BFS

```
// Program to print BFS traversal from a given source vertex. BFS(int s)
// traverses vertices reachable from s.
#include<iostream>
#include <list>

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it visited
        // and enqueue it
        for(i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if(!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

DFS

```
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v. It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
```

```
    // Call the recursive helper function to print DFS traversal
    DFSUtil(v, visited);
}

//////////////////////////////////////////////////////////////DETECT CYCLE DIRECTED
// A C++ Program to detect cycle in a graph
#include<iostream>
#include <list>
#include <limits.h>

// This function is a variation of DFSUytil() in http://www.geeksforgeeks.org/archives/1821
2
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
        // Mark the current node as visited and part of recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if ( !visited[*i] && isCyclicUtil(*i, visited, recStack) )
                return true;
            else if (recStack[*i])
                return true;
        }

    }
    recStack[v] = false;  // remove the vertex from recursion stack
    return false;
}

// Returns true if the graph contains a cycle, else false.
// This function is a variation of DFS() in http://www.geeksforgeeks.org/archives/18212
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for(int i = 0; i < V; i++)
    {
        visited[i] = false;
        recStack[i] = false;
    }

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}

 ////////////////////////////////////////////////////////////////////DETECT CYCLE UNDIRECTED
 GRAPH
 // A C++ Program to detect cycle in an undirected graph
#include<iostream>
#include <list>
#include <limits.h>

// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
```

```
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // If an adjacent is not visited, then recur for that adjacent
        if (!visited[*i])
        {
           if (isCyclicUtil(*i, visited, v))
              return true;
        }

        // If an adjacent is visited and not parent of current vertex,
        // then there is a cycle.
        else if (*i != parent)
           return true;
    }
    return false;
}

// Returns true if the graph contains a cycle, else false.
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for (int u = 0; u < V; u++)
        if (!visited[u]) // Don’t recur for u if it is already visited
          if (isCyclicUtil(u, visited, -1))
             return true;

    return false;
}

//////////////////////////////////////////////////////////////////////////top sort
// A C++ program to print topological sorting of a DAG
#include<iostream>
#include <list>
#include <stack>
using namespace std;

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int> &Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack which stores result
    Stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
```

```
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
      if (visited[i] == false)
        topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}

//////////////////////////////////////////////////Boggle (Find all possible words in a board
 of characters)
// C++ program for Boggle game
#include<iostream>
#include<cstring>
using namespace std;

#define M 3
#define N 3

// Let the given dictionary be following
string dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
int n = sizeof(dictionary)/sizeof(dictionary[0]);

// A given function to check if a given string is present in
// dictionary. The implementation is naive for simplicity. As
// per the question dictionary is givem to us.
bool isWord(string &str)
{
    // Linearly search all words
    for (int i=0; i<n; i++)
        if (str.compare(dictionary[i]) == 0)
          return true;
    return false;
}

// A recursive function to print all words present on boggle
void findWordsUtil(char boggle[M][N], bool visited[M][N], int i,
                   int j, string &str)
{
    // Mark current cell as visited and append current character
    // to str
    visited[i][j] = true;
    str = str + boggle[i][j];

    // If str is present in dictionary, then print it
    if (isWord(str))
        cout << str << endl;

    // Traverse 8 adjacent cells of boggle[i][j]
    for (int row=i-1; row<=i+1 && row<M; row++)
      for (int col=j-1; col<=j+1 && col<N; col++)
        if (row>=0 && col>=0 && !visited[row][col])
          findWordsUtil(boggle,visited, row, col, str);
```

```
    // Erase current character from string and mark visited
    // of current cell as false
    str.erase(str.length()-1);
    visited[i][j] = false;
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N])
{
    // Mark all characters as not visited
    bool visited[M][N] = {{false}};

    // Initialize current string
    string str = "";

    // Consider every character and look for all words
    // starting with this character
    for (int i=0; i<M; i++)
       for (int j=0; j<N; j++)
             findWordsUtil(boggle, visited, i, j, str);
}

//////////////////////////////Shortest path with exactly k edges in a directed and weighted
 graph

// C++ program to find shortest path with exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and inifinite value
#define V 4
#define INF INT_MAX

// A naive recursive function to count walks from u to v with k edges
int shortestPath(int graph[][V], int u, int v, int k)
{
   // Base cases
   if (k == 0 && u == v)             return 0;
   if (k == 1 && graph[u][v] != INF) return graph[u][v];
   if (k <= 0)                       return INF;

   // Initialize result
   int res = INF;

   // Go to all adjacents of u and recur
   for (int i = 0; i < V; i++)
   {
       if (graph[u][i] != INF && u != i && v != i)
       {
           int rec_res = shortestPath(graph, i, v, k-1);
           if (rec_res != INF)
              res = min(res, graph[u][i] + rec_res);
       }
   }
   return res;
}

/////////////////////////////////////////////////////////////////////////////////dp solution
///////////////////////// Dynamic Programming based C++ program to find shortest path with
/////////////////////////// exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and inifinite value
#define V 4
#define INF INT_MAX
```

```
// A Dynamic programming based function to find the shortest path from
// u to v with exactly k edges.
int shortestPath(int graph[][V], int u, int v, int k)
{
    // Table to be filled up using DP. The value sp[i][j][e] will store
    // weight of the shortest path from i to j with exactly k edges
    int sp[V][V][k+1];

    // Loop for number of edges from 0 to k
    for (int e = 0; e <= k; e++)
    {
        for (int i = 0; i < V; i++)  // for source
        {
            for (int j = 0; j < V; j++) // for destination
            {
                // initialize value
                sp[i][j][e] = INF;

                // from base cases
                if (e == 0 && i == j)
                    sp[i][j][e] = 0;
                if (e == 1 && graph[i][j] != INF)
                    sp[i][j][e] = graph[i][j];

                //go to adjacent only when number of edges is more than 1
                if (e > 1)
                {
                    for (int a = 0; a < V; a++)
                    {
                        // There should be an edge from i to a and a
                        // should not be same as either i or j
                        if (graph[i][a] != INF && i != a &&
                            j!= a && sp[a][j][e-1] != INF)
                          sp[i][j][e] = min(sp[i][j][e], graph[i][a] +
                                                             sp[a][j][e-1]);
                    }
                }
           }
        }
    }
    return sp[u][v][k];
}


////////////////////////////////////// C++ program to find out whether a given graph is Bip
artite or not
#include <iostream>
#include <queue>
#define V 4
using namespace std;

// This function returns true if graph G[V][V] is Bipartite, else false
bool isBipartite(int G[][V], int src)
{
    // Create a color array to store colors assigned to all veritces. Vertex
    // number is used as index in this array. The value '-1' of  colorArr[i]
    // is used to indicate that no color is assigned to vertex 'i'.  The value
    // 1 is used to indicate first color is assigned and value 0 indicates
    // second color is assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // Assign first color to source
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers and enqueue source vertex
```

```
    // for BFS traversal
    queue <int> q;
    q.push(src);

    // Run while there are vertices in queue (Similar to BFS)
    while (!q.empty())
    {
        // Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )
        int u = q.front();
        q.pop();

         // Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v)
        {
            // An edge from u to v exists and destination v is not colored
            if (G[u][v] && colorArr[v] == -1)
            {
                // Assign alternate color to this adjacent v of u
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }

            //  An edge from u to v exists and destination v is colored with
            // same color as u
            else if (G[u][v] && colorArr[v] == colorArr[u])
                return false;
        }
    }

    // If we reach here, then all adjacent vertices can be colored with
    // alternate color
    return true;
}

///////////////////////////////////////////// C++ program to print transitive closure of a
graph
#include<bits/stdc++.h>
using namespace std;

class Graph
{
    int V; // No. of vertices
    bool **tc; // To store transitive closure
    list<int> *adj; // array of adjacency lists
    void DFSUtil(int u, int v);
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w) { adj[v].push_back(w); }

    // prints transitive closure matrix
    void transitiveClosure();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];

    tc = new bool* [V];
    for (int i=0; i<V; i++)
    {
        tc[i] = new bool[V];
        memset(tc[i], false, V*sizeof(bool));
    }
}
```

```
// A recursive DFS traversal function that finds
// all reachable vertices for s.
void Graph::DFSUtil(int s, int v)
{
    // Mark reachability from s to t as true.
    tc[s][v] = true;

    // Find all the vertices reachable through v
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (tc[s][*i] == false)
            DFSUtil(s, *i);
}

// The function to find transitive closure. It uses
// recursive DFSUtil()
void Graph::transitiveClosure()
{
    // Call the recursive helper function to print DFS
    // traversal starting from all vertices one by one
    for (int i = 0; i < V; i++)
        DFSUtil(i, i); // Every vertex is reachable from self.

    for (int i=0; i<V; i++)
    {
        for (int j=0; j<V; j++)
            cout << tc[i][j] << " ";
        cout << endl;
    }
}

// C Program for Floyd Warshall Algorithm
#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 floydWarshell (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
      distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
      ---> Before start of a iteration, we have shortest distances between all
      pairs of vertices such that the shortest distances consider only the
      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is added to the set of
      intermediate vertices and the set becomes {0, 1, 2, .. 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 ("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 program to test above function
int main()
{
    /* Let us create the following weighted graph
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3           */
    int graph[V][V] = { {0,   5,  INF, 10},
                        {INF, 0,   3, INF},
                        {INF, INF, 0,   1},
                        {INF, INF, INF, 0}
                      };

    // Print the solution
    floydWarshell(graph);
    return 0;
}

// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board
#include<iostream>
#include <queue>
using namespace std;

// An entry in queue used in BFS
struct queueEntry
{
```

```
    int v;     // Vertex number
    int dist;  // Distance of this vertex from source
};

// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder at i
// takes to.
int getMinDiceThrows(int move[], int N)
{
    // The graph has N vertices. Mark all the vertices as
    // not visited
    bool *visited = new bool[N];
    for (int i = 0; i < N; i++)
        visited[i] = false;

    // Create a queue for BFS
    queue<queueEntry> q;

    // Mark the node 0 as visited and enqueue it.
    visited[0] = true;
    queueEntry s = {0, 0};  // distance of 0't vertex is also 0
    q.push(s);  // Enqueue 0'th vertex

    // Do a BFS starting from vertex at index 0
    queueEntry qe;  // A queue entry (qe)
    while (!q.empty())
    {
        qe = q.front();
        int v = qe.v; // vertex no. of queue entry

        // If front vertex is the destination vertex,
        // we are done
        if (v == N-1)
            break;

        // Otherwise dequeue the front vertex and enqueue
        // its adjacent vertices (or cell numbers reachable
        // through a dice throw)
        q.pop();
        for (int j=v+1; j<=(v+6) && j<N; ++j)
        {
            // If this cell is already visited, then ignore
            if (!visited[j])
            {
                // Otherwise calculate its distance and mark it
                // as visited
                queueEntry a;
                a.dist = (qe.dist + 1);
                visited[j] = true;

                // Check if there a snake or ladder at 'j'
                // then tail of snake or top of ladder
                // become the adjacent of 'i'
                if (move[j] != -1)
                    a.v = move[j];
                else
                    a.v = j;
                q.push(a);
            }
        }
    }

    // We reach here when 'qe' has last vertex
    // return the distance of vertex in 'qe'
    return qe.dist;
```

```
}

// Driver program to test methods of graph class
int main()
{
    // Let us construct the board given in above diagram
    int N = 30;
    int moves[N];
    for (int i = 0; i<N; i++)
        moves[i] = -1;

    // Ladders
    moves[2] = 21;
    moves[4] = 7;
    moves[10] = 25;
    moves[19] = 28;

    // Snakes
    moves[26] = 0;
    moves[20] = 8;
    moves[16] = 3;
    moves[18] = 6;

    cout << "Min Dice throws required is " << getMinDiceThrows(moves, N);
    return 0;
}
```