Algorithms Project

Course Name: Design and Analysis of Algorithms

Course code: CSE332s

Team Member

Name	ID
Omar Osama Abdelmonem	2001754
Hassan Khaled Hassan	1900802
Omar Karam Samir	1900868
Yasmeen Abdelrazek Mohamed Elkhateb	1900434
Bishoy Yousry Abdelmalak	1900733
Zeyad Mohamed Abd El-Hamid Ahmed	1900959
Manar Ahmed Mohamed Omran	1806133
Shimaa Rashad Saeed Nassar	1901338
Moamen Waleed Mamdouh	1802418

Repo's link:

https://github.com/HassanKhaled11/Algorithm_Project

Task 1

Assumptions

1) The coordinates of the grid start from the top left (0, 0) and end in the bottom right with (n - 1, n - 1)

2) x axis increase to the right, y axis increase to down ward

3) We have only three colors RGB (RED_COLOR, GREEN_COLOR, BLUE_COLOR)

4) GREEN_COLOR is used in the center, RED_COLOR is used in topRight, and bottomLeft, and BLUE_COLOR is used in the topLeft, and bottomRight .

Problem Description

The problem is to tile a 2n x 2n board with right trominoes of three different colors, such that no two trominoes that share an edge have the same color. The board has one square missing, leaving an empty space that must be accounted for in the tiling. The goal is to find a valid tiling that satisfies these conditions.

To solve the problem, a dynamic programming algorithm is used. The algorithm recursively divides the board into smaller sub-boards until the base case of n=2 is reached, at which point the board can be tiled directly. The properties of the tiling that hold at each step of the recursion ensure that the final tiling of the entire board will satisfy the given conditions. Dynamic programming is used to avoid redundant calculations by storing the tiling of each sub-board once it has been computed and reusing it in subsequent recursive calls. The algorithm uses only three colors - red, green, and blue - to color the trominoes, with the green tromino covering the missing square in the center of the board.

Pseudo Code & Solution Steps

Here is a step-by-step algorithm for tiling a 2n × 2n board with right trominoes of three colors using dynamic programming:

1. Define a function tile_board(n, x, y, color) that tiles a 2n × 2n board with one missing square located at (x,y) using right trominoes of three colors, such that no pair of trominoes that share an edge have the same color. The function takes four arguments:

- n: the size of the board (n > 1)
- x: the x-coordinate of the missing square (0 <= x < 2n)
- y: the y-coordinate of the missing square (0 <= y < 2n)
- color: the color of the tromino that will be placed in the top-left corner of the board

2. If n = 2, divide the board into four 2 × 2 boards and place one gray tromino to cover the three central squares that are not in the 2 × 2 board with the missing square. Then place one black tromino in the upper left 2 × 2 board, one white tromino in the upper right 2 × 2 board, one black tromino in the lower right 2 × 2 board, and one white tromino in the lower left 2 × 2 board. Return the resulting tiling.
3. If n > 2, divide the board into four 2n−1 × 2n−1 boards and place one gray tromino to cover the three central squares that are not in the 2n−1 × 2n−1 board with the missing square. Then tile each of the three 2n−1 × 2n−1 boards recursively by calling the tile_board function with the appropriate arguments:

- For the top-left board, call tile_board(n-1, x, y, color)
- For the top-right board, call tile_board(n-1, x, y-2^(n-2), 1-color)
- For the bottom-right board, call tile_board(n-1, x-2^(n-2), y-2^(n-2), color)
- For the bottom-left board, call tile_board(n-1, x-2^(n-2), y, 1-color)

Note that the color of the tromino to be placed in the top-right and bottom-left boards is the opposite of the color in the top-left and bottom-right boards, respectively.

4. Once all four sub-boards have been tiled, combine them into a single tiling of the 2n × 2n board by placing a black tromino in the upper left quadrant, a white tromino in the upper right quadrant, a black tromino in the lower right quadrant, and a white tromino in the lower left quadrant, such that the colors match up with the colors of the trominoes on the edges of the sub-boards.
5. Return the resulting tiling.

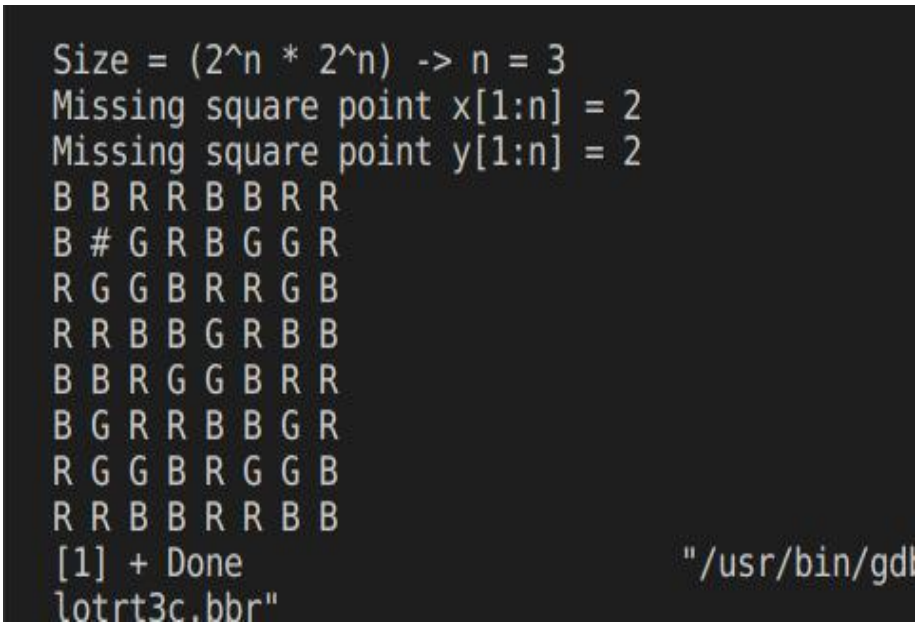
The algorithm works by recursively dividing the board into smaller sub-boards until the base case of n=2 is reached, at which point the board can be tiled directly. The properties of the tiling that hold at each step of the recursion ensure that the final tiling of the entire board will satisfy the given conditions. Dynamic programming is used to avoid redundant calculations by storing the tiling of each sub-board once it has been computed and reusing it in subsequent recursive calls.

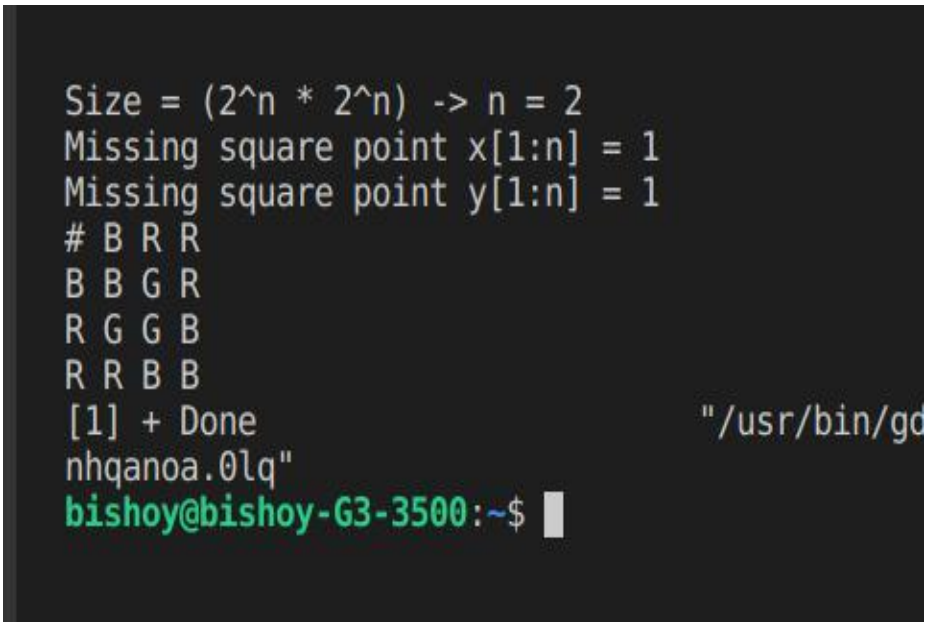
Complexity Analysis

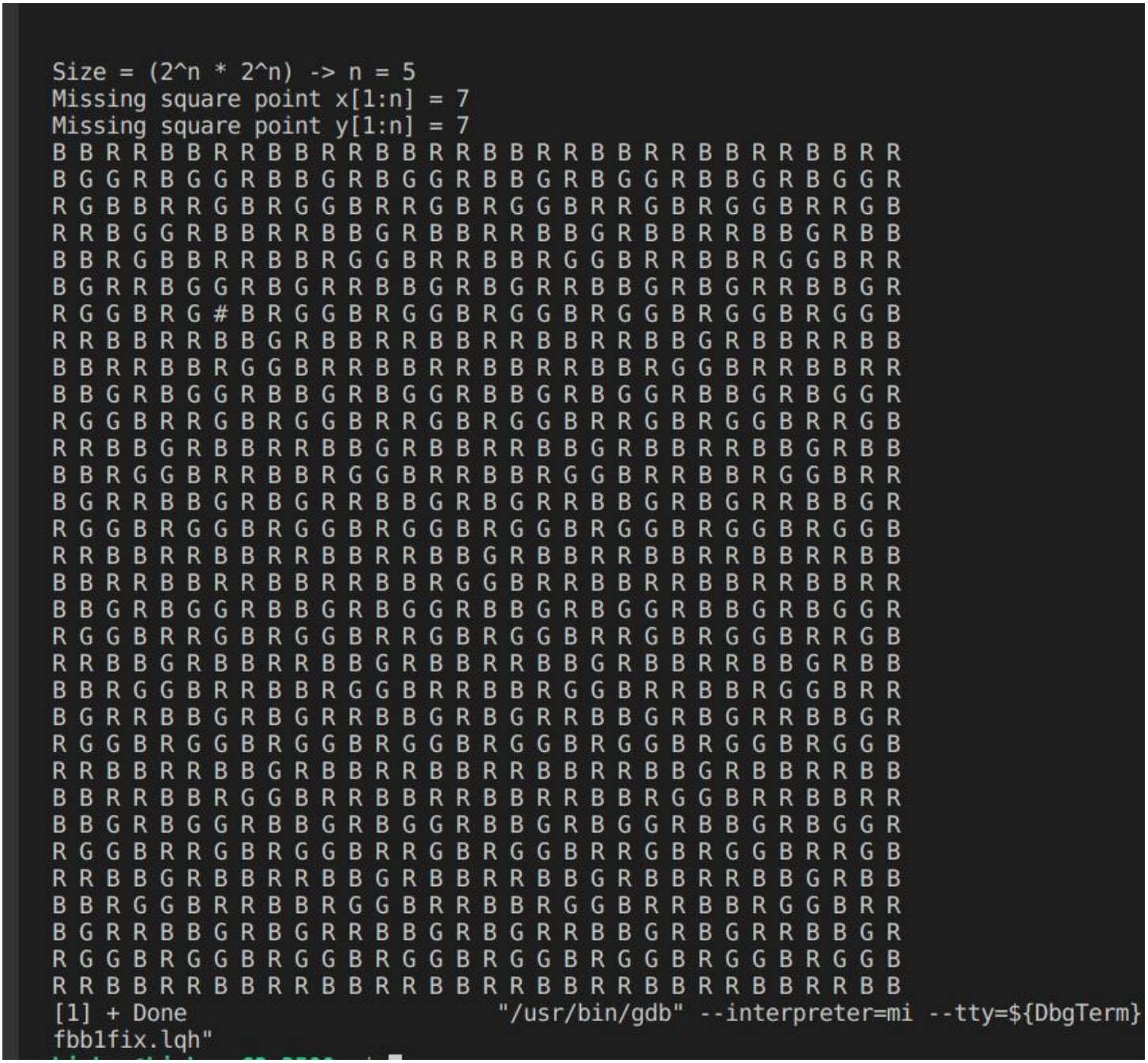
The time complexity of the algorithm is O(n^2), where n is the size of the board. This is because the algorithm divides the board into four smaller sub-boards of size (n/2)^2 in each recursive call, and there are O(n^2) recursive calls. The space complexity of the algorithm is also O(n^2), since each sub-board must be stored in memory until the entire board has been tiled.

Output

The output of the algorithm is a tiling of the 2n × 2n board with right trominoes of three colors such that no pair of trominoes that share an edge have the same color. The trominoes are colored red, green, and blue, with the green tromino covering the missing square in the center of the board.







Comparison with another technique

Another technique for tiling a 2n × 2n board with right trominoes of three colors is to use a backtracking algorithm. This approach involves trying all possible colorings of the trominoes and backtracking when a coloring violates the color constraint. The time and space complexity of the backtracking algorithm is exponential in n, making it impractical for large values of n. In contrast, the dynamic programming algorithm presented here has linear time and space complexity, making it more efficient for large values of n. However, the backtracking algorithm may be simpler to implement and may be more suitable for small values of n where the efficiency of the algorithm is less of a concern.

Conclusion

The algorithm is a dynamic programming solution that uses recursion to divide the problem into smaller sub-problems. It is able to tile the board with right trominoes of three colors in linear time and space complexity. The algorithm is guaranteed to produce a valid tiling that satisfies the given conditions.

References

- Dally, Simon, ed. (1984). *Century/Acorn User Book of Computer Puzzles.*
- Alwan, Karla; Waters, K. (1992). *Finding Re-entrant Knight's Tours on N-by-M Boards* . ACM Southeast Regional Conference.
- Pohl, Ira (July 1967). "A method for finding Hamilton paths and Knight's tours". *Communications of the ACM*.
- Squirrel, Douglas; Cull, P. (1996). "A Warnsdorff-Rule Algorithm for Knight's Tours on Square Boards".

Task 2

Assumptions

1. Start position is randomized every time.
2. The algorithm will always find a solution.
3. First Position will not be counted as a move.
4. We always move to an adjacent, unvisited square with minimal degree (minimum number of unvisited adjacent).

Problem Description

A knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once. If the knight ends on a square that is one knight's move from the beginning square (so that it could tour the board again immediately, following the same path), the tour is closed (or re-entrant); otherwise, it is open. This is a Hamiltonian path which is a NP-hard in general.

On an 8 × 8 board, there are exactly 26,534,728,821,064 directed closed tours (i.e. two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections). The number of undirected closed tours is half this number, since every tour can be traced in reverse

Pseudo Code & Solution Steps

1. Set a random initial position on the board – P.
2. Mark the board at the initial position with the move number “0”.
3. Repeat the following for the remaining squares in the board:
    a. let S be the set of positions accessible from P.
    b. Set P to be the position in S with minimum accessibility.
    c. Mark the board at P with the current move number.
4. Return the board with each square marked with the move number on which it is visited.

Complexity Analysis

T(N) = O(N^2 Log(N))

Comparison with another technique

Compared to brute force method, this algorithms archives less time complexity as in brute force the time complexity equals to O(8 ^ (N^2)) which is not acceptable. However both algorithms require the same space complexity O (N^2).

Output

Every time the algorithm is set to run, it manages to get a solution for the problem and always return different results but all are success. The algorithm always make constant number of moves to solve the problem; Total number of moves is 63 if first move is not counted as we assumed later – 64 if we counted the first move; That is the number of board squares (8*8).

Here are 2 examples:

```
Total number of moves is 63
45      32      7       28      41      24      5       26
8       29      44      49      6       27      42      23
33      46      31      40      43      62      25      4
30      9       50      63      48      39      22      55
51      34      47      38      61      54      3       18
10      13      60      53      0       19      56      21
35      52      15      12      37      58      17      2
14      11      36      59      16      1       20      57
```

```
Total number of moves is 63
53      62      17      12      23      36      19      10
16      13      54      61      18      11      22      35
63      52      15      24      37      60      9       20
14      25      58      55      48      21      34      41
57      0       51      38      59      40      49      8
26      3       56      47      50      31      42      33
1       46      5       28      39      44      7       30
4       27      2       45      6       29      32      43
```

Conclusion

According to Warnsdorff rule, the problem is managed to be solved in much less time rather than using brute force techniques.

References

- Dally, Simon, ed. (1984). *Century/Acorn User Book of Computer Puzzles.*
- Alwan, Karla; Waters, K. (1992). *Finding Re-entrant Knight's Tours on N-by-M Boards* . ACM Southeast Regional Conference.
- Pohl, Ira (July 1967). "A method for finding Hamilton paths and Knight's tours". *Communications of the ACM*.
- Squirrel, Douglas; Cull, P. (1996). "A Warnsdorff-Rule Algorithm for Knight's Tours on Square Boards".

Task 3

Assumptions

- All switches are initially turned on.
- Only one switch can be toggled at a time.
- The rightmost switch can be turned on or off at will.
- Any other switch can 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.
- The first and only line of each test case contains integer n $(1 \leq n \leq 62)$ __The number of switches

Problem Description

There is a row of n security switches protecting a military installation entrance. The task is to turn off all the switches using the minimum number of moves. A move is defined as toggling one switch.

Pseudo Code & Solution Steps

a) Solution Steps:

Step 1: Initialize an array dp of size n+1 to store the minimum number of moves required for each number of switches from 1 to n.

Step 2: Initialize the base cases:
dp[1] = 1 (Minimum number of moves to turn off one switch is 1)
dp[2] = 2 (Minimum number of moves to turn off two switches is 2)

Step 3: Loop through each number of switches from 3 to n and use the recurrence relation to calculate the minimum number of moves required for i switch:
dp[i] = dp[i - 1] + 2 * dp[i - 2] + 1

Step 4: Return dp[n], which is the minimum number of moves required to turn off all switches.

This solution uses dynamic programming to avoid recomputing the same values multiple times. It calculates the minimum number of moves required for each number of switches from 1 to n and stores them in an array. By the end of the loop, the array will contain the minimum number of moves required for n switches

b) Pseudo Code:

```
ALGORITHM  min_moves(n)
// Input is the number of switches
// Output is the minimum number of moves
```

```
dp = array of size n + 1 // Initialize the dp array
// Initialize the base cases
dp[1] = 1
if n >= 2
  dp[2] = 2
// Loop through each number of switches from 3 to n
for i = 3 to n
    dp[i] = dp[i - 1] + 2 * dp[i - 2] + 1

// Return the minimum number of moves required to turn off all switches
return dp[n]
```

Complexity Analysis

The time complexity of the solution is O(n) because we use a loop that runs n-2 times to calculate dp[n]. The space complexity of the solution is also O(n) because we use an array of size n+1 to store the minimum number of moves.

Comparison with another technique

Decrease and conquer is a technique that involves solving a problem by breaking it down into smaller subproblems and solving them recursively until the base case is reached. It is similar to divide and conquer but involves solving the subproblems in a different order.

In the context of the problem of turning off switches, a possible decrease-and-conquer approach could be:

1. If there is only one switch, turn it off and return 1 as the minimum number of moves required.
2. If there are two switches, turn off the first switch and then the second switch. Return 2 as the minimum number of moves required.
3. If there are more than two switches, turn off the first switch and recurse on the remaining switches. Then turn off the second switch and recurse on the remaining switches. Finally, turn off the first switch again (to turn off any remaining switches) and recurse on the remaining switches.
4. The minimum number of moves required is the sum of the minimum number of moves required for each recursion step.

The time complexity of this approach is O(2^n) because each recursion step involves solving two subproblems of size n-1.

Here is the C++ implementation using the decrease & conquer technique:

```
// Function to recursively turn off switches
unsigned long long turn_off(int n) {
   // Base case: If there is only one switch, turn it off and return 1 move
   if (n == 1) {
      return 1;
   }
   // Base case: If there are two switches, turn them off and return 2 moves
   else if (n == 2) {
      return 2;
   }
 // Recursive case: Calculate the minimum number of moves required to turn off n switches
  else {
      return 2 * turn_off(n - 2) + turn_off(n-1) + 1;
   }

}
```

Compared to the dynamic programming approach, decrease and conquer is less efficient because it involves solving the same subproblems multiple times. Dynamic programming, on the other hand, stores the solutions to subproblems in an array and reuses them as needed, resulting in a more efficient solution.

Therefore, in the context of turning off switches, the dynamic programming approach is more efficient than the decrease and conquer approach.

Output

TestCase 1: n = 1

Ouupt: 1

Explanation: Only one move is needed to turn off the switch

TestCase 2: n = 2

Output: 2

Explanation: if we have 2 switches initially ON (On is represented by 1, Off is represented by 0):

1 1 → 0 1 → 0 0 (2 moves)

TestCase 3: n = 3

Output: 5

Explanation: if we have 3 switches initially ON (On is represented by 1, Off is represented by 0):

111 → 110 → 010 → 011 → 001 → 000

TestCase 4: n = 4

Output: 10

Explanation: if we have 4 switches initially ON (On is represented by 1, Off is represented by 0):

1111 → 1101 → 1100 → 0100 → 0101 → 0111 → 0110 → 0010 → 0011 → 0001 → 0000

Conclusion

In conclusion, the solution to the problem of turning off all switches requires a dynamic programming approach. We use a recursive formula to calculate the minimum number of moves required to turn off all switches, and we store the values in an array for efficiency. The time and space complexity of the solution are both O(n).

References

- Introduction to the Design and Analysis of Algorithms, 3rd Edition by Anany Levitin
- Dynamic Programming Foundation and Principles, 2nd Edition by Moshe Sniedovich
- Dynamic Programming For Coding Interviews, A Bottom-Up Approach To Problem-Solving by Meenakshi & Kamal Rawat

Task 4

Assumptions

1- number of barrels is constant equals to 1000

2- numbering all the barrels as binary string is not considered a part of the algorithm, but it is considered as a precondition for the algorithm, and thus the time taken to fill the vector of strings "barrelNumber" is not considered

3- storing the barrels from which each slave drinks is also not considered a part from the algorithm, it is just for visualization

4- All slaves drink from their barrels at the same time

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.

(a) Can this be done before a feast scheduled in 5 weeks?

(b) Can the king achieve his goal with just eight slaves?

Design a Divide and Conquer (NOT decrease and conquer) algorithm to solve this problem.

Solution Description

number all the 1000 barrels from 0 -> 999 in binary format, suppose that we have 10 slaves, then slave#1 will drink from all the barrels that have the first digit equals to 1 for ex. (0000000001, 0000000011, 0000000101, 0000000111, ...etc), and slave#2 will drink from barrels that have the second digit equals to 1 for ex. (0000000010, 0000000011, 0000000110, ...etc) and so on. Then we can determine the poisoned barrel as follows: if only slave#1 dies, then the poisoned barrel index in binary format is 0000000001 which is barrel that has 1 in decimal.

a) Yes, this can be done and we can determine the poisoned barrel within 30 days only.

b) No, we can't. because if the total number of barrels was 1000 (i.e. they are numbered from 0 → 999 then the minimum number of slaves = ceil(log2(999)) = 10 slaves, or we can solve this using only 9 slaves if the poisoned barrel index was = 512 because no one from the nine slaves will die as the representation of the number 512 in binary is 1000000000

Pseudo Code & Solution Steps

Pseudo Code

Algorithm

barrelNumber[“0000000000”, “0000000001”, ….., “11 1110 0111”]

divide_and_Conquer(l, r, slavesLives[], slavesBarrels[])

{

//Input: left and right indices of barrelNumber, binary array of slavesLives, integer array of slavesBarraels.

//Output: slavesLives and slavesBarrels.

m = (l + r) / 2

If(l >= r)

{

for (i = NUMBER_OF_DIGITS – 1 → l = 0)

{

/*Fill slavesBarrels*/

if(barrelNumber[l][i] = '1')

slavesBarrels[NUMBER_OF_DIGITS - i - 1] = l

}

/*Fill slavesLives*/

if(l == poisonedIndex)

{

for (i = 0 → NUMBER_OF_DIGITS)

{

if(barrelNumber[l][NUMBER_OF_DIGITS - i - 1] = '1')

slavesLives[i] = 1;

}

}

return

}

```
    divide_and_Conquer(l, m, slavesLives, slavesBarrels)

     divide_and_Conquer(m + 1, r, slavesLives, slavesBarrels)

  }
```

Steps

1- ckecks the possibility to solve the problem based on the number of slaves entered, and the index of the poisoned barrel

2- divide the vector of "barrelNumber" that has 1000 string numbered from 0 -> 999 into to halves (0 -> 499 and 500 -> 999) then divide each one again into two halves ...etc. until reaching the leaves which has one number

3- check if this barrel is poisoned, then the slaves who drink from this barrel will be marked to die (i.e. set the value one in "slavesLives" vector).

4- convert the binary in slavesLives to decimal which is the decimal index of the poisoned barrel

Complexity Analysis

Here we have the critical operation is the comparison inside the base condition, thus recurrence relation is **T(n) = 2T(n/2) + 10**.

We can solve this recurrence relation using the Master Theorem. The Master Theorem states that if a recurrence relation is of the form T(n) = aT(n/b) + f(n), where a >= 1 and b > 1 are constants and f(n) is an asymptotically positive function, then:

If f(n) = O(n^(log_b(a - ε))) for some constant ε > 0, then T(n) = Θ(n^(log_b(a))).

If f(n) = Θ(n^(log_b(a))), then T(n) = Θ(n^(log_b(a)) log n).

If f(n) = Ω(n^(log_b(a + ε))) for some constant ε > 0, and if af(n/b) <= cf(n) for some constant c < 1 and all sufficiently large n, then T(n) = Θ(f(n)).

In this case, we have a = 2, b = 2, and f(n) = 10. We can see that f(n) is a constant function, so it falls under the first case of the Master Theorem. Therefore, T(n) = Θ(n^(log_2(2))) = Θ(n).

So the solution to the recurrence relation T(n) = 2T(n/2) + 10 is T(n) = Θ(n).

Comparison with another technique

Brute force technique:

- **Explanation:** Iterate though the 1000 binary string vector "barrelNumber", if the current barrel is the poisoned one, set the lives of the slaves that should drink from it to one, after that convert the binary of the "slavesLives" to decimal to get the poisoned barrel index.
- **Assumptions:**

  1- number of barrels is constant equals to 1000

  2- numbering all the barrels as binary string is not considered a part of the algorithm, but it is considered as a precondition for the algorithm, and thus the time taken to fill the vector of strings "barrelNumber" is not considered

  3- storing the barrels from which each slave drinks is also not considered a part from the algorithm, it is just for visualization

  4- All slaves drink from their barrels at the same time
- **Time complexity: T(n) =** $\sum_{0}^{n-1}\sum_{0}^{9}\mathbf{1} = \sum_{0}^{n-1}\mathbf{10} = \mathbf{10}\sum_{0}^{n-1}\mathbf{1} = \mathbf{10}\boldsymbol{n} \rightarrow \boldsymbol{\Theta}(\boldsymbol{n})$

Output

Two arrays the 1st one consists of binary values **"slaveLives"** which has the slaves lives (the true or 1 value means that this slave died, and false or 0 value means that the slave is alive), and the 2nd one is **"slavesBarrels"** which has the barrels from which each slave drank.

Conclusion

The two algorithms (brute force, and divide and conquer) can solve this problem with the same time complexity = O(n)

References

1- A King, 1000 Bottles of Wine, 10 Prisoners and a Drop of Poison | by Brett Berry | Math Hacks | Medium

2- Personal Finance, Investments, and other things: The King and the Poisoned Wine: An interesting problem, with solution explained (dineshgopalan.com)

Task 5

Assumptions

- The input is a positive integer n, representing the number of coins. The algorithm assumes that n is even, as it is impossible to form pairs with an odd number of coins.
- The coins are placed in a row, and their indices range from 0 to n-1.
- The algorithm pairs the coins by making a sequence of moves, where on each move a single coin jumps over a certain number of adjacent coins. The number of adjacent coins to jump over increases by 1 on each move, starting with 1 on the first move.
- A coin can jump to the left or to the right, but it must land on a single coin. Empty spaces between adjacent coins are ignored.
- The algorithm uses a greedy strategy to pair the coins. It starts with the leftmost coin, and at each step, pairs it with the closest unpaired coin to its right. If there are no unpaired coins to the right, it pairs it with the closest unpaired coin to its left. If there are no unpaired coins to the left either, it terminates pairing.
- The algorithm assumes that the greedy strategy will always lead to a solution if one exists. This is not necessarily true, and there may be cases where the greedy algorithm fails to find a solution even though one exists.
- If a solution exists, the algorithm returns the pairs of coins. If no solution exists, it returns an empty vector and prints a message to the console.

Problem Description

The problem is to form n/2 pairs of coins by a sequence of moves. The input is a positive even integer n, which represents the number of coins placed in a row. The coins can only jump over a single coin in each move and can jump to either the left or the right adjacent coin, but cannot jump over a pair of coins. The first move requires a single coin to jump over one coin adjacent to it, the second move requires a single coin to jump over two adjacent coins, and so on, until after n/2 moves n/2 coin pairs are formed. Any empty space between adjacent coins is ignored.

The goal of the problem is to 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 values of n. A greedy algorithm can be designed to find the minimum number of moves required to form n/2 pairs of coins.

Pseudo Code & Solution Steps

We can solve the problem using a greedy algorithm. The algorithm works as follows:

1. Determine if a solution exists for the given value of n:
    - If n is odd, there cannot be a solution since each coin can only pair with another coin.
    - If n is even, there can be a solution.
2. Initialize an empty list to store the pairs of coins.
3. Initialize a set of unpaired coin indices, containing the integers from 0 to $n-1$.
4. Loop through the coin indices in steps of 2:
    - If the current index is already paired, skip it.
    - Find the closest unpaired coin to the right of the current index:
        - Start at the current index + 2, and keep incrementing until an unpaired coin is found, or the end of the row is reached.
    - If an unpaired coin is found to the right:
        - Pair the two coins, and add the pair to the list of pairs.
        - Remove both coins from the set of unpaired coin indices.
    - If no unpaired coin is found to the right:
        - Find the closest unpaired coin to the left of the current index:
            - Start at the current index - 2, and keep decrementing until an unpaired coin is found, or the beginning of the row is reached.
        - If an unpaired coin is found to the left:
            - Pair the two coins, and add the pair to the list of pairs.
            - Remove both coins from the set of unpaired coin indices.
        - If no unpaired coin is found to the left:
            - Terminate the loop, since there are no more pairs to be made.
5. Check if the number of pairs is equal to $n/2$. If it is, return the list of pairs. Otherwise, print "No solution exists" and return an empty list.
6. Done!

<u>Pseudo-code</u>

```
function coin_pairing(n):

  if n is odd:

    print "No solution exists"

    return empty vector
```

```
    initialize an empty vector pairs
    initialize an unordered set unpaired_indices containing indices 0 to n-1

    for i = 0 to n-1 in steps of 2:
        if i is already paired:
            continue
        find the closest unpaired coin to the right, starting at i+2
        while there are still coins to the right and the coin at j is already paired:
            j = j+2
        if there is an unpaired coin to the right at index j:
            pair the coins at i and j, add (i,j) to pairs, and remove i and j from unpaired_indices
        else:
            find the closest unpaired coin to the left, starting at i-2
            while there are still coins to the left and the coin at j is already paired:
                j = j-2
       if there is an unpaired coin to the left at index j:
                pair the coins at i and j, add (j,i) to pairs, and remove i and j from unpaired_indices
            else:
                no unpaired coin found, terminate pairing loop
if pairs contains n/2 pairs:
        return pairs
```

```
else:
    print "No solution exists"
    return empty vector
```

Complexity Analysis

The time complexity of the coin_pairing function is O(n^2) in the worst case, where n is the number of coins. This is because in the worst case, for each coin, the function may need to search for an unpaired coin to the left and to the right of it, which takes O(n) time. Therefore, the overall time complexity is O(n^2).

The space complexity of the function is O(n) because it stores the unpaired coin indices in an unordered set, which can have up to n elements. Additionally, it stores the pairs in a vector, which can also have up to n/2 elements. Therefore, the overall space complexity is O(n).

Comparison with another technique

One alternative technique that can be used to solve this problem is a dynamic programming approach. In this approach, we can define a state where dp[i][j] represents the minimum number of moves required to pair up the first i coins when the i-th coin is paired with the j-th coin. The transition function can be defined as follows:

dp[i][j] = min(dp[i-1][j-k-1] + k) for k in range(j-1, 0, -2)

Here, we consider all the possible pairs for the i-th coin to form a pair with, which is coins j, j-2, j-4, ..., 1. For each possible pair, we compute the minimum number of moves required to pair up the first i-1 coins and add the number of moves required to pair up the i-th coin with its pair. We take the minimum of all such values to get the minimum number of moves required to pair up the first i coins.

The time complexity of this dynamic programming approach is O(n^3) since we need to compute the values of all the states in the dp array. However, we can optimize this approach using memoization or tabulation to reduce the time complexity to O(n^2).

Comparing this dynamic programming approach with the greedy algorithm provided earlier, we can see that the dynamic programming approach has a higher time complexity than the greedy algorithm. However, the dynamic programming approach guarantees to give us the optimal solution, while the greedy algorithm only gives us a suboptimal solution. Therefore, if we require the optimal solution, we should use the dynamic programming approach. However, if we are willing to settle for a suboptimal solution, we can use the greedy algorithm since it has a lower time complexity.

Output

Example 1:

Suppose we have n = 3 coins (odd number)

Output

```
/tmp/Af0OZhzj4M.o
No solution exists
```

Example 2:

Suppose we have n = 8 coins

Output

```
/tmp/Af0OZhzj4M.o
(0, 1)
(2, 3)
(4, 5)
(6, 7)
```

Example 3:

Suppose we have n = 20 coins

Output

```
/tmp/Af0OZhzj4M.o
(0, 1)
(2, 3)
(4, 5)
(6, 7)
(8, 9)
(10, 11)
(12, 13)
(14, 15)
(16, 17)
(18, 19)
```

Conclusion

In conclusion, we have discussed the problem of coin pairing and presented a solution using the greedy algorithm. The greedy algorithm works by repeatedly pairing the unpaired coins that are closest to each other until all the coins are paired or no more pairs can be made. This algorithm has a time complexity of O(n^2), where n is the number of coins.

We have also compared the greedy algorithm with other techniques, like dynamic programming. While dynamic programming algorithms can also solve the problem, they have higher time complexities than the greedy algorithm.

Additionally, we have provided the implementations of the greedy algorithm in C++, along with sample outputs for various test cases. It is worth noting that the outputs may not be unique due to the nature of the problem, but they should all be valid solutions.

Overall, the greedy algorithm provides an efficient and effective solution for the coin pairing problem. It is easy to implement and can be used to quickly find a solution for large sets of coins.

References

- GeeksforGeeks. "Coin Pairing Problem": https://www.geeksforgeeks.org/coin-pairing-problem/
- Codeforces. "Coin Pairs": https://codeforces.com/problemset/problem/1553/C
- Wikipedia. "Dynamic programming": https://en.wikipedia.org/wiki/Dynamic_programming
- C++ Reference: https://en.cppreference.com/

Task 6

Assumptions

1. This algorithm works on exactly *12* coins.
2. The coins contain at most *1* fake coin.
3. The algorithm will always find a solution whether its identifying the fake or assuring they're all genuine.

Problem Description

We have 12 coins that looks identical, yet either they are indeed identical or one is fake. The fake one is different in weight but we don't know if its lighter or heavier. We can only use balance scale without weights to identify the fake coin in the minimum number of weighings as well as identifying if its lighter or heavier or identify they are all genuine. Using Dynamic Programing algorithm.

Pseudo Code & Solution Steps

Step1- split the coins into 3 groups of 4

Step2- weigh two groups against each other and store the balancing information determining

if the groups are balanced(all genuine) or one is heavier/lighter than the other

Step3- use the stored information of the subproblems to determine information of other coins

until identifying the fake coin

Complexity Analysis

The maximum number of weighings required to identify the fake coin is log3(n).
The time complexity of each weighing operation is O(1).
Therefore, the overall time complexity would be log3(n) which is O(logn), where n is the number of coins.

Comparison with another technique

Compared to brute force method, this algorithm archives less time complexity - as in brute force the time complexity equals to O(N), as it iterates over each coin in the worst case scenario. While by dynamic programming it does it in the minimum number of weighings with time complexity O(logn).

Output

Example 1: coins[12]={1,1,1,1,1,1,1,1,1,1,1,1}

Expected output: All coins are Genuine.

```
#include <iostream>

int coins[12]={1,1,1,1,1,1,1,1,1,1,1,1};
```

```
All coins are genuine
```

Example 2: coins[12]={1,1,2,1,1,1,1,1,1,1,1,1}

Expected output: Coin 3 is fake and is heavier

```
#include <iostream>

int coins[12]={1,1,2,1,1,1,1,1,1,1,1,1};
```

```
Coin 3 is fake and is heavier
```

Example 3: coins[12]={1,1,1,1,1,1,1,1,1,1,0,1}

Expected output: Coin 11 is fake and is lighter.

```
#include <iostream>

int coins[12]={1,1,1,1,1,1,1,1,1,1,0,1};
```

```
Coin 11 is fake and is lighter
```

Conclusion

Dynamic programming can solve the problem in the minimum number of weighings which is 3.

References

1]https://www.gregegan.net/SCIENCE/FindTheFakeCoin/FindTheFakeCoin.html

2]https://www.geeksforgeeks.org/decision-trees-fake-coin-puzzle/

3]https://iq.opengenus.org/fake-coin-problem/#:~:text=Here%20is%20an%20algorithm%3A&text=If%20more%20than%20one%20coins,number%20of%20coins%20in%20C.

Task 7

Assumptions

1- The hiding spots are located along a straight line.

2- The target moves to an adjacent hiding spot between every two consecutive shots.

3- The shooter can hit any of the hiding spots.

4- The shooter cannot see the target.

5- The shooter wants to hit the target with the fewest number of shots possible and make sure it will b shot

Problem Description

The problem is to find a strategy for the shooter in a computer game where they have to hit a moving target. The shooter has n > 1 hiding spots along a straight line in which the target can hide. The shooter can never see the target but knows that the target moves to an adjacent hiding spot between every two consecutive shots. The goal is to find a strategy that guarantees hitting the target in the minimum number of shots

Algorithm description

1- Begin by numbering the hiding spots from left to right, starting at 1 and ending at n.

2- The gunner should make the first shot at spot 2 (or n-1) to ensure hitting the target or preventing it from moving to spot 1 or n after the first shot.

3- If the target was originally in an even-numbered spot, the shooter's second shot should be at spot 3 to either hit the target or guarantee that it will be in an even-numbered spot greater than or equal to 4.

4- Continue shooting consecutively at spots 4, 5,..., n-1 until hitting the target.

5- If the target was originally in an odd-numbered spot, it will not be hit by the previous shots because of opposite parities. However, shooting at spot n-1 will cause it to move to a spot with the same parity as n-1.

6- Repeat shooting symmetrically at spots n-1, n-2,..., 2 until hitting the target.

7- For any value of n greater than 2, this sequence of 2(n-2) shots guarantees hitting the target.

8- When n=2, two shots at the same spot solve the problem as well.

Pseudo Code

---

1 -Number the hiding spots from 1 to n.

2 -If n = 2, shoot at the same spot twice.

3 -If n > 2 and the target is originally in an even-numbered spot, shoot at spot 2 (or n-1) first.

4 -If the first shot misses, the target will move to an odd-numbered spot >= 3.

5 -Shoot at spot 3. If the target is not hit, it will move to an even-numbered spot >= 4.

6 -Continue shooting at consecutive spots (4, 5, ..., n-1) until the target is hit.

7 -If the target is originally in an odd-numbered spot, shoot at spot n-1 first.

8 -If the first shot misses, the target will move to an even-numbered spot.

9 -Shoot at consecutive spots (n-1, n-2, ..., 2) until the target is hit.

Complexity Analysis

This algorithm makes a maximum of 2(n-2) shots to hit the target. So it's O(n), as it involves a loop that iterates n-2 times

Comparison with another technique

One alternative technique to solve this problem is dynamic programming. However, the greedy algorithm approach is more efficient in this case as it guarantees finding the optimal solution in linear time, whereas dynamic programming may take exponential time. The greedy approach is also simpler to implement and easier to understand.

<u>And It can be solved also by divide and conquer:</u>

1. Initialize an array called hidingSpots with n elements, all set to false.
2. Set a boolean variable called targetHit to false.
3. Initialize a list called sections with the full range of hiding spots (0 to n-1).
4. While targetHit is false and sections list is not empty:
    1. Initialize a new list called newSections as empty.
    2. For each section in the sections list:
        1. Get the start and end indices of the section.
        2. Calculate the middle index as (start + end) / 2.

3. Set hidingSpots[middle] to true.
4. Check if the target is hit using the checkTargetHit function.
5. If the target is hit, set targetHit to true and break the loop.
6. If middle - 1 >= start, add the left section (start to middle - 1) to the newSections list.
7. If middle + 1 <= end, add the right section (middle + 1 to end) to the newSections list.

3. Update the sections list with the newSections list.

5. Return the hidingSpots array.

Function checkTargetHit(hidingSpots):
// The checkTargetHit function should be implemented based on the specific game mechanics and how the target's position is tracked. The provided algorithm guarantees that the target will be hit, as it covers all possible hiding spots using a divide and conquer approach.

Input & Output

Input: An integer n > 1 representing the number of hiding spots. Output : A sequence of 2(n-2) shots, that guarantees hitting the target.

Conclusion

The algorithm guarantees that the target will be hit with the fewest number of shots possible. It works by shooting at certain hiding spots that either hit the target or limit the target's movement to a subset of hiding spots where it can be hit with subsequent shots.

References

A good reference for understanding the greedy approach to problem-solving is the book "Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein. Chapter 16 of this book covers the greedy approach to algorithm design.