Lesson:

Backtracking-2

PW SKILLS

Prerequisite:

- Backtracking

Today's Checklist:

- Problem Solving on Backtracking

Problem 1: Sudoku Solver()

Consider a 9*9 2D array grid that is partially filled with numbers from 1 to 9. The Sudoku Solver problem is to fill remaining blocks with numbers from 1 to 9 so that every row, column and subgrid (3*3) contains exactly one instance of digits (1 to 9).

Example:
Input - (Unfilled cells are denoted as 0).

```
{ {3, 0, 6, 5, 0, 8, 4, 0, 0},
  {5, 2, 0, 0, 0, 0, 0, 0, 0},
  {0, 8, 7, 0, 0, 0, 0, 3, 1},
  {0, 0, 3, 0, 1, 0, 0, 8, 0},
  {9, 0, 0, 8, 6, 3, 0, 0, 5},
  {0, 5, 0, 0, 9, 0, 6, 0, 0},
  {1, 3, 0, 0, 0, 0, 2, 5, 0},
  {0, 0, 0, 0, 0, 0, 0, 7, 4},
  {0, 0, 5, 2, 0, 6, 3, 0, 0} }
```

Output -

```
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

Naive Approach – Try to generate all possible combinations and print the combination satisfying the condition.

Efficient Approach – We can solve this problem using Recursion:

- Create a function that will check if the assignment of the number to the cell is safe or not.
- Create a recursive function that will take the grid as input.
- Assign a number to an unfilled cell, after which one must check if it is safe to assign. If it is safe to assign, then call the function for all safe cases. If any recursive call returns true, simply return true. If no recursive call returns true, then return false.
- If there is no unassigned cell, return true.

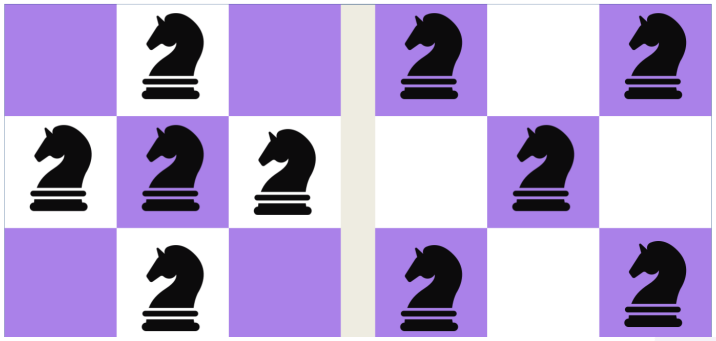
Code link
Time Complexity –
Time complexity will be O(9^(N*N)).
Space Complexity –
Space complexity will be O(N*N).

Problem 2: Place K-knights such that they do not attack each other

Given integers M, N and K, the task is to place K knights on an M*N chessboard such that they don't attack each other. The knights are expected to be placed on different squares on the board. A knight can move two squares vertically and one square horizontally or two squares horizontally and one square vertically. The knights attack each other if one of them can reach the other in single move. There are multiple ways of placing K knights on an M*N board or sometimes, no way of placing them. We are expected to list out all the possible solutions.

Examples:

Input: M = 3, N = 3, K = 5 **Output:** K A K A K A K A K A K K K A K A Total number of solutions : 2



Input: M = 5, N = 5, K = 13 **Output:** K A K A K A K A K A K A K A K A K A K A K A K A K
Total number of solutions : 1

Approach:

- This problem can be solved using backtracking. The idea is to place the knights one by one starting from first row and first column and moving forward to first row and second column such that they don't attack each other.
- When one row gets over, we move to the next row. Before placing a knight, we always check if the block is safe i.e. it is not an attacking position of some other knight.
- If it is safe, we place the knight and mark it's attacking position on the board else we move forward and check for other blocks.
- While following this procedure, we make a new board every time we insert a new knight into our board.
- This is done because if we get one solution and we need other solutions, then we can backtrack on our old board with the old configuration of knights which can then be checked for other possible solutions.
- The process of backtracking is continued till we get all our possible solutions.

Below is the implementation of the above approach:

Code link

Problem 3: Knight's Tour

Consider an N*N chessboard. The Knight's Tour problem is to print order when the Knight visits that block of the chessboard. Initially, the knight will be placed at the first block of the chessboard. The rule is that the Knight visits each block exactly once.

Example - Input - N = 8
Output -

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

Naive Approach – Try to generate all possible tours and print the tour that is satisfying the condition.

Efficient Approach –

- We first define two arrays row and col which contain the row and column numbers of all the possible moves of a knight on the chessboard.
- We define a function isSafe which checks whether a given move is safe or not. A move is safe if it is within the boundaries of the chessboard and the square has not already been visited.
- We define a function printSolution which prints the solution.
- We define a function solveKTUtil which is the main recursive function that tries all possible moves. It takes as input the current position of the knight, the current move number, and the current solution. It first checks if all the squares have been visited. If yes, it returns true. Otherwise, it tries all possible moves and checks if a solution exists by calling itself recursively. If a solution exists, it returns true. Otherwise, it backtracks by resetting the current square to -1 and returns false.
- We define a function solveKT which initializes the solution array and sets the first square to 0. It then calls the solveKTUtil function to solve the problem.
- Finally, we define the main function which calls the solveKT function.
- The time complexity of the above code is O(N * 2^N), where N is the length of the input array candidates. This is because for each element in the array, we have two choices: either include it in the combination or exclude it. So there can be a maximum of 2^N possible combinations, and we have to iterate over each of them to check if they sum up to the target. Sorting the array takes O(N log N) time, which is dominated by the backtracking algorithm.
- The space complexity of the above code is O(N), where N is the length of the input array candidates. This is because we use a temporary vector temp to store the current combination, and the maximum size of this vector is equal to the length of the input array. Additionally, the result vector can also have a maximum of O(2^N) elements, so the space complexity of the backtracking algorithm is also O(2^N). However, since we are only interested in the final result and not the intermediate combinations, we can consider the space complexity to be O(N).

Code link:https://pastebin.com/quYjcHkt

Output:

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

Problem 4: Combination Sum II – Find all unique combinations

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target. Each number in candidates may only be used once in the combination.

Note: The solution set must not contain duplicate combinations.

Examples:

Example 1:
Input: candidates = [10,1,2,7,6,1,5], target = 8
Output:
[
[1,1,6],
[1,2,5],
[1,7],
[2,6]]
Explanation: These are the unique combinations whose sum is equal to target.

Example 2:
Input: candidates = [2,5,2,1,2], target = 5
Output: [[1,2,2],[5]]
Explanation: These are the unique combinations whose sum is equal to target.

Solution:

Code link:

Output:
[[1 1 6][1 2 5][1 7][2 6]]

- The time complexity of the above code is O(N * 2^N), where N is the length of the input array candidates. This is because for each element in the array, we have two choices: either include it in the combination or exclude it. So there can be a maximum of 2^N possible combinations, and we have to iterate over each of them to check if they sum up to the target. Sorting the array takes O(N log N) time, which is dominated by the backtracking algorithm.
- The space complexity of the above code is O(N), where N is the length of the input array candidates. This is because we use a temporary vector temp to store the current combination, and the maximum size of this vector is equal to the length of the input array. Additionally, the result vector can also have a maximum of O(2^N) elements, so the space complexity of the backtracking algorithm is also O(2^N). However, since we are only interested in the final result and not the intermediate combinations, we can consider the space complexity to be O(N).

Upcoming Class Teasers:

- Trees