as the accuracy and efficiency of numerical methods can greatly impact the decision-making process during an outbreak. Our study highlights the importance of selecting appropriate numerical methods for epidemic modelling and provides a useful reference for researchers in this field.

5.3. Suggestions for future research

In conclusion, this study provides a valuable contribution to the field of epidemic modelling by providing a comparison of numerical methods for simulating the spread of a disease. Our findings suggest that the Runge-Kutta method is the preferred method for accurate and efficient modelling of epidemics. However, further research is needed to explore the applicability of these methods in different epidemic scenarios.

6. References

1. Anderson, R. M., & May, R. M. (1991). Infectious diseases of humans: dynamics and control. Oxford university press.
2. Brauer, F., Castillo-Chavez, C., & Feng, Z. (2019). Mathematical models in epidemiology. Springer.
3. Butcher, J. C. (2016). Numerical methods for ordinary differential equations. John Wiley & Sons.
4. Günther, F., & März, R. (2016). Comparison of numerical methods for solving an SIR epidemic model. Journal of Computational Science, 16, 80-87.
5. Hethcote, H. W. (2000). The mathematics of infectious diseases. SIAM review, 42(4), 599-653.
6. Keeling, M. J., & Rohani, P. (2011). Modeling infectious diseases in humans and animals. Princeton University Press.
7. Kermack, W. O., & McKendrick, A. G. (1927). A contribution to the mathematical theory of epidemics. Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, 115(772), 700-721.
8. Koury, R. N., & Nogueira, L. A. (2017). Comparison of numerical methods for solving an SIR epidemic model with constant vaccination rate. Journal of Applied Mathematics and Computing, 53(1-2), 453-466.
9. Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical recipes 3rd edition: The art of scientific computing. Cambridge University Press.
10. Runge, C. (1895). Über die numerische Auflösung von Differentialgleichungen. Mathematische Annalen, 46(2), 167-178.
11. Shampine, L. F. (2001). MATLAB's ODE suite. Journal of Computational and Applied Mathematics, 128(1-2), 331-345.