VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT on

Analysis and Design of Algorithms

Submitted by

Deekshith B(1BM22CS082)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Analysis and Design of Algorithms**" carried out by **Deekshith B(1BM22CS082),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester April-2024 to August-2024. The Lab report has been approved as it satisfies the academic requirements in respect of an **Analysis and Design of Algorithms (23CS4PCADA)** work prescribed for the said degree.

Dr. K. Panimozhi	Dr. Jyothi S Nayak
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

Index Sheet

Course Outcome

CO1	Analyze time complexity of Recursive and Non-recursive algorithms using asymptotic notations.
CO2	Apply various design techniques for the given problem.
CO3	Apply the knowledge of complexity classes P, NP, and NP-Complete and prove certain problems are NP-Complete
CO4	Design efficient algorithms and conduct practical experiments to solve problems.

1. Leetcode Exercise on Stacks: Count Collisions on a Road

There are n cars on an infinitely long road. The cars are numbered from 0 to n - 1 from left to right and each car is present at a unique point.

You are given a 0-indexed string directions of length n. directions[i] can be either 'L', 'R', or 'S' denoting whether the ith car is moving towards the left, towards the right, or staying at its current point respectively. Each moving car has the same speed.

The number of collisions can be calculated as follows:

When two cars moving in opposite directions collide with each other, the number of collisions increases by 2.

When a moving car collides with a stationary car, the number of collisions increases by 1.

After a collision, the cars involved can no longer move and will stay at the point where they collided. Other than that, cars cannot change their state or direction of motion.

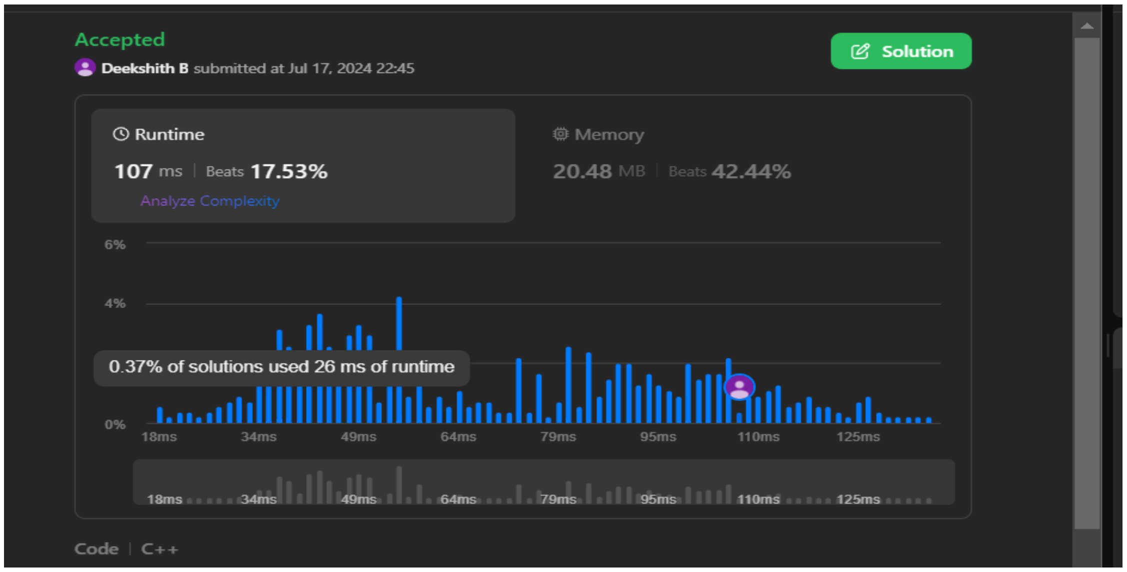
Return the total number of collisions that will happen on the road.

Program:

```
class Solution { public:     int

countCollisions(string directions) {


   int n = directions.size();

stack<char> s;     int ans =

0;     for(int i=0; i<n; i++)

   {

      if(s.empty()) s.push(directions[i]);        else if( (s.top()

== 'S') && (directions[i] == 'L')) ans++;        else if(s.top()

== 'R' && (directions[i] == 'L')) {

     s.pop();

directions[i] = 'S';
```

```
            i--;

            ans += 2;

        }

        else if( (s.top() == 'R') && (directions[i] == 'S')){

            s.pop();

directions[i] = 'S';

            i--;

            ans++;

        }

        else{

            s.push(directions[i]);

        }

    }

    return ans;

}

};
```

Output: Count Collisions on a Road



Accepted

Deekshith B submitted at Jul 17, 2024 22:45

Runtime

107 ms | Beats 17.53%

Memory

20.48 MB | Beats 42.44%

0.37% of solutions used 26 ms of runtime

Code | C++

2. Leetcode Exercise on DFS: Atlantic Ocean Water Flow:

There is an m x n rectangular island that borders both the Pacific Ocean and Atlantic Ocean. The Pacific Ocean touches the island's left and top edges, and the Atlantic Ocean touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an m x n integer matrix heights where heights[r][c] represents the height above sea level of the cell at coordinate (r, c).

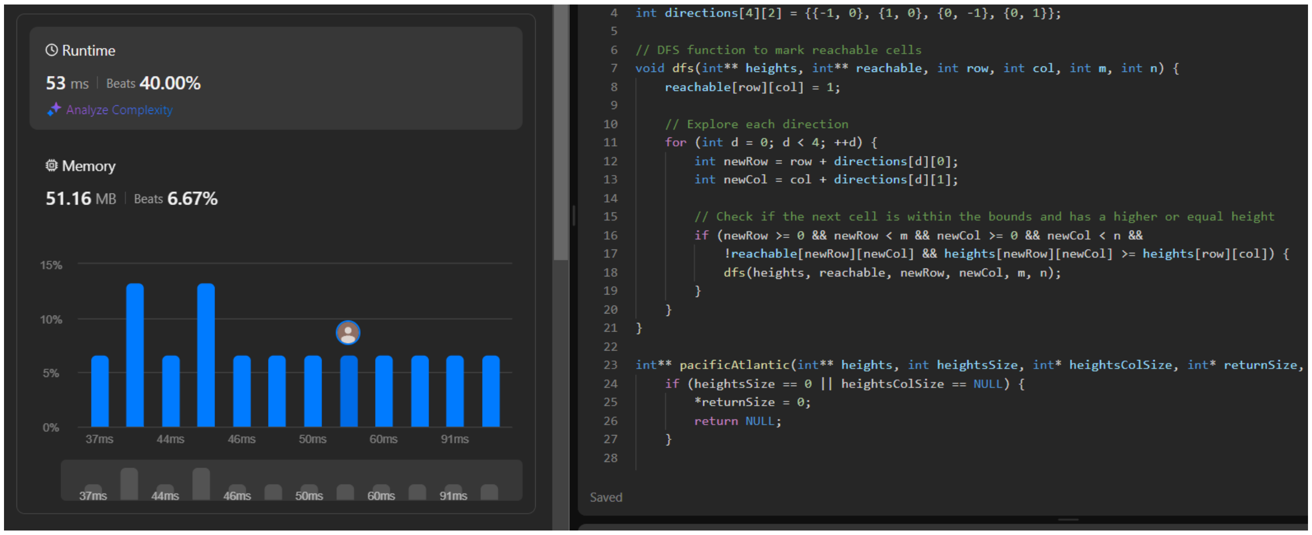
The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is less than or equal to the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return a 2D list of grid coordinates result where result[i] = [ri, ci] denotes that rain water can flow from cell (ri, ci) to both the Pacific and Atlantic oceans.

```
Program: class
Solution {
public:    int
m,n;
   vector<vector<bool>>atlantic,pacific;     vector<vector<int>>ans;
vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights) {
       if(!size(heights)) return ans;         m =
size(heights), n = size(heights[0]);

       atlantic = pacific = vector<vector<bool>>(m,vector<bool>(n, false));

       for(int i=0; i<m; i++) dfs(heights,pacific,i,0), dfs(heights, atlantic, i, n-1);
for(int i=0; i<n; i++) dfs(heights,pacific,0,i), dfs(heights, atlantic, m-1,i);         return
ans;
```

```
    }

    void dfs(vector<vector<int>>&mat, vector<vector<bool>>&visited, int i, int j)
    {
        if(visited[i][j]) return;
visited[i][j]=true;

        if(atlantic[i][j] && pacific[i][j]) ans.push_back(vector<int>{i,j});

        if(i+1 < m && mat[i+1][j] >=mat[i][j]) dfs(mat,visited,i+1,j);
if(i - 1 >= 0 && mat[i - 1][j] >= mat[i][j]) dfs(mat, visited, i - 1, j);
if(j + 1 <  n && mat[i][j + 1] >= mat[i][j]) dfs(mat, visited, i, j + 1);
if(j - 1 >= 0 && mat[i][j - 1] >= mat[i][j]) dfs(mat, visited, i, j - 1);
    }
};
```

Output: Atlantic Ocean Water Flow:


Runtime

53 ms | Beats 40.00%

Analyze Complexity

Memory

51.16 MB | Beats 6.67%

```
int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

// DFS function to mark reachable cells
void dfs(int** heights, int** reachable, int row, int col, int m, int n) {
    reachable[row][col] = 1;

    // Explore each direction
    for (int d = 0; d < 4; ++d) {
        int newRow = row + directions[d][0];
        int newCol = col + directions[d][1];

        // Check if the next cell is within the bounds and has a higher or equal height
        if (newRow >= 0 && newRow < m && newCol >= 0 && newCol < n &&
            !reachable[newRow][newCol] && heights[newRow][newCol] >= heights[row][col]) {
            dfs(heights, reachable, newRow, newCol, m, n);
        }
    }
}

int** pacificAtlantic(int** heights, int heightsSize, int* heightsColSize, int* returnSize,
    if (heightsSize == 0 || heightsColSize == NULL) {
        *returnSize = 0;
        return NULL;
    }

```

Saved

3. A) Topological Sorting:

Write a program to obtain the Topological ordering of vertices in a given digraph using Source Removal method

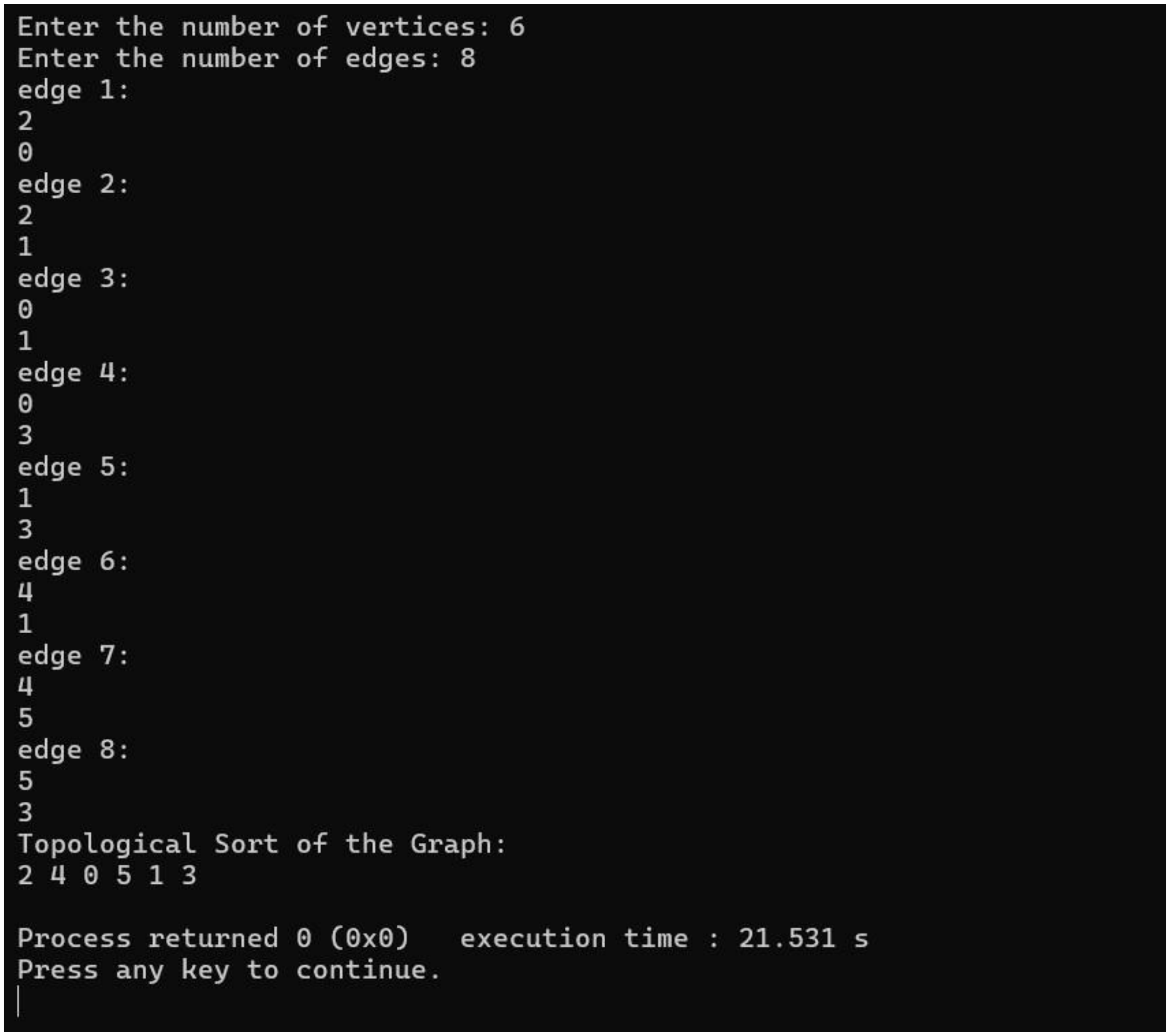
Program:

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_VERTICES 100


// Function to represent an adjacency list (modified for simplicity)

int graph[MAX_VERTICES][MAX_VERTICES];


// Function to add an edge to the graph (modified for simplicity)

void addEdge(int src, int dest) {   graph[src][dest] = 1;

}


// Function to perform topological sort using source removal

void topologicalSort(int V) {   int in_degree[MAX_VERTICES] =

{0};   int sorted[MAX_VERTICES];

  int index = 0;


  // Calculate in-degree of each vertex

  for (int v = 0; v < V; ++v) {

for (int w = 0; w < V; ++w) {

if (graph[v][w] == 1) {

in_degree[w]++;
```

```
      }
    }
  }

  // Find sources (nodes with in-degree 0)
  for (int i = 0; i < V; i++) {
if (in_degree[i] == 0) {
sorted[index++] = i;
    }
  }

  // Process sources and update in-degree of neighbors
  for (int i = 0; i < index; i++) {
int v = sorted[i];     for (int w =
0; w < V; ++w) {       if
(graph[v][w] == 1) {
in_degree[w]--;         if
(in_degree[w] == 0) {
sorted[index++] = w;
        }
      }
    }
  }

  // Check for cycle
if (index != V) {
printf("Graph
contains a cycle.
Topological sort not
possible.\n");
  } else {
    printf("Topological Sort of the Graph:\n");
    for (int i = 0; i < index; i++) {
printf("%d ", sorted[i]);
    }
    printf("\n");
  }
}

int main() {
    int i,a,b,V,E;
    printf("Enter the number of vertices: ");
scanf("%d",&V);     printf("Enter the
number of edges: ");     scanf("%d",&E);
    for(i=0;i<E;i++){
printf("edge %d:\n",i+1);
scanf("%d",&a);
scanf("%d",&b);
addEdge(a,b);
    }

  topologicalSort(V);

  return 0;
```

```
}
```

Output: Topological Sorting



```
Enter the number of vertices: 6
Enter the number of edges: 8
edge 1:
2
0
edge 2:
2
1
edge 3:
0
1
edge 4:
0
3
edge 5:
1
3
edge 6:
4
1
edge 7:
4
5
edge 8:
5
3
Topological Sort of the Graph:
2 4 0 5 1 3

Process returned 0 (0x0)   execution time : 21.531 s
Press any key to continue.
```

B) Leetcode Exercise on Graphs: Number of Restricted Paths from first to last node

There is an undirected weighted connected graph. You are given a positive integer n which denotes that the graph has n nodes labeled from 1 to n, and an array edges where each edges[i] = [ui, vi, weighti] denotes that there is an edge between nodes ui and vi with weight equal to weighti.

A path from node start to node end is a sequence of nodes [z0, z1, z2, ..., zk] such that z0 = start and zk = end and there is an edge between zi and zi+1 where 0 <= i <= k-1.

The distance of a path is the sum of the weights on the edges of the path. Let distanceToLastNode(x) denote the shortest distance of a path between node n and node x. A restricted path is a path that also satisfies that distanceToLastNode(zi) > distanceToLastNode(zi+1) where 0 <= i <= k-1.

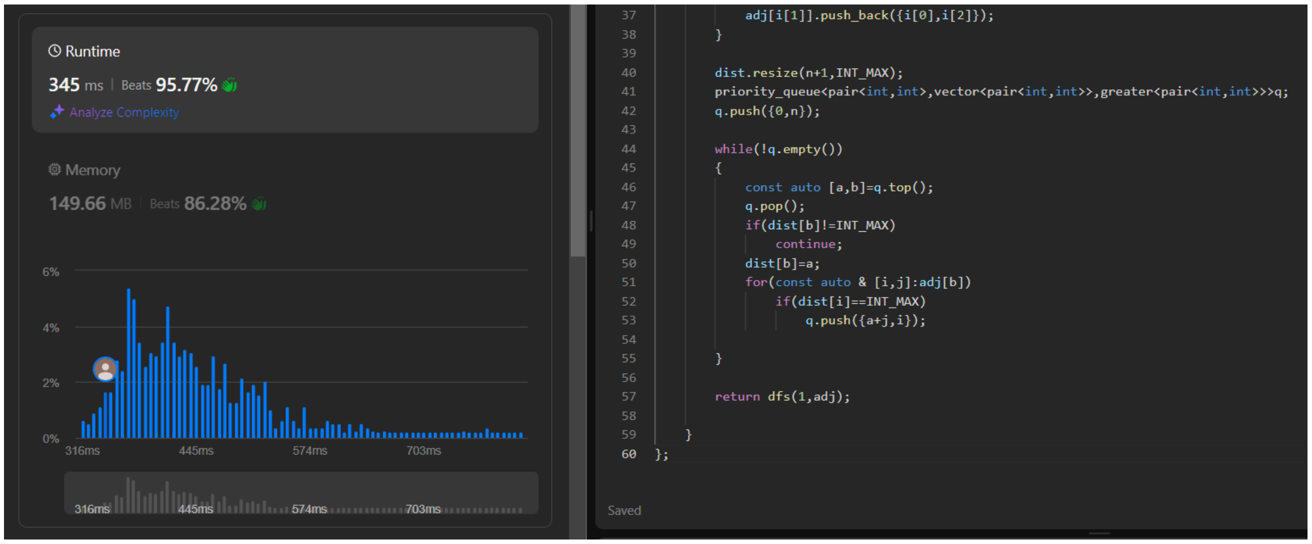
Return the number of restricted paths from node 1 to node n. Since that number may be too large, return it modulo 109 + 7.

Program:

```
class Solution {

vector<int>dist;     int

node;

  int MOD=1e9+7;     vector<int>dp;

int dfs(int i,vector<pair<int,int>>adj[])

  {

      if(i==node)           return 1;

if(dp[i]!=-1)           return dp[i];

int ans=0;        for(const auto &

[a,b]:adj[i])

          if(dist[i]>dist[a])

          {

              ans+=dfs(a,adj);

ans%=MOD;
```

```
          }

        return dp[i]=ans;
    }

public:
    // this makes the code run faster but is not needed for the code to function     Solution()
    {
        ios_base::sync_with_stdio(0);
        cin.tie(0);
    }
    int countRestrictedPaths(int n, vector<vector<int>>& edges) {
        node=n;
vector<pair<int,int>>adj[n+1];
dp.resize(n+1,-1);         for(const
auto & i:edges)
        {
            adj[i[0]].push_back({i[1],i[2]});
adj[i[1]].push_back({i[0],i[2]});
        }

        dist.resize(n+1,INT_MAX);
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>q;         q.push({0,n});

        while(!q.empty())
        {
            const auto [a,b]=q.top();
            q.pop();

            if(dist[b]!=INT_MAX)

               continue;           dist[b]=a;

for(const auto & [i,j]:adj[b])

if(dist[i]==INT_MAX)

                  q.push({a+j,i});


         }


         return dfs(1,adj);


      }

};
```

Output: Number of Restricted Paths from first to last node


Runtime
345 ms | Beats 95.77%

Memory
149.66 MB | Beats 86.28%

```
            adj[i[1]].push_back({i[0],i[2]});
        }

        dist.resize(n+1,INT_MAX);
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>q;
        q.push({0,n});

        while(!q.empty())
        {
            const auto [a,b]=q.top();
            q.pop();
            if(dist[b]!=INT_MAX)
                continue;
            dist[b]=a;
            for(const auto & [i,j]:adj[b])
                if(dist[i]==INT_MAX)
                    q.push({a+j,i});

        }

        return dfs(1,adj);

    }
};
```

4. Johnson Trotter Method:

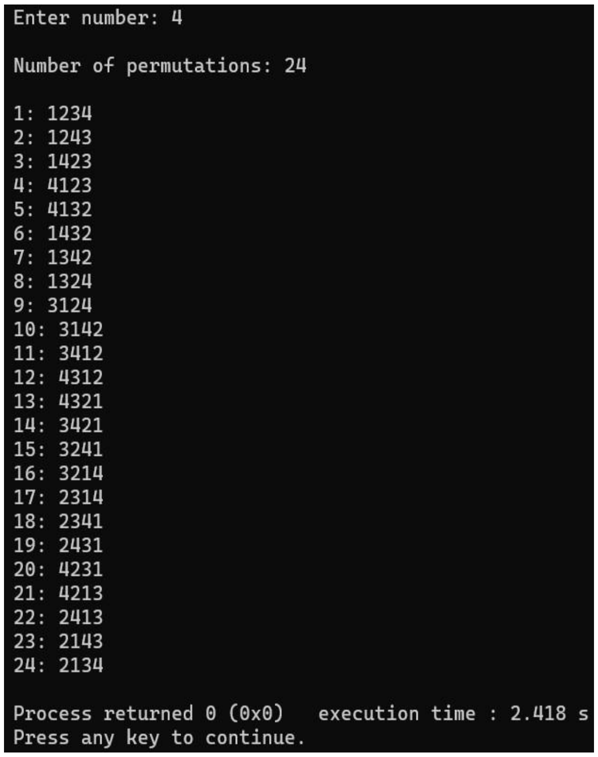
Implement Johnson Trotter algorithm to generate permutations.

Program:

```
#include <stdio.h>

#include <stdbool.h>


#define LEFT_TO_RIGHT true

#define RIGHT_TO_LEFT false


void swap(int *a, int *b) {

int temp = *a;     *a = *b;

   *b = temp;

}


int searchArr(int a[], int n, int mobile) {

   for (int i = 0; i < n; i++)

if (a[i] == mobile)

return i + 1;

   return -1; // should never be reached if mobile is in the array

}


int getMobile(int a[], bool dir[], int n) {

int mobile_prev = 0, mobile = 0;

   for (int i = 0; i < n; i++) {

      if (dir[a[i] - 1] == RIGHT_TO_LEFT && i != 0) {
            if (a[i] > a[i - 1] && a[i] > mobile_prev) {
                mobile = a[i];
                mobile_prev = mobile;
            }
        }
        if (dir[a[i] - 1] == LEFT_TO_RIGHT && i != n - 1) {
if (a[i] > a[i + 1] && a[i] > mobile_prev) {
                mobile = a[i];
                mobile_prev = mobile;
            }
        }
    }
    return mobile;
}

void printOnePerm(int a[], bool dir[], int n) {
int mobile = getMobile(a, dir, n);     int pos =
searchArr(a, n, mobile);

    if (dir[a[pos - 1] - 1] == RIGHT_TO_LEFT)
swap(&a[pos - 1], &a[pos - 2]);     else if
(dir[a[pos - 1] - 1] == LEFT_TO_RIGHT)
swap(&a[pos], &a[pos - 1]);

    for (int i = 0; i < n; i++)
printf("%d", a[i]);     printf("\n");
```

```
    // Change the direction of all elements greater than the current mobile
    for (int i = 0; i < n; i++) {         if
(a[i] > mobile) {             dir[a[i] -
1] = !dir[a[i] - 1];
        }
    }
}

int fact(int n) {     int res =
1;     for (int i = 1; i <= n;
i++)         res = res * i;
    return res;
}

void printPermutation(int n) {
    int a[n];
bool dir[n];

    printf("1: ");     for (int i
= 0; i < n; i++) {         a[i] = i
+ 1;
        printf("%d", a[i]);
    }
    printf("\n");

    for (int i = 0; i < n; i++)
        dir[i] = RIGHT_TO_LEFT;
    for (int i = 1; i < fact(n); i++){
printf("%d: ",i+1);        printOnePerm(a,
dir, n);
    }
}


int main() {
    int n;
    printf("Enter number: ");     scanf("%d", &n);
printf("\nNumber of permutations: %d\n\n", fact(n));
printPermutation(n);
    return 0;
}
```

Output: Johnson Trotter Method



```
Enter number: 4

Number of permutations: 24

1: 1234
2: 1243
3: 1423
4: 4123
5: 4132
6: 1432
7: 1342
8: 1324
9: 3124
10: 3142
11: 3412
12: 4312
13: 4321
14: 3421
15: 3241
16: 3214
17: 2314
18: 2341
19: 2431
20: 4231
21: 4213
22: 2413
23: 2143
24: 2134

Process returned 0 (0x0)   execution time : 2.418 s
Press any key to continue.
```

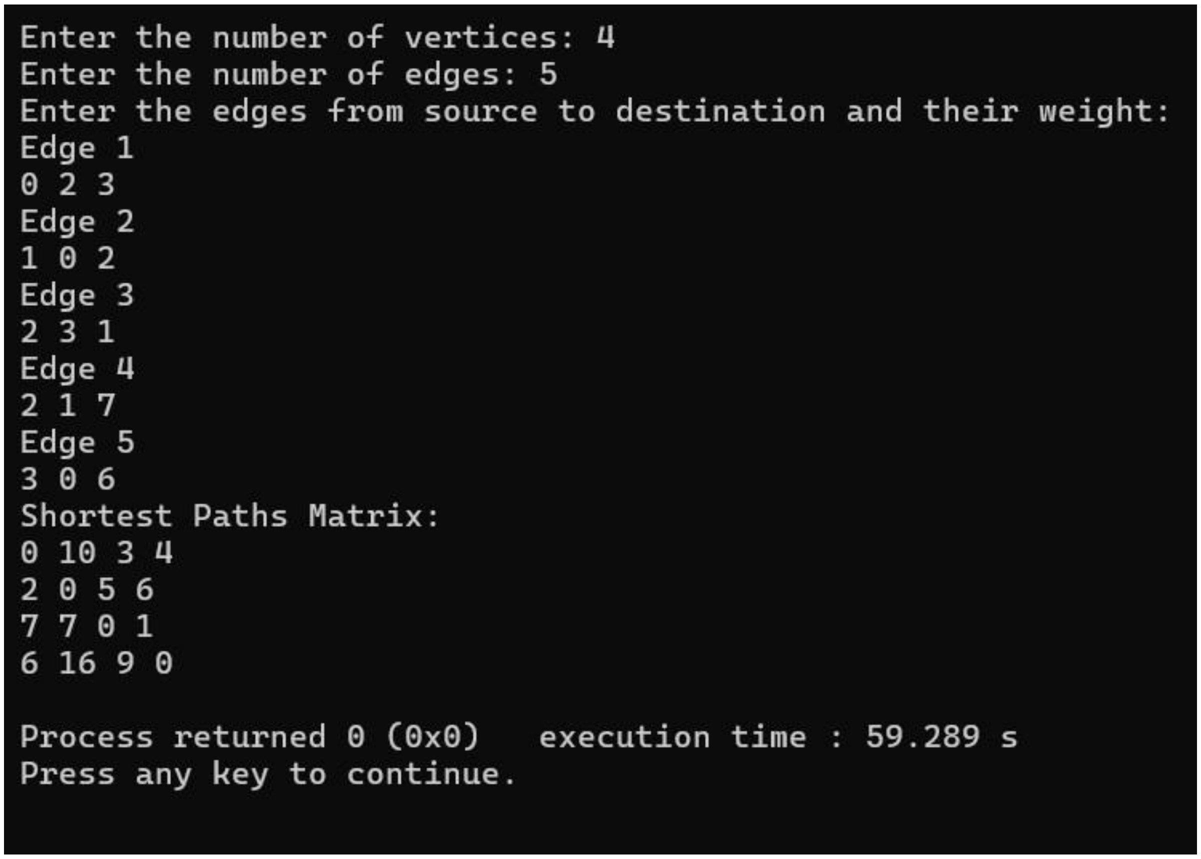
5. A) Floyd’s Algorithm: Implement All pair Shortest paths problem using Floyd’s Algorithm:

Program:

```
#include<stdio.h>
#include<limits.h> // Include limits.h for INT_MAX constant

// Function to find minimum of two integers
int min(int a, int b){
   if(a < b){
return a;     }
else {
return b;
   }
}

int main() {
   int n;
   printf("Enter the number of vertices: ");
scanf("%d", &n);

   // Input validation to ensure n is a positive integer
   if (n <= 0) {
       printf("Invalid number of vertices.\n");
return 1; // Exit with error code 1
   }
```

```
    int graph[n][n];

    // Initialize graph with a large value (INT_MAX)
    for(int i = 0; i < n; i++) {
for(int j = 0; j < n; j++) {
graph[i][j] = INT_MAX;
        }
    }

    // Set diagonal elements to 0
    for(int i = 0; i < n; i++) {
graph[i][i] = 0;
    }

    int edges;     printf("Enter the number
of edges: ");     scanf("%d", &edges);

    printf("Enter the edges from source to destination and their weight:\n");
    for(int i = 0; i < edges; i++) {        int src,
dest, wt;        printf("Edge %d\n", i + 1);
scanf("%d %d %d", &src, &dest, &wt);

        // Input validation to ensure vertices are within range
if (src < 0 || src >= n || dest < 0 || dest >= n) {
printf("Invalid edge. Vertex out of range.\n");           return
1; // Exit with error code 1

        }
        graph[src][dest] = wt;
    }

    // Floyd-Warshall Algorithm
    for(int k = 0; k < n; k++) {
for(int i = 0; i < n; i++) {
for(int j = 0; j < n; j++) {
                // Update shortest path if a shorter path is found
if(graph[i][k] != INT_MAX && graph[k][j] != INT_MAX) {
graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);
                }
            }
        }
    }

    // Output the shortest paths matrix
printf("Shortest Paths Matrix:\n");
    for(int i = 0; i < n; i++) {
for(int j = 0; j < n; j++) {
if(graph[i][j] == INT_MAX) {
                printf("INF ");
} else {
                printf("%d ", graph[i][j]);
            }
        }
        printf("\n");
    }

    return 0;
}
```

Output: Floyd's Algorithm



```
Enter the number of vertices: 4
Enter the number of edges: 5
Enter the edges from source to destination and their weight:
Edge 1
0 2 3
Edge 2
1 0 2
Edge 3
2 3 1
Edge 4
2 1 7
Edge 5
3 0 6
Shortest Paths Matrix:
0 10 3 4
2 0 5 6
7 7 0 1
6 16 9 0

Process returned 0 (0x0)   execution time : 59.289 s
Press any key to continue.
```

B) Leetcode Exercise on Dynamic programming: Unique Paths

There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

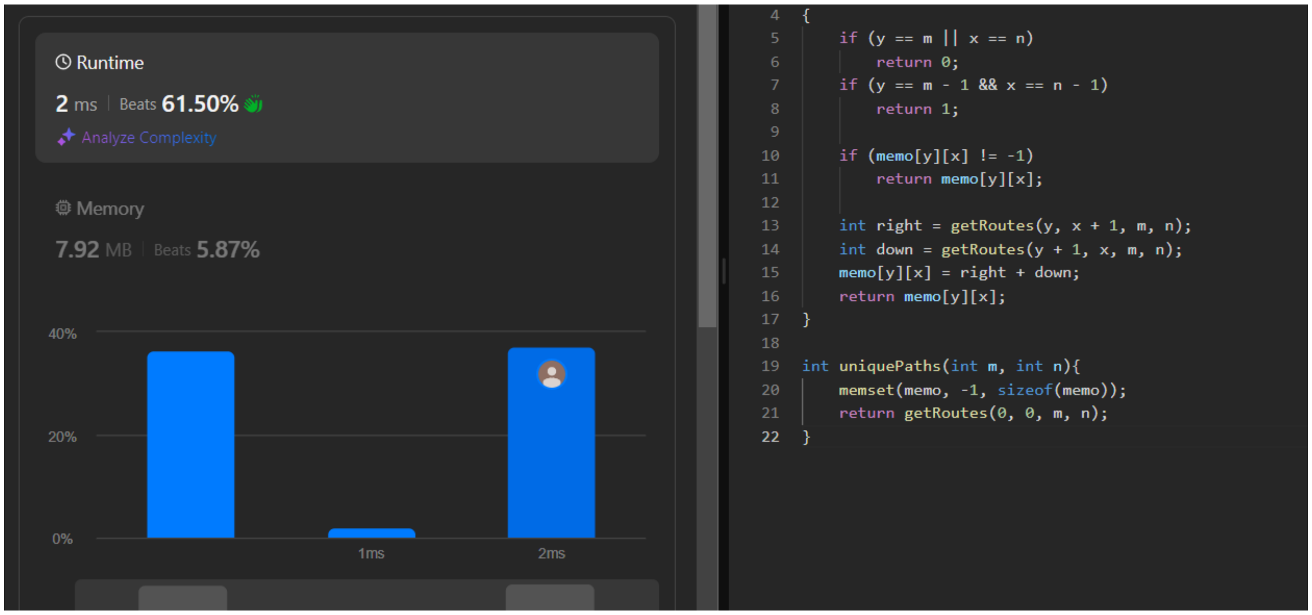
Given the two integers m and n, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to 2 * 10^9.

Program:

```
int memo[101][101];


int getRoutes(int y, int x, int m, int n)
{
   if (y == m || x == n)
return 0;    if (y == m - 1 &&
x == n - 1)
      return 1;


   if (memo[y][x] != -1)
return memo[y][x];


   int right = getRoutes(y, x + 1, m, n);
int down = getRoutes(y + 1, x, m, n);
memo[y][x] = right + down;    return
memo[y][x];
}


int uniquePaths(int m, int n){
   memset(memo, -1, sizeof(memo));
   return getRoutes(0, 0, m, n);
}
```

Output: Unique Paths



Runtime

2 ms | Beats 61.50%

Analyze Complexity

Memory

7.92 MB | Beats 5.87%

```
{
    if (y == m || x == n)
        return 0;
    if (y == m - 1 && x == n - 1)
        return 1;

    if (memo[y][x] != -1)
        return memo[y][x];

    int right = getRoutes(y, x + 1, m, n);
    int down = getRoutes(y + 1, x, m, n);
    memo[y][x] = right + down;
    return memo[y][x];
}

int uniquePaths(int m, int n){
    memset(memo, -1, sizeof(memo));
    return getRoutes(0, 0, m, n);
}
```

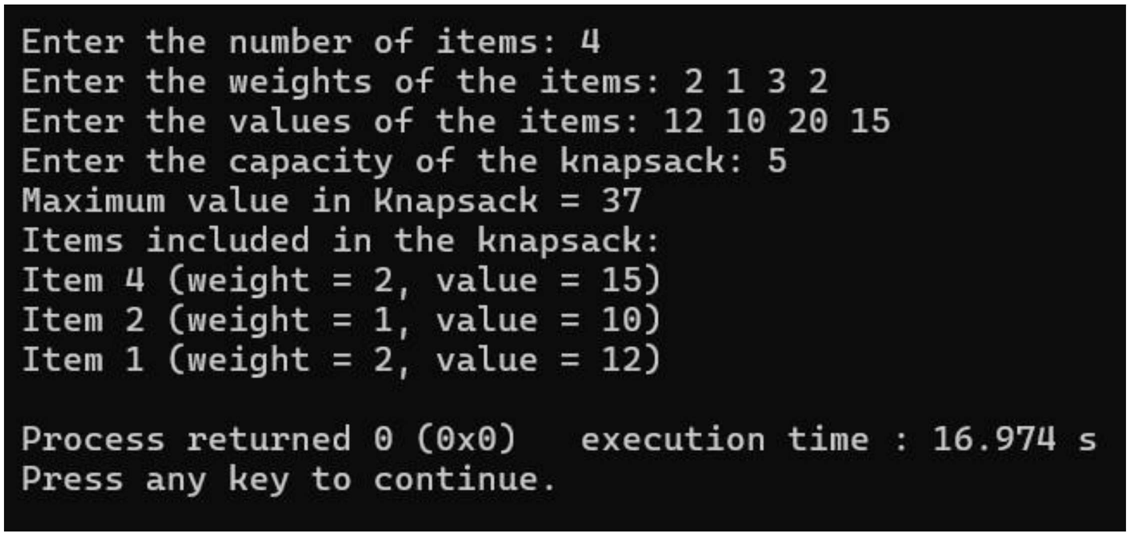
6. A) 0/1 Knapsack Problem using Dynamic Programming:

Program:

```
#include <stdio.h>

#define MAX_ITEMS 100
#define MAX_CAPACITY 1000

// Function to return the maximum of two integers
int max(int a, int b) {
return (a > b) ? a : b;
}

// Function to solve the 0/1 Knapsack problem using dynamic programming
void knapsack(int W, int wt[], int val[], int n) {
   int v[MAX_ITEMS+1][MAX_CAPACITY+1];

   // Build table v[][] in bottom-up manner
   for (int i = 0; i <= n; i++) {
for (int w = 0; w <= W; w++) {
         if (i == 0 || w == 0) {
v[i][w] = 0;
         } else if (wt[i-1] <= w) {              v[i][w] = max(val[i-1] +
v[i-1][w - wt[i-1]], v[i-1][w]);
         } else {
v[i][w] = v[i-1][w];
         }
      }
   }

   // The final value in v[n][W] is the maximum value that can be achieved
printf("Maximum value in Knapsack = %d\n", v[n][W]);

   // Print the items included in the knapsack
int w = W;     printf("Items included in the
knapsack:\n");
   for (int i = n; i > 0 && w > 0; i--) {        if (v[i][w] != v[i-1][w]) {
printf("Item %d (weight = %d, value = %d)\n", i, wt[i-1], val[i-1]);
           w -= wt[i-1];
       }
   }
}

int main() {
int n, W;
   int wt[MAX_ITEMS], val[MAX_ITEMS];

   printf("Enter the number of items: ");
scanf("%d", &n);     printf("Enter the weights
of the items: ");
   for (int i = 0; i < n; i++) {
scanf("%d", &wt[i]);
   }
   printf("Enter the values of the items: ");
```

```
    for (int i = 0; i < n; i++) {
scanf("%d", &val[i]);
    }
    printf("Enter the capacity of the knapsack: ");
scanf("%d", &W);

    knapsack(W, wt, val, n);

    return 0;
}
```

Output: 0/1 Knapsack Problem



```
Enter the number of items: 4
Enter the weights of the items: 2 1 3 2
Enter the values of the items: 12 10 20 15
Enter the capacity of the knapsack: 5
Maximum value in Knapsack = 37
Items included in the knapsack:
Item 4 (weight = 2, value = 15)
Item 2 (weight = 1, value = 10)
Item 1 (weight = 2, value = 12)

Process returned 0 (0x0)   execution time : 16.974 s
Press any key to continue.
```

B) Leetcode Exercise on Dynamic Programming: Edit Distance

Given two strings word1 and word2, return *the minimum number of operations required to convert word1 to word2*.

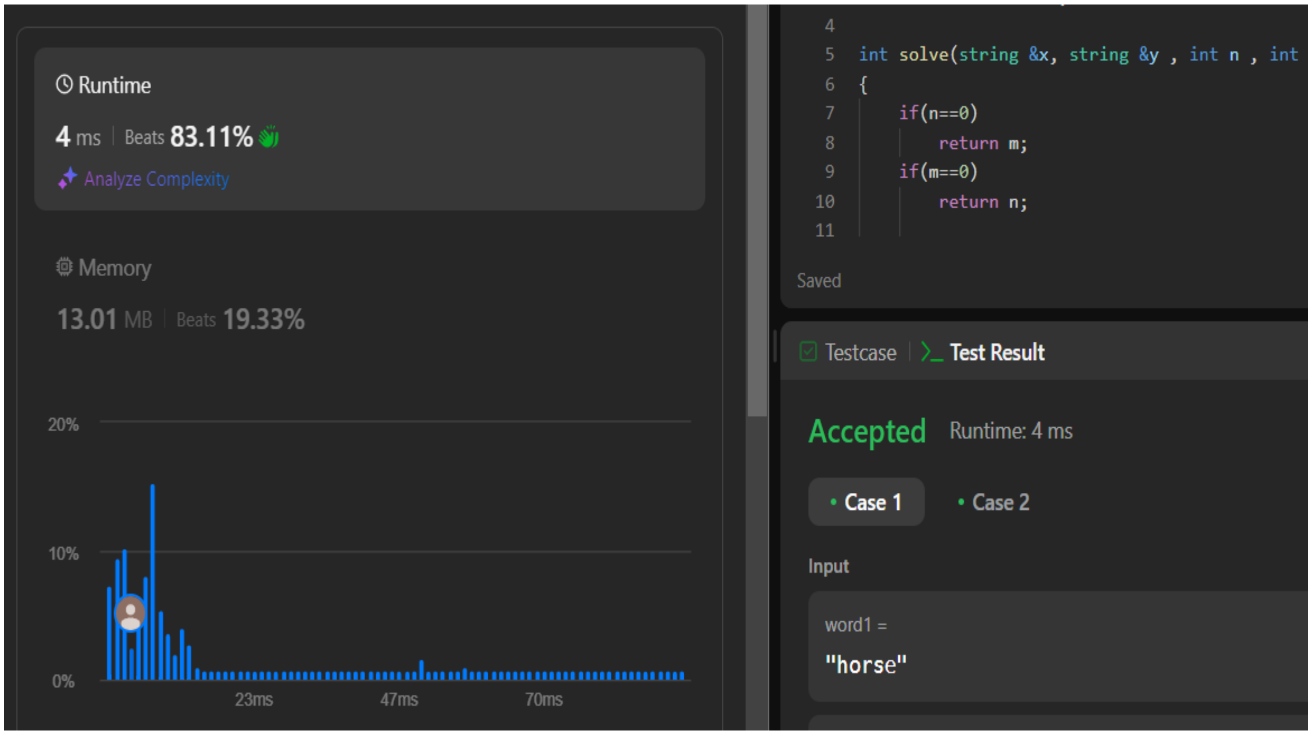
You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

```
Program: class
Solution {
public:
 int minDistance(string word1, string word2) {
const int m = word1.length();//first word length
const int n = word2.length();//second word length
   // dp[i][j] := min # of operations to convert word1[0..i) to word2[0..j)     vector<vector<int>>
dp(m + 1, vector<int>(n + 1));

   for (int i = 1; i <= m; ++i)
dp[i][0] = i;

   for (int j = 1; j <= n; ++j)
dp[0][j] = j;

   for (int i = 1; i <= m; ++i)
for (int j = 1; j <= n; ++j)
       if (word1[i - 1] == word2[j - 1])//same characters
dp[i][j] = dp[i - 1][j - 1];//no operation
```

```
      else
        dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]}) + 1;
                    //replace      //delete      //insert
    return dp[m][n];
  }
};
```

Output: Edit Distance



Runtime

4 ms | Beats 83.11%

Memory

13.01 MB | Beats 19.33%

```
int solve(string &x, string &y , int n , int
{
    if(n==0)
        return m;
    if(m==0)
        return n;
```

Saved

Testcase | Test Result

Accepted Runtime: 4 ms

Case 1 Case 2

Input

word1 =
"horse"

7. Merge Sort: Sort a given set of N integer elements using the Merge sort technique and compute its time taken.

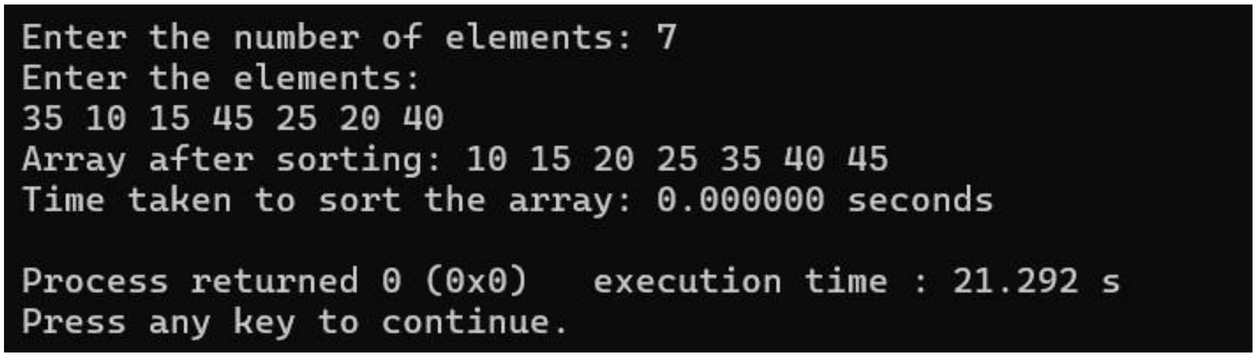
Program:

```
#include <stdio.h>

#include <time.h>


void merge(int a[], int low, int mid, int high) {

   int i, j, k;

   int c[high + 1]; // Temp array to store merged elements


   i = low; // Starting index for left subarray     j =

mid + 1; // Starting index for right subarray     k =

low; // Starting index to be sorted


   while (i <= mid && j <= high) {

      if (a[i] <= a[j]) {

c[k++] = a[i++];

      } else {

c[k++] = a[j++];

      }

   }

   while (i <= mid) {

c[k++] = a[i++];

   }


   while (j <= high) {

      c[k++] = a[j++];
```

```
    }

    for (i = low; i <= high; i++) {
        a[i] = c[i];
    }
}

void mergeSort(int a[], int low, int high) {
    int mid;     if (low < high) {
mid = (low + high) / 2;
mergeSort(a, low, mid);
mergeSort(a, mid + 1, high);
merge(a, low, mid, high);
    }
}
int main() {
    int n;
    printf("Enter the number of elements: ");
scanf("%d", &n);
    int a[n];
    printf("Enter the elements: \n");
    for (int i = 0; i < n; i++) {
scanf("%d", &a[i]);
    }

    // Start measuring time
clock_t start, end;
```

```
        double cpu_time_used;


        start = clock();


        mergeSort(a, 0, n - 1);


        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;


        printf("Array after sorting: ");
        for (int i = 0; i < n; i++) {
    printf("%d ", a[i]);
        }
        printf("\nTime taken to sort the array: %f seconds\n", cpu_time_used);
        return 0;
    }
```

Output:



```
Enter the number of elements: 7
Enter the elements:
35 10 15 45 25 20 40
Array after sorting: 10 15 20 25 35 40 45
Time taken to sort the array: 0.000000 seconds

Process returned 0 (0x0)   execution time : 21.292 s
Press any key to continue.
```

8. Quick Sort: Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

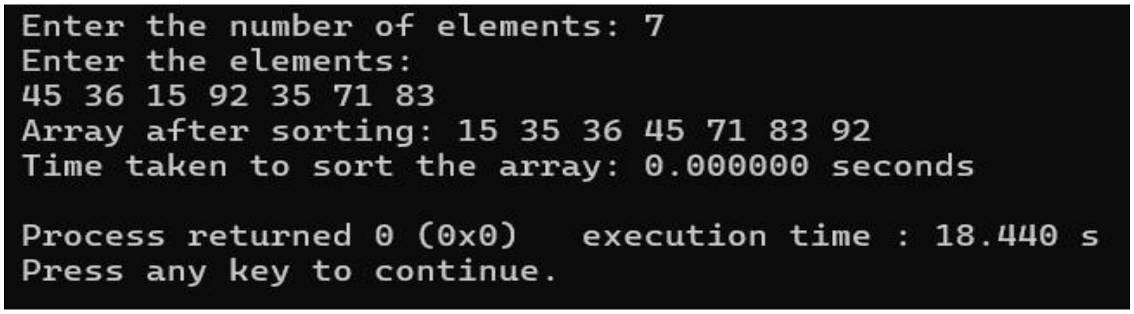
Program:

```
#include<stdio.h>

#include <time.h>


int partition(int a[], int low, int high);


void quickSort(int a[], int low, int high)

{

   if(low<high)

   {

      int mid = partition(a,low,high);

quickSort(a,low,mid-1);        quickSort(a,mid+1,high);

   }

}


int partition(int a[], int low, int high)

{

   int pivot = a[low];

int i = low+1;     int j

= high;     while(i<=j)

   {

      while(a[i]<=pivot)

         i++;

      while(a[j]>pivot)
```

```
            j--;
if(i<j)
        {
            int temp = a[i];
            a[i] = a[j];
a[j] = temp;
        }
    }
    int temp = a[low];
a[low] = a[j];     a[j] =
temp;     return j;
}

int main()
{     int
n;
    printf("Enter the number of elements: ");     scanf("%d", &n);
    int a[n];
    printf("Enter the elements: \n");
    for(int i=0; i<n; i++){
scanf("%d",&a[i]);
    }
        // Start measuring time
clock_t start, end;
    double cpu_time_used;
```

```
   start = clock();
quickSort(a,0,n-1);
end = clock();
   cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
   printf("Array after sorting: ");
for(int i=0; i<n; i++){
printf("%d ",a[i]);
   }
   printf("\nTime taken to sort the array: %f seconds\n", cpu_time_used);
   return 0;
}
```

Output: **Quick Sort**



```
Enter the number of elements: 7
Enter the elements:
45 36 15 92 35 71 83
Array after sorting: 15 35 36 45 71 83 92
Time taken to sort the array: 0.000000 seconds

Process returned 0 (0x0)   execution time : 18.440 s
Press any key to continue.
```

9. Heap Sort: Sort a given set of N integer elements using Heap Sort technique and compute its time taken.

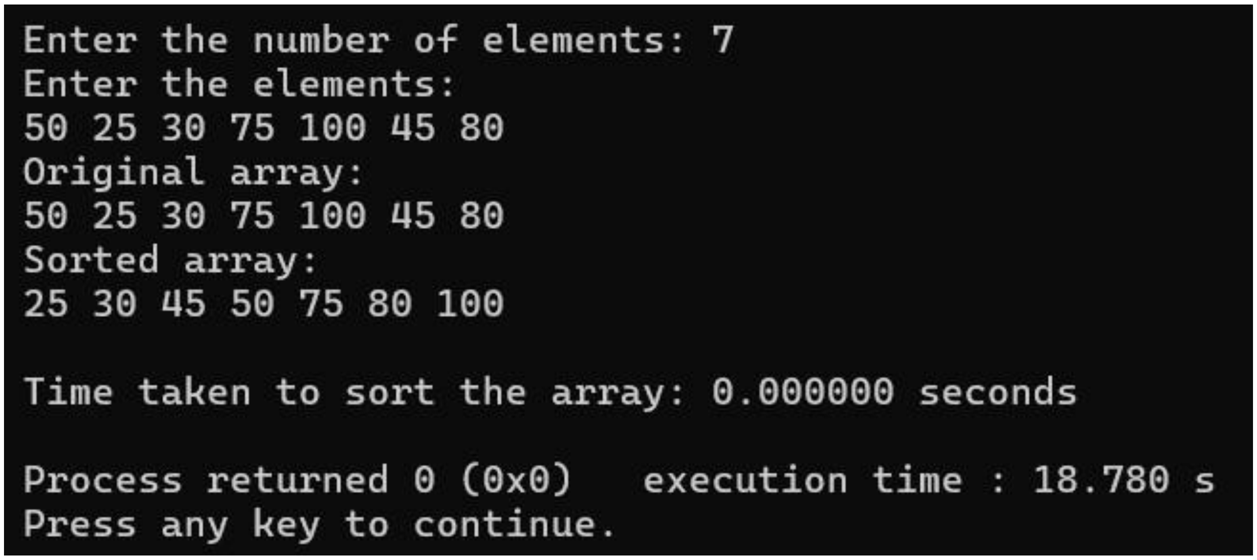
Program:

```
#include <stdio.h>

#include <time.h>


// Function to swap two integers

void swap(int *a, int *b) {     int

temp = *a;     *a = *b;

    *b = temp;

}


// Function to heapify a subtree rooted at index i

void heapify(int arr[], int n, int i) {     int

largest = i; // Initialize largest as root     int

left = 2 * i + 1; // left = 2*i + 1     int right =

2 * i + 2; // right = 2*i + 2


    // If left child is larger than root     if

(left < n && arr[left] > arr[largest])

        largest = left;


    // If right child is larger than largest so far

if (right < n && arr[right] > arr[largest])

        largest = right;
```

```
   // If largest is not root
if (largest != i) {
       swap(&arr[i], &arr[largest]);

       // Recursively heapify the affected sub-tree
heapify(arr, n, largest);
   }
}

// Main function to do heap sort
void heapSort(int arr[], int n) {
// Build heap (rearrange array)
   for (int i = n / 2 - 1; i >= 0; i--)
heapify(arr, n, i);

   // One by one extract an element from heap
   for (int i = n - 1; i >= 0; i--) {
// Move current root to end
swap(&arr[0], &arr[i]);

       // Call max heapify on the reduced heap
       heapify(arr, i, 0);
   }
}

// Utility function to print array void
printArray(int arr[], int n) {     for (int i
= 0; i < n; i++)        printf("%d ", arr[i]);
printf("\n");
}

// Driver program to test above functions int
main() {
   int n;
   printf("Enter the number of elements: ");
scanf("%d",&n);
   int arr[n];
   printf("Enter the elements: \n");
   for(int i=0; i<n; i++){
scanf("%d",&arr[i]);
   }

   printf("Original array: \n");
printArray(arr, n);

       // Start measuring time
clock_t start, end;
   double cpu_time_used;

   start = clock();
heapSort(arr, n);
end = clock();
   cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;     printf("Sorted array: \n");
printArray(arr, n);
```

```
    printf("\nTime taken to sort the array: %f seconds\n", cpu_time_used);


    return 0;
}
```

Output: Heap Sort



```
Enter the number of elements: 7
Enter the elements:
50 25 30 75 100 45 80
Original array:
50 25 30 75 100 45 80
Sorted array:
25 30 45 50 75 80 100

Time taken to sort the array: 0.000000 seconds

Process returned 0 (0x0)   execution time : 18.780 s
Press any key to continue.
```

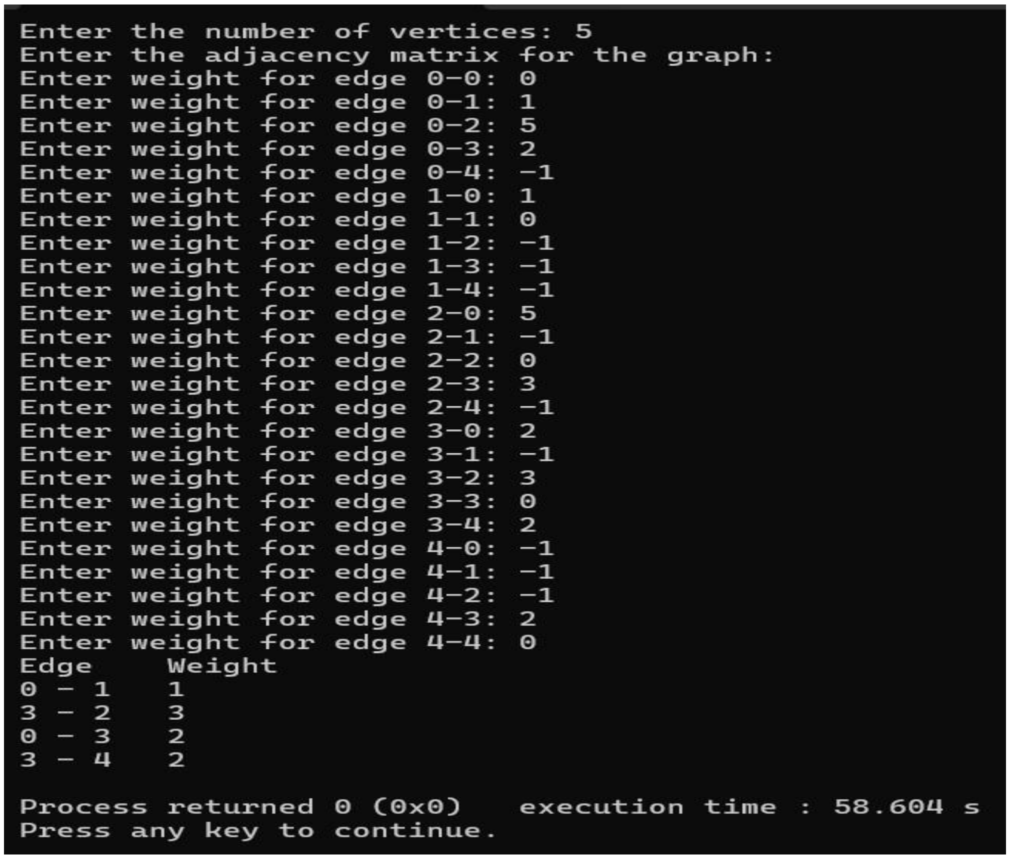
10. Prim’s Algorithm: Find Minimum Cost Spanning Tree of a given undirected graph using Prim’s Algorithm Program:

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

int V;

#define INF 99999 // Represents infinity or a very large number

// A utility function to find the vertex with the minimum key value
// from the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[]) {     int min =
INF, min_index;

   for (int v = 0; v < V; v++) {        if (mstSet[v]
== false && key[v] < min) {
           min = key[v];
min_index = v;
       }
   }

   return min_index;
}

// A utility function to print the constructed MST stored in parent[]
```

```
void printMST(int parent[], int graph[V][V]) {
printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency matrix
representation void primMST(int graph[V][V]) {    int parent[V]; // Array to store the
constructed MST    int key[V]; // Key values used to pick minimum weight edge in cut
bool mstSet[V]; // To represent set of vertices not yet included in MST

   // Initialize all keys as INFINITE
   for (int i = 0; i < V; i++) {
key[i] = INF;       mstSet[i]
= false;
   }

   // Always include first 1st vertex in MST.
   // Make key 0 so that this vertex is picked as first vertex.
   key[0] = 0;    parent[0] = -1; // First node is always
root of MST

   // The MST will have V vertices    for (int
count = 0; count < V - 1; count++) {
      // Pick the minimum key vertex from the set of vertices
      // not yet included in MST
int u = minKey(key, mstSet);
```

```
        // Add the picked vertex to the MST Set
mstSet[u] = true;

        // Update key value and parent index of the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not yet included in MST
        for (int v = 0; v < V; v++) {
            // graph[u][v] is non-zero only for adjacent vertices of u
            // mstSet[v] is false for vertices not yet included in MST
// Update the key only if graph[u][v] is smaller than key[v]            if
(graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;
key[v] = graph[u][v];
            }
        }
    }

    // print the constructed MST
printMST(parent, graph);
}

int main() {     printf("Enter the number of
vertices: ");     scanf("%d", &V);

    int graph[V][V];

    printf("Enter the adjacency matrix for the graph:\n");
    for (int i = 0; i < V; i++) {
for (int j = 0; j < V; j++) {
            printf("Enter weight for edge %d-%d: ", i, j);
scanf("%d", &graph[i][j]);

            // If there is no edge from i to j, represent it as -1
            if (graph[i][j] == -1) {
                graph[i][j] = INF; // Use INF to represent absence of edge
            }
        }
    }

    // Compute MST using Prim's algorithm
primMST(graph);

    return 0;
}
```

Output:



```
Enter the number of vertices: 5
Enter the adjacency matrix for the graph:
Enter weight for edge 0-0: 0
Enter weight for edge 0-1: 1
Enter weight for edge 0-2: 5
Enter weight for edge 0-3: 2
Enter weight for edge 0-4: -1
Enter weight for edge 1-0: 1
Enter weight for edge 1-1: 0
Enter weight for edge 1-2: -1
Enter weight for edge 1-3: -1
Enter weight for edge 1-4: -1
Enter weight for edge 2-0: 5
Enter weight for edge 2-1: -1
Enter weight for edge 2-2: 0
Enter weight for edge 2-3: 3
Enter weight for edge 2-4: -1
Enter weight for edge 3-0: 2
Enter weight for edge 3-1: -1
Enter weight for edge 3-2: 3
Enter weight for edge 3-3: 0
Enter weight for edge 3-4: 2
Enter weight for edge 4-0: -1
Enter weight for edge 4-1: -1
Enter weight for edge 4-2: -1
Enter weight for edge 4-3: 2
Enter weight for edge 4-4: 0
Edge    Weight
0 - 1   1
3 - 2   3
0 - 3   2
3 - 4   2

Process returned 0 (0x0)   execution time : 58.604 s
Press any key to continue.
```

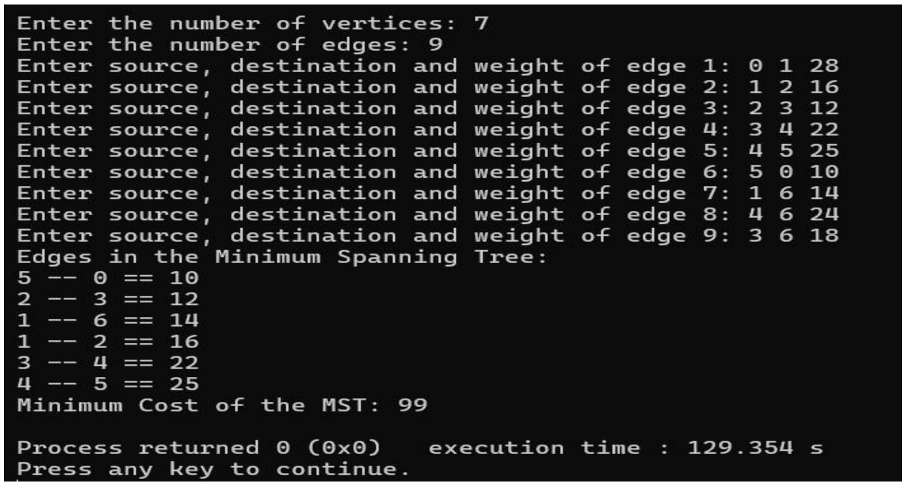
11. Kruskal’s Algorithm: Find minimum cost spanning tree of a given undirected graph using Kruskal’s algorithm

Program:

```
#include <stdio.h>

#include <stdlib.h>


// Structure to represent an edge in the graph

struct Edge {     int src, dest, weight;

};


// Function to sort edges based on their weights int

comparator(const void* p1, const void* p2)

{

    return ((struct Edge*)p1)->weight - ((struct Edge*)p2)->weight;

}


// Function to find the parent of a vertex int

findParent(int parent[], int v)

{

    if (parent[v] == v)

return v;

    return parent[v] = findParent(parent, parent[v]);

}


// Function to perform union of two sets void

unionSet(int u, int v, int parent[])

{
    parent[u] = v;
}

// Function to implement Kruskal's algorithm void
kruskalMST(int V, int E, struct Edge edges[])
{
    // Step 1: Sort all the edges in non-decreasing order of their weight
qsort(edges, E, sizeof(edges[0]), comparator);

    // Allocate memory for parent array     int
*parent = (int*) malloc(V * sizeof(int));

    // Initialize parent array
for (int v = 0; v < V; v++)
parent[v] = v;

    // Initialize variables to store minimum cost of MST
int minCost = 0;

    printf("Edges in the Minimum Spanning Tree:\n");

    // Step 2: Iterate through all sorted edges
    for (int i = 0; i < E; i++)
    {
        int rootSrc = findParent(parent, edges[i].src);
int rootDest = findParent(parent, edges[i].dest);
// If including this edge does not cause a cycle,
include it in the result and increment the index of

result for next edge

        if (rootSrc != rootDest)

        {

            printf("%d -- %d == %d\n", edges[i].src, edges[i].dest, edges[i].weight);

minCost += edges[i].weight;             unionSet(rootSrc, rootDest, parent);

        }

    }


    printf("Minimum Cost of the MST: %d\n", minCost);


    // Free dynamically allocated memory

    free(parent);

}


int main()

{    int V,

E;


    printf("Enter the number of vertices: ");

scanf("%d", &V);


    printf("Enter the number of edges: ");

scanf("%d", &E);


    // Allocate memory for edges array     struct Edge *edges = (struct

Edge*) malloc(E * sizeof(struct Edge));
```

```
    // Input edges
    for (int i = 0; i < E; i++)
    {
        printf("Enter source, destination and weight of edge %d: ", i + 1);
scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);
    }
    // Call Kruskal's algorithm function
kruskalMST(V, E, edges);

    // Free dynamically allocated memory
    free(edges);

    return 0;
}
```

Output:



```
Enter the number of vertices: 7
Enter the number of edges: 9
Enter source, destination and weight of edge 1: 0 1 28
Enter source, destination and weight of edge 2: 1 2 16
Enter source, destination and weight of edge 3: 2 3 12
Enter source, destination and weight of edge 4: 3 4 22
Enter source, destination and weight of edge 5: 4 5 25
Enter source, destination and weight of edge 6: 5 0 10
Enter source, destination and weight of edge 7: 1 6 14
Enter source, destination and weight of edge 8: 4 6 24
Enter source, destination and weight of edge 9: 3 6 18
Edges in the Minimum Spanning Tree:
5 -- 0 == 10
2 -- 3 == 12
1 -- 6 == 14
1 -- 2 == 16
3 -- 4 == 22
4 -- 5 == 25
Minimum Cost of the MST: 99

Process returned 0 (0x0)   execution time : 129.354 s
Press any key to continue.
```

12. Fractional Knapsack: Implement Fractional Knapsack using Greedy technique

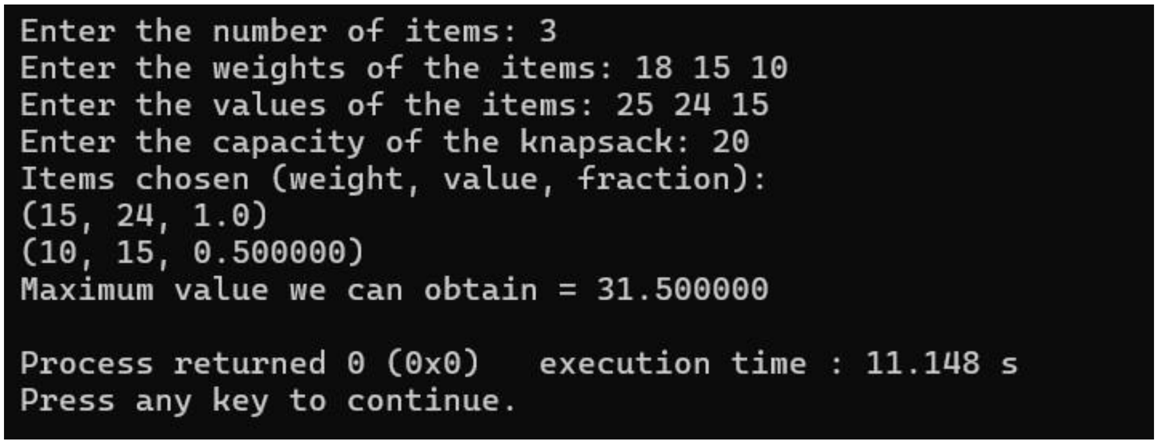
Program:

```
#include <stdio.h>


// Function to swap two elements void
swap(double* a, double* b) {     double
temp = *a;
   *a = *b;
   *b = temp;
}


// Function to sort items based on value per unit weight void sortItems(int
n, double valuePerWeight[], int weights[], int values[]) {
   for (int i = 0; i < n-1; i++) {
for (int j = i+1; j < n; j++) {
         if (valuePerWeight[i] < valuePerWeight[j]) {
swap(&valuePerWeight[i], &valuePerWeight[j]);                // Using
temp variable for weights and values swapping               int
tempWeight = weights[i];               weights[i] = weights[j];
weights[j] = tempWeight;
            int tempValue = values[i];
values[i] = values[j];               values[j]
= tempValue;
         }
      }
   }
```

```
}

    // Function to calculate maximum value we can get double
    fractionalKnapsack(int weights[], int values[], int n, int capacity) {     double
    valuePerWeight[n];
       for (int i = 0; i < n; i++) {
           valuePerWeight[i] = (double)values[i] / weights[i];
       }

       // Sort items by value per unit weight
    sortItems(n, valuePerWeight, weights, values);

       double totalValue = 0.0;

       printf("Items chosen (weight, value, fraction):\n");

       for (int i = 0; i < n; i++) {         if (capacity >=
    weights[i]) {            capacity -= weights[i];
    totalValue += values[i];            printf("(%d, %d,
    1.0)\n", weights[i], values[i]);
           } else {
               double fraction = (double)capacity / weights[i];
    totalValue += values[i] * fraction;            printf("(%d, %d,
    %lf)\n", weights[i], values[i], fraction);
               break;
           }
       }
       return totalValue;
    }

    int main() {     int
    n, capacity;

       printf("Enter the number of items: ");
    scanf("%d", &n);

       int weights[n];
    int values[n];

       printf("Enter the weights of the items: ");
       for (int i = 0; i < n; i++) {
           scanf("%d", &weights[i]);
       }

       printf("Enter the values of the items: ");
       for (int i = 0; i < n; i++) {
           scanf("%d", &values[i]);
       }

       printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);
```

```
    double maxValue = fractionalKnapsack(weights, values, n, capacity);     printf("Maximum
value we can obtain = %lf\n", maxValue);


    return 0;
}
```

Output: Fractional Knapsack



```
Enter the number of items: 3
Enter the weights of the items: 18 15 10
Enter the values of the items: 25 24 15
Enter the capacity of the knapsack: 20
Items chosen (weight, value, fraction):
(15, 24, 1.0)
(10, 15, 0.500000)
Maximum value we can obtain = 31.500000

Process returned 0 (0x0)   execution time : 11.148 s
Press any key to continue.
```

13. Dijkstra’s Algorithm: From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm

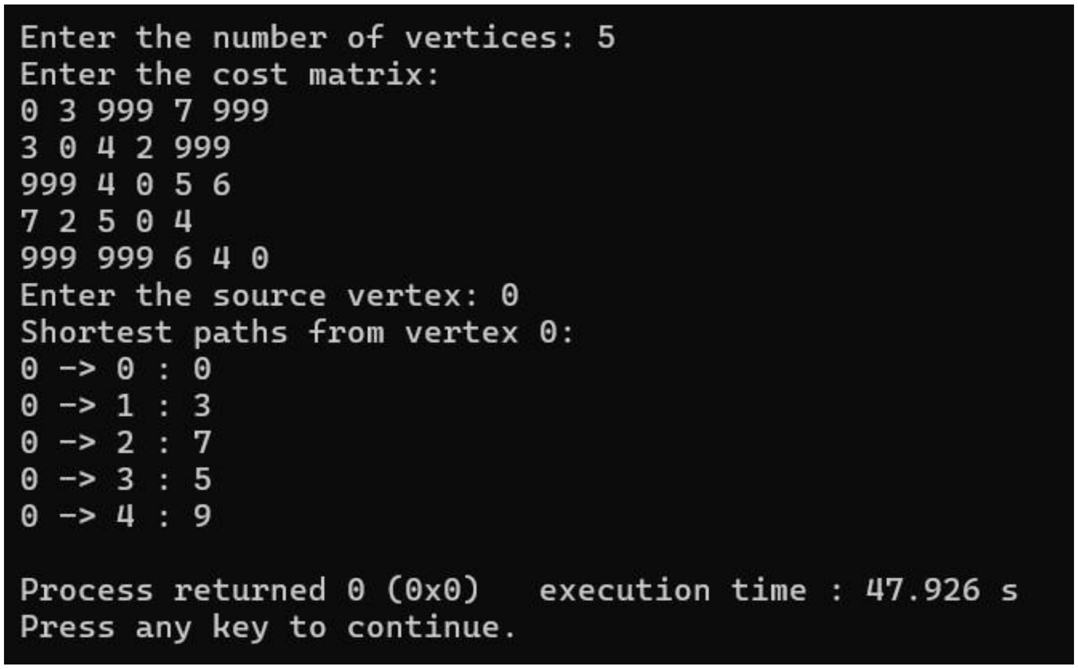
Program:

```
#include<stdio.h>


void dijkstra(int src, int n, int cost[10][10], int dest[10], int vis[10]) {     int
i, count, min, u;


   // Initialize arrays and variables
   for (i = 0; i < n; i++) {
      dest[i] = cost[src][i];  // Initialize distances from source to all vertices
      vis[i] = 0;  // Initialize all vertices as not visited
   }
   vis[src] = 1;  // Mark the source vertex as visited
dest[src] = 0;  // Distance from source to itself is 0     count
= 1;  // Initialize count of visited vertices


   while (count < n) {
min = 999;
      // Find the vertex with minimum distance which is not visited
      for (i = 0; i < n; i++) {            if
(dest[i] < min && !vis[i]) {
min = dest[i];
            u = i;
         }
      }
```

```
        vis[u] = 1;  // Mark the selected vertex as visited

        // Update distances of the adjacent vertices of the selected vertex
        for (i = 0; i < n; i++) {            if (!vis[i] &&
dest[u] + cost[u][i] < dest[i]) {                dest[i] =
dest[u] + cost[u][i];
            }
        }

        count++;
    }
}

int main() {
    int i, n, src;
    printf("Enter the number of vertices: ");
scanf("%d", &n);     int cost[10][10],
dest[10], vis[10];

    // Input the cost matrix
printf("Enter the cost matrix:\n");
    for (i = 0; i < n; i++) {         for
(int j = 0; j < n; j++) {
scanf("%d", &cost[i][j]);
        }
    }
```

```
   // Input the source vertex
printf("Enter the source vertex: ");
scanf("%d", &src);


   // Call Dijkstra's algorithm function
dijkstra(src, n, cost, dest, vis);


   // Output the shortest paths     printf("Shortest
paths from vertex %d:\n", src);
   for (i = 0; i < n; i++) {        printf("%d -> %d :
%d\n", src, i, dest[i]);     }


   return 0;
}
```

Output: **Dijkstra's Algorithm**



```
Enter the number of vertices: 5
Enter the cost matrix:
0 3 999 7 999
3 0 4 2 999
999 4 0 5 6
7 2 5 0 4
999 999 6 4 0
Enter the source vertex: 0
Shortest paths from vertex 0:
0 -> 0 : 0
0 -> 1 : 3
0 -> 2 : 7
0 -> 3 : 5
0 -> 4 : 9

Process returned 0 (0x0)   execution time : 47.926 s
Press any key to continue.
```