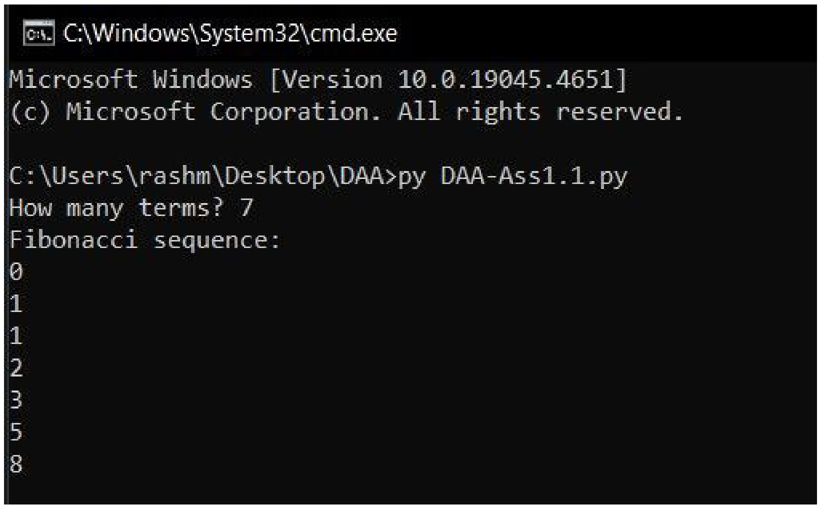
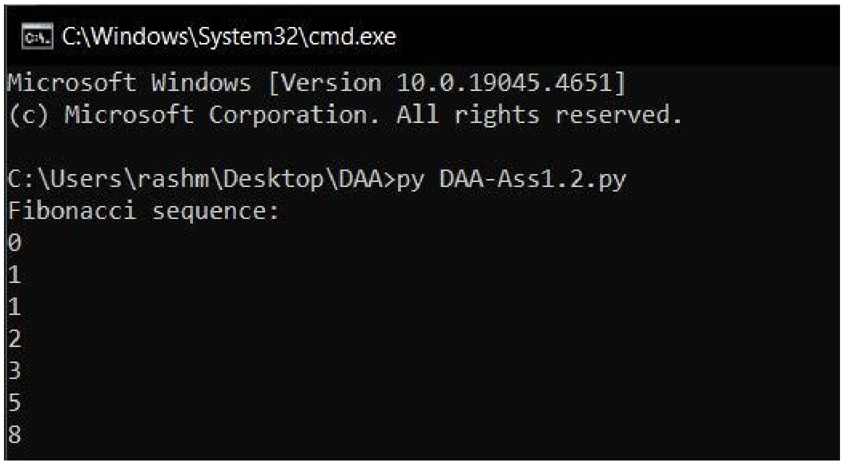
```python
# DAA Assignment No. 1
# non-recursive and recursive program to calculate Fibonacci numbers

# Method-1: NON-RECURSIVE


# Program to display the Fibonacci sequence up to n-th term
nterms = int(input("How many terms? "))

# first two terms
n1, n2 = 0, 1
count = 0

# check if the number of terms is valid
if nterms <= 0:
    print("Please enter a positive integer")

# if there is only one term, return n1
elif nterms == 1:
    print("Fibonaccisequence upto",nterms,":")
    print(n1)

# generate fibonacci sequence
else:
    print("Fibonacci sequence:")

    while count < nterms:
        print(n1)
        nth = n1 + n2

        # update values
        n1 = n2
        n2 = nth
        count += 1
```

```python
# DAA Assignment No. 1
# non-recursive and recursive program to calculate Fibonacci numbers

# Method-2: RECURSIVE


# Python program to display the Fibonacci sequence
def recur_fibo(n):

    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))

nterms = 7

# check if the number of terms is valid
if nterms <= 0:
    print("Plese enter a positive integer")

else:
    print("Fibonacci sequence:")

    for i in range(nterms):
        print(recur_fibo(i))
```

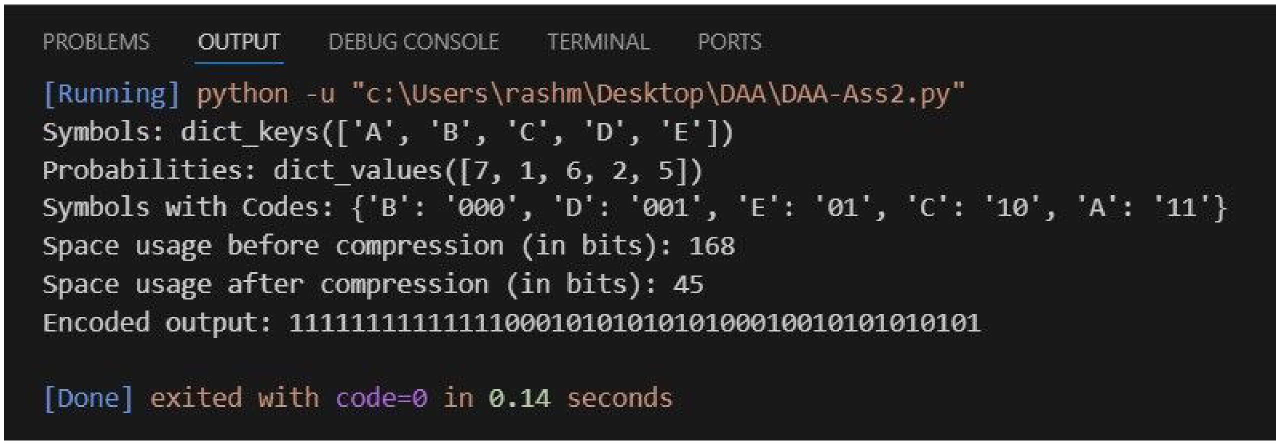


```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.4651]
(c) Microsoft Corporation. All rights reserved.

C:\Users\rashm\Desktop\DAA>py DAA-Ass1.1.py
How many terms? 7
Fibonacci sequence:
0
1
1
2
3
5
8
```


C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 10.0.19045.4651]
(c) Microsoft Corporation. All rights reserved.

C:\Users\rashm\Desktop\DAA>py DAA-Ass1.2.py
Fibonacci sequence:
0
1
1
2
3
5
8
```

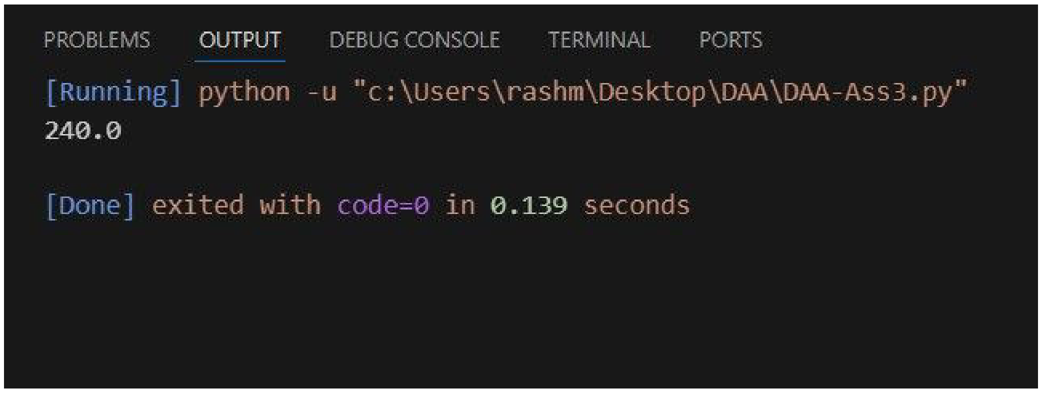
```python
# DAA Assignment No. 2
# Implement Huffman Encoding using a greedy strategy.

class Node:
    def __init__(self, prob, symbol, left=None, right=None):
        # probability of symbol
        self.prob = prob

        # symbol
        self.symbol = symbol

        # left node
        self.left = left

        # right node
        self.right = right

        # tree direction (0/1)
        self.code = ''

def Calculate_Probability(data):
    symbols = dict()
    for element in data:
        if symbols.get(element) is None:
            symbols[element] = 1
        else:
            symbols[element] += 1
    return symbols

def Calculate_Codes(node, val=''):
    newVal = val + str(node.code)

    # If the node is a leaf node, return its code
    if node.left is None and node.right is None:
        return {node.symbol: newVal}

    codes = {}
    if node.left:
        codes.update(Calculate_Codes(node.left, newVal))
    if node.right:
        codes.update(Calculate_Codes(node.right, newVal))

    return codes

def Output_Encoded(data, coding):
    encoding_output = []
    for c in data:
        encoding_output.append(coding[c])
    string = ''.join([str(item) for item in encoding_output])
    return string

def Total_Gain(data, coding):
    before_compression = len(data) * 8  # total bit space to store the data
before compression

    after_compression = 0
    symbols = coding.keys()
    for symbol in symbols:
        count = data.count(symbol)
        after_compression += count * len(coding[symbol])  # calculate how
many bits is required after compression

    print("Space usage before compression (in bits):",
before_compression)
    print("Space usage after compression (in bits):", after_compression)

def Huffman_Encoding(data):
    symbol_with_probs = Calculate_Probability(data)
    symbols = symbol_with_probs.keys()
    probabilities = symbol_with_probs.values()
    print("Symbols:", symbols)
    print("Probabilities:", probabilities)

    nodes = []

    # Converting symbols and probabilities into Huffman tree nodes
    for symbol in symbols:
        nodes.append(Node(symbol_with_probs.get(symbol), symbol))

    while len(nodes) > 1:
        # Sort all the nodes in ascending order based on their probability
        nodes = sorted(nodes, key=lambda x: x.prob)

        # Pick 2 smallest nodes
        left = nodes[0]
        right = nodes[1]

        left.code = '0'
        right.code = '1'

        # Combine the 2 smallest nodes to create a new node
        newNode = Node(left.prob + right.prob, left.symbol + right.symbol,
left, right)

        nodes.remove(left)
        nodes.remove(right)
        nodes.append(newNode)

    huffman_encoding = Calculate_Codes(nodes[0])
    print("Symbols with Codes:", huffman_encoding)
    Total_Gain(data, huffman_encoding)
    encoded_output = Output_Encoded(data, huffman_encoding)
    print("Encoded output:", encoded_output)
    return encoded_output, nodes[0]

# Driver code
if __name__ == "__main__":
    data = "AAAAAAABCCCCCCDDEEEEE"
    encoded_data, tree = Huffman_Encoding(data)
```



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

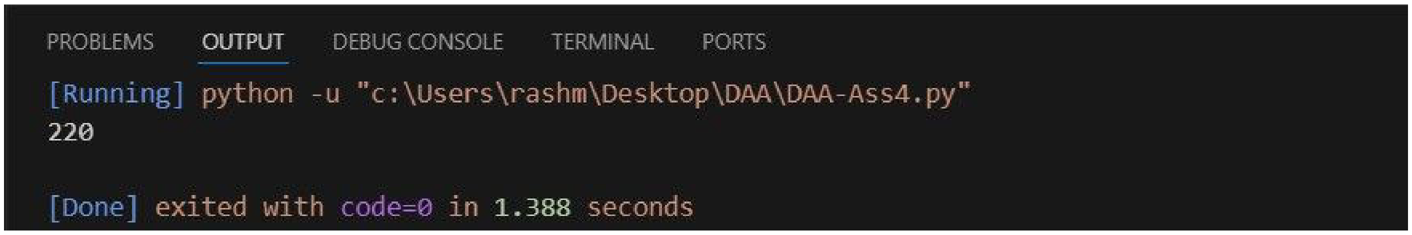
```
[Running] python -u "c:\Users\rashm\Desktop\DAA\DAA-Ass2.py"
Symbols: dict_keys(['A', 'B', 'C', 'D', 'E'])
Probabilities: dict_values([7, 1, 6, 2, 5])
Symbols with Codes: {'B': '000', 'D': '001', 'E': '01', 'C': '10', 'A': '11'}
Space usage before compression (in bits): 168
Space usage after compression (in bits): 45
Encoded output: 111111111111110001010101010100010010101010101

[Done] exited with code=0 in 0.14 seconds
```

```python
# DAA Assignment No. 3
# Solve a fractional Knapsack problem using a greedy method

class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight

def fractionalKnapsack(W, arr):
    # Sorting Items on basis of ratio
    arr.sort(key=lambda x: (x.value/x.weight), reverse=True)

    # Result (value in Knapsack)
    finalvalue = .0

    # Looping through all Items
    for item in arr:
        # If adding the Item won't overflow, add it completely
        if item.weight <= W:
            W -= item.weight
            finalvalue += item.value
        # If we can't add the current Item,
        # add fractional part of it
        else:
            finalvalue += item.value * (W / item.weight)
            break

    # Returning final value
    return finalvalue

# Driver Code
if __name__ == "__main__":
    W = 50
    arr = [Item(60, 10), Item(100, 20), Item(120, 30)]

    # Function call
    max_val = fractionalKnapsack(W, arr)
    print(max_val)
```



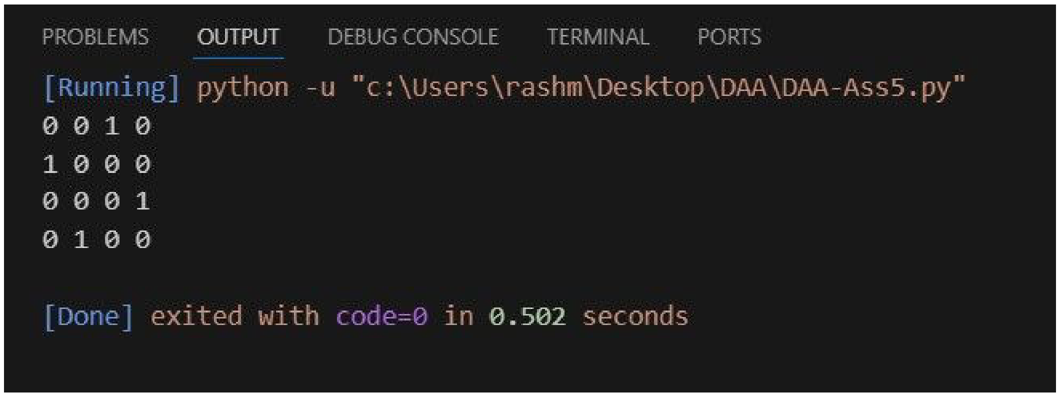
```
[Running] python -u "c:\Users\rashm\Desktop\DAA\DAA-Ass3.py"
240.0

[Done] exited with code=0 in 0.139 seconds
```

```
# DAA Assignment No. 4

# code
# A Dynamic Programming based Python
# Program for 0-1 Knapsack problem
# Returns the maximum value that can
# be put in a knapsack of capacity W

def knapSack(W, wt, val, n):
    dp = [0 for i in range(W+1)] # Making the dp array

    for i in range(1, n+1): # taking first i elements

        for w in range(W, 0, -1): # starting from back,so that we also have
data of
                                 # previous computation when taking i-1 items

            if wt[i-1] <= w:

                # finding the maximum value
                dp[w] = max(dp[w], dp[w-wt[i-1]]+val[i-1])

    return dp[W] # returning the maximum value of knapsack

# Driver code
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```



```
[Running] python -u "c:\Users\rashm\Desktop\DAA\DAA-Ass4.py"
220

[Done] exited with code=0 in 1.388 seconds
```

```
# DAA Assignment No. 5

# Python3 program to solve N Queen
# Problem using backtracking

global N
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end = " ")
        print()

# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens

def isSafe(board, row, col):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if board[i][j] == 1:
                return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solveNQUtil(board, col):
    # base case: If all queens are placed
    # then return true
    if col >= N:
        return True

    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):
        if isSafe(board, i, col):
        # Place this queen in board[i][col]
            board[i][col] = 1
            # recur to place rest of the queens

            if solveNQUtil(board, col + 1) == True:
                return True

            # If placing queen in board[i][col
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0

    # if the queen can not be placed in any row in
    # this column col then return false

    return False

# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.

def solveNQ():
    board = [ [0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0] ]
    if solveNQUtil(board, 0) == False:
        print ("Solution does not exist")
        return False

    printSolution(board)
    return True

# Driver Code
solveNQ()
```



```
[Running] python -u "c:\Users\rashm\Desktop\DAA\DAA-Ass5.py"
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

[Done] exited with code=0 in 0.502 seconds
```

DESIGN AND ANALYSIS OF ALGORITHMS

MINI PROJECT

Title: Write a Program to Implement Matrix Multiplication. Also Implement Multithreaded Matrix Multiplication with Either One Thread per Row or One Thread per Cell. Analyze and Compare Their Performance.

Abstract:

This document presents a program that demonstrates matrix multiplication using both single-threaded and multithreaded approaches. The performance of both methods is analyzed and compared based on execution time. The multithreaded approach leverages the capabilities of modern processors by distributing the workload across multiple threads, thus potentially reducing computation time. The study highlights the efficiency of multithreading in matrix operations, providing insights into performance optimization techniques.

Introduction:

Matrix multiplication is a fundamental operation in various fields, including computer science, physics, and engineering. Traditional matrix multiplication has a time complexity of $O(n^3)$, which can be computationally expensive for large matrices. This paper explores how multithreading can enhance performance by allowing simultaneous computation, thereby utilizing multiple processor cores effectively. The implementation focuses on multiplying two matrices of fixed size, demonstrating both single-threaded and multithreaded methods.

Objectives:

1. To implement matrix multiplication using a single-threaded approach.
2. To implement matrix multiplication using a multithreaded approach (one thread per row).
3. To analyze and compare the performance of both methods based on execution time.
4. To provide insights into the efficiency of multithreading in computational tasks.

System Requirements:

- A system with a minimum of dual-core processor.
- C++ compiler supporting C++11 or later (e.g., g++, MinGW).
- Basic libraries: `<iostream>`, `<pthread.h>`, `<cstdlib>`, `<chrono>`.

Methodology:

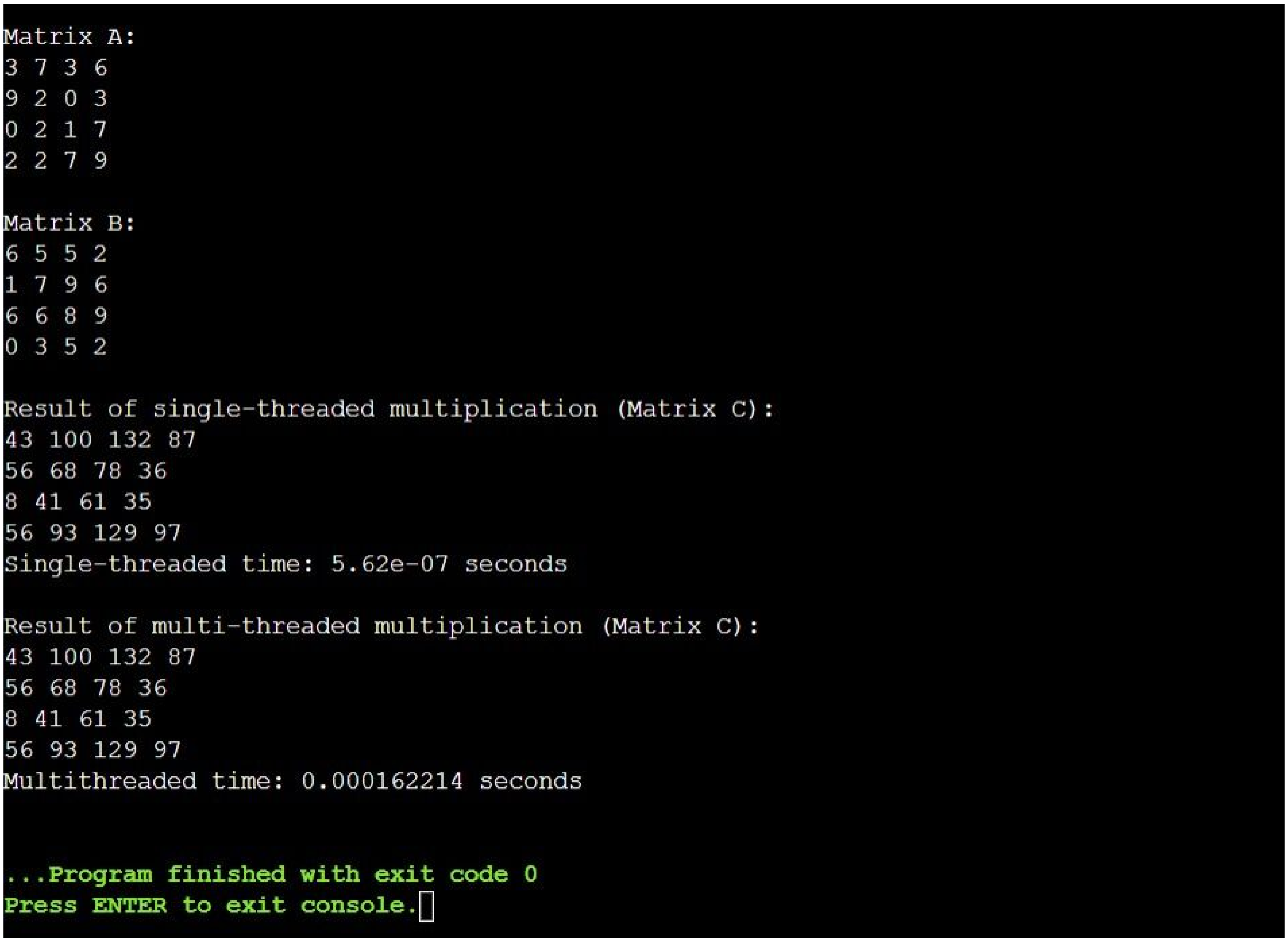
1. **Matrix Generation**: Random matrices are generated for multiplication.
2. **Single-threaded Multiplication**: A function is created to perform matrix multiplication using nested loops.
3. **Multithreaded Multiplication**:
    - Multiple threads are created, with each thread responsible for computing one row of the resultant matrix.
    - The `pthread_create` function is used to create threads, and `pthread_join` is used to ensure all threads complete before proceeding.
4. **Performance Measurement**: The execution time of both methods is measured using `chrono` for accurate timing.
5. **Output**: The resultant matrices from both methods are displayed, along with their execution times.

Conclusion:

The program successfully demonstrates matrix multiplication using both single-threaded and multithreaded approaches. The results indicate that while the single-threaded approach is straightforward, the multithreaded approach can lead to performance improvements, particularly in larger matrices. However, for smaller matrices, the overhead of thread creation may negate these benefits. This exploration highlights the importance of choosing the appropriate method based on matrix size and system architecture, emphasizing the value of multithreading in enhancing computational efficiency.

```
# DAA MINI PROJECT (Multithreaded Matrix Multiplication: A Comparative
Performance Analysis)


#include <iostream>
#include <pthread.h>
#include <cstdlib>
#include <chrono>

using namespace std;

#define MAX 4 // Maximum size of the matrix
#define MAX_THREAD 4 // Maximum number of threads

int matA[MAX][MAX];
int matB[MAX][MAX];
int matC[MAX][MAX];
int step_i = 0;

// Function for single-threaded matrix multiplication
void multiplySingleThreaded()
{
    for (int i = 0; i < MAX; i++)
     {
        for (int j = 0; j < MAX; j++)
           {
            matC[i][j] = 0; // Initialize the element
            for (int k = 0; k < MAX; k++)
                {
                matC[i][j] += matA[i][k] * matB[k][j];
            }
        }
    }
}

// Function for multithreading (one thread per row)
void* multi(void* arg)
{
    int i = step_i++; // i denotes row number of resultant matC
    for (int j = 0; j < MAX; j++)
     {
        for (int k = 0; k < MAX; k++)
           {
            matC[i][j] += matA[i][k] * matB[k][j];
        }
    }
```

```
}

void printMatrix(int matrix[MAX][MAX])
{
    for (int i = 0; i < MAX; i++)
     {
        for (int j = 0; j < MAX; j++)
           {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}

int main()
{
    // Generating random values in matA and matB
    for (int i = 0; i < MAX; i++)
     {
        for (int j = 0; j < MAX; j++)
           {
            matA[i][j] = rand() % 10;
            matB[i][j] = rand() % 10;
        }
    }

    cout << "\nMatrix A:" << endl;
    printMatrix(matA);

    cout << "\nMatrix B:" << endl;
    printMatrix(matB);

    // Measure time for single-threaded multiplication
    auto start = chrono::high_resolution_clock::now();
    multiplySingleThreaded();
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> singleThreadedTime = end - start;

    cout << "\nResult of single-threaded multiplication (Matrix C):" <<
endl;
    printMatrix(matC);
    cout << "Single-threaded time: " << singleThreadedTime.count() << "
seconds" << endl;

    // Reset matC
    fill(&matC[0][0], &matC[0][0] + sizeof(matC) / sizeof(matC[0][0]),0);
    // Declaring four threads for multithreading
    pthread_t threads[MAX_THREAD];

    // Measure time for multithreaded multiplication
    start = chrono::high_resolution_clock::now();
    for (int i = 0; i < MAX_THREAD; i++)
      {
        pthread_create(&threads[i], NULL, multi, NULL);
    }
    for (int i = 0; i < MAX_THREAD; i++)
      {
        pthread_join(threads[i], NULL);
    }
    end = chrono::high_resolution_clock::now();
    chrono::duration<double> multiThreadedTime = end - start;

    cout << "\nResult of multi-threaded multiplication (Matrix C):" <<
endl;
    printMatrix(matC);
    cout << "Multithreaded time: " << multiThreadedTime.count() << "
seconds" << endl;

    return 0;
}
```



```
Matrix A:
3 7 3 6
9 2 0 3
0 2 1 7
2 2 7 9

Matrix B:
6 5 5 2
1 7 9 6
6 6 8 9
0 3 5 2

Result of single-threaded multiplication (Matrix C):
43 100 132 87
56 68 78 36
8 41 61 35
56 93 129 97
Single-threaded time: 5.62e-07 seconds

Result of multi-threaded multiplication (Matrix C):
43 100 132 87
56 68 78 36
8 41 61 35
56 93 129 97
Multithreaded time: 0.000162214 seconds


...Program finished with exit code 0
Press ENTER to exit console.
```