Backtracking

Mustafa Muhammad

25 September 2021

List of Questions:

1. subsets in a set
2. permutations of a string
3. permutation with spaces
4. rat in a maze
5. N queens problem (2)
6. combination sum (3)
7. word search
8. letter case permutation

1 Identification

Backtracking is an approach to find solution to a problem by searching through all possible outcomes.

It follows the same algorithm as Depth First Search.

Examples:

1. Decision making (Rat in a maze, NQueens).
2. Optimization (Graphs – DFS).
3. Permutations (Permutations of array or strings).
4. Subsets.

Identification

1. Exponential Time Complexity problems (N is a small number, $N < 10$ or $N < 50$).
2. Constraints.
3. We dont have a guarantee of which outcome gives the solution.

2 All Subsets in a set

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        res = []
        def perms(nums, i, subset):
            if i == 0:
                res.append(subset.copy())
                return


            perms(nums, i-1, subset)
            subset.append(nums[i-1])
            perms(nums, i-1, subset)
            subset.pop()
            return
        perms(nums, len(nums), [])

        return res
```

Note: append a shallow copy in the results array otherwise an empty set will show up in the result array.

3 Permutations of a string

```
class Solution:
    def find_permutation(self, S):
        string_set = set()
        def swap(arr, pos, i):
            temp = arr[pos]
            arr[pos] = arr[i]
            arr[i] = temp
            return "".join(arr)

        def helper(pos, current_string):
            if pos == len(current_string):
                string_set.add(current_string)
                return
            for i in range(pos, len(current_string)):
                current_string = swap(list(current_string), pos, i)
                helper(pos+1, current_string)
                current_string = swap(list(current_string), pos, i)

            return

        helper(0, S)

        return string_set
```

Note: backtracking step is done when are performing the recusion between two swaps.

Also Note: we use a set to avoid duplicates, we may use an array to include duplicates in our answer.

4 Permutations of string with spaces

```
class Solution:
    def permutation (self, S):
        set_of_strings = set()

        def helper(pos, current_string):
            if pos == len(current_string):
                if current_string[-1] == " ":
                    current_string = current_string[:-1]
                set_of_strings.add(current_string)
                return

            helper(pos+1, current_string)
            #strings are immutable
            current_string = list(current_string)
            current_string.insert(pos+1, " ")
            current_string = "".join(current_string)
            helper(pos+2, current_string)
            current_string = list(current_string)
            current_string.pop(pos+1)

            return

        helper(0, S)
        return list(set_of_strings)
```

Note: Similar to generating permutation of strings, for loop is missing since we dont have to do swaps.

Also Note: Strings are immutable so convert to list before adding and removing values.

Also pay close attention to changing of index, after adding a space we increment the index by 2 and then remove the space.

5 Rat in a maze

```
class Solution:
    def findPath(self, m):

        paths = []
        def dfs(matrix, i, j, path):
            if i < 0 or j<0 or i > len(matrix)-1 or j > len(matrix[0])-1:
                return

            if matrix[i][j] == 0:
                return

            if i == len(matrix)-1 and j == len(matrix[0])-1:
                path.append((i,j))
                paths.append(path.copy())
                return

            matrix[i][j] = 0
            path.append((i,j))
            dfs(matrix, i+1, j, path.copy())
            dfs(matrix, i-1, j, path.copy())
            dfs(matrix, i, j+1, path.copy())
            dfs(matrix, i, j-1, path.copy())
            matrix[i][j] = 1
            path.pop()

            return

        dfs(m, 0, 0, [])

        return paths
```

Note: Simple dfs algorithm, make sure to pass copy of the array to avoid problems in popping and adding paths.

Also Note: Set visited back to one once the recursion is complete, no need for a set in this particular case.

6 N queens problem

```
class Solution:
    def nQueen(self, n):
        matrix = [[0]*n for i in range(n)]

        def is_safe(matrix, row, col):
            for i in range(col):
                if matrix[row][i] == 1:
                    return False

            for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
                if matrix[i][j] == 1:
                    return False

            for i, j in zip(range(row, len(matrix), 1), range(col, -1, -1)):
                if matrix[i][j] == 1:
                    return False

            return True

        def helper(matrix, j):
            if j == len(matrix):
                return True

            for i in range(len(matrix)):

                if is_safe(matrix, i, j):
                    matrix[i][j] = 1
                    if helper(matrix.copy(), j+1):
                        return True
                    matrix[i][j] = 0

            return False

        helper(matrix, 0)

        return matrix
```

Hardest thing to implement is the is-safe method, otherwise the solution is trivial.

7 Generating all permutations of the n queens problem

```
class Solution:
    def totalNQueens(self, n: int) -> int:
        matrix = [[0]*n for i in range(n)]

        def is_safe(matrix, row, col):
            for i in range(0, col):
                if matrix[row][i] == 1:
                    return False
            for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
                if matrix[i][j] == 1:
                    return False
            for i , j in zip(range(row, len(matrix)), range(col, -1, -1)):
                if matrix[i][j] == 1:
                    return False
            return True

        count = [0]
        def helper(matrix, j):
            if j == len(matrix):
                count[0] += 1
                return
            for i in range(0, len(matrix)):
                if is_safe(matrix, i, j):
                    matrix[i][j] = 1
                    helper(matrix.copy(), j+1)
                    matrix[i][j] = 0
            return

        helper(matrix, 0)
        return count[0]
```

Passing an integer as a global variable in a nested function causes problem, which is why it is passed as a list.

8 Combination Sum - Including duplicates

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int)
     -> List[List[int]]:
        sol = []


        def helper(i, candidates, current_sum, target, res):
            if i > len(candidates)-1 or current_sum > target:
                return

            if current_sum == target:
                sol.append(res.copy())
                return


            current_sum += candidates[i]
            res.append(candidates[i])
            helper(i, candidates, current_sum, target ,res)
            current_sum -= candidates[i]
            res.pop()
            helper(i+1, candidates, current_sum, target, res)

            return

        helper(0, candidates, 0, target, [])

        return sol
```

We are making two recursive calls - either to include or not to include, followed by the backtracking step

9 Combination Sum - Excluding duplicates

```
class Solution:
    def combinationSum2(self, candidates: List[int], target: int)
     -> List[List[int]]:
        candidates = sorted(candidates)
        sol = []


        def helper(i, current_sum, target, candidates, res):
            if current_sum == target:
                sol.append(res.copy())
                return

            if i > len(candidates)-1 or current_sum > target:
                return

            current_sum += candidates[i]
            res.append(candidates[i])
            helper(i+1, current_sum, target, candidates, res)
            current_sum -= candidates[i]
            res.pop()

            #if it is the same as the last one on the current sum (see last if)

            if len(res) == 0 or res[-1] != candidates[i]:
                helper(i+1, current_sum, target, candidates, res)

            return

        helper(0, 0, target, candidates, [])

        return result
```

Note: Slight difference in code, instead of i and i+1 calls we only make i+1 calls and check the last if statement.

Otherwise the code does not differ that much from combination sum with duplicates.

10 Combination Sum - No duplicates and with specified limit on number of elements

```
class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        result = []
        thresh = [k]
        nums = [i for i in range(1,10)]
        def helper(i, nums, target, current_sum, res):
            if current_sum == target and len(res) == thresh[0]:
                result.append(res.copy())
                return

            if len(res) > thresh[0] or i > len(nums)-1 or current_sum > target:
                return

            current_sum += nums[i]
            res.append(nums[i])
            helper(i+1, nums, target, current_sum, res)
            current_sum -= nums[i]
            res.pop()

            if len(res) == 0 or res[-1] != nums[i]:
                helper(i+1, nums, target, current_sum, res)

            return

        helper(0, nums, n, 0, [])

        return result
```

Find all valid combinations of k numbers that sum up to n such that the following conditions are true:

Only numbers 1 through 9 are used. Each number is used at most once. Return a list of all possible valid combinations. The list must not contain the same combination twice, and the combinations may be returned in any order.

Input: k = 3, n = 7 Output: [[1,2,4]] Explanation: 1 + 2 + 4 = 7 There are no other valid combinations.

11 Word Search

```
class Solution:
    def __init__(self):
        self.temp = ""
        self.visited = set()

    def exist(self, matrix: List[List[str]], word: str) -> bool:
        for i in range(0, len(matrix)):
            for j in range(0, len(matrix[0])):
                if matrix[i][j] == word:
                    return True

                elif word[0] == matrix[i][j] and self.dfs(matrix, i, j, 0, word):
                    return True

        return False



    def dfs(self, matrix, i, j, count, target):
        if count == len(target):
            return True

        if i < 0 or j < 0 or i > len(matrix)-1 or j > len(matrix[0])-1
        or matrix[i][j] != target[count]:
            return False

        temp = matrix[i][j]
        matrix[i][j] = None


        ans = self.dfs(matrix, i+1, j, count+1, target)
        or self.dfs(matrix, i-1, j, count+1, target)
        or self.dfs(matrix, i, j+1, count+1, target)
        or self.dfs(matrix, i, j-1, count+1, target)

        matrix[i][j] = temp

        return ans
```

12 Letter case permutation

```
class Solution:
    def letterCasePermutation(self, s: str) -> List[str]:
        res = []

        s = [letter for letter in s]

        def perms(pos, s):
            if pos >= len(s):
                if "".join(s) not in res:
                    res.append("".join(s))
                return

            s[pos] = s[pos].lower()
            perms(pos+1, s)
            s[pos] = s[pos].upper()
            perms(pos+1, s)


            return

        perms(0, s)

        return res
```

Given a string s, we can transform every letter individually to be lowercase or uppercase to create another string.

Return a list of all possible strings we could create. You can return the output in any order.

Input: s = "a1b2" Output: ["a1b2","a1B2","A1b2","A1B2"]

13 Letter Combinations of a Phone Number

```
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if len(digits) == 0:
            return []
        hash_map = {
            '2': "abc",
            '3': "def",
            '4': "ghi",
            '5': "jkl",
            '6': "mno",
            '7': "pqrs",
            '8': "tuv",
            '9': "wxyz"
        }
        res = []
        def backtrack(pos, current_str):
            if len(current_str) == len(digits):
                res.append(current_str)
                return

            for c in hash_map[digits[pos]]:
                backtrack(pos+1, current_str+c)

            return

        backtrack(0, "")

        return res
```

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

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

Input: digits = ”23” Output: [”ad”,”ae”,”af”,”bd”,”be”,”bf”,”cd”,”ce”,”cf”]