LAB ASSESSMENT 3

Name: Karan Sehgal

Registration Number: 22BCE3939

Course Name: Design and Analysis of Algorithms

Course Code: BCSE204P

Submitted to: : Prof. Gayatri P

Question 1:

Naïve string-matching algorithm

Problem Statement:

Given a text array, T [1.....n], of n character (i.e. length n) and a pattern array, P [1......m], of m characters (i.e. length m) such that m <= n.
The problem is to find an integer s, called valid shift where 0 ≤ s < n-m and T [s+1......s+m] = P [1......m].
If P occurs with shift s in T, then we say that s is a valid shift. Otherwise, we call s an invalid shift.
We say that pattern P occurs with shift s in text T if 0 ≤ s ≤ n-m and T [s+1......s+m] = P [1......m].
String matching problem is to find all valid shifts.
The item of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... z}.

Pseudo-Code:

Naive String Matching Algorithm

22BCE3939

```
function naive Stringmatch (text, pat)
textlength = length of text
pattern length = length of pattern
for i from 0 to textlength - pattern length:
    j = 0
    While j < patternlength:
        if text[i+j]!= pattern[j]:
            break
        j = j+1
    if j equals patternlength:
        print "pattern found at shift", i-1

function main()
read text from user
read pattern from user
print "Text", text
print "Pattern", pattern
print "Naive String Matching Algorithm:"
Call naivestring match( text, pat)
```

Source code:

```c
//22BCE3939
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void naiveStringMatch(char *text, char *pattern) {
    int textLength = strlen(text);
    int patternLength = strlen(pattern);

    for (int i = 0; i <= textLength - patternLength; i++) {
        int j;
        for (j = 0; j < patternLength; j++) {
            if (text[i + j] != pattern[j])
                break;
        }
        if (j == patternLength)
            printf("Pattern found at shift %d\n", i);
    }
}

int main() {
    char text[1000], pattern[100];

    printf("Enter the text: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // Removing trailing newline character

    printf("Enter the pattern: ");
    fgets(pattern, sizeof(pattern), stdin);
    pattern[strcspn(pattern, "\n")] = '\0'; // Removing trailing newline
character

    printf("Text: %s\n", text);
    printf("Pattern: %s\n", pattern);
    printf("Naive String Matching Algorithm:\n");
    naiveStringMatch(text, pattern);

    return 0;
}
```

Output :

```
"C:\Users\karan\Documents\(
Enter the text: 000010001010001
Enter the pattern: 0001
Text: 000010001010001
Pattern: 0001
Naive String Matching Algorithm:
Pattern found at shift 1
Pattern found at shift 5
Pattern found at shift 11

Process returned 0 (0x0)   execution time : 20.587 s
Press any key to continue.
```

```
"C:\Users\karan\Documents\(
Enter the text: ABEDACEFGDACE
Enter the pattern: ACE
Text: ABEDACEFGDACE
Pattern: ACE
Naive String Matching Algorithm:
Pattern found at shift 4
Pattern found at shift 10

Process returned 0 (0x0)   execution time : 11.669 s
Press any key to continue.
```

Question 2:

KMP string-matching algorithm

Pseudo-Code:

KMP String Matching Algorithm

22BCE3939

```
function KMP Search (pat, text):
  M = length of pat
  N = length of text
// create lps[] that will hold longest
  prefix suffix
// values for pattern
  lps[M] = new integer array
// preprocess the pattern
          [calculate lps array]

compute LPSarray (pat, M, lps)
  i = 0   // index for text[]
  j = 0   // index for pat[]
  while (N - i) >= (M - j):
       if pat[j] == text[i]:
          j = j + 1
          i = i + 1

    if j == M
       print "Found pattern at index", i-j
       j = lps[j-1]

// mismatch after j matches
```

```
else if i<N and pat[j] != text[i]:
                         // Do not match
    if j != 0              lps[0]...[j-1] character,
    j = lps[j-1]           they will match
                           anyway.
    else
       i = i+1

function computeLPSArray (pat, M, lps):

  len = 0
  lps[0] = 0
  // the loop calculates lps[i] for
     i = 1 to M-1

  i = 1
  while i < M:
    if pat[i] == pat[len]:
        len = len + 1
        lps[i] = len
        i = i + 1
    else:
        if len != 0:
            len = lps[len - 1]
        else:
            lps[i] = 0
            i = i + 1
```

```
// Driver code
function main():
read txt
read pattern
KMPSearch(pat, txt)
```

Source Code:

```
//22BCE3939
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Function to compute the LPS (Longest Proper Prefix which is also a Suffix)
array
void computeLPSArray(char *pattern, int M, int *lps) {
    int len = 0; // Length of the previous longest prefix suffix

    lps[0] = 0; // lps[0] is always 0

    int i = 1;
    while (i < M) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// Function to perform string matching using KMP algorithm
void KMPSearch(char *pattern, char *text) {
    int M = strlen(pattern);
    int N = strlen(text);

    int *lps = (int *)malloc(sizeof(int) * M);
    if (lps == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }

    computeLPSArray(pattern, M, lps);

    int i = 0; // Index for text[]
```

```c
    int j = 0; // Index for pattern[]

    while (i < N) {
        if (pattern[j] == text[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Pattern found at index %d\n", i - j);
            j = lps[j - 1];
        } else if (i < N && pattern[j] != text[i]) {
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }

    free(lps);
}

int main() {
    char text[1000], pattern[1000];

    printf("Enter the text: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // Remove newline character

    printf("Enter the pattern: ");
    fgets(pattern, sizeof(pattern), stdin);
    pattern[strcspn(pattern, "\n")] = '\0'; // Remove newline character

    printf("Text: %s\n", text);
    printf("Pattern: %s\n", pattern);
    printf("KMP String Matching Algorithm:\n");
    KMPSearch(pattern, text);

    return 0;
}
```

Output:

```
"C:\Users\karan\Documents\(
Enter the text: AABAACAADAABAAABAA
Enter the pattern: AABA
Text: AABAACAADAABAAABAA
Pattern: AABA
KMP String Matching Algorithm:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13

Process returned 0 (0x0)   execution time : 141.383 s
Press any key to continue.
```

```
"C:\Users\karan\Documents\(
Enter the text: 000010001010001
Enter the pattern: 0001
Text: 000010001010001
Pattern: 0001
KMP String Matching Algorithm:
Pattern found at index 1
Pattern found at index 5
Pattern found at index 11

Process returned 0 (0x0)   execution time : 106.399 s
Press any key to continue.
```

Question 3:

Rabin Karp string-matching algorithm

Problem Statement:

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character substring of text to be compared.
If the hash values are unequal, the algorithm will determine the hash value for
next M-character substring sequence.
If the hash values are equal, the algorithm will analyse the pattern and the Mcharacter
sequence (i.e) compare characters in pattern and M-character substring of text.
Therefore, character matching is only required when the hash values match.
Valid match (Valid Shift) – hash values of pattern and M-character substring
(ts) of text are equal and T [s+1......s+m] = P [1......m].

Pseudocode:

Rabin Karp String Matching Algorithm
22BCE3939

function
RabinKarp (txt, pat, q):
    txtlen = length(txt)
    patlen = length (pat)
    d = 256 // No. of characters in the input alphabet

    h = 1
    // Calculate h = (d^ (pattern length - 1)) % q
    for i from 0 to patlen - 2:
        h = (h *d) % q
    // calculate hash value of pattern and first window of text

    p = 0
    t = 0
    for i from 0 to patlen - 1:
        p = (d*p + pat[i]) % q
        t = (d* t + txt[i]) % q
    // Slide the pattern over text one by one
    for i from 0 to txtlen - patlen:
        // Check hash values of current window
        if p equals t:
            // Check for match character by character

```
for j from 0 to patlen-1:
    if txt[i+j] != pat[j]:
        break
    if j equals patlen:
        print "pattern found at shift", i-1
// Calculate hash value for next window of text: Removing leading digit, add trailing digit
if i < txtlen - patlen:
  t = (d*(t-txt[i]*h) +
          text[i+patlen]) % q
// Handle negative
  if t<0:
     t = t+q
function main():
 read text
 read pat
 q=13  // prime no. for hashing.
 call RabinKarp (text, pat, q)
```

Source Code:

```
//22BCE3939
#include <stdio.h>
#include <string.h>

#define d 256 // Number of characters in the input alphabet

void rabinKarp(char *text, char *pattern, int q) {
    int textLength = strlen(text);
    int patternLength = strlen(pattern);
    int i, j;
    int p = 0; // Hash value for pattern
    int t = 0; // Hash value for text
    int h = 1;

    // Calculate hash value of pattern and first window of text
    for (i = 0; i < patternLength - 1; i++)
        h = (h * d) % q;

    for (i = 0; i < patternLength; i++) {
        p = (d * p + pattern[i]) % q;
        t = (d * t + text[i]) % q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= textLength - patternLength; i++) {
        // Check hash values of current window of text and pattern
        if (p == t) {
            // Check for match character by character
            for (j = 0; j < patternLength; j++) {
                if (text[i + j] != pattern[j])
                    break;
            }
            if (j == patternLength)
                printf("Pattern found at shifts %d\n", i);
        }

        // Calculate hash value for next window of text: Remove leading digit,
add trailing digit
        if (i < textLength - patternLength) {
            t = (d * (t - text[i] * h) + text[i + patternLength]) % q;

            // Handle negative hash value
            if (t < 0)
```

```c
                t = (t + q);
        }
    }
}

int main() {
    char text[1000];
    char pattern[1000];
    int q;

    printf("Enter the text: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // Remove newline character

    printf("Enter the pattern: ");
    fgets(pattern, sizeof(pattern), stdin);
    pattern[strcspn(pattern, "\n")] = '\0'; // Remove newline character

    printf("Enter the prime number for hashing: ");
    scanf("%d", &q);

    printf("Text: %s\n", text);
    printf("Pattern: %s\n", pattern);
    printf("Rabin-Karp String Matching Algorithm:\n");
    rabinKarp(text, pattern, q);

    return 0;
}
```

Output:

```
"C:\Users\karan\Documents\(
Enter the text: AABAACAADAABAAABAA
Enter the pattern: AABA
Enter the prime number for hashing: 13
Text: AABAACAADAABAAABAA
Pattern: AABA
Rabin-Karp String Matching Algorithm:
Pattern found at shifts 0
Pattern found at shifts 9
Pattern found at shifts 13

Process returned 0 (0x0)   execution time : 21.164 s
Press any key to continue.
```

```
"C:\Users\karan\Documents\(
Enter the text: 000010001010001
Enter the pattern: 0001
Enter the prime number for hashing: 11
Text: 000010001010001
Pattern: 0001
Rabin-Karp String Matching Algorithm:
Pattern found at shifts 1
Pattern found at shifts 5
Pattern found at shifts 11

Process returned 0 (0x0)   execution time : 20.090 s
Press any key to continue.
```

Question 4:

Bellman Ford shortest path algorithm

Problem Statement:

Bellman ford algorithm is a single-source shortest path algorithm.
Computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.

• It is capable of handling graphs in which some of the edge weights are negative numbers.

• If the weighted graph contains the negative weight values, then the Dijkstra's algorithm does not confirm whether it produces the correct answer or not.

• Bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.

• Weight of edges can represent everything in real world. Example: Consider
a graph simulating behaviour of a molecule in a chemical reaction (i.e.) which paths it can take during reaction. Weights represents energy absorbed or released in the transition. We represent released energy with +ve weights and absorbed energy with -ve.

• Negative cycles can also be detected using Bellman Ford algorithm.

Pseudocode:

Bellman Ford shortest path Algo
22BCE3939

```
function CreateGraph (V, E):
    graph = new graph
    graph.V = V
    graph.E = E
    graph.edge = new Edge[E]
    return graph

function printArr (dist, n):
    print "Vertex Dist from source"
    for i from 0 to n-1:
        print i, dist[i]

function Bellmanford (graph, src):
    V = graph.V
    E = graph.E
    dist = new array of size V filled
           with INF values
    dist[src] = 0
    // Relax all edges |V|-1 times
    for i from 1 to V-1:
        for j from 0 to E-1:
```

```
u = graph.edge [j] . src
v = graph. edge [j] . dest
weight = graph . edge [j]. weight
if dist [u] != INFINITE and
   dist [u] + weight < dist [v]
   dist [v] = dist [u] + weight.

//Check for negative weight cycles
for i from 0 to E-1:
 u = graph . edge [i] . src
 V= graph . edge (i) . dest
 weight = graph. edge [i]. weight
 if dist [u] != INFINITE and
          dist [u] + weight < dist [v]:
 print "Graph contains negative
   weight cycle"
 return
print Arr (dist, V)
```

```
function main ()
  V = 5
  E = 8
  graph = creategraph (V, E)
 // Add edges to the graph
  graph. edge [0] src = 0
  graph. edge [0]. dest = 1
  graph. edge [0]. weight = 3
   // Add another edge similarly :-

 //Function call
 BellmanFord (graph , 0)
```

Source Code:

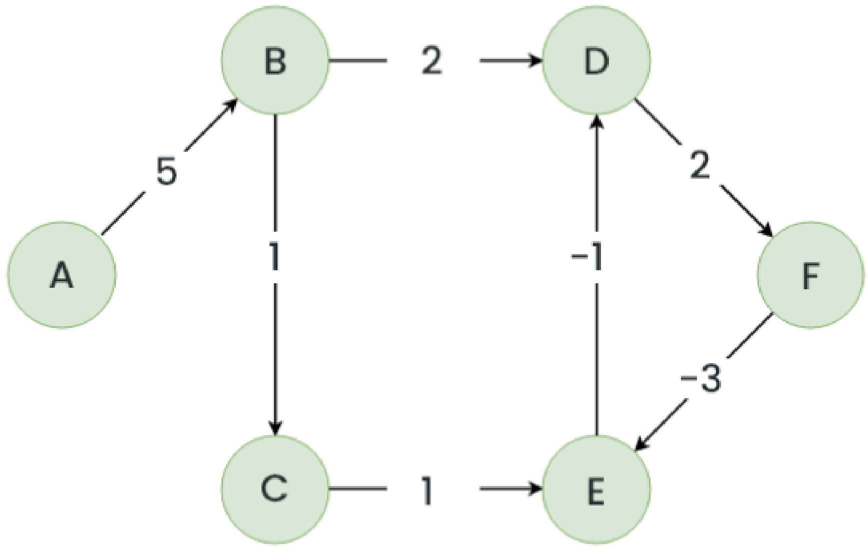
```
// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
//22BCE3939
#include <bits/stdc++.h>
using namespace std;

// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph {
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src
// to all other vertices using Bellman-Ford algorithm. The
// function also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
```

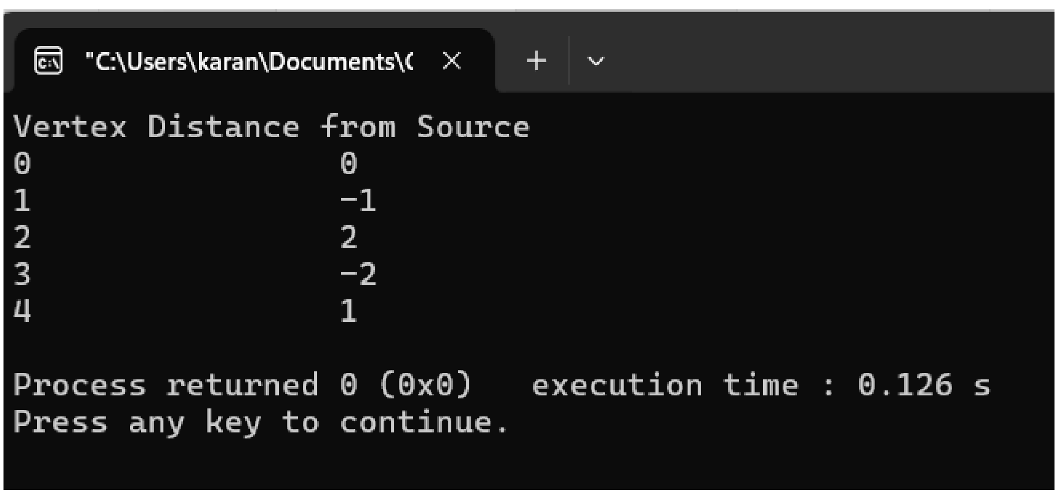
```
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other
    // vertices as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple
    // shortest path from src to any other vertex can have
    // at-most |V| - 1 edges
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX
                && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles. The above
    // step guarantees shortest distances if graph doesn't
    // contain negative weight cycle. If we get a shorter
    // path, then there is a cycle.
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX
            && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle");
            return; // If negative cycle is detected, simply
                    // return
        }
    }

    printArr(dist, V);

    return;
}
```

```c
// Driver's code
int main()
{
    /* Let us create the graph given in above example */
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;

    // Function call
    BellmanFord(graph, 0);

    return 0;
```

```
}
```

Output:






```
Vertex Distance from Source
0               0
1               -1
2               2
3               -2
4               1

Process returned 0 (0x0)   execution time : 0.126 s
Press any key to continue.
```

Question 5:

Floyd Warshall shortest path algorithm

Problem Statement:

Floyd Warshall Algorithm is for solving the All-Pairs Shortest Path problem.
The problem is to find shortest distances between every pair of vertices in a given weighted directed Graph.

Pseudo-code:

Floyd Warshall shortest path Algo

22BCE3939

```
function floydWarshall (dist[][]):
  V = number of vertices in the
      graph
  // Initialize dist[][] with input
  graph's distances
  for i from 0 to V-1 :
      for j from 0 to V-1 :
          if i equals j :
              dist[i][j] = 0
          else if graph[i][j] != INF :
              dist[i][j] = graph[i][j]
          else :
              dist[i][j] = INF
// Main loop :-
  for k from 0 to V-1 :
      for i from 0 to V-1 :
          for j from 0 to V-1 :
  // if vertex k is on the shortest path
     from i to j, update dist[i][j]
  if dist[i][k] + dist[k][j] < dist[i][j]
      dist[i][j] = dist[i][k] +
                   dist[k][j]

print Solution (dist)
```

```
function : print_Solution ( dist[][] ):
print "The following matrix shows the shortest distances between every pair of vertices"
   for i from 0 to V-1 :
      for j from 0 to V-1 :
         if   dist[i][j] equals INF :
               print "INF"
         else
               print dist[i][j]
         print newline

function main():
   Initialize graph with the weighted edges.
   Call FloydWarshall (graph)
```

Source code:

```c
// C Program for Floyd Warshall Algorithm
//22BCE3939
#include <stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough
value. This value will be used
for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{
    int i, j, k;

    for (k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++) {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++) {
                // If vertex k is on the shortest path from
                // i to j, then update the value of
                // dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
```

```
    printf(
        "The following matrix shows the shortest distances"
        " between every pair of vertices \n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver's code
int main()
{
    int graph[V][V] = { { 0, 5, INF, 10 },
                        { INF, 0, 3, INF },
                        { INF, INF, 0, 1 },
                        { INF, INF, INF, 0 } };

    // Function call
    floydWarshall(graph);
    return 0;
}
```

Output:

Example Graph:

```
        10
   (0)------->(3)
    |         /|\
  5 |          |
    |          | 1
   \|/         |
   (1)------->(2)
        3
```

Input: adjacency matrix representation of above graph

```
graph[][] = { {0,   5,  INF, 10},
              {INF,  0,  3,  INF},
              {INF, INF, 0,   1},
              {INF, INF, INF, 0} }
```

```
"C:\Users\karan\Documents\(
The following matrix shows the shortest distances between every pair of vertices
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0

Process returned 0 (0x0)   execution time : 0.064 s
Press any key to continue.
```

Question 6:

Ford – Fulkerson maximum flow algorithm

Problem statement:

To find the maximum flow, the ford Fulkerson algorithm repeatedly finds augmenting paths through the residual graph and augments the flow until no more augmenting paths can be found.

Pseudocode:

Ford – Fulkerson maxᵐ flow Algorithm
22BCE3939

```
function bfs (rGraph, s, t, parent):
 // Create a visited array and
  mark all vertices as not
  visited
 vist[] = new boolean array of size V
 for each vertex V:
     visited [v] = false
 // Create a queue, enqueue source
 vertex, and mark it as visited
  q = new Queue
  q.push(s)
  vist [s] = True
  parent [s] = -1
 // standard BFS loop
 while q is not empty:
     u = q.front()
     q.pop()
 for each vertex v adjacent to u:
    if not visited [v] and rGraph [u][v] > 0:
       if v == t:
          parent [v] = u
          return true
      q.push (v)
      parent [v] = u
      visited [v] = True
return false
```

function fordFulkerson (graph, s, t):

// Create a residual graph and fill it with given capacities

rGraph [][] = new 2D array of size V x V

for each edge (u, v) in graph :

rGraph [u][v] = graph [u][v]

// Initialize parent array

parent [] = new array of size V

max flow = 0 // Initialise max flow = 0

// Augment the flow while there is a path from source to sink while there exists a path from s to t in rGraph :

// Find max residual capacity along path filled by BFS

path_flow = INT_MAX

for each vertex v in the path from t to s:

u = parent[v]

path_flow = min (path_flow, rGraph[u][v])

```
// update residual capacities and
reverse edges along the path for
each vertex v in the path from t to s:
    u = parent[v]
    rGraph[u][v] -= path_flow
    rGraph[v][u] += path_flow
    // Add path flow to overall flow
    max_flow += path_flow
    return max_flow

function main():
    Create a graph
    graph[][]
    // Find the maxm flow from
    source (0) to sink(5)
    max_flow = fordFulkerson(graph, 0, 5)
    print "MAX POSSIBLE FLOW", max_flow
```

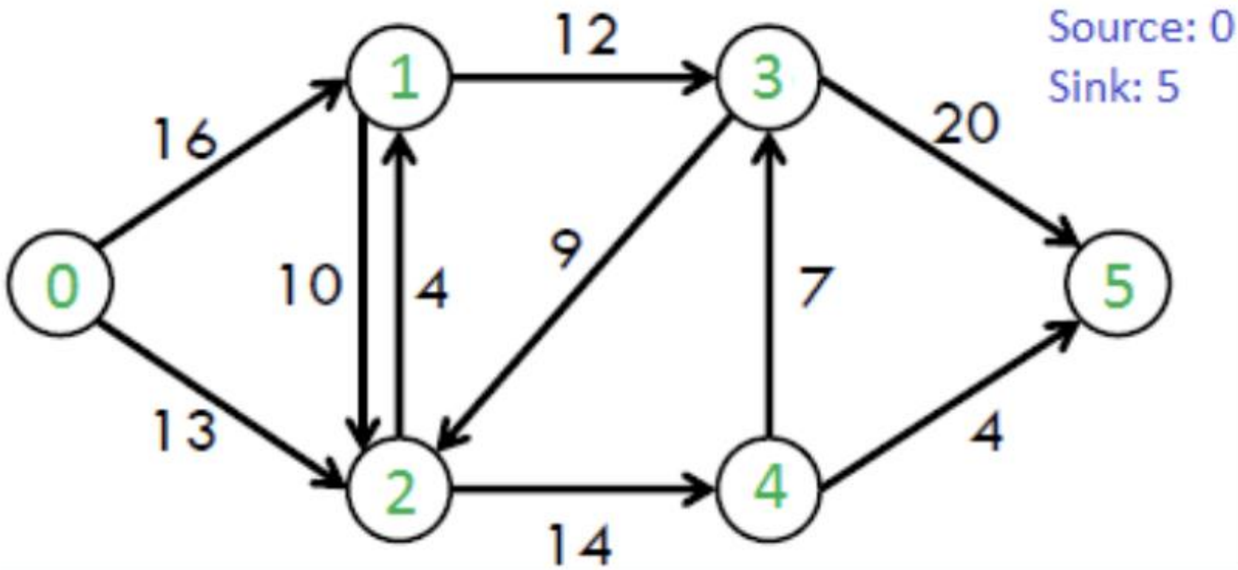
Source Code:

```
// C++ program for implementation of Ford Fulkerson
// algorithm
//22BCE3939
#include <iostream>
#include <limits.h>
#include <queue>
#include <string.h>
using namespace std;

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink
't' in residual graph. Also fills parent[] to store the
path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not
    // visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

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

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

        for (int v = 0; v < V; 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, so
    // return false
    return false;
}

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


    int rGraph[V]
            [V];
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to
                // store path

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

    // Augment the flow while there is path from source to
    // sink
    while (bfs(rGraph, 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()
{
    // Let us create a graph shown in the above example
    int graph[V][V]
        = { { 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 } };

    cout << "The maximum possible flow is "
        << fordFulkerson(graph, 0, 5);

    return 0;
}
```

Output:




```
"C:\Users\karan\Documents\(
The maximum possible flow is 23
Process returned 0 (0x0)   execution time : 0.079 s
Press any key to continue.
```

Question 7:

Edmond – Karp maximum flow algorithm

Psuedo Code:

Edmonds - Karp maximum flow Algo

22BCE3939

function bfs (rGraph, s, t, parent):

Initialize an array visited of size V and set all elements to false.

Create an empty queue q

Push the source vertex s to the queue

Mark vertex s as visited

Set parent [s] to −1

While q is not empty:

Dequeue a vertex u from front of the queue

for each vertex v from 0 to V−1:

if v is not visited and there is residual cap from u to v:

Mark vertex v as visited

Set parent [v] to u

Enqueue vertex v to the queue

return true if vertex t is visited, otherwise false.

```
function edmondsKarp (graph, s, t):
Initialise rGraph as copy of given graph
Initialise an array parent of size V
 to store augmenting path.
Inialize maxflow to 0

while bfs (rGraph, s, t, parent) returns true:
    Set pathflow to positive infinity
    Iterate from vertex t back to s
        using parent[]:
          Update pathflow to the
          minimum ( pathflow, residual cap of
                 edge from parent[v] to v)

    Iterate from vertex t back to s
        using parent[]:
          Update the residual capacities of
          forward and backward edge
          along the augmenting path

    Add pathflow to maxflow

return maxflow
```

function main ()

Initialize the flow network graph as a 2D array with given capacities.

Set source to 0 and sink to 5

Call edmondsKarp (graph, source, sink)

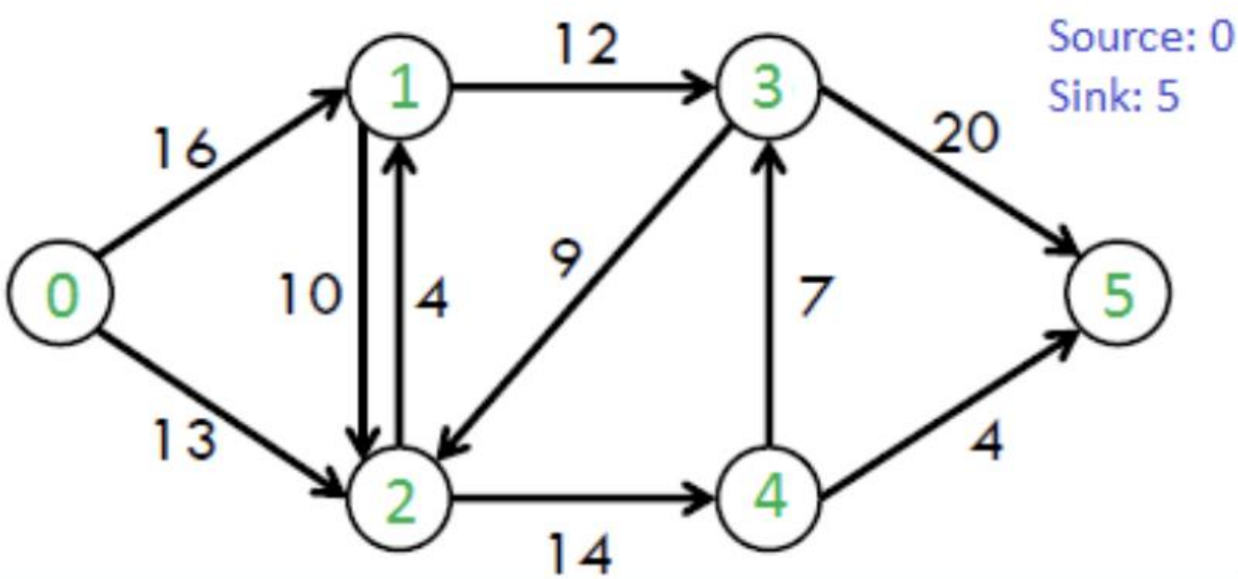
print maximum flow

Source Code:

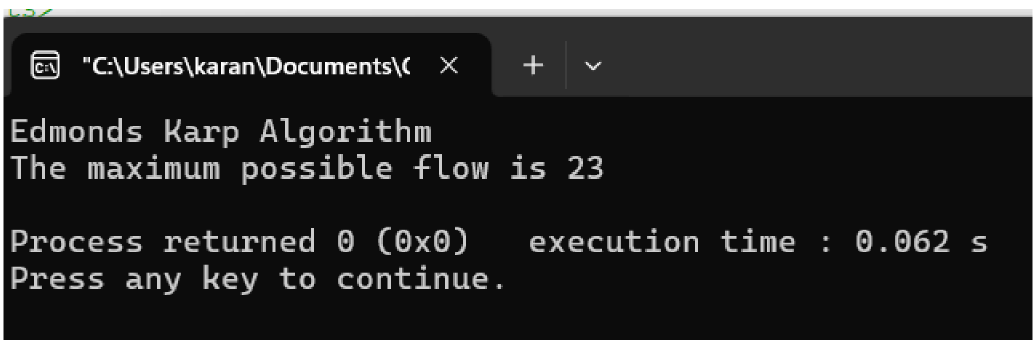
```
#include <iostream>
#include <queue>
#include <cstring>
#include <climits>
//22BCE3939
using namespace std;

#define V 6 // Number of vertices in the given graph

// Returns true if there is a path from source 's' to sink 't' in residual graph.
// Also fills parent[] to store the path.
bool bfs(int rGraph[V][V], int s, int t, int parent[]) {
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    queue<int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

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

        for (int v = 0; v < V; v++) {
            if (visited[v] == false && rGraph[u][v] > 0) {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }
    return (visited[t] == true);
}

// Returns the maximum flow from s to t in the given graph
int edmondsKarp(int graph[V][V], int s, int t) {
    int u, v;

    int rGraph[V][V];
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];
```

```
    int parent[V];
    int max_flow = 0;

    while (bfs(rGraph, 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]);
        }

        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
        max_flow += path_flow;
    }
    return max_flow;
}

int main() {
    int graph[V][V] = { {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} };

    int source = 0, sink = 5;
    cout<<"Edmonds Karp Algorithm"<<endl;
    cout << "The maximum possible flow is " << edmondsKarp(graph, source, sink)
<< endl;
    return 0;
}
```

Output:



Source: 0
Sink: 5
0
1
2
3
4
5
16
13
10
4
12
9
14
7
20
4



"C:\Users\karan\Documents\(
Edmonds Karp Algorithm
The maximum possible flow is 23
Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.