# Write a program to calculate Fibonacci numbers and find its step count

# Function to calculate Fibonacci number using recursion

def fibonacci\_recursive(n):

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    else:

        return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2)

# Function to calculate Fibonacci number using iteration (dynamic programming)

def fibonacci\_iterative(n):

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    fib = [0] \* (n + 1)

    fib[1] = 1

    for i in range(2, n + 1):

        fib[i] = fib[i - 1] + fib[i - 2]

    return fib[n]

# Function to count the number of steps needed to calculate a Fibonacci number using recursion

def count\_steps\_recursive(n):

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    else:

        return 1 + count\_steps\_recursive(n - 1) + count\_steps\_recursive(n - 2)

# Function to count the number of steps needed to calculate a Fibonacci number using iteration

def count\_steps\_iterative(n):

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    steps = [0] \* (n + 1)

    steps[1] = 1

    for i in range(2, n + 1):

        steps[i] = 1 + steps[i - 1] + steps[i - 2]

    return steps[n]

# Input the desired Fibonacci number 'n'

n = int(input("Enter the value of n: "))

# Calculate and print the Fibonacci number using recursion

fib\_recursive = fibonacci\_recursive(n)

print(f"Fibonacci number (Recursive) for n = {n}: {fib\_recursive}")

# Calculate and print the number of steps needed using recursion

steps\_recursive = count\_steps\_recursive(n)

print(f"Number of steps (Recursive) to calculate Fibonacci number for n = {n}: {steps\_recursive}")

# Calculate and print the Fibonacci number using iteration

fib\_iterative = fibonacci\_iterative(n)

print(f"Fibonacci number (Iterative) for n = {n}: {fib\_iterative}")

# Calculate and print the number of steps needed using iteration

steps\_iterative = count\_steps\_iterative(n)

print(f"Number of steps (Iterative) to calculate Fibonacci number for n = {n}: {steps\_iterative}")

def generate\_fibonacci\_sequence(n):

    if n <= 0:

        return []

    fibonacci\_sequence = []

    a, b = 0, 1

    while a <= n:

        fibonacci\_sequence.append(a)

        a, b = b, a + b

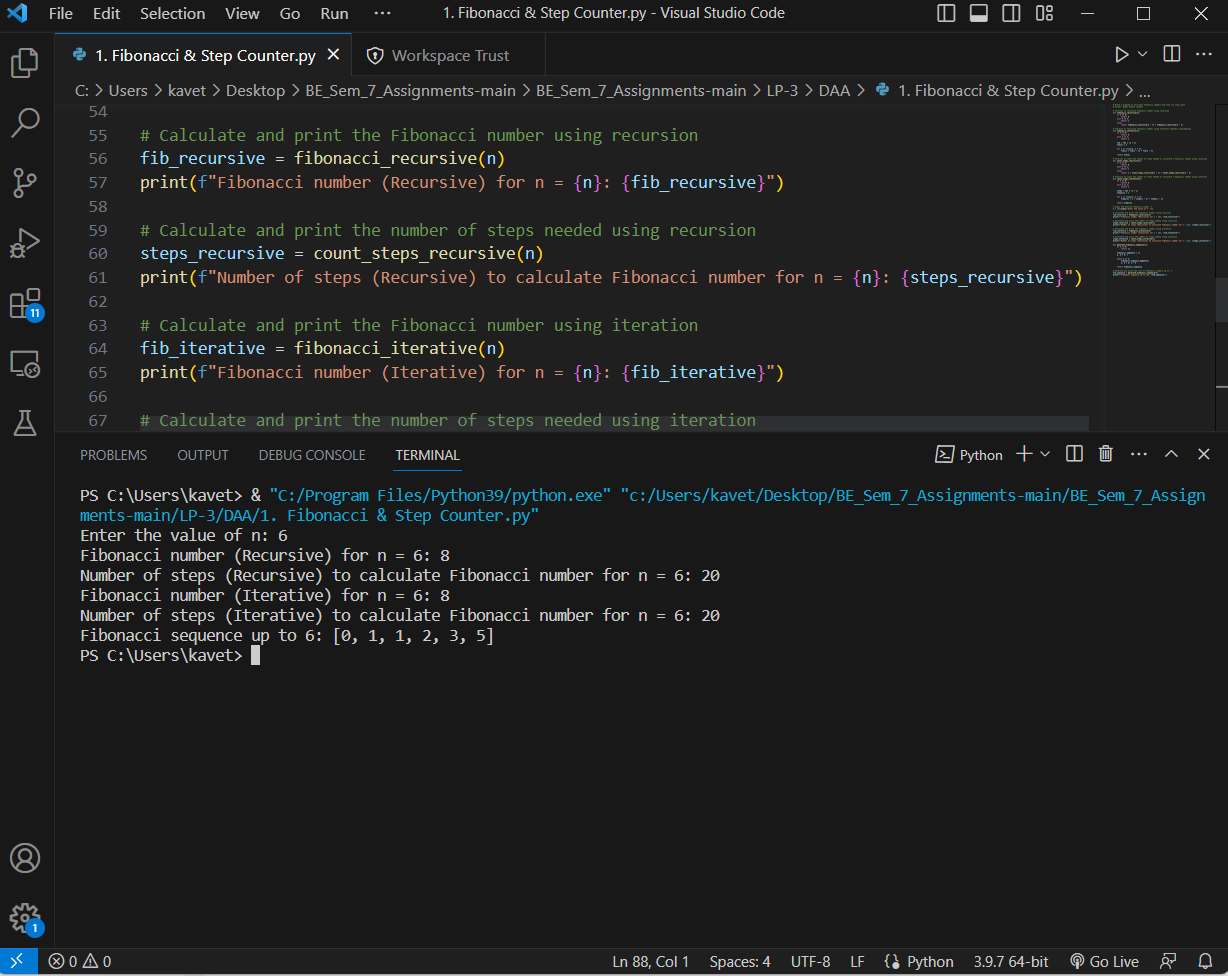
    return fibonacci\_sequence

# Generate and print the list of Fibonacci numbers up to 'n'

fib\_sequence = generate\_fibonacci\_sequence(n)

print(f"Fibonacci sequence up to {n}: {fib\_sequence}")

OUTPUT:



# Write a program to implement Huffman Encoding using a greedy strategy.

import collections, heapq

# d - tree direction (0/1)

Node = collections.namedtuple('Node',['d','freq','lchild','rchild'])

def print\_codes(root, code):

    if root is not None:

        if(root.d != "$"):

            print(f"{root.d}: {code}")

        print\_codes(root.lchild, code + "0")

        print\_codes(root.rchild, code + "1")

def HuffmanCodes(data, frequency):

    min\_heap = []

    for i in range(len(data)):

        heapq.heappush(min\_heap, Node(data[i], frequency[i], None, None))

    while(len(min\_heap) > 1):

        lchild = heapq.heappop(min\_heap)

        rchild = heapq.heappop(min\_heap)

        top = Node("$", lchild.freq + rchild.freq, lchild, rchild)

        heapq.heappush(min\_heap, top)

    print("Huffman Code: ")

    print\_codes(min\_heap[0], "")

def main():

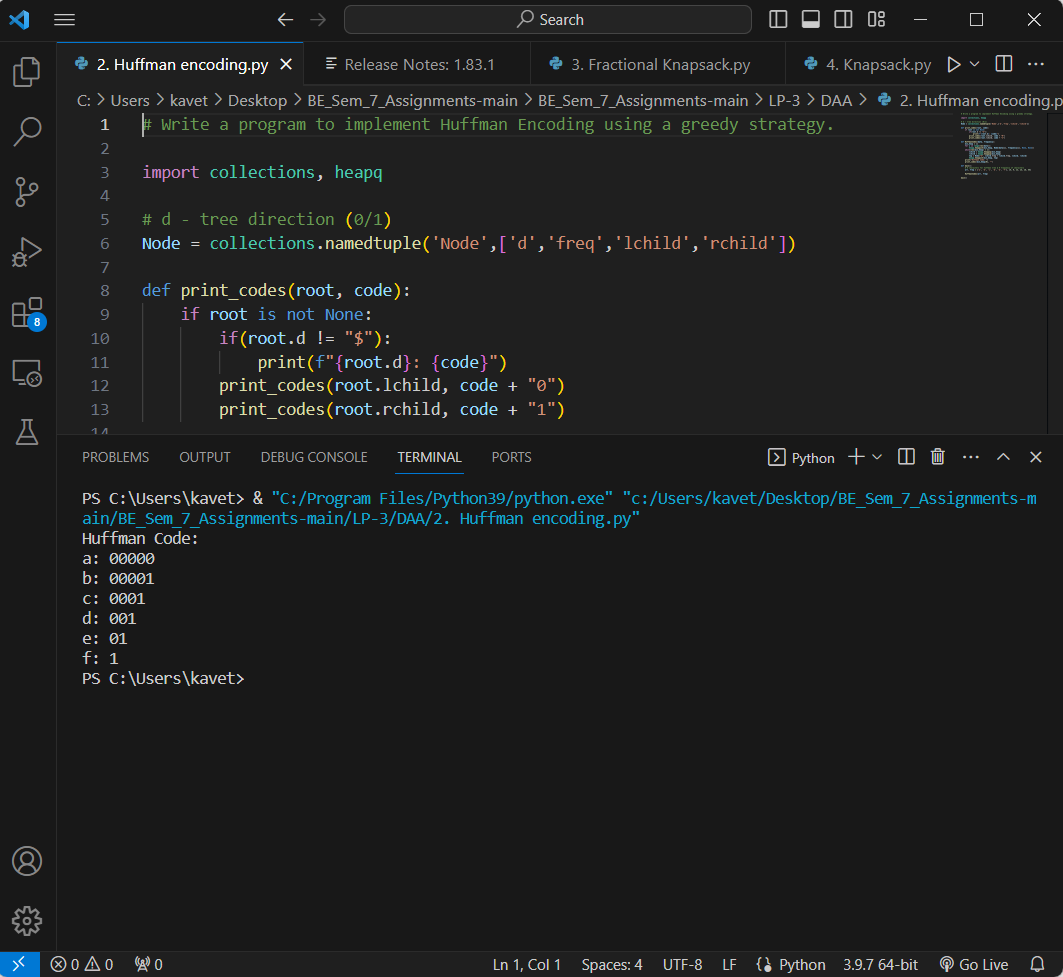
    # # characters for huffman tree & # frequency of characters

    arr, freq = ['a', 'b', 'c', 'd', 'e', 'f'], [5, 9, 12, 13, 16, 45]

    HuffmanCodes(arr, freq)

main()

OUTPUT:



import collections

Item = collections.namedtuple('Item', ['profit', 'weight'])

def FractionalKnapsack2(arr, n, W):

    summ, tot = W, 0

    for i in range(n):

        summ -= arr[i].weight

        if(summ >= 0):

            tot += arr[i].profit

        elif(arr[i].weight >= summ):

            summ += arr[i].weight

            # print("Sum ", summ)

            tot += arr[i].profit \* summ // arr[i].weight

            summ -= arr[i].weight

            # print("Tota ", tot)

        # print("tot ", tot)

    # print("Total ", tot)

    return tot

def FractionalKnapsack(arr, W):

    arr.sort(key=lambda x: (x.profit/x.weight), reverse=True)

    # print(arr)

    ans = 0.0

    for i in arr:

        if(i.weight <= W):

            W -= i.weight

            ans += i.profit

        else:

            ans += i.profit \* W // i.weight

            break

    return ans

def main():

    # # characters for huffman tree & # frequency of characters

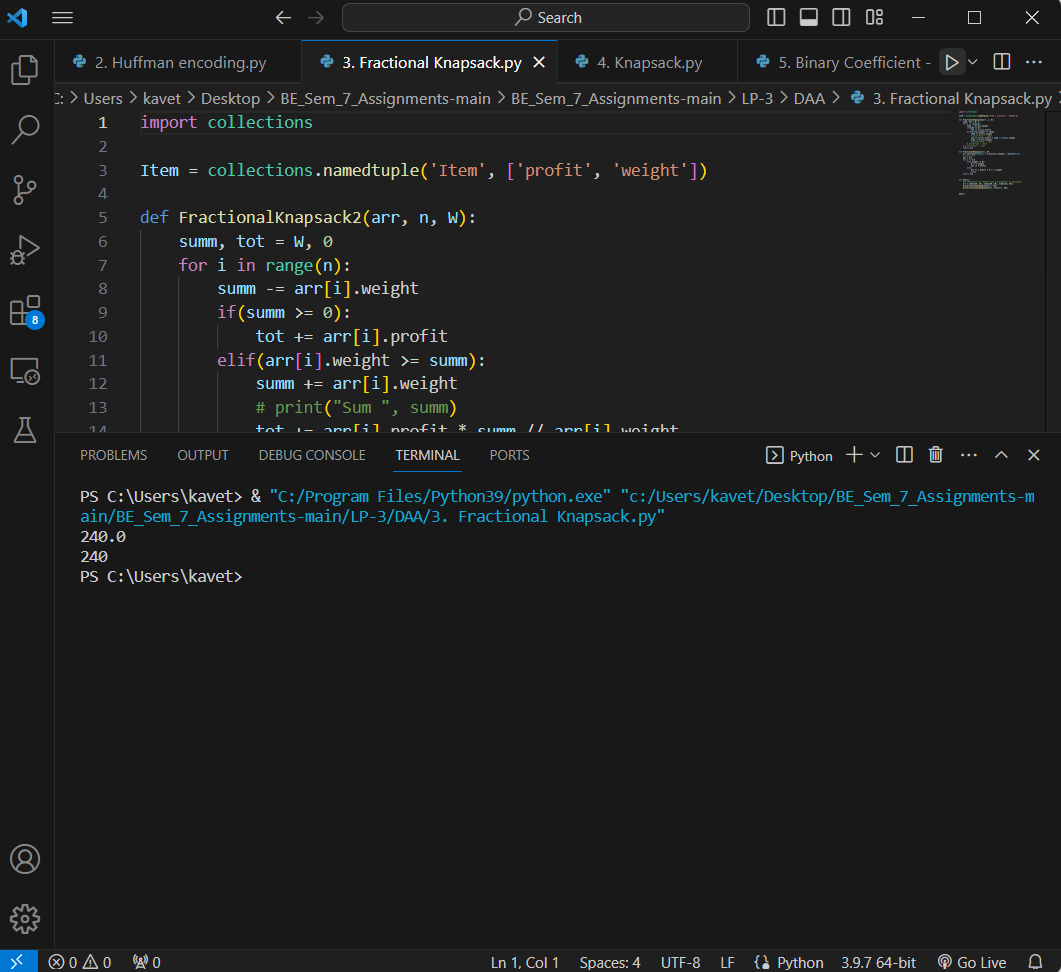
    arr = [Item(60, 10), Item(100, 20), Item(120, 30)]

    print(FractionalKnapsack(arr, 50))

    print(FractionalKnapsack2(arr, len(arr), 50))

main()

OUTPUT:



import collections

Item = collections.namedtuple('Item', ['profit', 'weight'])

def Knapsack(arr, n, w):

    if(n == 0 or w == 0):

        return 0

    if(arr[n-1].weight <= w):

        return max(Knapsack(arr, n-1, w - arr[n-1].weight) + arr[n-1].profit ,

                    Knapsack(arr, n-1, w)

                   )

    else:

        return Knapsack(arr, n-1, w)

def main():

    # # characters for huffman tree & # frequency of characters

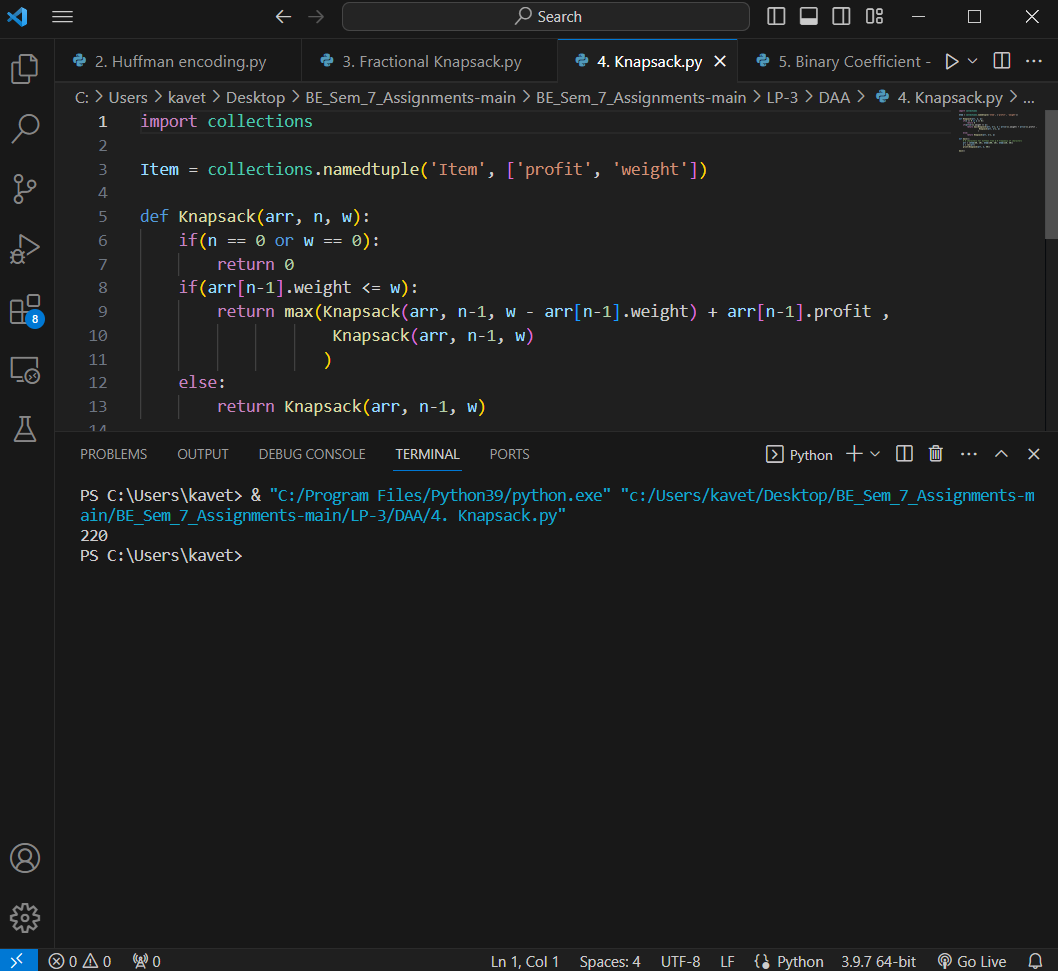
    arr = [Item(60, 10), Item(100, 20), Item(120, 30)]

    n = len(arr)

    print(Knapsack(arr, n, 50))

main()

OUTPUT:



# 5. Write a program to generate binomial coefficients using dynamic programming.

class Solution:

    def getRow(self, rowIndex):

        prev = [1]

        for i in range(1, rowIndex + 1):

            curr = [0] \* (i + 1)

            for j in range(i + 1):

                left = prev[j - 1] if j > 0 else 0

                right = prev[j] if j < i else 0

                curr[j] = left + right

            prev = curr

        return prev

    def generateBinaryCoefficientsTable(self, rows):

        if(rows <= 0):

            return

        print([1])

        for i in range(1, rows+1):

            row = self.getRow(i)

            print(row)

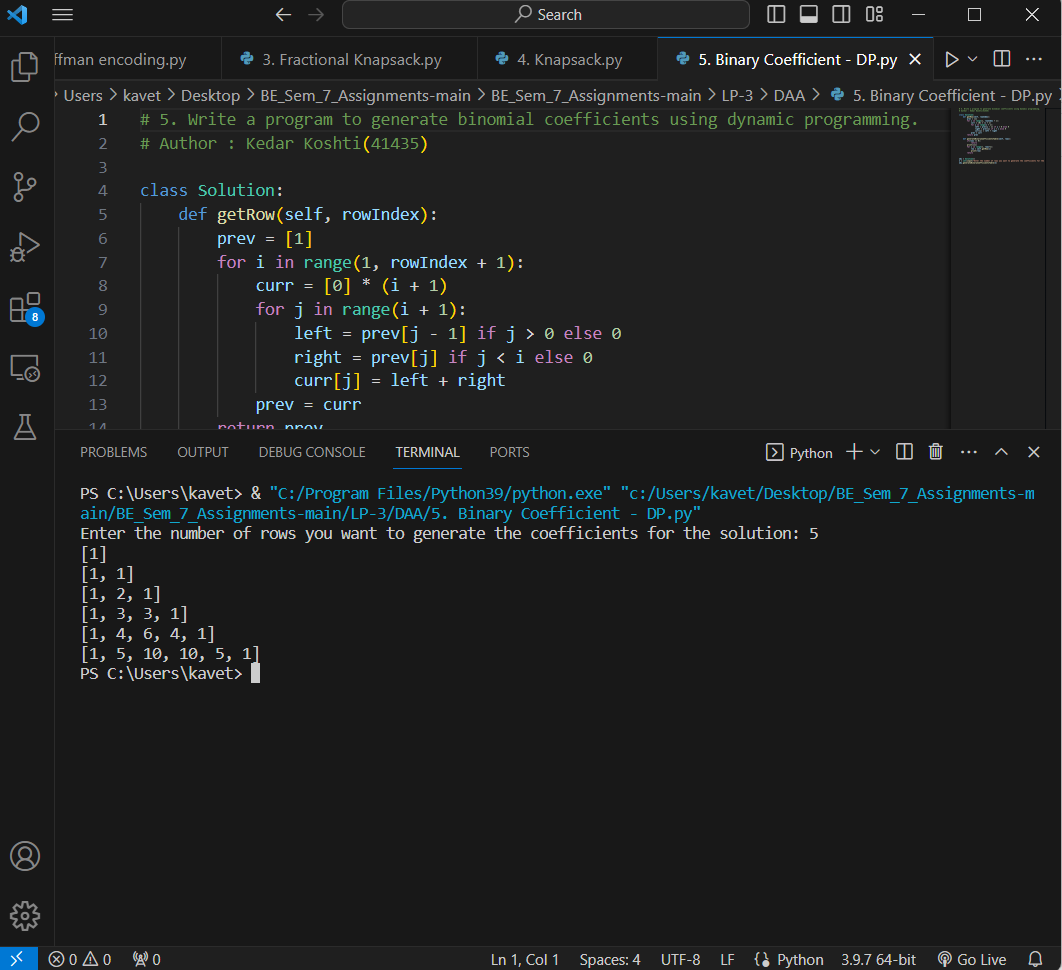
        return

obj = Solution()

n = int(input("Enter the number of rows you want to generate the coefficients for the solution: "))

obj.generateBinaryCoefficientsTable(n)

OUTPUT:



from threading import Thread

MAX, MAX\_THREAD = 4, 4

FinalMatrix = [[0 for i in range(MAX)] for j in range(MAX)]

step\_i = 0

# Function to multiply a row of matrix A  with entire matrix B to get a row of matrix C

def multi():

    global step\_i, FinalMatrix

    i = step\_i

    step\_i += 1

    for j in range(MAX):

        for k in range(MAX):

            FinalMatrix[i][j] += (A[i][k] \* B[k][j])

def normalMultiplication(a, b):

    ans = [[0 for i in range(len(b[0]))] for j in range(len(a))]

    for m in range(len(a)):

        for n in range(len(b[0])):

            for o in range(len(b)):

                ans[m][n] += (a[m][o] \* b[o][n])

    print(ans)

if \_\_name\_\_ == "\_\_main\_\_":

    # A = [[5, 4, 3],

    #      [2, 4, 6],

    #      [4, 7, 9]]

    # B = [[3, 2, 4],

    #      [4, 3, 6],

    #      [2, 7, 5]]

    # [37, 43, 59]

    # [34, 58, 62]

    # [58, 92, 103]

    # normalMultiplication(A, B)

    A = [[3, 7, 3, 6],

         [9, 2, 0, 3],

         [0, 2, 1, 7],

         [2, 2, 7, 9]]

    B = [[6, 5, 5, 2],

         [1, 7, 9, 6],

         [6, 6, 8, 9],

         [0, 3, 5, 2]]

    # creating list of size MAX\_THREAD

    thread = list(range(MAX\_THREAD))

    for i in range(MAX\_THREAD):

        thread[i] = Thread(target=multi)

        thread[i].start()

    # Waiting for all threads to finish

    for i in range(MAX\_THREAD):

        thread[i].join()

    print(FinalMatrix)

    normalMultiplication(A, B)

    # A = [[7 2 6 8

    # 7 0 6 6

    # 6 1 7 5

    # 3 7 7 2]]

    # B = 3 5 3 6

    # 5 7 5 8

    # 8 9 4 9

    # 6 5 7 2

    # Multiplication of A and B

    # 127 143 111 128

    # 105 119 87 108

    # 109 125 86 117

    # 112 137 86 141

# main()

OUTPUT:

