2024

Data structures and algorithms

ANDREAS KOKKINOS

glyndŵr
PRIFYSGOL GLYNDŴR WRECSAM
GLYNDŴR UNIVERSITY WREXHAM

Computing

Faculty of Arts, Science and Technology

BSc (Hons) Cyber Security / Computer Science / Computer Networks & Security

Level:	**5**
Module:	**COM539 Data Structures and Algorithms**
Assignment:	**Portfolio**
Issue Date:	**Monday 11th November 2024**
Review Date:	**by Friday 3rd January 2025**
Submission Date :	**Friday 3rd January 2025**
Estimated Completion time:	**Session time *plus 50 Hours***
Module Leader:	**Vic Grout**
Internal Verifier:	**Nigel Houlden**

To be completed by student:

I certify that, other than where collaboration has been explicitly permitted, this work is the result of my individual effort and that all sources for materials have been acknowledged. I also confirm that I have read and understood the codes of practice on plagiarism contained within the Glyndwr Academic Regulations and that, by signing this printed form or typing my name on an electronically submitted version, I am agreeing to be dealt with accordingly in any case of suspected unfair practice. I also certify that my attendance for the module has been at least 70%

Name: Andreas Kokkinos

Student Number: 1452

Date Submitted: .....

Student Signature: AK

Are extenuating circumstances being claimed? YES / NO

If YES, give reference number: ----------

To be completed by lecturer
Comments:

Grade / Mark
(Indicative: may change when moderated)

Table of Contents

The code: **3**

The first case: **23**

Second case: **25**

Third case: **26**

Fourth case: **27**

Fifth case: **29**

Sixth and Seventh case: **30**

Sources **31**

The code:

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    // Insertion Sort with Counter
public static void InsertionSort(double[] arr)
{
    int counter = 0; // Counter to count the number of iterations in the outer loop
    for (int i = 1; i < arr.Length; i++)
    {
        counter++; // Increment the counter for each iteration of the outer loop
        double key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1] that are greater than key
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
    Console.WriteLine($"The Insertion Sort loop executed {counter} times.");
}
```

```
    // 2D Array Addition
    public static double[,] AddMatrices(double[,] matrix1, double[,] matrix2)
{
    if (matrix1.GetLength(0) != matrix2.GetLength(0) || matrix1.GetLength(1) !=
matrix2.GetLength(1))
        throw new ArgumentException("Matrices dimensions do not match.");

    int rows = matrix1.GetLength(0);
    int cols = matrix1.GetLength(1);
    double[,] result = new double[rows, cols];

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            result[i, j] = matrix1[i, j] + matrix2[i, j];
        }
    }

    return result;
}


    public static double GetValidDouble(string prompt)
{
    double result;
    while (true)
    {
```

```
        Console.Write(prompt);
        string input = Console.ReadLine();
        if (double.TryParse(input, out result))
        {
            return result;
        }
        else
        {
            Console.WriteLine("Invalid input. Please enter a valid number.");
        }
    }
}


    // Function to get a valid positive integer input from the user
    public static int GetValidPositiveInteger(string prompt)
    {
        int result;
        while (true)
        {
            Console.Write(prompt);
            string input = Console.ReadLine();
            if (int.TryParse(input, out result) && result > 0)
            {
            return result;
            }
            else
            {
                Console.WriteLine("Invalid input. Please enter a positive integer.");
```

```
            }

        }

    }

    public static void PrintMatrix(double[,] matrix)
{
    int rows = matrix.GetLength(0);
    int cols = matrix.GetLength(1);
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            Console.Write(matrix[i, j] + " ");
        }
        Console.WriteLine();
    }
}



    // 2D Array Multiplication (Modified for Double)
public static double[,] MultiplyMatrices(double[,] matrix1, double[,] matrix2)
{
    if (matrix1.GetLength(1) != matrix2.GetLength(0))
        throw new ArgumentException("Matrix multiplication is not possible. Columns of
Matrix 1 must match rows of Matrix 2.");


    int rows = matrix1.GetLength(0);
    int cols = matrix2.GetLength(1);
```

```
    int common = matrix1.GetLength(1);
    double[,] result = new double[rows, cols];

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            for (int k = 0; k < common; k++)
            {
                result[i, j] += matrix1[i, k] * matrix2[k, j];
            }
        }
    }

    return result;
}


    public static void PrintMatrix(int[,] matrix)
    {
        int rows = matrix.GetLength(0);
        int cols = matrix.GetLength(1);
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                Console.Write(matrix[i, j] + " ");
            }
            Console.WriteLine();
        }
```

```
    }

    // Dijkstra's Algorithm
    public static void Dijkstra(Dictionary<int, List<Tuple<int, int>>> graph, int source)
    {
        if (!graph.ContainsKey(source))
        {
            Console.WriteLine("Source node not found in the graph.");
            return;
        }

        var distances = new Dictionary<int, int>();
        var priorityQueue = new SortedSet<Tuple<int, int>>(Comparer<Tuple<int,
int>>.Create((x, y) =>
        {
            return x.Item2 == y.Item2 ? x.Item1.CompareTo(y.Item1) :
x.Item2.CompareTo(y.Item2);
        }));

        foreach (var node in graph.Keys)
        {
            distances[node] = int.MaxValue;
        }

        distances[source] = 0;
        priorityQueue.Add(new Tuple<int, int>(source, 0));

        while (priorityQueue.Count > 0)
        {
            var current = priorityQueue.Min;
            priorityQueue.Remove(current);
```

```
            int currentNode = current.Item1;
            int currentDistance = current.Item2;

            foreach (var neighbor in graph[currentNode])
            {
                int nextNode = neighbor.Item1;
                int weight = neighbor.Item2;
                int newDist = currentDistance + weight;

                if (newDist < distances[nextNode])
                {
                    priorityQueue.Remove(new Tuple<int, int>(nextNode, distances[nextNode]));
                    distances[nextNode] = newDist;
                    priorityQueue.Add(new Tuple<int, int>(nextNode, newDist));
                }
            }
        }

        Console.WriteLine("Shortest distances from node " + source + ":");
        foreach (var kvp in distances)
        {
            Console.WriteLine($"Node {kvp.Key}: Distance {kvp.Value}");
        }
    }

    // Function to generate a random graph for Dijkstra's algorithm
    public static Dictionary<int, List<Tuple<int, int>>> GenerateRandomGraph(int
numNodes)
    {
```

```
        Random rand = new Random();
        var graph = new Dictionary<int, List<Tuple<int, int>>>();

        for (int i = 1; i <= numNodes; i++)
        {
            var neighbors = new List<Tuple<int, int>>();
            for (int j = 1; j <= numNodes; j++)
            {
                if (i != j) // Do not connect a node to itself
                {
                    int weight = rand.Next(1, 10); // Random weight between 1 and 9
                    neighbors.Add(new Tuple<int, int>(j, weight));
                }
            }
            graph[i] = neighbors;
        }

        return graph;
    }
public static string CaesarCipherEncrypt(string text, int shift)
{
    char[] encrypted = new char[text.Length];
    shift = (shift % 26 + 26) % 26;  // Adjust for negative shifts

    for (int i = 0; i < text.Length; i++)
    {
        char ch = text[i];
        if (char.IsLetter(ch))
        {
            char offset = char.IsUpper(ch) ? 'A' : 'a';
```

```
            encrypted[i] = (char)((ch + shift - offset) % 26 + offset);
        }
        else
        {
            encrypted[i] = ch;
        }
    }
    return new string(encrypted);
}

public static string CaesarCipherDecrypt(string text, int shift)
{
    return CaesarCipherEncrypt(text, 26 - (shift % 26 + 26) % 26);  // Adjust for negative shifts
}


    // Function to get a valid integer input from the user
    public static int GetValidInteger(string prompt)
    {
        int result;
        while (true)
        {
            Console.Write(prompt);
            string input = Console.ReadLine();
            if (int.TryParse(input, out result))
            {
                return result;
            }
            else
```

```
        {
            Console.WriteLine("Invalid input. Please enter a valid integer.");
        }
    }
}

// Prim's Algorithm for Minimum Spanning Tree
public static void MinimumSpanningTree(Dictionary<int, List<Tuple<int, int>>> graph)
{
    Console.WriteLine("Finding Minimum Spanning Tree (MST) using Prim's Algorithm...");

    // Initialize the required structures
    var mstEdges = new List<Tuple<int, int, int>>(); // Store MST edges (node1, node2, weight)
    var visited = new HashSet<int>();
    var priorityQueue = new SortedSet<Tuple<int, int, int>>(Comparer<Tuple<int, int, int>>.Create((x, y) =>
    {
        return x.Item3 == y.Item3 ? (x.Item1.CompareTo(y.Item1)) : x.Item3.CompareTo(y.Item3);
    }));

    // Start from node 1 (or any node)
    visited.Add(1);
    foreach (var neighbor in graph[1])
    {
        priorityQueue.Add(new Tuple<int, int, int>(1, neighbor.Item1, neighbor.Item2));
    }

    while (priorityQueue.Count > 0)
```

```
        {
            var edge = priorityQueue.Min;
            priorityQueue.Remove(edge);

            int node1 = edge.Item1;
            int node2 = edge.Item2;
            int weight = edge.Item3;

            if (!visited.Contains(node2))
            {
                mstEdges.Add(edge);
                visited.Add(node2);

                foreach (var neighbor in graph[node2])
                {
                    if (!visited.Contains(neighbor.Item1))
                    {
                        priorityQueue.Add(new Tuple<int, int, int>(node2, neighbor.Item1,
neighbor.Item2));
                    }
                }
            }
        }

        Console.WriteLine("Minimum Spanning Tree (MST):");
        foreach (var edge in mstEdges)
        {
            Console.WriteLine($"{edge.Item1} - {edge.Item2}: {edge.Item3}");
        }
    }
```

```
// Main Function
static void Main()
{
    while (true)
    {
        Console.WriteLine("Welcome! Choose an operation to perform:");
        Console.WriteLine("1. Insertion Sort");
        Console.WriteLine("2. 2D Array Addition");
        Console.WriteLine("3. 2D Array Multiplication");
        Console.WriteLine("4. Dijkstra's Algorithm");
        Console.WriteLine("5. Caesar Cipher");
        Console.WriteLine("6. Graph Algorithms (Minimum Spanning Tree, etc.)");
        Console.WriteLine("7. Exit");

        Console.Write("Enter the number of the operation you want to perform (1-7): ");
        string input = Console.ReadLine();

        int operation;
        while (!int.TryParse(input, out operation) || operation < 1 || operation > 7)
        {
            Console.WriteLine("Invalid choice. Please enter a number between 1 and 7.");
            input = Console.ReadLine();
        }

       if (operation == 7) break;

        if (operation == 1)
{
```

```
    double[] numbers = new double[10];  // Create an array of size 10
    int count = 0;  // Count how many numbers the user has entered

    while (count < 10)
    {
        Console.WriteLine($"Enter number {count + 1} of up to 10 numbers to sort:");

        string inputNumber = Console.ReadLine();
        bool success = double.TryParse(inputNumber, out numbers[count]); // Allow real numbers

        if (success)
        {
            count++;
        }
        else
        {
            Console.WriteLine("Invalid input. Please enter a valid number.");
        }
    }

    InsertionSort(numbers); // Call the updated InsertionSort method
    Console.WriteLine("Sorted array: " + string.Join(", ", numbers));
}

            else if (operation == 2)
{
    try
    {
        Console.WriteLine("Enter the number of rows and columns for the matrices:");
```

```
    int rows = GetValidPositiveInteger("Number of rows: ");
    int cols = GetValidPositiveInteger("Number of columns: ");

    double[,] matrix1 = new double[rows, cols];
    double[,] matrix2 = new double[rows, cols];

    Console.WriteLine("Enter the elements of the first matrix:");
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            matrix1[i, j] = GetValidDouble($"Matrix1[{i}, {j}]: ");
        }
    }

    Console.WriteLine("Enter the elements of the second matrix:");
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            matrix2[i, j] = GetValidDouble($"Matrix2[{i}, {j}]: ");
        }
    }

    double[,] result = AddMatrices(matrix1, matrix2);
    Console.WriteLine("Sum of the matrices:");
    PrintMatrix(result);
}
catch (Exception ex)
```

```
    {
        Console.WriteLine("Error: " + ex.Message);
    }
}


else if (operation == 3)
{
    try
    {
        int rows1, cols1, rows2, cols2;

        while (true) // Loop until valid dimensions are provided
        {
            // Prompt the user for dimensions of both matrices
            rows1 = GetValidPositiveInteger("Number of rows in Matrix 1: ");
            cols1 = GetValidPositiveInteger("Number of columns in Matrix 1: ");
            rows2 = GetValidPositiveInteger("Number of rows in Matrix 2: ");
            cols2 = GetValidPositiveInteger("Number of columns in Matrix 2: ");

            if (cols1 == rows2) // Check if dimensions are valid for multiplication
                break; // Exit the loop if valid
            else
                Console.WriteLine("Matrix multiplication is not possible. The number of columns in Matrix 1 must match the number of rows in Matrix 2. Please try again.");
        }

        double[,] matrix1 = new double[rows1, cols1];
        double[,] matrix2 = new double[rows2, cols2];
```

```
        // Input elements for the first matrix
        Console.WriteLine("Enter the elements of the first matrix:");
        for (int i = 0; i < rows1; i++)
        {
            for (int j = 0; j < cols1; j++)
            {
                matrix1[i, j] = GetValidDouble($"Matrix1[{i}, {j}]: ");
            }
        }

        // Input elements for the second matrix
        Console.WriteLine("Enter the elements of the second matrix:");
        for (int i = 0; i < rows2; i++)
        {
            for (int j = 0; j < cols2; j++)
            {
                matrix2[i, j] = GetValidDouble($"Matrix2[{i}, {j}]: ");
            }
        }

        // Perform matrix multiplication
        double[,] result = MultiplyMatrices(matrix1, matrix2);
        Console.WriteLine("Result of matrix multiplication:");
        PrintMatrix(result);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error: " + ex.Message);
    }
}
```

```
        else if (operation == 4)
        {
            int numNodes;
            // Loop until the user enters a valid number of nodes (up to 10)
            while (true)
            {
                numNodes = GetValidInteger("Enter the number of nodes in the graph (up to 10): ");
                if (numNodes > 0 && numNodes <= 10)  // Ensure number of nodes is within valid range
                    break;  // If valid, break out of the loop
                else
                Console.WriteLine("Error: You can enter up to 10 nodes only. Please enter a valid number.");

            }

            var graph = GenerateRandomGraph(numNodes);
            Console.WriteLine("Generated Graph (Node -> (Neighbor, Weight)):");
            foreach (var node in graph)
            {
                Console.Write($"{node.Key}: ");
                foreach (var neighbor in node.Value)
                {
                    Console.Write($"({neighbor.Item1}, {neighbor.Item2}) ");

                }
```

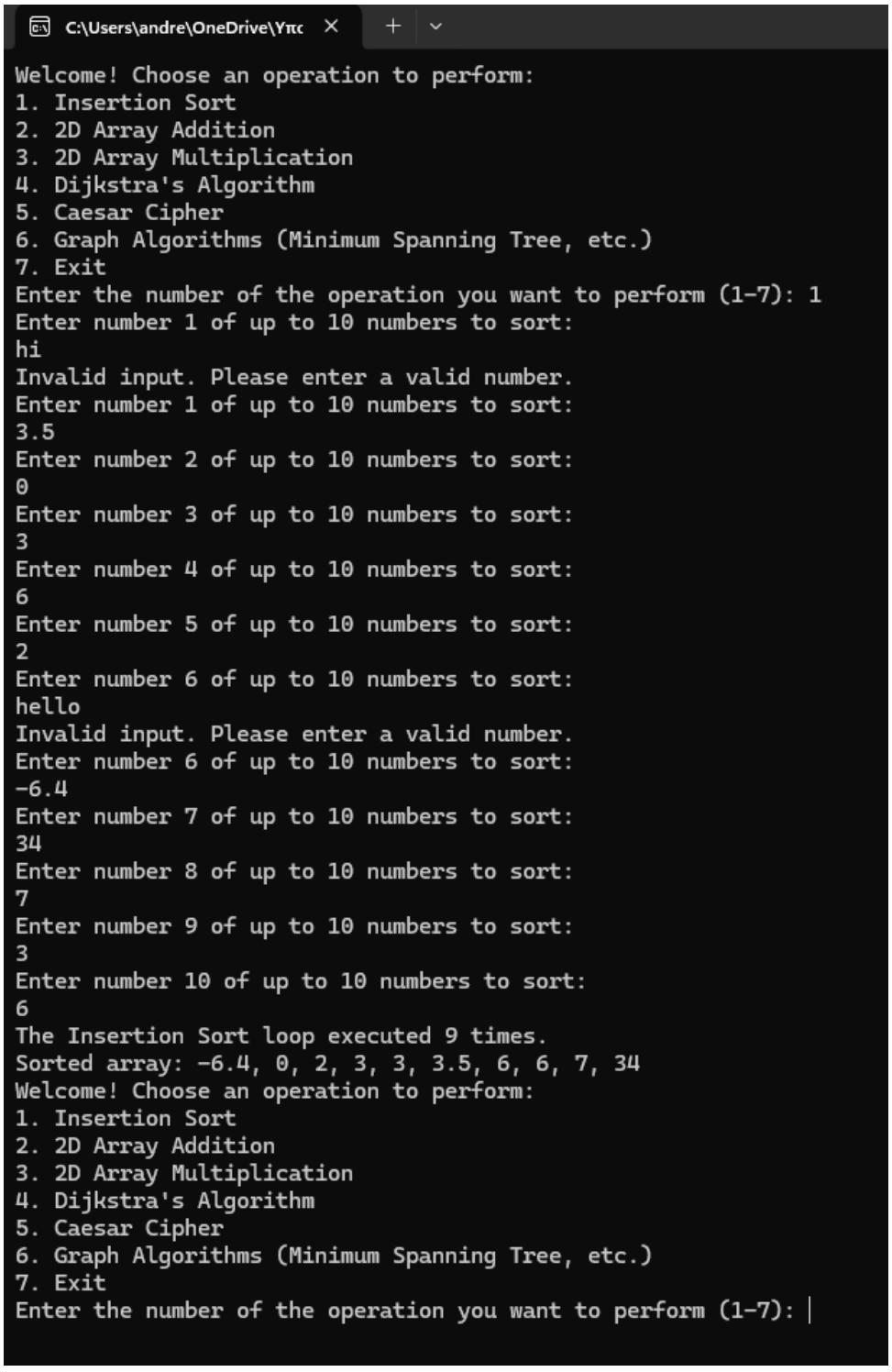
```
            Console.WriteLine();
        }

        int source;
        while (true)
        {
            source = GetValidInteger("Enter the source node for Dijkstra's algorithm (1 to " +
numNodes + "): ");

            // Validate that the source node is within the valid range
            if (source >= 1 && source <= numNodes)
            break;  // If valid, break out of the loop
            else
            Console.WriteLine("Error: Please enter a node between 1 and " + numNodes + ".");

        }

        Dijkstra(graph, source);

       }

    else if (operation == 5)
{
   Console.WriteLine("Caesar Cipher");
   string text;
   while (true)
   {
      Console.Write("Enter text to encrypt (letters only): ");
      text = Console.ReadLine();
```

```
        // Ensure input contains only letters (no spaces or special characters)
        if (!string.IsNullOrEmpty(text) && text.All(c => char.IsLetter(c)))
        {
            break; // Exit loop if valid
        }
        else
        {
            Console.WriteLine("Error: Please enter only letters (no spaces, digits, or special characters).");
        }
    }

    int shift = GetValidInteger("Enter the shift value: ");
    string encryptedText = CaesarCipherEncrypt(text, shift);
    Console.WriteLine($"Encrypted Text: {encryptedText}");

    string decryptedText = CaesarCipherDecrypt(encryptedText, shift);
    Console.WriteLine($"Decrypted Text: {decryptedText}");
}
```

```
        if (operation == 6)
        {
            int numNodes;
            // Loop until the user enters a valid number of nodes (up to 10)
            while (true)
            {
                numNodes = GetValidInteger("Enter the number of nodes in the graph (up to 10): ");
```

```
            if (numNodes > 0 && numNodes <= 10)  // Ensure number of nodes is within valid range
                break;  // If valid, break out of the loop
            else
                Console.WriteLine("Error: You can enter up to 10 nodes only. Please enter a valid number.");
        }

        var graph = GenerateRandomGraph(numNodes);
        Console.WriteLine("Generated Graph (Node -> (Neighbor, Weight)):");
        foreach (var node in graph)
        {
            Console.Write($"{node.Key}: ");
            foreach (var neighbor in node.Value)
            {
                Console.Write($"({neighbor.Item1}, {neighbor.Item2}) ");
            }
            Console.WriteLine();
        }

        MinimumSpanningTree(graph); // Call the new Minimum Spanning Tree function
    }


}

}
```

The first case:



As we can see, the user chose the first case**: "Insertion Sort"**. The program asks the user to enter 1 to up 10 numbers and the user checks all the cases to see whether the program accepts

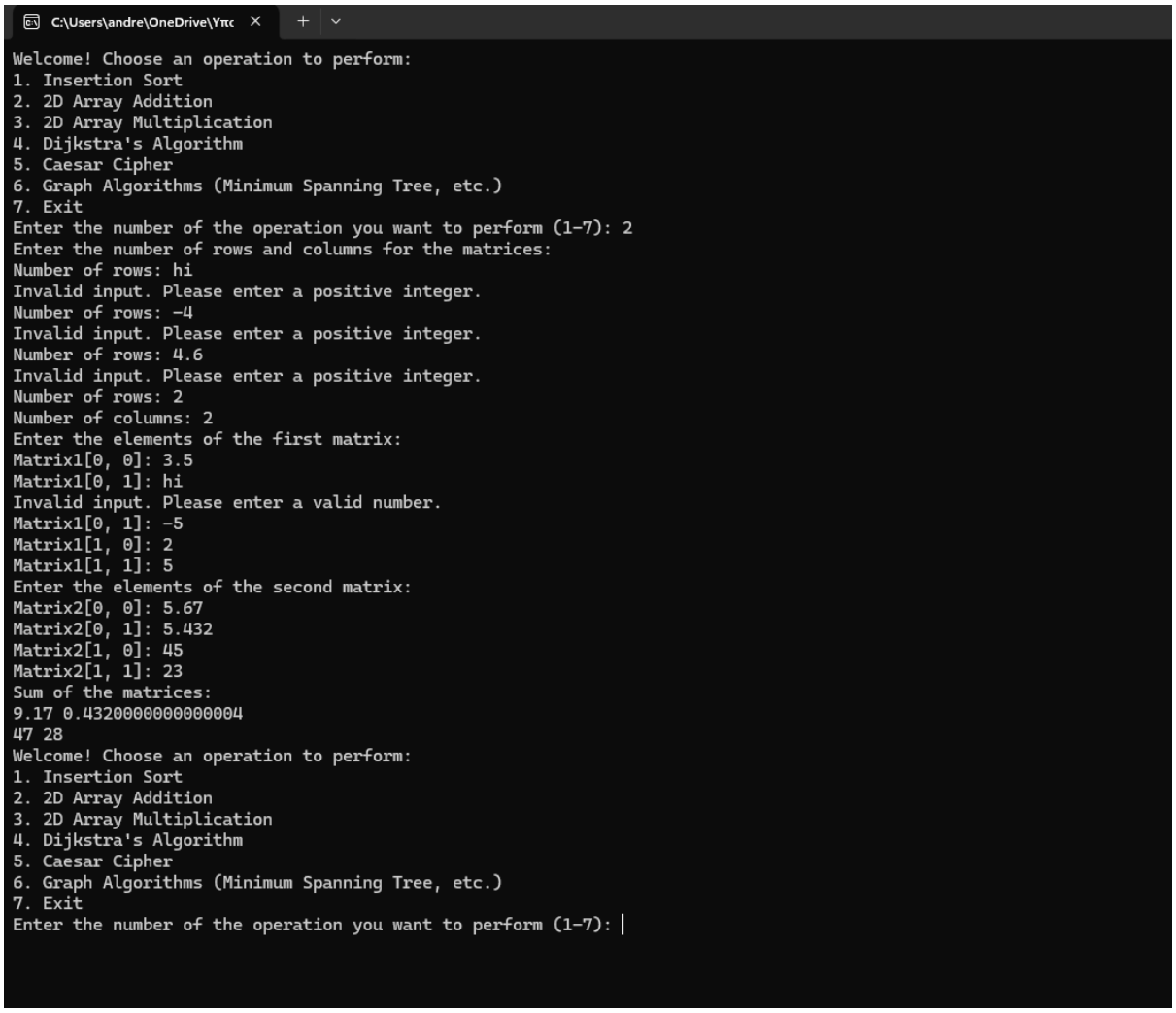
negative numbers, if it accepts letters, and if it gives the same number will the program sort them correctly? As we can see, yes, the program correctly navigates the numbers.

However, the Insertion Sort is the most simple and intuitive sorting algorithm. It works just like sorting cards in your hand-it inserts each card into its proper place concerning the sorted ones.

Steps of the Insertion Sort algorithm:

1. **Initialization**: Start from the second element of the array, considering the first element as already "sorted" by itself.
2. **Insertion of the element**: Take the second element (called the "key") and compare it with the first element:
    - If the key is smaller than the first element, place it before it. Otherwise, leave the first element in place.
3. **Shifting elements**: If the key is smaller than an element before it, shift that element one position to the right to make room for the key.
4. **Repeat**: This process is repeated for the next element in the array until you reach the end.
5. **Final arrangement**: By the end, all the elements are placed in the correct order. [1]

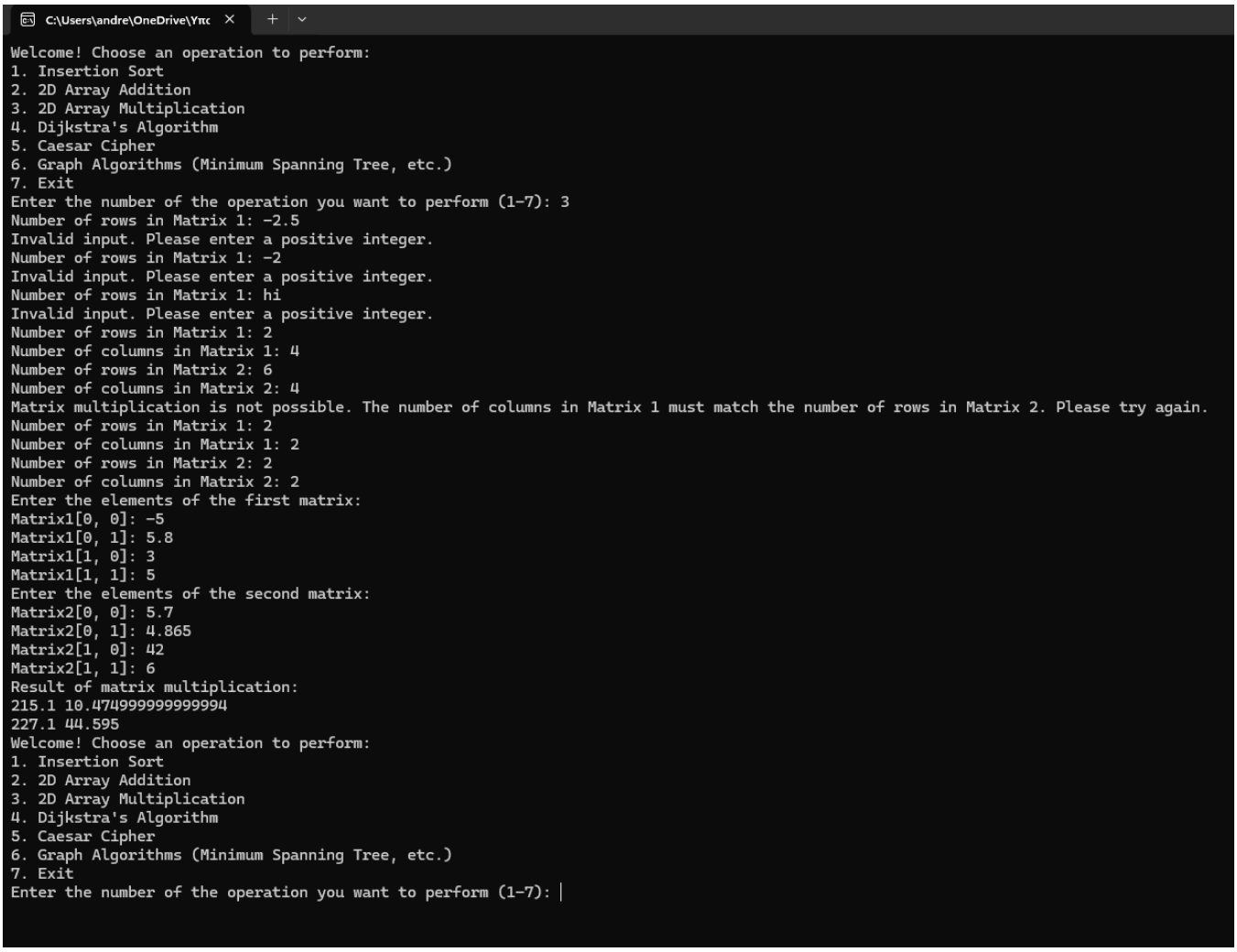
Second case:



The user has selected choice 2 for **"2D Array Addition"**. So, the program further asks to enter the matrices' number of rows and columns. If it asks the user to input positive integer values, it instantly rejects invalid inputs such as just integers or negative values, prompting positive integers.

After giving the valid dimensions, that is, 2 rows and 2 columns, the program asked the user to input the elements of two matrices. The program checked for valid inputs in each element of the matrices, allowing only numbers. After the definition of both matrices, the program calculated their sum and showed the resulting matrix. [2]

Third case:



The user chose **"2D Array Multiplication"** from the menu through option 3. Then, the program asked the user for the number of rows in Matrix 1. For instance, against invalid inputs-like non-integer or negative values **(" -2" or " -1")**, the program rejected such inputs and asked for positive integers.

Once the user provided valid input, for instance, 2 rows for Matrix 1, the program would ask for the number of columns in Matrix 1. Again, a similar type of validation is done here, where negative numbers or non-numeric inputs are not accepted. It then asked for the number of rows and columns for Matrix 2. The same type of validation was conducted for positive integers, including appropriate error messages when an invalid entry was given.

With that said, having derived valid dimensions for both matrices rows × 2 columns each for Matrix 1 and Matrix 2-the program now asked the user to enter the elements of both matrices. Once the matrices were populated with valid values, the program carried out the multiplication of the matrix and showed the results. [3]

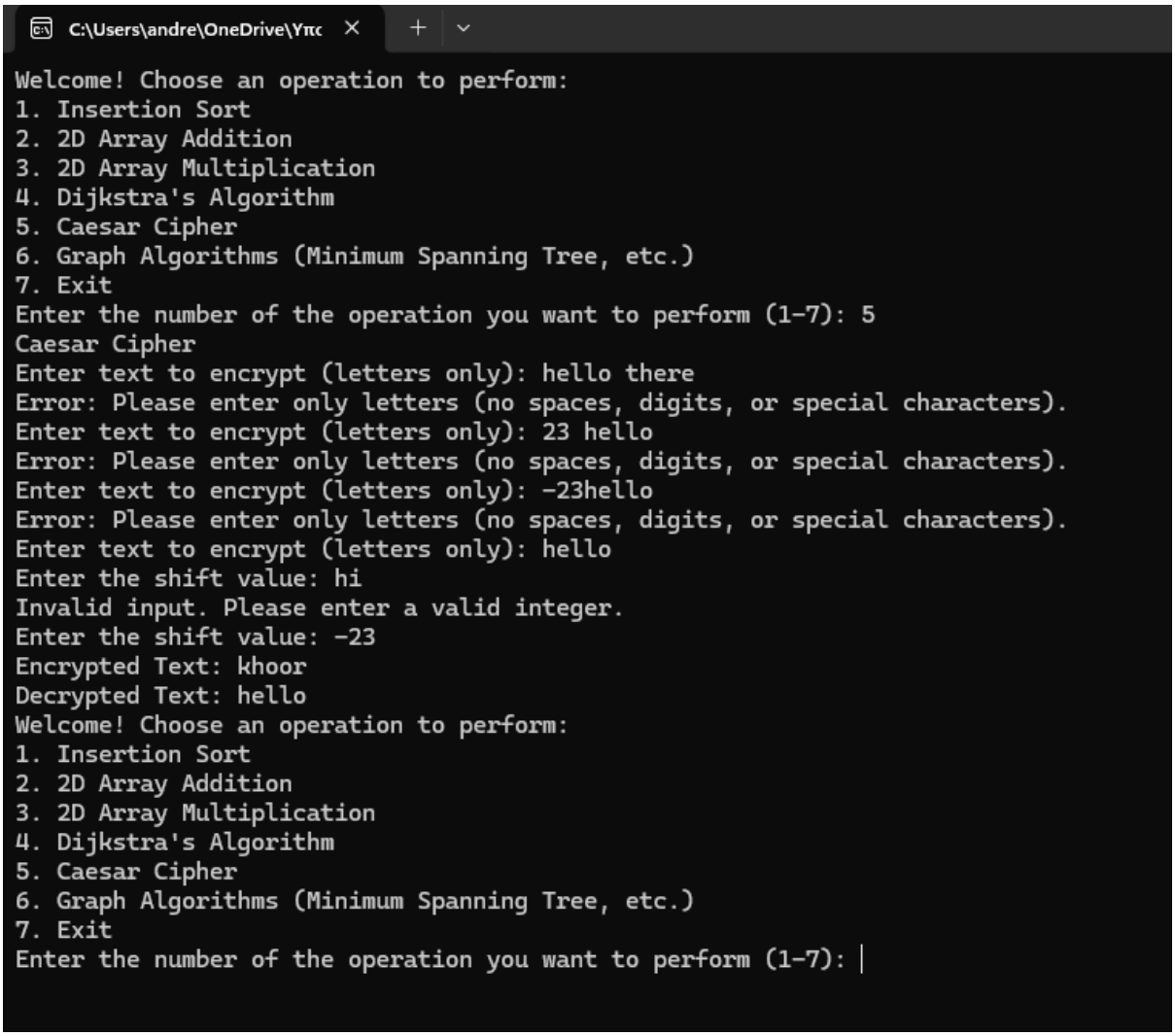
Fourth case:

Here, the user has selected 4 for **"Dijkstra's Algorithm"**. Then, the program asked for the number of nodes for the graph. It asked to enter up to 10 nodes. Given that the user entered incorrect inputs like non-integer values or out-of-range values, such as **"11", "0", "-4", or "4.6"**, the program rejected such inputs and asked for a valid positive integer.

It took several attempts until finally, the user inputted 3, which was valid input; thus, the program accepted it.

After this, the program created the graph with nodes and edges, with weights to indicate the link between nodes. The program asked for the source node to which Dijkstra's algorithm needed to be applied. Providing inputs in invalid formats, that is, non-integer and out of the specified range, like **"-6", "2.4",** or **"hello"**, resulted in their rejection by the program in the form of asking for the correct valid number of a node. Lastly, the user inputted 3 as the source node and the result came out. [4]

Fifth case:

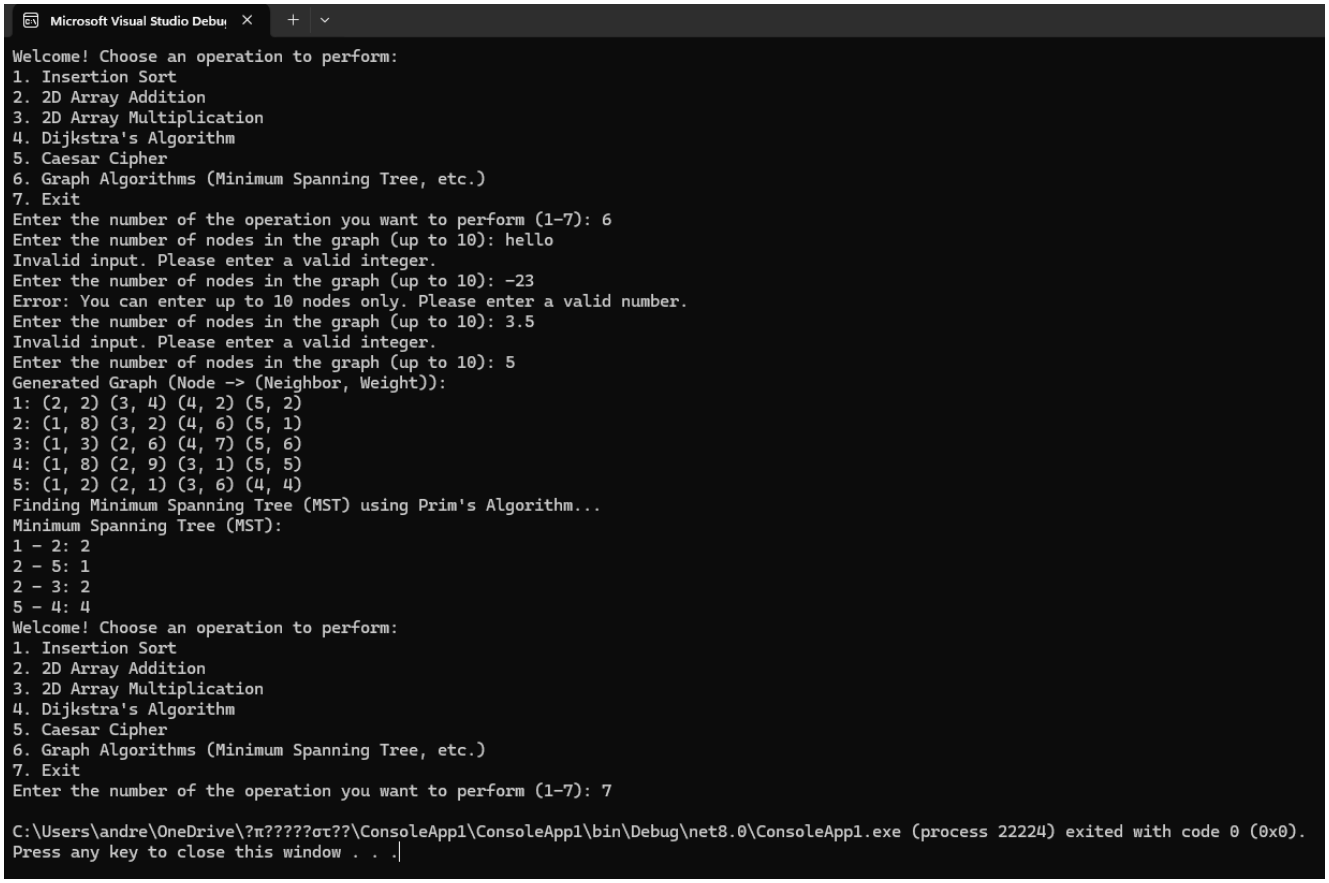


Here, the user has selected option 5, **"Caesar Cipher"**. The program is now going to ask for text to encrypt, precisely asking for only letters and not spaces, digits, or special characters. Given the inputs **"hello there"**, **"23 hello"**, and **"-23hello"** in turn, it will reject all of them because they are invalid inputs due to containing a space, numbers, and a negative number, respectively, and asking again for just letters.

Once the user input **"hello"**, a valid input, the program moves to request the shift value for the Caesar Cipher. Once **"hi"** was entered as an integer program rejected it, asking for a valid integer. Then with -23 being entered, the program accepted that as the valid shift value.

It in turn encrypted **"hello"** with the shift value of -23 and got **"khoor"** as the encrypted text. It also decrypted **"khoor"** back into **"hello"**. Once encryption and decryption were done, it came out to the main menu to have the user select another operation. [5]

Sixth and Seventh case:



The user chose option 6: **"Graph Algorithms Minimum Spanning Tree, etc."**. Prompt the user to enter the number of nodes (up to 10). In the absence of valid input that is non-integer and out of the range- for instance, **"hello", "-23",** or **"3.5"**, the program would not accept such input but insist on an integer input.

It took an umpteen number of tries before the user finally entered 5, a valid input that the program accepted. Then the program generated a graph with 5 nodes and displayed the graph structure: nodes and their links with weights.

After this, the program executed Prim's Algorithm to find the Minimum Spanning Tree and showed edges and the weights thus obtained. Finally, the user chose option 7 to quit the program, and the program did indeed shut down, as indicated by the exit code message. [6]

Sources

[1] I. J. o. C. Applications, «Enhanced Insertion Sort Algorithm,» 2013.

[2] W.-R. S. Shen-Fu Hsiao, «A new hardware-efficient algorithm and architecture for computation of 2-D DCTs on a linear array,» 2001.

[3] G. BLELLOCH, «Recursive array layouts and fast parallel matrix multiplication,» 1999.

[4] A. Javaid, «Understanding Dijkstra's Algorithm,» 2014.

[5] S. N. Gowda, «Innovative enhancement of the Caesar cipher algorithm for cryptography,» 2016.

[6] C. U. Press, «Graph Algorithms,» 2011.