

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

$$O(4^N)$$

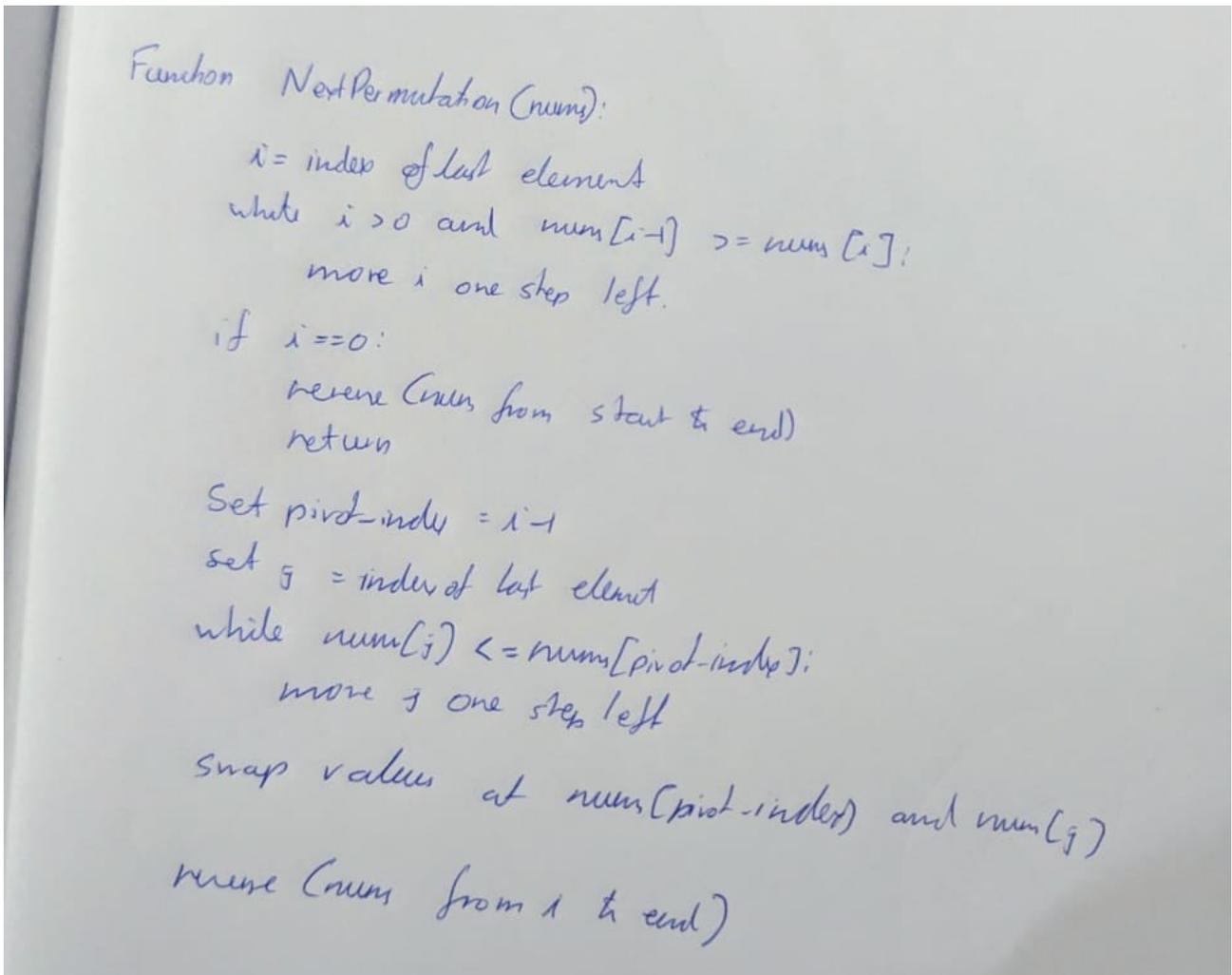
Screenshot of Output

The screenshot displays a coding interface for the problem '17. Letter Combinations of a Phone Number'. The problem description includes a mapping of digits to letters (1-9) and asks for all possible letter combinations for a given string of digits. The solution code is shown in Python, implementing a recursive backtracking algorithm. The test results show 'Accepted' with a runtime of 0 ms. The input is '23' and the output is ['ad', 'ae', 'af', 'bd', 'be', 'bf', 'cd', 'ce', 'cf'].

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

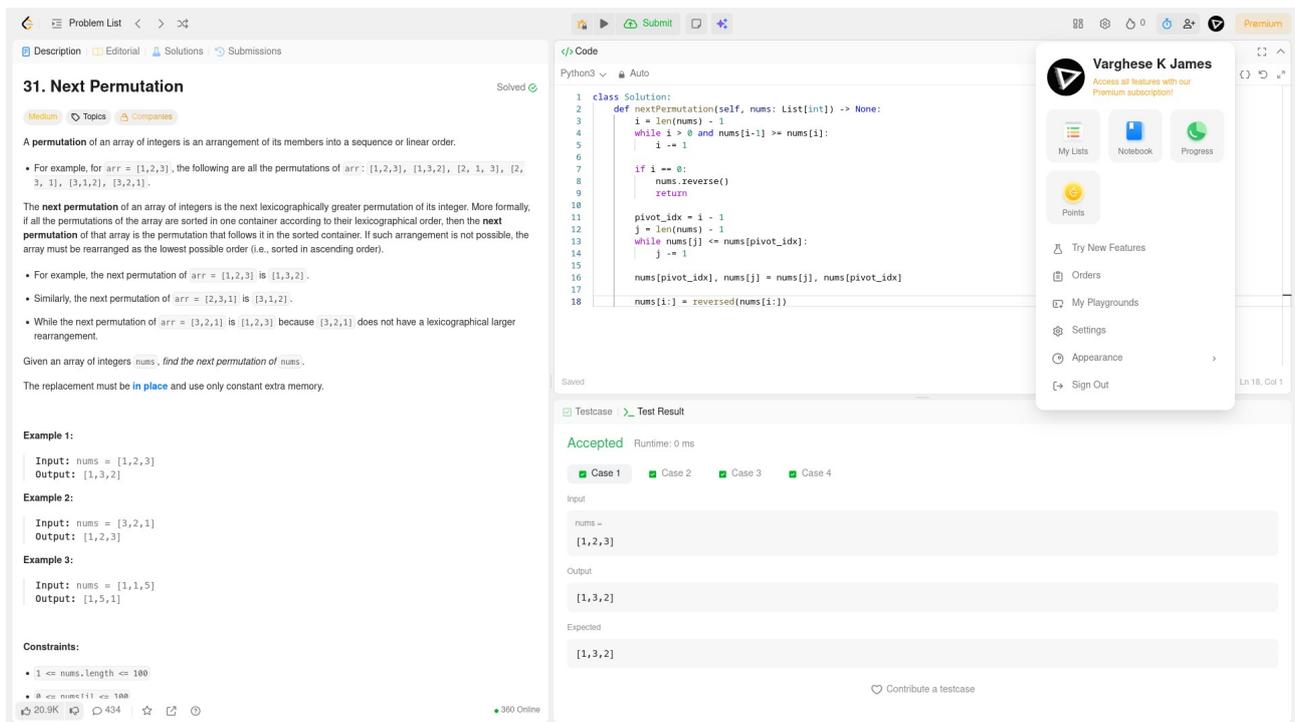
    nums[pivot_idx], nums[j] = nums[j], nums[pivot_idx]

    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



Question 3

Aim: Wildcard Matching

Algorithm

Function is Match (text, pattern):

Initialise 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 coding interface for a problem titled "44. Wildcard Matching". The problem description asks to implement wildcard pattern matching with support for '?' and '*'. The solution is implemented in Python using a dynamic programming (DP) approach. The code defines a class Solution with a method isMatch that takes a text string and a pattern string. It uses a 2D DP array to store the results of subproblems. The DP array is initialized with False values, and the base case dp[0][0] is set to True. The algorithm iterates over the pattern and text characters, updating the DP array based on the current characters and the previous state. The final result is dp[i][j], where i and j are the indices of the current characters in the text and pattern, respectively.

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

The test case shows the input s = "aa" and p = "a", resulting in the output false. The constraints are: 0 <= s.length, p.length <= 2000; s contains only lowercase English letters; p contains only lowercase English letters, '?', or '*'. The solution is accepted, and the runtime is 0 ms.

Learning Outcomes

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