CSE245 - Advanced Algorithms and Complexity

Spring 2025 – Final Project

Submitted to:

Dr. Gamal Abdel-Shafy

Eng. Sally Shaker

Mai Fakhry	23P0108
Mariam Riyadh	23P0147
Mohamed Morsi	23P0097
Omar Shaker	23P0188
Rana Tarek	23P0264

Table of Contents

Table of Figures

Credits

Mai Fakhry 23P0108	- Task 2: Knight's Tour + Visualization - Research Task 2: Partition Problem
Mariam Riyadh 23P0147	- Task 3: Tower of Hanoi - Research Task 3: Graph Coloring Problem
Mohamed Morsi 23P0097	- Task 1: Colored Tromino Tiling + Visualization - (1/2) Task 6: Crossing Dots
Omar Shaker 23P0188	- Task 4: Six Knights - Research Task 1: Hamiltonian Circuit
Rana Tarek 23P0264	- Task 5: Hitting a Moving Target - (1/2) Task 6: Crossing Dots

Task 1: Colored Tromino Tiling

Definition

Devise an algorithm for the following task: given a $2^n \times 2^n$ ($n > 1$) board with one missing square, tile it with right trominoes of only three colors so that no pair of trominoes that share an edge have the same color. Recall that the right tromino is an L-shaped tile formed by three adjacent squares. Use Divide and conquer technique to solve this problem.

Assumptions

- Input n is greater than one
- Input for missing cell is 1-based indexing
- Board is initially empty except for one missing square
- Any right tromino can take one of four different orientations
- Adjacency is only checked in four directions (up, down, left and right)

Explanation and Pseudo-Code

The problem we are addressing involves tiling a square board of size 2^n x 2^n, where $n > 1$, using L-shaped tiles known as right trominoes, under the constraint that exactly one square on the board is missing. The objective is to completely tile the rest of the board using these trominoes such that no two adjacent trominoes (i.e., those sharing an edge) are assigned the same color, using only three distinct colors. The right tromino is a polyomino consisting of three squares connected in an L-shape. Visually, it resembles the shape of the letter "L", covering two adjacent cells in a row and one additional cell in the next row (or vice versa depending on rotation).

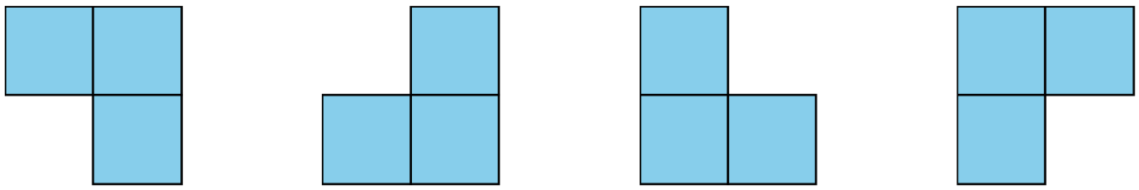


Figure 1: Right tromino orientations (Left T, Mirrored L-shape, L-shape and Right T)

This problem has roots in classic algorithm design and is notably a variant of the defective chessboard problem, which was discussed by Donald E. Knuth in *The Art of Computer Programming, Volume 1* (Knuth, 1997). The problem is commonly used to illustrate the divide and conquer paradigm, a method where a problem is divided into smaller subproblems of the same type, solved recursively, and then the solutions are combined to solve the original problem.

The insight that makes this problem tractable using divide and conquer lies in the regularity and symmetry of the board size. Since the board is of size $2^n \times 2^n$, it can always be evenly divided into four smaller quadrants of equal size. Each time we divide the board into four parts, one of the quadrants contains the missing cell. By cleverly placing a single tromino at the center—covering one square in each of the remaining three quadrants—we reduce the problem to four subproblems of smaller size, each now with its own missing square. This recursive structure continues until the base case is reached: a 1 x 1 board which requires no further action.

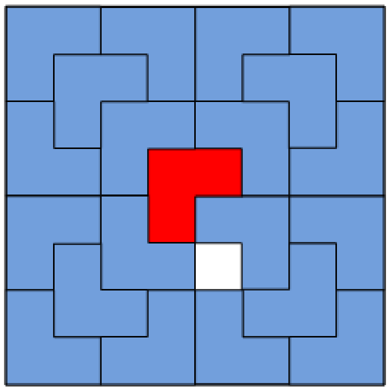


Figure 2: Initial overview of the tromino placement in a 8x8 grid

```
tromino_id = 0 // unique identifier for each placed tromino
```

To manage the identity of each tromino we place on the board, the algorithm introduces a global variable called tromino_id. This variable is initially set to zero and is incremented every time a new tromino is placed. Its role is to ensure that each tromino has a unique identifier that can be tracked independently across the entire board. This is crucial because, in later stages of the algorithm, particularly during the coloring phase, we need to differentiate between adjacent trominoes to avoid assigning the same color to two touching pieces. The uniqueness of the tromino IDs effectively creates a mapping from cells on the board to the tromino they belong to, which simplifies the task of building an adjacency graph and applying graph coloring techniques later on.

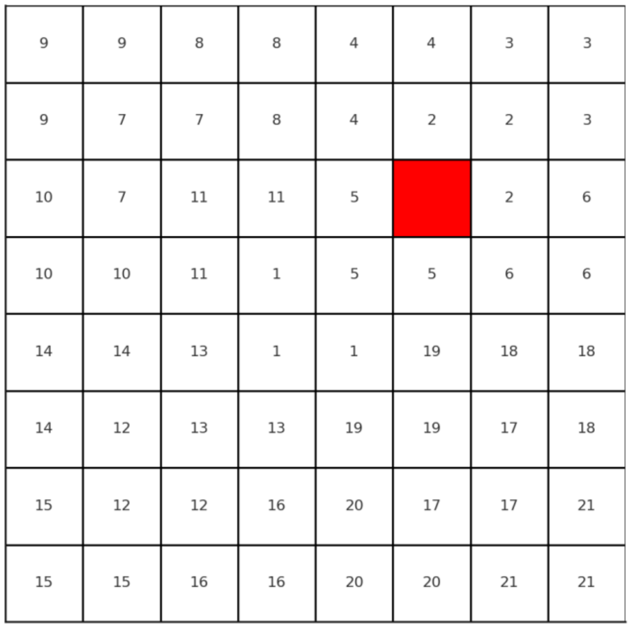
```
function locateTromino(board, size, istart, iend, jstart, jend, missing_i, missing_j):
    if size == 1:
        return

    determine which quadrant (quad 1 to 4) the missing cell is in

    increment global tromino_id

    based on the quadrant:
        place a tromino at the center of the current board
        (cover the other 3 quadrants' center cells)

    recursively call locateTromino on all 4 sub-quadrants:
        - for the original missing quadrant, pass the actual missing cell
        - for the other 3 quadrants, set a new "missing" at the position covered by the tromino
```

The core of the divide and conquer approach lies in the function locateTromino, which recursively places trominoes on sub-regions of the board. This function accepts as input the current board, the size of the sub-board being worked on, the starting and ending indices of rows and columns for the region, and the coordinates of the missing square within that region.

Upon each call, the function first checks for the base case: if the size of the sub-board is 1, it returns immediately because no tromino can be placed on a 1 x 1 board. Otherwise, the board is divided conceptually into four equal quadrants: the top-left, top-right, bottom-left, and bottom-right. The quadrant that contains the missing square is determined by comparing the coordinates of the missing cell to the midpoints of the row and column ranges.

Once the missing quadrant is identified, the function increments tromino_id and places a tromino in the center of the board such that it covers one square in each of the three remaining quadrants. For example, if the missing square is in the top-right quadrant, then the placed tromino will cover one square each in the top-left, bottom-left, and bottom-right quadrants—specifically the squares closest to the center. This strategically placed tromino creates new "missing" squares in the three other quadrants, enabling the recursion to proceed.

The function then makes four recursive calls, one for each quadrant. The quadrant containing the original missing square is passed down with the same missing coordinates. For the other three quadrants, the newly placed tromino has created a missing square at the center, and these new coordinates are passed instead. This systematic process guarantees that every quadrant always has exactly one missing square, thus maintaining the problem's invariant at every recursive level.



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

Figure 3: The result of the first function: a successfully tiled grid with unique tromino IDs

After the entire board has been successfully tiled using uniquely identified trominoes, the next task is to color them using only three colors in such a way that no two adjacent trominoes share the same color.

```
function colorBoard(board, size):
    // STEP 1: Build adjacency graph
    create an empty adjacency map: tromino_id → set of adjacent tromino_ids

    for each cell in the board:
        if cell has valid tromino_id:
            for each neighbor (up, down, left, right):
                if neighbor has a different tromino_id:
                    add mutual adjacency in the map

    // STEP 2: Greedy Coloring
    create empty map tromino_color

    for each tromino_id in adjacency:
        create boolean array used[3] initialized to false

        for each neighbor:
            if neighbor has a color assigned:
                mark used[color] = true

        assign the first available color (0, 1, or 2) to current tromino_id

    // STEP 3: Apply Colors to Board
    for each cell in the board:
        if cell has a valid tromino_id:
            overwrite board[i][j] with tromino_color[tromino_id] + 1
```

This requirement can be reframed as a graph coloring problem, where each tromino represents a node, and an edge exists between two nodes if the corresponding trominoes share at least one side on the board. The goal is then to color the nodes of this graph such that no two connected nodes have the sae color, using the minimum number of colors—in this case, restricted to three.

The coloring process begins with the construction of the adjacency graph. For every cell on the board, the algorithm examines its four immediate neighbors—up, down, left, and right. If any of these neighbors belong to a different tromino (i.e., their tromino_id is different), an undirected edge is added between the two IDs in an adjacency map. This is implemented using a hash map where the keys are tromino IDs and the values are sets containing the IDs of all adjacent trominoes. This ensures that each connection is recorded only once, preventing redundant computation in later steps.

With the adjacency graph constructed, the algorithm proceeds to apply a greedy coloring strategy. For each tromino ID, it examines the colors assigned to its adjacent trominoes and marks them as unavailable. It then selects the first available color (from the options 0, 1, or 2) that is not used by its neighbors and assigns it to the current tromino. This method ensures that no two adjacent trominoes receive the same color.

The use of only three colors is theoretically justified by results from graph theory. Specifically, in a planar tiling where each tile (tromino) touches only a limited number of neighbors (in this case, at most four), Brooks' Theorem (Brooks, 1941) tells us that the chromatic number of the graph is at most equal to the maximum degree of any vertex, unless the graph is a complete graph or an odd cycle. In this setting, our adjacency graph is neither of those special cases, and hence three colors suffice.

Finally, once the colors are determined, the function makes a final pass over the board. Each cell that was marked with a tromino_id is overwritten with the corresponding color code assigned to that ID. To convert from 0-based color codes to a more human-readable form, the colors are incremented by 1, resulting in final color values of 1, 2, or 3.

The output of this function is a completely tiled and colored board where no two adjacent trominoes are of the same color, satisfying all constraints of the original problem.

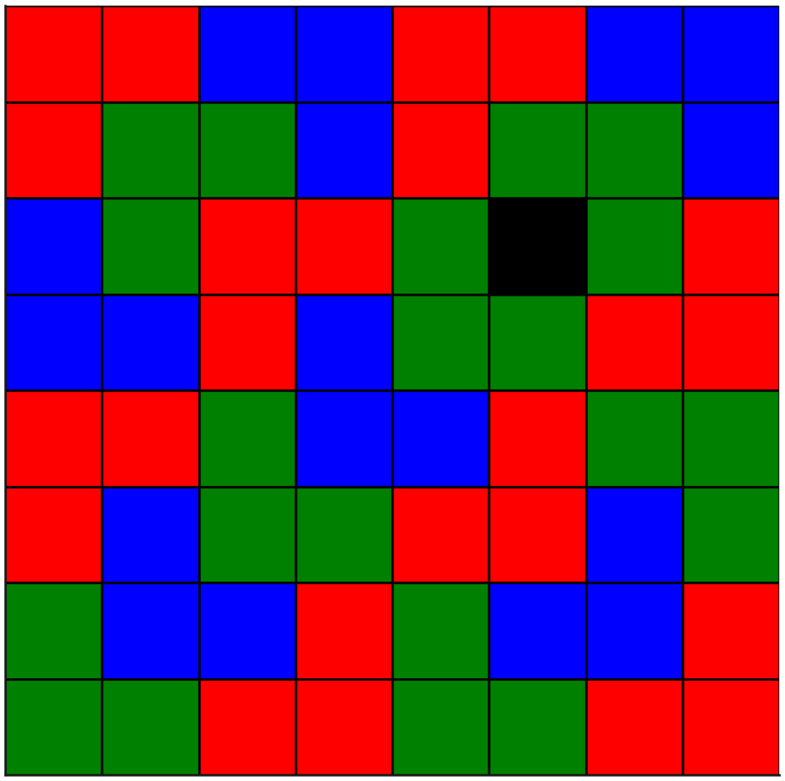


Figure 4: The result of the second function: a feasible coloring combination for the trominoes (1 is mapped to red, 2 to green and 3 to blue)

C++ Code

```
/*
Computer Engineering and Artificial Intelligence Program
CSE245: Advanced Algorithms and Complexity – Spring 2025
Task 1 - Tiling Trominoes
*/

#include <bits/stdc++.h>
using namespace std;

int tromino_id = 0;

// The main recursive function to place trominoes
void locateTromino(vector<vector<int>> &board, int size, int istart, int iend, int jstart, int jend,
int missing_i, int missing_j){

    // Base case (reached a single cell)
    if(size == 1) return;

    // Determine quad of missing cell
    int quad;
    if(missing_i >= istart && missing_i <= (istart+iend)/2){
        if(missing_j >= ((jstart+jend)/2)+1 && missing_j <= jend) quad = 1;
        else quad = 2;
    }
    else if(missing_i >= ((istart+iend)/2)+1 && missing_i <= iend){
        if(missing_j >= jstart && missing_j <= (jstart+jend)/2) quad = 3;
        else quad = 4;
    }

    // Increment tromino ID (to place a different one)
    tromino_id++;

    // Determine tromino orientation based on quad of missing cell
    switch(quad){
        case 1:
            // 1st quad: L shape
            board[(istart+iend)/2][(jstart+jend)/2] = tromino_id; // Top left
            board[((istart+iend)/2)+1][(jstart+jend)/2] = tromino_id; // Bottom left
            board[((istart+iend)/2)+1][((jstart+jend)/2)+1] = tromino_id; // Bottom right

            // Quadruple recursion
            locateTromino(board, size/2, istart, istart+(size/2)-1, jstart+(size/2), jstart+(size-
1), missing_i, missing_j); // 1ST QUAD
            locateTromino(board, size/2, istart, istart+(size/2)-1, jstart, jstart+(size/2)-1,
istart+(size/2)-1, jstart+(size/2)-1); // 2nd quad
            locateTromino(board, size/2, istart+(size/2), istart+(size-1), jstart, jstart+(size/2)-
1, istart+(size/2), jstart+(size/2)-1); // 3rd quad
            locateTromino(board, size/2, istart+(size/2), istart+size-1, jstart+(size/2),
jstart+size-1, istart+(size/2), jstart+(size/2)); // 4th quad
        break;

        case 2:
            // 2nd quad: Mirrored L shape
            board[(istart+iend)/2][((jstart+jend)/2)+1] = tromino_id; // Top right
            board[((istart+iend)/2)+1][(jstart+jend)/2] = tromino_id; // Bottom left
            board[((istart+iend)/2)+1][((jstart+jend)/2)+1] = tromino_id; // Bottom right

            // Quadruple recursion
            locateTromino(board, size/2, istart, istart+(size/2)-1, jstart+(size/2), jstart+(size-
1), istart+(size/2)-1, jstart+(size/2)); // 1st quad
            locateTromino(board, size/2, istart, istart+(size/2)-1, jstart, jstart+(size/2)-1,
missing_i, missing_j); // 2ND QUAD
```

```
            locateTromino(board, size/2, istart+(size/2), istart+(size-1), jstart, jstart+(size/2)-
1, istart+(size/2), jstart+(size/2)-1); // 3rd quad
            locateTromino(board, size/2, istart+(size/2), istart+size-1, jstart+(size/2),
jstart+size-1, istart+(size/2), jstart+(size/2)); // 4th quad
        break;

        case 3:
            // 3rd quad: Left T
            board[(istart+iend)/2][((jstart+jend)/2)+1] = tromino_id; // Top right
            board[(istart+iend)/2][(jstart+jend)/2] = tromino_id; // Top left
            board[((istart+iend)/2)+1][((jstart+jend)/2)+1] = tromino_id; // Bottom right

            // Quadruple recursion
            locateTromino(board, size/2, istart, istart+(size/2)-1, jstart+(size/2), jstart+(size-
1), istart+(size/2)-1, jstart+(size/2)); // 1st quad
            locateTromino(board, size/2, istart, istart+(size/2)-1, jstart, jstart+(size/2)-1,
istart+(size/2)-1, jstart+(size/2)-1); // 2nd quad
            locateTromino(board, size/2, istart+(size/2), istart+(size-1), jstart, jstart+(size/2)-
1, missing_i, missing_j); // 3RD QUAD
            locateTromino(board, size/2, istart+(size/2), istart+size-1, jstart+(size/2),
jstart+size-1, istart+(size/2), jstart+(size/2)); // 4th quad
        break;

        case 4:
            // 4th quad: Right T
            board[(istart+iend)/2][(jstart+jend)/2] = tromino_id; // Top left
            board[(istart+iend)/2][((jstart+jend)/2)+1] = tromino_id; // Top right
            board[((istart+iend)/2)+1][(jstart+jend)/2] = tromino_id; // Bottom left

            // Quadruple recursion
            locateTromino(board, size/2, istart, istart+(size/2)-1, jstart+(size/2), jstart+(size-
1), istart+(size/2)-1, jstart+(size/2)); // 1st quad
            locateTromino(board, size/2, istart, istart+(size/2)-1, jstart, jstart+(size/2)-1,
istart+(size/2)-1, jstart+(size/2)-1); // 2nd quad
            locateTromino(board, size/2, istart+(size/2), istart+(size-1), jstart, jstart+(size/2)-
1, istart+(size/2), jstart+(size/2)-1); // 3rd quad
            locateTromino(board, size/2, istart+(size/2), istart+size-1, jstart+(size/2),
jstart+size-1, missing_i, missing_j); // 4TH QUAD
        break;
    }
}

// To display board in the console
void displayBoard(vector<vector<int>> &board, int n){
    for(int i = 0; i < pow(2, n); i++){
        for(int j = 0; j < pow(2, n); j++){
            cout << setw(3) << board[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}
```

```
// Coloring the board after filling the board with trominoes
void colorBoard(vector<vector<int>> &board, int size){

    // STEP 1: ADJACENCY GRAPH
    // Mapping each node ID to its adjacent node IDs (used a set to prevent repetition)
    unordered_map<int, unordered_set<int>> adjacency;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int current = board[i][j];
            if (current <= 0) continue; // Ignore missing cell
            // To check all adjacent cells of a single cell (left, right, down, up)
            vector<pair<int, int>> directions = {{-1,0}, {1,0}, {0,-1}, {0,1}};
            for (auto [di, dj] : directions) {
                int xi = i + di, xj = j + dj;
                if (xi >= 0 && xi < size && xj >= 0 && xj < size) {
                    int neighbor = board[xi][xj];
                    // Ignore the neighbor if its missing OR has the same ID as the current cell
                    if (neighbor > 0 && neighbor != current) {
                        // Add adjacency to both nodes: current and neighbor
                        adjacency[current].insert(neighbor);
                        adjacency[neighbor].insert(current);
                    }
                }
            }
        }
    }

    // STEP 2: GREEDY COLORING
    // To track each tromino ID and its color
    unordered_map<int, int> tromino_color;
    for (auto &entry : adjacency) {
        int tromino_id = entry.first;
        const unordered_set<int> &neighbors = entry.second;

        // Track used colors by neighbors for EACH entry
        bool used[3] = {false, false, false};

        for (int neighbor : neighbors) {
            if (tromino_color.count(neighbor)) { // Returns 1 if it has been colored before, 0 if
not
                used[tromino_color[neighbor]] = true; // Mark it as used to prevent the current
tromino_id to be the same color
            }
        }

        // Assign the first available color (GREEDY COLORING)
        for (int c = 0; c < 3; c++) { // 0, 1, 2 to navigate used array and also colors
            if (!used[c]) {
                tromino_color[tromino_id] = c;
                break;
            }
        }
    }

    // STEP 3: OVERWRITE EXISTING IDS WITH COLORS
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (board[i][j] > 0) { // If not missing
                board[i][j] = tromino_color[board[i][j]] + 1; // Shifted all colors by 1 to make
them 1, 2, 3
            }
        }
    }
}
```

```
int main() {

    int n;
    cout << "n: ";
    cin >> n;

    int missing_row, missing_col;
    cout << "Row of missing cell: ";
    cin >> missing_row;
    cout << "Column of missing cell: ";
    cin >> missing_col;

    int size = pow(2, n);
    // Initialize board
    vector<vector<int>> board(size, vector<int>(size, 0));
    // Remove missing cell
    board[missing_row-1][missing_col-1] = -1;

    // Display board
    displayBoard(board, n);

    // Start the divide and conquer algorithm
    locateTromino(board, size, 0, size-1, 0, size-1, missing_row-1, missing_col-1);
    // Color the board after tromino placement
    colorBoard(board, size);

    // Display board
    displayBoard(board, n);

    return 0;
}
```

The program begins by declaring a global variable named tromino_id and initializing it to zero. This variable is essential to the tiling process as it uniquely labels each L-shaped tromino placed on the board. The uniqueness is important for identifying which group of three connected cells belong together and later for distinguishing each tromino during the coloring process.

The first core function in the program is locateTromino, which performs the recursive tiling. It takes in a reference to the board (a two-dimensional vector), the size of the current sub-board, and the starting and ending row and column indices for that region. It also receives the coordinates of the missing cell within that sub-region. The function first checks whether the current region is a single cell in size. If so, it returns immediately, as no tromino can be placed in a 1x1 space. This serves as the base case for the recursive process.

Next, the function determines which quadrant of the current board region contains the missing square. The quadrants are calculated by finding the midpoint of the row and column ranges. If the missing square falls in the top half and right side, it is considered to be in quadrant 1. If it is in the top-left, it is in quadrant 2. If it is in the bottom-left, it falls into quadrant 3, and if it is in the bottom-right, it is in quadrant 4. This quadrant detection is key to placing the L-shaped tromino in the correct orientation such that it avoids the real missing cell while simulating a new "missing" cell in the other three quadrants.

After identifying the quadrant with the real missing square, the program increments the tromino ID to assign a new, unique label to the tromino that is about to be placed. It then places the tromino in the center of the board, such that it covers one square from each of the three quadrants that do not contain the actual missing cell. The placement is done manually by referencing the midpoint rows and columns of the current region and assigning the current tromino_id to three specific cells forming the L-shape.

Following this placement, the function calls itself recursively on each of the four quadrants. The quadrant that originally contained the missing square is passed the original coordinates of the missing cell, ensuring consistency in the recursive breakdown. The other three quadrants are each passed the coordinates of the square just filled in by the center tromino, effectively treating them as the new missing cells. Each recursive call is made on a region of half the size, and this process continues until all subregions are reduced to size one, at which point recursion terminates.

After the board has been completely filled with tromino IDs, the function colorBoard is called to assign one of three colors to each distinct tromino. This function begins by building an adjacency graph that maps each tromino ID to a set of neighboring IDs. This is done by iterating through every cell on the board. For each cell, if its value is greater than zero, the function checks its four immediate neighbors: up, down, left, and right. If any neighbor is within bounds and has a different tromino ID, the function adds an edge between the current cell's ID and the neighbor's ID in an adjacency list represented by an unordered map of sets. This adjacency map ensures that we can later assign colors such that no two adjacent trominoes are the same.

Once the adjacency graph is complete, the function proceeds with the coloring phase using a greedy algorithm. It iterates over each entry in the adjacency list, representing a tromino and its set of neighbors. For each tromino, the function initializes a boolean array of size three to keep track of which colors are already used by its neighbors. It then loops through the set of adjacent trominoes and, if any of them have already been assigned a color, it marks that color as used. After this check, the function selects the first unused color and assigns it to the current tromino. Because each tromino is adjacent to at most four others and only three colors are needed, the greedy approach guarantees that no two connected trominoes end up sharing the same color.

Following the coloring, the function makes one last pass through the board. For each cell with a positive value (indicating that it belongs to a tromino), it replaces the tromino ID with its assigned color plus one. The color values originally ranged from 0 to 2, so adding one shifts them to 1, 2, and 3, making them more readable for output. The missing cell, which was marked with -1, remains unchanged.

The main function of the program acts as the entry point. It begins by prompting the user to enter the value of n. It then asks the user for the row and column of the missing square. These indices are adjusted from 1-based to 0-based to match the internal representation used in the board vector. A two-dimensional vector is created and initialized to all zeroes, and the missing cell is marked with -1. The board is then displayed to show the initial state. After that, the recursive tiling function is invoked to fill the board with trominoes, and the coloring function is applied to assign colors to each tromino. Finally, the board is displayed again, now showing the completed and colored result. The output confirms that the board has been tiled correctly and that the color constraints are met.

Time Complexity Analysis

The locateTromino function is a recursive divide-and-conquer algorithm responsible for tiling the entire board with L-shaped trominoes. In each invocation, the function first checks whether the size of the current sub-board is 1. If so, it returns immediately as no tromino can be placed in such a space. Otherwise, it determines which quadrant contains the missing square and places one L-shaped tromino in the center of the current sub-board to balance the problem. This placement covers three cells and takes constant time. Following that, the function recursively calls itself four times, each on a quadrant of size exactly half the current side length. Since each level of recursion divides the board into four smaller sub-boards and performs only constant-time work at that level (placing one tromino), the time complexity follows the recurrence relation

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

where n is the length of one side of the board. By solving this recurrence using the Master Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$
$$such\ that\ a = 4, b = 2\ and\ d = 1$$

we find that the total time complexity of the locateTromino function is

$$O\left(\mathrm{n}^{\log_2 4}\right) = O(n^2)$$

which corresponds to the number of cells on the board. In the worst-case scenario, the function recursively visits and processes every section of the board until all $n^2 - 1$ cells (excluding the missing one) are tiled, resulting in a total of $(n^2 - 1)/3$ trominoes being placed, confirming the quadratic time.

The colorBoard function is responsible for coloring all the trominoes on the board such that no two adjacent trominoes share the same color, using only three distinct colors. This function consists of three major phases, each of which runs in worst-case linear time with respect to the number of cells on the board.

In the first phase, it builds an adjacency graph by scanning every cell on the board. For each of the $n \times n$ cells, it checks up to four neighbors to detect connections between different trominoes. Since each neighbor check and graph insertion operation is constant time, this phase takes time proportional to $O(n^2)$.

In the second phase, the function performs greedy coloring on each unique tromino. The total number of trominoes is also proportional to $O(n^2)$, and each tromino can be adjacent to at most four others. For each tromino, the function examines its neighbors and selects one of the three available colors that hasn't been used, all in constant time. This results in another $O(n^2)$ step.

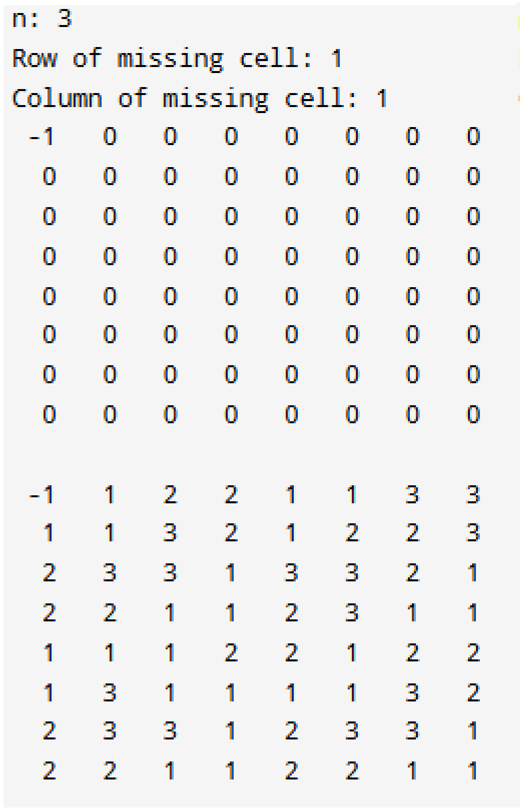
Finally, in the third phase, the function updates the board by replacing every cell's tromino ID with its assigned color. This final sweep again processes all n^2 cells, performing a constant-time operation per cell. As all three phases individually run in $O(n^2)$ time, the overall worst-case time complexity of the colorBoard function is $O(n^2)$.

Since both the tiling and coloring functions independently run in $O(n^2)$ time, the overall worst-case time complexity of the entire program:

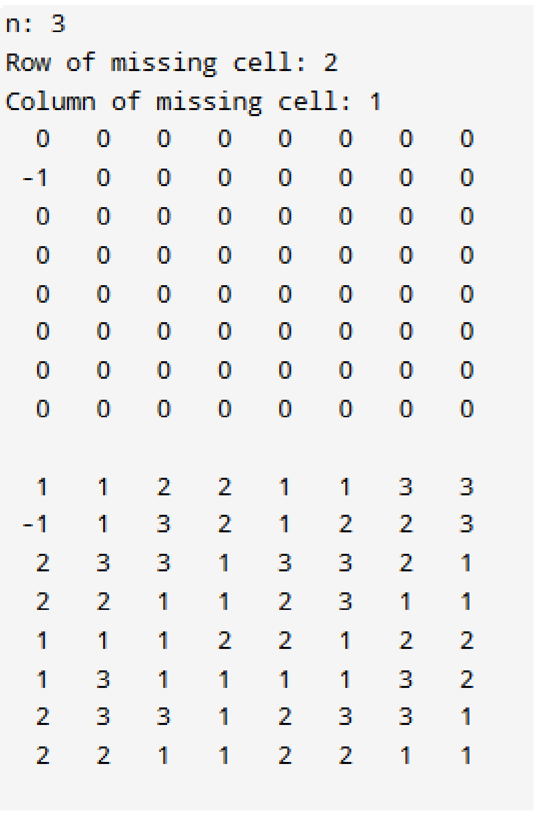
$$O(n^2)$$

Output

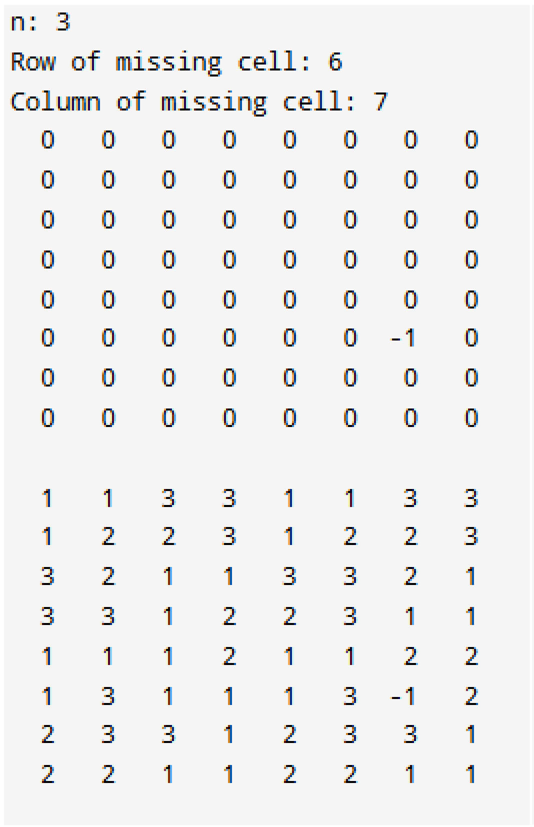
```
n: 4
Row of missing cell: 7
Column of missing cell: 7
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0  -1   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0

   1   1   3   3   1   1   2   2   1   1   2   2   1   1   2   2
   1   2   2   3   1   3   3   2   1   3   3   2   1   3   3   2
   3   2   1   1   2   2   3   1   2   3   1   1   2   2   3   1
   3   3   1   1   1   2   1   1   2   2   1   3   3   2   1   1
   1   1   2   1   3   3   2   2   1   1   2   2   3   1   2   2
   1   3   2   2   3   1   1   2   1   3   3   2   1   1   3   2
   2   3   3   1   2   1  -1   1   2   2   3   1   2   3   3   1
   2   2   1   1   2   2   1   1   3   2   1   1   2   2   1   1
   1   1   2   2   1   1   2   3   3   1   2   2   1   1   2   2
   1   3   3   2   1   3   2   2   1   1   3   2   1   3   3   2
   2   3   1   1   2   3   3   1   2   3   3   1   2   2   3   1
   2   2   1   3   2   2   1   1   2   2   1   1   3   2   1   1
   1   1   2   3   3   1   2   2   1   1   2   3   3   1   2   2
   1   3   2   2   1   1   3   2   1   3   2   2   1   1   3   2
   2   3   3   1   2   3   3   1   2   3   3   1   2   3   3   1
   2   2   1   1   2   2   1   1   2   2   1   1   2   2   1   1
```



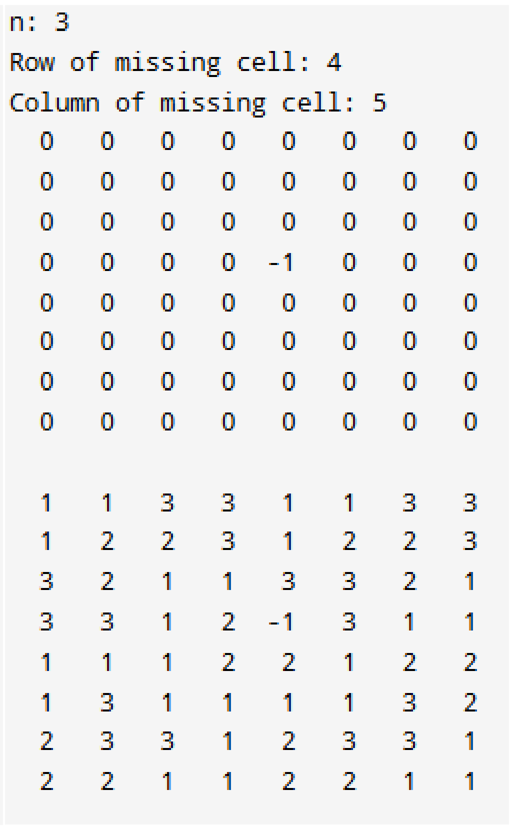
```
n: 3
Row of missing cell: 1
Column of missing cell: 1
  -1   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0

  -1   1   2   2   1   1   3   3
   1   1   3   2   1   2   2   3
   2   3   3   1   3   3   2   1
   2   2   1   1   2   3   1   1
   1   1   1   2   2   1   2   2
   1   3   1   1   1   1   3   2
   2   3   3   1   2   3   3   1
   2   2   1   1   2   2   1   1
```



```
n: 3
Row of missing cell: 2
Column of missing cell: 1
   0   0   0   0   0   0   0   0
  -1   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0

   1   1   2   2   1   1   3   3
  -1   1   3   2   1   2   2   3
   2   3   3   1   3   3   2   1
   2   2   1   1   2   3   1   1
   1   1   1   2   2   1   2   2
   1   3   1   1   1   1   3   2
   2   3   3   1   2   3   3   1
   2   2   1   1   2   2   1   1
```



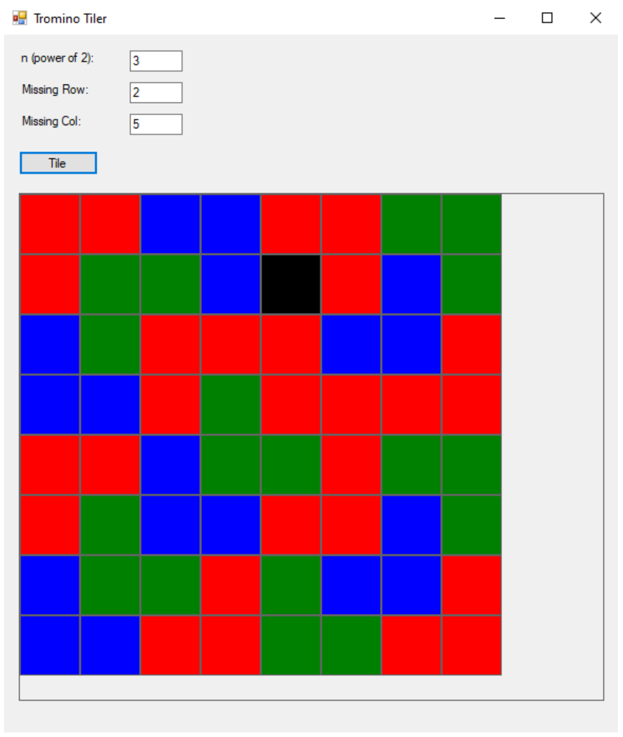
```
n: 3
Row of missing cell: 6
Column of missing cell: 7
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0  -1   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0

   1   1   3   3   1   1   3   3
   1   2   2   3   1   2   2   3
   3   2   1   1   3   3   2   1
   3   3   1   2   2   3   1   1
   1   1   1   2   1   1   2   2
   1   3   1   1   1   3  -1   2
   2   3   3   1   2   3   3   1
   2   2   1   1   2   2   1   1
```



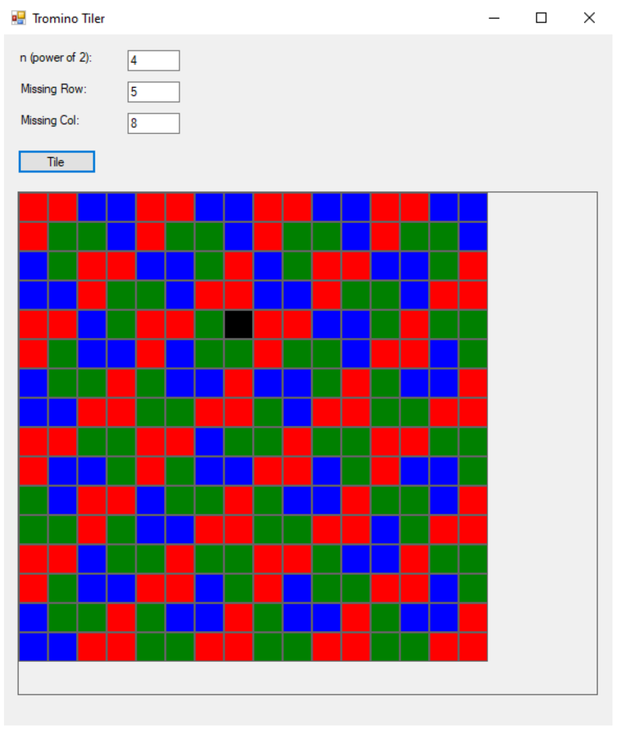
```
n: 3
Row of missing cell: 4
Column of missing cell: 5
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0  -1   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0

   1   1   3   3   1   1   3   3
   1   2   2   3   1   2   2   3
   3   2   1   1   3   3   2   1
   3   3   1   2  -1   3   1   1
   1   1   1   2   2   1   2   2
   1   3   1   1   1   1   3   2
   2   3   3   1   2   3   3   1
   2   2   1   1   2   2   1   1
```

Visualization using C#

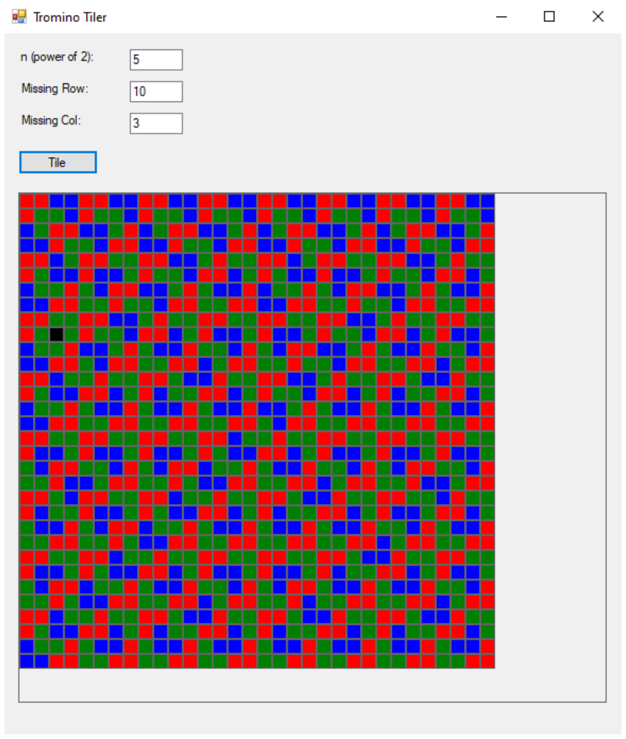
```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;

namespace TrominoTiler
{
    public class TrominoForm : Form
    {
        TextBox txtN, txtRow, txtCol;
        Button btnTile;
        Panel boardPanel;

        int[,] board;
        int trominoId;
        int size;
        readonly Color[] trominoColors = { Color.Red, Color.Green, Color.Blue };

        public TrominoForm()
        {
            Text = "Tromino Tiler";
            Width = 600; Height = 700;

            var lblN = new Label { Text = "n (power of 2):", AutoSize = true, Top = 15, Left = 15 };
            txtN = new TextBox { Top = lblN.Top, Left = 120, Width = 50, Text = "3" };
            var lblR = new Label { Text = "Missing Row:", AutoSize = true, Top = 45, Left = 15 };
            txtRow = new TextBox { Top = lblR.Top, Left = 120, Width = 50, Text = "1" };
            var lblC = new Label { Text = "Missing Col:", AutoSize = true, Top = 75, Left = 15 };
            txtCol = new TextBox { Top = lblC.Top, Left = 120, Width = 50, Text = "1" };

            btnTile = new Button { Text = "Tile", Top = 110, Left = 15 };
            btnTile.Click += BtnTile_Click;

            boardPanel = new Panel
            {
                Top = 150,
                Left = 15,
                AutoScroll = true,
                BorderStyle = BorderStyle.FixedSingle,
                Width = ClientSize.Width - 30,
                Height = ClientSize.Height - 180,
                Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right
            };

            Controls.AddRange(new Control[] { lblN, txtN, lblR, txtRow, lblC, txtCol, btnTile, boardPanel
});
        }

        private void BtnTile_Click(object sender, EventArgs e)
        {
            if (!int.TryParse(txtN.Text, out int n) || n < 1)
            {
                MessageBox.Show("Enter a valid integer n ≥ 1");
                return;
            }
            size = 1 << n;  // 2^n
            if (!int.TryParse(txtRow.Text, out int mr) ||
                !int.TryParse(txtCol.Text, out int mc) ||
                mr < 1 || mr > size || mc < 1 || mc > size)
            {
                MessageBox.Show($"Row/Col must be between 1 and {size}");
                return;
            }

            // Initialize
            board = new int[size, size];
            trominoId = 0;
            // Mark missing cell as -1
            board[mr - 1, mc - 1] = -1;

            // Tile
            LocateTromino(0, size - 1, 0, size - 1, mr - 1, mc - 1);

            // Color
            ColorBoard();

            // Display
            BuildBoardUI();
        }

        // Recursive tiling
        private void LocateTromino(int r0, int r1, int c0, int c1, int mr, int mc)
        {
            int s = r1 - r0 + 1;
            if (s == 1) return;

            int rm = (r0 + r1) / 2, cm = (c0 + c1) / 2;
            int quad;
            if (mr <= rm)
                quad = (mc > cm) ? 1 : 2;
            else
                quad = (mc <= cm) ? 3 : 4;

            trominoId++;
            // place central L
            // each case covers the 3 squares not in the missing quadrant
            if (quad == 1)
            {
                board[rm, cm] = trominoId;
                board[rm + 1, cm] = trominoId;
                board[rm + 1, cm + 1] = trominoId;
            }
            else if (quad == 2)
            {
                board[rm, cm + 1] = trominoId;
                board[rm + 1, cm] = trominoId;
                board[rm + 1, cm + 1] = trominoId;
            }
            else if (quad == 3)
            {
                board[rm, cm] = trominoId;
                board[rm, cm + 1] = trominoId;
                board[rm + 1, cm + 1] = trominoId;
            }
            else // quad 4
            {
                board[rm, cm] = trominoId;
                board[rm, cm + 1] = trominoId;
                board[rm + 1, cm] = trominoId;
            }

            // Recurse into 4 quadrants, supplying each one's "missing" cell
            // Top-right
            LocateTromino(r0, rm, cm + 1, c1,
                quad == 1 ? mr : rm,
                quad == 1 ? mc : cm + 1);

            // Top-left
            LocateTromino(r0, rm, c0, cm,
                quad == 2 ? mr : rm,
                quad == 2 ? mc : cm);

            // Bottom-left
            LocateTromino(rm + 1, r1, c0, cm,
                quad == 3 ? mr : rm + 1,
                quad == 3 ? mc : cm);

            // Bottom-right
            LocateTromino(rm + 1, r1, cm + 1, c1,
                quad == 4 ? mr : rm + 1,
                quad == 4 ? mc : cm + 1);
        }

        // Build adjacency and greedy color
        private void ColorBoard()
        {
            // STEP 1: Build adjacency graph
            var adj = new Dictionary<int, HashSet<int>>();
            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
                    int id = board[i, j];
                    if (id <= 0) continue;               // skip missing cell
                    if (!adj.ContainsKey(id))
                        adj[id] = new HashSet<int>();

                    foreach (var d in new (int di, int dj)[] { (-1, 0), (1, 0), (0, -1), (0, 1) })
                    {
                        int x = i + d.di, y = j + d.dj;
                        if (x < 0 || x >= size || y < 0 || y >= size) continue;
                        int nid = board[x, y];
                        if (nid > 0 && nid != id)
                        {
                            // add both directions
                            adj[id].Add(nid);
                            if (!adj.ContainsKey(nid))
                                adj[nid] = new HashSet<int>();
                            adj[nid].Add(id);
                        }
                    }
                }
            }

            // STEP 2: Initialize and greedy-color every ID
            // Make sure every key gets at least a “-1” placeholder
            var colorMap = new Dictionary<int, int>();
            foreach (var id in adj.Keys)
                colorMap[id] = -1;

            // Now do the greedy coloring
            foreach (var kv in adj)
            {
                bool[] used = new bool[3];
                foreach (var nb in kv.Value)
                {
                    int c = colorMap[nb];
                    if (c >= 0 && c < 3) used[c] = true;
                }
                // pick the first unused color
                int pick = 0;
                while (pick < 3 && used[pick]) pick++;
                if (pick >= 3) pick = 0;  // fallback
                colorMap[kv.Key] = pick;
            }

            // STEP 3: Overwrite board IDs with (color + 1), but only if we actually have a mapping
            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
                    int id = board[i, j];
                    if (id > 0 && colorMap.TryGetValue(id, out int c))
                    {
                        board[i, j] = c + 1;
                    }
                    // else leave it (either the missing cell or something went wrong)
                }
            }
        }


        // Draw the grid of colored Panels
        // Draw the grid of colored Panels
        private void BuildBoardUI()
        {
            boardPanel.Controls.Clear();
            boardPanel.SuspendLayout();

            int cellSize = Math.Min(
                (boardPanel.ClientSize.Width - 20) / size,
                (boardPanel.ClientSize.Height - 20) / size);

            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
                    var cell = new Panel
                    {
                        Width = cellSize,
                        Height = cellSize,
                        Left = j * cellSize,
                        Top = i * cellSize,
                        BorderStyle = BorderStyle.FixedSingle
                    };

                    // classic switch to pick the back color
                    int val = board[i, j];
                    Color bg;
                    switch (val)
                    {
                        case -1:
                            bg = Color.Black;
                            break;
                        case 1:
                            bg = trominoColors[0];
                            break;
                        case 2:
                            bg = trominoColors[1];
                            break;
                        case 3:
                            bg = trominoColors[2];
                            break;
                        default:
                            bg = Color.White;
                            break;
                    }
                    cell.BackColor = bg;

                    boardPanel.Controls.Add(cell);
                }
            }

            boardPanel.ResumeLayout();
        }

    }
}
```



n (power of 2): 3
Missing Row: 2
Missing Col: 5



n (power of 2): 4
Missing Row: 5
Missing Col: 8



n (power of 2): 5
Missing Row: 10
Missing Col: 3

Comparison

Brute-Force Approach

Explore all possible tromino placements on the board. For each configuration, try assigning each of the 3 colors to the trominoes. Backtrack if any two adjacent trominoes share the same color or if the board cannot be fully covered. This can be seen in the following pseudo-code:

```
function brute_force(board, trominoes, position):
    if board is fully covered:
        return is_valid_coloring(trominoes)

    for each possible L-shaped placement starting from current position:
        for color in {red, green, blue}:
            if placement is valid:
                place tromino with color
                if brute_force(updated_board, trominoes + new_tromino, next_position):
                    return true
                undo placement
    return false
```

Let T be the number of trominoes on the board. For a board of size 2^n x 2^n with one missing square, the number of trominoes is T = $(4^n - 1) / 3$. For each tromino, we consider up to three color choices and up to four tromino orientations, and for each board configuration, we attempt every possible placement of trominoes. Since tilings and colorings must be validated recursively and backtracked, the total time complexity is

$$O(12^T)$$

where T is the number of trominoes on the board.

Dynamic Programming Approach

The dynamic programming solution applies recursion with memoization to cache solutions of subproblems involving smaller boards. Each subproblem captures the configuration of a subboard along with the coloring of its boundary edges to ensure valid merges of adjacent subboards. The recursion proceeds by dividing the board into four quadrants and solving them individually while placing a central tromino in the region that does not contain the missing square. The key insight is to reuse previously computed solutions of subboards by encoding the missing square's position and boundary constraints into the state.

The pseudocode below captures the essence of this recursive, memoized dynamic programming strategy:

```
function dp(x, y, size, missing, boundary_colors):
    key = (x, y, size, missing, boundary_colors)
    if key in memo:
        return memo[key]

    if size == 2:
        return base_case(x, y, missing, boundary_colors)

    mid = size / 2
    split board into 4 quadrants
    place central tromino at the center avoiding the quadrant with the missing square

    total_ways = 0
    for each valid coloring of the central tromino:
        propagate boundary constraints
        for each quadrant:
            total_ways += dp(quadrant_parameters)

    memo[key] = total_ways
    return total_ways
```

Let B(n) denote the number of distinct boundary configurations for subproblems of size n. For each subproblem, there are approximately $O(n^2)$ positions and B(n) configurations to evaluate. The total number of subproblems is therefore $O(n^2 * B(n))$. For a 2^n x 2^n board, the number of trominoes is $T = (4^n - 1) / 3$. Each subproblem requires constant or low-degree polynomial time depending on how efficiently the boundary constraints are propagated. Consequently, the overall time complexity of the dynamic programming approach is:

$$O(4^n * B(n))$$

The following table below provides a comparative overview of the three well-known techniques. Each method varies significantly in its strategy, efficiency, and suitability depending on the structure and constraints of the input board. The Divide and Conquer method, which is the one used in this report, stands out for its optimal performance. In contrast, brute-force methods attempt to exhaustively explore all possible configurations, making them highly inefficient. Dynamic Programming approaches, though powerful in counting valid configurations under certain constraints, are generally better suited for analytical purposes.

Algorithm Comparison

Technique	Description	Time Complexity	Advantages	Disadvantages
Divide and Conquer	Recursively divide the board into quadrants, place one tromino at center, and recurse with a simulated missing cell in each sub-board.	$O(n^2)$	- Reuses subproblem solutions - Lowest complexity	- Complex state representation
Bruteforce	Try every possible placement of trominoes and backtrack when a solution fails. Explores all configurations.	Exponential	- Simple to implement - Guaranteed to find all solutions	- Extremely slow and unscalable
Dynamic Programming	Use memorization to count the number of valid tiling configurations under certain constraints.	Exponential or polynomial	- Efficient use of overlapping subproblems	- Requires careful state encoding - Large memory

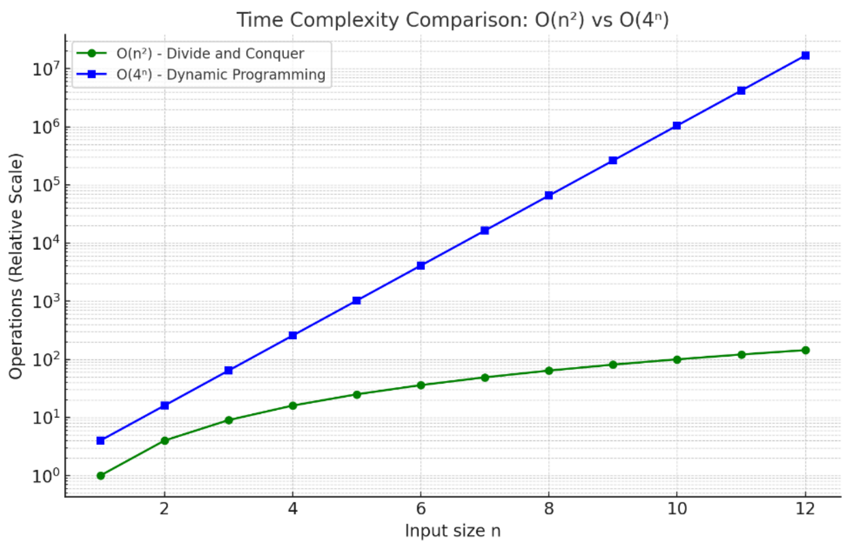


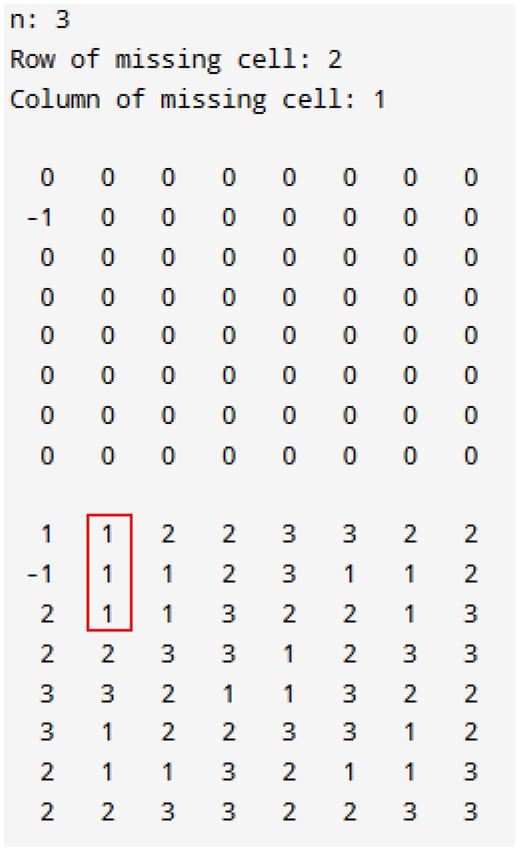
Figure 5: Divide and Conquer (green) versus Dynamic Programming (blue)

Additional Notes

An alternative coloring approach was explored during the development phase. Instead of building an adjacency graph to identify neighboring trominoes and applying a separate coloring phase afterward, the initial idea was to assign a color to each tromino immediately upon placement during the recursive tiling by calling the function determineColor.

```
// Determine tromino color
short determineColor(vector<vector<int>>& board, int center_i, int center_j, short quad) {
    // Track whether each color is used or not
    vector<bool> used(3, false);

    // Get the cells of the current tromino based on its orientation
    vector<pair<int, int>> trominoCells;
    switch(quad) {
        case 1: // Standard L-shape
            trominoCells = {{center_i, center_j}, {center_i+1, center_j}, {center_i+1, center_j+1}};
            break;
        case 2: // Mirrored L-shape
            trominoCells = {{center_i, center_j+1}, {center_i+1, center_j}, {center_i+1,
center_j+1}};
            break;
        case 3: // Rotated L-shape
            trominoCells = {{center_i, center_j}, {center_i, center_j+1}, {center_i+1, center_j}};
            break;
        case 4: // Other rotated L-shape
            trominoCells = {{center_i, center_j}, {center_i, center_j+1}, {center_i+1, center_j+1}};
            break;
    }

    // Check adjacent cells
    for (auto& cell : trominoCells) {
        int i = cell.first, j = cell.second;

        // Check all 4 directions
        if (i > 0 && board[i-1][j] > 0 && board[i-1][j] <= 3) // Up
            used[board[i-1][j]-1] = true;
        if (i < board.size()-1 && board[i+1][j] > 0 && board[i+1][j] <= 3) // Down
            used[board[i+1][j]-1] = true;
        if (j > 0 && board[i][j-1] > 0 && board[i][j-1] <= 3) // Left
            used[board[i][j-1]-1] = true;
        if (j < board.size()-1 && board[i][j+1] > 0 && board[i][j+1] <= 3) // Right
            used[board[i][j+1]-1] = true;

        // Each check makes sure i and j are within limits and the board number is between 1 and 3
(inclusive)
    }

    // Find the first available color
    for (short color = 1; color <= 3; color++) {
        if (!used[color-1]) {
            return color;
        }
    }

    // This should NEVER happen
    return -1;
}
```

While this greedy strategy seemed promising in early test cases, it ultimately failed in more complex configurations—especially where multiple adjacent trominoes converged at the quadrant centers. The lack of contextual awareness during placement meant that trominoes could unknowingly be colored the same as a neighbor, violating the adjacency constraint.


```
n: 3
Row of missing cell: 2
Column of missing cell: 1

  0   0   0   0   0   0   0   0
 -1   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0

  1   1   2   2   3   3   2   2
 -1   1   1   2   3   1   1   2
  2   1   1   3   2   2   1   3
  2   2   3   3   1   2   3   3
  3   3   2   1   1   3   2   2
  3   1   2   2   3   3   1   2
  2   1   1   3   2   1   1   3
  2   2   3   3   2   2   3   3
```

These errors highlighted the importance of building a proper adjacency graph and using a global greedy coloring algorithm after tiling is complete. This correction ensured that no two connected trominoes share a color, and that the final visual output is both accurate and clean.

Conclusion

Through this thorough research, we managed to successfully implement and analyze a classic algorithm for solving the tromino tiling problem using a divide and conquer approach. By recursively dividing the board into quadrants and placing a central L-shaped tromino to simulate missing cells, the algorithm achieves an efficient and elegant solution with a guaranteed runtime of $O(n^2)$ on a 2^n x 2^n board with one missing square. Beyond simply tiling the board, the project extends this technique to apply a valid 3-coloring to the trominoes using an adjacency graph and a greedy coloring algorithm, ensuring no two adjacent tiles share the same color. Visual examples, benchmark comparisons, and complexity plots further support the correctness and efficiency of the method. Compared to other common strategies—such as brute force, greedy filling, or dynamic programming—the divide and conquer method proves to be not only more scalable but also mathematically grounded and visually interpretable. The project also includes comprehensive testing across a variety of board sizes and missing cell positions to validate both correctness and performance.

References

[1] Knuth, D. E. The Art of Computer Programming, Volume 1: Fundamental Algorithms. 3rd Edition. Section 2.2.2, Exercise 6 (Defective Chessboard Problem).

[2] Golomb, S. W. Polyominoes: Puzzles, Patterns, Problems, and Packings. Dover Publications, 1996.

[3] I-Ping Chu, Richard Johnsonbaugh: Tiling and recursion. SIGCSE 1987: 261-263

Task 2: Knight’s Tour

Definition

Is it possible for a chess knight to visit all the cells of an 8 × 8 chessboard exactly once, ending at a cell one knight’s move away from the starting cell? (Such a tour is called closed or re-entrant. Note that a cell is considered visited only when the knight lands on it, not just passes over it on its move.) What if the chessboard is n × n? is all n where n>8 is possible for a chess knight to visit all the cells? If not, what is the cases? design a greedy algorithm to find the minimum number of moves the chess knight needs. And study the different cases of n.

Assumptions

- The knight's starting position must be a valid square within the boundaries of the n×n chessboard.
- The tour is closed if the knight visits all squares exactly once and ends at a square one knight's move away from the starting position.
- The algorithm follows Warnsdorff’s heuristic, selecting moves that lead to squares with fewer available moves, aiming to avoid dead-ends.
- The algorithm will keep attempting to find a closed knight's tour until a solution is found, with random starting positions for each run.
- algorithm does not reconsider previous moves once the knight has moved, meaning it may fail if it reaches a dead-end.
- The algorithm works for any n×n chessboard, but the feasibility of a closed tour depends on the size of n.
- The knight moves in an "L" shape, either 2 squares in one direction and 1 square in the other, or 1 square in one direction and 2 squares in the other.
- The algorithm assumes that a valid closed knight's tour is possible for valid chessboard sizes and that the knight will visit all squares exactly once.

Explanation and Pseudo-Code

The Knight’s Tour problem is a classic challenge in computer science and mathematics that involves moving a knight across an n x n chessboard in such a way that it visits every square exactly once. The knight moves in an “L” shape—two squares in one direction and then one square perpendicular to that. A particularly interesting variation of this problem is the **closed knight’s tour**, where the knight must not only visit each square once but also end on a square that is one legal knight’s move away from the starting position, forming a complete loop. This closed tour, also called a **re-entrant tour**, is the focus of this assignment. The objective is to determine whether a closed knight’s tour is possible for various board sizes, starting with the standard 8×8 chessboard and extending to larger sizes where n>8. The algorithm used follows a greedy approach that selects moves based on specific heuristics to efficiently explore possible paths. The assignment also involves analyzing the behavior and performance of this method to identify which board sizes allow for successful closed tours and to evaluate the minimum number of steps needed to complete them.

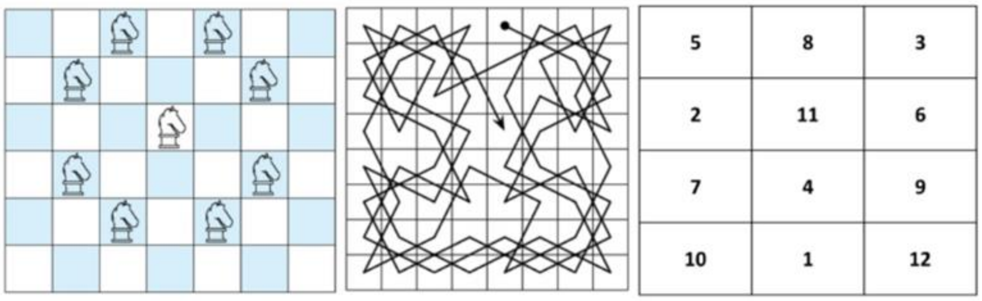


Figure 6: Legal moves of a knight, knight's tour moves and an example

```
DEFINE N = 8  // Chessboard size is 8x8

DEFINE knight's possible moves
    cx = [1, 1, 2, 2, -1, -1, -2, -2]
    cy = [2, -2, 1, -1, 2, -2, 1, -1]

FUNCTION limits(x, y):
    IF x >= 0 AND y >= 0 AND x < N AND y < N:
        RETURN TRUE
    ELSE:
        RETURN FALSE

FUNCTION isempty(board, x, y):
    IF limits(x, y) AND board[y * N + x] == -1:
        RETURN TRUE
    ELSE:
        RETURN FALSE

FUNCTION getDegree(board, x, y):
    count = 0
    FOR i = 0 to 7:
        nx = x + cx[i]
        ny = y + cy[i]
        IF isempty(board, nx, ny):
            count += 1
    RETURN count

FUNCTION nextMove(board, x, y):
    Initialize:
        min_deg = N + 1
        min_deg_index = -1
        start = random integer from 0 to 7

    FOR count = 0 to 7:
        i = (start + count) mod 8
        nx = x + cx[i]
        ny = y + cy[i]


        IF isempty(board, nx, ny):
            degree = getDegree(board, nx, ny)
            IF degree < min_deg:
                min_deg = degree
                min_deg_index = i

    IF min_deg_index == -1:  // No valid move found
        RETURN FALSE

    // Update the knight's position and mark the move
    x = x + cx[min_deg_index]
    y = y + cy[min_deg_index]
    board[y * N + x] = board[(y - cy[min_deg_index]) * N + (x - cx[min_deg_index])] + 1

    RETURN TRUE

FUNCTION neighbour(x, y, start_x, start_y):
    FOR i = 0 to 7:
        IF (x + cx[i] == start_x) AND (y + cy[i] == start_y):
            RETURN TRUE
    RETURN FALSE

FUNCTION printBoard(board):
    FOR each row from 0 to N-1:
        FOR each column from 0 to N-1:
            PRINT board[column * N + row] with tab spacing
        PRINT new line

FUNCTION findClosedTour():
    Initialize board of size N * N with all values = -1

    Generate random start_x and start_y
    Set current position x = start_x, y = start_y
    board[y * N + x] = 1 (mark the first move)

    moveCount = 1  // Initialize move count
    FOR i = 1 to N*N - 1:
        IF nextMove(board, x, y) == FALSE:
            RETURN FALSE

    IF neighbour(x, y, start_x, start_y) == FALSE:  // Check if it's a closed tour
        RETURN FALSE

    printBoard(board)
    PRINT "Minimum number of moves: ", moveCount  // Print the move count
    RETURN TRUE

MAIN FUNCTION:
    Seed the random number generator

    WHILE findClosedTour() == FALSE:
        Try again until a closed tour is foundDefine
```

Knight's Possible Moves

The knight’s movement is defined by two arrays, cx[] and cy[], which represent the possible changes in x and y coordinates, respectively. These arrays capture the eight possible directions a knight can move in an "L" shape on a chessboard.

Limits Function

This function checks whether a given position (x, y) is within the boundaries of the chessboard. It returns TRUE if the position lies between 0 and N−1N - 1N−1 for both coordinates, ensuring the knight does not move off the board.

IsEmpty Function

The isEmpty function determines whether a given square is both within bounds and unvisited. A square is considered unvisited if its value in the board array is -1. This function helps the algorithm avoid revisiting squares.

GetDegree Function

This function calculates the number of valid unvisited squares that the knight can move to from a given position. Known as the “degree” of the square, this value is used to prioritize moves in the next step, with fewer onward moves being preferred.

NextMove Function

Using Warnsdorff’s heuristic, this function evaluates all eight potential knight moves from the current position and selects the one with the lowest degree (fewest future move options). If a valid next move is found, it updates the knight’s position and move count. If no move is valid, it returns FALSE.

Neighbour Function

After completing the tour, this function checks whether the knight’s final position is one knight move away from the starting square. This validation ensures that the tour is closed (re-entrant), which is a requirement of the problem.

PrintBoard Function

This function displays the chessboard after the tour is completed. Each square is printed with the number corresponding to the move when the knight visited it, providing a clear representation of the tour path.

FindClosedTour Function

This function manages the overall process of the knight’s tour. It initializes the board, selects a random starting position, and repeatedly calls nextMove() to attempt to build a valid closed tour. If the tour is successful, it prints the final board configuration.

Main Function

The main function initializes the program and repeatedly calls findClosedTour() until a valid closed knight’s tour is found. It ensures that different random starting points are tested, increasing the chance of finding a valid solution.

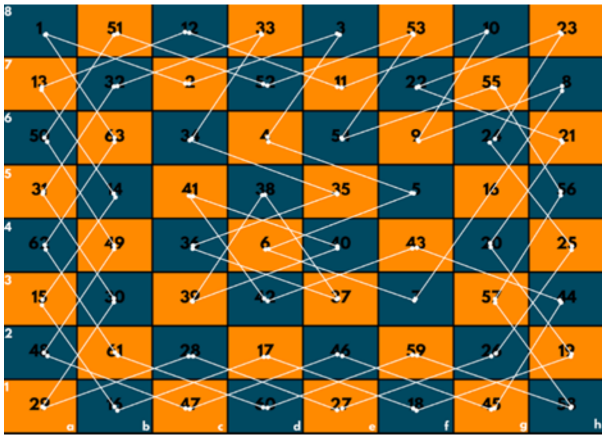


*Figure 7: Sample solution of the knight moves in a 8*8 closed tour problem*

<u>C++ Code</u>

The code implements a solution for the Knight's Tour problem on an 8x8 chessboard using Warnsdorff's Heuristic, a greedy algorithm. The core idea is to move the knight around the board such that it visits every square exactly once, ultimately ending on a square one move away from the starting position, creating a closed tour. The knight's possible moves are defined by two arrays, cx[] and cy[], which represent changes in the x and y coordinates, respectively. The limits() function checks if a given position lies within the chessboard's boundaries, while the isempty() function ensures the knight moves to unvisited squares. The getDegree() function calculates the number of valid moves from a given square, aiding the selection of the next move based on Warnsdorff’s heuristic, which prioritizes squares with fewer possible moves. The nextMove() function implements this heuristic, choosing the square with the minimum degree of future moves. After the tour, the neighbour() function verifies if the knight’s final position is one knight's move away from the starting position, ensuring the tour is closed. The print() function displays the chessboard with the sequence of knight’s moves. The findClosedTour() function coordinates the knight’s movements, starting from a random position, attempting to complete the tour, and checking if it is closed. The main() function initializes the random number generator and repeatedly attempts to find a valid closed tour until a solution is found and printed. The sequence of steps when solving the problem begins with the initialization of the chessboard, where all squares are marked as -1, indicating they haven't been visited. A random starting position for the knight is then chosen, and the first square is marked as 1, representing the first move. The moveCount is set to 1, and the algorithm enters a loop to find the knight’s next move. Using the nextMove() function, the knight selects the next square based on Warnsdorff's Heuristic, which favors squares with fewer valid future moves. This process repeats until the knight visits every square on the board. After completing the tour, the neighbour() function checks whether the knight’s final position is one move away from the starting square to ensure the tour is closed. Finally, the solution is printed, and the process continues until a valid closed tour is found.

```
/*
Computer Engineering and Artificial Intelligence Program
CSE245: Advanced Algorithms and Complexity – Spring 2025
Task 2 – Knight’s Tour
*/

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cstdio>
#define N 8
// Move pattern on basis of the change of
// x coordinates and y coordinates respectively
static int cx[N] = {1, 1, 2, 2, -1, -1, -2, -2};
static int cy[N] = {2, -2, 1, -1, 2, -2, 1, -1};

// function restricts the knight to remain within
// the 8x8 chessboard
bool limits(int x, int y)
{
    return ((x >= 0 && y >= 0) && (x < N && y < N));
}

/* Checks whether a square is valid and empty or not */
bool isempty(int a[], int x, int y)
{
    return (limits(x, y)) && (a[y * N + x] < 0);
}

/* Returns the number of empty squares adjacent
to (x, y) */
int getDegree(int a[], int x, int y)
{
    int count = 0;
    for (int i = 0; i < N; ++i)
        if (isempty(a, (x + cx[i]), (y + cy[i])))
            count++;

    return count;
}

// Picks next point using Warnsdorff's heuristic.
// Returns false if it is not possible to pick
// next point.
bool nextMove(int a[], int *x, int *y)
{
    int min_deg_idx = -1, c, min_deg = (N + 1), nx, ny;

    // Try all N adjacent of (*x, *y) starting
    // from a random adjacent. Find the adjacent
    // with minimum degree.
    int start = rand() % N;
    for (int count = 0; count < N; ++count)
    {
        int i = (start + count) % N;
        nx = *x + cx[i];
        ny = *y + cy[i];
        if ((isempty(a, nx, ny)) &&
            (c = getDegree(a, nx, ny)) < min_deg)
        {
            min_deg_idx = i;
            min_deg = c;
        }
    }

    // IF we could not find a next cell
    if (min_deg_idx == -1)
        return false;

    // Store coordinates of next point
    nx = *x + cx[min_deg_idx];
    ny = *y + cy[min_deg_idx];

    // Mark next move
    a[ny * N + nx] = a[(*y) * N + (*x)] + 1;

    // Update next point
    *x = nx;
    *y = ny;

    return true;
}

/* displays the chessboard with all the
legal knight's moves */
void print(int a[])
{
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
            printf("%d\t", a[j * N + i]);
        printf("\n");
    }
}

/* checks its neighbouring squares */
/* If the knight ends on a square that is one
knight's move from the beginning square,
then tour is closed */
bool neighbour(int x, int y, int xx, int yy)
{
    for (int i = 0; i < N; ++i)
        if (((x + cx[i]) == xx) && ((y + cy[i]) == yy))
            return true;

    return false;
}

/* Generates the legal moves using Warnsdorff's
heuristics. Returns false if not possible */
bool findClosedTour()
{
    // Filling up the chessboard matrix with -1's
    int a[N * N];
    for (int i = 0; i < N * N; ++i)
        a[i] = -1;

    // Random initial position
    int sx = rand() % N;
    int sy = rand() % N;

    // Current points are same as initial points
    int x = sx, y = sy;
    a[y * N + x] = 1; // Mark first move.

    int moveCount = 1;  // Start the move count



    // Keep picking next points using
    // Warnsdorff's heuristic
    for (int i = 0; i < N * N - 1; ++i)
    {
        if (nextMove(a, &x, &y) == 0)
            return false;
        moveCount++; // Increment the move count
    }

    // Check if tour is closed (Can end
    // at starting point)
    if (!neighbour(x, y, sx, sy))
        return false;

    print(a);
    std::cout << "Minimum number of moves: " << moveCount << std::endl;
    return true;
}

int main()
{
    // To make sure that different random
    // initial positions are picked.
    srand(time(NULL));

    // While we don't get a solution
    while (!findClosedTour())
    {
        ;
    }

    return 0;
}
```

Time Complexity Analysis

The time complexity of the Knight's Tour code can be understood by analyzing the various components of the algorithm. First, the findClosedTour() function initializes a chessboard of size N x N with all squares set to -1, which takes O(N^2) time. A random starting position is then selected in constant time,O(1). The main tour loop involves repeatedly selecting the next move using Warnsdorff's Heuristic. For each move, the nextMove() function evaluates all 8 possible knight moves, checks if they are valid, and selects the one with the fewest valid future moves (minimum degree). Evaluating each move and checking its degree takes constant time, O(1), and since there are N^2 moves to make, this step contributes O(N^2) to the overall time complexity. After completing the tour, the algorithm checks if the final position is one move away from the starting position to ensure the tour is closed, which takes O(1) time. Finally, the print() function is called to display the board, which takes O(N^2) time since it prints each square on the board. Overall, the total time complexity is dominated by the operations related to the squares on the chessboard, making the overall time complexity O(N^2). The recurrence relation for the time complexity of the algorithm can be expressed as:

$$T(n) = 8T(n-1) + O(1)$$

Where:

- T(N) represents the time taken to solve the Knight's Tour on an N×NN \times N chessboard.
- T(N−1) is the time to solve the problem on a smaller N−1×Nchessboard (after making one move).
- The factor 8 accounts for checking all 8 possible knight moves from each square.
- O(1) represents constant-time operations, like updating the board.

By solving this recurrence relation, it simplifies to O(N^2), indicating that the time complexity is proportional to the number of squares on the chessboard, which is:

$$O(n^2)$$

as the board requires N^2 positions to store the knight’s path.

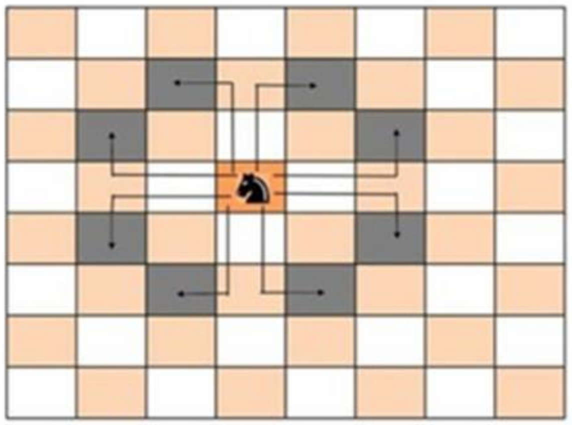
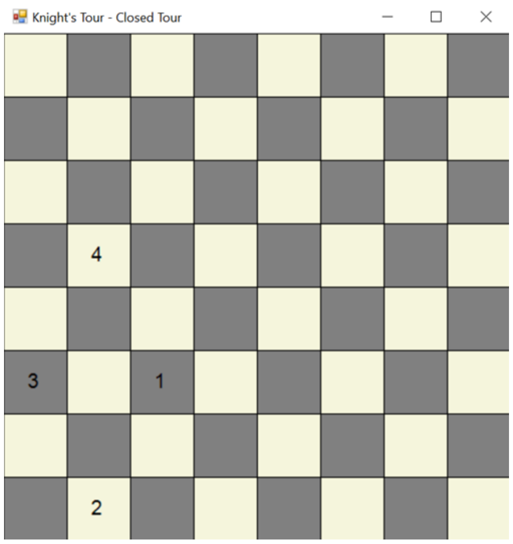


Figure 8: The 8 available moves for the Knight at each square (Represented by The Factor 8)

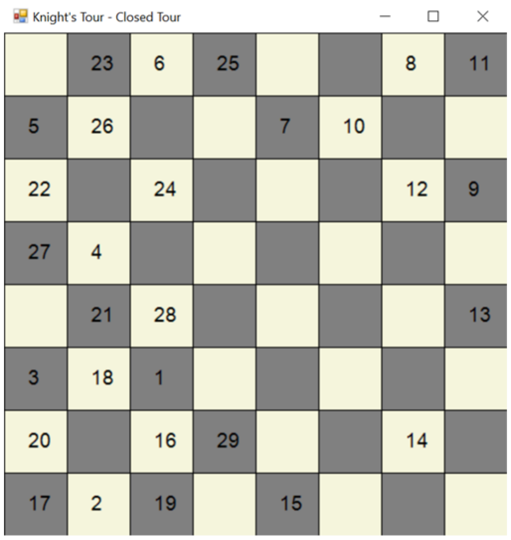
Visualization Code (using C#)

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WindowsFormsApp1
{
    public partial class KnightTour : Form
    {
        const int N = 8;
        int[] board = new int[N * N];
        int[] cx = { 1, 1, 2, 2, -1, -1, -2, -2 };
        int[] cy = { 2, -2, 1, -1, 2, -2, 1, -1 };
        int squareSize = 60;
        Timer timer = new Timer();
        int moveCount = 1;
        int x, y, sx, sy;
        int displayIndex = 1;

        public KnightTour()
        {
            InitializeComponent();
            this.ClientSize = new Size(N * squareSize, N * squareSize);
            this.Text = "Knight's Tour - Closed Tour";
            timer.Interval = 200;
            timer.Tick += Timer_Tick;
            StartTour();
        }

        void StartTour()
        {
            Random rand = new Random();
            bool success = false;

            while (!success)
            {
                moveCount = 1;
                for (int i = 0; i < N * N; i++) board[i] = -1;

                sx = rand.Next(N);
                sy = rand.Next(N);
                x = sx;
                y = sy;
                board[y * N + x] = moveCount;

                success = FindTour();
            }

            displayIndex = 1;
            timer.Start();
        }

        bool FindTour()
        {
            for (int i = 0; i < N * N - 1; ++i)
            {
                if (!NextMove(ref x, ref y)) return false;
            }

            return IsNeighbour(x, y, sx, sy);
        }

        bool Limits(int x, int y) => (x >= 0 && y >= 0 && x < N && y < N);
        bool IsEmpty(int x, int y) => Limits(x, y) && board[y * N + x] == -1;

        int GetDegree(int x, int y)
        {
            int count = 0;
            for (int i = 0; i < N; i++)
                if (IsEmpty(x + cx[i], y + cy[i])) count++;
            return count;
        }

        bool NextMove(ref int x, ref int y)
        {
            int min_deg_idx = -1, min_deg = N + 1, nx, ny;
            Random rand = new Random();
            int start = rand.Next(N);

            for (int count = 0; count < N; ++count)
            {
                int i = (start + count) % N;
                nx = x + cx[i];
                ny = y + cy[i];

                if (IsEmpty(nx, ny) && GetDegree(nx, ny) < min_deg)
                {
                    min_deg_idx = i;
                    min_deg = GetDegree(nx, ny);
                }
            }

            if (min_deg_idx == -1) return false;

            nx = x + cx[min_deg_idx];
            ny = y + cy[min_deg_idx];

            board[ny * N + nx] = ++moveCount;
            x = nx;
            y = ny;

            return true;
        }

        bool IsNeighbour(int x, int y, int xx, int yy)
        {
            for (int i = 0; i < N; i++)
                if ((x + cx[i] == xx) && (y + cy[i] == yy)) return true;
            return false;
        }

        private void Timer_Tick(object sender, EventArgs e)
        {
            displayIndex++;
            if (displayIndex > N * N)
                timer.Stop();

            this.Invalidate(); // Redraw
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            Font font = new Font("Arial", 14);
            for (int row = 0; row < N; row++)
            {
                for (int col = 0; col < N; col++)
        {
          int idx = board[row * N + col];
          bool filled = idx != -1 && idx <= displayIndex;

          Brush bg = (row + col) % 2 == 0 ? Brushes.Beige : Brushes.Gray;
          g.FillRectangle(bg, col * squareSize, row * squareSize, squareSize, squareSize);
          g.DrawRectangle(Pens.Black, col * squareSize, row * squareSize, squareSize,
squareSize);

          if (filled)
          {
            string num = idx.ToString();
            g.DrawString(num, font, Brushes.Black, col * squareSize + 20, row *
squareSize + 18);
          }
        }
      }
    }
  }
}
```

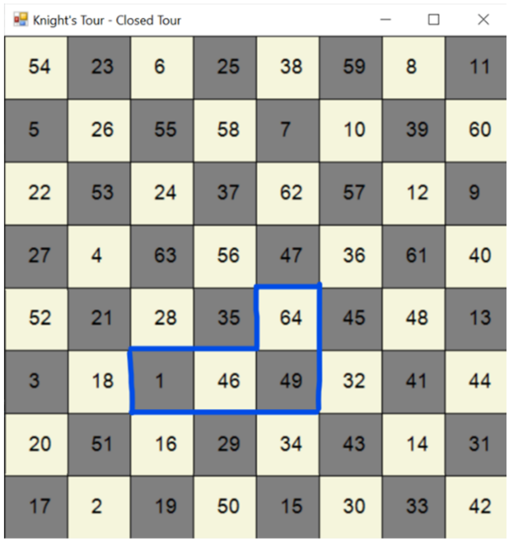
Sample Output



	4						
3		1					
	2						



	23	6	25			8	11
5	26			7	10		
22		24				12	9
27	4						
	21	28					13
3	18	1					
20		16	29			14	
17	2	19		15			



54	23	6	25	38	59	8	11
5	26	55	58	7	10	39	60
22	53	24	37	62	57	12	9
27	4	63	56	47	36	61	40
52	21	28	35	64	45	48	13
3	18	1	46	49	32	41	44
20	51	16	29	34	43	14	31
17	2	19	50	15	30	33	42

Advantages of Warnsdorff's Rule

- Efficiency:
  Warnsdorff's Rule is much more efficient than backtracking because it reduces the search space. By always choosing the next move with the least number of onward moves, it prevents exploring less promising paths early on.

- Simple Heuristic:
  The algorithm is based on a simple, intuitive rule (choosing the square with the fewest onward moves), which is easy to implement and understand.

- Scalability:
  Since it's a heuristic, Warnsdorff's Rule can be applied to larger chessboards more quickly than backtracking or exhaustive search, making it suitable for bigger problems where brute-force algorithms would be too slow.

- Frequently Finds Solutions:
  It often finds a solution without needing to backtrack, especially for closed tours, if the heuristic is well-chosen.

Disadvantages of Warnsdorff's Rule

- Doesn't Always Guarantee a Solution:
  While it's efficient, Warnsdorff's Rule doesn't guarantee that it will always find a solution, especially for all initial positions or for certain board sizes. The heuristic may fail to find a valid path if the board size or starting position isn't ideal.

- Heuristic Nature:
  As a heuristic, it may not find the optimal path or may get "stuck" in certain cases, leading to suboptimal results. It's not a complete search like backtracking.

- Closed Tour Difficulties:
  For closed Knight's Tours (where the knight ends on a square one move away from the start), Warnsdorff's Rule is less reliable. It can be harder to find a closed tour using this heuristic compared to an open tour.

- Lack of Flexibility:
  The algorithm is limited by its predefined heuristic. If the heuristic is tweaked or if a different approach is needed, it can require significant changes or adaptations.

Comparison

Backtracking solution (Alternative solution)

The Knight’s Tour problem is solved using a backtracking approach in this code. Backtracking is a recursive technique used to explore all possible paths in a systematic way. The knight starts at the initial position (typically the top-left corner of the board) and attempts to visit every cell exactly once by making only valid knight moves. At each step, the algorithm tries all eight possible directions a knight can move. If a move leads to an unvisited and valid square, it proceeds recursively from that new position. If a dead end is reached (i.e., no further valid moves are possible), the algorithm backtracks by undoing the last move and trying a different direction. This process continues until either all cells are visited (indicating a successful tour) or all paths are exhausted without visiting every cell (indicating that no tour is possible from the starting point). This method ensures that every possible configuration is tried.

```
Function isSafe(x, y, sol):
    Return true if (x, y) is within board bounds AND sol[x][y] is -1
    Else return false
Function printSolution(sol):
    For each row in sol:
        For each column in row:
            Print sol[row][col] with spacing
Function solveKT():
    Create 2D array sol[N][N], initialize all to -1
    Initialize moveCount = 0
    Define knight's 8 possible moves in arrays xMove[8] and yMove[8]
    Set starting point sol[0][0] = 0
    If solveKTUtil(0, 0, 1, sol, xMove, yMove, moveCount) returns false:
        Print "Solution does not exist"
        Return 0
    Else:
        Call printSolution(sol)
        Print "Minimum number of steps: " + moveCount
        Return 1
Function solveKTUtil(x, y, movei, sol, xMove, yMove, moveCount):
    If movei == N * N:
        Return true
    For k = 0 to 7:
        next_x = x + xMove[k]
        next_y = y + yMove[k]
        If isSafe(next_x, next_y, sol):
            Set sol[next_x][next_y] = movei
            Increment moveCount
            If solveKTUtil(next_x, next_y, movei + 1, sol, xMove, yMove, moveCount) returns true:
                Return true
            Else:
                // Backtrack
                Set sol[next_x][next_y] = -1
                Decrement moveCount
    Return false
Main:
    Call solveKT()
```

isSafe Function

function checks whether the knight can move to a given position (x, y) on the chessboard. It validates two conditions: whether the position is within the chessboard bounds and whether the square has not been visited yet (indicated by the value -1 in the solution matrix sol). If both conditions are true, it returns true, meaning the move is safe; otherwise, it returns false.

printSolution Function

function is responsible for printing the knight's tour solution. It traverses the solution matrix sol and prints the move sequence on the chessboard, providing a visual representation of the knight's journey. Each cell of the matrix contains a number indicating the step in the sequence, which is printed with appropriate spacing.

solveKT Function

function sets up the chessboard and initiates the knight's tour process. It creates a 2D array sol[N][N] and initializes all values to -1, marking all squares as unvisited. The knight begins at the top-left corner, with sol[0][0] set to 0. The function then calls the recursive solveKTUtil function to attempt solving the knight's tour. If no solution is found, it prints "Solution does not exist"; if a solution is found, it calls printSolution to display the solution and prints the total number of moves.

solveKTUtil Function

function is the core of the solution, implementing the recursive backtracking approach. Starting from the given position (x, y), it tries to move the knight to all possible positions using the knight's 8 possible moves. For each valid move, it updates the sol matrix and recursively calls itself to continue solving the tour. If it finds a valid move that leads to a complete solution, it returns true. If no valid moves are found, it backtracks by resetting the square to -1 and tries a different move. This process continues until all squares are visited or no solution exists.

C++ Code

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
using namespace std;

#define N 8

// Function declarations
int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[], int yMove[], int &moveCount);

bool isSafe(int x, int y, int sol[N][N]) {
    return (x >= 0 && x < N && y >= 0 && y < N && sol[x][y] == -1);
}
```

```
void printSolution(int sol[N][N]) {
    for (int x = 0; x < N; x++) {
        for (int y = 0; y < N; y++)
            cout << " " << setw(2) << sol[x][y] << " ";
        cout << endl;
    }
}

int solveKT() {
    int sol[N][N];
    int moveCount = 0;  // To track number of steps

    // Initialize all cells to -1
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            sol[x][y] = -1;

    // Possible knight moves
    int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
    int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

    // Starting position
    sol[0][0] = 0;

    // Try to solve the tour
    if (solveKTUtil(0, 0, 1, sol, xMove, yMove, moveCount) == 0) {
        cout << "Solution does not exist" << endl;
        return 0;
    } else {
        printSolution(sol);
        cout << "Minimum number of steps: " << moveCount << endl;
    }

    return 1;
}

int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[8], int yMove[8], int &moveCount) {
    int k, next_x, next_y;

    if (movei == N * N)
        return 1;

    // Try all next moves from the current coordinate x, y
    for (k = 0; k < 8; k++) {
        next_x = x + xMove[k];
        next_y = y + yMove[k];
        if (isSafe(next_x, next_y, sol)) {
            sol[next_x][next_y] = movei;
            moveCount++; // Count this step

            if (solveKTUtil(next_x, next_y, movei + 1, sol, xMove, yMove, moveCount) == 1)
                return 1;
            else {
                // Backtracking
                sol[next_x][next_y] = -1;
                moveCount--; // Undo the step
            }
        }
    }
    return 0;
}

// Main function
int main() {
    solveKT();
    return 0;
}
```

Time Complexity Analysis

The time complexity of the given code can be analyzed based on the recursive backtracking approach used in the solveKTUtil() function. In the worst case, the algorithm attempts all possible moves from each square, recursively exploring every potential path the knight can take on the chessboard.

Since the knight must visit every square exactly once and there are NN squares on the board, the algorithm will make a recursive call for each square. For each recursive call, the algorithm evaluates 8 possible moves (since the knight has 8 possible moves). This leads to a branching factor of 8 at each level of recursion. Hence, in the worst case, the algorithm explores all 8 possible moves for each of the NN squares.

Therefore, the time complexity follows a recurrence relation similar to:

$$T(n) = 8T(n-1) + O(1)$$

Where:

- T(N) represents the time taken to solve the Knight's Tour.
- T(N−1) is the time to solve the problem on a smaller N - 1 x N chessboard (after making one move).
- The factor 8 accounts for checking all 8 possible knight moves from each square.
- O(1) represents constant-time operations, like updating the boardBy

solving this recurrence relation, it simplifies to O(N^2), indicating that the time complexity is proportional to the number of squares on the chessboard which is:

$$O(n^2)$$

Advantages of Backtracking

1. Guaranteed to Find a Solution (if one exists)
   - It exhaustively searches all possibilities, so it will always find a valid tour if there is one.
2. Simple to Implement
   - Conceptually and programmatically straightforward — just try all moves and backtrack if you get stuck.
3. Flexible
   - Works on boards of any size, not just the standard 8×8.
   - Can be easily adapted for finding all possible tours, not just one.
4. Great for Learning
   - Excellent for teaching recursion, decision trees, and algorithmic thinking.

Disadvantages of Backtracking

1. Very Slow
   - Especially on larger boards — time complexity is O(N!), which grows extremely fast.
   - Explores many unnecessary paths before finding the solution.
2. High Memory Usage
   - Recursive calls consume a lot of memory (stack space).
   - May lead to stack overflow if not optimized or for very large boards.
3. No Heuristics
   - Doesn't make "smart" decisions — it blindly explores all possibilities.
4. Not Practical for Real-Time Applications
   - Too slow for situations that require fast or real-time results.

Divide-and-Conquer

The Knight's Tour problem can be approached using **divide-and-conquer** by breaking the chessboard into smaller sub-boards, solving the tour for each section independently, and then carefully merging the solutions while ensuring valid knight moves connect them. The board is divided into quadrants (or smaller sections), and each is solved recursively—either using a known base case (like a 6×6 board) or further subdivision. The key challenge lies in **combining** the partial tours by identifying connecting moves between adjacent sections and adjusting coordinates to form a seamless, closed loop. While not as straightforward as backtracking, this method can work efficiently for larger boards by reducing problem complexity through decomposition. However, ensuring correct transitions between sub-tours requires careful handling of entry and exit points.

Pseudocode

```
function divideAndConquerKnightTour(boardSize):
    // Base case: known knight's tour for small board
    if boardSize == 6:
        return knownSolutionFor6x6Board()

    // Step 1: Divide the board into four quadrants
    halfSize = boardSize / 2

    // Quadrant definitions:
    // Quadrant 1: Top-left         (0, 0)
    // Quadrant 2: Top-right        (0, halfSize)
    // Quadrant 3: Bottom-left      (halfSize, 0)
    // Quadrant 4: Bottom-right     (halfSize, halfSize)

    // Step 2: Recursively solve each smaller quadrant
    tour1 = divideAndConquerKnightTour(halfSize)   // Q1
    tour2 = divideAndConquerKnightTour(halfSize)   // Q2
    tour3 = divideAndConquerKnightTour(halfSize)   // Q3
    tour4 = divideAndConquerKnightTour(halfSize)   // Q4

    // Step 3: Offset coordinates for each tour to fit their quadrant
    // tour2 is shifted right
    adjustedTour2 = offsetTour(tour2, 0, halfSize)

    // tour3 is shifted down
    adjustedTour3 = offsetTour(tour3, halfSize, 0)

    // tour4 is shifted right and down
    adjustedTour4 = offsetTour(tour4, halfSize, halfSize)

    // Step 4: Connect the tours together with valid knight moves
    // These functions search for a valid move between two sub-tours
    connection1 = findConnectingMove(tour1, adjustedTour2)   // Connect Q1 → Q2
    connection2 = findConnectingMove(adjustedTour2, adjustedTour4) // Q2 → Q4
    connection3 = findConnectingMove(adjustedTour4, adjustedTour3) // Q4 → Q3
    connection4 = findConnectingMove(adjustedTour3, tour1)   // Q3 → Q1

    // Step 5: Combine the tours and connections in a single path
    fullTour = combineTours(
        tour1,
        adjustedTour2,
        adjustedTour4,
        adjustedTour3,
        connection1,
        connection2,
        connection3,
        connection4
    )

    return fullTour
function offsetTour(tour, rowOffset, colOffset):
    newTour = []
    for move in tour:
        newRow = move.row + rowOffset
        newCol = move.col + colOffset
        newTour.append((newRow, newCol))
    return newTour

function findConnectingMove(tourA, tourB):
    // Try to find a knight move from the end of tourA to the start of tourB
    for endMove in lastKMoves(tourA):
        for startMove in firstKMoves(tourB):
            if isKnightMove(endMove, startMove):
                return (endMove, startMove)
    raise Error("No valid knight connection found")

function combineTours(tour1, tour2, tour3, tour4, conn1, conn2, conn3, conn4):
    // Combine tours in the order with connecting moves
    return tour1 + [conn1[1]] + tour2 + [conn2[1]] + tour4 + [conn3[1]] + tour3 + [conn4[1]]
```

divideAndConquerKnightTour Function

Solves the knight's tour using divide-and-conquer. Splits the board into quadrants, solves each one, adjusts positions, connects them, and combines into a full tour.

knownSolutionFor6x6Board Function

Returns a precomputed valid knight's tour for a 6x6 board.

offsetTour Function

Shifts all the moves in a tour by adding row and column offsets to place it in the correct quadrant.

findConnectingMove Function

Finds a valid knight move from the end of tourA to the start of tourB.

isKnightMove Function

Checks if a move from one square to another is a valid knight's move.

combineTours Function

Merges the four quadrant tours and their connecting moves into a single complete tour.

lastKMoves Function

Returns the last k moves of a tour to check possible connections.

firstKMoves Function

Returns the first k moves of a tour to check possible connections.

Time Complexity Analysis

The time complexity of the given divide-and-conquer algorithm can be analyzed as follows:

Recurrence Relation

1. Divide: The board of size n × n is split into 4 subproblems of size (n/2) × (n/2).
2. Combine: Merging the solutions requires:
    - Adjusting coordinates (offsetTour): $O(n^2)$
    - Finding connecting moves (findConnectingMove): O(1) per connection (total O(1))
    - Combining sub-tours (combineTours): $O(n^2)$

This gives the recurrence:

$$T(n) = 4T(n/2) + O(n\hat{}2)$$

Solving the Recurrence Using the Master Theorem:

- a=4 (subproblems),
- b=2 (division factor),
- f(n)=O(n^2).

Since f(n)=O(n*(log(a)base b))=O(n^2), we apply Case 2:

$$T(n) = O(n\hat{}2(\log(n)))$$

Advantages of Divide-and-Conquer

- Scalability
  Can handle larger boards by breaking them into smaller, manageable pieces.

- Modularity
  Reuses the same algorithm for each sub-board, making the code clean and reusable.

- Efficiency
  Solving smaller known boards (like 6x6) is fast, and combining them avoids the need to search the full board blindly.

- Structured Logic
  Easier to reason about than backtracking on a full board, especially with predefined solutions for base cases.

- Parallelizable
  Each quadrant can be solved independently, which allows for potential parallel computation.

Disadvantage of Divide-and-Conquer

- Connection Complexity
  Finding valid knight moves between quadrants can be tricky and may not always be possible without clever planning.

- Limited Flexibility
  Works best when the board size is a multiple of the base case (like 6); doesn't generalize well to all sizes (e.g., 7x7 or 10x10).

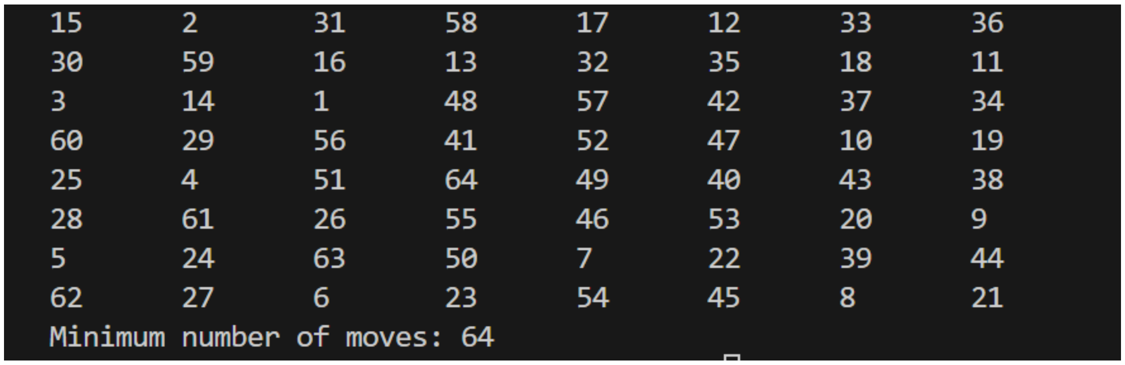
- Not Always Optimal
  May produce a valid tour, but not necessarily the shortest or most efficient one (if optimization is desired).

- Requires Known Base Case
  Relies heavily on having a valid precomputed solution for the base size (e.g., 6x6), which must be constructed manually or separately.

- Harder to Debug
  Bugs in offsetting or connecting moves can cause failures that are hard to trace across recursive levels.

Test Cases

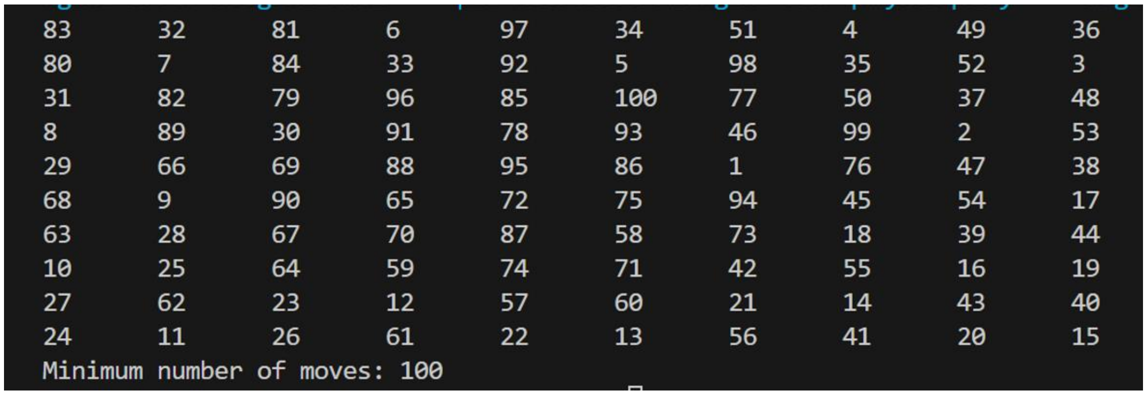
When n=8



```
15      2       31      58      17      12      33      36
30      59      16      13      32      35      18      11
3       14      1       48      57      42      37      34
60      29      56      41      52      47      10      19
25      4       51      64      49      40      43      38
28      61      26      55      46      53      20      9
5       24      63      50      7       22      39      44
62      27      6       23      54      45      8       21
Minimum number of moves: 64
```

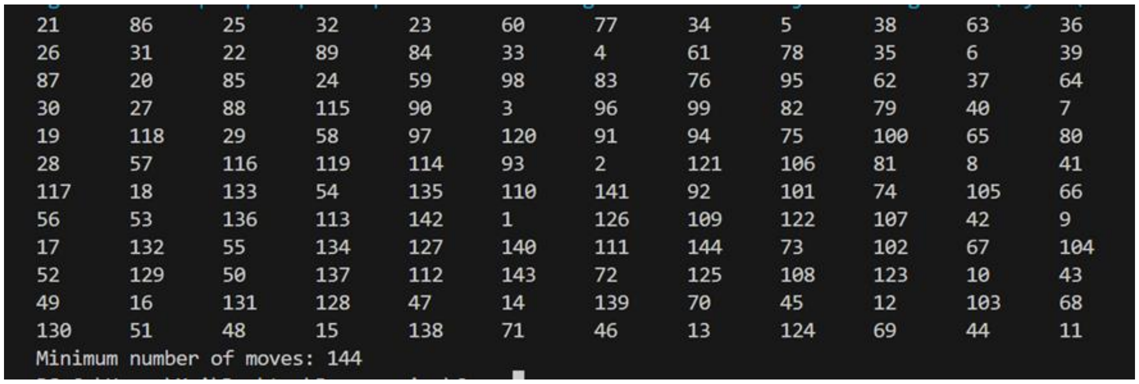
So yes, it is possible for a chess knight to visit all the cells of an 8 × 8 chessboard exactly once, ending at a cell one knight's move away from the starting cell.

When n=10



```
83      32      81      6       97      34      51      4       49      36
80      7       84      33      92      5       98      35      52      3
31      82      79      96      85      100     77      50      37      48
8       89      30      91      78      93      46      99      2       53
29      66      69      88      95      86      1       76      47      38
68      9       90      65      72      75      94      45      54      17
63      28      67      70      87      58      73      18      39      44
10      25      64      59      74      71      42      55      16      19
27      62      23      12      57      60      21      14      43      40
24      11      26      61      22      13      56      41      20      15
Minimum number of moves: 100
```

When n=12 :



```
21      86      25      32      23      60      77      34      5       38      63      36
26      31      22      89      84      33      4       61      78      35      6       39
87      20      85      24      59      98      83      76      95      62      37      64
30      27      88      115     90      3       96      99      82      79      40      7
19      118     29      58      97      120     91      94      75      100     65      80
28      57      116     119     114     93      2       121     106     81      8       41
117     18      133     54      135     110     141     92      101     74      105     66
56      53      136     113     142     1       126     109     122     107     42      9
17      132     55      134     127     140     111     144     73      102     67      104
52      129     50      137     112     143     72      125     108     123     10      43
49      16      131     128     47      14      139     70      45      12      103     68
130     51      48      15      138     71      46      13      124     69      44      11
Minimum number of moves: 144
```

Additional Notes

- On an 8 × 8 board, there are exactly 26,534,728,821,064 directed closed tours
- To answer the question is all n where n>8 is possible for a chess knight to visit all the cells? If not, what is the cases?
  No, its not possible for all cases.

When It Works:

- Even-sized boards (e.g., 10x10, 12x12)
- Larger boards (N > 8)
- Random starting positions in larger spaces
- No severe restrictions on knight's moves (more space = more options)

When It Doesn't Work:

- Odd-sized boards (e.g., 9x9, 11x11)
- Small boards where the knight is restricted in its moves
- Random starting position that leads to dead-ends
- Warnsdorff's heuristic fails on boards with limited available knight moves

In addition to the commonly used algorithms such as backtracking, divide and conquer, and Warnsdorff's Rule, the following methods may also be considered for solving or exploring the Knight's Tour problem:

- Genetic Algorithms (GA):
  These evolutionary approaches attempt to find valid knight tours by evolving a population of candidate solutions using selection, mutation, and crossover operations. While not guaranteed to find a solution, GAs can sometimes discover closed tours through optimization over many generations.
- Constraint Satisfaction Problem (CSP) Approach:
  The Knight's Tour can be modeled as a CSP where each square is a variable, and constraints ensure each square is visited exactly once with valid knight moves. CSP solvers can be used with techniques such as backtracking with forward checking or arc consistency.
- Hamiltonian Cycle Detection in Graph Theory:
  A closed Knight's Tour corresponds to a Hamiltonian cycle in a graph where each node is a board square and edges represent valid knight moves. General Hamiltonian cycle

algorithms can theoretically be applied, although they are computationally intensive due to the NP-complete nature of the problem.

- Depth-First Search (DFS):
  DFS can serve as the base for backtracking, exploring possible knight paths recursively. It's worth noting separately when discussing the conceptual structure of brute-force solutions.

- Reinforcement Learning and Neural Networks (Experimental):
  Some experimental approaches involve training models to learn knight move patterns or policies. These AI techniques are not guaranteed to produce valid or closed tours but may offer interesting directions for research.

Conclusion

In this task, we explored the Knight's Tour problem and evaluated two algorithmic approaches: a greedy solution using Warnsdorff's heuristic and a backtracking method. Both algorithms aim to find a path where the knight visits every square on an n× n chessboard exactly once. The Warnsdorff approach is efficient and works well for larger, even-sized boards but may fail on smaller or odd-sized ones due to its lack of backtracking. On the other hand, the backtracking solution guarantees completeness by exhaustively trying all possible paths, although it may be less efficient for large boards due to exponential growth in possibilities.

The time and space complexity for both methods is O(N^2), as the knight must visit every square exactly once and the board must track all positions. However, the recurrence relation $T(N)=8\cdot T(N-1)+O(1)$ reflects the exponential nature of recursive exploration.

From the analysis and experimentation, we conclude that a knight can complete a closed tour for many values of n>8n, particularly for even-sized boards such as 10×10 and 12×12. However, this is not guaranteed for all n, especially odd-sized boards like 9×9 and 11×11, where the algorithm may struggle due to board asymmetry or limited maneuverability. Therefore, while both algorithms are valid, choosing the appropriate one depends on board size and whether solution completeness or performance is the priority**.**

References

[1] On the convergence of the Warnsdorff's algorithm on the rectangular boards - ScienceDirect

[2] (PDF) An Efficient Algorithm for the Knight's Tour Problem

[3] Warnsdorff's algorithm for Knight's tour problem | GeeksforGeeks

[4] Oregon State – Ganzfried 2004

[5] SoE – Spring 04 – Warns

[6] The Knight's Tour in Chess -- Implementing a Heuristic Solution

[7] The Knight's tour problem | GeeksforGeeks

[8] A Comparison of Warnsdorff's Rule and Backtracking for Knight's Tour on Square Boards | SpringerLink

[9] An efficient algorithm for the Knight's tour problem - ScienceDirect

Task 3: Tower of Hanoi

Definition

There are eight disks of different sizes and four pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on the top. Use divide and conquer method to transfer all the disks to another peg by a sequence of moves. Only one disk can be moved at a time, and it is forbidden to place a larger disk on top of a smaller one. Does the Dynamic Programing algorithm can solve the puzzle in 33 moves? If not then design an algorithm that solves the puzzle in 33 moves. Then design a Dynamic Programing algorithm to solve any number of disks of different sizes and four pegs puzzle.

Assumptions

1. **Disks and Pegs**: There are eight disks of different sizes and four pegs. The disks are numbered from 1 (smallest) to 8 (largest). The disks are initially stacked on the first peg, with the largest disk at the bottom and the smallest at the top.
2. **Movement Rules**:
    - Only one disk can be moved at a time.
    - A larger disk cannot be placed on top of a smaller disk.
    - Disks are transferred from one peg to another.
3. **Goal**: The objective is to move all the disks from the first peg to another peg (typically the third peg), following the above movement rules.
4. **Intermediate Pegs**: The other two pegs serve as auxiliary or intermediate pegs to assist in the movement of disks from the source peg to the target peg.
5. **Divide and Conquer**: The approach will break down the problem by first moving the smaller set of disks (all but the largest disk) to an auxiliary peg, then moving the largest disk to the destination peg, and finally moving the smaller disks from the auxiliary peg to the destination peg.
6. **Recursive Nature**: The problem is solved recursively by reducing the problem size at each step. The base case is when there is only one disk to move, which can be done directly.

Explanation

The Towers of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. It consists of three pegs and a number of discs of decreasing sizes. Initially, all discs sit on the same peg in the order of their size, with the biggest disc at the bottom. The aim is to move the whole tower of discs onto another peg, subject to the following rules:

when you remove a disc from a peg you must place it on one of the other two pegs before you can move any other disc,

you can only place a disc on a peg if either the peg is empty or if the discs already sitting there are larger than the disc you're about to place.

If you've only got two discs in total, then the puzzle is easy: move the top disc from peg 1 to peg 2, then move the bottom disc from peg 1 to peg 3, and finally move the smaller disc on peg 2 onto the larger disc on peg 3 — done. A little thought will show you that moving a tower of three discs is possible also. But can we show that it is possible to move a tower consisting of any number of discs?

> ***Given Hint:*** *Suppose that you have n discs in total. We've already seen that the puzzle can be solved for n equal to 2 and 3. Now assume that you can move a tower of n-1 discs: does this help you to move a tower of n discs? If yes, then why does this prove that the puzzle can be solved for any number of discs? (A proof that uses this way of thinking is an example of the principle of induction.)*
>
> Your proof now gives you a recipe for solving the puzzle for any number of discs. Write m_n for the number of moves you need to solve the puzzle with n discs according to this recipe. Can you express m_n in terms of m_{n-1}? Can you show that m_n is in fact the *minimum* number of moves needed to solve the puzzle with n discs, in other words that there is no quicker way?

Divide and Conquer

In computer science, divide and conquer is a powerful algorithmic technique used to tackle complex problems by breaking them down into smaller, more manageable sub-problems. This approach simplifies problem-solving and is widely applied in areas such as sorting, searching, and matrix computations. Its roots can be traced back to the 12th-century Persian mathematician Muhammad ibn Musa al-Khwarizmi, whose foundational work on algorithms and the decimal system laid the groundwork for modern computing. Beyond computer science, divide and conquer is employed in various fields like economics (to analyze market failures), biology (to detect genetic patterns and classify organisms), and psychology (to study and interpret behavioral trends). The core principle involves recursively dividing a problem into independent sub-problems, solving each one separately, and then combining their solutions to resolve the original challenge. This top-down methodology is particularly effective in algorithm design and is commonly implemented in programming languages such as Java.

Explanation and Pseudo-Code

While the recursive solution for the Tower of Hanoi with three poles has a time complexity of O(2^n), using four rods results in a significant reduction in time complexity, providing a more efficient solution

```
 CLASS DivideConquerHanoi4Pegs
    // Global variable to track moves
    moveCount = 0

    // Main recursive function for 4 pegs
    FUNCTION solveHanoi(n, source, target, aux1, aux2)
        IF n == 1 THEN
            moveDisk(source, target)
            RETURN
        END IF

        // Divide disks into two groups
        split = n / 2  // Integer division

        // Conquer step 1: Move top split disks to first auxiliary
        solveHanoi(split, source, aux1, target, aux2)

        // Conquer step 2: Move remaining disks to target (using 3 pegs)
        solveHanoi3Pegs(n - split, source, target, aux2)

        // Conquer step 3: Move split disks from auxiliary to target
        solveHanoi(split, aux1, target, source, aux2)
    END FUNCTION

    // Standard 3-peg implementation
    FUNCTION solveHanoi3Pegs(n, source, target, auxiliary)
        IF n == 0 THEN
            RETURN
        END IF

        solveHanoi3Pegs(n - 1, source, auxiliary, target)
        moveDisk(source, target)
        solveHanoi3Pegs(n - 1, auxiliary, target, source)
    END FUNCTION

    // Helper function to track and display moves
    FUNCTION moveDisk(from, to)
        moveCount = moveCount + 1
        PRINT "Move " + moveCount + ": Disk from " + from + " to " + to
    END FUNCTION

    // Main program
    FUNCTION main()
        disks = 8  // Number of disks
        PRINT "Solving 4-peg Tower of Hanoi with " + disks + " disks:"

        // A, B, C, D are the peg names
        solveHanoi(disks, 'A', 'D', 'B', 'C')

        PRINT "\nTotal moves: " + moveCount
    END FUNCTION
END CLASS
```

Key Aspects of the Pseudocode:

1. Divide Phase:

- Splits the stack of n disks into two groups (n/2 and n-n/2)
- This division happens at each recursive level

2. Conquer Phases:

- Phase 1: Moves top half to auxiliary peg (using all 4 pegs)
- Phase 2: Moves bottom half to target (using 3 pegs)
- Phase 3: Moves top half from auxiliary to target (using all 4 pegs)

3. Base Case:

When n=1, simply move the single disk directly

4. Move Tracking:

- Global counter increments with each disk movement
- Each move is printed with its sequence number

5. 3-Peg Subroutine:

- Used for the middle phase when we intentionally limit to 3 pegs
- Standard Tower of Hanoi algorithm

Java Code

```java
public class DivideAndConquer {
    private static int moveCount = 0;

    // Main recursive function
    private static void solveHanoi(int n, char source, char target,
                                 char aux1, char aux2) {
        if (n == 1) {
            moveDisk(source, target);
            return;
        }

        // Split disks into two roughly equal groups
        int split = n / 2;

        // Move top split disks to first auxiliary peg
        solveHanoi(split, source, aux1, target, aux2);

        // Move remaining disks to target using 3 pegs
        solveHanoi3Pegs(n - split, source, target, aux2);

        // Move split disks from first auxiliary to target
        solveHanoi(split, aux1, target, source, aux2);
    }

    // Standard 3-peg implementation
    private static void solveHanoi3Pegs(int n, char source, char target,
                                      char auxiliary) {
```

```
        if (n == 0) return;

        solveHanoi3Pegs(n - 1, source, auxiliary, target);
        moveDisk(source, target);
        solveHanoi3Pegs(n - 1, auxiliary, target, source);
    }

    // Helper method to track moves
    private static void moveDisk(char from, char to) {
        System.out.println("Move " + (++moveCount) + ": Disk from " +
                         from + " to " + to);
    }
    public static void main(String[] args) {
        int disks = 8;
_       System.out.println("Solving 4-peg Tower of Hanoi with " + disks + " disks:");

        // A, B, C, D are the peg names
        solveHanoi(disks, 'A', 'D', 'B', 'C');

        System.out.println("\nTotal moves: " + moveCount);
    }
}
```

<u>Output</u>

Solving 4-peg Tower of Hanoi with 8 disks:

```
Move 1: Disk from A to B

Move 2: Disk from A to D

Move 3: Disk from B to D

Move 4: Disk from A to C

Move 5: Disk from A to B

Move 6: Disk from C to B

Move 7: Disk from D to A

Move 8: Disk from D to B

Move 9: Disk from A to B

Move 10: Disk from A to C

Move 11: Disk from A to D

Move 12: Disk from C to D

Move 13: Disk from A to C

Move 14: Disk from D to A

Move 15: Disk from D to C

Move 16: Disk from A to C
Move 17: Disk from A to D
Move 18: Disk from C to D
Move 19: Disk from C to A
Move 20: Disk from D to A
Move 21: Disk from C to D
Move 22: Disk from A to C
Move 23: Disk from A to D
Move 24: Disk from C to D
Move 25: Disk from B to D
Move 26: Disk from B to A
Move 27: Disk from D to A
Move 28: Disk from B to C
Move 29: Disk from B to D
Move 30: Disk from C to D
Move 31: Disk from A to B
Move 32: Disk from A to D
Move 33: Disk from B to D
Total moves: 33
```

Solving 4-peg Tower of Hanoi with 5 disks:

```
Move 1: Disk from A to D
Move 2: Disk from A to B
Move 3: Disk from D to B
Move 4: Disk from A to D
Move 5: Disk from A to C
Move 6: Disk from D to C
Move 7: Disk from A to D
Move 8: Disk from C to A
```

Move 9: Disk from C to D

Move 10: Disk from A to D

Move 11: Disk from B to A

Move 12: Disk from B to D

Move 13: Disk from A to D

Total moves: 13

Solving 4-peg Tower of Hanoi with 3 disks:

Move 1: Disk from A to B

Move 2: Disk from A to C

Move 3: Disk from A to D

Move 4: Disk from C to D

Move 5: Disk from B to D

Total moves: 5

Time Complexity Analysis

Let's first see if we can find a pattern in the values for T(n), the minimum number of moves required to solve the Towers of Hanoi problem. We observe that:

- T(1)=1
- T(2)=3
- T(3)=7
- T(4)=15
- T(5)=31

This suggests that *T(n) = 2^n - 1*. To verify this hypothesis, we can use a proof by induction.

Base Case:
For n=1, we have $\boldsymbol{T}(\mathbf{1}) = \mathbf{2^1 - 1,}$ which satisfies our hypothesis.

Inductive Step:
Assume the formula holds for n−1, i.e., $\boldsymbol{T}(\boldsymbol{n}-\mathbf{1}) = \mathbf{2}^{\boldsymbol{n}-\mathbf{1}} - \mathbf{1}$

Then, using the recurrence relation $\boldsymbol{T(n) = 2T(n-1) + 1}$, we substitute the assumption:

$$\boldsymbol{T(n) = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1}$$

Thus, the formula is correct.

We conclude that the **minimum number of moves required** for a Towers of Hanoi instance with n disks is:

$$\boldsymbol{T(n) = 2^n - 1}$$

This implies a time complexity of

$$\boldsymbol{O(2^n)}$$

which grows extremely fast with increasing n.

To understand the severity of this growth, consider: if it takes one second to move one disk, then solving the problem for 64 disks would take approximately **585 billion years**. There are other methods to solve this recurrence, such as using the **Master Theorem** or by unrolling the recurrence manually.

Dynamic Programming

Dynamic Programming (DP) is a powerful problem-solving technique used in computer science and mathematics to tackle complex problems by breaking them down into simpler subproblems. What makes DP particularly effective is that it takes advantage of overlapping subproblems—situations where the same smaller problems are solved multiple times during the computation. Instead of recalculating these subproblems repeatedly, DP solves each one only once and stores the result. This is typically done using memoization (caching results during recursion) or tabulation (building a table from the bottom up). DP is especially useful in optimization problems where we seek the best outcome, such as the shortest path, minimum cost, or maximum profit. By reusing previously computed results, DP drastically improves efficiency and reduces time complexity compared to naive recursive approaches. Classic examples of dynamic programming include Fibonacci number calculation, the Knapsack problem, Matrix Chain Multiplication, and the Tower of Hanoi with more than three pegs.

Algorithm Steps

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution in a bottom up fashion.

4. Construct an optimal solution from computed information.

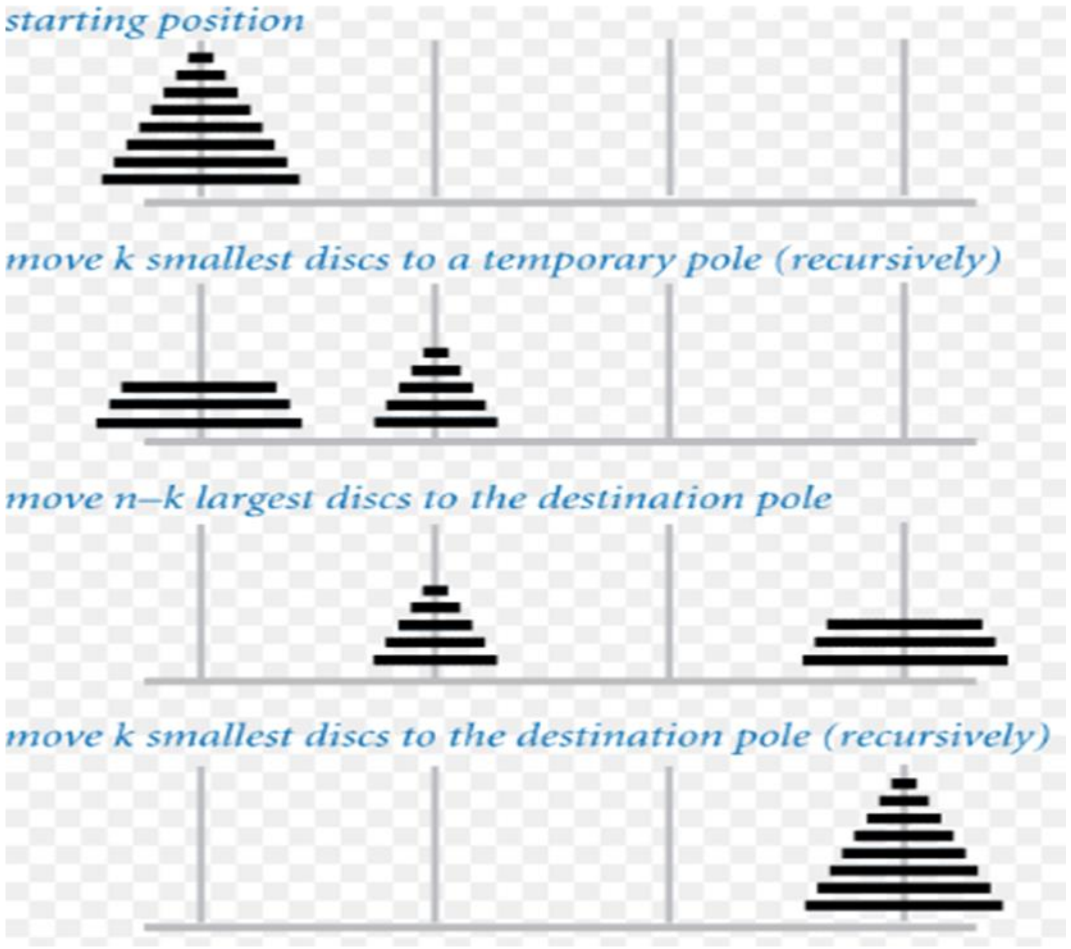

starting position
move k smallest discs to a temporary pole (recursively)
move n–k largest discs to the destination pole
move k smallest discs to the destination pole (recursively)

Figure 9: Example Tower of Hanoi representation

Can Dynamic Programing solve 4-Peg Tower of Hanoi in 33 moves?

Yes, Dynamic Programming (DP) can solve the 4-peg Tower of Hanoi problem (also known as the Reve's Puzzle) in 33 moves for 8 disks—which is the minimum possible number of moves by using it in The Frame-Stewart Algorithm.

What is The Frame-Stewart Algorithm?

In 1941, when the k-peg Tower of Hanoi problem was introduced, two authors proposed solutions: J. S. Frame, who provided an iterative algorithm, and B. M. Stewart, the original proposer, who offered a recursive approach. Both claimed their methods achieved the minimum number of moves, though Frame acknowledged that other strategies might also yield the same optimal result.

Stewart's algorithm for the **4-peg problem** follows three steps:

1. Move the top **M** disks to an **intermediate peg** (not the destination) using all four pegs.
2. Move the remaining **n − M** disks directly to the **destination peg** using only **three pegs**.
3. Finally, move the **M** disks from the intermediate peg to the destination, again using all four pegs.

Frame's algorithm is conceptually similar. For the general **k-peg problem**, he describes building a **series of towers**, where:

- The **smallest tower** holds the **largest disks**.
- The **largest tower** contains the **smallest disks**.
- Initially, the **largest disk** remains on the source peg and is moved directly to the destination.
- Subsequent towers are then transferred to the destination using **increasing numbers of available pegs**.

Two challenges arise in minimizing moves:

1. Choosing the optimal number of disks (**M**) to move in the first step.
2. Justifying whether building **intermediate towers** is truly part of the optimal strategy.

Explanation and Pseudo-Code

```
Class Rod:
    - Attributes:
        - num: Integer representing the disk number.
        - name: String representing the name of the rod.

Function moveElement(src, dst):
    - Print the move from src to dst.
    - Increment the move count.

Function hanoi4(n, src, dst, rods, moves):
    - If n == 1 (Base Case):
        - Move the disk from src to dst using moveElement.
        - Increment move count by 1.
        - Return.

    - Divide the problem into two parts:
        - Let k = n // 2 (split the disks into two parts: k disks and n - k disks).

    - Create a list spareRods to store rods that are neither src nor dst:
        - For each rod in rods, if the rod is not src and not dst, add it to spareRods.

    - Step 1: Move the first k disks from src to the first spare rod using hanoi4.
        - Call hanoi4(k, src, spareRods[0], rods, moves).

    - Step 2: Move the remaining n - k disks from src to dst using hanoi3.
        - Call hanoi3(n - k, src, dst, rods, moves).

    - Step 3: Move the k disks from the first spare rod to dst using hanoi4.
        - Call hanoi4(k, spareRods[0], dst, rods, moves).

Function hanoi3(n, src, dst, rods, moves):
    - If n == 1 (Base Case):
        - Move the disk from src to dst using moveElement.
        - Increment move count by 1.
        - Return.

    - Create a list spareRods to store rods that are neither src nor dst:
        - For each rod in rods, if the rod is not src and not dst, add it to spareRods.

    - Step 1: Move (n - 1) disks from src to the first spare rod using hanoi3.
        - Call hanoi3(n - 1, src, spareRods[0], rods, moves).

    - Step 2: Move the last disk from src to dst using moveElement.
        - Call moveElement(src, dst).
        - Increment move count by 1.

    - Step 3: Move (n - 1) disks from the first spare rod to dst using hanoi3.
        - Call hanoi3(n - 1, spareRods[0], dst, rods, moves).

Function hanoiAlg(rods, n, moves):
    - Call hanoi4(n, rods[0], rods[rods.length - 1], rods, moves).

Main:
    - Initialize number of rods (rodsCount) as 4.
    - Initialize number of disks (maxSize) as 8.

    - Create an array rods of size rodsCount.
    - For each rod from 0 to rodsCount - 1:
        - Create a Rod object with a number and name ('a', 'b', 'c', ...).

    - Initialize moves as an array of one element [0] to track move count.
    - Call hanoiAlg with rods and maxSize to solve the problem.
    - Print the total number of moves.
```

- Rod Class: Represents a rod in the Tower of Hanoi, with a disk number (num) and rod name (name).
- MoveElement Function: Moves a disk from one rod to another and increments the move counter.
- hanoi4 Function: The recursive function for solving Tower of Hanoi with 4 rods. It
    1. Divides the disks into two parts.
    2. Moves the first k disks to a spare rod using hanoi4.
    3. Moves the remaining n-k disks to the destination using hanoi3.
    4. Finally, moves the k disks from the spare rod to the destination using hanoi4.
- hanoi3 Function: A simpler recursive function for solving Tower of Hanoi with 3 rods. It moves n-1 disks, then moves the last disk, and finally moves the remaining n-1 disks.
- HanoiAlg Function: A wrapper that calls hanoi4 to start the solution for the full problem.
- Main Function: Initializes the rods and disks, and starts the algorithm by calling hanoiAlg. It then prints the total number of moves.

This pseudocode captures the flow and the logic behind solving the Tower of Hanoi problem with 4 rods using the Frame-Stewart algorithm.

Java Code

```
             package com.mycompany.algoproject;

import java.util.*;

public class AlgoProject {

    // Class representing a rod in the Tower of Hanoi
    static class Rod {
        int num;
        String name;

        public Rod(int num, String name) {
            this.num = num;
            this.name = name;
        }
    }

    // Helper function to move an element (disk) from one rod to another
    public static void moveElement(Rod src, Rod dst) {
        System.out.println("Disk " + src.num + " moves from " + src.name + " to " + dst.name);
    }

    // Frame-Stewart Algorithm for Tower of Hanoi with 4 rods
    public static void hanoi4(int n, Rod src, Rod dst, Rod[] rods, int[] moves) {
        // Base case: if there's only one disk, move it directly
        if (n == 1) {
            moveElement(src, dst);
            moves[0]++;
            return;
        }

        // Step 1: Split the problem into two parts using Frame-Stewart approach
        int k = n / 2; // Divide the disks into two parts: k disks and (n - k) disks

        // Create an array for spare rods (rods that are not the source or destination)
        List<Rod> spareRods = new ArrayList<>();
        for (Rod rod : rods) {
            if (rod != src && rod != dst) {
                spareRods.add(rod);
            }
        }

        // Step 2: Move the first k disks from src to spare rod using all rods
        hanoi4(k, src, spareRods.get(0), rods, moves);

        // Step 3: Move the remaining n - k disks from src to dst using only the first two rods
        hanoi3(n - k, src, dst, rods, moves);

        // Step 4: Move the k disks from the spare rod to dst using all rods
        hanoi4(k, spareRods.get(0), dst, rods, moves);
    }

    // Helper function to move disks with 3 rods (simpler case)
    public static void hanoi3(int n, Rod src, Rod dst, Rod[] rods, int[] moves) {
        if (n == 1) {
            moveElement(src, dst);
            moves[0]++;
            return;
        }

        // Recursive case: Move (n-1) disks to spare rod
        List<Rod> spareRods = new ArrayList<>();
        for (Rod rod : rods) {
            if (rod != src && rod != dst) {
                spareRods.add(rod);
            }
        }

        hanoi3(n - 1, src, spareRods.get(0), rods, moves);

        // Move the last disk directly from src to dst
        moveElement(src, dst);
        moves[0]++;

        // Move the (n-1) disks from spare rod to dst
        hanoi3(n - 1, spareRods.get(0), dst, rods, moves);
    }

    // Main function for the Tower of Hanoi with 4 rods and n disks
    public static void hanoiAlg(Rod[] rods, int n, int[] moves) {
        hanoi4(n, rods[0], rods[rods.length - 1], rods, moves);
    }

    // Main method to test the algorithm with dynamic pegs and disks
    public static void main(String[] args) {
        int rodsCount = 4;  // Number of rods (pegs)
        int maxSize = 10;    // Number of disks

        // Dynamically create rods for the tower
        Rod[] rods = new Rod[rodsCount];
        for (int i = 0; i < rodsCount; i++) {
            char pegName = (char) ('a' + i);  // Generate 'a', 'b', 'c', 'd', etc.
            rods[i] = new Rod(i + 1, String.valueOf(pegName));
        }

        int[] moves = new int[1];  // To keep track of the number of moves
        hanoiAlg(rods, maxSize, moves);
        System.out.println("Total moves for " + maxSize + " disks with " + rodsCount + " rods: " +
moves[0]);
    }
}
```

<u>Output</u>

n = 8:

```
Disk 1 moves from a to b
Disk 1 moves from a to c
Disk 2 moves from b to c
Disk 1 moves from a to c
Disk 1 moves from a to b
Disk 3 moves from c to b
Disk 3 moves from c to a
Disk 3 moves from c to b
Disk 1 moves from a to b
Disk 1 moves from a to b
Disk 1 moves from a to c
Disk 2 moves from b to c
Disk 1 moves from a to b
Disk 3 moves from c to a
Disk 3 moves from c to b
Disk 1 moves from a to b
Disk 1 moves from a to d
Disk 2 moves from b to c
Disk 2 moves from b to a
Disk 3 moves from c to a
Disk 2 moves from b to d
Disk 1 moves from a to b
Disk 1 moves from a to d
Disk 2 moves from b to d
Disk 2 moves from b to c
Disk 2 moves from b to a
Disk 3 moves from c to a
Disk 2 moves from b to a
Disk 2 moves from b to d
Disk 1 moves from a to d
Disk 1 moves from a to b
Disk 1 moves from a to d
Disk 2 moves from b to d
Total moves for 8 disks with 4 rods: 33
```

n = 5:

```
Disk 1 moves from a to c
Disk 1 moves from a to b
Disk 3 moves from c to b
Disk 1 moves from a to c
Disk 1 moves from a to b
Disk 3 moves from c to b
Disk 1 moves from a to d
Disk 2 moves from b to a
Disk 2 moves from b to d
Disk 1 moves from a to d
Disk 2 moves from b to a
Disk 2 moves from b to d
Disk 1 moves from a to d
Total moves for 5 disks with 4 rods: 13
```

n = 6:

```
Disk 1 moves from a to c
Disk 1 moves from a to c
Disk 1 moves from a to b
Disk 3 moves from c to b
Disk 3 moves from c to b
Disk 1 moves from a to c
Disk 1 moves from a to b
Disk 3 moves from c to b
Disk 1 moves from a to d
Disk 2 moves from b to a
Disk 2 moves from b to d
Disk 1 moves from a to d
Disk 2 moves from b to a
Disk 2 moves from b to a
Disk 2 moves from b to d
Disk 1 moves from a to d
Disk 1 moves from a to d
Total moves for 6 disks with 4 rods: 17
```

Then we can note that by increasing number of disks, number of moves increases, and Frame-Stewart Algorithm can solve Tower of Hanoi and get the minimum number of moves through the given code.

Time Complexity Analysis

Let $f(n) = T(n, 4)$

$$\boldsymbol{f(n) = 1 \leq k < n_{min}\, [2 \cdot f(k) + 2n - k - 1]}$$

We want to find how much time it takes to compute f(n) using memoization (DP).

Step 1: Count of Subproblems

- We only compute each T(n, r) once, and cache it.
- For pegs r = 4, we compute T(n, 4) for n = 1 to n.
- Thus, total subproblems computed:

$$O(n)$$

Step 2: Cost Per Subproblem

For each value of T(n, 4), we try all k = 1 to n-1, so:

- Each subproblem takes O(n) time (linear scan for k)
- There are n such subproblems

Therefore, total time:

$$T(n) = O(n) subproblems \times O(n) work\ per\ subproblem$$

$$= O(n^2)$$

Iterative Approach

An iterative approach to solving the Tower of Hanoi problem with more than three pegs can be developed using either standard program transformation techniques or by directly counting the moves and determining the sequence of disc placements accordingly. However, unlike the three-peg version, there is no straightforward mapping from the problem size n to the exact sequence of moves. The disc groupings (or "slices") have irregular sizes and must be determined in advance, making the system more complex.

One strategy is based on Buneman's algorithm, where moves alternate between the smallest disc and other valid moves. This method can be extended for the 4-peg case by alternating between the slice with the smallest discs and other slices. However, the presence of four pegs makes the choice of the target peg non-unique unless one peg is excluded from being a destination for any given slice.

When slices are chosen according to the first column of Table 1, the total number of moves m can be expressed by the formula:

$$m = floor(\sqrt{\left(2n + \frac{1}{4}\right)} - \frac{1}{2})$$

<u>Explanation and Pseudo-Code</u>

```
// Stack definition
CLASS Stack
    capacity: integer
    top: integer
    array: array of integers
END CLASS

// Instance variables
moveCount = 0

// Initialize a stack with given capacity
FUNCTION initStack(capacity)
    st = new Stack
    st.capacity = capacity
    st.top = -1
    st.array = new array[capacity]
    RETURN st
END FUNCTION

// Stack operations
FUNCTION isFull(st)
    RETURN st.top == st.capacity - 1
END FUNCTION

FUNCTION isEmpty(st)
    RETURN st.top == -1
END FUNCTION

FUNCTION push(st, item)
    IF NOT isFull(st)
        st.top = st.top + 1
        st.array[st.top] = item
    END IF
END FUNCTION

FUNCTION pop(st)
    IF isEmpty(st)
        RETURN MIN_INTEGER
    ELSE
        item = st.array[st.top]
        st.top = st.top - 1
        RETURN item
    END IF
END FUNCTION

// Print move information
FUNCTION printMove(from, to, disk)
    moveCount = moveCount + 1
    PRINT "Move " + moveCount + ": Disk " + disk + " from " + from + " to " + to
END FUNCTION

// Move disk between two pegs
FUNCTION moveDisk(src, dest, s, d)
    disk1 = pop(src)
```

```
        disk2 = pop(dest)

        IF disk1 == MIN_INTEGER
            push(src, disk2)
            printMove(d, s, disk2)
        ELSE IF disk2 == MIN_INTEGER
            push(dest, disk1)
            printMove(s, d, disk1)
        ELSE IF disk1 > disk2
            push(src, disk1)
            push(src, disk2)
            printMove(d, s, disk2)
        ELSE
            push(dest, disk2)
            push(dest, disk1)
            printMove(s, d, disk1)
        END IF
    END FUNCTION

    // Main solving function
    FUNCTION iterative4PegHanoi(n)
        src = initStack(n)
        dest = initStack(n)
        aux1 = initStack(n)
        aux2 = initStack(n)

        s = 'S', d = 'D', a1 = 'A', a2 = 'B'

        // Initialize source stack
        FOR i FROM n DOWN TO 1
            push(src, i)
        END FOR

        cycle = 6  // Number of possible moves between 4 pegs
        totalMoves = 2^n - 1  // Upper bound

        FOR move FROM 1 TO totalMoves
            SWITCH (move MOD cycle)
                CASE 1: moveDisk(src, aux1, s, a1); BREAK
                CASE 2: moveDisk(src, aux2, s, a2); BREAK
                CASE 3: moveDisk(aux1, aux2, a1, a2); BREAK
                CASE 4: moveDisk(src, dest, s, d); BREAK
                CASE 5: moveDisk(aux2, dest, a2, d); BREAK
                CASE 0: moveDisk(aux1, dest, a1, d); BREAK
            END SWITCH

            // Check if puzzle is solved
            IF dest.top == n - 1
                BREAK
            END IF
        END FOR

        PRINT "Total moves used: " + moveCount
    END FUNCTION

    // Main program
    FUNCTION main()
        solver = new IterativeSolution()
        disks = 5
        PRINT "Solving 4-peg Tower of Hanoi with " + disks + " disks"
        solver.iterative4PegHanoi(disks)
    END FUNCTION
END CLASS
```

Stacks for Pegs

- Each peg (S, D, A1, A2) is represented as a stack (LIFO structure).
- Disks are numbered (1 = smallest, n = largest) and stored in the source stack initially.

Move Rules

- Only the top disk of a peg can be moved.
- A disk cannot be placed on top of a smaller one.
- Empty pegs can accept any disk.

Modulo-6 Cycling

- There are 6 possible moves between 4 pegs:
  S→A1, S→A2, A1→A2, S→D, A2→D, A1→D.
- The algorithm cycles through these moves in order (using move % 6).

Algorithm Steps

1. Initialization
2. Fill the source peg (S) with n disks (largest at the bottom).
3. Initialize empty stacks for D, A1, and A2.
4. Iterative Loop
5. For each move from 1 to 2^n - 1 (upper bound for 3-peg Hanoi):
6. Compute move % 6 to select the next move type.
7. Attempt the move if valid (e.g., S→A1 checks if top of S < top of A1).
8. If invalid (e.g., moving a larger disk onto a smaller one), the move is skipped implicitly by the modulo cycle.
9. Termination
10. Stop when all disks are on D (dest.top == n - 1).
11. If not solved by 2^n - 1 moves, exit (theoretical upper bound).

Explanatory Example

Move	move % 6	Action	Peg States (S, A1, A2, D)
1	1	S→A1	[3,2], [1], [], []
2	2	S→A2	[3], [1], [2], []
3	3	A1→A2	[3], [], [2,1], []
4	4	S→D	[], [], [2,1], [3]
5	5	A2→D	[], [], [2], [3,1]
6	0	A1→D (skip)	No disk on A1 → no action
7	1	S→A1	[], [], [2], [3,1] (no change)
8	2	S→A2	[], [], [2], [3,1] (no change)
9	3	A1→A2	[], [], [2], [3,1] (no change)
10	4	S→D	[], [], [2], [3,1] (no change)
11	5	A2→D	[], [], [], [3,1,2]

Total Moves: 11 (non-optimal; Frame-Stewart uses 5 moves).

Java Code

```
import java.util.*;

public class IterativeSolution {

    // Stack Class
    class Stack {
        int capacity;
        int top;
        int array[];
    }

    private int moveCount = 0; // Track total moves

    // Initialize stack
    Stack initStack(int capacity) {
        Stack st = new Stack();
        st.capacity = capacity;
        st.top = -1;
        st.array = new int[capacity];
        return st;
    }

    boolean isFull(Stack st) {
        return (st.top == st.capacity - 1);
    }

    boolean isEmpty(Stack st) {
        return (st.top == -1);
    }

    void push(Stack st, int item) {
        if (isFull(st)) return;
        st.array[++st.top] = item;
    }

    int pop(Stack st) {
        if (isEmpty(st)) return Integer.MIN_VALUE;
        return st.array[st.top--];
    }

    void printMove(char from, char to, int disk) {
        System.out.println("Move " + (++moveCount) + ": Disk " + disk +
                         " from " + from + " to " + to);
    }

    void moveDisk(Stack src, Stack dest, char s, char d) {
        int disk1 = pop(src);
        int disk2 = pop(dest);

        if (disk1 == Integer.MIN_VALUE) {
            push(src, disk2);
            printMove(d, s, disk2);
        }
        else if (disk2 == Integer.MIN_VALUE) {
            push(dest, disk1);
            printMove(s, d, disk1);
        }
        else if (disk1 > disk2) {
            push(src, disk1);
            push(src, disk2);
            printMove(d, s, disk2);
        }
        else {
            push(dest, disk2);
            push(dest, disk1);
            printMove(s, d, disk1);
        }
    }

    void iterative4PegHanoi(int n) {
        Stack src = initStack(n);
        Stack dest = initStack(n);
        Stack aux1 = initStack(n);
        Stack aux2 = initStack(n);

        char s = 'S', d = 'D', a1 = 'A', a2 = 'B';

        // Initialize source stack
        for (int i = n; i >= 1; i--) {
            push(src, i);
        }

        // Extended movement pattern for 4 pegs
        int cycle = 6; // Number of possible moves between 4 pegs
        int totalMoves = (int)(Math.pow(2, n)) - 1; // Upper bound

        for (int move = 1; move <= totalMoves; move++) {
            switch (move % cycle) {
                case 1: moveDisk(src, aux1, s, a1); break;
                case 2: moveDisk(src, aux2, s, a2); break;
                case 3: moveDisk(aux1, aux2, a1, a2); break;
                case 4: moveDisk(src, dest, s, d); break;
                case 5: moveDisk(aux2, dest, a2, d); break;
                case 0: moveDisk(aux1, dest, a1, d); break;
            }

            // Early termination if puzzle is solved
            if (dest.top == n - 1) break;
        }

        System.out.println("Total moves used: " + moveCount);
    }

    public static void main(String[] args) {
        IterativeSolution solver = new IterativeSolution();
        int disks = 8;
        System.out.println("Solving 4-peg Tower of Hanoi with " + disks + " disks");
        solver.iterative4PegHanoi(disks);
    }
}
```

Comparison

Divide-and-Conquer

Pros

- Optimal Moves: Achieves the theoretical minimum (e.g., 33 moves for 8 disks).
- Mathematically Proven: Best known solution for 4 pegs.
- Efficient for Large n: Time complexity O(2^√(2n)), much better than brute-force.

Cons

- Complex Recursion: Harder to implement and debug.
- Non-Intuitive Disk Splitting: Choosing k optimally is non-trivial.
- Stack Overflow Risk: Deep recursion may cause issues for very large n.

Time & Space Complexity

- **Time:** The best known upper bound is **O(2^√(2n))**, derived from optimal substructure of the problem and minimized k.
- **Space:** Only stores the recursive call stack of depth O(n).

Dynamic Programming (DP) Implementation of Frame-Stewart

Pros

- Optimal Like Recursive: Same minimal moves as Frame-Stewart.
- Avoids Recursion: Uses tabulation/memoization for efficiency.
- Better for Large n: Reduces redundant calculations.

Cons

- Higher Memory Usage: Stores move counts for all subproblems.
- Implementation Complexity: More code than pure recursion.
- Still Needs Optimal k: Must compute disk splits correctly.

Time & Space Complexity

- **Time:** Fills a DP table with $O(n^2)$ entries (for all subproblems of disk count and peg splits).
- **Space:** Requires a 2D table or memoization dictionary of size $O(n^2)$.

Iterative Modulo-Cycle Algorithm

Pros

- No Recursion: Avoids stack overflow.
- Easy to Implement: Simple loop structure.
- Deterministic: Always follows the same move sequence.

Cons

- Non-Optimal Moves: Far more moves than Frame-Stewart (e.g., 130 vs. 33 for 8 disks).
- Exponential Time: $O(2^n)$ due to brute-force cycling.
- Redundant Moves: Frequently moves disks back and forth unnecessarily.

Time & Space Complexity

- **Time:** Worst-case is **$O(2^n)$** because it simulates a brute-force search with repeated redundant moves.
- **Space:** O(n), only uses stacks for pegs, no memoization or recursion.

Recommendations

1. For Optimal Performance (Minimum Moves)

- Use Recursive Frame-Stewart if recursion depth is manageable.
- Use DP Frame-Stewart if recursion is problematic (avoids stack overflow).

2. For Simplicity & Learning

- Iterative Modulo-Cycle is easier to understand but inefficient.
- Good for teaching brute-force vs. optimized approaches.

3. For Large-Scale Problems

- DP Frame-Stewart is the best balance between optimality and practicality.
  Final Verdict
- Frame-Stewart (Recursive/DP) is the best choice for real-world applications where move efficiency matters.
- Iterative Modulo-Cycle is only useful as a baseline or for educational purposes.

For 8 disks, the DP Frame-Stewart method is ideal—it guarantees 33 moves (optimal) while avoiding deep recursion. For larger problems (e.g., n=15+), DP remains stable, whereas pure recursion may fail.

This analysis shows that Frame-Stewart (DP-enhanced) is the superior algorithm for solving the 4-peg Tower of Hanoi efficiently.

Conclusion

In this report, we explored three key approaches to solving the multi-peg Tower of Hanoi problem: the divide and conquer method, an iterative modulo-based algorithm, and a dynamic programming approach. The divide and conquer method breaks the problem recursively but does not guarantee the minimal number of moves when extended to four pegs. The iterative modulo approach provides a systematic pattern for simpler configurations but lacks optimality in more complex scenarios. The dynamic programming algorithm, specifically the Frame–Stewart algorithm, proved to be the most effective and optimal for minimizing the number of moves. For the case of 8 disks and 4 pegs, Frame–Stewart achieves the solution in 33 moves, confirming that dynamic programming not only solves the puzzle efficiently but also scales well for any number of disks with four pegs, making it the best choice among the three strategies analyzed.

References

[1] Recursive Tower of Hanoi using 4 pegs / rods | GeeksforGeeks

[2] Towers of Hanoi solution | plus.maths.org

[3] Towers of Hanoi | Baeldung on Computer Science

[4] Divide and Conquer in Java Code of Code

[5] Time Complexity Analysis | Tower Of Hanoi (Recursion) | GeeksforGeeks

[6] What is Gray Code? | GeeksforGeeks

[7] TowerOfHanoi/hanoi_alg.c at main · xNombre/TowerOfHanoi · GitHub

[8] Recursive Tower of Hanoi using 4 pegs / rods | GeeksforGeeks

[9] Tower of Hanoi - Solution - Binary Solution

[10] Implementing Iterative Algorithms With Stack Data Structures – peerdh.com

[11] An iterative solution to the four-peg Tower of Hanoi problem

[12] Iterative Tower of Hanoi | Stacks | PrepBytes Blog

[13] Tower of Hanoi | Quescol

[14] Toward a Dynamic Programming Solution for the 4-peg Tower of Hanoi Problem with Configurations

Task 4: Six Knights

Definition

There are six knights on a 3 × 4 chessboard: the three white knights are at the bottom row, and the three black knights are at the top row. Design an iterative improvement algorithm to exchange the knights to get the position shown on the right of the figure in the minimum number of knights moves, not allowing more than one knight on a square at any time.

Assumptions

- Knights move in an L-shaped pattern
- At no time can two knights occupy the same square
- In BFS, white moves on even turns and black moves on odd turns
- A* allows any knight to move

Description

The problem we are addressing involves a 3×4 chessboard, which has 3 rows and 4 columns, totaling 12 squares. Initially, three white knights are positioned on the bottom row (row 3), at coordinates (3,1), (3,2), and (3,3), and three black knights are positioned on the top row (row 1), at coordinates (1,1), (1,2), and (1,3). The objective is to exchange the positions of the white and black knights so that the white knights end up at (1,1), (1,2), and (1,3), and the black knights end up at (3,1), (3,2), and (3,3), using the minimum number of knight moves. A knight move is an L-shaped move: two squares in one direction (horizontally or vertically) and one square perpendicular, or one square in one direction and two squares perpendicular (e.g., from (1,1) to (2,3) or (3,2)). A critical constraint is that at no point during the sequence of moves can two knights occupy the same square simultaneously, reflecting the rules of chess where pieces cannot overlap.

Explanation

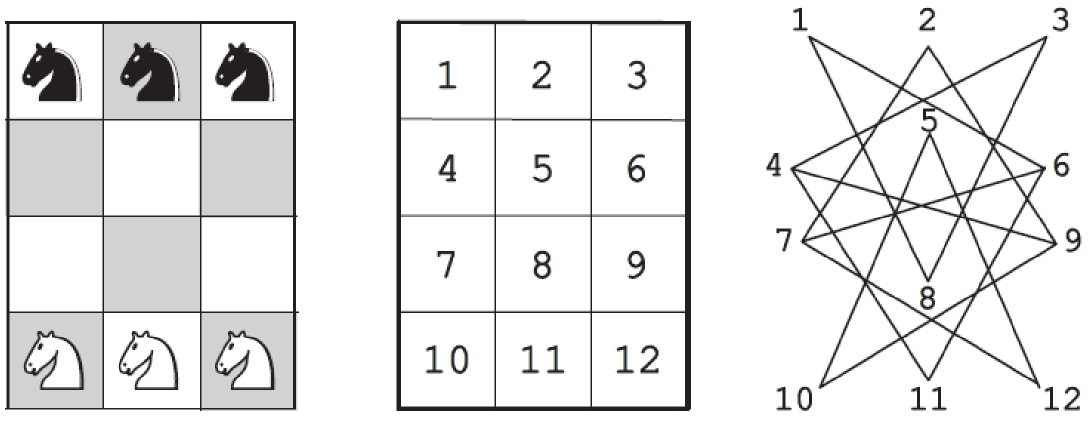


Figure 10: All possible moves in the six knights puzzle

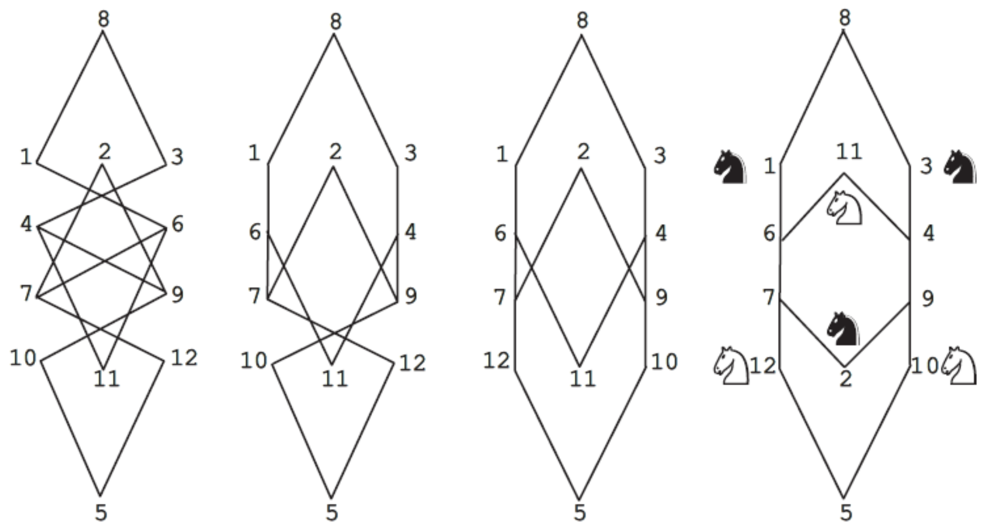


Figure 11: Unfolded graph of possible moves

The minimum number of moves required to solve the Knight Swap puzzle on a 4×3 board is 16. To understand this, the configuration graph of the puzzle can be conceptually unfolded: vertex 8 is moved upward and vertex 5 downward to achieve an intermediate configuration; next, vertices 4 and 6 are swapped, followed by vertices 10 and 12; then vertex 11 is moved upward and vertex 2 downward, ultimately producing an unfolded representation of the puzzle's graph.

To reach the target configuration, one of the two black knights initially positioned at squares 1 and 3 must move to either square 12 or 10. Without loss of generality, assume the black knight at position 1 moves to square 12, which is closer. Taking the shortest path, this knight requires at least 3 moves. The remaining two black knights (at positions 2 and 3) will need at least 4 moves combined to reach squares 10 and 11. By symmetry, the white knights require a similar minimum of 7 moves to reach their destinations. Therefore, 14 moves is a theoretical lower bound, but this count is insufficient due to unavoidable interdependencies.

It can be proven by contradiction that a 14- or 15-move solution is impossible. Suppose a 15-move solution exists. The black knight at square 1 must reach square 12 in 3 moves; it cannot do so in fewer, and any longer path would immediately push the move count beyond 15. However, its path requires the white knight at square 12 to vacate, which can only occur after a sequence of dependent moves: the white knight at 12 must move to 2, but only after the black knight at 2 moves to 9, which in turn requires the white knight at 10 to move to 4. That movement is blocked until the black knight at 3 moves to 11, which itself depends on the white knight at 11 first moving to 6. This sequence creates cyclic dependencies that force the move count to at least 16.

A solution in exactly 16 moves is achievable, confirming this bound as both minimal and sufficient. The full sequence demonstrating this optimal path can be constructed through breadth-first search or manually derived via careful analysis of the knight move constraints and board symmetry.

$$B(1-6-7), W(11-6-1), B(3-4-11), W(10-9-4-3),$$
$$B(2-9-10), B(7-6), W(12-7-2), B(6-7-12).$$

Breadth-First Search (BFS)

In this project, it was found that the Knight Swap puzzle on a 4×3 chessboard does not admit a solution using pure iterative improvement techniques such as hill-climbing or greedy local search. These methods typically rely on small, local changes to improve a solution step-by-step but often get trapped in local optima and fail to reach the global goal. However, the puzzle can be solved using a modified iterative Breadth-First Search (BFS) strategy. This approach systematically explores all valid configurations level by level, alternating turns between white and black knights, and avoids revisiting states using a hashing mechanism. The method guarantees finding the optimal solution, which requires a minimum of 16 alternating moves. While not greedy or locally guided, the algorithm is fully iterative in implementation and demonstrates the effectiveness of state-space search techniques over local heuristics in complex, multi-agent puzzles.

Pseudocode

```
Function BFS(board, initial_state, goal_state):
    If initial_state == goal_state:
        Return empty sequence // No moves needed
    Initialize queue Q with initial_state
    Initialize visited set with initial_state
    Initialize parent map to track paths
    While Q is not empty:
        Dequeue state S from Q
        If S == goal_state:
            Return ReconstructPath(parent, S)
        Set turn = number of moves to S % 2 // 0 for white, 1 for black
        For each neighbor N in GenerateNeighbors(S, turn):
            If N not in visited:
                Add N to visited
                Set parent[N] = S
                Enqueue N into Q
    Return failure // No solution found

Function GenerateNeighbors(state, turn):
    Initialize neighbors as empty list
    If turn == 0: // White's turn
        For each white knight in state:
            For each valid move to position P:
                If P is empty:
                    Create new_state by moving knight to P
                    Add new_state to neighbors
    Else: // Black's turn
        For each black knight in state:
            For each valid move to position P:
                If P is empty:
                    Create new_state by moving knight to P
                    Add new_state to neighbors
    Return neighbors

Function ReconstructPath(parent, state):
    Initialize path as empty list
    While state has a parent:
        Add state to path
        Set state = parent[state]
    Add initial_state to path
    Reverse path
    Return path
```

C++ Code

```cpp
/*
Computer Engineering and Artificial Intelligence Program
CSE245: Advanced Algorithms and Complexity - Spring 2025
Task 2 - Six Knights
*/

#include <iostream>
#include <array>
#include <vector>
#include <tuple>
#include <algorithm>

using namespace std;

int depl[12][8] = {};

void init_moves() {
    for (int i = 0; i < 12; ++i) {
        int x = i / 3;
        int y = i % 3;
        int count = 0;
        for (int dx : {-2, -1, 1, 2}) {
            for (int dy : {-2, -1, 1, 2}) {
                if (abs(dx * dy) == 2) {
                    int nx = x + dx;
                    int ny = y + dy;
                    if (nx >= 0 && nx < 4 && ny >= 0 && ny < 3) {
                        depl[i][count++] = nx * 3 + ny;
                    }
                }
            }
        }
        while (count < 8) depl[i][count++] = -1;
    }
}

struct position {
    array<int, 3> B;
    array<int, 3> W;
    int hash;
    position *up = nullptr;
    position(int w0, int w1, int w2, int b0, int b1, int b2) {
        W = {w0, w1, w2};
        B = {b0, b1, b2};
        sort(W.begin(), W.end());
        sort(B.begin(), B.end());
        cal_hash();
    }
    position() {};
    void cal_hash() {
        hash = 0;
        for (int i = 0; i < 3; ++i) {
            hash = hash * 12 + B[i];
        }
        for (int i = 0; i < 3; ++i) {
            hash = hash * 12 + W[i];
        }
    }
    vector<position> gene_mov(int white_black) {
        vector<position> res;
        if (!white_black) {
            for (int i = 0; i < 3; ++i) {
                for (int j = 0; j < 8 && depl[W[i]][j] != -1; ++j) {
                    int pos = depl[W[i]][j];
                    if (find(W.begin(), W.end(), pos) == W.end() &&
                        find(B.begin(), B.end(), pos) == B.end()) {
                        auto Wnew = W; Wnew[i] = pos; sort(Wnew.begin(), Wnew.end());
                        res.emplace_back(Wnew[0], Wnew[1], Wnew[2], B[0], B[1], B[2]);
                    }
                }
            }
        } else {
            for (int i = 0; i < 3; ++i) {
                for (int j = 0; j < 8 && depl[B[i]][j] != -1; ++j) {
                    int pos = depl[B[i]][j];
                    if (find(B.begin(), B.end(), pos) == B.end() &&
                        find(W.begin(), W.end(), pos) == W.end()) {
                        auto Bnew = B; Bnew[i] = pos; sort(Bnew.begin(), Bnew.end());
                        res.emplace_back(W[0], W[1], W[2], Bnew[0], Bnew[1], Bnew[2]);
                    }
                }
            }
        }
        return res;
    }
    void print(int shift = 0) {
        char displ[4][3] = {{'.','.','.'},{'.','.','.'},{'.','.','.'},{'.','.','.'}};
        for (auto w : W) displ[3 - w / 3][w % 3] = 'W';
        for (auto b : B) displ[3 - b / 3][b % 3] = 'B';
        for (int i = 0; i < 4; ++i) {
            for (int j = 0; j < 3; j++) {
                cout << "| " << displ[i][j] << " ";
            }
            cout << "|\n";
            cout << "+---+---+---+\n";
        }
    }
};

tuple<bool, int, vector<position>> find_moves(position &first, position &end) {
    vector<position> moves;
    vector<bool> pos_found(2985984, false);

    vector<vector<position>> dfs;
    dfs.push_back({first});

    int iter = 1;
    int white_black = 0;
    while (true) {
        int n = dfs[iter-1].size();
        if (n == 0) return make_tuple(false, 0, moves);
        dfs.push_back({});
        for (int i = 0; i < n; ++i) {
            auto candidates = dfs[iter-1][i].gene_mov(white_black);
            for (auto &c : candidates) {
                if (pos_found[c.hash]) continue;
                c.up = &dfs[iter-1][i];
                if (c.hash == end.hash) {
                    moves.resize(iter+1);
                    moves[iter] = c;
                    for (int i = iter - 1; i >= 0; i--) {
                        moves[i] = *(moves[i+1].up);
                    }
                    return make_tuple(true, iter, moves);
                }
                pos_found[c.hash] = true;
                dfs[iter].push_back(c);
            }
        }
        iter++;
        white_black = 1 - white_black;
    }
    return make_tuple(false, -1, moves);
}

int main() {
    init_moves();
    position first(9, 10, 11, 0, 1, 2);
    position end(0, 1, 2, 9, 10, 11);

    bool success;
    int n_iter;
    vector<position> moves;
    tie(success, n_iter, moves) = find_moves(first, end);
    cout << "success = " << success << "\n";
    cout << "n_iter = " << n_iter << "\n\n";

    for (auto &m : moves) {
        m.print();
        cout << "\n";
    }
    return 0;
}
```

Explanation

The BFS implementation uses an iterative, level-by-level search strategy to guarantee the shortest path to the goal state, making it a reliable choice for finding the optimal move sequence. The code begins with the init_moves() function, which precomputes valid knight moves for each square on the chessboard. It calculates possible L-shaped knight moves (two squares one way, one perpendicular, or vice versa) and checks board boundaries, storing results in an array with invalid moves marked distinctly. This precomputation enables efficient move generation during the search. The core data structure is the position struct, which represents a state with two arrays: one for the positions of the three white knights and another for the three black knights, using board indices. The struct includes a hash field for state identification, computed as a unique identifier based on knight positions, and a pointer to the parent state for path reconstruction. The gene_mov() method generates next states by moving one knight—either white or black, depending on the turn—to an empty square, ensuring no overlaps and enforcing alternation between white and black turns. The print() method visualizes the board, displaying 'W' for white knights, 'B' for black, and '.' for empty squares.

The BFS algorithm is implemented in the find_moves() function, which uses a simulated queue to explore states systematically, organized by depth levels. It starts with the initial state (white knights on the bottom row, black on the top) and tracks visited states using a hash-based array to avoid duplicates. The algorithm alternates between white and black turns, generates neighbors for each state at the current depth, and enqueues unvisited states into the next depth level. When the goal state (white knights on the top row, black on the bottom) is reached, it reconstructs the path by following parent pointers back to the initial state, returning a success flag, the move count, and the move sequence. The main() function sets up the initial and goal states, runs BFS, and prints the solution sequence with board visualizations. BFS ensures optimality by exploring all possible moves at each level before proceeding, making it thorough but resource-intensive due to the need to store and process many states.

Time Complexity Analysis

BFS explores all states level by level, starting from the initial state at depth 0 and expanding to depth d = 16 d=16. At each depth k k, BFS explores all states reachable in exactly k k moves, with the number of states at depth k k being approximately b^k where b is the branching factor. The total number of states explored is the sum of a geometric series:

$$1 + b + b^2 + b^d = \frac{b^{d+1} - 1}{b - 1}$$

thus, total time complexity is:

$$O(b^d \, . b) = O(b^{d+1})$$

However, in standard literature (e.g., Introduction to Algorithms, Cormen et al., 2009), BFS is often cited as:

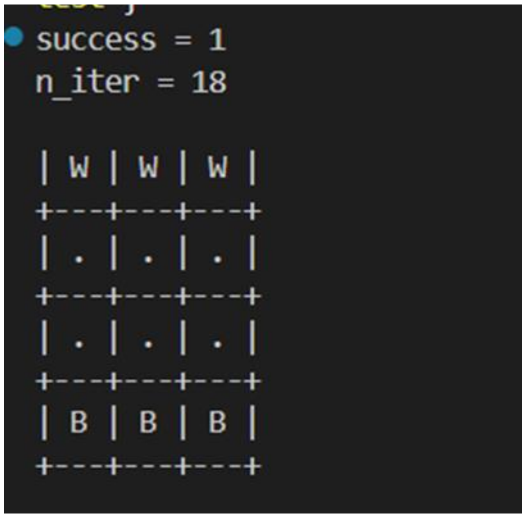
$$O\ (b^d)$$

B is the branching factor and it represents the number of possible moves for the knights in a turn.

D is the depth and it represents the number of moves required to reach the optimal solution, which is 16.

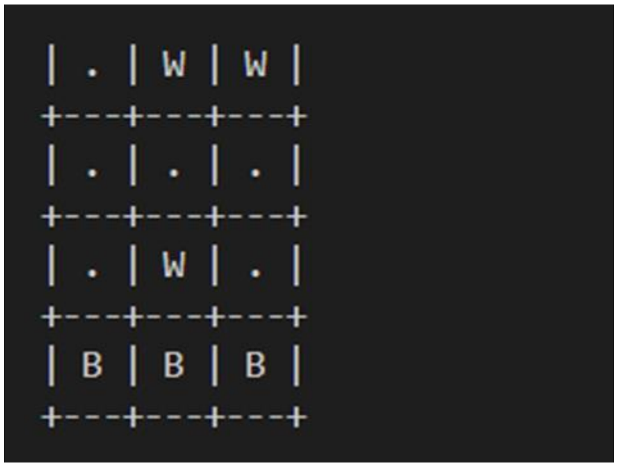
Output

First Iteration



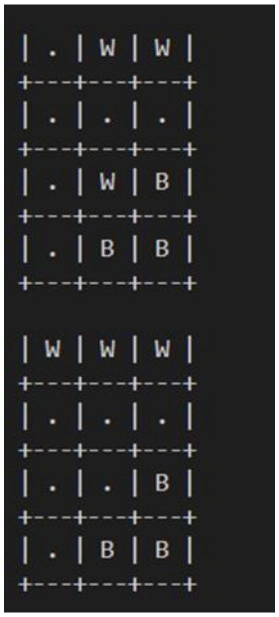
```
success = 1
n_iter = 18

| W | W | W |
+---+---+---+
| . | . | . |
+---+---+---+
| . | . | . |
+---+---+---+
| B | B | B |
+---+---+---+
```

Second Iteration



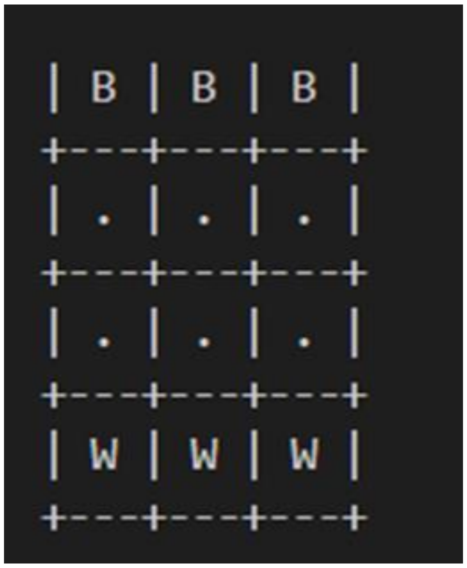
```
| . | W | W |
+---+---+---+
| . | . | . |
+---+---+---+
| . | W | . |
+---+---+---+
| B | B | B |
+---+---+---+
```

Third and Fourth Iteration



```
| . | W | W |
+---+---+---+
| . | . | . |
+---+---+---+
| . | W | B |
+---+---+---+
| . | B | B |
+---+---+---+

| W | W | W |
+---+---+---+
| . | . | . |
+---+---+---+
| . | . | B |
+---+---+---+
| . | B | B |
+---+---+---+
```

Last Iteration



```
| B | B | B |
+---+---+---+
| . | . | . |
+---+---+---+
| . | . | . |
+---+---+---+
| W | W | W |
+---+---+---+
```

A-Star (A*)

The A* implementation, in contrast, employs a heuristic-guided search to focus on promising states, offering a more efficient path to the optimal solution.

Pseudocode

```
Function AStarSearch(board, initial_state, goal_state):
    If initial_state == goal_state:
        Return empty sequence // No moves needed
    Initialize open_list with initial_state, f = h(initial_state)
    Initialize closed_list as empty
    Initialize g_score with g(initial_state) = 0
    Initialize parent map
    While open_list is not empty:
        Select S from open_list with minimum f(S)
        If S == goal_state:
            Return ReconstructPath(parent, S)
        Remove S from open_list
        Add S to closed_list
        Set turn = g_score[S] % 2 // 0 for white, 1 for black
        For each neighbor N in GenerateNeighbors(S, turn):
            If N in closed_list:
                Skip N
            Set tentative_g = g_score[S] + 1
            If N not in open_list or tentative_g < g_score[N]:
                Set g_score[N] = tentative_g
                Set f_score[N] = g_score[N] + h(N)
                Set parent[N] = S
                If N not in open_list:
                    Add N to open_list
    Return failure // No solution found

Function h(state):
    Initialize total_distance = 0
    For each white knight in state:
        Add Manhattan distance to its target position
    For each black knight in state:
        Add Manhattan distance to its target position
    Return total_distance

Function GenerateNeighbors(state, turn):
    // Same as BFS GenerateNeighbors
    Return list of valid next states

Function ReconstructPath(parent, state):
    // Same as BFS ReconstructPath
    Return path
```

C++ code

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>
#include <string>
#include <sstream>
#include <chrono>

using namespace std;

const int ROWS = 4;
const int COLS = 3;
const int NUM_KNIGHTS = 3;

const int dx[] = {-2, -2, -1, -1, 1, 1, 2, 2};
const int dy[] = {-1, 1, -2, 2, -2, 2, -1, 1};

enum KnightColor { EMPTY = 0, WHITE = 1, BLACK = 2 };

struct Knight {
    int row;
    int col;
    KnightColor color;

    Knight(int r, int c, KnightColor clr) : row(r), col(c), color(clr) {}

    bool operator==(const Knight& other) const {
        return row == other.row && col == other.col && color == other.color;
    }
};

struct State {
    vector<vector<KnightColor>> board;
    vector<Knight> knights;
    int g;
    int h;
    int f;
    string moveHistory;
    string lastMove;

    State() : g(0), h(0), f(0) {
        board = vector<vector<KnightColor>>(ROWS, vector<KnightColor>(COLS, EMPTY));
        moveHistory = "";
        lastMove = "";
    }

    bool operator==(const State& other) const {
        for (int i = 0; i < ROWS; ++i) {
            for (int j = 0; j < COLS; ++j) {
                if (board[i][j] != other.board[i][j]) {
                    return false;
                }
            }
        }
        return true;
    }
};

namespace std {
    template <>
    struct hash<State> {
        size_t operator()(const State& state) const {
            size_t hash_val = 0;
            for (int i = 0; i < ROWS; ++i) {
                for (int j = 0; j < COLS; ++j) {
                    hash_val = hash_val * 3 + state.board[i][j];
                }
            }
            return hash_val;
        }
    };
}

bool isValid(int row, int col) {
    return row >= 0 && row < ROWS && col >= 0 && col < COLS;
}

void printBoard(const State& state) {
    cout << "  a b c" << endl;
    for (int i = ROWS - 1; i >= 0; --i) {
        cout << i + 1 << " ";
        for (int j = 0; j < COLS; ++j) {
            if (state.board[i][j] == WHITE) {
                cout << "W ";
            } else if (state.board[i][j] == BLACK) {
                cout << "B ";
            } else {
                cout << ". ";
            }
        }
        cout << endl;
    }
    cout << endl;
}

bool isGoalState(const State& state) {
    for (int j = 0; j < COLS; ++j) {
        if (state.board[ROWS - 1][j] != WHITE) {
            return false;
        }
    }
    for (int j = 0; j < COLS; ++j) {
        if (state.board[0][j] != BLACK) {
            return false;
        }
    }
    return true;
}

int minKnightMoves(int srcRow, int srcCol, int destRow, int destCol) {
    if (srcRow == destRow && srcCol == destCol) {
        return 0;
    }
    vector<vector<bool>> visited(ROWS, vector<bool>(COLS, false));
    queue<pair<pair<int, int>, int>> q;
    q.push({{srcRow, srcCol}, 0});
    visited[srcRow][srcCol] = true;
    while (!q.empty()) {
        auto curr = q.front();
        q.pop();
        int row = curr.first.first;
        int col = curr.first.second;
        int dist = curr.second;
        if (row == destRow && col == destCol) {
            return dist;
        }
        for (int k = 0; k < 8; ++k) {
            int newRow = row + dx[k];
            int newCol = col + dy[k];
            if (isValid(newRow, newCol) && !visited[newRow][newCol]) {
                visited[newRow][newCol] = true;
                q.push({{newRow, newCol}, dist + 1});
            }
        }
    }
    return -1;
}

int calculateHeuristic(const State& state) {
    int total = 0;
    for (const Knight& knight : state.knights) {
        if (knight.color == WHITE) {
            int minMoves = INT_MAX;
            for (int j = 0; j < COLS; ++j) {
                if (state.board[ROWS - 1][j] == EMPTY || (state.board[ROWS - 1][j] == WHITE && knight.row
== ROWS - 1 && knight.col == j)) {
                    int moves = minKnightMoves(knight.row, knight.col, ROWS - 1, j);
                    minMoves = min(minMoves, moves);
                }
            }
            total += minMoves;
        }
        else if (knight.color == BLACK) {
            int minMoves = INT_MAX;
            for (int j = 0; j < COLS; ++j) {
                if (state.board[0][j] == EMPTY || (state.board[0][j] == BLACK && knight.row == 0 &&
knight.col == j)) {
                    int moves = minKnightMoves(knight.row, knight.col, 0, j);
                    minMoves = min(minMoves, moves);
                }
            }
            total += minMoves;
        }
    }
    return total;
}

vector<State> getNextStates(const State& current) {
    vector<State> nextStates;
    for (int k = 0; k < current.knights.size(); ++k) {
        const Knight& knight = current.knights[k];
        for (int i = 0; i < 8; ++i) {
            int newRow = knight.row + dx[i];
            int newCol = knight.col + dy[i];
            if (isValid(newRow, newCol) && current.board[newRow][newCol] == EMPTY) {
                State nextState = current;
                nextState.board[knight.row][knight.col] = EMPTY;
                nextState.board[newRow][newCol] = knight.color;
                nextState.knights[k].row = newRow;
                nextState.knights[k].col = newCol;
                nextState.g = current.g + 1;
                nextState.h = calculateHeuristic(nextState);
                nextState.f = nextState.g + nextState.h;
                char colFrom = 'a' + knight.col;
                char colTo = 'a' + newCol;
                nextState.lastMove = (knight.color == WHITE ? "W" : "B") +
                    string(1, colFrom) + to_string(knight.row + 1) + "-" +
                    string(1, colTo) + to_string(newRow + 1);
                nextState.moveHistory = current.moveHistory + (current.moveHistory.empty() ? "" : ", ") +
nextState.lastMove;
                nextStates.push_back(nextState);
            }
        }
    }
    return nextStates;
}

State solveKnightSwap() {
    auto startTime = chrono::high_resolution_clock::now();
    State initialState;
    for (int j = 0; j < COLS; ++j) {
        initialState.board[0][j] = WHITE;
        initialState.knights.push_back(Knight(0, j, WHITE));
    }
    for (int j = 0; j < COLS; ++j) {
        initialState.board[ROWS - 1][j] = BLACK;
        initialState.knights.push_back(Knight(ROWS - 1, j, BLACK));
    }
    initialState.h = calculateHeuristic(initialState);
    initialState.f = initialState.h;
    auto compare = [](const State& a, const State& b) {
        if (a.f != b.f) {
            return a.f > b.f;
        }
        return a.g < b.g;
    };
    priority_queue<State, vector<State>, decltype(compare)> openSet(compare);
    unordered_set<State> closedSet;
    unordered_map<size_t, State> stateMap;
    openSet.push(initialState);
    int nodesExplored = 0;
    while (!openSet.empty()) {
        State current = openSet.top();
        openSet.pop();
        nodesExplored++;
        if (isGoalState(current)) {
            auto endTime = chrono::high_resolution_clock::now();
            auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime).count();
            cout << "Solution found in " << current.g << " moves" << endl;
            cout << "Nodes explored: " << nodesExplored << endl;
            cout << "Time taken: " << duration << " ms" << endl;
            return current;
        }
        if (closedSet.find(current) != closedSet.end()) {
            continue;
        }
        closedSet.insert(current);
        vector<State> nextStates = getNextStates(current);
        for (const State& nextState : nextStates) {
            if (closedSet.find(nextState) != closedSet.end()) {
                continue;
            }
            openSet.push(nextState);
        }
    }
    cout << "No solution found" << endl;
    return initialState;
}

void visualizeSolution(const State& solution) {
    if (solution.moveHistory.empty()) {
        cout << "No moves to visualize." << endl;
        return;
    }
    State currentState;
    for (int j = 0; j < COLS; ++j) {
        currentState.board[0][j] = WHITE;
        currentState.knights.push_back(Knight(0, j, WHITE));
    }
    for (int j = 0; j < COLS; ++j) {
        currentState.board[ROWS - 1][j] = BLACK;
        currentState.knights.push_back(Knight(ROWS - 1, j, BLACK));
    }
    cout << "Initial state:" << endl;
    printBoard(currentState);
    istringstream iss(solution.moveHistory);
    string move;
    int moveNumber = 1;
    while (getline(iss, move, ',')) {
        move.erase(0, move.find_first_not_of(" \t"));
        move.erase(move.find_last_not_of(" \t") + 1);
        char color = move[0];
        char fromCol = move[1];
        int fromRow = move[2] - '0';
        char toCol = move[4];
        int toRow = move[5] - '0';
        int srcRow = fromRow - 1;
        int srcCol = fromCol - 'a';
        int destRow = toRow - 1;
        int destCol = toCol - 'a';
        for (int i = 0; i < currentState.knights.size(); ++i) {
            Knight& knight = currentState.knights[i];
            if (knight.row == srcRow && knight.col == srcCol) {
                currentState.board[srcRow][srcCol] = EMPTY;
                currentState.board[destRow][destCol] = knight.color;
                knight.row = destRow;
                knight.col = destCol;
                break;
            }
        }
        cout << "Move " << moveNumber << ": " << move << endl;
        printBoard(currentState);
        moveNumber++;
    }
}

int main() {
    cout << "Knight Swap Puzzle Solver" << endl;
    cout << "-------------------------" << endl;
    cout << "Initial state: White knights at bottom, Black knights at top" << endl;
    cout << "Goal state: White knights at top, Black knights at bottom" << endl;
    cout << "Finding minimal solution using A* algorithm..." << endl << endl;
    State solution = solveKnightSwap();
    cout << endl << "Solution path: " << solution.moveHistory << endl << endl;
    cout << "Visualizing solution step by step:" << endl;
    visualizeSolution(solution);
    return 0;
}
```

Explanation

The code defines a Knight struct to represent a knight's row, column, and color (white or black) and a State struct to encapsulate the board state, including a grid of knight positions, a vector of six knights, and cost metrics: moves taken, heuristic estimate, total estimated cost, and fields to track the solution path. The minKnightMoves() function calculates the shortest knight move path between two squares, used in the heuristic calculation. The calculateHeuristic() function estimates the remaining moves needed by summing the minimum knight moves for each knight to its target row (top for white, bottom for black), selecting the closest viable target square. This heuristic ensures A* finds the optimal path by never overestimating the true cost.

The A* algorithm is implemented in solveKnightSwap(), which initializes the board with white knights on the top row and black knights on the bottom (noting a potential grid dimension inconsistency in the code). It uses a priority queue to explore states with the lowest total estimated cost, prioritizing those closer to the goal. A hash-based set prevents re-exploration of states, and a custom hash function identifies states based on the board configuration. The getNextStates() function generates next states by moving any knight to a valid, empty square, updating costs and move history without enforcing turn alternation, relying on the heuristic to guide the search. When the goal state is reached (verified by checking knight positions), the algorithm returns the solution state, logging the move count and path details. The visualizeSolution() function replays the move sequence, printing the board after each step for clarity.

<u>Time Complexity Analysis</u>

For each generation of successors, the complexity is $O(b)$, and each insertion in the priority queue is $O(logM)$

$$O(M \cdot b\ logM)$$

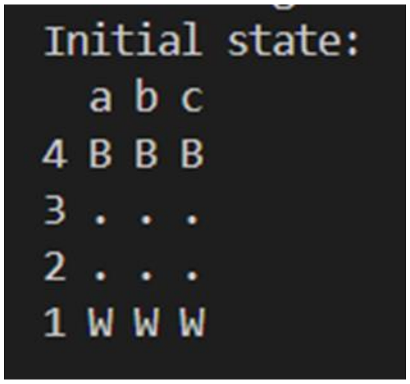
B is the branching factor and it represents the number of possible moves for the knights in a turn.

D is the depth and it represents the number of moves required to reach the optimal solution, which is 16.

M is the number of states explored

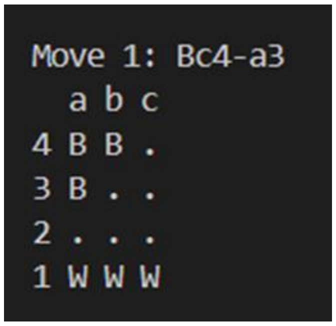
<u>Output</u>

First Iteration



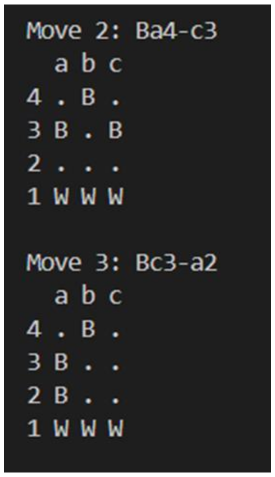
```
Initial state:
  a b c
4 B B B
3 . . .
2 . . .
1 W W W
```

Second Iteration



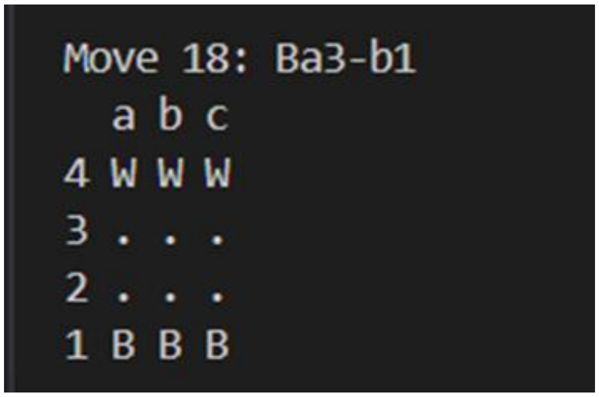
```
Move 1: Bc4-a3
  a b c
4 B B .
3 B . .
2 . . .
1 W W W
```

Third and Fourth Iteration



```
Move 2: Ba4-c3
  a b c
4 . B .
3 B . B
2 . . .
1 W W W

Move 3: Bc3-a2
  a b c
4 . B .
3 B . .
2 B . .
1 W W W
```

Last Iteration



```
Move 18: Ba3-b1
  a b c
4 W W W
3 . . .
2 . . .
1 B B B
```

<u>Iterative Deepening Depth-First Search (IDDFS)</u>

<u>Pseudocode</u>

```
BEGIN
    1. For depth from 0 to infinity:
       a. Call Depth-Limited Search (DLS) with current depth
       b. If DLS returns a solution, return it
    2. If no solution is found, return failure
END

Depth-Limited Search (DLS):
BEGIN
     ascended
    1. If depth = 0 and current state is goal, return success
    2. If depth = 0 and current state is not goal, return failure
    3. For each successor S of current state:
       a. If DLS(S, depth - 1) succeeds, return success
    4. Return failure
END
```

<u>Time Complexity</u>

$$\sum_{k=0}^{m} b^k = \frac{b^{m+1}-1}{b-1}$$

$$\sum_{m=0}^{d}\sum_{k=0}^{m} b^k = \sum_{m=0}^{d} \frac{b^{m+1}-1}{b-1}$$

$$\frac{b^{d+1}-1}{b-1} \approx O(b^d)$$

With $O\ (b^d)$ total states explored, each requiring O (b)

$$O(b^d \,.\, b) \ = \ O(b^{d+1})$$

However, in standard literature, it is often cited as:

$$O\ (\ b^d)$$

B: is the branching factor and it represents the number of possible moves for the knights in a turn.

D: is the depth and it represents the number of moves required to reach the optimal solution, which is 16

Bidirectional Search

Pseudocode

```
BEGIN
    1. Initialize two queues: Q1 with initial state, Q2 with goal state
    2. Initialize two visited sets: V1 with initial state, V2 with goal state
    3. While Q1 and Q2 are not empty:
       a. Expand Q1:
          i. Dequeue S1 from Q1
          ii. For each neighbor N1 of S1:
             - If N1 is in V2, return path from S1 to N1 via Q1 and Q2
             - If N1 not in V1, enqueue N1 to Q1, add to V1
       b. Expand Q2:
          i. Dequeue S2 from Q2
          ii. For each neighbor N2 of S2:
             - If N2 is in V1, return path from S2 to N2 via Q2 and Q1
             - If N2 not in V2, enqueue N2 to Q2, add to V2
    4. If no path found, return failure
END
```

Time Complexity

$$\sum_{i=0}^{k} b^{i} \approx O(b^{k})$$

$$\sum_{i=0}^{d-k} b^{i} \approx O(b^{d-k})$$

Total states when k=d/2:

$$O\left(b^{\frac{d}{2}}\right) + O\left(b^{\frac{d}{2}}\right) \approx O\left(b^{\frac{d}{2}}\right)$$

With $\boldsymbol{O}\ (\boldsymbol{b}^{\mathbf{d}})$ total states explored, each requiring O(b):

$$O(b^{\frac{d}{2}}.b) = O(b^{\frac{d}{2}+1})$$

However, in standard literature, it is often cited as:

$$\mathrm{O}(\mathrm{b}^{\mathrm{d}/2})$$

B: is the branching factor and it represents the number of possible moves for the knights in a turn.

D: is the depth and it represents the number of moves required to reach the optimal solution, which is 16

Comparison

Algorithm	Time Complexity	Space Complexity	Advantages	Disadvantages
Bidirectional Search	$O(b^{d/2})$	$O(b^{d/2})$	Highly efficient, optimal, leverages symmetry	Complex implementation, higher memory than IDDFS, meeting point uncertainty
A*	O(M·b.logM)	O(M)	Efficient with good heuristic, optimal, flexible	Complex implementation, memory intensive with poor heuristic, heuristic dependency
IDDFS	$O(b^d)$	O(d)	Low memory usage, optimal, complete	Redundant exploration, slower than optimal, implementation overhead
BFS	$O(b^d)$	O(bd)	Guaranteed optimality, complete, simple implementation	High memory usage, slow for deep searches, redundant exploration

Conclusion

The Six Knights puzzle, requiring the exchange of three white and three black knights on a 3×4 chessboard in a minimum of 16 moves, is a complex state-space search problem that demands systematic exploration to achieve an optimal solution. The provided C++ implementations of BFS and A* Search successfully solve this puzzle, each with distinct strengths. BFS, an uninformed search method, exhaustively explores all possible configurations level by level, guaranteeing the shortest path of 16 moves through its iterative, queue-based approach and strict alternation of white and black moves. However, its time complexity of O(b^d) (where b≈12−18 and d = 16) and high memory demands make it computationally expensive. In contrast, A* Search, an informed algorithm, leverages a heuristic (the sum of minimum knight moves to target positions) to prioritize promising states, significantly reducing the number of explored nodes while still ensuring optimality. Its time complexity makes it more efficient, despite the overhead of heuristic computation.

Pure iterative improvement techniques like hill-climbing or simulated annealing fail here, as they get trapped in local optima due to the puzzle's interdependent move constraints. BFS and A*, while not greedy or locally guided, fit the iterative requirement through their systematic state exploration. A* emerges as the superior choice for this task, balancing efficiency and optimality, making it practical for large state spaces. Both algorithms confirm the theoretical minimum of 16 moves, validated by the cyclic dependencies that preclude shorter solutions, as proven in the problem analysis. This project highlights the power of state-space search over local heuristics in multi-agent puzzles, with A* offering a scalable solution for similar combinatorial challenges.

References

[1] How to solve black & white knights problem in 3×3 grid

[2] Knights Exchange Puzzle—Teaching the Efficiency of Modeling

[3] Algorithmic Puzzles

[4] Introduction to Algorithms

[5] Artificial Intelligence: A Modern Approach:

Task 5: Hitting a Moving Target

Definition

A computer game has a shooter and a moving target. The shooter can hit any of n > 1 hiding spot located along a straight line in which the target can hide. The shooter can never see the target; all he knows is that the target moves to an adjacent hiding spot between every two consecutive shots. Design a Divide and conquer algorithm that guarantees hitting the target.

Assumptions

1. There are n>1 hiding spots.
2. The shooter can never see the target.
3. The target moves to an adjacent hiding spot between every two consecutive shots.
4. The shooter can choose any hiding spot to shoot.
5. The number of hiding spots (n) is finite and known.
6. The target cannot leave the line.

Explanation and Pseudo-code

By using the Divide & Conquer technique, I will divide an array of size n at its midpoint into left and right halves and conquer the middle point by assigning the shooter to hit it and if the shot misses recursively call the function to the left and right subarrays until the target is hit [1], [2].

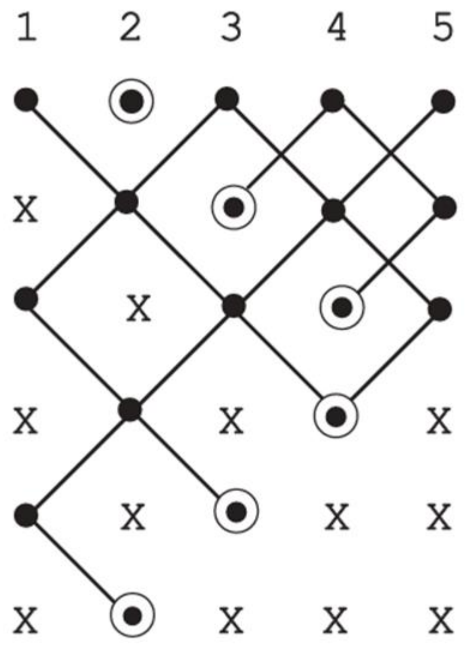


Figure 12: Hitting a Moving Target algorithm for n = 5, Row i, i = 1,… ,2n − 4, depicts the possible (shown by the small black circles) and impossible (shown by the X's) locations for the moving target before the ith shot; the shot is indicated by the ring around the location

```
function shoot_target(left, right):
    // Base case: only one hiding spot remains
    if left == right:
        hit(left)
        return
    // Divide step
    mid ← ceil((left + right) / 2)
// Step 1: Shoot at the midpoint
    hit(mid)
// Step 2: Shoot again before the target moves
    // Prefer midpoint of the half not containing 'mid' to as we already have shot at the midpoint
    // Choose a second shot strategically (could be at the midpoint of one half)
    // We'll assume a symmetric divide and shoot second midpoint
    // Compute midpoints for left and right halves
    if left < mid:
        left_mid ← ceil((left + mid - 1) / 2)
        hit(left_mid)
    elif mid < right:
        right_mid ← ceil((mid + 1 + right) / 2)
        hit(right_mid)
// Step 3
    // the target moves ±1 after two shots
    // So we expand the next search range by 1 in both directions
    new_left ← max(1, left - 1) //So that we don't go lower than 1
    new_right ← min(n, right + 1) //So that we don't go higher than n
// Step 4
    // Recurse on the expanded search range
    shoot_target(new_left, new_right)
```

C++ Code

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

int n;                  // Total number of hiding spots
int target_position;    // Actual position of the moving target
int shots_taken = 0;    // Counter for number of shots

// Function to move the target to an adjacent position
void move_target() {
    if (shots_taken%2!=0) return; // Only move after  2 consective shots

    int direction = rand() % 2; // 0 for left, 1 for right
    if (direction == 0 && target_position > 1) {
        target_position--;
    } else if (direction == 1 && target_position < n) {
        target_position++;
    }
    // If can't move in chosen direction, stays in place
}

bool hit(int position) {
    shots_taken++;
    cout << "Shot " << shots_taken << ": Shooting at position " << position;
    cout << " (Target is at " << target_position << ")";

    if (position == target_position) {
        cout << " – HIT!" <<endl;
        cout << "\nTarget destroyed in " << shots_taken << " shots!" <<endl;
        return true;
    } else {
        cout << " – Missed." << endl;
        move_target(); // Target moves only after 2 consective shots
        return false;
    }
}

// Recursive Divide & Conquer shooting strategy
void shoot_target(int left, int right,bool& s) {
    if (left > right || s) return; // stop recursion if already hit
    // Base case - single position
    if (left == right) {
        s=hit(left);
        return;
    }
    // Midpoint shoot
    int mid = left + (right - left) / 2;
    s=hit(mid);
    if (s) return; // Stop further shooting if target hit
    // Recursively search left and right halves
    shoot_target(left, mid - 1,s);
    shoot_target(mid + 1, right,s);
}

int main() {
    srand(time(0));
    cout << "Enter the number of hiding spots (n): ";
    cin >> n;
    bool s = false;
    if (n < 1) {
        cerr << "Invalid value for n." << endl;
        return 1;
    }
    target_position = 1 + rand() % n;
    cout << "Target starts at position " << target_position << endl;
    cout << "Target will move after the 2nd shot\n" << endl;
    while (!s) {
        shoot_target(1, n, s);
    }
    return 0;
}
```

Time Complexity Analysis

The shoot_target function is a divide-and-conquer algorithm that:

1. Shoots at the midpoint (mid).
2. Recursively searches the left half (left to mid-1).
3. Recursively searches the right half (mid+1 to right).

This resembles a binary search, but unlike binary search (which discards half the space), this function visits every position exactly once (like a preorder traversal of a binary tree).

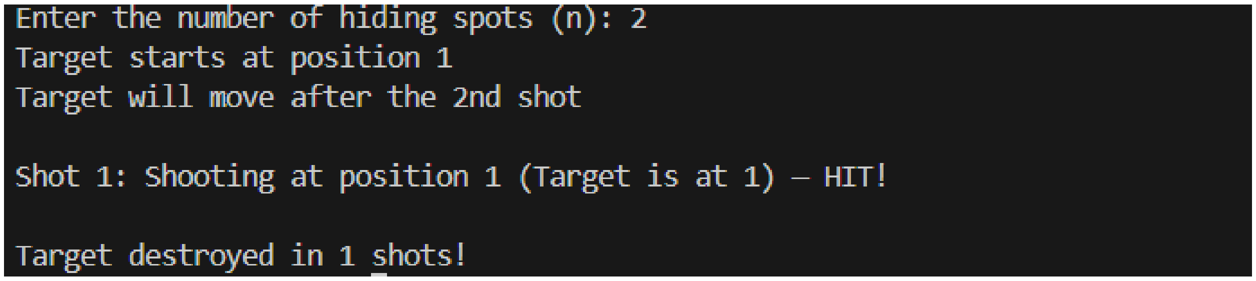
Which gives the following recurrence relation :

$$T(n) = 2T\left(\frac{n}{2}\right) + 2$$

We can solve this using the Master Theorem as a=2 and b=2 and d=0 therefore 2 =2^1 so it is the second case so $T(n) \in \theta(nlogn) = O(n)$

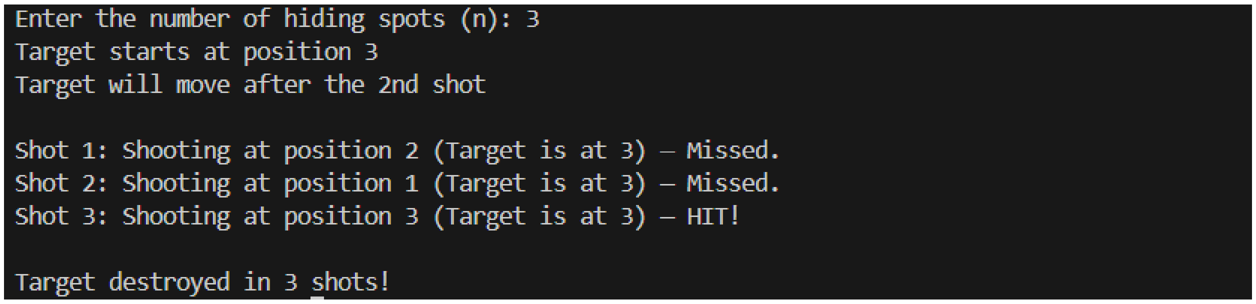
Output

n=2



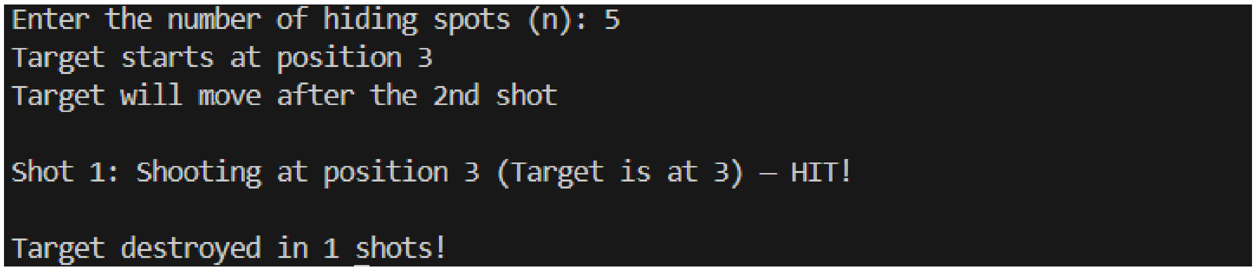
```
Enter the number of hiding spots (n): 2
Target starts at position 1
Target will move after the 2nd shot

Shot 1: Shooting at position 1 (Target is at 1) – HIT!

Target destroyed in 1 shots!
```

n=3



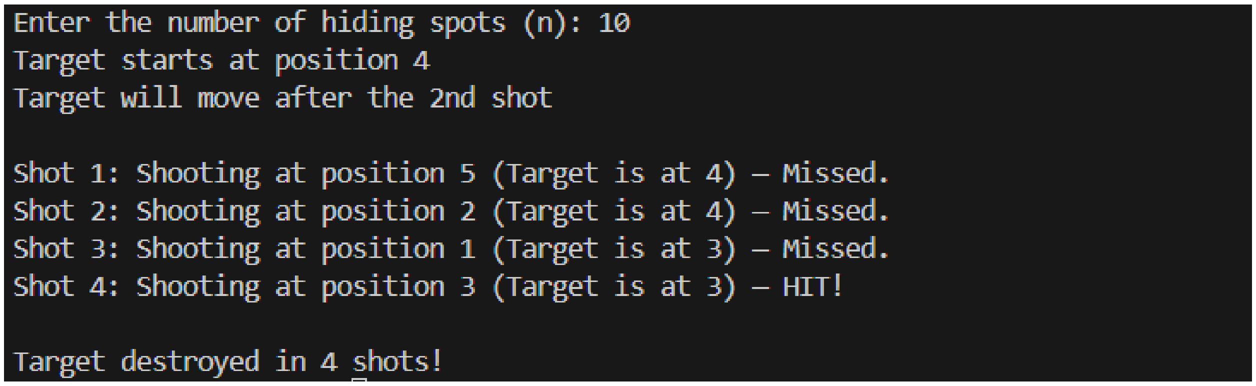
```
Enter the number of hiding spots (n): 3
Target starts at position 3
Target will move after the 2nd shot

Shot 1: Shooting at position 2 (Target is at 3) – Missed.
Shot 2: Shooting at position 1 (Target is at 3) – Missed.
Shot 3: Shooting at position 3 (Target is at 3) – HIT!

Target destroyed in 3 shots!
```

n=5 (Best case, target is at the middle)



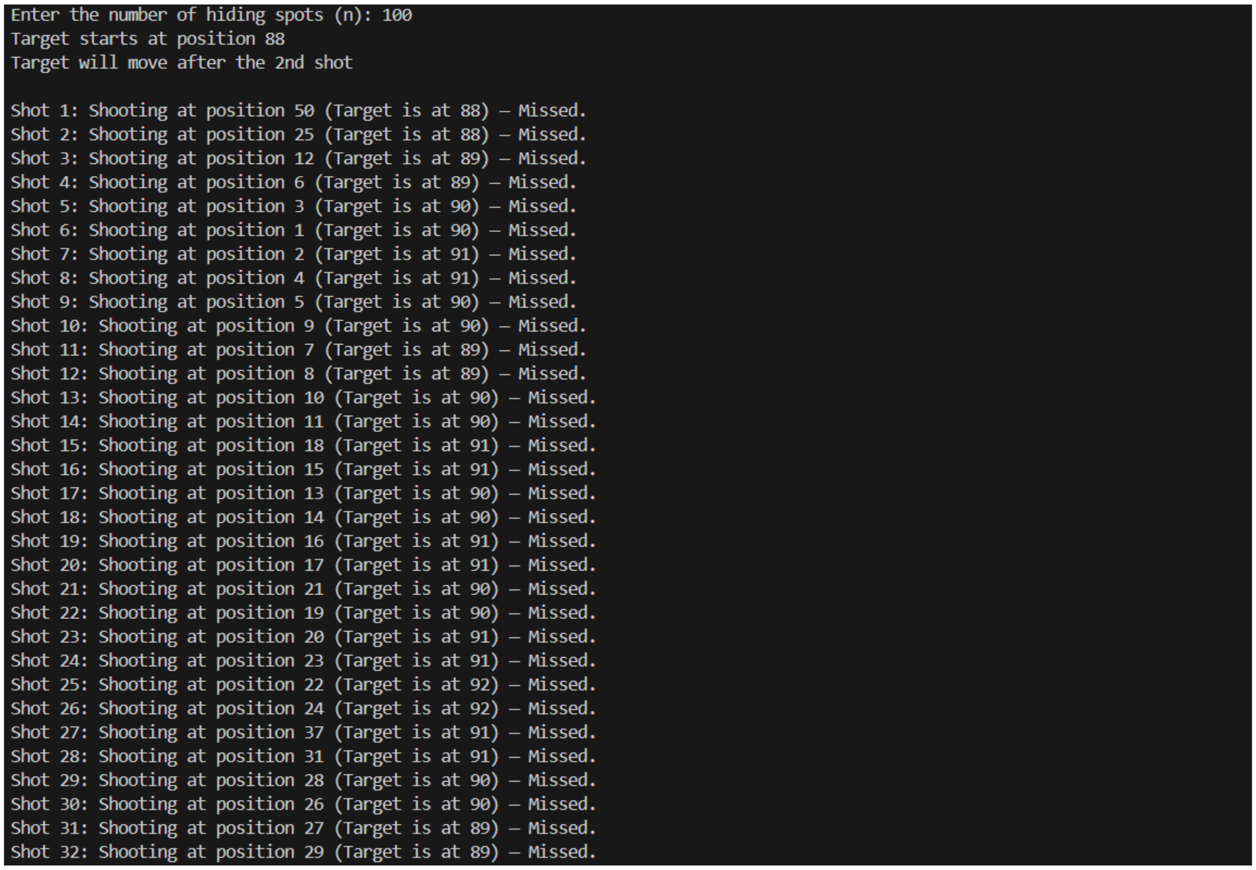
```
Enter the number of hiding spots (n): 5
Target starts at position 3
Target will move after the 2nd shot

Shot 1: Shooting at position 3 (Target is at 3) – HIT!

Target destroyed in 1 shots!
```

n=10

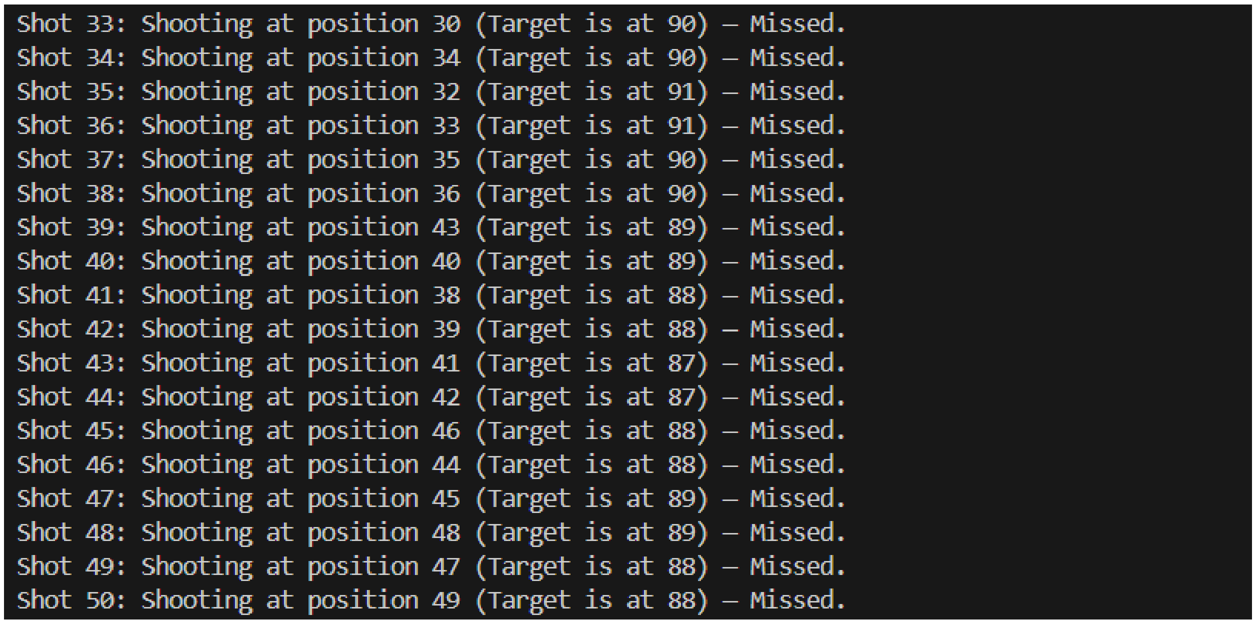


```
Enter the number of hiding spots (n): 10
Target starts at position 4
Target will move after the 2nd shot

Shot 1: Shooting at position 5 (Target is at 4) – Missed.
Shot 2: Shooting at position 2 (Target is at 4) – Missed.
Shot 3: Shooting at position 1 (Target is at 3) – Missed.
Shot 4: Shooting at position 3 (Target is at 3) – HIT!

Target destroyed in 4 shots!
```

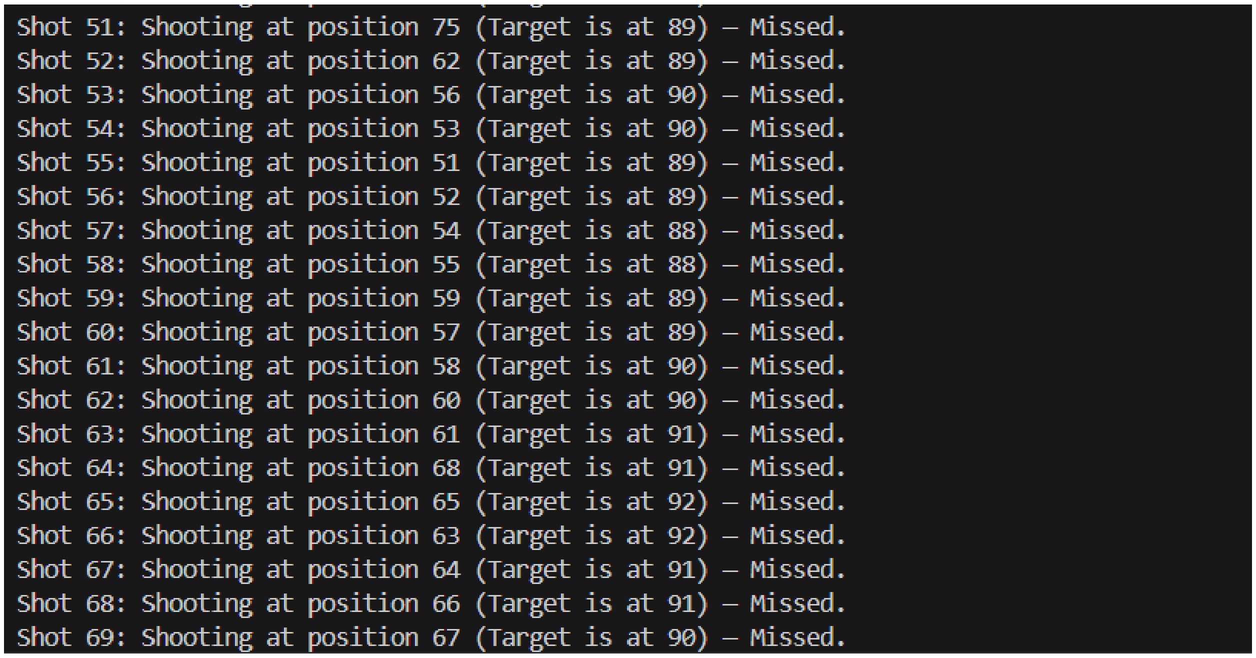
n=100



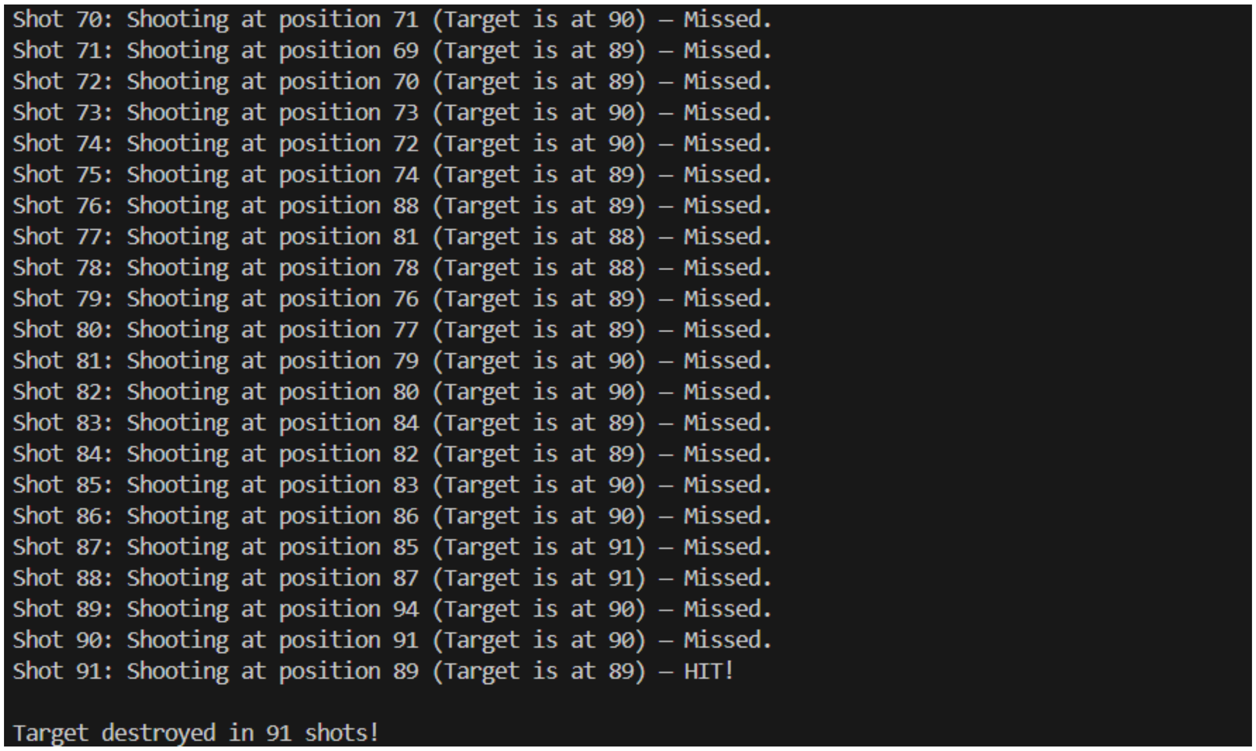
```
Enter the number of hiding spots (n): 100
Target starts at position 88
Target will move after the 2nd shot

Shot 1: Shooting at position 50 (Target is at 88) — Missed.
Shot 2: Shooting at position 25 (Target is at 88) — Missed.
Shot 3: Shooting at position 12 (Target is at 89) — Missed.
Shot 4: Shooting at position 6 (Target is at 89) — Missed.
Shot 5: Shooting at position 3 (Target is at 90) — Missed.
Shot 6: Shooting at position 1 (Target is at 90) — Missed.
Shot 7: Shooting at position 2 (Target is at 91) — Missed.
Shot 8: Shooting at position 4 (Target is at 91) — Missed.
Shot 9: Shooting at position 5 (Target is at 90) — Missed.
Shot 10: Shooting at position 9 (Target is at 90) — Missed.
Shot 11: Shooting at position 7 (Target is at 89) — Missed.
Shot 12: Shooting at position 8 (Target is at 89) — Missed.
Shot 13: Shooting at position 10 (Target is at 90) — Missed.
Shot 14: Shooting at position 11 (Target is at 90) — Missed.
Shot 15: Shooting at position 18 (Target is at 91) — Missed.
Shot 16: Shooting at position 15 (Target is at 91) — Missed.
Shot 17: Shooting at position 13 (Target is at 90) — Missed.
Shot 18: Shooting at position 14 (Target is at 90) — Missed.
Shot 19: Shooting at position 16 (Target is at 91) — Missed.
Shot 20: Shooting at position 17 (Target is at 91) — Missed.
Shot 21: Shooting at position 21 (Target is at 90) — Missed.
Shot 22: Shooting at position 19 (Target is at 90) — Missed.
Shot 23: Shooting at position 20 (Target is at 91) — Missed.
Shot 24: Shooting at position 23 (Target is at 91) — Missed.
Shot 25: Shooting at position 22 (Target is at 92) — Missed.
Shot 26: Shooting at position 24 (Target is at 92) — Missed.
Shot 27: Shooting at position 37 (Target is at 91) — Missed.
Shot 28: Shooting at position 31 (Target is at 91) — Missed.
Shot 29: Shooting at position 28 (Target is at 90) — Missed.
Shot 30: Shooting at position 26 (Target is at 90) — Missed.
Shot 31: Shooting at position 27 (Target is at 89) — Missed.
Shot 32: Shooting at position 29 (Target is at 89) — Missed.
```



```
Shot 33: Shooting at position 30 (Target is at 90) — Missed.
Shot 34: Shooting at position 34 (Target is at 90) — Missed.
Shot 35: Shooting at position 32 (Target is at 91) — Missed.
Shot 36: Shooting at position 33 (Target is at 91) — Missed.
Shot 37: Shooting at position 35 (Target is at 90) — Missed.
Shot 38: Shooting at position 36 (Target is at 90) — Missed.
Shot 39: Shooting at position 43 (Target is at 89) — Missed.
Shot 40: Shooting at position 40 (Target is at 89) — Missed.
Shot 41: Shooting at position 38 (Target is at 88) — Missed.
Shot 42: Shooting at position 39 (Target is at 88) — Missed.
Shot 43: Shooting at position 41 (Target is at 87) — Missed.
Shot 44: Shooting at position 42 (Target is at 87) — Missed.
Shot 45: Shooting at position 46 (Target is at 88) — Missed.
Shot 46: Shooting at position 44 (Target is at 88) — Missed.
Shot 47: Shooting at position 45 (Target is at 89) — Missed.
Shot 48: Shooting at position 48 (Target is at 89) — Missed.
Shot 49: Shooting at position 47 (Target is at 88) — Missed.
Shot 50: Shooting at position 49 (Target is at 88) — Missed.
```



```
Shot 51: Shooting at position 75 (Target is at 89) — Missed.
Shot 52: Shooting at position 62 (Target is at 89) — Missed.
Shot 53: Shooting at position 56 (Target is at 90) — Missed.
Shot 54: Shooting at position 53 (Target is at 90) — Missed.
Shot 55: Shooting at position 51 (Target is at 89) — Missed.
Shot 56: Shooting at position 52 (Target is at 89) — Missed.
Shot 57: Shooting at position 54 (Target is at 88) — Missed.
Shot 58: Shooting at position 55 (Target is at 88) — Missed.
Shot 59: Shooting at position 59 (Target is at 89) — Missed.
Shot 60: Shooting at position 57 (Target is at 89) — Missed.
Shot 61: Shooting at position 58 (Target is at 90) — Missed.
Shot 62: Shooting at position 60 (Target is at 90) — Missed.
Shot 63: Shooting at position 61 (Target is at 91) — Missed.
Shot 64: Shooting at position 68 (Target is at 91) — Missed.
Shot 65: Shooting at position 65 (Target is at 92) — Missed.
Shot 66: Shooting at position 63 (Target is at 92) — Missed.
Shot 67: Shooting at position 64 (Target is at 91) — Missed.
Shot 68: Shooting at position 66 (Target is at 91) — Missed.
Shot 69: Shooting at position 67 (Target is at 90) — Missed.
```



```
Shot 70: Shooting at position 71 (Target is at 90) — Missed.
Shot 71: Shooting at position 69 (Target is at 89) — Missed.
Shot 72: Shooting at position 70 (Target is at 89) — Missed.
Shot 73: Shooting at position 73 (Target is at 90) — Missed.
Shot 74: Shooting at position 72 (Target is at 90) — Missed.
Shot 75: Shooting at position 74 (Target is at 89) — Missed.
Shot 76: Shooting at position 88 (Target is at 89) — Missed.
Shot 77: Shooting at position 81 (Target is at 88) — Missed.
Shot 78: Shooting at position 78 (Target is at 88) — Missed.
Shot 79: Shooting at position 76 (Target is at 89) — Missed.
Shot 80: Shooting at position 77 (Target is at 89) — Missed.
Shot 81: Shooting at position 79 (Target is at 90) — Missed.
Shot 82: Shooting at position 80 (Target is at 90) — Missed.
Shot 83: Shooting at position 84 (Target is at 89) — Missed.
Shot 84: Shooting at position 82 (Target is at 89) — Missed.
Shot 85: Shooting at position 83 (Target is at 90) — Missed.
Shot 86: Shooting at position 86 (Target is at 90) — Missed.
Shot 87: Shooting at position 85 (Target is at 91) — Missed.
Shot 88: Shooting at position 87 (Target is at 91) — Missed.
Shot 89: Shooting at position 94 (Target is at 90) — Missed.
Shot 90: Shooting at position 91 (Target is at 90) — Missed.
Shot 91: Shooting at position 89 (Target is at 89) — HIT!

Target destroyed in 91 shots!
```

Comparison

Brute Force Approach

Algorithm

- The shooter starts at position 1 and shoots sequentially (1 → 2 → 3 → ... → n)
- If the target is not hit, it wraps around (1 → 2 → ... again)

Pseudocode

```
def brute_force_shoot(n, target_pos):
    shots = 0
    while True:
        for pos in range(1, n + 1):
            shots += 1
            if pos == target_pos:
                return shots  # Target hit!
            if shots % 2 == 0:
                target_pos = move_target(target_pos, n)  # Move after every 2 shots
```

Time Complexity

The target keeps moving away, forcing the shooter to traverse the entire list multiple times. $O(n^2)$ because in the worst case, the shooter may need n full passes (each taking n shots) before hitting the target.

Advantages

- Simple to implement.
- Works for small n.

Disadvantages

- Very slow for large n (quadratic time).

Greedy Approach

Algorithm

- The shooter predicts where the target will be based on its movement pattern.
- Since the target moves every 2 shots, the shooter can:
    1. Shoot at the current estimated position.
    2. If missed, adjust the next shot based on possible movement.

Optimal Strategy

- The target moves at most 1 position every 2 shots.
- The shooter should minimize the maximum distance the target can escape.
- Best approach: Alternate between two positions (since the target can't escape far).

Pseudocode

```
def greedy_shoot(n, target_pos):
    shots = 0
    while True:
        # Shoot at the best predicted position
        shoot_pos = predict_target_position(target_pos, shots)
        shots += 1
        if shoot_pos == target_pos:
            return shots
        if shots % 2 == 0:
            target_pos = move_target(target_pos, n)
```

Time Complexity

The target moves away optimally, but the greedy strategy ensures it is caught in O(n) time. Similar to Divide and Conquer approach, but it has better average performance.

Advantages

- Faster than brute force (linear time).
- More efficient than Divide & Conquer in practice (fewer shots).

Disadvantages

- Requires movement prediction, which may be complex.

Overall Comparison

Metric	Brute Force	Greedy	Divide & Conquer
Time Complexity	$O(n^2)$	$O(n)$	$O(n)$
Best For	Small n	Moving targets	Static targets
Worst Case	Very slow	Fast	Moderate
Implementation	Simple	Complex	Moderate

Table 1 Comparison between the 3 approaches

Conclusion

While all three approaches solve the problem, the Greedy method provides the best combination of theoretical guarantees and practical performance for this moving target scenario. The Divide & Conquer approach offers a strong alternative when target movement is less predictable, while the Brute Force method serves primarily as a baseline for comparison.

References

[1] A. Levitin and M. Levitin, "Algorithmic Puzzles."

[2] M. Beckmann et al., "Lecture Notes in Economics and Mathematical Systems."

Task 6: Crossing Dots

Definition

Given an n × n point lattice (intersection points of n consecutive horizontal and n consecutive vertical lines on common graph paper), where n > 2, cross out all the points by 2n − 2 straight lines without lifting your pen from the paper. You may cross the same point more than once, but you cannot redraw any portion of the same line. Design a dynamic programming algorithm to solve this problem.

Assumptions

- For simplification, the board is limited to 15x15 which can accommodate an answer from n = 3 to a higher limited number
- The line is drawn in code as asterisks and can be horizontal, vertical or diagonal

Explanation and Pseudo-Code

The problem involves crossing all points of an n × n point lattice (grid) using 2n − 2 straight lines without lifting the pen. The lines may cross the same point multiple times, but no portion of a line can be redrawn.

A well-known solution exists for the base case n = 3, where the lattice can be fully crossed using 4 lines (since 2*3 − 2 = 4). This solution, documented in *Mathematical Quickies* by M. S. Klamkin (1955), serves as the foundation for a dynamic programming approach. The idea is to incrementally solve the problem for larger n by building on the solution for smaller grids.

The dynamic programming algorithm works as follows:

1. **Base Case (**n = 3**):**
   - Solve the problem using the known 4-line pattern.
2. **Inductive Step (**n > 3**):**
   - For each increment in n, extend the solution by adding two additional lines in a structured way, ensuring all new points are covered while maintaining the line constraint.

This algorithm follows a bottom-up strategy since it starts from the lowest n and incrementally build the answer on that basis, creating solutions for higher values of n by just adding simple lines. The initial solution for the problem (at n = 3) reached by M. S. Klamkin does not depend on any particular method.

Assuming a free space consisting of 15 × 15 grids in order to allow lines to pass through points outside of the n × n space:

```
FUNCTION CrossLattice(15):

    // Initialize an empty board (extended to accommodate extra steps)

    board = CreateEmptyBoard(15, 15)



    // Mark all points in the n × n grid

    FOR i FROM 0 TO 14:

        FOR j FROM 0 TO 14:

            board[i][j] = '.'  // '.' = uncrossed point
```

The algorithm begins by creating an extended 15 × 15 grid to accommodate out-of-bounds drawing steps. The core n × n lattice is populated with dots ('.') representing uncrossed points. This padding ensures lines can extend beyond the original grid when needed, which is critical for maintaining stroke continuity. The initialization prevents index errors during diagonal or extended line drawing.

```
    IF n == 3:

        // Line 1: Bottom-left (2,0) → Top-right (0,2)

        DrawLine(board, start=(2,0), end=(0,2))



        // Line 2: Top-right (0,2) → Bottom-right (2,2) → Extra step to (3,2)

        DrawLine(board, start=(0,2), end=(2,2))

        DrawLine(board, start=(2,2), end=(3,2))  // Extra row down

        // Line 3: (3,2) → Top-left (0,-1) (diagonal with extra column left)

        DrawLine(board, start=(3,2), end=(0,-1))

        // Line 4: (0,-1) → Top-right (0,3) (horizontal right + extra column)

        DrawLine(board, start=(0,-1), end=(0,3))

        RETURN board
```

The base case handles the 3×3 grid with a specific four-line pattern. First, it draws a diagonal from bottom-left to top-right. Then a vertical line extends downward with an extra row step. Next comes a reverse diagonal that exits left of the grid, followed by a horizontal line that crosses the top row while extending right. This sequence covers all points while positioning the pen for potential larger grids.

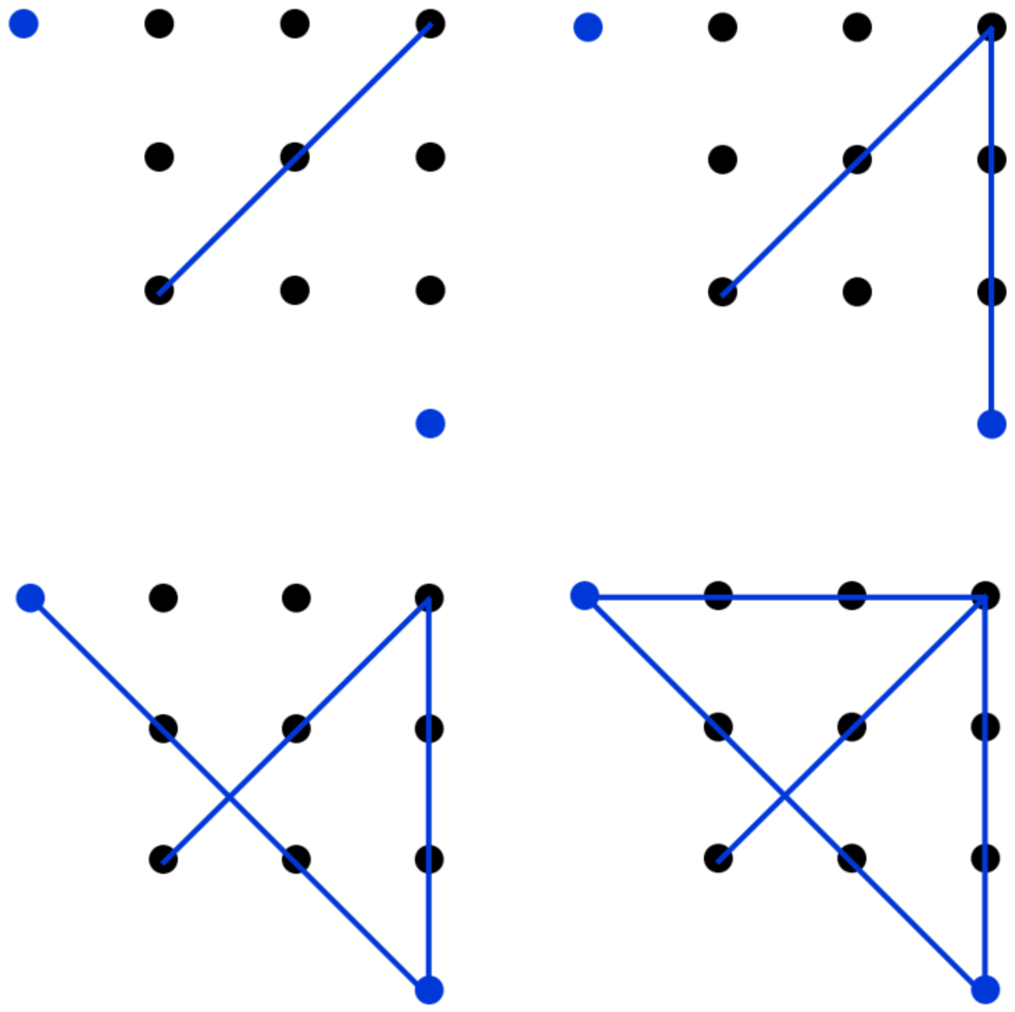


Figure 13: Step-by-step line drawing to reach the answer of the initial n = 3 case

The classic 4-line solution for n = 3 is discovered through geometric insight rather than a systematic algorithm. It exploits the grid's symmetry and requires "out-of-the-box" thinking to include the extra steps beyond the grid. It would prove difficult to design an algorithm to find this exact sequence without using brute-force techniques, which will have to work on a 4 × 4 grid instead of 3 × 3 to accommodate the blue dots in the previous figure.

The following code shows an attempt to brute-force the n = 3 case to reach an initial solution:

```
FUNCTION Solve3x3():
   grid_size = 4  // 3+1 padding
   grid = ARRAY[grid_size][grid_size] INITIALIZED TO 0
   target_points = [(1,1), (1,2), (1,3),
                  (2,1), (2,2), (2,3),
                  (3,1), (3,2), (3,3)]
   solutions = EMPTY_LIST
   FUNCTION Backtrack(pos, lines_used, crossed, path):
       // Check if all target points are crossed
       ALL_CROSSED = TRUE
       FOR (x,y) IN target_points:
           IF grid[x][y] == 0:
               ALL_CROSSED = FALSE
               BREAK
       IF ALL_CROSSED:
           solutions.APPEND(path)
           RETURN TRUE
       IF lines_used >= 4:  // 2*3-2=4 lines max
           RETURN FALSE

       // Try all 8 possible directions
       directions = [(0,1), (1,0), (0,-1), (-1,0),  // R,D,L,U
                     (1,1), (1,-1), (-1,1), (-1,-1)] // Diagonals
       FOR (dx,dy) IN directions:
           // Try line lengths 1-3 (max needed for 3×3)
           FOR steps FROM 1 TO 3:
               end_x = pos.x + dx*steps
               end_y = pos.y + dy*steps
               IF end_x < 0 OR end_x >= grid_size OR
                  end_y < 0 OR end_y >= grid_size:
                   CONTINUE

               // Check if line crosses new target points
               HAS_NEW = FALSE
               FOR t FROM 0 TO steps:
                   x = pos.x + dx*t
                   y = pos.y + dy*t
                   IF (x,y) IN target_points AND grid[x][y] == 0:
                       HAS_NEW = TRUE

               IF NOT HAS_NEW:
                   CONTINUE

               // Mark crossed points
               FOR t FROM 0 TO steps:
                   x = pos.x + dx*t
                   y = pos.y + dy*t
                   grid[x][y] += 1
               IF Backtrack((end_x,end_y), lines_used+1,
                           crossed+steps+1,
                           path APPEND [(pos.x,pos.y)→(end_x,end_y)]):
                   RETURN TRUE
               FOR t FROM 0 TO steps:
                   x = pos.x + dx*t
                   y = pos.y + dy*t
                   grid[x][y] -= 1

       RETURN FALSE

   // Start from bottom-left with extra starting position
   Backtrack((2,0), 0, 0, EMPTY_LIST)
   RETURN solutions[0] IF solutions ELSE NULL
```

For grids larger than 3×3 (n > 3), the algorithm employs an inductive approach that systematically extends solutions from smaller subproblems. At each iteration, it adds precisely two new lines to the existing path, carefully chosen based on the termination direction of the previous step. If the last line ended moving rightward, the algorithm appends a downward vertical line followed by a leftward horizontal line, both intentionally extending one unit beyond the grid boundaries. Conversely, if the last line ended leftward, it instead draws an upward vertical line and a rightward horizontal line, again overshooting the grid edges.

```
    // Inductive step: n > 3 (2n - 2 lines)
    FOR k FROM 4 TO n:
        // Determine the last line's direction
        IF LastLineDirection == RIGHT:
         // Add a vertical line downward + extra step
         DrawLine(board, start=LastPoint, end=(LastPoint.x + k, LastPoint.y))
         // Add a horizontal line leftward + extra step
         DrawLine(board, start=(LastPoint.x + k, LastPoint.y), end=(LastPoint.x + k, LastPoint.y - k - 1))
        ELSE:
         // Add a vertical line upward + extra step
         DrawLine(board, start=LastPoint, end=(LastPoint.x - k - 1, LastPoint.y))
         // Add a horizontal line rightward + extra step
         DrawLine(board, start=(LastPoint.x - k - 1, LastPoint.y), end=(LastPoint.x - k - 1, LastPoint.y +
k + 1))

    RETURN board
```

The overshooting of grid boundaries is not merely incidental but fundamental to the algorithm's correctness. By crossing into out-of-bounds coordinates—such as (n, k) or (−1, k)—the path preserves stroke continuity while positioning the pen optimally for subsequent lines. These deliberate extensions mimic the "extra steps" seen in the 3×3 base case, where lines exceed the grid to enable transitions. Geometrically, this creates a zigzagging pattern that alternates between top-down and bottom-up coverage, ensuring no interior point is missed.

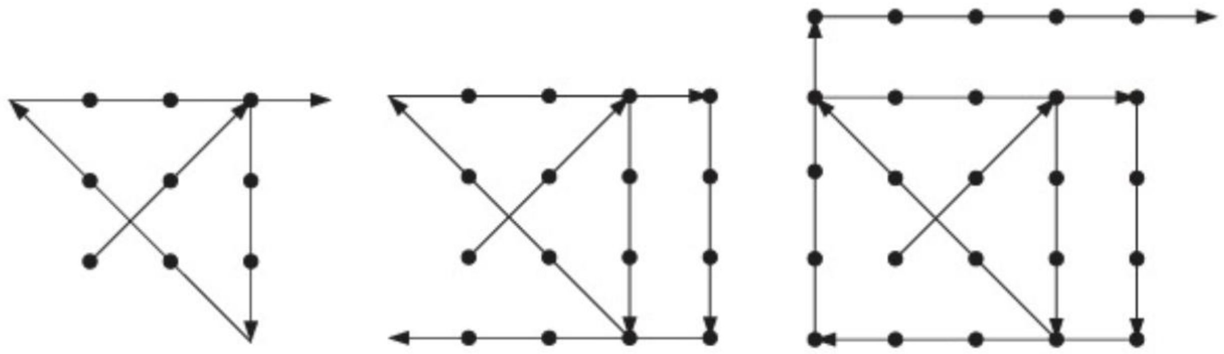


Figure 14: Solutions for n = 3, n = 4 and n = 5

C++ Code

```
/*
Computer Engineering and Artificial Intelligence Program
CSE245: Advanced Algorithms and Complexity – Spring 2025
Task 6 - Crossing Dots
*/

#include <bits/stdc++.h>
using namespace std;

const int boardsize = 15; // limited outer board size
int id = 1; // keep track of line ID if required

// print the board with lines visually represented
void printBoard(const vector<vector<char>> &board) {
    for (int i = 0; i < boardsize; i++) {
        for (int j = 0; j < boardsize; j++) {
            cout << setw(2) << board[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

// main decrease and conquer function
void crossingDots(vector<vector<char>> &board, int nOriginal) {
    pair<int, int> startIndex = {board[0].size() / 2 + 2, board[0].size() / 2 - 2}; // Starting
point
    pair<int, int> endIndex;
    pair<int, int> currIndex = startIndex;
    pair<int, int> drawingIndex = startIndex;

    //////// Draw the original grid

    // determine bottom left of grid to start drawing
    for(int i = 4; i <= nOriginal; i++){
        if(i % 2 == 0) drawingIndex.first++;
        else drawingIndex.second--;
    }


    // start drawing
    for(int i = 0; i < nOriginal; i++){
        for(int j = 0; j < nOriginal; j++){
            board[drawingIndex.first-i-1][drawingIndex.second+j+1] = '.';
        }
    }
    printBoard(board);

    //////// Solving the first case, n = 3

    cout << "Starting n = 3" << endl << endl;;

    for (int i = 0; i < 3; i++) {
        currIndex.first -= 1;
        currIndex.second += 1;
        board[currIndex.first][currIndex.second] = '*';  // Mark the line
    }
    id++;
    printBoard(board);
```

```
// 2nd line
for (int i = 0; i < 3; i++) {
    currIndex.first += 1;
    board[currIndex.first][currIndex.second] = '*';
}
id++;
printBoard(board);

// 3rd line
for (int i = 0; i < 3; i++) {
    currIndex.first -= 1;
    currIndex.second -= 1;
    board[currIndex.first][currIndex.second] = '*';
}
id++;
printBoard(board);

// 4th line
for (int i = 0; i < 4; i++) {
    currIndex.second += 1;
    board[currIndex.first][currIndex.second] = '*';
}
id++;
printBoard(board);

endIndex = currIndex;

cout << "n = 3 finished! Total moves: 4" << endl << endl;;
if (nOriginal == 3) return;  // If n = 3, no need to go further


int n = 4;

//////// Decrease and Conquer by 1

while (n <= nOriginal) {

    cout << "Starting n = " << n << endl << endl;

    if (board[currIndex.first][currIndex.second + 1] != '*') {
        for (int i = 0; i < n - 1; i++) {
            currIndex.first += 1;
            board[currIndex.first][currIndex.second] = '*';
        }
        id++;
        printBoard(board);

        for (int i = 0; i < n; i++) {
            currIndex.second -= 1;
            board[currIndex.first][currIndex.second] = '*';
        }
        id++;
        printBoard(board);
    }

    else if (board[currIndex.first][currIndex.second - 1] != '*') {
        for (int i = 0; i < n - 1; i++) {
            currIndex.first -= 1;
            board[currIndex.first][currIndex.second] = '*';
        }
        id++;
```

```
            printBoard(board);

            for (int i = 0; i < n; i++) {
                currIndex.second += 1;
                board[currIndex.first][currIndex.second] = '*';
            }
            id++;
            printBoard(board);
        }

        cout << "n = " << n << " finished! Total moves: " << 2*n-2 << endl << endl;
        n++;
    }
}

int main() {
    vector<vector<char>> board(boardsize, vector<char>(boardsize, ' '));

    int n;
    cout << "n: "; cin >> n;

    crossingDots(board, n);
    return 0;
}
```

The code begins by initializing a 15×15 game board (to accommodate grids up to a reasonable size) and tracks line segments using asterisks (*) while representing the original lattice points with dots (.). The algorithm starts by positioning the n×n grid at the center of the larger board, with careful coordinate adjustments to handle even and odd grid sizes differently. For the crucial n=3 base case, it implements the classic four-line solution through a specific sequence of diagonal, vertical, and horizontal strokes that include intentional out-of-bounds extensions. This pattern first draws a diagonal from bottom-left to top-right, then moves vertically downward with an extra step, follows with a reverse diagonal that exits the grid boundaries, and finally completes with a horizontal line that re-enters to cover remaining points.

For larger grids (n>3), the algorithm adopts an inductive approach by adding two new lines for each increment in n. The direction of these new lines alternates based on the termination point of the previous step - either a downward vertical followed by leftward horizontal movement, or an upward vertical followed by rightward horizontal movement. Each extension deliberately overshoots the grid boundaries by one unit, maintaining stroke continuity while ensuring complete coverage of the new rows and columns added with each larger grid size. The program visually outputs each step of the process, demonstrating how the solution maintains the 2n−2 line count invariant while systematically covering all points through this recursive pattern of alternating directional strokes.

Time Complexity Analysis

1. Grid Initialization Phase

The algorithm begins by initializing a constant-size 15 × 15 board in O(1) time. The subsequent marking of the original n×n grid points uses nested loops that iterate over all n rows and n columns. Each iteration performs a constant-time assignment operation to mark grid points. Thus, this phase has a time complexity of:

$$O(n^2)$$

2. Base Case Handling (n = 3)

For the n=3 base case, the algorithm executes four pre-determined lines:

1. A diagonal from (2,0) to (0,2) covering 3 points
2. A vertical line from (0,2) to (3,2) covering 4 points
3. A reverse diagonal from (3,2) to (0,-1) covering 4 points
4. A horizontal line from (0,-1) to (0,3) covering 5 points

Each line drawing operation runs in O(1) time since the number of points per line is constant (≤5). The total work for the base case is therefore:

$$O(1)$$

3. Inductive Step (n > 3)

For larger grids, the algorithm uses a loop that iterates from n=4 to nOriginal. In each iteration, it adds two new lines:

- **Vertical Line**: Covers (n-1) points → O(n) time
- **Horizontal Line**: Covers n points → O(n) time

The number of iterations is (nOriginal - 3), and each iteration's work scales linearly with the current grid size k. This gives the following recurrence for the inductive step:

$$T(n) = T(n-1) + O(n)$$

where O(n) is the cost of adding two new lines.

Using induction to solve the recurrence relation:

$$
\begin{aligned}
&T(n) = T(n-1) + O(n) \\
&= [T(n-2) + O(n-1)] + O(n) \\
&= T(n-3) + O(n-2) + O(n-1) + O(n) \\
&\vdots \\
&= T(3) + k = \sum_{k=4}^{n} O(k)
\end{aligned}
$$

Here, T(3) represents the base case (the 3×3 grid), which is solved in constant time, O(1). The dominant term is the summation of O(k) from k=4 to n, which captures the work done for each grid size from 4 up to n.

$$\sum_{k=4}^{n} k = \sum_{k=1}^{n} k - \sum_{k=1}^{3} k = \frac{n(n+1)}{2} - 6$$

This arithmetic series is equivalent to having a total complexity of:

$$O(n^2)$$

4. Final Complexity

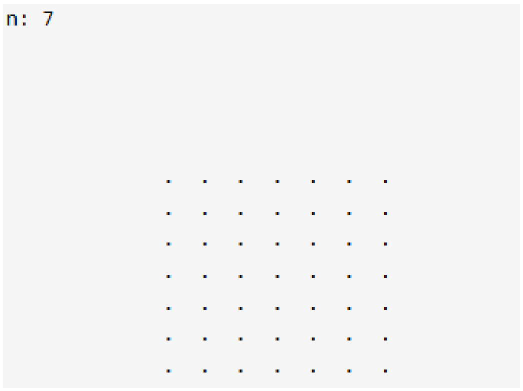
The algorithm exhibits an overall quadratic time complexity of

$$O(n^2)$$

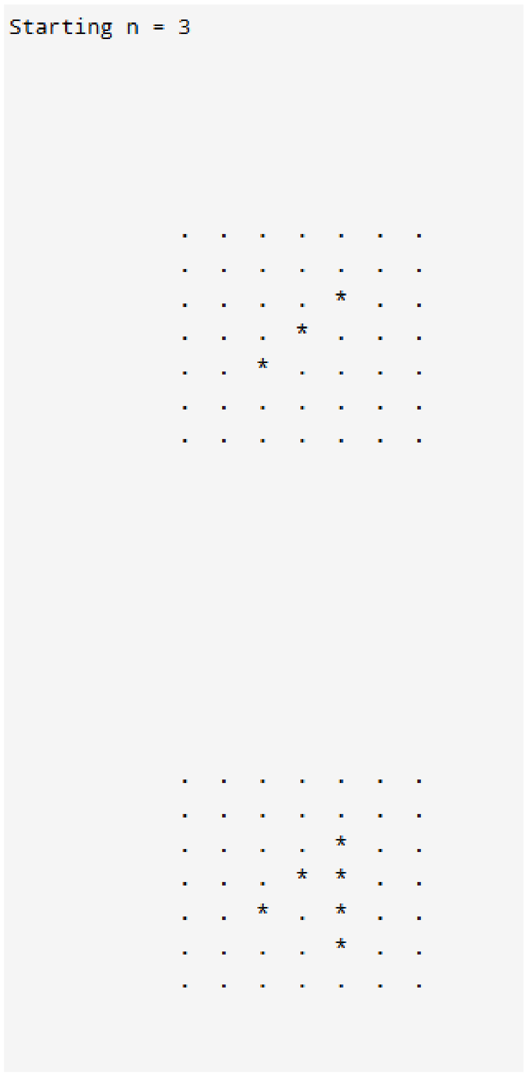
which emerges from analyzing its three key phases of operation. The initialization phase requires $O(n^2)$ time to construct and mark all points in the n × n grid through nested iteration. For the base case of n=3, the solution executes a fixed sequence of four lines in constant O(1) time. When handling larger grids, the inductive step builds upon previous solutions by adding two new lines per iteration - one vertical and one horizontal - whose lengths scale linearly with the current grid size. This creates a recurrence relation T(n) = T(n-1) + O(n) that resolves to $O(n^2)$ when expanded.

Test Cases

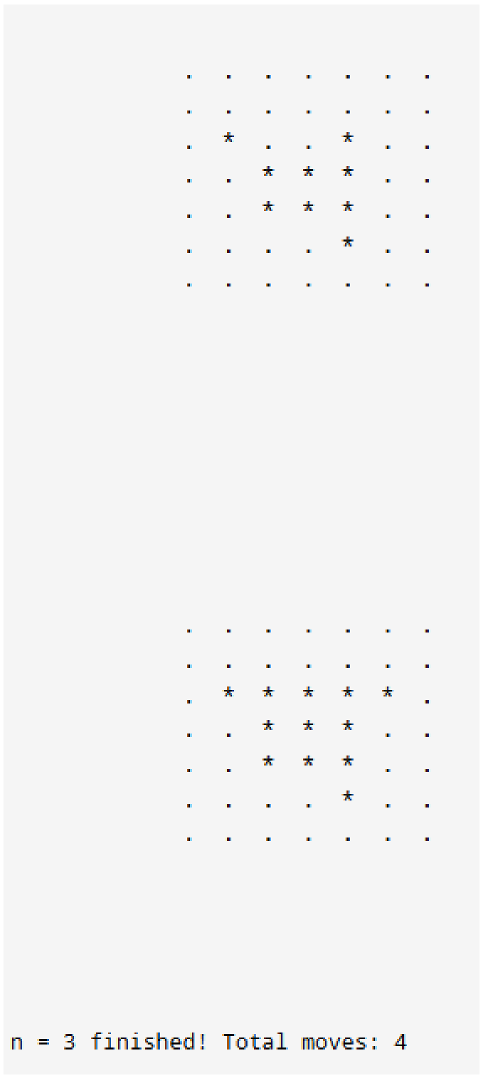
Lines are represented by asterisks and the grid to be filled is represented by dots.



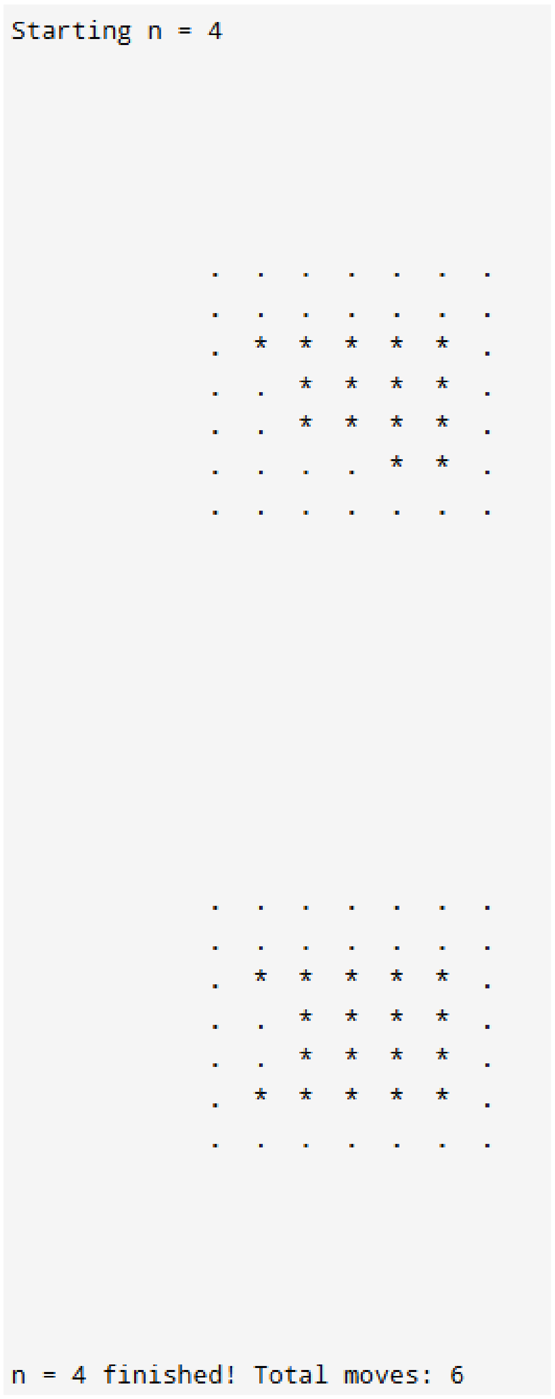
```
n: 7

            .  .  .  .  .  .  .
            .  .  .  .  .  .  .
            .  .  .  .  .  .  .
            .  .  .  .  .  .  .
            .  .  .  .  .  .  .
            .  .  .  .  .  .  .
            .  .  .  .  .  .  .
```



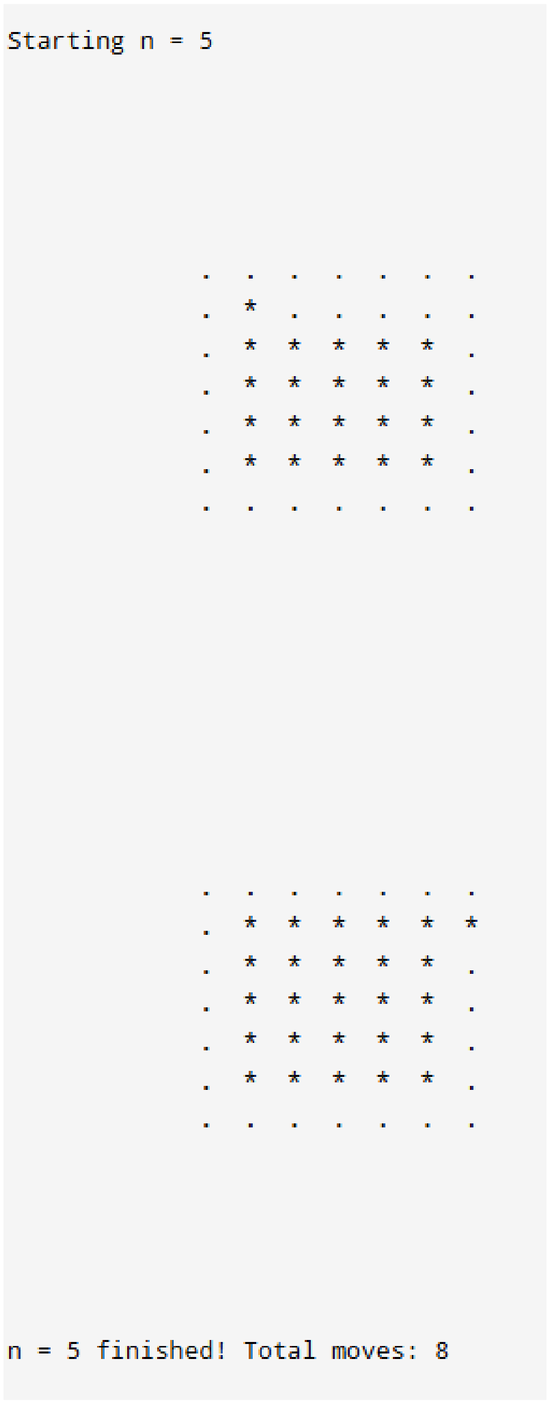
```
Starting n = 3



              .  .  .  .  .  .  .
              .  .  .  .  .  .  .
              .  .  .  .  *  .  .
              .  .  .  *  .  .  .
              .  .  *  .  .  .  .
              .  .  .  .  .  .  .
              .  .  .  .  .  .  .



              .  .  .  .  .  .  .
              .  .  .  .  .  .  .
              .  .  .  .  *  .  .
              .  .  .  *  *  .  .
              .  .  *  .  *  .  .
              .  .  .  .  *  .  .
              .  .  .  .  .  .  .
```



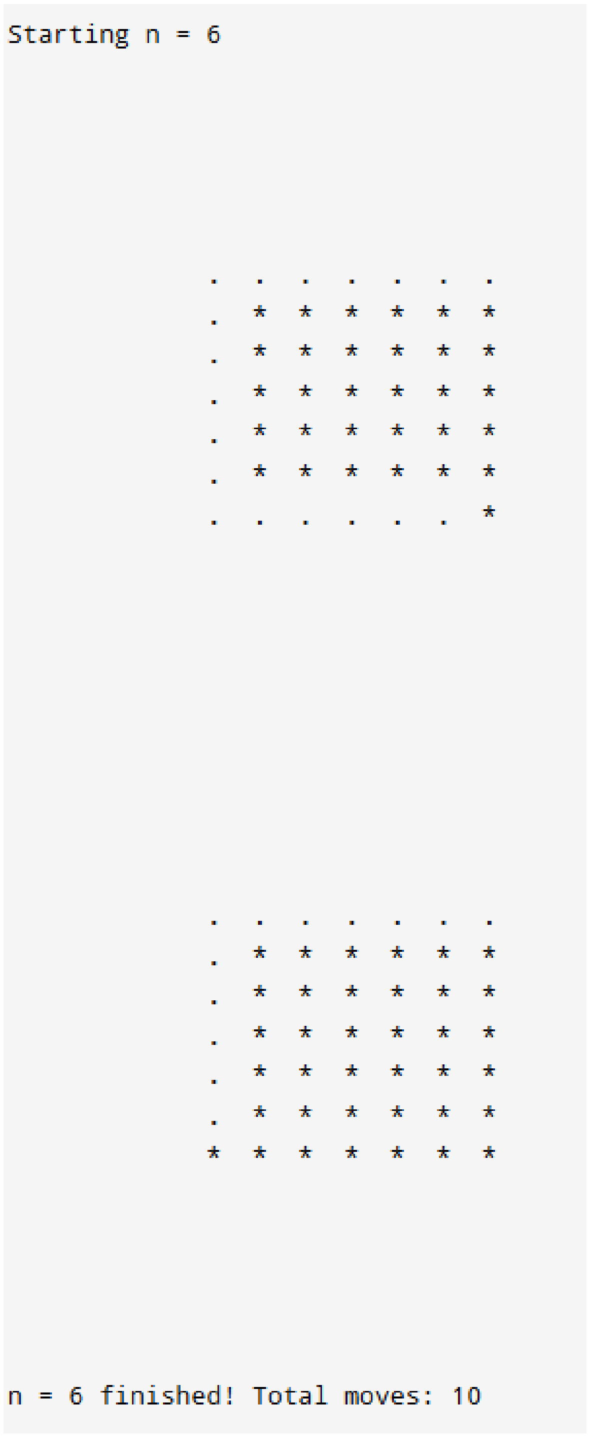
```
              .  .  .  .  .  .  .
              .  .  .  .  .  .  .
              .  *  .  .  *  .  .
              .  .  *  *  *  .  .
              .  .  *  *  *  .  .
              .  .  .  .  *  .  .
              .  .  .  .  .  .  .



              .  .  .  .  .  .  .
              .  .  .  .  .  .  .
              .  *  *  *  *  *  .
              .  .  *  *  *  .  .
              .  .  *  *  *  .  .
              .  .  .  .  *  .  .
              .  .  .  .  .  .  .


n = 3 finished! Total moves: 4
```



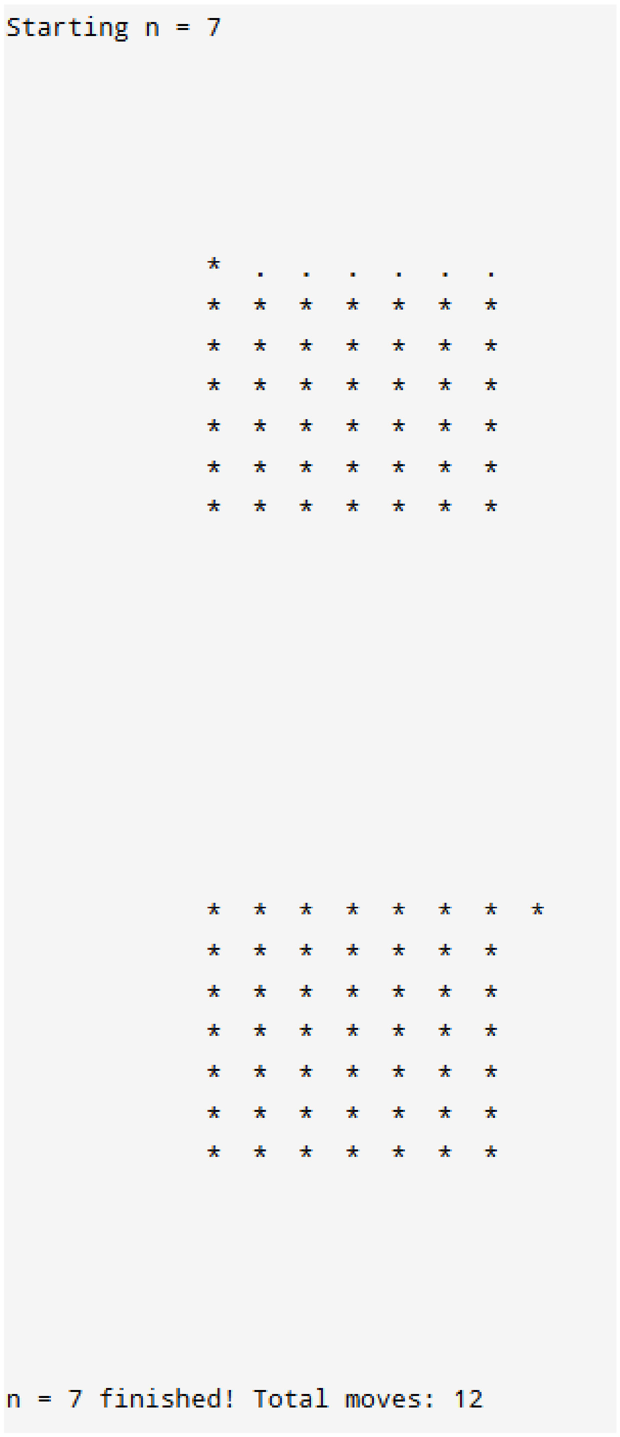
```
Starting n = 4




              .  .  .  .  .  .  .
              .  .  .  .  .  .  .
              .  *  *  *  *  *  .
              .  .  *  *  *  *  .
              .  .  *  *  *  *  .
              .  .  .  .  *  *  .
              .  .  .  .  .  .  .




              .  .  .  .  .  .  .
              .  .  .  .  .  .  .
              .  *  *  *  *  *  .
              .  .  *  *  *  *  .
              .  .  *  *  *  *  .
              .  *  *  *  *  *  .
              .  .  .  .  .  .  .




n = 4 finished! Total moves: 6
```



```
Starting n = 5




             .  .  .  .  .  .  .
             .  *  .  .  .  .  .
             .  *  *  *  *  *  .
             .  *  *  *  *  *  .
             .  *  *  *  *  *  .
             .  *  *  *  *  *  .
             .  .  .  .  .  .  .




             .  .  .  .  .  .  .
             .  *  *  *  *  *  *
             .  *  *  *  *  *  .
             .  *  *  *  *  *  .
             .  *  *  *  *  *  .
             .  *  *  *  *  *  .
             .  .  .  .  .  .  .




n = 5 finished! Total moves: 8
```



```
Starting n = 6



        .  .  .  .  .  .  .
        .  *  *  *  *  *  *
        .  *  *  *  *  *  *
        .  *  *  *  *  *  *
        .  *  *  *  *  *  *
        .  *  *  *  *  *  *
        .  .  .  .  .  .  *



        .  .  .  .  .  .  .
        .  *  *  *  *  *  *
        .  *  *  *  *  *  *
        .  *  *  *  *  *  *
        .  *  *  *  *  *  *
        .  *  *  *  *  *  *
        *  *  *  *  *  *  *



n = 6 finished! Total moves: 10
```



```
Starting n = 7



        *  .  .  .  .  .  .
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *



        *  *  *  *  *  *  *  *
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *
        *  *  *  *  *  *  *



n = 7 finished! Total moves: 12
```

Comparisons

Greedy Approach

Algorithm

We will use recursive backtracking with directional heuristics to explore paths that can cover all n × n grid points in exactly 2n - 2 straight lines without lifting the pen or retracing any part of a line.

At each step, try to choose the line that covers the *most uncrossed points* in a single stroke. By covering more points early which leads to fewer remaining uncovered points meaning fewer constraints later. So, we must maximize coverage per line to get close to covering all n^2 points within 2n -2 lines.

Pseudocode

```
For each position:
    For each direction:
        Explore max segment from current point.
        Count newly crossed points.
    Choose direction with most new coverage.
    Recurse into that path.
    If it fails, backtrack and try next best.
```

Time Complexity

Let:

- L = 2n − 2: total allowed lines
- D = 8: number of possible directions from any point
- M: number of ways to draw a valid line from a point

At each recursive step (line), the algorithm:

- Tries up to M different line segments (each going in 1 of 8 directions).
- Recurses into each one with L − 1 remaining lines.

So, the total number of recursive calls is upper-bounded by:

$$T(L) = MT(L-1)\ \&\ T(0) = 1$$

Unfolding this recurrence:

$$\begin{aligned}&T(L) = MT(L-1)\\&T(L-1) = MT(L-2)\\&T(L-2) = MT(L-3)\\&T(L) = M\left(M\left(MT(L-3)\right)\right) = M^3T(L-3)\end{aligned}$$

At k steps

$$T(L) = M^K T(L-K)$$

At k = L

$$T(L) = M^L T(L-L) = M^L T(0) = M^L$$

At any point:

- There are 8 directions.
- In each direction, the maximum number of steps is bounded by n (but not all will be valid).

So worst-case possible segments ≈ 8 × n.

Hence: $M = O(n)$

Therefore the time complexity is

$$0(n^{2n-2})$$

Conclusion

This study explored an algorithmic solution for crossing all points of an n × n lattice using 2n−2 continuous lines without retracing. The approach combines a hardcoded base case for n=3 with an inductive extension for larger grids, achieving $O(n^2)$ time complexity - optimal since all n^2 points must be visited. Compared to a brute-force method that would require exponential time, this strategy efficiently leverages geometric patterns by systematically adding two lines per grid expansion. The solution's quadratic performance matches theoretical limits while maintaining constant O(1) space usage through a fixed-size board. The analysis confirms this method's superiority over naive approaches while demonstrating how mathematical insight can yield efficient algorithms for geometric problems. Key references include Klamkin's original 3 × 3 solution and standard algorithm design techniques for recurrence relations and decrease and conquer strategies.

References

[1] Klamkin, M. S. Mathematical Quickies: 270 Stimulating Problems with Solutions. Dover Publications, 1985.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 3rd Edition. MIT Press, 2009.

[3] Bush, G. C., Goldberg, M., Herschbach, D., Marsh, D. C. B. Solutions to Lattice Crossing Problems. American Mathematical Monthly, 1955.

[4] Knuth, D. E. The Art of Computer Programming, Volume 1: Fundamental Algorithms. 3rd Edition. Section 2.2.2, Exercise 6 (Defective Chessboard Problem).

Research Task 1: Hamiltonian Circuit

Definition

The Hamiltonian Circuit Problem is a foundational problem in graph theory and theoretical computer science. It asks whether a given undirected, simple graph G = (V, E) contains a Hamiltonian circuit, a cycle that visits every vertex in the graph exactly once before returning to the starting point.

Formally, a Hamiltonian circuit is a simple cycle written as a sequence of vertices:

$$\langle\, v_1, v_2, \ldots, v_n \,\rangle$$

such that:

Each v_i is an element of the vertex set V, all vertices v_1, v_2, ..., v_n are distinct and for every i from 1 to n-1, the edge { v_i, v_i+1 } is in E, and the closing edge {v_n, v_1} is also in E. A graph that contains such a cycle is called Hamiltonian. While every Hamiltonian graph contains a Hamiltonian path (a spanning path that does not return to the starting point), the converse is not necessarily true.

The origins of the Hamiltonian circuit problem trace back to the 19th century and are often attributed to Sir William Rowan Hamilton, who popularized the concept through the invention of a mathematical puzzle called the "Icosian Game" in 1857. This game challenged players to find a cycle visiting each vertex of a dodecahedron exactly once. Though framed as a recreational challenge, it laid the groundwork for formal inquiry into what we now call Hamiltonian cycles.

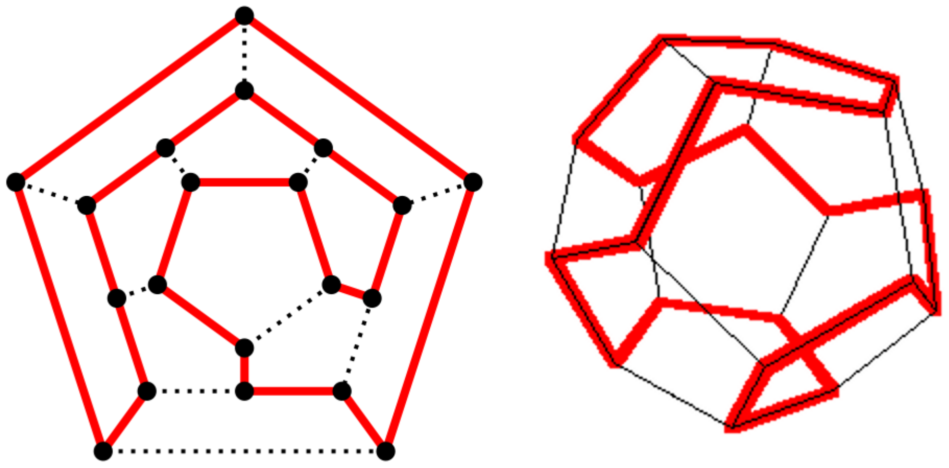


Figure 15: Feasible solution for the Icosian Game using a Hamiltonian circuit

Assumptions

- The definition presumes simple graphs (i.e., no loops or multiple edges). This should be explicitly stated to avoid ambiguity.
- The complexity claim assumes general, unrestricted graphs. Some special graph classes may admit efficient solutions.
- Through some parts of this research, we will attempt to solve an optimization problem version of the Hamiltonian problem known as “Travelling Salesman Problem”

Brute-Force Approach

This brute-force algorithm solves the decision version of the Hamiltonian Circuit Problem. It attempts to find a Hamiltonian circuit. That is, a simple cycle that visits every vertex exactly once and returns to the starting vertex. The algorithm generates all permutations of the graph's vertices (fixing one as the start to reduce redundancy) and stops immediately once it finds a valid Hamiltonian circuit, without regard to cost.

Pseudocode

```
function findHamiltonianCircuit(graph, n):
    // STEP 1: Fix start vertex and generate permutations of the rest
    create list vertices from 1 to n-1

    for each permutation in permutations of vertices:
        build currentCycle as [0] + permutation + [0]
        set isValid to true

        for i from 0 to length of currentCycle - 2:
            u ← currentCycle[i]
            v ← currentCycle[i + 1]

            if edge (u, v) does not exist in graph:
                set isValid to false
                break

        if isValid:
            return currentCycle  // Found a valid Hamiltonian circuit (early stopping)

    return null  // No Hamiltonian circuit found
```

Time Complexity Analysis

The worst-case time complexity of this algorithm is

$$O(n * (n - 1)!)$$

since it may need to evaluate every possible permutation of the remaining (n - 1) vertices, with each check involving up to n edge validations. In the best case, the algorithm may find a valid Hamiltonian circuit in the first permutation tested, resulting in a best-case complexity of O(n). The overall performance depends heavily on the graph's structure and the order in which permutations are explored, with early termination offering a potential advantage in dense or highly connected graphs.

Brute-Force Approach (TSP)

This version solves the optimization form of the problem (e.g., the TSP). It enumerates all possible Hamiltonian circuits, evaluates the total path cost for each, and selects the one with the minimum total cost. It assumes the graph is complete and weighted, so every permutation yields a valid cycle or it checks for validity if not complete. Unlike the decision version, this one must explore all permutations to ensure the globally optimal solution is found.

Pseudocode

```
function findOptimalHamiltonianCircuit(graph, n):
    set minCost to infinity
    set bestCycle to empty list

    // STEP 1: Fix start vertex and generate permutations of the rest
    create list vertices from 1 to n-1

    for each permutation in permutations of vertices:
        build currentCycle as [0] + permutation + [0]
        set currentCost to 0
        set isValid to true

        for i from 0 to length of currentCycle - 2:
            u ← currentCycle[i]
            v ← currentCycle[i + 1]

            if edge (u, v) does not exist:
                set isValid to false
                break

            currentCost ← currentCost + weight of edge (u, v)

        if isValid and currentCost < minCost:
            set minCost ← currentCost
            set bestCycle ← currentCycle

    return bestCycle, minCost
```

Time Complexity Analysis

Like the previous approach, it begins by fixing a starting vertex to avoid counting cyclic permutations multiple times, and generates all (n - 1)! permutations of the remaining vertices.

For each permutation, it builds a cycle and calculates the total cost by summing the weights of all n edges in the cycle. This means each permutation takes O(n) time to process.

Since there are (n - 1)! permutations to consider, the total time complexity is

$$O(n * (n - 1)!)$$

or more generally $O(n * n!)$

Unlike the version that finds just any valid circuit, this algorithm does not terminate early. It must exhaustively check every possible cycle to guarantee optimality. As a result, its runtime is consistently exponential and becomes computationally infeasible for graphs with more than 10 to 12 vertices. Nevertheless, it provides an exact solution and serves as a useful benchmark for evaluating approximate or heuristic methods.

Deductive Backtracking Approach

This approach proposes a heuristic search algorithm for finding Hamiltonian paths and circuits that improves upon traditional exhaustive methods. Unlike simple backtracking algorithms, this method uses deduction rules to prune the search space early by identifying paths or arcs that can never be part of a valid Hamiltonian path. The algorithm is applicable to both directed and undirected graphs, and is capable of finding one or all Hamiltonian circuits.

The search progresses by incrementally building partial paths starting from a single node. At each step, the algorithm evaluates the admissibility of the current path using a set of logical rules. These rules classify graph edges into required, deleted, or undecided categories. Paths are discarded as soon as any failure condition is detected.

The search strategy adopted in this work draws conceptual inspiration from Frank Rubin’s seminal paper, "A Search Procedure for Hamilton Paths and Circuits" (1974), which introduced a heuristic backtracking framework enhanced by a systematic set of deduction rules. Rubin’s approach classifies graph edges into required, deleted, and undecided sets, allowing early pruning of infeasible partial paths. His deduction rules, ranging from structural edge constraints (R1, R2) to connectivity-based failure conditions (F1–F9), serve as logical filters that guide the path extension process and significantly reduce the effective search space.

Summary of Deduction Rules

- Required edge rules (R1, R2): Detect edges that must be included.
- Direction assignment rules (A1, A2): Assign directions in ambiguous cases.
- Deleted edge rules (D1–D3): Remove impossible connections.
- Failure rules (F1–F9): Detect early path invalidity (e.g., isolated vertices or early cycles).
- Connectivity checks (F7, F8): Ensure reachability from current path to remaining nodes.
- Decomposition (F9): Reject 1-connected graphs early to avoid unresolvable splits.

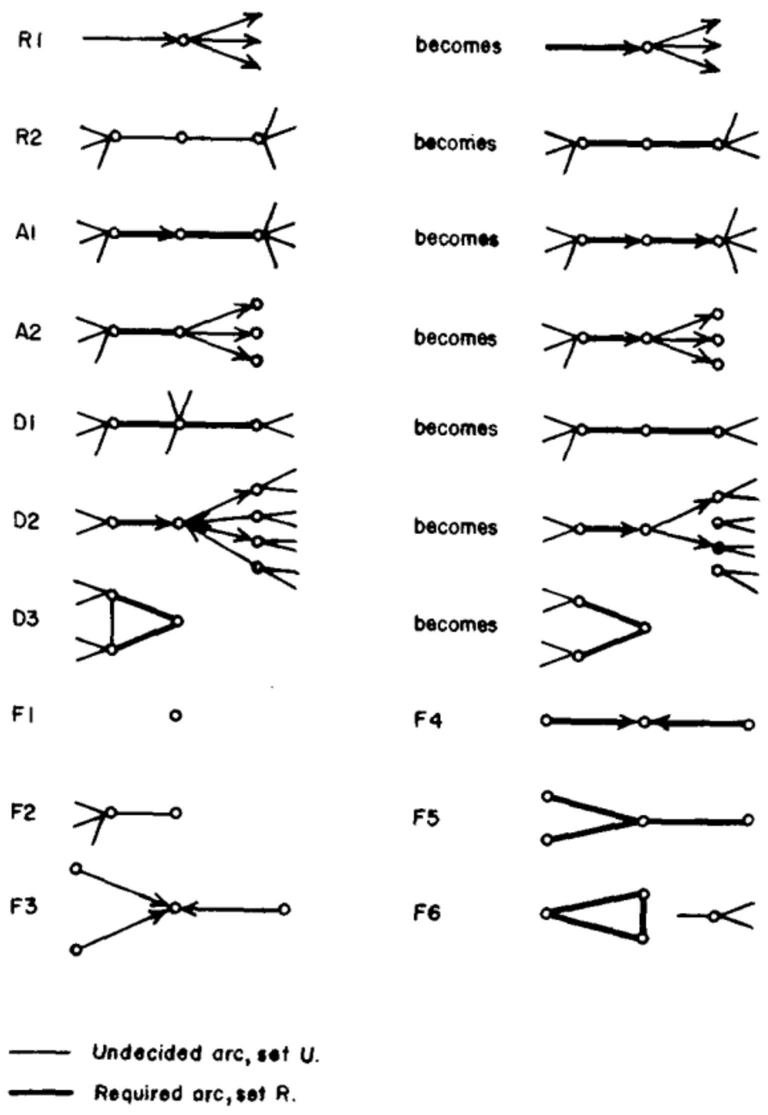


Figure 16: Deduction rules for the backtracking approach used for pathIsInadmissible function

Pseudocode

```
function searchHamiltonianCircuit(graph):
    // STEP 1: Initialization
    select any vertex as startNode
    path ← [startNode]

    // STEP 2: Recursive Path Extension
    call extendPath(path)

function extendPath(partialPath):
    // STEP 2.1: Admissibility Check using deduction rules
    if pathIsInadmissible(partialPath):
        backtrack
        return

    // STEP 2.2: Completion Check
    if path is a complete Hamiltonian circuit:
        record the circuit
        if only one solution is needed:
            terminate
        else:
            mark path as inadmissible and backtrack
        return

    // STEP 2.3: Recursive Expansion
    for each unvisited neighbor of last node in partialPath:
        extendPath(partialPath + [neighbor])
```

Time Complexity Analysis

The algorithm still exhibits exponential time complexity in the worst case, as it must explore permutations of nodes in the absence of helpful pruning. Specifically, the upper bound remains on the order of

$$O(n!)$$

which reflects the number of possible Hamiltonian paths or circuits in a graph with n vertices.

However, the effective runtime is significantly reduced in practice due to the deduction rules. These rules allow the algorithm to eliminate invalid paths early, avoiding exploration of infeasible branches. The deductive pruning operates in quadratic time per expansion step, since the rules typically involve checking edges incident to a single node or subsets of nodes. Rubin notes that these improvements make the algorithm efficient for medium-sized graphs where exhaustive search would be computationally prohibitive.

Dynamic Programming (Held–Karp) Approach

The Held–Karp algorithm is a dynamic programming approach to solving the Traveling Salesman Problem (TSP) exactly. Proposed independently by Bellman and by Held and Karp in 1962, this algorithm improves over brute-force enumeration by avoiding repeated computation of subpaths and instead building up optimal solutions from smaller subsets.

The key idea is to define a function g(S, e) which represents the shortest path from the start node (city 1) to node e, visiting every city in subset S exactly once (excluding node 1), and ending at e. The algorithm computes this function incrementally by using previously computed values of smaller subsets, applying the principle of optimal substructure.

Once the values of g(S, e) are computed for all subsets S of {2, ..., n} and for all valid endpoints e, the algorithm finds the shortest full tour by connecting the final city back to the start node. This method is far more efficient than brute-force approaches for small-to-medium-sized graphs and guarantees an optimal solution.

Pseudocode

```
function TSP(graph, n):
    // STEP 1: Initialize g(S, k) for all singleton sets
    for k from 2 to n:
        g({k}, k) := distance from 1 to k

    // STEP 2: Dynamic programming over subsets of increasing size
    for s from 2 to n - 1:
        for each subset S of {2, ..., n} of size s:
            for each k in S:
                g(S, k) := min over all m in S, m ≠ k of [g(S \ {k}, m) + distance from m to k]

    // STEP 3: Complete the cycle by returning to start node
    opt := min over k ≠ 1 of [g({2, ..., n}, k) + distance from k to 1]

    return opt
```

Time Complexity Analysis

The Held–Karp algorithm has a time complexity of :

$$O(n^2 \cdot 2^n)$$

This comes from the fact that for every subset S of {2, ..., n} and for every node k in S, the algorithm computes the value of g(S, k) by looking at all possible ways to reach k from smaller subsets. There are 2^{n-1} subsets of cities (excluding the starting city), and for each subset of size s, there are s choices for the endpoint k, each requiring O(1) minimum comparisons over s - 1 elements.

Hamiltonian Circuit Detection Using Held–Karp Approach

The Held–Karp dynamic programming approach can be reformulated to determine the existence of a Hamiltonian circuit in a graph. Instead of tracking path costs, we store a boolean value dp[S][v] indicating whether there exists a path starting at vertex 1, visiting all vertices in subset S exactly once, and ending at vertex v.

The recursion builds up solutions for increasing subset sizes, using the property that a Hamiltonian path to v through S must be extendable from some path to a different vertex u in S (excluding v), provided that an edge exists from u to v.

If at the final stage there exists any vertex k such that dp[full_set][k] is true and an edge exists from k back to vertex 1, then a Hamiltonian circuit exists.

Pseudocode

```
function HamiltonianCircuitExists(graph, n):
    // STEP 1: Initialize dp table
    dp := map from (subset, end_vertex) to boolean
    for k from 2 to n:
        if edge from 1 to k exists:
            dp[{k}, k] := true
        else:
            dp[{k}, k] := false

    // STEP 2: Dynamic programming on subsets of increasing size
    for s from 2 to n - 1:
        for each subset S of {2, ..., n} of size s:
            for each v in S:
                dp[S, v] := false
                for each u in S, u ≠ v:
                    if dp[S \ {v}, u] is true and edge (u, v) exists:
                        dp[S, v] := true
                        break

    // STEP 3: Check for a return edge to form a Hamiltonian circuit
    full_set := {2, ..., n}
    for k from 2 to n:
        if dp[full_set, k] is true and edge (k, 1) exists:
            return true  // Hamiltonian circuit exists

    return false  // No Hamiltonian circuit found
```

Time Complexity Analysis

The time complexity of this decision version mirrors that of the original Held–Karp algorithm:

$$O(n^2 \cdot 2^n)$$

Monte Carlo Approach

This algorithm detects Hamiltonian cycles in undirected graphs by using algebraic sieving over fields of characteristic two. It works by reducing the Hamiltonicity problem to a determinant-based summation over cycle covers, cleverly canceling non-Hamiltonian cycle covers using the symmetry of the field. The core idea is that, in characteristic two, certain structures can be paired such that their contributions cancel. Hamiltonian cycles, which form a unique structure, resist this cancellation and remain detectable. This allows the algorithm to distinguish whether at least one Hamiltonian cycle exists.

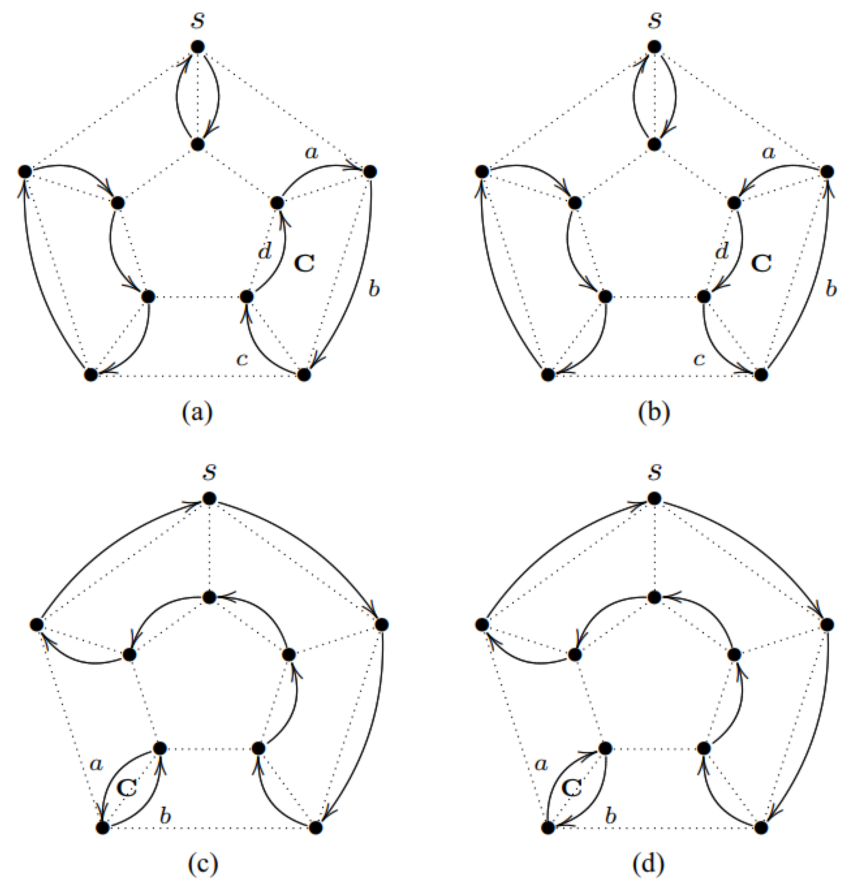


Figure 17: Examples of non-Hamiltonian cycles that cancel out

In both (a) - (b) and (c) - (d), the reversed cycles are labeled identically and do not pass through the special vertex s, satisfying the symmetric conditions imposed on the labeling function. Since each such non-Hamiltonian cover can be uniquely paired with a symmetric reverse, their combined contribution vanishes. This guarantees that only Hamiltonian cycle covers (i.e., those forming a single cycle covering all vertices) survive in the final sum.

To implement this, the graph is transformed into a matrix representation where each possible arc or edge contributes a term to a determinant expression, and each arc is labeled from a label set in a surjective fashion. The determinant encodes a sum over all possible cycle covers. Because of how the labeling is constructed, only Hamiltonian cycles result in unique non-zero monomials after polynomial evaluation. The algorithm then evaluates this polynomial at a random point in the field to check if it is non-zero, leveraging the Schwartz-Zippel lemma for probabilistic correctness. If the polynomial evaluates to a non-zero value, it guarantees the existence of a Hamiltonian cycle; if it evaluates to zero, it suggests absence with a small probability of error. Repeating the process reduces the error probability exponentially.

Pseudocode

```
Input: Undirected graph G with n vertices

Repeat r = O(n^2) times:
    1. Randomly split vertices into V1 and V2 (equal size)
    2. For m = 0 to floor(n/4):   # try up to n/4 internal edges in V1
        a. Build small graph D on V1
        b. Use labels from V2 and m extra dummy labels
        c. Construct function f that encodes paths and edges
        d. Compute determinant sum Λ(D, labels, f) over a finite field
        e. If Λ ≠ 0, return YES (Hamiltonian cycle found)

Return NO
```

Time Complexity Analysis

The expected runtime of the algorithm is $O(1.657^n)$. This is a Monte Carlo algorithm, meaning it may return false negatives but never false positives. The base of 1.657 is achieved by optimizing the tradeoff between the number of labels (which determine the subset size and hence the exponential blowup from subset enumeration) and the structural properties of Hamiltonian cycles under random partitioning of the vertex set. The algorithm repeats over several random vertex partitions to increase the chance that a Hamiltonian cycle, if one exists, is preserved in a favorable configuration. In bipartite graphs or graphs with known large independent sets, this structure can be exploited to reduce the label set size deterministically. When the independent set is of size i, the time complexity improves to $O(2^{n-i})$. For bipartite graphs, the balanced structure ensures that the labeling aligns naturally with Hamiltonian cycle alternation between the partitions, reducing the base to approximately 1.414, which matches the theoretical lower bound imposed by the structure's entropy.

Boolean satisfiability (SAT) Approach

To solve the Hamiltonian Path Problem (HPP) using Boolean satisfiability (SAT), we encode the problem into a SAT instance. This approach leverages the capabilities of modern SAT solvers to handle complex combinatorial problems.

Given a directed graph G = (V, E), the goal is to determine whether there exists a Hamiltonian path—a path that visits each vertex exactly once. Model the problem by assigning positions to vertices in the path. Each vertex must appear in exactly one position, and each position must be occupied by exactly one vertex. For each vertex v in V and each position p in {1, 2, ..., n}, define a Boolean variable x[v][p] that is true if vertex v is at position p in the path.

Vertex Position Constraint:

For each vertex v in V:

$$x[v][1] + x[v][2] + \ldots + x[v][n] = 1$$

Position Occupancy Constraint:

For each position p in {1, ..., n}:

$$x[v_1][p] + x[v_2][p] + \ldots + x[v_n][p] = 1$$

(where v_1, v_2, ..., v_n are all vertices in V)

Adjacency Constraint:

For each position p in {1, ..., n−1} and for each pair of vertices (u, v) such that (u, v) is not an edge in E:

$$\neg x[u][p] \lor \neg x[v][p+1]$$

This encoding ensures that if x[u][p] and x[v][p+1] are both true, then (u, v) must be an edge in the graph. If not, the clause prevents that assignment.

Pseudocode

```
function EncodeHamiltonianPathToSAT(Graph G = (V, E)):
    n := number of vertices in G
    // Create boolean variables x[i][j] for vertex i at position j
    vars := matrix of boolean variables x[i][j] for i in 1..n, j in 1..n

    clauses := []

    // Constraint 1: Each position in the path is occupied by exactly one vertex
    for j in 1 to n:
        clause := []
        for i in 1 to n:
            clause.append(x[i][j])
        clauses.append(OR of all x[i][j])  // At least one vertex at position j
        for i1 in 1 to n:
            for i2 in i1+1 to n:
                clauses.append(NOT x[i1][j] OR NOT x[i2][j])  // At most one

    // Constraint 2: Each vertex appears in exactly one position
    for i in 1 to n:
        clause := []
        for j in 1 to n:
            clause.append(x[i][j])
        clauses.append(OR of all x[i][j])  // Vertex i appears somewhere
        for j1 in 1 to n:
            for j2 in j1+1 to n:
                clauses.append(NOT x[i][j1] OR NOT x[i][j2])  // At most one

    // Constraint 3: Enforce adjacency
    for j in 1 to n-1:
        for i in 1 to n:
            for k in 1 to n:
                if (i, k) not in E:
                    clauses.append(NOT x[i][j] OR NOT x[k][j+1])
    return clauses
```

Time Complexity Analysis

The SAT encoding for the Hamiltonian Path problem introduces O(n^2) Boolean variables, one for each possible assignment of a vertex to a position, and O(n^3) clauses to enforce the constraints that each vertex occupies exactly one position, each position is uniquely occupied, and that consecutive positions correspond to valid edges in the graph. While the encoding process itself runs in polynomial time, solving the resulting SAT instance dominates the complexity. Because SAT is NP-complete, the worst-case time complexity of solving the encoded instance is exponential. Therefore, the overall time complexity of this approach is

$$O(2^{\mathrm{n}} \cdot poly(n))$$

References

[1] *Hamiltonian Circuit* - ScienceDirect Topic Overview.

[2] *On Hamiltonian Cycles and Hamiltonian Paths*, ResearchGate.

[3] *The Icosian Game and Hamiltonian Circuits* – That's Maths.

[4] Rubin, Frank. *A Search Procedure for Hamilton Paths and Circuits.*

[5] *A Heuristic Method for the Determination of a Hamiltonian Circuit in a Graph*, Proceedings of the Edinburgh Mathematical Society.

[6] *Held–Karp Algorithm* — Overview of the dynamic programming approach for TSP and related problems.

[7] Zhou, Neng-Fa. *Encoding the Hamiltonian Cycle Problem into SAT Based on Vertex Elimination.*

[8] Shah, S. and Kothari, A. *Solving Hamiltonian Cycle Problem using SAT Solvers.* IEEE, 2010.

Research Task 2: Partition Problem

Definition

The **Partition Problem** is a fundamental challenge in the fields of computer science and combinatorial optimization. It involves determining whether a given finite set of positive integers can be divided into two subsets such that the sum of the elements in both subsets is exactly equal. In formal terms, given a set **S={s1,s2,...,sn},**the goal is to identify whether there exists a subset **S_1 ⊆ S** such that the sum of the elements in **S_1** is equal to the sum of the elements in the complement of **S_1** within the original set, that is, **sum(S_1)=sum(S− S_1)**, or equivalently, **sum(S_1)=total_sum/2**.This problem is known to be NP-complete, meaning that there is no known algorithm capable of solving all instances of the problem efficiently in polynomial time. However, it can be solved in pseudo-polynomial time using dynamic programming techniques, where the runtime depends on the total sum of the elements rather than just their count. This approach makes the problem tractable for inputs of moderate size. For instance, given the set **{1, 5, 11, 5},** which has a total sum of **22**, it is possible to divide it into two subsets **{11}** and **{1, 5, 5},** each summing to **11**. In contrast, a set like **{1, 2, 3, 5}** has a total sum of **11**, which is an odd number and therefore **cannot** be split into two equal integer parts, making a valid partition impossible. The Partition Problem is a special case of the Subset Sum Problem and is closely related to other optimization problems such as the Knapsack Problem and Multi-way Partitioning. It also has practical applications in various domains, including load balancing, resource allocation, scheduling, and cryptography.

Assumptions

- **All Elements Are Positive Integers**
  The standard Partition Problem assumes that all numbers in the input set are **positive integers**. Negative numbers or zeros introduce additional complexity and may lead to different problem variants.
- **Finite Set Size**
  The input set must be **finite**. Infinite sets or streams are not considered in this problem.
- **Even Total Sum**
  If the **total sum of all elements is odd**, the partition is **immediately impossible**. This is a basic check that can be done before applying any algorithm.
- **Subset Existence (Binary Decision)**
  The problem is framed as a **decision problem**: does a valid partition exist? It does not require finding **all** such partitions or the most optimal one (though those are valid extensions).

- **Subset Must Be Disjoint and Complete**
  The two subsets must be **disjoint** (no overlapping elements) and together must include **all elements** of the original set — i.e., a **partition**, not just any two equal-sum subsets.
- **Elements Can Only Be Used Once**
  The same element cannot be reused in both subsets. Each element is **used exactly once**, assigned to one of the two subsets.
- **Standard Version Is Unweighted**
  The elements are not associated with external weights or values — the numbers themselves represent the "weight" or "cost" being balanced.

Brute Force (Exhaustive Search)

The Brute Force (Exhaustive Search) approach to solving the Partition Problem systematically checks all possible ways to partition a set of integers into two subsets and verifies if their sums are equal. For a set of n elements, there are 2^n possible subsets, as each element can either be in the first subset or the second. The algorithm generates every subset and computes the sums of both subsets, comparing them to see if they are equal. If a partition is found with equal sums, the algorithm immediately returns True, indicating that the set can be divided into two subsets with equal sums. If no valid partition exists, it returns False. Although the brute force approach guarantees finding a solution if one exists, its time complexity is O(2^n * n), making it impractical for larger sets due to its exponential growth. This method is useful for small input sizes, quick prototyping, or situations where every possible partition needs to be checked for completeness. However, it is not efficient for large sets because of its exhaustive nature.

Pseudocode

```
Function CanPartition(S: array of positive integers) -> boolean:
    total_sum ← sum of all elements in S
    if total_sum is odd:
        return False  // Cannot partition an odd total

    target_sum ← total_sum / 2
    n ← length of S

    // Generate all possible subsets using binary representation
    for i from 0 to (2^n) - 1:
        subset_sum ← 0
        for j from 0 to n - 1:
            if (i >> j) & 1 == 1:  // If j-th bit in i is set
                subset_sum ← subset_sum + S[j]
        if subset_sum == target_sum:
            return True  // Found a valid partition

    return False  // No valid partition found
```

Time Complexity Analysis

The brute force approach has the following time complexity:

- There are 2^n possible subsets for a set with n elements.
- For each subset, we compute the sum of its elements, which takes $O(n)$ time.

Thus, the total time complexity is:

$$O(2^n \cdot n)$$

Where 2^n comes from generating all subsets, and n comes from summing the elements of each subset. This exponential time complexity makes brute force impractical for large values of n.

Dynamic Programming (DP)

The **Dynamic Programming (DP)** approach to solving the Partition Problem focuses on efficiently determining whether a subset of a given set of integers sums to half of the total sum of the set. The problem is first checked for feasibility: if the total sum of the elements is odd, the set cannot be partitioned into two equal subsets, and the algorithm returns False. If the total sum is even, the algorithm attempts to find a subset with a sum equal to half of the total. It uses a boolean array dp[] where each index j represents whether a subset exists that sums to j. The array is initialized with dp[0] = True, since a sum of 0 is always achievable. For each number in the set, the algorithm iterates backward through the dp array, updating each entry to reflect whether including that number allows achieving the current sum. If at the end dp[total_sum / 2] is True, it means a valid partition exists, and the algorithm returns True. Otherwise, it returns False. This approach has a time complexity of **O(n * total_sum)** and a space complexity of **O(total_sum)**, where n is the number of elements. While it is not polynomial in the traditional sense, it is significantly more efficient than brute force for moderate-sized inputs and is especially suitable when the sum of elements is not excessively large.

Pseudocode

```
Function canPartition(S: array of positive integers) -> boolean:
    total_sum ← sum of all elements in S
    if total_sum is odd:
        return False

    target ← total_sum / 2
    dp ← array of boolean values of size (target + 1), initialized to False
    dp[0] ← True

    for num in S:
        for j from target down to num:
            dp[j] ← dp[j] or dp[j - num]

    return dp[target]
```

Time Complexity Analysis

The algorithm has two nested loops:

the outer loop runs for n elements, and the inner loop runs for **target = total_sum / 2** in the worst case.

Therefore, time complexity is:

$$O(n \cdot total\_sum/2) = O(n \cdot total\_sum)$$

Backtracking approach

The Backtracking approach to solving the Partition Problem is a refined form of brute force that incrementally builds subsets and abandons paths that cannot lead to a solution. The goal is to determine whether a given set of positive integers can be partitioned into two subsets with equal sums. First, the total sum of the set is calculated, and if it is odd, the algorithm immediately returns False. If the total is even, the problem reduces to finding a subset that sums to half of the total. Using backtracking, the algorithm explores combinations of elements, deciding at each step whether to include the current element in the subset. If the partial sum exceeds the target (half the total sum), that path is abandoned (pruned), reducing unnecessary computation. This pruning makes backtracking more efficient than pure brute force by avoiding exploration of invalid branches early. If a valid subset is found that sums to the target, the algorithm returns True. Otherwise, it continues exploring. The worst-case time complexity is still O(2^n), similar to brute force, but in practice, it performs better due to pruning. The backtracking method is suitable for small to medium input sizes and provides a balance between completeness and performance by intelligently narrowing down the search space.

Pseudocode

```
Function canPartitionBacktrack(nums: array of positive integers) -> boolean:
    total_sum ← sum(nums)
    if total_sum is odd:
        return False

    target ← total_sum / 2

    Function backtrack(index: int, current_sum: int) -> boolean:
        if current_sum == target:
            return True
        if current_sum > target or index == length of nums:
            return False

        // Choose to include the current number
        if backtrack(index + 1, current_sum + nums[index]):
            return True

        // Choose to exclude the current number
        return backtrack(index + 1, current_sum)

    return backtrack(0, 0)
```

Time Complexity Analysis

In the worst case, the algorithm explores every subset of the input array.

This results in time complexity of

$$O(2^{n})$$

where n is the number of elements.

However, pruning (i.e., stopping recursion when the current sum exceeds the target) significantly reduces the number of recursive calls in many practical cases.

The Greedy Algorithm

The Greedy Algorithm is a heuristic approach that builds a solution step-by-step by making locally optimal choices at each step, with the hope of finding a global optimum. In the context of the Partition Problem, a greedy strategy typically attempts to assign each number in the set to the subset with the currently smaller sum. The goal is to keep the sums of the two subsets as balanced as possible during the construction.

For example, after sorting the numbers in descending order, the algorithm assigns each number to the subset with the smaller current sum. This method is fast and simple, and in many cases produces a reasonably balanced partition. However, it does not guarantee a correct answer in all cases. This is because greedy algorithms do not consider future consequences and may miss optimal combinations that a backtracking or dynamic programming approach would find.

While it can be useful for approximations or when an exact solution is not required (as in load balancing or scheduling), the greedy algorithm cannot be relied upon for solving the Partition Problem exactly due to its failure on certain inputs.

Pseudocode

```
Function canPartitionGreedy(nums: array of positive integers) -> boolean:
    total_sum ← sum(nums)
    if total_sum is odd:
        return False

    Sort nums in descending order

    subset1 ← empty list, sum1 ← 0
    subset2 ← empty list, sum2 ← 0

    for num in nums:
        if sum1 <= sum2:
            subset1.add(num)
            sum1 += num
        else:
            subset2.add(num)
            sum2 += num

    return sum1 == sum2
```

Time Complexity Analysis

The greedy approach has the following time complexity:

- First, we calculate the total sum of the set, which takes **O(n)** time.
- Then, we sort the set of n elements in descending order, which takes **O(n log n)** time.
- After sorting, we iterate through the set once, making a constant-time comparison and assignment for each element, which takes **O(n)** time.

Thus, the total time complexity is:

$$O(n\ logn)$$

Where $n \log n$ comes from the sorting step, and n comes from the iteration over the elements. This makes the greedy approach much more efficient than brute force, especially for large inputs.

Approximation Algorithm

The Approximation Algorithm for the Partition Problem aims to find a near-optimal solution efficiently, without exploring all possible partitions. It does not guarantee an exact answer, but it typically produces a good approximation, especially when a perfectly balanced partition isn't strictly required.

The most common approximation method uses a greedy strategy, where the elements are sorted in descending order and each is assigned to the subset with the smaller current total sum. The idea is to balance the two subsets by always placing the next largest number in the "lighter" subset.

This method is simple and efficient, making it useful for large datasets where exact solutions (like brute force or dynamic programming) are computationally too expensive.

Pseudocode

```
function ApproximatePartition(set S):
    sort S in descending order
    initialize subset1 and subset2 as empty
    sum1 = 0, sum2 = 0

    for each element in S:
        if sum1 <= sum2:
            add element to subset1
            sum1 += element
        else:
            add element to subset2
            sum2 += element

    return subset1, subset2
```

Time Complexity Analysis

The approximation approach has the following time complexity:

- First, the total sum of the set is calculated, which takes **O(n)** time.
- Then, the input set is sorted in descending order, which takes **O(n log n)** time.
- After sorting, each element is placed into the subset with the smaller current sum in a single pass through the array, which takes **O(n)** time.

Thus, the total time complexity is:

$$O(n\ logn)$$

Where $n\ logn$ comes from the sorting step, and n comes from assigning each element to a subset. Although this approach does not provide an exact solution, it is highly efficient and works well in many real-world scenarios.

Integer Programming (IP)

The **Integer Programming (IP)** approach transforms the Partition Problem into a **mathematical optimization problem** with linear constraints and integer variables. The goal is to decide whether the input set can be divided into two subsets with equal sums by assigning each element to one of two groups, such that the total difference between the groups is minimized.

Each element is associated with a **binary decision variable**, which indicates which subset it belongs to (e.g., 0 for one subset, 1 for the other). The objective is to **minimize the absolute difference between the sums of the two subsets**, and the constraints enforce valid subset assignments.

This formulation allows the use of **Integer Linear Programming (ILP)** solvers (like CPLEX, Gurobi, or open-source solvers like CBC or GLPK) to find an optimal partition or determine that none exists.

Pseudocode

```
Input: A set S = {$s_1$, $s_2$, ..., $s_n$} of positive integers

Let total_sum = sum of all elements in S

Define binary variables:
    For each i from 1 to n:
        x[i] ∈ {0, 1}  // 1 means $s_i$ goes to Subset A, 0 means Subset B

Objective:
    Minimize |2 * sum(x[i] * s[i] for i = 1 to n) - total_sum|
Constraints:
    x[i] ∈ {0, 1}  for all i = 1 to n

Output:
    If the objective equals 0, return True (partition exists)
    Else, return False
```

Time Complexity Analysis

The integer programming approach has the following time complexity characteristics:

- **Integer Linear Programming is NP-hard**, meaning there is no known polynomial-time algorithm that can solve all cases efficiently.
- **In the worst case**, solving an integer program with **n binary variables** requires **exponential time**:

$$O(2^n)$$

However, modern **ILP solvers** use advanced techniques like **branch and bound**, **cutting planes**, and **preprocessing**, which perform well in practice on moderately sized problems.

Thus, the worst-case time complexity is:

$$O(2^n)$$

But with efficient solvers and heuristics, practical performance is often much better than brute force or backtracking.

The Branch and Bound

The Branch and Bound algorithm is an optimized version of backtracking designed to reduce the number of recursive calls by eliminating suboptimal or impossible solutions early. It explores the search tree systematically, but prunes branches that cannot lead to a better or valid solution based on calculated bounds.

In the context of the Partition Problem, the goal is to assign elements to two subsets such that their sums are as close as possible. The algorithm maintains a current difference between the two subset sums and attempts to minimize this difference.

At each decision point (node), the algorithm tries including the current element in either subset and continues recursively. If a current partial solution leads to a subset sum difference larger than a previously found best solution (or can't possibly improve it), the branch is bounded (pruned) and not explored further.

This technique avoids checking every possible partition and focuses only on promising paths, reducing unnecessary computation compared to brute force or backtracking.

Pseudocode

```
function BranchAndBoundPartition(set S):
    sort S in descending order
    best_diff = ∞
    best_partition = null

    function recurse(index, subset1_sum, subset2_sum):
        if index == length of S:
            diff = abs(subset1_sum - subset2_sum)
            if diff < best_diff:
                best_diff = diff
                update best_partition
            return

        if abs(subset1_sum - subset2_sum) >= best_diff:
            return  // prune this branch

        // Include current element in subset1
        recurse(index + 1, subset1_sum + S[index], subset2_sum)

        // Include current element in subset2
        recurse(index + 1, subset1_sum, subset2_sum + S[index])

    recurse(0, 0, 0)
    return best_partition
```

Time Complexity Analysis

The branch and bound approach has the following time complexity:

- In the **worst case**, it explores the entire subset tree similar to brute force, which has 2^{n} nodes.
- At each level, the algorithm does constant-time calculations for pruning and tracking differences, so each recursive call takes **O(1)** time apart from recursive overhead.
- Therefore, the **worst-case time complexity is:**

$$O(2^{\mathrm{n}})$$

However, **in practice**, due to effective **pruning**, many branches are skipped, especially when larger elements are processed first (hence the importance of sorting). This makes the **average case performance significantly better** than brute force or backtracking, though still exponential in the worst case.

References

[1] *Reducibility among Combinatorial Problems | SpringerLink*

[2] *Berkeley Algorithms – Chapter 7*

[3] *The Algorithm Design Manual | SpringerLink*

[4] *An Algorithm for Dualization in Products of Lattices and Its Applications | SpringerLink*

[5] *Partition a Set into Two Subsets of Equal Sum | GeeksforGeeks*

[6] *Partition problem - Wikipedia*

Research Task 3: Graph Coloring Problem

Definition

We are given an undirected graph and asked to assign colors to all its vertices using exactly **M colors**, such that **no two adjacent vertices share the same color**. If it is possible to color the graph according to this rule, we must output the coloring result. Otherwise, the output should be: No solution possible. The **minimum number of colors** needed to color the graph without any two connected vertices having the same color is called the **chromatic number** of the graph.

Consider the following illustration of a graph:

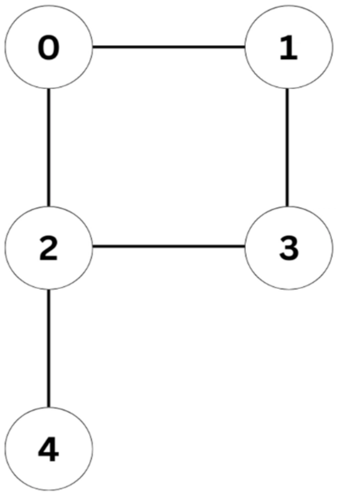


Figure 18: Sample graph

In this example, each circle represents a node, and the lines connecting them represent edges. Now, let's explore different ways to color this graph:

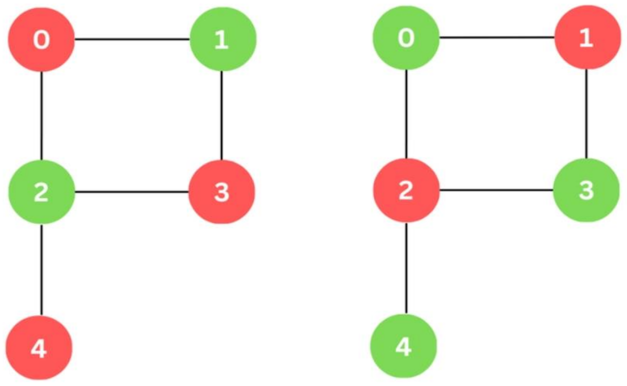


Figure 19: Different coloring ways

Based on the two possibilities of coloring a graph, it's evident that we require a minimum of two colors to ensure that no two adjacent nodes share the same color, thus satisfying the constraint. Through the process of graph coloring, we'll explore how this seemingly simple task can provide an easy solution to a complex problem.

Assumptions

1. **Each vertex must be assigned one of the M colors**
2. **For every edge (u, v), the colors of u and v must be different**
3. Graph may be **sparse or dense**, but must be **undirected** and **simple** (no loops or multi-edges)

Backtracking Algorithm

The backtracking algorithm makes the process efficient by avoiding many bad decisions made in naïve approaches. In this approach, we color a single vertex and then move to its adjacent (connected) vertex to color it with different color. After coloring, we again move to another adjacent vertex that is uncolored and repeat the process until all vertices of the given graph are colored. In case, we find a vertex that has all adjacent vertices colored and no color is left to make it color different, we backtrack and change the color of the last-colored vertices and again proceed further. If by backtracking, we come back to the same vertex from where we started and all colors were tried on it, then it means the given number of colors (i.e. 'm') is insufficient to color the given graph and we require more colors (i.e. a bigger chromatic number).

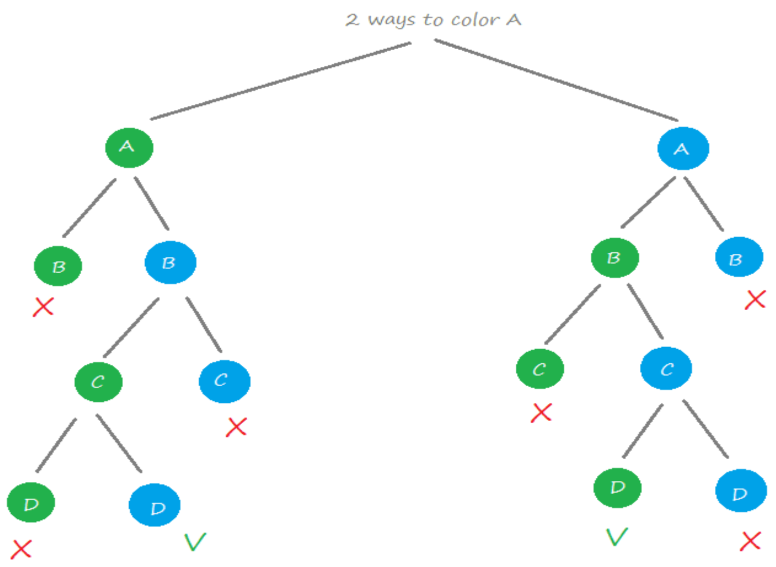


Figure 20: Backtracking tree

Steps To color graph using the Backtracking Algorithm:

1. Different colors:

    1. Confirm whether it is valid to color the current vertex with the current color (by checking whether any of its adjacent vertices are colored with the same color).

    2. If yes then color it and otherwise try a different color.

    3. Check if all vertices are colored or not.

    4. If not then move to the next adjacent uncolored vertex.

2. If no other color is available then backtrack (i.e. un-color last colored vertex).

*Here **backtracking means to stop further recursive calls** on adjacent vertices by returning false. In this algorithm Step-1.2 (Continue) and Step-2 (backtracking) is causing the program to try different color option.*

Continue *– try a different color for current vertex.*
Backtrack *– try a different color for last colored vertex.*

Pseudocode with Explanation

```
Function Main:
    Create an array of Vertex objects for the graph
    Connect vertices using addNeighbor (bidirectional)
    Define an array of available colors
    Create a Coloring object with the colors and number of vertices
    Call setColors starting from the first vertex
    If setColors returns true:
        Print the color of each vertex
    Else:
        Print "No Solution"

Class Vertex:
    Properties:
        name
        list of adjacent vertices
        colored (boolean)
        color (string)
    Constructor(name):
        Initialize name, empty adjacency list, colored=false, color=""
    Method addNeighbor(vertex):
        Add vertex to this.adjacentVertices
        Add this to vertex.adjacentVertices (bidirectional)

Class Coloring:
    Properties:
        colors[]
        numberOfVertices
        colorCount = 0
    Constructor(colors[], N):
        Store the color array and number of vertices
    Method setColors(vertex):
        For each color in colors:
            If it is valid to assign (canColorWith returns true):
                Assign color to vertex
                Mark as colored
                Increment colorCount
                If all vertices are colored:
                    Return true
                For each neighbor of vertex:
                    If neighbor is not colored:
                        Recursively call setColors(neighbor)
                        If successful:
                            Return true
        Backtrack:
            Unmark vertex as colored
            Remove its color
        Return false

    Method canColorWith(colorIndex, vertex):
        For each adjacent vertex:
            If that vertex is already colored with this color:
                Return false
        Return true
```

Best-Case

1. **Graph is Sparse**:
    - Few edges, so fewer adjacency constraints → easier to assign valid colors.
2. **Colors Are Sufficient and Well-Suited**:
    - For example, if you provide 3 colors to a tree or bipartite graph (which only needs 2), then the algorithm finds a solution early.
3. **Vertex Traversal Order is Favorable**:
    - If you start coloring from a vertex with **fewer neighbors** and progress in an order that minimizes conflicts, the algorithm will require little or no backtracking.

Average Case

This is the **typical case**:

- The algorithm **sometimes backtracks** when an assigned color leads to a conflict later.
- Most vertices are colored successfully on the first or second attempt.
- A **moderate number of recursive calls and color reassignments** are required.
- Occurs when:
    - The graph is **moderately connected**.
    - The number of colors is **close to the chromatic number**.

Worst Case

This is the **most expensive case**:

- The algorithm tries **all possible color combinations**.

- **Heavy backtracking** is needed for many vertices.
- Happens when:
    - The number of colors MMM is **too small** to produce a valid coloring.
    - The graph is **dense** or **complete** (many edges).
    - Vertices are colored in a **bad order** (conflicts occur early).

<u>Java Implementation</u>

```
import java.util.ArrayList;
import java.util.List;

class Vertex {
  String name;
  List<Vertex> adjacentVertices;
  boolean colored;
  String color;

  public Vertex(String name) {
    this.name = name;
    this.adjacentVertices = new ArrayList<>();
    this.colored =false;
    this.color = "";
  }

  //connect two verticess bi-directional
  public void addNeighbor(Vertex vertex){
    this.adjacentVertices.add(vertex);
    vertex.adjacentVertices.add(this);
  }
}

class Coloring {
  String colors[];
  int colorCount;
  int numberOfVertices;

  public Coloring(String[] colors, int N) {
    this.colors = colors;
    this.numberOfVertices = N;
  }

  public boolean setColors(Vertex vertex){
    //Step: 1
    for(int colorIndex=0; colorIndex<colors.length; colorIndex++){
      //Step-1.1: checking validity
      if(!canColorWith(colorIndex, vertex))
        continue;

      //Step-1.2: continue coloring
      vertex.color=colors[colorIndex];
      vertex.colored=true;
      colorCount++;

      //Step-1.3: check whether all vertices colored?
      if(colorCount== numberOfVertices) //base case
        return true;

      //Step-1.4: next uncolored vertex
      for(Vertex nbrvertex: vertex.adjacentVertices){
        if (!nbrvertex.colored){
          if(setColors(nbrvertex))
            return true;
          }
      }

    }

    //Step-4: backtrack
    vertex.colored = false;
    vertex.color = "";
    return false;
  }

  //Function to check whether it is valid to color with color[colorIndex]
  boolean canColorWith(int colorIndex, Vertex vertex) {
    for(Vertex nbrvertex: vertex.adjacentVertices){
      if(nbrvertex.colored && nbrvertex.color.equals(colors[colorIndex]))
        return  false;
    }
    return true;
  }
}

public class Main{
  public static void main(String args[]){
    //define vertices
    Vertex vertices[]= {new Vertex("A"), new Vertex("B"), new Vertex("C"), new Vertex("D")};

    //join verices
    vertices[0].addNeighbor(vertices[1]);
    vertices[1].addNeighbor(vertices[2]);
    vertices[2].addNeighbor(vertices[3]);
    vertices[0].addNeighbor(vertices[3]);

    //define colors
    String colors[] = {"Green","Blue"};

    //create coloring object
    Coloring coloring = new Coloring(colors, vertices.length);

    //start coloring with vertex-A
    boolean hasSolution = coloring.setColors(vertices[0]);

    //check if coloring was successful
    if (!hasSolution)
        System.out.println("No Solution");
    else {
        for (Vertex vertex: vertices){
            System.out.println(vertex.name + " "+ vertex.color +"\n");
        }
    }
  }
}
```

Output

```
A Green

B Blue

C Green

D Blue
```

Time Complexity

Let's define:

V: The number of vertices in the graph.

M: The number of colors.

E: The number of edges.

Worst-Case Time Complexity

The backtracking algorithm has to explore multiple potential colorings of the graph. The worst-case time complexity is based on two key factors:

Choice of Colors for Each Vertex

For each vertex, there are M possible color choices.

So, if we have V vertices, there are M^V possible ways to assign colors, assuming each vertex could be colored independently.

Checking for Validity

For each coloring, we need to check whether it's valid (i.e., no two adjacent vertices share the same color). This check involves examining the adjacency list (or edges) for each vertex. In the worst case, each check involves examining the neighbors of the vertex, and for each of V vertices, this would take O(E) time.

Thus, validating a coloring could take up to O(E) time per coloring attempt.

So, in the worst case, the backtracking algorithm will explore M^V possible color assignments, and for each assignment, it will take O(E) time to check if the coloring is valid.

Therefore, the worst-case time complexity is:

$$\boldsymbol{O(M^V \cdot E)}$$

This is because:

We have at most M^V possible ways to color the graph (each vertex can be assigned one of M colors).

For each coloring configuration, we must check the edges, which requires O(E) time.

Space Complexity

The space complexity of this approach primarily comes from:

Storing the graph, which requires O(V + E) space for the adjacency list representation.

The recursion stack, which can go up to depth V (one recursion call per vertex).

Thus, the space complexity is:

$$\boldsymbol{O(V+E)}$$

Greedy Algorithm

Brooks' theorem states that a connected graph can be colored with only x colors, where x is the maximum degree of any vertex in the graph except for complete graphs and graphs containing an odd length cycle, which requires x+1 colors.

Greedy coloring *considers the vertices of the graph in sequence and assigns each vertex its first available color*, i.e., vertices are considered in a specific order v_1, v_2, … v_n, and v_i and assigned the smallest available color which is not used by any of v_i's neighbors.

Greedy coloring doesn't always use the minimum number of colors possible to color a graph. For a graph of maximum degree x, greedy coloring will use at most x+1 color. Greedy coloring can be arbitrarily bad; for example, the following crown graph (a complete bipartite graph), having n vertices, can be 2–colored but greedy coloring resulted in n/2 colors.

Steps To color graph using the Greedy Algorithm:

- Color first vertex with first color.
- Do following for remaining **V-1** vertices.
    - Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to **v**, assign a new color to it.

Pseudocode with Explanation

```
Class Greedy:
    Variables:
        V: Integer (Number of vertices)
        adj: Array of Linked Lists (Adjacency list for the graph)

    Constructor Greedy(v):
        Set V = v
        Initialize adj as an array of V linked lists
        For i from 0 to V - 1:
            adj[i] = empty linked list

    Method addEdge(v, w):
        Add w to adj[v]
        Add v to adj[w]  // Because the graph is undirected

    Method greedyColoring():
        result = Array of size V, initialized with -1 (all vertices uncolored)
        result[0] = 0  // Assign first color to first vertex

        available = Boolean array of size V, initialized to true (all colors available)

        For u from 1 to V - 1:
            For each adjacent vertex i of u:
                If result[i] != -1:
                    available[result[i]] = false  // Mark the color as unavailable

            For cr from 0 to V - 1:
                If available[cr] is true:
                    Break (found available color cr)

            Assign color cr to result[u]
            Reset all values in available to true for next iteration

        For u from 0 to V - 1:
            Print "Vertex u ---> Color result[u]"

Main Method:
    Create object g1 with 5 vertices
    Add edges to g1
    Print "Coloring of graph 1"
    Call g1.greedyColoring()

    Create object g2 with 5 vertices
    Add edges to g2
    Print "Coloring of graph 2"
    Call g2.greedyColoring()
```

Best case

- The vertices are processed in an order such that each vertex has **minimal color conflict** with already-colored neighbors.
- Example: If the graph is a **tree** (acyclic connected graph), greedy coloring will use only **2 colors**.

Average case

- Vertices are processed in **random or natural order**, not optimized.
- Moderate conflicts among neighbors; coloring is not optimal but not poor either.

Worst case

- vertices are processed in an order that **maximizes color conflict**.
- A classic example is a **complete graph** KnK_nKn, where every vertex is connected to every other vertex:
- It **must** use n colors because all vertices are adjacent.
- Another bad case is when a **path graph** is processed in reverse zig-zag or when **high-degree vertices** are processed late.

Java Implementation

```
// A Java program to implement greedy algorithm for graph coloring
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
class Graph
{
          private int V; // No. of vertices
          private LinkedList<Integer> adj[]; //Adjacency List

          //Constructor
          Graph(int v)
          {
                    V = v;
                    adj = new LinkedList[v];
                    for (int i=0; i<v; ++i)
                              adj[i] = new LinkedList();
          }

          //Function to add an edge into the graph
          void addEdge(int v,int w)
          {
                    adj[v].add(w);
                    adj[w].add(v); //Graph is undirected
          }

          // Assigns colors (starting from 0) to all vertices and
          // prints the assignment of colors
          void greedyColoring()
          {
                    int result[] = new int[V];

                    // Initialize all vertices as unassigned
                    Arrays.fill(result, -1);

                    // Assign the first color to first vertex
                    result[0] = 0;

                    // A temporary array to store the available colors. False
                    // value of available[cr] would mean that the color cr is
                    // assigned to one of its adjacent vertices
                    boolean available[] = new boolean[V];

                    // Initially, all colors are available
                    Arrays.fill(available, true);

                    // Assign colors to remaining V-1 vertices
                    for (int u = 1; u < V; u++)
                    {
                              // Process all adjacent vertices and flag their colors
                              // as unavailable
                              Iterator<Integer> it = adj[u].iterator() ;
                              while (it.hasNext())
                              {
                                        int i = it.next();
                                        if (result[i] != -1)
                                                  available[result[i]] = false;
                              }

                              // Find the first available color
                              int cr;
                              for (cr = 0; cr < V; cr++){
                                        if (available[cr])
                                                  break;
                              }
```

```
                                result[u] = cr; // Assign the found color

                                // Reset the values back to true for the next iteration
                                Arrays.fill(available, true);
                        }

                        // print the result
                        for (int u = 0; u < V; u++)
                                System.out.println("Vertex " + u + " ---> Color "
                                                                    + result[u]);
                }

                // Driver method
                public static void main(String args[])
                {
                        Graph g1 = new Graph(5);
                        g1.addEdge(0, 1);
                        g1.addEdge(0, 2);
                        g1.addEdge(1, 2);
                        g1.addEdge(1, 3);
                        g1.addEdge(2, 3);
                        g1.addEdge(3, 4);
                        System.out.println("Coloring of graph 1");
                        g1.greedyColoring();

                        System.out.println();
                        Graph g2 = new Graph(5);
                        g2.addEdge(0, 1);
                        g2.addEdge(0, 2);
                        g2.addEdge(1, 2);
                        g2.addEdge(1, 4);
                        g2.addEdge(2, 4);
                        g2.addEdge(4, 3);
                        System.out.println("Coloring of graph 2 ");
                        g2.greedyColoring();
                }
}
// This code is contributed by Aakash Hasija
```

<u>Output</u>

```
Coloring of graph 1

                Vertex 0 ---> Color 0

                Vertex 1 ---> Color 1

                Vertex 2 ---> Color 2

                Vertex 3 ---> Color 0

                Vertex 4 ---> Color 1

Coloring of graph 2

                Vertex 0 ---> Color 0

                Vertex 1 ---> Color 1

                Vertex 2 ---> Color 2
```

Vertex 3 ---> Color 0

Vertex 4 ---> Color 3

Time Complexity

In the greedy approach to the graph coloring problem, the time complexity is $O(V^2+E)$ in the worst case.

Space Complexity

In the greedy approach to the graph coloring problem, we are not using any extra space. So, the space complexity is **O(1)**.

Comparison

1. Backtracking Algorithm

Pros:

- **Guaranteed solution:** Finds a solution if one exists.
- **Accurate chromatic number detection:** Can find the **minimum number of colors** required (i.e., the **chromatic number**) if implemented to try increasing values of M.
- **Flexible:** Works for arbitrary graph structures (sparse, dense, cycles, etc.).

Cons:

- **Very slow for large graphs** due to exponential time complexity.
- **Excessive backtracking** in dense graphs or with low M.
- **Inefficient for real-time or large input cases.**

Time Complexity:

- **Worst-case:** O(M^V · E)
  - M = number of colors
  - V = number of vertices
  - E = number of edges (used in validity checks)
- **Note:** Exponential due to all color permutations.

Space Complexity:

- O(V + E) for adjacency list and vertex properties.
- O(V) additional for recursion stack (one call per vertex).

2. Greedy Algorithm

Pros:

- **Fast and efficient:** Runs in linear time with respect to the number of vertices and edges.
- **Simple to implement.**
- **Good for sparse or tree-like graphs.**
- **Deterministic:** Always gives a result.

Cons:

- **Not optimal:** May use more colors than necessary.
- **Highly order-dependent:** Coloring may vary significantly based on the vertex traversal order.
- **Poor worst-case performance** (e.g., crown graphs, complete graphs).

Time Complexity:

- **Time:** O(V + E)
    - Each vertex and edge is visited once.
- **Worst-case colors used:** $\Delta + 1$, where Δ is the maximum degree of the graph.

Space Complexity:

- O(V + E) for the graph.
- O(V) for color tracking arrays.

Conclusion

The Graph Coloring Problem is a classic example of a constraint satisfaction problem with wide-ranging applications in scheduling, register allocation, and frequency assignment. It seeks to color the vertices of an undirected graph using a limited number of colors such that no two adjacent vertices share the same color. This seemingly simple problem encapsulates significant computational complexity, especially as the size and density of the graph increase.

Two primary approaches explored in this article—the **Backtracking Algorithm** and the **Greedy Algorithm**—offer contrasting strategies. The **backtracking method** is exhaustive but accurate, capable of finding an optimal solution if one exists. However, it suffers from exponential time complexity O(M^V * E) in the worst case and is impractical for large or dense graphs. In contrast, the **greedy approach** is efficient with linear time complexity O(V^2) (or O(V + E) with optimizations), but it often fails to find the optimal coloring and is highly sensitive to vertex ordering.

Choosing between these methods depends on the problem constraints:

- Use **backtracking** when accuracy is paramount and the graph is small or sparse.
- Use **greedy coloring** when performance is critical, and approximate solutions are acceptable.

Understanding these trade-offs equips developers and researchers to select or design more efficient algorithms tailored to specific real-world scenarios.

References

[1] Graph Coloring Problem: Cracking Complexity with Elegant Solutions - DEV Community

[2] Graph Coloring Algorithm using Backtracking – Pencil Programmer

[3] Graph Coloring Using Greedy Algorithm | GeeksforGeeks

[4] Graph Coloring Problem | Techie Delight

[5] Introduction to Algorithms (Cormen, Leiserson, Rivest, Stein)