ALGORITHM AND ANALYSIS GROUP PROJECT

Lecturer: Hua Chong Ling

Group name: Group 16
Group member:

- Le Minh Thai Hoa (S3979194)
- Cho Jaesuk (s3914532)
- Jimin Cho (S3940575)
- Nguyen Thanh Long (S3980490)

TABLE OF CONTENTS

1. Algorithm overview:

1.1. Overview:

Our solution uses the backtracking method to solve a grid pathfinding issue. The grid size is 8x8, and the objective is to calculate the number of pathways beginning in the first cell [0,0] while adhering to the input movement constraint.The path string includes any movement ("*") or direction constraints (U, R, L, and D). The approach uses backtracking to investigate potential routes while adhering to restrictions like grid boundaries and avoiding repeating visits to the same cells.

1.2. Classes:

- Path class:
  - Handles the main logic for the backtracking
  - Contains the grid, visited cells, and movement constraints
  - Count the number of valid paths
  - Calculate the time to count in millisecond

1.3. Attributes:

- count: A static variable to store the number of valid paths
- size: Define the size of the grid
- visited [ ] [ ]: A 2D Boolean array to track visited cells on the grid.
- moves [ ] [ ]: A 2D array represent the direction of move (U,R,D,L)
- path [ ]: Represent the sequence of moves in the path
- s: String input that contains the movement for each step of the path

1.4. Methods:

- start ():
  Initialize the grid boundaries, set up the visited array, and trigger the backtracking at the initial position.

- permute(int number, int row, int col):
  The recursive backtracking function to explore paths step-by-step

based on movement constraints.

1.5. The main() function:

- serves as a program entry point.
- generates a Path class object and calls the start() function.
- evaluates execution time for benchmarking performance.

1.6. Key functional components:

- Grid initialization:
    - The grid is surrounded by a boundary (using visited array)
- Backtracking Algorithm:
    - The permute method explores all the possible paths:
        - If a constraint (U, R, L, D) exists at the current step, only that direction is explored.
        - For no constraint step (using “*” as representation), all the direction is considered
- The recursion stops when:
    - A complete path of valid steps is found
    - A dead-end is found

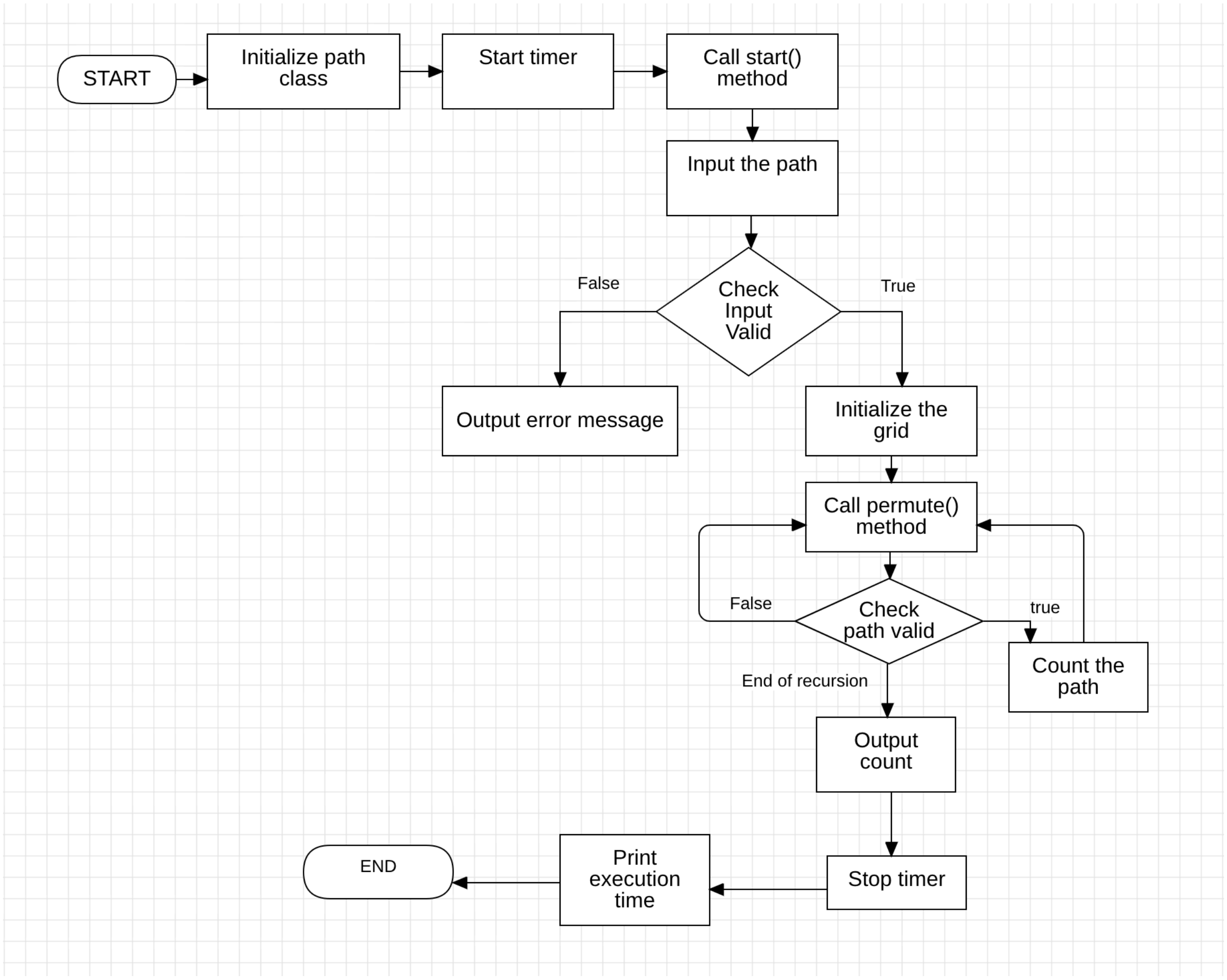
1.7. Software Design Patterns:

- Backtracking pattern:
  When limitations are broken, this method of problem-solving "backtracks" and stops working on the solution.

- Single responsibility principle:
  The Path class contains all the pathfinding logic, keeping it separate from external concerns like performance measurement (which is handled in main()).

- Encapsulation:
  The Path class encapsulates all grid and pathfinding-related variables and methods.

1.8. Algorithm's code flow:



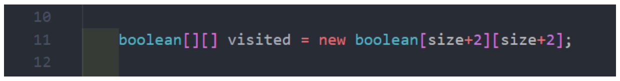
<Figure 1> Flow chart show Algorithm's code flow

2. The data structures and algorithms you apply or develop

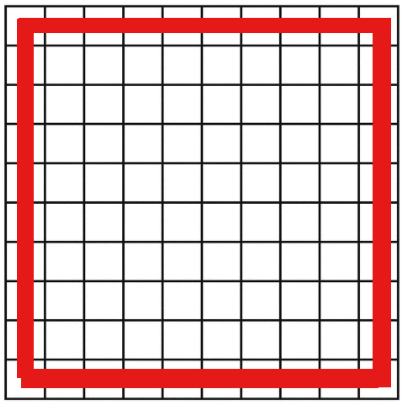
2.1 Data Structures Used

2.1.1 2D Array (visited)

- **Purpose**:
  The 2D array visited serves to track whether each cell in the grid has already been visited, preventing duplicate visits.
- **Implementation**:
  boolean[][] visited is declared with dimensions size+2 x size+2 to include the boundaries surrounding the 8x8 grid. This expanded grid size (10x10) ensures efficient boundary handling during exploration.

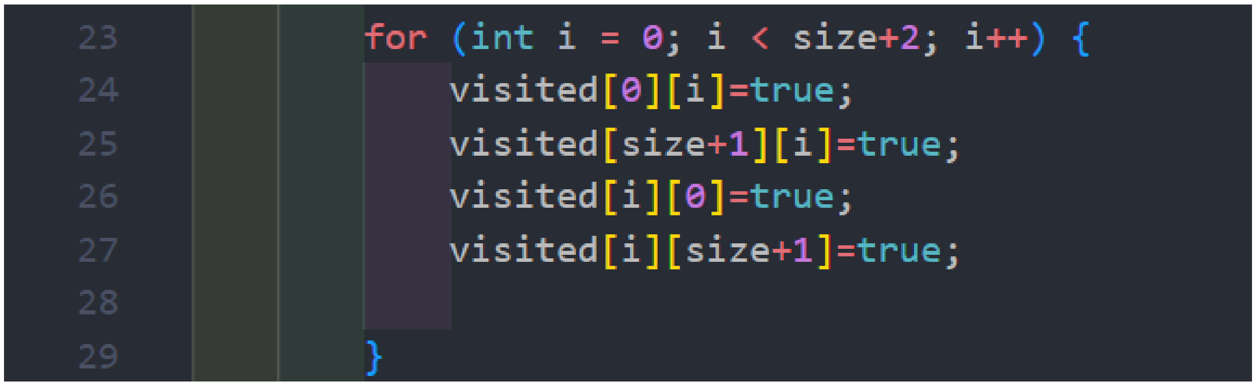


```
boolean[][] visited = new boolean[size+2][size+2];
```



<Figure 2> 2D Array (visited) implementation and 10x10 grid with boundary (red line)

- **Initialization**:
  During path exploration, the movement may exceed the boundaries of the 8x8 grid. To address this, the initialization step is designed to efficiently block movements that exceed the grid's boundary.
  By expanding the visited array to a size+2 x size+2 grid, an external boundary is created. During initialization, all external boundary cells are set to true, marking them as already visited. This design prevents the program from exploring these out-of-bound cells.



```
for (int i = 0; i < size+2; i++) {
    visited[0][i]=true;
    visited[size+1][i]=true;
    visited[i][0]=true;
    visited[i][size+1]=true;

}
```

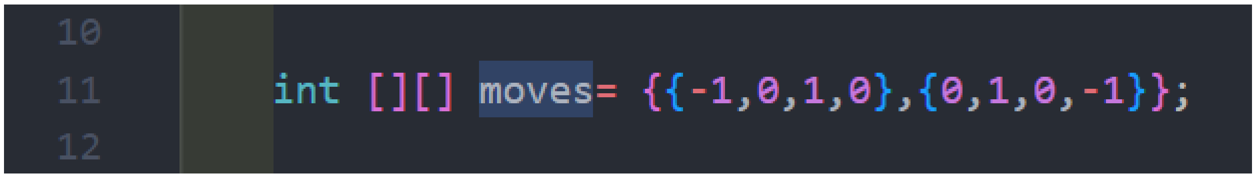
<Figure 3> Initialization implementation

- **Reason for Selection**:
  This data structure is the most suitable for solving the given problem due to the following advantages:
  - **Fast Visit Check**:
    Checking whether a cell has been visited has a time complexity of O(1), allowing instant verification.
  - **Scalability**:
    This structure is flexible and can be used for grids of varying sizes,

which is ideal for testing with different grid dimensions as required by the assignment.

2.1.2 Array (moves)

- **Purpose**:
  The moves array is a directional array that defines the values for moving in four directions: up, down, left, and right.
- **Implementation**:
  Each element in the array represents the changes in row and column values for each movement. This array is combined with a loop to process movement logic concisely and efficiently.
- **Structure of the Array**:
  - moves[0][i]: Change in **row** for direction i.
  - moves[1][i]: Change in **column** for direction i.



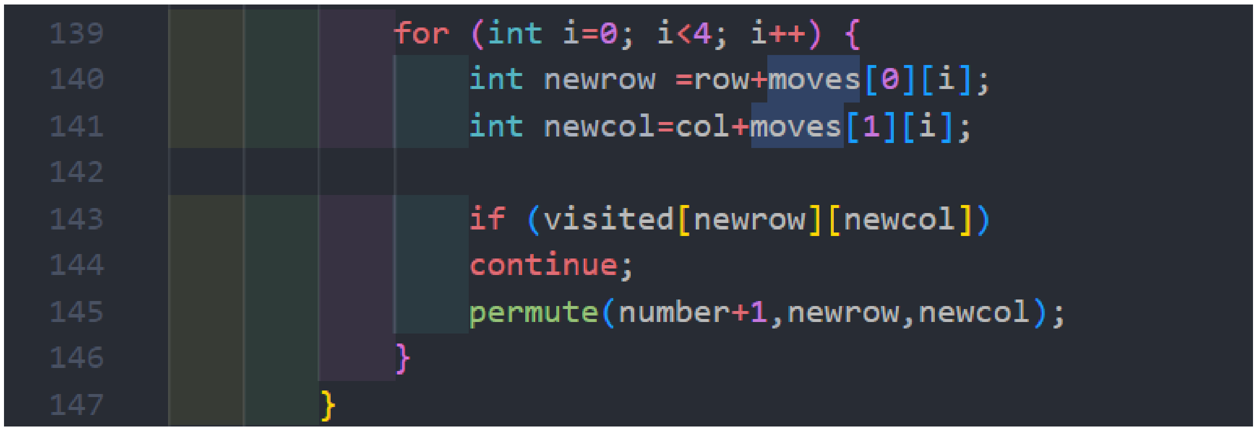
```
    int [][] moves= {{-1,0,1,0},{0,1,0,-1}};
```

<Figure 4> moves array implementation

- Meaning of Each Value

Directional Movement	moves[0][i]: Row change	moves[1][i]: Column change	Explanation
Up, U	-1	0	Row decreases by -1 (move up)
RIght, R	0	1	Column increases by +1 (move right)
Down, D	1	0	Row increases by +1 (move down)
Left, L	0	-1	Column decreases by -1 (move left)

- **Usage Example**:



```
        for (int i=0; i<4; i++) {
            int newrow =row+moves[0][i];
            int newcol=col+moves[1][i];

            if (visited[newrow][newcol])
            continue;
            permute(number+1,newrow,newcol);
        }
    }
```

<Figure 5> Directional Exploration Using Moves Array

- i = 0 (Up): newRow = row - 1, newCol = col (move up).
- i = 1 (Right): newRow = row, newCol = col + 1 (move right).
- i = 2 (Down): newRow = row + 1, newCol = col (move down).
- i = 3 (Left): newRow = row, newCol = col - 1 (move left).

- **Execution Example**:
  Assuming the current position is (3, 3) and we are checking unvisited cells:
  row = 3, col = 3

  moves[0][0], moves[1][0] → Up: (2, 3)

  moves[0][1], moves[1][1] → Right: (3, 4)

  moves[0][2], moves[1][2] → Down: (4, 3)

  moves[0][3], moves[1][3] → Left: (3, 2)

- **Reason for Selection**:
  The moves array is highly effective in solving four-directional exploration problems due to the following reasons:
  1. **Simplified Loops**:
     Without the moves array, conditional statements for each direction would significantly increase code complexity and execution time. By combining the array with a loop, we improve code readability and reduce complexity.
  2. **Consistent Direction Handling**:
     The array allows all directions to be processed in a uniform manner which avoid the need for separate logic for each direction.

3. **High Performance**:
   Since no conditional checks are needed for direction handling, the exploration time is reduced.
   - Array Access: O(1).
   - Four-directional exploration within the loop: O(4).

2.2 Algorithms Used

2.2.1 Backtracking

- A search technique that explores all possible paths but efficiently terminates paths that do not meet the required conditions early.
- It is used to find paths that visit each cell exactly once.

2.2.2 Pruning

- A technique used to reduce computational overhead by preemptively eliminating invalid paths during the search process.
- It prevents unnecessary recursive calls and optimizes execution time.

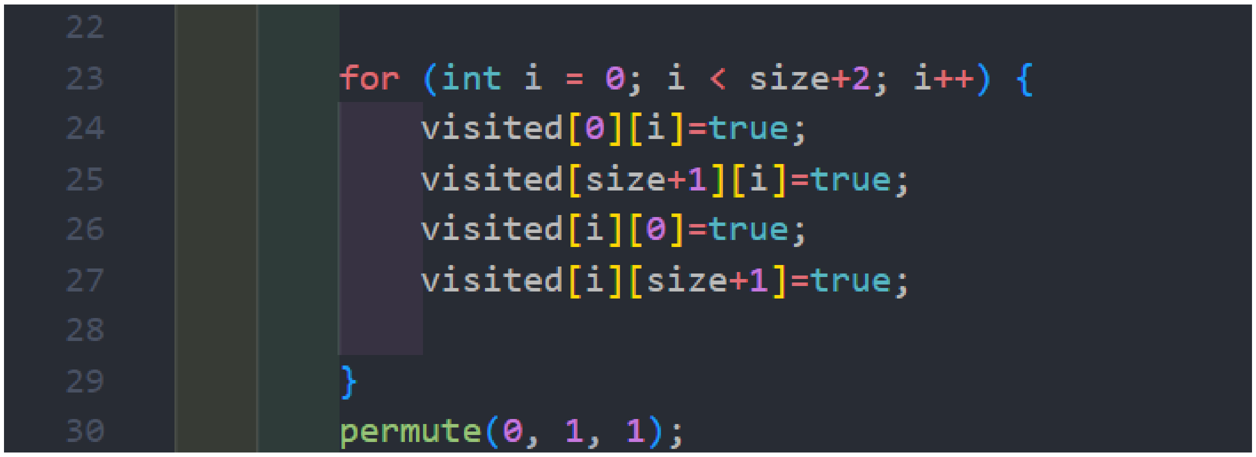
2.2.3 Special Character '*' Handling

- When the command is '*' instead of a specific direction, the algorithm explores all four directions, thereby expanding the search space.

2.3 How the Algorithm Works and is Implemented

2.3.1 Initialization of Exploration

- External boundaries are added to the grid to block movements beyond the 8x8 grid.
- Exploration starts from the initial point (1, 1).
- Code Implementation:

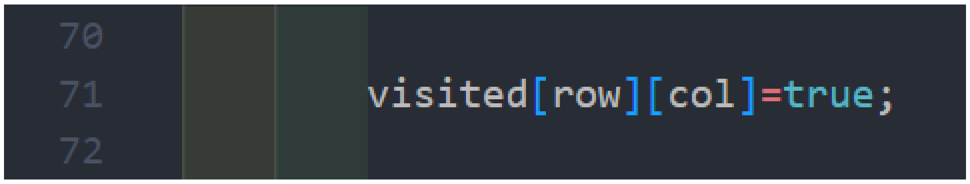


```
for (int i = 0; i < size+2; i++) {
    visited[0][i]=true;
    visited[size+1][i]=true;
    visited[i][0]=true;
    visited[i][size+1]=true;

}
permute(0, 1, 1);
```

<Figure 6> Initialization of Exploration implementation

2.3.2 Recursive Path Exploration

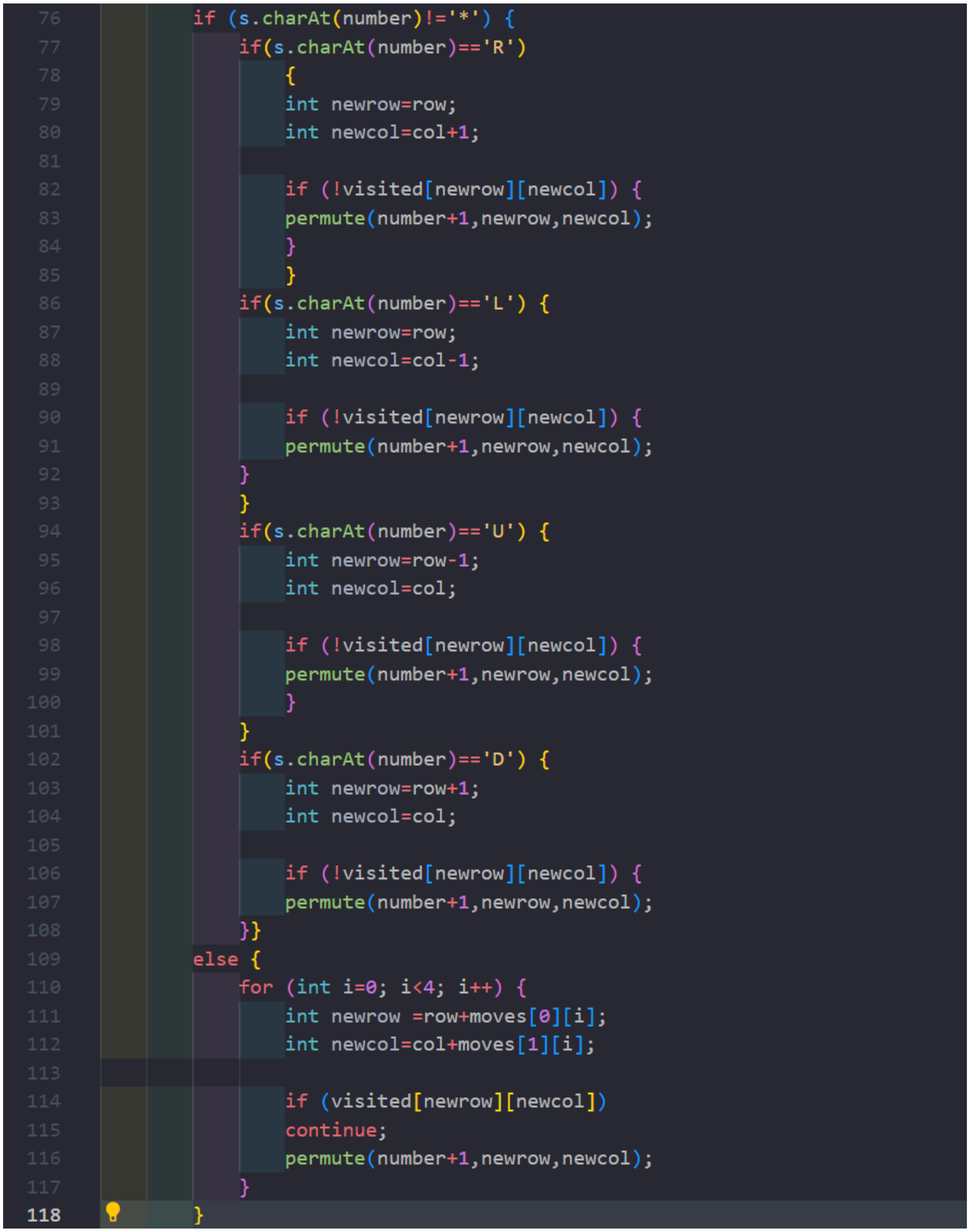
1. **Mark the Current Position as Visited**:
   - Purpose: Prevents revisiting the same cell
   - Code Implementation:



```
visited[row][col]=true;
```

<Figure 7> Updating Visited State for the Current Cell

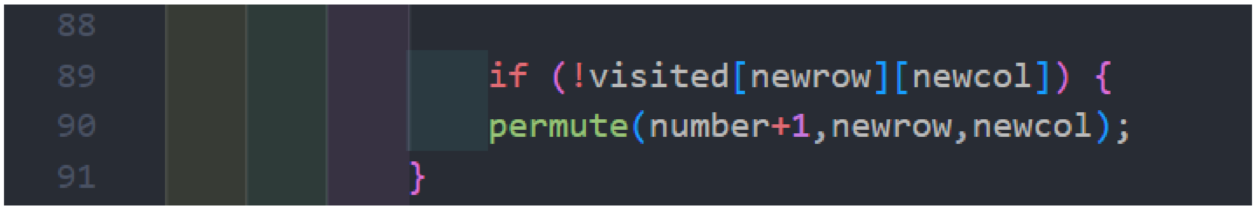
2. **Direction Exploration**:
   - If s.charAt(number) corresponds to a specific direction (‘R’, ‘L’, ‘U’, ‘D’), the algorithm moves in that direction.
   - If *, the algorithm explores all four directions.
   - Code Implementation:



```java
if (s.charAt(number)!='*') {
    if(s.charAt(number)=='R')
        {
        int newrow=row;
        int newcol=col+1;

        if (!visited[newrow][newcol]) {
        permute(number+1,newrow,newcol);
        }
        }
    if(s.charAt(number)=='L') {
        int newrow=row;
        int newcol=col-1;

        if (!visited[newrow][newcol]) {
        permute(number+1,newrow,newcol);
    }
    }
    if(s.charAt(number)=='U') {
        int newrow=row-1;
        int newcol=col;

        if (!visited[newrow][newcol]) {
        permute(number+1,newrow,newcol);
        }
    }
    if(s.charAt(number)=='D') {
        int newrow=row+1;
        int newcol=col;

        if (!visited[newrow][newcol]) {
        permute(number+1,newrow,newcol);
    }}
else {
    for (int i=0; i<4; i++) {
        int newrow =row+moves[0][i];
        int newcol=col+moves[1][i];

        if (visited[newrow][newcol])
        continue;
        permute(number+1,newrow,newcol);
    }
}
```

<Figure 8> Direction Exploration implementation

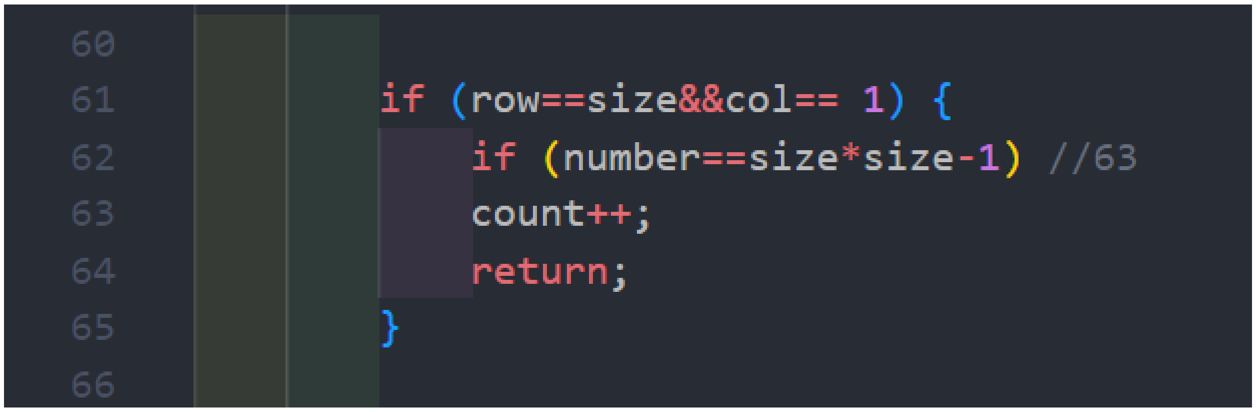
3. **Validation of Paths**:
   - Concept: Ensures that the path is valid by checking boundary and visitation constraints.
   - Code Implementation:



```
        if (!visited[newrow][newcol]) {
        permute(number+1,newrow,newcol);
    }
```

<Figure 9> Validation of Paths implementation

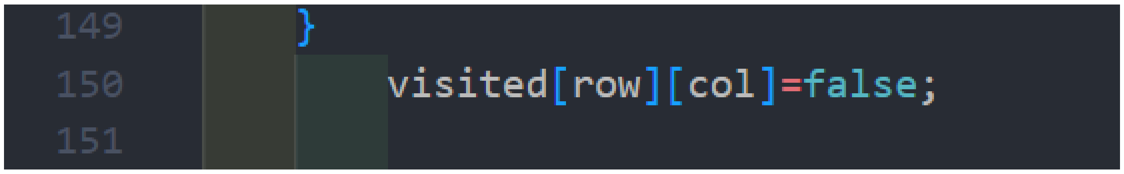
4. **Checking for Termination:**
   - Concept: Validates if all cells have been visited and the endpoint (size, 1) is reached.
   - Code Implementation:



```

    if (row==size&&col== 1) {
        if (number==size*size-1) //63
        count++;
        return;
    }

```

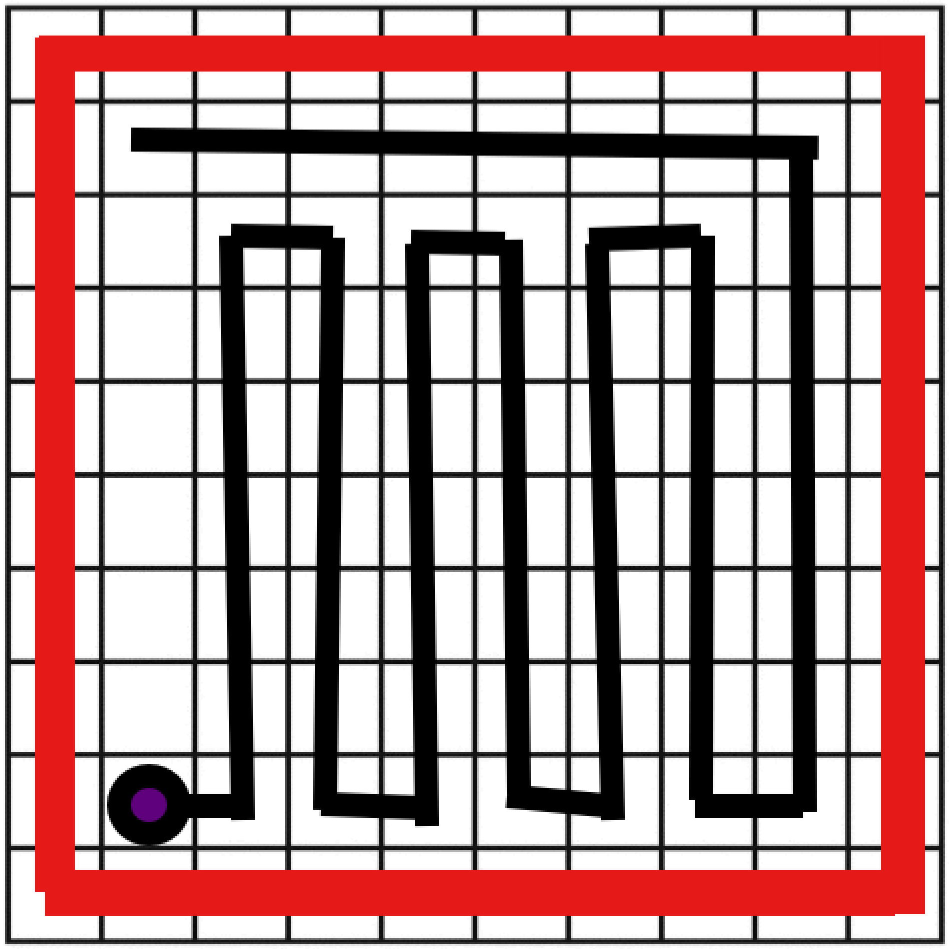
<Figure 10> Checking for Termination implementation

5. **Backtracking**
   - **Concept:** Resets the current cell to "unvisited" for exploring alternative paths.
   - Code Implementation:



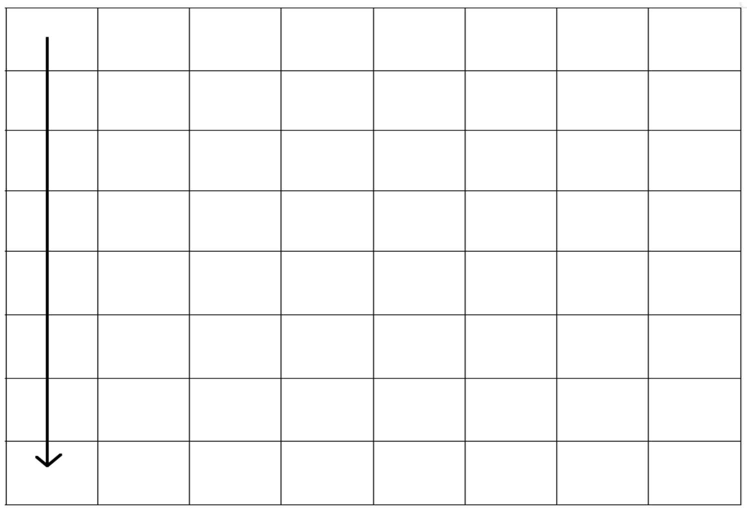
```
    }
        visited[row][col]=false;

```

<Figure 11> Backtracking - Unmarking Visited Cells



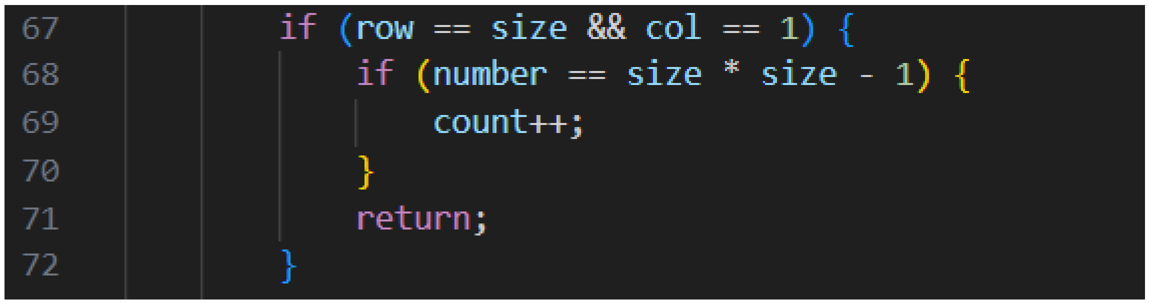
<Figure 12> Backtracking - example(The purple dots indicate where backtracking occurs.)

6. **Pruning to Reduce Search Space**:
   - **Concept**: This technique eliminates invalid paths early, reducing unnecessary recursive calls and calculations.
   1. It is obvious that the solution cannot be finished if the path arrives at the lower-left square before it has passed through every other square in the grid. The following path serves as an illustration of this:



<Figure 13>Example of hitting the target cell

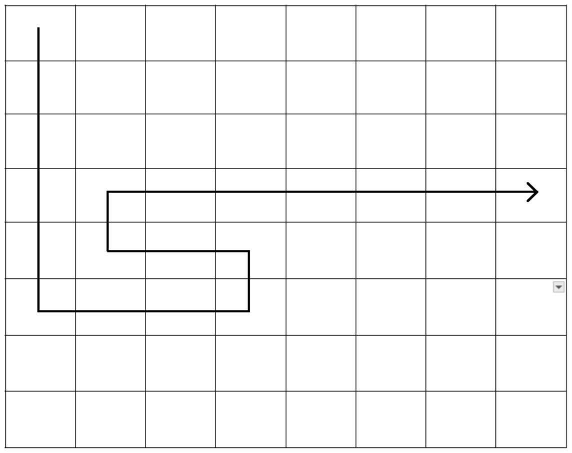
- **Scenario**:
    - When it reaches (8,1),if the number of cells traveled is correct (size*size-1), then increase count by 1, otherwise stop that case because it is wrong.

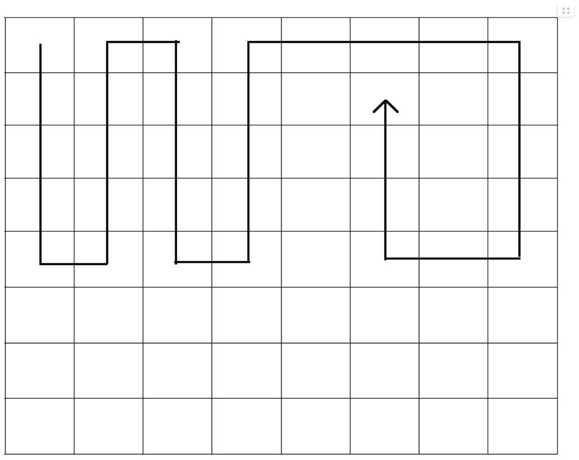


```
        if (row == size && col == 1) {
            if (number == size * size - 1) {
                count++;
            }
            return;
        }
```

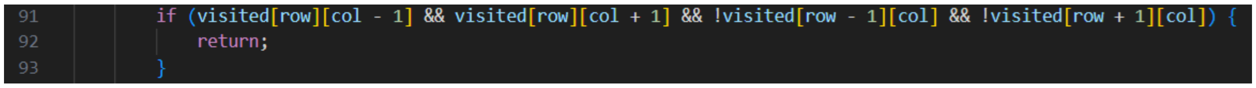
<Figure 14> Check to hit the target cell or not

2. The grid divides into two sections that both include unexplored squares if the path may turn left or right but cannot go forward or down (or if the path may go forward or downward but can not turn right or left). Take the following route, for instance:


<Figure 15>   Horizontal check


<Firgue 16>Vertical checking

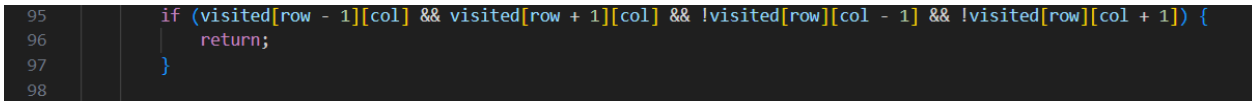
- **Scenario**:
    - The first case (Firgue 15):
    - The left (col - 1) and right (col + 1) cells have already been visited.
    - The upper (row - 1) and lower (row + 1) cells have not yet been visited.



```
        if (visited[row][col - 1] && visited[row][col + 1] && !visited[row - 1][col] && !visited[row + 1][col]) {
            return;
        }
```

<Figure 17> The statement check if the path is divided into 2 sides with unvisited squares horizontally

-
    - The second case (Firgue 16):
    - The upper (row -1) and lower (row +1) cells have already been visited.
    - The left(col -1) and right (col+1) cells have not yet been visited

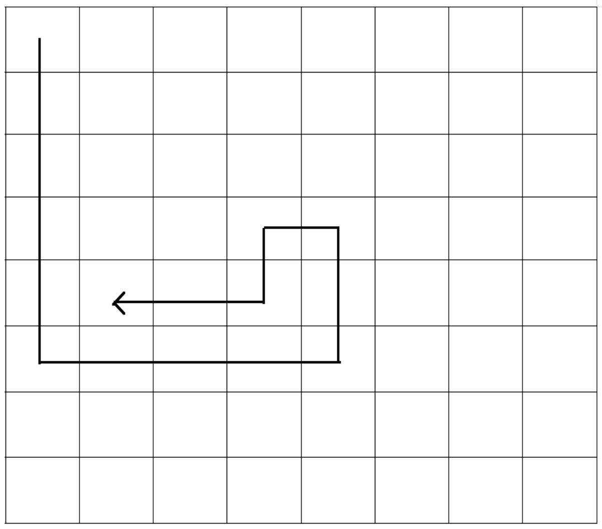


```
        if (visited[row - 1][col] && visited[row + 1][col] && !visited[row][col - 1] && !visited[row][col + 1]) {
            return;
        }

```

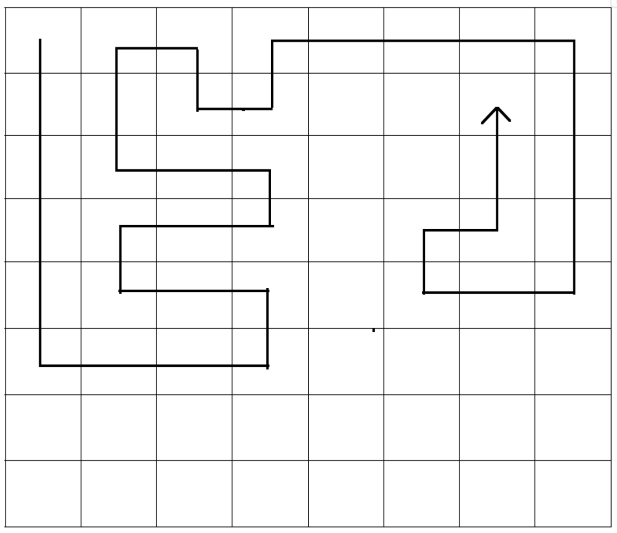
<Figure 18> The statement check if the path is divided into 2 sides with unvisited squares vertically

In such cases, when moving to the next cells until it can no longer be considered, it will definitely not be able to complete all the cells. Therefore, the function uses return to immediately end the current recursive call while preventing unnecessary recursive calls.

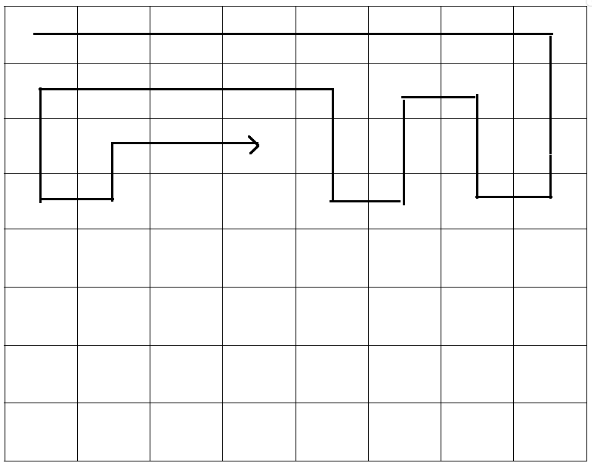
3. If the next step goes through a direction that creates a dead end, we should give up and go back because the path will definitely not go through all the cells. The next step we are forced to go to that dead end cell.



<Figure 19> Leftward move case

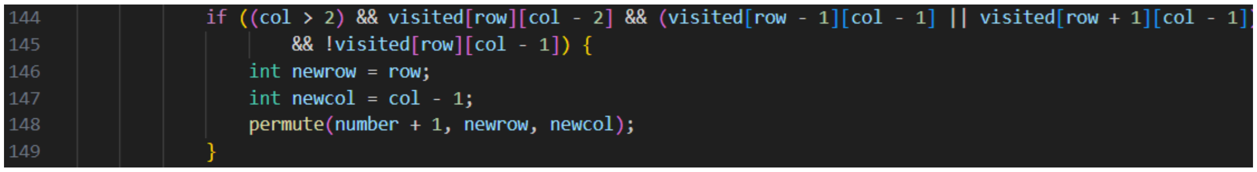


<Figure 20> Upward move case



<Figure 21>Rightward move case

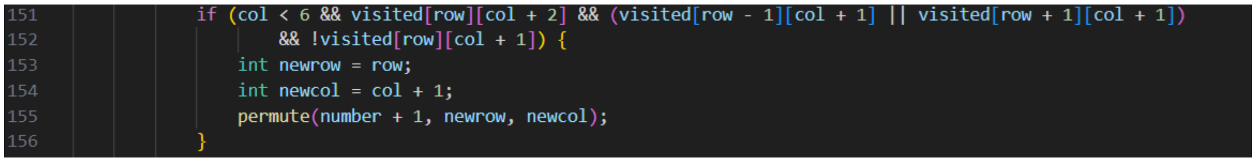
- **Scenario**:
  - **Leftward Move Conditions (col - 1) (Figure 19):**
    - The column > 2
    - The cell two steps to the left (col - 2) has already been visited
    - The diagonal cells to the left (row - 1, col - 1) or (row + 1, col - 1) has been visited
    - The cell to the left (col - 1) has not been visited



```
        if ((col > 2) && visited[row][col - 2] && (visited[row - 1][col - 1] || visited[row + 1][col - 1]
            && !visited[row][col - 1]) {
            int newrow = row;
            int newcol = col - 1;
            permute(number + 1, newrow, newcol);
        }
```

<Figure 22> The statement check if the move is forced to the left

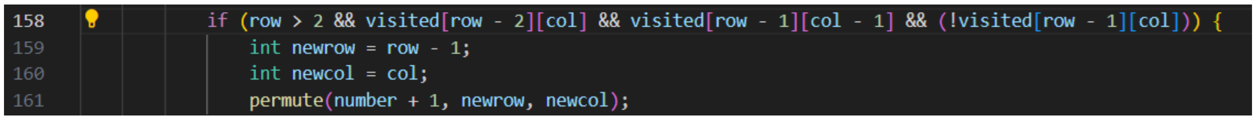
- **Rightward Move Conditions (col + 1) (Figure 21):**
  - The column < 6
  - The cell two steps to the right (col + 2) has already been visited
  - The diagonal cell to the right (row - 1, col + 1) or (row + 1, col + 1) has been visited
  - The cell to the right (col + 1) has not been visited:



```
        if (col < 6 && visited[row][col + 2] && (visited[row - 1][col + 1] || visited[row + 1][col + 1])
            && !visited[row][col + 1]) {
            int newrow = row;
            int newcol = col + 1;
            permute(number + 1, newrow, newcol);
        }
```

<Figure 23> The statement check if the move is forced to the right

- **Upward Move Conditions (row - 1) (Figure 20)**
  - The row > 2
  - The cell two steps up (row - 2) has already been visited
  - The diagonal cell to the left above (row - 1, col - 1) has been visited
  - The cell above (row - 1) has not been visited



```
        if (row > 2 && visited[row - 2][col] && visited[row - 1][col - 1] && (!visited[row - 1][col])) {
            int newrow = row - 1;
            int newcol = col;
            permute(number + 1, newrow, newcol);
```

<Figure 24> The statement check if the move is forced to go up

7. **Special Character Handling (*)**:
   - Concept: When '*' is encountered, the algorithm explores all four directions (up, down, left, right), expanding the search space.
   - For each direction, it calculates the new coordinates (newRow, newCol) and checks whether the position has not been visited.
   - Code Implementation: see Figure 5

3. Complexity Analysis

This section provides a detailed analysis of the algorithm which is used to determine the number of unique paths in a grid. The analysis will consider many aspects of the algorithm such as the algorithm's recursion stack, data structures, and how effective pruning invalid paths. In addition, the main objective of this analysis is to examine the best, worst, and average-case scenarios in terms of time complexity and space complexity.

3.1 Detailed Algorithm Analysis

The algorithm works by recursing through possible paths on the grid while accepting a lot of constraints. The main steps include:

- **Recursive Backtracking**: The algorithm generates permutations of moves on the grid. Additionally, its task generates possible paths from the starting position to the goal.
- **Visited Array**: A 2D array visited[][] is used to mark the cells that have been visited while the path is searched. This array plays an important role on checking the revisit cell and infinite loops
- **Move Control**: The algorithm controls movement by using 4 directions (left, right, up, down)
- **Path Counting**: The algorithm follows valid paths by ultilizing a counter variable count. It increments this counter whenever a valid path is found.
- **Stopping Conditions**:
    - **Reaching the Target**: When the destination cell is reached, the algorithm terminates that path and updates the path count or not.
    - **Dead Ends**: The algorithm backtracks if a path reaches a point where no valid moves exists
    - **Split :** If a path reaches a point which is visited and divide 2 parts which are both unvisited.

3.2 Complexity analysis

- Assumption
    - Let **N** be the size of each side of the grid.
    - Let **B** represent the average branching factor which indicates the number of valid directions available at each cell.

3.2.1 Complexity of space:

Certainly, all the memory allocations, such as variables and data structures, should be considered to analyze the space complexity of the algorithm.

- **visited array**

- The visited[][] array follows each cell that has already been visited or not. It has a size of (N+2) * (N+2) to control the width and avoid going out of the border.

  => The space complexity of this array is **O(N^2)**.

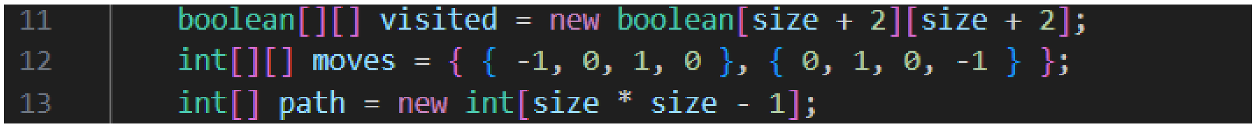
- **moves array**:
  - The moves array holds the movement directions (left, right, up, down), which has a fixed size of 4.

    => The space complexity of this array is **O(1)**.

- **Other variables**:
  - A lot of variables like number, row, col, and other loop variables are scalar which use about constant memory.

    => The space complexity of these variables is **O(1)**.

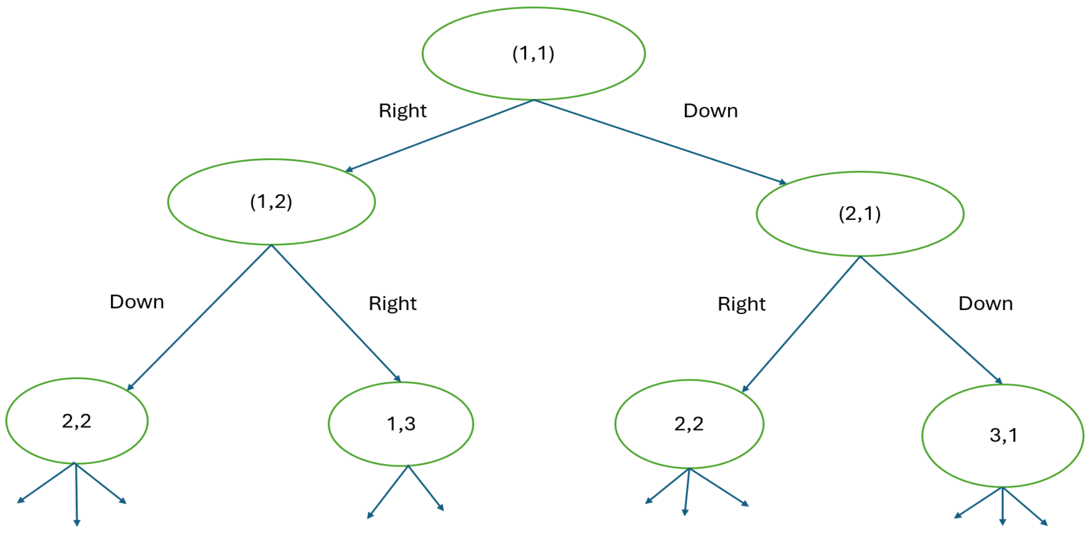
- **Overall of Space Complexity**
  - **The space complexity is: O(N^2)**



```
boolean[][] visited = new boolean[size + 2][size + 2];
int[][] moves = { { -1, 0, 1, 0 }, { 0, 1, 0, -1 } };
int[] path = new int[size * size - 1];
```

<Figure 25> All of arrays which are used in algorithm

3.2.2 Time complexity



<Figure 26> Tree representing the first path of every grid with size >3

The time complexity analysis involves considering the best, worst, and average-case scenarios. Let's analyze each case:

a. Best Case

This problem depends heavily on the input string. In the best case, we analyze 2 cases:

1. When the total number of paths is 0
2. When the total number of paths is greater than 0

- **Analysis**

With input-based analysis, assuming the grid size is n = 8 and the starting position is at cell, different scenarios with the input sequence can be analyzed as follows:

1. Unsuitable input

- Figure 26 illustrates the first cases when starting from cell (1, 1). It can be seen that from cell (1,1) The first 2 steps can only go right or down.
- If the first character is not valid (e.g., starting with U which is up from the initial position), the path will be immediately infeasible.
- It results in the following situations:

  +Pruning early: When the first character is not valid, the algorithm will return immediately.

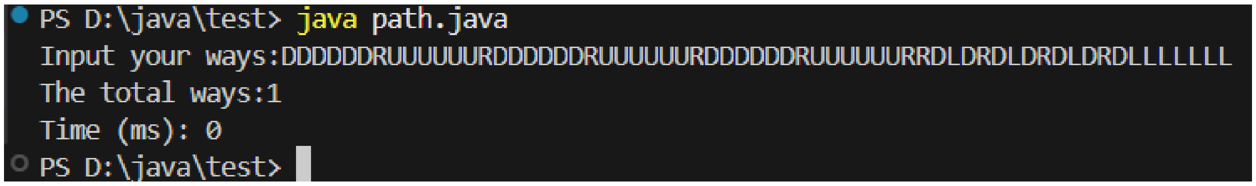
+Processing especially: Because it does not have to consider all of the invalid branches, the algorithm can be very fast for various strings.

+When the first step is U (up), the new position is (0,1) is out of the grid.

+Pruning from the start: Because the !visited[newrow][newcol] check fails,the permute function will not run subsequent branches.

=>Best case: almost O(1). as the path count is 0, and the algorithm terminates almost immediately.

2. **Optimal Input**
    - If the input was specifically a unique and valid path , the algorithm only needs to check one path.
    - For instance, if the input directly moves from (1, 1) to (8, 1) in a straight line, it only visits a single path.
    - For example:



```
PS D:\java\test> java path.java
Input your ways:DDDDDDRUUUUUURDDDDDDRUUUUUURDDDDDDRUUUUUURRDLDRDLDRDLDRDLLLLLLL
The total ways:1
Time (ms): 0
PS D:\java\test>
```

<Figure 27>

- Because the input is a specific description of a path, it will recurse with N^2-1 times.
- **Best case: O(N^2)**

b. Worst case

- **Re-analyse the algorithm:**
    - In the worst case, all the characters of the input string is '*'
- **The number of possible moves:**
    - At each cell of the grid, it can move in a maximum of 4 directions (left, right, up, down).
    - So, in the worst case, each cell is considered in all 4 directions without any restrictions.
    - Suppose there are no restrictions to traverse the entire grid

Possible Paths: 4^{N*N}

! But in reality, the complexity in this case is much smaller.

- **Why is the real case less?**

In the algorithm, the constraints reduce the number of recursions significantly:

1. **Visited state**
    - This eliminates many redundant paths. Therefore, after visiting a cell, the number of cells that can be visited will gradually decrease.
    - A cell can only be visited once in each path.
    - The total possible directions of all cells will be reduced by 1 or 2, because the outermost cells have only 2 paths of movement.
2. **Pruning Conditions:**
    - Dead ends and redundant paths are pruned while during exploration, so it often reduces choices at each step to 3, 2, or even 1.

- **Conclusion**
    - Instead of 4^{N*N}, the actual number of paths becomes proportional to B^{N*N} where $1< B < 3$

    Worst Case Complexity: $O(B^{N*N})$, where $1< B < 3$

c.Average Case:

- In general:
    - In the average case, the algorithm handles a mix of valid and invalid moves.
    - The pruning mechanism significantly reduces the number of explored paths.
- Performance
    - While it is difficult to calculate an exact complexity, the average case lies between the best-case and worst-case scenarios.
    - Due to the mix of valid and invalid moves, the algorithm's performance is improved compared to the worst case.

Average Case Complexity: $O(B^{N*N})$ where $1< B << 3$

3.3. OVERALL COMPLEXITY

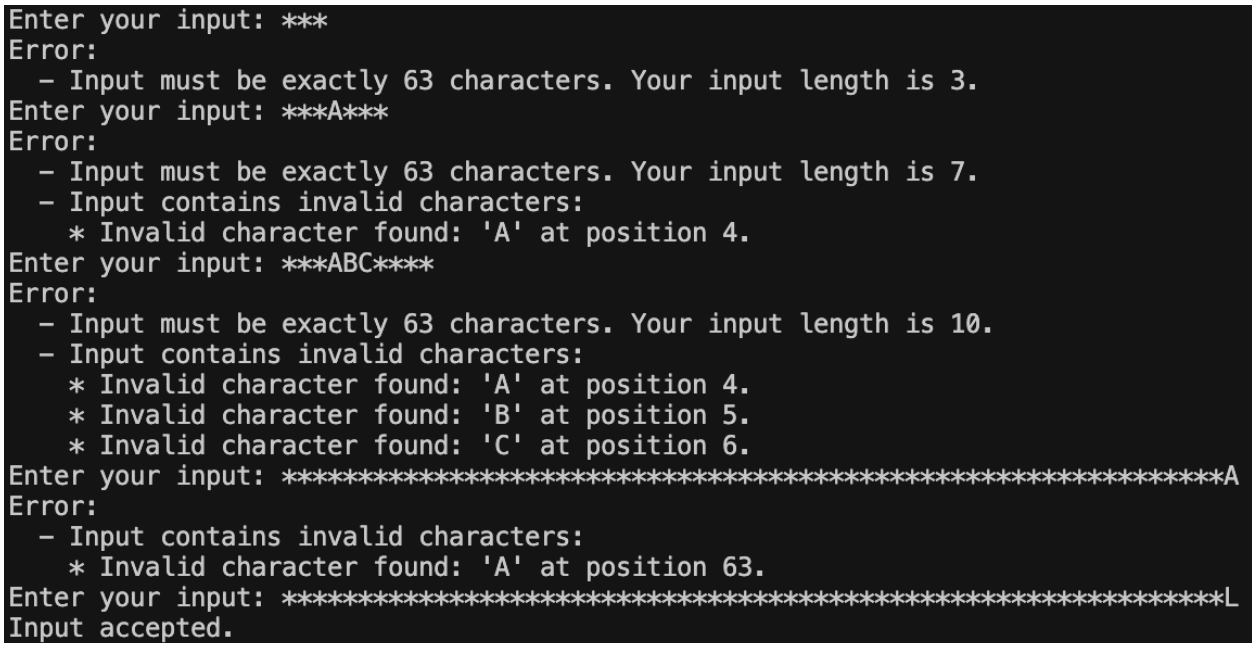
Case	**Time complexity**	**Space complexity**
Best case (None of total way)	O(1)	$O(N^2)$
Best case (More than 0 of total way)	$O(N^2)$	$O(N^2)$
Worst case	$O(B^{N*N})$ with $1<B<3$	$O(N^2)$
Average case	$O(B^{N*N})$ where $1< B << 3$	$O(N^2)$

4. Evaluation

4.1 Correctness Evaluation

The **Algorithm.java** file ensures that all inputs provided by the user are valid before the computation begins. This validation is performed with the following checks:

- The input string must be exactly 63 characters long.
- The input string can only contain the following characters: U (Up), D (Down), L (Left), R (Right), and *.
- If the user input fails either of these checks, the program clearly identifies the errors and provides feedback. For example:
- If the input length is incorrect, the program notifies the user of the exact number of characters entered.
- If invalid characters are present, the program pinpoints the character and its position.
- This ensures that only valid strings proceed to computation, reducing unnecessary errors and enhancing program robustness.



```
Enter your input: ***
Error:
  - Input must be exactly 63 characters. Your input length is 3.
Enter your input: ***A***
Error:
  - Input must be exactly 63 characters. Your input length is 7.
  - Input contains invalid characters:
    * Invalid character found: 'A' at position 4.
Enter your input: ***ABC****
Error:
  - Input must be exactly 63 characters. Your input length is 10.
  - Input contains invalid characters:
    * Invalid character found: 'A' at position 4.
    * Invalid character found: 'B' at position 5.
    * Invalid character found: 'C' at position 6.
Enter your input: **************************************************************A
Error:
  - Input contains invalid characters:
    * Invalid character found: 'A' at position 63.
Enter your input: **************************************************************L
Input accepted.
```

<Figure 28> Input validation example

The algorithm ensures that each path generated meets the following criteria:

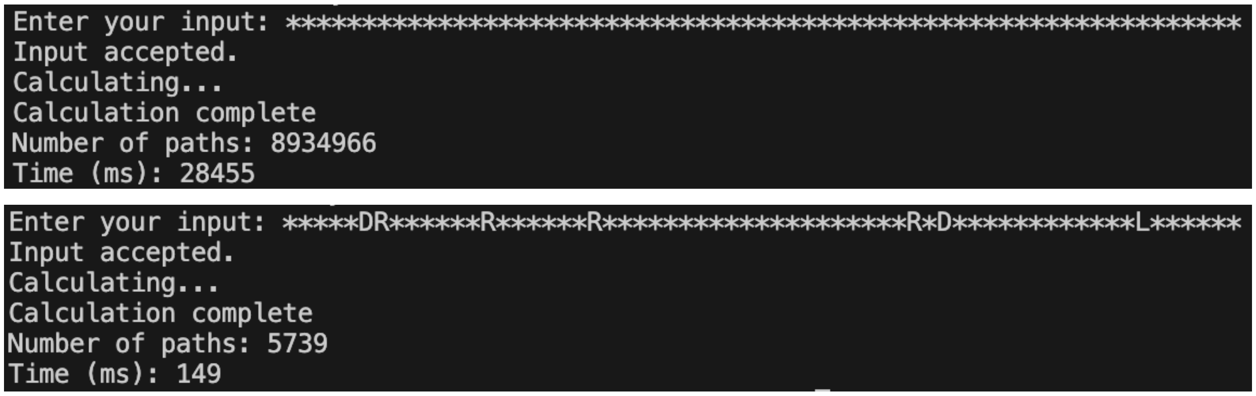
- The path must traverse the entire grid exactly once.
- The traversal must follow valid directions, without revisiting any cell or stepping outside the grid boundaries.
- The recursive method carefully explores all possibilities while respecting these rules. Any invalid path is automatically pruned, preventing further computation on it.

4.2. Performance Evaluation

Execution Time Measurement:

To evaluate the performance of the **Algorithm.java** implementation, the execution time is measured using System.currentTimeMillis() at the start and end of the start() method. This approach provides precise time tracking for various grid sizes.

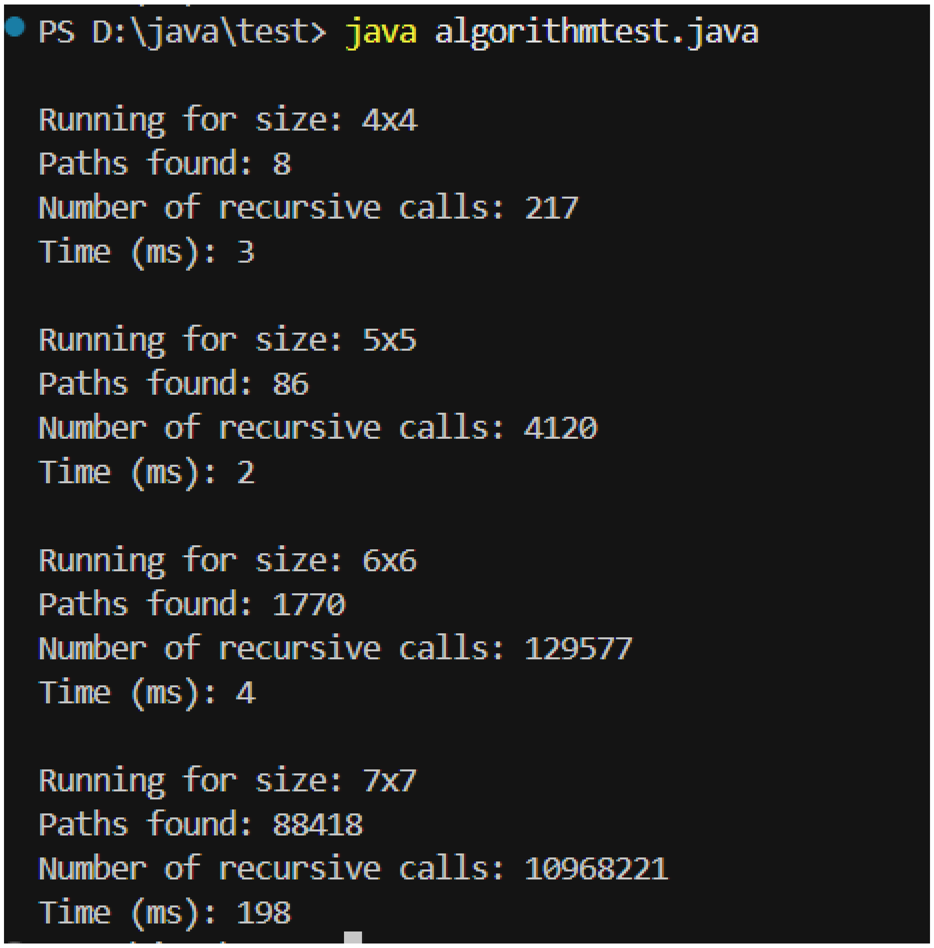
Below are the measured results for a detailed observation of the 8x8 environment (Figure 29) and a grid size from 4x4 to 7x7 (Figure 30).



```
Enter your input: ***************************************************************
Input accepted.
Calculating...
Calculation complete
Number of paths: 8934966
Time (ms): 28455
```

```
Enter your input: *****DR******R******R********************R*D************L******
Input accepted.
Calculating...
Calculation complete
Number of paths: 5739
Time (ms): 149
```

<Figure 29> **Algorithm.java** execution results



```
PS D:\java\test> java algorithmtest.java

Running for size: 4x4
Paths found: 8
Number of recursive calls: 217
Time (ms): 3

Running for size: 5x5
Paths found: 86
Number of recursive calls: 4120
Time (ms): 2

Running for size: 6x6
Paths found: 1770
Number of recursive calls: 129577
Time (ms): 4

Running for size: 7x7
Paths found: 88418
Number of recursive calls: 10968221
Time (ms): 198
```

<Figure 30> **AlgorithmTest.java** execution results for grid sizes 4x4 to 7x7

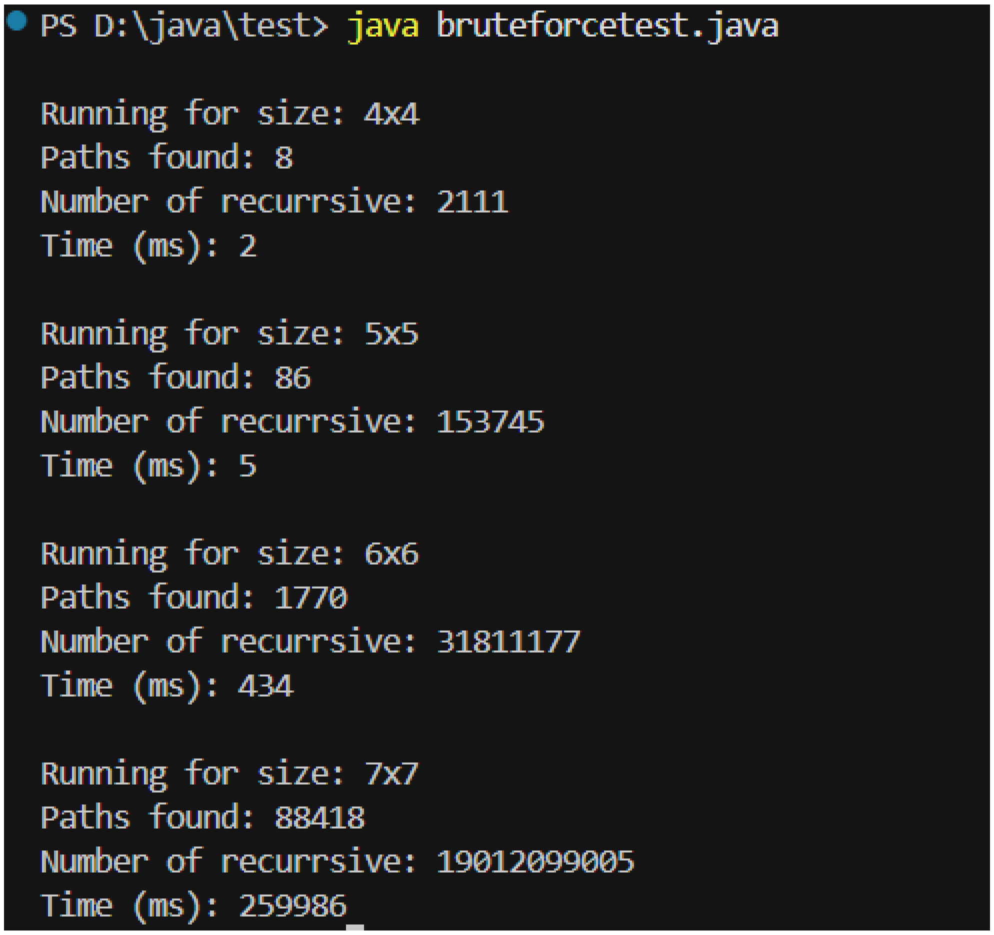
Grid Size	AlgorithmTest Paths	AlgorithmTest Time (ms)	AlgorithmTest Recursive calls
4x4	8	3	217
5x5	86	2	4,120
6x6	1,770	4	129,577
7x7	88,418	198	10,968,221

These results demonstrate that the optimized algorithm efficiently handles large grid sizes, providing accurate outputs within a reasonable timeframe.

4.3. Brute Force Comparison

To evaluate the efficiency of the implemented algorithm, a **BruteForceTest.java** file was created. The brute force approach generates all possible paths without optimization or pruning, leading to excessive computation time.

Below are the results of the **brute force** approach for grid sizes 4x4 to 7x7 (Figure 31):



```
PS D:\java\test> java bruteforcetest.java

Running for size: 4x4
Paths found: 8
Number of recurrsive: 2111
Time (ms): 2

Running for size: 5x5
Paths found: 86
Number of recurrsive: 153745
Time (ms): 5

Running for size: 6x6
Paths found: 1770
Number of recurrsive: 31811177
Time (ms): 434

Running for size: 7x7
Paths found: 88418
Number of recurrsive: 19012099005
Time (ms): 259986
```

<Figure 31> **BruteForceTest.java** execution results

Grid Size	BruteForceTest Paths	BruteForceTest Time (ms)	BruteForceTest Recursive calls
4x4	8	2	2111
5x5	86	5	153745
6x6	1,770	434	31,811,177
7x7	88,418	259,986	19,012,099,005

- Comparison:

To illustrate the performance difference between the optimized algorithm(AlgorithmTest.java) and the brute force approach(BruteForceTest.java), the results for grid sizes 4x4 to 7x7 are summarized in two tables below:

Grid Size	AlgorithmTest Paths	AlgorithmTest Time (ms)	BruteForceTest Paths	BruteForceTest Time (ms)
4x4	8	3	8	2
5x5	86	2	86	5
6x6	1,770	4	1,770	434
7x7	88,418	198	88,418	259,986

The table clearly highlights the efficiency of the optimized algorithm. For instance:

In the 7x7 grid, the optimized algorithm completes execution in 198 ms, whereas the brute force approach requires 259,986 ms (~6.3 minutes).

Grid Size	AlgorithmTest Number of recursive calls	BruteForceTest Number of recursive calls
4x4	217	2,111
5x5	4,120	153,745
6x6	129,577	31,811,177
7x7	10,968,221	19,012,099,005

Finally, the table above shows the number of times the program makes recursive calls. Each recursive call creates a new version of the function on the call stack which waste time and memory. It can be seen that the number of recursions of the brute force is overwhelmingly higher than the optimized algorithm.

- **Review of comparison:**

This significant difference demonstrates how pruning and optimization techniques drastically reduce redundant computations and execution time, making the optimized algorithm far superior for larger grid sizes.

The brute force approach explores every possible path exhaustively, which becomes infeasible for larger grid sizes due to its exponential time complexity. In contrast, the optimized algorithm efficiently prunes invalid paths early, ensuring superior performance in both time and memory usage.

4.4. Evaluation Summary

The evaluations above highlight the correctness and efficiency of the implemented algorithm. Through robust input validation and optimized path exploration, the **AlgorithmTest.java** significantly outperforms a brute force approach, especially for larger grid sizes. The performance evaluation and comparison demonstrate that the algorithm is both reliable and efficient.

5. Conclusions

5.1. Advantages:

Efficient problem solving:

The backtracking algorithm prunes unnecessary branches, reducing the possibilities to compute.

Systematic search:

Backtracking ensures a systematic search of the solution. It guarantees that all the solutions will
be explored.

Clear and Intuitive:

The algorithm is conceptually more simple and easier to implement than other algorithms.

Complex constraints handle capability:

Backtracking excels in solving problems with movement constraints.

5.2. Limitations:

Grid Size Limitations

The algorithm developed by our group is specifically optimized for the 8x8 grid requirement outlined in the assignment. However, due to the recursive nature of the backtracking algorithm, the execution time increases exponentially when applied to larger grids, such as 10x10 or 12x12.

Increased Memory Usage

To prevent the algorithm from traversing out-of-bounds, we expand the visited array to a size of (N+2) * (N+2). While this is efficient for smaller grids, it leads to significantly higher memory consumption for larger grid sizes, making the approach less scalable.

Lack of Optimization

The current implementation relies on a straightforward backtracking approach combined with basic pruning. The lack of advanced optimization techniques results in inefficiencies, particularly when dealing with larger search spaces.
For instance, as the number of * characters increase, the number of possible paths grows exponentially. In the worst-case scenario, this results in a time complexity of $O(B^{N*N})$ with $1<B<3$, significantly impacting performance in terms of both execution time and computational overhead.

5.3. Future Work:

Advanced Pruning for Reduced Search Space

We plan to implement an advanced pruning mechanism based on specific conditions to eliminate unnecessary exploration paths preemptively. This approach aims to minimize the overall search space and improve efficiency.

Researching Optimization Techniques

Our team will explore and study algorithms that can optimize the search process further. By identifying and integrating advanced optimization techniques, we aim to significantly reduce execution time while maintaining accuracy.

Integration of Parallel Processing

To enhance performance, we propose adopting a multi-threaded approach to explore multiple paths simultaneously. By leveraging parallel processing, execution time can be significantly reduced. Our team intends to study and implement parallel processing techniques to integrate them effectively into the current system.

Support for Flexible Grid Sizes

We aim to develop a mechanism for dynamically configuring grid sizes to support various dimensions beyond 8x8. Additionally, we will explore the use of bit masking or compressed data structures to optimize memory usage, enabling the system to scale efficiently for larger grids.

These improvements are expected to address the identified limitations, making the system more robust and scalable.

6.Reference:

1. Instructure.com. 2024 [cited 2024 Dec 20]. Available from: https://rmit.instructure.com/courses/149277/pages/w6-learning-materials?module_item_id=6669375
2. Instructure.com. 2024 [cited 2024 Dec 20]. Available from: https://rmit.instructure.com/courses/149277/pages/w3-learning-materials?module_item_id=6669353
3. MyApps Portal [Internet]. Instructure.com. 2019 [cited 2024 Dec 20]. Available from: https://rmit.instructure.com/courses/149277/pages/w4-learning-materials?module_item_id=6669360
4. Calculating Time Complexity | Data Structures and Algorithms| GeeksforGeeks