Topic - Recursion

Difficulty Medium: Generate Div Tags

Write a function that takes in a positive integer numberOfTags and returns a list of all the valid strings that you can generate with that number of matched tags.

A string is valid and contains matched tags if for every opening tag , there is a closing tag that comes after the opening tag and that isn't used as a closing tag for another opening tag. Each output string should contain exactly same numberOfTags opening tags and numberOfTags closing tags.

For example, given numberOfTags=2. the valid strings to return would be:

[

" ",

" "

]

Note that the output strings don't need to be in any particular order.

Hint:

The brute-force approach to solve this problem is to generate every single possible string

that contains numberOfTags tags and to then check all of those strings to see if they're valid. Can you think of a better way to do this? To solve this problem optimally, you'll have to incrementally build valid strings by adding and tags to already valid partial strings. While doing this, you can avoid creating strings that will never lead to a valid final string by following two rules:

1. If a string has fewer opening tags than numberOfTags, it's valid to add an opening tag to the end of it.

2. If a string has fewer closing tags than opening tags, it's valid to add a closing tag to the end of it.

Using the rules defined in Hint #2, write a recursive algorithm that generates all possible valid strings. You'll need to keep track of how many opening and closing tags each partial string has available (at each recursive call), and you'll simply follow the rules outlined in Hint #2. Once a string has no more opening and closing tags available, you can add it to your final list of strings. Your first call to the function will start with an empty string as the partial string and with numberOfTags as the number of opening and closing tags available. For example, after you add an opening tag to a partial string, you'll recursively call the function like this: recursiveFunction (partialStringWithExtraOpening Tag, openingTags - 1, closing Tags)

Space-TIme Complexity:

O((2n)!/((n!((n + 1)!)))) time | O((2n)!/((n!((n + 1)!)))) space - where n is the input number

Solution:

```python
def generateDivTags(numberOfTags):
    matchedDivTags = []
    generateDivTagsFromPrefix(numberOfTags, numberOfTags, "", matchedDivTags)
    return matchedDivTags


def generateDivTagsFromPrefix(openingTagsNeeded, closingTagsNeeded, prefix, result):
    if openingTagsNeeded > 0:
        newPrefix = prefix + "<div>"
        generateDivTagsFromPrefix(openingTagsNeeded - 1, closingTagsNeeded, newPrefix, result)

    if openingTagsNeeded < closingTagsNeeded:
        newPrefix = prefix + "</div>"
        generateDivTagsFromPrefix(openingTagsNeeded, closingTagsNeeded - 1, newPrefix, result)

    if closingTagsNeeded == 0:
        result.append(prefix)
```

Difficulty Medium: Ambiguous Measurements

This problem deals with measuring cups that are missing important measuring labels. Specifically, a measuring cup only has two measuring lines. a Low (L) line and a High (H) line. This means that these cups can't precisely measure and can only guarantee that the substance poured into them will be between the Land H line. For example, you might have a measuring cup that has a Low line at 400ml and a high line at 435ml. This means that when you use this measuring cup, you can only be sure that what you're measuring is between 400ml and 435ml. You're given a list of measuring cups containing their low and high lines as well as one low integer and one high integer representing a range for a target measurement. Write a function that returns a Boolean representing whether you can use the cups to accurately measure a volume in the specified [Low, high] range (the range is inclusive).

Note that:

• Each measuring cup will be represented by a pair of positive integers [L, H]. where 0 <= L <= H

• You'll always be given at least one measuring cup, and the low and high input parameters will always satisfy the following constraint: 0 <= Low <= high.

• Once you've measured some liquid, it will immediately be transferred to a larger bowl that will eventually (possibly) contain the target measurement.

• You can't pour the contents of one measuring cup into another cup.

Sample Input

```
measuringCups = [
  [200, 210],
  [450, 465],
  [800, 850],
]
low = 2100
high = 2300
```

Sample Output

```
true
// We use cup [450, 465] to measure four volumes:
// First measurement: Low = 450, High = 465
// Second measurement: Low = 450 + 450 = 900, High = 465 + 465 = 930
// Third measurement: Low = 900 + 450 = 1350, High = 930 + 465 = 1395
// Fourth measurement: Low = 1350 + 450 = 1800, High = 1395 + 465 = 1860

// Then we use cup [200, 210] to measure two volumes:
// Fifth measurement: Low = 1800 + 200 = 2000, High = 1860 + 210 = 2070
// Sixth measurement: Low = 2000 + 200 = 2200, High = 2070 + 210 = 2280

// We've measured a volume in the range [2200, 2280].
// This is within our target range, so we return `true`.

// Note: there are other ways to measure a volume in the target range.
```

Hint:

Start by considering the last cup that you'll use in your sequence of measurements. If it isn't possible to use any of the cups as the last cup, then you can't measure the desired volume. If the cup that you're going to use last has a measuring range of [100, 110] and you want to measure in the range of [500, 550], then after you pick this cup as the last cup, you need to measure a range of [400, 440]. Now, you can simply pick the last cup you'll use to measure this new range. If you continue these steps, you'll eventually know if you're able to measure the entire range or not. This should give you an idea of how to solve this problem recursively. Try every cup as the last cup for the starting range, then recursively try to measure the new ranges created after using the selected last cups. If you ever reach a point where one cup can measure the entire range, then you're finished and you can measure the target range. Try to think of a way to optimize this recursive approach, since it might involve a lot of repeated calculations.

Space-TIme Complexity:

O(low * high * n) time | O(low * high) space - where n is the number of measuring cups

Solution:

```python
def ambiguousMeasurements(measuringCups, low, high):
    memoization = {}
    return canMeasureInRange(measuringCups, low, high, memoization)


def canMeasureInRange(measuringCups, low, high, memoization):
    memoizeKey = createHashableKey(low, high)
    if memoizeKey in memoization:
        return memoization[memoizeKey]

    if low <= 0 and high <= 0:
        return False

    canMeasure = False
    for cup in measuringCups:
        cupLow, cupHigh = cup
        if low <= cupLow and cupHigh <= high:
            canMeasure = True
            break

        newLow = max(0, low - cupLow)
        newHigh = max(0, high - cupHigh)
        canMeasure = canMeasureInRange(measuringCups, newLow, newHigh, memoization)
        if canMeasure:
            break

    memoization[memoizeKey] = canMeasure
    return canMeasure


def createHashableKey(low, high):
    return str(low) + ":" + str(high)
```

Difficulty Hard: Number Of Binary Tree Topologies

Write a function that takes in a non-negative integer n and returns the number of possible Binary Tree topologies that can be created with exactly n nodes. A Binary Tree topology is defined as any Binary Tree configuration, irrespective of node values. For instance, there exist only two Binary Tree topologies when n is equal to 2:a root node with a left node, and a root node with a right node. Note that when n is equal to 0, there's one topology that can be created: the none/null node.

Sample Input

```
n = 3
```

Sample Output

```
5
```

Hint:

Every Binary Tree topology of n nodes where n is greater than 0 must have a root node and an amount of nodes on both of its sides totaling n - 1. For instance, one such topology could have a root node, n - 3 nodes in its left subtree, and 2 nodes in its right subtree. Another one could have a root node, 4 nodes in its left subtree, and n - 3 nodes in its right subtree. How many distinct Binary Tree topologies with a root node, a left subtree of x nodes, and a right subtree of n - 1 - x nodes are there? Consider a Binary Tree topology of n nodes with a root node, x nodes in its left subtree, and n - 1 - x nodes in its right subtree, and call this topology T1. This is one of possibly many topologies of n nodes. Realize that for every distinct topology T-Lk of x nodes (i.e. for every distinct topology of T1's left subtree) there is a corresponding, distinct topology of as many nodes as T1. Similarly, for every distinct topology T-Rk of n - 1 - x nodes (i.e. for every distinct topology of T1's right subtree) there is a corresponding, distinct topology of as many nodes as T1. In fact, every unique combination of left and right topologies T-Lk and T-Rk forms a distinct topology of as many nodes as T1, and this is true for every x between 0 and n - 1. Realizing this, can you implement a recursive algorithm that solves this problem? Iterate through every number x between 0 and n - 1 inclusive; at every number x, recursively calculate the number of distinct topologies of x nodes and multiply that by the number of distinct topologies of n - 1 - x nodes. Sum all of the products that you calculate to find the total number of distinct topologies of n nodes. Can you improve the recursive algorithm mentioned above by using a caching system (memoization)? Can you implement the algorithm iteratively? Is there any advantage to doing so?

Space-TIme Complexity:

O(n^2) time | O(n) space - where n is the input number

Solution:

A.

```
def numberOfBinaryTreeTopologies(n):
    if n == 0:
        return 1
    numberOfTrees = 0
    for leftTreeSize in range(n):
        rightTreeSize = n - 1 - leftTreeSize
        numberOfLeftTrees = numberOfBinaryTreeTopologies(leftTreeSize)
        numberOfRightTrees = numberOfBinaryTreeTopologies(rightTreeSize)
        numberOfTrees += numberOfLeftTrees * numberOfRightTrees
    return numberOfTrees
```

B.

```python
def numberOfBinaryTreeTopologies(n, cache={0: 1}):
    if n in cache:
        return cache[n]
    numberOfTrees = 0
    for leftTreeSize in range(n):
        rightTreeSize = n - 1 - leftTreeSize
        numberOfLeftTrees = numberOfBinaryTreeTopologies(leftTreeSize, cache)
        numberOfRightTrees = numberOfBinaryTreeTopologies(rightTreeSize, cache)
        numberOfTrees += numberOfLeftTrees * numberOfRightTrees
    cache[n] = numberOfTrees
    return numberOfTrees
```

C.

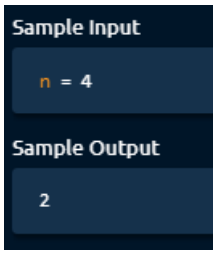
```python
def numberOfBinaryTreeTopologies(n):
    cache = [1]
    for m in range(1, n + 1):
        numberOfTrees = 0
        for leftTreeSize in range(m):
            rightTreeSize = m - 1 - leftTreeSize
            numberOfLeftTrees = cache[leftTreeSize]
            numberOfRightTrees = cache[rightTreeSize]
            numberOfTrees += numberOfLeftTrees * numberOfRightTrees
        cache.append(numberOfTrees)
    return cache[n]
```

Difficulty Hard: Non-Attacking Queens

Write a function that takes in a positive integer n and returns the number of non-attacking placements of n queens on an nxn chessboard. A non-attacking placement is one where no queen can attack another queen in a single turn. In other words, it's a placement where no queen can move to the same position as another queen in a single turn. In chess, queens can move any number of squares horizontally, vertically, or diagonally in a single turn.

```
+--+--+--+--+
|  |Q |  |  |
+--+--+--+--+
|  |  |  |Q |
+--+--+--+--+
|Q |  |  |  |
+--+--+--+--+
|  |  |Q |  |
+--+--+--+--+
```

The chessboard above is an example of a non-attacking placement of 4 queens on a 4l chessboard. For reference, there are only 2 non-attacking placements of 4 queens on a 4x4 chessboard.



Hint:

As soon as the input gets relatively large, this problem can no longer be solved with a brute-force approach. For example, there are 16,777,216 possible placements of 8 queens on an 8x8 chessboard. To consider all of these placements and to check if they're non-attacking isn't viable. Can you come up with an approach that limits the number of placements to consider? In order to generate a placement of n queens, you naturally have to place queens one at a time. Try only placing queens such that they're in a non-attacking position, given where you've previously placed queens. This should drastically limit the total number of placements that you consider. For example, if you place the first queen in the first row and in the first column, then don't consider a placement where any other queen is in the first row, in the first column, or in the down diagonal that starts at the first queen. When placing a queen in order to generate a full placement of n queens, you'll have to check if the position that you're considering is non-attacking. This can be done in linear time or in constant time, depending on if and how you store what columns and diagonals are blocked by previously placed queens. See the Conceptual Overview section of this question's video explanation for a more in-depth explanation.

Space-TIme Complexity:

Upper Bound: O(n!) time | O(n) space - where n is the input number

Solution:

A.

```python
def nonAttackingQueens(n):
    # Each index of `columnPlacements` represents a row of the chessboard,
    # and the value at each index is the column (on the relevant row) where
    # a queen is currently placed.
    columnPlacements = [0] * n
    return getNumberOfNonAttackingQueenPlacements(0, columnPlacements, n)


def getNumberOfNonAttackingQueenPlacements(row, columnPlacements, boardSize):
    if row == boardSize:
        return 1

    validPlacements = 0
    for col in range(boardSize):
        if isNonAttackingPlacement(row, col, columnPlacements):
            columnPlacements[row] = col
            validPlacements += getNumberOfNonAttackingQueenPlacements(row + 1, columnPlacements, boardSize)

    return validPlacements


# As `row` tends to `n`, this becomes an O(n)-time operation.
def isNonAttackingPlacement(row, col, columnPlacements):
    for previousRow in range(row):
        columnToCheck = columnPlacements[previousRow]
        sameColumn = columnToCheck == col
        onDiagonal = abs(columnToCheck - col) == row - previousRow
        if sameColumn or onDiagonal:
            return False

    return True
```

B.

```python
def nonAttackingQueens(n):
    blockedColumns = set()
    blockedUpDiagonals = set()
    blockedDownDiagonals = set()
    return getNumberOfNonAttackingQueenPlacements(0, blockedColumns, blockedUpDiagonals, blockedDownDiagonals, n)


def getNumberOfNonAttackingQueenPlacements(row, blockedColumns, blockedUpDiagonals, blockedDownDiagonals, boardSize):
    if row == boardSize:
        return 1

    validPlacements = 0
    for col in range(boardSize):
        if isNonAttackingPlacement(row, col, blockedColumns, blockedUpDiagonals, blockedDownDiagonals):
            placeQueen(row, col, blockedColumns, blockedUpDiagonals, blockedDownDiagonals)
            validPlacements += getNumberOfNonAttackingQueenPlacements(
                row + 1, blockedColumns, blockedUpDiagonals, blockedDownDiagonals, boardSize
            )
            removeQueen(row, col, blockedColumns, blockedUpDiagonals, blockedDownDiagonals)

    return validPlacements


# This is always an O(1)-time operation.
def isNonAttackingPlacement(row, col, blockedColumns, blockedUpDiagonals, blockedDownDiagonals):
    if col in blockedColumns:
        return False
    if row + col in blockedUpDiagonals:
        return False
    if row - col in blockedDownDiagonals:
        return False

    return True


def placeQueen(row, col, blockedColumns, blockedUpDiagonals, blockedDownDiagonals):
    blockedColumns.add(col)
    blockedUpDiagonals.add(row + col)
    blockedDownDiagonals.add(row - col)


def removeQueen(row, col, blockedColumns, blockedUpDiagonals, blockedDownDiagonals):
    blockedColumns.remove(col)
    blockedUpDiagonals.remove(row + col)
    blockedDownDiagonals.remove(row - col)
```