SNAKE AND LADDER PROBLEM

- Project by:

1. Sahil Mokkapati – RA2011003010879
2. Sheetal Jatav – RA2011003010885
3. Aum Shah – RA2011003010872

- Department – Ctech
- Subject – DAA
- Section – C2

PROBLEM STATEMENT

Like a traditional snakes and ladders game, we are provided with a 10*10 board numbered from one to one hundred and dice with faces numbered from one to six. We start from square one, and the game is considered finished when a player reaches square one hundred.

It is all based on the player-rolling dice, for example if the player gets a number from the 6-numbered dice and he/she might get a ladder and win another roll or get a snake and lose the position. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.

So, to dig into this problem we'll be focusing on some algorithms on how this game actually works.

REAL-TIME EXAMPLES

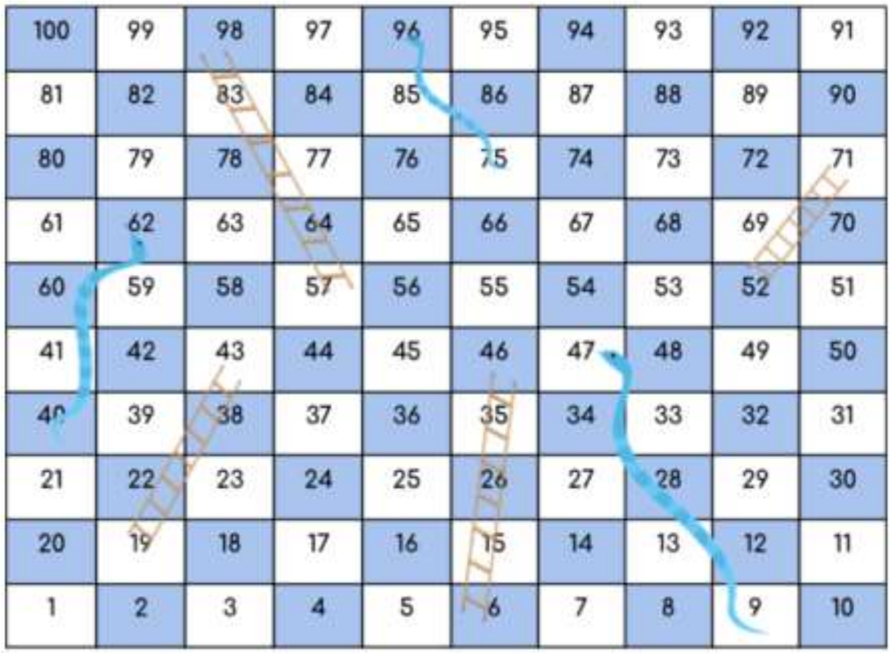
Covering two other aspects of snakes and ladders we are given two components of ladder and snake where,

1. Ladder[i][0] denotes the starting square of the ith ladder and ladder[i][1] denotes the end of the ith ladder.

2. Snake[i][0] denotes the mouth of the ith snake and snake[i][1] denotes the tail of the ith snake.

Now we are required to find the minimum number of rolls to reach the hundredth square.

A real-time example says:



We’ll be creating array inputs for both snake and ladder.

Input Ladder = [[6,46],[19,43],[52,71],[57,98]]
Snake = [[47,9],[62,40],[96,75]]

This array will tell the start and end point of either snake or ladder.

<u>BASIC IDEA</u>:

The idea is to consider the given snake and ladder board as a directed graph with the number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to the next six vertices if the next 6 vertices do not have a snake or ladder.

<u>IMPLEMENTATION</u>:

Following is the implementation of the above idea. The input is represented by two things, first is ‘N’ which is number of cells in the given board, second is an array ‘move[0…N-1]’ of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

APPROACH

From the above-mentioned idea, we see that If any of the next six vertices has a snake or ladder, then the edge from the current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find the shortest path using **Breadth First Search** of the graph.

For this approach, we consider the board a graph and every square as the vertex of the graph. Here our starting vertex is one, and one hundred is the final, or destination, vertex. So, since there are six faces on a dice, we have six possible squares to move to from a particular square.

If we encounter ladder i, then it would take us from ladder[i][0] to ladder[i][1] in zero moves. Likewise, If we encounter a snake j, then it would take us from snake[j][0] to snake[j][1] in zero moves.

Coming to the core approach we do a level order search using BFS, where we push all possible positions in a queue, and in the next iteration loop over only the next level positions, and increment the final answer with the increasing levels in the algorithm.

BREADTH-FIRST SEARCH

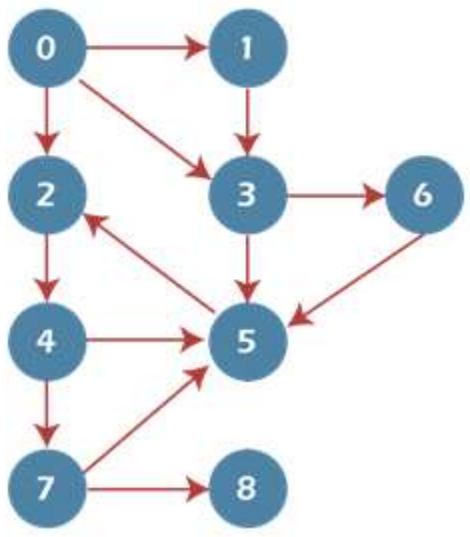
Introduction:

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node

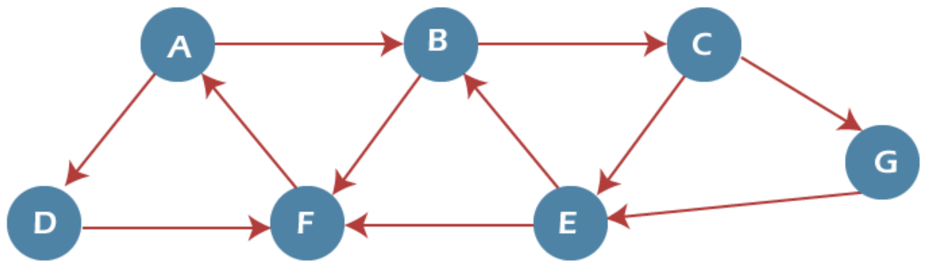
It is also known as the best method to transverse the graph BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure.

Implementation:

Implementing the Breadth-First Search algorithm makes it much easier to deal with the adjacency list since we only have to travel through the list of nodes attached to each node once the node is dequeued from the head start) of the queue.



EXAMPLE:



Using the BFS that will start from Node A and end at Node E. The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

As we saw, how BFS actually works, now let's move on to the snake and ladder problem. Let’s try to implement the BFS algorithm in the problem.
Below is the algorithm used for the problem given.

ALGORITHM:

Step-1 Create an explicit map and insert the ladder, snake jumping possibilities.

Step-2 Now declare a queue and a variable level incremented with 1.

Step-3 Now push 1 to queue and start doing a level order search.

Step-4 Calculate the size of the queue and traverse on each possible value.

Step-5 The new position will be x + i (1<=i<=6) if x+i is a value in map then we move to that like pos = m[x+i].

Step-6 If the new value is 100 then we return the variable level.

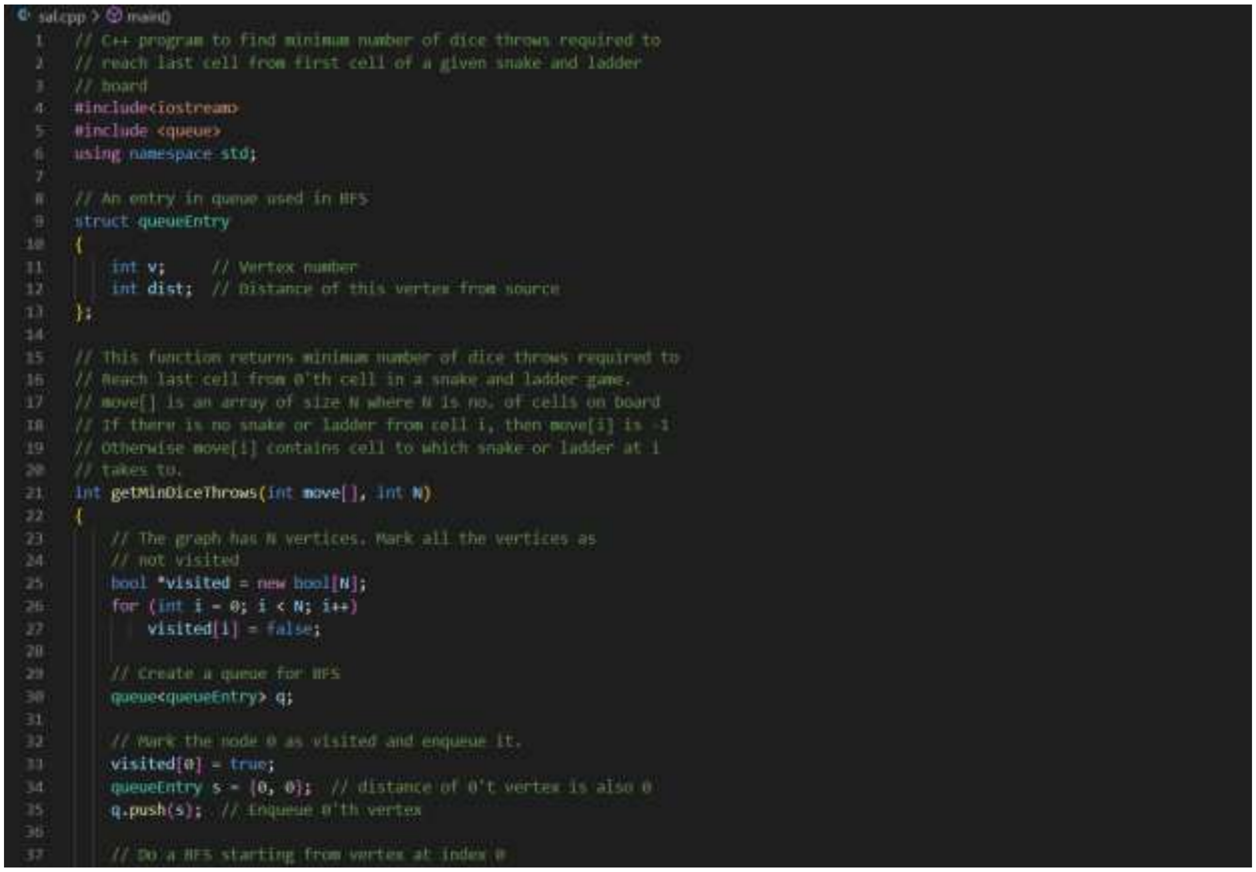
Step-7 Otherwise if the new position was not visited before then we mark it visited and push it into the queue.

Step-8 If we are out of the queue then we return -1.

MAPPING CODE:

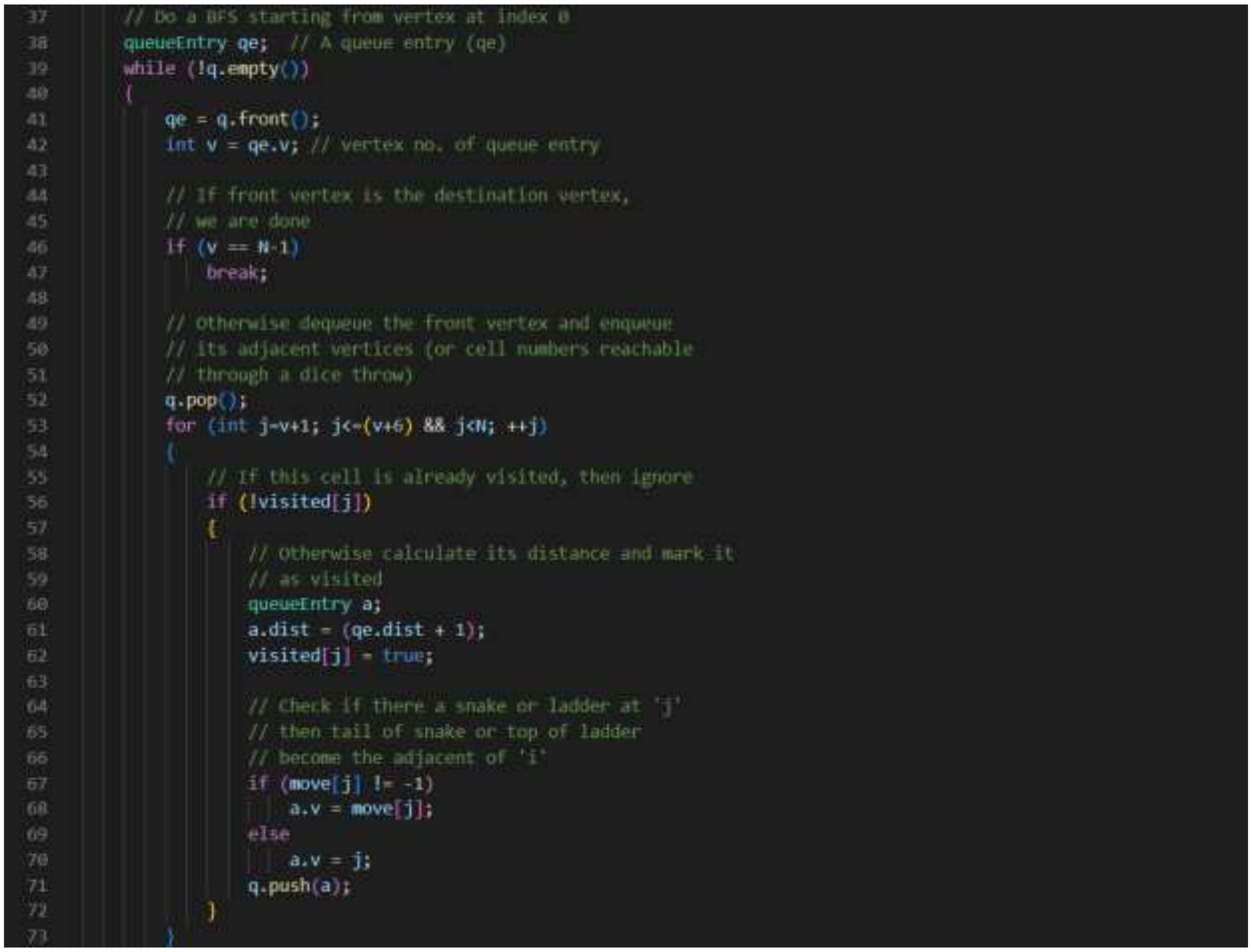
```
unordered_map < int, int > m;
for (vector < int > & i: ladder)
  m[i[0]] = i[1];

for (vector < int > & i: snake)
  m[i[0]] = i[1]
```

CODE:

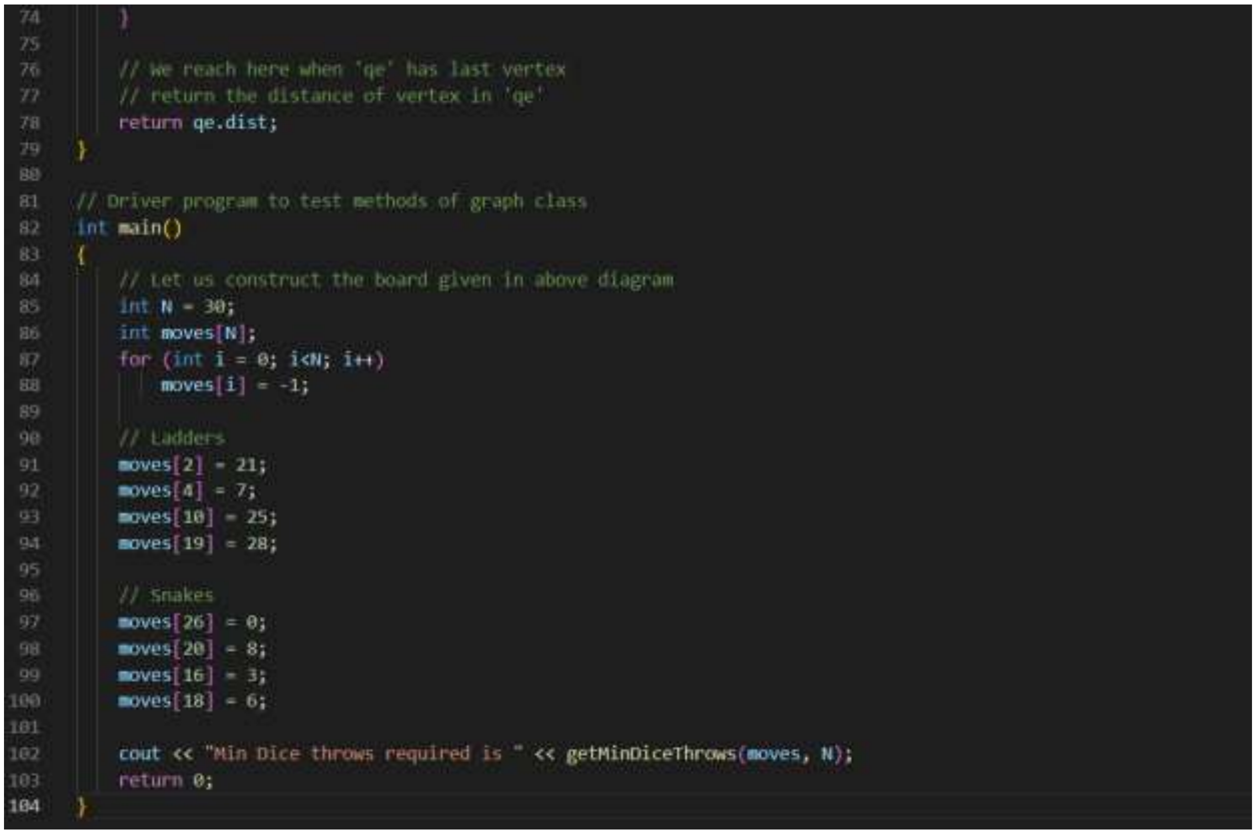


sal.cpp > main()

```cpp
// 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
```

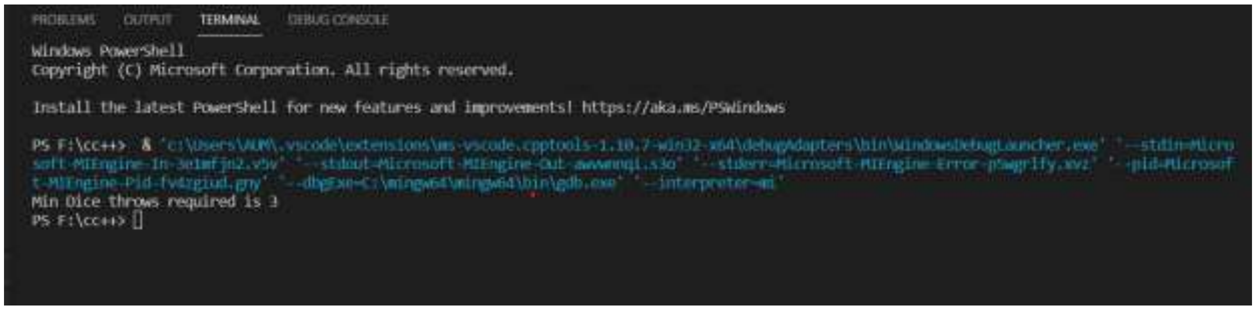


```
        // Do a BFS starting from vertex at index 0
        queueEntry qe;  // A queue entry (qe)
        while (!q.empty())
        {
            qe = q.front();
            int v = qe.v; // vertex no. of queue entry

            // 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);
                }
            }
```



```cpp
    }

    // 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;
}
```

OUTPUT:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS F:\cc++>  & 'c:\Users\AUM\.vscode\extensions\ms-vscode.cpptools-1.10.7-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Micro
soft-MIEngine-In-3e1mfjn2.v5v' '--stdout=Microsoft-MIEngine-Out-awwwnqi.s3o' '--stderr=Microsoft-MIEngine-Error-p5wgrlfy.xvz' '--pid=Microsof
t-MIEngine-Pid-fvdzgiud.gny' '--dbgExe=C:\mingw64\mingw64\bin\gdb.exe' '--interpreter=mi'
Min Dice throws required is 3
PS F:\cc++> []
```

TIME COMPLEXITY AND ANALYSIS

The time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of the BFS algorithm is O(V+E), since in the worst case, the BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

Complexity is as follows:-

$V * (O(1) + O(E_{aj}) + O(1))$
$V + V * E_{aj} + V$
2V + E(total number of edges in graph)
V + E

Space Complexity:

The space complexity of BFS can be expressed as O(V).

CONCLUSION/RESULT:

To reach the hundredth square, minimum dice rolls were used and its implementation was shown.

CONTRIBUITONS:

NAME	REG. NO.	CONTRIBUTION
SAHIL MOKKAPATI	RA2011003010879	DESIGN TECHNIQUE AND CODE
SHEETAL JATAV	RA2011003010885	COMPLEXITY ANALYSIS AND CODE
AUM SHAH	RA2011003010872	PROBLEM STATEMENT AND EXPLANATION AND CODE