## Problem Description

The first time this project appeared, my colleagues and I had a hard time discussing what we would create. Then we came up with the idea to make the knight’s tour. The first idea we had is to make the knight’s tour using the BFS and DFS. BFS stands for breadth-first search, the data structure behind BFS is a queue. Meanwhile, in the middle of the process, we decided to replace DFS with a backtracking algorithm. The first time when we did the research for this, we had the hypothesis that the BFS will be more efficient.

Backtracking is an algorithm that has the function to find the shortest path. The backtracking algorithm is an upgraded algorithm of brute force. In this project, we use arrays on the backtracking, which are a linear data structure. In this problem, we use backtracking to find all possible moves for the knight.

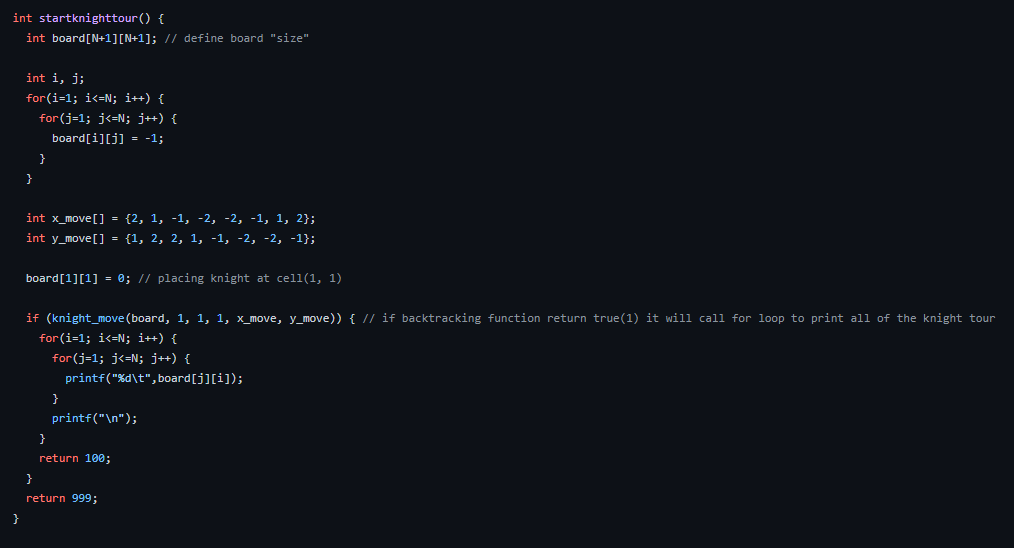
For the comparison, we are proposing BFS to be the solution to this problem. In BFS we applied queue data structure. In the BFS algorithm, it will go from the parent node and check all of the children nodes first, then go to the lower level.

## Program Manual

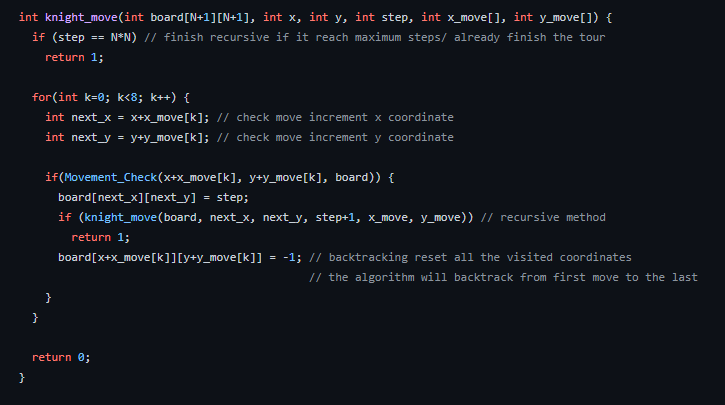
### 2.1 Backtracking



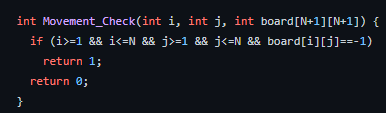
At the first, we will start with this main function. In this function, it will call the startknighttour function. Later it will count the time of the CPU.



In the startknighttour function, first, it will define the board size. The board size itself is already defined as 8, which is the standard size of a chessboard. There are 2 for loops to mark all off the part of the board as -1. Below that there are x\_move and y\_move which is the all possible movement for the knight. After that, we initialize the place of the knight, and we set it at the top left or (1,1).

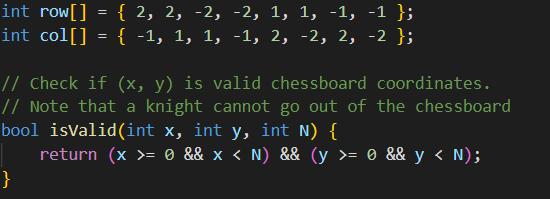


The first if’s function is to stop the recursive if the step is already the same with the number of the size of the board, but the maximum step is 63 because it starts at 0. The first for loops is to increment the current position with the possible move of the knight.

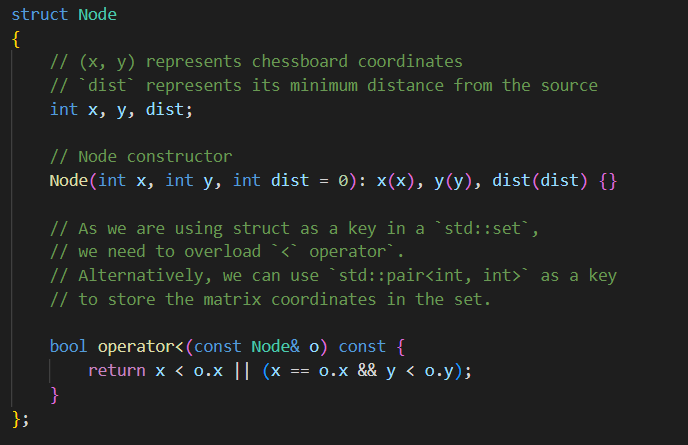


The movement\_check is a function to check if the move of the knight is valid, valid means the knight is still on the board. Continuing the knight\_move function, if the step that it checks will make a solution, then it will increment the step variable. If the checked step is not returning a solution, then it will backtrack and mark the coordinate as -1. If all of the loops inside knight\_move return 1 it will end the recursive function and it will continue to startknighttour function. It will print each of the elements on the board with the step of the knight tour.

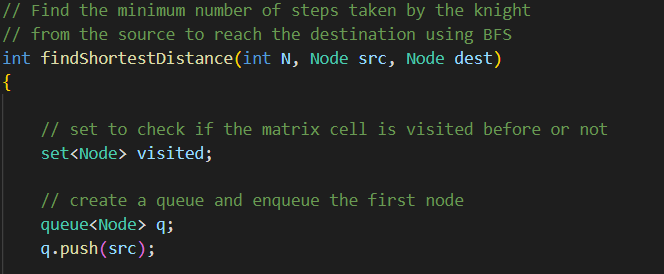
2.2 Queue (BFS)



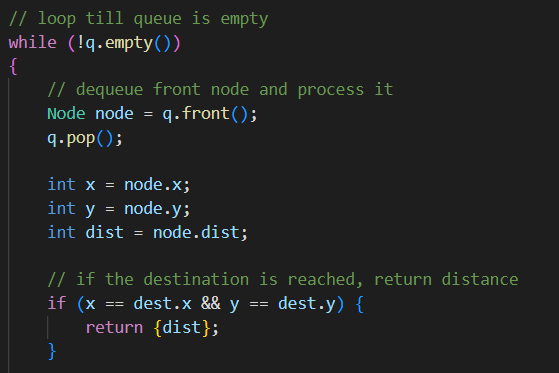
To begin, these are the possible moves of the knight and the subsequent check to see if the knight’s moves are valid or not.



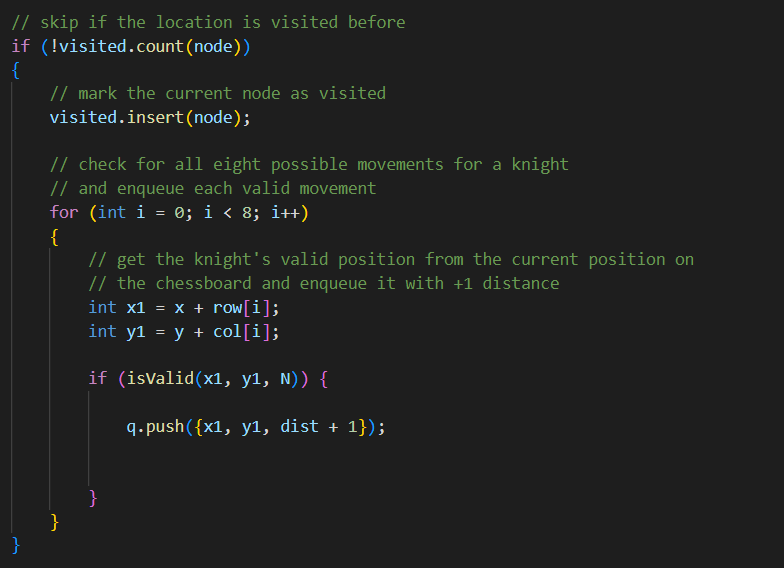
This is the function that constructs the nodes as well as the operator function that returns a boolean and then pushes it into the matrix, which will then be used in the findShortestDistance



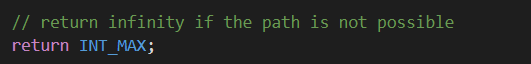
In the findShortestDistance, it uses queues to find the shortest distance to the set node that has been structured, it will push the size from board to the node from the queue.



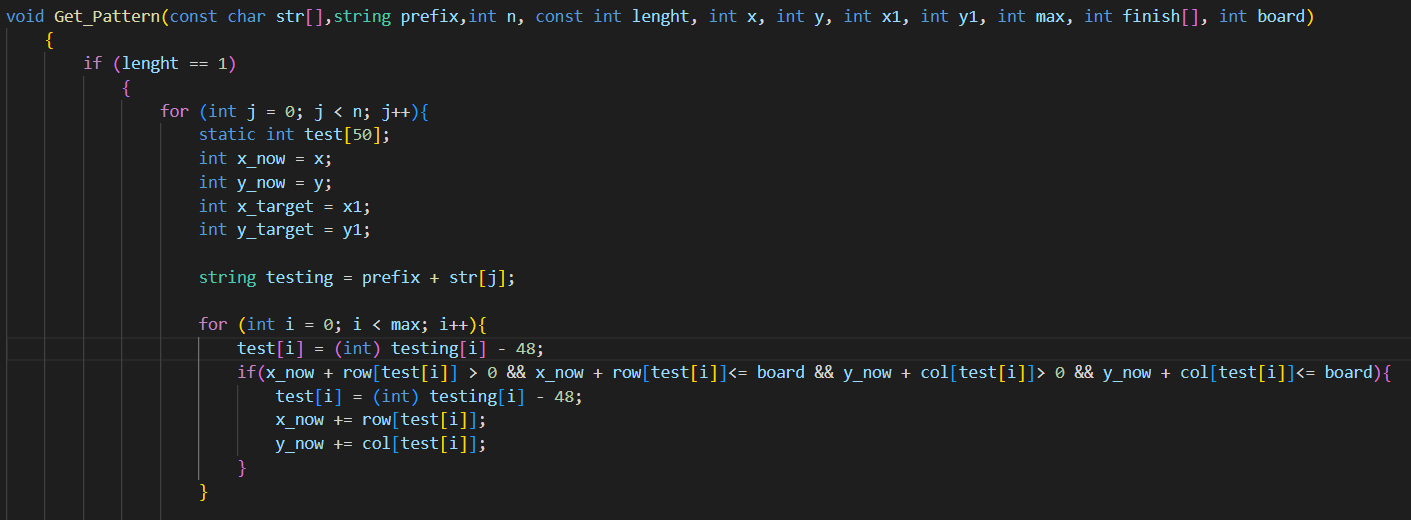
After that, it loops at the start so the node that’s defined in the beginning equals to get q.front() which is the size, after that q is popped. It will keep looping until it finally meets the if statement which is the destination x and y and will show the minimum amount of moves.



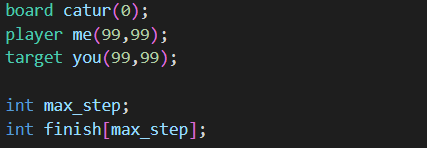
After q is popped, x,y and distance from the structured is defined. If the location that is placed isn’t there yet, it will be turned into an insert node. In the insert node, there’s a loop with a total of 8 possible moves, if the move is valid or if in the board queue with a valid x,y coordinate and the distance added by 1 to count the moves done.



If it is impossible to reach the final destination, the function will return INT\_MAX or infinity.



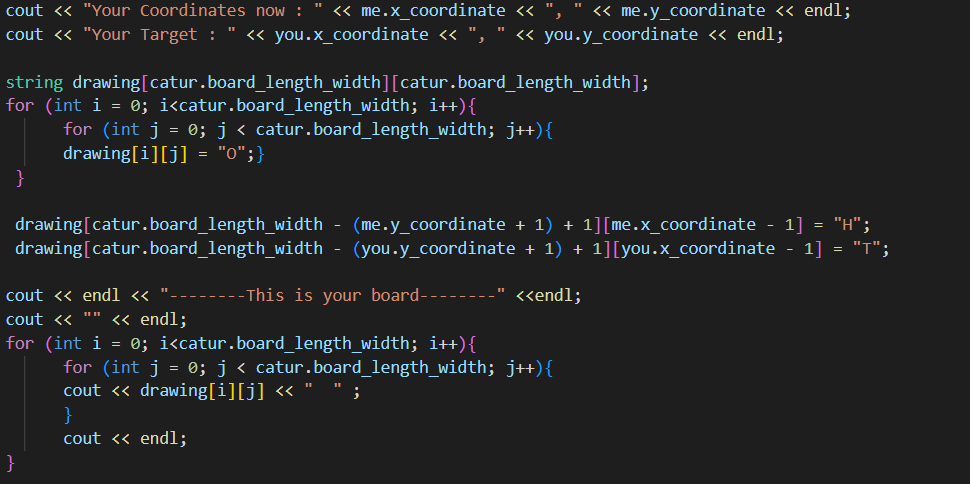
This is the Get Pattern function,it is used for finding the correct pattern. What this algorithm is actually doing is it will give a max step number of patterns. For example the minimum step is 5, it would give 00000 until 88888. And if it find the right pattern, it would return the pattern to be used in the history algorithm.



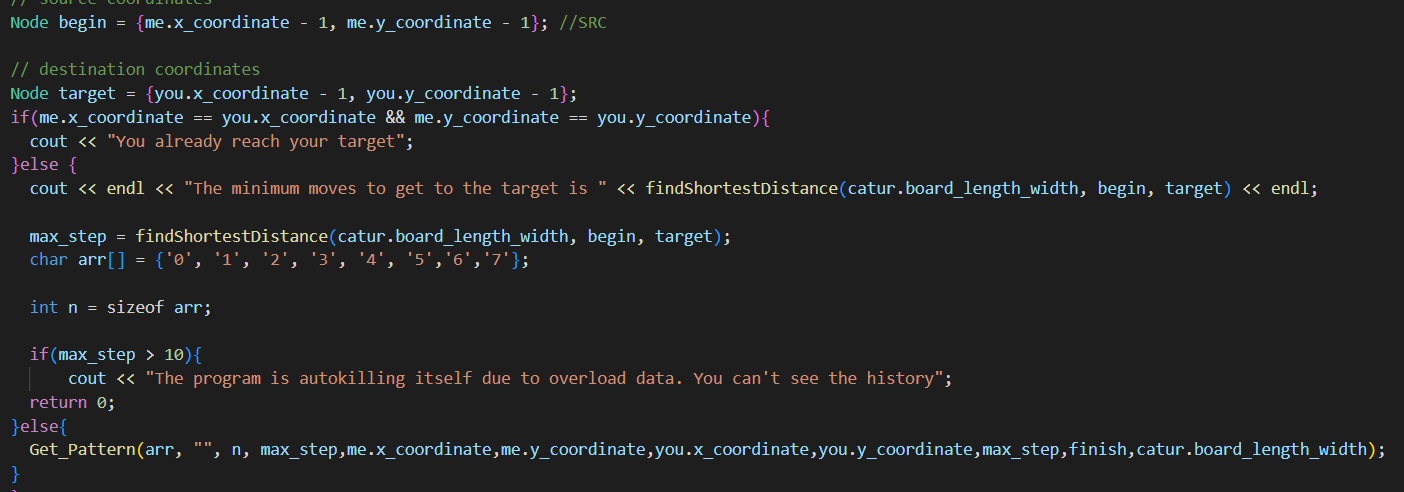
Calling every classes that needed for the program, also declaring variable for max step and an array for the patternText

Description automatically generated

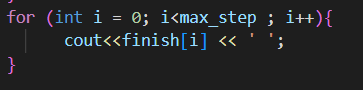
For the user to input coordinates and target, as well as the board size.



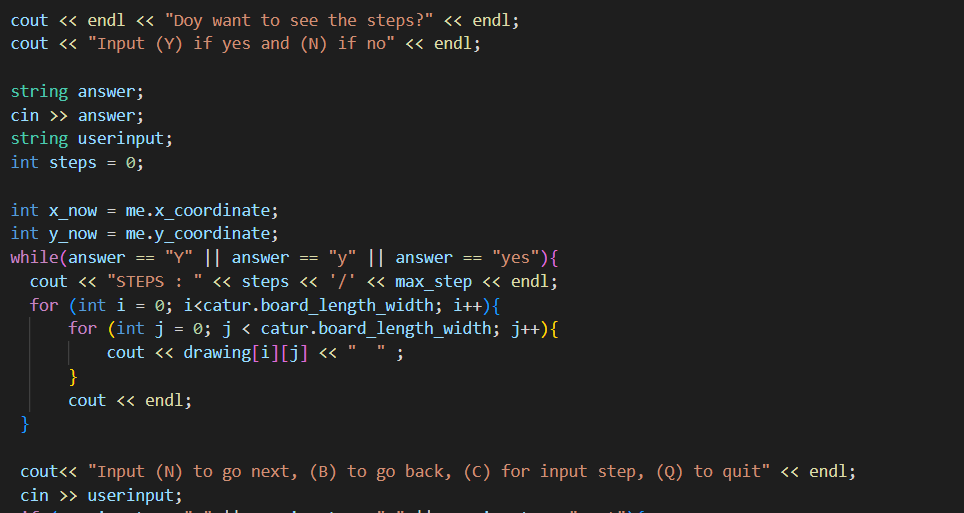
Visualize the board that has been declared from the user input.



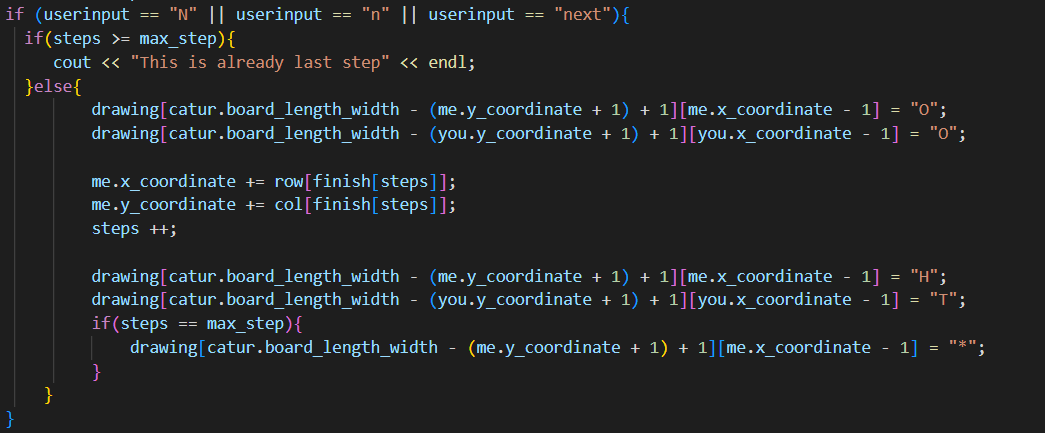
Running until get the shortest distance and one of the possible patter to be use for the history.



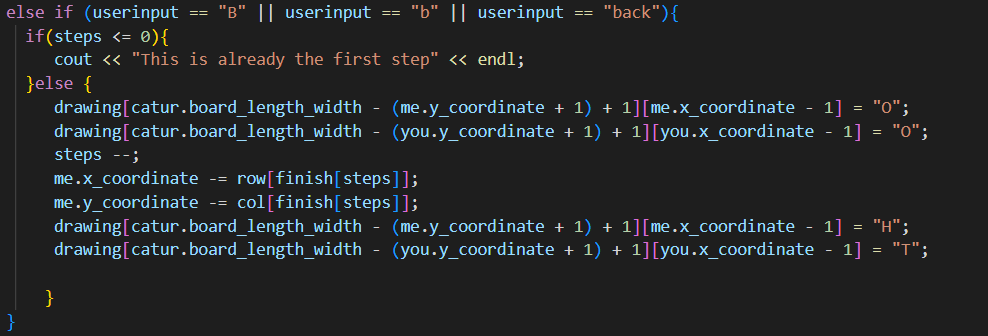
Printing the pattern.



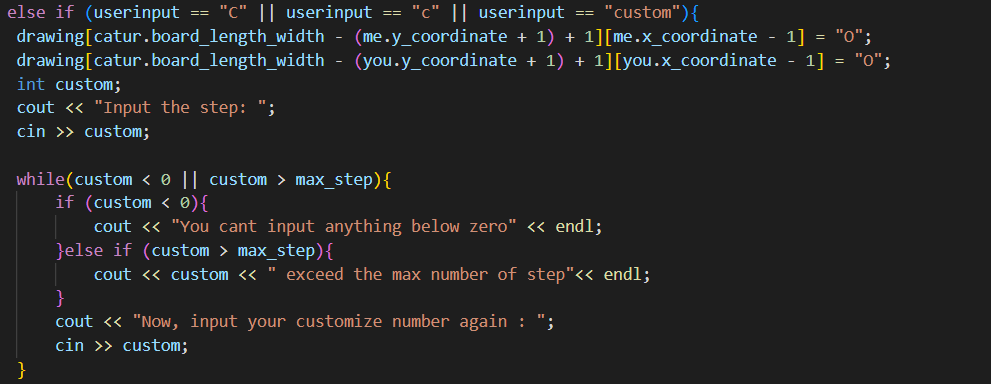
Ask the user if they want to see the history or not, if yes they will be given the history program.



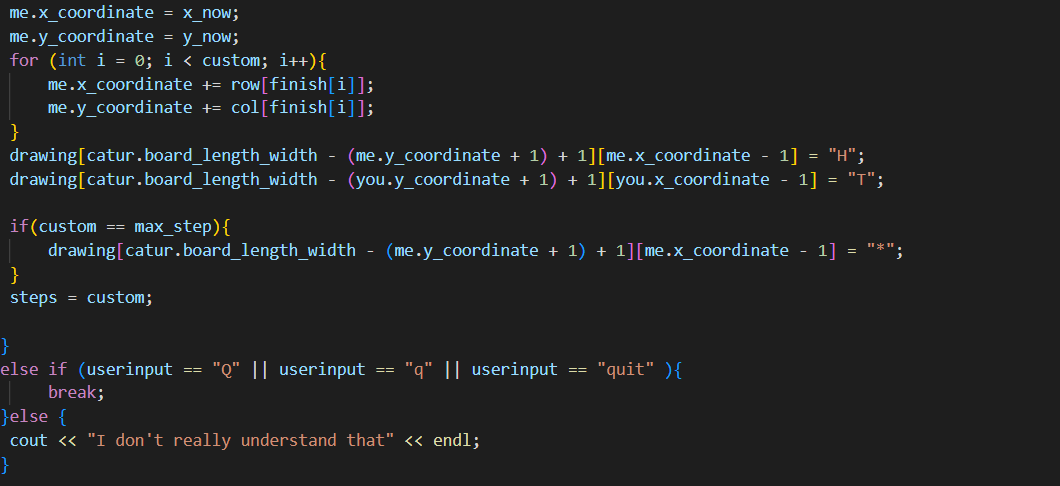
User can input N to go next step, and if already in max step, they can’t do it again.



This is if the player on user want to go back, if they already in the first step. They can’t do it again.



This is for the custom input, user can choose which step they want to see.

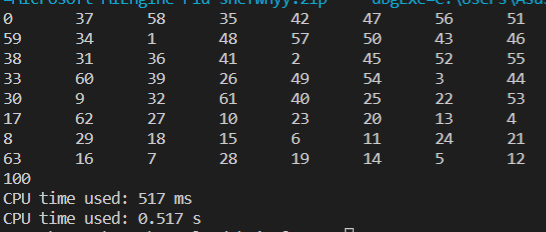


Here is just for telling the program to give sign if the user already hit the target in custom input.

## Example Result

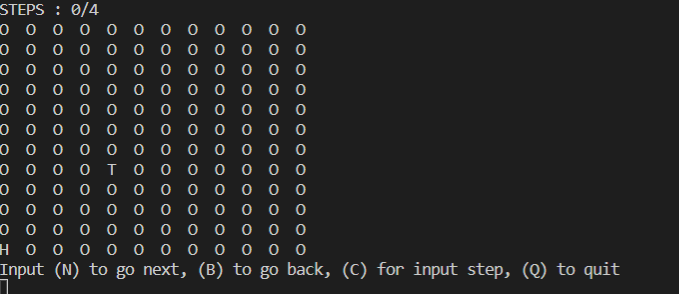
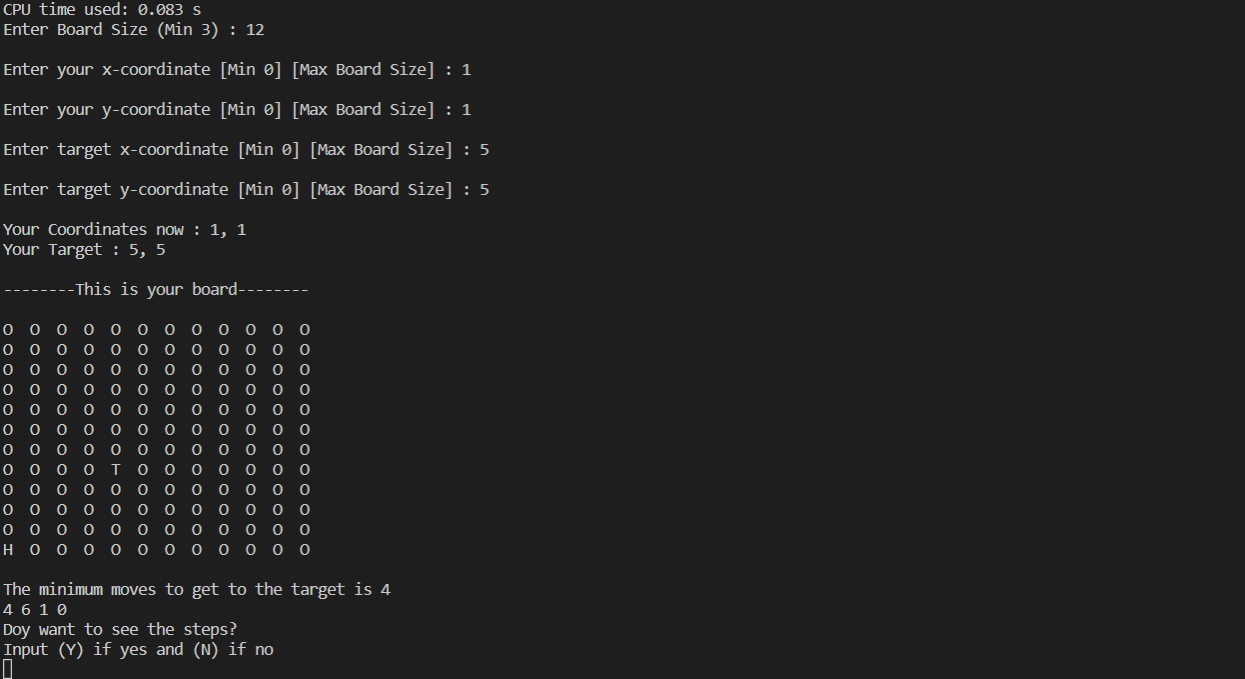
### 3.1 Backtracking

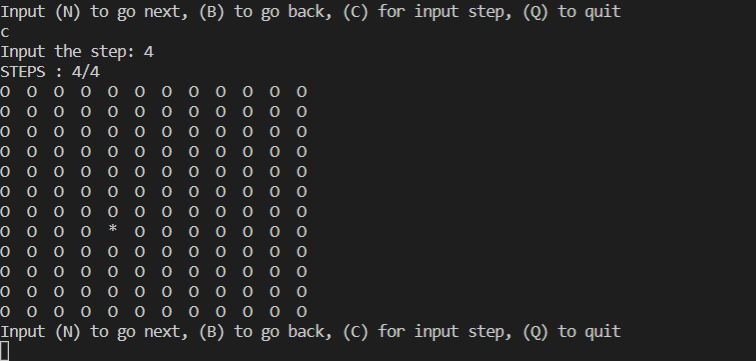
The result if we run the program will show the board of the chess and the steps of the knight’s tour and for a success parameter, it will return value 100 and also print it in the terminal. It will also print the CPU time, so we can compare the time between backtracking and the BFS.



### 3.2 Queue (BFS)

### The result is that if we run the BFS code, first we can input the number/size of the board, next we need to input our coordinates and target coordinates. And then it will give/return the smallest number of needed moves to get to the target coordinate from your coordinate. And then, it will also print one of the possible patterns from the given smallest moves. And then we will have the history/visualisation of the steps. We can also input whether we want to go next or back, we also add a custom feature where you can input what step you want.





### 3.3 Table of Comparison

|  |  |  |
| --- | --- | --- |
| **Parameter** | Backtracking | Queue(BFS) |
| Flexibility | In our backtracking code, it does not have the flexibility to change the coordinates, and also the size of the board. And it only resulted in giving the complete knight tour. External code is needed to have flexibility to change the value of the coordinate and size. | In the BFS code, you can input an unlimited number of size of the board, you can also change target coordinate as well as your coordinate. We also add some algorithms that can use the return value of the BFS algorithm to get the right pattern resulting in visualisation of the steps. However, the history can be shown up to 10 steps, if it is higher than 10 steps we add stop statement so your computer not go boom boom. |
| Run Time |  |  |
| Complexity | Backtracking is kinda simple to code, caused by the simple algorithm and also the use of looping from start until the end. In this case, we are using recursion for the loop. | BFS is subjectively for us is kinda hard to code and understand, caused by the use of complex nodes and also some external algorithm to get all of the patterns using mathematical formulas. |