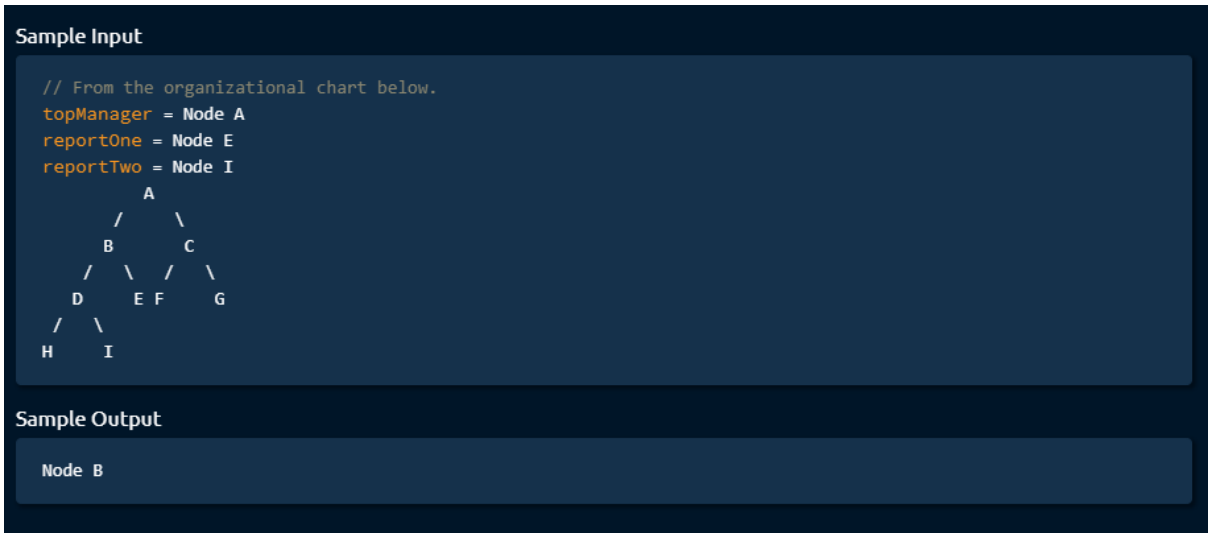
Topic - Recursion

Difficulty Medium: Lowest Common Manager:

You're given three inputs, all of which are instances of an OrgChart class that have a directReports property pointing to their direct reports. The first input is the top manager in an organizational chart (i.e., the only instance that isn't anybody else's direct report), and the other two inputs are reports in the organizational chart. The two reports are guaranteed to be distinct. -

Write a function that returns the lowest common manager to the two reports.



Solution:

Lowest Common Manager Hint:

Given a random subtree in the organizational chart, the manager at the root of that subtree is common to any two reports in the subtree. The lowest common manager to two reports in an organizational chart is the root of the lowest subtree containing those two reports. Find that lowest subtree to find the

lowest common manager. To find the lowest subtree containing both of the input reports, try recursively traversing the organizational chart and keeping track of the number of those reports contained in each subtree as well as the lowest common manager in each subtree. Some subtrees might contain neither of the two reports, some might contain one of them, and others might contain both; the first to contain both should return the lowest common manager for all of the subtrees above it that contain it, including the entire organizational chart.

Space-Time Complexity:

O(n) time | O(d) space - where n is the number of people in the org and d is the depth (height) of the org chart.

```
def getLowestCommonManager(topManager, reportOne, reportTwo):
    return getOrgInfo(topManager, reportOne, reportTwo).lowestCommonManager


def getOrgInfo(manager, reportOne, reportTwo):
    numImportantReports = 0
    for directReport in manager.directReports:
        orgInfo = getOrgInfo(directReport, reportOne, reportTwo)
        if orgInfo.lowestCommonManager is not None:
            return orgInfo
        numImportantReports += orgInfo.numImportantReports
    if manager == reportOne or manager == reportTwo:
        numImportantReports += 1
    lowestCommonManager = manager if numImportantReports == 2 else None
    return OrgInfo(lowestCommonManager, numImportantReports)


class OrgInfo:
    def __init__(self, lowestCommonManager, numImportantReports):
        self.lowestCommonManager = lowestCommonManager
        self.numImportantReports = numImportantReports


# This is the input class.
class OrgChart:
    def __init__(self, name):
        self.name = name
        self.directReports = []
```

Difficulty Medium: Interweaving Strings

Write a function that takes in three strings and returns a Boolean representing whether the third string can be formed by interweaving the first two strings. To interweave strings means to merge them by alternating their letters without any specific pattern. For instance, the strings "abc" and "123" can be interwoven as "alb2c3", as "abc123", and as "ab1c23" (this list is nonexhaustive). Letters within a string must maintain their relative ordering in the interwoven string.

Sample Input

```
one = "algoexpert"
two = "your-dream-job"
three = "your-algodream-expertjob"
```

Sample Output

```
true
```

Interweaving Strings Hint:

Try traversing the three strings with three different pointers to solve this problem. Declare three variables (i, j, and k, for instance) pointing to indices in the three strings, respectively, and starting at 0. At any given combination of indices, if neither the character at i in the first string nor the character at j in the second string is equal to the character at k in the third string, then the first two strings can't interweave to form the third one (at least not in whatever way led to the values of i, j, and k in question). If at any given combination of the indices i, j, and k mentioned in Hint #2, the character at i in the first string or the character at j in the second string is equal to the character at k in the third string, then you can potentially interweave the first two strings to get the third one. In such a case, try incrementing the two relevant indices (i and k or j and k) and repeating this process until you confirm whether or not the first two strings can be interwoven to form the third one. Try using recursion to implement this algorithm. you'll perform, in some cases, many computations multiple times. How can you use caching to improve the time complexity of this algorithm?

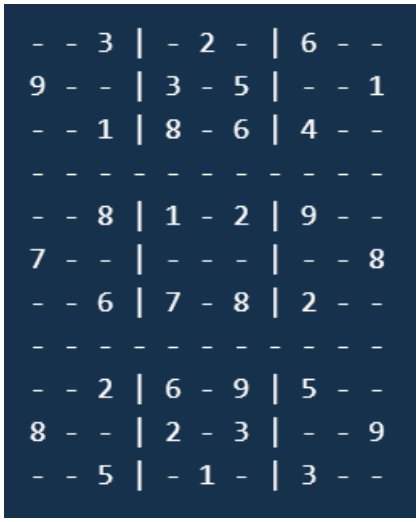
Space-TIme Complexity:

O(nm) time | O(nm) space - where n is the length of the first string and m is the length of the second string

```python
def interweavingStrings(one, two, three):
    if len(three) != len(one) + len(two):
        return False

    return areInterwoven(one, two, three, 0, 0)


def areInterwoven(one, two, three, i, j):
    k = i + j
    if k == len(three):
        return True

    if i < len(one) and one[i] == three[k]:
        if areInterwoven(one, two, three, i + 1, j):
            return True

    if j < len(two) and two[j] == three[k]:
        return areInterwoven(one, two, three, i, j + 1)

    return False
```

Difficulty Medium: Solve Sudoku

You're given a two-dimensional array that represents a 9x9 partially filled Sudoku board. Write a function that returns the solved Sudoku board. Sudoku is a famous number-placement puzzle in which you need to fill a 9x9 grid with integers in the range of 1-9. Each 9x9 Sudoku board is split into 9 3x3 subgrids, as seen in the illustration below, and starts out partially filled.



The objective is to fill the grid such that each row, column, and 3x3 subgrid contains the numbers 1-9 exactly once. In other words, no row may contain the same digit more than once, no column may contain the same digit more than once, and none of the 9 3x3 subgrids may contain the same digit more than once. Your input for this problem will always be a partially filled 9x9 two-dimensional array that represents a solvable Sudoku puzzle. Every element in the array will be an integer in the range of 8-9, where a represents an empty square that must be filled by your algorithm. Note that you may modify the input array and that there will always be exactly one solution to each input Sudoku board.

Sample Input

```
board =
[
  [7, 8, 0, 4, 0, 0, 1, 2, 0],
  [6, 0, 0, 0, 7, 5, 0, 0, 9],
  [0, 0, 0, 6, 0, 1, 0, 7, 8],
  [0, 0, 7, 0, 4, 0, 2, 6, 0],
  [0, 0, 1, 0, 5, 0, 9, 3, 0],
  [9, 0, 4, 0, 6, 0, 0, 0, 5],
  [0, 7, 0, 3, 0, 0, 0, 1, 2],
  [1, 2, 0, 0, 0, 7, 4, 0, 0],
  [0, 4, 9, 2, 0, 6, 0, 0, 7],
]
```

Sample Output

```
[
  [7, 8, 5, 4, 3, 9, 1, 2, 6],
  [6, 1, 2, 8, 7, 5, 3, 4, 9],
  [4, 9, 3, 6, 2, 1, 5, 7, 8],
  [8, 5, 7, 9, 4, 3, 2, 6, 1],
  [2, 6, 1, 7, 5, 8, 9, 3, 4],
  [9, 3, 4, 1, 6, 2, 7, 8, 5],
  [5, 7, 8, 3, 9, 4, 6, 1, 2],
  [1, 2, 6, 5, 8, 7, 4, 9, 3],
  [3, 4, 9, 2, 1, 6, 8, 5, 7],
]
```

Solve Sudoku Hint:

The brute-force approach to this problem is to generate every possible Sudoku board and to check each one until you find one that's valid. The issue with this approach is that there is 9^81 possible 9x9 Sudoku boards. This is an extremely large number, which makes it practically impossible to take this approach. How can you avoid generating every possible Sudoku board? Keep in mind that a Sudoku board doesn't need to be entirely filled to figure out if it's invalid and won't lead to a solution. Try generating partially filled Sudoku boards until they become invalid, thereby abandoning solutions that will never lead to a

properly solved board. This method is more formally known as backtracking. This involves attempting to place digits into empty positions in the Sudoku board and checking at each insertion if that newly inserted digit makes the Sudoku board invalid. If it does, then you try to insert another digit until you find one that doesn't invalidate the board. If it doesn't invalidate the board, you temporarily place that digit and continue to try to solve the rest of the board. If you ever reach a position where there are no valid digits to be inserted (every digit placed in that position leads to an invalid board), that means that one of the previously inserted digits is incorrect. Thus, you must backtrack and change previously placed digits. For more details on this approach, refer to the Conceptual Overview section of this question's video explanation.

Space-TIme Complexity:

O(1) time | O(1) space - assuming a 9x9 input board

```python
def solveSudoku(board):
    solvePartialSudoku(0, 0, board)
    return board


def solvePartialSudoku(row, col, board):
    currentRow = row
    currentCol = col

    if currentCol == len(board[currentRow]):
        currentRow += 1
        currentCol = 0
        if currentRow == len(board):
            return True  # board is completed

    if board[currentRow][currentCol] == 0:
        return tryDigitsAtPosition(currentRow, currentCol, board)

    return solvePartialSudoku(currentRow, currentCol + 1, board)


def tryDigitsAtPosition(row, col, board):
    for digit in range(1, 10):
        if isValidAtPosition(digit, row, col, board):
            board[row][col] = digit
            if solvePartialSudoku(row, col + 1, board):
                return True

    board[row][col] = 0
    return False


def isValidAtPosition(value, row, col, board):
    rowIsValid = value not in board[row]
    columnIsValid = value not in map(lambda r: r[col], board)

    if not rowIsValid or not columnIsValid:
        return False

    # Check subgrid constraint.
    subgridRowStart = (row // 3) * 3
    subgridColStart = (col // 3) * 3
    for rowIdx in range(3):
        for colIdx in range(3):
            rowToCheck = subgridRowStart + rowIdx
            colToCheck = subgridColStart + colIdx
            existingValue = board[rowToCheck][colToCheck]

            if existingValue == value:
                return False

    return True
```