VIETNAM GENERAL CONFEDERATION OF LABOR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**NGUYỄN TRẦN HOÀNG NHÂN – 523H0164**

**BÀI TẬP LỚN HỆ CƠ SỞ DỮ LIỆU  
2024 - 2025**

**HỆ CƠ SỞ DỮ LIỆU**

**HO CHI MINH CITY, 2024**

VIETNAM GENERAL CONFEDERATION OF LABOR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**NGUYỄN TRẦN HOÀNG NHÂN – 523H0164**

**BÀI TẬP LỚN CƠ SỞ DỮ LIỆU**

**HỆ CƠ SỞ DỮ LIỆU**

Advised by

**Dr. HUỲNH THỊ THU THỦY**

**HO CHI MINH CITY, 2024**

**TABLE OF CONTENT**

[LIST OF FIGURES ii](#_Toc163396281)

[CHAPTER 1. METHODOLOGY OF SOLVING TASKS 1](#_Toc163396282)

[1.1 Declaration: 1](#_Toc163396283)

[1.2 Task 1a: 1](#_Toc163396284)

[1.3 Task 1b: 2](#_Toc163396285)

[1.4 Task 1c: 2](#_Toc163396286)

[1.5 Task 1d: 2](#_Toc163396287)

[1.6 Task 1e: 3](#_Toc163396288)

[1.7 Task 1f: 3](#_Toc163396289)

[1.8 Task 1g: 4](#_Toc163396290)

[1.9 Task 1h: 4](#_Toc163396291)

[CHAPTER 2. SOURCE CODES AND OUTPUTS 6](#_Toc163396292)

[2.1 Declaration: 6](#_Toc163396293)

[2.2 Task 1a: 7](#_Toc163396294)

[2.3 Task 1b: 8](#_Toc163396295)

[2.4 Task 1c: 9](#_Toc163396296)

[2.5 Task 1d: 10](#_Toc163396297)

[2.6 Task 1e: 10](#_Toc163396298)

[2.7 Task 1f: 10](#_Toc163396299)

[2.8 Task 1g: 11](#_Toc163396300)

[2.9 Task 1h: 11](#_Toc163396301)

# LIST OF FIGURES

[Figure 1: Import numpy library, declare matrix A, matrix B and matrix C. 6](#_Toc163396302)

[Figure 2: Print matrix A, matrix B and matrix C 6](#_Toc163396303)

[Figure 3: Matrix A 7](#_Toc163396304)

[Figure 4: Matrix B 7](#_Toc163396305)

[Figure 5: Matrix C 7](#_Toc163396306)

[Figure 6: Implement task 1a 8](#_Toc163396307)

[Figure 7: Answer of task 1a 8](#_Toc163396308)

[Figure 8: Implement task 1b 8](#_Toc163396309)

[Figure 9: Answer of task 1b 9](#_Toc163396310)

[Figure 10: Implement task 1c 9](#_Toc163396311)

[Figure 11: Answer of task 1c 10](#_Toc163396312)

[Figure 12: Implement task 1d 10](#_Toc163396313)

[Figure 13: Answer of task 1d 10](#_Toc163396314)

[Figure 14: Implement task 1e 10](#_Toc163396315)

[Figure 15: Answer of task 1e 10](#_Toc163396316)

[Figure 16: Implement task 1f 11](#_Toc163396317)

[Figure 17: Answer of task 1f 11](#_Toc163396318)

[Figure 18: Implement task 1g 11](#_Toc163396319)

[Figure 19: Answer of task 1g 11](#_Toc163396320)

[Figure 20: Implement task 1h 12](#_Toc163396321)

[Figure 21: Answer of task 1h 12](#_Toc163396322)

# METHODOLOGY OF SOLVING TASKS

## Declaration:

* The numpy library is imported and aliased as np.
* The sympy library is imported and aliased as sp.
* Three matrices (A, B, and C) are defined using the np.random.randint function from the numpy library.
* The rows and cols variables define the number of rows and columns for each matrix, respectively.
* The np.random.randint function generates a random integer matrix with values between low (inclusive) and high (exclusive). In this case, low is set to 1 and high is set to rows \* cols + 1 to ensure values within a reasonable range for the matrix size. Size is a tuple specifying the shape of the matrix (number of rows, number of columns).

## Task 1a:

Calculate: .

* Performs the core calculations and stores the result in a variable named result. It combines four elements:
* a\_matrix + a\_matrix.T: This is the addition operation between the matrix a\_matrix and its transpose a\_matrix.T. The transpose of a matrix is computed using the .T method in NumPy.
* np.matmul(c\_matrix, b\_matrix): This is the matrix multiplication operation between c\_matrix and b\_matrix using the np.matmul() function in NumPy.
* np.matmul(b\_matrix.T, c\_matrix.T): Similarly, this is the matrix multiplication operation between the transpose of b\_matrix and the transpose of c\_matrix.
* print(result): After the computations, the result is stored in the variable result, and then printed to the screen using the print statement.

## Task 1b:

Calculate

* result = 0: Initializes the variable result to 0.
* for i in range(10): This is a loop that iterates i from 0 to 9 (inclusive). It will repeat the subsequent code block 10 times.
* result += np.linalg.matrix\_power(a\_matrix / (10 + i), i + 1):
* np.linalg.matrix\_power(a\_matrix / (10 + i), i + 1): This calculates the matrix power of the expression (a\_matrix / (10 + i)) raised to the power of (i + 1). np.linalg.matrix\_power() function computes the matrix power of a given matrix.
* result += ...: This adds the result of the matrix power calculation to the result variable. It's equivalent to result = result + ....
* print(result): Prints out the final result after the loop completes.

## Task 1c:

Save odd rows of the matrix into a new matrix, and print the resultant matrix to the screen.

* odd\_row = [value for index, value in enumerate(a\_matrix) if index % 2 == 0]: This line creates a list comprehension that iterates over the rows of the matrix a\_matrix and selects only the rows whose index is odd (i.e., index % 2 == 0). It extracts the values of these rows and stores them in the list odd\_row.
* result\_matrix = np.array(odd\_row): Converts the list odd\_row into a NumPy array and assigns it to the variable result\_matrix. This creates a new matrix containing only the odd-indexed rows from the original matrix a\_matrix.
* print(result\_matrix): Prints out the resulting matrix result\_matrix.

## Task 1d:

Save odd integer numbers in the matrix into a new vector, and print the resultant vector to the screen.

* mask = a\_matrix % 2 != 0: This line creates a boolean mask by applying the condition a\_matrix % 2 != 0. This condition checks each element of the matrix a\_matrix to see if it's odd (i.e., not divisible by 2). If an element satisfies this condition, the corresponding entry in the mask will be True; otherwise, it will be False.
* result\_matrix = a\_matrix[mask]: This line uses the boolean mask to select the elements from the matrix a\_matrix where the mask is True. In other words, it filters out the elements of a\_matrix that are odd according to the condition specified by the mask.
* print(result\_matrix): Prints out the resulting matrix result\_matrix.

## Task 1e:

Save prime numbers in the matrix into a new vector, and print the resultant vector to the screen.

* prime\_numbers = np.array([num for row in a\_matrix for num in row if sp.isprime(num)]): This line creates a NumPy array prime\_numbers containing prime numbers extracted from the matrix a\_matrix. It uses a list comprehension to iterate over each element (num) in each row (row) of a\_matrix. The condition if sp.isprime(num) checks if num is a prime number using the sp.isprime() function from the SymPy library.
* print(prime\_numbers): Prints out the resulting array prime\_numbers containing prime numbers extracted from a\_matrix.

## Task 1f:

Given a matrix , reverse elements in the odd rows of the matrix , and print the resultant matrix to the screen.

* d\_matrix = np.matmul(c\_matrix, b\_matrix): This line computes the matrix product of c\_matrix and b\_matrix using the np.matmul() function in NumPy and assigns the result to the variable d\_matrix.
* for i in range(len(d\_matrix)): This line initiates a loop that iterates over the indices of the rows in d\_matrix.
* if i % 2 == 0: Inside the loop, this line checks if the index i is even.
* d\_matrix[i] = np.flip(d\_matrix[i]): If the index i is even, this line flips the elements of the row i in the matrix d\_matrix using the np.flip() function. Flipping the row means reversing the order of its elements.
* print(d\_matrix): Finally, the resulting matrix d\_matrix, after every other row has been flipped, is printed.

## Task 1g:

Regarding the matrix , find the rows which have maximum count of prime numbers, and print the rows to the screen.

* max\_prime\_count = max(sum(sp.isprime(num) for num in row) for row in a\_matrix): This line calculates the maximum count of prime numbers in any row of the matrix a\_matrix. It uses a generator expression within the max() function to iterate over each row of a\_matrix, calculate the sum of prime numbers in that row, and then find the maximum of these sums.
* rows\_with\_max\_primes = [row for row in a\_matrix if sum(sp.isprime(num) for num in row) == max\_prime\_count]: This line creates a list rows\_with\_max\_primes containing the rows of a\_matrix that have the maximum count of prime numbers. It uses a list comprehension to iterate over each row of a\_matrix, and selects only those rows whose sum of prime numbers equals max\_prime\_count.
* for row in rows\_with\_max\_primes: This initiates a loop that iterates over each row in rows\_with\_max\_primes.
* print(row): Inside the loop, this line prints out each row that has the maximum count of prime numbers.

## Task 1h:

Regarding the matrix , find the rows which have the longest contiguous odd numbers sequence, and print the rows to the screen.

* def longest\_odd\_sequence(lst): This line defines a nested function named longest\_odd\_sequence(lst). This function calculates the length of the longest sequence of consecutive odd numbers in a given list lst. It iterates through the elements of the list, updating current\_length if the element is odd and resetting it to 0 if the element is even. It returns the max\_length found.
* max\_odd\_sequence\_lengths = [longest\_odd\_sequence(row) for row in a\_matrix]: This line creates a list max\_odd\_sequence\_lengths containing the lengths of the longest sequences of consecutive odd numbers in each row of the matrix a\_matrix. It uses a list comprehension to iterate over each row of a\_matrix and apply the longest\_odd\_sequence() function to it.
* max\_length = max(max\_odd\_sequence\_lengths): This line calculates the maximum length of consecutive odd numbers among all rows of the matrix a\_matrix.
* rows\_with\_max\_length = [row for row, length in zip(a\_matrix, max\_odd\_sequence\_lengths) if length == max\_length]: This line creates a list rows\_with\_max\_length containing the row(s) of a\_matrix that have the maximum length of consecutive odd numbers. It uses a list comprehension and the zip() function to iterate over pairs of rows and their corresponding lengths, and selects only those rows whose length matches max\_length.
* for row in rows\_with\_max\_length: This initiates a loop that iterates over each row in rows\_with\_max\_length.
* print(row): Inside the loop, this line prints out each row that has the maximum length of consecutive odd numbers.

# SOURCE CODES AND OUTPUTS

## Declaration:

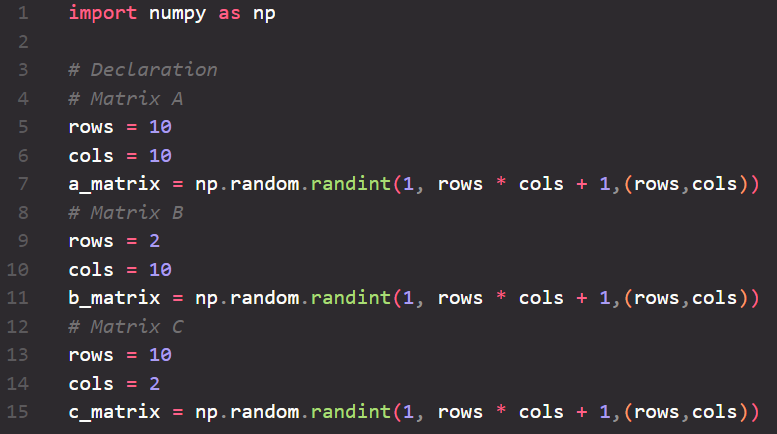


Figure 1: Import numpy library, declare matrix A, matrix B and matrix C.

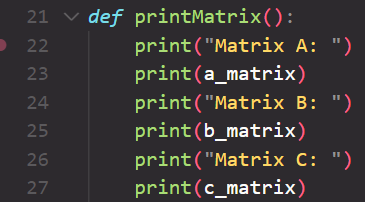


Figure 2: Print matrix A, matrix B and matrix C

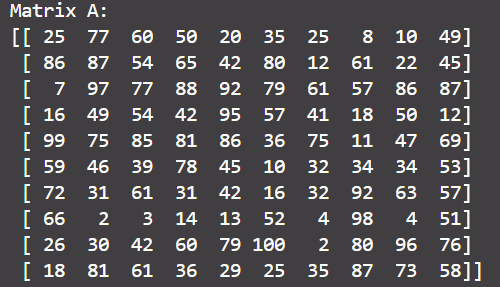


Figure 3: Matrix A

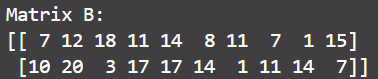


Figure 4: Matrix B

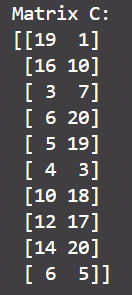


Figure 5: Matrix C

## Task 1a:

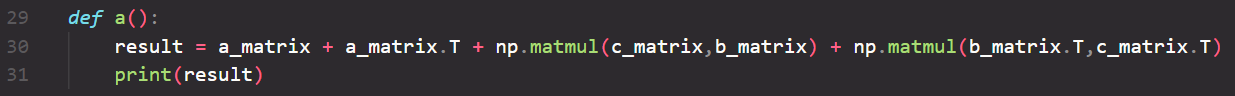


Figure 6: Implement task 1a

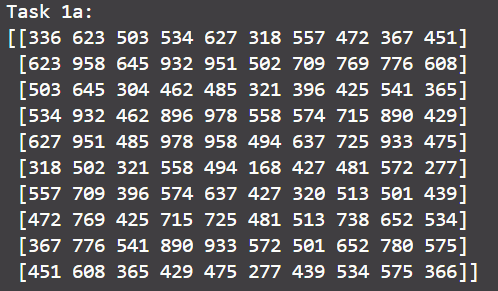


Figure 7: Answer of task 1a

## Task 1b:

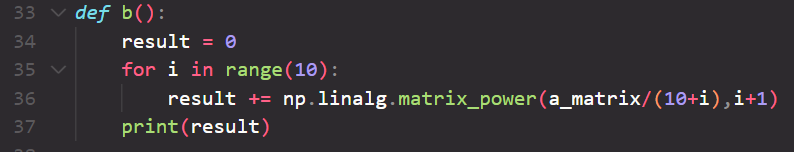


Figure 8: Implement task 1b

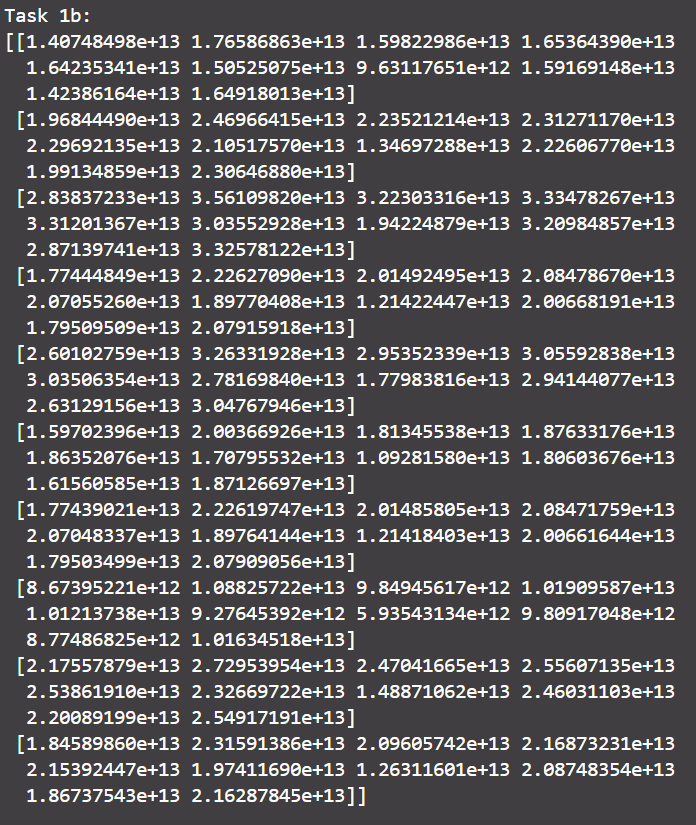


Figure 9: Answer of task 1b

## Task 1c:

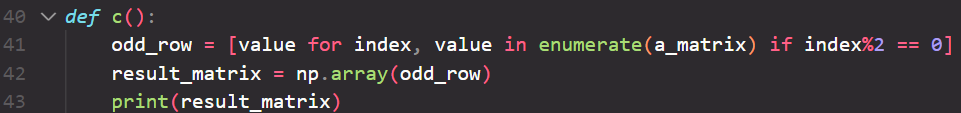


Figure 10: Implement task 1c

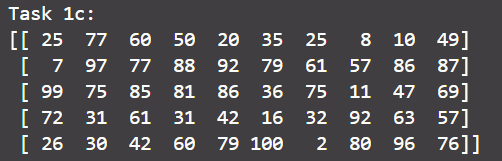


Figure 11: Answer of task 1c

## Task 1d:

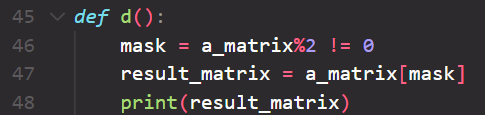


Figure 12: Implement task 1d

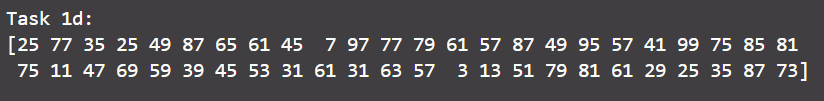


Figure 13: Answer of task 1d

## Task 1e:

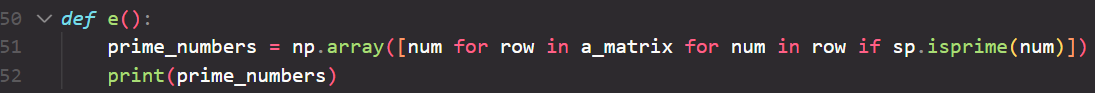


Figure 14: Implement task 1e



Figure 15: Answer of task 1e

## Task 1f:

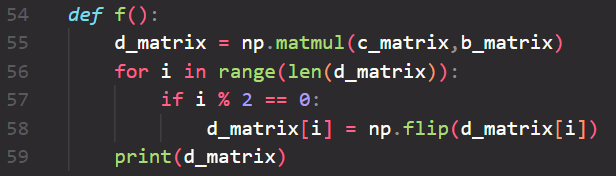


Figure 16: Implement task 1f

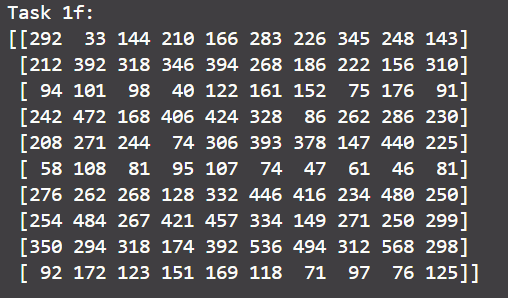


Figure 17: Answer of task 1f

## Task 1g:

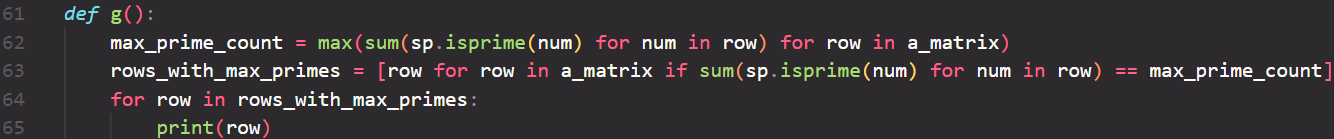


Figure 18: Implement task 1g



Figure 19: Answer of task 1g

## Task 1h:

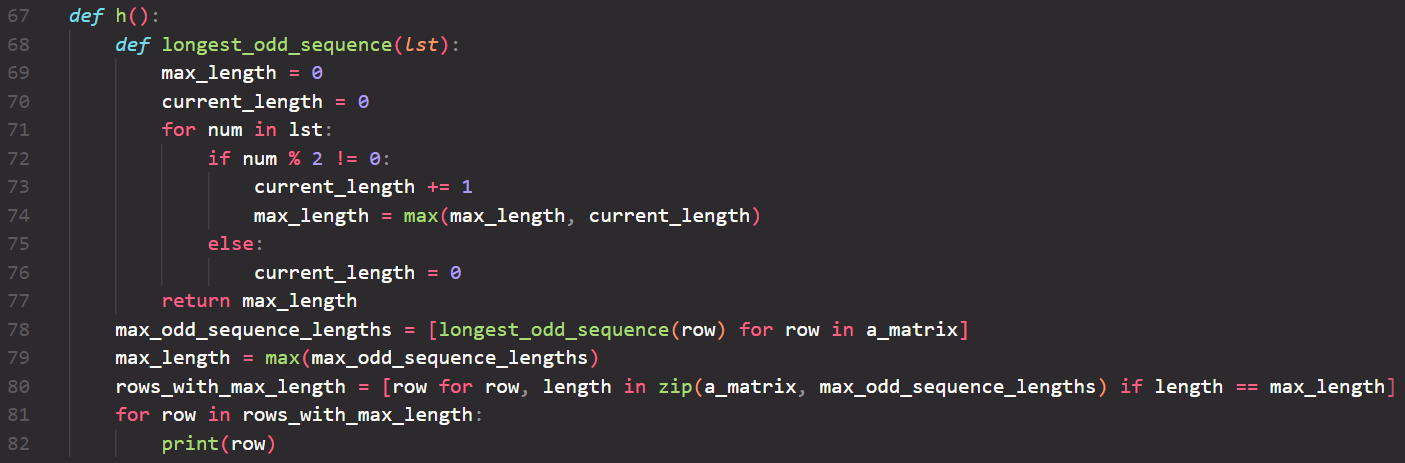


Figure 20: Implement task 1h



Figure 21: Answer of task 1h