Technique: Backtracking

Level: Medium

Sudoku Solver: **Given a Sudoku board, find a solution. The board can have some squares filled out already. You have to fill the rest of the squares.**

(Rules of Sudoku are as follows: In each column, row and 3 x 3 square, you cannot have duplicate numbers. Also, only numbers 1-9 are allowed.)

Questions to Clarify:
Q. How is the input presented?
A. You're given a 2D array that represents the board. Each empty square has a value of 0.

Q. Will the board be of a fixed size?
A. Yes, the board will always be 9X9.

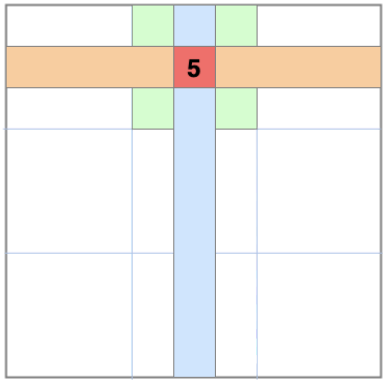
Q. How do you want the output?
A. Fill up the board with a valid answer.

Q. What if there is no solution. What do we return?
A. You can throw an error if no solution is found, i.e, the board is invalid.

Solution:
We use Backtracking here. We start with the first index - *[0,0]*. We try to place all 9 numbers in the box. We only place a number if it doesn't violate the rules, i.e, it's unique in that row, column and 3x3 box.

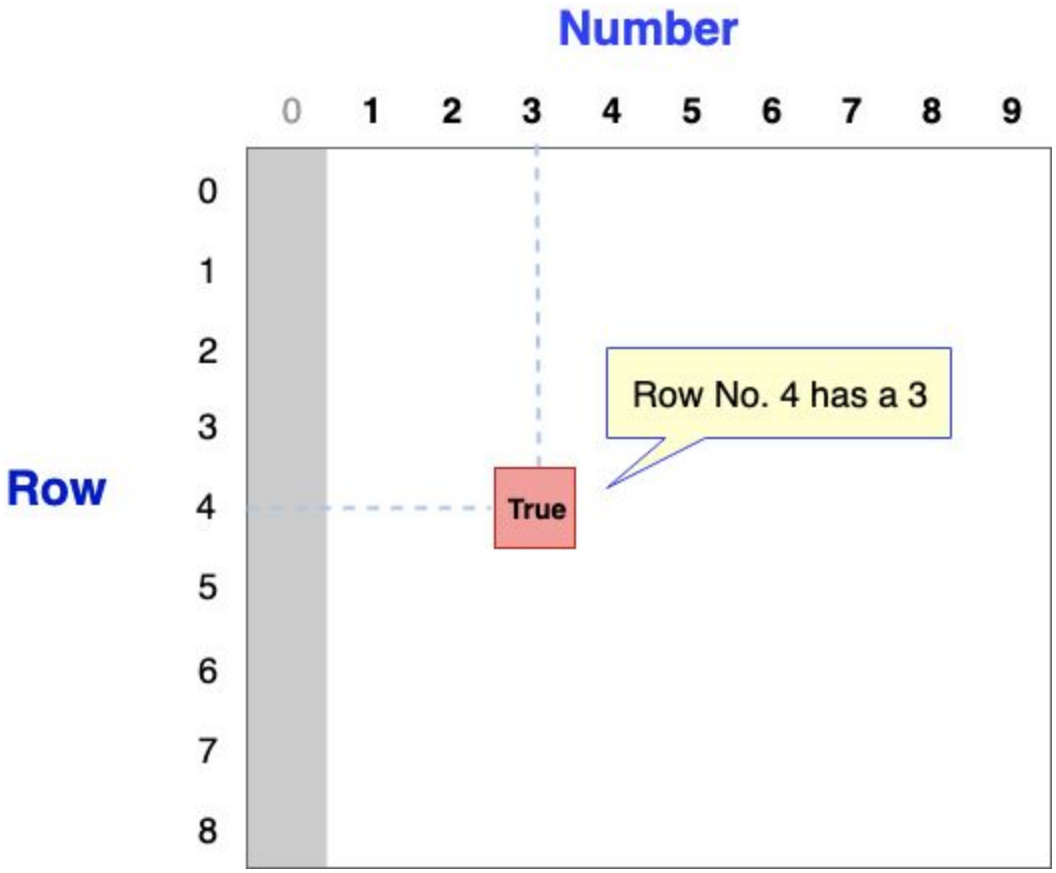
After placing each number, we recurse to the next index to check if there is a valid solution from there. Eventually, if we reach the end, it means we have found a an answer.

This is a case of simple backtracking. However, the key to this problem is quickly checking if a number is valid, i.e, it is unique in that row, column and 3x3 box.



The brute force way to do this - let's say we're checking if the number 5 is valid at index *[i,j]*. We loop through the entire row *i* and check if 5 is not taken. We do the same for column *j* and the 3x3 box around *[i,j]*. However, this can be improved by maintaining an index.

We can keep a boolean matrix that keeps track of numbers in each row:



We keep two more matrices for columns and 3x3 boxes.
We can number the 3x3 boxes as follows:

0	1	2
3	4	5
6	7	8

Now, to check if 5 is valid at index *[i,j]*, we simply assert the following:

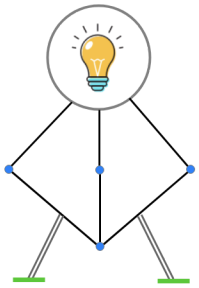
```
row[i][5] == false
column[j][5] == false
box[boxNumber(i,j)][5] == false
```

If all of the above are false, 5 can be placed there.

`boxNumber(i,j)` is a function we can implement. It finds the box number for an index. In an interview, we can suggest to the interviewer that we'll implement it later. This way, we're focusing on the core backtracking algorithm instead of getting distracted by helper functions.

Now that we have the index down, our algorithm can do the job quickly. Below, we summarize the backtracking function using the spaceship.

Backtracking Spaceship:



Core Idea	Try a number in an index, then go to the next index.
Steps/Splits	Possibly 9 splits per index - one for each number we try.
Converge/Collect	If any number returns true, return true.
Memoization	No possibility for memoization because each call is a unique combination. We're not repeating any work.
Base Cases	If we reach the end of the matrix, return true. If the index's number is provided by the board (i.e, *a[i][j]* is not 0), move to the next index.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a
few lines quickly to plan out your solution. Your actual solution should be in
a real language and use good syntax.)

solveSudoku(board, i, j, checker):
     if i == board.length:
          return true, we've found the solution

     if board[i][j] is provided:
          // recurse to next index
          return solveSudoku(board, next_i, next_j, checker):

      for candidate: 1 to 9 {
          if (candidate can be placed) {
              place candidate in checker
              board[i][j] = candidate

              if solveSudoku(board, next_i, next_j, checker):
                  return true

              remove candidate from checker
              board[i][j] = 0 // remove from board
          }
      }
      // no solution found
      return false
```

Test Cases:

Edge Cases: empty board, invalid board (no solution possible)
Base Cases: board with 1 element provided
Regular Cases: normal board with many elements provided

Time Complexity: $O(9^n)$, where *n* is the number of squares on the board.

Why 9^n? Because we do at most 9 splits each time, and we do this up to *n* times (for all *n* squares).
In this case, since the board size is fixed, our complexity is technically *O(1)* since *n* is constant.

Space Complexity: O(n), where *n* is the number of squares on the board.
We use this space both on the recursion stack and on the index (board checker).

```
public static void solveSudoku(int[][] board) {
    BoardChecker checker = new BoardChecker(board);

    boolean success = solveSudoku(board, 0, 0, checker);
    if (!success) {
        throw new IllegalArgumentException("Board has no solution");
    }
}

public static boolean solveSudoku(int[][] board, int i, int j, BoardChecker
checker) {
    if (i == board.length)
        return true;

    Pair next = nextSquare(i, j);

    if (board[i][j] != 0)
        return solveSudoku(board, next.i(), next.j(), checker);

    for (int candidate = 1; candidate <= 9; candidate++) {
        if (checker.canPlace(i, j, candidate)) {

            // place candidate on square
            checker.place(i, j, candidate);
            board[i][j] = candidate;

            if (solveSudoku(board, next.i(), next.j(), checker))
                return true;

            // remove candidate from square
            checker.remove(i, j, candidate);
            board[i][j] = 0;
        }
    }

    // no solution found
    return false;
}

/*
 * Helper Code: Ask the interviewer if they want you to implement this.
 */

public static Pair nextSquare(int i, int j) {
    if (j == 8)
        return new Pair(i + 1, 0);
    else
```

```
        return new Pair(i, j + 1);
}

public static class BoardChecker {

    boolean[][] row = new boolean[9][10];
    boolean[][] column = new boolean[9][10];
    boolean[][] box = new boolean[9][10];

    public BoardChecker(int[][] board) {
        // add existing numbers to checker
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (board[i][j] != 0) {
                    place(i, j, board[i][j]);
                }
            }
        }
    }

    public boolean canPlace(int i, int j, int number) {
        if (row[i][number])
            return false;

        if (column[j][number])
            return false;

        if (box[boxNumber(i,j)][number])
            return false;

        return true;
    }

    public boolean place(int i, int j, int number) {
        if (!canPlace(i, j, number))
            return false;

        row[i][number] = true;
        column[j][number] = true;
        box[boxNumber(i,j)][number] = true;

        return true;
    }

    public void remove(int i, int j, int number) {
        row[i][number] = false;
        column[j][number] = false;
```

```
            box[boxNumber(i,j)][number] = false;
        }


        public int boxNumber(int i, int j) {
            // Note: (i/3) * 3 is not the same as i.
            // for int values, i/3 gets reduced to floor(i/3).
            return (i/3)*3 + (j/3);
        }
    }

    public static class Pair {
        int i;
        int j;

        public Pair(int i, int j) {
            this.i = i;
            this.j = j;
        }

        public int i() {
            return i;
        }

        public int j() {
            return j;
        }
    }
```

Technique: Backtracking

Level: Medium

***Word Break Problem*: Given a String S, which contains letters and no spaces, determine if you can break it into valid words. Return one such combination of words.**
You can assume that you are provided a dictionary of English words.

For example:

S = "ilikemangotango"

Output:

Return any one of these:
"i like mango tango"
"i like man go tan go"
"i like mango tan go"
"i like man go tango"

Questions to Clarify:
Q. Can I return the result as a list of strings (each string is a word)?
A. Yes

Q. What to return if no result is found?
A. Just return null.

Q. What if there are multiple possible results?
A. Return any one valid result.

Q. What if the String is empty or null?
A. Return null.

Solution:
We use the following recursion: iterate *i* from 0 to the end of the string, and
check if *s[0..i]* is a valid word. If it's a valid word, do the same for the remainder of the string.

For example:

We first iterate from 0. When *i* is 0, we find the first word - just "i".

```
i l i k e m a n g o t a n g o
```

We then repeat the process with the rest of the string:

```
result = ["i"]
```

l i k e m a n g o t a n g o

We now find a new word - "like". So we repeat the process for the remaining string:

result = ["i", "like"]

m a n g o t a n g o

Similarly, we find "mango", so we repeat the process:

t a n g o

result = ["i", "like", "mango"]

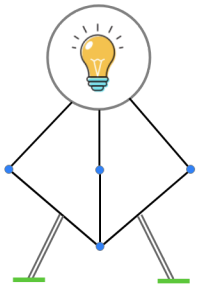
We find that "tango" is also a word.

Now, we have processed the entire string, so we return true. The result array contains a valid word combo:

result = ["i", "like", "mango", "tango"]

If there was no possible word combo, we would've reached the end without finding a valid word. We would return false in that case.

Backtracking Spaceship:



Core Idea	For every string, check if [0..i] forms a valid word, then do the same for [i+1..string.length-1].
Steps/Splits	Every time we find a valid word, we start a new function call with the rest of the string.
Converge/Collect	If any of the branches returns *true*, we return *true*. We also keep track of the resulting words in a list. At the end, the list will contain the result.
Memoization	If we don't find a result from a certain index,

	we can note that index, so we don't recurse through it again.
Base Cases	If we reached the end of the string, return true.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a
few lines quickly to plan out your solution. Your actual solution should be in
a real language and use good syntax.)

word_break(s, start, dict, list<string> result, memo)
     // base case
     if (start == s.length):
         return true

     // check memo
     if memo[start] is NOT_FOUND:
         return false;

     for i: start to s.length - 1 {
         if s[start..i] is a valid word:
             add s[start..i] to result
             // try recursing from i+1
             if word_break(s, i+1, dict, result, memo) returns true:
                 return true. result contains the valid words.
             else
                 remove s[start..i] from result
                 set memo[i+1] to NOT_FOUND, because there is no combo from i
     }

     // no result found
     return false;
```

Test Cases:

Edge Cases: string is empty or null
Base Cases: single character (in dict, not in dict), single word
Regular Cases: two words, no valid result, many words.

Time Complexity: $O(n^2)$ with memoization, because we go over each substring at most once.

Without memoization, the time complexity of this problem is too complicated for interviews. If you present the memoized problem, you don't generally have to worry about non-memoized time complexity anyway.

The reason this problem has a more difficult time complexity is the varying number of branches each function can call. Don't trust time complexities online for this problem. Different sites explain different complexities. It's safe to ignore this for interviews.

Space Complexity: O(n) with memoization - both on the memo and on the recursion stack.

Why O(n) on the recursion stack? Because in the worst case, the depth of the tree will be as deep as the length of the string. For example:

If S = "aaa" and dictionary contains "a", the first word combination will be "a a a", which will take 3 recursive calls.

```
public static List<String> wordBreak(String s, HashSet<String> dictionary) {
    if (s == null || s.isEmpty())
        return null;

    State[] memo = new State[s.length()];
    Arrays.fill(memo, State.UNVISITED);

    List<String> result = new ArrayList<String>();

    if (wordBreak(s, 0, memo, result, dictionary)) {
        return result;
    }

    return null;
}

public static boolean wordBreak(String s, int start, State[] memo,
List<String> result,
                                HashSet<String> dictionary) {
    if (start == s.length())
        return true;

    if (memo[start] == State.NOT_FOUND)
        return false;

    for (int i = start; i < s.length(); i++) {
        String candidate = s.substring(start, i + 1);

        if (dictionary.contains(candidate)) {
            result.add(candidate);
            if (wordBreak(s, i + 1, memo, result, dictionary)) {
                return true;
            } else {
                result.remove(result.size() - 1); // remove candidate
                memo[i+1] = State.NOT_FOUND;
```

```
            }
        }
    }

    return false;
}

/*
 * Helper code. Ask if your interviewer if they want you to implement this.
 */

/*
 * Note: Technically, you could use a boolean instead of saving 2 states.
 * However, we find that the State enum is more readable and doesn't take time
 * to implement.
 */
public enum State {
    UNVISITED,
    NOT_FOUND;
}
```