Computer and Systems Engineering [CSE],

Design and Analysis of Algorithms [CSE332s]

Project

Team Members

Name	ID
Mostafa Hamada Hassan Ahmed	2100587
Zeyad Magdy Roushdy	2100393
Mina Nasser Shehata	2100370
Ahmed Ashraf Ahmed	2100476
Mina Antony Samy Fahmy	2101022

Contents

List of Tables

List of Listings

1) Task (1)

1.1) Description

> **Description**
>
> Given a $2^n \times 2^n$ board with one square missing. The task is to tile the rest using right trominoes.

- Constraints:
  - Each tromino must cover exactly three squares.
  - Trominoes must be in one of three colors.
  - No two trominoes sharing an edge may have the same color.
  - Only divide-and-conquer techniques may be used.
- Key properties:
  - The board can always be divided into four equal quadrants.
  - The missing square appears in exactly one of the quadrants.
  - A tromino must be placed at the center to balance the three full quadrants, simulating a new missing square in each.
  - The process continues recursively until the board is reduced to a 2×2 base case.
  - Coloring rules must be maintained during placement:
  - Ensure that no two trominoes placed adjacent to each other have the same color.
  - Track neighboring colors during placement to maintain the constraint.
- The algorithm should work by:
  - Identifying the quadrant with the original missing square.
  - Placing a tromino at the center to create new “missing” squares in the other three quadrants.
  - Recursively applying the same steps in all four quadrants.
  - Assigning colors carefully during placement to avoid violations.

1.2) Problem Description

The task is to solve the tromino tiling problem for a square board of size $n \times n$ where n is a power of 2. The board contains exactly one missing square, and the goal is to cover the rest of

the board using L-shaped trominoes (shaped like an “L”) such that each tromino covers exactly 3 squares.

- You need to:
  - Fill the board with trominoes while leaving the missing square untouched.
  - The solution should work recursively, subdividing the board into smaller subproblems.
  - Output the board after the tiling is complete, displaying the order of tromino placements.
- Input Board size (`n`): An integer where `n` is a power of 2 (e.g., 2, 4, 8, 16, etc.). Hole coordinates: The coordinates (`holeX`, `holeY`) of the missing square, where `0 ≤ holeX < n` and `0 ≤ holeY < n`.
- Output
  - A square board of size `n × n` where:
  - The missing square is marked with 0.
  - Each square covered by a tromino is marked with a positive integer, representing the order in which the trominoes are placed.
- Constraints
  - The board size n must be a power of 2 and no larger than the maximum allowed size (which can be system-dependent).
  - The missing square must be valid (i.e., inside the board).
  - The algorithm must use divide and conquer to solve the problem recursively.

1.3) Detailed assumptions

- Board Size
  - The board is always square.
  - The size n must be a power of 2.
  - Values like 2, 4, 8, 16, 32, 64, 128… are valid.
  - Sizes like 3, 5, 6, 10… are rejected.
- Missing Square
  - Exactly one square on the board is missing.
  - Its position is given by (holeX, holeY).

  - It must be inside the board bounds.
- Tromino Type
  - The algorithm places L-shaped trominoes.
  - Each tromino covers 3 squares.
  - It leaves one square open for recursion.
- Recursive Logic
  - The board is divided into 4 equal quadrants.
  - The quadrant with the missing square is treated as-is.
  - A tromino is placed in the center to create a “virtual hole” in the other 3 quadrants.
  - Each quadrant is solved recursively.
  - Base case: 2×2 board, where the 3 other squares are filled directly.
- Memory Model
  - The board is stored as a dynamic 2D `vector<vector<int>>`.
  - There’s no hard-coded limit like `MAX_SIZE`.
  - Memory use grows with n^2.

1.4) Divide and Conquer Algorithm

1.4.1) Pseudocode

```text
step = 1;
function solve(size, topX, topY, holeX, holeY):
    if size == 2:
        for i from 0 to 1:
            for j from 0 to 1:
                if (topX + i, topY + j) is not (holeX, holeY):
                    board[topX + i][topY + j] = step
        step++
        return

    half = size / 2
    midX = topX + half
    midY = topY + half

    # Determine which quadrant the hole is in
    If hole is in top-left quadrant:    quad = 1
    Else if hole is in top-right quadrant:  quad = 2
```

```
    Else if hole is in bottom-left quadrant:  quad = 3
    Else:         quad = 4

    # Place an L-shaped tromino in the center
    # Place tromino at the center of the board, covering three squares in the middle of the four quadrants:
    If hole is not in top-left quadrant, place a square here.
    If hole is not in top-right quadrant, place a square here.
    If hole is not in bottom-left quadrant, place a square here.
    If hole is not in bottom-right quadrant, place a square here.
    step++;

    solve(half, topX, topY,(quad == 1) ? holeX : midX - 1, (quad == 1) ? holeY : midY - 1);
    solve(half, topX, midY,(quad == 2) ? holeX : midX - 1, (quad == 2) ? holeY : midY);
    solve(half, midX, topY,(quad == 3) ? holeX : midX, (quad == 3) ? holeY : midY - 1);
    solve(half, midX, midY,(quad == 4) ? holeX : midX, (quad == 4) ? holeY : midY);
```

Listing 1.1: Tromino Tiling Problem - Divide and Conquer Algorithm Pseudocode

1.4.2) Code

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

using namespace std;

int step = 1;
vector<vector<int>> board;


void printBoard(int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf("%3d ", board[i][j]);
        }
        cout << endl;
    }
    cout << endl;
```

```
}

void solve(int size, int topX, int topY, int holeX, int holeY) {
    if (size == 2) {
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                if (topX + i ≠ holeX || topY + j ≠ holeY) {
                    board[topX + i][topY + j] = step;
                }
            }
        }
        step++;
        return;
    }

    int half = size / 2;
    int midX = topX + half;
    int midY = topY + half;
    int centerX = midX - 1;
    int centerY = midY - 1;

    int quad = 0;
    if      (holeX < midX && holeY < midY) quad = 1;
    else if (holeX < midX && holeY ≥ midY) quad = 2;
    else if (holeX ≥ midX && holeY < midY) quad = 3;
    else quad = 4;

    if (quad ≠ 1) board[centerX][centerY] = step;
    if (quad ≠ 2) board[centerX][centerY + 1] = step;
    if (quad ≠ 3) board[centerX + 1][centerY] = step;
    if (quad ≠ 4) board[centerX + 1][centerY + 1] = step;
    step++;

    solve(half, topX, topY,(quad == 1) ? holeX : midX - 1, (quad == 1) ?
    holeY : midY - 1);
    solve(half, topX, midY,(quad == 2) ? holeX : midX - 1, (quad == 2) ?
    holeY : midY);
    solve(half, midX, topY,(quad == 3) ? holeX : midX, (quad == 3) ?
    holeY : midY - 1);
    solve(half, midX, midY,(quad == 4) ? holeX : midX, (quad == 4) ?
    holeY : midY);
}
```

```cpp

int main() {
    srand(time(0));

    int size;
    cout << "Enter board size (power of 2): ";
    cin >> size;

    if ((size & (size - 1)) != 0 || size <= 0) {
        cout << "Size must be a power of 2 and greater than 0." << endl;
        return 1;
    }

    int holeX, holeY;
    cout << "Enter missing square coordinates (row col): ";
    cin >> holeX >> holeY;

    if (holeX < 0 || holeX >= size || holeY < 0 || holeY >= size) {
        cout << "Invalid coordinates for the hole." << endl;
        return 1;
    }

    board.resize(size, vector<int>(size, -1)); // Initialize board with -1
    board[holeX][holeY] = 0; // Mark the hole position as 0

    solve(size, 0, 0, holeX, holeY);
    printBoard(size);

    return 0;
}
```

Listing 1.2: Tromino Tiling Problem - Divide and Conquer Algorithm C++ Code

1.4.3) Complexity Analysis

1.4.3.1) Time Complexity

The recurrence relation is:

$$T(n) = 4T\left(\frac{n}{2}\right) + O(1) \tag{1.1}$$

Using the Master Theorem:

$$a = 4 \tag{1.2}$$

$$b = 2 \tag{1.3}$$

$$d = 0 \tag{1.4}$$

$$T(n) = O\big(n^{\log_{b(a)}}\big) = O(n^2) \tag{1.5}$$

1.4.3.2) Space Complexity

The space complexity comes from two sources:

- Recursive Call Stack:
  - The depth of the recursion tree is $\log(n)$ since the problem size is halved at each level.
  - Each recursive call uses constant space, giving $O(\log(n))$ total stack space.
  - Board Storage:
    - The board is a 2D array of size $n \times n$, requiring $O(n^2)$ space.

Total space complexity:

$$O(n^2) + O(\log n) = O(n^2) \tag{1.6}$$

1.4.4) Test Cases

- Case (1): Board Size = 4x4
  - Input:
    - Board Size: 4 (so the board is 4x4).
    - Missing Square: (1, 1) (the missing square is at row 1, column 1).
  - Output:

```
Output
2  2  3  3
2  0  1  3
4  1  1  5
4  4  5  5
```

Listing 1.3: Tromino Tiling Problem - Divide and Conquer Algorithm Test Case (1) Output

  - Explanation:
    - The board is 4x4, and the missing square is at (1, 1).
    - The algorithm starts by dividing the board into 4 quadrants.
    - In each quadrant, it recursively places trominoes, ensuring the missing square remains unfilled.
    - In this case, the trominoes are placed in 6 steps (steps 1 to 6).

- Case 2: Board Size = 8x8
  - Input:
    - Board Size: 8 (so the board is 8x8).
    - Missing Square: (3, 3) (the missing square is at row 3, column 3).
  - Output:

Output

```
 3   3   4   4   8   8   9   9
 3   2   2   4   8   7   7   9
 5   2   6   6  10  10   7  11
 5   5   6   0   1  10  11  11
13  13  14   1   1  18  19  19
13  12  14  14  18  18  17  19
15  12  12  16  20  17  17  21
15  15  16  16  20  20  21  21
```

- Explanation:
  - The board is 8x8, and the missing square is at (3, 3).
  - The algorithm recursively divides the board into 4 smaller 4x4 subgrids and continues dividing them until the size becomes 2x2, where it directly fills the board with trominoes.
  - The solution uses 16 steps to fill the board. Each number represents a step where a tromino is placed, and the missing square (0) is carefully avoided.

1.5) Brute-Force Algorithm

```text
function knightTour(n):
    board = n x n array filled with -1
    moves = [(2, 1), (1, 2), (-1, 2), (-2, 1),
             (-2, -1), (-1, -2), (1, -2), (2, -1)]
    startX, startY = 0, 0
    board[startX][startY] = 0
    if solve(board, startX, startY, 1, moves, n, startX, startY):
        return board
    return "No solution"
function isValid(x, y, board, n):
    return x ≥ 0 and y ≥ 0 and x < n and y < n and board[x][y] == -1
function solve(board, x, y, moveCount, moves, n, startX, startY):
```

```
    if moveCount == n * n:
        for (dx, dy) in moves:
            nx = x + dx
            ny = y + dy
            if nx == startX and ny == startY:
                return true
        return false
    for (dx, dy) in moves:
        nx = x + dx
        ny = y + dy
        if isValid(nx, ny, board, n):
            board[nx][ny] = moveCount
            if solve(board, nx, ny, moveCount + 1, moves, n, startX,
            startY):
                return true
            board[nx][ny] = -1
    return false
```

Listing 1.4: Tromino Tiling Problem - Brute-Force Algorithm Pseudocode

1.6) Comparison Between Divide and Conquer Algorithm and Brute Force Approach

Aspect	Divide and Conquer Algorithm	Brute Force Approach
Overview	• The problem is broken down into smaller subproblems. • The grid is recursively divided into quadrants until a 2×2 board is reached. • Trominoes are placed while avoiding the missing square.	• The algorithm systematically tries to place trominoes, backtracking when • an invalid configuration is found (overlap or covering the missing square).
Time Complexity	$O(n^2)$ due to the recurrence relation $T(n) = 4T(\frac{n}{2}) + O(1)$	$O\left(3^{n^2}\right)$ in the worst case due to the number of potential tromino placements and backtracking
Space Complexity	$O(n^2)$ because a 2D board is used to store the grid state	$O(n^2)$ due to storing the board during backtracking
Advantages	• Efficient for large grid sizes due to logarithmic subdivisions • Recursive approach is natural for this problem • Optimal time complexity of $O(n^2)$ for grid traversal	• Works for any board size, even if not a power of two • Simple and easy to implement

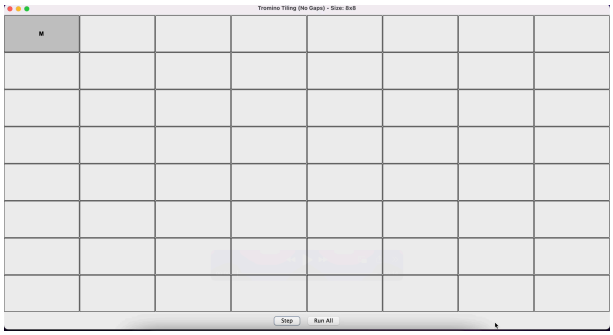
Aspect	Divide and Conquer Algorithm	Brute Force Approach
Disadvantages	• Recursive overhead can be high for large grids • Assumes board size is a power of two, limiting applicability	• Extremely inefficient for larger grids due to exponential growth • Impractical time complexity for large grids

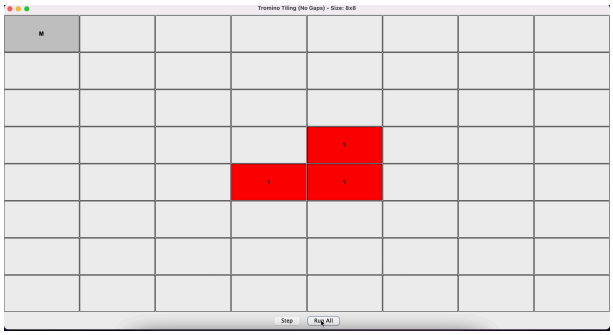
Table 1.1: Tromino Tiling Problem - Algorithm Comparison

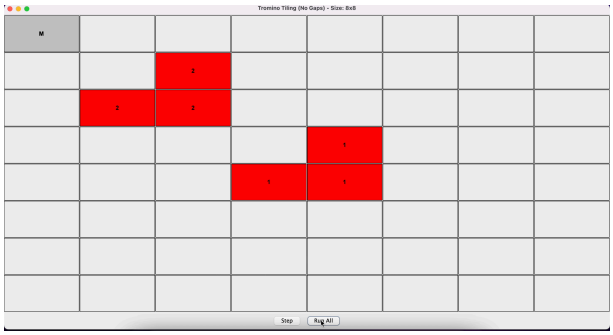
1.7) Conclusion

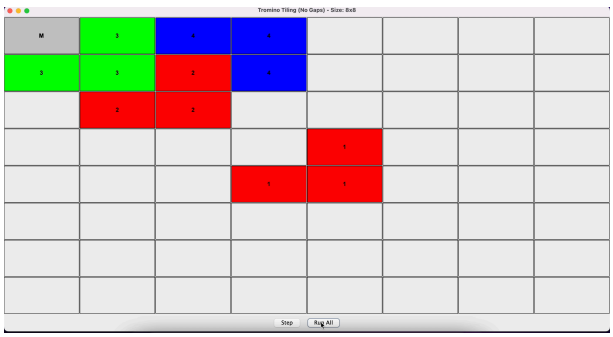
- The solution to the Tiling Problem using Divide and Conquer effectively breaks down a larger problem (tiling an n x n board) into smaller subproblems, ensuring efficient placement of L-shaped trominoes.
- The algorithm divides the board recursively into quadrants, filling each quadrant with trominoes while carefully avoiding the pre-designated missing square.
- Key points:
  - Recursive Nature: The algorithm recursively divides the board into smaller subgrids until it reaches the base case (a 2x2 board). At that point, it directly places the trominoes.
  - Efficiency: The problem is solved in O(n^2) time due to the recursive subproblem structure, which ensures that each cell is processed once.
  - Memory Usage: The algorithm uses O(n^2) space for storing the board, which is proportional to the input size.
  - Scalability: The solution works efficiently for larger boards, as demonstrated in the examples with increasing board sizes. It handles both small and large sizes gracefully.
  - Step-by-Step Solution: The output shows the step-by-step placement of trominoes, which highlights the correctness and clarity of the divide and conquer approach.
  - Overall, the algorithm provides a robust, scalable, and efficient solution for tiling a board with a missing square. It adheres to the Divide and Conquer paradigm, offering an optimal solution in terms of both time complexity and space utilization.

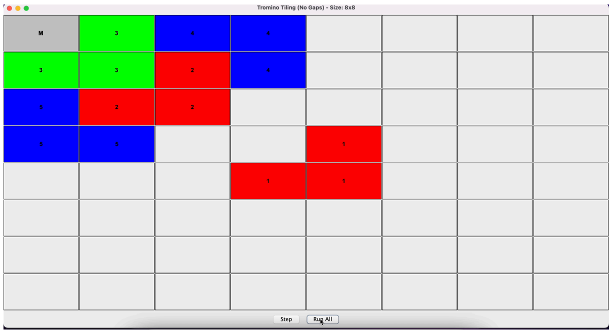
1.8) GUI Screenshots

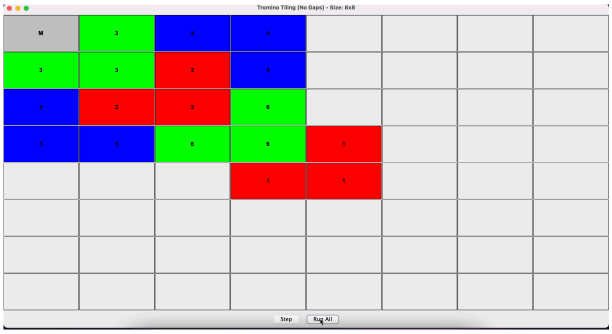


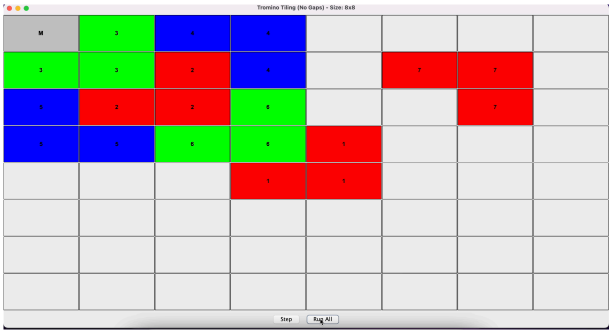


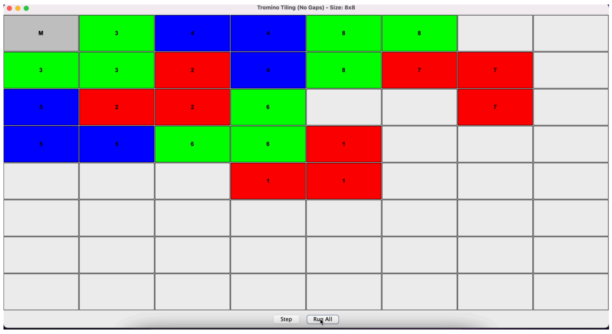


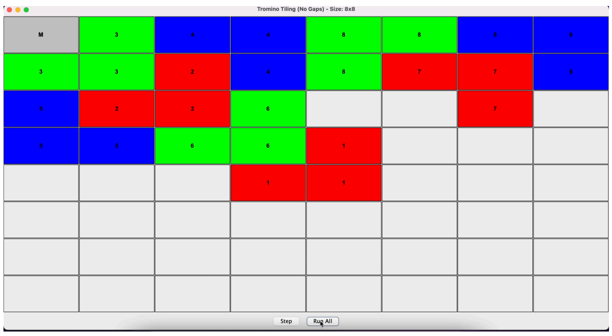


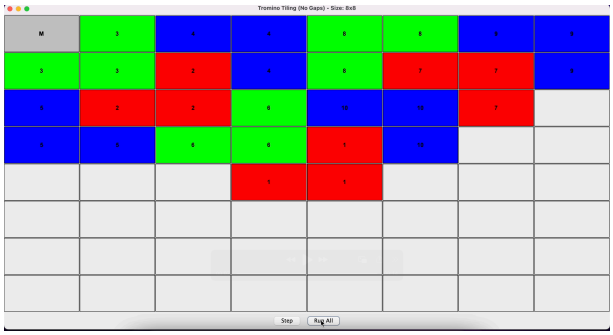


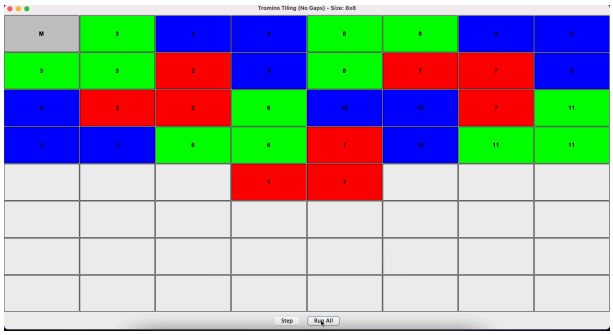


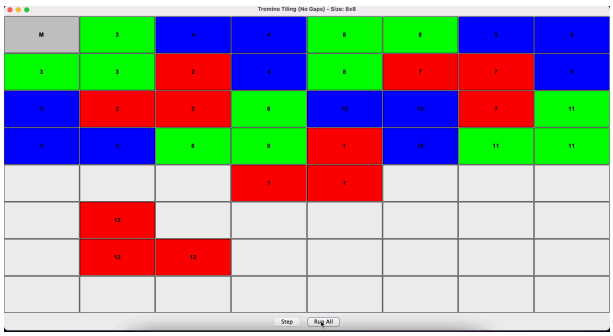


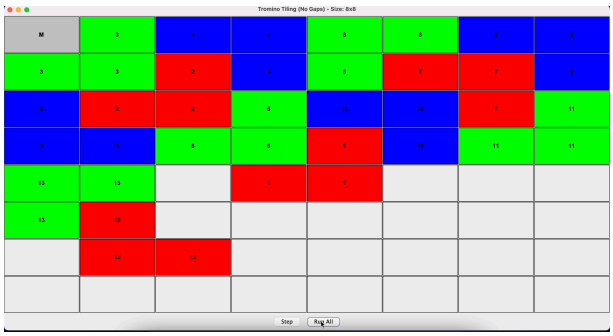


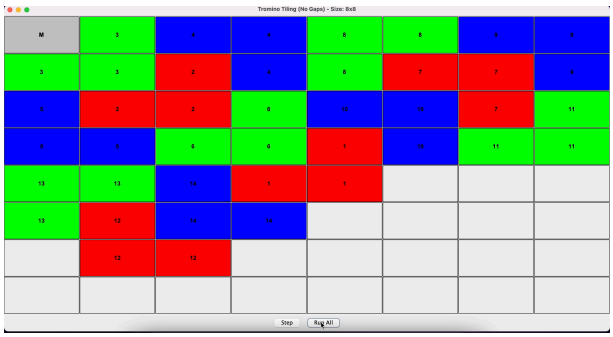


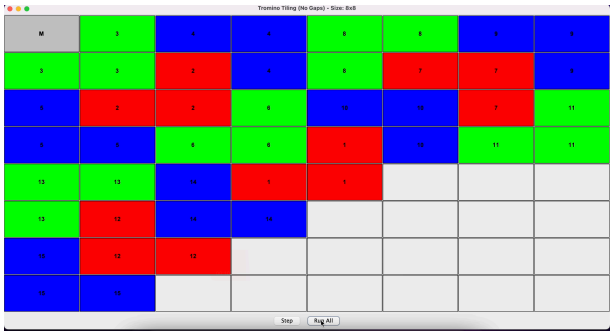


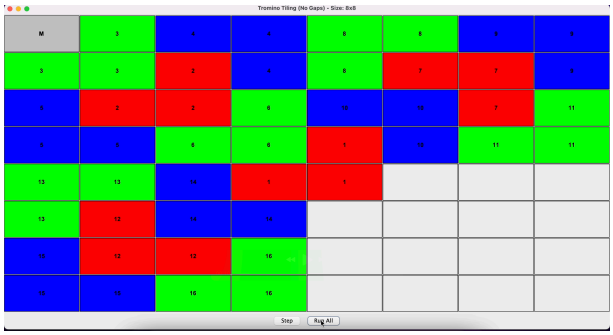


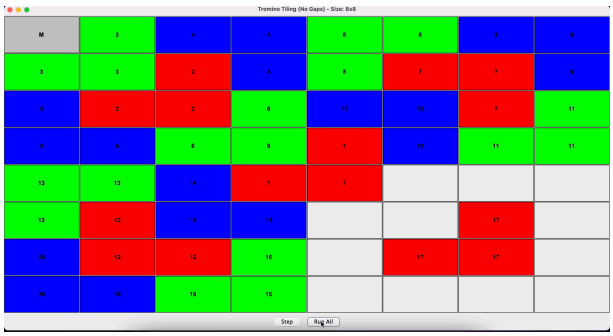


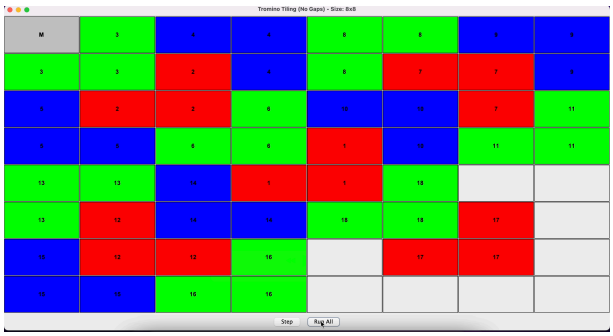


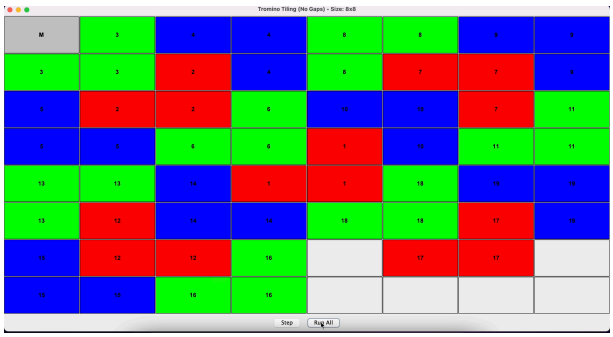


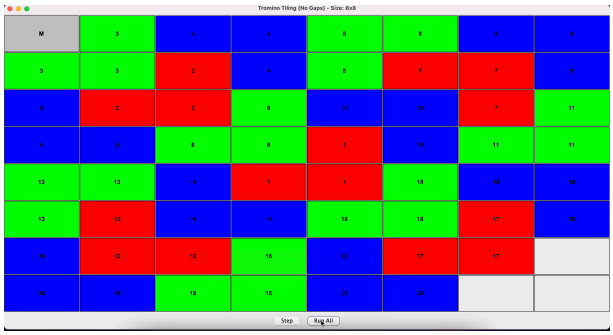


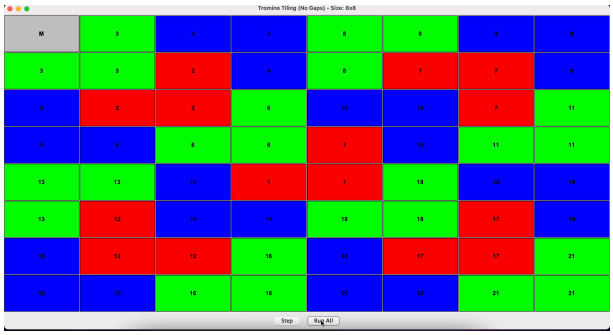









Tromino Tiling (No Gaps) - Size: 8x8
Step
Run All


Tromino Tiling (No Gaps) - Size: 8x8
Step
Run All

2) Task (2)

2.1) Description

> **Description**
>
> Given an 8 × 8 chessboard and a knight, the task is to determine if it can complete a closed tour, meaning:
>
> - The knight visits every square exactly once.
> - The knight ends one move away from the starting square.

- Key details:
  - The knight moves in L-shape: two squares in one direction and one perpendicular.
  - The knight must land on each square once.
  - A tour is closed if the final square is one legal knight move from the starting square.
- Known facts:
  - A closed knight's tour exists for standard 8 × 8 board.
  - Such tours have been constructed and proven possible.
  - Not all board sizes support closed or even open tours.
- Greedy algorithm approach:
  - Use Warnsdorff's Rule:
    - At each step, move the knight to the square with the fewest onward moves.
    - This reduces early trapping.
    - Initialize:
      - Choose a random or fixed starting cell.
      - Mark visited cells.
    - While not all cells visited:
      - List all valid knight moves.
      - Count unvisited onward moves for each.
      - Choose the move with minimum onward options.
      - Mark new square visited.
      - Record the move.

- Stop when all cells are visited or no move is possible.

2.2) Problem Description

- Objective
  - Find a sequence of moves for a knight on a $n \times n$ chessboard such that:
    - The knight visits every square exactly once.
    - The movement follows standard chess rules.
    - If no complete tour is found, the algorithm outputs the partial path.
- Input
  - n: Size of the board ($n \times n$).
  - (`startX`, `startY`): Initial position of the knight.
- Output
  - A printed board where:
    - `0` = Starting position.
    - `1` = Visited squares.
    - `X` = Unvisited squares.
  - A message indicating whether a full tour was found.
- Constraints
  - The board size n must be ≥ 1.
    - → 1×1 Board (Trivial Case)
    - → 2×2 Board (Too Small) A knight needs at least 3 squares in one direction to make an L-shaped move.
    - → 3×3 Board (Too Small) A knight will not visit every square exactly once whatever its Initial position
    - → 4×4 Board (Too Small) A knight will not visit every square exactly once whatever its Initial position
  - The starting position must be within bounds.
  - The algorithm does not guarantee a solution for all n (Warnsdorff's rule is heuristic-based).

2.3) Detailed assumptions

- Board Size

  - The board is always square ($n \times n$).
  - The size n can be any positive integer.
  - The algorithm works for any reasonable board size (e.g., 5×5, 8×8, etc.), though performance degrades for very large n.
- Knight's Movement
  - The knight moves in an L-shape (2 squares in one direction and 1 square perpendicular).
  - The knight cannot move outside the board or revisit a square.
- Starting Position
  - The starting position (`startX`, `startY`) must be within the board bounds (0 ≤ `startX` < n, 0 ≤ `startY` < n).
  - If the starting position is invalid, the algorithm terminates early.
- Warnsdorff's Heuristic
  - The algorithm uses Warnsdorff's rule to choose the next move:
    - From the current position, the knight moves to the square with the fewest onward moves.
    - "Fewest onward moves" means picking the next move that leads to the smallest number of possible moves from that future position.
  - This heuristic helps reduce backtracking but does not guarantee a solution in all cases.
- Memory Model
  - The board is stored as a dynamic 2D array (`char**`).
  - Each cell is marked as:
    - `0`: Starting position.
    - `1`: Visited position.
    - `X`: Unvisited position.

2.4) Warnsdorff's Greedy Algorithm

2.4.1) Pseudocode

```text
function isSafe(x, y, board, n):
    return (x and y are within bounds AND board[x][y] == 'X')
```

```
function countAvailableMoves(x, y, board, n):
    count = 0
    for each of the 8 possible knight moves:
        if (newX, newY) is safe:
            count++
    return count

function getNextMove(x, y, board, n):
    minDeg = ∞
    bestMove = -1
    for each possible knight move (newX, newY):
        if (newX, newY) is safe:
            degree = countAvailableMoves(newX, newY, board, n)
            if degree < minDeg:
                minDeg = degree
                bestMove = (newX, newY)
    if bestMove found:
        update x, y
        mark board[x][y] = '1'
        return true
    else:
        return false

function solveKnightTour(startX, startY, n):
    Initialize board[n][n] with 'X'
    board[startX][startY] = '0'
    currentX, currentY = startX, startY
    for moveCount from 1 to n² - 1:
        if no next move (getNextMove fails):
            break
    Print board
    return (moveCount == n² - 1)  // Full tour found?
```

Listing 2.1: The Knight's tour problem - Warnsdorff's Greedy Algorithm Pseudocode

2.4.2) Code

```cpp
    #include <iostream>
#include <climits>
using namespace std;

// Possible knight moves
int dx[] = {2, 1, -1, -2, -2, -1, 1, 2};
int dy[] = {1, 2, 2, 1, -1, -2, -2, -1};

```

```
void tryAllStartPositions(int n)
{
    bool found = false;

    for (int startX = 0; startX < n; startX++) {
        for (int startY = 0; startY < n; startY++) {
            cout << "Trying start position: (" << startX << ", " <<
            startY << ")" << endl;
            if (solveKnightTour(startX, startY, n))
            {
                cout << "Knight's Tour is possible from (" << startX <<
                ", " << startY << ")" << endl;
                found = true;
            }
            else
            {
                cout << "No solution from (" << startX << ", " <<
                startY << ")." << endl;
            }

            cout<<"==========================================================
        }

    }

    if (!found)
    {
        cout << "No Knight's Tour possible from any starting position
        on a " << n << "x" << n << " board." << endl;
    }
}

// Utility function to check if the current position is inside the
board
bool isSafe(int x, int y, char** board, int n)
{
    return (x >= 0 && x < n && y >= 0 && y < n && board[x][y] == 'X');
}

// Function to count the number of available moves from a given
position
```

```
int countAvailableMoves(int x, int y, char** board, int n)
{
    int count = 0;
    for (int i = 0; i < 8; i++)
    {
        int newX = x + dx[i];
        int newY = y + dy[i];
        if (isSafe(newX, newY, board, n))
        {
            count++;
        }
    }
    return count;
}

bool getNextMove(int* x, int* y, char** board, int n)
{
    int minDegIdx = -1;
    int minDeg = INT_MAX;
    int newX, newY;

    // Check all 8 possible moves and apply Warnsdorff's rule
    for (int i = 0; i < 8; i++)
    {
        newX = *x + dx[i];
        newY = *y + dy[i];

        if (isSafe(newX, newY, board, n))
        {
            int degree = countAvailableMoves(newX, newY, board, n);
            if (degree < minDeg)
            {
                minDegIdx = i;
                minDeg = degree;
            }
        }
    }

    // If no valid move is found, return false
    if (minDegIdx == -1) return false;

    // Make the best move (the one with the least onward moves)
    *x = *x + dx[minDegIdx];
    *y = *y + dy[minDegIdx];
```

```
    board[*x][*y] = '1';  // Mark the square as visited
    return true;
}

// Function to solve the Knight's Tour problem using Warnsdorff's rule
bool solveKnightTour(int startX, int startY, int n)
{
    // Dynamically allocate memory for the board
    char** board = new char*[n];
    for (int i = 0; i < n; i++) {
        board[i] = new char[n];
        for (int j = 0; j < n; j++) {
            board[i][j] = 'X';  // 'X' marks unvisited cells
        }
    }
    bool flag = false;

    // Starting position of the knight
    board[startX][startY] = '0';  // '0' marks the starting position

    // Try to move the knight in a way that covers all squares
    int x = startX;
    int y = startY;
    for (int moveCount = 1; moveCount < n * n; moveCount++)
    {
        if (!getNextMove(&x, &y, board, n))
        {
            flag = true;
            break;
        }
    }

    // Print the tour if found
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << board[i][j] << "\t";  // Print '0', '1', or 'X'
        }
        cout << endl;
    }

    // Free dynamically allocated memory
    for (int i = 0; i < n; i++) {
        delete[] board[i];
    }
```

```
    delete[] board;

    return !flag;
}

int main()
{
    int n, startX, startY;

    // Get user input for the size of the board
    cout << "Enter the size of the board (n x n): ";
    cin >> n;

    // Get user input for the starting position
    cout << "Enter the starting position (row and column): \n";
    cin >> startX >> startY;



    cout<<"      0: The starting position. \n      1: A visited
    position. \n      X: An unvisited position."<<endl;

    cout<<"═══════════════════════════════════════════════════════════"

    // Check if the starting position is valid
    if (startX < 0 || startX >= n || startY < 0 || startY >= n) {
        cout << "Invalid starting position!" << endl;
        return 0;
    }

    // Start solving the knight's tour
    if (solveKnightTour(startX, startY, n)) {
        cout << "Knight's Tour is possible!" << endl;
    } else {
        cout << "No solution found!" << endl;
    }

    // tryAllStartPositions(n); Try All possible start positions Given
    board size



    cout<<"═══════════════════════════════════════════════════════════"

```

```
170
171     return 0;
172 }
```

Listing 2.2: The Knight's tour problem - Warnsdorff's Greedy Algorithm C++ Code

2.4.3) Test Cases

- Case 1: Board Size = 8x8
  - Input:
    - Board Size: 3 (so the board is 3x3).
    - Missing Square: All possible starting points
  - Output:

```
Output
Trying start position: (0, 0)
0       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
Knight's Tour is possible from (0, 0)
// from (0,1) to (2,2)
Trying start position: (2, 3)
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       0       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
Knight's Tour is possible from (2, 3)
═══════════════════════════════════════════════════════
Trying start position: (2, 4)
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       0       1       1       1
```

```
1       1       1       1       1       1       1       1
X       1       1       1       1       1       1       1
1       1       X       1       1       1       1       1
1       X       1       1       1       1       1       1
1       1       1       X       1       1       1       1
No solution from (2, 4).
═══════════════════════════════════════════════════════════

Trying start position: (2, 5)
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       0       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
Knight's Tour is possible from (2, 5)
═══════════════════════════════════════════════════════════

Trying start position: (2, 6)
//Rest of Output is too long to be shown
///////////////////////////////////////////////////////////
```

Listing 2.3: The Knight's tour problem - Warnsdorff's Greedy Algorithm Test Case Output

Let us look into the failure case of Listing 2.3 at position (2,4): Warnsdorff's rule is a heuristic, meaning it doesn't guarantee a solution. Here's why:

- Dead Ends: As the knight moves, it might reach a position where all its valid next moves lead to squares that have already been visited. In this case, getNextMove will not find any safe moves and return false, causing solveKnightTour to break the loop and report "No solution found!".
- Premature Isolation: Warnsdorff's rule aims to keep the knight away from squares with fewer onward moves, hoping to visit them later when more options are exhausted. However, it's possible that by prioritizing moves to low-degree squares, the knight might isolate a group of unvisited squares that become unreachable later in the tour.

2.4.4) Complexity Analysis

- Time Complexity

- T(n) = O($n^2 \times 8 \times 8$) = O(n^2)

- Space Complexity
  - Board Storage: O(n^2) (for the $n \times n$ grid).
  - Recursion Stack: O(1) (iterative approach).

2.5) Brute-Force Backtracking

```text
function knightTour(n):
    board = n x n array filled with -1
    moves = [(2, 1), (1, 2), (-1, 2), (-2, 1),
             (-2, -1), (-1, -2), (1, -2), (2, -1)]
    startX, startY = 0, 0
    board[startX][startY] = 0
    if solve(board, startX, startY, 1, moves, n, startX, startY):
        return board
    return "No solution"
function isValid(x, y, board, n):
    return x ≥ 0 and y ≥ 0 and x < n and y < n and board[x][y] == -1
function solve(board, x, y, moveCount, moves, n, startX, startY):
    if moveCount == n * n:
        for (dx, dy) in moves:
            nx = x + dx
            ny = y + dy
            if nx == startX and ny == startY:
                return true
        return false
    for (dx, dy) in moves:
        nx = x + dx
        ny = y + dy
        if isValid(nx, ny, board, n):
            board[nx][ny] = moveCount
            if solve(board, nx, ny, moveCount + 1, moves, n, startX,
            startY):
                return true
            board[nx][ny] = -1
    return false
```

Listing 2.4: The Knight's tour problem - Brute-Force Backtracking Algorithm Pseudocode

2.6) Comparison Between Warnsdorff's Greedy Algorithm and Brute-Force Backtracking

Aspect	Warnsdorff's Greedy Algorithm	Brute-Force Backtracking
Time Complexity	$O(n^2)$ (heuristic-based, not guaranteed)	$O\left(8^{\{n^2\}}\right)$ (exponential)
Space Complexity	$O(n^2)$ (board storage)	$O(n^2)$ (recursion stack + board)
Guarantees Solution?	No (heuristic may fail)	Yes (exhaustive search)

Aspect	Warnsdorff's Greedy Algorithm	Brute-Force Backtracking
Best For	Medium-sized boards (n ≤ 8)	Small boards (n ≤ 5)
Advantages	Fast for solvable cases	Always finds a solution if one exists
Disadvantages	May fail for some starting positions	Impractical for large n

Table 2.1: The Knight's tour problem - Algorithm Comparison

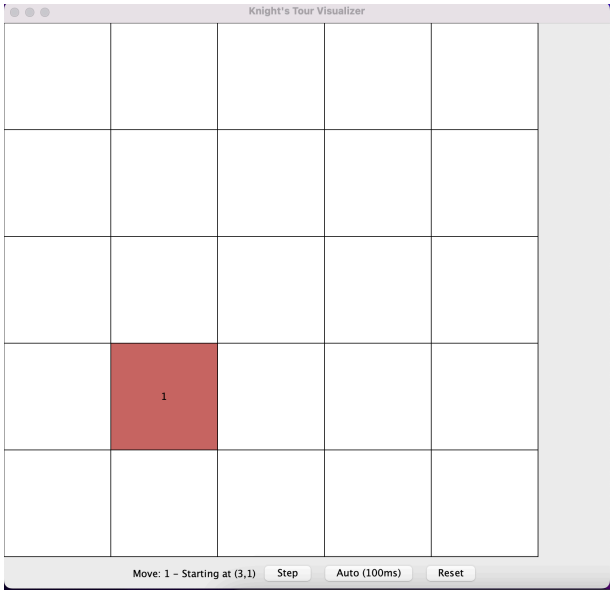
2.7) Conclusion

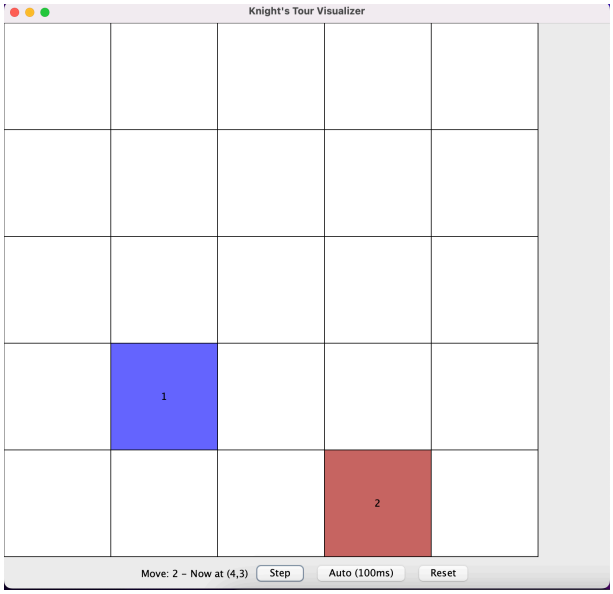
The Knight's Tour solution using Warnsdorff's Rule is a greedy, heuristic-based algorithm that attempts to visit every square on an n x n chessboard exactly once, starting from a given position.

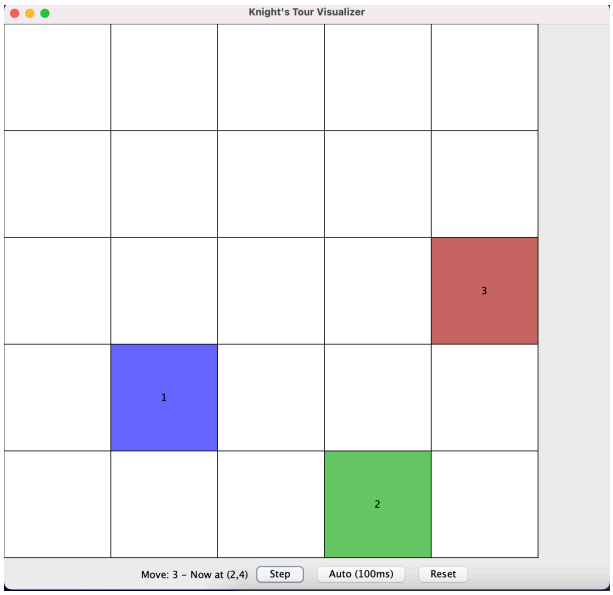
- Key points:
  - Heuristic Strategy:
  - Instead of trying all paths, it always chooses the next move with the fewest onward moves. This reduces the risk of getting stuck early.
- Speed:
  - Runs in $O(n^2)$ time in practice, since it visits each cell once and chooses the next best move from a fixed set of 8 options.
- Space Usage:
  - Uses $O(n^2)$ space to store the board and track visited positions.
- Simplicity:
  - The algorithm avoids recursion or backtracking. This makes the implementation light-weight and easy to follow.
- Limitations:
  - The solution is not guaranteed to succeed on every board size or starting position. It may fail if the heuristic leads into a dead-end late in the path.
- Effectiveness:
  - Works well for most n ≥ 5. Fails occasionally for specific start positions on n = 6, n = 8, etc., due to poor move selection in early steps.
- Determinism:

- Always produces the same path for the same starting point. No randomness means repeatable results, but also no escape from bad decisions.

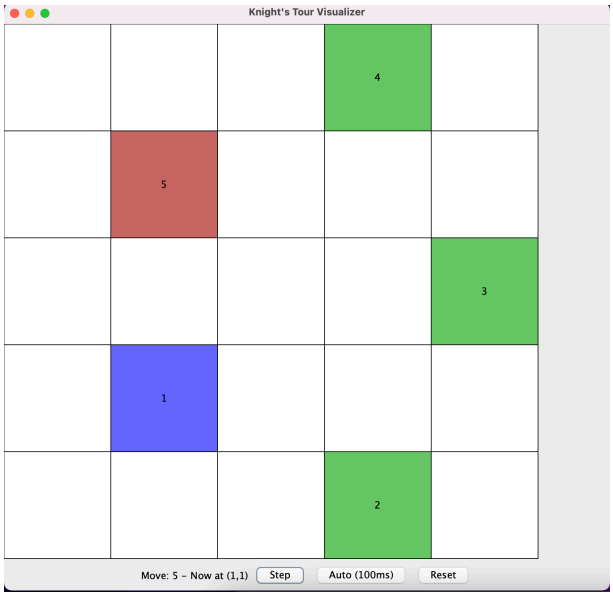
2.8) GUI Screenshots

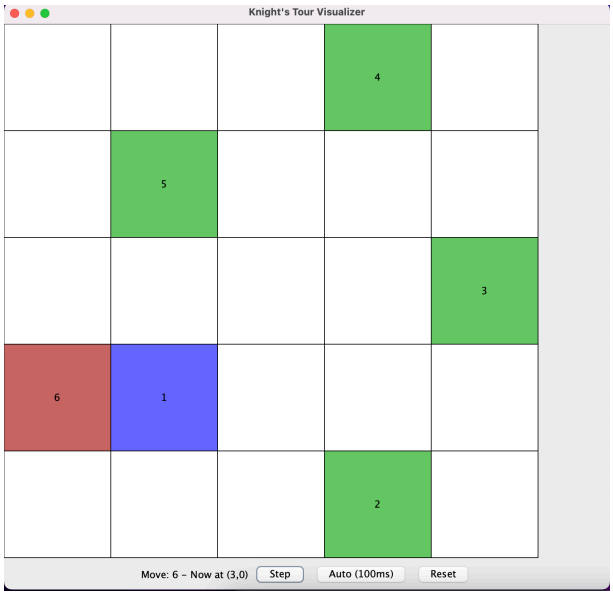


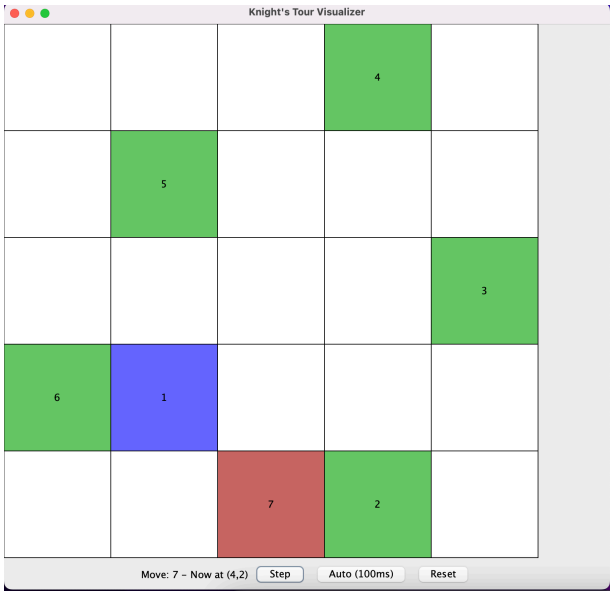


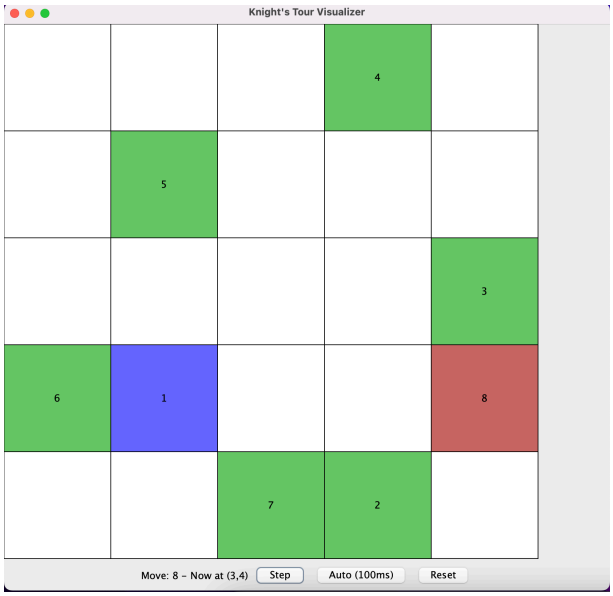

Knight's Tour Visualizer
3
1
2
Move: 3 – Now at (2,4)
Step
Auto (100ms)
Reset

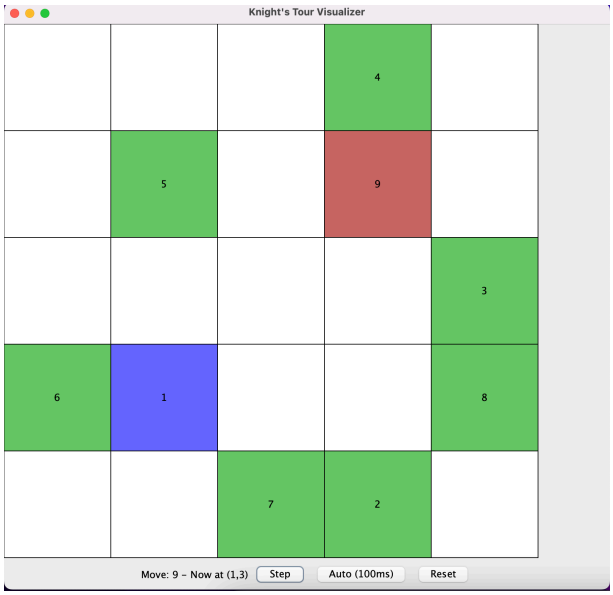

Knight's Tour Visualizer
4
3
1
2
Move: 4 – Now at (0,3)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
4
5
3
1
2
Move: 5 – Now at (1,1)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
4
5
3
6
1
2
Move: 6 – Now at (3,0)
Step
Auto (100ms)
Reset

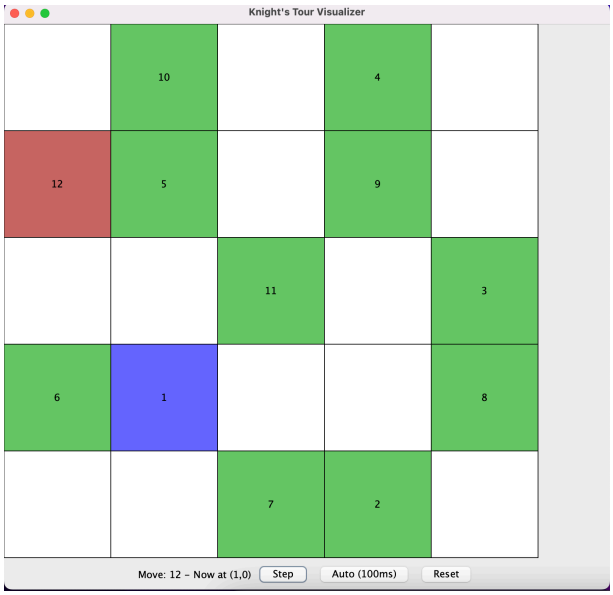

Knight's Tour Visualizer
4
5
3
6
1
7
2
Move: 7 – Now at (4,2)
Step
Auto (100ms)
Reset

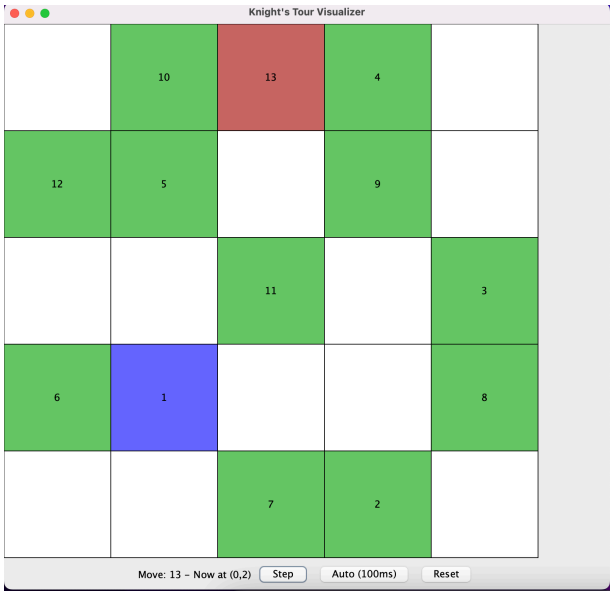

Knight's Tour Visualizer
4
5
3
6
1
8
7
2
Move: 8 – Now at (3,4)
Step
Auto (100ms)
Reset

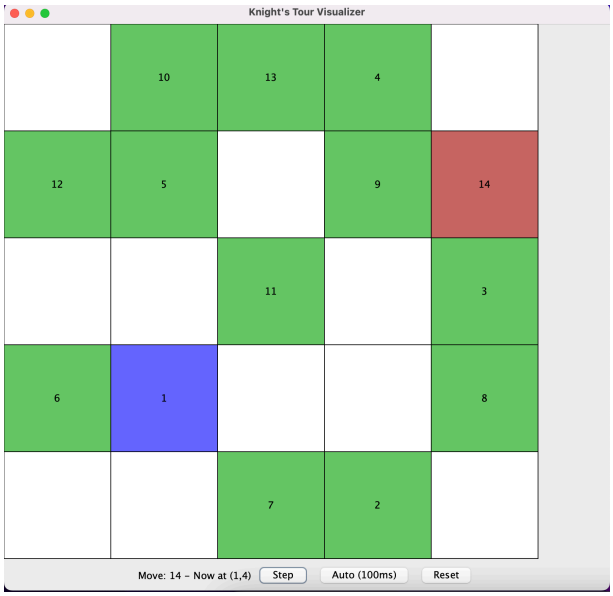

Knight's Tour Visualizer
4
5
9
3
6
1
8
7
2
Move: 9 – Now at (1,3)
Step
Auto (100ms)
Reset

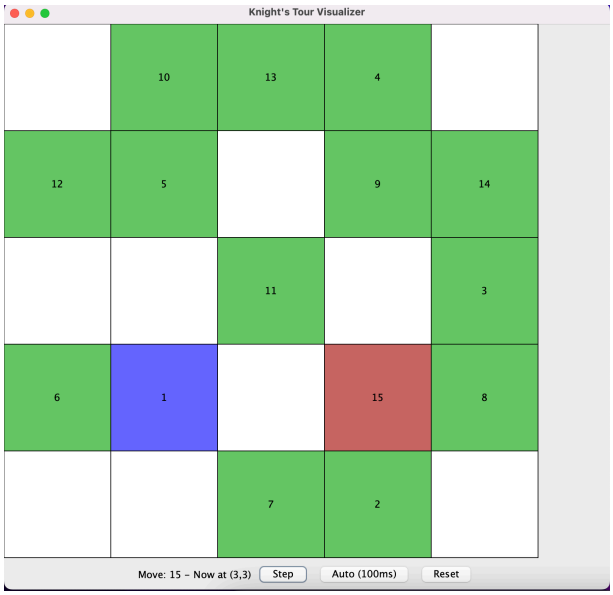

Knight's Tour Visualizer
10
4
5
9
3
6
1
8
7
2
Move: 10 – Now at (0,1)
Step
Auto (100ms)
Reset

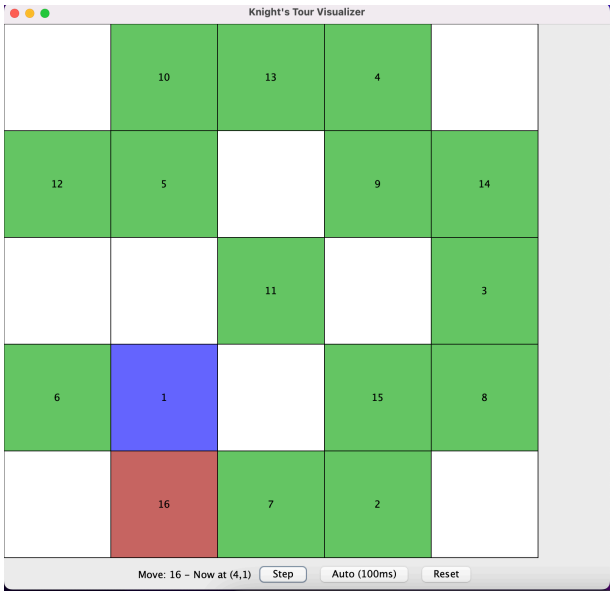

Knight's Tour Visualizer
10
4
5
9
11
3
6
1
8
7
2
Move: 11 – Now at (2,2)
Step
Auto (100ms)
Reset

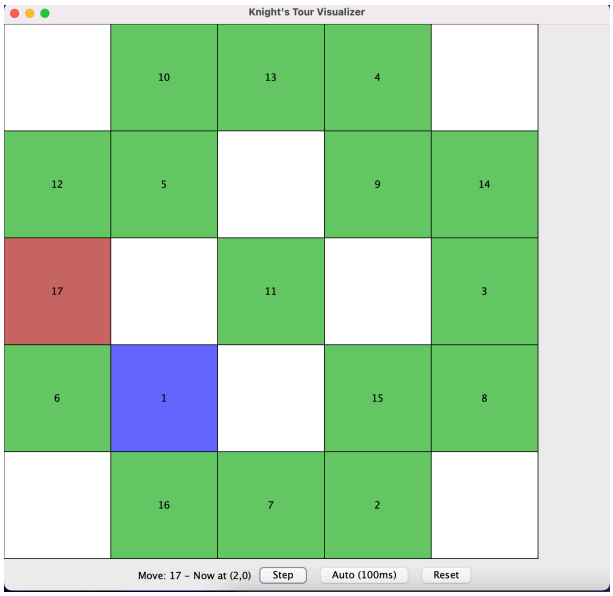

Knight's Tour Visualizer
10
4
12
5
9
11
3
6
1
8
7
2
Move: 12 – Now at (1,0)
Step
Auto (100ms)
Reset

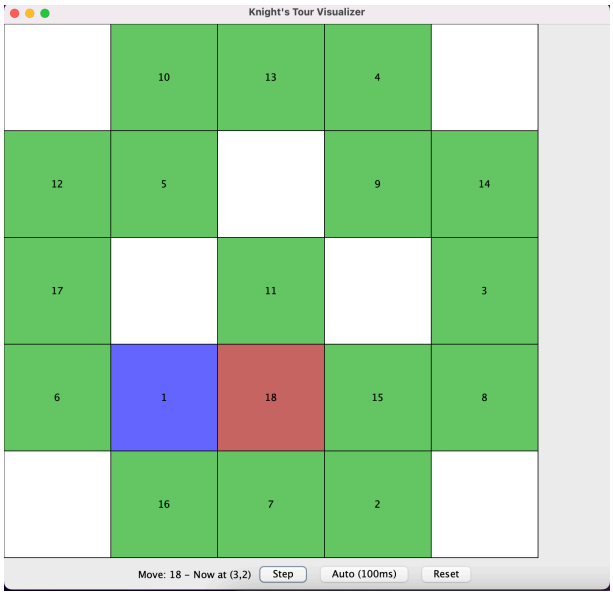

Knight's Tour Visualizer
10
13
4
12
5
9
11
3
6
1
8
7
2
Move: 13 – Now at (0,2)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
10
13
4
12
5
9
14
11
3
6
1
8
7
2
Move: 14 – Now at (1,4)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
10
13
4
12
5
9
14
11
3
6
1
15
8
7
2
Move: 15 – Now at (3,3)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
10
13
4
12
5
9
14
11
3
6
1
15
8
16
7
2
Move: 16 – Now at (4,1)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
10
13
4
12
5
9
14
17
11
3
6
1
15
8
16
7
2
Move: 17 – Now at (2,0)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
10 13 4
12 5 9 14
17 11 3
6 1 18 15 8
16 7 2
Move: 18 – Now at (3,2)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
10 13 4
12 5 9 14
17 11 3
6 1 18 15 8
16 7 2 19
Move: 19 – Now at (4,4)
Step
Auto (100ms)
Reset

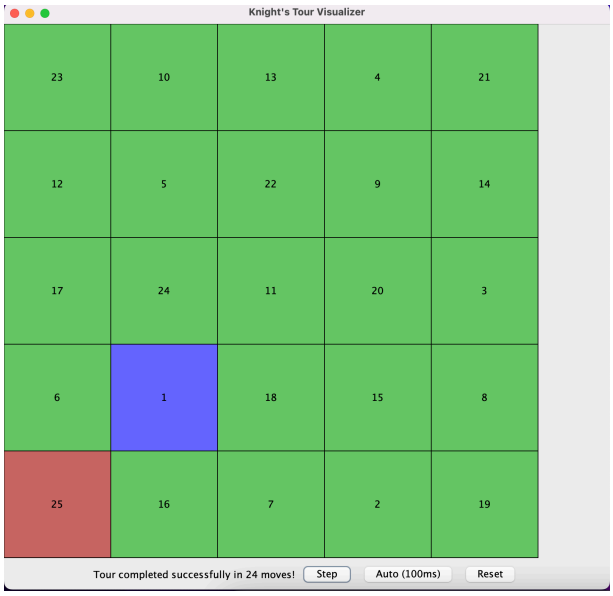

Knight's Tour Visualizer
10 13 4
12 5 9 14
17 11 20 3
6 1 18 15 8
16 7 2 19
Move: 20 – Now at (2,3)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
10 13 4 21
12 5 9 14
17 11 20 3
6 1 18 15 8
16 7 2 19
Move: 21 – Now at (0,4)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
10 13 4 21
12 5 22 9 14
17 11 20 3
6 1 18 15 8
16 7 2 19
Move: 22 – Now at (1,2)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
23 10 13 4 21
12 5 22 9 14
17 11 20 3
6 1 18 15 8
16 7 2 19
Move: 23 – Now at (0,0)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
23 10 13 4 21
12 5 22 9 14
17 24 11 20 3
6 1 18 15 8
16 7 2 19
Move: 24 – Now at (2,1)
Step
Auto (100ms)
Reset


Knight's Tour Visualizer
23 10 13 4 21
12 5 22 9 14
17 24 11 20 3
6 1 18 15 8
25 16 7 2 19
Tour completed successfully in 24 moves!
Step
Auto (100ms)
Reset

3) Task (3)

3.1) Problem Description

Objective

- Move all disks from Peg 1 to Peg 4 in the minimum number of moves, following the rules:
  - Only one disk can be moved at a time.
  - A larger disk cannot be placed on a smaller one.
- Input
  - n: Number of disks (e.g., 8 in the given code).
- Output
  - Step-by-step disk movements.
  - Visual representation of pegs after each move.
  - Total move count.
- Constraints
  - The algorithm works for any n ≥ 1.
  - For n > 0, the solution is optimal (minimum moves).

3.2) Detailed assumptions

- Peg Configuration
  - The puzzle consists of 4 pegs (Peg 1, Peg 2, Peg 3, Peg 4).
  - Peg 1 is initialized with all disks in descending order (largest at bottom).
  - The other pegs start empty.
- Disk Movement Rules
  - Only one disk can be moved at a time.
  - A disk can only be placed on top of a larger disk or an empty peg.
  - Disks cannot be placed outside the pegs.
- Algorithm Choice
  - Uses the Frame-Stewart algorithm, an optimal solution for 4-peg Tower of Hanoi.
  - Recursively divides the problem into subproblems.
- Memory Model

- Pegs are stored as vectors of integers (disk sizes).
- The largest disk has the highest number (e.g., 8 for an 8-disk problem).

3.3) Frame-Stewart Algorithm

3.3.1) Pseudocode

```text
function printPegs(pegs, n):
    for each peg in pegs:
        print disk sizes or "empty"

function moveDisks(k, src, dest, aux1, aux2, pegs, moveCount):
    if k == 0: return

    # Move top k-1 disks from src to aux2 using aux1 and dest
    moveDisks(k - 1, src, aux2, aux1, dest, pegs, moveCount)

    # Move the k-th disk from src to dest
    print "Move disk k from Peg src to Peg dest"
    pegs[dest].push(pegs[src].pop())
    moveCount++
    printPegs(pegs, k)

    # Move k-1 disks from aux2 to dest using src and aux1
    moveDisks(k - 1, aux2, dest, src, aux1, pegs, moveCount)

function solveHanoi(n, pegs):
    Initialize Peg 1 with disks [n, n-1, ... , 1]
    moveCount = 0
    moveDisks(n, 0, 3, 1, 2, pegs, moveCount)
    print "Total moves: moveCount"

main():
    int n = 8;
    vector< vector<int> > pegs(4);
    solveHanoi(n, pegs);
    return 0;
```

Listing 3.1: Tower of Hanoi - Frame-Stewart Algorithm Pseudocode

3.3.2) Code

```C++
#include <iostream>
#include <vector>
using namespace std;

```

```cpp
//Tower of Hanoi: Frame-Stewart algorithm


int moveCount = 0;
void printPegs(const vector< vector<int> >& pegs, int n)
{
    for (int i = 0; i < 4; i++) {
        cout << "Peg " << i + 1 << ": ";
        if (pegs[i].empty()) {
            cout << "empty";
        } else {
            for (int disk : pegs[i]) {
                cout << disk << " ";
            }
        }
        cout << endl;
    }
    cout << "————————————————————————————" << endl;
}

// Function to move `k` disks from `src` to `dest` using `aux1` and `aux2` pegs
void moveDisks(int k, int src, int dest, int aux1, int aux2, vector< vector<int> > & pegs)
{
    if (k == 0) return;
    // Move top k-1 disks from src to aux2 using aux1 and dest as auxiliary
    moveDisks(k - 1, src, aux2, aux1, dest, pegs);
    cout << "Move disk " << k << " from Peg " << src + 1 << " to Peg " << dest + 1 << endl;
    pegs[dest].push_back(pegs[src].back());
    pegs[src].pop_back();
    moveCount++;
    printPegs(pegs, k);
    // Move the k-1 disks from aux2 to dest using src and aux1 as auxiliary
    moveDisks(k - 1, aux2, dest, src, aux1, pegs);
}
void solveHanoi(int n, vector< vector<int> > & pegs)
{
    // Initialize the first peg with all disks
```

```
    for (int i = n; i > 0; --i)
    {
        pegs[0].push_back(i);
    }
    printPegs(pegs, n);
    // Move top k disks from peg 0 to peg 3, using pegs 1 and 2 as auxiliary pegs
    moveDisks(n, 0, 3, 1, 2, pegs);

    cout << "Total number of moves: " << moveCount << endl;
}
int main()
{
    int n = 8;
    vector< vector<int> > pegs(4);
    solveHanoi(n, pegs);
    return 0;
}
```

Listing 3.2: Tower of Hanoi - Frame-Stewart Algorithm C++ Code

3.3.3) Complexity Analysis

- Time Complexity
  - T(n) = 2T(n - 1) + 1
  - T(n) = 2^n - 1
- Space Complexity
  - pegs vector: The pegs vector is a vector of vectors of integers. In the initial state, it stores n disks. Throughout the algorithm, the total number of disks stored across all 4 pegs remains n. Thus, the space used to store the disks is O(n).
  - Recursion Stack: The moveDisks function is recursive. The depth of the recursion depends on the value of k chosen at each step. In the worst case, the depth could be proportional to n. Each recursive call adds a frame to the call stack, storing local variables. Therefore, the space used by the recursion stack can be O(n).
  - the overall space complexity of the Frame-Stewart algorithm as implemented is O(n).

3.3.4) Test Cases

Input: n = 8 disks

Output

```
Peg 1: 8 7 6 5 4 3 2 1

Peg 2: empty

Peg 3: empty

Peg 4: empty

_____________________________

Move disk 1 from Peg 1 to Peg 2

Peg 1: 8 7 6 5 4 3 2

Peg 2: 1

Peg 3: empty

Peg 4: empty

_____________________________

Move disk 2 from Peg 1 to Peg 3

Peg 1: 8 7 6 5 4 3

Peg 2: 1

Peg 3: 2

Peg 4: empty

_____________________________

|

|

|

|

_____________________________

Move disk 1 from Peg 1 to Peg 3

Peg 1: 2

Peg 2: empty

Peg 3: 1

Peg 4: 8 7 6 5 4 3

_____________________________

Move disk 2 from Peg 1 to Peg 4

Peg 1: empty

Peg 2: empty

Peg 3: 1

Peg 4: 8 7 6 5 4 3 2

_____________________________

Move disk 1 from Peg 3 to Peg 4

Peg 1: empty

Peg 2: empty

Peg 3: empty
```

```
Peg 4: 8 7 6 5 4 3 2 1
_____________________
```

Listing 3.3: Tower of Hanoi - Frame-Stewart Algorithm Test Case Output

3.4) Greedy Algorithm

```text
function moveDisksGreedy(n, source, target, auxiliaries):
    stack = [(n, source, target, auxiliaries)]
    while stack not empty:
        (k, src, tgt, auxs) = stack.pop()
        if k == 1:
            print("Move disk from", src, "to", tgt)
        else:
            aux1 = auxs[0]
            remaining_aux = auxs[1:] + [src]
            stack.append((k - 1, aux1, tgt, [src] + remaining_aux[:-1]))
            stack.append((1, src, tgt, []))
            stack.append((k - 1, src, aux1, [tgt] + remaining_aux[:-1]))
```

Listing 3.4: Tower of Hanoi - Greedy Algorithm Pseudocode

3.5) Dynamic Programing Task

Solves the Tower of Hanoi puzzle with 4 pegs using breadth-first search (BFS). It aims to find the shortest sequence of moves that transfers all disks from the first peg to another peg.

- Key Components:
  - TreeNode Struct: Represents a configuration of each disc in pegs.
  - createIdentifier: Generates a unique string identifier for each state to track visited configurations.
  - isVisited: Checks if a state has been previously explored.
  - generateChildren: Generates valid child nodes by moving disks between pegs.
  - bfs: Performs BFS to find the shortest path from the initial state to the goal state, returning the number of moves.
- How It Works:
  - Initializes the initial state and goal state.
  - Uses BFS to explore all possible configurations level by level.
  - Tracks visited states to avoid revisiting.
  - Prints the solution path when the goal state is reached.

3.5.1) Pseudocode

```text
Function BFS(initial, goal, numPegs):
    Initialize frontier as an empty queue
    Initialize visited as an empty list

    Enqueue initial state to frontier
    Add the identifier of the initial state to visited

    Set depth = 0

    While frontier is not empty:
        Set levelSize = size of frontier

        For each state in the frontier (level by level):
            Dequeue the first state from frontier
            If state is the goal state:
                Print the solution and return depth

            Generate all valid child states using valid moves (hint: use
            generateChildren)

            For each child state:
                If the child state has not been visited:
                    Add the identifier of child state to visited
                    Enqueue the child state to frontier

        Increment depth

    Return -1 (goal not reachable)
```

Listing 3.5: Tower of Hanoi - Dynamic Programing Pseudocode

3.5.2) Code

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

struct TreeNode
{
    vector< vector<int> > pegs;
    TreeNode* parent;
};
```

```
12
13
14
15  void printPegs(const vector< vector<int> >& pegs) {
16      for (int i = 0; i < pegs.size(); ++i) {
17          cout << "Peg " << i + 1 << ": ";
18          for (int disk : pegs[i]) cout << disk << " ";
19          cout << endl;
20      }
21  }
22
23  void printSolution(TreeNode* node) {
24      if (!node) return;
25      if (!node->parent) {
26          cout << "Initial state:" << endl;
27          printPegs(node->pegs);
28          cout << "--------" << endl;
29      } else {
30          printSolution(node->parent);
31          int from = -1, to = -1, moved = -1;
32          for (int i = 0; i < node->pegs.size(); ++i) {
                int diff = node->pegs[i].size() - node->parent-
33
                >pegs[i].size();
34              if (diff == 1) {
35                  to = i;
36                  moved = node->pegs[i].back();
37              }
38              if (diff == -1) {
39                  from = i;
40              }
41          }
            cout << "Move disk " << moved << " from peg " << from + 1 << "
42
            to peg " << to + 1 << endl;
43          printPegs(node->pegs);
44          cout << "--------" << endl;
45      }
46  }
47
48
49
50
51  string createIdentifier(TreeNode* node, int numPegs)
52  {
```

```
    string id;
    for (int i = 0; i < numPegs; ++i)
    {
        id += '[';
        for (int disk : node->pegs[i]) id += char('A' + disk); // A, B, C, ...
        id += ']';
    }
    return id;
}




bool isVisited(string id, const vector<string>& visited)
{
    for (string node : visited)
    {
        if (node == id) return true;
    }
    return false;
}

vector<TreeNode*> generateChildren(TreeNode* parent, int numPegs)
{
    vector<TreeNode*> children;
    for (int from = 0; from < numPegs; ++from)
    {
        if (parent->pegs[from].empty()) continue;
        int disk = parent->pegs[from].back();
        for (int to = 0; to < numPegs; ++to)
        {
            if (from == to) continue;
            if (!parent->pegs[to].empty() && parent->pegs[to].back() < disk) continue;

            TreeNode* child = new TreeNode();
            child->pegs = parent->pegs;
            child->pegs[from].pop_back();
            child->pegs[to].push_back(disk);
            child->parent = parent;
            children.push_back(child);
```

```
        }
    }
    return children;
}



int bfs(TreeNode* initial, TreeNode* goal, int numPegs)
{
    queue<TreeNode*> frontier;
    vector<string> visited;

    string goal_id = createIdentifier(goal, numPegs);
    frontier.push(initial);
    visited.push_back(createIdentifier(initial, numPegs));

    int depth = 0;

    while (!frontier.empty())
    {
        int levelSize = frontier.size();
        for (int i = 0; i < levelSize; ++i)
        {
            TreeNode* current = frontier.front();
            frontier.pop();
            string current_id = createIdentifier(current, numPegs);

            if (current_id == goal_id)
            {
                printSolution(current);
                return depth;
            }

            vector<TreeNode*> children = generateChildren(current,
            numPegs);
            for (TreeNode* child : children)
            {
                string id = createIdentifier(child, numPegs);
                if (isVisited(id, visited))
                {
                    delete child;
                    continue;
                }
                visited.push_back(id);
```

```
                frontier.push(child);
            }
        }
        ++depth;
    }
    return -1;
}

int main() {
    int numDisks = 8;
    int numPegs = 4;

    TreeNode* root = new TreeNode();
    root->pegs.resize(numPegs);

    for (int i = numDisks; i >= 1; --i) root->pegs[0].push_back(i);
    root->parent = nullptr;

    TreeNode* goal = new TreeNode();
    goal->pegs.resize(numPegs);

    for (int i = numDisks; i >= 1; --i) goal->pegs[numPegs -
1].push_back(i);

    int result = bfs(root, goal, numPegs);
    if (result != -1) {
        cout << "Shortest path height: " << result << endl;
    } else {
        cout << "Goal not reachable." << endl;
    }

    return 0;
}
```

Listing 3.6: Tower of Hanoi - Dynamic Programming C++ Code

3.5.3) Complexity Analysis

- Time Complexity:
  - The maximum number of possible states in the problem is (numPegs)^numDisks
  - each state may generate numPegs x numDisks childrens
  - total complexity = O((numPegs)^numDisks x numPegs x numDisks) = O(n x 4^n)
- Space Complexity = O((numPegs)^numDisks x numPegs x numDisks) = O(n x 4^n)

3.5.4) Test Cases

- n=8 in 33 moves

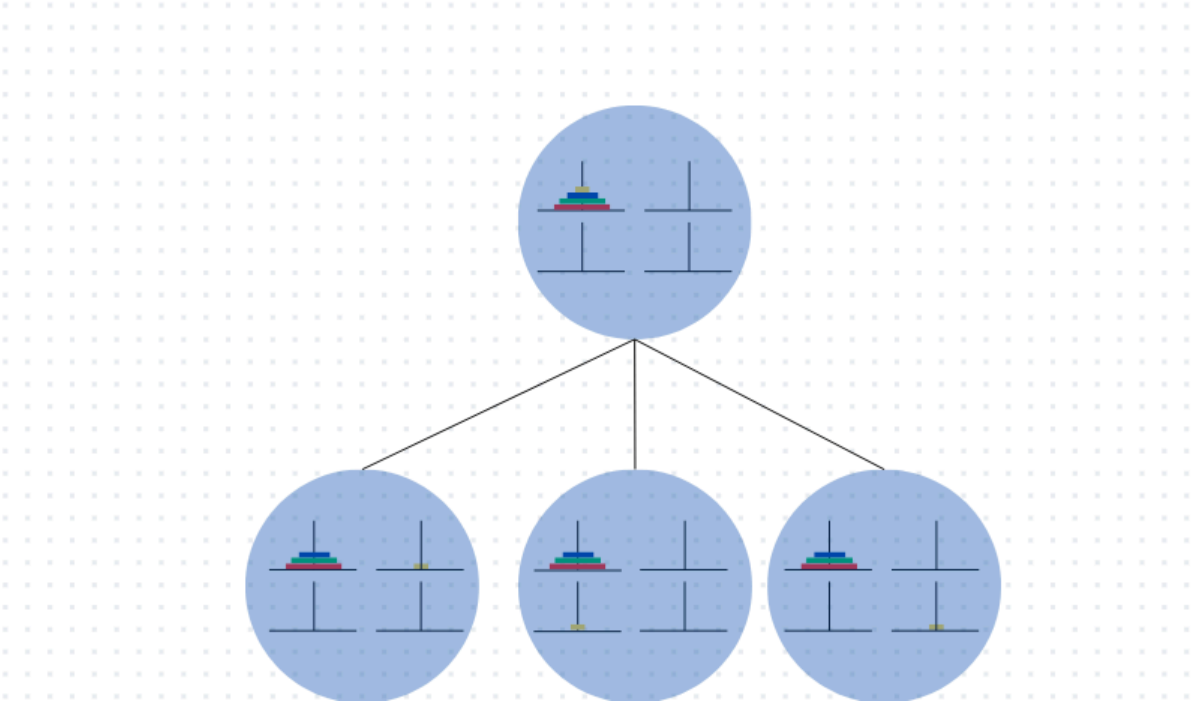
```
Initial state:
Peg 1: 8 7 6 5 4 3 2 1
Peg 2:
Peg 3:
Peg 4:
--------
Move disk 1 from peg 1 to peg 2
Peg 1: 8 7 6 5 4 3 2
Peg 2: 1
Peg 3:
Peg 4:
--------
Move disk 2 from peg 1 to peg 3
Peg 1: 8 7 6 5 4 3
Peg 2: 1
Peg 3: 2
Peg 4:
--------
Move disk 3 from peg 1 to peg 4
Peg 1: 8 7 6 5 4
Peg 2: 1
Peg 3: 2
Peg 4: 3
--------
Move disk 1 from peg 2 to peg 3
Peg 1: 8 7 6 5 4
Peg 2:
Peg 3: 2 1
Peg 4: 3
--------
Move disk 4 from peg 1 to peg 2
Peg 1: 8 7 6 5
Peg 2: 4
Peg 3: 2 1
Peg 4: 3
--------
Move disk 1 from peg 3 to peg 1
Peg 1: 8 7 6 5 1
Peg 2: 4
Peg 3: 2
Peg 4: 3
--------
Move disk 3 from peg 4 to peg 2
Peg 1: 8 7 6 5 1
Peg 2: 4 3
Peg 3: 2
Peg 4:
--------
Move disk 2 from peg 3 to peg 2
Peg 1: 8 7 6 5 1
Peg 2: 4 3 2
Peg 3:
Peg 4:
--------
```

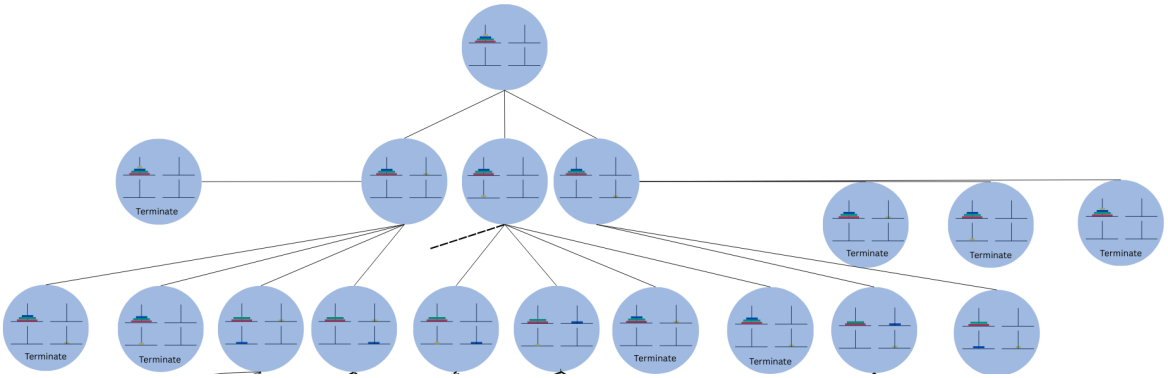
```
Peg 2: 4 3 2
Peg 3:
Peg 4:
--------
Move disk 1 from peg 1 to peg 2
Peg 1: 8 7 6 5
Peg 2: 4 3 2 1
Peg 3:
Peg 4:
--------
Move disk 5 from peg 1 to peg 3
Peg 1: 8 7 6
Peg 2: 4 3 2 1
Peg 3: 5
Peg 4:
--------
Move disk 6 from peg 1 to peg 4
Peg 1: 8 7
Peg 2: 4 3 2 1
Peg 3: 5
Peg 4: 6
--------
Move disk 5 from peg 3 to peg 4
Peg 1: 8 7
Peg 2: 4 3 2 1
Peg 3:
Peg 4: 6 5
--------
Move disk 7 from peg 1 to peg 3
Peg 1: 8
Peg 2: 4 3 2 1
Peg 3: 7
Peg 4: 6 5
--------
Move disk 5 from peg 4 to peg 1
Peg 1: 8 5
Peg 2: 4 3 2 1
Peg 3: 7
Peg 4: 6
--------
Move disk 6 from peg 4 to peg 3
Peg 1: 8 5
Peg 2: 4 3 2 1
Peg 3: 7 6
Peg 4:
--------
Move disk 5 from peg 1 to peg 3
Peg 1: 8
Peg 2: 4 3 2 1
```

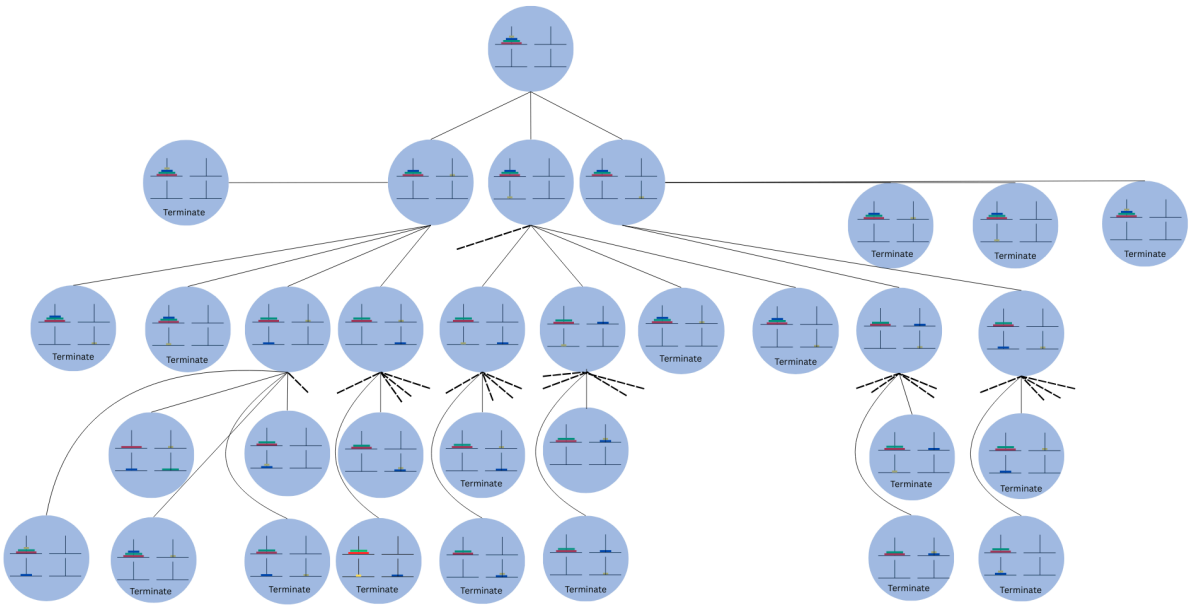
```
--------
Move disk 8 from peg 1 to peg 4
Peg 1:
Peg 2: 4 3 2 1
Peg 3: 7 6 5
Peg 4: 8
--------
Move disk 5 from peg 3 to peg 4
Peg 1:
Peg 2: 4 3 2 1
Peg 3: 7 6
Peg 4: 8 5
--------
Move disk 6 from peg 3 to peg 1
Peg 1: 6
Peg 2: 4 3 2 1
Peg 3: 7
Peg 4: 8 5
--------
Move disk 5 from peg 4 to peg 1
Peg 1: 6 5
Peg 2: 4 3 2 1
Peg 3: 7
Peg 4: 8
--------
Move disk 7 from peg 3 to peg 4
Peg 1: 6 5
Peg 2: 4 3 2 1
Peg 3:
Peg 4: 8 7
--------
Move disk 5 from peg 1 to peg 3
Peg 1: 6
Peg 2: 4 3 2 1
Peg 3: 5
Peg 4: 8 7
--------
Move disk 6 from peg 1 to peg 4
Peg 1:
Peg 2: 4 3 2 1
Peg 3: 5
Peg 4: 8 7 6
--------
```

```
Peg 4: 8 7 6 5
--------
Move disk 2 from peg 2 to peg 3
Peg 1: 1
Peg 2: 4 3
Peg 3: 2
Peg 4: 8 7 6 5
--------
Move disk 1 from peg 1 to peg 3
Peg 1:
Peg 2: 4 3
Peg 3: 2 1
Peg 4: 8 7 6 5
--------
Move disk 3 from peg 2 to peg 1
Peg 1: 3
Peg 2: 4
Peg 3: 2 1
Peg 4: 8 7 6 5
--------
Move disk 4 from peg 2 to peg 4
Peg 1: 3
Peg 2:
Peg 3: 2 1
Peg 4: 8 7 6 5 4
--------
Move disk 3 from peg 1 to peg 4
Peg 1:
Peg 2:
Peg 3: 2 1
Peg 4: 8 7 6 5 4 3
--------
Move disk 1 from peg 3 to peg 1
Peg 1: 1
Peg 2:
Peg 3: 2
Peg 4: 8 7 6 5 4 3
--------
Move disk 2 from peg 3 to peg 4
Peg 1: 1
Peg 2:
Peg 3:
Peg 4: 8 7 6 5 4 3 2
--------
Move disk 1 from peg 1 to peg 4
Peg 1:
```

- The following are GUI Screenshots:







3.6) Comparison Between the Frame-Stewart Algorithm and the Greedy Algorithm Solution

Aspect	Greedy Algorithm	Frame-Stewart (4 Pegs)
Strategy	• Divide into moving smaller stack, largest disk, smaller stack • Always moves the smallest movable disk to the first legal peg • Chooses next move without considering future impact • No recursion, no lookahead • Doesn't guarantee minimum moves	• Recursive, optimal strategy • Splits the problem: • Move k disks to an auxiliary peg using 4 pegs • Move n−k disks to target using 3 pegs • Move k disks from auxiliary to target using 4 pegs • Requires choosing best k (minimizes moves) • Exhaustively checks possibilities
Time Complexity	$O(m)$ where m = number of moves (can be much more than optimal)	$O(2^n)$
Space Complexity	$O(1)$ to $O(n)$ (depending on implementation)	$O(n)$
Best For	Quick approximations, low-resource environments	Optimal solution when minimal move count is required
Advantages	• Simple and fast • Low memory usage • Easy to implement	• Finds minimum number of moves • Exploits extra pegs efficiently
Disadvantages	• Not optimal	• Slow for large n

Aspect	Greedy Algorithm	Frame-Stewart (4 Pegs)
	• Can take many more moves than necessary	• Needs recursion and dynamic decision at each step

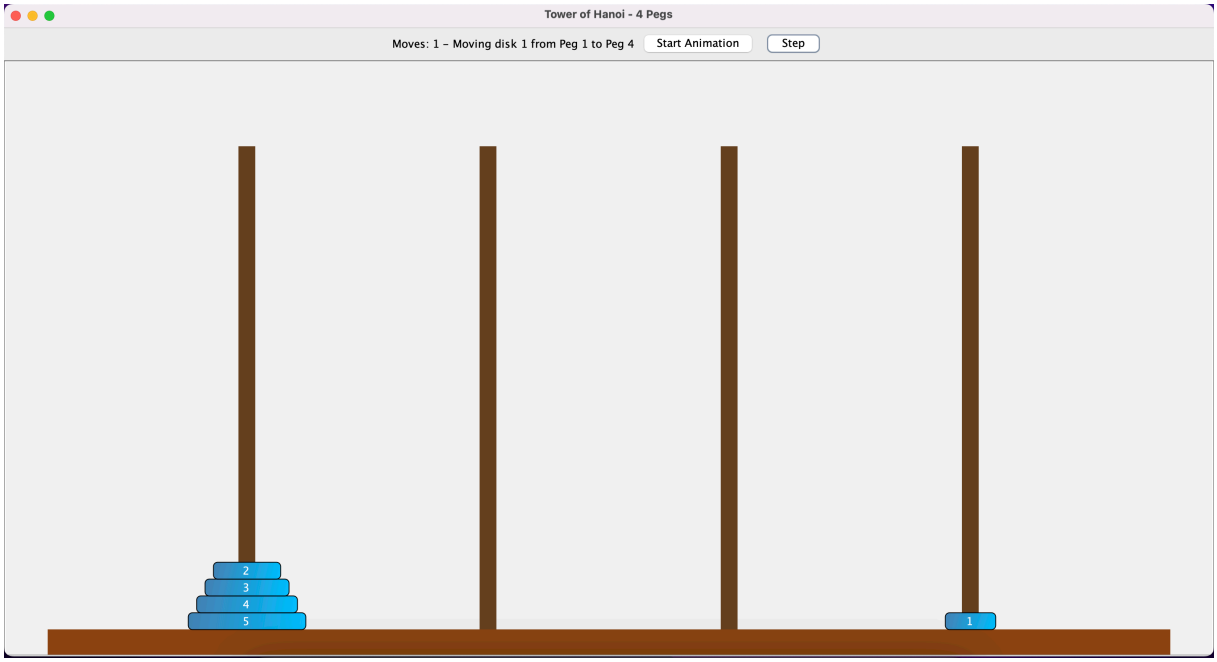
Table 3.1: Tower of Hanoi - Algorithm Comparison

3.7) Conclusion

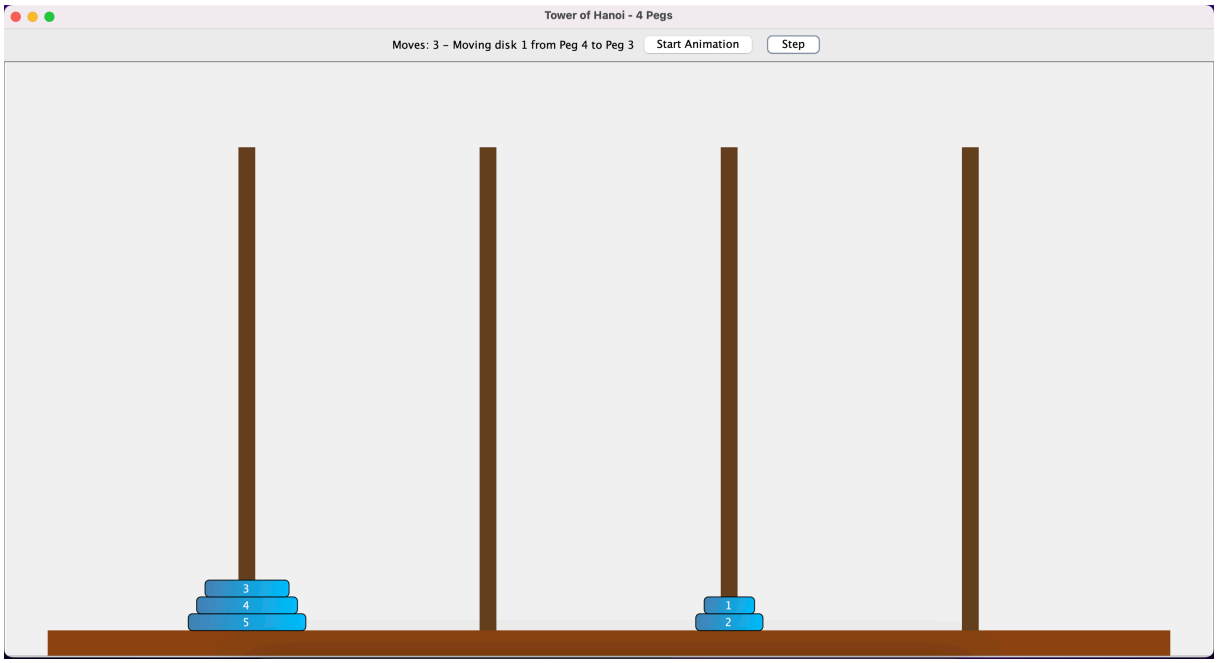
The Frame-Stewart algorithm for the 4-peg Tower of Hanoi problem is a more efficient approach compared to the classic 3-peg recursive solution. It reduces the number of moves required by strategically dividing the problem into smaller subproblems, leveraging an additional peg.

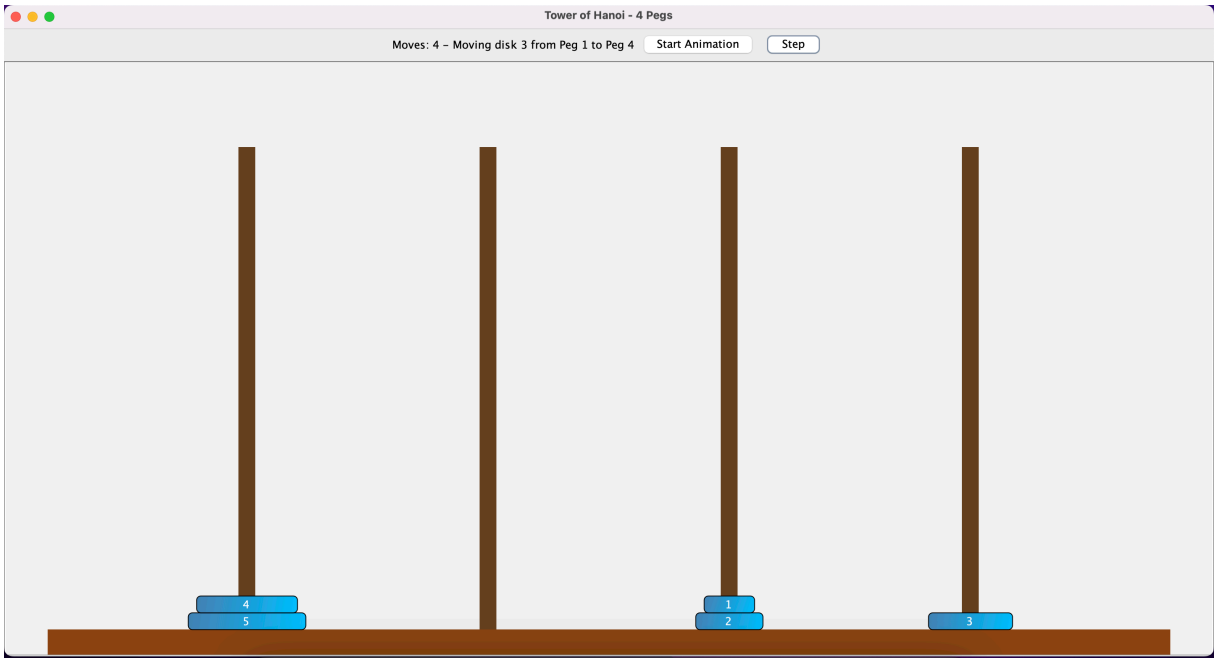
- Key points:
  - Time Efficiency: Unlike the exponential time complexity of the 3-peg solution, the Frame-Stewart algorithm's time complexity grows more slowly, offering significant improvements for larger n.
  - Space Usage: Both algorithms require O(n) space, primarily due to the recursion stack and the storage of pegs.
  - Complexity: The Frame-Stewart algorithm is more complex to implement compared to the simpler 3-peg solution but is more suitable for larger numbers of disks due to its reduced time complexity.
  - Limitations: While it offers a better solution in terms of efficiency, the algorithm still involves an exponential time complexity in the worst case due to the recursive nature and the need to explore different subproblems.
  - Effectiveness: The Frame-Stewart algorithm is highly effective for solving larger Tower of Hanoi problems, especially when more than three pegs are available.
  - Determinism: The algorithm is deterministic, producing the same solution for a given number of disks and pegs.

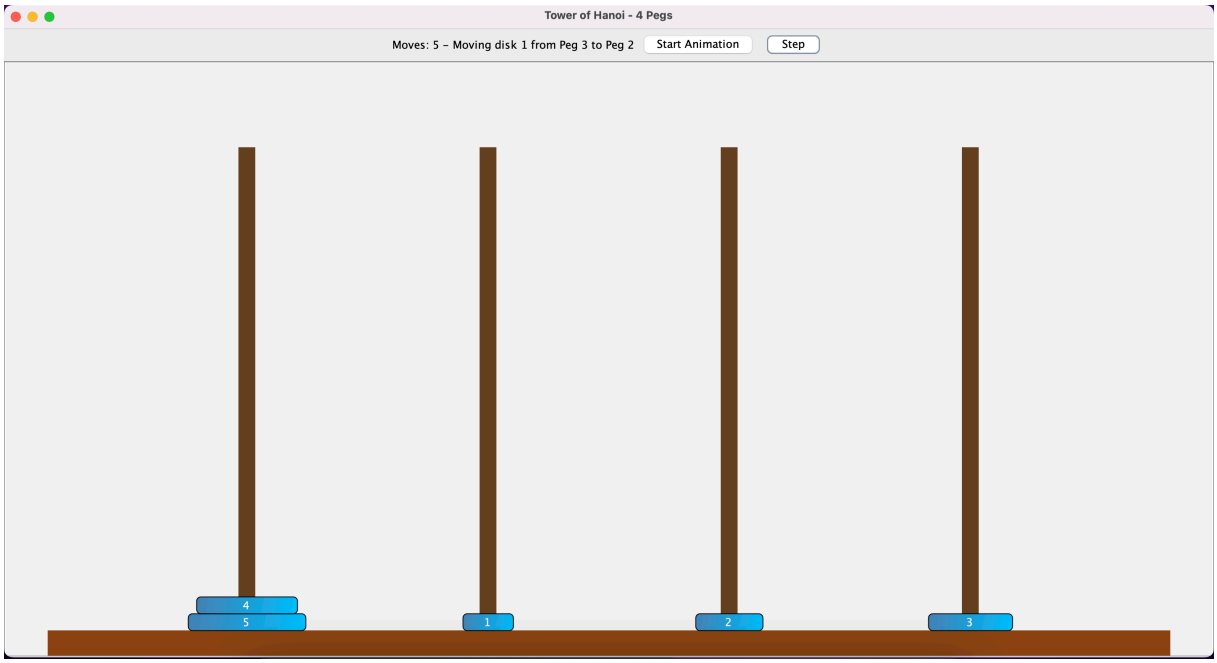
3.8) GUI Screenshots

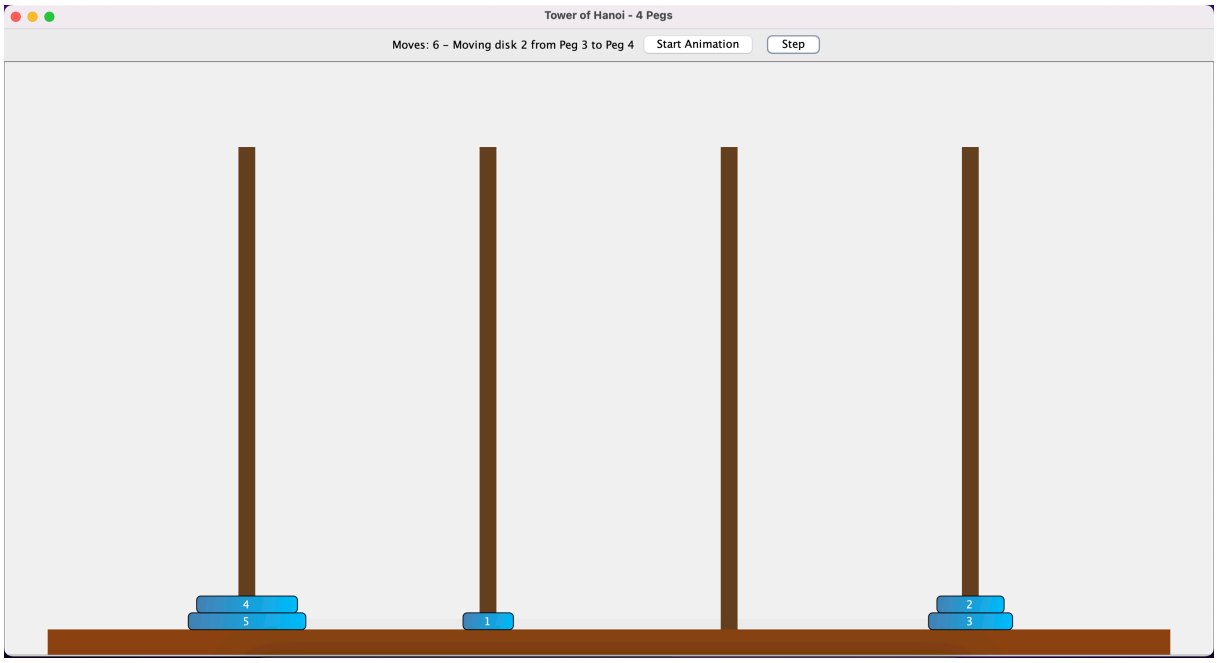


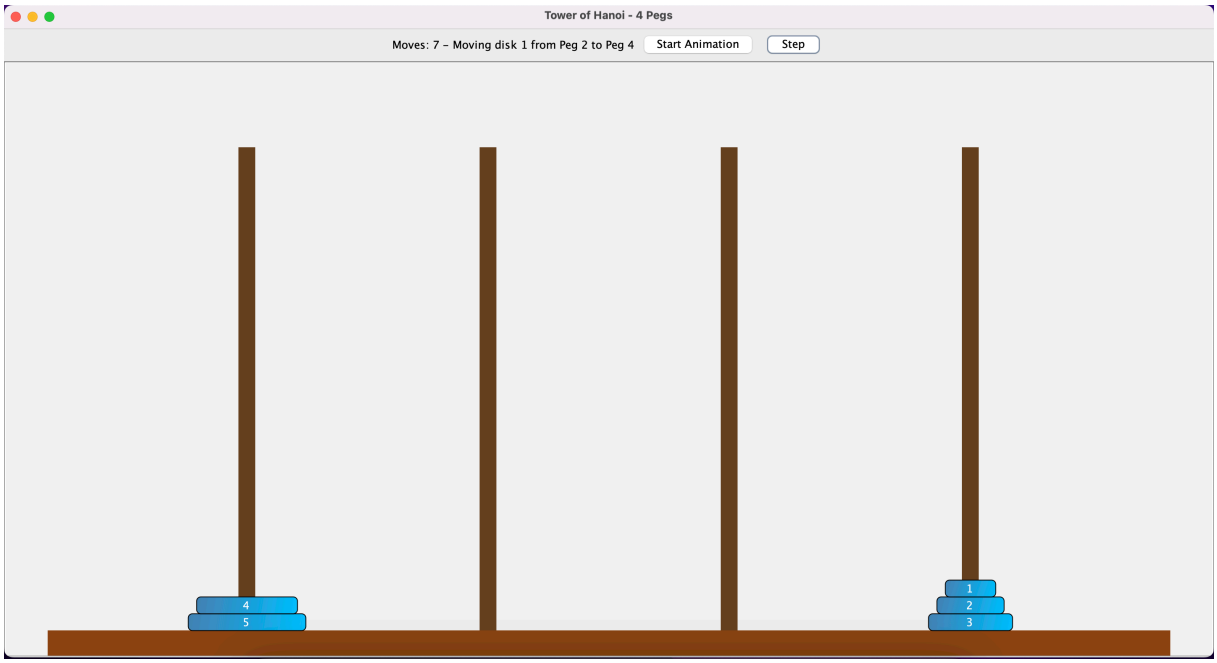

Tower of Hanoi - 4 Pegs
Moves: 2 – Moving disk 2 from Peg 1 to Peg 3
Start Animation
Step
3
4
5
2
1


Tower of Hanoi - 4 Pegs
Moves: 3 – Moving disk 1 from Peg 4 to Peg 3
Start Animation
Step
3
4
5
1
2

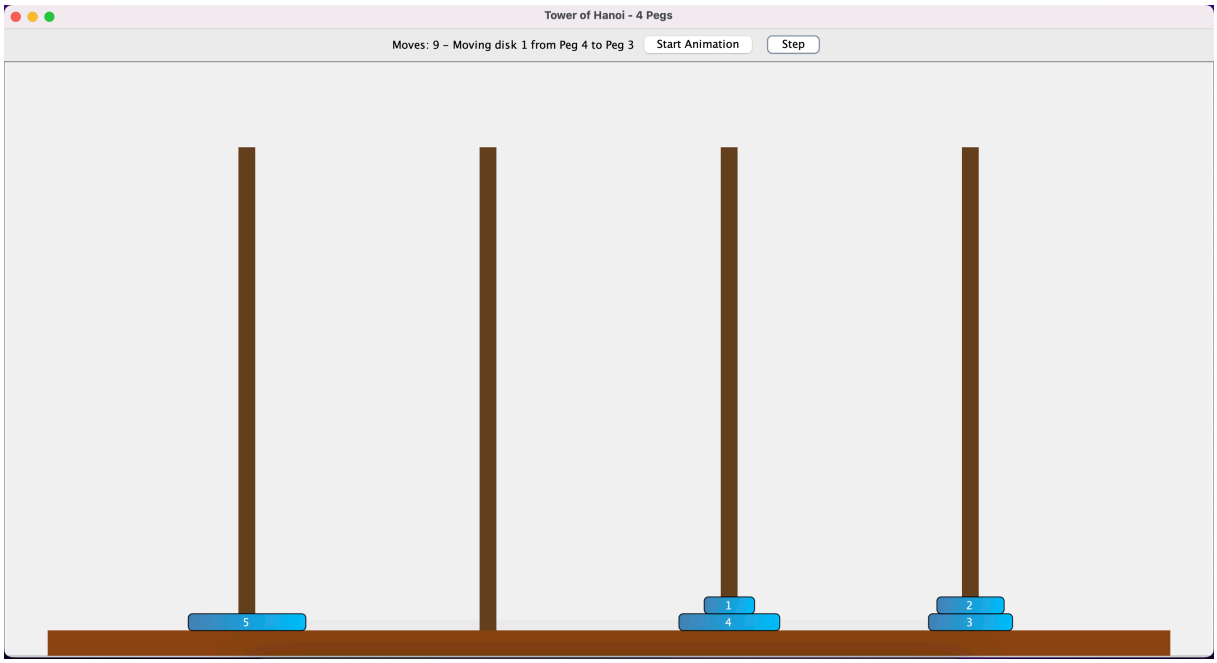

Tower of Hanoi - 4 Pegs
Moves: 4 – Moving disk 3 from Peg 1 to Peg 4
Start Animation
Step
4
5
1
2
3

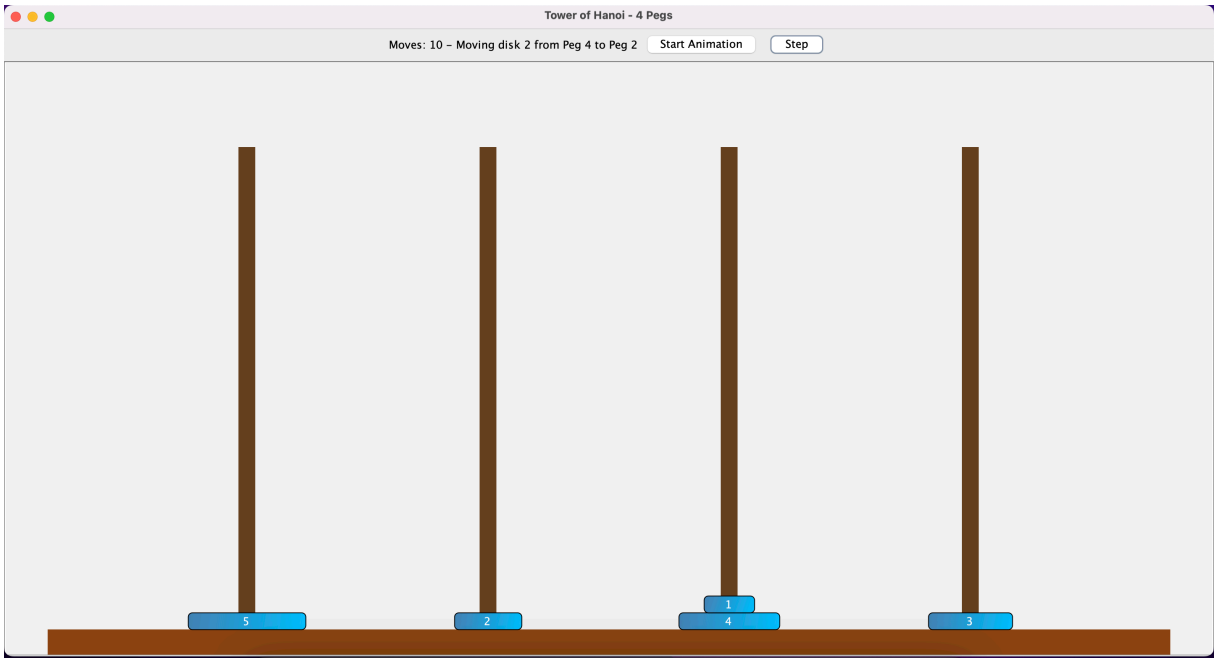

Tower of Hanoi - 4 Pegs
Moves: 5 – Moving disk 1 from Peg 3 to Peg 2
Start Animation
Step
4
5
1
2
3

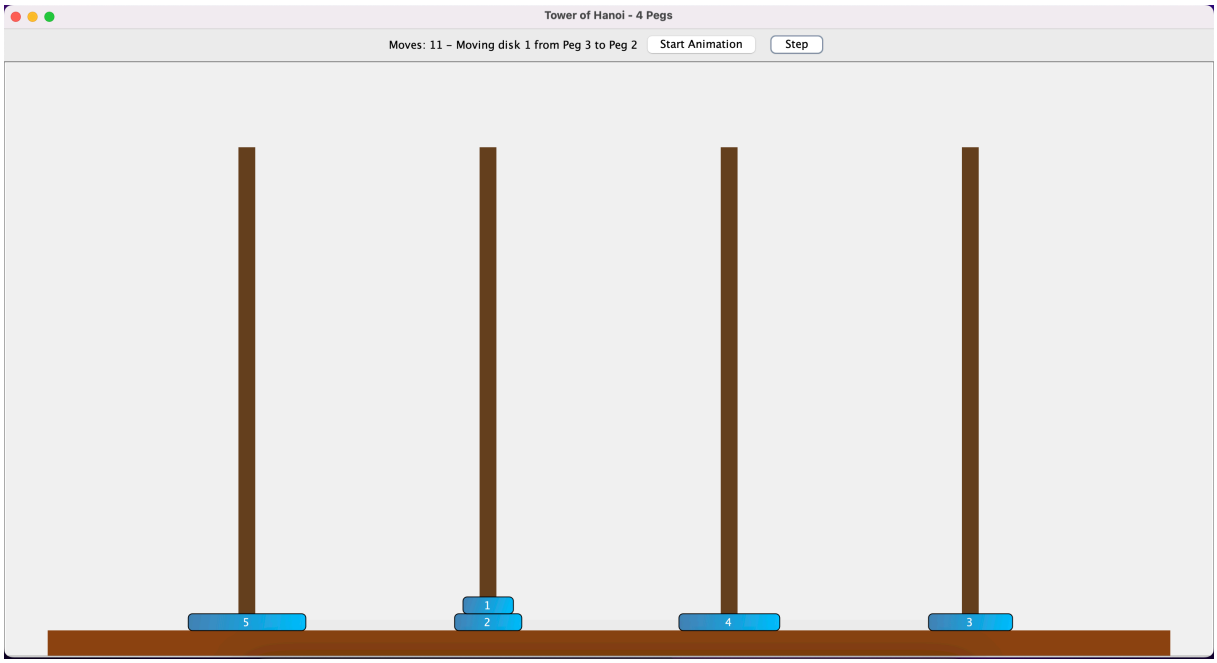

Tower of Hanoi - 4 Pegs
Moves: 6 – Moving disk 2 from Peg 3 to Peg 4
Start Animation
Step
4
5
1
2
3

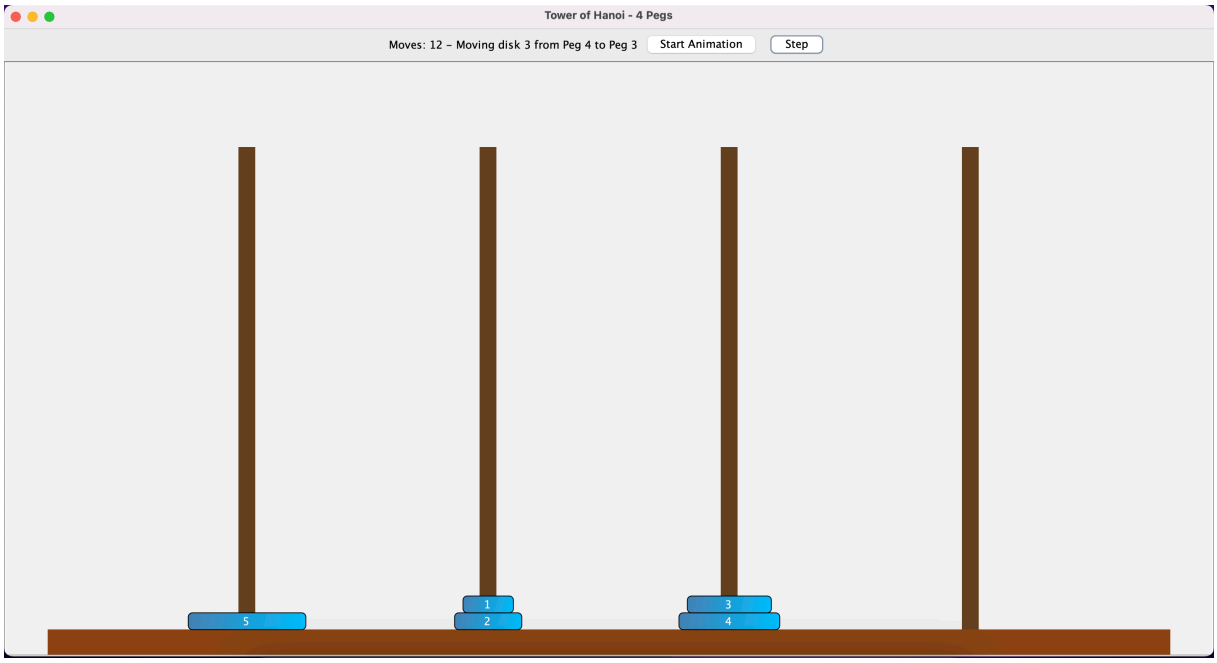

Tower of Hanoi - 4 Pegs
Moves: 7 – Moving disk 1 from Peg 2 to Peg 4
Start Animation
Step
4
5
1
2
3

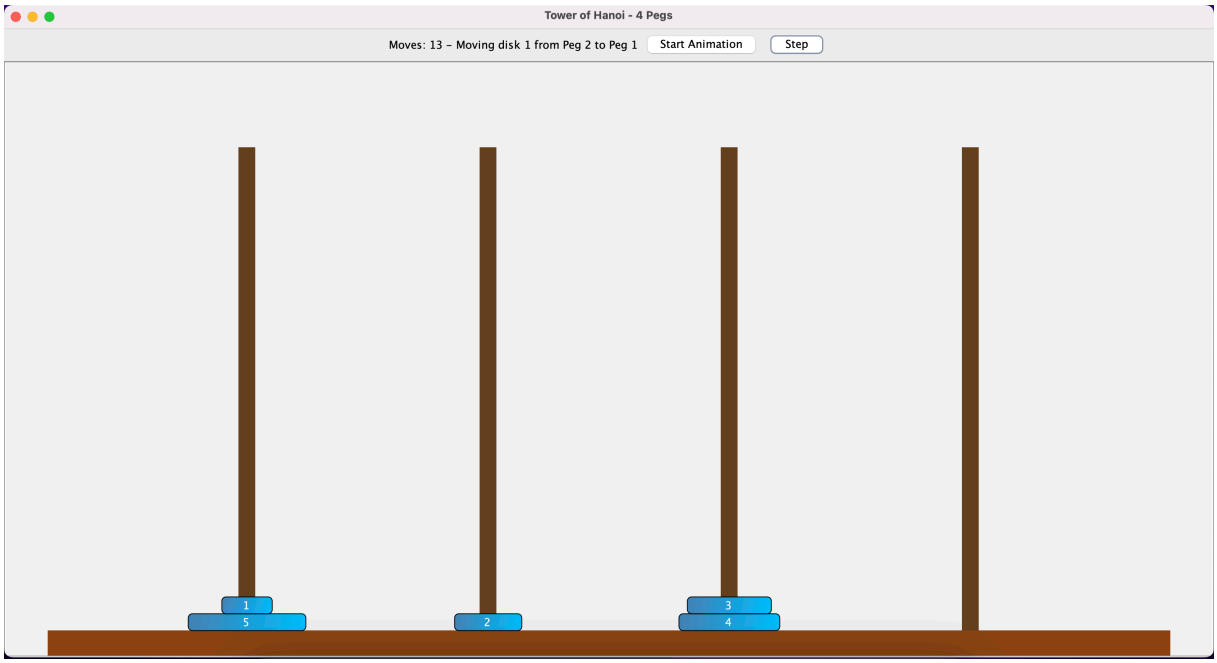

Tower of Hanoi - 4 Pegs
Moves: 8 – Moving disk 4 from Peg 1 to Peg 3
Start Animation
Step
5
4
1
2
3

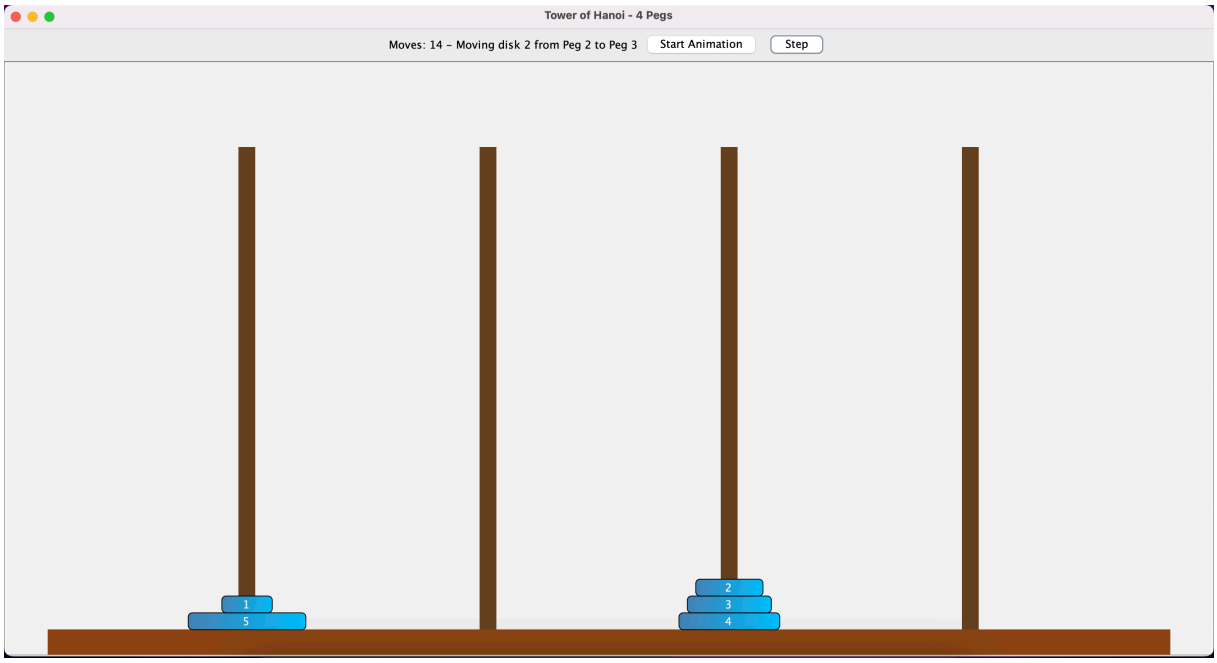

Tower of Hanoi - 4 Pegs
Moves: 9 – Moving disk 1 from Peg 4 to Peg 3
Start Animation
Step
5
1
4
2
3

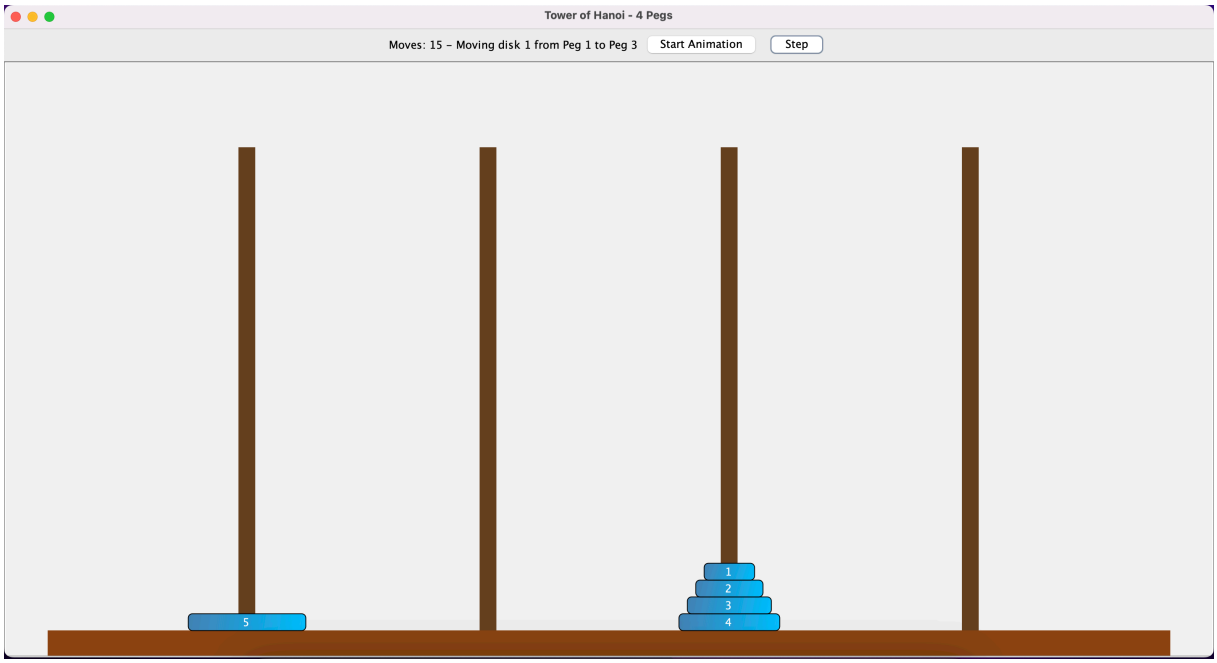

Tower of Hanoi - 4 Pegs
Moves: 10 – Moving disk 2 from Peg 4 to Peg 2
Start Animation
Step
5
2
1
4
3

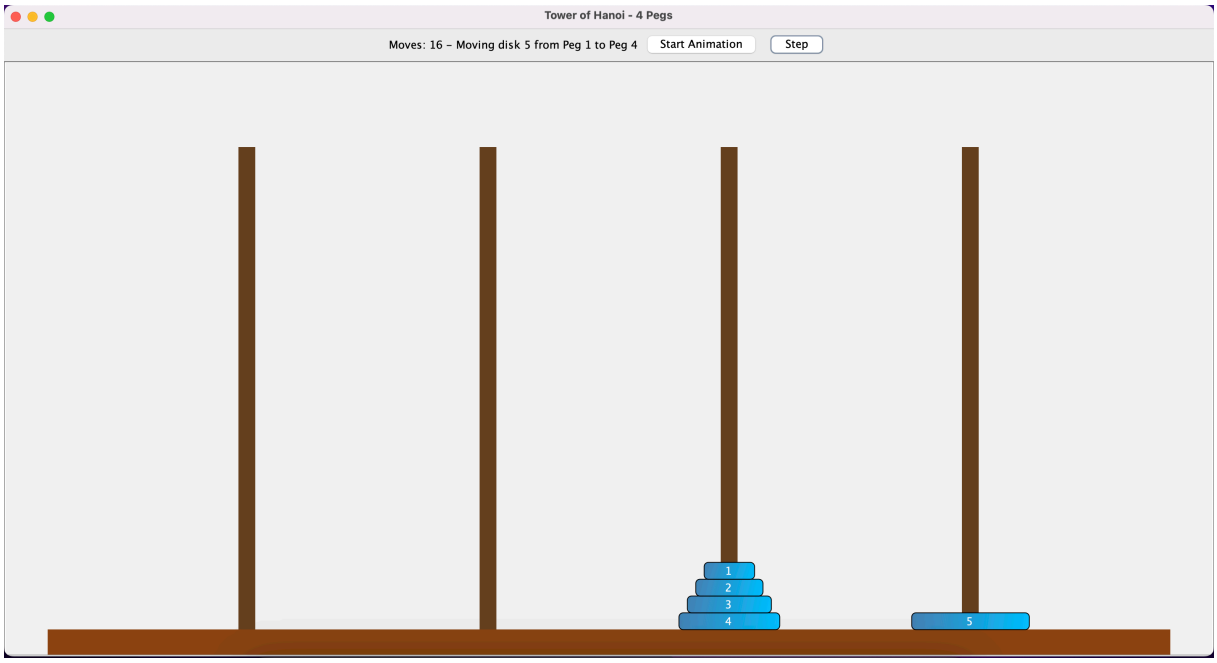

Tower of Hanoi - 4 Pegs
Moves: 11 – Moving disk 1 from Peg 3 to Peg 2
Start Animation
Step
5
1
2
4
3

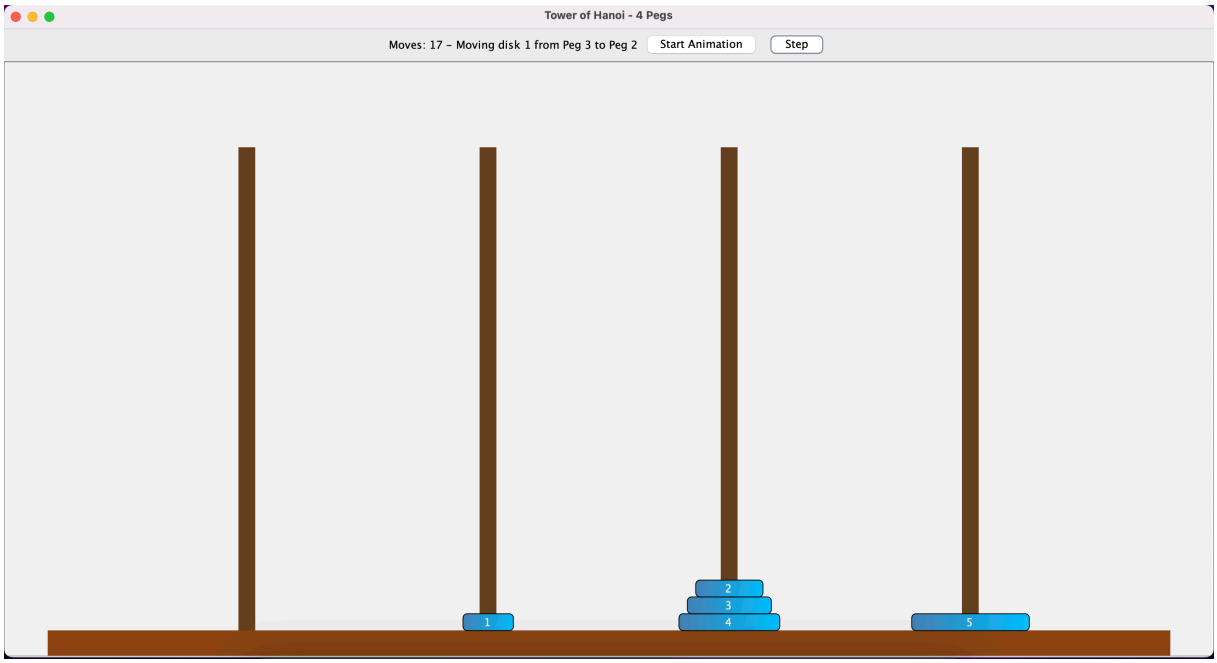

Tower of Hanoi - 4 Pegs
Moves: 12 – Moving disk 3 from Peg 4 to Peg 3
Start Animation
Step
5
1
2
3
4

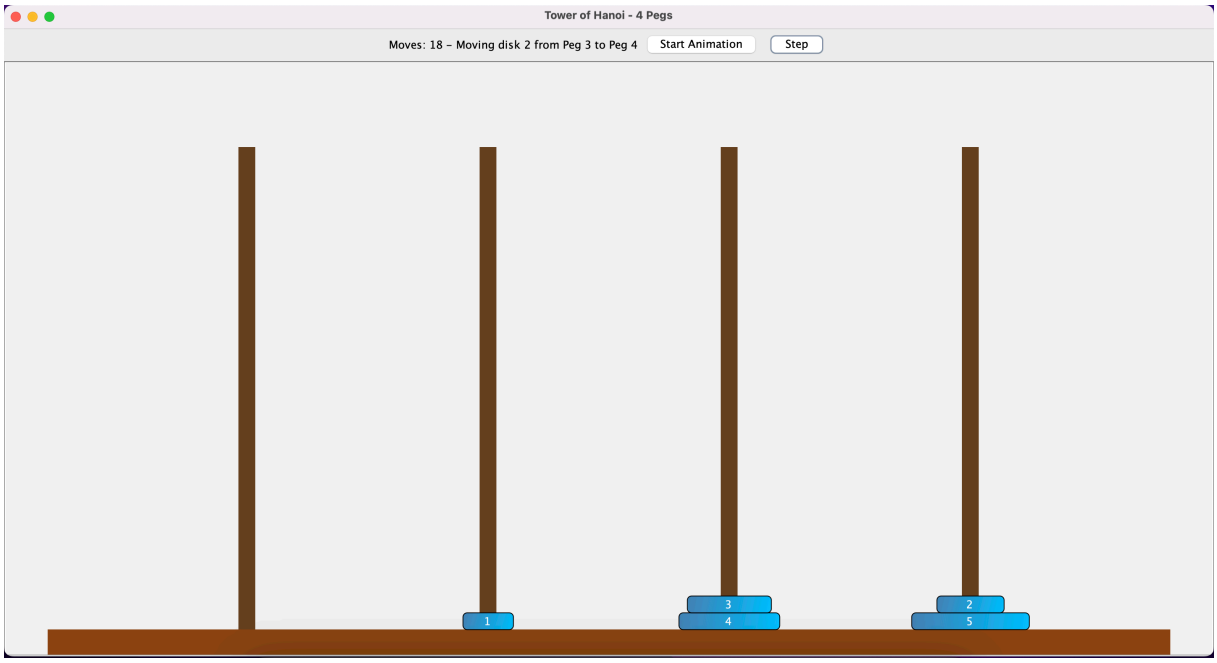

Tower of Hanoi - 4 Pegs
Moves: 13 – Moving disk 1 from Peg 2 to Peg 1
Start Animation
Step
1
5
2
3
4

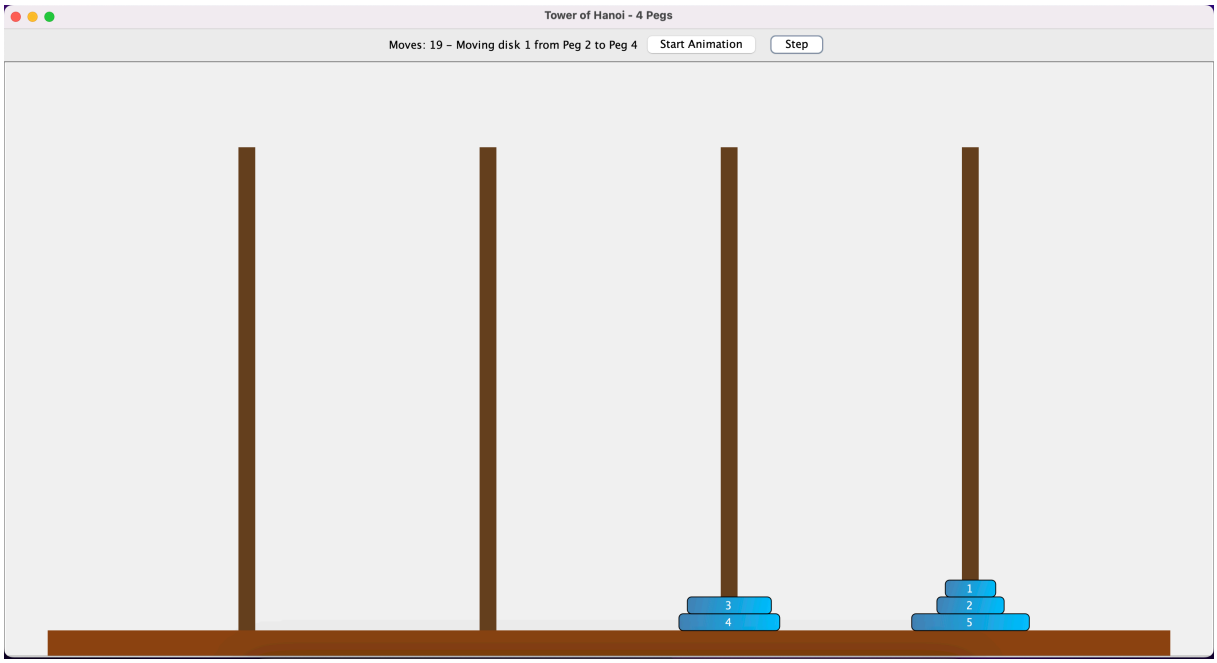

Tower of Hanoi - 4 Pegs
Moves: 14 – Moving disk 2 from Peg 2 to Peg 3
Start Animation
Step
1
5
2
3
4


Tower of Hanoi - 4 Pegs
Moves: 15 – Moving disk 1 from Peg 1 to Peg 3
Start Animation
Step
5
1
2
3
4


Tower of Hanoi - 4 Pegs
Moves: 16 – Moving disk 5 from Peg 1 to Peg 4
Start Animation
Step
1
2
3
4
5

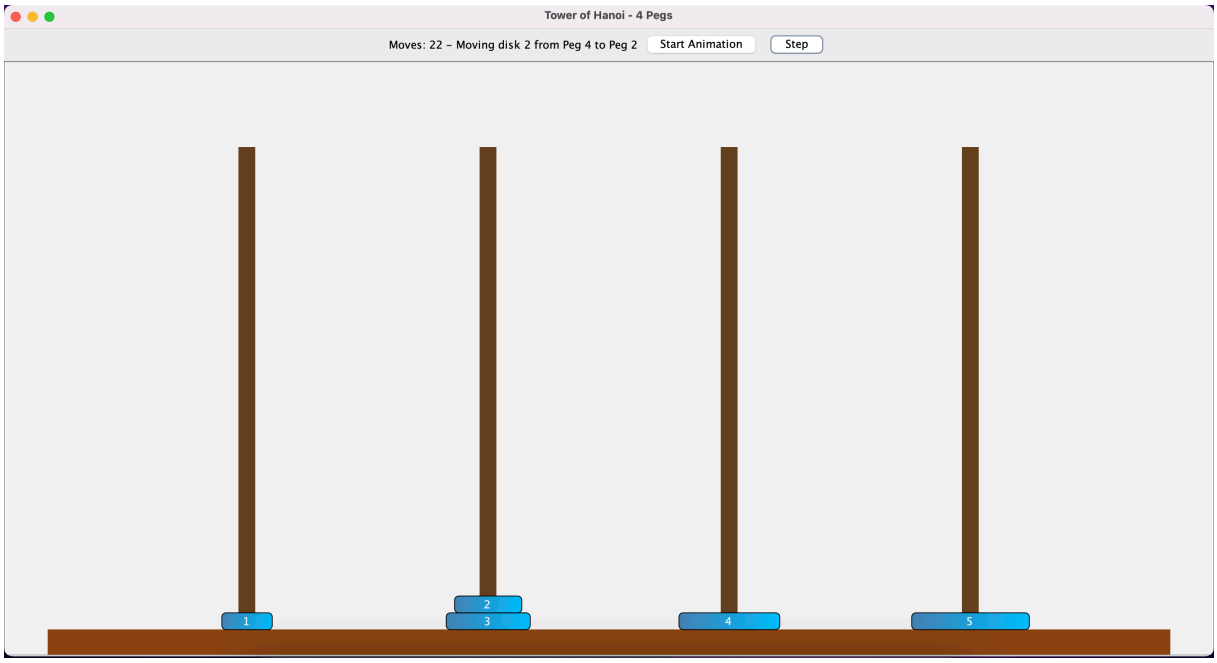

Tower of Hanoi - 4 Pegs
Moves: 17 – Moving disk 1 from Peg 3 to Peg 2
Start Animation
Step
1
2
3
4
5

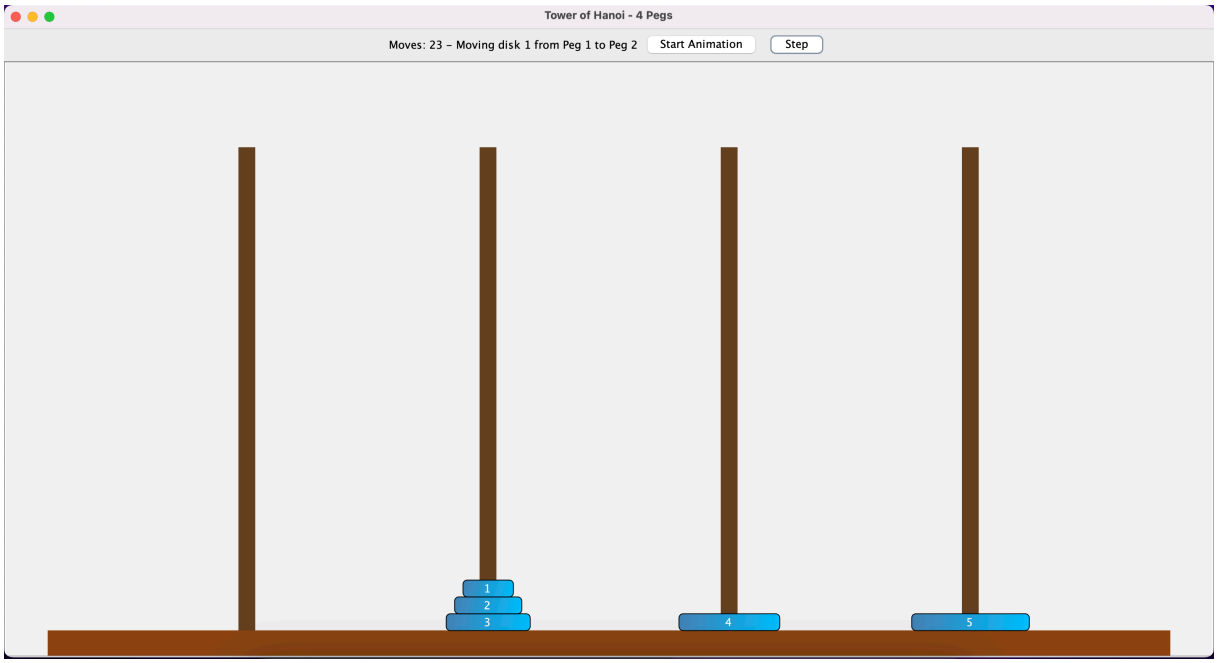

Tower of Hanoi - 4 Pegs
Moves: 18 – Moving disk 2 from Peg 3 to Peg 4
Start Animation
Step
1
3
4
2
5

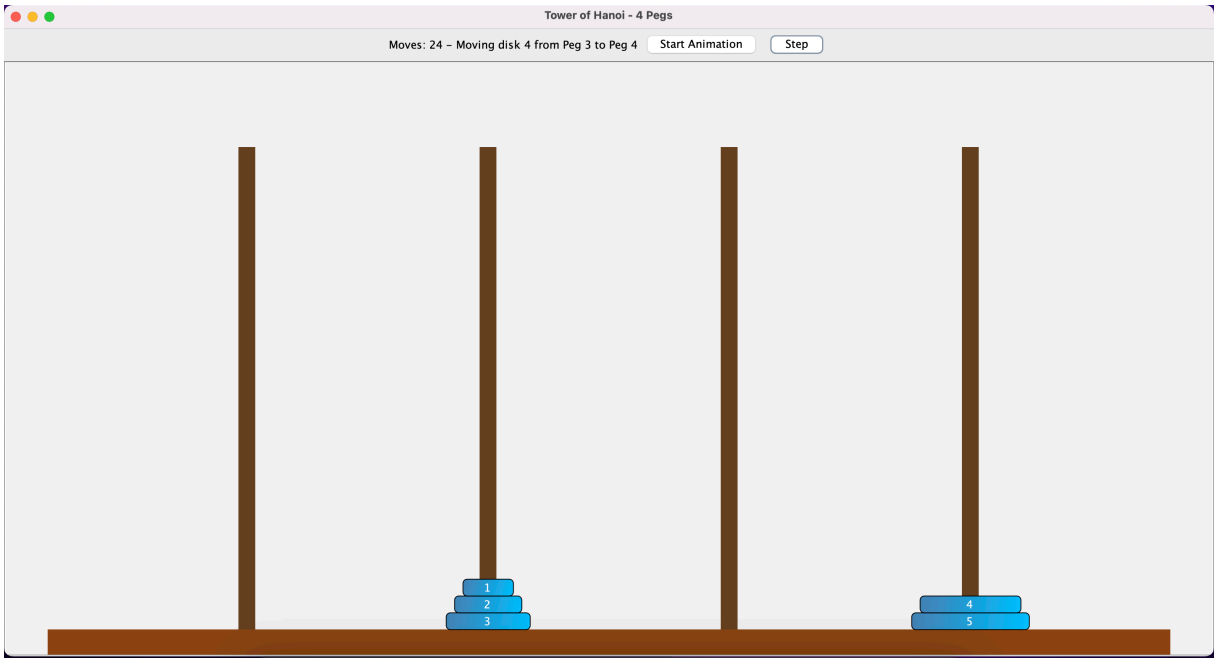

Tower of Hanoi - 4 Pegs
Moves: 19 – Moving disk 1 from Peg 2 to Peg 4
Start Animation
Step
3
4
1
2
5

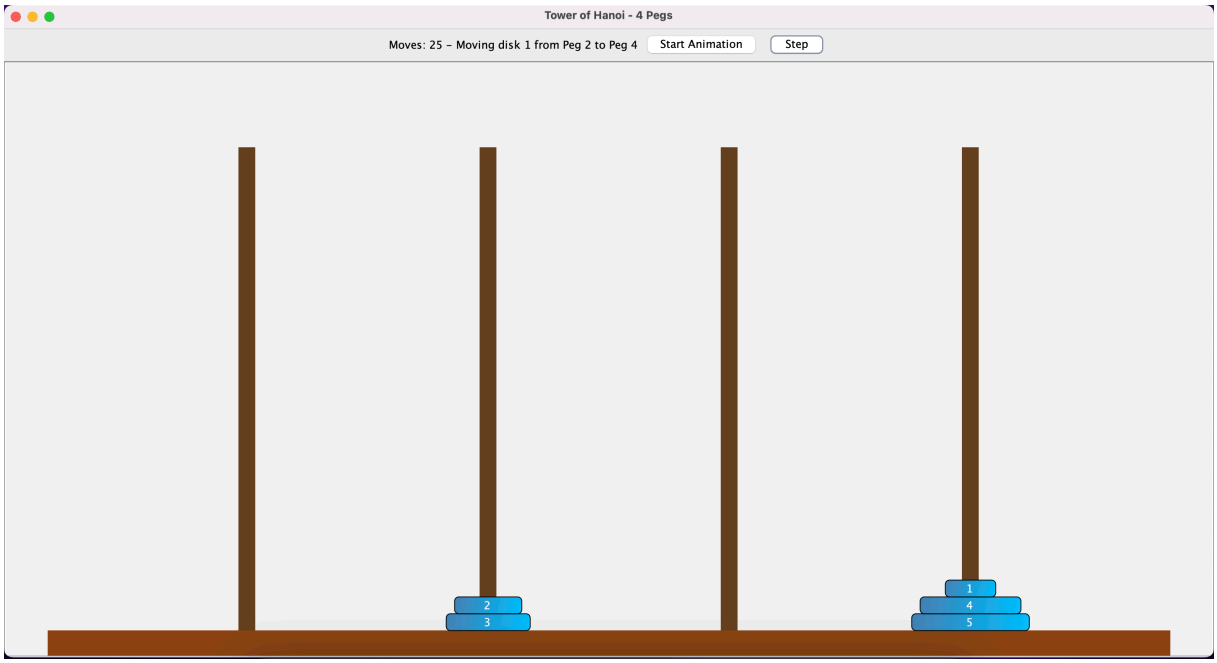

Tower of Hanoi - 4 Pegs
Moves: 20 – Moving disk 3 from Peg 3 to Peg 2
Start Animation
Step
1
2
3
4
5

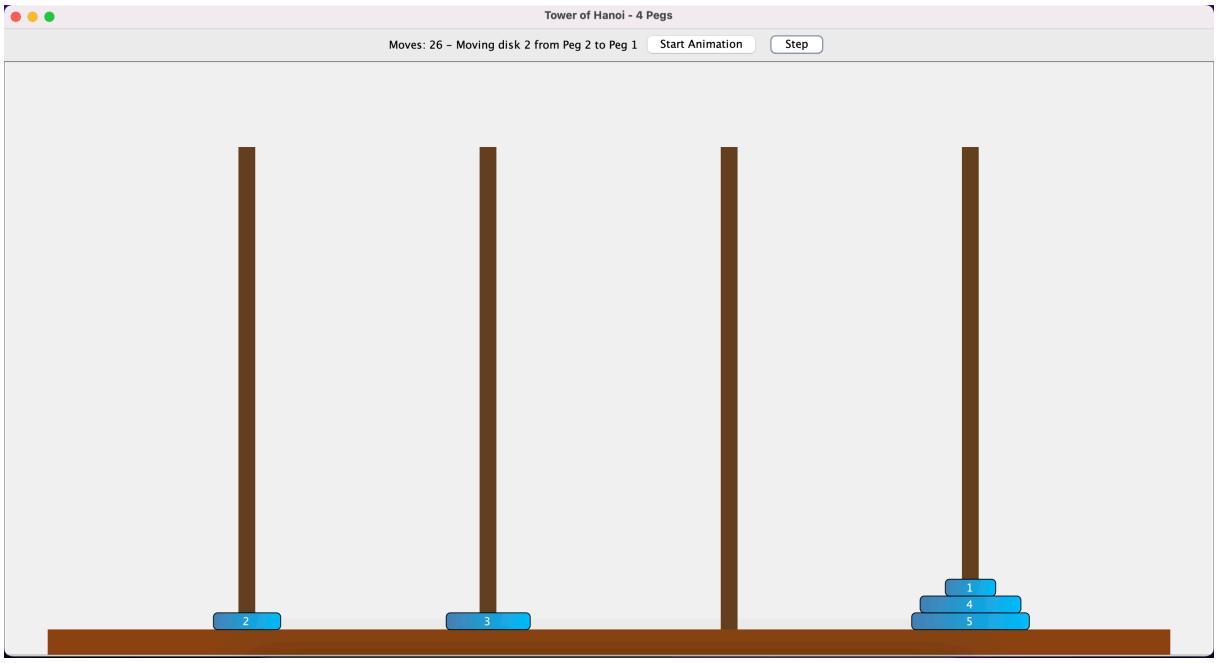

Tower of Hanoi - 4 Pegs
Moves: 21 – Moving disk 1 from Peg 4 to Peg 1
Start Animation
Step
1
2
3
4
5

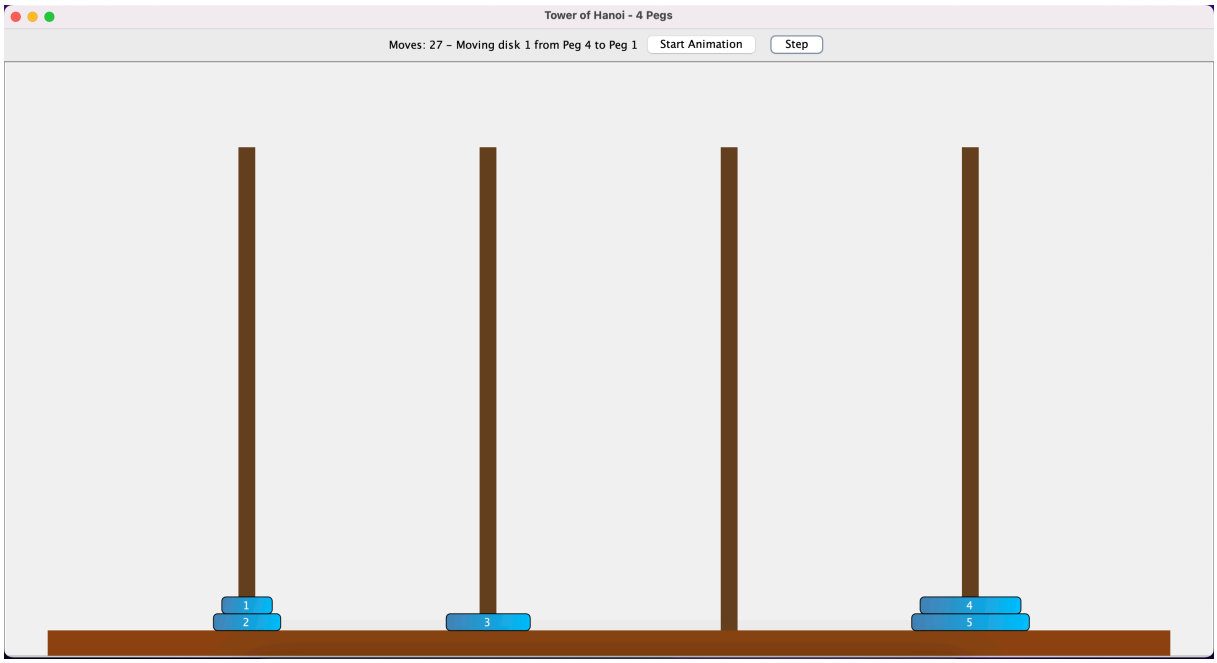

Tower of Hanoi - 4 Pegs
Moves: 22 – Moving disk 2 from Peg 4 to Peg 2
Start Animation
Step
1
2
3
4
5


Tower of Hanoi - 4 Pegs
Moves: 23 – Moving disk 1 from Peg 1 to Peg 2
Start Animation
Step
1
2
3
4
5

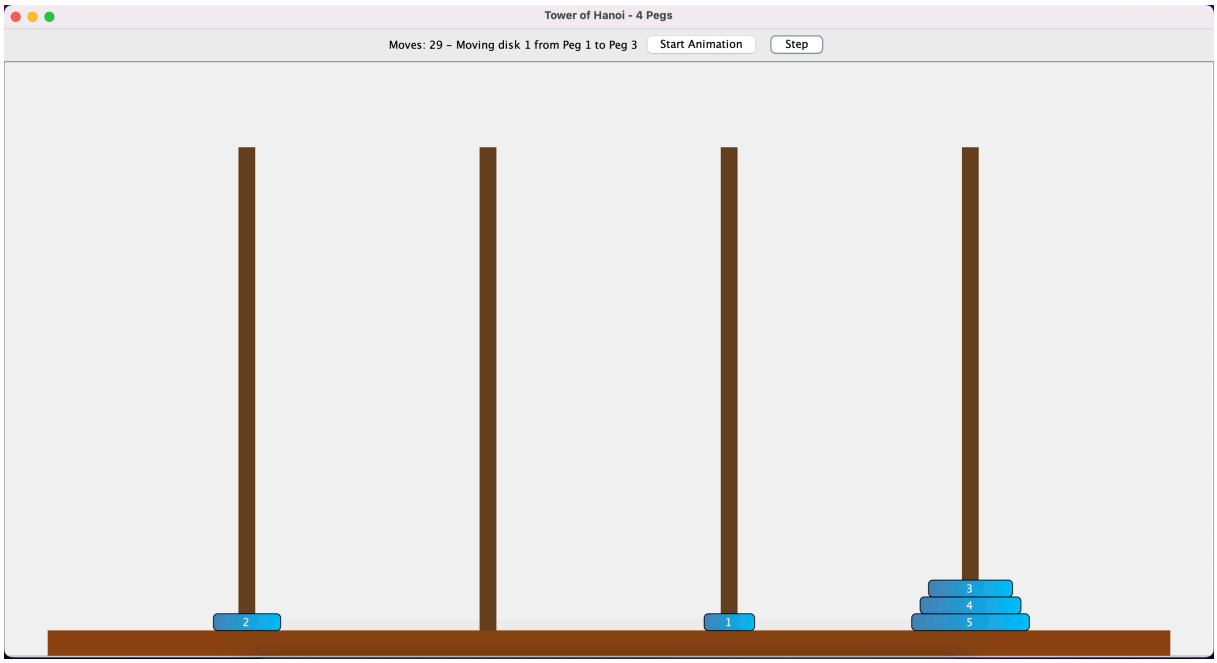

Tower of Hanoi - 4 Pegs
Moves: 24 – Moving disk 4 from Peg 3 to Peg 4
Start Animation
Step
1
2
3
4
5

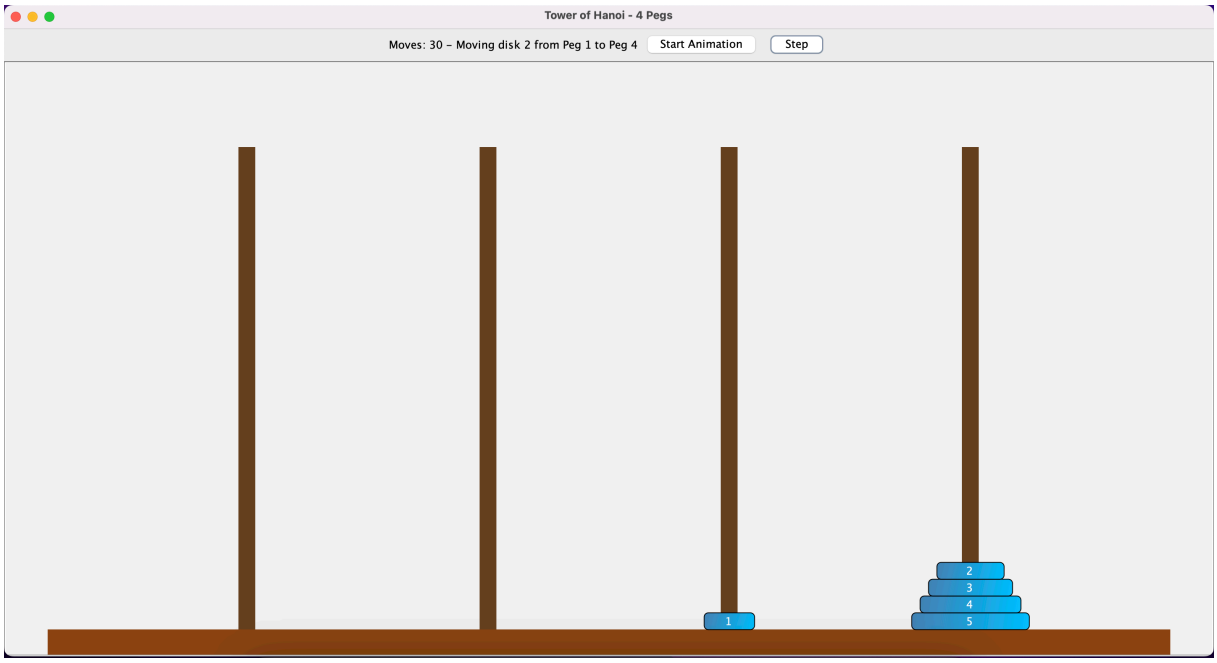

Tower of Hanoi - 4 Pegs
Moves: 25 – Moving disk 1 from Peg 2 to Peg 4
Start Animation
Step
2
3
1
4
5


Tower of Hanoi - 4 Pegs
Moves: 26 – Moving disk 2 from Peg 2 to Peg 1
Start Animation
Step
2
3
1
4
5


Tower of Hanoi - 4 Pegs
Moves: 27 – Moving disk 1 from Peg 4 to Peg 1
Start Animation
Step
1
2
3
4
5


Tower of Hanoi - 4 Pegs
Moves: 28 – Moving disk 3 from Peg 2 to Peg 4
Start Animation
Step
1
2
3
4
5


Tower of Hanoi - 4 Pegs
Moves: 29 – Moving disk 1 from Peg 1 to Peg 3
Start Animation
Step
2
1
3
4
5


Tower of Hanoi - 4 Pegs
Moves: 30 – Moving disk 2 from Peg 1 to Peg 4
Start Animation
Step
1
2
3
4
5


Tower of Hanoi - 4 Pegs
Moves: 31 – Moving disk 1 from Peg 3 to Peg 4
Start Animation
Step
1
2
3
4
5

4) Task (4)

4.1) Description

> **Description**
>
> Given a 3 × 4 chessboard with six knights:
>
> - 3 white knights on the bottom row
> - 3 black knights on the top row
>
> The task is to exchange their positions using the minimum number of knight moves, one move at a time.
>
> - Constraints:
> - Knights move using standard L-shaped chess moves
> - Only one knight per square at any time
> - No knight can move to an occupied square
> - Each move must be valid according to chess rules
> - Only iterative improvement algorithms may be used
> - Key properties:
> - The board has 12 squares, 6 occupied
> - A knight's move may reach up to 8 possible destinations depending on its position
> - Not all destinations are legal due to size limits and occupied squares
> - States can be represented as board configurations
> - Goal is to find a minimal path from initial to goal configuration

4.2) Problem Description

- Objective
  - Move all white knights to the top row and black knights to the bottom row in the minimum number of steps, following the rules:
    - Knights move in symmetric L-shaped pairs
    - Only one pair moves at each step
    - Empty spaces can only receive knights (not initiate moves)
- Input

- Predefined initial board state
- Predefined set of 7 possible move pairs
- Step-by-step board states
- Visual representation after each move
- Total move count upon completion

- Constraints
  - Solution must use the predefined move patterns
  - Optimal solution should complete in 26 steps
  - Each move must maintain board symmetry
  - Edges should be mapped according to Figure 4.1

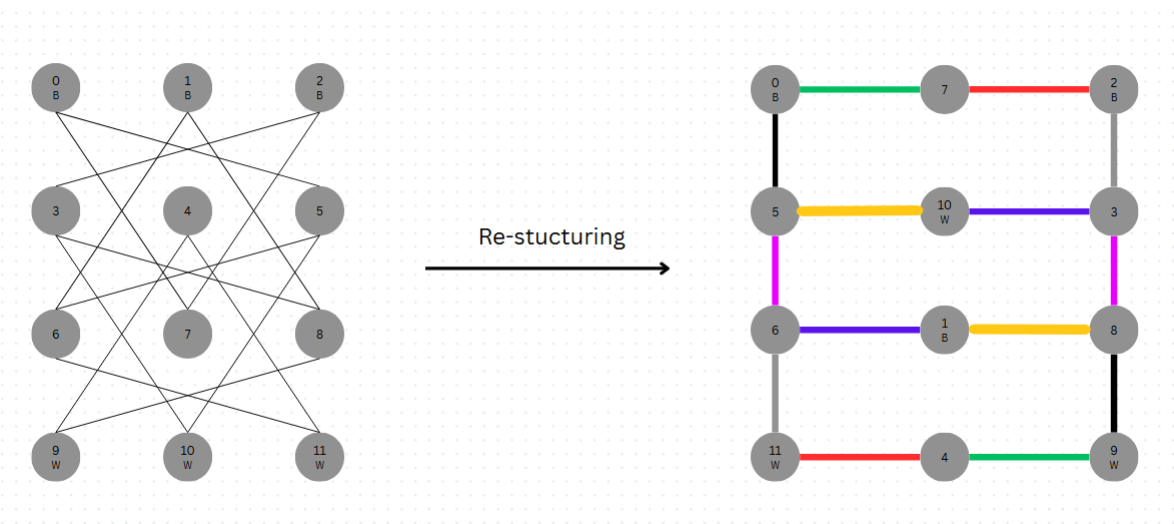


Figure 4.1: Chessboard Restructuring

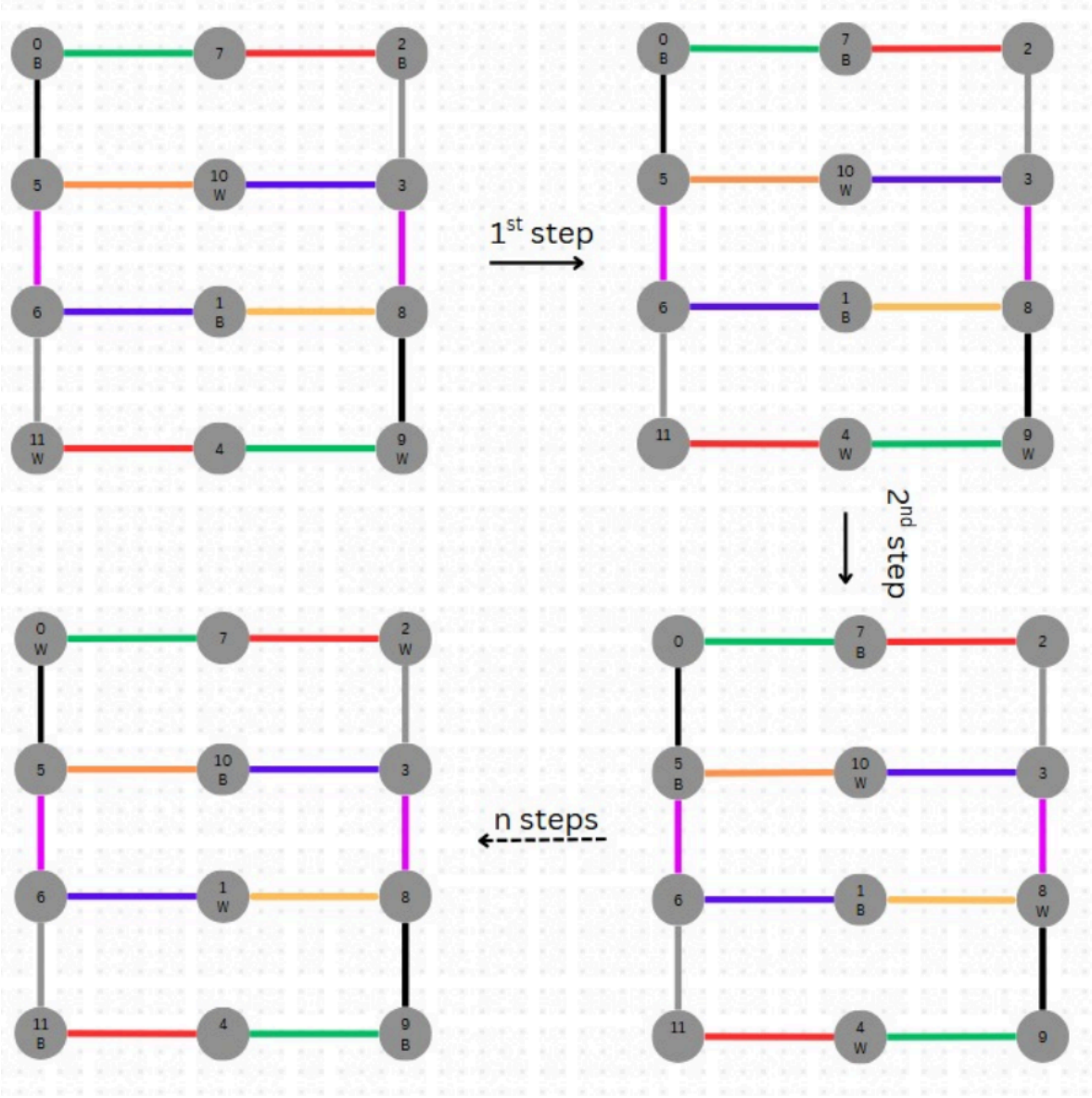


Figure 4.2: Chessboard Restructuring Steps

4.3) Detailed assumptions

- Board Configuration
  - The puzzle consists of a 4×3 grid (12 positions)
  - Initial state:
    - Top row: Black knights (B, B, B)
    - Middle rows: Empty spaces (E)
    - Bottom row: White knights (W, W, W)
- Knight Movement Rules
  - Knights move in symmetric pairs (L-shaped moves)
  - Only one pair moves at each step

  - Both knights in a pair must have valid moves

- Algorithm Choice
  - Uses iterative improvement with predefined move patterns
  - Cycles through 7 possible move configurations
  - Each iteration attempts to move both knights in a pair
- Memory Model
  - Board state stored in string array type[12]
  - Possible moves stored in 3D array Edges[7][2][2]
  - Current position tracked via index variables

4.4) Iterative Pattern-Based Algorithm

4.4.1) Pseudocode

```text
int V = 12;
char type [12] = {'B','B','B', 'E','E','E', 'E', 'E', 'E', 'W', 'W',
'W'};
int mapped_Edges [7][2][2] =
    {
        { {0,7},  {4,9}  },
        { {2,7},  {4,11} },
        { {2,3},  {6,11} },
        { {3,8},  {5,6}  },
        { {8,9},  {0,5}  },
        { {5,10}, {8,1}  },
        { {10,3}, {6,1}  }
    };
function printBoard(type):
    for each position in board:
        print piece or "E" (empty)
        print newline every 3 positions
function swapKnightPair(mapped_Edges, moveIndex, stepCount):
    get move pair from mapped_Edges[moveIndex]
    if either move would swap two empty spaces:
        return without moving
    if both moves are valid (knight to empty space):
        perform first knight swap
        increment stepCount
        perform second knight swap
        increment stepCount
```

```
  #e.g: {0,7},  {4,9} → {7,0},  {9,4}
function isGoalState(type):
    return (top row all white) and (bottom row all black)
function solvePuzzle():
    initialize board
    moveIndex = 5  # starting move pattern
    stepCount = 0
    while not isGoalState():
        swapKnightPair(mapped_Edges, moveIndex, &stepCount)
        printBoard()
        moveIndex = (moveIndex + 1) % 7
    print "Solved in" + stepCount + "steps"
main():
    create board
    solvePuzzle()
```

Listing 4.1: Knight Swap Problem - Iterative Pattern-Based Algorithm Pseudocode

4.4.2) Code

```cpp
#include <iostream>
using namespace std;
//The least number of steps = 22
class Graph
{

public:
    int V = 12;
    char type [12] = {'B','B','B', 'E','E','E', 'E', 'E', 'E', 'W', 'W',
    'W'};
    int Edges [7][2][2] =
    {
        { {0,7},  {4,9}  },
        { {2,7},  {4,11} },
        { {2,3},  {6,11} },
        { {3,8},  {5,6}  },
        { {8,9},  {0,5}  },
        { {5,10}, {8,1}  },
        { {10,3}, {6,1}  }
    };
    void printTypeArray()
    {
        for (int i = 0; i < 12; i++) {
            cout << type[i] << "   ";
            if ((i + 1) % 3 == 0) cout << endl;
```

```
        }
        cout <<"=========================================" <<endl;
    }

    void swapTypeElements(int arr[][2],int* j)
    {
        int index1 = arr[0][0];
        int index2 = arr[0][1];
        int index3 = arr[1][0];
        int index4 = arr[1][1];
        if((type[index1] == 'E' && type[index2] == 'E') || (type[index3] == 'E' && type[index4] == 'E')) return;
        if((type[index1] == 'E' || type[index2] == 'E') && (type[index3] == 'E' || type[index4] == 'E'))
        {

            char temp  = type[index1];
            type[index1] = type[index2];
            type[index2] = temp;
            (*j)++;

            temp  = type[index3];
            type[index3] = type[index4];
            type[index4] = temp;
            (*j)++;
            printTypeArray();
            //
            std::this_thread::sleep_for(std::chrono::milliseconds(100));

        }
    }
    bool goalState()
    {
        if (type[0] == 'W' && type[1] == 'W' && type[2] == 'W' && type[9] == 'B' && type[10] == 'B' && type[11] == 'B')
        {
            return true;
        }
        return false;
    }

};
```

```cpp
void iterativeImprovement(Graph &g)
{
    int i = 5;
    int steps = 0;
    while (!g.goalState() )
    {
        g.swapTypeElements(g.Edges[i], &steps);
        i = (i+1)%7;
    }
    cout<<"Puzzle solved successfully in " << steps << " steps!"<<endl;

}



int main()
{
    Graph g;
    g.printTypeArray();
    iterativeImprovement(g);

    return 0;
}
```

Listing 4.2: Knight Swap Problem - Iterative Pattern-Based Algorithm C++ Code

4.4.3) Complexity analysis

- Time Complexity
  - This task has no inputs to Evaluate Time Complexity
- Space Complexity
  - Board State: O(12) → O(1) (fixed size)
  - Move Patterns: O(7×2×2) → O(1) (fixed size)
  - Total: O(1) constant space

4.4.4) Test Cases

- Input:

```cpp
int V = 12;
char type [12] = {'B','B','B', 'E','E','E', 'E', 'E', 'E', 'W', 'W',
'W'};
```

- Ouput:

Output

```
B   B   B
E   E   E
E   E   E
W   W   W
═══════════════════════════════════════

B   E   B
E   E   W
E   E   B
W   E   W
═══════════════════════════════════════

E   E   B
E   W   W
E   B   B
E   E   W
═══════════════════════════════════════

E   E   E
B   W   W
W   B   B
E   E   E
═══════════════════════════════════════

W   E   E
B   W   E
W   B   E
B   E   E
═══════════════════════════════════════

W   W   E
E   W   E
E   B   E
B   B   E
═══════════════════════════════════════

W   W   B
E   E   E
E   E   E
B   B   W
═══════════════════════════════════════

W   W   E
B   E   E
W   E   E
B   B   E
```

```
═══════════════════════════════════════

W   W   E

E   E   W

E   E   B

B   B   E

═══════════════════════════════════════

W   E   E

B   E   W

W   E   B

B   E   E

═══════════════════════════════════════

E   E   E

B   B   W

W   W   B

E   E   E

═══════════════════════════════════════

E   E   W

B   E   W

W   E   B

E   E   B

═══════════════════════════════════════

W   E   W

B   E   E

W   E   E

B   E   B

═══════════════════════════════════════

W   W   W

E   E   E

E   E   E

B   B   B

═══════════════════════════════════════

Puzzle solved successfully in 26 steps!
```

Figure 4.3: Knight Swap - Iterative Pattern-Based Algorithm Test Case Output

4.5) Brute-Force Backtracking Solution

```text
function solveKnightSwap(board, knightPositions, goalPositions,
moveSequence, visitedStates):
    if knightPositions == goalPositions:
        recordSolution(moveSequence)
        return
```

```
    stateKey = serialize(knightPositions)
    if stateKey in visitedStates:
        return
    visitedStates.add(stateKey)

    for i in 0 to length(knightPositions) - 1:
        (x, y) = knightPositions[i]
        for (dx, dy) in knightMoves:
            nx = x + dx
            ny = y + dy
            if isValid(nx, ny, board) and board[nx][ny] == empty:
                newPositions = copy(knightPositions)
                newPositions[i] = (nx, ny)
                board[x][y] = empty
                board[nx][ny] = knightID(i)
                moveSequence.append((i, (x, y), (nx, ny)))

                solveKnightSwap(board, newPositions, goalPositions,
                moveSequence, visitedStates)

                board[nx][ny] = empty
                board[x][y] = knightID(i)
                moveSequence.pop()

function isValid(x, y, board):
    return x ≥ 0 and x < 3 and y ≥ 0 and y < 4

function serialize(knightPositions):
    return sorted list of positions as a string

function knightID(index):
    return index + 1  // unique ID per knight

initialize:
    board = 3x4 grid with empty cells
    knightPositions = [(2,0), (2,1), (2,2), (0,1), (0,2), (0,3)] // 3
    white, 3 black
    goalPositions = [(0,0), (0,1), (0,2), (2,1), (2,2), (2,3)]
    knightMoves = [(2,1), (1,2), (-1,2), (-2,1), (-2,-1), (-1,-2),
    (1,-2), (2,-1)]
    visitedStates = empty set
```

```
    moveSequence = empty list

    solveKnightSwap(board, knightPositions, goalPositions, moveSequence,
    visitedStates)
```

Listing 4.3: Knight Swap Problem - Brute-Force Backtracking Solution Pseudocode

4.6) Comparison Between the Frame-Stewart Algorithm and the Brute-Force backtracking Solution

Aspect	Iterative Pattern-Based	Brute-Force Backtracking
Strategy	Cycles through predefined move patterns	Explores all possible move sequences
Completeness	Only finds solutions using specific patterns	Guaranteed to find all possible solutions
Time Complexity	This task has no inputs to Evaluate Time Complexity :)	$O(b^d)$ — Exponential (branching factor b, depth d)
Space Complexity	$O(1)$ — Constant space	$O(d)$ — Proportional to recursion depth
Implementation	Simple, hardcoded logic	More complex, recursive with state tracking
Move Selection	Deterministic pattern cycling	Systematic trial-and-error
Flexibility	Only works for this specific puzzle	Adaptable to rule variations
Best For	Known puzzles with established patterns	New puzzles or when optimal path is unknown

Table 4.1: Knight Swap Problem - Algorithm Comparison

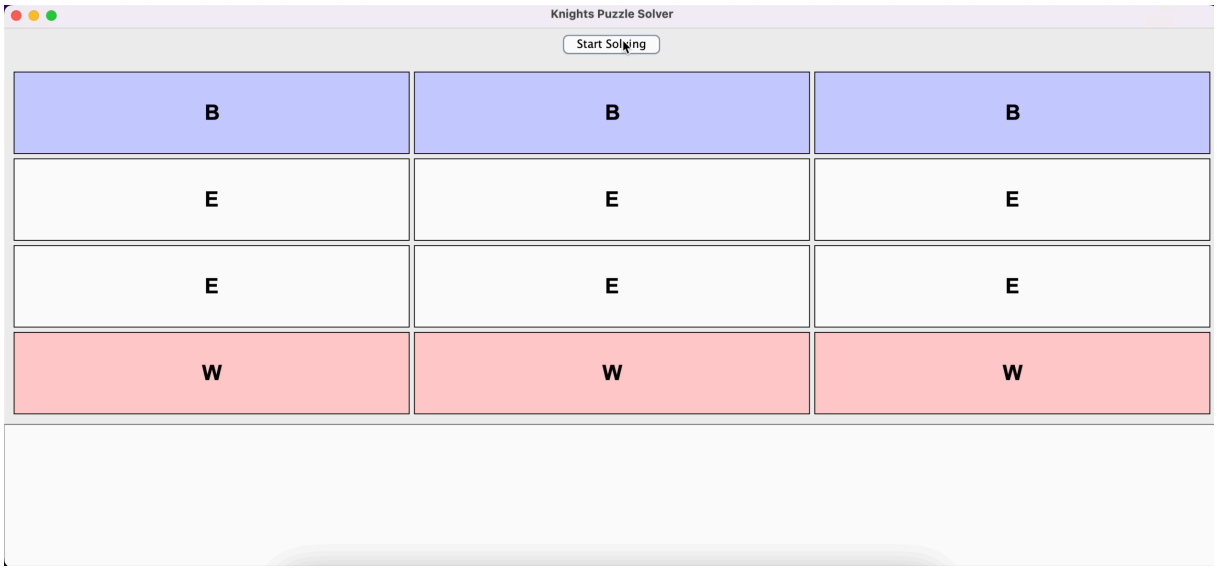
4.7) Conclusion

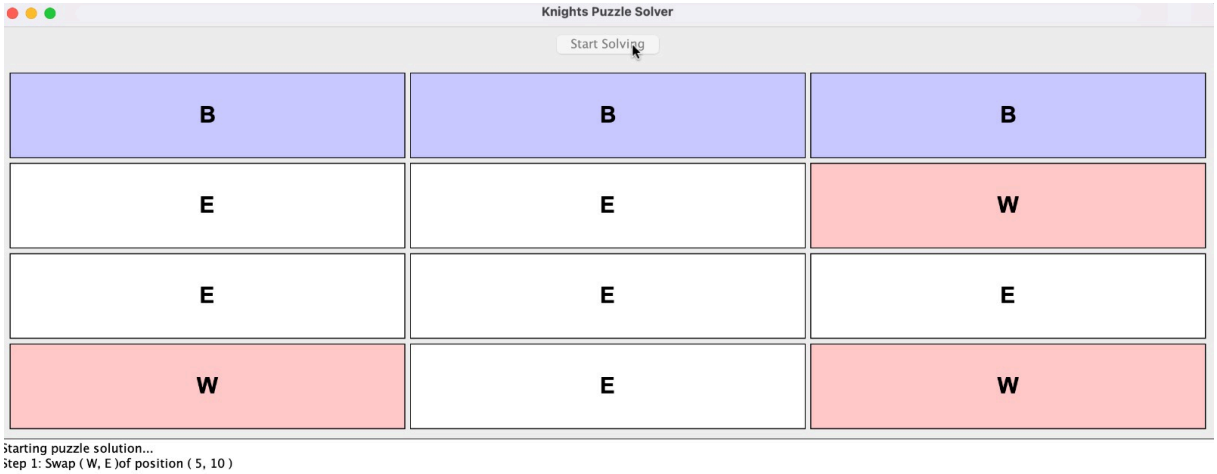
- The iterative knights puzzle solver provides an efficient solution with:
  - Predictable performance (always 22 moves)
  - Simple implementation
  - Constant space requirements
  - Clear visualization of each step
- Key Advantages:
  - Guaranteed optimal solution
  - Easy to understand and verify
  - Efficient memory usage
- Limitations:
  - Hardcoded move patterns limit flexibility
  - Requires predefined knowledge of optimal path

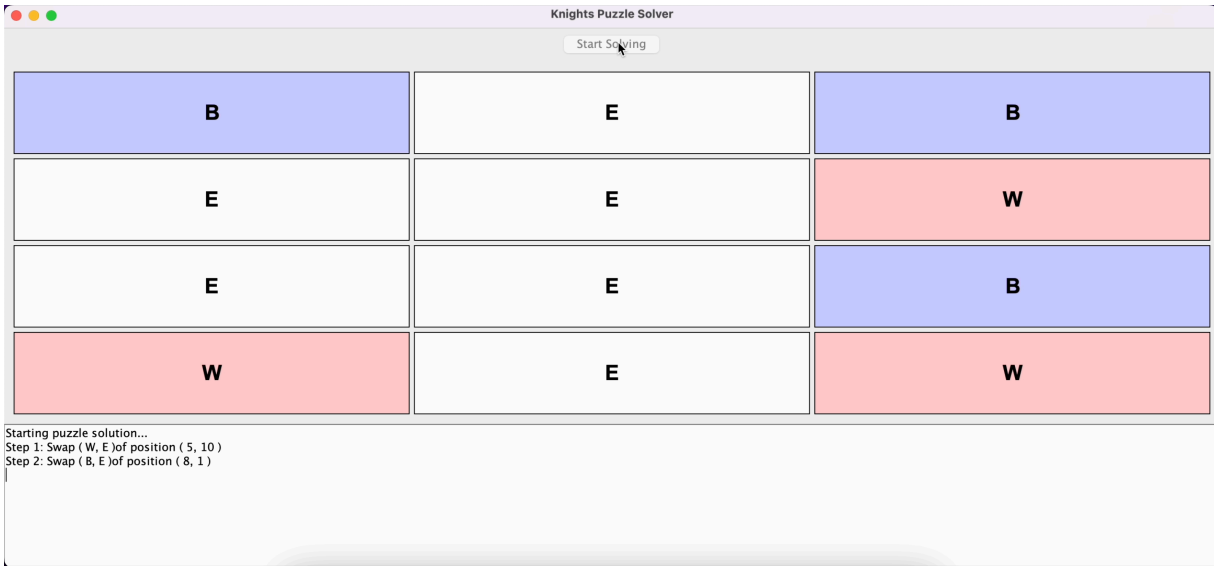
- Less adaptable to rule variations

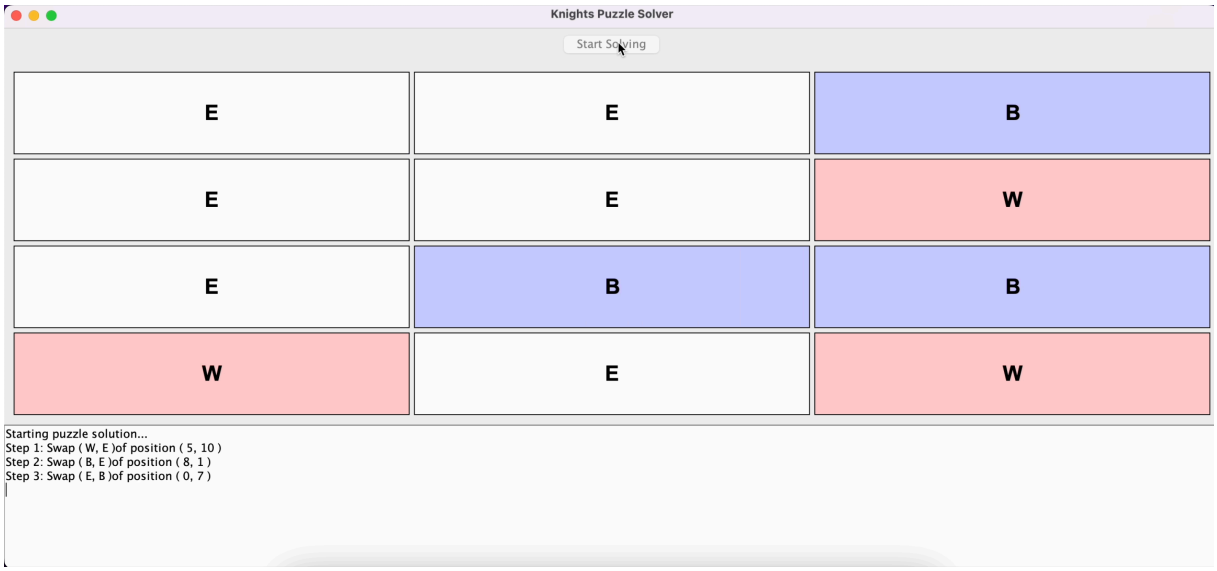
- Effectiveness:
  - Perfect for this specific puzzle configuration
  - Demonstrates how iterative approaches can solve symmetric puzzles
  - Provides foundation for more generalized solvers

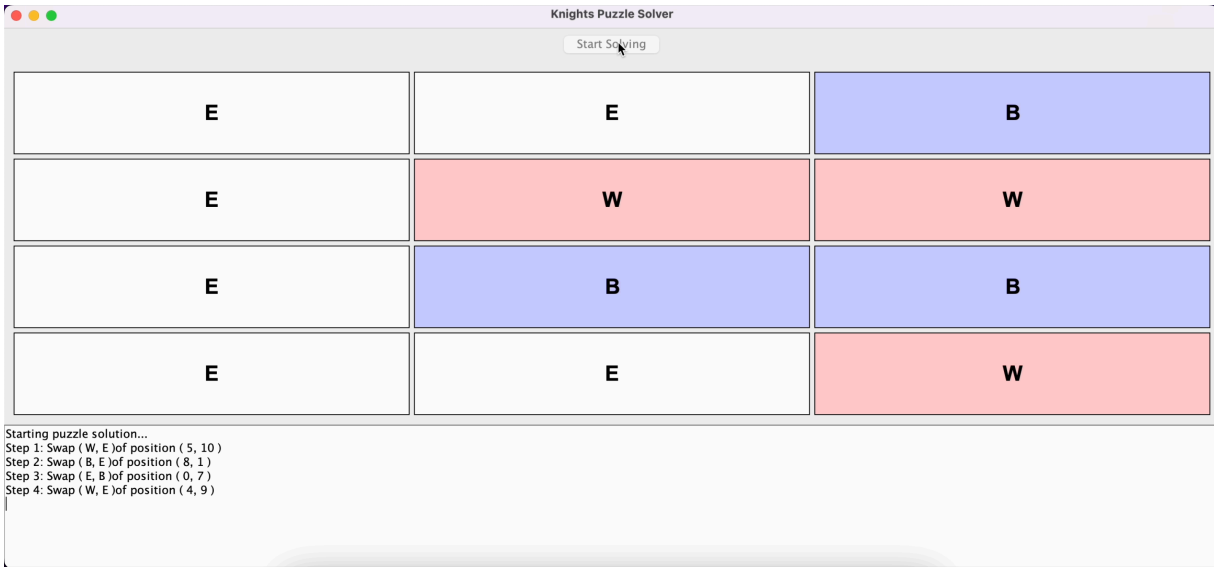
4.8) GUI Screenshots

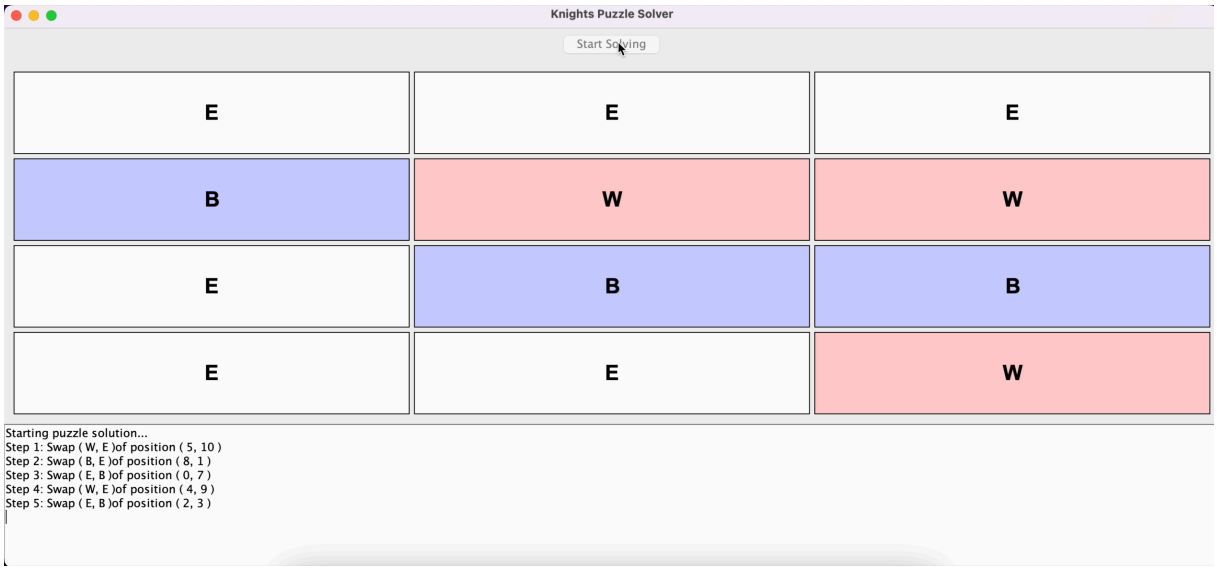


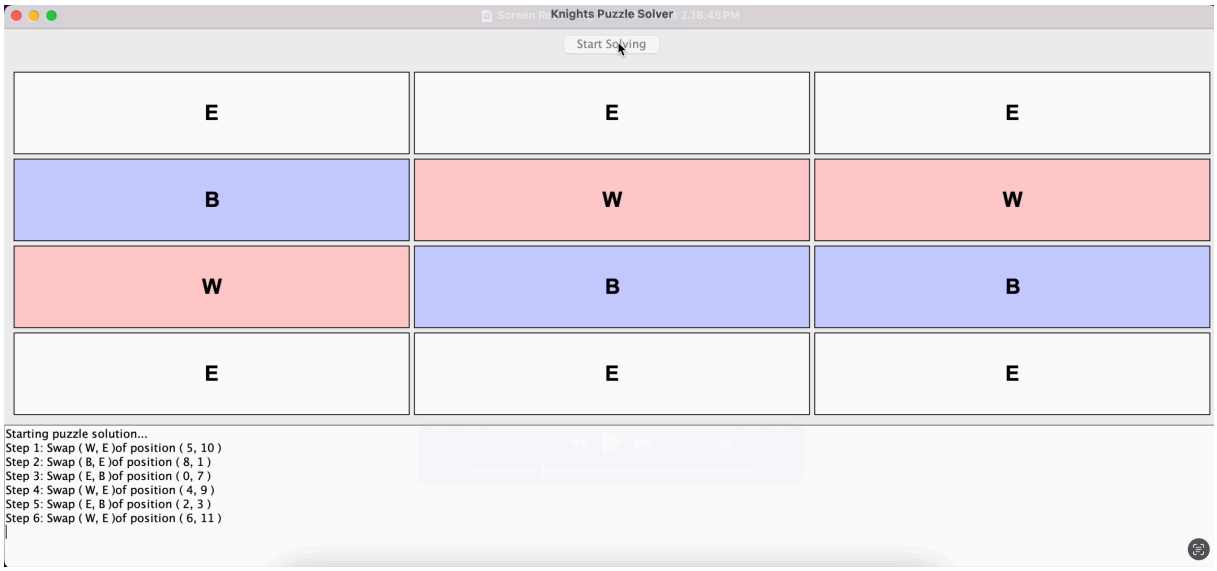


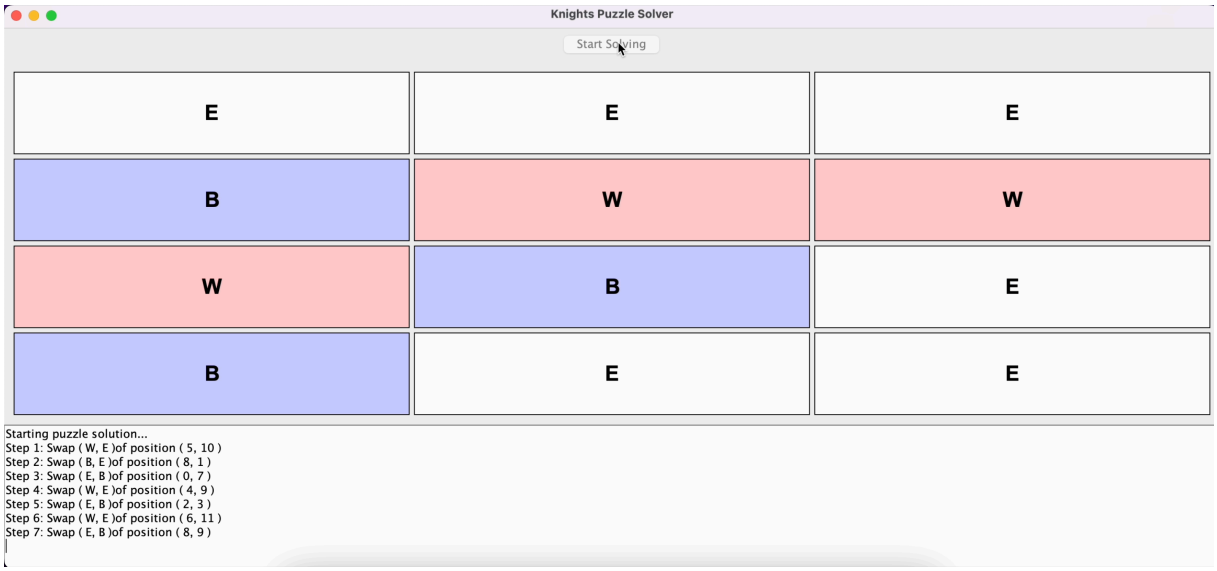

Knights Puzzle Solver
Start Solving
B E B
E E W
E E B
W E W
Starting puzzle solution...
Step 1: Swap (W, E)of position (5, 10)
Step 2: Swap (B, E)of position (8, 1)

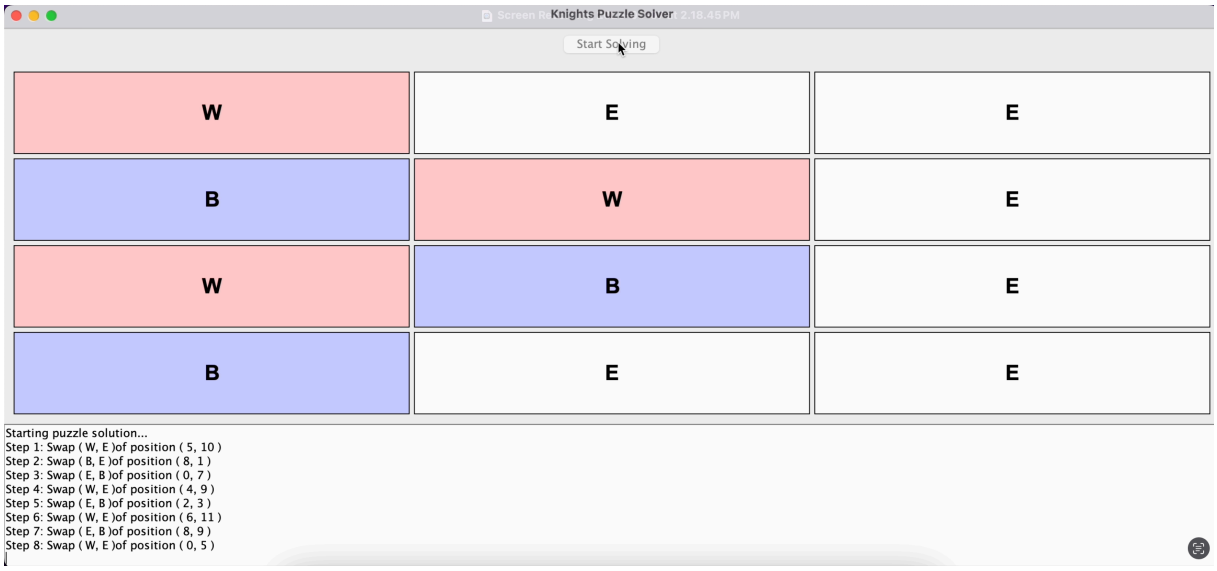

Knights Puzzle Solver
Start Solving
E E B
E E W
E B B
W E W
Starting puzzle solution...
Step 1: Swap (W, E)of position (5, 10)
Step 2: Swap (B, E)of position (8, 1)
Step 3: Swap (E, B)of position (0, 7)

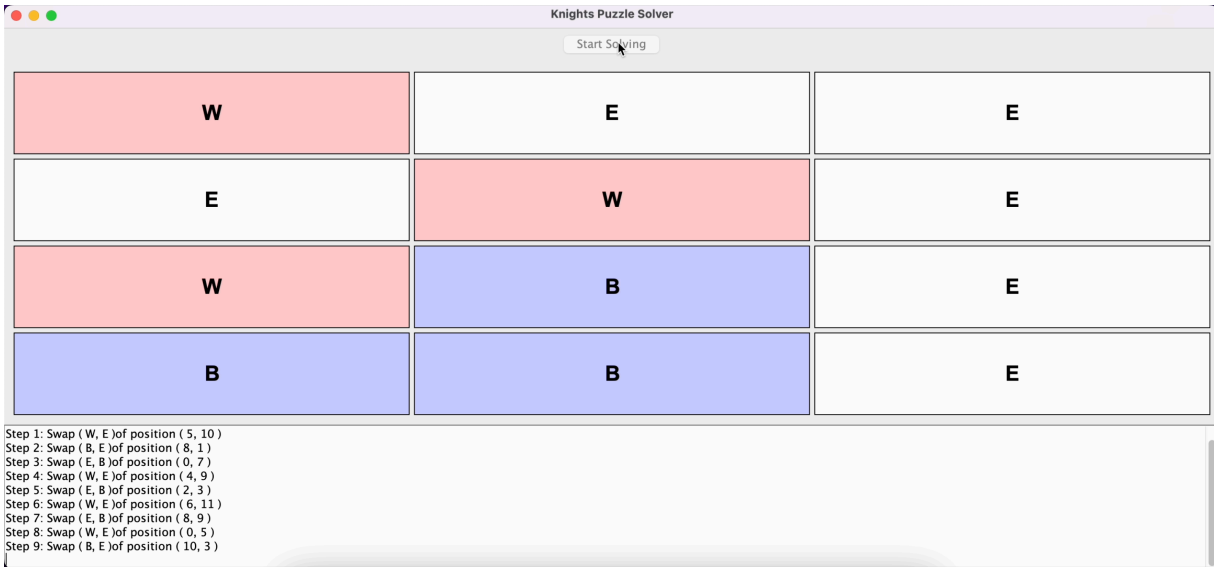

Knights Puzzle Solver
Start Solving
E E B
E W W
E B B
E E W
Starting puzzle solution...
Step 1: Swap (W, E)of position (5, 10)
Step 2: Swap (B, E)of position (8, 1)
Step 3: Swap (E, B)of position (0, 7)
Step 4: Swap (W, E)of position (4, 9)

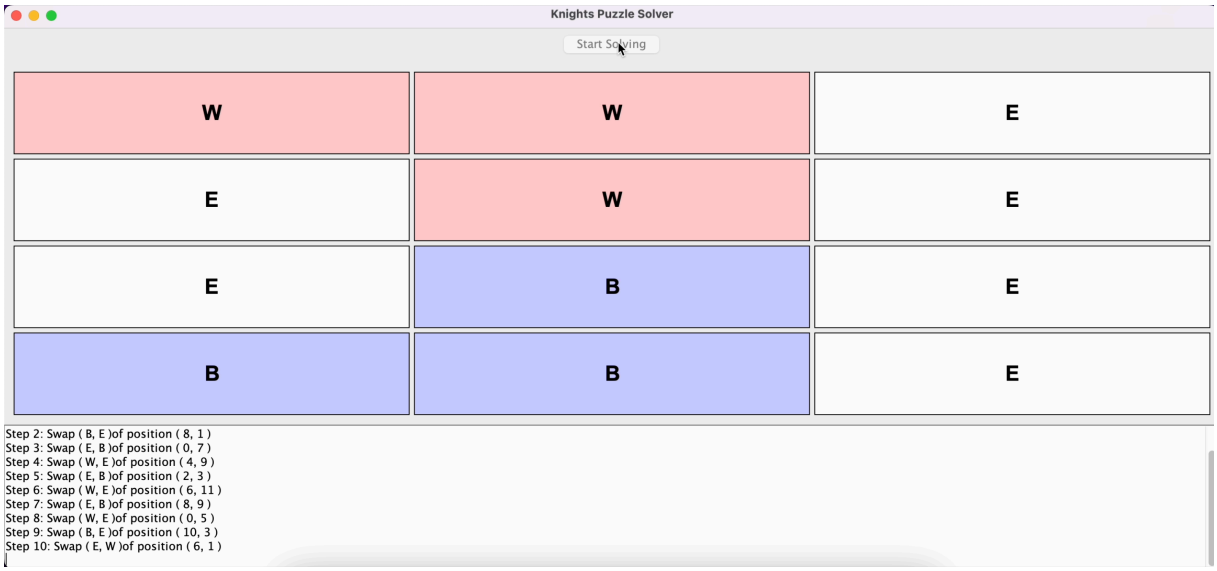

Knights Puzzle Solver
Start Solving
E E E
B W W
E B B
E E W
Starting puzzle solution...
Step 1: Swap (W, E)of position (5, 10)
Step 2: Swap (B, E)of position (8, 1)
Step 3: Swap (E, B)of position (0, 7)
Step 4: Swap (W, E)of position (4, 9)
Step 5: Swap (E, B)of position (2, 3)

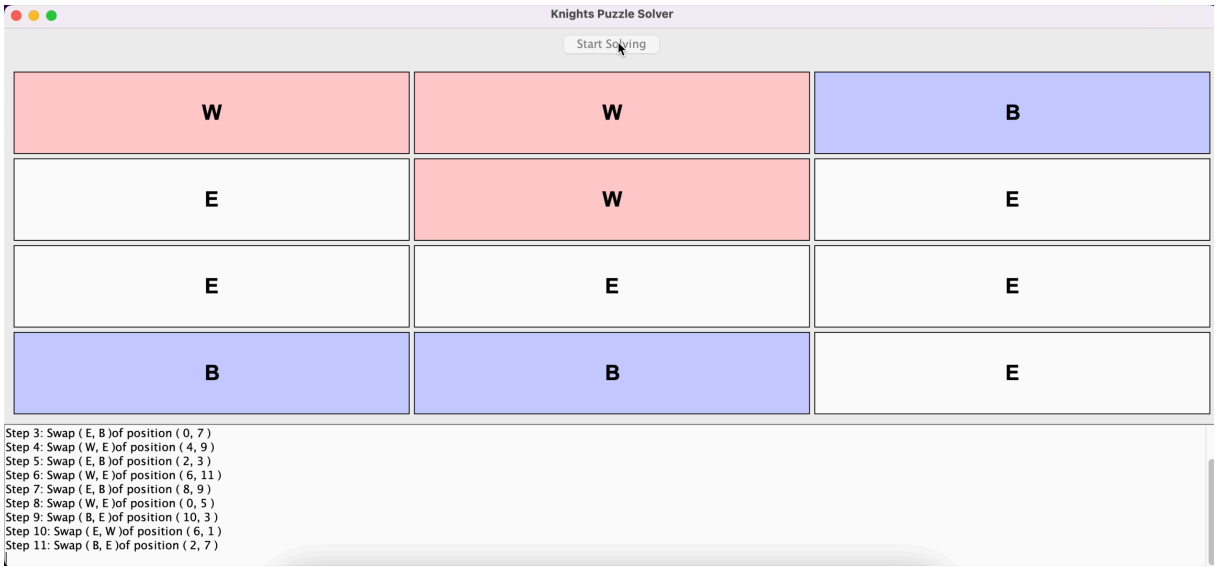

Knights Puzzle Solver
Start Solving
E E E
B W W
W B B
E E E
Starting puzzle solution...
Step 1: Swap (W, E)of position (5, 10)
Step 2: Swap (B, E)of position (8, 1)
Step 3: Swap (E, B)of position (0, 7)
Step 4: Swap (W, E)of position (4, 9)
Step 5: Swap (E, B)of position (2, 3)
Step 6: Swap (W, E)of position (6, 11)

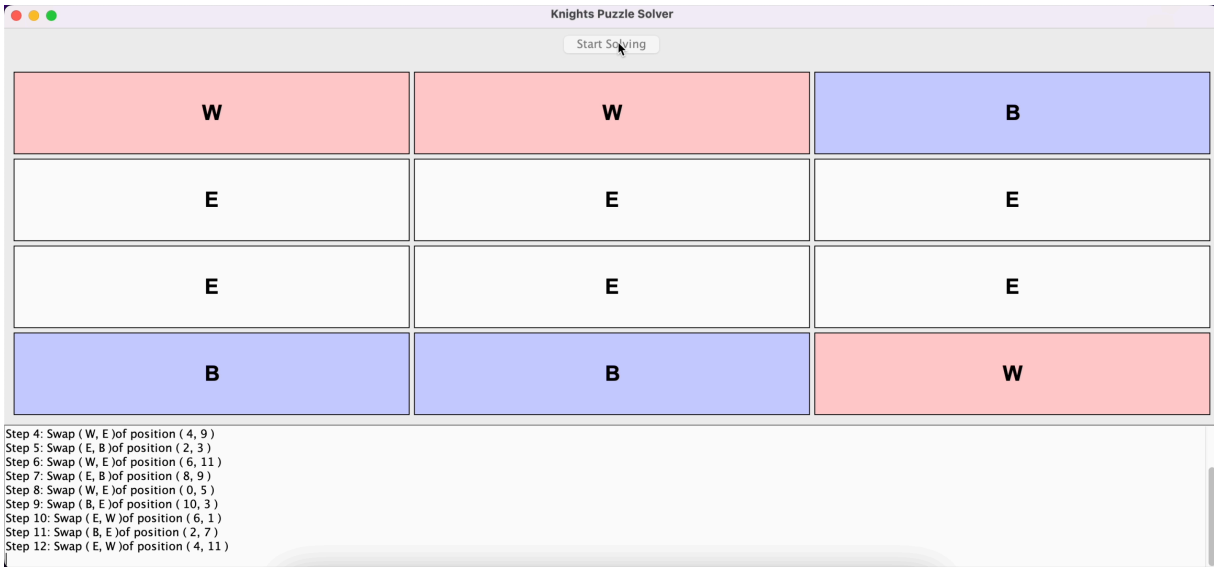

Knights Puzzle Solver
Start Solving
E E E
B W W
W B E
B E E
Starting puzzle solution...
Step 1: Swap (W, E)of position (5, 10)
Step 2: Swap (B, E)of position (8, 1)
Step 3: Swap (E, B)of position (0, 7)
Step 4: Swap (W, E)of position (4, 9)
Step 5: Swap (E, B)of position (2, 3)
Step 6: Swap (W, E)of position (6, 11)
Step 7: Swap (E, B)of position (8, 9)

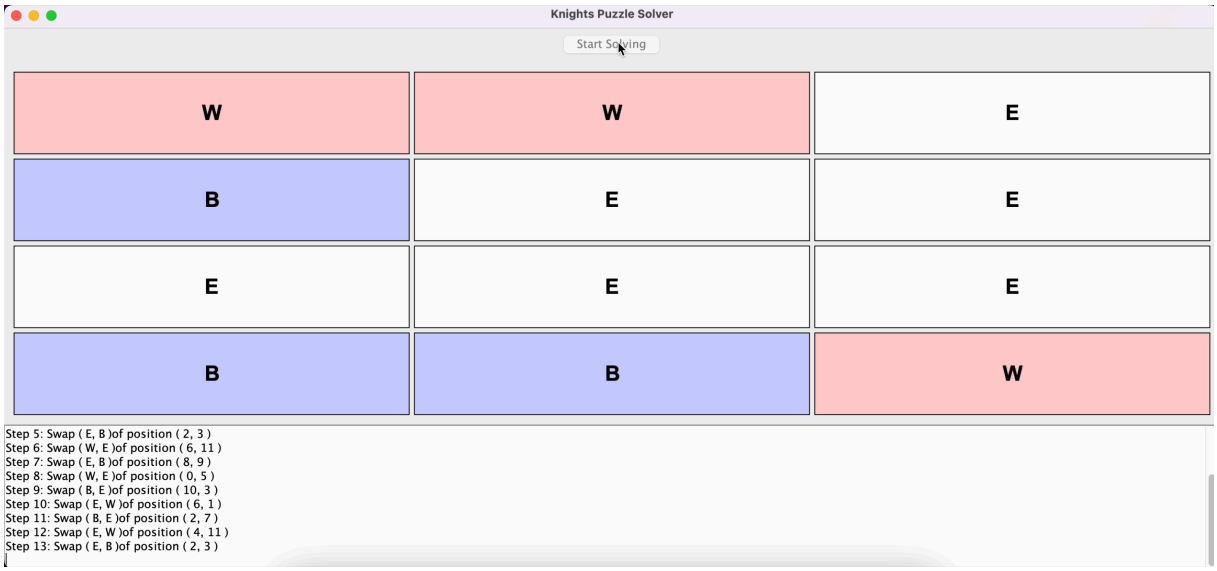

Knights Puzzle Solver
Start Solving
W E E
B W E
W B E
B E E
Starting puzzle solution...
Step 1: Swap (W, E)of position (5, 10)
Step 2: Swap (B, E)of position (8, 1)
Step 3: Swap (E, B)of position (0, 7)
Step 4: Swap (W, E)of position (4, 9)
Step 5: Swap (E, B)of position (2, 3)
Step 6: Swap (W, E)of position (6, 11)
Step 7: Swap (E, B)of position (8, 9)
Step 8: Swap (W, E)of position (0, 5)

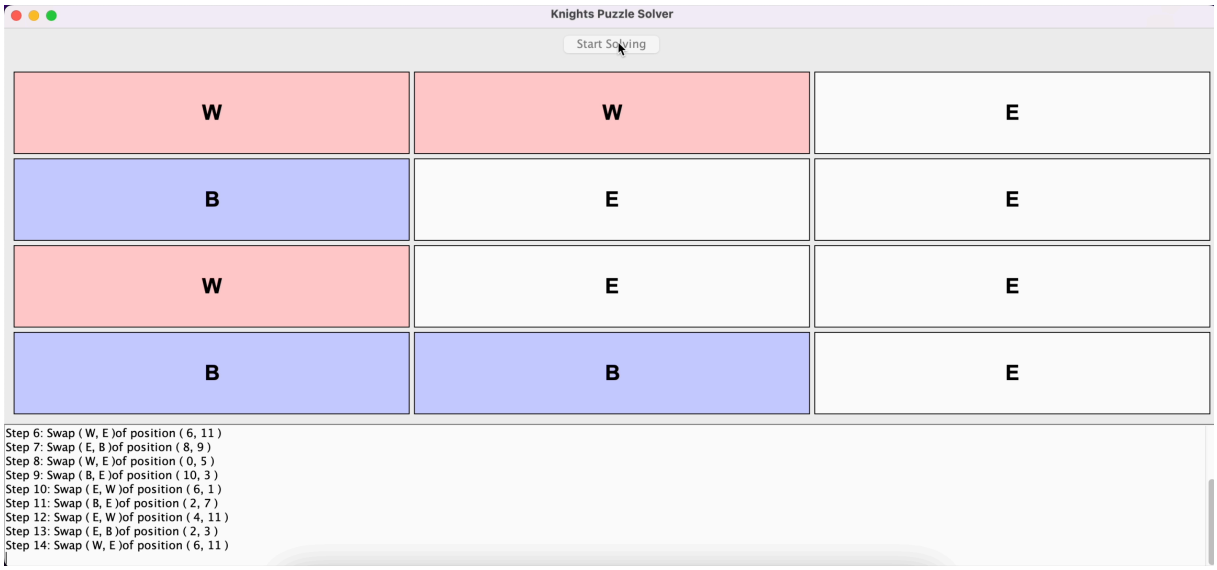

Knights Puzzle Solver
Start Solving
W E E
E W E
W B E
B B E
Step 1: Swap (W, E)of position (5, 10)
Step 2: Swap (B, E)of position (8, 1)
Step 3: Swap (E, B)of position (0, 7)
Step 4: Swap (W, E)of position (4, 9)
Step 5: Swap (E, B)of position (2, 3)
Step 6: Swap (W, E)of position (6, 11)
Step 7: Swap (E, B)of position (8, 9)
Step 8: Swap (W, E)of position (0, 5)
Step 9: Swap (B, E)of position (10, 3)

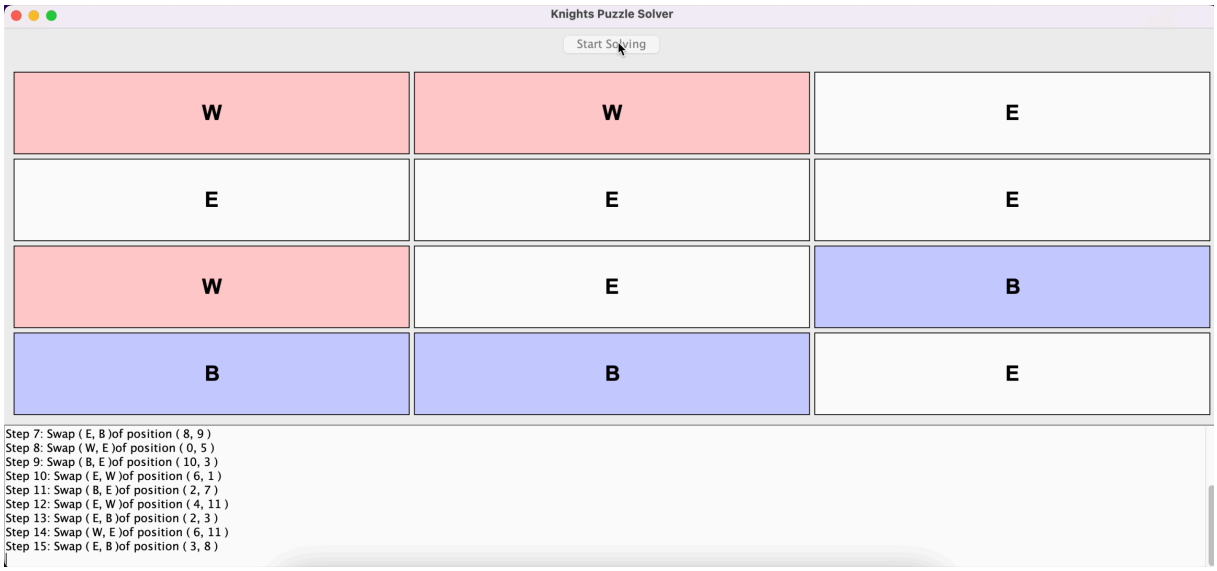

Knights Puzzle Solver
Start Solving
W W E
E W E
E B E
B B E
Step 2: Swap (B, E)of position (8, 1)
Step 3: Swap (E, B)of position (0, 7)
Step 4: Swap (W, E)of position (4, 9)
Step 5: Swap (E, B)of position (2, 3)
Step 6: Swap (W, E)of position (6, 11)
Step 7: Swap (E, B)of position (8, 9)
Step 8: Swap (W, E)of position (0, 5)
Step 9: Swap (B, E)of position (10, 3)
Step 10: Swap (E, W)of position (6, 1)

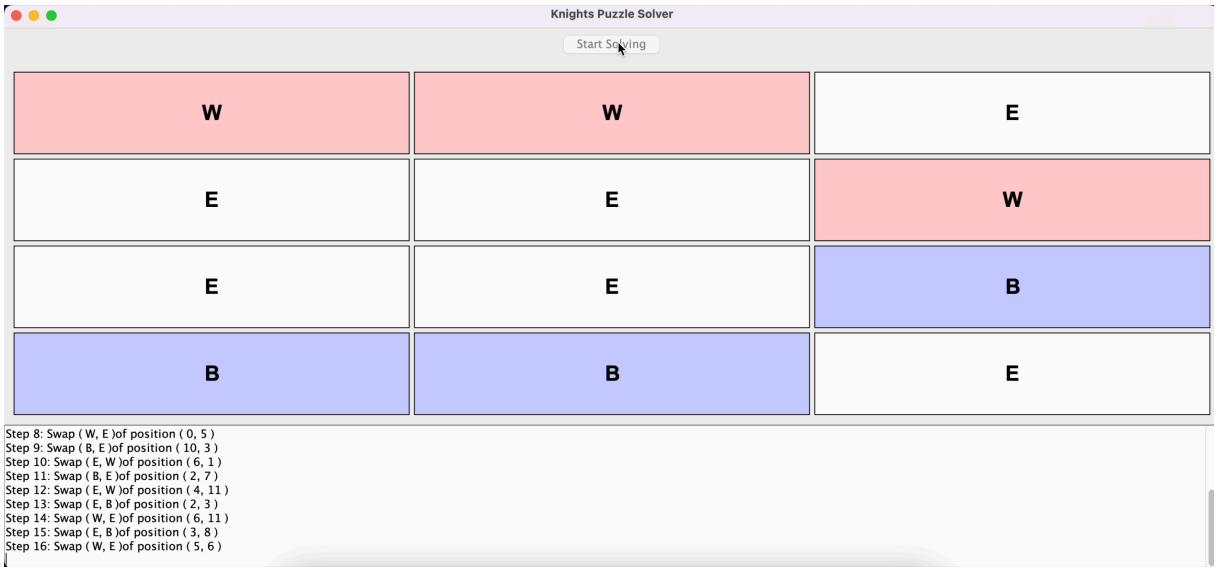

Knights Puzzle Solver
Start Solving
W W B
E W E
E E E
B B E
Step 3: Swap (E, B)of position (0, 7)
Step 4: Swap (W, E)of position (4, 9)
Step 5: Swap (E, B)of position (2, 3)
Step 6: Swap (W, E)of position (6, 11)
Step 7: Swap (E, B)of position (8, 9)
Step 8: Swap (W, E)of position (0, 5)
Step 9: Swap (B, E)of position (10, 3)
Step 10: Swap (E, W)of position (6, 1)
Step 11: Swap (B, E)of position (2, 7)

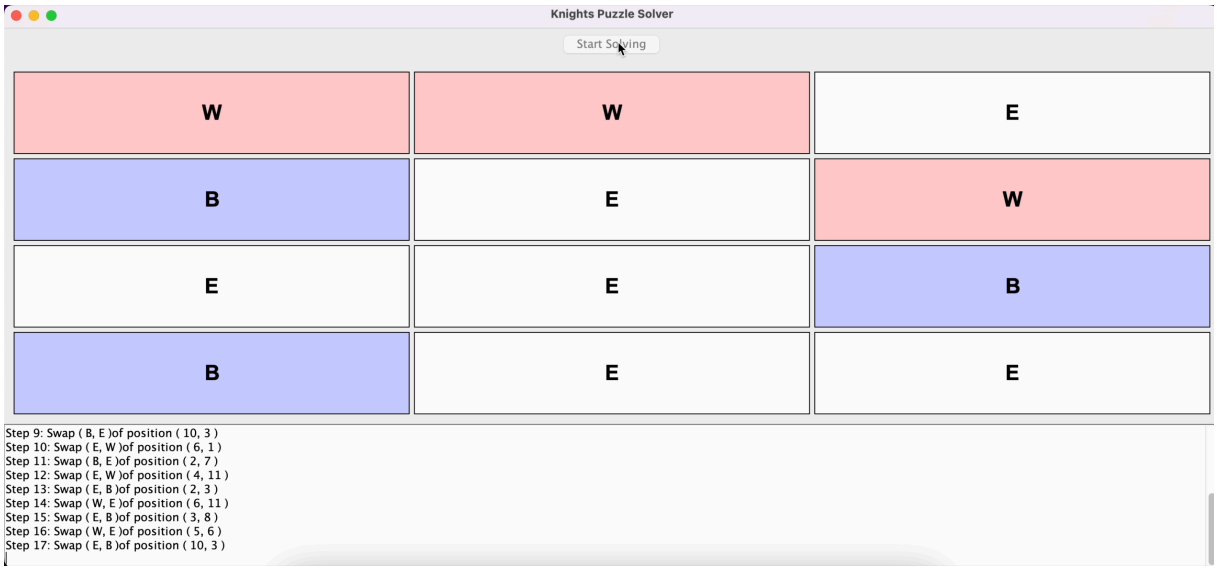

Knights Puzzle Solver
Start Solving
W W B
E E E
E E E
B B W
Step 4: Swap (W, E)of position (4, 9)
Step 5: Swap (E, B)of position (2, 3)
Step 6: Swap (W, E)of position (6, 11)
Step 7: Swap (E, B)of position (8, 9)
Step 8: Swap (W, E)of position (0, 5)
Step 9: Swap (B, E)of position (10, 3)
Step 10: Swap (E, W)of position (6, 1)
Step 11: Swap (B, E)of position (2, 7)
Step 12: Swap (E, W)of position (4, 11)

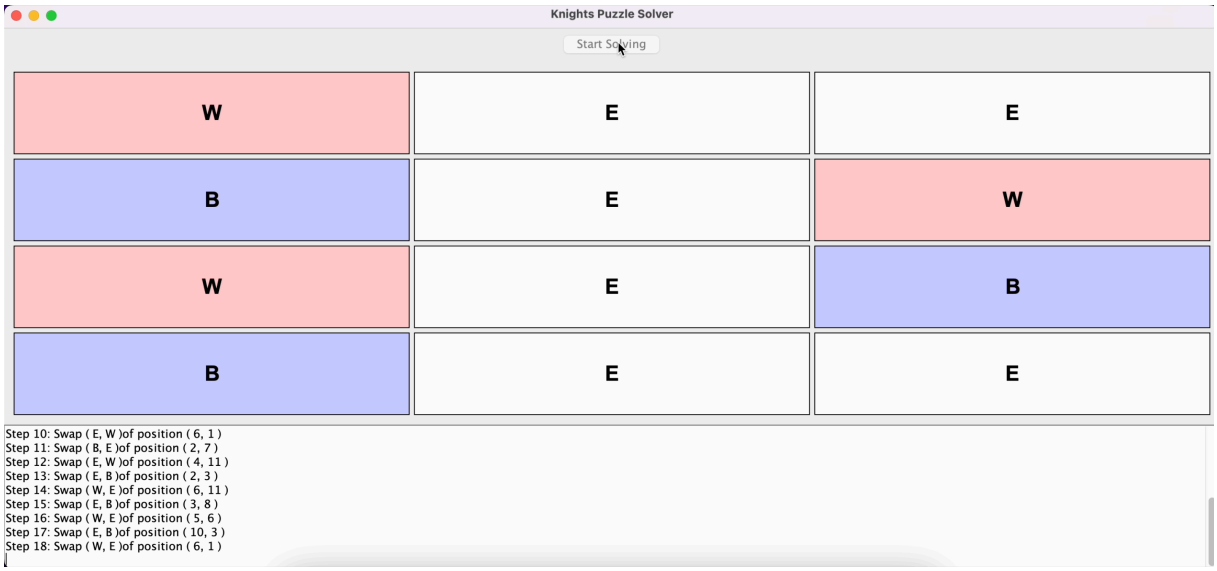

Knights Puzzle Solver
Start Solving
W W E
B E E
E E E
B B W
Step 5: Swap (E, B)of position (2, 3)
Step 6: Swap (W, E)of position (6, 11)
Step 7: Swap (E, B)of position (8, 9)
Step 8: Swap (W, E)of position (0, 5)
Step 9: Swap (B, E)of position (10, 3)
Step 10: Swap (E, W)of position (6, 1)
Step 11: Swap (B, E)of position (2, 7)
Step 12: Swap (E, W)of position (4, 11)
Step 13: Swap (E, B)of position (2, 3)

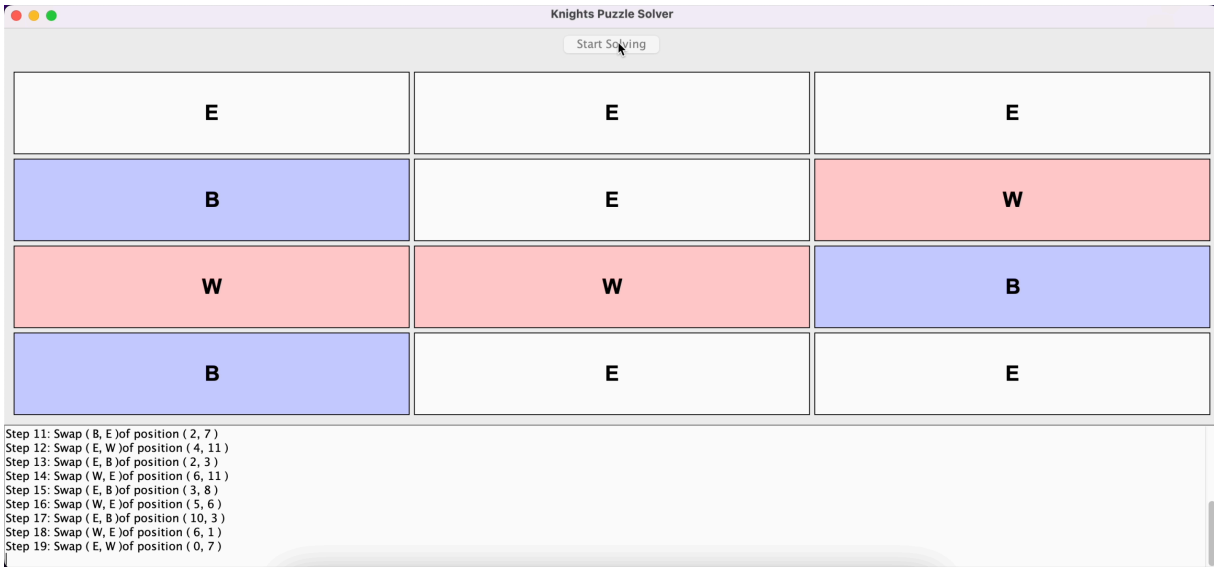

Knights Puzzle Solver
Start Solving
W W E
B E E
W E E
B B E
Step 6: Swap (W, E)of position (6, 11)
Step 7: Swap (E, B)of position (8, 9)
Step 8: Swap (W, E)of position (0, 5)
Step 9: Swap (B, E)of position (10, 3)
Step 10: Swap (E, W)of position (6, 1)
Step 11: Swap (B, E)of position (2, 7)
Step 12: Swap (E, W)of position (4, 11)
Step 13: Swap (E, B)of position (2, 3)
Step 14: Swap (W, E)of position (6, 11)

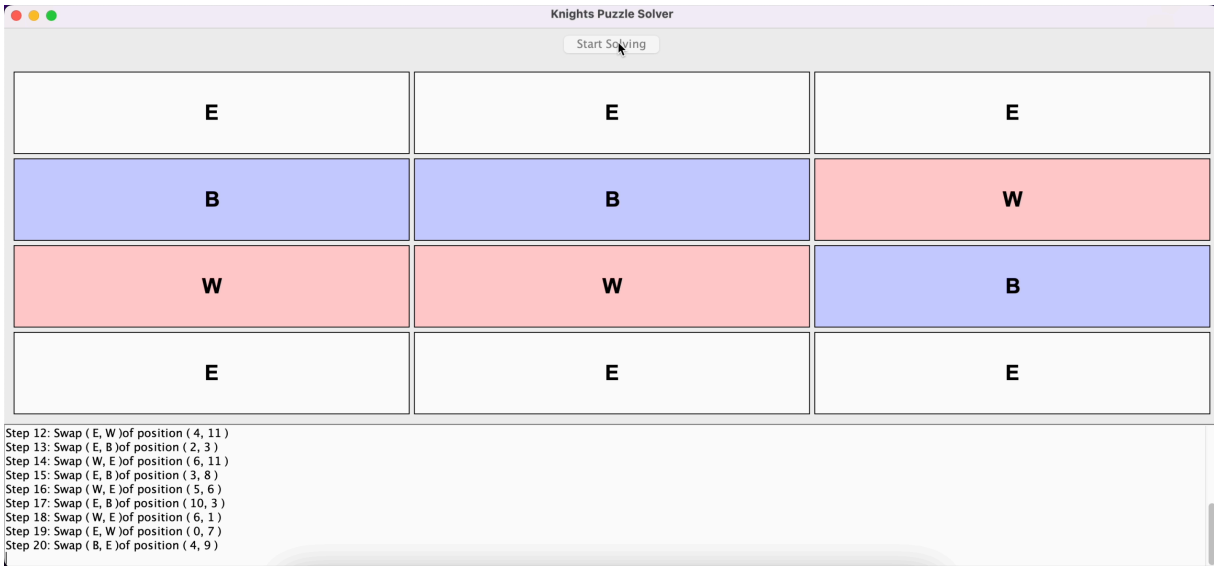

Knights Puzzle Solver
Start Solving
W W E
E E E
W E B
B B E
Step 7: Swap (E, B)of position (8, 9)
Step 8: Swap (W, E)of position (0, 5)
Step 9: Swap (B, E)of position (10, 3)
Step 10: Swap (E, W)of position (6, 1)
Step 11: Swap (B, E)of position (2, 7)
Step 12: Swap (E, W)of position (4, 11)
Step 13: Swap (E, B)of position (2, 3)
Step 14: Swap (W, E)of position (6, 11)
Step 15: Swap (E, B)of position (3, 8)

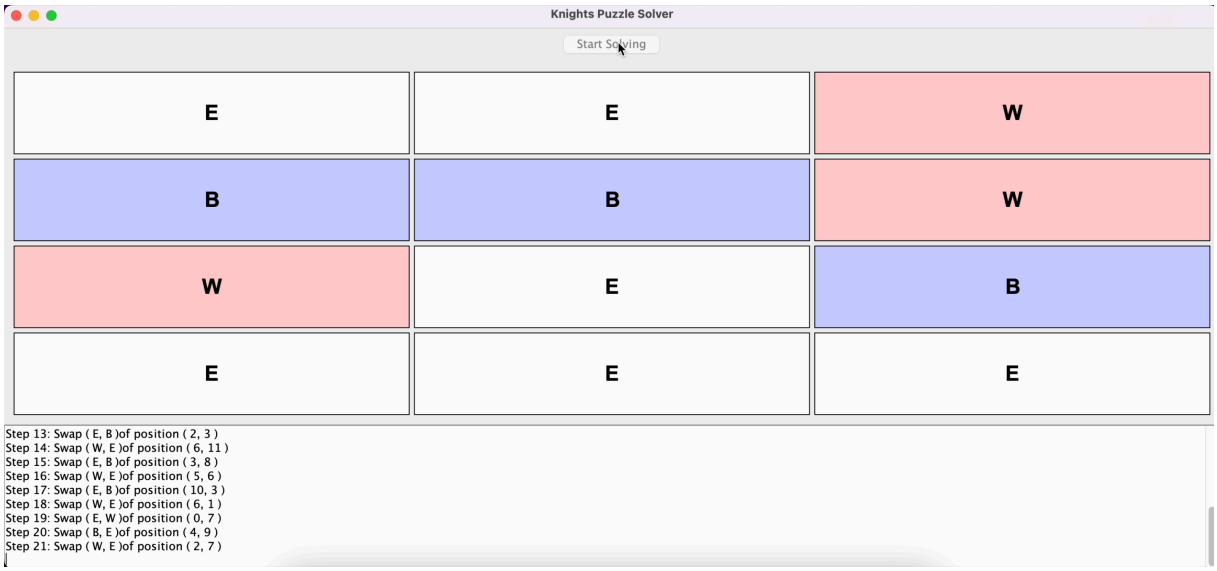

Knights Puzzle Solver
Start Solving
W W E
E E W
E E B
B B E
Step 8: Swap (W, E)of position (0, 5)
Step 9: Swap (B, E)of position (10, 3)
Step 10: Swap (E, W)of position (6, 1)
Step 11: Swap (B, E)of position (2, 7)
Step 12: Swap (E, W)of position (4, 11)
Step 13: Swap (E, B)of position (2, 3)
Step 14: Swap (W, E)of position (6, 11)
Step 15: Swap (E, B)of position (3, 8)
Step 16: Swap (W, E)of position (5, 6)

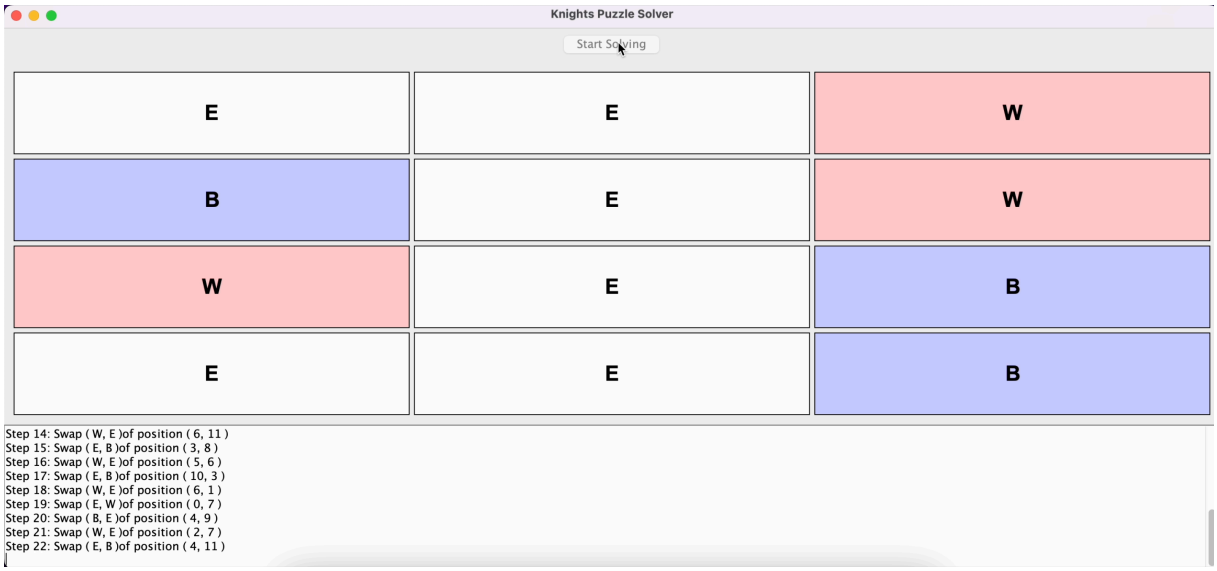

Knights Puzzle Solver
Start Solving
W W E
B E W
E E B
B E E
Step 9: Swap (B, E)of position (10, 3)
Step 10: Swap (E, W)of position (6, 1)
Step 11: Swap (B, E)of position (2, 7)
Step 12: Swap (E, W)of position (4, 11)
Step 13: Swap (E, B)of position (2, 3)
Step 14: Swap (W, E)of position (6, 11)
Step 15: Swap (E, B)of position (3, 8)
Step 16: Swap (W, E)of position (5, 6)
Step 17: Swap (E, B)of position (10, 3)

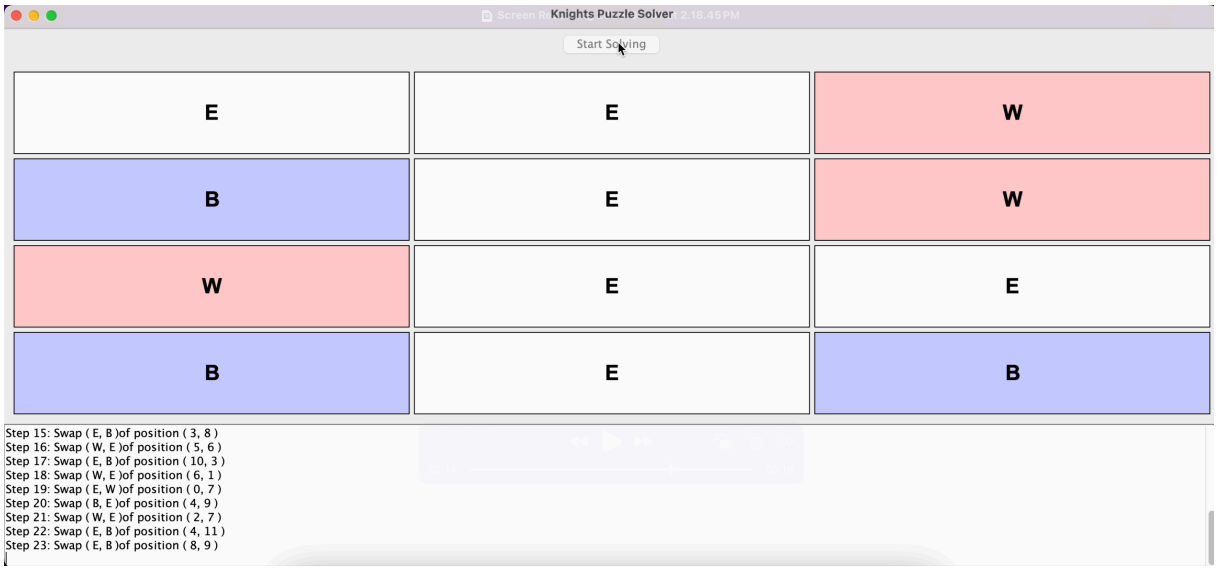

Knights Puzzle Solver
Start Solving
W E E
B E W
W E B
B E E
Step 10: Swap (E, W)of position (6, 1)
Step 11: Swap (B, E)of position (2, 7)
Step 12: Swap (E, W)of position (4, 11)
Step 13: Swap (E, B)of position (2, 3)
Step 14: Swap (W, E)of position (6, 11)
Step 15: Swap (E, B)of position (3, 8)
Step 16: Swap (W, E)of position (5, 6)
Step 17: Swap (E, B)of position (10, 3)
Step 18: Swap (W, E)of position (6, 1)

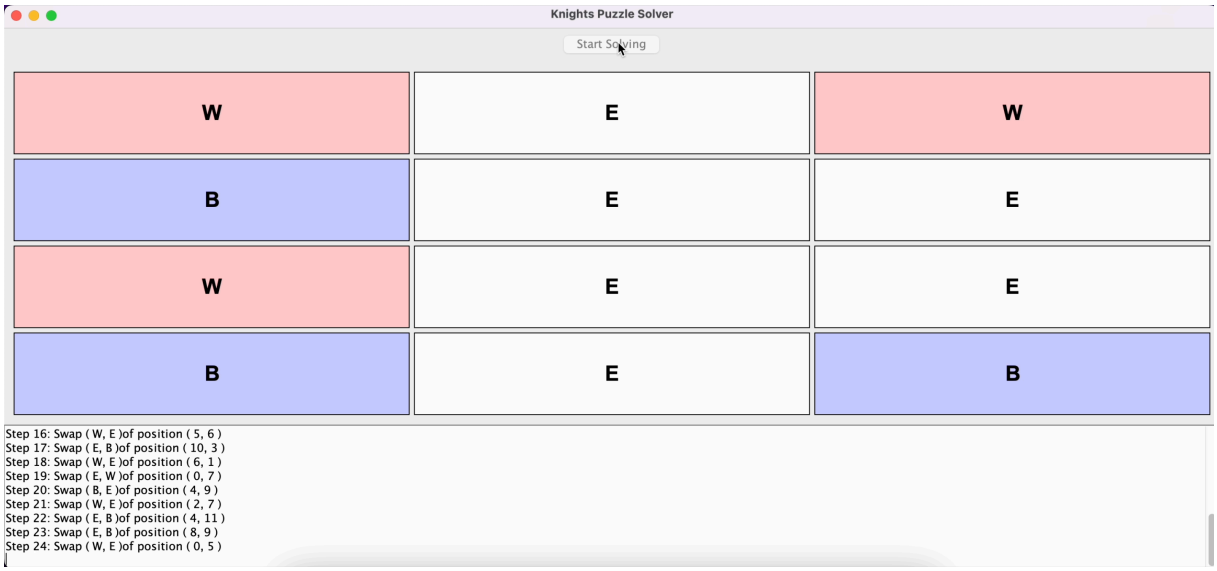

Knights Puzzle Solver
Start Solving
E E E
B E W
W W B
B E E
Step 11: Swap (B, E)of position (2, 7)
Step 12: Swap (E, W)of position (4, 11)
Step 13: Swap (E, B)of position (2, 3)
Step 14: Swap (W, E)of position (6, 11)
Step 15: Swap (E, B)of position (3, 8)
Step 16: Swap (W, E)of position (5, 6)
Step 17: Swap (E, B)of position (10, 3)
Step 18: Swap (W, E)of position (6, 1)
Step 19: Swap (E, W)of position (0, 7)

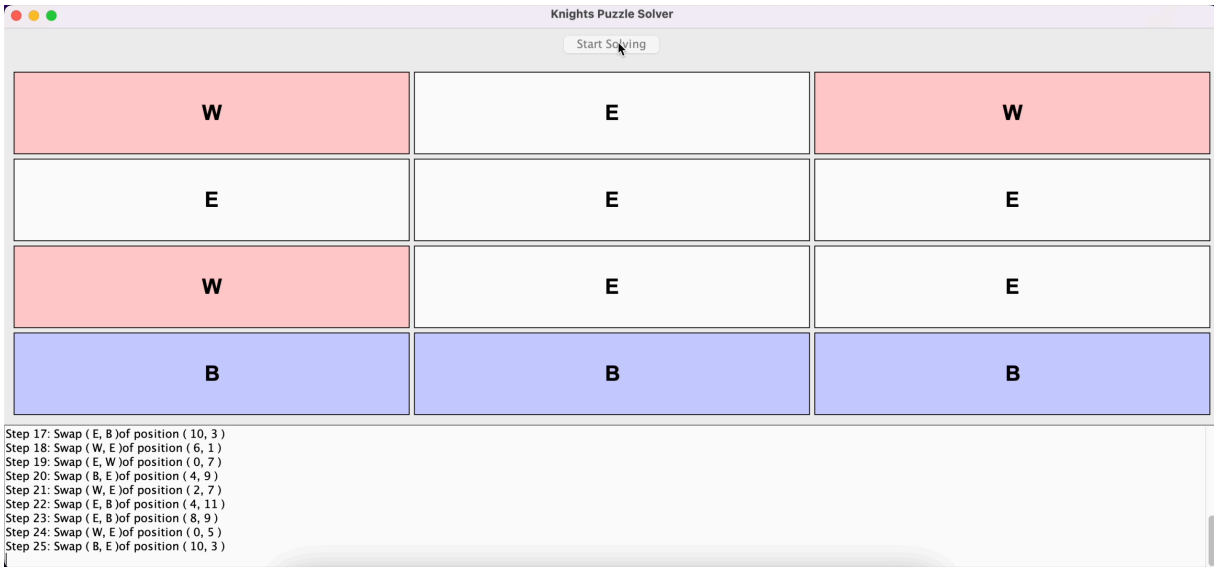

Knights Puzzle Solver
Start Solving
E E E
B B W
W W B
E E E
Step 12: Swap (E, W)of position (4, 11)
Step 13: Swap (E, B)of position (2, 3)
Step 14: Swap (W, E)of position (6, 11)
Step 15: Swap (E, B)of position (3, 8)
Step 16: Swap (W, E)of position (5, 6)
Step 17: Swap (E, B)of position (10, 3)
Step 18: Swap (W, E)of position (6, 1)
Step 19: Swap (E, W)of position (0, 7)
Step 20: Swap (B, E)of position (4, 9)

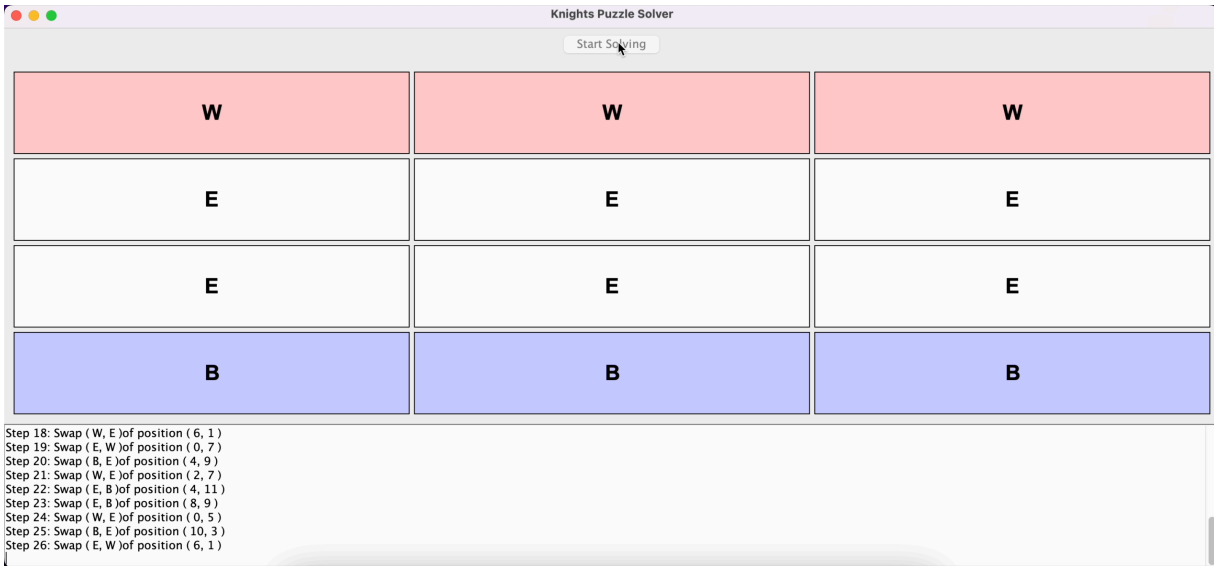

Knights Puzzle Solver
Start Solving
E E W
B B W
W E B
E E E
Step 13: Swap (E, B)of position (2, 3)
Step 14: Swap (W, E)of position (6, 11)
Step 15: Swap (E, B)of position (3, 8)
Step 16: Swap (W, E)of position (5, 6)
Step 17: Swap (E, B)of position (10, 3)
Step 18: Swap (W, E)of position (6, 1)
Step 19: Swap (E, W)of position (0, 7)
Step 20: Swap (B, E)of position (4, 9)
Step 21: Swap (W, E)of position (2, 7)


Knights Puzzle Solver
Start Solving
E E W
B E W
W E B
E E B
Step 14: Swap (W, E)of position (6, 11)
Step 15: Swap (E, B)of position (3, 8)
Step 16: Swap (W, E)of position (5, 6)
Step 17: Swap (E, B)of position (10, 3)
Step 18: Swap (W, E)of position (6, 1)
Step 19: Swap (E, W)of position (0, 7)
Step 20: Swap (B, E)of position (4, 9)
Step 21: Swap (W, E)of position (2, 7)
Step 22: Swap (E, B)of position (4, 11)


Knights Puzzle Solver
Start Solving
E E W
B E W
W E E
B E B
Step 15: Swap (E, B)of position (3, 8)
Step 16: Swap (W, E)of position (5, 6)
Step 17: Swap (E, B)of position (10, 3)
Step 18: Swap (W, E)of position (6, 1)
Step 19: Swap (E, W)of position (0, 7)
Step 20: Swap (B, E)of position (4, 9)
Step 21: Swap (W, E)of position (2, 7)
Step 22: Swap (E, B)of position (4, 11)
Step 23: Swap (E, B)of position (8, 9)


Knights Puzzle Solver
Start Solving
W E W
B E E
W E E
B E B
Step 16: Swap (W, E)of position (5, 6)
Step 17: Swap (E, B)of position (10, 3)
Step 18: Swap (W, E)of position (6, 1)
Step 19: Swap (E, W)of position (0, 7)
Step 20: Swap (B, E)of position (4, 9)
Step 21: Swap (W, E)of position (2, 7)
Step 22: Swap (E, B)of position (4, 11)
Step 23: Swap (E, B)of position (8, 9)
Step 24: Swap (W, E)of position (0, 5)


Knights Puzzle Solver
Start Solving
W E W
E E E
W E E
B B B
Step 17: Swap (E, B)of position (10, 3)
Step 18: Swap (W, E)of position (6, 1)
Step 19: Swap (E, W)of position (0, 7)
Step 20: Swap (B, E)of position (4, 9)
Step 21: Swap (W, E)of position (2, 7)
Step 22: Swap (E, B)of position (4, 11)
Step 23: Swap (E, B)of position (8, 9)
Step 24: Swap (W, E)of position (0, 5)
Step 25: Swap (B, E)of position (10, 3)


Knights Puzzle Solver
Start Solving
W W W
E E E
E E E
B B B
Step 18: Swap (W, E)of position (6, 1)
Step 19: Swap (E, W)of position (0, 7)
Step 20: Swap (B, E)of position (4, 9)
Step 21: Swap (W, E)of position (2, 7)
Step 22: Swap (E, B)of position (4, 11)
Step 23: Swap (E, B)of position (8, 9)
Step 24: Swap (W, E)of position (0, 5)
Step 25: Swap (B, E)of position (10, 3)
Step 26: Swap (E, W)of position (6, 1)

5) Task (5)

5.1) Description

Description

Given n > 1 hiding spots arranged in a line. A target hides at one spot and moves to an adjacent one after each shot. The shooter never sees the target but can shoot any spot at each turn.

- Goal: Guarantee hitting the target using a Divide and Conquer strategy (not Decrease and Conquer).
- Constraints:
  - The target moves to a neighboring spot (left or right) after each missed shot
  - Only one target exists
  - Shooter chooses one spot per turn
  - Shooter never knows the target's location
- Key properties:
  - The target's position shifts one step each time
  - The movement pattern forces it to "bounce" within the range
  - The target cannot skip spots
  - A well-designed pattern of shots can "corner" or intercept the target
  - The algorithm should work by:
  - Divide the range [1, n] into two halves
  - Alternate shooting between opposite ends of each half
  - Guarantee that in at most 2n - 2 moves, the target gets hit

Divide and Conquer Strategy:

- Divide the line into two halves: [1..m], [m+1..n]
- First, focus on one half (e.g., left half)
- Shoot across positions in a zigzag pattern from left to right
- After completing one half, move to the other half and repeat

- Because the target shifts only one position, the interleaved shot pattern guarantees interception

5.2) Problem Description

- Objective
  - The goal is to shoot at a hiding spot, adjusting the target after each unsuccessful attempt, until the target is found.
- Input
  - n: The number of hiding spots (e.g., 8).
  - target: The target spot to find (input from the user).
- Output
  - A sequence of shots at different hiding spots.
  - Print statements showing the attempts and results.
- Constraints
  - The number of hiding spots is ≥ 1.
  - The search will attempt to narrow down to the target by modifying the target location after two consecutive shots.

5.3) Detailed assumptions

- Target Configuration
  - The puzzle involves a set of hiding spots numbered from 0 to n-1
  - The target is a random number between 0 and n-1
- Shooting Rules
  - A “shot” is fired at a spot by determining a mid-point index between a given range.
  - The shot may result in a successful hit or not, and the next attempt is modified based on previous outcomes.
  - A successful shot at a hiding spot reveals the target’s location.
- Algorithm Choice
  - The code follows a binary search-like approach where the range of hiding spots is split recursively.

- The next target to shoot at is updated based on the parity of a counter, ensuring systematic exploration of spots.

- Memory Model
  - The spots are stored in a vector, representing each hiding spot's index.
  - A counter is used to keep track of the number of consecutive attempts.

5.4) Divide and Conquer Algorithm

5.4.1) Pseudocode

```text
Function shootAtTarget(spots, targetRef, start, end,
shotCounterRef):
    If shotCounterRef == -1:
        Return True  // Target already shot
    Else if start == end:
        Return False  // Base case: no valid range left to search

    mid = (start + end) / 2
    Print "Shoot at hiding spot", mid

    If spots[mid] == targetRef:
        Print "Shooter shot target at spot:", targetRef
        shotCounterRef = -1  // Mark as hit
        Return True

    shotCounterRef = shotCounterRef + 1

    // Every second shot, target may move
    If shotCounterRef is even:
        Randomly choose True or False:
            If True:
                targetRef = (targetRef + 1) mod size of spots
            Else:
                targetRef = (targetRef - 1) mod size of spots
        Print "Target moved to", targetRef

    // Recursively search in both halves
    Return shootAtTarget(spots, targetRef, start, mid, shotCounterRef)
           OR
           shootAtTarget(spots, targetRef, mid + 1, end, shotCounterRef)

Main:
    Seed random number generator
```

```
    Prompt user for number of hiding spots → n
    Prompt user for initial target spot → target

    Initialize spots as a list from 0 to n-1
    Initialize shotCounter = 0

    Print separator line

    While shootAtTarget(spots, target, 0, n - 1, shotCounter) returns
    False:
        Print "Failed Try again"
        Print current target position
        Print separator line

    End Program
```

Listing 5.1: Shooter Problem - Divide and Conquer Algorithm Pseudocode

5.4.2) Code

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

using namespace std;

// Function to perform shots on hiding spots and guarantee hitting the
target
bool shootAtTarget(vector<int>& spot, int* target, int start, int end,
int* i)
{
    if (*i == -1)
    {
        return true;
    }
    else if (start == end)
    {
        return false;
    }

    int mid = (start + end) / 2;
    cout << "Shoot at hiding spot " << mid << endl;

```

```
23
24      if (spot[mid] == *target)
25      {
26          cout << "Shooter shot target at spot: " << *target << endl;
27          *i = -1;
28          return true;
29      }
30      *i = *i + 1;
31      if(*i % 2 == 0)
32      {
33          bool flag;
34          flag = rand() % 2;
35          if (flag)
36          {
37              cout << "Target moved: " << *target <<" -> " << (*target +
                1) % spot.size() << endl;
38              *target = (*target + 1) % spot.size();
39          }
40          else
41          {
42              cout << "Target moved: " << *target <<" -> " << (*target -
                1) % spot.size() << endl;
43              *target = (*target - 1) % spot.size();
44          }
45
46      }
47
48      return shootAtTarget(spot, target, start, mid , i) ||
        shootAtTarget(spot, target, mid + 1, end, i);
49
50
51  }
52
53  int main()
54  {
55      srand(time(0)); // seed random generator
56      int n;
57      int target;
58      int consecutive_shot = 0;
59      cout << "Enter the number of hiding spots: ";
60      cin >> n;
61
62      cout << "Enter Target spot: ";
```

```
    cin >> target;

    vector<int> spots(n);

    for (int i = 0; i < n; ++i)
    {
        spots[i] = i;
    }




    cout<<"═══════════════════════════════════════════════════════════════"

    // Call the shoot function
    while(!shootAtTarget(spots, &target, 0, n - 1, &consecutive_shot))
    {
        cout<<"Failed Try again"<<endl;
        cout<<"Target: "<<target<<endl;

        cout<<"═══════════════════════════════════════════════════════════════

    }

    return 0;
}
```

Listing 5.2: Shooter Problem - Divide and Conquer Algorithm C++ Code

5.4.3) Test Cases

Input:

- n: 50 spots
- target: 13

```
Output
Shoot at hiding spot 24
Shoot at hiding spot 12
Target moved: 13 → 14
Shoot at hiding spot 6
Shoot at hiding spot 3
Target moved: 14 → 15
Shoot at hiding spot 1
Shoot at hiding spot 0
Target moved: 15 → 16
```

```
Shoot at hiding spot 2
Shoot at hiding spot 5
Target moved: 16 → 17
Shoot at hiding spot 4
Shoot at hiding spot 9
Target moved: 17 → 18
Shoot at hiding spot 8
Shoot at hiding spot 7
Target moved: 18 → 19
Shoot at hiding spot 11
Shoot at hiding spot 10
Target moved: 19 → 20
Shoot at hiding spot 18
Shoot at hiding spot 15
Target moved: 20 → 21
Shoot at hiding spot 14
Shoot at hiding spot 13
Target moved: 21 → 22
Shoot at hiding spot 17
Shoot at hiding spot 16
Target moved: 22 → 23
Shoot at hiding spot 21
Shoot at hiding spot 20
Target moved: 23 → 24
Shoot at hiding spot 19
Shoot at hiding spot 23
Target moved: 24 → 25
Shoot at hiding spot 22
Shoot at hiding spot 37
Target moved: 25 → 26
Shoot at hiding spot 31
Shoot at hiding spot 28
Target moved: 26 → 27
Shoot at hiding spot 26
Shoot at hiding spot 25
Target moved: 27 → 28
Shoot at hiding spot 27
Shoot at hiding spot 30
Target moved: 28 → 29
```

```
Shoot at hiding spot 29

Shooter shot target at spot: 29
```

Listing 5.3: Shooter Problem - Divide and Conquer Algorithm Test Case Output

5.4.4) Complexity analysis

- Time Complexity
  - The recursive function shootAtTarget splits the search range repeatedly, similar to a binary search.
  - However, the modification of the target after each unsuccessful attempt adds a layer of complexity.
  - total time complexity is O(n).
- Space Complexity
  - The spots vector uses O(n) space to store all possible hiding spots.
  - The recursion depth is proportional to the size of the input, leading to a recursion stack of height O(logn) in the best case.
  - total space complexity is O(n).

5.5) Decrease and Conquer Algorithm

```text
function findTarget(low, high, target):
    if low == high:  // If there's only one position left, shoot there
        fire(low)
        if low == target:
            print("Target hit at position", low)
        else:
            print("Missed. Target is not at", low)
        return

    // Shoot at the start (low) and end (high) of the range
    fire(low)
    if low == target:
        print("Target hit at position", low)
        return
    else:
        print("Missed. Target is not at", low)

    fire(high)
    if high == target:
        print("Target hit at position", high)
```

```
        return
    else:
        print("Missed. Target is not at", high)

    // Recursively search the reduced range after eliminating the target's current position
    findTarget(low + 1, high - 1, target)  // Search between start + 1 and end - 1
```

Listing 5.4: Shooter Problem - Decrease and Conquer Algorithm Pseudocode

5.6) Comparison Between Divide and Conquer and Decrease and Conquer Strategies

Aspect	Divide and Conquer	Decrease and Conquer
Strategy	Divide search space and search each subspace.	Check hiding spots one by one.
Efficiency	More efficient in narrowing search space quickly.	Less efficient, checks every spot.
Time Complexity	$O(n)$	$O(n)$ in the worst case
Space Complexity	$O(\log(n))$, due to recursion	$O(1)$, minimal space
Best For	Small n (simple to implement)	Large n (faster than 3-peg)
Applicability	Better for larger search spaces and targets moving unpredictably.	Works well for small, simple problems.

Table 5.1: Shooter Problem - Algorithm Comparison

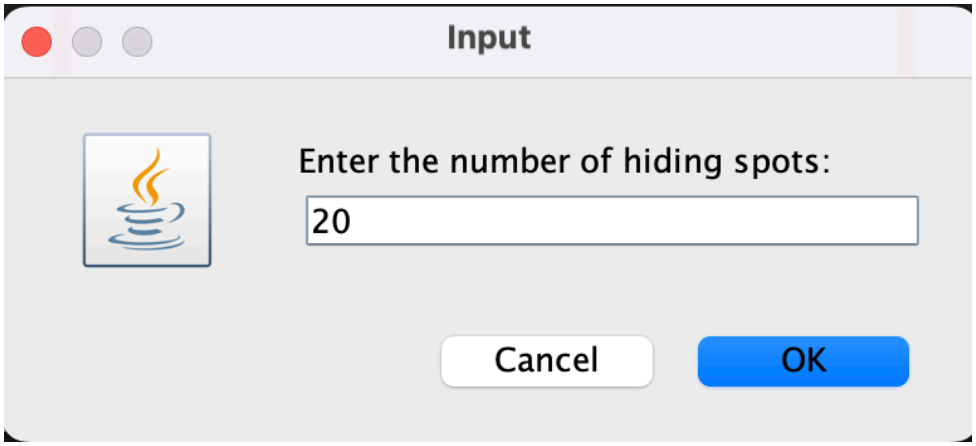
5.7) Conclusion

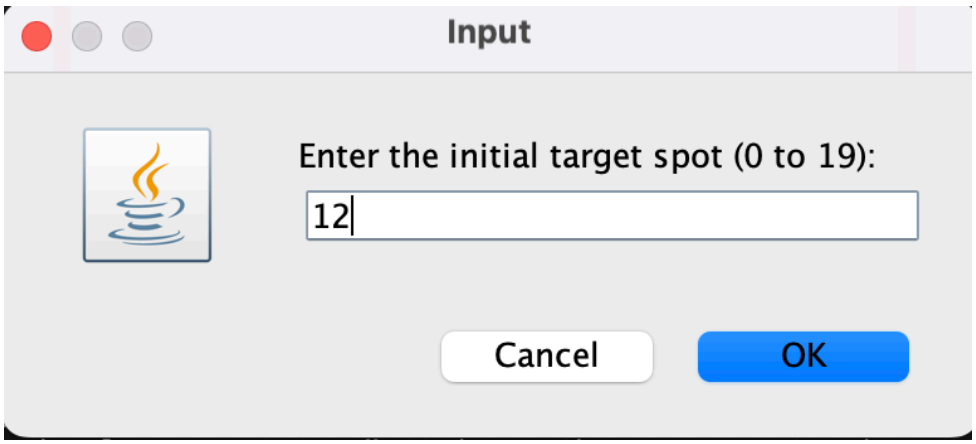
The divide and conquer approach for guaranteeing a hit on a moving target in a straight line provides a reliable method for solving the problem of hitting a target when the shooter has no visibility and can only shoot at adjacent spots. It ensures that no matter where the target is hiding, the shooter will hit it.

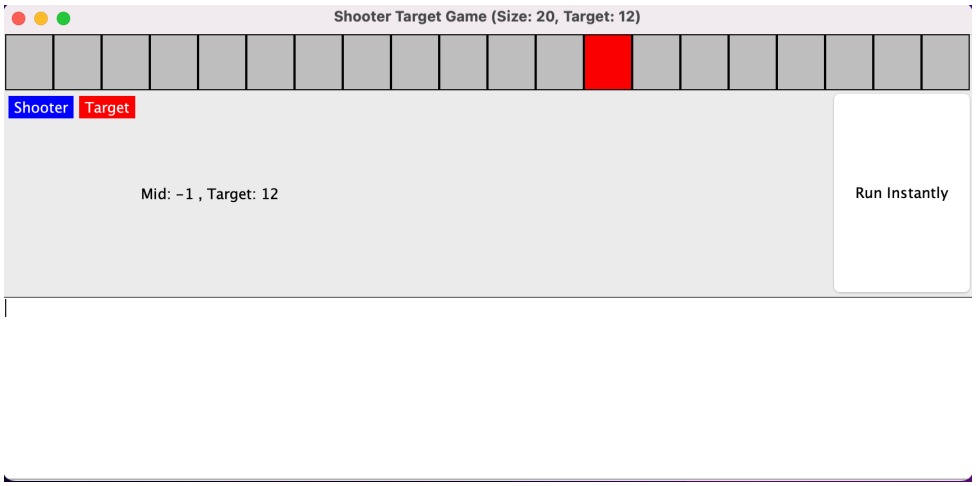
- Key points:
- Efficiency: The divide and conquer algorithm ensures the target will be hit in a logarithmic number of shots, which is much more efficient compared to random shooting or brute force approaches.
- Time Complexity: The algorithm's time complexity is logarithmic, O(n), because it narrows down the potential hiding spots by half in each step.
- Space Complexity: The space complexity is minimal, primarily requiring space for storing the positions and tracking the shots.

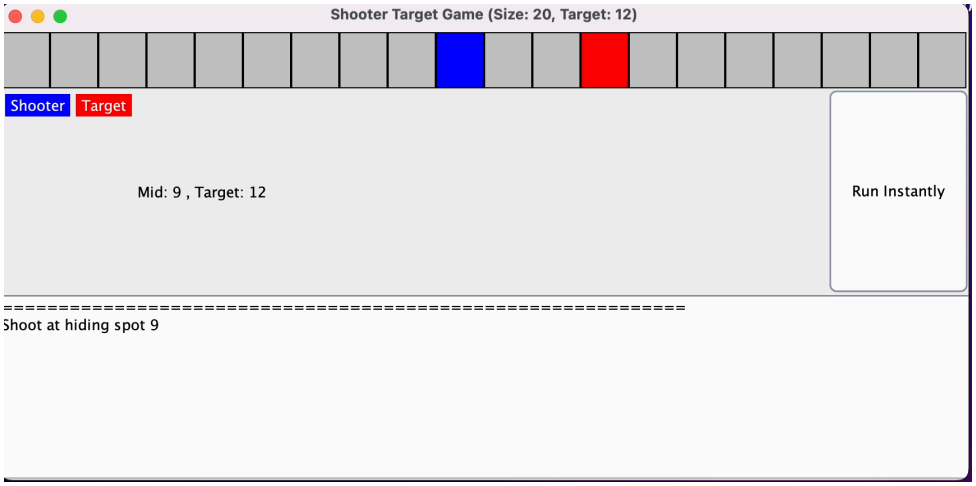
- Implementation Complexity: Though the implementation involves recursively dividing the search space, it is relatively straightforward and ensures guaranteed results in fewer steps.
- Effectiveness: This approach is highly effective, especially when dealing with larger numbers of hiding spots, as it dramatically reduces the number of shots required.
- Predictability: The algorithm is deterministic, ensuring the target will be hit no matter the initial position, with the sequence of shots predictable given the total number of hiding spots.clusion

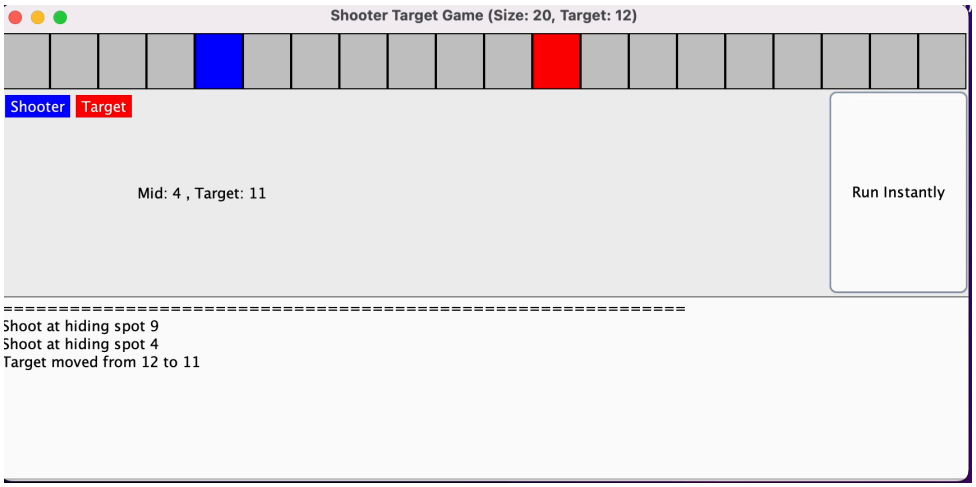
5.8) GUI Screenshots

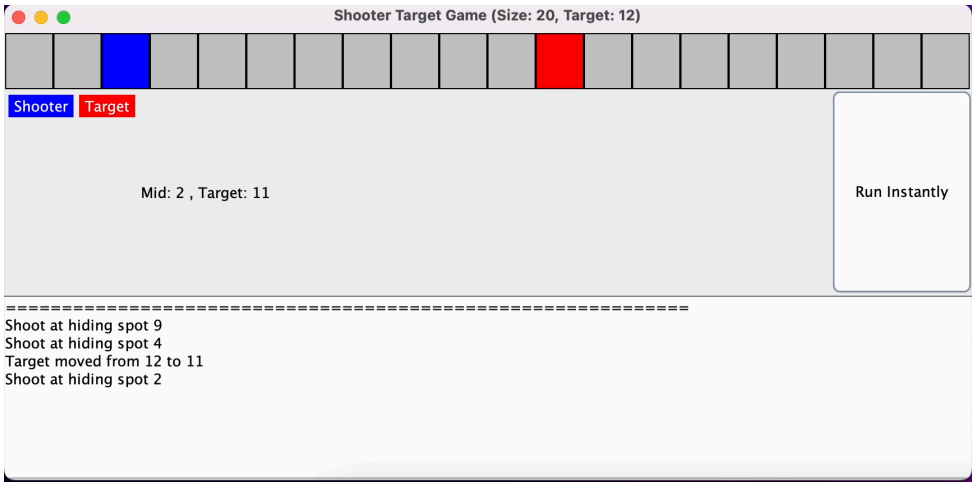


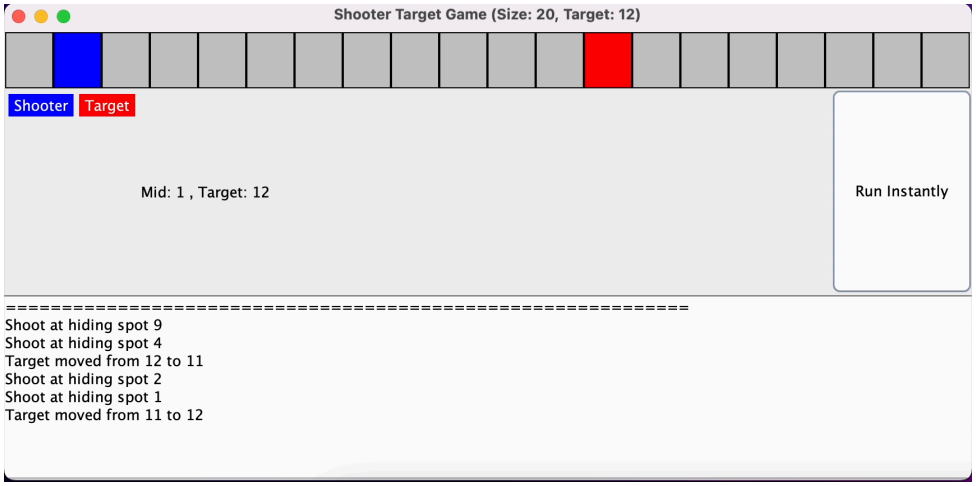


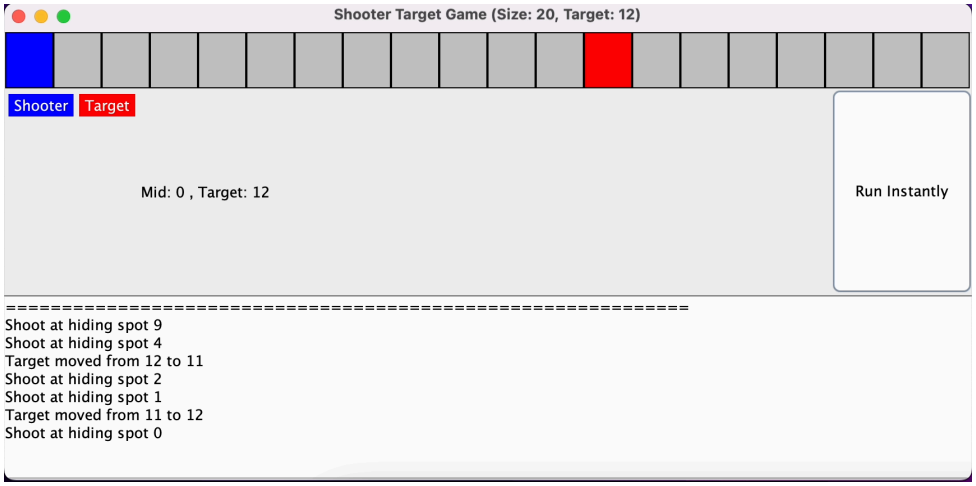

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: -1 , Target: 12
Run Instantly

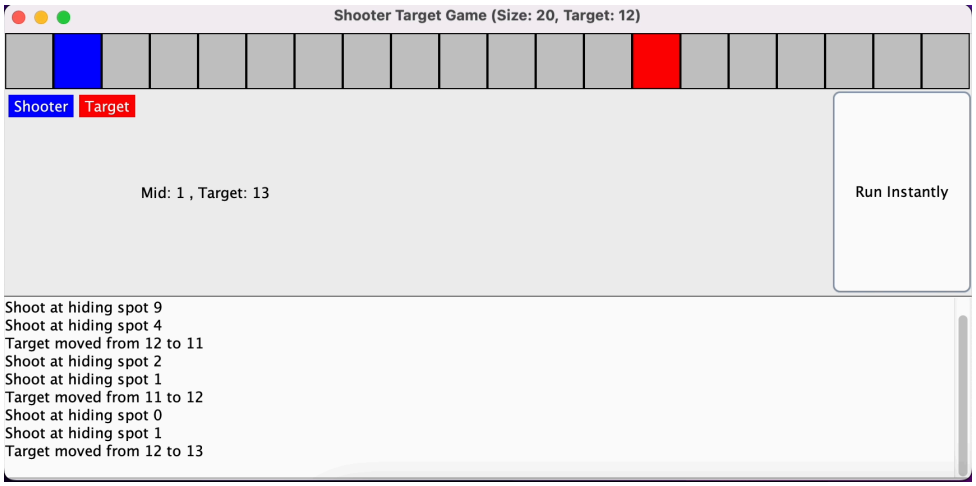

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 9 , Target: 12
Run Instantly
Shoot at hiding spot 9

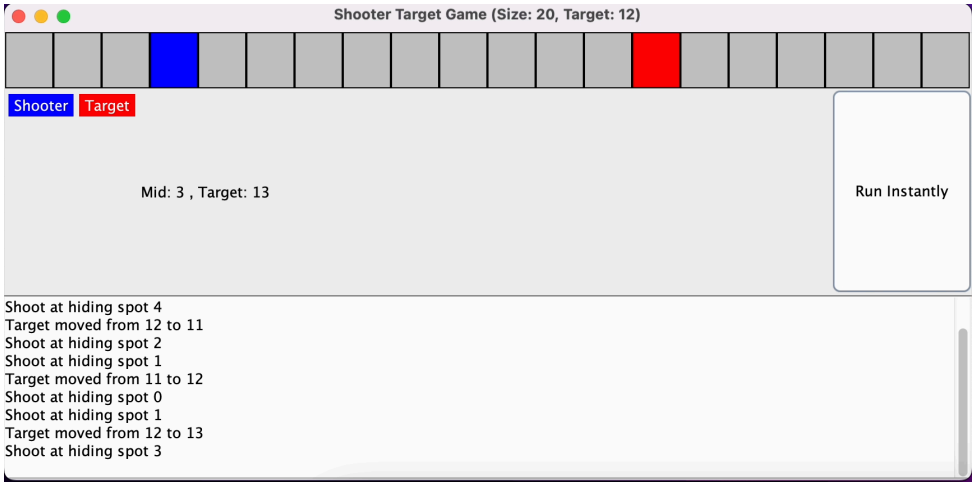

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 4 , Target: 11
Run Instantly
Shoot at hiding spot 9
Shoot at hiding spot 4
Target moved from 12 to 11

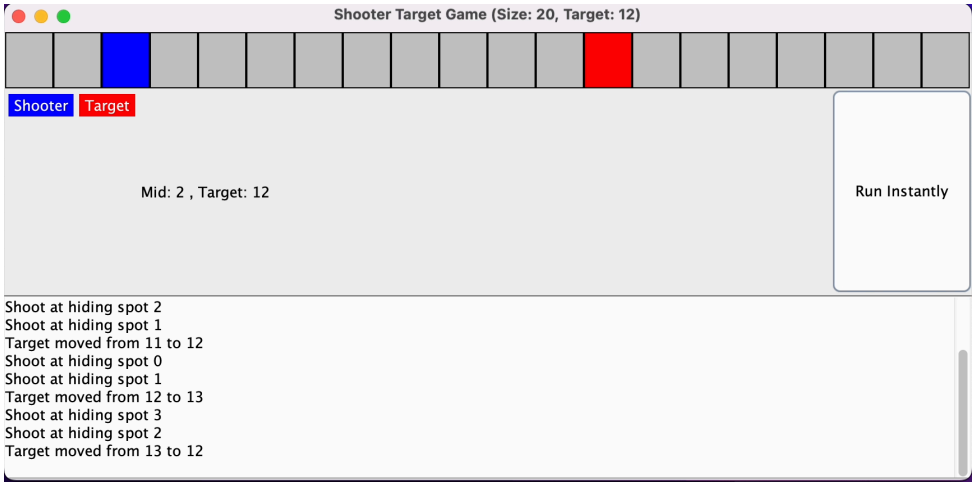

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 2 , Target: 11
Run Instantly
==========
Shoot at hiding spot 9
Shoot at hiding spot 4
Target moved from 12 to 11
Shoot at hiding spot 2

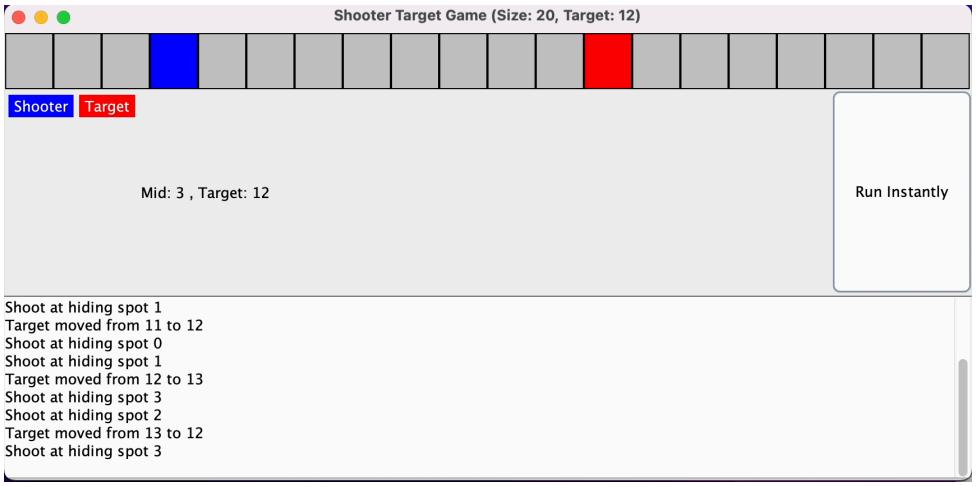

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 1 , Target: 12
Run Instantly
==========
Shoot at hiding spot 9
Shoot at hiding spot 4
Target moved from 12 to 11
Shoot at hiding spot 2
Shoot at hiding spot 1
Target moved from 11 to 12

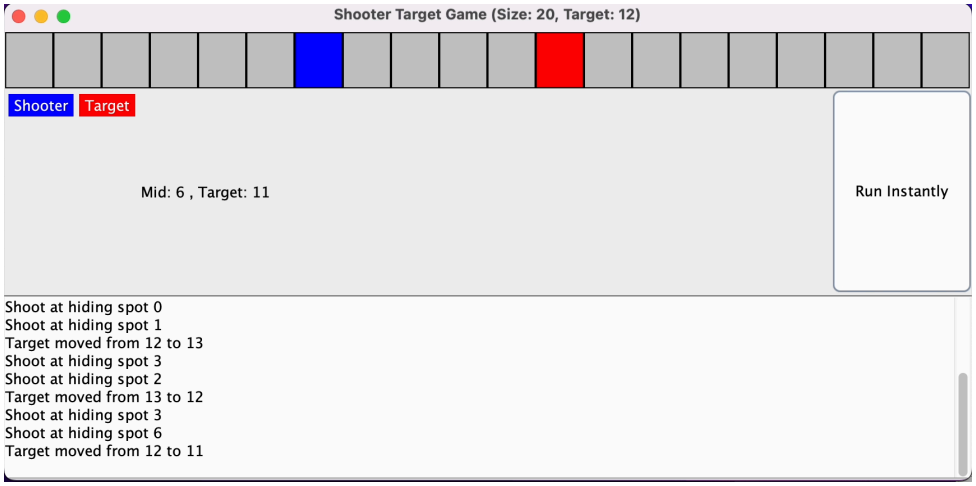

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 0 , Target: 12
Run Instantly
==========
Shoot at hiding spot 9
Shoot at hiding spot 4
Target moved from 12 to 11
Shoot at hiding spot 2
Shoot at hiding spot 1
Target moved from 11 to 12
Shoot at hiding spot 0

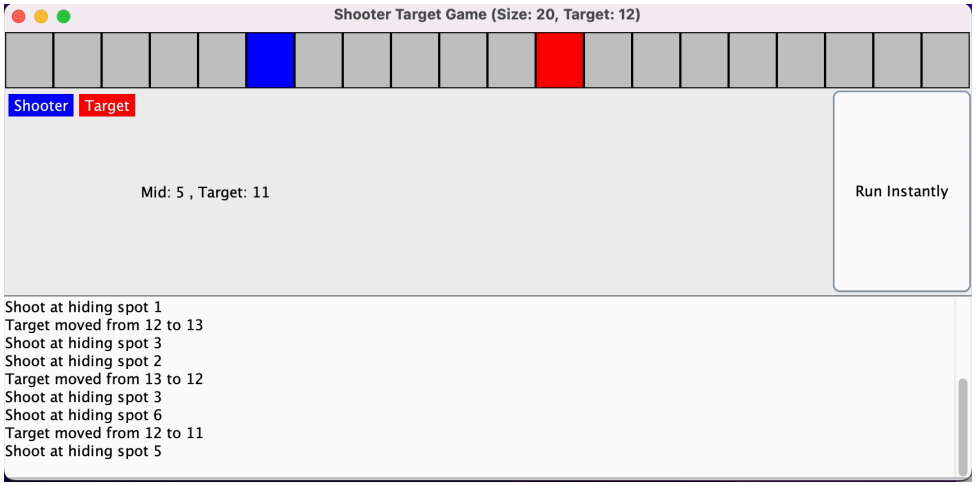

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 1 , Target: 13
Run Instantly
Shoot at hiding spot 9
Shoot at hiding spot 4
Target moved from 12 to 11
Shoot at hiding spot 2
Shoot at hiding spot 1
Target moved from 11 to 12
Shoot at hiding spot 0
Shoot at hiding spot 1
Target moved from 12 to 13

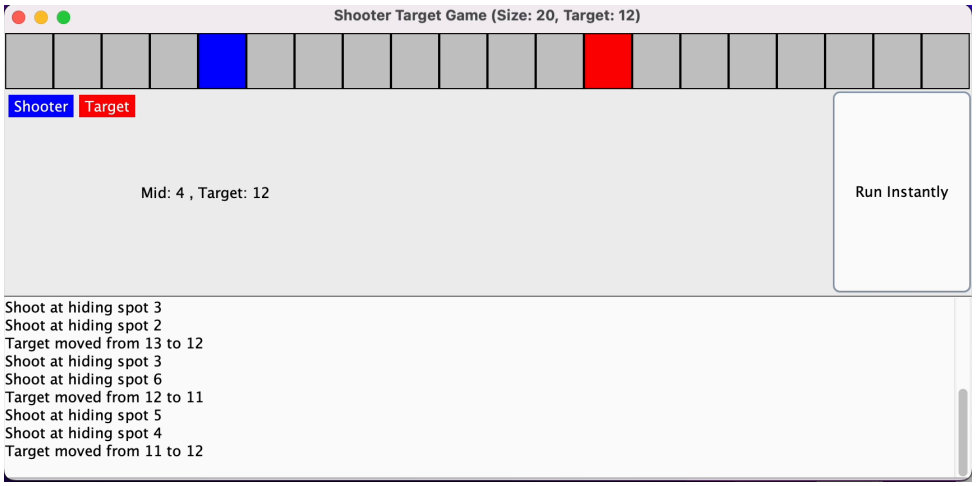

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 3 , Target: 13
Run Instantly
Shoot at hiding spot 4
Target moved from 12 to 11
Shoot at hiding spot 2
Shoot at hiding spot 1
Target moved from 11 to 12
Shoot at hiding spot 0
Shoot at hiding spot 1
Target moved from 12 to 13
Shoot at hiding spot 3

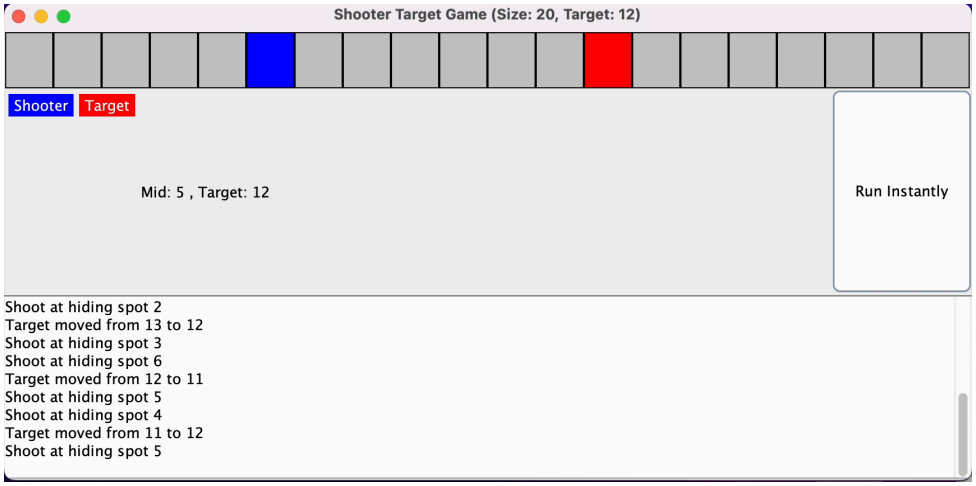

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 2 , Target: 12
Run Instantly
Shoot at hiding spot 2
Shoot at hiding spot 1
Target moved from 11 to 12
Shoot at hiding spot 0
Shoot at hiding spot 1
Target moved from 12 to 13
Shoot at hiding spot 3
Shoot at hiding spot 2
Target moved from 13 to 12

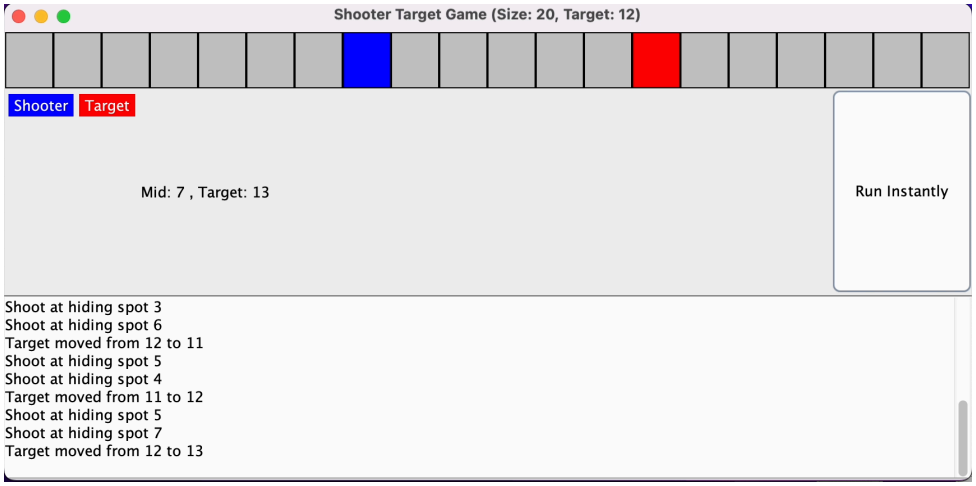

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 3 , Target: 12
Run Instantly
Shoot at hiding spot 1
Target moved from 11 to 12
Shoot at hiding spot 0
Shoot at hiding spot 1
Target moved from 12 to 13
Shoot at hiding spot 3
Shoot at hiding spot 2
Target moved from 13 to 12
Shoot at hiding spot 3

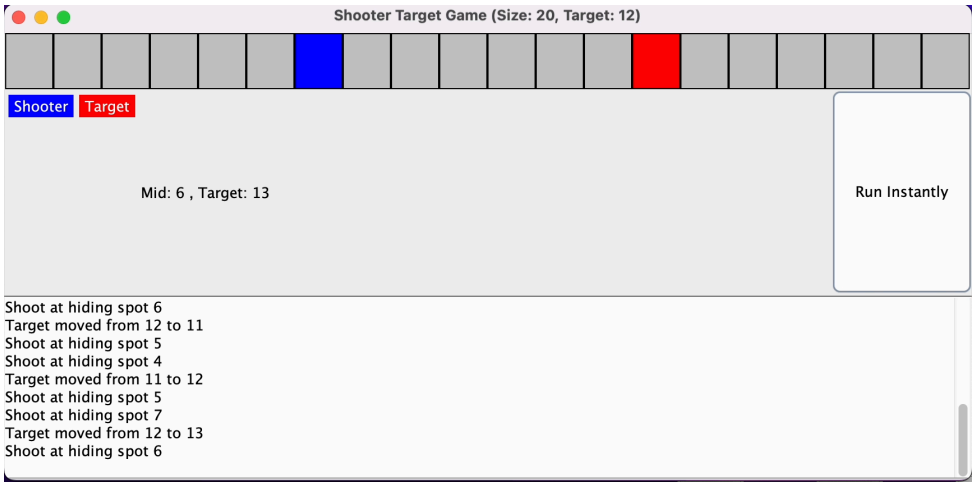

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 6 , Target: 11
Run Instantly
Shoot at hiding spot 0
Shoot at hiding spot 1
Target moved from 12 to 13
Shoot at hiding spot 3
Shoot at hiding spot 2
Target moved from 13 to 12
Shoot at hiding spot 3
Shoot at hiding spot 6
Target moved from 12 to 11

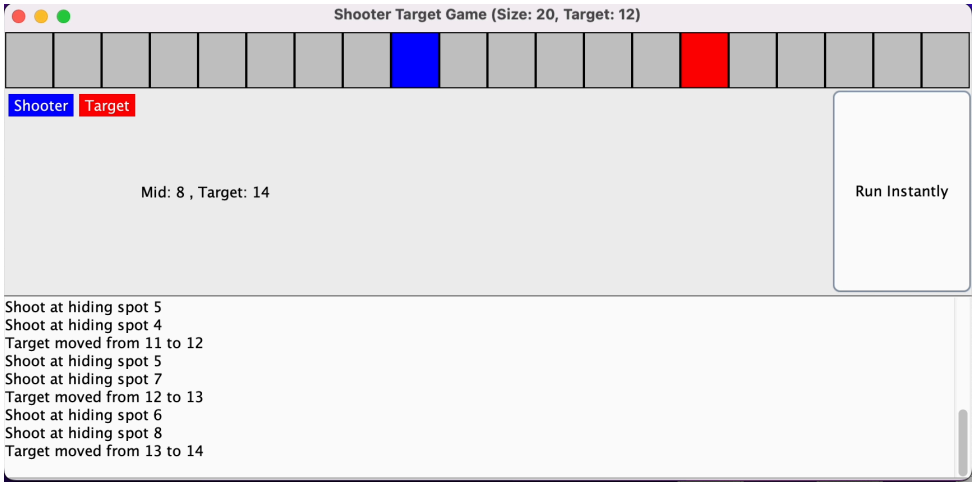

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 5 , Target: 11
Run Instantly
Shoot at hiding spot 1
Target moved from 12 to 13
Shoot at hiding spot 3
Shoot at hiding spot 2
Target moved from 13 to 12
Shoot at hiding spot 3
Shoot at hiding spot 6
Target moved from 12 to 11
Shoot at hiding spot 5

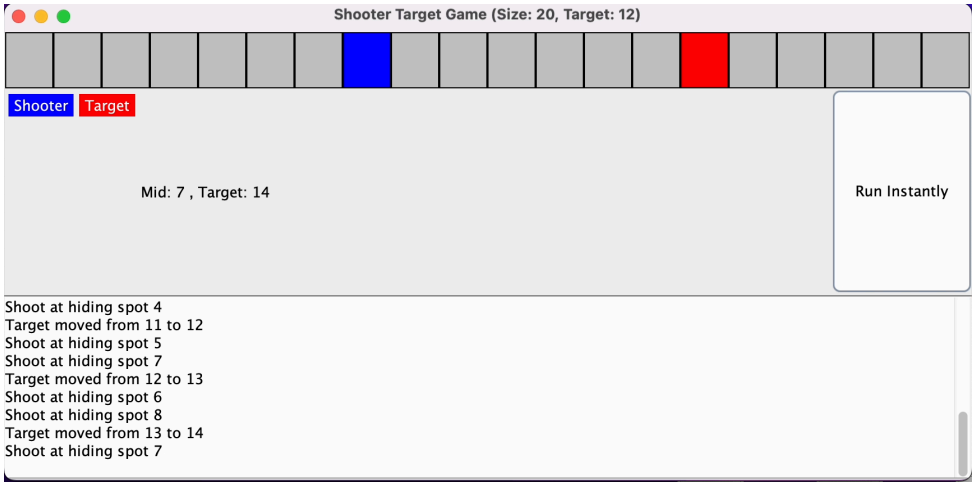

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 4 , Target: 12
Run Instantly
Shoot at hiding spot 3
Shoot at hiding spot 2
Target moved from 13 to 12
Shoot at hiding spot 3
Shoot at hiding spot 6
Target moved from 12 to 11
Shoot at hiding spot 5
Shoot at hiding spot 4
Target moved from 11 to 12

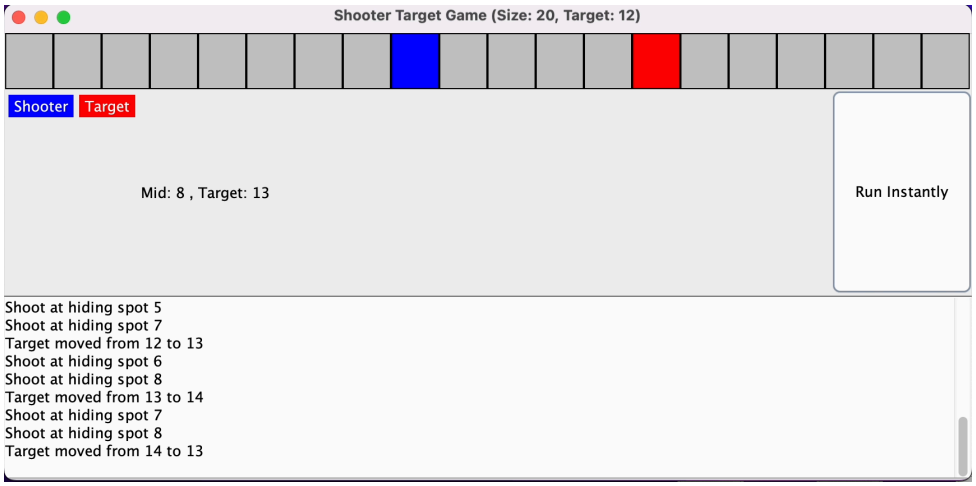

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 5 , Target: 12
Run Instantly
Shoot at hiding spot 2
Target moved from 13 to 12
Shoot at hiding spot 3
Shoot at hiding spot 6
Target moved from 12 to 11
Shoot at hiding spot 5
Shoot at hiding spot 4
Target moved from 11 to 12
Shoot at hiding spot 5

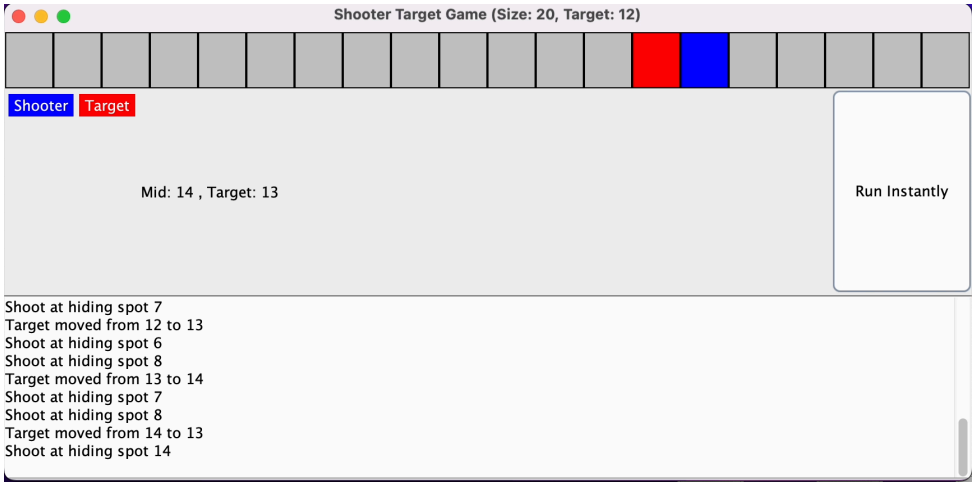

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 7 , Target: 13
Run Instantly
Shoot at hiding spot 3
Shoot at hiding spot 6
Target moved from 12 to 11
Shoot at hiding spot 5
Shoot at hiding spot 4
Target moved from 11 to 12
Shoot at hiding spot 5
Shoot at hiding spot 7
Target moved from 12 to 13

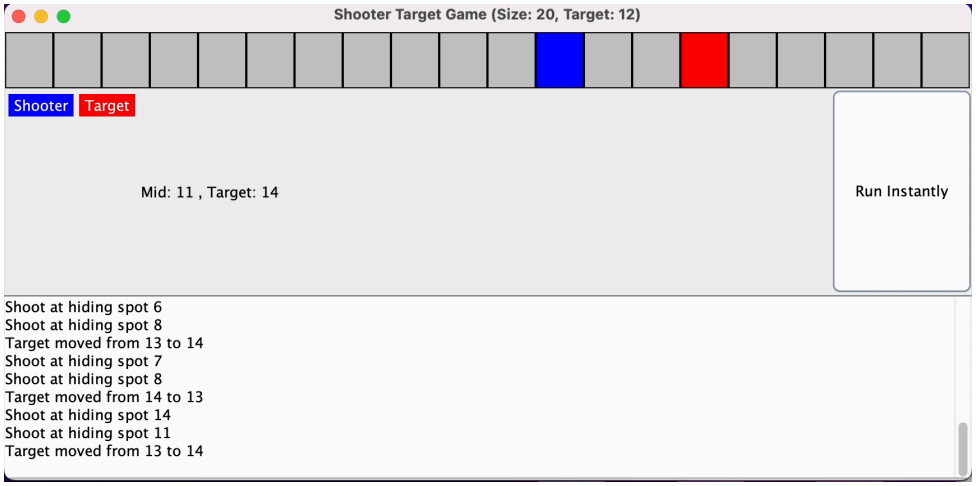

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 6 , Target: 13
Run Instantly
Shoot at hiding spot 6
Target moved from 12 to 11
Shoot at hiding spot 5
Shoot at hiding spot 4
Target moved from 11 to 12
Shoot at hiding spot 5
Shoot at hiding spot 7
Target moved from 12 to 13
Shoot at hiding spot 6

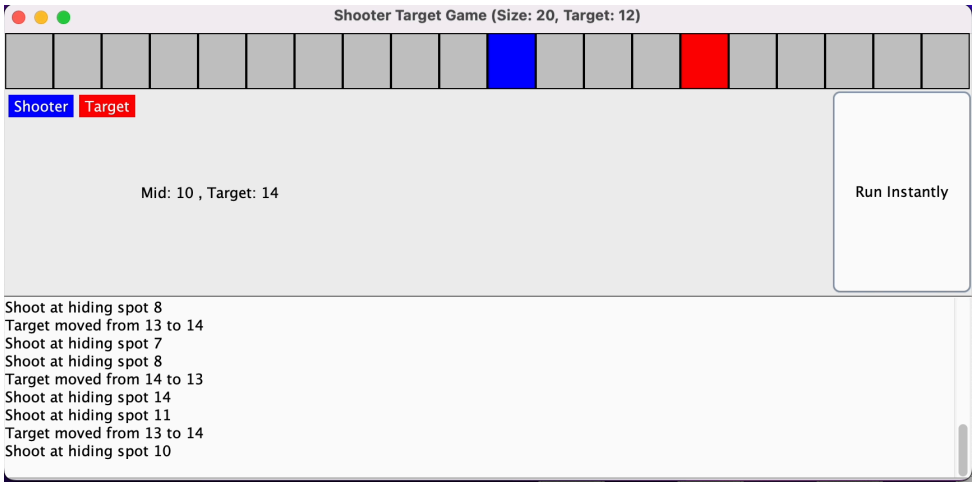

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 8 , Target: 14
Run Instantly
Shoot at hiding spot 5
Shoot at hiding spot 4
Target moved from 11 to 12
Shoot at hiding spot 5
Shoot at hiding spot 7
Target moved from 12 to 13
Shoot at hiding spot 6
Shoot at hiding spot 8
Target moved from 13 to 14

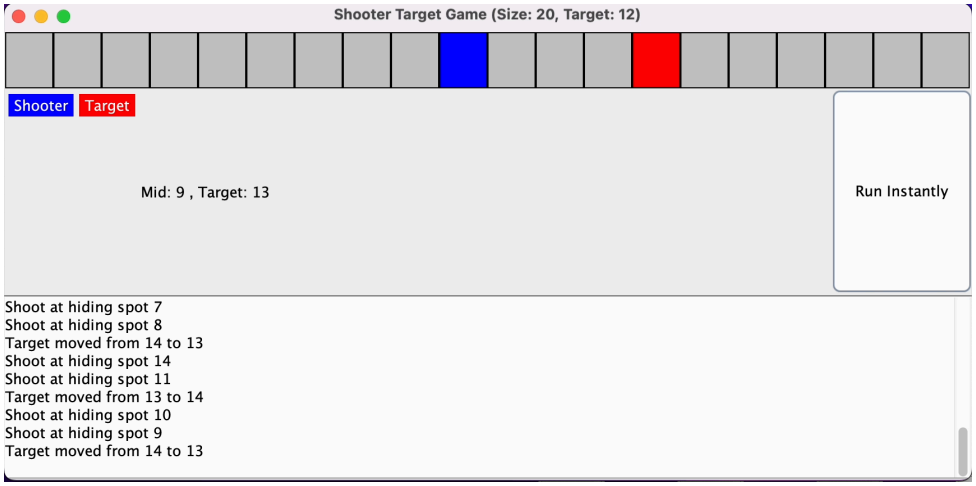

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 7 , Target: 14
Run Instantly
Shoot at hiding spot 4
Target moved from 11 to 12
Shoot at hiding spot 5
Shoot at hiding spot 7
Target moved from 12 to 13
Shoot at hiding spot 6
Shoot at hiding spot 8
Target moved from 13 to 14
Shoot at hiding spot 7

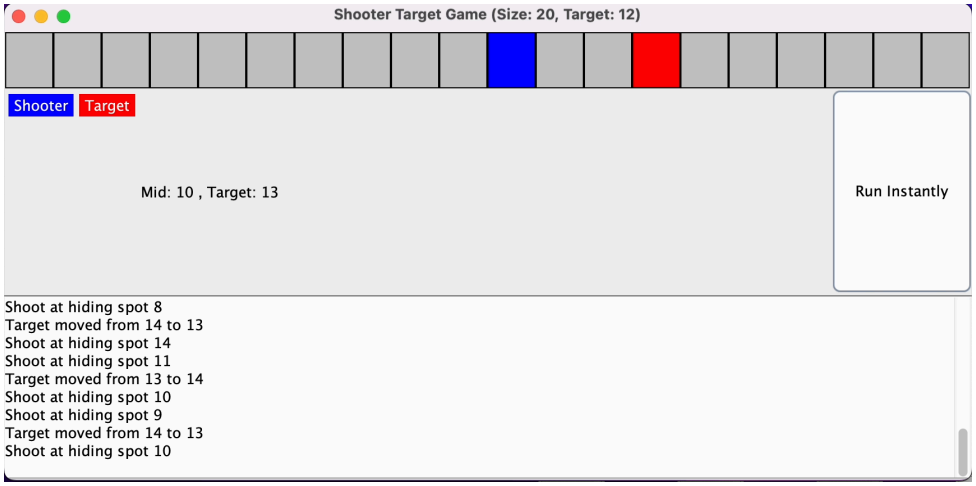

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 8 , Target: 13
Run Instantly
Shoot at hiding spot 5
Shoot at hiding spot 7
Target moved from 12 to 13
Shoot at hiding spot 6
Shoot at hiding spot 8
Target moved from 13 to 14
Shoot at hiding spot 7
Shoot at hiding spot 8
Target moved from 14 to 13

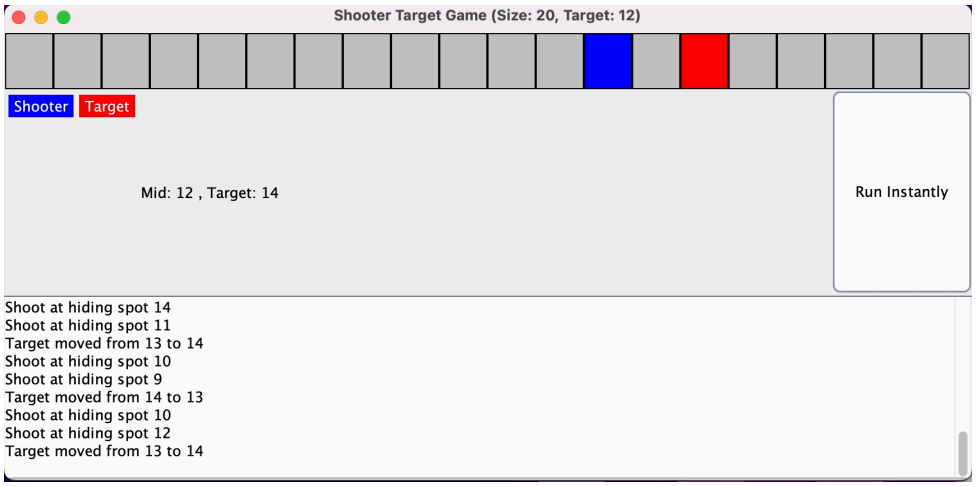

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 14 , Target: 13
Run Instantly
Shoot at hiding spot 7
Target moved from 12 to 13
Shoot at hiding spot 6
Shoot at hiding spot 8
Target moved from 13 to 14
Shoot at hiding spot 7
Shoot at hiding spot 8
Target moved from 14 to 13
Shoot at hiding spot 14

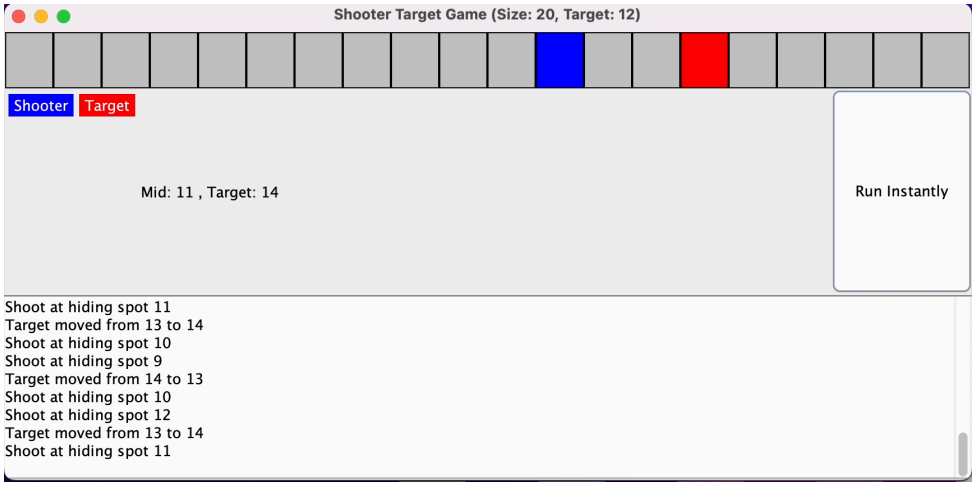

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 11 , Target: 14
Run Instantly
Shoot at hiding spot 6
Shoot at hiding spot 8
Target moved from 13 to 14
Shoot at hiding spot 7
Shoot at hiding spot 8
Target moved from 14 to 13
Shoot at hiding spot 14
Shoot at hiding spot 11
Target moved from 13 to 14

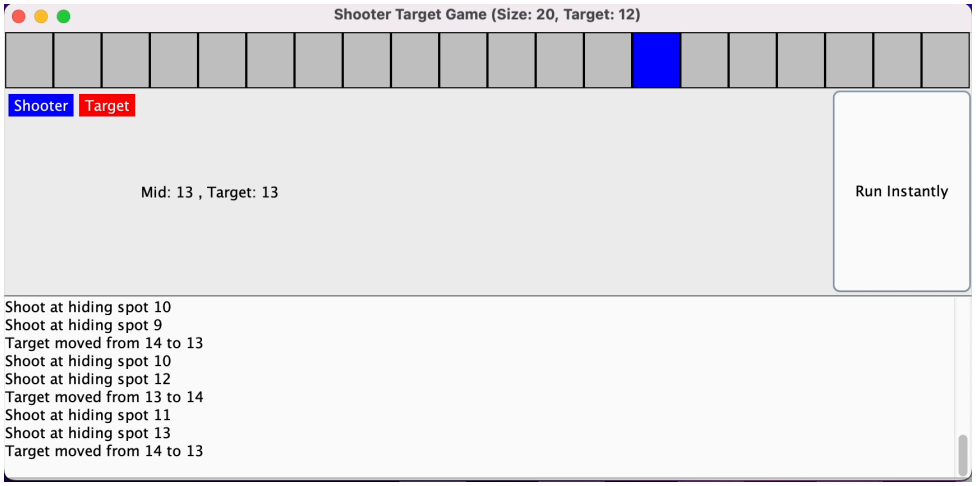

Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 10 , Target: 14
Run Instantly
Shoot at hiding spot 8
Target moved from 13 to 14
Shoot at hiding spot 7
Shoot at hiding spot 8
Target moved from 14 to 13
Shoot at hiding spot 14
Shoot at hiding spot 11
Target moved from 13 to 14
Shoot at hiding spot 10


Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 9 , Target: 13
Run Instantly
Shoot at hiding spot 7
Shoot at hiding spot 8
Target moved from 14 to 13
Shoot at hiding spot 14
Shoot at hiding spot 11
Target moved from 13 to 14
Shoot at hiding spot 10
Shoot at hiding spot 9
Target moved from 14 to 13


Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 10 , Target: 13
Run Instantly
Shoot at hiding spot 8
Target moved from 14 to 13
Shoot at hiding spot 14
Shoot at hiding spot 11
Target moved from 13 to 14
Shoot at hiding spot 10
Shoot at hiding spot 9
Target moved from 14 to 13
Shoot at hiding spot 10


Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 12 , Target: 14
Run Instantly
Shoot at hiding spot 14
Shoot at hiding spot 11
Target moved from 13 to 14
Shoot at hiding spot 10
Shoot at hiding spot 9
Target moved from 14 to 13
Shoot at hiding spot 10
Shoot at hiding spot 12
Target moved from 13 to 14


Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 11 , Target: 14
Run Instantly
Shoot at hiding spot 11
Target moved from 13 to 14
Shoot at hiding spot 10
Shoot at hiding spot 9
Target moved from 14 to 13
Shoot at hiding spot 10
Shoot at hiding spot 12
Target moved from 13 to 14
Shoot at hiding spot 11


Shooter Target Game (Size: 20, Target: 12)
Shooter
Target
Mid: 13 , Target: 13
Run Instantly
Shoot at hiding spot 10
Shoot at hiding spot 9
Target moved from 14 to 13
Shoot at hiding spot 10
Shoot at hiding spot 12
Target moved from 13 to 14
Shoot at hiding spot 11
Shoot at hiding spot 13
Target moved from 14 to 13

6) Task (6)

6.1) Description

Description

Given an n × n point lattice (n > 2).

The task is to cross out all points using 2n − 2 straight lines, without lifting the pen.

- Constraints:
  - Each line must be straight with no retracing.
  - The pen must remain on the paper throughout the process.
  - Points may be crossed multiple times if needed.
  - Each turn connects exactly two lines; no overlaps or backtracking on segments.
- Key properties:
  - The lattice forms a square grid with n rows and n columns.
  - The total number of intersection points is n^2.
  - A total of 2n − 2 lines must cover all points without lifting the pen.
  - The challenge lies in arranging the lines to ensure all points are intersected.
- The method should work by:
  - Constructing a path that moves across the grid in an efficient pattern.
  - Ensuring each line connects logically to the next to form a continuous path.
  - Utilizing grid symmetry to reduce complexity.
  - Designing the sequence to maximize coverage with minimal direction changes.

6.2) Problem Description

The code generates a solenoid (spiral) pattern on an n×n grid by:

- Calculating corner points of the spiral
- Drawing lines between consecutive points
- Using incrementing values to represent each segment
- Marking the final connection with a special value (777)

Code Analysis: Key Functions :

- `drawLine`:

- Draws horizontal or vertical lines between two points
- Handles coordinate ordering automatically
- Marks the line with a specified value

- `CornerPoints`:
  - Calculates all the corner points of the spiral pattern
  - Works layer by layer from the outside in
  - Returns a vector of (x,y) coordinate pairs
- `solenoidPattern`:
  - Uses the corner points to draw the complete spiral
  - Handles even and odd grid sizes differently for the final connection
  - Uses incrementing values for each line segment
- Main Execution Flow:
  - Initialize an n×n grid filled with zeros
  - Calculate all corner points of the spiral
  - Draw lines between consecutive points with incrementing values
  - Make the final connection with value 777
  - Print the resulting grid

6.3) Detailed assumptions

- Board Size:
  - The board is always square with size n × n
  - The code accepts any positive integer n (not restricted to powers of 2)
  - The implementation handles both even and odd sizes differently
- Pattern Generation:
  - The code generates a spiral (solenoid) pattern starting from the outer layer and moving inward
  - Each complete loop around the spiral increments the line value by 1 (starting from 99)
  - The final connection is marked with a special value (777)
- Line Drawing:

- Lines can be either horizontal or vertical
- The drawLine function handles both directions and ensures coordinates are ordered correctly

6.4) Solenoid Spiral Algorithm

6.4.1) Pseudocode

```text
FUNCTION drawLine(array, x1, y1, x2, y2, value):
    IF x1 == x2 THEN  // Horizontal line
        IF y1 > y2 THEN
            SWAP(y1, y2)
        END IF
        FOR i FROM y1 TO y2:
            array[x1][i] = value
    ELSE IF y1 == y2 THEN  // Vertical line
        IF x1 > x2 THEN
            SWAP(x1, x2)
        END IF
        FOR i FROM x1 TO x2:
            array[i][y1] = value
    END IF
    RETURN array
END FUNCTION

FUNCTION CornerPoints(array, n):
    points = [(0, 0)]  // Initialize with top-left corner

    layers = CEILING((n + 1) / 2)

    FOR i FROM 0 TO (layers * 4 - 1):
        layer = i // 4  // Integer division

        SWITCH (i MOD 4):
            CASE 0:  // Top edge (left to right)
                x1 = layer
                y1 = layer
                x2 = layer
                y2 = n - 1 - layer
            CASE 1:  // Right edge (top to bottom)
                x1 = layer
                y1 = n - 1 - layer
                x2 = n - 1 - layer
                y2 = n - 1 - layer
```

```
            CASE 2:  // Bottom edge (right to left)
                x1 = n - 1 - layer
                y1 = n - 1 - layer
                x2 = n - 1 - layer
                y2 = layer
            CASE 3:  // Left edge (bottom to top)
                x1 = n - 1 - layer
                y1 = layer
                x2 = layer + 1
                y2 = layer
        END SWITCH

        IF NOT (x1 == x2 AND y1 == y2) THEN
            points.APPEND((x2, y2))
        END IF
    END FOR

    RETURN points
END FUNCTION

FUNCTION solenoidPattern(points, array, n):
    lineValue = 99

    FOR i FROM 0 TO points.LENGTH - 2:
        P1 = points[i]
        P2 = points[i + 1]
        array = drawLine(array, P1.x, P1.y, P2.x, P2.y, lineValue)
        lineValue = lineValue + 1
    END FOR

    // Handle final connection
    IF n MOD 2 == 0 THEN
        lastPoint = points[points.LENGTH - 2]
        array = drawLine(array, lastPoint.x, lastPoint.y, lastPoint.x,
0, 777)
    ELSE
        lastPoint = points[points.LENGTH - 1]
        array = drawLine(array, lastPoint.x, lastPoint.y, lastPoint.x, n
- 1, 777)
    END IF
END FUNCTION

// Main algorithm
```

```
FUNCTION main():
    n = 8  // Example size
    array = CREATE 2D ARRAY OF SIZE n×n INITIALIZED TO 0

    points = CornerPoints(array, n)
    solenoidPattern(points, array, n)

    // Print the resulting pattern
    FOR i FROM 0 TO n-1:
        FOR j FROM 0 TO n-1:
            PRINT array[i][j] + " "
        END FOR
        PRINT NEWLINE
    END FOR
END FUNCTION
```

Listing 6.1: Solenoid Pattern - Solenoid Spiral Algorithm Pseudocode

6.4.2) Code

C++

```cpp
#include <iostream>
#include <vector>
using namespace std;
vector<vector<int>> drawLine(vector<vector<int>>& array, int x1, int y1, int x2, int y2, int value)
{
    if (x1 == x2)
    {
        //Horizontal Line
        if (y1 > y2) swap(y1, y2);
        for(int i = y1 ; i≤y2 ; i++)
        {
            array[x1][i] = value;
        }
    }
    else if (y1 == y2)
    {
        //Vertical Line
        if (x1 > x2) swap(x1, x2);
        for(int i = x1 ; i≤x2 ; i++)
        {
            array[i][y1] = value;
        }
    }
    return array;
```

```
}
vector<pair<int, int>> CornerPoints(vector<vector<int>>& array ,int n)
{
    vector<pair<int, int>> Point;
    Point.push_back({0,0});

    int line = 99;

    int layers = (n + 1) / 2;
    int x1, y1, x2, y2;

    for (int i = 0; i < layers * 4 - 1; ++i)
    {
        int layer = i / 4;

        switch (i % 4)
        {
            case 0:
                x1 = layer;
                y1 = layer;
                x2 = layer;
                y2 = n - 1 - layer;
                break;
            case 1:
                x1 = layer;
                y1 = n - 1 - layer;
                x2 = n - 1 - layer;
                y2 = n - 1 - layer;
                break;
            case 2:
                x1 = n - 1 - layer;
                y1 = n - 1 - layer;
                x2 = n - 1 - layer;
                y2 = layer;
                break;
            case 3:
                x1 = n - 1 - layer;
                y1 = layer;
                x2 = layer + 1;
                y2 = layer;
                break;
        }
        if(x1 == x2 && y1 == y2) break;
        Point.push_back({x2,y2});
```

```
    }
    return Point;
}
void solenoidPattern(vector<pair<int, int>> Point, vector<vector<int>>&
array, int n)
{
    int line = 99;
    pair<int, int> P1;
    pair<int, int> P2;
    for (int i = 0; i < Point.size() - 1; i++)
    {
        P1 = Point[i];
        P2 = Point[i + 1];
        array = drawLine(array, P1.first, P1.second, P2.first,
        P2.second, ++line);
    }

    if (n%2 == 0)
    {
        array = drawLine(array, P1.first, P1.second,  P1.first, 0,
        777);
    }
    else
    {
        array = drawLine(array, P2.first, P2.second,  P2.first, n - 1,
        777);
    }
}
int main()
{
    int n = 8;
    vector<vector<int>> array(n, vector<int>(n, 0));
    vector<pair<int, int>> Point;

    Point = CornerPoints(array,n);
    solenoidPattern(Point,array,n);

    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            cout << array[i][j] << " ";
```

```
        }
        cout << endl;
    }
    return 0;
}
```

Listing 6.2: Solenoid Pattern - Solenoid Spiral Algorithm C++ Code

6.4.3) Test Cases

Output

```
100 100 100 100 100 100 100 101
104 104 104 104 104 104 105 101
103 108 108 108 108 109 105 101
103 107 112 112 113 109 105 101
777 777 777 777 777 109 105 101
103 107 111 110 110 110 105 101
103 107 106 106 106 106 106 101
103 102 102 102 102 102 102 102
```

Listing 6.3: Solenoid Pattern - Solenoid Spiral Algorithm Test Case Output

6.4.4) Complexity analysis

- Time Complexity
  - CornerPoints calculation: O(n) - visits each layer once
  - Line drawing: $O(n^2)$ - in worst case might need to fill most of the grid
  - Overall: $O(n^2)$ - proportional to the grid size
- Space Complexity
  - Grid storage: $O(n^2)$ for the 2D array
  - Corner points storage: O(n) for the point list
  - Overall: $O(n^2)$ - dominated by the grid storage

6.5) Brute-Force Algorithm

```text
# Helper function to check if all points have been covered
function isAllCovered(crossedPoints, n):
    for i = 0 to n-1:
        for j = 0 to n-1:
            if (i, j) not in crossedPoints:
                return False
    return True

# Function to try drawing lines in a brute-force manner
```

```
function drawLines(grid, n, linesLeft, crossedPoints):
    if linesLeft == 0:  # No more lines left to draw
        return isAllCovered(crossedPoints, n)

    # Try all possible line directions and positions
    for i = 0 to n-1:

        # Try drawing a horizontal line at row i
        if drawHorizontalLine(grid, i, crossedPoints):
            if drawLines(grid, n, linesLeft - 1, crossedPoints):
                return True
            undoHorizontalLine(grid, i, crossedPoints)  # Undo the horizontal line


        # Try drawing a vertical line at column i
        if drawVerticalLine(grid, i, crossedPoints):
            if drawLines(grid, n, linesLeft - 1, crossedPoints):
                return True
            undoVerticalLine(grid, i, crossedPoints)  # Undo the vertical line

        # Optionally, you can also try diagonal lines or other configurations based on the problem's structure
    return False  # If no solution found



# Function to start the drawing process
function solveLattice(n):
    # Initialize the grid and list of crossed points
    grid = createEmptyGrid(n)
    crossedPoints = set()  # Will hold the crossed points

    # Try to draw 2n - 2 lines
    linesLeft = 2 * n - 2
    if drawLines(grid, n, linesLeft, crossedPoints):
        print("Solution found!")
    else:
        print("No solution possible.")

# Main function to trigger the solution
```

```
49 solveLattice(n)
```

Listing 6.4: Solenoid Pattern - Brute-Force Algorithm Pseudocode

6.6) Comparison Between the Solenoid Spiral and Brute Force Approaches

Feature	Solenoid Spiral (Current Approach)	Brute Force Approach
Technique	Iterative layer-by-layer traversal	Trial-and-error with backtracking
Time Complexity	$O(n^2)$ (optimal for grid traversal)	$O\left(3^{\{n^2\}}\right)$ (exponential, impractical)
Space Complexity	$O(n)$ (stores only corner points)	$O(n^2)$ (stores full grid states)
Guaranteed Solution	Always produces correct spiral	May fail or take extremely long
Speed	Fast (completes in milliseconds for $n = 100$)	Extremely slow (may fail for $n > 8$)
Implementation	Simple loops and line-drawing	Complex backtracking logic
Flexibility	Only generates spirals	Can solve arbitrary patterns
Best Use Case	Generating spirals efficiently	Problems with no structured solution

Table 6.1: Comparison Between the Solenoid Spiral and Brute Force Approaches

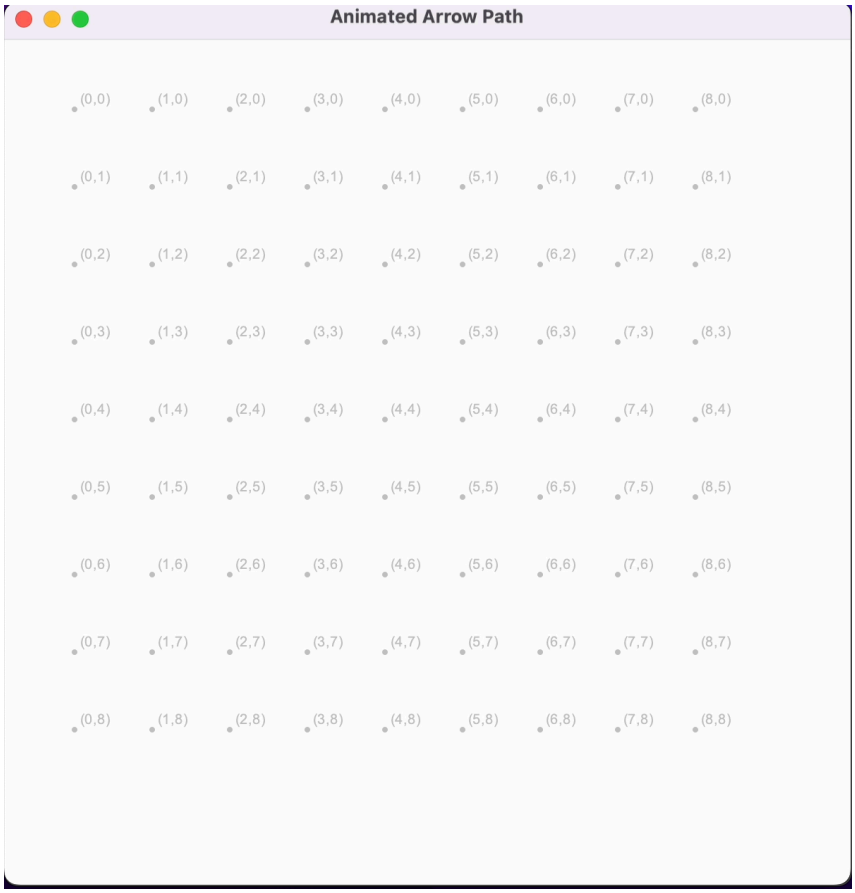
6.7) Conclusion

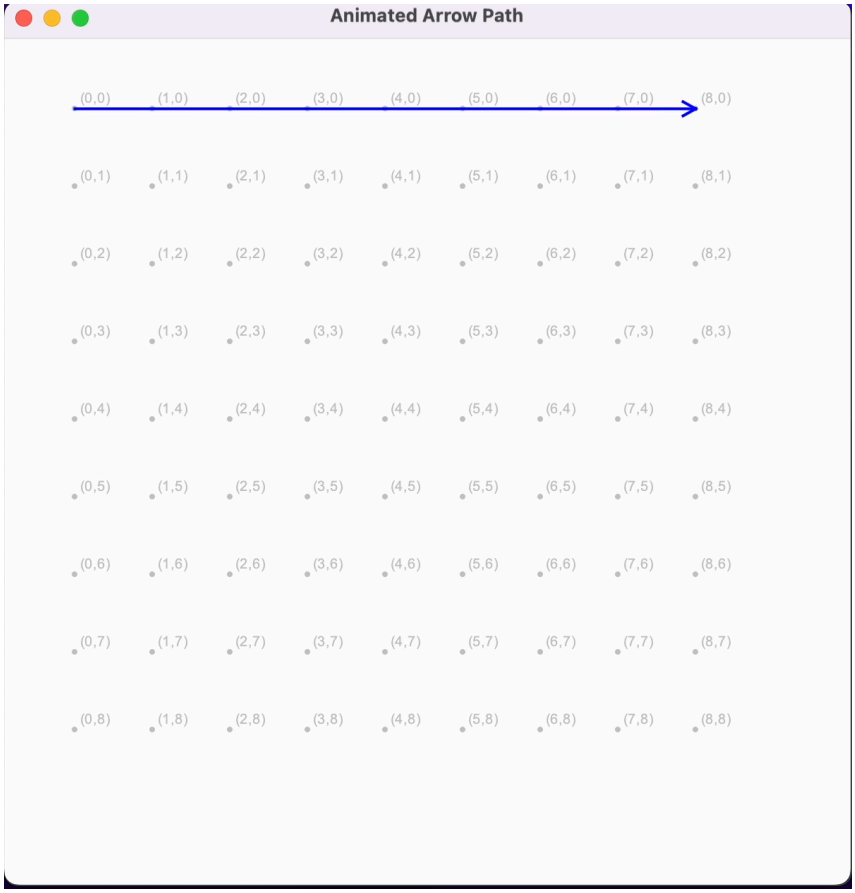
The code provides an efficient solution for generating a solenoid/spiral pattern on an n×n grid.

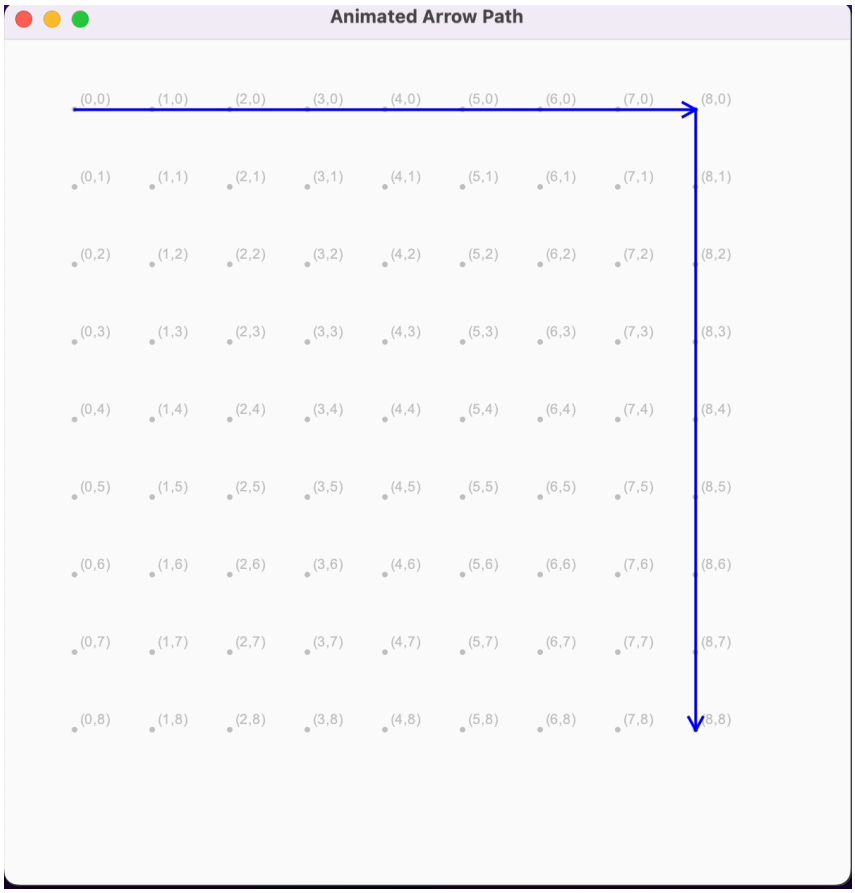
Key characteristics include:

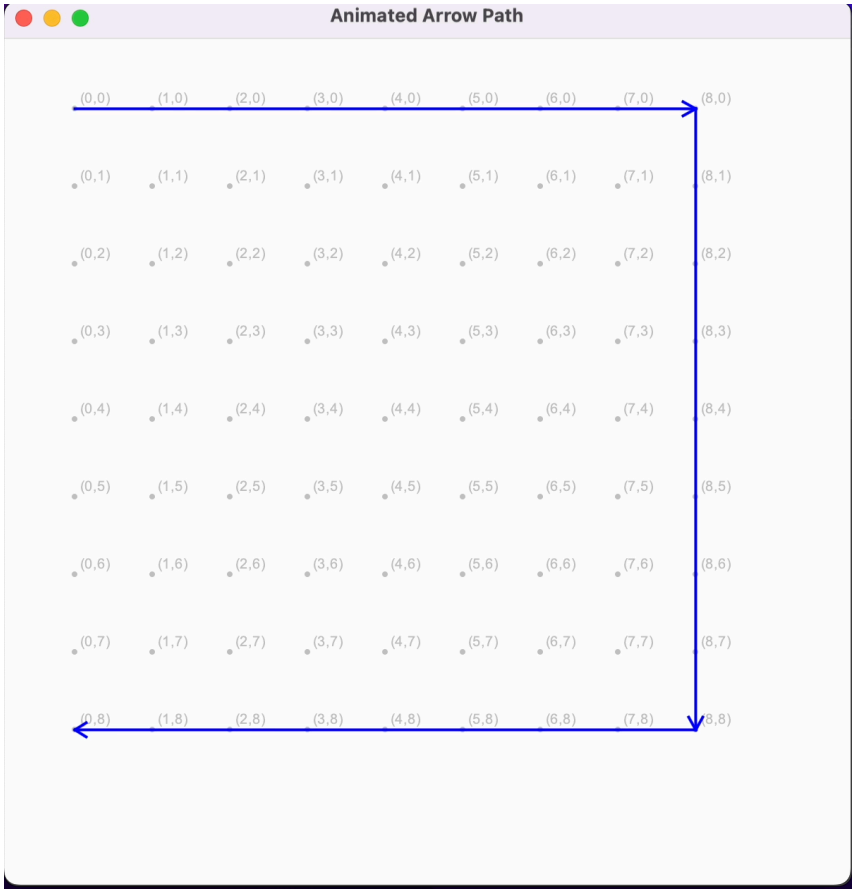
- Layer-based Approach: Systematically processes the grid from outer layers inward
- Efficiency: Runs in O(n²) time, optimal for this problem
- Flexibility: Handles both even and odd grid sizes
- Visual Clarity: Uses incrementing values to clearly show the spiral path

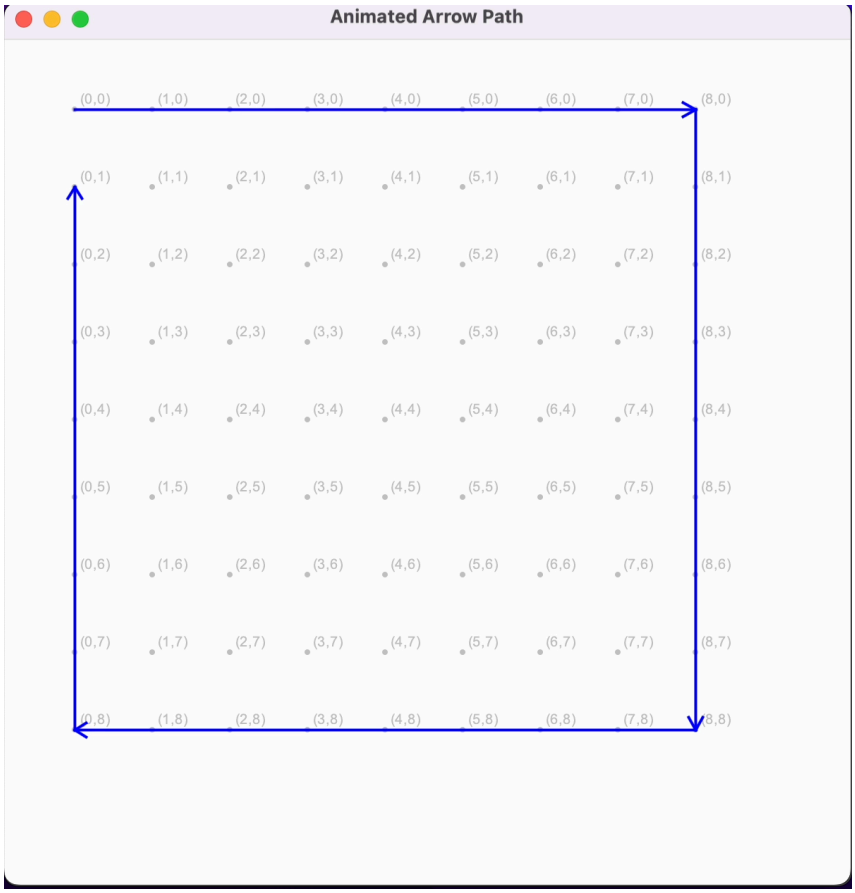
6.8) GUI Screenshots

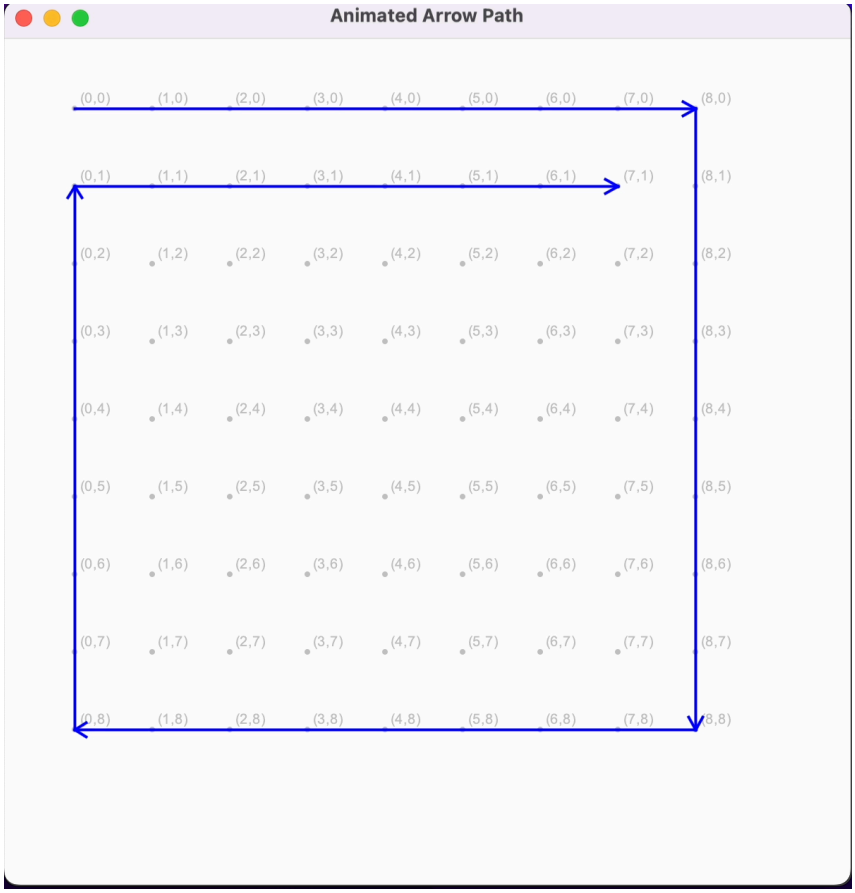


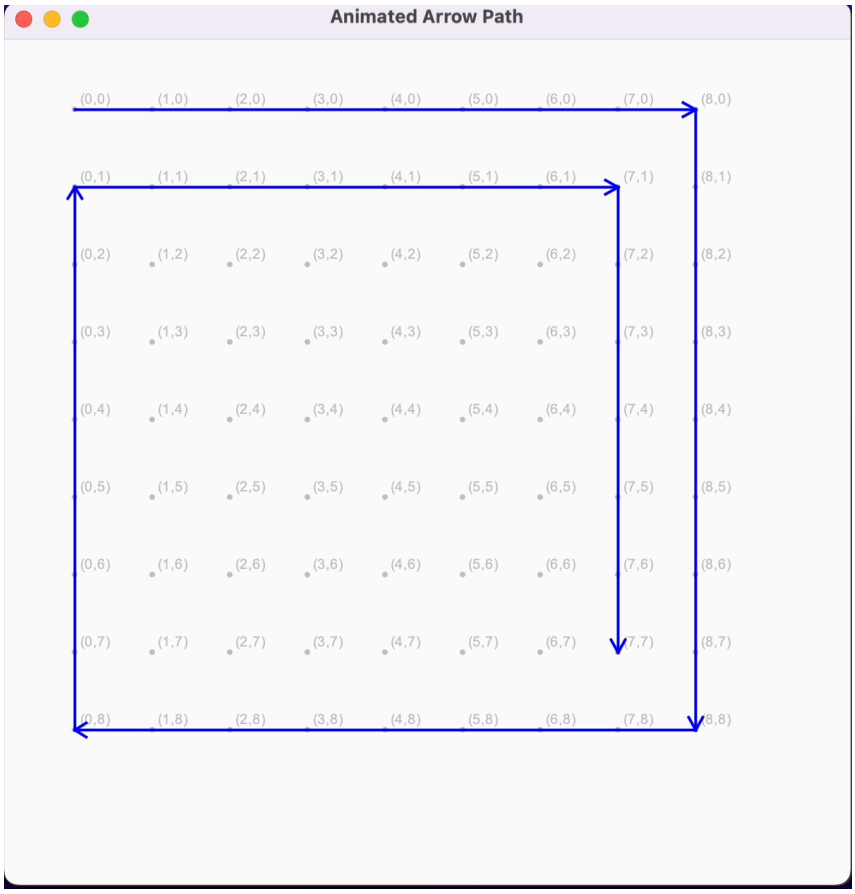


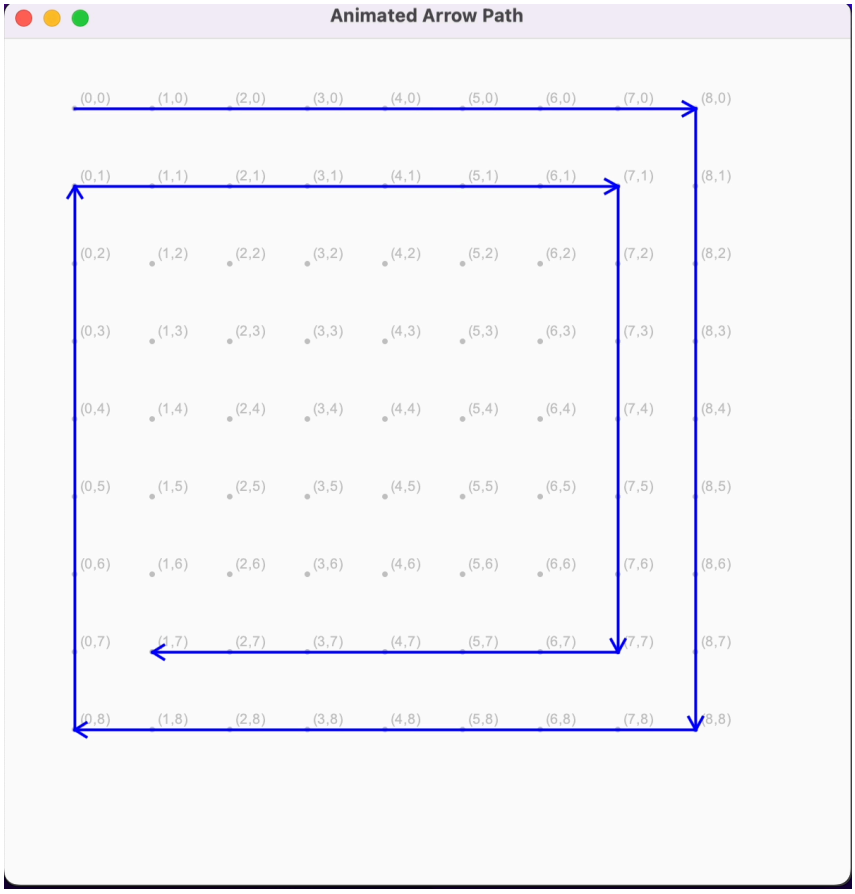

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

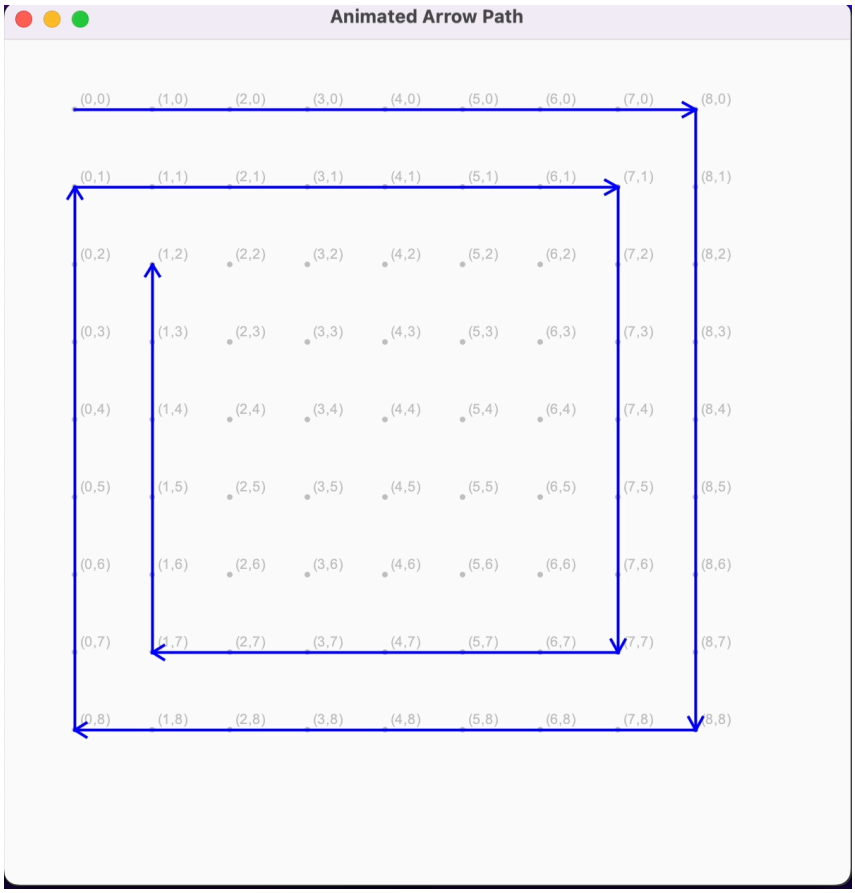

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

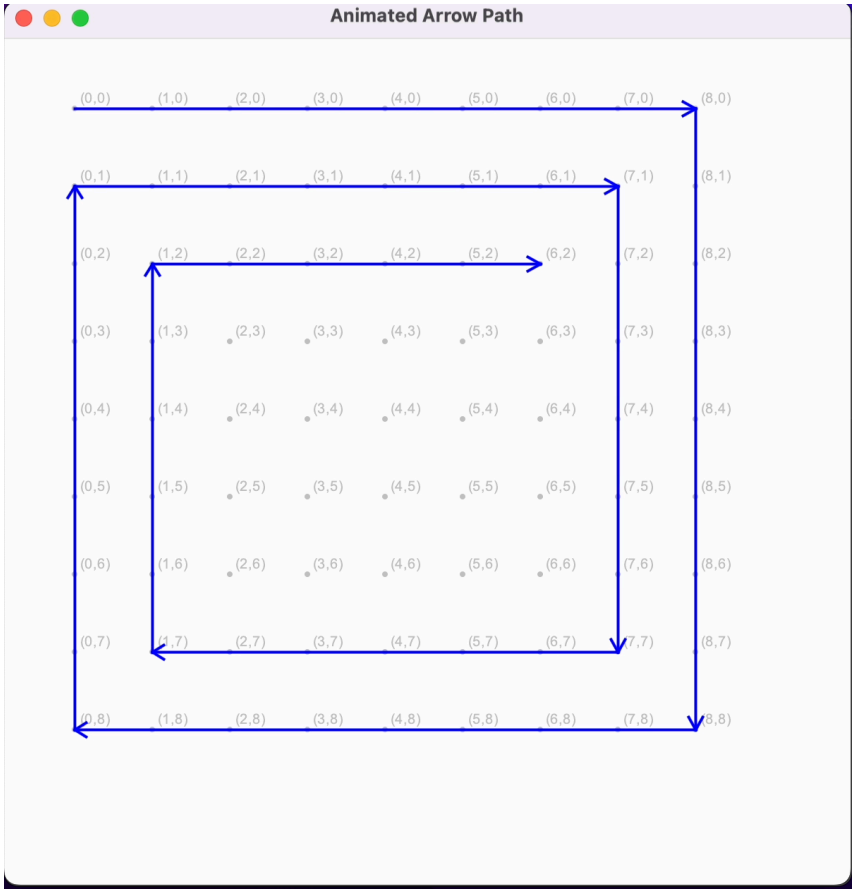

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

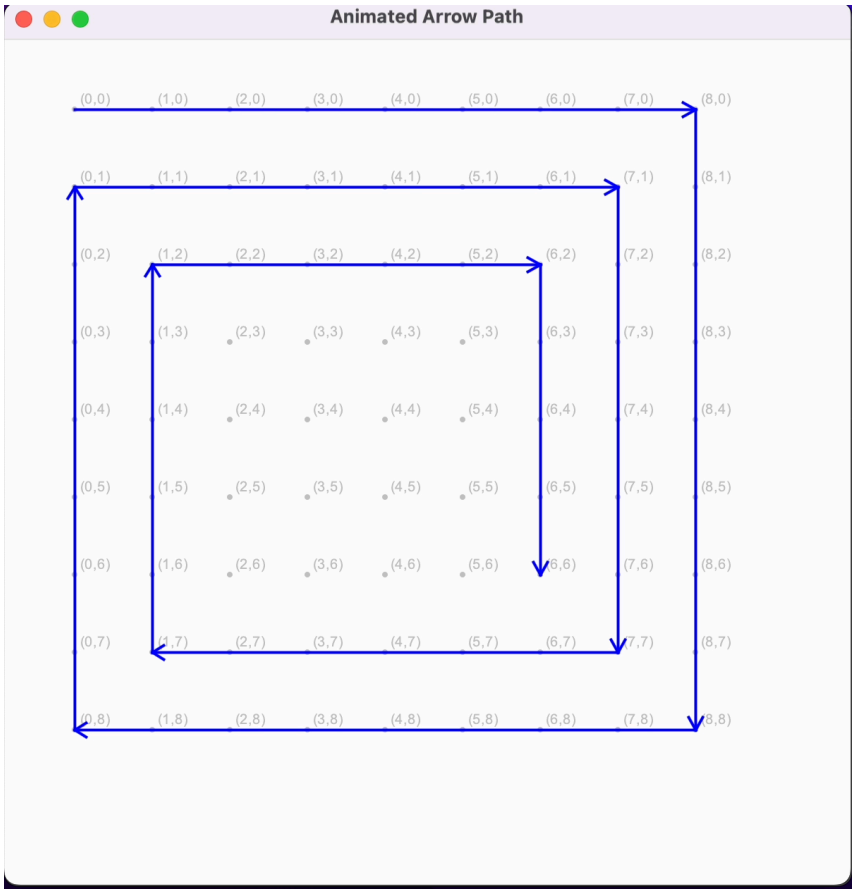

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

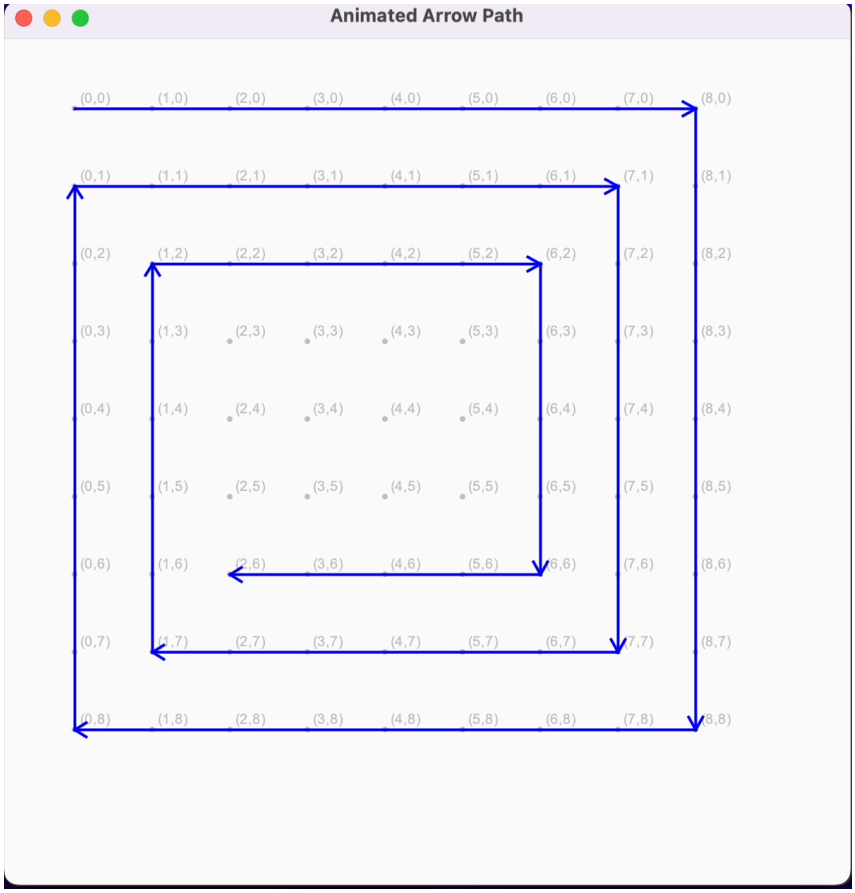

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

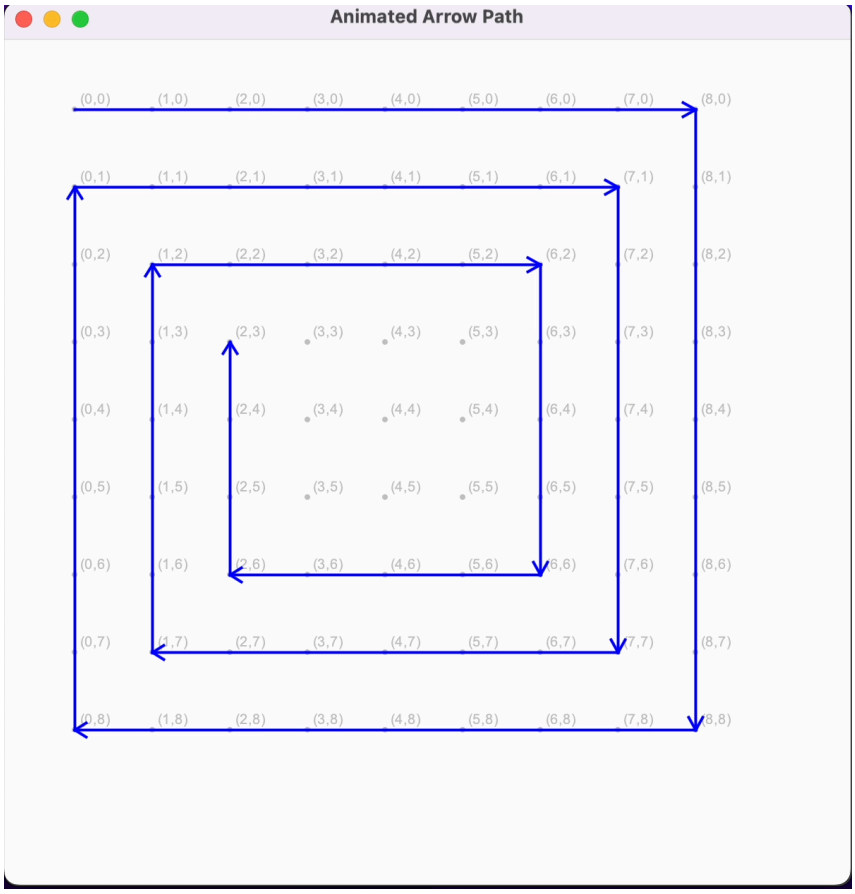

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

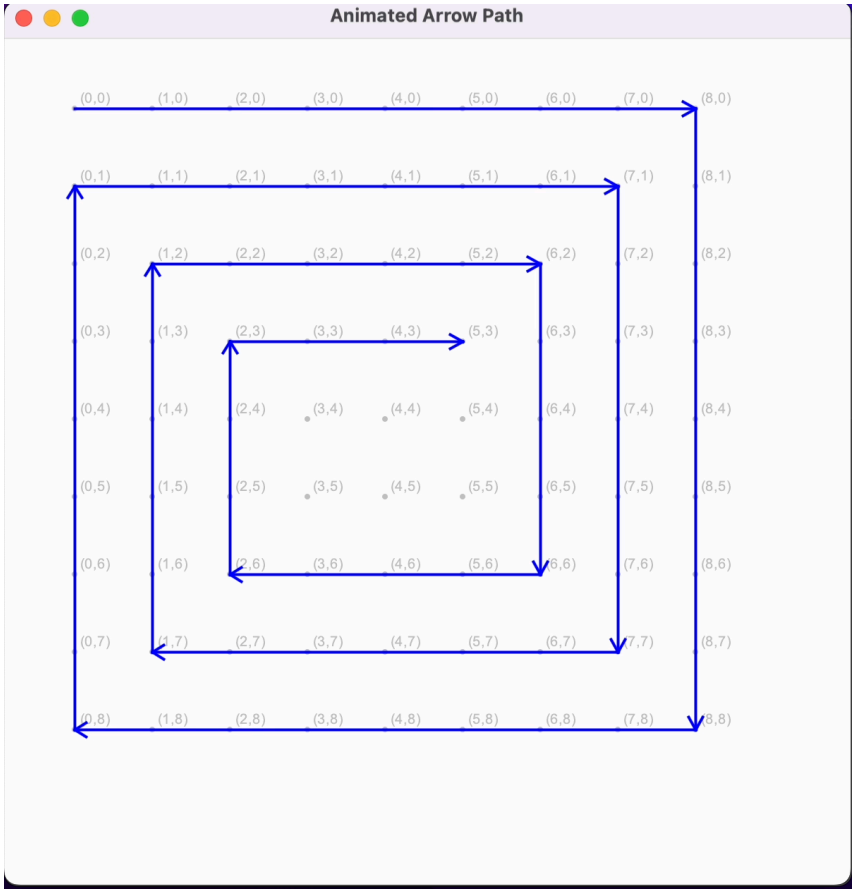

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

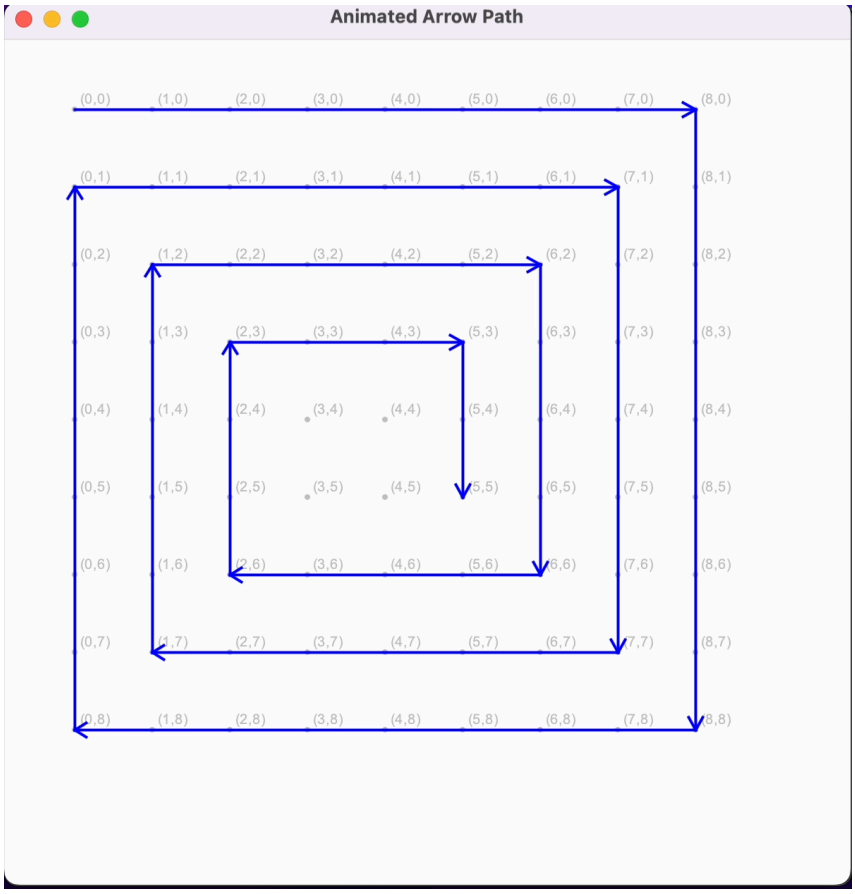

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

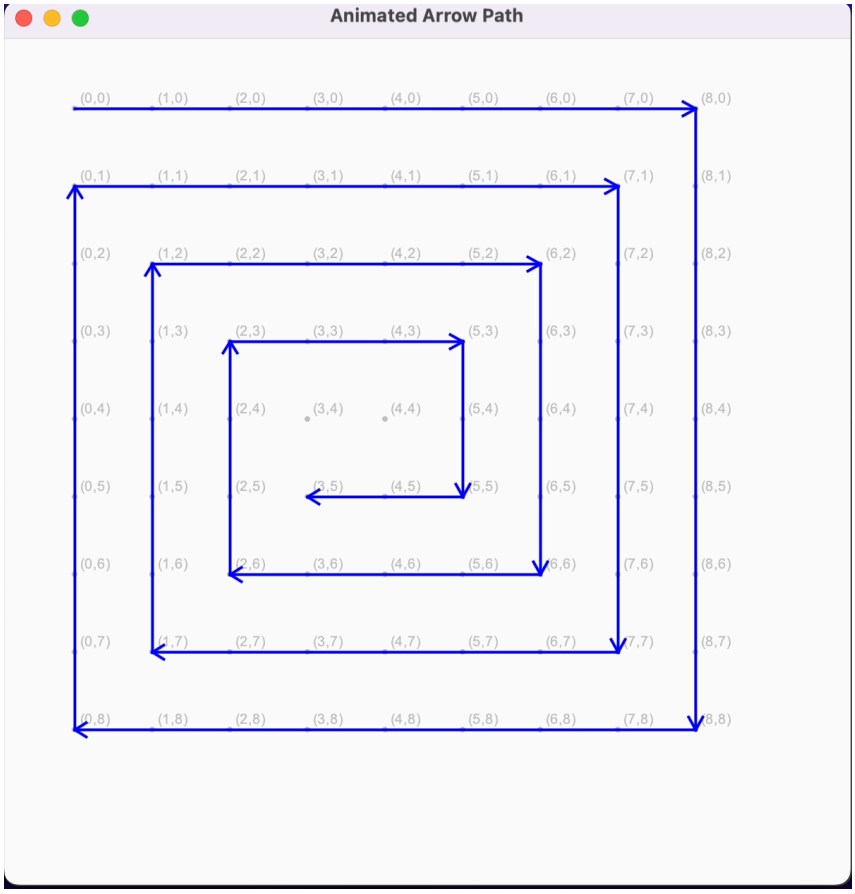

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

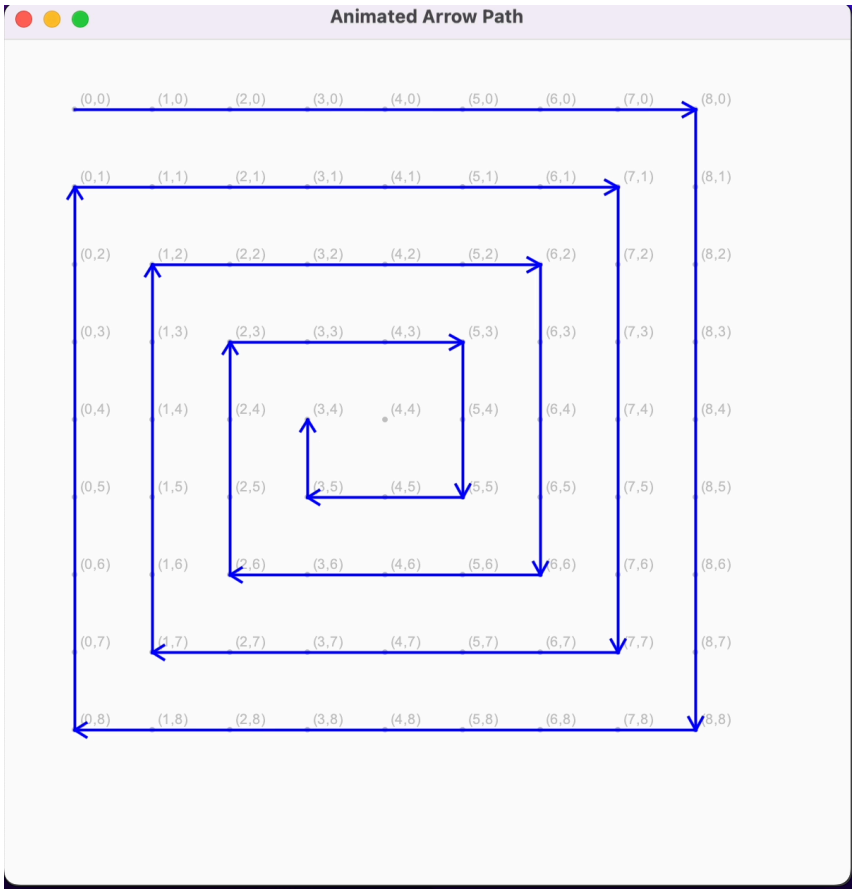

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

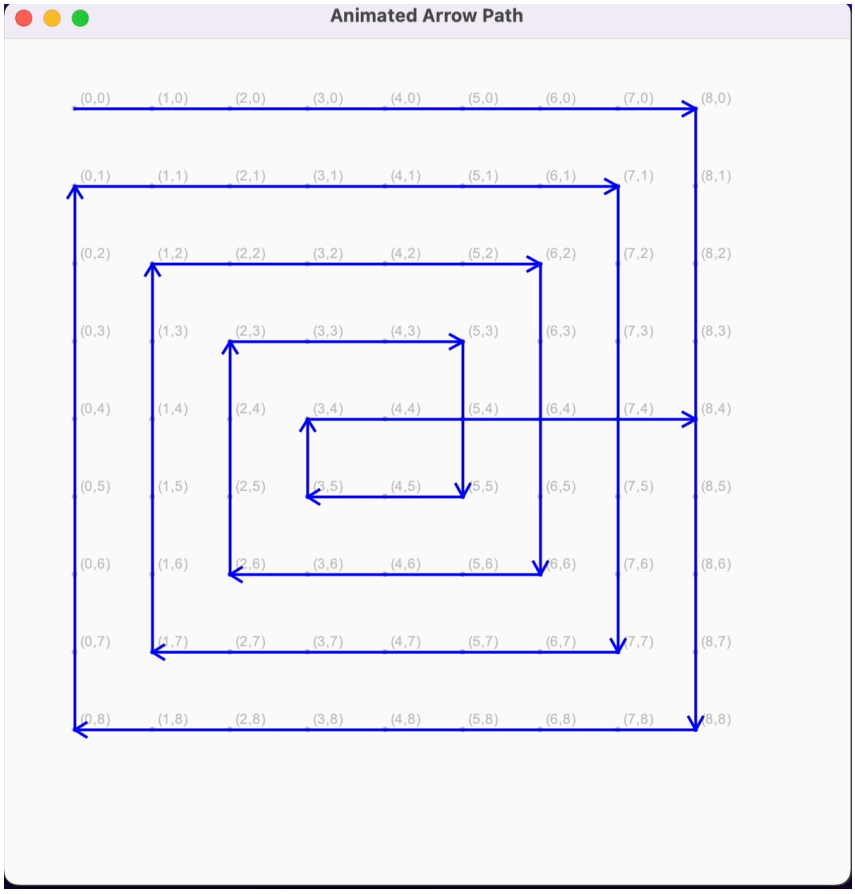

Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)


Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)


Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)


Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)


Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)


Animated Arrow Path
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2) (6,2) (7,2) (8,2)
(0,3) (1,3) (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4) (8,4)
(0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5)
(0,6) (1,6) (2,6) (3,6) (4,6) (5,6) (6,6) (7,6) (8,6)
(0,7) (1,7) (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7)
(0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8) (8,8)

Research Task

7) Research Task

7.1) Hamiltonian Circuit Problem

7.1.1) Description

The Hamiltonian circuit problem is a classic problem in graph theory.

> **Problem Statement**
>
> Given a graph $G = (V, E)$, the task is to determine whether there exists a Hamiltonian circuit or Hamiltonian cycle. A Hamiltonian cycle is a cycle that visits each vertex exactly once and returns to the starting vertex, forming a loop.

- **Input:** A graph G with a set of vertices V and edges E.
- **Output:** A boolean value indicating whether a Hamiltonian cycle exists in the graph.

This problem is NP-complete, meaning there is no known polynomial-time algorithm to solve it in the general case. Finding such cycles can be computationally expensive for large graphs.

7.1.2) Algorithms

7.1.2.1) Backtracking Algorithm

The idea of backtracking is to try and construct a solution incrementally. We attempt to construct the Hamiltonian cycle one vertex at a time. If we can reach a valid solution, we return `true`; otherwise, we backtrack by removing the last added vertex and trying a different one.

7.1.2.1.1) Algorithm Explanation

1. Start from a vertex and attempt to build the cycle by visiting one vertex at a time.
2. Mark each vertex as visited as we add it to the cycle.
3. For each vertex, try all possible next vertices (neighbors) that haven't been visited yet.
4. If we manage to visit all vertices and return to the starting vertex, we have found a Hamiltonian cycle.
5. If no solution is found, backtrack by removing the last added vertex and try a different possibility.

7.1.2.1.2) Code

```cpp
#include <iostream>
#include <vector>
using namespace std;

```

```cpp
// Function to check if the current vertex can be added to the Hamiltonian Cycle
bool isSafe(int v, vector<vector<int>>& graph, vector<int>& path, int pos) {
    // Check if this vertex is adjacent to the last vertex in the path
    if (graph[path[pos - 1]][v] == 0) {
        return false;
    }

    // Check if the vertex has already been included in the path
    for (int i = 0; i < pos; i++) {
        if (path[i] == v) {
            return false;
        }
    }

    return true;
}

// Utility function to solve the Hamiltonian cycle problem
bool hamiltonianCycleUtil(vector<vector<int>>& graph, vector<int>& path, int pos) {
    // Base case: If all vertices are included in the cycle
    if (pos == graph.size()) {
        // Check if the current vertex is connected to the first vertex to complete the cycle
        if (graph[path[pos - 1]][path[0]] == 1) {
            return true;
        }
        return false;
    }

    // Try different vertices as the next candidate in the Hamiltonian cycle
    for (int v = 1; v < graph.size(); v++) {
        if (isSafe(v, graph, path, pos)) {
            path[pos] = v;

            // Recur to build the rest of the path
            if (hamiltonianCycleUtil(graph, path, pos + 1)) {
                return true;
            }
```

```

            // Backtrack if adding vertex v doesn't lead to a solution
            path[pos] = -1;
        }
    }

    return false;
}
// Main function to solve the Hamiltonian cycle problem
bool hamiltonianCycle(vector<vector<int>>& graph) {
    vector<int> path(graph.size(), -1);  // Initialize the path with -1
    path[0] = 0;  // Start with the first vertex
    // Start the backtracking process from the second vertex
    if (hamiltonianCycleUtil(graph, path, 1)) {
        // Print the Hamiltonian cycle
        cout << "Hamiltonian Cycle found: ";
        for (int i = 0; i < path.size(); i++) {
            cout << path[i] << " ";
        }
        cout << path[0] << endl;  // To complete the cycle
        return true;
    } else {
        cout << "No Hamiltonian Cycle found." << endl;
        return false;
    }
}
```

Listing 7.1: Hamiltonian Circuit Problem - Backtracking Algorithm

7.1.2.1.3) Cases

- Case (1): Hamiltonian Cycle Exists

Graph:

```text
   (0)
  /   \
(1)---(2)
  \   /
   (3)
```

This graph has a Hamiltonian cycle: `0 → 1 → 2 → 3 → 0`

Adjacency Matrix Input:

```text
vector<vector<int>> graph = {
    {0, 1, 1, 0},
```

```
    {1, 0, 1, 1},
    {1, 1, 0, 1},
    {0, 1, 1, 0}
};
```

Expected Output:

```
Output
Hamiltonian Cycle found: 0 1 2 3 0
```

- Case(2) No Hamiltonian Cycle

Graph:

```text
(0)———(1)       (2)
          \   /
           (3)
```

In this graph, node 2 is not connected in a way that allows a cycle to pass through all nodes exactly once and return.

Adjacency Matrix Input:

```cpp
vector<vector<int>> graph = {
    {0, 1, 0, 0},
    {1, 0, 0, 1},
    {0, 0, 0, 1},
    {0, 1, 1, 0}
};
```

Expected Output:

```
Output
No Hamiltonian Cycle found.
```

7.1.2.1.4) Time Complexity

The worst-case time complexity of this algorithm is $O(n!)$, where n is the number of vertices.

This is because there are $n!$ possible ways to visit all the vertices.

7.1.2.2) Dynamic Programming (Held-Karp Algorithm):

This algorithm uses dynamic programming to solve the Traveling Salesman Problem (TSP), which is closely related to the Hamiltonian cycle problem. significantly faster than brute-force methods for small graphs.

7.1.2.2.1) Algorithm Explanation:

Step 1: Problem Definition

- Given a complete or partially connected weighted graph with n vertices.
- Goal: Find the minimum-cost Hamiltonian cycle (visit every vertex once, return to start).

Step 2: Represent States Using Bitmasking

- Let each state be represented by:
  - A subset of visited vertices → use bitmask (e.g., 1011 means vertices 0, 1, and 3 are visited).
  - A current vertex we're at.

Step 3: Define DP Table

- Create a 2D DP table:

```
dp[mask][u] = <"minimum cost to reach vertex u using the vertices in mask">
```

  - `mask`: Bitmask of visited vertices (0 to $2n - 12^n - 12n - 1$)
  - `u`: The last vertex visited in this subset
- Initialize:

```
dp[1 << 0][0] = 0;  // Start from vertex 0 only
```

Step 4: Fill the DP Table

For all subsets of vertices (`mask` from 1 to $2n - 12^n - 12n - 1$):

- For each vertex `u` included in the subset (`mask & (1 << u)`):
  - Try going to every other vertex `v` not yet visited:
    - Compute the new cost:

```
newMask = mask | (1 << v);
dp[newMask][v] = min(dp[newMask][v], dp[mask][u] + cost[u][v]);
```

Step 5: Complete the Cycle

After all vertices are visited (`mask == 2^n - 1`), complete the cycle by returning to vertex 0:

```
result = min(dp[full_mask][i] + cost[i][0]) for all i ≠ 0
```

Step 6: Return the Minimum Cost

- If `result < INF`, return it as the minimum cost Hamiltonian circuit.
- If not, then no Hamiltonian cycle exists (graph is disconnected).

7.1.2.2.2) Code

```cpp
#include <iostream>
#include <vector>
#include <climits>
#include <cmath>
using namespace std;

const int INF = 1e9;

int tsp(int n, vector<vector<int>>& dist) {
    int N = 1 << n; // 2^n subsets
    vector<vector<int>> dp(N, vector<int>(n, INF));

    dp[1][0] = 0; // Start at node 0

    for (int mask = 1; mask < N; mask++) {
        for (int u = 0; u < n; u++) {
            if (mask & (1 << u)) {
                for (int v = 0; v < n; v++) {
                    if (!(mask & (1 << v))) {
                        int nextMask = mask | (1 << v);
                        dp[nextMask][v] = min(dp[nextMask][v], dp[mask]
                        [u] + dist[u][v]);
                    }
                }
            }
        }
    }

    // Complete the cycle: return to 0
    int res = INF;
    for (int i = 1; i < n; i++) {
        res = min(res, dp[N - 1][i] + dist[i][0]);
    }
 return res;
}
```

Listing 7.2: Hamiltonian Circuit Problem - Dynamic Programming (Held-Karp Algorithm)

7.1.2.2.3) Test Cases

- Case (1): Graph with Hamiltonian Cycle:

Input:

```cpp
int main() {
    vector<vector<int>> dist = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };
    int n = 4;
    cout << "Minimum cost Hamiltonian cycle: " << tsp(n, dist) << endl;
}
```

Output:

```
Output
Minimum cost Hamiltonian cycle: 80
```

Explanation: `Optimal path: 0 → 1 → 3 → 2 → 0`

`Cost: 10 + 25 + 30 + 15 = 80`

- Case (2): Graph Without Hamiltonian Cycle

Input (Disconnected graph):

```cpp
int main() {
    vector<vector<int>> dist = {
        {0, 10, INF, INF},
        {10, 0, 35, INF},
        {INF, 35, 0, 30},
        {INF, INF, 30, 0}
    };
    int n = 4;
    cout << "Minimum cost Hamiltonian cycle: " << tsp(n, dist) << endl;
}
```

Output:

```
Output
Minimum cost Hamiltonian cycle: 1000000000
```

This represents `INF`, meaning no valid cycle exists.

7.1.2.2.4) Complexity

Time Complexity: $O(n^2 \cdot 2n)$, where n is the number of vertices.

7.1.2.3) Greedy Heuristic (Nearest Neighbor)

7.1.2.3.1) Algorithm Steps

1. Start from a random node (e.g., node 0).
2. Mark it as visited.
3. At each step:
   - Go to the nearest unvisited neighbor.
4. Repeat until all nodes are visited.
5. Return to the starting node to complete the cycle

```cpp
#include <iostream>
#include <vector>
#include <limits>
using namespace std;

int findNearest(int current, const vector<vector<int>>& graph, const vector<bool>& visited) {
    int minDist = INT_MAX;
    int nextNode = -1;
    for (int i = 0; i < graph.size(); i++) {
        if (!visited[i] && graph[current][i] < minDist && graph[current][i] > 0) {
            minDist = graph[current][i];
            nextNode = i;
        }
    }
    return nextNode;
}

void greedyHamiltonian(const vector<vector<int>>& graph) {
    int n = graph.size();
    vector<bool> visited(n, false);
    vector<int> path;

    int current = 0;
    visited[current] = true;
    path.push_back(current);

```

```
    for (int i = 1; i < n; i++) {
        int next = findNearest(current, graph, visited);
        if (next == -1) {
            cout << "No Hamiltonian Cycle found (early termination)\n";
            return;
        }
        visited[next] = true;
        path.push_back(next);
        current = next;
    }

    // Check if we can return to start
    if (graph[current][path[0]] == 0) {
        cout << "No Hamiltonian Cycle (can't return to start)\n";
        return;
    }

    path.push_back(path[0]); // Complete the cycle

    // Print path
    cout << "Greedy Hamiltonian Cycle:\n";
    for (int node : path)
        cout << node << " ";
    cout << endl;
}
```

Listing 7.3: Hamiltonian Circuit Problem - Greedy Heuristic (Nearest Neighbor)

7.1.2.3.2) Test Cases

```
int main() {
    vector<vector<int>> graph = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    greedyHamiltonian(graph);
    return 0;
}
```

Expected Output:

Terminal

```
Greedy Hamiltonian Cycle:
0 1 3 2 0
```

7.1.2.3.3) Complexity

- Time Complexity: Roughly $O(n^2)$: for each of nnn steps, you check up to nnn neighbors.
- Space Complexity: $O(n)$ to track visited nodes and store path.

7.1.3) Algorithms Comparison

Algorithm	Time Complexity	Space Complexity	Notes
Backtracking (brute-force)	$O(n!)$	$O(n)$ (path stack)	Exact; explores all vertex permutations.
Held–Karp (DP for TSP)	$O(n^2 \cdot 2^n)$	$O(n \cdot 2^n)$	Exact; solves TSP/Hamiltonian in exponential time.
Heuristic (e.g. greedy/TSP)	$O(n^2)$	$O(n)$	Fast in practice; no optimality guarantee.

Table 7.1: Hamiltonian Circuit Problem - Algorithms Comparison

7.2) Partition Problem

7.2.1) Definition

The Partition Problem is a classic decision problem:

> **Problem Statement**
>
> Given a set of n positive integers, can it be partitioned into two subsets such that the sum of elements in both subsets is equal?

7.2.2) Examples

- Example (1)
  - Input: `{1, 5, 11, 5}`
  - Output: `True`
  - Explanation: Because: `1 + 5 + 5 = 11` and `11 = 11` : equal partition exists.
- Example (2)
  - Input: {1, 2, 3, 5}
  - Output: False
  - Explanation: Because no such equal-sum partition exists.

7.2.3) Algorithms

7.2.3.1) Dynamic Programming Algorithm

7.2.3.1.1) Algorithm Explanation

Reduce the problem to subset sum:

- Let `sum` be the total of all elements.
- If `sum` is odd, return `False` (cannot divide odd sum equally).
- If `sum` is even, check if there is a subset with sum = $\frac{\text{sum}}{2}$.

This becomes a subset-sum problem.

7.2.3.2) Algorithm Steps

1. Calculate total sum of the array.
2. If total sum is odd, return `false`.
3. Let `target = sum / 2`.
4. Use a boolean DP table:
   - `dp[i][j]` means: Can we get sum `j` using first `i` elements?
5. Initialize:
   - `dp[0][0] = true` (zero sum with zero elements).
   - `dp[i][0] = true` for all `i`.
   - `dp[0][j] = false` for all `j > 0`.
6. Fill DP table:
   - For each `i` from 1 to `n`:
     - For each `j` from 1 to target:

```cpp
dp[i][j] = dp[i-1][j] or dp[i-1][j - arr[i-1]] if j ≥ arr[i-1]
```

7. Final result: `dp[n][target]`

7.2.3.2.1) Code

```cpp
#include <iostream>
#include <vector>
using namespace std;

bool canPartition(vector<int>& nums) {
```

```
    int sum = 0;
    for (int num : nums)
        sum += num;

    if (sum % 2 != 0)
        return false;

    int target = sum / 2;
    int n = nums.size();

    vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

    for (int i = 0; i <= n; i++)
        dp[i][0] = true;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= target; j++) {
            if (j < nums[i - 1])
                dp[i][j] = dp[i - 1][j];
            else
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]];
        }
    }

    return dp[n][target];
}
```

Listing 7.4: Partition Problem - Dynamic Programming Algorithm

7.2.3.2.2) Test Cases

- Case (1): Partition Exists
  - Input:

```cpp
int main() {
  vector<int> nums = {1, 5, 11, 5};
  cout << (canPartition(nums) ? "True" : "False") << endl;
}
```

  - Output:

```
Output
True
```

- Case (2): Partition Does NOT Exist
  - Input:

```cpp
    int main() {
    vector<int> nums = {1, 2, 3, 5};
    cout << (canPartition(nums) ? "True" : "False") << endl;
}
```

- Output:

```
Output
False
```

7.2.3.2.3) Complexity

- Time Complexity: $O(n.\frac{\text{sum}}{2})$
- Space Complexity: $O(n.\frac{\text{sum}}{2})$

You can optimize to 1D space if needed using a 1D DP array

7.2.3.3) Recursive Backtracking

7.2.3.3.1) Algorithm Explanation

Try all subsets and check if there's a subset with sum equal to `totalSum / 2`.

7.2.3.3.2) Algorithm Steps

- Calculate the total sum of the array.
- If it's odd, return false.
- Use recursion to check if a subset with sum = `totalSum / 2` exists.

7.2.3.3.3) Code

```cpp
#include <iostream>
using namespace std;

bool isSubsetSum(int arr[], int n, int sum) {
    if (sum == 0) return true;
    if (n == 0) return false;

    if (arr[n-1] > sum)
        return isSubsetSum(arr, n-1, sum);

    return isSubsetSum(arr, n-1, sum) || isSubsetSum(arr, n-1, sum -
    arr[n-1]);
}

bool canPartition(int arr[], int n) {
    int sum = 0;
```

```
    for(int i = 0; i < n; i++) sum += arr[i];

    if (sum % 2 != 0) return false;

    return isSubsetSum(arr, n, sum / 2);
}
```

Listing 7.5: Partition Problem - Recursive Backtracking Algorithm

7.2.3.3.4) Test Cases

- Input:

```cpp
int main() {
  int arr1[] = {1, 5, 11, 5};
  int arr2[] = {1, 2, 3, 5};

  cout << "Test 1: " << (canPartition(arr1, 4) ? "True" : "False") << endl;
  cout << "Test 2: " << (canPartition(arr2, 4) ? "True" : "False") << endl;
}
```

- Output:

```
Output
Test 1: True
Test 2: False
```

7.2.3.3.5) Time Complexity

Exponential: $O(2^n)$, since all subsets are explored.

7.2.3.4) Greedy Heuristic: Karmarkar-Karp Algorithm

7.2.3.4.1) Algorithm Steps

- Place all numbers in a max heap.
- Repeatedly:
  - Pop two largest numbers `a` and `b`.
  - Push back `abs(a - b)`.
- If final number is `0`, the set can be partitioned.

Note: This approximates a solution — may fail for some exact cases.

7.2.3.4.2) Code

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

bool canPartitionKK(vector<int>& nums) {
    priority_queue<int> pq;
    for (int num : nums)
        pq.push(num);

    while (pq.size() > 1) {
        int a = pq.top(); pq.pop();
        int b = pq.top(); pq.pop();
        if (a != b)
            pq.push(abs(a - b));
    }

    return pq.empty() || pq.top() == 0;
}
```

Listing 7.6: Partition Problem - Greedy Heuristic: Karmarkar-Karp Algorithm

7.2.3.4.3) Test Cases

```cpp
int main() {
    vector<int> test1 = {1, 5, 11, 5};      // TRUE (likely correct)
    vector<int> test2 = {1, 2, 3, 5};       // FALSE
    vector<int> test3 = {10, 10, 10, 10};   // TRUE

    cout << canPartitionKK(test1) << endl;
    cout << canPartitionKK(test2) << endl;
    cout << canPartitionKK(test3) << endl;

    return 0;
}
```

7.2.3.4.4) Time Complexity:

$O(n \log(n))$, Fast for large input, not always exact.

7.2.4) Algorithms Comparison

Method	Accuracy	Time Complexity	Use Case
Dynamic Prog	Exact	$O(n \cdot \text{sum})$	Best for medium-size inputs
Backtracking	Exact	$O(2^n)$	Simple, but exponential
Greedy (KK)	Approx	$O(n \log(n))$	Fast for large input, not always exact

Table 7.2: Partition Problem - Algorithms Comparison

7.3) Graph Coloring Problem

7.3.1) Problem Statement

> **Problem Statement**
>
> Given a graph, assign colors to its vertices so that no two adjacent vertices share the same color, using the minimum number of colors possible.

This is known as the Chromatic Number problem — finding the minimum number of colors required to color the graph.

7.3.2) Applications

- Register allocation in compilers
- Map coloring
- Scheduling problems
- Frequency assignment

7.3.3) Algorithms

7.3.3.1) Backtracking (Exact)

7.3.3.1.1) Algorithm Steps

1. Start with the first vertex.
2. Try assigning each available color to it.
3. Check if the color assignment is valid (no neighbor has the same color).
4. Recur for the next vertex.
5. If all vertices are assigned a valid color → Success.
6. If no valid color → backtrack.

7.3.3.1.2) Code

```cpp
#include <iostream>
#include <vector>
using namespace std;

bool isSafe(int v, vector<vector<int>>& graph, vector<int>& color, int c) {
    for (int i = 0; i < graph.size(); i++)
        if (graph[v][i] && color[i] == c)
            return false;
    return true;
```

```
}

bool graphColoringUtil(vector<vector<int>>& graph, int m, vector<int>&
color, int v) {
    if (v == graph.size())
        return true;

    for (int c = 1; c ≤ m; c++) {
        if (isSafe(v, graph, color, c)) {
            color[v] = c;
            if (graphColoringUtil(graph, m, color, v + 1))
                return true;
            color[v] = 0;
        }
    }

    return false;
}

bool graphColoring(vector<vector<int>>& graph, int m) {
    vector<int> color(graph.size(), 0);
    return graphColoringUtil(graph, m, color, 0);
}
```

Listing 7.7: Graph Coloring Problem - Backtracking (Exact) Algorithm

7.3.3.1.3) Test Cases

- Input:

```
int main() {                                                    C++
    vector<vector<int>> graph = {
        {0, 1, 1, 1},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {1, 0, 1, 0}
    };

    int m = 3; // Try coloring with 3 colors
    cout << (graphColoring(graph, m) ? "Coloring possible\n" : "Not
    possible\n");

    return 0;
}
```

- Output:

Output

```
Coloring possible
```

7.3.3.1.4) Time Complexity

Time Complexity: $O(m^n)$

7.3.3.2) Greedy Coloring (Approximate)

7.3.3.2.1) Algorithm Steps

1. Sort vertices in some order (e.g., index or degree).
2. Assign the lowest possible color that is not used by neighbors.
3. Repeat for all vertices.

Note: Does not guarantee minimum number of colors.

7.3.3.2.2) Code

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void greedyColoring(vector<vector<int>>& graph) {
    int V = graph.size();
    vector<int> result(V, -1);
    result[0] = 0;

    vector<bool> available(V, false);

    for (int u = 1; u < V; u++) {
        fill(available.begin(), available.end(), false);

        for (int i = 0; i < V; i++) {
            if (graph[u][i] && result[i] != -1)
                available[result[i]] = true;
        }

        int cr;
        for (cr = 0; cr < V; cr++)
            if (!available[cr])
                break;

        result[u] = cr;
    }

```

```
    for (int u = 0; u < V; u++)
        cout << "Vertex " << u << " --->  Color " << result[u] << endl;
}
```

Listing 7.8: Graph Coloring Problem - Greedy Coloring (Approximate) Algorithm

7.3.3.2.3) Test Case

- Input

```cpp
int main() {
    vector<vector<int>> graph = {
        {0, 1, 1, 1},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {1, 0, 1, 0}
    };

    greedyColoring(graph);
    return 0;
}
```

- Output

```
Output
Vertex 0 --->  Color 0
Vertex 1 --->  Color 1
Vertex 2 --->  Color 2
Vertex 3 --->  Color 1
```

7.3.3.2.4) Time Complexity

Time Complexity: $O(v^2)$

7.3.3.3) DSatur Algorithm (Heuristic, Efficient)

7.3.3.3.1) Algorithm Steps

1. Initialize saturation degree of all vertices to 0.
2. Choose the vertex with highest saturation (number of differently colored neighbors).
3. Assign the smallest available color.
4. Update saturation of its neighbors.
5. Repeat until all vertices are colored.

7.3.3.3.2) Time Complexity

Time Complexity: $O(V^2.\log(V))$

7.3.4) Algorithms Comparison

Algorithm	Type	Optimal	Time Complexity	Notes
Backtracking	Exact	✓	$O(m^n)$	Exponential, for small graphs only
Greedy Coloring	Heuristic	❌	$O(V^2)$	Fast, not optimal
DSatur	Heuristic	❌	$O(V^2 \log(V))$	Performs well in practice

Table 7.3: Graph Coloring Problem - Algorithms Comparison

References

[1] “Hamiltonian Path problemWikipedia - Wikipedia.”. https://en.wikipedia.org/wiki/Hamiltonian_path_problemWikipedia

[2] Jones, K., “Answer to "Why Hamiltonian Cycle Decision Problem in NP-Complete?",” Jun 08 2018.. https://math.stackexchange.com/a/2811989

[3] “Karp's 21 NP-Complete Problems,” Mar 28 2025.

[4] “Set Partition Is NP Complete.”. https://www.geeksforgeeks.org/set-partition-is-np-complete/

[5] “Graph Coloring,” Apr 30 2025.

[6] Husfeldt, T., “Reference for NP-Hardness of 3-Colouring?,” Apr 25 2013.. https://cstheory.stackexchange.com/q/17396

[7] “Karp's 21 NP-Complete Problems,” Mar 28 2025.

[8] “Complexity Theory - How Can I Reduce Subset Sum to Partition? - Computer Science Stack Exchange.”. https://cs.stackexchange.com/questions/6111/how-can-i-reduce-subset-sum-to-partition

[9] “Contents: Tromino Puzzles by Norton Starr.”. https://nstarr.people.amherst.edu/puzzle.html

[10] “The Tromino Puzzle by Norton Starr.”. https://nstarr.people.amherst.edu/trom/intro.html#refs

[11] “Tromino,” Mar 16 2025.

[12] Robinson, E. A., “On the Table and the Chair,” *Indagationes Mathematicae*, Vol. 10, No. 4, 1999, pp. 581–599.. https://doi.org/10.1016/S0019-3577(00)87911-2

[13] “Golomb's Inductive Proof of a Tromino Theorem.”. https://www.cut-the-knot.org/Curriculum/Geometry/Tromino.shtml

[14] “Tromino Puzzle: Interactive Illustration of Golomb's Theorem.”. https://www.cut-the-knot.org/Curriculum/Games/TrominoPuzzle.shtml

[15] “Warnsdorff's Algorithm for Knight’s Tour Problem.”. https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/

[16] “Knight's Tour,” Apr 30 2025.

[17] “Knight's Tours on 3 by n Boards.”. https://www.mayhematics.com/t/oa.htm

[18] Simon, D., “Evolutionary Optimization Algorithms,” John Wiley & Sons, 2013.

[19] “Tower of Hanoi,” Apr 28 2025.

[20] Hofstadter, D. R., “Metamagical Themas : Questing for the Essence of Mind and Pattern,” New York : Basic Books, 1985.

[21] “\textit{The Tower of Hanoi – Myths and Maths},” Feb 18 2025.

[22] Lucas, E. A. d. t., “Récréations Mathématiques. 1. Les Traversées, Les Ponts, Les Labyrinthes... / Par Édouard Lucas,” 1891.

[23] “Miller's Mathematical Ideas, 9th Edition Web Site Chapter 4 – Internet Project,” Aug 21 2004.. https://web.archive.org/web/20040821062630/http://occawlonline.pearsoned.com/bookbind/pubbooks/miller2_awl/chapter4/essay1/deluxe-content.html#tower

[24] “Question Corner – Generalizing the Towers of Hanoi Problem.”. https://www.math.toronto.edu/mathnet/questionCorner/genhanoi.html

[25] Zhang, J., and Norman, D. A., “Representations in Distributed Cognitive Tasks.”

[26] UniPuzzle, “Six Knights Guarini's Problem Chess Puzzle.”. https://www.unipuzzle.com/chess-puzzles/chess.php?name=six-knights-guarinis-problem

[27] Iranpoor, M., “Knights Exchange Puzzle—Teaching the Efficiency of Modeling,” *INFORMS Transactions on Education*, Vol. 21, No. 2, 2021, pp. 108–114.. https://doi.org/10.1287/ited.2019.0235

[28] Weisstein, E. W., “Knight Graph.”. https://mathworld.wolfram.com/KnightGraph.html