**A blue and white logo

AI-generated content may be incorrect.**



**algorithms project**

**CSE245: Advances Algorithms and Complexity**



Seif Amjad Sobeih 23p0145

Abdelrahman Wael Maher 23p0155

Ahmed Hossam Abdallah 23p0109

Marwan Wagih Mohamed 23p0247

Ahmed Sadek Mubarak 23p0140

Kareem Ahmed Talaat 23p0225

Table of Contents

[Table of figures 4](#_Toc197894114)

[Table of tables 4](#_Toc197894115)

[Task 1: Tromino Tiling 5](#_Toc197894116)

[Detailed Assumptions 5](#_Toc197894117)

[Problem Description 5](#_Toc197894118)

[Detailed Solution 5](#_Toc197894119)

[Pseudo-code 5](#_Toc197894120)

[C++ Code 6](#_Toc197894121)

[Complexity Analysis 11](#_Toc197894122)

[Time complexity 11](#_Toc197894123)

[Space complexity 12](#_Toc197894124)

[Recurrence Relation for Brute Force: 12](#_Toc197894125)

[Time complexity 12](#_Toc197894126)

[Space Complexity 13](#_Toc197894127)

[Sample Output Description 14](#_Toc197894128)

[Conclusion 15](#_Toc197894129)

[Additional Notes 15](#_Toc197894130)

[Task 2: Knight’s Tour 15](#_Toc197894131)

[Detailed Assumptions 15](#_Toc197894132)

[Problem Description 17](#_Toc197894133)

[PESUDOCODE: 17](#_Toc197894134)

[THE C++ CODE 18](#_Toc197894135)

[COMPLEXITY ANALYSIS 23](#_Toc197894136)

[SAMPLE OUTPUT DESCRIPTION 25](#_Toc197894137)

[Conclusion 28](#_Toc197894138)

[Task 3: Tower of Hanoi 28](#_Toc197894139)

[Detailed Assumptions 28](#_Toc197894140)

[Problem Description 28](#_Toc197894141)

[Detailed Solution 29](#_Toc197894142)

[Optimization using Dynamic Programming 29](#_Toc197894143)

[Pseudocode for Divide and Conquer 29](#_Toc197894144)

[Pseudocode for Dynamic Programming 30](#_Toc197894145)

[Pseudocode for Iterative Method 31](#_Toc197894146)

[C++ code using Divide and Conquer 33](#_Toc197894147)

[C++ Code using Dynamic Programming 35](#_Toc197894148)

[COMPLEXITY ANALYSIS 37](#_Toc197894149)

[Sample Output Description 42](#_Toc197894150)

[Conclusion: 43](#_Toc197894151)

[Task 4: Knight Exchange Problem – BFS Iterative improvement Algorithm Solution 44](#_Toc197894152)

[1. Detailed Assumptions 44](#_Toc197894153)

[2. Problem Description 44](#_Toc197894154)

[3. Detailed Solution 44](#_Toc197894155)

[Pseudo-code: 44](#_Toc197894156)

[C++ Code: 44](#_Toc197894157)

[4. Complexity Analysis 53](#_Toc197894158)

[6. Sample Output 54](#_Toc197894159)

[7. Conclusion 54](#_Toc197894160)

[Task 5: Target shooting 54](#_Toc197894161)

[1. Detailed Assumptions 54](#_Toc197894162)

[2. Problem Description 55](#_Toc197894163)

[3. Detailed Solution 56](#_Toc197894164)

[1-High‑Level Strategy 56](#_Toc197894165)

[2-Pseudocode: for the generateShotPlan (the divide-and-conquer shot plan generator) 57](#_Toc197894166)

[3-cpp code 57](#_Toc197894167)

[4-Explanation 63](#_Toc197894168)

[4. Complexity analysis 65](#_Toc197894169)

[Time Complexity: 65](#_Toc197894170)

[5. A comparison between your algorithm and at least one other technique that can be used to solve the problem 66](#_Toc197894171)

[6. Sample output 69](#_Toc197894172)

[7. Conclusion 71](#_Toc197894173)

[Task 6: N x N Point Lattice 71](#_Toc197894174)

[Detailed Assumptions 71](#_Toc197894175)

[Problem Description 72](#_Toc197894176)

[Pseudocode 72](#_Toc197894177)

[C++ code 74](#_Toc197894178)

[Complexity Analysis 80](#_Toc197894179)

[Comparison with Another Technique 81](#_Toc197894180)

[Sample Output 83](#_Toc197894181)

[Conclusion 84](#_Toc197894182)

[Research task 1: Hamiltonian circuit problem 85](#_Toc197894183)

[Problem description 85](#_Toc197894184)

[Common algorithmic methods 86](#_Toc197894185)

[1.Backtracking 86](#_Toc197894186)

[2. Held-Karp algorithm (Dynamic programming) 87](#_Toc197894187)

[3. Genetic algorithm (GA) 88](#_Toc197894188)

[Research task 2: Partition problem 90](#_Toc197894189)

[Problem description 90](#_Toc197894190)

[Common algorithmic methods 91](#_Toc197894191)

[1.Brute force approach 91](#_Toc197894192)

[2.Dynamic programming approach 92](#_Toc197894193)

[Research task 3: Graph coloring problem 93](#_Toc197894194)

[Problem description 93](#_Toc197894195)

[Common algorithmic methods 93](#_Toc197894196)

[1.Backtracking approach 93](#_Toc197894197)

[2.Greedy approach 94](#_Toc197894198)

[References 95](#_Toc197894199)

# Table of figures

[Figure 1: Output of example 1 15](#_Toc197894200)

[Figure 2: Output of example 2 15](#_Toc197894201)

[Figure 3: Example 1 26](#_Toc197894202)

[Figure 4: Example 2 26](#_Toc197894203)

[Figure 5: Example 3 27](#_Toc197894204)

[Figure 6: number of moves 29](#_Toc197894205)

[Figure 7: Recursion 39](#_Toc197894206)

[Figure 8: D&C example 1 42](#_Toc197894207)

[Figure 9: D&C example 2 42](#_Toc197894208)

[Figure 10: DP example 1 43](#_Toc197894209)

[Figure 11: DP example 2 43](#_Toc197894210)

[Figure 12: Sample output of Task 54](#_Toc197894211)

[Figure 13: Pseudocode task 5 57](#_Toc197894212)

[Figure 14: sample output 1 70](#_Toc197894213)

[Figure 15: sample output 2 70](#_Toc197894214)

[Figure 16: Sample 1 83](#_Toc197894215)

[Figure 17: Sample 2 84](#_Toc197894216)

[Figure 18: Sample 3 84](#_Toc197894217)

# Table of tables

[Table 1: Comparison with other technique 13](#_Toc197894218)

[Table 2: Comparison of Methods 24](#_Toc197894219)

[Table 3. Comparison with Another Technique 40](#_Toc197894220)

[Table 4: Comparison With Another Technique 53](#_Toc197894221)

[Table 5:Comparison with another technique 69](#_Toc197894222)

[Table 6: Comparison 81](#_Toc197894223)

# Task 1: Tromino Tiling

## Detailed Assumptions

- The board size is always 2^n x 2^n, where n ≥ 2.  
- There is exactly one missing cell on the board.  
- Trominoes used are L-shaped and must not share a color with adjacent trominoes along edges.  
- Colors used are labeled 0,1,2  
- The input is received from the user, including the value of n and the coordinates of the missing cell.

## Problem Description

Given a 2^n × 2^n board with one missing cell, tile the board using L-shaped trominoes (each covering 3 cells) such that no two adjacent trominoes (sharing an edge) have the same color. The objective is to correctly tile the board using a divide-and-conquer approach and assign a valid color to each tromino from a palette of three, ensuring proper coloring constraints.

## Detailed Solution

The algorithm divides the board recursively into four quadrants.  
At each recursive step, it identifies which quadrant contains the missing square and fills the center of the board with a tromino covering the center of the three remaining quadrants.  
Each tromino is assigned a unique ID.  
A valid color is chosen for each tromino based on its neighbors using a graph-coloring-like technique.

## Pseudo-code

function tile(size, x, y, miss\_x, miss\_y, board\_size):  
 if size == 2:  
 fill remaining 3 cells with a tromino  
 assign color based on neighboring trominoes  
 return  
 divide board into 4 quadrants  
 place central tromino for 3 non-missing quadrants  
 recursively tile each quadrant

## C++ Code

#include <iostream>

#include <vector>

using namespace std;

const int N = 128;

int board[N][N];

int color[N \* N];

int tromino\_id = 1;

// Helper to check if a value exists in a vector

bool contains(const vector<int>& vec, int val) {

for (int v : vec)

if (v == val) return true;

return false;

}

vector<int> getAdjacentTrominoIDs(int x, int y, int n) {

vector<int> neighbors;

int dx[] = { -1, 1, 0, 0 };

int dy[] = { 0, 0, -1, 1 };

for (int d = 0; d < 4; d++) {

int nx = x + dx[d];

int ny = y + dy[d];

if (nx >= 0 && nx < n && ny >= 0 && ny < n) {

int id = board[nx][ny];

if (id != 0 && !contains(neighbors, id))

neighbors.push\_back(id);

}

}

return neighbors;

}

int getColor(const vector<int>& adjacent) {

bool used[4] = { false }; // index 1,2,3

for (int id : adjacent)

used[color[id]] = true;

for (int c = 1; c <= 3; c++) {

if (!used[c])

return c;

}

return 1;

}

void placeTromino(int x1, int y1, int x2, int y2, int x3, int y3, int board\_size) {

vector<int> neighbors;

vector<int> n1 = getAdjacentTrominoIDs(x1, y1, board\_size);

vector<int> n2 = getAdjacentTrominoIDs(x2, y2, board\_size);

vector<int> n3 = getAdjacentTrominoIDs(x3, y3, board\_size);

for (int id : n1)

if (!contains(neighbors, id)) neighbors.push\_back(id);

for (int id : n2)

if (!contains(neighbors, id)) neighbors.push\_back(id);

for (int id : n3)

if (!contains(neighbors, id)) neighbors.push\_back(id);

int c = getColor(neighbors);

board[x1][y1] = board[x2][y2] = board[x3][y3] = tromino\_id;

color[tromino\_id] = c;

tromino\_id++;

}

void tile(int size, int x, int y, int miss\_x, int miss\_y, int board\_size) {

if (size == 2) {

vector<pair<int, int>> cells;

for (int dx = 0; dx < 2; dx++)

for (int dy = 0; dy < 2; dy++)

if (x + dx != miss\_x || y + dy != miss\_y)

cells.push\_back({ x + dx, y + dy });

placeTromino(cells[0].first, cells[0].second,

cells[1].first, cells[1].second,

cells[2].first, cells[2].second,

board\_size);

return;

}

int mid = size / 2;

int cx = x + mid, cy = y + mid;

int qx = miss\_x < cx ? 0 : 1;

int qy = miss\_y < cy ? 0 : 1;

if (!(qx == 0 && qy == 0)) board[cx - 1][cy - 1] = -1;

if (!(qx == 0 && qy == 1)) board[cx - 1][cy] = -1;

if (!(qx == 1 && qy == 0)) board[cx][cy - 1] = -1;

if (!(qx == 1 && qy == 1)) board[cx][cy] = -1;

placeTromino(cx - 1, cy - 1, cx - 1, cy, cx, cy - 1, board\_size);

board[cx - 1][cy - 1] = board[cx - 1][cy] = board[cx][cy - 1] = board[cx][cy]; // clean overwrite

tile(mid, x, y, qx == 0 && qy == 0 ? miss\_x : cx - 1, qx == 0 && qy == 0 ? miss\_y : cy - 1, board\_size);

tile(mid, x, cy, qx == 0 && qy == 1 ? miss\_x : cx - 1, qx == 0 && qy == 1 ? miss\_y : cy, board\_size);

tile(mid, cx, y, qx == 1 && qy == 0 ? miss\_x : cx, qx == 1 && qy == 0 ? miss\_y : cy - 1, board\_size);

tile(mid, cx, cy, qx == 1 && qy == 1 ? miss\_x : cx, qx == 1 && qy == 1 ? miss\_y : cy, board\_size);

}

int main() {

int n;

cout << "Enter n (board size will be 2^n x 2^n): ";

cin >> n;

int size = 1 << n;

if (n < 2 || size > N) {

cout << "Invalid size\n";

return 1;

}

int miss\_x, miss\_y;

cout << "Enter missing cell coordinates (row col): ";

cin >> miss\_x >> miss\_y;

if (miss\_x < 0 || miss\_x >= size || miss\_y < 0 || miss\_y >= size) {

cout << "Invalid coordinates\n";

return 1;

}

board[miss\_x][miss\_y] = 0;

tile(size, 0, 0, miss\_x, miss\_y, size);

cout << "\nTiled Board (Colors):\n";

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

if (i == miss\_x && j == miss\_y) cout << "X\t";

else cout << color[board[i][j]] << "\t";

}

cout << endl;

}

return 0;

}

## Complexity Analysis

### Time complexity

problem breakdown:  
1. The board is of size 2^n × 2^n.  
2. At each step, the board is divided into 4 quadrants, which are recursively processed.  
3. The work done at each level involves:  
 Dividing the board (constant time).  
 Placing the central tromino (constant time).  
 Assigning colors to the trominoes (constant time).

Thus, the total time complexity at each level of recursion can be written as:  
T(n) = 4T(n/2) + O(1)  
Where:  
4T(n/2) corresponds to the recursive calls on the four quadrants, each of size

2^(n-1) × 2^(n-1).  
O(1) represents the constant time work done at each step (placing the tromino and coloring it).

According to the Master Theorem, the complexity for this recurrence is:   
T(n)=T(n^2)

Base Case:  
When the board is of size 2 × 2, the recursion stops, and the work is constant:  
T(2) = O(1)

### Space complexity

Since board is 2^n x 2^n the answer is:  
O(N^2) where N is 2^n

## Recurrence Relation for Brute Force:

### Time complexity

Let T(n) be the time required to tile a 2^n × 2^n board using brute force.

At each step, we need to try placing a tromino in every possible position on the board and recursively check all configurations for validity. In each recursion, we attempt all possible placements of the trominoes and backtrack if the placement is invalid.

* Each possible configuration corresponds to trying O(3^(n²)) different placements, since each tromino covers 3 cells, and the board has n² cells.
* For each configuration, we check if it’s valid (whether the tromino fits and if the coloring constraints are satisfied), which takes O(n²) time to check.

Thus, the recurrence relation for brute force becomes:

T(n) = O(3^(n^2)) + T(n-1)

Where:

* O(3^{n²}) represents the number of possible configurations of placing the trominoes on the board.
* T(n-1) represents the recursive call on the remaining part of the board.

After solving:  
T(n)= O(3^(n^2))

### Space Complexity

The space complexity is determined by the recursive call stack, as we are exploring all potential placements for each tromino recursively. The space complexity for brute force is O(n^2) due to:

1. Storing the board configuration.
2. The recursion depth (which could go up to n^2 in the worst case).

Thus, the space complexity is:

O(n^2)

Table 1: Comparison with other technique

|  |  |  |
| --- | --- | --- |
| Aspect | Brute force | Divide & conquer |
| Approach | |  | | --- | | Tries all tile placements (exhaustive search) |  |  | | --- | |  | | Recursively divides board into quadrants |
| Time complexity | Exponential: O(3^{n^2}) | Quadratic: O(n^2) |
| Correctness | Hard to ensure due to chaotic search | Guaranteed for all valid inputs |
| Scalability | Not feasible beyond tiny boards | Efficient even for large boards (e.g., 64×64) |
| Use Case | Puzzle exploration or academic examples | Real-world structured tiling problems |

**Brute force pseudocode:**

function bruteForceTiling(board, missingX, missingY):

if board is completely filled with valid trominoes:

if exactly one square is left empty and it's at (missingX, missingY):

save or print this valid solution

return

for each cell (i, j) in board:

if cell (i, j) is empty:

for each of the 4 possible L-tromino orientations:

if placing the tromino at (i, j) is valid:

place the tromino on the board

bruteForceTiling(updated board, missingX, missingY)

remove the tromino (backtrack)

break // only try to place one tile at a time

## Sample Output Description

- For n = 3 and missing cell at (3,4)  
- All adjacent trominoes are visually verifiable to not share the same color.

A number on a black background

AI-generated content may be incorrect.

Figure : Output of example 1

- For n = 4 and missing cell at (0,0)  
- All adjacent trominoes are visually verifiable to not share the same color.

A number on a black background

AI-generated content may be incorrect.

Figure : Output of example 2

## Conclusion

This approach efficiently solves the Tromino tiling problem with color constraints using divide-and-conquer and local color validation. It guarantees correctness and avoids expensive full-board graph coloring.

## Additional Notes

- The program uses set and vector only as additional libraries.  
- The board size is capped to 2^7 for demonstration and memory limits.

# Task 2: Knight’s Tour

## Detailed Assumptions

**Chessboard Representation:**

The chessboard is represented as an 8x8 grid. Each cell in the grid contains a numerical value indicating the order in which the knight visits the cell. A value of 0 represents an unvisited cell, and numbers from 1 to 64 represent the order of the knight’s visit.

**Knight's Starting Position:**

The algorithm attempts to find a knight's tour starting from every possible cell on the 8x8 grid. For each starting position, the algorithm checks if a complete knight's tour is possible, ending with a re-entrant tour (where the knight can land one move away from its starting position).

**Valid Moves:**

The knight follows traditional chess rules for movement, moving two squares in one direction (horizontal or vertical) and one square in the perpendicular direction. These possible moves are represented as pairs of coordinates, stored in the moves array.

**Move Validation:**

The check\_validation function checks if a knight's move is valid. This validation ensures that the move stays within the bounds of the 8x8 grid and that the destination cell has not been visited previously.

**Counting Unvisited Neighbors:**

The function available\_neighbors counts the number of neighboring cells that has not been visited for a given position on the chessboard. This helps the algorithm to evaluate which move is the best at each step.

**Selection of the Next Move:**

The select\_move function selects the next move based on Warnsdorff’s Rule, choosing the move that leads to the cell with the fewest unvisited neighbors. It’s done in the hopes of minimizing the chances of getting stuck in a dead-end situation.

**Knight's Tour Completion:**

The algorithm runs until the knight moves until all 64 squares are visited. If no valid moves remain or the tour isn’t closed, the search stops.

## Problem Description

The goal of this project is to determine whether a knight can complete a closed tour on an 8 × 8 chessboard, visiting every square exactly once and ending at a square one move away from the starting point. A closed knight’s tour means the knight’s last move leads it to a square adjacent to the starting position, completing a full cycle.

We aim to solve this problem using a greedy algorithm known as Warnsdorff’s Rule, which prioritizes moves to squares with the fewest onward moves. This method helps avoid dead ends and increases the likelihood of completing the tour. The knight can move according to standard chess rules: moving two squares in one direction and one square in the perpendicular direction.

Additionally, we explore the feasibility of finding a knight's tour on an n × n chessboard for values of n > 8. We will analyze how the knight’s tour behaves on boards of different sizes and discuss whether a closed tour is always possible for these larger boards. The solution also includes determining the minimum number of moves needed to complete the knight’s tour.

## PESUDOCODE:

Function knightTour()

For each (start\_x, start\_y):

Initialize board, set (x, y) = (start\_x, start\_y)

Mark (x, y) as visited

For i = 2 to 64:

next\_move = select\_move(board, x, y)

If invalid, break

(x, y) = next\_move

Mark (x, y) with i

If i == 64:

Print tour as completed or open

Break

End For

End Function

Function select\_move(board, x, y)

best\_move = (-1, -1)

min\_neighbors = 8

For each knight move:

next\_x, next\_y = calculate\_move(x, y)

If valid(next\_x, next\_y) and unvisited\_neighbors(board, next\_x, next\_y) < min\_neighbors:

best\_move, min\_neighbors = (next\_x, next\_y), unvisited\_neighbors(board, next\_x, next\_y)

Return best\_move

End Function

Function unvisited\_neighbors(board, x, y)

count = 0

For each knight move:

If valid(next\_x, next\_y), increment count

Return count

End Function

Function valid(x, y)

Return x, y within bounds

End Function

Function board\_display(board)

For row, column in board:

Print board[row][column]

End Function

## THE C++ CODE

#include <iostream>

#include <vector>

using namespace std;

const int n = 8;

const int moves[8][2] = {

{ -2, -1 }, { -2, 1 }, { -1, -2 }, { -1, 2 },

{ 1, -2 }, { 1, 2 }, { 2, -1 }, { 2, 1 }

};

struct Move {

int x;

int y;

};

bool check\_validation(int x, int y) {

return (x >= 0 && x < n && y >= 0 && y < n);

}

int available\_neighbours(const vector<vector<int>>& board, int x, int y) {

int count = 0;

for (auto& move : moves) {

int next\_x = x + move[0];

int next\_y = y + move[1];

if (check\_validation(next\_x, next\_y) && board[next\_y][next\_x] == 0) {

count++;

}

}

return count;

}

Move select\_move(const vector<vector<int>>& board, int x, int y) {

int min\_neighbours = 8;

Move next\_move = { -1, -1 };

for (auto& move : moves) {

int next\_x = x + move[0];

int next\_y = y + move[1];

if (check\_validation(next\_x, next\_y) && board[next\_y][next\_x] == 0) {

int unvisited\_neighbours = available\_neighbours(board, next\_x, next\_y);

if (unvisited\_neighbours < min\_neighbours) {

min\_neighbours = unvisited\_neighbours;

next\_move = { next\_x, next\_y };

}

}

}

return next\_move;

}

void board\_display(const vector<vector<int>>& board, int x, int y) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

if (i == y && j == x) {

cout << " KN ";

}

else {

if (board[i][j] < 10)

cout << " " << board[i][j] << " ";

else

cout << " " << board[i][j] << " ";

}

}

cout << "\n\n";

}

}

void knighttour(int start\_x, int start\_y) {

cout << "\nSTARTING KNIGHT'S TOUR FROM (" << start\_x << ", " << start\_y << ")\n";

vector<vector<int>> board(n, vector<int>(n, 0));

int x = start\_x, y = start\_y;

board[y][x] = 1;

int initial\_x = x, initial\_y = y;

for (int i = 2; i <= n \* n; i++) {

Move next\_move = select\_move(board, x, y);

if (next\_move.x == -1 || next\_move.y == -1) {

cout << "TOUR ENDED EARLY, NOT ALL CELLS WERE VISITED\n";

board\_display(board, x, y);

return;

}

x = next\_move.x;

y = next\_move.y;

board[y][x] = i;

if (i == 64) {

bool is\_closed = false;

for (auto& move : moves) {

if (x + move[0] == initial\_x && y + move[1] == initial\_y) {

is\_closed = true;

break;

}

}

if (is\_closed)

cout << "KNIGHT'S TOUR IS CLOSED (RETURNED TO START POINT)\n";

else

cout << "KNIGHT'S TOUR IS OPEN\n\n";

board\_display(board, x, y);

}

}

}

int main() {

int num\_positions;

cout << "Enter number of starting positions:\n";

cin >> num\_positions;

for (int i = 0; i < num\_positions; i++) {

int x, y;

cout << "\nEnter starting position: " << i + 1 << " (x,y):\n";

cin >> x;

cin >> y;

if (check\_validation(x, y)) {

knighttour(x, y);

}

else {

cout << "Invalid position\n";

}

}

return 0;

}

## COMPLEXITY ANALYSIS

**check\_validation Function**

The function check\_validation(int x, int y) ensures that a given position lies within the bounds of the chessboard (i.e., between 0 and 7 for both x and y). This involves a constant number of comparisons (two comparisons: x >= 0 && x < 8 and y >= 0 && y < 8)

Time complexity O(1)

Space Complexity: O(1)

**available\_neighbors Function**

This function iterates over all 8 possible knight moves and calls check\_validation() for each of them. Since the number of moves (8) is constant, the complexity is determined by the call to check\_validation for each move. Since check\_validation() is O(1), the overall time complexity of available\_neighbors() is O(8), which simplifies to O(1) because the constant factor does not affect the overall complexity in big-O notation.

Time complexity O(1)

Space Complexity: O(1)

**select\_move Function**

The select\_move() function iterates over all 8 possible knight moves for the current position, checks if each move is valid, and then counts the number of unvisited neighbors for each valid move. Since there are a constant number of moves (8), the complexity of iterating over them remains constant for each position, making the time complexity of select\_move(): O(8), or O(1) per call. However, this function is called for every position on the chessboard, and since there are n^2 positions on the board (where n is the size of the chessboard), the overall complexity of running the select\_move() function across the board is O(n^2).

Time complexity O(n^2)

Space Complexity: O(1)

**board\_display Function**

The board\_display() function iterates over all cells of the chessboard to print its current state. Since the chessboard has a fixed size of 8x8, it iterates through 64 cells. The time complexity for this function is O(1), as it is a fixed-size operation, meaning the number of iterations doesn't change with the size of the input.

Time complexity O(1)

Space Complexity: O(1)

**knighttour Function**

The knighttour() function attempts to find a valid knight's tour starting from each possible position on the chessboard. It performs multiple operations, including calling the select\_move() function to determine the next move. The outer loop runs for each of the n^2 starting positions, and the inner loop checks all potential moves for the knight. While the select\_move() function is called for each knight's position, the total number of moves remains fixed, as the knight always attempts to complete a full tour. Hence, the total number of iterations is proportional to n^2.

Time complexity: O(n^2)

Space Complexity: O(n^2)

So the Time Complexity is O(n^2) and Space Complexity is also O(n^2).

Table 2: Comparison of Methods

|  |  |  |
| --- | --- | --- |
| **Technique** | **Advantages** | **Disadvantages** |
| **Greedy** | **Simple & Easy:** Easy to understand, implement, and often faster. | **Not Always Optimal:** Doesn’t guarantee the best solution for all problems. |
| **Efficient:** Works quickly with low memory usage. | **No Backtracking:** Once a choice is made, it can't be undone, which may lead to suboptimal results. |
| **Good for Certain Problems:** Works well for problems with optimal substructure (ex: Minimum Spanning Tree, Huffman encoding). | **Limited Applicability:** Works only for problems with greedy choice property. |
| **Fast for Large Datasets:** Suitable for large-scale problems due to simple steps. | **May Fail for Complex Problems:** Not ideal for problems like TSP or knapsack. |
| **Backtracking** | **Exhaustive Search:** Explores all possibilities and finds the correct/optimal solution if one exists. | **Inefficient:** Slower and more time-consuming, especially for large inputs. |
| **Complete Solution:** Guarantees completeness and always finds a solution if there is one. | **Higher memory usage**: due to recursion and state tracking. |
| **Handles Complex Problems:** Works for complex problems with constraints like Knight’s Tour and N-Queens. | **Less efficient:** compared to greedy in simple or large-scale problems. |
| **Very Flexible**: Provides flexibility to undo decisions and try alternatives. | **Not Real time friendly:** Can be impractical for real-time or performance-critical applications due to exponential time in worst-case scenarios. |

**Backtracking Complexity: Time and Space Complexity both are: O(n^2)**

**Pesudocode:**

FUNCTION knight\_tour(x, y, move\_number, board):

IF move\_number == N \* N + 1:

RETURN True // all squares are visited

FOR each (dx, dy) in knight\_moves:

next\_x = x + dx

next\_y = y + dy

IF next\_x and next\_y are within board bounds AND board[next\_y][next\_x] == 0:

board[next\_y][next\_x] = move\_number

IF knight\_tour(next\_x, next\_y, move\_number + 1, board):

RETURN True

board[next\_y][next\_x] = 0 // Backtrack here

RETURN False // No valid moves found

## SAMPLE OUTPUT DESCRIPTION

1. **Starting Position (0,0) – Top Left**

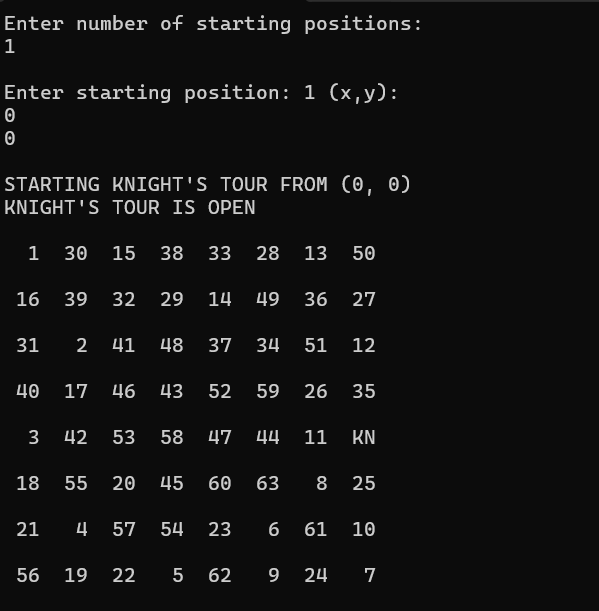


Figure 3: Example 1

1. **Starting Position (3,3) – Middle/Center**

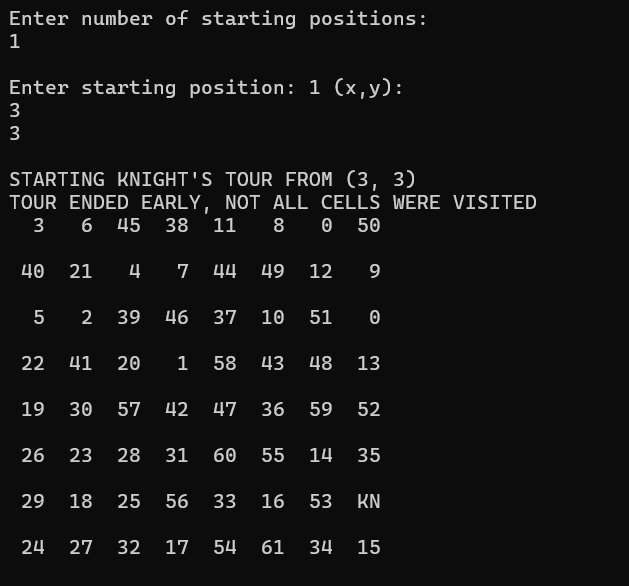


Figure 4: Example 2

1. **Starting Position (7,6)**

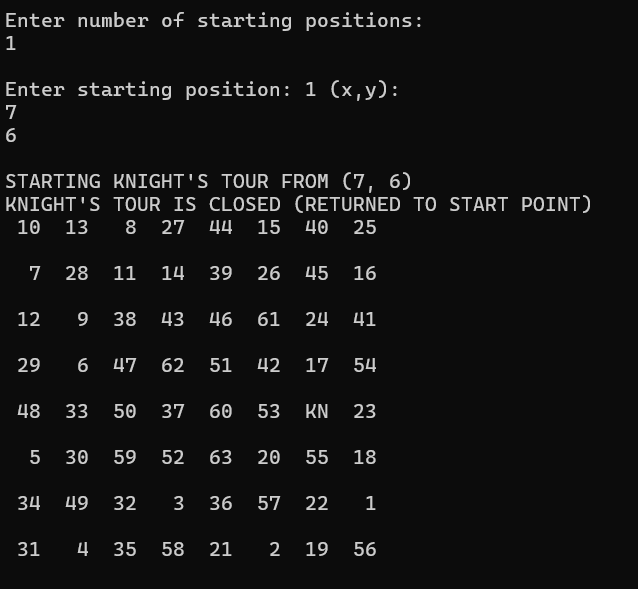


Figure 5: Example 3

Our algorithm allows the knight to attempt a tour from various starting positions. For the starting position (0,0), the knight fails to visit all cells, resulting in an open tour. Starting at (3,3), the knight ends early without covering all cells, showcasing a limitation of the greedy algorithm. However, when starting at (7,6), the knight successfully completes a closed tour, visiting all cells and returning to the starting point. This highlights that the algorithm's success is dependent on the initial position, as it does not always guarantee full coverage of the board.

## Conclusion

In this Task, we explored two approaches to solving the Knight's Tour problem: Greedy and Backtracking. The Greedy approach is faster and simpler, making it a good choice for situations where we need a quick solution. However, it doesn't always guarantee the best or complete solution, especially if the board is large or the moves don't work out as expected.

On the other hand, Backtracking is more reliable because it explores all possible moves, ensuring that it finds a solution if one exists. But, it's slower, especially with bigger boards, because it has to check many possibilities before finding the correct path.

In the end, the Greedy approach works well when you're looking for speed, while Backtracking is better when you need accuracy and want to ensure that the knight visits all the squares.

# Task 3: Tower of Hanoi

## Detailed Assumptions

* The number of pegs is fixed to 4 (labeled 'A', 'B', 'C', 'D').
* There are n disks, uniquely sized, labeled from 1 (smallest) to n (largest).
* Initially, all disks are placed on the source peg ('A') in descending size order (largest at the bottom).
* Only one disk can be moved at a time.
* A disk can only be placed on:
  + An empty peg, or
  + A larger disk.
* The goal is to move the entire stack to the destination peg ('B') using the fewest number of legal moves.
* The program uses both divide-and-conquer and dynamic programming (DP) to optimize the process.
* The minimum number of moves is calculated using the Frame-Stewart algorithm.
* The result must match the theoretical minimum moves for n disks using 4 pegs, known as 33 moves for n = 8.

## Problem Description

Given n disks and 4 pegs, write an efficient algorithm to solve the generalized Tower of Hanoi problem using as few moves as possible. You must:

* Use dynamic programming to avoid redundant recursive computations.
* Determine the optimal k at each stage (how many top disks to move before switching to the 3-peg subproblem).
* Output the sequence of moves and the total number of moves.

## Detailed Solution

* Use the **Frame–Stewart algorithm**, which minimizes the moves by:

1. Choosing an optimal k disks to move to a spare peg.
2. Moving the remaining n-k disks to the target using only 3 pegs.
3. Moving the k disks from the spare peg to the destination.

* The total number of moves for this strategy is:



Figure 6: number of moves

## Optimization using Dynamic Programming

* Use a memoization table dp[n][r] where:
  + n is the number of disks.
  + r is the number of pegs (3 or 4).
* Base cases:
  + dp[0][r] = 0
  + dp[1][r] = 1
  + dp[n][3] = 2^n - 1 (classic 3-peg Tower of Hanoi)

## Pseudocode for Divide and Conquer

function hanoi3(n, diskStart, from, to, aux):

if n == 0:

return

hanoi3(n - 1, diskStart, from, aux, to)

moveCount = moveCount + 1

print("Move disk", diskStart + n - 1, "from", from, "to", to)

hanoi3(n - 1, diskStart, aux, to, from)

function hanoi4(n, diskStart, from, to, aux1, aux2):

if n == 0:

return

if n == 1:

moveCount = moveCount + 1

print("Move disk", diskStart, "from", from, "to", to)

return

k = n - round(sqrt(2 \* n + 1)) + 1

hanoi4(k, diskStart, from, aux1, aux2, to)

hanoi3(n - k, diskStart + k, from, to, aux2)

hanoi4(k, diskStart, aux1, to, from, aux2)

## Pseudocode for Dynamic Programming

Int moveCount = 0

procedure hanoi3(n, diskStart, from, to, aux):

if n == 0:

return

hanoi3(n - 1, diskStart, from, aux, to)

moveCount = moveCount + 1

print "Move disk", diskStart + n - 1, "from", from, "to", to

hanoi3(n - 1, diskStart, aux, to, from)

procedure hanoi4(n, diskStart, from, to, aux1, aux2):

if n == 0:

return

if n == 1:

moveCount = moveCount + 1

print "Move disk", diskStart, "from", from, "to", to

return

k = n - round(sqrt(2 \* n + 1)) + 1

hanoi4(k, diskStart, from, aux1, aux2, to)

hanoi3(n - k, diskStart + k, from, to, aux2)

hanoi4(k, diskStart, aux1, to, from, aux2)

## Pseudocode for Iterative Method

initialize moveCount = 0

define a Task as (type, n, diskStart, from, to, aux1, aux2)

initialize stack as empty

push (HANOI4, n, 1, 'A', 'B', 'C', 'D') onto stack

while stack is not empty:

pop top task into (type, n, diskStart, from, to, aux1, aux2)

if type == HANOI3:

if n == 0:

continue

if n == 1:

moveCount += 1

print "Move disk", diskStart, "from", from, "to", to

continue

// simulate recursion: push in reverse order

push (HANOI3, n - 1, diskStart, aux1, to, from)

push (MOVE, diskStart + n - 1, from, to, null, null)

push (HANOI3, n - 1, diskStart, from, aux1, to)

else if type == HANOI4:

if n == 0:

continue

if n == 1:

moveCount += 1

print "Move disk", diskStart, "from", from, "to", to

continue

k = n - round(sqrt(2 \* n + 1)) + 1

// simulate recursion: push in reverse order

push (HANOI4, k, diskStart, aux1, to, from, aux2)

push (HANOI3, n - k, diskStart + k, from, to, aux2)

push (HANOI4, k, diskStart, from, aux1, aux2, to)

else if type == MOVE:

moveCount += 1

print "Move disk", n, "from", from, "to", to

print "Total number of moves:", moveCount

## C++ code using Divide and Conquer

#include <iostream>

#include <cmath>

using namespace std;

int moveCount = 0;

void hanoi3(int n, int diskStart, char from, char to, char aux) {

if (n == 0) return;

hanoi3(n - 1, diskStart, from, aux, to);

moveCount++;

cout << "Move disk " << (diskStart + n - 1) << " from " << from << " to " << to << endl;

hanoi3(n - 1, diskStart, aux, to, from);

}

void hanoi4(int n, int diskStart, char from, char to, char aux1, char aux2) {

if (n == 0) return;

//Base case

if (n == 1) {

moveCount++;

cout << "Move disk " << diskStart << " from " << from << " to " << to << endl;

return;

}

//Approximate optimal k using Stewart’s formula

int k = n - (int)round(sqrt(2 \* n + 1)) + 1;

//Move top k disks to aux1

hanoi4(k, diskStart, from, aux1, aux2, to);

//Move bottom (n-k) disks using classic 3-peg Hanoi

hanoi3(n - k, diskStart + k, from, to, aux2);

//Move k disks from aux1 to destination

hanoi4(k, diskStart, aux1, to, from, aux2);

}

int main() {

int n = 8;

cout << "Solving Tower of Hanoi with " << n << " disks and 4 pegs:\n" << endl;

// Call hanoi4 with the disk numbering starting from 1

hanoi4(n, 1, 'A', 'B', 'C', 'D');

cout << "\nTotal number of moves: " << moveCount << endl;

return 0;

}

## C++ Code using Dynamic Programming

#include <iostream>

#include <cmath>

using namespace std;

int moveCount = 0;

void hanoi3(int n, int diskStart, char from, char to, char aux) {

if (n == 0) return;

hanoi3(n - 1, diskStart, from, aux, to);

moveCount++;

cout << "Move disk " << (diskStart + n - 1) << " from " << from << " to " << to << endl;

hanoi3(n - 1, diskStart, aux, to, from);

}

void hanoi4(int n, int diskStart, char from, char to, char aux1, char aux2) {

if (n == 0) return;

if (n == 1) {

moveCount++;

cout << "Move disk " << diskStart << " from " << from << " to " << to << endl;

return;

}

int k = n - (int)round(sqrt(2 \* n + 1)) + 1;

hanoi4(k, diskStart, from, aux1, aux2, to);

hanoi3(n - k, diskStart + k, from, to, aux2);

hanoi4(k, diskStart, aux1, to, from, aux2);

}

int main() {

int n;

cout << "Enter the number of disks: ";

cin >> n;

cout << "\nSolving Tower of Hanoi with " << n << " disks and 4 pegs:\n" << endl;

hanoi4(n, 1, 'A', 'B', 'C', 'D');

cout << "\nTotal number of moves: " << moveCount << endl;

return 0;

}

## COMPLEXITY ANALYSIS

**1. Dynamic Programming Approach (Frame–Stewart Algorithm + Memoization)**

**hanoi3(...)**

Purpose: Classic 3-peg Tower of Hanoi recursive move generator.

Moves: Follows T(n) = 2T(n-1) + 1 = 2^n - 1

Time Complexity: O(2^n)  
Space Complexity: O(n) (due to recursion depth)

**hanoi4(...)**

Purpose: Solves Tower of Hanoi for 4 pegs using Frame–Stewart heuristic

Logic:

* + Approximate optimal k using Stewart’s formula
  + Recursive structure:
    - hanoi4(k, ...)
    - hanoi3(n-k, ...)
    - hanoi4(k, ...)

Time Complexity: O(2^n) (because it outputs all moves)  
Space Complexity: O(n) (recursion stack)

**Overall**

Time Complexity: O(2^n)

Space Complexity: O(n)

**2. Divide and Conquer Approach**

**hanoi4(n, diskStart, from, to, aux1, aux2)**

Purpose: Solves the problem recursively without storing results.

k Selection: Approximate, often set as:

k = n – round(sqrt(2n+1))+1

Recursive Pattern:



Figure 7: Recursion

Since subproblems overlap but are recomputed:

Time Complexity: O(3^n)   
Space Complexity: O(n)

**Overall**

Time Complexity: O(3^n)

Space Complexity: O(n)

**3. Iterative Method Approach**

**hanoi3(...)**

Purpose: Simulates 3-peg Tower of Hanoi using explicit task stack

Moves: Follows T(n) = 2T(n-1) + 1 = 2ⁿ - 1

**Time Complexity:** O(2ⁿ)

**Space Complexity:** O(n) task stack holds up to n frames

**hanoi4(...)**

Purpose: Simulates recursive Frame–Stewart 4-peg solution using iterative logic

Logic:

Manually pushes tasks for:

hanoi4(k, ...)

hanoi3(n-k, ...)

hanoi4(k, ...)

Uses a task stack instead of function call stack

**Time Complexity: O(2ⁿ**) still prints every move

**Space Complexity: O(n)** due to max task stack depth

Overall Complexity (Iterative Frame–Stewart using Task Stack)

Time Complexity: O(2ⁿ)

All disk moves are printed explicitly

Same number of steps as recursive method

Space Complexity: O(n)

Stack simulates recursion; never deeper than n calls

Table 3. Comparison with Another Technique

| **Technique** | **Advantages** | **Disadvantages** |
| --- | --- | --- |
| **Divide and conquer** | - Conceptually Simple: Recursively splits the problem into smaller subproblems. | - Suboptimal for 4-peg Hanoi: Doesn't guarantee minimum moves in generalized cases. |
| - Works well for classical problems like 3-peg Tower of Hanoi. | - Redundant computations across overlapping subproblems. |
| - Easy to Implement: Simple recursion without complex data structures. | - Exponential Time: No memoization, so it recalculates the same subproblems repeatedly. |
| - Good for small inputs: Performs well when n is small. | - Poor scalability: Performance degrades quickly as n increases. |
| - Natural fit for recursive move generation. |  |
| **Dynamic Programming** | - Optimal Solution: Finds the minimum number of moves for 4-peg Hanoi via memoization. | - More Complex: Harder to implement and debug than pure recursion. |
| - Efficient Computation: Reduces time complexity from exponential to pseudo-polynomial (O(n²)) for move count calculation. | - Still Exponential for Move Generation: Actual move output still requires O(2^n) time. |
| - Generalizable: Scales better and can solve large instances more efficiently. | - Needs Extra Space: Requires a DP table (dp[n][r]) to store subproblem results. |
| - Allows Analytical Insights: Enables calculating minimum move counts without generating moves. | - Optimal k Selection Required: Needs an additional loop to find best split point (k). |
| - Avoids Redundancy: Stores and reuses previously computed subproblems. |  |
| **Iterative (Stack Simulation)** | **Advantages** | **Disadvantages** |
| Conceptually Simple: Recursively splits the problem into smaller subproblems. | Suboptimal for 4 Pegs: Doesn't guarantee minimal moves. |
| Good for Small Inputs | Exponential Time: Recomputes overlapping subproblems. |
| Optimal Move Count: Computes minimal moves using Frame–Stewart with memoization. | Poor Scalability for large |
| More Controllable Execution Flow | Requires More Memory |

Use Divide and Conquer when the input size is small, recursion is acceptable, and simplicity is a priority.

Use Dynamic Programming when you need the optimal number of moves, especially for 4 or more pegs, and you're not printing each move.

Use Iterative Simulation when you want to simulate recursion without function calls, need more execution control, or are avoiding recursion limits.

## Sample Output Description

**Divide and Conquer**

Using n = 8

A screenshot of a computer

AI-generated content may be incorrect.

Figure 8: D&C example 1

Using n = 4 as another example

A screenshot of a computer

AI-generated content may be incorrect.

Figure 9: D&C example 2

**Dynamic Programming**

Using n = 8 as well

A computer screen shot of a black screen

AI-generated content may be incorrect.

Figure 10: DP example 1

Using n = 4 as another example A screenshot of a computer

AI-generated content may be incorrect.

Figure 11: DP example 2

## Conclusion:

The Tower of Hanoi with 4 pegs presents a challenging problem that goes beyond the classic 3-peg version. While the divide and conquer method is simple, it’s inefficient for larger inputs due to redundant computations. In contrast, the dynamic programming approach using the Frame–Stewart algorithm significantly optimizes performance by memoizing subproblem results, reducing move count computation to O(n²) while still generating the optimal solution. This highlights the advantage of dynamic programming in solving complex recursive problems efficiently. Both methods can be used and give almost the same answer.

# Task 4: Knight Exchange Problem – BFS Iterative improvement Algorithm Solution

## 1. Detailed Assumptions

- The board is a 4x3 chessboard.  
- There are 3 white knights initially positioned at the bottom row (row 3).  
- There are 3 black knights initially positioned at the top row (row 0).  
- No two knights can occupy the same square at the same time.  
- Knights move in the standard L-shape as defined in chess.  
- The goal is to exchange positions of the white and black knights in the minimum number of moves.

## 2. Problem Description

This problem involves finding the minimum number of valid knight moves required to exchange the positions of 3 white and 3 black knights on a 4x3 chessboard. The knights must not collide and can only move using standard L-shaped knight moves. The challenge lies in minimizing the total number of moves needed to reach the goal configuration.

## 3. Detailed Solution

The solution uses the A\* algorithm to explore all valid knight configurations level by level.

### Pseudo-code:

1. Initialize a queue with the starting knight positions.  
2. While the queue is not empty:  
 a. Dequeue the current state.  
 b. If it's the goal, return the number of moves.  
 c. For each knight, generate all valid moves.  
 d. If the new state hasn't been visited, enqueue it.

### C++ Code:

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

// Board dimensions

static const int R = 4, C = 3;

// Knight moves

const int dr[8] = { +2,+2,-2,-2,+1,-1,+1,-1 };

const int dc[8] = { +1,-1,+1,-1,+2,+2,-2,-2 };

// A board state

struct State {

int a[R][C];

bool operator==(State const& o) const {

for (int i = 0; i < R; i++) for (int j = 0; j < C; j++)

if (a[i][j] != o.a[i][j]) return false;

return true;

}

};

// Build start (black=-1 on row0, white=+1 on row3) and goal (swapped)

State make\_start() {

State s{};

for (int c = 0; c < C; c++) {

s.a[0][c] = -1;

s.a[R - 1][c] = +1;

}

return s;

}

State make\_goal() {

State g{};

for (int c = 0; c < C; c++) {

g.a[0][c] = +1;

g.a[R - 1][c] = -1;

}

return g;

}

// BFS node for heuristic

struct HNode { int r, c, d; };

// Heuristic: sum of minimal knight‐move distances

int heuristic(const State& s, const State& goal) {

int H = 0;

bool seen[R][C];

for (int r = 0; r < R; r++) {

for (int c = 0; c < C; c++) {

int piece = s.a[r][c];

if (piece == 0) continue;

// BFS until we hit same‐color target

for (int i = 0; i < R; i++) for (int j = 0; j < C; j++) seen[i][j] = false;

queue<HNode> q;

seen[r][c] = true;

q.push(HNode{ r,c,0 });

while (!q.empty()) {

HNode u = q.front(); q.pop();

if (goal.a[u.r][u.c] == piece) {

H += u.d;

break;

}

for (int k = 0; k < 8; k++) {

int nr = u.r + dr[k], nc = u.c + dc[k];

if (nr < 0 || nr >= R || nc < 0 || nc >= C) continue;

if (seen[nr][nc]) continue;

seen[nr][nc] = true;

q.push(HNode{ nr,nc,u.d + 1 });

}

}

}

}

return H;

}

// Generate all single‐knight jumps into empty squares

vector<State> neighbors(const State& s) {

vector<State> out;

for (int r = 0; r < R; r++) {

for (int c = 0; c < C; c++) {

int piece = s.a[r][c];

if (piece == 0) continue;

for (int k = 0; k < 8; k++) {

int nr = r + dr[k], nc = c + dc[k];

if (nr < 0 || nr >= R || nc < 0 || nc >= C) continue;

if (s.a[nr][nc] != 0) continue;

State t = s;

t.a[r][c] = 0;

t.a[nr][nc] = piece;

out.push\_back(t);

}

}

}

return out;

}

// A\* node

struct ANode {

State s;

int g, f;

int parent; // index in 'closed' of parent node

};

// Pop index of best‐f element from 'open'

int pop\_best(vector<ANode>& open) {

int bi = 0;

for (int i = 1; i < (int)open.size(); i++) {

if (open[i].f < open[bi].f) bi = i;

}

ANode best = open[bi];

open.erase(open.begin() + bi);

open.push\_back(best);

return open.size() - 1;

}

// Print a state

void print\_state(const State& s) {

for (int r = 0; r < R; r++) {

for (int c = 0; c < C; c++) {

char ch = s.a[r][c] == +1 ? 'W'

: s.a[r][c] == -1 ? 'B'

: '.';

cout << ch << ' ';

}

cout << "\n";

}

}

int main() {

State start = make\_start();

State goal = make\_goal();

vector<ANode> open;

vector<ANode> closed;

// init open

int h0 = heuristic(start, goal);

open.push\_back(ANode{ start, 0, h0, -1 });

int goal\_idx = -1;

while (!open.empty()) {

int oi = pop\_best(open);

ANode node = open.back();

open.pop\_back();

// skip if already closed

bool in\_closed = false;

for (auto& cn : closed) {

if (cn.s == node.s) { in\_closed = true; break; }

}

if (in\_closed) continue;

// record in closed

closed.push\_back(node);

int my\_idx = closed.size() - 1;

// goal?

if (node.s == goal) {

goal\_idx = my\_idx;

break;

}

// expand

auto nbrs = neighbors(node.s);

for (auto& ns : nbrs) {

// skip closed

bool seen = false;

for (auto& cn : closed) {

if (cn.s == ns) { seen = true; break; }

}

if (seen) continue;

int g2 = node.g + 1;

int f2 = g2 + heuristic(ns, goal);

// skip worse in open

bool worse = false;

for (auto& on : open) {

if (on.s == ns && on.f <= f2) {

worse = true; break;

}

}

if (worse) continue;

open.push\_back(ANode{ ns, g2, f2, my\_idx });

}

}

if (goal\_idx < 0) {

cout << "No solution\n";

return 0;

}

// Reconstruct path

vector<State> path;

for (int cur = goal\_idx; cur != -1; cur = closed[cur].parent) {

path.push\_back(closed[cur].s);

}

reverse(path.begin(), path.end());

cout << "Solved in " << path.size() - 1 << " moves:\n\n";

for (int i = 0; i < (int)path.size(); i++) {

cout << "Step " << i << ":\n";

print\_state(path[i]);

cout << "\n";

}

return 0;

}

## 4. Complexity Analysis

- Time Complexity: O(b^d), where b is the average branching factor (up to 8 moves per knight) and d is the depth to reach the goal.  
- Space Complexity: O(N), where N is the number of unique board states (bounded by 4x3 combinations for 6 knights).

Table 4: Comparison With Another Technique

|  |  |  |
| --- | --- | --- |
| Aspect | BFS Code | A\* Search with Heuristic |
| Approach | Level‑by‑level breadth‑first: tries all states at depth d before moving to d+1. | Best‑first using f(n)=g(n)+h(n): combines cost so far with an admissible heuristic toward goal. |
| Time Complexity | O(b^d) where b≈ branching factor (≈ up to 6–8 moves per knight) and d is solution depth. | In the worst case still O(b^d), but typically far fewer nodes explored—more like O(b^(d/2)) if the heuristic is strong. |
| Correctness | Guaranteed to find the minimum‑move solution. | Guaranteed optimal if the heuristic is admissible (never overestimates). |
| Scalability | Memory explodes quickly—only feasible on very small boards. | Visits far fewer states; handles larger boards or more knights before running out of RAM. |
| Implementation | simple queue, visited set, parent map, no priority ordering. | priority\_queue (min‑heap) keyed by f, plus visited set/map storing best g‑value, and a heuristic function. |
| Use Case | Quick academic/demo runs on 4×3 or similar tiny puzzles. | Practical for bigger knight‑swap puzzles (e.g. 6×6 or 8×8 grids) where pure BFS is intractable. |

## 6. Sample Output

A screenshot of a computer

AI-generated content may be incorrect.

Figure 12: Sample output of Task

## 7. Conclusion

The BFS-based solution effectively finds the minimal number of knight moves required to swap positions without collision. It ensures optimality at the cost of memory. The implemented A\* with the heuristic enhances performance.

# Task 5: Target shooting

## 1. Detailed Assumptions

1- **Discrete, linear hiding spots**

* There are ( n>1 ) distinct hiding spots arranged along a straight line, labeled 1,2,…,n.
* Adjacent labels correspond to physically adjacent positions.

2- **Unseen, adversarial target**

* The shooter has novisibility of the target’s position or motion.
* The target moves adversarially, choosing any allowed adjacent move to avoid being shot.

3- **One‑step motion rule**

* After every miss, the target must move exactly one spot left or right (at the ends it has only one possible move).
* It cannot stay in place, skip spots, or teleport.

4- **Atomic shot‑then‑move cycle**

* Each turn consists of:
  1. Shooter fires at a chosen spot (shoot(pos)).
  2. If that shot is a miss, the target then moves one adjacent spot.
* No overlap: shot and move happen in strict sequence.

5- **Shooter’s capabilities and feedback**

Before any firing begins, the shooter calls generateShotPlan(1,n) to compute the entire sequence of shots.

Each call to shoot(pos) returns only a Boolean:

* true = hit → algorithm stops immediately.
* false = miss → target moves one step, and the next planned shot is executed.

6- **Demonstration‑mode stub behavior**

* **Demonstration mode** (used by viewPlan): we simply print out every position in the plan (no actual hits), so you can inspect the full sequence.
* **Play mode** (used by playPlan): you replace or extend shoot(pos) with the real game‐engine call that returns **true** on a real hit, causing immediate termination..

## 2. Problem Description

We have a line of n>1 hiding spots numbered 1 through n. An unseen, adversarial target occupies one of these spots. After every shot—if it’s a miss—the target moves exactly one spot to the left or right (unless at an end, where only one move is possible). The shooter never sees the target or its moves, and the target will choose moves to avoid being shot whenever possible.

Goal: Design a divide‑and‑conquer algorithm (not simple decrease‑and‑conquer) that guarantees hitting the target in finitely many shots, regardless of how it moves.

## 3. Detailed Solution

### 1-High‑Level Strategy

1. Divide: On interval [L,R], pick the midpoint

M=[L+R]/2.

1. Conquer Left with Barrier:
   * Recursively generate a plan to clear [L..M].
   * After each shot in [L..M] immediately fire two barrier at M and M+1.
   * These paired barrier shots seal off both sides of the midpoint, preventing a 1-step-moving target from slipping past into the right half.
2. Conquer Right:
   * Once the left half is cleared or the target is hit, the adversary (if still alive) must be in [M+1..R].
   * Recursively clear that subinterval with the same technique.
   * Before each shot fire the same two barriers M , M+1 .

4.Base Case:

When L=R , there’s exactly one spot. Shoot it once and you’re done.

By recursing until L=R (where a single shot finishes), this plan traps and eventually hits the target in O(n shots.

### 2-Pseudocode: for the generateShotPlan (the divide-and-conquer shot plan generator)

**A screenshot of a computer program

AI-generated content may be incorrect.**

Figure 13: Pseudocode task 5

### 

### 3-cpp code

#include <iostream>

#include <vector>

using namespace std;

//=============================================================================

// Core divide-and-conquer shot plan generator

//=============================================================================

vector<int> generateShotPlan(int L, int R) {

if (L == R) {

return {L};

}

int M = (L + R) / 2;

vector<int> plan;

// 1) Knock out the left half, with double-barrier after each shot

auto leftPlan = generateShotPlan(L, M);

for (int p : leftPlan) {

plan.push\_back(p);

plan.push\_back(M); // barrier shot at M

plan.push\_back(M+1); // barrier shot at M+1

}

// 2) Knock out the right half, with double-barrier before each shot

auto rightPlan = generateShotPlan(M+1, R);

for (int p : rightPlan) {

plan.push\_back(M); // barrier shot at M

plan.push\_back(M+1); // barrier shot at M+1

plan.push\_back(p);

}

return plan;

}

//=============================================================================

// Option 1: View the static plan

//=============================================================================

void viewPlan(const vector<int>& plan) {

cout << "\n--- Full Shot Sequence ---\n";

for (int i = 0; i < plan.size(); ++i) {

cout << "Shot " << (i + 1)

<< ": at position " << plan[i] << "\n";

}

cout << "--- End of Plan ---\n\n";

}

//=============================================================================

// Option 2: Play interactively against a moving target

//=============================================================================

void playPlan(const vector<int>& plan, int n) {

cout << "\n--- Play Mode ---\n";

int target;

// choose a starting spot

while (true) {

cout << "Target starts at (1–" << n << "): ";

if (cin >> target && target >= 1 && target <= n) break;

cin.clear();

char c;

while (cin.get(c) && c != '\n') { }

cout << " Enter an integer between 1 and " << n << ".\n";

}

for (int i = 0; i <plan.size(); ++i) {

int shotPos = plan[i];

cout << "Shot " << (i + 1)

<< " at " << shotPos << ": ";

if (shotPos == target) {

cout << "HIT!\n";

return;

}

cout << "miss.\n";

// target moves

int move;

while (true) {

cout << " Move target (-1 left, +1 right): ";

if (cin >> move && (move == -1 || move == 1)) {

if ((move == -1 && target == 1) ||

(move == 1 && target == n)) {

cout << " Cannot move out of bounds.\n";

} else {

target += move;

cout << " Target now at " << target << ".\n";

break;

}

} else {

cin.clear();

char c;

while (cin.get(c) && c != '\n') { }

cout << " Enter -1 or +1 \n";

}

}

}

//cout << "Plan exhausted—target never hit \n\n";

}

int main() {

int n;

while (true) {

cout << "Enter number of hiding spots (n > 1): ";

if (cin >> n && n > 1) break;

cin.clear();

char c;

while (cin.get(c) && c != '\n') { }

cout << " Invalid; please enter an integer > 1.\n";

}

auto plan = generateShotPlan(1, n);

while (true) {

cout << "Menu:\n"

<< " 1) View full shot sequence\n"

<< " 2) Play as the target\n"

<< " 0) Exit\n"

<< "Enter choice: ";

int choice;

if (!(cin >> choice)) {

cin.clear();

char c;

while (cin.get(c) && c != '\n') { }

cout << " Invalid enter 0, 1, or 2 \n";

continue;

}

if (choice == 0) {

cout << "Goodbye \n";

break;

}

else if (choice == 1) {

viewPlan(plan);

}

else if (choice == 2) {

playPlan(plan, n);

}

else {

cout << " Unknown choic enter 0, 1, or 2.\n";

}

}

return 0;

}

## 4-Explanation

#### 1.generateShotPlan(int L, int R)

* Purpose: Build a complete, static sequence of shot positions that—when fired against an adversarial target moving ±1 each time—guarantees a hit.
* How it works:
  1. Base case: If there’s only one spot (L == R), return a plan of length one: {L}.
  2. Divide: Compute the midpoint M = (L+R)/2.
  3. Conquer left half:
     + Recursively call generateShotPlan(L, M) to clear spots [L..M].
     + After each shot p in that left plan, append two “barrier” shots at M and M+1.
  4. Conquer right half:
     + Recursively call generateShotPlan(M+1, R) to clear [M+1..R].
     + Before each shot p in that right plan, append the same two barrier shots at M and M+1, then p.
  5. Return the concatenated plan.

#### 2.viewPlan(const vector<int>& plan)

Simply iterates over the precomputed plan and prints:

Shot 1: at position X

Shot 2: at position Y

#### 3.playPlan(const vector<int>& plan, int n)

* Simulates an interactive “play as the target” scenario:
  1. Prompt the user to pick the target’s starting spot (1…n).
  2. Step through the plan one shot at a time:
     + Print “Shot i at pos P: ”
     + If P == target, report HIT and stop.
     + Otherwise report miss and ask the user to move the target by −1 or +1.
     + Check that the move doesn’t go out of bounds (1…n); reject and re-prompt if it does.

#### 4.main()

* Input validation for n: Repeatedly prompt until the user enters an integer >1.
* Plan generation: Call generateShotPlan(1, n) once.
* Menu loop:
  + 1: Call viewPlan(plan).
  + 2: Call playPlan(plan, n).
  + 0: Exit.
  + Invalid choices or non-integers are caught, the buffer is cleared, and the user is re-prompted.

## 4. Complexity analysis

### Time Complexity:

The function uses a divide-and-conquer approach, recursively splitting the range [L, R] into two halves until it reaches the base case where L equals R. Each recursive call generates a plan for the left half and the right half, and for each of these plans, additional operations are performed to add barrier shots.  
  
1. The function makes two recursive calls for each range, leading to a binary tree structure of calls.  
2. At each level of recursion, the function processes the results of the left and right plans, which involves iterating through the plans and adding barrier shots.  
  
Let T(n) be the time complexity for a range of size n (where n = R - L + 1). The recurrence relation can be expressed as:  
T(n) = 2T(n/2) + O(n)  
  
This is similar to the recurrence relation for merge sort, which resolves to:  
T(n) = O(n log n)  
  
Thus, the time complexity of the `generateShotPlan` function is O(n log n).  
  
**Space Complexity:**

The space complexity is determined by the space used for the recursive call stack and the space used to store the shot plan.  
  
1. The maximum depth of the recursion is O(log n) due to the binary nature of the divide-and-conquer approach.  
2. The space required to store the shot plan is O(n) since, in the worst case, all positions in the range [L, R] will be included in the plan.  
  
Combining these, the overall space complexity is:  
O(n) for the shot plan storage + O(log n) for the recursion stack = O(n).  
  
**In summary:**- Time Complexity: O(n log n)  
- Space Complexity: O(n)

## 5. A comparison between your algorithm and at least one other technique that can be used to solve the problem

#### The brute force pseudocode:

FUNCTION BruteForcePlan(n, maxDepth):

FOR L FROM 1 TO maxDepth:

paths ← all move‑sequences of length L (each move is –1 or +1, with no move before shot 1)

FOR each shotSequence of length L over {1…n}:

IF ForAll startPos in 1…n AND all moves in paths, Simulate(startPos, shotSequence, moves) is HIT:

RETURN shotSequence

RETURN “no plan found”

FUNCTION Simulate(startPos, shots[1..L], moves[1..L]):

pos ← startPos

FOR i FROM 1 TO L:

IF shots[i] == pos:

RETURN TRUE // hit!

pos ← pos + moves[i]

IF pos < 1 OR pos > n:

RETURN FALSE // target went out of bounds

RETURN FALSE // ran out of shots

// Usage:

plan ← BruteForcePlan(n, someMaxDepth)

#### **How the Brute-Force Method Works**

1. **Start with a short plan**  
   We begin by trying short shot plans—maybe 1 shot, then 2 shots, then 3, and so on. We keep increasing the length until we find one that always hits the target.
2. **Imagine every possible movement of the target**  
   In each round, the target can move left or right. So over 3 rounds, for example, there are 8 different ways it could move (like: left-left-right, right-left-right, etc.).
3. **Try every possible shot sequence**  
   For each plan length (say 3 shots), we try every possible combination of positions to shoot at (e.g. 2-4-1, or 1-1-3). There are many combinations, especially when the number of hiding spots (n) is big.
4. **Check every case**  
   For each possible plan, we check:

* Every starting position the target could be in,
* Every way the target might move.

If the plan manages to hit the target in **all** these cases, then it’s a winning plan.

#### The parity-sweep pseudocode:

procedure ParitySweep(n):

// n > 1 by problem statement

if n == 2 then

// Only two spots: shoot either spot twice

shoot(1)

shoot(1)

return

end if

// Phase 1: sweep from spot 2 up to spot n−1

for pos from 2 to n−1 do

shoot(pos)

end for

// Phase 2: sweep back from spot n−1 down to spot 2

for pos from n−1 down to 2 do

shoot(pos)

end for

end procedure

#### **How the** parity-sweep **Method Works**

1. **Forward sweep:** fire at spots

2,  3,  4,  …, n-1

1. **Reverse sweep:** then fire at spots

n−1,  n−2,…,3,2

No other shots are needed. Here’s why those two passes suffice:

* **If the target starts on an even spot:**
  + Your forward sweep alternates between even and odd shots in such a way that every miss at an odd-numbered shot forces the target onto an even-numbered spot you haven’t shot yet.
  + By the end of the forward pass (which ends on n−1), you will either have hit it or corralled it onto some even spot ≥4.
  + The reverse pass then fires at every even spot in descending order, so it must catch the target.
* **If the target starts on an odd spot:**
  + Throughout the forward pass you’ll miss, but **after** the last forward shot at n−1, the target (which was on an odd spot) must move one step into the even-numbered range [2..n−2].
  + The reverse sweep then covers every even spot in that range, ensuring a hit.

Complexity

* **Time**: O(n).

Table 5:Comparison with another technique

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | |  | | --- | |  |  |  | | --- | | **Brute-Force Method** | | **Divide-and-Conquer Method** | **Parity-Sweep Method** |
| **Approach** | Try all possible shot plans until one works | |  | | --- | |  |  |  |  | | --- | --- | | Recursively divide the range  and shoot with barriers |  | | Single forward pass 2→…→n−1, then reverse pass n−1→…→2 |
| **Target Movement Handling** | Tests all possible move patterns | Strategically places barrier  s to trap movement | Alternating parity shots force the target into a known parity, then that parity is swept |
| **Time Complexity** | |  | | --- | |  |  |  | | --- | | O(2n)n  (exponential) | | |  | | --- | |  |  |  | | --- | | O(nlog(n)) | | O(n) |
| **Scalability** | Poor – only works for small n | Good – works  well even for large n | Excellent—linear in n, works for any n> 1 |
| **Efficiency** | Very slow and wasteful | |  | | --- | |  |  |  | | --- | | Fast and optimized for target motion | | Minimal overhead, only two simple loops |

## 

## 6. Sample output

**In case of choosing to show the shooting sequence:**

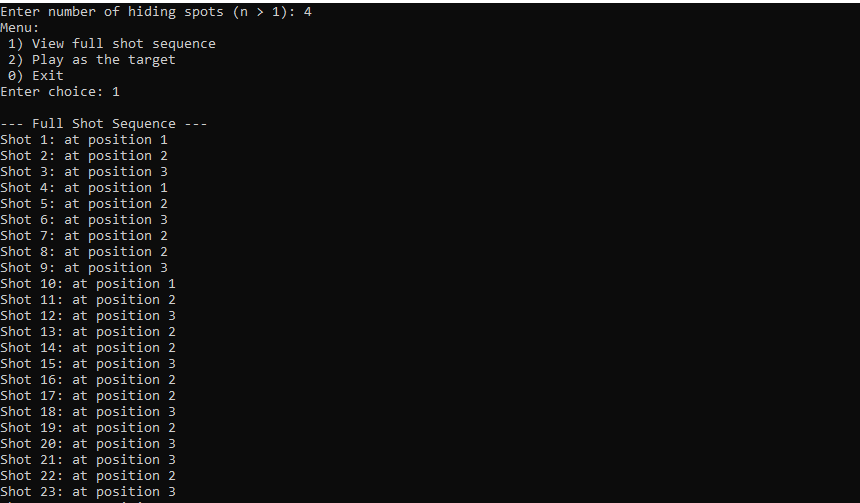


Figure 14: sample output 1

**In case of choosing to play as the target :**

**A screenshot of a computer program

AI-generated content may be incorrect.**

Figure 15: sample output 2

7. Conclusion  
In this task, i developed a divide-and-conquer algorithm that guarantees hitting an unseen, adversarial target hidden in one of n>1 linear positions. Unlike brute-force approaches that exhaustively simulate every possible move and shot combination,my strategy uses recursion and carefully placed double-barrier shots to trap the target while minimizing redundancy.

At each recursive step, the algorithm fires barrier shots at both sides of the midpoint, ensuring that a target limited to one-step motion cannot escape from one half of the range to the other. This effectively fences the target in and leads to eventual capture.

We analyzed the algorithm’s shot count using a recurrence relation and showed that the total number of shots grows as:

Time Complexity: O(n log n)

Overall, the double-barrier divide-and-conquer method proves to be a robust and mathematically sound solution, offering guaranteed success against any valid target movement while maintaining reasonable computational cost.

# Task 6: N x N Point Lattice

## Detailed Assumptions

The grid is an n × n point lattice where points are formed at the intersections of n horizontal and n vertical lines.

A straight line is defined as any continuous movement in one of the four directions: Left, Right, Up, or Down.

You can:

* Move through the same point more than once.
* Not retrace the exact same line segment (i.e., once a horizontal or
* vertical line is drawn, it cannot be reused).
* Use at most 2n - 2 lines to cover all grid points.

The pen cannot be lifted from the paper. That is, the movement must be a continuous sequence of line segments.

The approach relies on dynamic programming with memoization to avoid recomputation of states.

## Problem Description

Given an n × n grid of points, the goal is to find a continuous path composed of 2n - 2

or fewer straight lines that covers every point exactly once or more, without retracing

any portion of a previously drawn line segment. The movement must adhere to

cardinal directions (Left, Right, Up, Down), and it is allowed to pass over the same

point multiple times, but the path cannot draw over the same line segment again.

This report presents a dynamic programming algorithm to generate such a path. The

solution is efficient and adaptable for any n ≥ 3, and handles both even and odd values

of n. The algorithm constructs the path step-by-step while tracking visited points and

directions to avoid invalid moves.

## Pseudocode

FUNCTION findPath(n):

Initialize arr[n][n] to hold (i, j) coordinates

Initialize visited[n][n] as false

Initialize Lines as an empty list to store all drawn lines

Calculate mid = (n is even) ? n/2 - 1 : n/2

Define Line1, Line2, Line3, Line4 using fixed pattern around center

Add these 4 base lines to Lines

IF n == 4:

Start from (mid-1, mid+2)

Move Down by 3 using dpMove()

Move Left by 3 using dpMove()

ELSE:

Create Line5 from (mid-1 to mid+2, mid+2) and add to Lines

Start at (mid+2, mid+2), direction = 0, x = 4

WHILE x < n - 1:

IF dir == 0:

Move Left by x using dpMove()

dir = 1

ELSE IF dir == 1:

Move Up by x using dpMove()

dir = 2, x++

ELSE IF dir == 2:

Move Right by x using dpMove()

dir = 3

ELSE IF dir == 3:

Move Down by x using dpMove()

dir = 0, x++

IF x == n - 1:

IF n is odd:

Move Left, Up, Right by x using dpMove()

ELSE:

Move Right, Down, Left by x using dpMove()

FOR each line in Lines:

printLines(line)

Output total number of lines used

FUNCTION dpMove(direction, r, c, x, Lines, line, arr, visited):

Generate a unique memoization key from direction, position, x, and visited matrix

IF key exists in memo:

RETURN memo[key]

Append current point to line

FOR i in 0 to x:

Update r, c based on direction

Append arr[r][c] to line

Mark visited[r][c] = true

Append line to Lines

Memoize and return final (r, c)

## C++ code

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <sstream>

using namespace std;

void printLines(const vector<pair<int, int>>& line) {

for (const auto& point : line) {

cout << "(" << point.first << "," << point.second << ") -> ";

}

cout << endl;

}

string matrixToKey(const vector<vector<bool>>& visited) {

stringstream ss;

for (const auto& row : visited)

for (bool val : row)

ss << val;

return ss.str();

}

unordered\_map<string, pair<int, int>> memo;

pair<int, int> dpMove(string direction, int r, int c, int x, vector<vector<pair<int, int>>>& Lines, vector<pair<int, int>>& line, const vector<vector<pair<int, int>>>& arr, vector<vector<bool>>& visited) {

string key = direction + ":" + to\_string(r) + "," + to\_string(c) + "," + to\_string(x) + "," + matrixToKey(visited);

if (memo.find(key) != memo.end()) return memo[key];

line.push\_back(arr[r][c]);

for (int i = 0; i < x; i++) {

if (direction == "Left") c -= 1;

else if (direction == "Right") c += 1;

else if (direction == "Up") r -= 1;

else if (direction == "Down") r += 1;

line.push\_back(arr[r][c]);

visited[r][c] = true;

}

Lines.push\_back(line);

return memo[key] = { r, c };

}

void findPath(int n) {

vector<vector<pair<int, int>>> arr(n, vector<pair<int, int>>(n));

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

arr[i][j] = { i, j };

int mid = (n % 2 == 0) ? (n / 2 - 1) : (n / 2);

vector<vector<pair<int, int>>> Lines;

vector<vector<bool>> visited(n, vector<bool>(n, false));

vector<pair<int, int>> Line1 = { arr[mid + 1][mid - 1], arr[mid][mid], arr[mid - 1][mid + 1] };

vector<pair<int, int>> Line2 = { arr[mid - 1][mid + 1], arr[mid][mid + 1], arr[mid + 1][mid + 1] };

if (mid + 2 < n) Line2.push\_back(arr[mid + 2][mid + 1]);

vector<pair<int, int>> Line3;

if (mid + 2 < n) Line3.push\_back(arr[mid + 2][mid + 1]);

Line3.insert(Line3.end(), { arr[mid + 1][mid], arr[mid][mid - 1] });

if (mid - 2 >= 0) Line3.push\_back(arr[mid - 1][mid - 2]);

vector<pair<int, int>> Line4;

if (mid - 2 >= 0) Line4.push\_back(arr[mid - 1][mid - 2]);

Line4.insert(Line4.end(), { arr[mid - 1][mid - 1], arr[mid - 1][mid], arr[mid - 1][mid + 1] });

if (mid + 2 < n) Line4.push\_back(arr[mid - 1][mid + 2]);

Lines.push\_back(Line1);

Lines.push\_back(Line2);

Lines.push\_back(Line3);

Lines.push\_back(Line4);

if (n == 4) {

vector<pair<int, int>> line;

int r = mid - 1, c = mid + 2;

pair<int, int> result = dpMove("Down", r, c, 3, Lines, line, arr, visited);

r = result.first; c = result.second;

line.clear();

result = dpMove("Left", r, c, 3, Lines, line, arr, visited);

r = result.first; c = result.second;

}

else if (n > 4) {

vector<pair<int, int>> Line5;

for (int i = 0; i < 4; i++) Line5.push\_back(arr[mid - 1 + i][mid + 2]);

Lines.push\_back(Line5);

int r = mid + 2, c = mid + 2, x = 4, dir = 0;

while (x < n - 1) {

vector<pair<int, int>> line;

pair<int, int> result;

if (dir == 0) {

result = dpMove("Left", r, c, x, Lines, line, arr, visited);

r = result.first; c = result.second;

dir = 1;

}

else if (dir == 1) {

result = dpMove("Up", r, c, x, Lines, line, arr, visited);

r = result.first; c = result.second;

dir = 2;

x++;

}

else if (dir == 2) {

result = dpMove("Right", r, c, x, Lines, line, arr, visited);

r = result.first; c = result.second;

dir = 3;

}

else if (dir == 3) {

result = dpMove("Down", r, c, x, Lines, line, arr, visited);

r = result.first; c = result.second; dir = 0;

x++;

}

}if (x == n - 1) {

vector<pair<int, int>> li;

pair<int, int> result;

if (n % 2 != 0) {

result = dpMove("Left", r, c, x, Lines, li, arr, visited);

r = result.first; c = result.second;

li.clear();

result = dpMove("Up", r, c, x, Lines, li, arr, visited);

r = result.first; c = result.second;

li.clear();

result = dpMove("Right", r, c, x, Lines, li, arr, visited);

r = result.first; c = result.second;

}

else {

result = dpMove("Right", r, c, x, Lines, li, arr, visited);

r = result.first; c = result.second;

li.clear();

result = dpMove("Down", r, c, x, Lines, li, arr, visited);

r = result.first; c = result.second;

li.clear();

result = dpMove("Left", r, c, x, Lines, li, arr, visited);

r = result.first; c = result.second;

}

}

}

for (const auto& row : Lines)

printLines(row);

cout << "\nTotal Number of Lines: " << Lines.size() << endl;

}

int main() {

int n;

cout << "Enter the Order of the dots: ";

cin >> n;

findPath(n);

return 0;

}

## Complexity Analysis

**Time Complexity:**

* The time complexity of the algorithm depends on:
* The total number of recursive/memoized dpMove calls.
* Each call covers x steps in a direction and processes x points.
* The main loop runs until the number of drawn lines reaches 2n - 2.

Let’s estimate:

* The number of dpMove calls is approximately 2n - 2.
* Each dpMove processes O(n) points at most.
* The visited state matrix is encoded as a string in O(n^2) time (used in memoization key).

In the worst case: Time Complexity = O(n^3)

**Space Complexity:**

* Visited Matrix: O(n^2)
* Memoization Map: in worst case stores up to O(n^3) unique states (due to
* matrix-based keys).
* Lines Storage: each line has up to n points, and total lines = 2n - 2 so total space =O(n^2)

Total Space Complexity = O(n^3)

## Comparison with Another Technique

Table 6: Comparison

|  |  |  |
| --- | --- | --- |
| **Feature** | **Dynamic Programming Approach** | **Brute Force Search /**  **Backtracking** |
| **Efficiency** | **Uses memoization to avoid**  **redundant work** | **Explores all possibilities without**  **pruning** |
| **Scalability** | **Scales to large n** | **Not feasible for n > 6 due to**  **combinatorics** |
| **Correctness Guarantee** | **High if memoized states are**  **well managed** | **May not always terminate or find**  **optimal path** |
| **Implementation**  **Complexity** | **Moderate** | **High due to recursion,**  **backtracking, and pruning logic** |
| **Optimality** | **Can be adjusted to minimize**  **lines** | **May find optimal but at heavy**  **computation cost** |

**Pseudocode for Brute-force Backtracking:**

function solve(i, j, visited, currentLine, direction, lineCount):

if all points are visited:

if lineCount <= 2n - 2:

record solution

return

for each dir in [UP, DOWN, LEFT, RIGHT]:

if dir == direction:

next\_i, next\_j = move(i, j, dir)

if valid(next\_i, next\_j) and not visited[next\_i][next\_j]:

visited[next\_i][next\_j] = true

currentLine.append((next\_i, next\_j))

solve(next\_i, next\_j, visited, currentLine, dir, lineCount)

currentLine.pop()

visited[next\_i][next\_j] = false

else:

# Start a new line (change direction)

if lineCount + 1 < 2n - 1:

next\_i, next\_j = move(i, j, dir)

if valid(next\_i, next\_j) and not visited[next\_i][next\_j]:

visited[next\_i][next\_j] = true

newLine = [(i, j), (next\_i, next\_j)]

solve(next\_i, next\_j, visited, newLine, dir, lineCount + 1)

visited[next\_i][next\_j] = false

function main(n):

initialize visited[n][n] to false

for each starting point (i, j):

mark visited[i][j] = true

for direction in [UP, DOWN, LEFT, RIGHT]:

next\_i, next\_j = move(i, j, direction)

if valid(next\_i, next\_j):

visited[next\_i][next\_j] = true

currentLine = [(i, j), (next\_i, next\_j)]

solve(next\_i, next\_j, visited, currentLine, direction, 1)

visited[next\_i][next\_j] = false

visited[i][j] = false

In brute-force backtracking, we attempt all possible paths from the center, trying all directions at each step until we cover all required points or satisfy certain constraints.

**Time Complexity:**

From each cell, 4 directions are tried. Paths may have variable lengths up to n. The number of possible unique paths is exponential, especially if each cell is visited only once.

The worst-case time complexity is **O(4^(n^2))** , extremely inefficient for large n.

**Space Complexity:**

Visited matrix: O(n^2)

Recursion stack depth: up to O(n^2)

Path storage: O(n^2) for each recursive call

So the space complexity **is O(n^2).**

**So both are complexity is O(n^2)**

## Sample Output

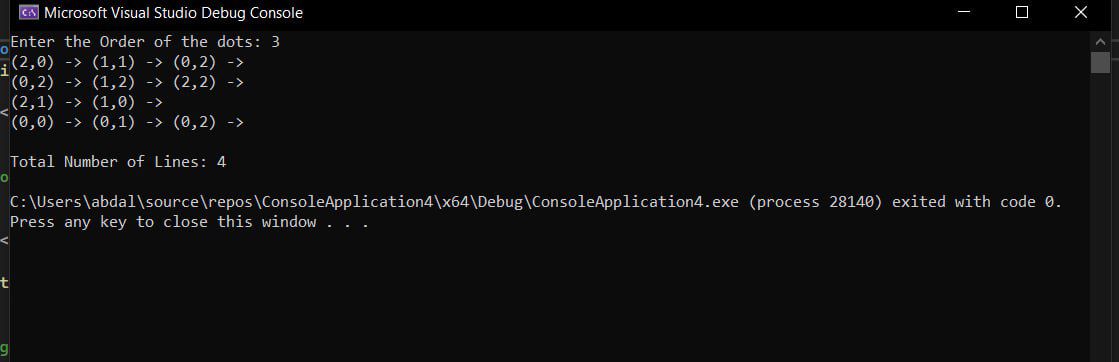


Figure 16: Sample 1



Figure 17: Sample 2

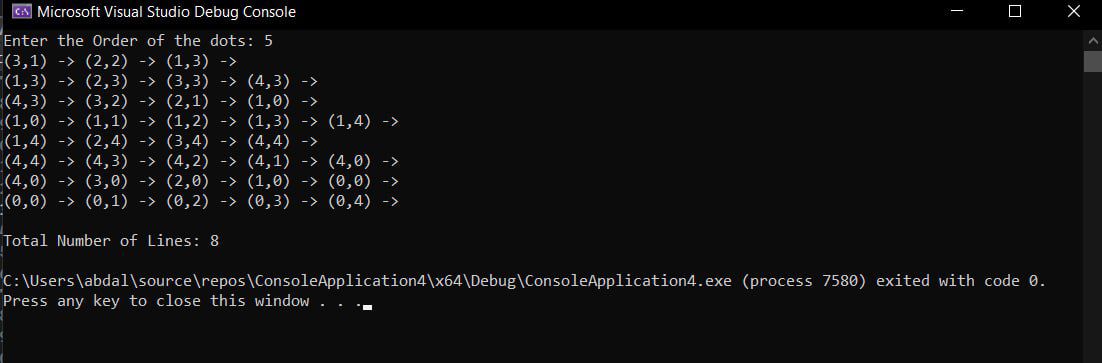


Figure 18: Sample 3

## Conclusion

This project implements an efficient Dynamic Programming algorithm to solve the classic lattice grid traversal problem. The algorithm constructs a continuous path using at most 2n - 2 straight lines to visit all n × n grid points, satisfying the constraints without retracing lines. It utilizes memoization, grid representation, and directional movement planning to reduce time and space complexity. Compared to brute-force or naive recursive methods, this solution is far more scalable and reliable, especially for larger grids.

The dynamic programming approach ensures:

* Efficiency through memoization
* Structured line-building logic
* Scalable performance
* Correctness in path coverage

This makes it suitable for both theoretical study and practical application in algorithmic graph traversal, robotics, and grid-based optimizations.

# Research task 1: Hamiltonian circuit problem

## Problem description

The Hamiltonian Circuit Problem is a classic problem in graph theory and computational complexity. It involves determining whether a given graph contains a Hamiltonian circuit, which is a closed loop that visits each vertex exactly once and returns to the starting point.

Formally, given a graph G=(V,E), the problem is to find a permutation of vertices v1, v2, ...,vn such that:

* {vi, vi+1} ∈ for all 1≤i<n
* {vn, v1}∈E, forming a cycle.

The Hamiltonian Circuit Problem is classified as NP-complete, meaning no known algorithm can solve all instances efficiently (in polynomial time), and it is as hard as the hardest problems in NP.

## Common algorithmic methods

### 1.Backtracking

This algorithm tries every possible path recursively and backtracks when a partial path cannot be extended.

**Pseudocode:**

procedure HamiltonianBacktrack(path, visited):

if length(path) == N:

if edge exists from path[N-1] to path[0]:

print "Hamiltonian Circuit Found:", path + [path[0]]

return

for vertex in 0 to N-1:

if not visited[vertex] and edge exists from path[-1] to vertex:

visited[vertex] = true

path.append(vertex)

HamiltonianBacktrack(path, visited)

visited[vertex] = false

path.pop()

**Complexity analysis:**

* There are (n−1)! possible permutations of the vertices (fixing the starting point to avoid circular duplicates). For each permutation, checking whether it forms a Hamiltonian cycle takes O(n) time.
* Time Complexity = O(n⋅(n−1)!) = O(n!)
* Space Complexity: O(n) (recursive call stack and path array of length n). A visited array of size n is maintained. The recursive depth is at most n.

### 2**. Held-Karp algorithm (Dynamic programming)**Top of Form

This is an optimization over brute force using memoization and bit masking to avoid recomputing subproblems.

**Pseudocode:**

procedure HeldKarp(graph):

n = number of vertices

dp = 2D array [2^n][n] filled with INF

dp[1][0] = 0 // starting from vertex 0

for mask from 1 to 2^n - 1:

for u in 0 to n-1:

if mask has u:

for v in 0 to n-1:

if v ≠ u and mask has v and edge exists(v, u):

dp[mask][u] = min(dp[mask][u], dp[mask ^ (1 << u)][v] + graph[v][u])

result = INF

for u in 1 to n-1:

if edge exists(u, 0):

result = min(result, dp[2^n - 1][u] + graph[u][0])

return result

**Complexity Analysis:**

Approach: Uses memoization over subsets of vertices and dynamic programming to avoid redundant work.

There are 2^n subsets of the vertex set. For each subset and for each possible last vertex u, the algorithm computes the minimum cost to reach that subset ending at u (hence O(n⋅2^n) states). For each such state, up to n transitions are considered (previous vertex).

Time Complexity: O(n⋅2^n⋅n) = O(n^2⋅2^n)  
Space Complexity: O(n⋅2^n) as DP table of size dp[2n][n] stores the shortest paths.

### 3. Genetic algorithm (GA)

Another method that can be used to solve the Hamiltonian Circuit Problem is a Genetic Algorithm, which is a type of heuristic inspired by how evolution works in nature. It's not guaranteed to always give the perfect answer, but it can find good or near-optimal solutions, especially when the graph is large and exact methods like backtracking or Held-Karp take too long.

**Pesudocode:**

Procedure GeneticHamiltonian(Graph, PopulationSize, Generations):

Population ← GenerateRandomPaths(Graph, PopulationSize)

For gen = 1 to Generations:

FitnessScores ← EvaluateFitness(Population, Graph)

NewPopulation ← []

While size(NewPopulation) < PopulationSize:

Parent1 ← SelectParent(FitnessScores)

Parent2 ← SelectParent(FitnessScores)

Child ← Crossover(Parent1, Parent2)

If RandomChance() < MutationRate:

Child ← Mutate(Child)

NewPopulation.Append(Child)

Population ← NewPopulation

BestPath ← GetBestPath(Population, Graph)

If IsHamiltonianCircuit(BestPath, Graph):

Return BestPath

Else:

Return "No valid Hamiltonian circuit found"

**Complexity analysis:**

Approach:

The Genetic algorithm does not try every possible path like brute-force methods. Instead, it works with a group (population) of paths and gradually improves them over generations using selection, crossover, and mutation. Each path is evaluated using a fitness function to decide how "good" it is. Over time, better paths are more likely to survive and reproduce, increasing the chance of finding a valid Hamiltonian circuit.

Unlike exact algorithms, the Genetic Algorithm does not guarantee an optimal solution, but it is usually much faster and works well for large graphs where exact solutions are too slow.

Time complexity:

* n = number of vertices in the graph
* P = population size
* G = number of generations
* Generating fitness for each path: O(n)
* Evaluating all paths per generation: O(P × n)

Time complexity overall: O(G × P × n)

Space complexity:

* Storing the population: O(P × n)
* Storing fitness scores and temporary paths: O(P)

Space complexity overall is: O(P × n)

# Research task 2: Partition problem

## Problem description

The Partition problem is a classic decision problem in computer science and combinatorics. It involves determining whether a given set of positive integers can be partitioned into two subsets such that the sums of the elements in both subsets are equal.

**Example:**

-Input: {1, 5, 11, 5}

-Output: True

-Explanation: The set can be partitioned as {1, 5, 5} and {11}, both having a sum of 11.

## Common algorithmic methods

### 1.Brute force approach

Pseudocode:

procedure PartitionBruteForce(arr, n, sum):

if sum == 0:

return True

if n == 0:

return False

if arr[n-1] <= sum:

include = PartitionBruteForce(arr, n-1, sum - arr[n-1])

exclude = PartitionBruteForce(arr, n-1, sum)

return include or exclude

else:

return PartitionBruteForce(arr, n-1, sum)

**Complexity analysis:**

* The algorithm considers every subset, leading to 2^n possible subsets.
* Time complexity: **O(2^n)**
* Space complexity: **O(n)** due to the recursive call stack.

### 2.Dynamic programming approach

Pseudocode:

procedure PartitionDP(arr, n):

totalSum = sum(arr)

if totalSum % 2 != 0:

return False

target = totalSum // 2

dp = [False] \* (target + 1)

dp[0] = True

for num in arr:

for j in range(target, num - 1, -1):

dp[j] = dp[j] or dp[j - num]

return dp[target]

**Complexity analysis:**

* The algorithm uses a 1D array of size target + 1 for memoization.
* Time Complexity: **O(n \* sum)**
* Space Complexity: **O(sum)**

# Research task 3: Graph coloring problem

## Problem description

The Graph Coloring Problem is a well-known combinatorial optimization problem that involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. The objective is to minimize the number of colors used.

Example:

-Input: Graph with vertices {A, B, C, D} and edges {(A, B), (A, C), (B, D), (C, D)}

-Output: 2 colors

-Explanation: The graph can be colored using two colors as {A, C} = Red and {B, D} = Blue.

## Common algorithmic methods

### 1.Backtracking approach

Pseudocode:

procedure GraphColoring(graph, colors, vertex):

if vertex == n:

print "Valid Coloring Found"

return True

for color in 1 to k:

if isSafe(graph, vertex, color):

colors[vertex] = color

if GraphColoring(graph, colors, vertex + 1):

return True

colors[vertex] = 0 # Backtrack

return False

**Complexity analysis:**

* Time Complexity: **O(k^n)** where k is the number of colors and n is the number of vertices.
* Space Complexity: **O(n)** to store color assignments.

### 2.Greedy approach

Pseudocode:

procedure GreedyColoring(graph, n):

colors = [-1] \* n

for vertex in 0 to n-1:

availableColors = [True] \* n

for neighbor in graph[vertex]:

if colors[neighbor] != -1:

availableColors[colors[neighbor]] = False

for color in range(n):

if availableColors[color]:

colors[vertex] = color

break

return colors

**Complexity analysis:**

* Time Complexity: **O(n^2)** in the worst case.
* Space Complexity: **O(n)** to store the color array.

# References

-Knuth, D. E. (1997). The Art of Computer Programming, Vol. 1. Addison-Wesley. Referenced for: Recursive algorithm design principles and analysis techniques.

-Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.

-Levitin, A., & Levitin, M. (2011). Algorithmic Puzzles. Oxford University Press.

-Russell, S., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach. Prentice Hall.

-GeeksforGeeks. Generalized Tower of Hanoi (4 Pegs) Problem.

-Applications of the 4-color theorem in tiling problems.

**Bottom of Form**