Design and Analysis of ALGORITHMS

Spring 2023
CESS - Junior
Faculty of Engineering, Ain Shams University

Team Members	IDs
Ahmed Tarek Aboelmakarem	20P6897
Fady Fady Fouad	20P7341
Kareem Wael Hasan	2001151
Karim Bassel Samir	20P6794
Matthew Sherif Shalaby	20P6785
Mina Fadi Mohsen	20P6996
Moisis George Mounir	20P8708
Omar Hisham Gouda	20P6484
Shady Emad Sabry	20P7239

Table of Contents

Task 1
- Assumptions
- Problem Description
- Approach
- Solution Steps
- Pseudocode
- Code Implementation
- Complexity Analysis
- Sample Output
- Comparison with an Alternative Algorithm
- Conclusion

Task 2
- Assumptions
- Problem Description
- Solution Steps
- Pseudocode
- Code Implementation
- Complexity Analysis
- Sample Output
- Comparison with an Alternative Algorithm
- Conclusion

Task 3
- Assumptions
- Problem Description
- Solution Steps
- Pseudocode
- Code Implementation
- Complexity Analysis
- Sample Output
- Comparison with an Alternative Algorithm
- Conclusion

Task 4
- Assumptions
- Problem Description
- Solution Steps
- Pseudocode
- Code Implementation
- Complexity Analysis

Sample Output
Comparison with an Alternative Algorithm
Conclusion

Task 5
Assumptions
Problem Description
Solution Steps
Pseudocode
Code Implementation
Complexity Analysis
Sample Output
Comparison with an Alternative Algorithm
Conclusion

Task 6
Assumptions
Problem Description
Solution Steps
Pseudocode
Code Implementation
Complexity Analysis
Sample Output
Comparison with an Alternative Algorithm
Conclusion

Task 7
Assumptions
Problem Description
Solution Steps
Pseudocode
Code Implementation
Complexity Analysis
Sample Output
Comparison with an Alternative Algorithm
Conclusion

References

Task 1

Assumptions

- The board is initially filled with -1 (indicating no tromino placed) except for the missing square which has value 0
- X-coordinate represent 2D-array's row, Y-coordinate represents 2D-array's column

Problem Description

Devise an algorithm for the following task: given a 2^n × 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 dynamic programming to solve this problem.

Approach

Before designing the recursive algorithm using a dynamic programming approach, let's define the coloring notation that will be used to solve and visualize the problem and the recursive and base cases which lead to the observation that is used in the dynamic programming technique for this problem.

Coloring Notation

using the following integers to represent the 3 colors in code and the corresponding colors for visualization

1: red

2 : green

3 : blue

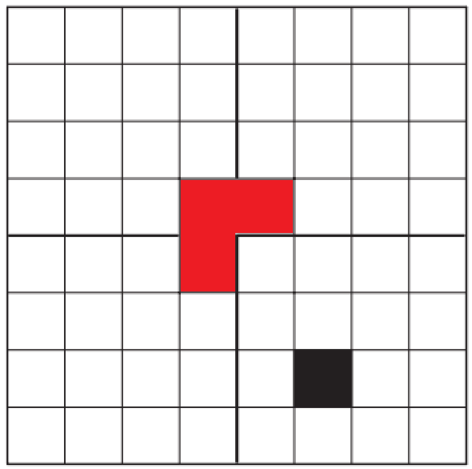
0 : black (denoting the missing square)

Recursive Case

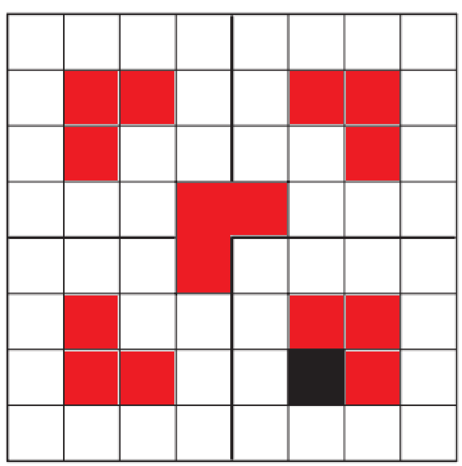
Let the problem (or any subproblem) of a board size 2^n × 2^n be defined as:

P { n, {xt, yt}, {xm, ym} }, where n is the problem size, {xt, yt} are the coordinates of the top left square in the subproblem's board, and {xm, ym} are the coordinates of the missing square.

Divide the problem P into 4 subproblems each of problem size n-1 (board size 2^(n-1) × 2^(n-1)). Since the problem P contains only 1 missing square that exists in one of the 4 subproblems, 3 out of the 4 center squares of Problem P that are not in the same subproblem as the original square will form a tromino and be colored with color 1 (red in visualization).



Since none of the 4 center squares are on the edge of any problem size n > 1, tiling all center trominoes of each subproblem (n > 1) with color 1 (red) will not result in a pair of trominoes that share an edge have the same color. While colors 2 (green) and 3 (blue) will be used to tile subproblems with n = 1 which will be the base case of the recursive algorithm.

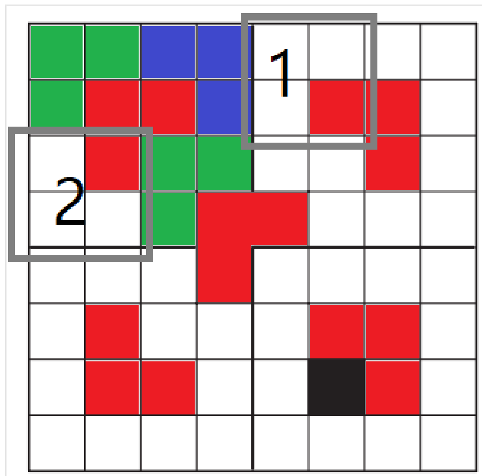


Base Case

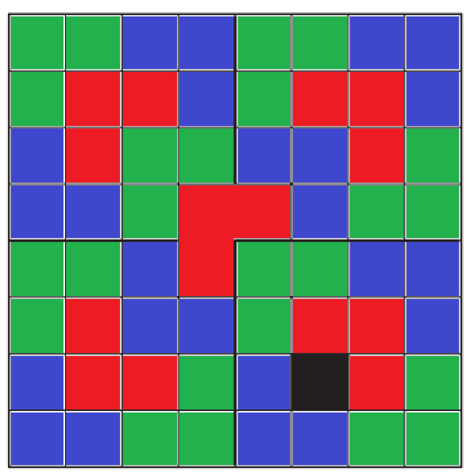
A subproblem of size n = 1 has a board of size 2^1 x 2^1 = 2 x 2, 4 squares with 1 missing square results in 3 squares that fit for placing 1 tromino of either color 2 or 3.

The color is decided by looking at the neighboring trominoes that are not subproblem center (red) and excluding the color that is used with them. Since the trominoes are placed from top left to bottom right, only neighboring trominoes on top and left (if exists) will be examined.

Note that in order to have no pair of trominoes that share an edge have the same color, neighboring trominoes on top and left have the same color but since not all trominoes have trominoes on their top, left checking is necessary.



In the previous figure, tromino 1 has no trominoes on its top, so it will examine the tromino on the left which is of color 3 (blue) so its color will be 2 (green). Tromino 2 will examine the tromino on its top which is of color 2 (green) so it will have color 3 (blue).



Observation

The tiling method mentioned above results in an interesting observation, all subproblems of size n = 2 (2^2 x 2^2 boards) are composed of 5 trominoes:

1 red tromino on the center

2 green trominoes on top left and bottom right

2 blue trominoes on top right and bottom left

Their orientation changes according to the location of the sub problem's missing square, but is the same for every subproblem with the same location of the missing square.

However, there can be only 5 possible locations for the missing square inside each subproblem of size n: top left, top right, bottom left, bottom right, and original missing square of the main problem. Therefore, **for each problem size 2 (for now) there are a maximum of 5 unique subproblems and all other subproblems are overlapping with them,** having the exact trominoes orientation and color.

Since a subproblem of size n = 3 with the same location for the missing square is divided into the same 4 subproblems of size n = 2 (according to the problem definition), the same observation applies for problem size n = 3. This serves as an inductive step if this observation is transformed into inductive proof, so **for each problem size n there are a maximum of 5 unique subproblems and all other subproblems are overlapping with them**

Dynamic Programming

It is not important to store the solution of a subproblem containing the original missing square of the main problem since it appears only once for each problem size n.

Therefore there are only 4 types of overlapping subproblems in each one; the missing square is in one of the corners. Denote them as follows:

Type 3: missing square is in top left

Type 2 : missing square is in bottom left

Type 1 : missing square is in top right

Type 0 : missing square is in bottom right

The memory used to store the results of overlapping subproblems will be in the form of 2D array : dp[n][type] = {xt, yt}

First dimension n refers to problem size (since each problem size have 5 unique subproblems)

Second dimension type refers to the problem type according to the location of the missing square.

Values stored in the memory are the coordinates of the top left square for the solved subproblem.

The **Recursive Step** is modified so that before it the subproblem type is defined and the memory is checked for a solution to a previous subproblem with the same size and type, if the solution exists then the board colors are copied from the subproblem which top left coordinates are stored in the memory to the current subproblem and no further recursion is made.

If no solution exists yet in the memory or the subproblem has the original missing square of the same subproblem then recursion is made like normal and the subproblem solution is stored in the memory to be copied later to overlapping subproblems.

Solution Steps

The problem (and each subproblem) will be one of three cases:

1. Base case: if problem size n = 1 then the subproblem is a 2*2 board with 1 missing square, a space for 1 tromino

    1.1. Decide the color for this tromino, this is done by looking in the neighboring solved subproblems and excluding their colors. Coloring is explained in detail in the Approach section.

    1.2. Place a tromino with the decided color in the non-missing squares.

2. Dynamic Programming Part: Check memory
    2.1. Define problem type as explained in detail in the Approach section.
    2.2. Check if a solution to an overlapping subproblem exists in memory
        2.2.1. If it exists, copy the colors values from the stored subproblem to the original one and stop recursion for this subproblem.
        2.2.2. If solution doesn't exist yet continue to step 3
3. Recursive Part: Divide the problem into 4 subproblems, for each subproblem repeat the following steps:
    3.1. Define top left coordinates for the subproblem
    3.2. Check if the missing square of the big problem exists in this subproblem
        3.2.1. If it is, mark its coordinates as the coordinates of the subproblem missing square as well
        3.2.2. If it isn't. Color the center square of the big problem which is inside the subproblem with color 1 (red) as part of the center tromino and mark its coordinates as the coordinates of the subproblem missing square
    3.3. Recursively call the function with problem size n - 1, top left coordinates defined in 3.1, and missing square coordinates defined in 3.2
4. After Recursive step: store subproblem solution (denoted by top left coordinates) if its missing square is not the original missing square of the main problem.

Pseudocode

ALGORITHM makeTrominoes(board[0..size-1, 0..size-1], dp[0..n][0..3], n, missing_x, missing_y, topLeft_x, topLeft_y)

```
// Input: 2d array Total board of size (size*size)
//      array of top left coordinates for solverd subproblems (Dynamic Programming)
//      problem size: the exponent of the board's edge size n
//      (x,y) coordinates of the missing square
//      (x,y) coordinates of the top left square of the sub-board

edge_size ← 2^n
// base case: 2^1 * 2^1 board = 1 tromino colored with the parameter color
if n = 1 then
        // Define the color of this tromino
    color ← defineColor(board, topLeft_x, topLeft_y)

    for i ← topLeft_x to topLeft_x + edge_size do
            for j ← topLeft_y to topLeft_y + edge_size do
                    if i ≠ missing_x or j ≠ missing_y
                        board[i][j] ← color
    return

// Dynamic Programming Part
// Determine subproblem type
type ← - 1
if board[missing_x][missing_y] ≠ 0 then
    type ← 0
        if topLeft_x = missing_x then
                type ←  type + 1
        if topLeft_y = missing_y then
```

```
            type ← type + 2

// Check if the solution to this subproblem exists in memory
if type ≥ 0 and dp[n][type] ≠ {-1, -1} then
        // Solution exists, copy squares colors and stop recursion
    {original_topLeft_x, original_topLeft_y} ← dp[n][type]

        for i ← topLeft_x to topLeft_x + edge_size do
            for j ← topLeft_y to topLeft_y + edge_size do
                original_x ← original_topLeft_x + i - topLeft_x
                original_y ← original_topLeft_y + j - topLeft_y
                board[i][j] ← board[original_x][original_y]
        return

// 4 subproblems:

// 1. Top Left
subproblem_topLeft_x ← topLeft_x        // row coordinate of sub-board topLeft
subproblem_topLeft_y ← topLeft_y        // column coordinate of sub-board topLeft

if missing_x < topLeft_x + edge_size / 2 and missing_y < topLeft_y + edge_size / 2 then
        // missing square is in this subproblem
        subproblem_missing_x ← missing_x
        subproblem_missing_y ← missing_y
else
        // missing square is NOT in this subproblem, color the center square in this
        // sub-board side and denote it as the missing square in this subproblem
        subproblem_missing_x ← topLeft_x + edge_size / 2 - 1
        subproblem_missing_y ← topLeft_y + edge_size / 2 - 1
```

```
        board[subproblem_missing_x][subproblem_missing_y] ← 1

makeTrominoes(board, dp, n - 1, subproblem_missing_x, subproblem_missing_y
      ,subproblem_topLeft_x, subproblem_topLeft_y)

// 2. Top Right
// row coordinate of sub-board topLeft is the same
subproblem_topLeft_y ← topLeft_y + edge_size / 2       // column coordinate of sub-board
                                                       // topLeft

if missing_x < topLeft_x + edge_size / 2 and missing_y ≥ topLeft_y + edge_size / 2 then
        // missing square is in this subproblem
        subproblem_missing_x ← missing_x
        subproblem_missing_y ← missing_y
else
        // missing square is NOT in this subproblem, color the center square in this
        // sub-board side and denote it as the missing square in this subproblem
        subproblem_missing_x ← topLeft_x + edge_size / 2 - 1
        subproblem_missing_y ← topLeft_y + edge_size / 2
        board[subproblem_missing_x][subproblem_missing_y] ← 1

makeTrominoes(board, dp, n - 1, subproblem_missing_x, subproblem_missing_y
                , subproblem_topLeft_x, subproblem_topLeft_y)

// 3. Bottom Left
subproblem_topLeft_x ← topLeft_x + edge_size / 2 // row coordinate of sub-board topLeft
subproblem_topLeft_y ← topLeft_y         // column coordinate of sub-board topLeft
```

```
if missing_x ≥ topLeft_x + edge_size / 2 and missing_y < topLeft_y + edge_size / 2 then
        // missing square is in this subproblem
        subproblem_missing_x ← missing_x
        subproblem_missing_y ← missing_y
else
        // missing square is NOT in this subproblem, color the center square in this
        // sub-board side and denote it as the missing square in this subproblem
        subproblem_missing_x ← topLeft_x + edge_size / 2
        subproblem_missing_y ← topLeft_y + edge_size / 2 - 1
        board[subproblem_missing_x][subproblem_missing_y] ← 1

makeTrominoes(board, dp, n - 1, subproblem_missing_x, subproblem_missing_y
                , subproblem_topLeft_x, subproblem_topLeft_y)

// 4. Bottom Left
// row coordinate of sub-board topLeft is the same
subproblem_topLeft_y ← topLeft_y + edge_size / 2                // column coordinate of
                                                                // sub-board topLeft

if missing_x ≥ topLeft_x + edge_size / 2 and missing_y ≥ topLeft_y + edge_size / 2 then
        // missing square is in this subproblem
        subproblem_missing_x ← missing_x
        subproblem_missing_y ← missing_y
else
        // missing square is NOT in this subproblem, color the center square in this
        // sub-board side and denote it as the missing square in this subproblem
        subproblem_missing_x ← topLeft_x + edge_size / 2
        subproblem_missing_y ← topLeft_y + edge_size / 2
```

```
        board[subproblem_missing_x][subproblem_missing_y] ← 1

  makeTrominoes(board, dp, n - 1, subproblem_missing_x, subproblem_missing_y
                , subproblem_topLeft_x, subproblem_topLeft_y)

  // Store solution in the memory
  if type ≥ 0 Then
        // not original missing square, store this subproblem's top left coordinates
        dp[n][type] ← {topLeft_x, topLeft_y}
```

Helper function defineColor pseudocode:

```
procedure defineColor(board[0..size-1, 0..size-1], topLeft_x, topLeft_y)
  // Check if there is a tromino with color = 2 on top
  if topLeft_x > 0 and (board[topLeft_x - 1][topLeft_y] = 2
                        or board[topLeft_x - 1][topLeft_y + 1] = 2) Then
    return 3

  // Check if there is a tromino with color = 2 on left
  if topLeft_y > 0 and (board[topLeft_x][topLeft_y - 1] = 2
                        or board[topLeft_x + 1][topLeft_y - 1] = 2) Then
    return 3

  // No color neighboring tromino with color 2 (green), tile with it
  return 2
```

Code Implementation

```
#include<iostream>

using namespace std;

int power(int n) {
	// calculate 2^n using my own implementation
	int ans = 1;
	while (n--) {
		ans *= 2;
	}
	return ans;
}

int defineColor(int** board, int topLeft_x, int topLeft_y) {
	// Check if there is a tromino with color = 2 on top
	if (topLeft_x > 0 && (board[topLeft_x-1][topLeft_y] == 2 ||
board[topLeft_x - 1][topLeft_y + 1] == 2)) {
		return 3;
	}

	// Check if there is a tromino with color = 2 on left
	if (topLeft_y > 0 && (board[topLeft_x][topLeft_y - 1] == 2 ||
board[topLeft_x + 1][topLeft_y - 1] == 2)) {
		return 3;
	}

	// No color neighboring tromino with color 2 (green), tile with it
	return 2;
}

void makeTrominoes(int** board, pair<int, int>** dp, int n, int missing_x,
int missing_y, int topLeft_x = 0, int topLeft_y = 0) {
	int edge_size = power(n);
	int subproblem_missing_x, subproblem_missing_y, subproblem_topLeft_x,
subproblem_topLeft_y, color;

	// base case: 2^1 * 2^1 board = 1 tromino colored by checking
previously colored trominoes
	if (n == 1) {
		color = defineColor(board, topLeft_x, topLeft_y);
```

```
        for (int i = topLeft_x; i < topLeft_x + edge_size; i++) {
            for (int j = topLeft_y; j < topLeft_y + edge_size; j++) {
                if (i != missing_x || j != missing_y) {
                    board[i][j] = color;
                }
            }
        }
        return;
    }

    // Check if the solution exists in the memory

    // Determine subproblem type
    int type = -1;
    if (board[missing_x][missing_y] != 0) {
        type = 0;
        if (topLeft_x == missing_x) type += 1;
        if (topLeft_y == missing_y) type += 2;
    }


    // Check if the solution to this subproblem exists in memory
    if (type >= 0 && dp[n][type].first != -1) {
        // Solution exists, copy squares colors and stop recursion
        int original_topLeft_x = dp[n][type].first;
        int original_topLeft_y = dp[n][type].second;
        int original_x, original_y;
        for (int i = topLeft_x; i < topLeft_x + edge_size; i++) {
            for (int j = topLeft_y; j < topLeft_y + edge_size; j++) {
                original_x = original_topLeft_x + i - topLeft_x;
                original_y = original_topLeft_y + j - topLeft_y;
                board[i][j] = board[original_x][original_y];
            }
        }
        return;
    }

    // 4 subproblems:

    // 1. Top Left
    subproblem_topLeft_x = topLeft_x;
    subproblem_topLeft_y = topLeft_y;
```

```
	if (missing_x < topLeft_x + edge_size / 2 && missing_y < topLeft_y + edge_size / 2) {
		subproblem_missing_x = missing_x;
		subproblem_missing_y = missing_y;
	}
	else {
		subproblem_missing_x = topLeft_x + edge_size / 2 - 1;
		subproblem_missing_y = topLeft_y + edge_size / 2 - 1;
		board[subproblem_missing_x][subproblem_missing_y] = 1;
	}
	makeTrominoes(board, dp, n - 1, subproblem_missing_x, subproblem_missing_y, subproblem_topLeft_x, subproblem_topLeft_y);

	// 2. Top Right
	subproblem_topLeft_y = topLeft_y + edge_size / 2;
	if (missing_x < topLeft_x + edge_size / 2 && missing_y >= topLeft_y + edge_size / 2) {
		subproblem_missing_x = missing_x;
		subproblem_missing_y = missing_y;
	}
	else {
		subproblem_missing_x = topLeft_x + edge_size / 2 - 1;
		subproblem_missing_y = topLeft_y + edge_size / 2;
		board[subproblem_missing_x][subproblem_missing_y] = 1;
	}
	makeTrominoes(board, dp, n - 1, subproblem_missing_x, subproblem_missing_y, subproblem_topLeft_x, subproblem_topLeft_y);

	// 3. Bottom Left
	subproblem_topLeft_x = topLeft_x + edge_size / 2;
	subproblem_topLeft_y = topLeft_y;
	if (missing_x >= topLeft_x + edge_size / 2 && missing_y < topLeft_y + edge_size / 2) {
		subproblem_missing_x = missing_x;
		subproblem_missing_y = missing_y;
	}
	else {
		subproblem_missing_x = topLeft_x + edge_size / 2;
		subproblem_missing_y = topLeft_y + edge_size / 2 - 1;
		board[subproblem_missing_x][subproblem_missing_y] = 1;
	}
	makeTrominoes(board, dp, n - 1, subproblem_missing_x, subproblem_missing_y, subproblem_topLeft_x, subproblem_topLeft_y);
```

```
        // 4. Bottom Right
        subproblem_topLeft_y = topLeft_y + edge_size / 2;
        if (missing_x >= topLeft_x + edge_size / 2 && missing_y >= topLeft_y
+ edge_size / 2) {
                subproblem_missing_x = missing_x;
                subproblem_missing_y = missing_y;
        }
        else {
                subproblem_missing_x = topLeft_x + edge_size / 2;
                subproblem_missing_y = topLeft_y + edge_size / 2;
                board[subproblem_missing_x][subproblem_missing_y] = 1;
        }
        makeTrominoes(board, dp, n - 1, subproblem_missing_x,
subproblem_missing_y, subproblem_topLeft_x, subproblem_topLeft_y);

        // Store solution in the memory
        if (type >= 0) {
                dp[n][type] = make_pair(topLeft_x, topLeft_y);
        }

}

int main() {
        int n, missing_x, missing_y, edge_size;

        cout << "The board will be of size 2^n * 2^n\n"
                << "Enter n : ";
        cin >> n;
        edge_size = power(n);
        // edge_size = 2^n

        cout << "\nTop left cell is at coordinate (0,0)\n"
                << "Enter coordinates of the missing square : ";
        cin >> missing_x >> missing_y;

        int** board = new int* [edge_size];
        for (int i = 0; i < edge_size; i++) {
                board[i] = new int[edge_size];
                for (int j = 0; j < edge_size; j++) {
                        board[i][j] = -1;
        // Not colored yet
                }
```

```
	}
	board[missing_x][missing_y] = 0;                                        //
Indicate missing square

	pair<int, int> ** dp = new pair<int, int>* [n+1];
	for (int i = 0; i < n + 1; i++) {
		dp[i] = new pair<int, int> [4];
		for (int j = 0; j < 4; j++) {
			dp[i][j] = make_pair(-1, -1);                           //
Initial value to indicate no solution exists in memory
		}
	}

	makeTrominoes(board, dp, n, missing_x, missing_y, 0, 0);    // 0, 0
are topLeft coordinates of the board (optional parameters in C++)

	cout << "\nThe board:\n";
	for (int i = 0; i < edge_size; i++) {
		for (int j = 0; j < edge_size; j++) {
			cout << board[i][j] << " ";
		}
		cout << endl;
	}

	cout << "\nThe board visualized: \n";
	for (int i = 0; i < edge_size; i++) {
		for (int j = 0; j < edge_size; j++) {
			if (board[i][j] == 1)
				cout << "\033[91mO\033[0m";
			else if (board[i][j] == 2)
				cout << "\033[92mO\033[0m";
			else if (board[i][j] == 3)
				cout << "\033[94mO\033[0m";
			else
				cout << "\033[30mO\033[0m";
		}
		cout << endl;
	}

	// Delete Dynamically allocated memory
	for (int i = 0; i < edge_size; i++) {
		delete[] board[i];
	}
```

```
        delete[] board;
        for (int i = 0; i < n; i++) {
                delete[] dp[i];
        }
        delete[] dp;
}
```

Complexity Analysis

The basic operation is assignment to an element in the 2^n*2^n board.

Analysis could also be done considering comparisons to the coordinates of the missing square as a basic operation, this will reveal the same results.

In one function call there are either solution using recursion or solution using dynamic programming memoization:

In recursion solution there are 4 recursive calls + 3 assignments (the tromino placed in the center)

$T(2^n) = 4*T(2^{(n-1)}) + 3,$ $n > 1,$ $T(2^1) = 3$

Using Backward Substitution:

$T(2^n) = 4*T(2^{n-1}) + 3 = 4(4*T(2^{n-2}) + 3) + 3 = 4(4(4*T(2^{n-3}) + 3) + 3) + 3$

$= 4^k\, T(2^{n-k}) + 3\Sigma^{k-1}_{j=0}\, 4^j = 4^k\, T(2^{n-k}) + 3(4^k-1)/3 = 4^k * (1 + T(2^{n-k})) - 1$

At k = n-1

$T(2^n) = 4^{n-1} * (1 + T(2^1)) - 1 = 4^{n-1} * (1 + 3) - 1 = 4^n - 1 \in \mathbf{O(4\text{^}n)}$

Another solution in terms of board's edge size: Denote e as

the edge size of the board: $e = 2^n$ $T(e) = 4*T(e/2) + 3$

Using Master Theorem: $a = 4$, $b = 2$, $d = 0 \Rightarrow b^d = 2^0 = 1$ Since $a > b^d$, Then

$T(e) \in O(e^{\log_2 4})$

$T(e) \in$ **O(e^2)**

In Dynamic Programming memoization solution: there are 6 assignments and 2 nested loops of size edge_size e ($e = 2^n$) inside them 3 assignments

$$T(2^n) = 6 + \Sigma^{2^n-1}_{i=0} \Sigma^{2^n-1}_{j=0} 3 = 6 + 3 \Sigma^{2^n-1}_{i=0} (2^n - 1 + 0 + 1)$$

$$= 6 + 3 \Sigma^{2^n-1}_{i=0} 2^n = 6 + 3 * 2^n * (2^n - 1 + 0 + 1) = 6 + 3 * 2^n * 2^n$$

$$= 6 + 3 * (2^n)^2 = 6 + 3 * (2^2)^n = 6 + 3 * 4^n \in O(4n)$$

This shows that either the subproblem is new and needs to be solved recursively or it is already memoized and can get its solution using dynamic programming technique, Time complexity doesn't change

In terms of Input size n: $T(n) \in O(4^n)$

In terms of board's edge size: $T(e) \in O(e^2)$

Sample Output

n = 2, missing square at (1,1)

```
The board will be of size 2^n * 2^n
Enter n : 2

Top left cell is at coordinate (0,0)
Enter coordinates of the missing square : 1 1

The board:
2 2 3 3
2 0 1 3
3 1 1 2
3 3 2 2

The board visualized:
0000
0 00
0000
0000
```

n = 3, missing square at (6, 5)

```
The board will be of size 2^n * 2^n
Enter n : 3

Top left cell is at coordinate (0,0)
Enter coordinates of the missing square : 6 5

The board:
2 2 3 3 2 2 3 3
2 1 1 3 2 1 1 3
3 1 2 2 3 3 1 2
3 3 2 1 1 3 2 2
2 2 3 1 2 2 3 3
2 1 3 3 2 1 1 3
3 1 1 2 3 0 1 2
3 3 2 2 3 3 2 2

The board visualized:
00000000
00000000
00000000
00000000
00000000
00000000
00000 00
00000000
```

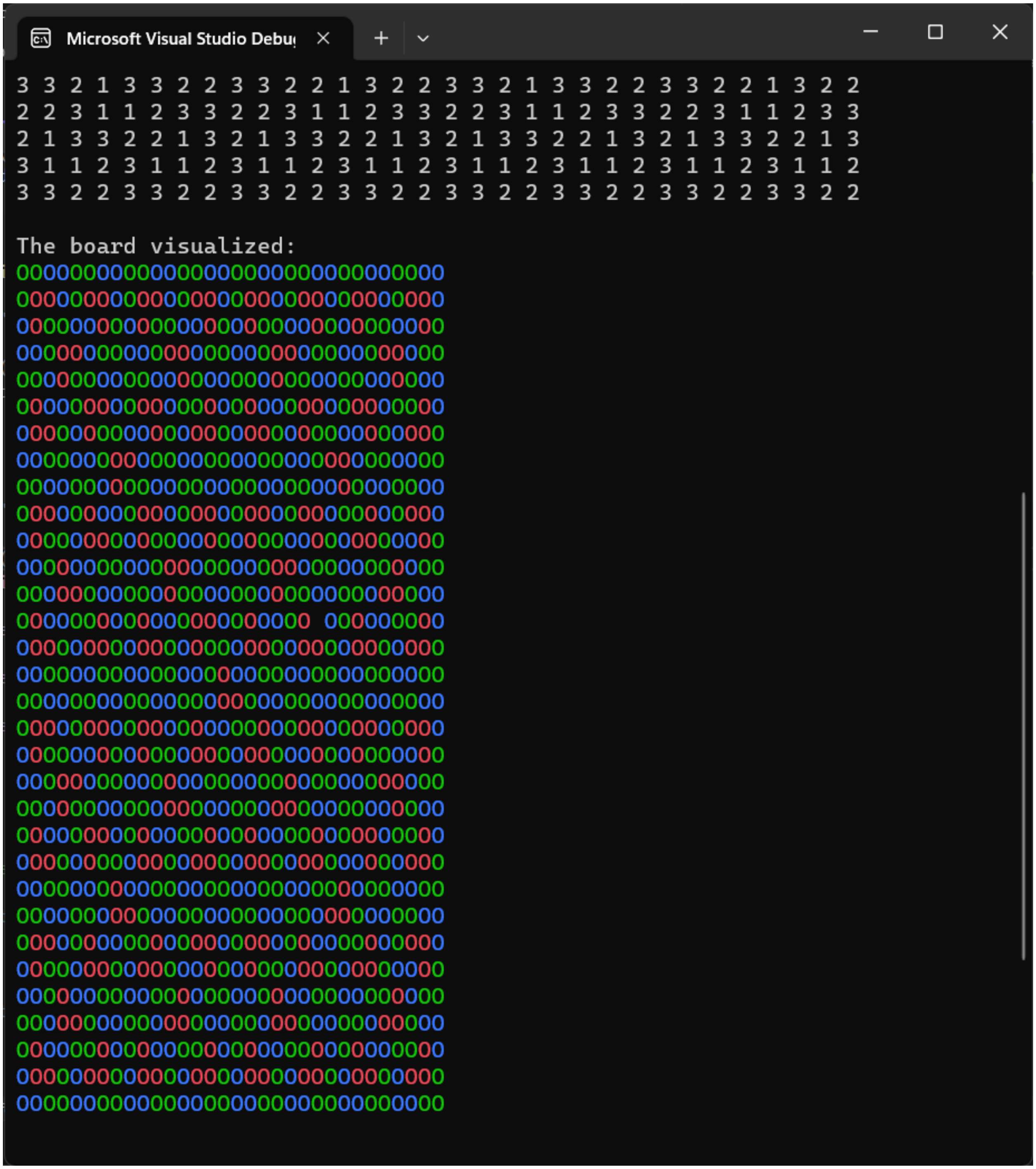
n = 4, missing square at (0,0)

Microsoft Visual Studio Debu

```
The board will be of size 2^n * 2^n
Enter n : 4

Top left cell is at coordinate (0,0)
Enter coordinates of the missing square : 0 0

The board:
0 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3
2 2 1 3 2 1 1 3 2 1 1 3 2 1 1 3
3 1 1 2 3 3 1 2 3 1 2 2 3 3 1 2
3 3 2 2 1 3 2 2 3 3 2 1 1 3 2 2
2 2 3 1 1 2 3 3 2 2 3 3 1 2 3 3
2 1 3 3 2 2 1 3 2 1 1 3 2 2 1 3
3 1 1 2 3 1 1 2 3 3 1 2 3 1 1 2
3 3 2 2 3 3 2 2 1 3 2 2 3 3 2 2
2 2 3 3 2 2 3 1 1 2 3 3 2 2 3 3
2 1 1 3 2 1 3 3 2 2 1 3 2 1 1 3
3 1 2 2 3 1 1 2 3 1 1 2 3 3 1 2
3 3 2 1 3 3 2 2 3 3 2 2 1 3 2 2
2 2 3 1 1 2 3 3 2 2 3 1 1 2 3 3
2 1 3 3 2 2 1 3 2 1 3 3 2 2 1 3
3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2
3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2

The board visualized:
 000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

n = 5, missing square at (13, 22)

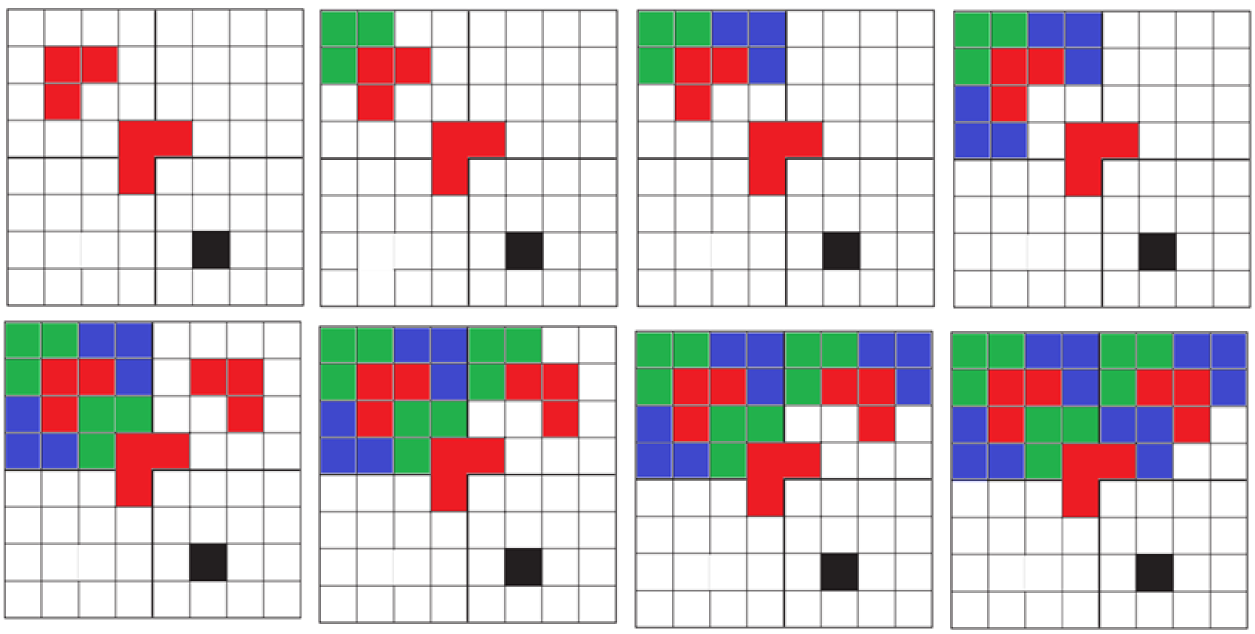
```
Microsoft Visual Studio Debu

The board will be of size 2^n * 2^n
Enter n : 5

Top left cell is at coordinate (0,0)
Enter coordinates of the missing square : 13 22

The board:
2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3
2 1 1 3 2 1 1 3 2 1 1 3 2 1 1 3 2 1 1 3 2 1 1 3 2 1 1 3 2 1 1 3
3 1 2 2 3 3 1 2 3 1 2 2 3 3 1 2 3 1 2 2 3 3 1 2 3 1 2 2 3 3 1 2
3 3 2 1 1 3 2 2 3 3 2 1 1 3 2 2 3 3 2 1 1 3 2 2 3 3 2 1 1 3 2 2
2 2 3 1 2 2 3 3 2 2 3 3 1 2 3 3 2 2 3 1 2 2 3 3 2 2 3 3 1 2 3 3
2 1 3 3 2 1 1 3 2 1 1 3 2 2 1 3 2 1 3 3 2 1 1 3 2 1 1 3 2 2 1 3
3 1 1 2 3 1 2 2 3 3 1 2 3 1 1 2 3 1 1 2 3 1 2 2 3 3 1 2 3 1 1 2
3 3 2 2 3 3 2 1 1 3 2 2 3 3 2 2 3 3 2 2 3 3 2 1 1 3 2 2 3 3 2 2
2 2 3 3 2 2 3 1 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 1 2 3 3 2 2 3 3
2 1 1 3 2 1 3 3 2 1 1 3 2 1 1 3 2 1 1 3 2 1 1 3 2 2 1 3 2 1 1 3
3 1 2 2 3 1 1 2 3 1 2 2 3 3 1 2 3 1 2 2 3 3 1 2 3 1 1 2 3 3 1 2
3 3 2 1 3 3 2 2 3 3 2 1 1 3 2 2 3 3 2 1 1 3 2 2 3 3 2 2 1 3 2 2
2 2 3 1 1 2 3 3 2 2 3 1 2 2 3 3 2 2 3 1 2 2 3 3 2 2 3 1 1 2 3 3
2 1 3 3 2 2 1 3 2 1 3 3 2 1 1 3 2 1 3 3 2 1 0 3 2 1 3 3 2 2 1 3
3 1 1 2 3 1 1 2 3 1 1 2 3 1 2 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2
3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 1 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2
2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 1 1 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3
2 1 1 3 2 1 1 3 2 1 1 3 2 1 3 3 2 2 1 3 2 1 1 3 2 1 1 3 2 1 1 3
3 1 2 2 3 3 1 2 3 1 2 2 3 1 1 2 3 1 1 2 3 3 1 2 3 1 2 2 3 3 1 2
3 3 2 1 1 3 2 2 3 3 2 1 3 3 2 2 3 3 2 2 1 3 2 2 3 3 2 1 1 3 2 2
2 2 3 1 2 2 3 3 2 2 3 1 1 2 3 3 2 2 3 1 1 2 3 3 2 2 3 3 1 2 3 3
2 1 3 3 2 1 1 3 2 1 3 3 2 2 1 3 2 1 3 3 2 2 1 3 2 1 1 3 2 2 1 3
3 1 1 2 3 1 2 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2 3 3 1 2 3 1 1 2
3 3 2 2 3 3 2 1 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 1 3 2 2 3 3 2 2
2 2 3 3 2 2 3 1 1 2 3 3 2 2 3 3 2 2 3 3 2 2 3 1 1 2 3 3 2 2 3 3
2 1 1 3 2 1 3 3 2 2 1 3 2 1 1 3 2 1 1 3 2 1 3 3 2 2 1 3 2 1 1 3
3 1 2 2 3 1 1 2 3 1 1 2 3 3 1 2 3 1 2 2 3 1 1 2 3 1 1 2 3 3 1 2
3 3 2 1 3 3 2 2 3 3 2 2 1 3 2 2 3 3 2 1 3 3 2 2 3 3 2 2 1 3 2 2
2 2 3 1 1 2 3 3 2 2 3 1 1 2 3 3 2 2 3 1 1 2 3 3 2 2 3 1 1 2 3 3
2 1 3 3 2 2 1 3 2 1 3 3 2 2 1 3 2 1 3 3 2 2 1 3 2 1 3 3 2 2 1 3
3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2
3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2
```


Microsoft Visual Studio Debu
3 3 2 1 3 3 2 2 3 3 2 2 1 3 2 2 3 3 2 1 3 3 2 2 3 3 2 2 1 3 2 2
2 2 3 1 1 2 3 3 2 2 3 1 1 2 3 3 2 2 3 1 1 2 3 3 2 2 3 1 1 2 3 3
2 1 3 3 2 2 1 3 2 1 3 3 2 2 1 3 2 1 3 3 2 2 1 3 2 1 3 3 2 2 1 3
3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2 3 1 1 2
3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2 3 3 2 2
The board visualized:

Comparison with an Alternative Algorithm

An alternative algorithm using **Divide and Conquer technique** will follow the same steps of the original algorithm designed in this report except it has no memoization due to being not dynamic programming, which means all subproblems now are considered unique and solved every time recursively.

For the base case, to completely avoid using information from previously solved subproblems, coloring trominoes will no longer depend on neighboring colors. Instead, another observation is made:



A pattern is noticed: first tromino is colored with color 2 (for example) then color is alternated every 2 base case trominoes, this can be implemented using binary representation of a variable *count* where the second rightmost bit alternates every 2 increments.

For the Recursive step, if the base case is not true recursion will be made at once in the same way as the main algorithm instead of looking up in the memory first, after it there will also be no storing in memory.

Pseudocode

ALGORITHM makeTrominoesDandC(board[0..size-1, 0..size-1], n, missing_x, missing_y, topLeft_x, topLeft_y)

```
// Input: 2d array Total board of size (size*size)
//       the exponent of the board's edge size n
//       (x,y) coordinates of the missing square
//       (x,y) coordinates of the top left square of the sub-board

edge_size ← 2^n
// base case: 2^1 * 2^1 board = 1 tromino colored with the parameter color
if n = 1 then
        // Switch coloring every two trominoes
    count ← count + 1       // Count of base case trominoes (can be global or static variable)
        if count & 2 = 1 then                    // second rightmost bit of count
            color = 3
        else
            color = 2

    for i ← topLeft_x to topLeft_x + edge_size do
            for j ← topLeft_y to topLeft_y + edge_size do
                    if i ≠ missing_x or j ≠ missing_y
                        board[i][j] ← color
    return

// 4 subproblems:

// 1. Top Left
```

```
subproblem_topLeft_x ← topLeft_x          // row coordinate of sub-board topLeft
subproblem_topLeft_y ← topLeft_y          // column coordinate of sub-board topLeft

if missing_x < topLeft_x + edge_size / 2 and missing_y < topLeft_y + edge_size / 2 then
    // missing square is in this subproblem
    subproblem_missing_x ← missing_x
    subproblem_missing_y ← missing_y
else
    // missing square is NOT in this subproblem, color the center square in this
    // sub-board side and denote it as the missing square in this subproblem
    subproblem_missing_x ← topLeft_x + edge_size / 2 - 1
    subproblem_missing_y ← topLeft_y + edge_size / 2 - 1
    board[subproblem_missing_x][subproblem_missing_y] ← 1

makeTrominoes(board, n - 1, subproblem_missing_x, subproblem_missing_y
   ,subproblem_topLeft_x, subproblem_topLeft_y)

// 2. Top Right
// row coordinate of sub-board topLeft is the same
subproblem_topLeft_y ← topLeft_y + edge_size / 2          // column coordinate of sub-board
                                                          // topLeft

if missing_x < topLeft_x + edge_size / 2 and missing_y ≥ topLeft_y + edge_size / 2 then
    // missing square is in this subproblem
    subproblem_missing_x ← missing_x
    subproblem_missing_y ← missing_y
else
    // missing square is NOT in this subproblem, color the center square in this
    // sub-board side and denote it as the missing square in this subproblem
```

```
        subproblem_missing_x ← topLeft_x + edge_size / 2 - 1
        subproblem_missing_y ← topLeft_y + edge_size / 2
        board[subproblem_missing_x][subproblem_missing_y] ← 1

makeTrominoes(board, n - 1, subproblem_missing_x, subproblem_missing_y
                , subproblem_topLeft_x, subproblem_topLeft_y)

// 3. Bottom Left
subproblem_topLeft_x ← topLeft_x + edge_size / 2 // row coordinate of sub-board topLeft
subproblem_topLeft_y ← topLeft_y                  // column coordinate of sub-board topLeft

if missing_x ≥ topLeft_x + edge_size / 2 and missing_y < topLeft_y + edge_size / 2 then
        // missing square is in this subproblem
        subproblem_missing_x ← missing_x
        subproblem_missing_y ← missing_y
else
        // missing square is NOT in this subproblem, color the center square in this
        // sub-board side and denote it as the missing square in this subproblem
        subproblem_missing_x ← topLeft_x + edge_size / 2
        subproblem_missing_y ← topLeft_y + edge_size / 2 - 1
        board[subproblem_missing_x][subproblem_missing_y] ← 1

makeTrominoes(board, n - 1, subproblem_missing_x, subproblem_missing_y
                , subproblem_topLeft_x, subproblem_topLeft_y)

// 4. Bottom Left
// row coordinate of sub-board topLeft is the same
subproblem_topLeft_y ← topLeft_y + edge_size / 2        // column coordinate of sub-board
                                                        // topLeft
```

```
if missing_x ≥ topLeft_x + edge_size / 2 and missing_y ≥ topLeft_y + edge_size / 2 then
    // missing square is in this subproblem
    subproblem_missing_x ← missing_x
    subproblem_missing_y ← missing_y
else
    // missing square is NOT in this subproblem, color the center square in this
    // sub-board side and denote it as the missing square in this subproblem
    subproblem_missing_x ← topLeft_x + edge_size / 2
    subproblem_missing_y ← topLeft_y + edge_size / 2
    board[subproblem_missing_x][subproblem_missing_y] ← 1

makeTrominoes(board, n - 1, subproblem_missing_x, subproblem_missing_y
        , subproblem_topLeft_x, subproblem_topLeft_y)
```

Time Complexity Analysis

The basic operation is assignment to an element in the 2^n*2^n board.

Analysis could also be done considering comparisons to the coordinates of the missing square as a basic operation, this will reveal the same results.

In one function call there are either solution using recursion or solution using dynamic programming memoization:

In recursion solution there are 4 recursive calls + 3 assignments (the tromino placed in the center)

$T(2^n) = 4*T(2^{(n-1)}) + 3,$ $n > 1,$ $T(2^1) = 3$

Using Backward Substitution:

$T(2^n) = 4*T(2^{n-1}) + 3 = 4(4*T(2^{n-2}) + 3) + 3 = 4(4(4*T(2^{n-3}) + 3) + 3) + 3$

$= 4^k\ T(2^{n-k}) + 3\Sigma^{k-1}_{j=0}\ 4^j = 4^k\ T(2^{n-k}) + 3(4^k-1)/3 = 4^k * (1 + T(2^{n-k})) – 1$

At k = n-1

$T(2^n) = 4^{n-1} * (1 + T(2^1)) – 1 = 4^{n-1} * (1 + 3) – 1 = 4^n – 1 \in O(4\text{^}n)$

Algorithm Comparison

As mentioned in the Complexity Analysis section and noticed in this section, solving the problem using dynamic programming technique and divide and conquer technique didn't change the Time complexity. However the key difference is in the memory or to be precise: **Call Stack**.

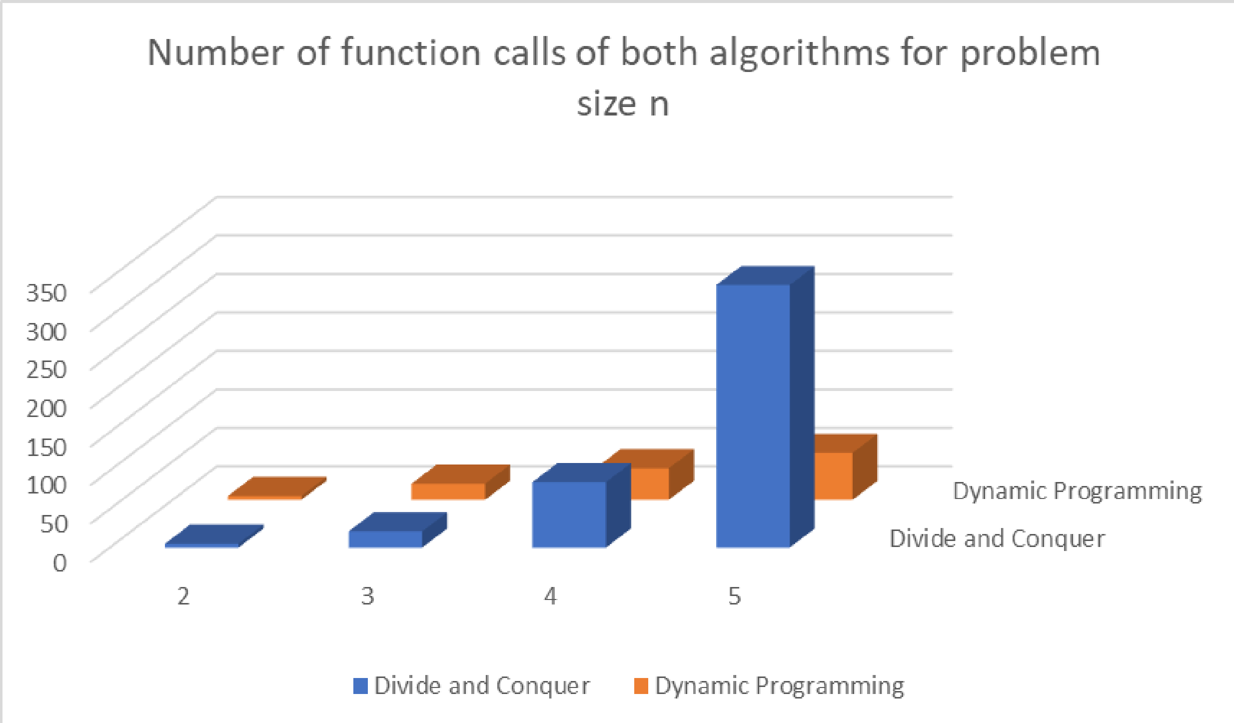
The dynamic programming approach reduces the number of recursive calls significantly at the expense of a 2D array of size n x 4 x 2 (as it is storing a pair of integers) which is negligible compared to the 2^n x 2^n board.

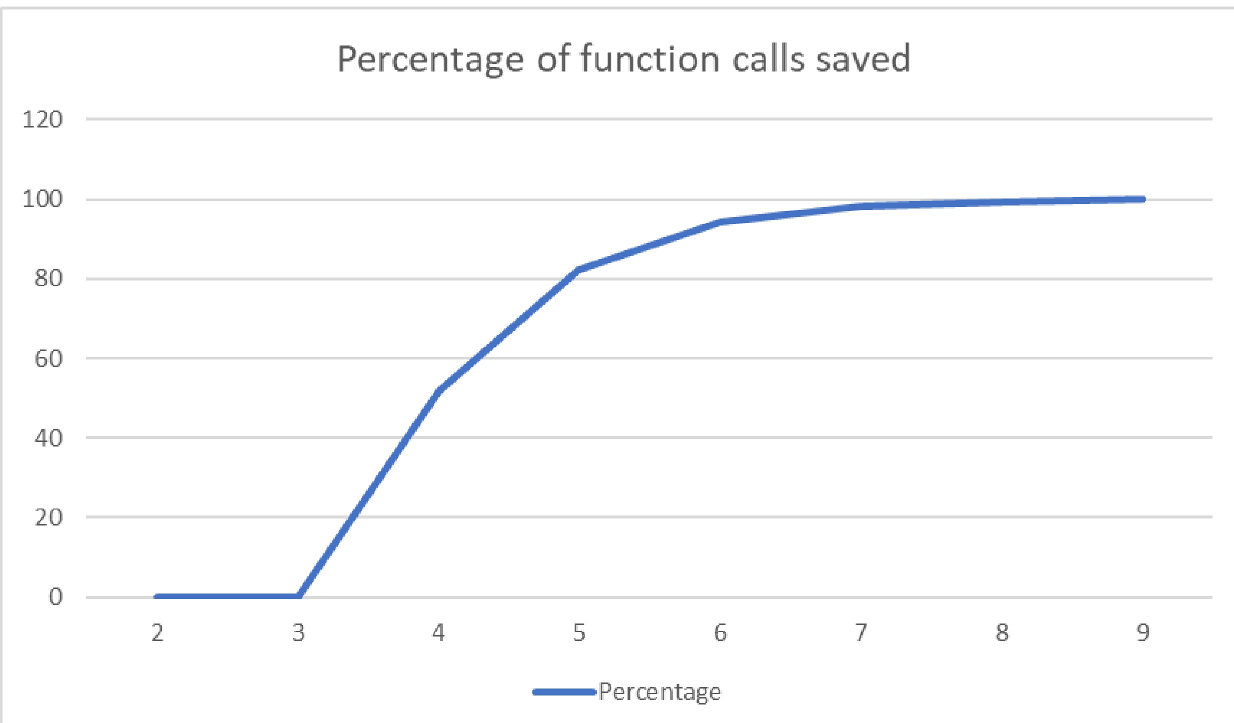
In the divide and conquer approach a function call either solves a base case (places one tromino) or generates extra 4 function calls while placing a tromino in the center of the subproblem board. Either way, it can be concluded that the number of function calls equals the number of trominoes of the problem = ((4^n) - 1) / 3 which increases **exponentially** with the increase of problem size n.

However, in the dynamic programming approach the number of function calls has a maximum of 5 for each subproblem size n > 1 + a maximum of 20 function calls for each subproblem of size 1. And it never exceeds the number of function calls of the divide and conquer approach. Therefore it increases **linearly** with the increase of problem size n.

Problem size n	Divide and Conquer function calls	Dynamic programming function calls	Percentage of function calls saved
2	5	5	0%
3	21	21	0%
4	85	41	51.76%
5	341	61	82.11%
6	1365	81	94.07%
7	5461	101	98.15%
8	21875	121	99.45%
9	87381	141	99.84%

This result was obtained by modifying the code to calculate function calls.





It is noticed that no saving of function calls for problem size 2 and 3, This is because no overlapping subproblems exist.

For example: a problem of size 3 has 4 subproblems of size 2 and all are unique, and 16 subproblems of size 1 which are base cases.

Conclusion

The Dynamic programming algorithm of this problem is theoretically capable of solving any version of this problem with any input in terms of problem size and missing square location. However, due to its exponential time complexity it is capable of solving problems with maximum size n of 15 (on average cpu).

For problems with problem size n larger than 7 it can't be visualized on the console, that's the reason why there are no large n in the output sample, however this can be solved by writing the output to the file instead.

The key difference of this algorithm and the popular divide and conquer algorithm is the reduction of Call Stack usage by reducing function calls at the expense of a small memory storage, exploiting the observation that many subproblems are actually overlapping with each other.

Even though both the main and the alternative complexity are of complexity $O(4^n)$ where n is the problem size, it can be proved that this problem has a tight lower bound using trivial lower bound method:

Since input can be minimized to the coordinates of the missing square and problem size n = 3, and the output is a colored board of size 2^n x 2^n = 4^n, the lower bound of this problem ∈ O(4^n) which is the efficiency class of both algorithms.

Task 2

Assumptions

- The chess board only contains one knight; there are no other pieces on the board.
- A cell is considered visited only when the knight lands on it, not just passes over it on its move

Problem Description

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).

Solution Steps

1. Start from any square on the board.
2. From that square, move to the neighboring square with the fewest accessible neighbors. An accessible neighbor is a square that can be reached by a knight move without going off the board and without visiting any square twice.
3. If there are multiple squares with the same minimum number of accessible neighbors, choose one at random.
4. Repeat step 2 from the newly visited square, until there is no other squares to move to
5. Check if all squares are visited and the knight can move from the last positions to the started square then end else chose another square from board and repeat from step 2

Pseudocode

```
FUNCTION valid_moves(x, y):
    ans = empty vector of pairs
    IF x - 1 >= 0 AND y + 2 <= 7 THEN
        IF passed_on[{x - 1, y + 2}] == 0 THEN
            ans.push_back({ x - 1,y + 2 })
```

```
  IF x - 2 >= 0 AND y + 1 <= 7 THEN
    IF passed_on[{x - 2, y + 1}] == 0 THEN
      ans.push_back({ x - 2,y + 1 })
  IF x - 2 >= 0 AND y - 1 >= 0 THEN
    IF passed_on[{x - 2, y - 1}] == 0 THEN
      ans.push_back({ x - 2,y - 1 })
  IF x - 1 >= 0 AND y - 2 >= 0 THEN
    IF passed_on[{x - 1, y - 2}] == 0 THEN
      ans.push_back({ x - 1,y - 2 })
  IF x + 1 <= 7 AND y + 2 <= 7 THEN
    IF passed_on[{x + 1, y + 2}] == 0 THEN
      ans.push_back({ x + 1,y + 2 })
  IF x + 1 <= 7 AND y - 2 >= 0 THEN
    IF passed_on[{x + 1, y - 2}] == 0 THEN
      ans.push_back({ x + 1,y - 2 })
  IF x + 2 <= 7 AND y + 1 <= 7 THEN
    IF passed_on[{x + 2, y + 1}] == 0 THEN
      ans.push_back({ x + 2,y + 1 })
  IF x + 2 <= 7 AND y - 1 >= 0 THEN
    IF passed_on[{x + 2, y - 1}] == 0 THEN
      ans.push_back({ x + 2,y - 1 })
  RETURN ans

FUNCTION next_move(x, y):
  vc = valid_moves(x, y)
  ans = { -1, -1 }
  IF vc is empty THEN
```

```
    PRINT "no valid moves"
    PRINT "you are at ", x, ' ', y
    RETURN ans
  END IF
  mn = MAX_INT
  FOR EACH e IN vc DO
    temp = size_of(valid_moves(e.first, e.second))
    IF temp < mn THEN
      mn = temp
      ans = e
    END IF
  END FOR
  RETURN ans
END FUNCTION

FUNCTION solution_found(chess[][]) RETURNS boolean
  passed_on.clear()
  FOR i = 0 to 7
    FOR j = 0 to 7
      IF chess[i][j] == 64 THEN
        vc = valid_moves(i, j)
        FOR EACH x IN vc DO
          IF chess[x.first][x.second] == 1 THEN
            RETURN true
          END IF
        END FOR
      END IF
```

```
    END FOR
  END FOR
  RETURN false
END FUNCTION

FUNCTION main() RETURNS int {
  n ← 8
  chess[n][n] ← 0
  for i ← 0 to n-1 do
    for j ← 0 to n-1 do
      chess[i][j] ← 0
    end for
  end for
  start ← (7,0)
  for x ← 0 to n-1 do
    for y ← 0 to n-1 do
      start ← (x,y)
      chess[start.first][start.second] ← 1
      curr ← start
      passed_on[curr] ← passed_on[curr] + 1
      for i ← 2 to 64 do
        curr ← next_move(curr.first, curr.second)
        chess[curr.first][curr.second] ← i
        passed_on[curr] ← passed_on[curr] + 1
      end for
      if (solution_found(chess)) then
        for i ← 0 to n-1 do
          for j ← 0 to n-1 do
            print chess[i][j], " "
```

```
          end for
          print '\n'
        end for
        print "**********************\n"
        return 0
      end if
    end for
  end for
END FUNCTION
```

Code Implementation

```
#include <iostream>
#include <cstring>
#include <string>
#include <vector>
#include<map>
#include<limits.h>

using namespace std;

#define N 18
map<pair<int, int>, int> passed_on;

vector<pair<int, int>> valid_moves(int x, int y) {
      vector<pair<int, int>> ans;
      if (x - 1 >= 0 && y + 2 <= N-1) {
            if (passed_on[{x - 1, y + 2}] == 0)
                  ans.push_back({ x - 1,y + 2 });
      }
      if (x - 2 >= 0 && y + 1 <= N-1) {
            if (passed_on[{x - 2, y + 1}] == 0)
                  ans.push_back({ x - 2,y + 1 });
      }
      if (x - 2 >= 0 && y - 1 >= 0) {
            if (passed_on[{x - 2, y - 1}] == 0) {
                  ans.push_back({ x - 2,y - 1 });
            }
      }
      if (x - 1 >= 0 && y - 2 >= 0) {
            if (passed_on[{x - 1, y - 2}] == 0) {
                  ans.push_back({ x - 1,y - 2 });
            }
      }
      if (x + 1 <= N-1 && y + 2 <= N-1) {
            if (passed_on[{x + 1, y + 2}] == 0) {
                  ans.push_back({ x + 1,y + 2 });
            }
      }
      if (x + 1 <= N-1 && y - 2 >= 0) {
            if (passed_on[{x + 1, y - 2}] == 0) {
                  ans.push_back({ x + 1,y - 2 });
            }
      }
```

```
      if (x + 2 <= N-1 && y + 1 <= N-1) {
            if (passed_on[{x + 2, y + 1}] == 0) {
                  ans.push_back({ x + 2,y + 1 });
            }
      }
      if (x + 2 <= N-1 && y - 1 >= 0) {
            if (passed_on[{x + 2, y - 1}] == 0) {
                  ans.push_back({ x + 2,y - 1 });
            }
      }
      return ans;
}

pair<int, int> next_move(int x, int y) {

      vector<pair<int, int>> vc = valid_moves(x, y);
      pair<int, int> ans = { -1,-1 };
      if (vc.empty()) {
            return ans;
      }
      int mn = INT_MAX;
      for (const auto& e : vc) {
            int temp = valid_moves(e.first, e.second).size();
            if (temp < mn) {
                  mn = temp;
                  ans = e;
            }
      }

      return ans;
}

bool solution_found(int chess[][N],pair<int,int> lp) {
      passed_on.clear();
      if (lp.first == -1)return 0;
      int i = lp.first;
      int j = lp.second;
      if (chess[i][j] == N*N) {
            vector<pair<int, int>> vc = valid_moves(i, j);
            for (const auto x : vc) {
                  if (chess[x.first][x.second] == 1)return 1;
            }
            return 0;
```

```
    }

    return 0;
}

int main() {
    int counter = 0;
    int chess[N][N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            chess[i][j] = 0;
        }
    }
    pair<int, int> start = { 7,0 };
    for (int x = 0; x < N; x++) {
        for (int y = 0; y < N; y++) {
            start.first = x;
            start.second = y;
            if (x >= N || x< 0 || y >= N || y < 0) {
                cout << "stack error\n";
                return 0;
            }
            pair<int, int> lastPostion = { -1,-1 };
            chess[start.first][start.second] = 1; // start postion
            pair<int, int> curr = start;
            passed_on[curr]++;
            for (int i = 2; i <= N*N; i++) {
                curr = next_move(curr.first, curr.second);
                if (curr.first < 0 || curr.first >= N || curr.second
< 0 || curr.second >= N) {
                    break;
                }
                chess[curr.first][curr.second] = i;
                passed_on[curr]++;
                if (chess[curr.first][curr.second] == N * N) {
                    lastPostion = curr;
                }

            }
            if (solution_found(chess,lastPostion)) {
                for (int i = 0; i < N; i++) {
                    for (int j = 0; j < N; j++) {
```

```
                                    cout << chess[i][j] << "\t";
                              }
                              cout << '\n';
                        }
                        cout << "**********************\n";
                        return 0;
                  }
                  else {
                        if(lastPostion.first >0 && lastPostion.second >0)
                              chess[lastPostion.first][lastPostion.second] =
0;
                  }

            }
      }

}
```

Complexity Analysis

The basic operation in the algorithm is finding the next move which is calling the **next_move()** function

the next move function have a loop that is executed at max 8 times as this the max moves that can be done by knight independent of the board size each time of this its calling the **valid_moves()** function the complexity of **valid_moves()** if O(1) so the complexity of the **next_move()** is $\sum_{i=0}^{7} \quad (1)$ which is O(8) that is equivalent to O(1)

The basic operation is done in the main function $\sum_{x=0}^{n} \quad \sum_{y=0}^{n} \quad \sum_{i=2}^{n^2} \quad (1)$=$(n^2-1)(n)(n)$ = $n^4 - n^2$ times where n is the dimensions of the board (a 8*8 board will have n equal to 8) so the time complexity of the program is **O(n^4)**

Sample Output

Given n = 8

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

Given n = 18

```
Select Microsoft Visual Studio Debug Console
50      71      8       75      52      69      10      79      54      67      12      89      56      65      14      139     58      63
7       74      51      70      9       80      53      68      11      90      55      66      13      156     57      64      15      138
72      49      84      81      76      175     86      91      78      151     88      159     148     153     140     155     62      59
83      6       73      176     85      92      77      174     87      220     149     152     157     160     165     60      137     16
48      97      82      93      178     181     248     221     150     173     158     219     170     147     154     141     164     61
5       94      177     182     247     244     179     230     249     222     227     172     161     218     169     166     17      136
98      47      96      245     180     285     278     243     228     231     252     223     226     171     146     163     142     167
95      4       183     284     279     246     229     286     277     250     225     232     253     162     217     168     135     18
46      99      280     185     298     283     288     319     242     261     276     251     224     215     234     145     208     143
3       184     45      282     289     320     299     296     287     314     241     260     233     254     209     216     19      134
44      281     100     303     186     297     318     313     322     295     262     275     256     239     214     235     144     207
101     2       187     290     317     304     321     300     315     274     311     240     259     236     255     210     133     20
188     43      306     1       302     291     316     323     312     263     294     257     272     213     238     131     206     123
33      102     193     190     305     324     301     266     293     310     273     200     237     258     211     124     21      132
42      189     34      307     194     191     292     309     268     199     264     271     212     125     130     205     122     113
35      32      103     192     39      308     267     198     265     270     201     126     129     204     121     114     117     22
104     41      30      37      106     195     28      269     108     197     26      203     110     127     24      119     112     115
31      36      105     40      29      38      107     196     27      202     109     128     25      120     111     116     23      118
***********************
```

Comparison with an Alternative Algorithm

Another solution for this problem can be done by backtracking, the backtracking solution goes as following

1. Initialize a chessboard with all cells marked as unvisited.
2. Start from a given cell, mark it as visited, and add it to the path.
3. Generate all valid moves from the current cell that lead to unvisited cells.
4. For each valid move, mark the new cell as visited, add it to the path, and recursively repeat the above steps.
5. If a Hamiltonian path is found (i.e., the path length equals the number of cells on the board), print the path and return true.
6. If no valid moves are left, backtrack to the previous cell, remove it from the path, and mark it as unvisited.
7. If all cells have been visited but no Hamiltonian path has been found, return false.

Code

```
#include <iostream>
#include <cstring>
#include <vector>
#include <map>
#include <iostream>
using namespace std;
#define N 6
map<pair<int, int>, int> passed_on;
pair<int, int> start;
vector<pair<int, int>> moves;
vector<pair<int, int>> valid_moves(int x, int y) {
	vector<pair<int, int>> ans;
	if (x - 1 >= 0 && y + 2 <= N - 1) {
		if (passed_on[{x - 1, y + 2}] == 0)
			ans.push_back({ x - 1,y + 2 });
	}
	if (x - 2 >= 0 && y + 1 <= N - 1) {
		if (passed_on[{x - 2, y + 1}] == 0)
			ans.push_back({ x - 2,y + 1 });
	}
	if (x - 2 >= 0 && y - 1 >= 0) {
		if (passed_on[{x - 2, y - 1}] == 0) {
			ans.push_back({ x - 2,y - 1 });
		}
	}
	if (x - 1 >= 0 && y - 2 >= 0) {
		if (passed_on[{x - 1, y - 2}] == 0) {
			ans.push_back({ x - 1,y - 2 });
		}
	}
	if (x + 1 <= N - 1 && y + 2 <= N - 1) {
		if (passed_on[{x + 1, y + 2}] == 0) {
			ans.push_back({ x + 1,y + 2 });
		}
	}
	if (x + 1 <= N - 1 && y - 2 >= 0) {
		if (passed_on[{x + 1, y - 2}] == 0) {
			ans.push_back({ x + 1,y - 2 });
		}
	}
	if (x + 2 <= N - 1 && y + 1 <= N - 1) {
		if (passed_on[{x + 2, y + 1}] == 0) {
```

```
                  ans.push_back({ x + 2,y + 1 });
            }
      }
      if (x + 2 <= N - 1 && y - 1 >= 0) {
            if (passed_on[{x + 2, y - 1}] == 0) {
                  ans.push_back({ x + 2,y - 1 });
            }
      }
      return ans;
}


bool solution_found(int chess[][N], pair<int, int> lp) {
      if (moves.size() < N * N)return false;
      passed_on[start] = 0;
      auto x = valid_moves(lp.first, lp.second);
      for (auto ans : x) {
            if (ans==start) {
                  return true;
            }
      }

      passed_on[start] = 1;
      return 0;
}


bool solve(int chess[][N],pair<int,int> curr) {
      passed_on[curr] = 1;
      if (solution_found(chess, !moves.empty() ? moves.back() : make_pair(-
1,-1))) {
            int i = 1;
            for (const auto& move : moves) {
                  chess[move.first][move.second] = i;
                  i++;
            }
            for (int i = 0; i < N; i++) {
                  for (int j = 0; j < N; j++) {
                        cout << chess[i][j] << '\t';
                  }
                  cout << '\n';
            }
            return true;
```

```
      }
      auto vec = valid_moves(curr.first, curr.second);
      bool ans = false;
      for (const auto &move : vec) {
            moves.push_back(move);
            passed_on[move] = 1;
            ans = ans || solve(chess, move);
            if (ans) break;
            moves.pop_back();
            passed_on[move] = 0;
      }
      return ans;
}


int main(){
      int chess[N][N];
      for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                  chess[i][j] = 0;
            }
      }
      start = { 2,2 };
      moves.push_back(start);
      cout << solve(chess, start);
}
```

In this case on an n x n chessboard, the branching factor is at most 8 (since a knight can make at most 8 moves from any position) and the depth of the search tree is n^2 (since the tour requires visiting each cell on the chessboard exactly once). Therefore, the time complexity of the algorithm is $O(8^{n^2})$.

So in the context of solving the knight's tour problem, while the backtracking algorithm guarantees a solution, it suffers from exponential growth in complexity with increasing board size. On the other hand, the greedy solution has a polynomial complexity, making it more practical for larger board sizes. Therefore, it can be argued that the greedy solution is more practical than the backtracking solution in practice.

Conclusion

The problem of finding a closed knight tour on a chess board can be approached using different algorithms. One such algorithm is a greedy approach, which selects the next move based on the fewest accessible neighbors. This method has been shown to provide a solution for an 8 x 8 chess board. The time complexity of the greedy algorithm is O(n^4), where n represents the dimensions of the board.

Alternatively, a solution can be obtained using a backtracking algorithm. Backtracking guarantees a solution, but its complexity grows exponentially with increasing board size. Specifically, the time complexity of the backtracking algorithm for a chess board is O(8^(n^2)). This exponential growth poses challenges when dealing with larger board sizes and makes the greedy solution more practical in practice.

It is worth mentioning that the greedy algorithm may encounter ties, where multiple moves have the same minimum number of accessible neighbors. The method chosen to resolve these ties can have a significant impact on the resulting solution and adds complexity to the implementation. In the provided implementation, the first move among the tied moves was selected.

However, it is important to note that the closed knight tour problem does not have a solution for all board sizes. Specifically, there are no solutions for boards of size n x n where n is odd or n is less than 6. This limitation arises due to the mathematical properties of knight movements and the constraints imposed by odd dimensions and small board sizes. Therefore, when considering the feasibility of finding a closed knight tour, it is crucial to consider the board size and its compatibility with the problem's constraints.

Task 3

Assumptions

- We will number the switches left to right from 1 to n
- Denote the “on” and “off” states of a switch by a 1 and 0, respectively.

Problem Description

There is a row of n security switches protecting a military installation entrance. The switches can be manipulated as follows:

(i) The rightmost switch may be turned on or off at will.

(ii) Any other switch may be turned on or off only if the switch to its immediate right is on and all the other switches to its right, if any, are off.

(iii) Only one switch may be toggled at a time.

Solution Steps

1. Declare an array switches and an integer n to store the number of switches.
2. Read the value of n from the user.
3. Declare integer variables count, pos, end, and begin to store the count of moves, the current position, the ending position of a group of switches that can be toggled together, and the starting position of the group of switches, respectively.
4. Initialize all switches to be turned on by setting each element of the switches array to 1.
5. Print the initial state of the switches by looping through the switches array and printing each element.
6. Call the dynamic function with parameters switches, pos, count, n, begin, and n.
7. Inside the dynamic function, check if all switches are turned off by counting the number of switches with a value of 0.
8. If all switches are turned off, return 1.

9. If the current position pos is the rightmost switch, toggle the switch and print the move by calling the printmove function. Then check for special cases and recurse accordingly.
10. If the current position pos is the second-to-the-rightmost switch, check if the switch to the right is turned on. If it is, toggle the current switch and print the move by calling the printmove function. Then recurse with the new position pos-1. If the switch to the right is not turned on, move to the next position by recursing with the new position pos+1.
11. If the current position pos is not the rightmost or second-to-the-rightmost switch, check if the switches to the right are turned off. If they are, toggle the current switch and print the move by calling the printmove function. Then move to the next group of switches by recursing with the new position begin and updated values of rest and begin. If only the switch to the right is turned on and the switch at the current position is also turned on, toggle the current switch, print the move, and move to the next group of switches by recursing with the new position begin and updated values of rest and begin. If only the switch to the right is turned on and the switch at the current position is turned off, toggle the current switch, print the move, and move to the next position by recursing with the new position pos+1. If none of the above conditions are met, move to the next position by recursing with the new position pos+2.
12. In the main function, print the minimum number of moves by printing the value of count.
13. End of the program.

Pseudocode

```
function dynamic(array, pos, count, size, begin, rest):
    // Check if all switches are turned off
    y = 0
    for i from 0 to size-1:
        if array[i] == 0:
            y = y + 1
    if y == size:
        return count

    // For the rightmost switch
    if pos == size-1:
        // Toggle the switch
        if array[pos] == 0:
            array[pos] = 1
        else:
            array[pos] = 0

        // Print the move
        printmove(size, array, count)

        // Check the special cases and recurse accordingly
        if begin == size-1:
            return count
        else if size == 4 and rest == 3 and array[size-1] == 1:
            return dynamic(array, pos-1, count, size, begin, rest)
        else if size == 5 and array[size-1] == 0 and array[size-2] == 1:
```

```
    return dynamic(array, pos-2, count, size, begin, rest)
  else if size == 5 and array[size-1] == 0 and array[size-2] == 0 and array[size-3] == 0:
    return dynamic(array, pos=begin, count, size, begin=0, rest=size)
  else if size == 5 and array[size-1] == 1 and rest == 3:
    return dynamic(array, pos-1, count, size, begin, rest)
  else:
    return dynamic(array, pos=begin, count, size, begin, rest)

// For the second-to-the-rightmost switch
else if pos == size-2:
  if array[pos+1] == 1:
    // Toggle the switch
    if array[pos] == 0:
      array[pos] = 1
    else:
      array[pos] = 0

    // Print the move
    printmove(size, array, count)

    return dynamic(array, pos-1, count, size, begin, rest)
  else:
    return dynamic(array, pos+1, count, size, begin, rest)

// For all other switches
else:
  // Check if the switches to the right are off
```

```
m = 0
for i from size-1 down to pos+1:
    if array[i] == 0:
        m = m + 1

// Check the conditions for toggling the switch at position pos
if array[pos+1] == 1 and m == size-pos-2 and array[pos] == 1:
    // Toggle the switch
    array[pos] = 0

    // Print the move
    printmove(size, array, count)

    return dynamic(array, pos=begin, count, size, begin=begin+1, rest=rest-1)
else if array[pos+1] == 1 and m == size-pos-2 and array[pos] == 0:
    // Toggle the switch
    array[pos] = 1

    // Print the move
    printmove(size, array, count)

    return dynamic(array, pos+1, count, size, begin, rest)
else:
    dynamic(array, pos+2, count, size, begin, rest)
    if array[pos] == 0 and begin != size-1:
        dynamic(array, pos=pos+1, count, size, begin=begin+1, rest=rest)
    else if array[begin] == 1:
```

dynamic(array, pos, count, size, begin, rest)

Code Implementation

```
#include <iostream>

using namespace std;

void printmove(int size, int array[] , int &count)
{
	for (int i = 0; i < size ; i++)
		cout << " " << array[i] << " ";
	cout << endl;
	count++;
}

int dynamic(int array[], int pos , int &count , int size , int &begin , int rest)
{
	//Check if all switches are closed
	int y = 0;
	for (int i = 0; i < size; i++)
	{
		if (array[i] == 0)
			y++;
	}

	if (y == size)
		return 1;


	//For the rignt most switch
	if (pos == size - 1)
	{
		if (array[pos] == 0)
			array[pos] = 1;
		else
			array[pos] = 0;

		printmove(size, array, count);

		if (begin == size - 1)
			return count;
		else if(size == 4 && rest == 3 && array[size - 1]== 1)
			return dynamic(array, pos - 1, count, size, begin, rest);
```

```
		else if(size == 5 && array[size - 1] == 0  && array[size -2])
			return dynamic(array, pos - 2, count, size, begin, rest);
		else if(size == 5 && array[size - 1] == 0 && array[size - 2] ==0 && array[size - 3]
== 0)
			return dynamic(array, pos = begin, count, size, begin = 0, rest = size);
		else if(size == 5 && array[size - 1] == 1 && rest == 3)
			return dynamic(array, pos - 1, count, size, begin, rest);
		else
			return dynamic(array, pos = begin, count, size, begin , rest);
	}

	//For the second to the right most switch
	else if (pos == size - 2)
	{
		if (array[pos + 1] == 1)
		{
			if (array[pos] == 0)
				array[pos] = 1;
			else
				array[pos] = 0;

			printmove(size, array, count);

			return dynamic(array, --pos, count, size, begin , rest);
		}

		else
			return dynamic(array, ++pos, count, size , begin , rest);
	}

	else
	{
		int m = 0;
		for (int i = size - 1; array[i] == 0; i--)
			m++;

		if (array[pos + 1] == 1 && m == size -pos- 2 && array[pos] == 1)
		{
			array[pos] = 0;

			printmove(size, array, count);

			return dynamic(array, pos = begin , count, size , ++begin , --rest);
```

```
        }
        else if (array[pos + 1] == 1 && m == size - pos - 2 && array[pos] == 0)
        {
            array[pos] = 1;

            printmove(size, array, count);

            return dynamic(array, pos + 1, count, size, begin, rest);
        }
        else
        {
            dynamic(array , pos + 2, count, size , begin , rest);

            if(array[pos] == 0 && begin != size -1)
                dynamic(array, pos++ , count, size , ++begin , rest);
            else if(array[begin] == 1)
                dynamic(array, pos, count, size, begin , rest);
        }
    }
}



int main()
{
    int switches[100];
    int n; //Number of Switehes
    cout << "Enter the Number of Switches => ";
    cin >> n;
    int count = 0;    //Count the number of moves
    int pos = 0;
    int end = 0;
    int begin = 0;
    for (int i = 0; i < n; i++) //All switches are turned on
    {
        switches[i] = 1;
        cout << " " << switches[i] << " ";
    }
    cout << endl;

    int x = dynamic(switches, pos, count, n , begin , n);

    cout << "Minimum Number of moves is = " << count << endl;
```

```
}
```

Complexity Analysis

The time complexity of this function is exponential, O(2^n), where n is the input to the function. This is because each recursive call to the function creates two more recursive calls, leading to an exponential growth in the number of function calls and computations. The space complexity of this function is also proportional to the depth of the recursion tree, which is O(n) in this case since each call stores its local variables and parameters on the call stack. However, since the maximum depth of the recursion tree is also exponential in n, the space complexity is dominated by the time complexity and is also O(2^n).

Sample Output

```
Microsoft Visual Studio Debug Console
Enter the Number of Switches => 3
 1  1  1
 1  1  0
 0  1  0
 0  1  1
 0  0  1
 0  0  0
Minimum Number of moves is = 5
```

```
Microsoft Visual Studio Debug Console
Enter the Number of Switches => 4
 1  1  1  1
 1  1  0  1
 1  1  0  0
 0  1  0  0
 0  1  0  1
 0  1  1  1
 0  1  1  0
 0  0  1  0
 0  0  1  1
 0  0  0  1
 0  0  0  0
Minimum Number of moves is = 10
```

```
Microsoft Visual Studio Debug Console
Enter the Number of Switches => 5
 1  1  1  1  1
 1  1  1  1  0
 1  1  0  1  0
 1  1  0  1  1
 1  1  0  0  1
 1  1  0  0  0
 0  1  0  0  0
 0  1  0  0  1
 0  1  0  1  1
 0  1  0  1  0
 0  1  1  1  0
 0  1  1  1  1
 0  1  1  0  1
 0  1  1  0  0
 0  0  1  0  0
 0  0  1  0  1
 0  0  1  1  1
 0  0  1  1  0
 0  0  0  1  0
 0  0  0  1  1
 0  0  0  0  1
 0  0  0  0  0
Minimum Number of moves is = 21
```

Comparison with an Alternative Algorithm

We can define a state dp[i][j] as the minimum number of moves required to turn off all switches from position i to position n-1, assuming that the switch at position i-1 is currently in state j (where j=0 means off and j=1 means on).

We can then use dynamic programming to fill in the dp array, starting from the rightmost switch (position n-1) and working backwards to the leftmost switch (position 0). At each position i, we can consider two cases:

If we leave the switch at position i-1 in its current state j, then the minimum number of moves required to turn off all switches from position i to position n-1 is dp[i+1][1] (assuming that the rightmost switch is on).

If we toggle the switch at position i-1 to state j', then the minimum number of moves required is 1 + dp[i+1][j'] (assuming that the rightmost switch is on and all switches to the right of position i are off).

We can then take the minimum of these two cases to fill in dp[i][j].

Finally, the minimum number of moves required to turn off all switches is given by dp[0][0].

Here are the steps for solving this problem using dynamic programming:

Initialize the dp array to all infinity values (except for dp[n-1][0] which is initialized to 0).

For each position i from n-2 down to 0, and each state j from 0 to 1:

a. Compute dp[i][j] using the recurrence relation described above.

The minimum number of moves required to turn off all switches is given by dp[0][0].

Also for Calculating the number of moves

```
function dp(n) {
if(n <= 2) return n;
return 2*dp(n-2) + dp(n-1) + 1;
}
```

To prove the given formula, we will use mathematical induction.

Base case:

For $n = 1$, we have $M(1) = 1$, which satisfies the given formula.

For $n = 2$, we have $M(2) = 3$, which also satisfies the given formula.

Inductive step:

Assuming that the formula holds for all values up to n-1, we will show that it also holds for n.

$$M(n) = 2M(n-2) + M(n-1) + 1$$

$$= 2(2M(n-4) + M(n-3) + 1) + (2M(n-3) + M(n-2) + 1)$$

$$= 4M(n-4) + 3M(n-3) + 2M(n-2) + M(n-1) + 3$$

Now, we can use the inductive hypothesis to substitute in the expressions for M(n-4), M(n-3), M(n-2), and M(n-1):

$$= 4(2M(n-6) + M(n-5) + 1) + 3(2M(n-5) + M(n-4) + 1) + 2(2M(n-4) + M(n-3) + 1) + M(n-1) + 3$$

$$= 8M(n-6) + 10M(n-5) + 10M(n-4) + 6M(n-3) + 3M(n-2) + M(n-1) + 7$$

$$= 2M(n-1) + (8M(n-6) + 10M(n-5) + 10M(n-4) + 6M(n-3) + 3M(n-2) + 7)$$

$$= 2M(n-1) + Q$$

where Q is a constant that only depends on n.

Therefore, we can see that the formula holds for n as well, completing the inductive step.

Thus, by mathematical induction, we have proven the formula $M(n) = 2M(n-2) + M(n-1) + 1$ for all positive integers n.

Conclusion

The algorithm solving the puzzle is based on the decrease and conquer strategy. Although solving the second-order recurrence for the number of moves by applying the standard techniques is both natural and easy, one can avoid this by following either Ball and Coxeter ,or Averbach and Chein.In our view, both methods are more cumbersome than the above solution. An entirely different approach was proposed by the French mathematician Louis A. Gros in 1872. His method amounted to representing states of the switches by bit strings anticipating modern-day Gray codes.The puzzle was proposed by C. E. Greenes [Gre73]. It imitates operations of a very old and well-known mechanical puzzle called the Chinese Rings. The abounding literature on the Chinese Rings is annotated by D. Singmaster.

Task 4

Assumptions

No assumptions.

Problem Description

An evil king is informed that one of his 1000 wine barrels has been poisoned. The poison is so potent that a miniscule amount of it, no matter how diluted, kills a person in exactly 30 days. The king is prepared to sacrifice 10 of his slaves to determine the poisoned barrel.

Solution Steps

1 Initialize the variables y and days to 0 to keep track of the poisoned barrel number and the number of days taken to identify the poisoned barrel.

2 Define the constants N (number of barrels), D (number of slaves), power (power of D equal to or greater than N), and size (minimum number greater than N that is a multiple of D so that it can be divided among the slaves properly).

3 Declare an array poison of size size to represent whether each group of barrels is poisoned or not.

4 Set the poisoned barrel number y to the index of the poisoned group (e.g., poison[124] = 1 means that barrel no 124 is poisoned and y is initially set to 0).

5 Set the variable tens to the maximum group size (size/D) to represent the positional value of the current group being tested.

6 Define the win function, which takes in the poison array, the start and end indices of the group of barrels being tested, and the slave number (1 to D).

7 Iterate through the barrels in the group being tested using a for loop.

8 If a poisoned barrel is found(slave dies), update the poisoned barrel number y accordingly ,divide tens by D, increment days and return.

9 If no poisoned barrel is found, continue iterating through the barrels.

10 Define the divide function, which takes in the poison array and the start and end indices of the group of barrels to be tested.

11 Use a for loop to divide the group of barrels into D (number of slaves) subgroups and call the win function for each subgroup.

12 Repeat steps 6-9

13 If the size of the current group being tested is greater than or equal to D, recursively call the divide function (step no. 10)for each subgroup.

14 When the poisoned barrel is found, print the poisoned barrel number y and the number of days taken to identify the poisoned barrel.

Pseudocode

```
Set y, z, and days to 0

Set N to the number of barrels
Set D to the number of slaves
Set size to the smallest multiple of D that is greater than or equal to N

Create a boolean array poison of size size and set all elements to false
Set the index of the poisoned barrel to true in the poison array

Set tens to size/D

Define function win(poison, start, end, slave):
    For i from start to end in the poison array:
        If poison[i] is true:
            Update y and z
            Return

Define function divide(poison, start, end):
```

```
    If end - start < D:
        Return

    For i from 1 to D:
        Call win(poison, start + (i-1)*(end-start)/D, start + i*(end-start)/D, i)

    For i from 1 to D:
        Call divide(poison, start + (i-1)*(end-start)/D, start + i*(end-start)/D)

    Set tens to tens/D
    Increment days

Call divide(poison, 0, size)

Print the values of y and days
```

Code Implementation

```
#include <iostream>
#include <cmath>
using namespace std;
int y=0;//poisoned barrel number
int days=0;//dayes required to solve this problem
const int N=1000;//Number of barrels
static int D=10;//Number of slaves
const int power=ceil(log(N)/log(D));
const int size= pow(D,power);//minimum number greater than number of barrels that is a multiple of number of slaves so that it can be divided among them properly
static int tens= size/D;
```

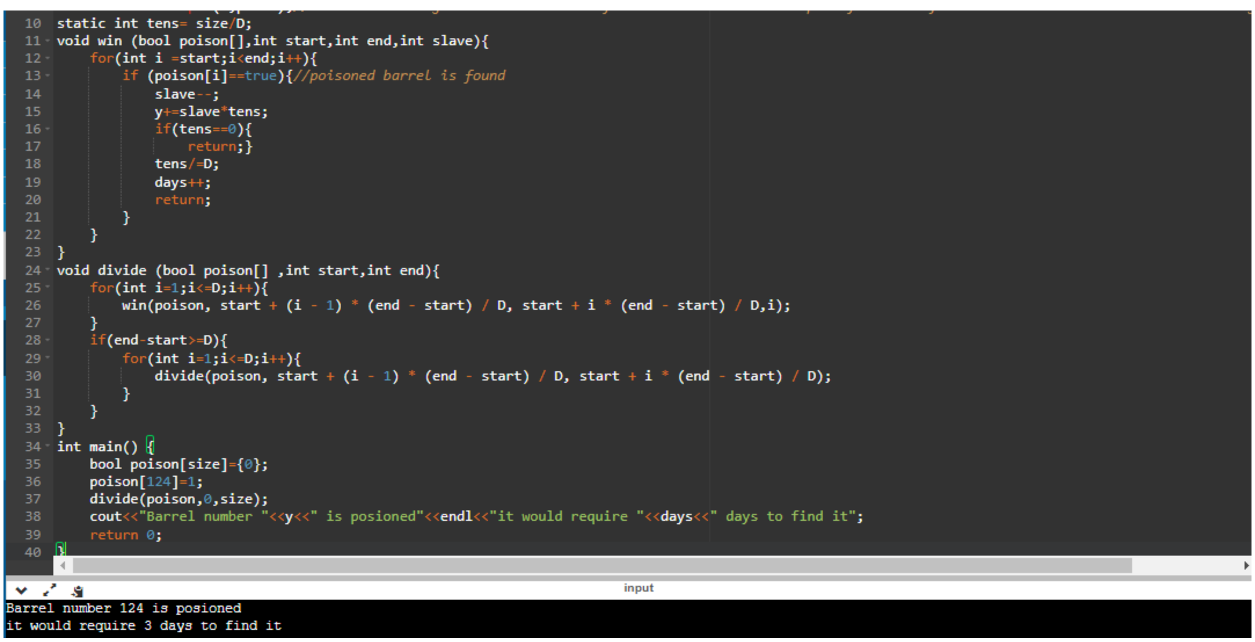
```
void win (bool poison[],int start,int end,int slave){
   for(int i =start;i<end;i++){
      if (poison[i]==true){//poisoned barrel is found
         slave--;
         y+=slave*tens;
         if(tens==0){
            return;}
         tens/=D;
         days++;
         return;
      }
   }
}
void divide (bool poison[] ,int start,int end){
   for(int i=1;i<=D;i++){
      win(poison, start + (i - 1) * (end - start) / D, start + i * (end - start) / D,i);
   }
   if(end-start>=D){
      for(int i=1;i<=D;i++){
         divide(poison, start + (i - 1) * (end - start) / D, start + i * (end - start) / D);
      }
   }
}
int main() {
   bool poison[size]={0};
   poison[124]=1;
   divide(poison,0,size);
```

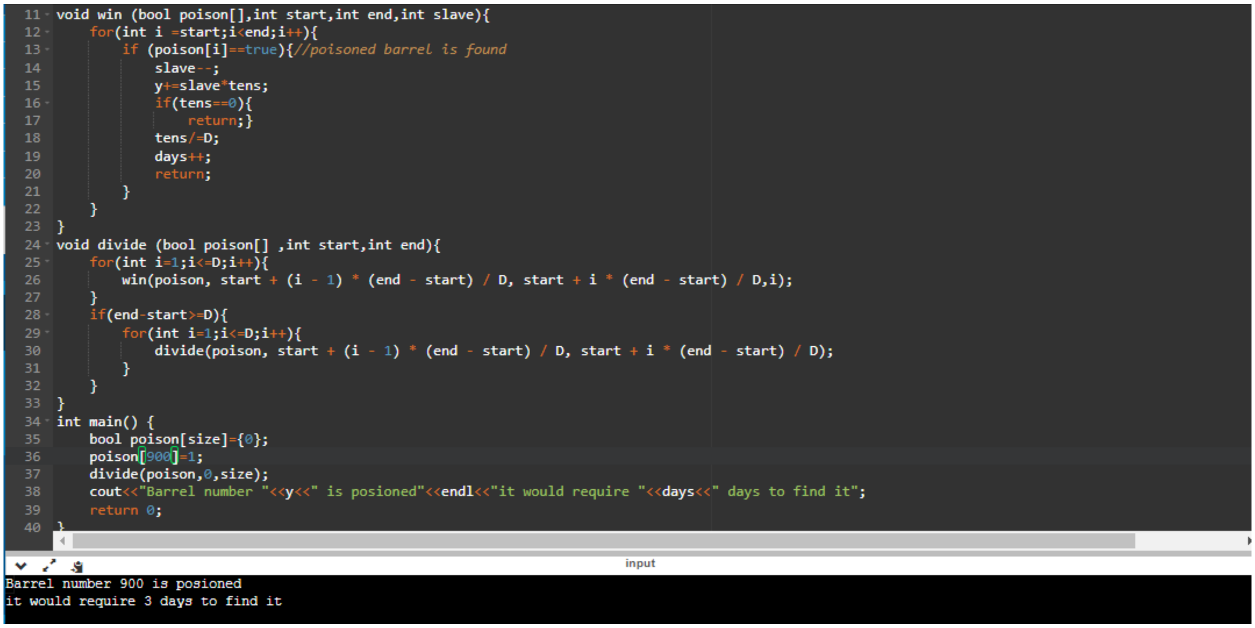
```
    cout<<y<<endl<<days;

    return 0;

}
```

Complexity Analysis

Time complexity of the provided code is O(N * log_D N).The reason for this time complexity is that each recursive call of the divide function partitions the barrels into D subgroups, each of size N/D. The win function then iterates over the barrels in each subgroup, which takes O(N/D) time. The divide function is called recursively until the subgroup size is less than D, which takes O(log_D N) recursive calls. Therefore, the total time complexity of the algorithm is the product of the number of iterations of the win function and the number of recursive calls to the divide function, which is O(D * (N/D) * log_D N). Simplifying this expression gives O(N * log_D N), since D is constant.

Sample Output





```
void win (bool poison[],int start,int end,int slave){
    for(int i =start;i<end;i++){
        if (poison[i]==true){//poisoned barrel is found
            slave--;
            y+=slave*tens;
            if(tens==0){
                return;}
            tens/=D;
            days++;
            return;
        }
    }
}
void divide (bool poison[] ,int start,int end){
    for(int i=1;i<=D;i++){
        win(poison, start + (i - 1) * (end - start) / D, start + i * (end - start) / D,i);
    }
    if(end-start>=D){
        for(int i=1;i<=D;i++){
            divide(poison, start + (i - 1) * (end - start) / D, start + i * (end - start) / D);
        }
    }
}
int main() {
    bool poison[size]={0};
    poison[900]=1;
    divide(poison,0,size);
    cout<<"Barrel number "<<y<<" is posioned"<<endl<<"it would require "<<days<<" days to find it";
    return 0;
}
```

input

```
Barrel number 900 is posioned
it would require 3 days to find it
```

Comparison with an Alternative Algorithm

Binary search

```
#include <ctime>

#include <cstdlib>

#include <iostream>

using namespace std;



int main() {

    srand(time(0)); // seed the random number generator

    int poisoned_barrel = rand() % 1000 + 1; // choose a random poisoned barrel

    cout << "The poisoned barrel is: " << poisoned_barrel << endl;


    int left = 1;

    int right = 1000;


    while (left <= right) {
```

```
        int mid = (left + right) / 2;
        cout << "Testing barrels " << left << " to " << right << endl;

        if (poisoned_barrel == mid) {
            cout << "Barrel " << mid << " is poisoned!" << endl;
            break;
        }
        else if (poisoned_barrel < mid) {
            right = mid - 1;
        }
        else {
            left = mid + 1;
        }
    }

    return 0;
}
```

The time complexity of the provided code is O(log N), where N is the number of barrels. However, it would take a lot of days more than that described above The reason behind that is that each iteration is dependant on the one before so we have to wait 30 days after each iteration to see which half we are going to test next

Brute Force:

```
int main() {
    int poisoned_barrel = rand() % 1000 + 1; // choose a random poisoned barrel
    cout << "The poisoned barrel is: " << poisoned_barrel << endl;

    for (int i = 1; i <= 1000; i++) { // test each barrel in sequence
        if (i == poisoned_barrel) {
            cout << "Barrel " << i << " is poisoned!" << endl;
```

```
            break;
        }
    }

    return 0;
}
```

The time complexity of the provided code is O(N), where N is the number of barrels. However, it would take a lot of days more than that described above The reason behind that we have to test each barrel separately so we have to wait 30 days to see whether a barrel is poisoned (30 days*1000 barrels) so, this solution might be impractical at all.

Conclusion

Best option would be to divide barrels in D groups and repeat. And after 30 days positioned value of each slave is added to get number of poisoned barrel and that solution would let the king know which barrel is poisoned before the feast if 10 slaves are used .

Otherwise if any other algorithm is used he wouldn't be able to find poisoned barrel before the feast.

Task 5

Assumptions

No Assumptions!

Problem Description

There are n coins placed in a row. The goal is to form n/2 pairs of them by a sequence of moves. On the first move a single coin has to jump over one coin adjacent to it, on the second move a single coin has to jump over two adjacent coins, on the third move a single coin has to jump over three adjacent coins, and so on, until after n/2 moves n/2 coin pairs are formed. (On each move, a coin can jump right or left but must land on a single coin. Jumping over a coin pair counts as jumping over two coins. Any empty space between adjacent coins is ignored.) Determine all the values of n for which the problem has a solution and design an algorithm that solves it in the minimum number of moves for those n's. Design a greedy algorithm to find the minimum number of moves

Solution Steps

The problem has a solution if and only if N is a multiple of 4

Let's imagine we are at the final move – all coins are paired except 2 single ones – so one of 2 remaining coins has to jump over coin pairs – even number of coins – in order to reach the other single coin and form a pair with it. But if N is even but not a multiple of 4 (ex: N=10), and we know that at the final move, the coin has to jump over N/2 coins, N/2 here is 5 (odd). In this case, the problem has no solution.

Greedy Solution 1

1. There are 2 restrictions on the number of jumps: First, we must start with jumping over 1 coin. Second, the number of moves increases by 1 on each move.
2. If we try to pair the first coin, then the second, and so on, we will get stuck.
3. For the pairing to run smoothly, let's ignore the first restriction; we will start by jumping the first coin over [(n/2) - 1] coins, the second on (n/2) coins, and so on. We will end up with (n/2) coin pairs smoothly.
4. So, we need some process until the number of jumps allowed equals half the number of the remaining single coins.

5. This process is a backward pairing; we will put coin number (n-2) on coin (n), then coin number (n-5) on coin (n-1), then coin number (n-8) on coin (n-2), and so on, until the number of jumps allowed is equal to half the number of the remaining single coins. And so, we can peacefully perform step 3, without ignoring either of the 2 restrictions.

Greedy Solution 2

1. Find the first single coin.

2. Try to jump forward the legal number of jumps depending on the move number.
   - If the coin is paired with another one, continue and repeat **step 1.**
   - If not, go to **step 3.**

3. If the number of jumps is odd and the right-adjacent coin is single, or the number of jumps is even and half the number of jumps is odd, move the coin 2 positions forward, and continue and repeat **step 1.**

4. Else, increment the number of moves, and the number of jumps, and go to **step 2.**

Pseudocode

Greedy Solution 1

```
FUNCTION coin_pairs(n):
    IF n % 4 != 0 OR n < 4:
        RETURN false
    coins = array of n 1's
    jumps = 2
    single_coins = n
    IF n == 4:
        forward_pairing(coins, single_coins, jumps)
        RETURN true
    ELSE:
        backward_pairing(coins, single_coins, jumps)
        forward_pairing(coins, single_coins, jumps)
        RETURN true

FUNCTION forward_pairing(coins[], single_coins, jumps):
    source_index = 0
    WHILE single_coins > 0:
        coins[source_index] = 0
        target_index = source_index + 1
        jumps_count = 0
        WHILE jumps_count < jumps - 1:
            jumps_count += coins[target_index]
            target_index += 1
        IF coins[target_index] == 0:
```

```
            target_index += 1
        coins[target_index] = 2
        single_coins -= 2
        source_index += 1
        jumps += 1

FUNCTION backward_pairing(coins[], single_coins, jumps):
    down3_idx = n - 3
    down1_idx = n - 1
    WHILE single_coins > (2 * jumps):
        coins[down3_idx] = 0
        IF coins[down1_idx] == 0:
            down1_idx -= 1
        coins[down1_idx] = 2
        down3_idx -= 3
        down1_idx -= 1
        single_coins -= 2
        jumps += 1
```

Greedy Solution 2

```
FUNCTION pair_coins(n):
      IF n % 4 != 0 THEN
      RETURN false
      END IF

      coins = array of n 1's
      jumps = 2
      moves = 0
      single_coins = n
      source_index = 0

      WHILE single_coins > 0:
            coins[source_index] = 0
            target_index = source_index + 1
            jumps_count = 0

            WHILE jumps_count < jumps - 1 AND target_index < n:
                  jumps_count = jumps_count + coins[target_index]
                  target_index = target_index + 1
                  END WHILE

            IF jumps_count == jumps - 1:
                  coins[target_index] = coins[target_index] + 1
                  single_coins = single_coins - 2
                  jumps = jumps + 1
                  moves = moves + 1
            ELSE IF (((jumps - 1) % 2 != 0) AND (coins[source_index + 1] == 1))
                  OR (((jumps - 1) % 2 == 0) AND (((jumps - 1) / 2) % 2 != 0)):
                  target_index = source_index + 2
                  coins[target_index] = coins[target_index] + 1
                  single_coins = single_coins - 2
```

```
                jumps = jumps + 1
                moves = moves + 1
          ELSE
                jumps = jumps + 1
                moves = moves + 1
                CONTINUE
          END IF

          WHILE coins[source_index] != 1:
                source_index = source_index + 1
          END WHILE
     END WHILE
     RETURN true
END FUNCTION
```

Code Implementation

Greedy Solution 1

```
#include <iostream>
using namespace std;
#define N 16

void print_coin_row(int coins[], int n) {
      cout << "The final paired coins row:" << endl;
      for (int i = 0; i < n; i++)
            cout << coins[i] << " ";
      cout << endl << endl;
      cout << "For N = " << N << ", the coins are paired in " << N / 2 <<
      " moves." << endl;
}

void forward_pairing(int coins[], int& single_coins, int& jumps) {
      int source_index = 0;   // index of the coin to be moved
      int jumps_count = 0;    // counting number of coins to jump over
      int target_index = 0;   // index of the position to where the coin is moved
      // pair until the number of single coins = 0
      while (single_coins != 0) {
            // remove the single coin to be paired
            coins[source_index] = 0;
            target_index = source_index + 1;
            // count jumps
            jumps_count = 0;
            while (jumps_count < jumps - 1) {
                  jumps_count += coins[target_index];
                  target_index++;
            }
            // ignore the empty space
            if (coins[target_index] == 0) target_index++;
            // pair the coin
            coins[target_index] = 2;
            // update variables
            single_coins -= 2;
            source_index++;
```

```
            jumps++;
      }
}
```

```
void backward_pairing(int coins[], int& single_coins, int& jumps) {
      int down3_idx = N - 3;  // index of the coin to be moved
      int down1_idx = N - 1;  // index of the position to where the coin is moved
      // pair until number of single coins = 2 * number of jumps
      while (single_coins > jumps * 2) {
            // remove the single coin to be paired
            coins[down3_idx] = 0;
            // ignore the empty space
            if (coins[down1_idx] == 0) down1_idx--;
            // pair the coin
            coins[down1_idx] = 2;
            // update variables
            down3_idx -= 3;
            down1_idx--;
            single_coins -= 2;
            jumps++;
      }
}

bool coin_pairs(int n) {
      // make sure that number of coins is POSITIVE, EVEN, and MULTIPLE OF 4
      If  (n % 4 != 0 || n < 4) return 0;
      static int coins[N];
      std::fill_n(coins, N, 1);     // row of N single coins
      static int jumps = 2;         // jumps = 2 means jumping over 1 coin,
                                       or moving forward/backward by 2 positions
      static int single_coins = n;  // number of remaining single coins
      if (n == 4) {
            forward_pairing(coins, single_coins, jumps);
            print_coin_row(coins, n);
            return 1;
      }
      else {
            backward_pairing(coins, single_coins, jumps);
            forward_pairing(coins, single_coins, jumps);
            print_coin_row(coins, n);
            return 1;
      }
}

int main() {
      coin_pairs(N);
}
```

Greedy Solution 2

```
#include <iostream>
using namespace std;
#define N 16
bool pair_coins(int n) {
      if (n % 4 == 0) {
            int coins[N];
            std::fill_n(coins, N, 1); // row of N single coins
            int jumps = 2;            // jumps = 2 means jumping over 1 coin,
                                      // or moving forward/backward by 2 positions
            int moves = 0;            // move count (to be minimized)
            int single_coins = n;     // number of remaining single coins
            int source_index = 0;     // index of the coin to be moved
            int jumps_count = 0;      // counting number of coins to jump
over
            int target_index = 0;     // index of the position to where the
                                      // coin is moved

            while (single_coins > 0) {
                  // remove the single coin to be paired
                  coins[source_index] = 0;
                  target_index = source_index + 1;
                  // count jumps
                  jumps_count = 0;

                  while (jumps_count < jumps - 1 && target_index < n - 1) {
                        jumps_count += coins[target_index];
                        target_index++;
                  }

                  // Forward Pairing from the 1st time
                  if (jumps_count == jumps - 1) {
                        // pair the coin
                        coins[target_index]++;
                        // update variables
                        single_coins -= 2;
                        jumps++;
                        moves++;
                  }

                  // Bounced Pairing from the 1st time
                        // jumps are odd AND OR jumps are even AND jumps/2 are odd
                  else if ((((jumps - 1) % 2 != 0) && (coins[source_index + 1] == 1))
                        || (((jumps - 1) % 2 == 0) && ((jumps - 1) / 2) % 2 != 0)) {

                        // set the pairing index
                        target_index = source_index + 2;
                        // pair the coin
```

```
                    coins[target_index]++;
                    // update variables
                    single_coins -= 2;
                    jumps++;
                    moves++;
                }
                // Pairing failed from the 1st time
                else {
                    // update variables and try once again
                    jumps++;
                    moves++;
                    continue;
                }

                // find the next single coin
                while (coins[source_index] != 1) source_index++;
            }

            // Print the paired coins row
            cout << "The final paired coins row:" << endl;
            for (int i = 0; i < n; i++)
                cout << coins[i] << " ";
            cout << endl << endl;

            // Print number of moves
            cout << "For N = " << N << ", the coins are paired in " << moves
                 << " moves." << endl;
            cout << "The optimal number of moves is " << N / 2 << " moves." << endl;
            return 1;
        }

        else return 0;
}

int main() {
        pair_coins(N);
}
```

Complexity Analysis

Greedy Solution 1

The **forward_pairing** function iterates over the coins in the row, and for every single coin, it performs a loop that jumps over a number of coins proportional to the number of jumps. Therefore, the time complexity of this function is **O(n*jumps)**, where jumps is the number of jumps performed by the function.

The **backward_pairing** function also iterates over the coins in the row, and for every single coin, it performs a constant number of operations, so its time complexity is **O(n)**.

The **coin_pairs** function calls **backward_pairing** first, which has a time complexity of **O(n)**, and then **forward_pairing**, which has a time complexity of **O(n*jumps)**. The value of jumps increases as the function progresses, so the time complexity of the entire function is dominated by the time complexity of **forward_pairing**.

Since the maximum value of jumps is n/2, the worst-case time complexity of the algorithm is **O(n^2)**. However, in practice, the value of jumps is likely to be much smaller than n/2, so the actual time complexity of the algorithm may be much lower.

Greedy Solution 2

The time complexity of the algorithm is **O(n^2)** as the **worst case**, where n is the number of coins. This is because the outer while loop iterates over **n/2** coins, and the nested while loops and conditional statements add an additional **O(n)** factor to the computation.

Sample Output

Greedy Solution 1

```
The final paired coins row:
0 0 2 2

For N = 4, the coins are paired in 2 moves.
```

```
The final paired coins row:
0 0 0 2 2 0 2 2

For N = 8, the coins are paired in 4 moves.
```

```
The final paired coins row:
0 0 0 0 2 2 0 2 2 0 2 2

For N = 12, the coins are paired in 6 moves.
```

```
The final paired coins row:
0 0 0 0 0 2 2 0 2 2 0 2 2 0 2 2

For N = 16, the coins are paired in 8 moves.
```

```
No solution for N = 10. N must be a multiple of 4.
```

```
No solution for N = 9. N must be a multiple of 4.
```

```
No solution for N = 2. N must be a multiple of 4.
```

Greedy Solution 2

```
The final paired coins row:
0 0 2 2

For N = 4, the coins are paired in 2 moves.
The optimal number of moves is 2 moves.
```

```
The final paired coins row:
0 0 2 2 0 0 2 2

For N = 8, the coins are paired in 6 moves.
The optimal number of moves is 4 moves.
```

```
The final paired coins row:
0 0 2 2 0 0 0 0 2 2 2 2

For N = 12, the coins are paired in 6 moves.
The optimal number of moves is 6 moves.
```

```
The final paired coins row:
0 0 2 2 0 0 0 0 2 2 2 2 0 0 2 2

For N = 16, the coins are paired in 10 moves.
The optimal number of moves is 8 moves.
```

```
No solution for N = 10. N must be a multiple of 4.
```

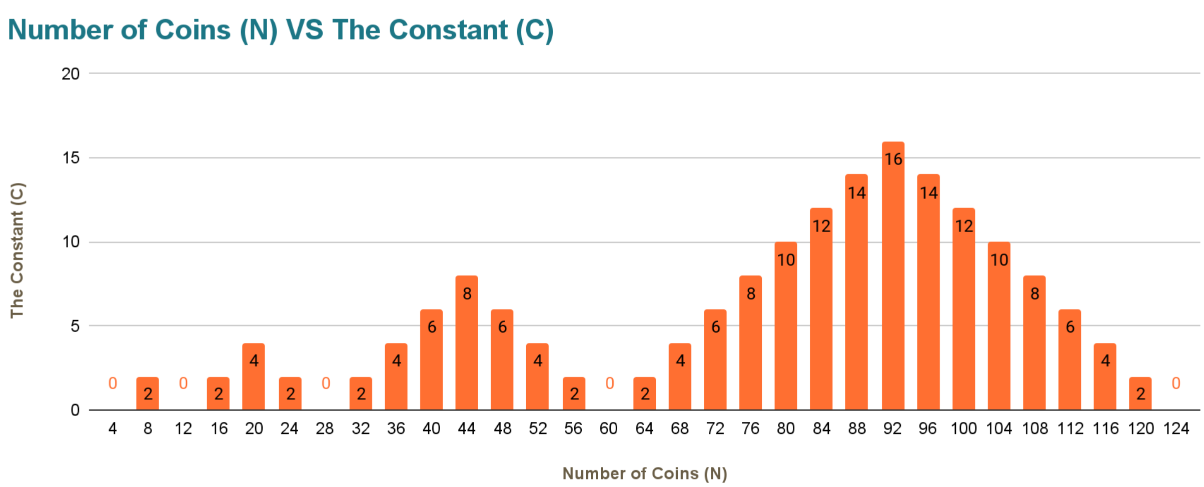
```
No solution for N = 9. N must be a multiple of 4.
```

```
No solution for N = 2. N must be a multiple of 4.
```

Comparison with an Alternative Algorithm

The first algorithm is faster than the second one, as it assumes that the problem is solvable for a given n and applies a **fixed sequence of pairings**. The second algorithm, on the other hand, tries **different pairings until all coins are paired**, which may require more moves.

- The 2 algorithms have almost the **same worst-case time complexity**: **O(n^2)**
- The **first algorithm** always solves the problem in the **minimum number of moves N/2**
- The **second algorithm** solves the problem in **N/2 + C moves**, where C < N/2



Conclusion

The first algorithm is guaranteed to solve the problem in the minimum number of moves N/2 for all legal N's, while the second algorithm is simple in implementation and straightforward in thinking. If the number of moves is critical, the first algorithm is the best choice. Meanwhile, if the number of moves is not a big matter, the second algorithm is the choice.

Task 6

Assumptions

- If the returned fakeIndex is = -999 then all of the coins are genuine
- Number of coins must be more than 2
- We have an integer called weightDifference, If it is returned with a value of 0 then the fake coin is lighter than the genuine Coins. If it is returned with a value of 1 then the fake coin is heavier than the genuine Coins.
- The balance of the scale is not faulty
- There cannot be more than 1 fake coin.

Problem Description

There are 12 coins identical in appearance; either all are genuine or exactly one of them is fake. It is unknown whether the fake coin is lighter or heavier than the genuine one. You have a two-pan balance scale without weights. The problem is to find whether all the coins are genuine and, if not, to find the fake coin and establish whether it is lighter or heavier than the genuine ones. Design a Dynamic Programming algorithm to solve the problem in the minimum number of weighings.

Solution Steps

1. Create 4 arrays, The first is called memory, The second is called temp, The third is called ThrownCoins, The fourth is called Coins.

2. Divide the bag of coins into two equal bags if the number of coins is divisible by 2

    2.1. If the number of coins is odd but not equal 3 (For example : 15) then throw away a coin, Store its index in the ThrownCoins array and divide them into two equal bags

    2.2. Set ThrownKeyFlag = true to indicate that we have coins outside of the subtree

3. Compare the two equal bags.

3.1. If they are equal in weight, Then all of the coins are genuine and we only needed to weigh them once.

3.2. If they are not equal then weigh them.

3.2.1. If the left bag is larger than the right bag : Fill the indexes of the left bag in temp array with 1s and the indexes of the right bag in temp array with -1s

3.2.2. If the left bag is smaller than the right bag : Fill the indexes of the left bag in temp array with -1s and the indexes of the right bag in temp array with 1s

4. Fill the memory array with the same values in temp.

5. (Recursive Step) : Divide the set of bags into left subtree and right subtree then compare the two subtrees.

5.1. If the left bag is larger than the right bag : Fill the indexes of the left bag in temp array with 1s and the indexes of the right bag in temp array with -1s

5.2. If the left bag is smaller than the right bag : Fill the indexes of the left bag in temp array with -1s and the indexes of the right bag in temp array with 1s

5.3. Compare the temp array with memory array

5.3.1. If temp[i] == memory[i] where i is one of the indexes in the left sub-tree :- Then our fake coin is in the left subtree and set boolean rightR = false so that we remove the right subtree.

5.3.2. If temp[j] == memory[j] where j is one of the indexes in the right subtree :- Then our fake coin is in the right subtree and set boolean leftR = false so that we remove the left subtree.

5.4. If the group of coins reaches only 2 coins :- 1 of them is fake. ***(Base case 1)***

5.4.1. Compare the temp[first coin] with memory[first coin]

5.4.1.1. if they are equal :- Then the first coin is the fake coin

5.4.1.1.1. set fakeIndex = temp[first coin]

5.4.1.2. if they are not equal :- Then the second coin is the fake coin

5.4.1.2.1. set fakeIndex = temp[second coin]

5.4.2. Exit the recursion

5.5. If the group of coins reaches only 3 coins :- 1 of them is fake. ***(Base case 2)***

5.5.1. Divide the group into 2 coins and 1 coin

5.5.2. Compare the first 2 coins, If they are equal then the fake coin is the third one

5.5.3. If they are not equal then compare the first coin with the third coin

5.5.3.1. If the first coin and the third are equal then the fake coin is the second one

5.5.3.2. If the first coin and third coin are not equal then the fake coin is the first one

5.5.3.3. Set isFound = true indicating that we have found the fake coin

5.5.3.4. Exit the recursion

5.6. Else : Return to step 5

6. After the recursion that occurs in step 5, If the *ThrownKeysFlag is = false (Meaning that we don't have any coins that were thrown during comparisons)* we will have found the fake coin.

7. If the *ThrownKeysFlag is = true and isFound = false (Meaning that we do have coins that were thrown during comparisons) Then we will start comparing the thrown coins with the middle coin (As we already determined that it is genuine)*

7.1. If the weight difference is equal for all thrown coins then all coins are genuine

7.2. If the weight difference is not equal, Then we have found the fake coin.

7.3. we set isFound = true then we retrieve its index from the ThrownCoins[] array

8. To find whether it is lighter than or heavier than the other coins, We check its index in memory.

    8.1. If memory[fakeIndex] == 1 then the fake coin is heavier than the genuine coins

    8.2. If memory[fakeIndex] == -1 then the fake coin is lighter than the genuine coins

By checking its index we will not need to weigh it against other coins again which gives us the optimal number of weighings.

Pseudocode

```
// Initialize some variables
weighings = 0
fakeIndex = -999
WeightDifference = -999 // 0 Means lighter, 1 Means Heavier
isFound = false
ThrownKeyFlag = false

// Define some constants
NCoins = 14 // Change this value to change number of coins
Memsize = NCoins+1
ThrownCoins[NCoins]
Thrown_maxIndex = NCoins - 1

result = -999
MidIndex = (NCoins - 1) / 2
Maxindex = NCoins - 1

// Fill the ThrownCoins array with a large negative value
fill()
{
  for i from 0 to NCoins
    ThrownCoins[i] = -900
```

```
}

// Compare the sum of two subsets of coins and return -1, 0 or 1
weigh(coins[NCoins], low1, high1, low2, high2)
{
    weighings = weighings + 1
    sum1 = 0
    sum2 = 0
    for i from low1 to high1
        sum1 = sum1 + coins[i]

    for i from low2 to high2
        sum2 = sum2 + coins[i]
    if sum1 == sum2 then return 0
    else if sum1 < sum2 then return -1
    else return 1
}

// Check if any of the thrown coins is fake and return its index and weight difference
checkThrownKeys(coins[NCoins], ThrownCoins[NCoins], mid)
{
    if not isFound and ThrownKeyFlag then
        i = 0
        while ThrownCoins[i] != -900
            thrownIndex = ThrownCoins[i]
            result = weigh(coins, ThrownCoins[i], ThrownCoins[i], mid, mid)
            if result == 0 then
            // This coin is genuine
            else if result == -1 then
            // This coin is fake and lighter
            fakeIndex = ThrownCoins[i]
            WeightDifference = 0
```

```
        isFound = true
        return (fakeIndex, WeightDifference)
        else if result == 1 then
        // This coin is fake and heavier
        fakeIndex = ThrownCoins[i]
        WeightDifference = 1
        isFound = true
        return (fakeIndex, WeightDifference)
        i = i + 1
    return (fakeIndex, WeightDifference)
}
// Find the fake coin among a subset of coins and return its index and weight difference
getFakeCoin(coins[NCoins], low, high, memory[Memsize])
{
        leftR = true
        rightR = true

        mid = (low + high) / 2
        temp[NCoins] = { 0 }

        result = -999

        if NCoins <= 2 then // Invalid entries
        return (-1, -1)

        if high - low == 1 then // For 2 coins
        result = weigh(coins, low, mid, mid + 1, high)
        if result == 0 then
        leftR = false
```

```
else if result == -1 then
    temp[low] = -1
temp[high] = 1

if temp[low] == memory[low] then
fakeIndex = low
WeightDifference = 0
isFound = true
return (fakeIndex, WeightDifference)
  else
fakeIndex = high
WeightDifference = 1
isFound = true
return (fakeIndex, WeightDifference)
else if result == 1 then
temp[low] = 1
temp[high] = -1

     if temp[low] == memory[low] then
fakeIndex = low
WeightDifference = 1
isFound = true
return (fakeIndex, WeightDifference)
else
fakeIndex = high
 WeightDifference = 0
isFound = true
```

```
    return (fakeIndex, WeightDifference)
}
if high - low == 2 then // For 3 coins
    result = weigh(coins, low, low, mid, mid)
    if result == 0 then
    result2 = weigh(coins, low, low, high, high)
    if result2 == 0 then
    // the 3 coins are genuine
    return (fakeIndex, WeightDifference)
    else
    fakeIndex = high
    isFound = true
    if memory[high] == 1 then
    WeightDifference = 1
    else
    WeightDifference = 0
    return (fakeIndex, WeightDifference)
    else if result == -1 then
    temp[low] = -1
    temp[mid] = 1
    if memory[low] == temp[low] then
    fakeIndex = low
    isFound = true
    if memory[low] == 1 then
    WeightDifference = 1
    else
    WeightDifference = 0
```

```
        return (fakeIndex, WeightDifference)
        else
        temp[low] = 1
        temp[mid] = -1
        fakeIndex = mid
        isFound = true
        if memory[mid] == 1 then
        WeightDifference = 1
        else
        WeightDifference = 0
        return (fakeIndex, WeightDifference)
else if result == 1 then
        temp[low] = 1
        temp[mid] = -1
        if memory[low] == temp[low] then
        fakeIndex = low
        isFound = true
        if memory[low] == 1 then
        WeightDifference = 1
        else
        WeightDifference = 0
        return (fakeIndex, WeightDifference)
        else
        fakeIndex = mid
        isFound = true
        if memory[mid] == 1 then
        WeightDifference = 1
```

```
        else
        WeightDifference = 0
        return (fakeIndex, WeightDifference)
if ( ( Maxindex % 2 != 0 or NCoins %2 != 0 ) and high % 2 == 0) then
        if (high - low+1) % 2 != 0 then // For odd coins except 3
        ThrownKeyFlag = true

        if NCoins % 2 != 0 then
        if ThrownCoins[low] == -900 then
        ThrownCoins[low] = low
        low = low + 1
        else if NCoins % 2 == 0 then
        if ThrownCoins[low] == -900 then
        ThrownCoins[low] = low
        if ThrownCoins[low + 1] == -900 then
        ThrownCoins[low + 1] = high
        low = low + 1
        high = high - 1
        Thrown_maxIndex = high - 1

if result < 0 then
        result = weigh(coins, low, mid, mid + 1, high)

if result == 0 and (high == MidIndex) then
        leftR = false // Remove the remaining left subtree
        mid = high
        if ThrownKeyFlag then
```

```
        high = Thrown_maxIndex
        else
        high = Maxindex
        // Coins are genuine
else if result == 0 and (high = Maxindex) and not ThrownKeyFlag then
        //All coins are genuine
        isFound = true
        return (fakeIndex, WeightDifference)
else if result == 0 and ThrownKeyFlag then
        //All non thrown coins are genuine
        leftR = false
        rightR = false
else if result == -1 then // sum1 < sum2
        for i from low to mid
        temp[i] = -1
        for j from mid + 1 to high
        temp[j] = 1

if memory[NCoins] == 0 then
        for i from low to mid
        memory[i] = -1
        for i from mid + 1 to high
        memory[i] = 1
        memory[NCoins] = 1
else // Memory is filled --> Start comparing values
        for i from low to mid
    if memory[i] != temp[i] then
        // do nothing
```

```
        else
        rightR = false
        memory[i] = temp[i]
        for i from mid + 1 to high
        if memory[i] != temp[i] then
        // do nothing
        else
        leftR = false
        memory[i] = temp[i]
}
else if result == 1 then // sum1 > sum2
        for i from low to mid
        temp[i] = 1
        for j from mid + 1 to high
        temp[j] = -1

        if memory[NCoins] == 0 then
        for i from low to mid
        memory[i] = 1
        for i from mid + 1 to high
        memory[i] = -1
        memory[NCoins] = 1
else
        for i from low to mid
        if memory[i] != temp[i] then
        // do nothing
        else
```

```
        rightR = false
        memory[i] = temp[i]
        for i from mid + 1 to high
        if memory[i] != temp[i] then
        // do nothing
        else
        leftR = false
        memory[i] = temp[i]
}


if leftR and not isFound then
        leftR = getFakeCoin(coins, low, mid, memory)
if rightR and not isFound then
        rightR = getFakeCoin(coins, mid + 1, high, memory)

if high == Thrown_maxIndex and not isFound then
        return checkThrownKeys(coins, ThrownCoins, mid)
else
        return (fakeIndex, WeightDifference)
}
```

Code Implementation

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

int weighings = 0;


int fakeIndex = -999;
int WeightDifference = -999; // 0 Means lighter, 1 Means
Heavier
bool isFound = false;
bool ThrownKeyFlag = false;



#define NCoins 14 // Change this value to change number
of coins
#define Memsize NCoins+1
int  ThrownCoins[NCoins];
int Thrown_maxIndex = NCoins - 1;


int result = -999;
int MidIndex = (NCoins - 1) / 2;
int Maxindex = NCoins - 1;

void fill()
{
    fill(ThrownCoins, ThrownCoins + NCoins, -900);
}



int weigh(int(&coins)[NCoins], int low1, int high1, int
low2, int high2)

{
    weighings++;
    int sum1 = 0, sum2 = 0;
    for (int i = low1; i <= high1; i++)
        sum1 += coins[i];

    for (int i = low2; i <= high2; i++)
        sum2 += coins[i];
    if (sum1 == sum2) return 0;
    else if (sum1 < sum2) return -1;
    else return 1;
}


pair<int, int> checkThrownKeys(int(&coins)[NCoins],
int(&ThrownCoins)[NCoins], int mid)
{

    if (!isFound && ThrownKeyFlag)
    {
        int i = 0;
        while (ThrownCoins[i] != -900)
        {
            int thrownIndex = ThrownCoins[i];
            int result = weigh(coins, ThrownCoins[i],
ThrownCoins[i], mid, mid);
            if (result == 0)
            {
                // This coin is genuine
            }
            else if (result == -1)
            {
                // This coin is fake and lighter
                fakeIndex = ThrownCoins[i];
                WeightDifference = 0;
                isFound = true;
                return make_pair(fakeIndex,
WeightDifference);
            }
            else if (result == 1)
            {
                // This coin is fake and heavier
                fakeIndex = ThrownCoins[i];
                WeightDifference = 1;
                isFound = true;
                return make_pair(fakeIndex,
WeightDifference);
            }
            i++;
        }
        return make_pair(fakeIndex, WeightDifference);


    }
}



pair<int, int> getFakeCoin(int(&coins)[NCoins], int low,
int high, int(&memory)[Memsize])
{
    bool leftR = true;
    bool rightR = true;

    int mid = (low + high) / 2;
    int temp[NCoins] = { 0 };

    result = -999;


    if (NCoins <= 2)   // Invalid entries
    {
        return make_pair(-1, -1);
    }
```

Dynamic Programming Algorithm Implementation (1)

```
if (high - low == 1) // For 2 coins
    {
        result = weigh(coins, low, mid, mid + 1, high);
        if (result == 0)
        {
            leftR = false;
        }
        else if (result == -1)
        {
            temp[low] = -1;
            temp[high] = 1;

            if (temp[low] == memory[low])
            {
                fakeIndex = low;
                WeightDifference = 0;
                isFound = true;
                return make_pair(fakeIndex,
WeightDifference);

            }
            else
            {
                fakeIndex = high;
                WeightDifference = 1;
                isFound = true;
                return make_pair(fakeIndex,
WeightDifference);
            }
        }
        else if (result == 1)
        {
            temp[low] = 1;
            temp[high] = -1;

            if (temp[low] == memory[low])
            {
                fakeIndex = low;
                WeightDifference = 1;
                isFound = true;
                return make_pair(fakeIndex,
WeightDifference);

            }

            else
            {
                fakeIndex = high;
                WeightDifference = 0;
                isFound = true;
                return make_pair(fakeIndex,
WeightDifference);


            }
        }



    }

    if (high - low == 2) // For 3 coins
    {
        result = weigh(coins, low, low, mid, mid);
        if (result == 0)
        {
            int result2 = weigh(coins, low, low, high,
high);
            if (result2 == 0)
            {
                // the 3 coins are genuine
                return make_pair(fakeIndex,
WeightDifference);
            }
            else
            {

                fakeIndex = high;
                isFound = true;
                if (memory[high] == 1)
                {
                    WeightDifference = 1;
                }
                else
                    WeightDifference = 0;
                return make_pair(fakeIndex,
WeightDifference);

            }
        }
        else if (result == -1)
        {
            temp[low] = -1;
            temp[mid] = 1;
            if (memory[low] == temp[low])
            {
                fakeIndex = low;
                isFound = true;
                if (memory[low] == 1)
                {
                    WeightDifference = 1;
                }
                else
                    WeightDifference = 0;
                return make_pair(fakeIndex,
WeightDifference);
            }
```

Dynamic Programming Algorithm Implementation (2)

```
else
            {
                temp[low] = 1;
                temp[mid] = -1;
                fakeIndex = mid;
                isFound = true;
                if (memory[mid] == 1)
                {
                    WeightDifference = 1;
                }
                else
                    WeightDifference = 0;
                return make_pair(fakeIndex,
WeightDifference);

            }
        }

        else if (result == 1)
        {
            temp[low] = 1;
            temp[mid] = -1;
            if (memory[low] == temp[low])
            {
                fakeIndex = low;
                isFound = true;
                if (memory[low] == 1)
                {
                    WeightDifference = 1;
                }
                else
                    WeightDifference = 0;
                return make_pair(fakeIndex,
WeightDifference);

            }
            else
            {
                fakeIndex = mid;
                isFound = true;
                if (memory[mid] == 1)
                {
                    WeightDifference = 1;
                }
                else
                    WeightDifference = 0;
                return make_pair(fakeIndex,
WeightDifference);

            }
        }
    }


    if ( ( Maxindex % 2 != 0 || NCoins %2 != 0 ) && high
% 2 == 0)
    {
        if ((high - low+1) % 2 != 0) // For odd coins
except 3
        {
            ThrownKeyFlag = true;

            if (NCoins % 2 != 0)
            {

                if (ThrownCoins[low] == -900)
                {
                    ThrownCoins[low] = low;
                }
                low = low + 1;

            }
            else if (NCoins % 2 == 0)
            {

                if (ThrownCoins[low] == -900)
                {
                    ThrownCoins[low] = low;
                }
                if (ThrownCoins[low + 1] == -900)
                {
                    ThrownCoins[low + 1] = high;
                }
                low = low + 1;
                high = high - 1;
                Thrown_maxIndex = high - 1;
            }



        }

    }

    if (result < 0)
    {
    result = weigh(coins, low, mid, mid + 1, high);
    }

    if (result == 0 && (high == MidIndex))
    {
        leftR = false; // Remove the remaining left
subtree
        mid = high;
        if (ThrownKeyFlag)
        {
            high = Thrown_maxIndex;
        }
```

Dynamic Programming Algorithm Implementation (3)

```
else
            high = Maxindex;
        // Coins are genuine
    }
    else if (result == 0 && (high = Maxindex) &&
!ThrownKeyFlag)
    {
        //All coins are genuine
        isFound = true;
        return make_pair(fakeIndex, WeightDifference);
    }
    else if (result == 0 &&  ThrownKeyFlag)
    {
        //All non thrown coins are genuine
        leftR = false;
        rightR = false;
    }

    else if (result == -1) // sum1 < sum2
    {
        for (int i = low; i <= mid; i++)
        {
            temp[i] = -1;
        }
        for (int j = mid + 1; j <= high; j++)
        {
            temp[j] = 1;
        }


        if (memory[NCoins] == 0)
        {
            for (int i = low; i <= mid; i++)
            {
                memory[i] = -1;
            }

            for (int i = mid + 1; i <= high; i++)
            {
                memory[i] = 1;
            }

            memory[NCoins] = 1;
        }

        else // Memory is filled --> Start comparing
values
        {
            for (int i = low; i <= mid; i++)
            {
                if (memory[i] != temp[i])
                {
                    // do nothing
                }
                else
                {
                    rightR = false;
                }
                memory[i] = temp[i];
            }

            for (int i = mid + 1; i <= high; i++)
            {
                if (memory[i] != temp[i])
                {
                    // do nothing
                }
                else
                {
                    leftR = false;
                }
                memory[i] = temp[i];
            }
        }
    }

    else if (result == 1) // sum1 > sum2
    {
        for (int i = low; i <= mid; i++)
        {
            temp[i] = 1;
        }
        for (int j = mid + 1; j <= high; j++)
        {
            temp[j] = -1;
        }

        if (memory[NCoins] == 0)
        {
            for (int i = low; i <= mid; i++)
            {
                memory[i] = 1;
            }

            for (int i = mid + 1; i <= high; i++)
            {
                memory[i] = -1;
            }
            memory[NCoins] = 1;

        }
```

Dynamic Programming Algorithm Implementation (4)

```
else
        {
            for (int i = low; i <= mid; i++)
            {
                if (memory[i] != temp[i])
                {
                    // do nothing
                }
                else
                {
                    rightR = false;
                }
                memory[i] = temp[i];
            }

            for (int i = mid + 1; i <= high; i++)
            {
                if (memory[i] != temp[i])
                {
                    // do nothing
                }
                else
                {
                    leftR = false;
                }
                memory[i] = temp[i];
            }

        }
    }


    if (leftR && !isFound)
    {
        pair<int, int> leftR = getFakeCoin(coins, low,
mid, memory);
    }
    if (rightR && !isFound)
    {
        pair<int, int> rightR = getFakeCoin(coins, mid +
1, high, memory);
    }

    if (high == Thrown_maxIndex && !isFound)
    {
        return checkThrownKeys(coins, ThrownCoins, mid);
    }

    else
        return make_pair(fakeIndex, WeightDifference);
}
```

Dynamic Programming Algorithm Implementation (5)

Complexity Analysis

The Worst case time complexity of this algorithm is ***O(log(NCoins) *NCoins)*** where NCoins is the number of coins provided by the user.

This is because in the algorithm in each iteration we divide the group of coins into two sub-groups and weigh them against each other. Based on the result of weighing the 2 groups, The algorithm will eliminate one of the sub-groups and will continue searching in the sub-group which isn't eliminated until the fake coin is found. This produces a recursion tree of height log(NCoins) and in each recursion we have a for loop that iterates over the memory and temp arrays which have the same size as the number of coins (NCoins) that's why the time complexity is ***O(log(NCoins) * NCoins)***

Sample Output

Sample 1 : Output for 8 coins where the fake coin is placed in index 2 with weight less than the genuine coins

Sample 2 : Output for 14 coins where the fake coin is placed in index 13 with weight less than the genuine coins

Sample 3 : Output for 12 coins where the fake coin is placed in index 4 with weight more than the genuine coins

Sample 4 : Output for 33 coins where the fake coin is placed in index 30 with weight more than the genuine coins

Sample 5 : Output for 6 coins where they are all genuine coins

Alternative Algorithm

The alternative algorithm used for this Task is a Divide-And-Conquer algorithm. This algorithm takes the set of coins and starts dividing it into left and right subtrees until it reaches 2 coins and compares them with each other and so on until it either :-

- *Find the fake coin and stop*
- *Doesn't find the fake coin and continues dividing.*

  This is very bad when all coins are genuine as it will go through the whole array of coins.

The algorithm has 3 base cases. The first base case is if there is only one coin then that coin is returned.

The second base case is if there are 2 coins, Then they are compared with each other.

The third base case is if there are 3 coins, Then we divide them into 2 sets of coins where the first set contains the first 2 coins while the second set contains the third. The 2 coins in the first set are compared with each other using the second base case. After comparing them with each other we can compare them with the third coin in the second set to find out whether they are all genuine or one of them is fake.

Alternative Algorithm Code Implementation

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

int weighings = 0;


int fakeIndex = -999;
int WeightDifference = -999; // 0 Means lighter, 1 Means
Heavier
int secondResult = -900;
int realWeight;

#define NCoins  25


int weigh(int(&coins)[NCoins], int low1, int high1, int
low2, int high2)

{

    weighings++;
    int sum1 = 0, sum2 = 0;

    for (int i = low1; i <= high1; i++)
        sum1 += coins[i];

    for (int i = low2; i <= high2; i++)
        sum2 += coins[i];
    if (sum1 == sum2) return 0;
    else if (sum1 < sum2) return -1;
    else return 1;

}


pair<int,int> getFakeCoin(int(&coins)[NCoins],int low,int
high)
{
    if (NCoins < 2)   // Invalid entries
    {
        return make_pair(-1, -1);
    }

    if (low == high) return
make_pair(fakeIndex,WeightDifference);



    if (high - low == 1) // Comparing between 2 coins
    {
        int result = weigh(coins, low, low, high, high);

        if (result == 0)
        {
```

Divide-And-Conquer Algorithm Implementation (1)

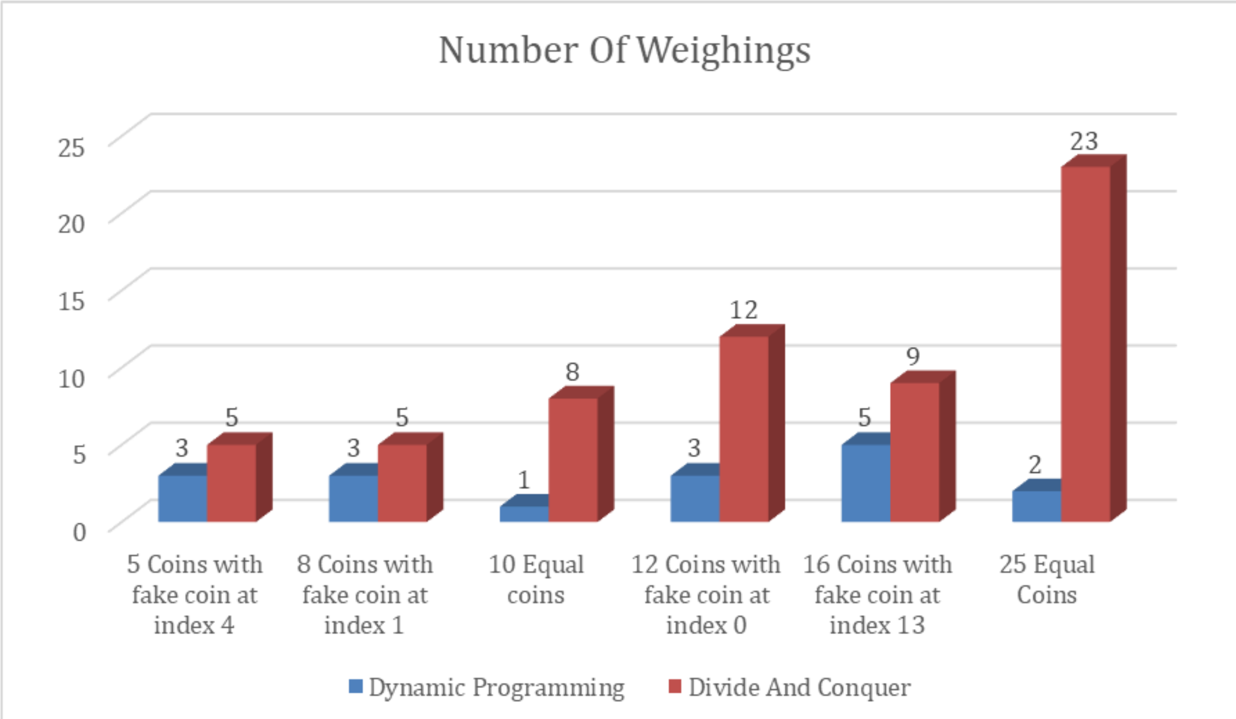
```
 // No Fake coin
        }
        else if (result == -1)
        {
            //Compare with 3rd coin
            if (high != NCoins-1)
            {
                 secondResult = weigh(coins, low, low,
high + 1, high + 1);
            }
            else
            {
                 secondResult = weigh(coins, low, low,
high - 2, high - 2);
            }
            if (secondResult == 0)
            {
                fakeIndex = high;
                WeightDifference = 1;
            }
            else
            {
                fakeIndex = low;
                WeightDifference = 0;
            }

        }
        else if (result == 1)
        {
            if (high != NCoins-1)
            {
                 secondResult = weigh(coins, low, low,
high + 1, high + 1);
            }
            else
                 secondResult = weigh(coins, low, low,
high - 2, high - 2);
            if (secondResult == 0)
            {
                fakeIndex = high;
                WeightDifference = 0;
            }
            else
            {
                fakeIndex = low;
                WeightDifference = 1;
            }
        }

    }

   if (high - low == 2) // Comparing 3 coins
    {
```

Divide-And-Conquer Algorithm Implementation (2)

```
int mid = (low + high) / 2;
        int result = weigh(coins, low, low, mid , mid);

        if (result == 0) // Both coins are equal,
Compare one of them with the 3rd
        {
            int result2 = weigh(coins, low, low, high,
high);
            if (result2 == 0)
            {
                // The 3 coins are genuine
            }
            else if (result2 == -1)
            {
                fakeIndex = high;
                WeightDifference = 1;
            }
            else if (result2 == 1)
            {
                fakeIndex = high;
                WeightDifference = 0;
            }
        }


    }


    int mid = (low+high) / 2;

        pair <int, int>leftSide = getFakeCoin(coins,
low, mid);
        pair <int, int> rightSide = getFakeCoin(coins,
mid+1, high);


        return make_pair(fakeIndex, WeightDifference);


}
```

Divide-And-Conquer Algorithm Implementation (3)

Comparison with an Alternative Algorithm



Conclusion

In all cases the dynamic-programming algorithm will have less number of weighings than the divide-and-conquer algorithm.

The dynamic programming algorithm is better than the divide-and-conquer algorithm as it will stop traversing the array of coins as soon as it finds the fake index. But in the divide-and-conquer algorithm, The traversing of the subtrees will continue even if the fake coin is found until the subtree has been divided into its base case which leads to way more number of weighings in the divide-and-conquer algorithm especially when they are all genuine coins as the divide-and-conquer algorithm will have to compare all of the coins.

In contrast, In the dynamic-programming algorithm, As soon as it finds that the first weighings are equal it will conclude that all the coins are genuine and it will not need to do more than 1 comparison. Incase the number of coins is odd and they are all genuine (Such as 25) it will just need to do 1 extra comparison for each thrown key (Which is still better than the divide and conquer algorithm).

Task 7

Assumptions

No Assumptions

Problem Description

A computer game has a shooter and a moving target. The shooter can hit any of n > 1 hiding spots 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 greedy algorithm that guarantees hitting the target.

Solution Steps

1. Shooter will start shooting at spot 2
2. Target will move randomly through the spots without being known by the shooter
3. If parity of the spot of the shooter is the same as the parity of spot of the target, then it is guaranteed that the target will be hit while progressing through the end of the array of spots
4. If parity of spot of shooter different from parity of spot of target then shooter will progress until reaching spot of N-1
5. Shooter will shot 2 times at spot N-1 to change parity and be the same parity as the target
6. While the shooter is progressing from spot N-1 to spot 2, it is guaranteed that the target will be hit as the target and the shooter having the same parity
7. Greedy Approach used by changing the parity of the shooter (local optimum achieved to reach a global optimum which is hitting the target)
8. Is is guaranteed that the target will be hit from 0 → 2n-4 shots either the first path 2 → N-1 if same parity or N-1 → 2 if the game started with different parities

Pseudocode

1. Initialize an array of integers spots with length n and assign values to represent the spots to shoot at.
2. Define n as a constant integer with value 5.
3. Initialize integer variable shots to 0, representing the number of shots taken.

4. Initialize integer variable shooter to the 3rd element of the spots array, representing the current location of the shooter.
5. Initialize integer variable dirshooter to 2, representing the direction of the shooter (0 for leftward and 1 for rightward).
6. Initialize integer variable target to the 2nd element of the spots array, representing the location of the target (unknown to the shooter).
7. Initialize integer variable dirtarget to 1, representing the index of the current target spot.
8. Initialize boolean variable endflag to false, representing whether or not the shooter has reached the rightmost spot.
9. Loop for i from 0 to 2n-5:

- Print the current dirtarget and dirshooter values.
- Increment the shots variable.
- If the shooter is at the same location as the target, break out of the loop.
- Generate a random integer randnum between 1 and 10 (inclusive).
- If dirtarget is greater than or equal to n-1, decrement it.
- Else if dirtarget is less than or equal to 0, increment it.
- Else if randnum is greater than 5, increment dirtarget.
- Else if randnum is less than or equal to 5, decrement dirtarget.
- Update the target variable to be the value of the spots array at the index dirtarget.
- If the dirshooter is equal to n-1 and endflag is false, set endflag to true and continue to the next iteration of the loop.
- Else if i is less than n-2, increment dirshooter and update the shooter variable to be the value of the spots array at the index dirshooter.
- Else if i is greater than or equal to n-2, decrement dirshooter and update the shooter variable to be the value of the spots array at the index dirshooter.

10. Print the message "target hit after shots shot(s)".

Code Implementation

```cpp
#include <iostream>

using namespace std;
const int n=5;
int spots[5]={1,2,3,4,5};
int main()
{
    int shots=0;
    int shooter = spots[2];
    int dirshooter=2;
    int target = spots[1]; //target location (not known by shooter)
    int dirtarget = 1; // current target spot
    bool endflag=false;
    /*
    start shooting at spot 2 then shoot concecutively until hitting spot n-1
    if target hit then target started at spot at even parity
    if not hit then hit spot n-1 again to change shooter parity
    then shoot from spot n-1 --> 2
    now the shooter parity is the same as the target parity
    so it is guaranteed to hit the target with maximum number of shots 2n-4
    */
    for(int i=0;i< 2*n-4;i++){
        cout << "target:" << dirtarget;
        cout << " shooter:" << dirshooter << endl;
        shots++;
        if(shooter==target)break;

        int randnum = rand()%10 + 1;
        if(dirtarget>=n-1)dirtarget--;
        else if(dirtarget<=0) dirtarget++;
        else if(randnum>5)dirtarget++;
        else if(randnum<=5)dirtarget--;
        target = spots[dirtarget];
        if(dirshooter==n-1 && endflag==false){
            endflag=true;
            continue;
        }
        else if(i<n-2)shooter = spots[++dirshooter];
        else if(i>= n-2) shooter = spots[--dirshooter];
    }
    cout << "target hit after " << shots << " shot(s)";
    return 0;
}
```

Complexity Analysis

The time complexity of this algorithm is O(n), where n is the length of the spots array. This is because the loop iterates 2n-5 times, and each iteration involves constant time operations. The array operations (such as getting the value at an index) are also constant time operations.

Sample Output

```
target:1 shooter:2
target:0 shooter:3
target:1 shooter:4
target:2 shooter:4
target:3 shooter:3
target hit after 5 shot(s)

...Program finished with exit code 0
Press ENTER to exit console.
```

Comparison with an Alternative Algorithm

Brute Force algorithm that shots randomly at any spot (random spot every time):

```cpp
*******************************************************************************/
#include <iostream>

using namespace std;
const int n=5;
int spots[5]={1,2,3,4,5};
int main()
{
    int target = spots[0]; //target location (not known by shooter)
    int dir=0;
    int randnum;
    int shots=0 ,counter=0;
    int i=0;
    while(true){
        counter++;
        //spots[i] is the current shot spot by the shooter
        //random number to randomize the movement of the target(left or right)
        randnum = rand()%10 + 1; // range 1-10
       // cout << randnum <<" rand"<< endl;
        if(dir>=n-1)dir--;
        else if(dir<=0) dir++;
        else if(randnum>5)dir++;
        else if(randnum<=5)dir--;
        target=spots[dir];
        int shooter = rand()%4;
        i = shooter;
        shots++;
        cout << "target:"<<target << " shooter:" << spots[i] << endl;
        if(target==spots[i])break;
    }
    cout << "target hitted successfully after "<<shots << " shot(s)" << endl;


    return 0;
}
```

Conclusion

The problem presented in the code is to simulate a shooting game where the shooter tries to hit a target at an unknown location, with the goal of hitting the target with the minimal number of shots possible. The algorithm implemented in the code involves shooting at a consecutive sequence of spots until hitting the target, and then changing the shooter's direction and shooting again until hitting the target again.

The algorithm appears to be effective in hitting the target with a small number of shots, although the exact number of shots required may depend on the initial positions of the shooter and the target, as well as the randomness of the target's movements. The time complexity of the algorithm is O(n), where n is the length of the spots array, and the space complexity is also O(n).

References

For Task 1 : I-Ping Chu, Richard Johnsonbaugh: Tiling and recursion. SIGCSE 1987: 261-263