NAME :- Rahi Agarwal
Batch :- F4
Enroll No. :- 9921103145

QUES-1)

```
#include <limits.h>
#include <string.h>

#include <iostream>
#include <queue>
using namespace std;

#define V 6

// Using BFS as a searching algorithm
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);
}

// Applying fordfulkerson algorithm
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];
  int max_flow = 0;

  // Updating the residual values of edges
  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;
    }

    // Adding the path flows
    max_flow += path_flow;
  }

  return max_flow;
}

int main() {
```

```
 int graph[V][V] =

        {{0, 11, 12, 0, 0, 0},

         {0, 0, 0, 12, 0, 0},

         {0, 1, 0, 0, 11, 0},

         {0, 0, 0, 0, 0, 19},

         {0, 0, 0, 7, 0, 4},

         {0, 0, 0, 0, 0, 0}};


 cout << "Max Flow: " << fordFulkerson(graph, 0, 5) << endl;

}
```

OUTPUT-

```
Output

/tmp/rklIRe33Lz.o
Max Flow: 23
```

QUES-2)

```
#include<cstdio>

#include<queue>

#include<cstring>

#include<vector>

#include<iostream>

using namespace std;

int c[10][10];

int flowPassed[10][10];

vector<int> g[10];

int parList[10];

int currentPathC[10];

int bfs(int sNode, int eNode)//breadth first search

{

  memset(parList, -1, sizeof(parList));

  memset(currentPathC, 0, sizeof(currentPathC));

  queue<int> q;//declare queue vector

  q.push(sNode);

  parList[sNode] = -1;//initialize parlist's source node

  currentPathC[sNode] = 999;//initialize currentpath's source node

  while(!q.empty())// if q is not empty

  {

    int currNode = q.front();

    q.pop();

    for(int i=0; i<g[currNode].size(); i++)
```

```
    {
      int to = g[currNode][i];
      if(parList[to] == -1)
      {
        if(c[currNode][to] - flowPassed[currNode][to] > 0)
        {
          parList[to] = currNode;
          currentPathC[to] = min(currentPathC[currNode],
          c[currNode][to] - flowPassed[currNode][to]);
          if(to == eNode)
          {
            return currentPathC[eNode];
          }
          q.push(to);
        }
      }
    }
  }
  return 0;
}
int edmondsKarp(int sNode, int eNode)
{
  int maxFlow = 0;
  while(true)
  {
```

```
    int flow = bfs(sNode, eNode);
    if (flow == 0)
    {
      break;
    }
    maxFlow += flow;
    int currNode = eNode;
    while(currNode != sNode)
    {
      int prevNode = parList[currNode];
      flowPassed[prevNode][currNode] += flow;
      flowPassed[currNode][prevNode] -= flow;
      currNode = prevNode;
    }
  }
return maxFlow;
}
int main()
{
  int nodCount, edCount;
  cout<<"enter the number of nodes and edges\n";
  cin>>nodCount>>edCount;
  int source, sink;
  cout<<"enter the source and sink\n";
  cin>>source>>sink;
```

```
  for(int ed = 0; ed < edCount; ed++)
  {
    cout<<"enter the start and end vertex along with capacity\n";
    int from, to, cap;
    cin>>from>>to>>cap;
    c[from][to] = cap;
    g[from].push_back(to);
    g[to].push_back(from);
  }
  int maxFlow = edmondsKarp(source, sink);
  cout<<endl<<endl<<"Max Flow is:"<<maxFlow<<endl;
}
```

```
enter the number of nodes and edges
7
11
enter the source and sink
0
6
enter the start and end vertex along with capacity
0
1
3
enter the start and end vertex along with capacity
1
2
2
enter the start and end vertex along with capacity
0
2
5
enter the start and end vertex along with capacity
1
3
1
enter the start and end vertex along with capacity
2
3
3
enter the start and end vertex along with capacity
```

```
enter the start and end vertex along with capacity
3
4
1
enter the start and end vertex along with capacity
4
6
7
enter the start and end vertex along with capacity
3
6
4
enter the start and end vertex along with capacity
0
5
8
enter the start and end vertex along with capacity
5
3
2
Max Flow is:6
```

QUES-3)

```
#include <iostream>

#define M 6

#define N 6

using namespace std;


bool bipartiteGraph[M][N] = {

  {0, 1, 1, 0, 0, 0},

  {0, 0, 0, 0, 0, 0},

  {1, 0, 0, 1, 0, 0},

  {0, 0, 1, 0, 0, 0},

  {0, 0, 1, 1, 0, 0},

  {0, 0, 0, 0, 0, 1}

};


bool bipartiteMatch(int u, bool visited[], int assign[]) {

  for (int v = 0; v < N; v++) {    //for all jobs 0 to N-1

    if (bipartiteGraph[u][v] && !visited[v]) {

      visited[v] = true;

      if (assign[v] < 0 || bipartiteMatch(assign[v], visited, assign)) {

        assign[v] = u;

        return true;

      }

    }

  }
```

```
    return false;
}


int maxMatch() {
    int assign[N];
    for(int i = 0; i<N; i++)
        assign[i] = -1;
    int jobCount = 0;


    for (int u = 0; u < M; u++) {
        bool visited[N];
        for(int i = 0; i<N; i++)
            visited[i] = false;
        if (bipartiteMatch(u, visited, assign))
            jobCount++;
    }
    return jobCount;
}


int main() {
    cout << "Maximum number of applicants matching for job: " << maxMatch();
}
```

OUTPUT-

```
/tmp/rklIRe33Lz.o
Maximum number of applicants matching for job: 5
```