DFS collection

Derrick Guo

February 2020

Difference between combination and permutations: Combination doesn't care about order. I just care about if I used this number or not. So we use an index in the combination array, and in the dfs call forwards, we only select elements that are in input[idx:]. We don't touch the things that we touched before.

However, permutations care about order. For example, [1,2,3] and [1,3,2] are different permutation, but they are the same combination. Thus we don't use an index to track if some number has been used and just select from input[idx:], instead, we keep a used array or dictionary. Then in every dfs call, we loop from the start of the array to find unused element, and thus create different order of arrays.

1 46.Permutations

Given a collection of distinct integers, return all possible permutations.

Example:
Input: [1,2,3]
Output: [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
Idea:
Use a visited array. The index in the visited array matches to the index of nums. If visited[i] == True, that means nums[i] has been visited in this dfs run. The base case of dfs is that len(path) == len(nums), which means we foud a permutation, then we return. Otherwise loop through the whole array to find unvisited numbers and add it to the path.
The reason to set visited[i] = True before calling dfs is to make sure we don't reuse it in the future dfs of this round, and when it returns from the dfs we set it back to False so that we can reuse this number to form the next permutation.

```
class Solution(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        res = []
        visited = [False] * len(nums)
        self.dfs(nums, visited, res, [])
        return res


    def dfs(self, nums, visited, res, path):
        if len(path) == len(nums):
            res.append(path)
            return
        for i in range(len(nums)):
            if not visited[i]:
                visited[i] = True
                self.dfs(nums, visited, res, path+[nums[i]])
                visited[i] = False
```

2 47. Permutation II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

Example:

Input: [1,1,2] Output: [[1,1,2], [1,2,1], [2,1,1]]

idea:

In this question, there could be duplicates in the array. there are 2 ways to deal with this:
1. use a hash map, where the key is the number and the value is its count. In the dfs calls, loop through the keys of the hash map instead of the actual array, because the keys in the map are all unique and the actual array contains duplicates.
2. **For any dfs questions where duplicates can be an issue, we sort the array and skip the duplicates.** Note we need to check the visited list to see if the first one of the duplicates are used. If it is used, then we are visiting the second one of the duplicate, we keep doing the dfs; if the first one of the duplicates are not used, we are using the second one of the duplicates first in our dfs, which is exactly the same as starting with the first one in the duplicate, so we skip.

```
class Solution(object):
    def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        res = []
        self.dfs(nums, [], res, collections.Counter(nums))
        return res

    def dfs(self, nums, temp, res, counter):
        if len(nums) == len(temp):
            res.append(temp)
            return
        for x in counter:
            if counter[x] > 0:
                counter[x] -= 1
                self.dfs(nums, temp + [x], res, counter)
                counter[x] += 1
```

Solution II:

```
class Solution(object):
    def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        res = []
        visited = [False] * len(nums)
        nums.sort()
        self.dfs(nums, visited, [], res)
        return res

    def dfs(self, nums, visited, path, res):
        if len(path) == len(nums):
            res.append(path)
            return
        for i in range(len(nums)):
            if i > 0 and nums[i] == nums[i-1] and not visited[i-1]:
                continue # skip the duplicates
            if not visited[i]:
                visited[i] = True
                self.dfs(nums, visited, path + [nums[i]], res)
                visited[i] = False
```

3 60. Permutation Sequence

The set [1,2,3,...,n] contains a total of n! unique permutations.

By listing and labeling all of the permutations in order, we get the following sequence for n = 3:

"123" "132" "213" "231" "312" "321" Given n and k, return the kth permutation sequence.

Note:

Given n will be between 1 and 9 inclusive. Given k will be between 1 and n! inclusive.

Example 1:

Input: n = 3, k = 3 Output: "213" Example 2:

Input: n = 4, k = 9 Output: "2314"

idea:

Technically this is not a dfs question, but we can think of it in the dfs way. There are n! permutations in the set [1...n], if we fix the first number of a particular permutation, there is (n-1)! of them. So we can think of this n! permutations as a group of n items, where each item has (n-1)! permutations in it and all permutations in the same group starts with the same number. We can keep dividing until the last digit. So now our goal is to find the which group to go into in the process if we want to find the kth permutation. the index of the group is k / (n-1)!, then we know the starting number of the group and we add it to the result. Then we get rid of that number because it's used and we can't use it again.

```python
class Solution(object):
    def getPermutation(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
        """
        nums = range(1, n+1)
        k -= 1 # the kth in the sequence has index k-1
        res = ''
        while n > 0:
            n -= 1
            index, k = divmod(k, math.factorial(n))
            res += str(nums[index])
            nums.remove(nums[index])
        return res
```

4 39. Combination Sum

Given a set of candidate numbers (candidates) (without duplicates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

The same repeated number may be chosen from candidates unlimited number of times.

Note:

All numbers (including target) will be positive integers. The solution set must not contain duplicate combinations. Example 1:

Input: candidates = [2,3,6,7], target = 7,

A solution set is: [[7], [2,2,3]]

Example 2:

Input: candidates = [2,3,5], target = 8,

A solution set is: [[2,2,2,2], [2,3,3], [3,5]]

idea:

DFS. Sort the Candidates first because otherwise there will be duplicates. ex: [2,2,3] could also be summed as [2,3,2], which are essentially the same. As DFS goes on, decrement the target until it reaches 0, which is the base case. Using an index here in the DFS call helps to track the position in the candidates array which tells us which number have been used to do the calculation, since the array is sorted and we only want to start from where we left from the previous dfs call.
Another thing to notice is that checking if the target ¡ n in the for loop, break if so. That will help to reduce the number of recursion stack, which saves both time and space.

```
class Solution(object):
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        res = []
        candidates.sort()
        self.dfs(candidates, 0, target, [],res)
        return res


    def dfs(self, candidates, idx, target, path, res):
        if target < 0:
            return
        if target == 0:
            res.append(path)
            return
        for i in range(idx, len(candidates)):
            self.dfs(candidates, i, target - candidates[i], path + [candidates[i]], res)
```

5 40. Combination Sum II

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

Each number in candidates may only be used once in the combination.

Note:

All numbers (including target) will be positive integers. The solution set must not contain duplicate combinations. Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8, A solution set is: [[1, 7], [1, 2, 5], [2, 6], [1, 1, 6]] Example 2:

Input: candidates = [2,5,2,1,2], target = 5, A solution set is: [[1,2,2], [5]]

idea:

Similar to the last one, but now since each number can be used only 1 time, so need to skip duplicates.

```python
class Solution(object):
    def combinationSum2(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        res = []
        candidates.sort()
        self.dfs(candidates, 0, target, [], res)
        return res

    def dfs(self, nums, index, target, temp, res):
        if target == 0:
            res.append(temp)
            return
        if target < 0:
            return
        for i in range(index, len(nums)):
            # Will be covered in the next dfs recursive call, so skip here
            if i > index and nums[i] == nums[i-1]: continue
            self.dfs(nums, i + 1, target - nums[i], temp + [nums[i]], res)
```

6 216. Combination Sum III

Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Note:

All numbers will be positive integers. The solution set must not contain duplicate combinations. Example 1:

Input: k = 3, n = 7 Output: [[1,2,4]] Example 2:

Input: k = 3, n = 9 Output: [[1,2,6], [1,3,5], [2,3,4]]

idea:

similar to last 2, just the base case is a bit different.

```python
class Solution(object):
    def combinationSum3(self, k, n):
        """
        :type k: int
        :type n: int
        :rtype: List[List[int]]
        """
        res = []
        self.dfs(k, n, 0, [], res)
        return res

    def dfs(self, k, n, idx, path, res):
        if n < 0: return
        if n == 0 and len(path) == k:
            res.append(path)
            return
        for i in range(idx+1, 10):
            self.dfs(k, n-i, i, path + [i], res)
```

7 22. Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

["((()))", "(()())", "(())()", "()(())", "()()()"]

idea:

Use 2 counters l, r to count the number of open parentheses and the close parentheses left to use. 3 cases:

1. l == r == 0: all of the open and close parentheses are used up, so we have a valid combination, add it to the returned list and return

2. l > 0: have some open parentheses left, don't need to care about r because r > 0 for sure. (Since we need valid parentheses, we can't have something ending with '(')

3. r > 0 and r > l: we only add closing parentheses when there are at least 1 unmatched opening parentheses in the path.

```python
class Solution(object):
    def generateParenthesis(self, n):
        """
        :type n: int
        :rtype: List[str]
        """
        res = []
        self.dfs(0, 0, n, '', res)
        return res

    def dfs(self, l, r, n, path, res):
        if l == r == n:
            res.append(path)
            return
        if l < n:
            self.dfs(l+1, r, n, path + '(', res)
        if r < l and r < n:
            self.dfs(l, r+1, n, path + ')', res)
```

8 17. Letter Combinations of a Phone Number

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example:

Input: "23" Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]. Note:

Although the above answer is in lexicographical order, your answer could be in any order you want.

idea:
Use a dictionary to get the mapping from the input digit to the characters it maps to. Then do DFS on this dictionary. Move one step each time on the given digits string, and expand the answers in the chars it maps to in the dictionary.

```
class Solution(object):
    def letterCombinations(self, digits):
        if not digits: return []
        dic = {2: 'abc', 3: 'def', 4: 'ghi', 5: 'jkl', 6: 'mno', 7: 'pqrs', 8: 'tuv', 9:
        res = []
        self.dfs(digits, dic, 0, '', res)
        return res

    def dfs(self, digits, dic, idx, path, res):
        if idx == len(digits):
            res.append(path)
            return
        for c in dic[int(digits[idx])]:
            self.dfs(digits, dic, idx+1, path+c, res)
```

9 200.Number of Islands

Given a 2d grid map of ’1’s (land) and ’0’s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

Input: 11110 11010 11000 00000

Output: 1 Example 2:

Input: 11000 11000 00100 00011

Output: 3

idea:

The key idea here is to flip the pixel when we hit it. Once we find a ’1’, we’ll mark it to something like ’’, basically as a way to tell the DFS algo that we’ve visited here before. Then the idea is to do dfs on a ’1’ we found, then expand that whole island. And each dfs call increments the counter by 1. Notice to check if we are actually visiting an island in dfs as well, because otherwise we are not expanding the island, we expanded the water as well.

```
class Solution(object):
    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        if not grid or not grid[0]: return 0
        R,C = len(grid), len(grid[0])
        res = 0

        for r in range(R):
            for c in range(C):
                if grid[r][c] == '1':
                    self.dfs(r,c, R, C, grid)
                    res += 1
        return res

    def dfs(self, r, c, R, C,  grid):
        if 0 <= r < R and 0 <= c < C and grid[r][c] == '1':
            grid[r][c] = '#'
            self.dfs(r-1, c, R, C, grid)
            self.dfs(r+1, c, R, C, grid)
            self.dfs(r, c+1, R, C, grid)
            self.dfs(r, c-1, R, C, grid)
```

10 Restore IP address

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

Example:

Input: ”25525511135” Output: [”255.255.11.135”, ”255.255.111.35”]

idea:

We need to track the number of cuts in the dfs process. For each valid ip address, it only has 4 parts. So we can cut 4 times. (By cut, I mean put a substring of s to the path, we need to do this process 4 times). Then the base case is that we cut 4 times and there is nothing left for us to cut. If both conditions are True, we record our path and return. If only one is True, that means either we made all the cuts and there are still things left to cut, or everything is cut and we still have cuts to perform. None of the above situations are valid, so we return without further operation.

If we are in the middle of the process, there are 3 cases. The piece of the sub ip address could contain 1, 2, or 3 digits. So we need to consider these 3 cases separately.

1. if s is not empty, then we can cut 1 character from s and go the the next step of dfs;

2. if s has at least 2 chars, and the first char is not ’0’, (since only 0 cannot start a piece of ip address that has more than 1 character) we can cut 2 chars from s and go to the next step of dfs;

3. if s at least 3 chars, we need to check, still the first char is not 0, but also the whole substring, s[:3] is between 0 and 255. if that satisfies, we can move forward to the next step of dfs call.

```
class Solution(object):
    def restoreIpAddresses(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        res = []
        self.dfs(s, '', res, 0)
        return res

    def dfs(self, s, path, res, cuts):
        if s == '' and cuts == 4:
            res.append(path[:-1])
            return
        if s == '' or cuts == 4: return

        self.dfs(s[1:], path + s[:1] + '.', res, cuts + 1)\
        # ssubstringchar
        if len(s) > 1 and s[0] != '0' :
            self.dfs(s[2:], path + s[:2] + '.', res, cuts + 1)
        # ssubstring3char
        if len(s) > 2 and s[0] != '0' and 0 <= int(s[:3]) <= 255:
            self.dfs(s[3:], path + s[:3] + '.', res, cuts + 1)
```

11 Subsets

Given a set of distinct integers, nums, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

Example:

Input: nums = [1,2,3] Output: [[3], [1], [2], [1,2,3], [1,3], [2,3], [1,2], []]

idea:

Since we are looking for all the subsets, we don't care about the order. Then we don't use used array, instead we use the index to track which element has been visited. Since we want all the subsets, there is no condition to satisfy to be included in the answer.

```
class Solution(object):
    def restoreIpAddresses(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        res = []
        self.dfs(s, '', res, 0)
        return res

    def dfs(self, s, path, res, cuts):
        if s == '' and cuts == 4:
            res.append(path[:-1])
            return
        if s == '' or cuts == 4: return

        self.dfs(s[1:], path + s[:1] + '.', res, cuts + 1)\
        # ssubstringchar
        if len(s) > 1 and s[0] != '0' :
            self.dfs(s[2:], path + s[:2] + '.', res, cuts + 1)
        # ssubstring3char
        if len(s) > 2 and s[0] != '0' and 0 <= int(s[:3]) <= 255:
            self.dfs(s[3:], path + s[:3] + '.', res, cuts + 1)
```

12 79. Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example:

board = [['A','B','C','E'], ['S','F','C','S'], ['A','D','E','E']]

Given word = "ABCCED", return true. Given word = "SEE", return true. Given word = "ABCB", return false.

idea:

Questions like this requires us to search in the board. To avoid double visiting a visited pixel in the board, we should flip it before calling the recursive dfs method, and flip it back once it is returned so that it is usable for the next dfs round.

Whenever we find a place where the first char in the word is the same as board[i][j], we know it's time to call dfs. In the dfs call, after making sure that the coordinates are valid, and word search is not finished, we check if word[0] is the same as board[i][j]. If so, we found one step further and this is the right way to go. **Now, flip board[i][j] to something like '#' which will never exist in the board, so in the next dfs call we know that this neighbour is visited and ignores it.** Then call dfs on the neighbours board[i][j] to try to move forward. The base case is word == ".

```
class Solution(object):
    def exist(self, board, word):
        """
        :type board: List[List[str]]
        :type word: str
        :rtype: bool
        """
        if not board or not board[0]: return False
        if not word: return True
        res = []
        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] == word[0]:
                    found = self.dfs(board, word, i, j)
                    if found: return True
        return False


    def dfs(self, board, word, i, j):
        if not word: return True
        if 0 <= i < len(board) and 0 <= j < len(board[0]):
            if board[i][j] == word[0]:
                val = board[i][j]
                board[i][j] = '#'
                found = False
                for x, y in ((i-1, j), (i+1, j), (i, j-1), (i, j+1)):
                    found = self.dfs(board, word[1:], x, y)
                    if found: break
                board[i][j] = val
                return found
        return False
```