DIGITAL ASSIGNMENT 5 [GRAPH ALGORITHMS]

CSE2012–DESIGN AND ANALYSIS OF ALGORITHMS (L25-26)[MRS GAYATHRI P]

MARCH 29, 2023

ANIRUDH VADERA
20BCE2940

ANIRUDH VADERA

DIGITAL ASSIGNMENT 5 [GRAPH ALGORITHMS]

QUESTION 1:

Implement KMP algorithm to check whether given pattern P is plagiarized in given Text T.

PSEUDOCODE:

The main logic behind the Knuth-Morris-Pratt (KMP) algorithm is to avoid re-matching previously matched characters of the pattern with the text by using a pre-computed array called the Longest Proper Prefix Suffix (LPS) array. The LPS array stores the length of the longest proper prefix that is also a suffix for each pattern substring.

The KMP algorithm works as follows:

- Preprocess the pattern string P to compute the LPS array.
- Initialize two pointers i and j to 0, representing the current positions in the text and pattern respectively.
- While i is less than the length of the text T, do the following:

    a. If the jth character of P matches the ith character of T, increment both i and j.

    b. If j equals the length of P, a match is found at index i-j in the text T. Reset j to the value of the LPS[j-1], which represents the length of the longest proper prefix that is also a suffix for the pattern substring P[0..j-1].

    c. If the jth character of P does not match the ith character of T, decrement j to the value of LPS[j-1]. If j equals 0, increment i and continue the loop.

- Repeat step 3 until the end of the text T is reached.

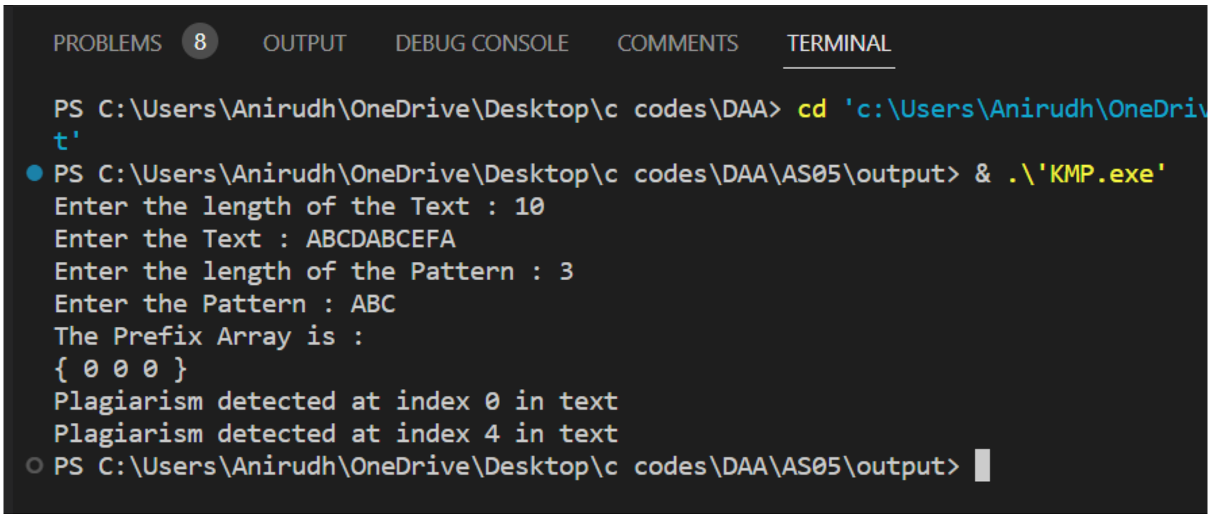
By using the LPS array, the KMP algorithm avoids re-matching previously matched characters in the pattern, resulting in a time complexity of O(m+n), where m and n are the lengths of the pattern and text respectively.

SOURCE CODE:

```
#include <bits/stdc++.h>
using namespace std;

void computeprefixArray(char *pattern, int M, int *prefixArray)
{
    int len = 0;
    prefixArray[0] = 0;
    int i = 1;
    while (i < M)
    {
        if (pattern[i] == pattern[len])
        {
            len++;
            prefixArray[i] = len;
            i++;
        }
        else
        {
            if (len != 0)
            {
                len = prefixArray[len - 1];
            }
            else
            {
                prefixArray[i] = 0;
                i++;
            }
        }
    }
}

void KMP(char *pattern, int M, char *text, int N)
{

    int *prefixArray = (int *)malloc(M * sizeof(int));

    computeprefixArray(pattern, M, prefixArray);

    cout << "The Prefix Array is : " << endl;
    cout << "{ ";
    for (int i = 0; i < M; i++)
    {
        cout << prefixArray[i] << " ";
    }
    cout << "}" << endl;
    int i = 0;
    int j = 0;
    while (i < N)
    {
        if (pattern[j] == text[i])
        {
            j++;
            i++;
        }

        if (j == M)
        {
            printf("Plagiarism detected at index %d in text\n", i - j);
            j = prefixArray[j - 1];
        }

        else if (i < N && pattern[j] != text[i])
        {
            if (j != 0)
            {
                j = prefixArray[j - 1];
            }
            else
            {
                i++;
            }
        }
    }
}

int main()
{
    int M, N;
    cout << "Enter the length of the Text : ";
    cin >> N;
    char *text = (char *)malloc(N * sizeof(char));
    cout << "Enter the Text : ";
    cin >> text;
    cout << "Enter the length of the Pattern : ";
    cin >> M;
    char *pattern = (char *)malloc(M * sizeof(char));
    cout << "Enter the Pattern : ";
    cin >> pattern;
    KMP(pattern, M, text, N);
    return 0;
}
```

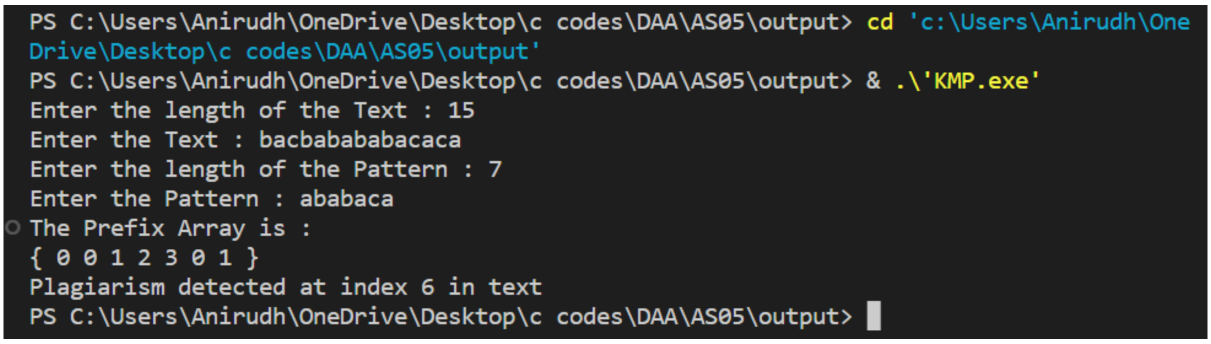
OUTPUT SCREENSHOT:

Test Case 1:



```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA> cd 'c:\Users\Anirudh\OneDriv
t'
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> & .\'KMP.exe'
Enter the length of the Text : 10
Enter the Text : ABCDABCEFA
Enter the length of the Pattern : 3
Enter the Pattern : ABC
The Prefix Array is :
{ 0 0 0 }
Plagiarism detected at index 0 in text
Plagiarism detected at index 4 in text
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
```

Test Case 2:



```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> cd 'c:\Users\Anirudh\One
Drive\Desktop\c codes\DAA\AS05\output'
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> & .\'KMP.exe'
Enter the length of the Text : 15
Enter the Text : bacbababababacaca
Enter the length of the Pattern : 7
Enter the Pattern : ababaca
The Prefix Array is :
{ 0 0 1 2 3 0 1 }
Plagiarism detected at index 6 in text
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
```

QUESTION 2:

Consider set of points Q as input and find the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior using Graham's Scan algorithm.

PSEUDOCODE:

The main logic behind the Graham scan algorithm for finding the Convex Hull of a set of points is as follows:

- Choose the point with the lowest y-coordinate. If there are multiple points with the same y-coordinate, choose the one with the lowest x-coordinate.
- Sort the remaining points by their polar angles with respect to the chosen point. To do this, first compute the polar angle of each point relative to the chosen point, and then sort the points in ascending order of their polar angles. In case of a tie, choose the point closest to the chosen point.
- Create an empty stack and push the first two points onto the stack.
- For each remaining point, repeat the following steps:

  a. While the current point forms a "right turn" with the top two points on the stack, pop the top point off the stack.

  b. Push the current point onto the stack.

- The stack now contains the points that form the convex hull in counterclockwise order, starting from the point with the lowest y-coordinate.
- Return the stack as the result.

The "right turn" test is performed using the cross product of two vectors formed by the three points: If the cross product is positive, the three points form a "left turn", indicating that the current point is on the boundary of the convex hull. If the cross product is negative, the three points form a "right turn", indicating that the top point on the stack is not part of the convex hull and should be removed. If the cross product is zero, the three points are collinear and the top point on the stack can be replaced with the current point.

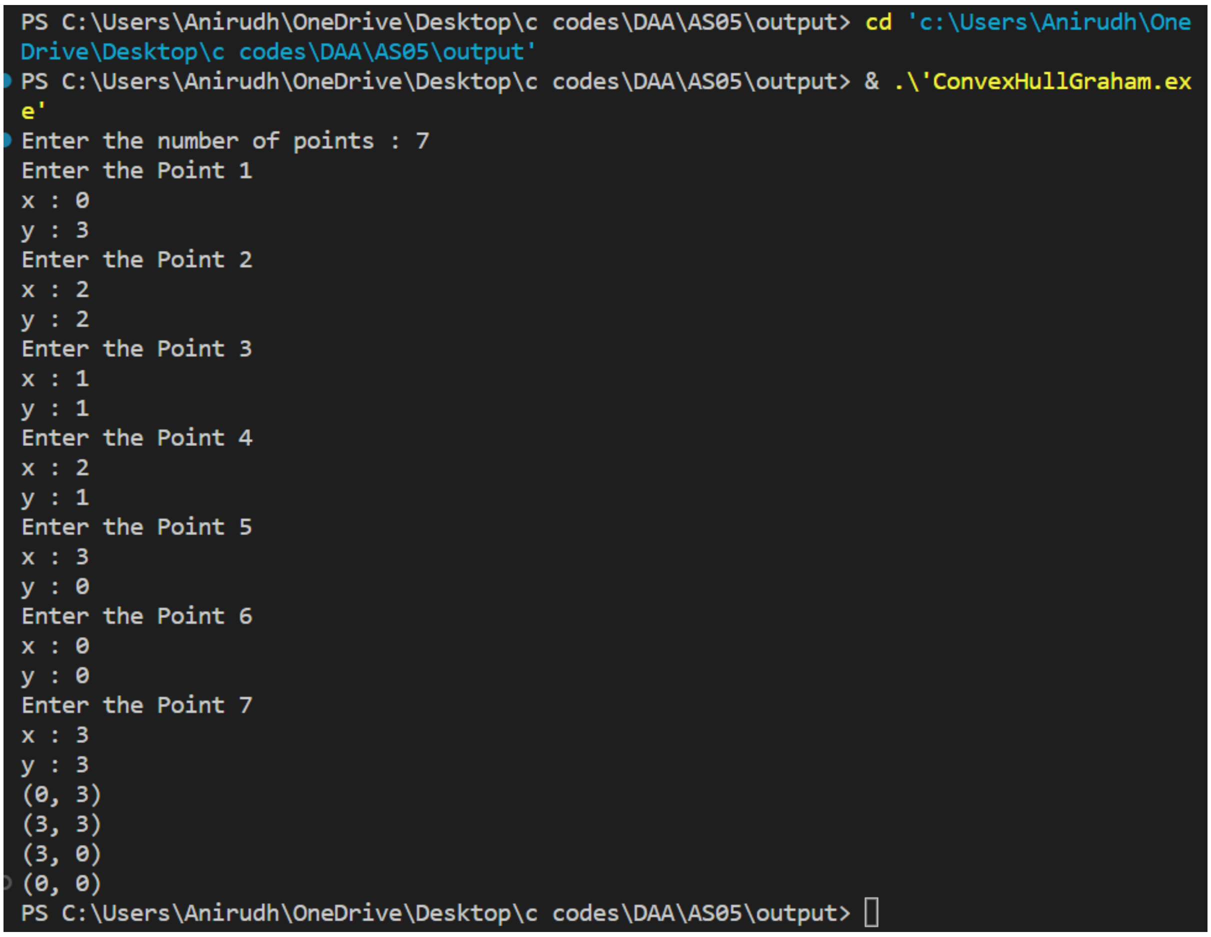
The Graham scan algorithm has a time complexity of O(n log n), where n is the number of points in the input set, due to the sorting step.

SOURCE CODE:

```
#include <iostream>
#include <stack>
#include <stdlib.h>
using namespace std;

// A simple Point
struct Point
{
    int x, y;
};

// The first point
Point p0;

// Top in Stack
Point nextToTop(stack<Point> &S)
{
    Point p = S.top();
    S.pop();
    Point res = S.top();
    S.push(p);
    return res;
}

void swap(Point &p1, Point &p2)
{
    Point temp = p1;
    p1 = p2;
    p2 = temp;
}

// squared dist btw 2 points
int distSq(Point p1, Point p2)
{
    return (p1.x - p2.x) * (p1.x - p2.x) +
           (p1.y - p2.y) * (p1.y - p2.y);
}

// 0 --> p, q and r are collinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);
    if (val == 0)
        return 0;             // collinear
    return (val > 0) ? 1 : 2; // clock or counterclock wise
}

// Helps in Sorting
int compare(const void *vp1, const void *vp2)
{
    Point *p1 = (Point *)vp1;
    Point *p2 = (Point *)vp2;

    // Find orientation
    int o = orientation(p0, *p1, *p2);
    if (o == 0)
        return (distSq(p0, *p2) >= distSq(p0, *p1)) ? -1 : 1;

    return (o == 2) ? -1 : 1;
}

// Prints convex hull of a set of n points.
void convexHull(Point *points, int n)
{
    // Find the bottommost point
    int ymin = points[0].y, min = 0;
    for (int i = 1; i < n; i++)
    {
        int y = points[i].y;

        // Pick the bottom-most or choose the left
        // most point in case of tie
        if ((y < ymin) || (ymin == y &&
                           points[i].x < points[min].x))
            ymin = points[i].y, min = i;
    }

    // Place the bottom-most point at first position
    swap(points[0], points[min]);

    // Sort n-1 points with respect to the first point.
    p0 = points[0];
    qsort(&points[1], n - 1, sizeof(Point), compare);

    // If two or more points make same angle with p0,
    // Remove all but the one that is farthest from p0
    int m = 1; // Size of modified array
    for (int i = 1; i < n; i++)
    {
        // Keep removing i while angle of i and i+1 is same
        // with respect to p0
        while (i < n - 1 && orientation(p0, points[i], points[i + 1]) == 0)
            i++;

        points[m] = points[i];
        m++;
    }

    // Atleast 3 points are required
    if (m < 3)
        return;

    // Stack
    stack<Point> S;
    S.push(points[0]);
    S.push(points[1]);
    S.push(points[2]);

    // Algo
    for (int i = 3; i < m; i++)
    {
        // Keep removing top while the angle formed by
        // points next-to-top, top, and points[i] makes
        // a non-left turn
        while (S.size() > 1 && orientation(nextToTop(S), S.top(), points[i])
!= 2)
            S.pop();
        S.push(points[i]);
    }

    // Now stack has the output points, print contents of stack
    while (!S.empty())
    {
        Point p = S.top();
        cout << "(" << p.x << ", " << p.y << ")" << endl;
        S.pop();
    }
}

// Driver program to test above functions
int main()
{
    int n;
    cout << "Enter the number of points : ";
    cin >> n;
    Point *points = (Point *)malloc(n * sizeof(Point));
    for (int i = 0; i < n; i++)
    {
        cout << "Enter the Point " << (i + 1) << endl;
        cout << "x : ";
        cin >> points[i].x;
        cout << "y : ";
        cin >> points[i].y;
    }
    convexHull(points, n);
    return 0;
}
```

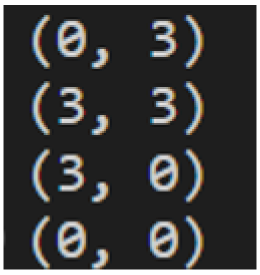
OUTPUT SCREENSHOT:

Test Case 1: {{0, 3}, {1, 1}, {2, 2}, {4, 4}, {0, 0}, {1, 2}, {3, 1}, {3, 3}};



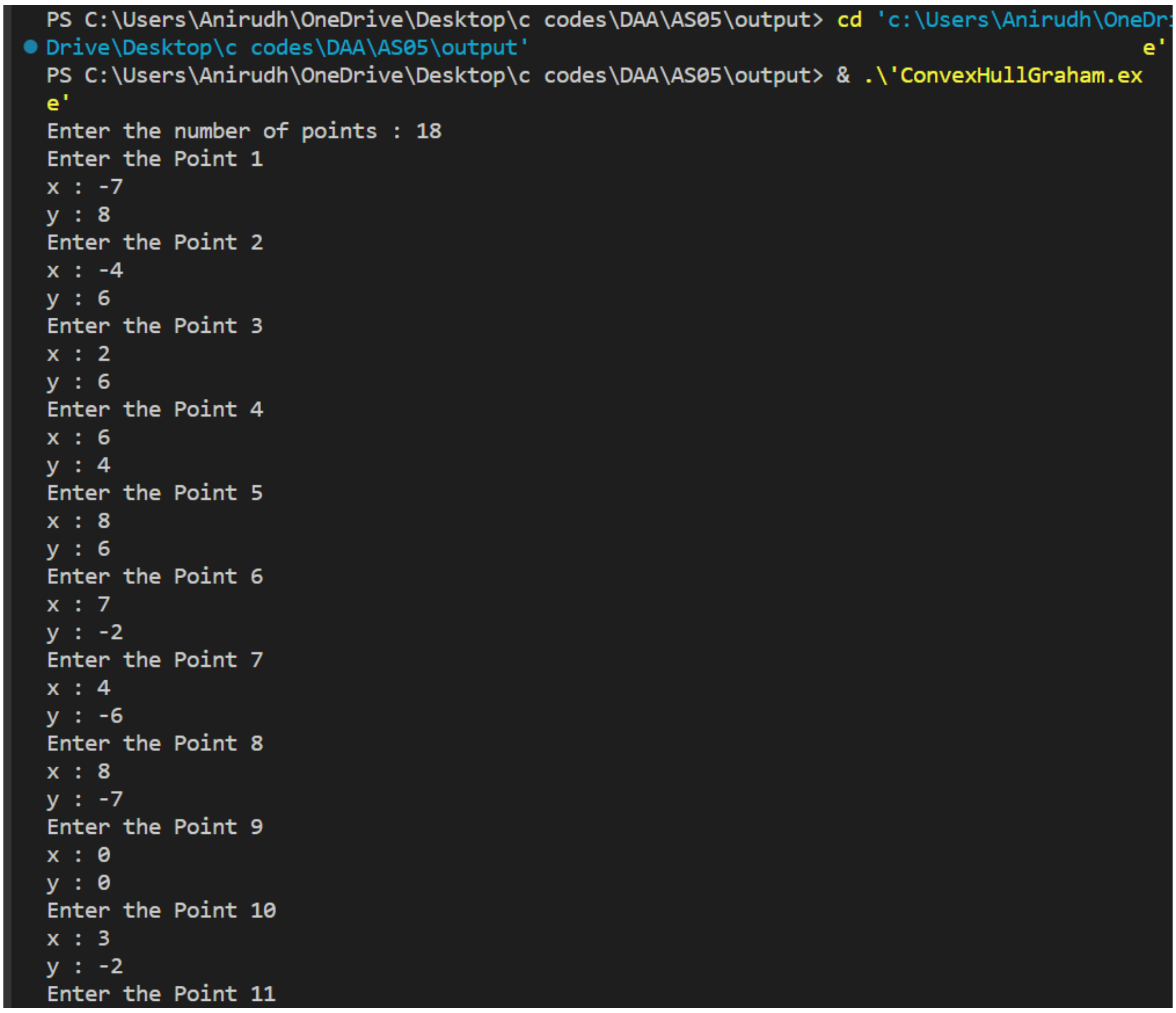
```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> cd 'c:\Users\Anirudh\One
Drive\Desktop\c codes\DAA\AS05\output'
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> & .\'ConvexHullGraham.ex
e'
Enter the number of points : 7
Enter the Point 1
x : 0
y : 3
Enter the Point 2
x : 2
y : 2
Enter the Point 3
x : 1
y : 1
Enter the Point 4
x : 2
y : 1
Enter the Point 5
x : 3
y : 0
Enter the Point 6
x : 0
y : 0
Enter the Point 7
x : 3
y : 3
(0, 3)
(3, 3)
(3, 0)
(0, 0)
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> 
```

Solution:

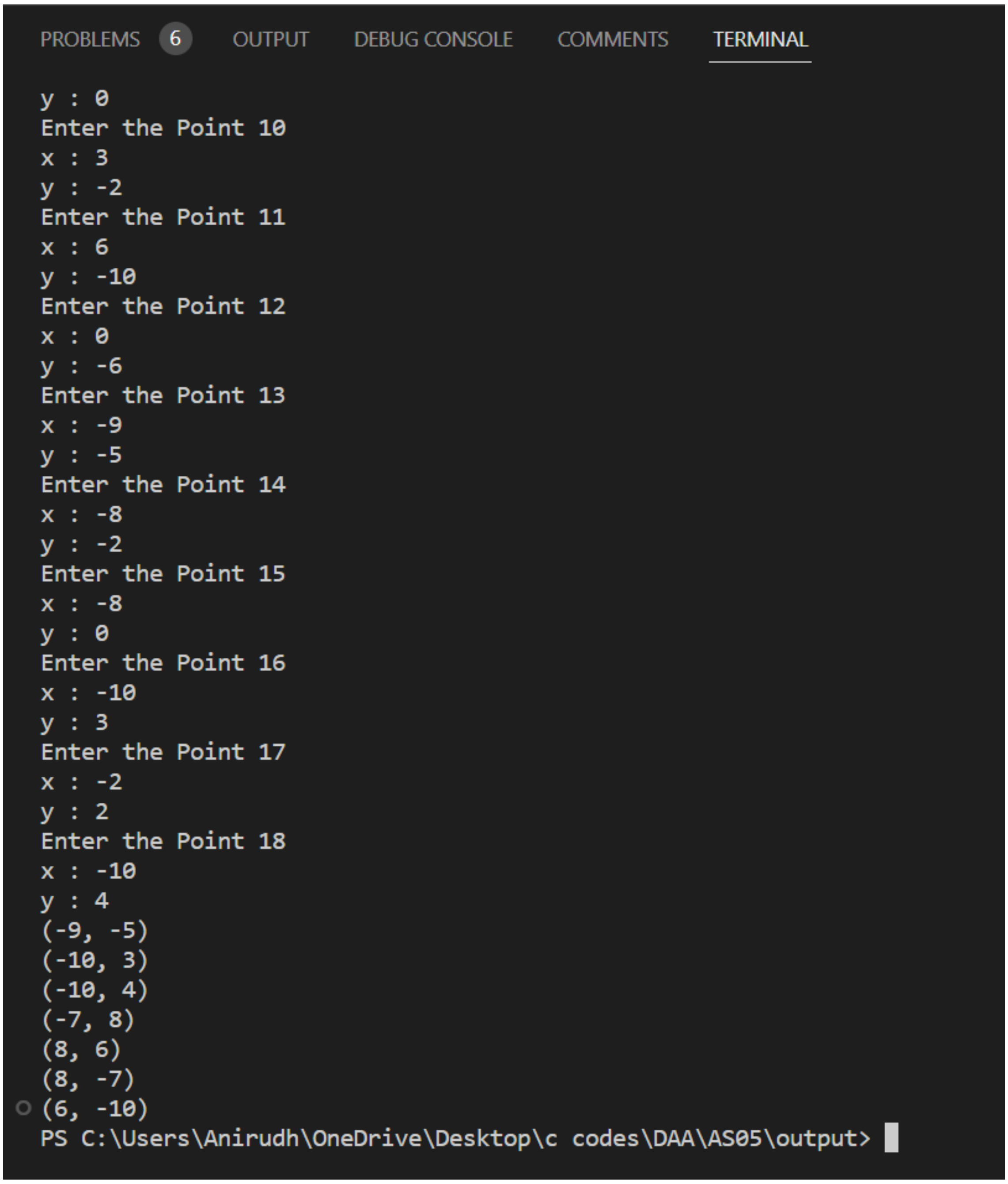


```
(0, 3)
(3, 3)
(3, 0)
(0, 0)
```

Test Case 2: {(-7,8), (-4,6), (2,6), (6,4), (8,6), (7,-2), (4,-6), (8,-7),(0,0), (3,-2),(6,-10),(0,-6),(-9,-5),(-8,-2),(-8,0),(-10,3),(-2,2),(-10,4)}

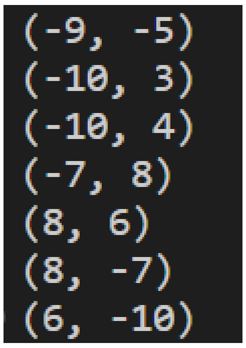


```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> cd 'c:\Users\Anirudh\OneDr
Drive\Desktop\c codes\DAA\AS05\output'                                                e'
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> & .\'ConvexHullGraham.ex
e'
Enter the number of points : 18
Enter the Point 1
x : -7
y : 8
Enter the Point 2
x : -4
y : 6
Enter the Point 3
x : 2
y : 6
Enter the Point 4
x : 6
y : 4
Enter the Point 5
x : 8
y : 6
Enter the Point 6
x : 7
y : -2
Enter the Point 7
x : 4
y : -6
Enter the Point 8
x : 8
y : -7
Enter the Point 9
x : 0
y : 0
Enter the Point 10
x : 3
y : -2
Enter the Point 11
```



```
y : 0
Enter the Point 10
x : 3
y : -2
Enter the Point 11
x : 6
y : -10
Enter the Point 12
x : 0
y : -6
Enter the Point 13
x : -9
y : -5
Enter the Point 14
x : -8
y : -2
Enter the Point 15
x : -8
y : 0
Enter the Point 16
x : -10
y : 3
Enter the Point 17
x : -2
y : 2
Enter the Point 18
x : -10
y : 4
(-9, -5)
(-10, 3)
(-10, 4)
(-7, 8)
(8, 6)
(8, -7)
(6, -10)
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
```

Solution:



```
(-9, -5)
(-10, 3)
(-10, 4)
(-7, 8)
(8, 6)
(8, -7)
(6, -10)
```

QUESTION 3:

Consider set of points Q as input and find the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior using Jarvis March algorithm.

PSEUDOCODE:

The main logic behind the Convex Hull using Jarvis March algorithm is to find the convex hull of a set of points in a 2D plane by starting from the leftmost point and traversing in a counterclockwise direction.

The Jarvis March algorithm works as follows:

- Find the leftmost point in the set of points and add it to the convex hull.
- Initialize the current point as the leftmost point and the endpoint as a point that is not on the convex hull.
- While the endpoint is not equal to the leftmost point, do the following:

  a. For each point in the set of points, calculate the cross product of the vector formed by the current point and the point being considered, and the vector formed by the current point and the endpoint. If the cross product is positive or the point being considered is the leftmost point, update the endpoint to be the point being considered.

  b. Add the endpoint to the convex hull.

  c. Update the current point to be the endpoint.

  d. Repeat step 3 until the endpoint is equal to the leftmost point.

- Return the convex hull.

The Jarvis March algorithm selects each point on the convex hull one at a time by selecting the point that has the largest counterclockwise angle with respect to the previous point. By selecting the leftmost point as the starting point, the algorithm guarantees that the first point added to the convex hull is part of the hull. The algorithm continues to select points that are part of the hull until it returns to the starting point. The resulting set of points forms the convex hull of the set of input points.

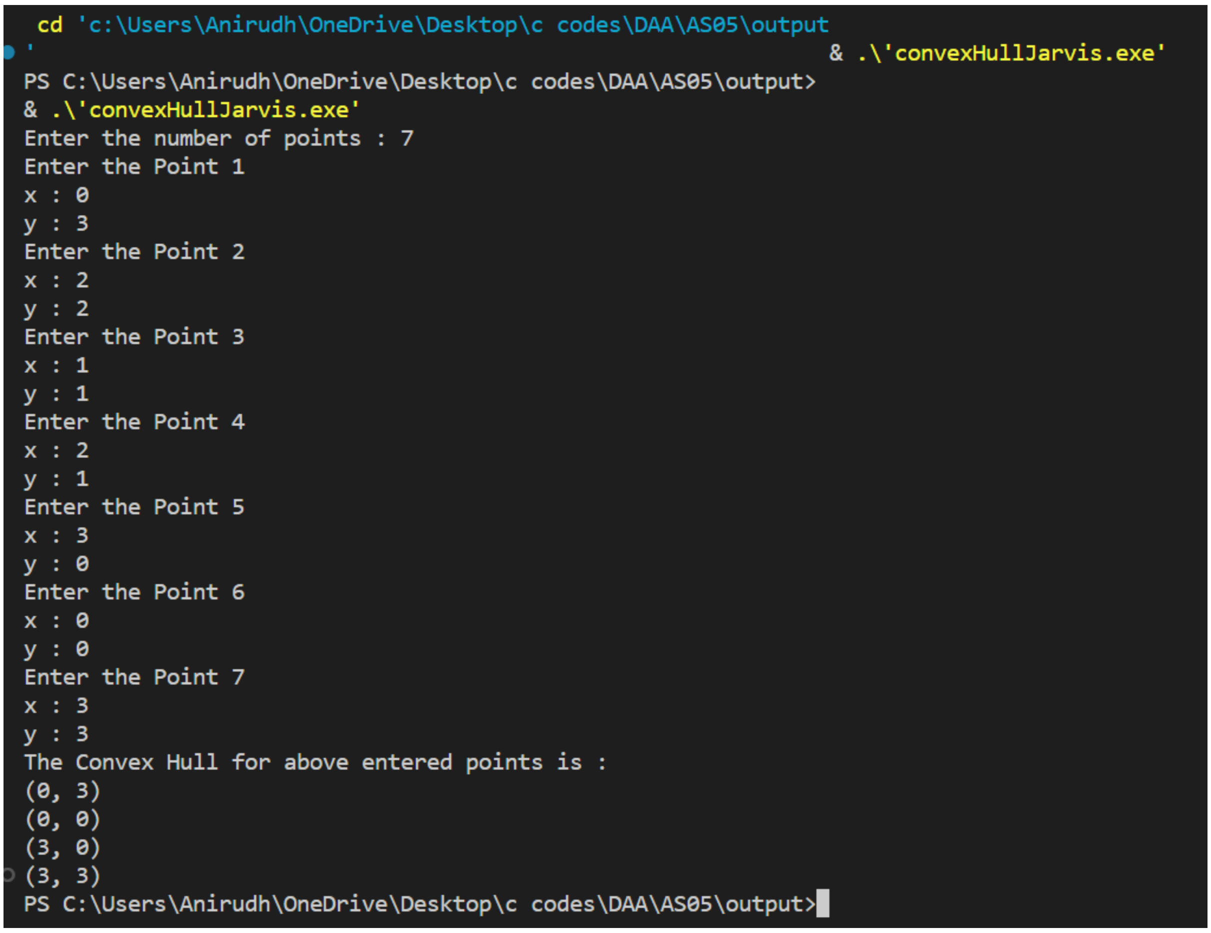
The time complexity of Jarvis March algorithm is O(nh), where n is the number of input points and h is the number of points on the convex hull.

SOURCE CODE:

```
#include <bits/stdc++.h>
using namespace std;

// A simple Point
struct Point
{
    int x, y;
};

// 0 --> p, q and r are collinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0)
        return 0;             // collinear
    return (val > 0) ? 1 : 2; // clock or counterclock wise
}

// Convex hull of a set of n points.
void convexHull(Point *points, int n)
{
    // At least 3 points
    if (n < 3)
        return;

    // Initialize Result
    vector<Point> hull;

    // Find the leftmost point
    int l = 0;
    for (int i = 1; i < n; i++)
        if (points[i].x < points[l].x)
            l = i;

    // Start from leftmost point, keep moving counterclockwise
    // until reach the start point again.
    int p = l, q;
    do
    {
        // Add current point to result
        hull.push_back(points[p]);
        q = (p + 1) % n;
        for (int i = 0; i < n; i++)
        {
            // If i is more counterclockwise than current q, then
            // update q
            if (orientation(points[p], points[i], points[q]) == 2)
                q = i;
        }

        // Now q is the most counterclockwise with respect to p
        // Set p as q for next iteration, so that q is added to
        // result 'hull'
        p = q;

    } while (p != l);

    // Print Result
    cout << "The Convex Hull for above entered points is : " << endl;
    for (int i = 0; i < hull.size(); i++)
        cout << "(" << hull[i].x << ", " << hull[i].y << ")\n";
}

// Driver program to test above functions
int main()
{
    int n;
    cout << "Enter the number of points : ";
    cin >> n;
    Point *points = (Point *)malloc(n * sizeof(Point));
    for (int i = 0; i < n; i++)
    {
        cout << "Enter the Point " << (i + 1) << endl;
        cout << "x : ";
        cin >> points[i].x;
        cout << "y : ";
        cin >> points[i].y;
    }
    convexHull(points, n);
    return 0;
}
```

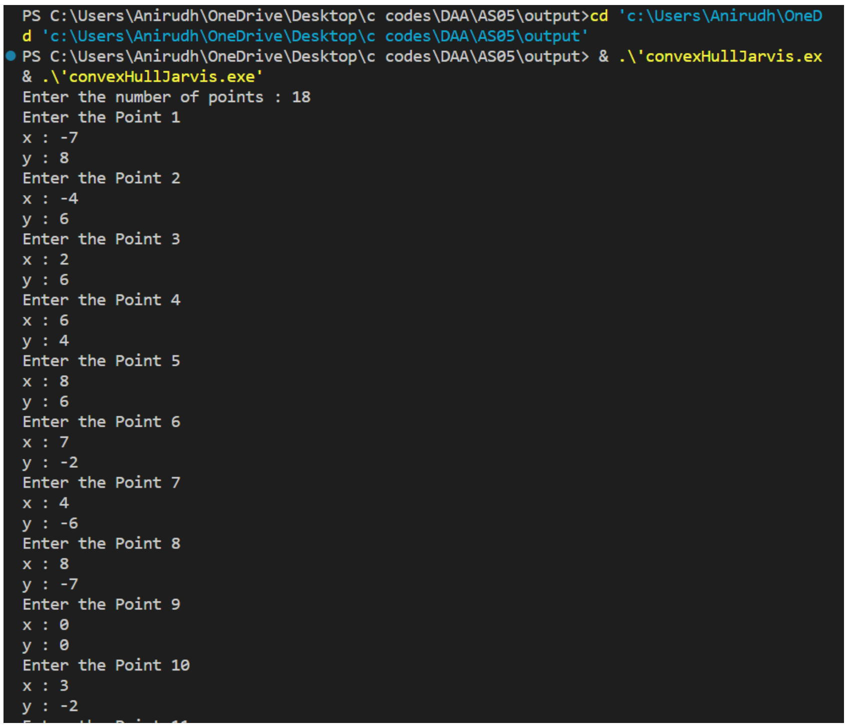
OUTPUT SCREENSHOT:

Test Case 1: {{0, 3}, {1, 1}, {2, 2}, {4, 4}, {0, 0}, {1, 2}, {3, 1}, {3, 3}};

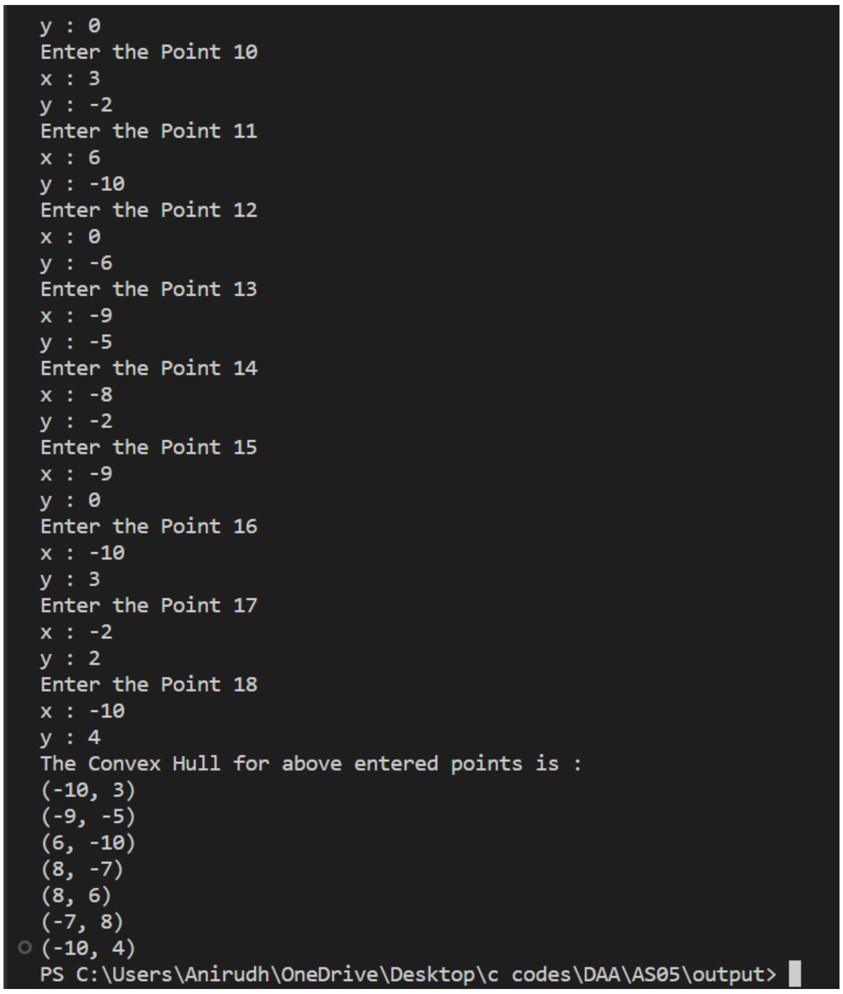


```
 cd 'c:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output
'                                                             & .\'convexHullJarvis.exe'
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
& .\'convexHullJarvis.exe'
Enter the number of points : 7
Enter the Point 1
x : 0
y : 3
Enter the Point 2
x : 2
y : 2
Enter the Point 3
x : 1
y : 1
Enter the Point 4
x : 2
y : 1
Enter the Point 5
x : 3
y : 0
Enter the Point 6
x : 0
y : 0
Enter the Point 7
x : 3
y : 3
The Convex Hull for above entered points is :
(0, 3)
(0, 0)
(3, 0)
(3, 3)
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
```

Test Case 2: {(-7,8), (-4,6), (2,6), (6,4), (8,6), (7,-2), (4,-6), (8,-7),(0,0), (3,-2),(6,-10),(0,-6),(-9,-5),(-8,-2),(-8,0),(-10,3),(-2,2),(-10,4)}



```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>cd 'c:\Users\Anirudh\OneD
d 'c:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output'
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> & .\'convexHullJarvis.ex
& .\'convexHullJarvis.exe'
Enter the number of points : 18
Enter the Point 1
x : -7
y : 8
Enter the Point 2
x : -4
y : 6
Enter the Point 3
x : 2
y : 6
Enter the Point 4
x : 6
y : 4
Enter the Point 5
x : 8
y : 6
Enter the Point 6
x : 7
y : -2
Enter the Point 7
x : 4
y : -6
Enter the Point 8
x : 8
y : -7
Enter the Point 9
x : 0
y : 0
Enter the Point 10
x : 3
y : -2
```



```
y : 0
Enter the Point 10
x : 3
y : -2
Enter the Point 11
x : 6
y : -10
Enter the Point 12
x : 0
y : -6
Enter the Point 13
x : -9
y : -5
Enter the Point 14
x : -8
y : -2
Enter the Point 15
x : -9
y : 0
Enter the Point 16
x : -10
y : 3
Enter the Point 17
x : -2
y : 2
Enter the Point 18
x : -10
y : 4
The Convex Hull for above entered points is :
(-10, 3)
(-9, -5)
(6, -10)
(8, -7)
(8, 6)
(-7, 8)
(-10, 4)
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
```

QUESTION 4:

Implement Floyd-Warshall algorithm and find the shortest path between each pair of vertices in the given graph G.

PSEUDOCODE:

The Floyd-Warshall algorithm is a dynamic programming algorithm that computes the shortest distance between every pair of vertices in a weighted directed graph. The main logic behind the algorithm is as follows:

- Create a 2D distance matrix D of size VxV, where V is the number of vertices in the graph.
- Initialize the distance matrix D such that D[i][j] = the weight of the edge from vertex i to vertex j if the edge exists, and D[i][j] = infinity otherwise. Also, initialize the diagonal elements of the matrix such that D[i][i] = 0 for all i.
- For each vertex k in the graph, do the following:

a. For each pair of vertices i and j in the graph, check if the shortest path from i to j passes through vertex k. If it does, update the distance matrix D by setting D[i][j] = min(D[i][j], D[i][k] + D[k][j]).

- Repeat step 3 for all vertices in the graph.

After all iterations are complete, the distance matrix D will contain the shortest distance between every pair of vertices in the graph.

The main idea behind the algorithm is to build up the solution by considering subproblems of increasing size. At each iteration, we update the distance matrix D to include the shortest path that passes through the current vertex k. By iterating through all vertices in the graph, we ensure that all possible shortest paths are considered and the resulting distance matrix D contains the shortest distance between every pair of vertices.

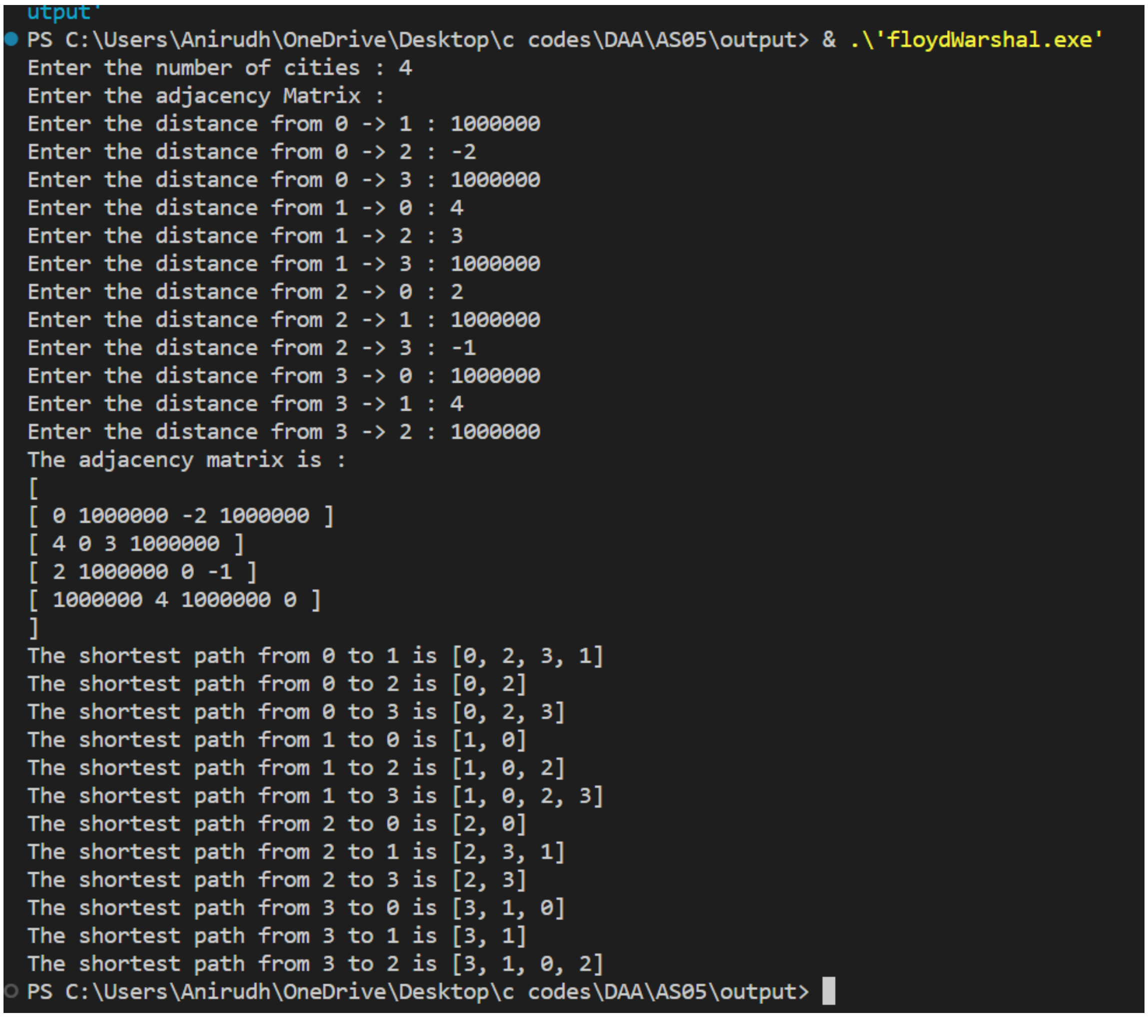
The time complexity of the Floyd-Warshall algorithm is O(V^3), where V is the number of vertices in the graph. This makes it suitable for small to medium sized graphs, but not practical for very large graphs.

SOURCE CODE:

```cpp
#include <iostream>
#include <vector>
#include <climits>
#include <iomanip>
using namespace std;

// Recursive function to print path of given vertex `u` from source vertex `v`
void printPath(vector<vector<int>> const &path, int v, int u)
{
    if (path[v][u] == v)
    {
        return;
    }
    printPath(path, v, path[v][u]);
    cout << path[v][u] << ", ";
}

// Function to print the shortest cost with path information between
// all pairs of vertices
void printSolution(vector<vector<int>> const &cost, vector<vector<int>> const
&path)
{
    int n = cost.size();
    for (int v = 0; v < n; v++)
    {
        for (int u = 0; u < n; u++)
        {
            if (u != v && path[v][u] != -1)
            {
                cout << "The shortest path from " << v << " to " << u << " is
["
                     << v << ", ";
                printPath(path, v, u);
                cout << u << "]" << endl;
            }
        }
    }
}

// Function to run the Floyd-Warshall algorithm
void floydWarshall(vector<vector<int>> const &adjMatrix, int n)
{
    // base case
    if (n == 0)
    {
        return;
    }

    // cost[] and path[] stores shortest path
    // (shortest cost/shortest route) information
    vector<vector<int>> cost(n, vector<int>(n));
    vector<vector<int>> path(n, vector<int>(n));

    // initialize cost[] and path[]
    for (int v = 0; v < n; v++)
    {
        for (int u = 0; u < n; u++)
        {
            // initially, cost would be the same as the weight of the edge
            cost[v][u] = adjMatrix[v][u];

            if (v == u)
            {
                path[v][u] = 0;
            }
            else if (cost[v][u] != INT_MAX)
            {
                path[v][u] = v;
            }
            else
            {
                path[v][u] = -1;
            }
        }
    }

    // run Floyd-Warshall
    for (int k = 0; k < n; k++)
    {
        for (int v = 0; v < n; v++)
        {
            for (int u = 0; u < n; u++)
            {
                // If vertex `k` is on the shortest path from `v` to `u`,
                // then update the value of cost[v][u] and path[v][u]

                if (cost[v][k] != INT_MAX && cost[k][u] != INT_MAX &&
cost[v][k] + cost[k][u] < cost[v][u])
                {
                    cost[v][u] = cost[v][k] + cost[k][u];
                    path[v][u] = path[k][u];
                }
            }
            // if diagonal elements become negative, the
            // graph contains a negative-weight cycle
            if (cost[v][v] < 0)
            {
                cout << "Negative-weight cycle found!!";
                return;
            }
        }
    }

    // Print the shortest path between all pairs of vertices
    printSolution(cost, path);
}

int main()
{
    int n;
    cout << "Enter the number of cities : ";
    cin >> n;

    vector<vector<int>> distance;
    int tempDis;

    cout << "Enter the adjacency Matrix : " << endl;
    for (int i = 0; i < n; i++)
    {
        vector<int> temp;

        for (int j = 0; j < n; j++)
        {
            if (i != j)
            {
                cout << "Enter the distance from " << i << " -> " << j << " :
";
                cin >> tempDis;
                temp.push_back(tempDis);
            }
            else
            {
                temp.push_back(0);
            }
        }
        distance.push_back(temp);
    }
    cout << "The adjacency matrix is : " << endl;
    cout << "[" << endl;
    for (int i = 0; i < n; i++)
    {
        cout << "[ ";
        for (int j = 0; j < n; j++)
        {
            cout << distance[i][j] << " ";
        }
        cout << "]" << endl;
    }
    cout << "]" << endl;
    floydWarshall(distance, n);

    return 0;
}
```

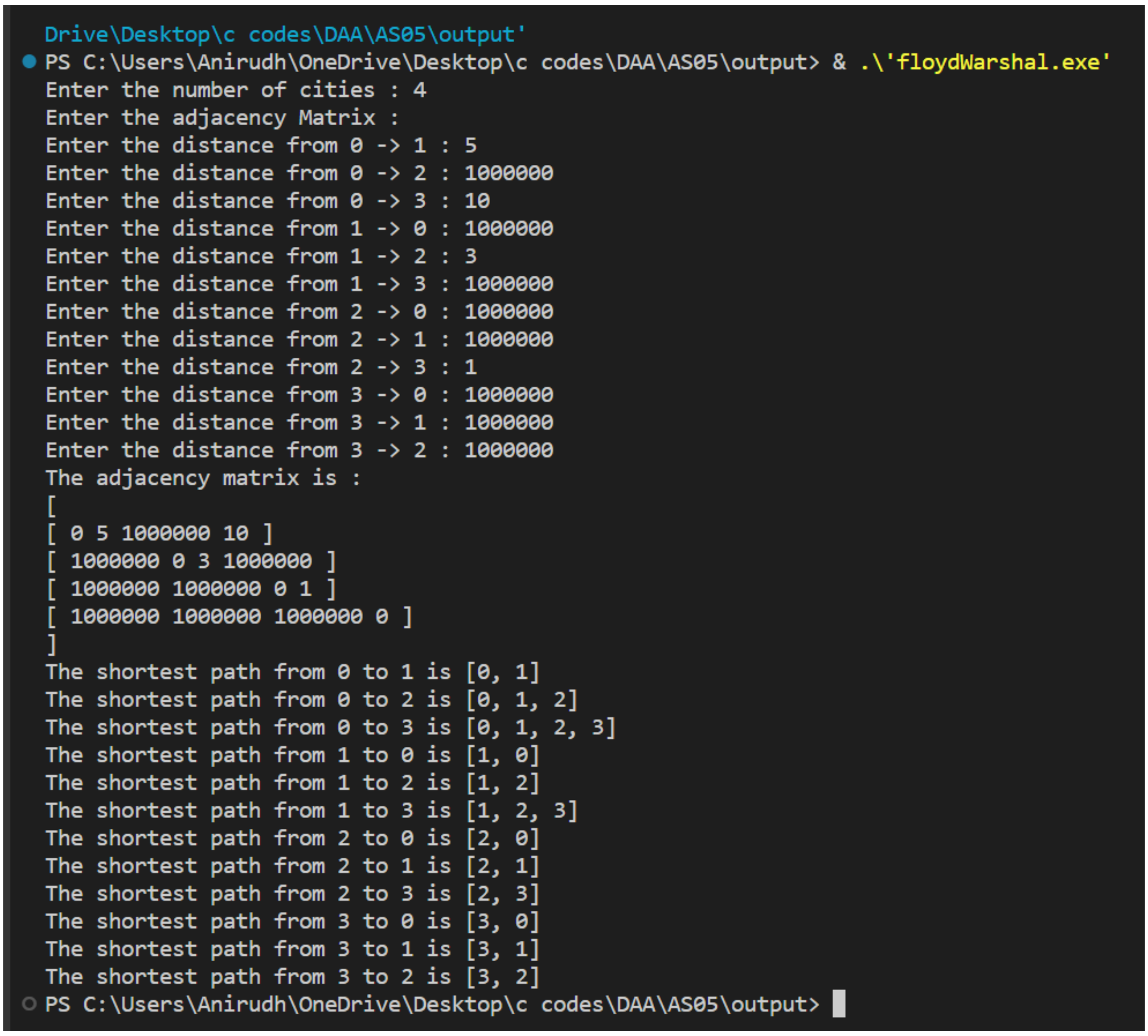
OUTPUT SCREENSHOT:

Test Case 1:



```
utput'
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> & .\'floydWarshal.exe'
Enter the number of cities : 4
Enter the adjacency Matrix :
Enter the distance from 0 -> 1 : 1000000
Enter the distance from 0 -> 2 : -2
Enter the distance from 0 -> 3 : 1000000
Enter the distance from 1 -> 0 : 4
Enter the distance from 1 -> 2 : 3
Enter the distance from 1 -> 3 : 1000000
Enter the distance from 2 -> 0 : 2
Enter the distance from 2 -> 1 : 1000000
Enter the distance from 2 -> 3 : -1
Enter the distance from 3 -> 0 : 1000000
Enter the distance from 3 -> 1 : 4
Enter the distance from 3 -> 2 : 1000000
The adjacency matrix is :
[
[ 0 1000000 -2 1000000 ]
[ 4 0 3 1000000 ]
[ 2 1000000 0 -1 ]
[ 1000000 4 1000000 0 ]
]
The shortest path from 0 to 1 is [0, 2, 3, 1]
The shortest path from 0 to 2 is [0, 2]
The shortest path from 0 to 3 is [0, 2, 3]
The shortest path from 1 to 0 is [1, 0]
The shortest path from 1 to 2 is [1, 0, 2]
The shortest path from 1 to 3 is [1, 0, 2, 3]
The shortest path from 2 to 0 is [2, 0]
The shortest path from 2 to 1 is [2, 3, 1]
The shortest path from 2 to 3 is [2, 3]
The shortest path from 3 to 0 is [3, 1, 0]
The shortest path from 3 to 1 is [3, 1]
The shortest path from 3 to 2 is [3, 1, 0, 2]
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
```

Test Case 2:



```
Drive\Desktop\c codes\DAA\AS05\output'
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output> & .\'floydWarshal.exe'
Enter the number of cities : 4
Enter the adjacency Matrix :
Enter the distance from 0 -> 1 : 5
Enter the distance from 0 -> 2 : 1000000
Enter the distance from 0 -> 3 : 10
Enter the distance from 1 -> 0 : 1000000
Enter the distance from 1 -> 2 : 3
Enter the distance from 1 -> 3 : 1000000
Enter the distance from 2 -> 0 : 1000000
Enter the distance from 2 -> 1 : 1000000
Enter the distance from 2 -> 3 : 1
Enter the distance from 3 -> 0 : 1000000
Enter the distance from 3 -> 1 : 1000000
Enter the distance from 3 -> 2 : 1000000
The adjacency matrix is :
[
[ 0 5 1000000 10 ]
[ 1000000 0 3 1000000 ]
[ 1000000 1000000 0 1 ]
[ 1000000 1000000 1000000 0 ]
]
The shortest path from 0 to 1 is [0, 1]
The shortest path from 0 to 2 is [0, 1, 2]
The shortest path from 0 to 3 is [0, 1, 2, 3]
The shortest path from 1 to 0 is [1, 0]
The shortest path from 1 to 2 is [1, 2]
The shortest path from 1 to 3 is [1, 2, 3]
The shortest path from 2 to 0 is [2, 0]
The shortest path from 2 to 1 is [2, 1]
The shortest path from 2 to 3 is [2, 3]
The shortest path from 3 to 0 is [3, 0]
The shortest path from 3 to 1 is [3, 1]
The shortest path from 3 to 2 is [3, 2]
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
```

QUESTION 5:

Implement Ford Fulkerson algorithm to compute the max-flow of a given graph.

PSEUDOCODE:

The Ford-Fulkerson algorithm is a method to find the maximum flow in a flow network. The main logic behind the Ford-Fulkerson algorithm is to repeatedly find augmenting paths from the source to the sink in the residual graph and update the flow along those paths until no more augmenting paths can be found.

The Ford-Fulkerson algorithm works as follows:

- Initialize the flow network with zero flow on all edges.
- While there exists an augmenting path P in the residual graph Gf, where Gf is the graph representing the residual capacity of each edge:

  a. Find an augmenting path P in Gf using any search algorithm such as breadth-first search or depth-first search.

  b. Find the bottleneck capacity c of the augmenting path P, which is the minimum residual capacity along the path.

  c. Update the flow f along the augmenting path P by adding c to the flow of each edge in the path and subtracting c from the residual capacity of each edge in the path.

- Return the maximum flow, which is the sum of the flow values on all edges leaving the source vertex.

The residual graph is a graph that represents the residual capacity of each edge after the current flow has been subtracted from the original capacity. An augmenting path is a path in the residual graph from the source to the sink that has positive residual capacity on all edges in the path.

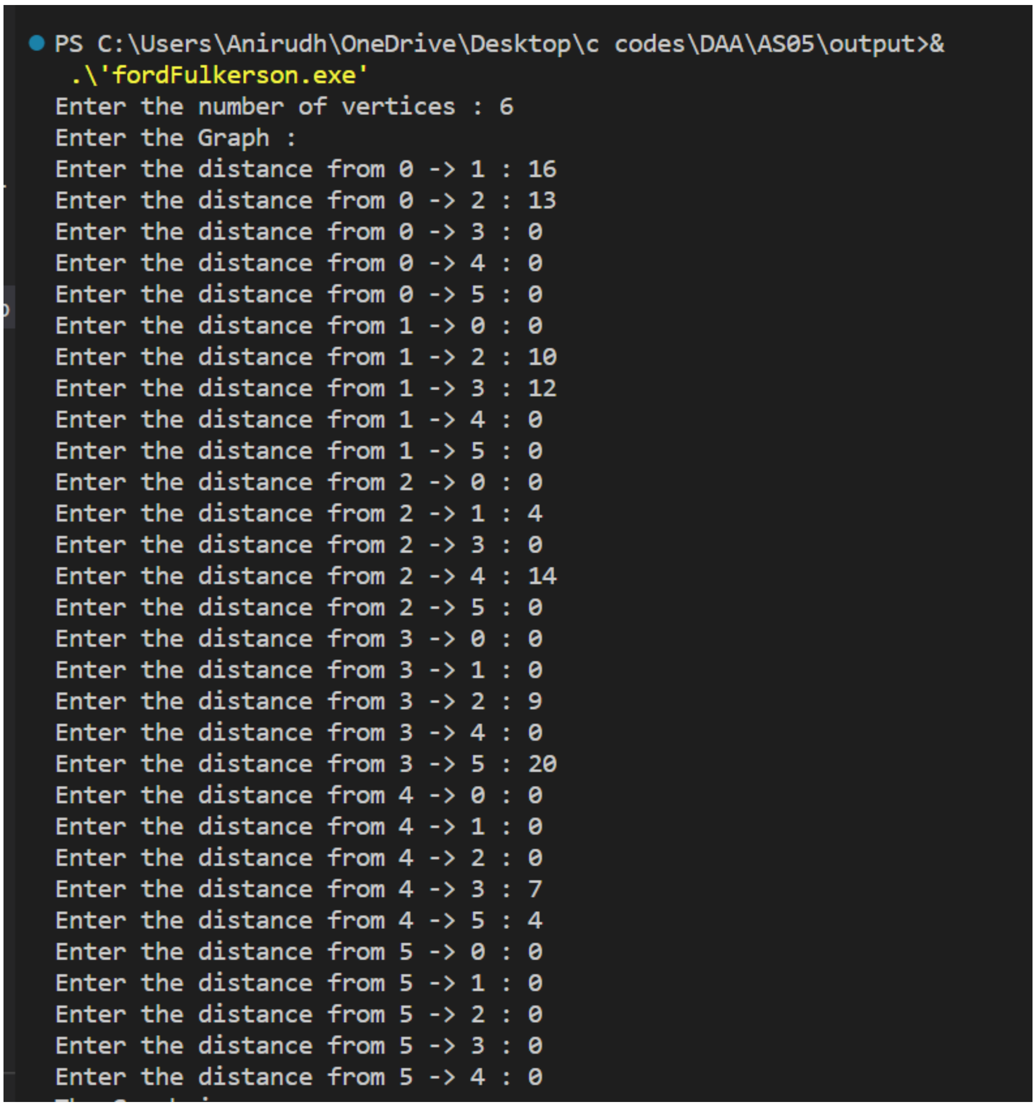
The Ford-Fulkerson algorithm can be implemented using various data structures and algorithms to find augmenting paths, such as depth-first search or breadth-first search. In some cases, the algorithm may not terminate due to the possibility of cycles in the residual graph, and so a variation of the algorithm called the Edmonds-Karp algorithm is used, which always finds the augmenting path with the smallest number of edges.

SOURCE CODE:

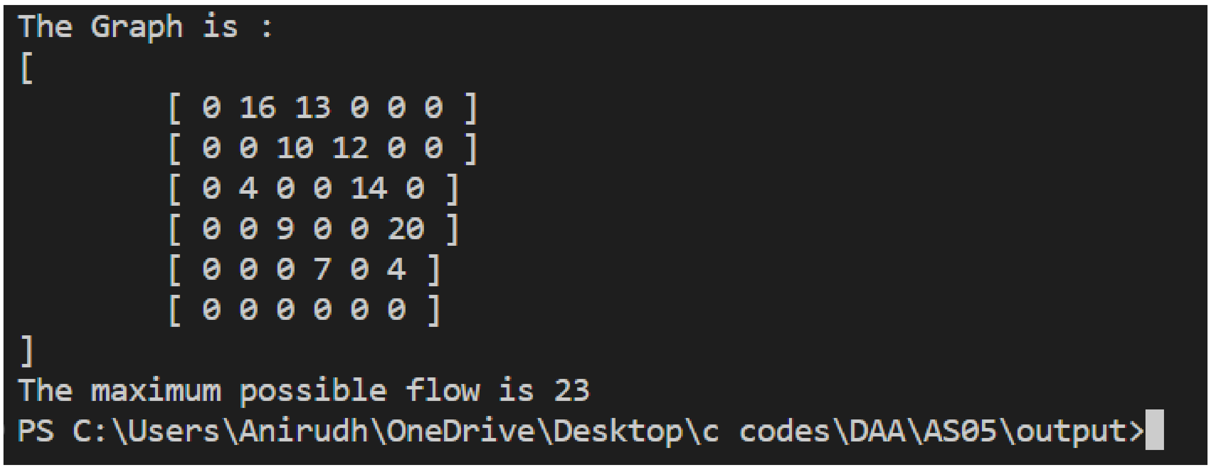
```
#include <iostream>
#include <limits.h>
#include <queue>
#include <string.h>
using namespace std;

// Returns true if there is a path
bool bfs(int **rGraph, int n, int s, int t, int *parent)
{
    // Create a visited array and mark all vertices as not
    // visited
    bool *visited = (bool *)malloc(n * sizeof(bool));
    for (int i = 0; i < n; i++)
    {
        visited[i] = 0;
    }

    // Create a queue, enqueue source vertex and mark source
    // vertex as visited
    queue<int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // BFS
    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v = 0; v < n; v++)
        {
            if (visited[v] == false && rGraph[u][v] > 0)
            {
                if (v == t)
                {
                    parent[v] = u;
                    return true;
                }
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }
    // We didn't reach sink in BFS starting from source
    return false;
}

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int **graph, int n, int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph
    // with given capacities in the original graph as
    // residual capacities in residual graph
    int **rGraph = (int **)malloc(n * sizeof(int *));
    for (u = 0; u < n; u++)
    {
        rGraph[u] = (int *)malloc(n * sizeof(int));
        for (v = 0; v < n; v++)
            rGraph[u][v] = graph[u][v];
    }

    int *parent = (int *)malloc(n * sizeof(int));

    int max_flow = 0; // There is no flow initially

    // Augment the flow while there is path from source to
    // sink
    while (bfs(rGraph, n, s, t, parent))
    {
        int path_flow = INT_MAX;
        for (v = t; v != s; v = parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // update residual capacities of the edges and
        // reverse edges along the path
        for (v = t; v != s; v = parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }
    // Return the overall flow
    return max_flow;
}

// Driver program to test above functions
int main()
{
    int n;
    cout << "Enter the number of vertices : ";
    cin >> n;
    cout << "Enter the Graph : " << endl;
    int **graph = (int **)malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++)
    {
        graph[i] = (int *)malloc(n * sizeof(int));
        for (int j = 0; j < n; j++)
        {
            if (i != j)
            {
                cout << "Enter the distance from " << i << " -> " << j << " :
";
                cin >> graph[i][j];
            }
            else
            {
                graph[i][j] = 0;
            }
        }
    }
    cout << "The Graph is : " << endl;
    cout << "[" << endl;
    for (int i = 0; i < n; i++)
    {
        cout << "\t[ ";
        for (int j = 0; j < n; j++)
        {
            cout << graph[i][j] << " ";
        }
        cout << "]" << endl;
    }
    cout << "]" << endl;
    cout << "The maximum possible flow is " << fordFulkerson(graph, n, 0, 5);

    return 0;
}
```

OUTPUT SCREENSHOT:

Test Case 1:

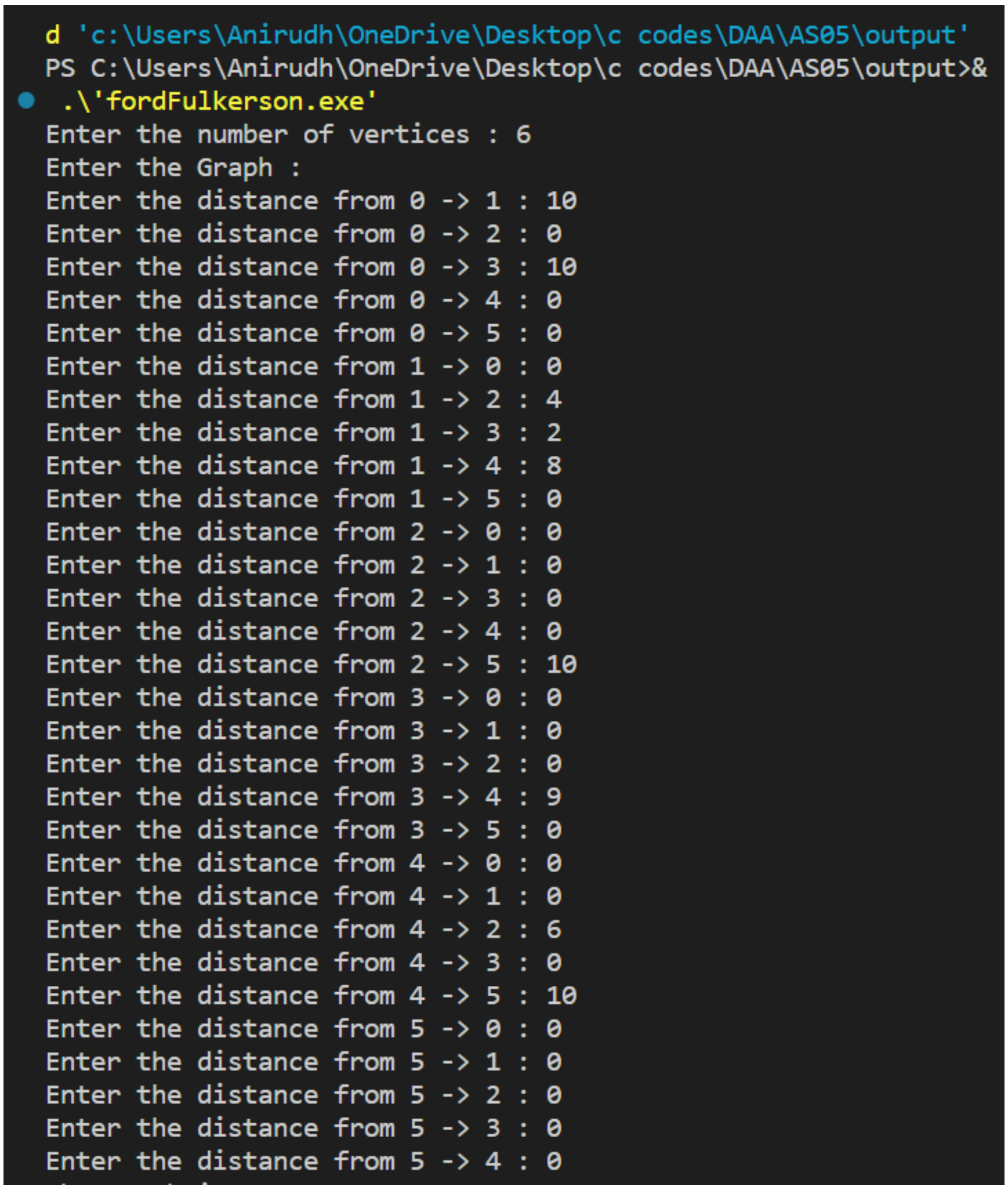


```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>&
 .\'fordFulkerson.exe'
Enter the number of vertices : 6
Enter the Graph :
Enter the distance from 0 -> 1 : 16
Enter the distance from 0 -> 2 : 13
Enter the distance from 0 -> 3 : 0
Enter the distance from 0 -> 4 : 0
Enter the distance from 0 -> 5 : 0
Enter the distance from 1 -> 0 : 0
Enter the distance from 1 -> 2 : 10
Enter the distance from 1 -> 3 : 12
Enter the distance from 1 -> 4 : 0
Enter the distance from 1 -> 5 : 0
Enter the distance from 2 -> 0 : 0
Enter the distance from 2 -> 1 : 4
Enter the distance from 2 -> 3 : 0
Enter the distance from 2 -> 4 : 14
Enter the distance from 2 -> 5 : 0
Enter the distance from 3 -> 0 : 0
Enter the distance from 3 -> 1 : 0
Enter the distance from 3 -> 2 : 9
Enter the distance from 3 -> 4 : 0
Enter the distance from 3 -> 5 : 20
Enter the distance from 4 -> 0 : 0
Enter the distance from 4 -> 1 : 0
Enter the distance from 4 -> 2 : 0
Enter the distance from 4 -> 3 : 7
Enter the distance from 4 -> 5 : 4
Enter the distance from 5 -> 0 : 0
Enter the distance from 5 -> 1 : 0
Enter the distance from 5 -> 2 : 0
Enter the distance from 5 -> 3 : 0
Enter the distance from 5 -> 4 : 0
```

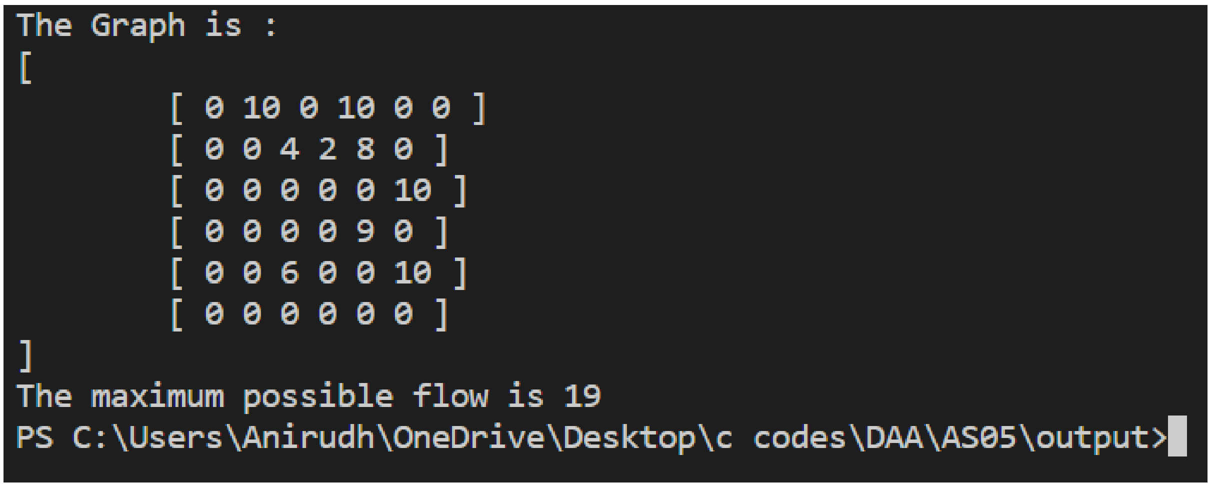


```
The Graph is :
[
        [ 0 16 13 0 0 0 ]
        [ 0 0 10 12 0 0 ]
        [ 0 4 0 0 14 0 ]
        [ 0 0 9 0 0 20 ]
        [ 0 0 0 7 0 4 ]
        [ 0 0 0 0 0 0 ]
]
The maximum possible flow is 23
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
```

Test Case 2:



```
 d 'c:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output'
 PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>&
● .\'fordFulkerson.exe'
 Enter the number of vertices : 6
 Enter the Graph :
 Enter the distance from 0 -> 1 : 10
 Enter the distance from 0 -> 2 : 0
 Enter the distance from 0 -> 3 : 10
 Enter the distance from 0 -> 4 : 0
 Enter the distance from 0 -> 5 : 0
 Enter the distance from 1 -> 0 : 0
 Enter the distance from 1 -> 2 : 4
 Enter the distance from 1 -> 3 : 2
 Enter the distance from 1 -> 4 : 8
 Enter the distance from 1 -> 5 : 0
 Enter the distance from 2 -> 0 : 0
 Enter the distance from 2 -> 1 : 0
 Enter the distance from 2 -> 3 : 0
 Enter the distance from 2 -> 4 : 0
 Enter the distance from 2 -> 5 : 10
 Enter the distance from 3 -> 0 : 0
 Enter the distance from 3 -> 1 : 0
 Enter the distance from 3 -> 2 : 0
 Enter the distance from 3 -> 4 : 9
 Enter the distance from 3 -> 5 : 0
 Enter the distance from 4 -> 0 : 0
 Enter the distance from 4 -> 1 : 0
 Enter the distance from 4 -> 2 : 6
 Enter the distance from 4 -> 3 : 0
 Enter the distance from 4 -> 5 : 10
 Enter the distance from 5 -> 0 : 0
 Enter the distance from 5 -> 1 : 0
 Enter the distance from 5 -> 2 : 0
 Enter the distance from 5 -> 3 : 0
 Enter the distance from 5 -> 4 : 0
```



```
The Graph is :
[
        [ 0 10 0 10 0 0 ]
        [ 0 0 4 2 8 0 ]
        [ 0 0 0 0 0 10 ]
        [ 0 0 0 0 9 0 ]
        [ 0 0 6 0 0 10 ]
        [ 0 0 0 0 0 0 ]
]
The maximum possible flow is 19
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS05\output>
```