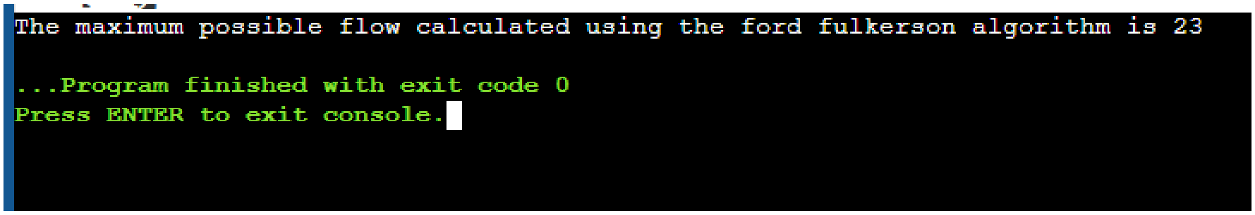
Nitin Chaudhary
9921103163
F7
APS Lab Week 9

Ans 1(a)-
Code-

```
#include <iostream>
#include <limits.h>
#include <queue>
#include <string.h>
using namespace std;
#define V 6

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) {
				if (v == t) {
					parent[v] = u;
					return true;
				}
				q.push(v);
				parent[v] = u;
				visited[v] = true;
			}
		}
```

```
        }
        return false;
}

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;
        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, 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 << "The maximum possible flow calculated using the ford fulkerson
algorithm is "<< fordFulkerson(graph, 0, 5);

        return 0;
}
```

Output



```
The maximum possible flow calculated using the ford fulkerson algorithm is 23

...Program finished with exit code 0
Press ENTER to exit console.
```

Ans 1(b)-

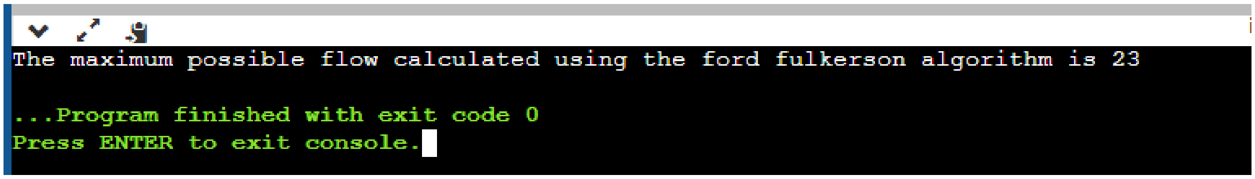
Code-

```
#include <iostream>
#include <limits.h>
#include <queue>
#include <string.h>
using namespace std;
#define V 6

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) {
				if (v == t) {
					parent[v] = u;
					return true;
				}
				q.push(v);
				parent[v] = u;
				visited[v] = true;
```

```
            }
        }
    }
    return false;
}

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;
    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, 11, 12, 0, 0, 0 }, { 0, 0, 1, 12, 0, 0 },
            { 0, 1, 0, 0, 11, 0 }, { 0, 12, 0, 0, 7, 19 },
            { 0, 0, 11, 7, 0, 4 }, { 0, 0, 0, 19, 4, 0 } };
    cout << "The maximum possible flow calculated using the ford fulkerson
algorithm is "<< fordFulkerson(graph, 0, 5);
```

```
        return 0;
}
```

Output-



```
The maximum possible flow calculated using the ford fulkerson algorithm is 23

...Program finished with exit code 0
Press ENTER to exit console.
```

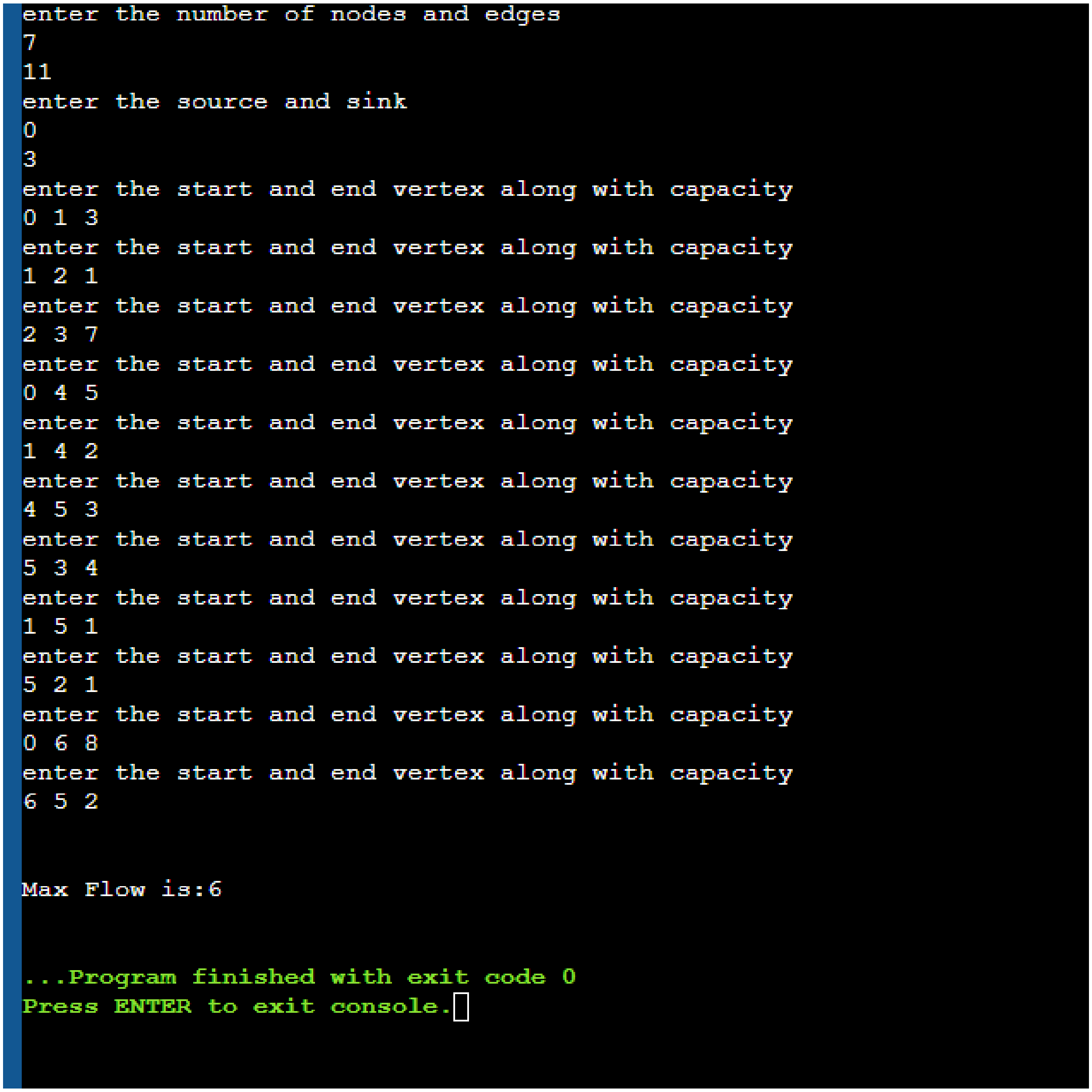
Ans 2-

(a)

```
#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)
{
  memset(parList, -1, sizeof(parList));
  memset(currentPathC, 0, sizeof(currentPathC));
  queue<int> q;
  q.push(sNode);
  parList[sNode] = -1;
  currentPathC[sNode] = 999;
  while(!q.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;
}
```

Output-

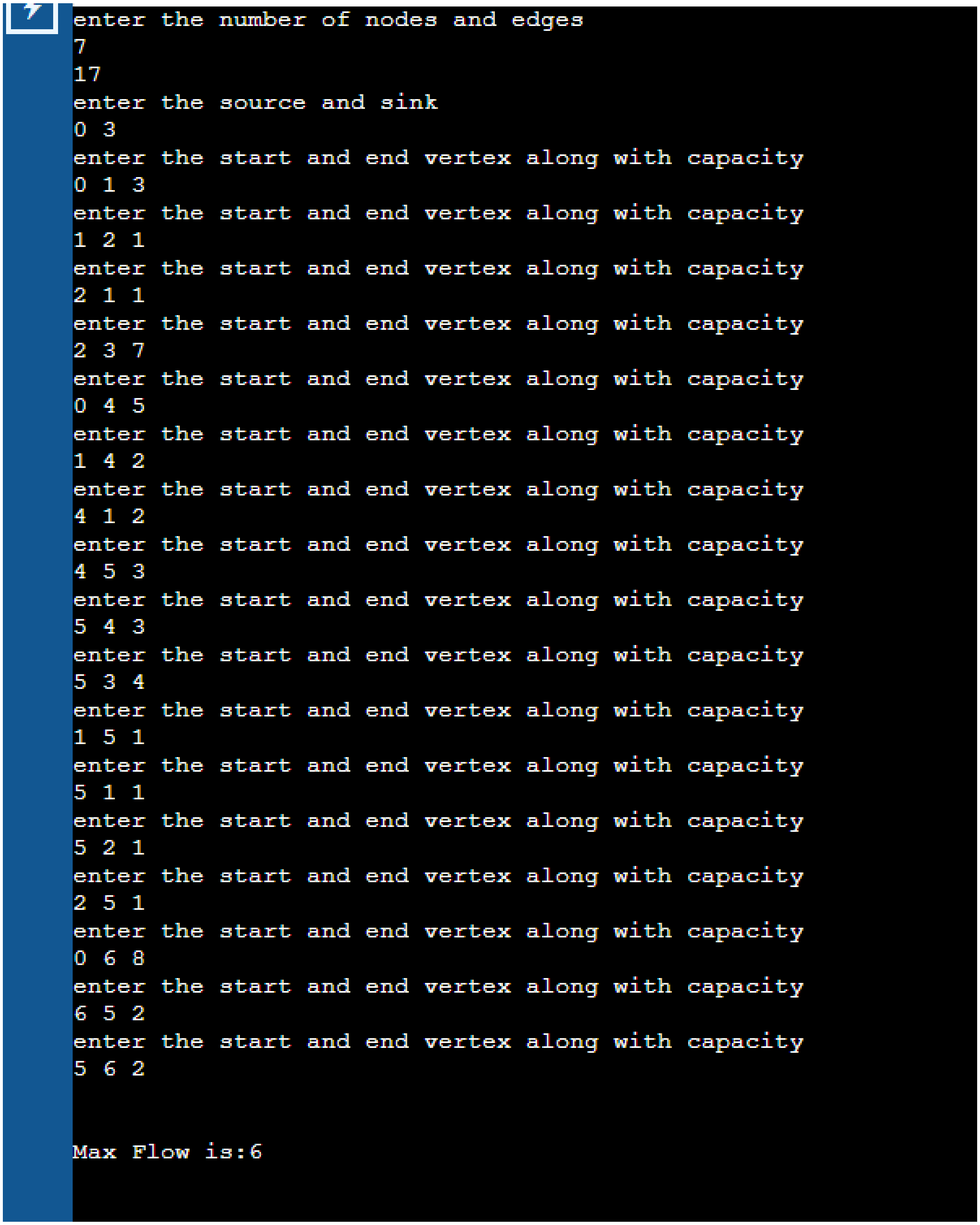


```
enter the number of nodes and edges
7
11
enter the source and sink
0
3
enter the start and end vertex along with capacity
0 1 3
enter the start and end vertex along with capacity
1 2 1
enter the start and end vertex along with capacity
2 3 7
enter the start and end vertex along with capacity
0 4 5
enter the start and end vertex along with capacity
1 4 2
enter the start and end vertex along with capacity
4 5 3
enter the start and end vertex along with capacity
5 3 4
enter the start and end vertex along with capacity
1 5 1
enter the start and end vertex along with capacity
5 2 1
enter the start and end vertex along with capacity
0 6 8
enter the start and end vertex along with capacity
6 5 2


Max Flow is:6


...Program finished with exit code 0
Press ENTER to exit console.
```

Ans 2-(b)-

```
#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)
{
  memset(parList, -1, sizeof(parList));
  memset(currentPathC, 0, sizeof(currentPathC));
  queue<int> q;
  q.push(sNode);
  parList[sNode] = -1;
  currentPathC[sNode] = 999;
  while(!q.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;
}
```



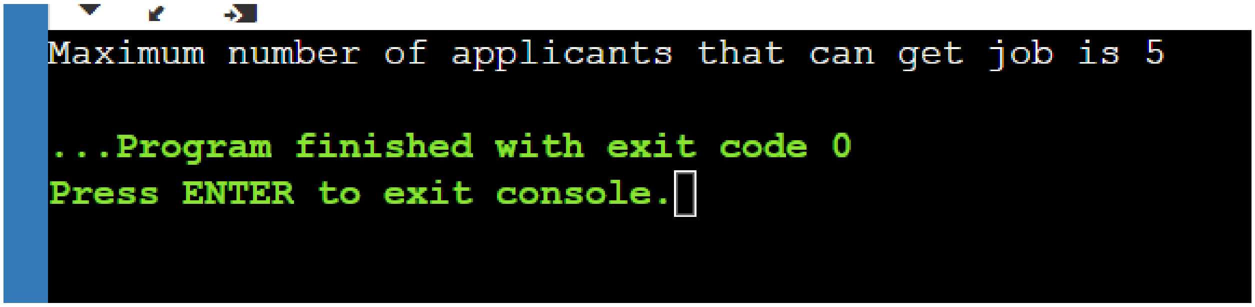
```
enter the number of nodes and edges
7
17
enter the source and sink
0 3
enter the start and end vertex along with capacity
0 1 3
enter the start and end vertex along with capacity
1 2 1
enter the start and end vertex along with capacity
2 1 1
enter the start and end vertex along with capacity
2 3 7
enter the start and end vertex along with capacity
0 4 5
enter the start and end vertex along with capacity
1 4 2
enter the start and end vertex along with capacity
4 1 2
enter the start and end vertex along with capacity
4 5 3
enter the start and end vertex along with capacity
5 4 3
enter the start and end vertex along with capacity
5 3 4
enter the start and end vertex along with capacity
1 5 1
enter the start and end vertex along with capacity
5 1 1
enter the start and end vertex along with capacity
5 2 1
enter the start and end vertex along with capacity
2 5 1
enter the start and end vertex along with capacity
0 6 8
enter the start and end vertex along with capacity
6 5 2
enter the start and end vertex along with capacity
5 6 2


Max Flow is:6
```

Ans 3-

```
#include <iostream>
```

```
#include <string.h>
using namespace std;
#define M 6
#define N 6
bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR[])
{
    for (int v = 0; v < N; v++)
    {
        if (bpGraph[u][v] && !seen[v])
        {
            seen[v] = true;
            if (matchR[v] < 0 || bpm(bpGraph, matchR[v],seen, matchR))
            {
                matchR[v] = u;
                return true;
            }
        }
    }
    return false;
}

int maxBPM(bool bpGraph[M][N])
{
    int matchR[N];
    memset(matchR, -1, sizeof(matchR));
    int result = 0;
    for (int u = 0; u < M; u++)
    {
        bool seen[N];
        memset(seen, 0, sizeof(seen));
        if (bpm(bpGraph, u, seen, matchR))
            result++;
    }
    return result;
}

int main()
{
    bool bpGraph[M][N] = {{0, 1, 1, 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, 0},
                          {0, 0, 0, 0, 0, 1}};

    cout << "Maximum number of applicants that can get job is "<<
maxBPM(bpGraph);

    return 0;
}
```

Output-



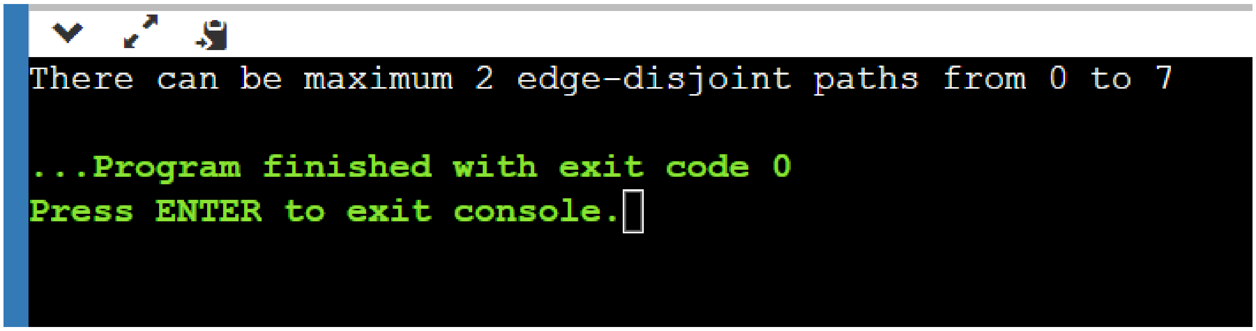
```
Maximum number of applicants that can get job is 5

...Program finished with exit code 0
Press ENTER to exit console.
```

Ans 4-

```
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;
#define V 8
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);
}
```

```
int findDisjointPaths(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, 1, 1, 1, 0, 0, 0, 0},
                        {0, 0, 1, 0, 0, 0, 0, 0},
                    {0, 0, 0, 1, 0, 0, 1, 0},
                    {0, 0, 0, 0, 0, 0, 1, 0},
                    {0, 0, 1, 0, 0, 0, 0, 1},
                    {0, 1, 0, 0, 0, 0, 0, 1},
                    {0, 0, 0, 0, 0, 1, 0, 1},
                    {0, 0, 0, 0, 0, 0, 0, 0}
                  };

    int s = 0;
    int t = 7;
    cout << "There can be maximum " << findDisjointPaths(graph, s, t)
      << " edge-disjoint paths from " << s <<" to "<< t ;

    return 0;
}
```

Output-



```
There can be maximum 2 edge-disjoint paths from 0 to 7

...Program finished with exit code 0
Press ENTER to exit console.
```