

Assignment 1: Iterative and Recursive Algorithms

Question 1

Aim: Letter Combinations of a Phone Number

Algorithm

```
Function backtrack (rem-dig, cur-comb, total-res):  
    If rem-dig is empty:  
        Add cur-comb to total-res  
        return  
    Else:  
        first = rem-dig[0]  
        possible = keypad[first]  
        For each letter in possible:  
            backtrack(rem-dig[1:], cur-comb + letter, total-res)
```

Code

```
class Solution:  
    comb = [  
        [], [],  
        ['a', 'b', 'c'],  
        ['d', 'e', 'f'],  
        ['g', 'h', 'i'],  
        ['j', 'k', 'l'],  
        ['m', 'n', 'o'],  
        ['p', 'q', 'r', 's'],  
        ['t', 'u', 'v'],  
        ['w', 'x', 'y', 'z']
```

]

```
def backtrack(self, inpstr: list, inplen: int, cur_comb: str, totcomb: List[str]):  
    if len(cur_comb) == inplen:  
        totcomb.append(cur_comb)  
        return  
  
    number = inpstr[0]  
  
    for letter in self.comb[int(number)]:  
        self.backtrack(inpstr[1:], inplen, (cur_comb + letter), totcomb)  
  
def letterCombinations(self, digits: str) -> List[str]:  
    daresult = list()  
    self.backtrack(list(digits), len(digits), "", daresult)  
    return daresult
```

Time complexity

This is a recursive solution with a branching factor of worst case 4 each time, hence the time complexity is

$O(4^N)$

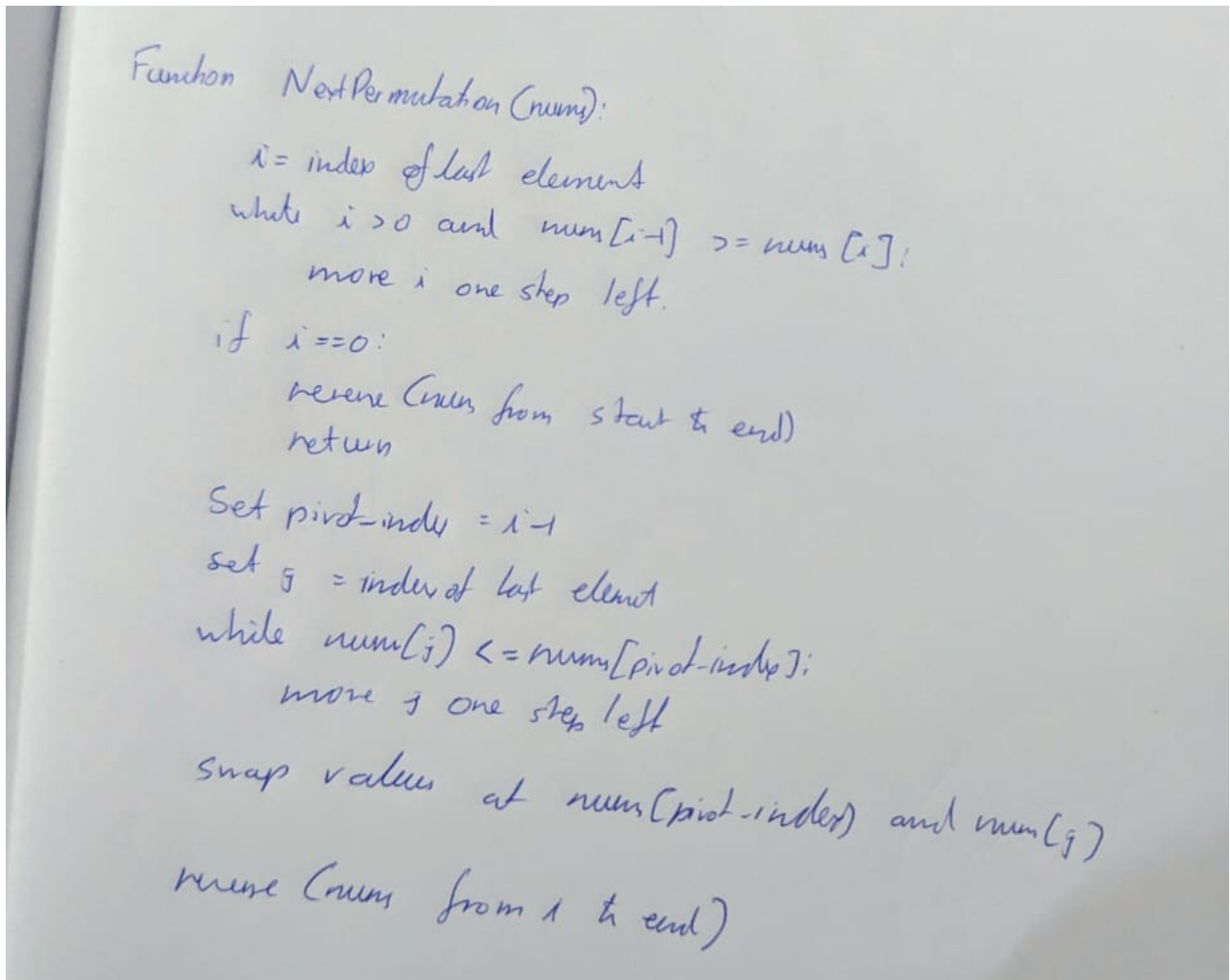
Screenshot of Output

The screenshot shows the LeetCode interface for the problem "17. Letter Combinations of a Phone Number". The problem description states: "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 digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters." A telephone keypad diagram is shown with digits 1-9 and their corresponding letters: 1 (no letters), 2 (a, b, c), 3 (d, e, f), 4 (g, h, i), 5 (j, k, l), 6 (m, n, o), 7 (p, q, r, s), 8 (t, u, v), 9 (w, x, y, z). Example 1: Input: digits = "23", Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]. Example 2: Input: digits = "2", Output: ["a","b","c"]. Constraints: 1 <= digits.length <= 4, digits[i] is a digit in the range ['2', '9']. The solution code is shown in Python3, implementing a recursive backtrack function. The test result shows "Accepted" with a runtime of 0 ms. A user profile for "Varghese K James" is visible on the right side of the interface.

Question 2

Aim: Next Permutation

Algorithm



Code

```
class Solution:  
    def nextPermutation(self, nums: List[int]) -> None:  
        i = len(nums) - 1  
        while i > 0 and nums[i-1] >= nums[i]:  
            i -= 1  
  
        if i == 0:  
            nums.reverse()  
            return
```

```
    pivot_idx = i - 1
    j = len(nums) - 1
    while nums[j] <= nums[pivot_idx]:
        j -= 1

    nums[pivot_idx], nums[j] = nums[j], nums[pivot_idx]

    nums[i:] = reversed(nums[i:])
```

Time complexity

This is a linear algorithm as even in worst case it only passes through the same element twice, hence it is of $O(N)$ time complexity.

Screenshot of Output

The screenshot displays a coding platform interface for the problem '31. Next Permutation'. The left panel shows the problem description, which includes examples and constraints. The right panel shows the user's solution code in Python3. The code defines a class 'Solution' with a method 'nextPermutation' that implements the algorithm. The bottom panel shows the test results, indicating that the solution is 'Accepted' and has a runtime of 0 ms. A user profile overlay for 'Varghese K James' is visible on the right side of the code editor.

31. Next Permutation

A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for `arr = [1,2,3]`, the following are all the permutations of `arr`: `[1,2,3]`, `[1,3,2]`, `[2,1,3]`, `[2,3,1]`, `[3,1,2]`, `[3,2,1]`.

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of `arr = [1,2,3]` is `[1,3,2]`.
- Similarly, the next permutation of `arr = [2,3,1]` is `[3,1,2]`.
- While the next permutation of `arr = [3,2,1]` is `[1,2,3]` because `[3,2,1]` does not have a lexicographical larger rearrangement.

Given an array of integers `nums`, find the **next permutation** of `nums`.

The replacement must be **in place** and use only constant extra memory.

Example 1:

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

Example 2:

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

Example 3:

```
Input: nums = [1,1,5]
Output: [1,5,1]
```

Constraints:

- $1 \leq \text{nums.length} \leq 100$
- $-10 \leq \text{nums}[i] \leq 10$

Testcase **Test Result**

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4

Input

```
nums =
[1,2,3]
```

Output

```
[1,3,2]
```

Expected

```
[1,3,2]
```

Contribute a testcase

Question 3

Aim: Wildcard Matching

Algorithm

Function isMatch(text, pattern):

 Initialize dp as a 2D grid with all false

 Set $dp[0][0] = \text{True}$.

 For each char in pattern:

 If char == '*':

 set its value to match whatever previous state was

 For each char in text & pattern:

 If pattern char is '*':

 match true if

 skip '*' or use '*'

 else if char is '?' or matches text char:

 match true if previous diag was true

 else
 match is false.

 return value at bottom right of grid dp

Code

```
class Solution:
    def isMatch(self, text: str, pattern: str) -> bool:
        n = len(text)
        m = len(pattern)

        dp = [[False] * (m + 1) for _ in range(n + 1)]

        dp[0][0] = True

        for j in range(1, m + 1):
            if pattern[j-1] == '*':
                dp[0][j] = dp[0][j-1]

        for i in range(1, n + 1):
            for j in range(1, m + 1):

                char_text = text[i-1]
                char_pattern = pattern[j-1]

                if char_pattern == '*':
                    dp[i][j] = dp[i][j-1] or dp[i-1][j]

                elif char_pattern == '?' or char_pattern == char_text:
                    dp[i][j] = dp[i-1][j-1]

                else:
                    dp[i][j] = False

        return dp[n][m]
```

Time complexity

This is a nested loop with the outer loop running for each character in text and the inner loop running for each character in pattern. Hence time complexity is $O(mn)$ where m is length of text and n is length of pattern

Screenshot of Output

The screenshot displays a LeetCode problem titled "44. Wildcard Matching" with a difficulty level of "Hard". The problem statement asks to implement wildcard pattern matching with support for '?' and '*' where '?' matches any single character and '*' matches any sequence of characters (including the empty sequence). The matching must cover the entire input string. Three examples are provided: Example 1 (s="aa", p="a" returns false), Example 2 (s="aa", p="a*" returns true), and Example 3 (s="cb", p="?a" returns false). Constraints include string lengths up to 2000 and only lowercase English letters. The solution is implemented in Python3 using a dynamic programming (DP) approach. The code defines a class Solution with a method isMatch that uses a 2D DP array to store results for subproblems. The DP array is initialized with False, and the base case dp[0][0] is set to True. The algorithm iterates through the string and pattern, updating the DP array based on whether the current characters match or if the pattern character is a wildcard. The final result is dp[n][m]. The test case result shows the solution is "Accepted" with a runtime of 0 ms for Case 1, Case 2, and Case 3. The input is s="aa" and p="a*", and the output is true.

44. Wildcard Matching Solved

Hard Topics Companies

Given an input string (*s*) and a pattern (*p*), implement wildcard pattern matching with support for '?' and '*' where:

- '?' Matches any single character.
- '*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

Example 1:

Input: *s* = "aa", *p* = "a"
Output: false
Explanation: "a" does not match the entire string "aa".

Example 2:

Input: *s* = "aa", *p* = "a*"
Output: true
Explanation: '*' matches any sequence.

Example 3:

Input: *s* = "cb", *p* = "?a"
Output: false
Explanation: '?' matches 'c', but the second letter is 'a', which does not match 'b'.

Constraints:

- 0 <= *s*.length, *p*.length <= 2000
- s* contains only lowercase English letters.
- p* contains only lowercase English letters, '?' or '*'.

Seen this question in a real interview before? 1/5

Yes No

Accepted 17.7K 100% 188 101 Online

Python3 Accepted x

```
1 class Solution:
2     def isMatch(self, text: str, pattern: str) -> bool:
3         n = len(text)
4         m = len(pattern)
5
6         dp = [[False] * (m + 1) for _ in range(n + 1)]
7
8         dp[0][0] = True
9
10        for j in range(1, m + 1):
11            if pattern[j-1] == '*':
12                dp[0][j] = dp[0][j-1]
13
14        for i in range(1, n + 1):
15            for j in range(1, m + 1):
16
17                char_text = text[i-1]
18                char_pattern = pattern[j-1]
19
20                if char_pattern == '*':
21                    dp[i][j] = dp[i][j-1] or dp[i-1][j]
22
23                elif char_pattern == '?' or char_pattern == char_text:
24                    dp[i][j] = dp[i-1][j-1]
25
26                else:
27                    dp[i][j] = False
```

Testcase Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

s =
"aa"

p =
"a*"

Output

true

Learning Outcomes

- Learned about using backtracking algorithms to solve questions involving permutations and combination
- Learned about bespoke optimisations done to solve competitive programming questions.