MAZE

Declaration of 2D array

```
int maze[N][N] = {
    {1, 3, 2, 0},
    {3, 2, 0, 1},
    {1, 5, 2, 0},
    {0, 1, 3, 4}
};
```

Here, the two-dimensional array (data-type: integer) represents the grids of the maze. The values 0 represent dead-end and non-zero numbers represent the maximum jumps that can be made from the specific cell.

Backtracking to find longest path

```
// Making recursive calls for all possible jumps from the current position
for (int i = 1; i <= maze[ratX][ratY]; i++) {
    solveMaze(maze, output, ratX - i, ratY, currWeight + i); // Moving towards left
    solveMaze(maze, output, ratX + i, ratY, currWeight + i); // Moving towards right
    solveMaze(maze, output, ratX, ratY - i, currWeight + i); // Moving towards top
    solveMaze(maze, output, ratX, ratY + i, currWeight + i); // Moving towards bottom
}

// Unmark the current cell as part of the path
output[ratX][ratY] = 0;
```

The solveMaze() function recursively travels towards all four directions.

```
void solveMaze(int maze[N][N], int output[N][N], int ratX, int ratY, int currWeight) {
    if (ratX >= N || ratY >= N || ratX < 0 || ratY < 0) {
        return;
    }

    if (maze[ratX][ratY] == 0) {
        return;
    }

    if (output[ratX][ratY] == 1) {
        return;
    }
```

These are the conditions to stop travelling the out of matrix (maze), avoiding the visited cell and the paths that leads to dead-end. This makes a proper backtracking.

Printing the solution matrix

```
if (destX == ratX && destY == ratY) { // Cheking if the rat reached the destionation.
    if (currWeight > maxWeight) { // Updating the longest weight.
        maxWeight = currWeight;
    }

    solutionFound = true;

    printSolution(output);

    cout << "Weight of longest path is " << maxWeight << endl;

    return;
}
```

When solution is found (i.e., destination cell is reached), update the longest path and print solution matrix.

Program output

```
1 1 1 0
1 1 0 0
1 1 1 0
0 0 1 1
Weight of longest path is 10
```

This is the output of program for the provided maze data.

Huffman Decoding

Declaration of node of Huffman tree

```
class Node { // Instance of this class represents a node of the huffman tree.
public:
    char data;
    int freq;
    Node* left;
    Node* right;

    Node(char value, int frequency) { // Constructor
        data = value; // if node is leaf there will be a character in attribute 'data' if node is not leaf 'data'
        freq = frequency; // This is not actually necessary for decoding algorithm but in huffman tree we used to
        // frequency makes it easier to choose which nodes to merge while designing a huffman tree.
        left = NULL;
        right = NULL;
    }
};
```

Here, the Huffman tree is represented as linked list. The node in this linked list has data, frequency, left and right pointer.

Decoding the encoded message

```
string decodeMessage (string encodedMsg, string decodedMsg, Node* currNode, Node* rootNode) {
    if (currNode->left == NULL && currNode-> right == NULL) { // This represent the current node is a leaf node.
        decodedMsg += currNode->data; // saving the character in decoded message.
        return decodeMessage(encodedMsg, decodedMsg, rootNode, rootNode); // here we return rootNode as currentNo
    }

    if (encodedMsg == "") { // At end of the encoded string, return decoded message.
        return decodedMsg;
    }

    char currBit = encodedMsg.front(); // Taking 1st bit from encoded string to test where to move.
    encodedMsg.erase(0, 1); // Removing the 1st bit from string to make recursion easier.

    if (currBit == '0') { // if current bit is 0, move towards left child.
        return decodeMessage(encodedMsg, decodedMsg, currNode->left, rootNode);
    } else { // if current bit is 1, move towards right child
        return decodeMessage(encodedMsg, decodedMsg, currNode->right, rootNode);
    }

}
```

This is the decoder function that takes encoded message and root node of the Huffman tree and gives the decoded message. The basic concept is to move towards left or right child node of the Huffman tree for each bit in the encoded string until anyone of the leaf node is found.

Output of program

```
Decoded Message is : EDDDFFCEDBACFEDAEBAFDCBEBDABEA
```

This is the output message for provided encoded string and Huffman tree.