# Count Vowels Permutation

Solution

Given an integer `n`, your task is to count how many strings of length `n` can be formed under the following rules:

- Each character is a lower case vowel (`'a'`, `'e'`, `'i'`, `'o'`, `'u'`)
- Each vowel `'a'` may only be followed by an `'e'`.
- Each vowel `'e'` may only be followed by an `'a'` or an `'i'`.
- Each vowel `'i'` **may not** be followed by another `'i'`.
- Each vowel `'o'` may only be followed by an `'i'` or a `'u'`.
- Each vowel `'u'` may only be followed by an `'a'.`

Since the answer may be too large, return it modulo `10^9 + 7.`

**Example 1:**

```
Input: n = 1
Output: 5
Explanation: All possible strings are: "a", "e", "i" , "o" and "u".
```

**Example 2:**

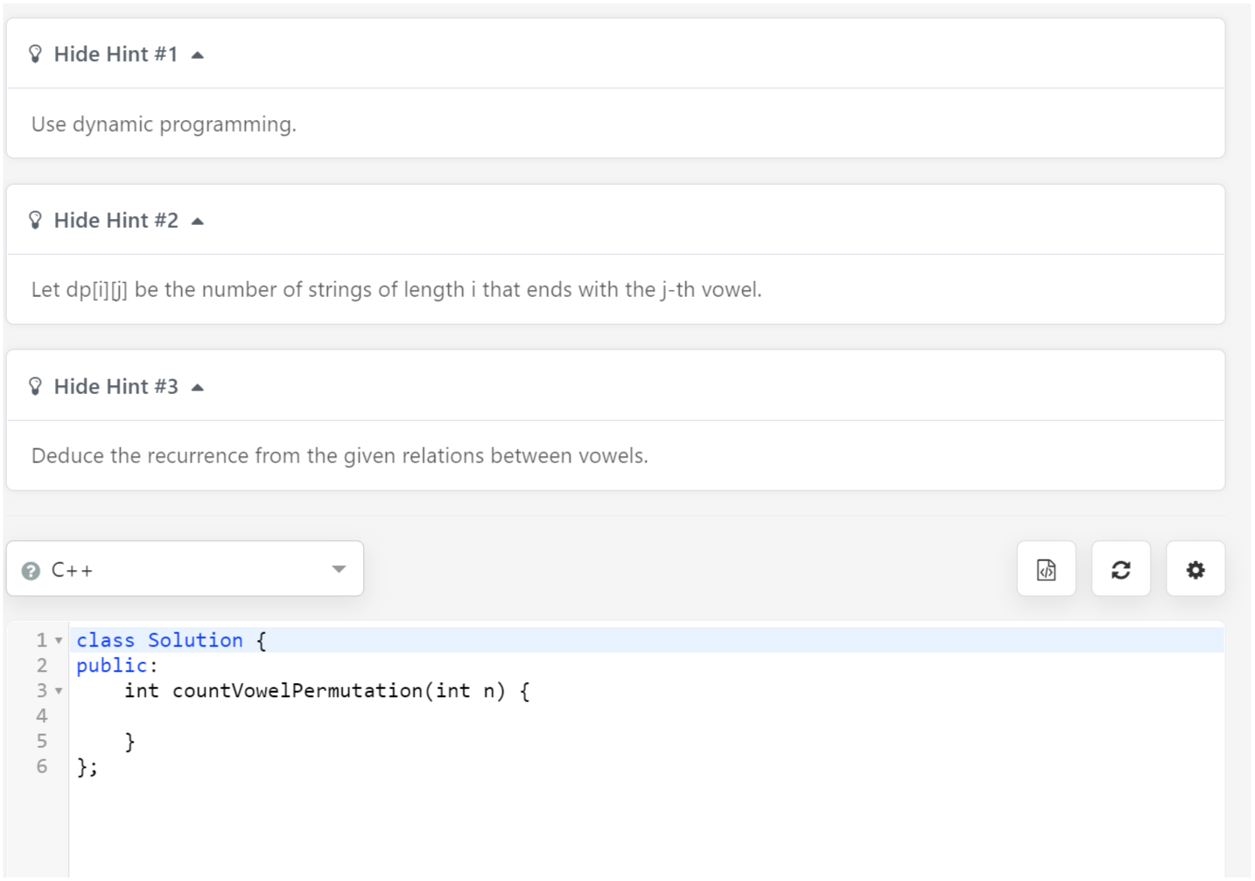
```
Input: n = 2
Output: 10
Explanation: All possible strings are: "ae", "ea", "ei", "ia", "ie", "io", "iu", "oi", "ou" and
```

**Example 3:**

```
Input: n = 5
Output: 68
```

**Constraints:**

- `1 <= n <= 2 * 10^4`



Use dynamic programming.

Let dp[i][j] be the number of strings of length i that ends with the j-th vowel.

Deduce the recurrence from the given relations between vowels.

```
class Solution {
public:
    int countVowelPermutation(int n) {

    }
};
```

## Solution

### Overview

Don't be scared by the *complex* rules! To solve this problem, we just need to tweak the rules a little bit.

There are five rules in the description (excluding the first bullet point) and each rule says **given a vowel, what vowels can be appended to it**. If we treat each vowel as a node, we can visualize the rules as shown in Figure 1. As you can see, Figure 1 illustrates all of the given rules.

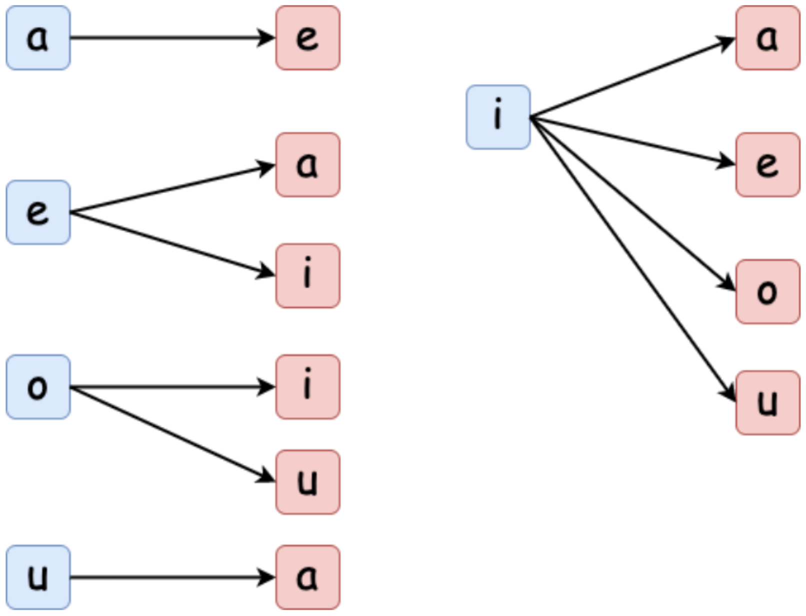


*Figure 1. Visualization of the rules.*

We must follow all of the rules while looking for permutations, so let's put all of the rules together. As shown in Figure 2, there are two ways to visualize the rules: (a) demonstrates the relationship between each pair of letters - the current letter and the following letter while (b) presents the rules as a directed cycle.

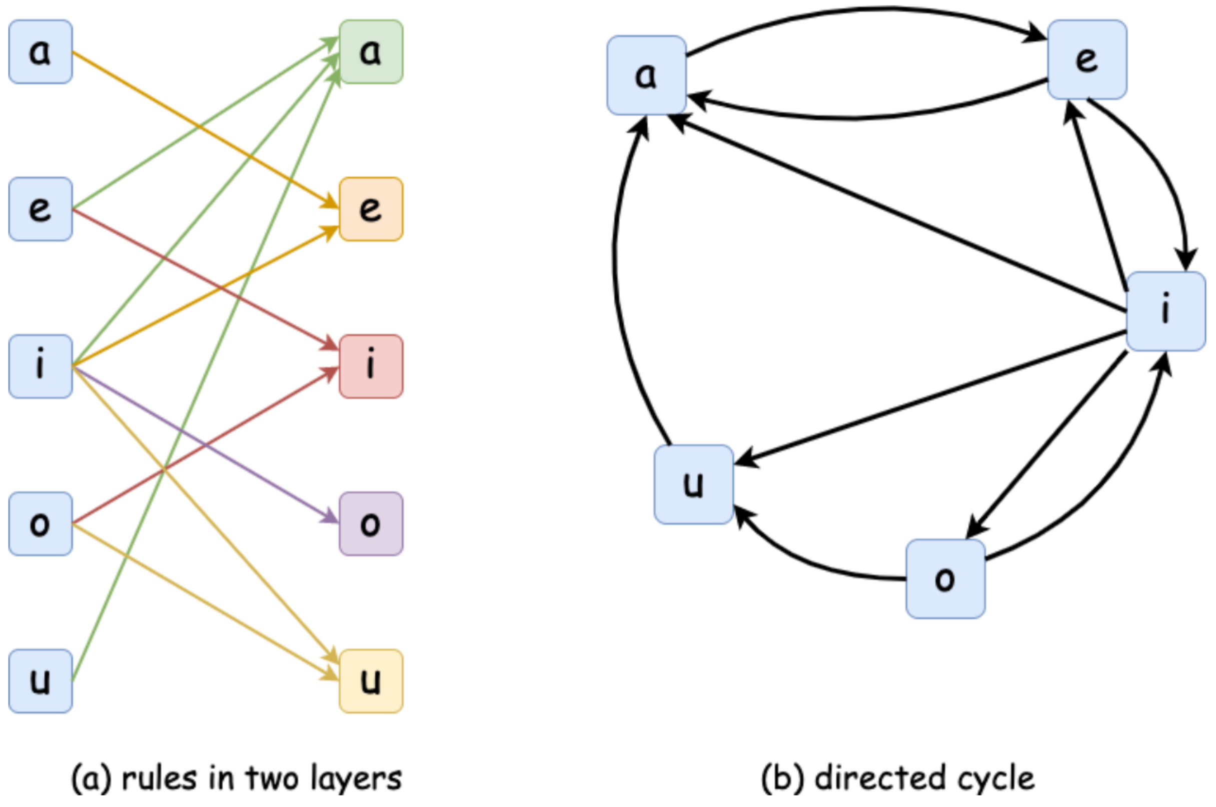


*Figure 2. Putting the rules together.*

> We can also model this problem using the state machine. State machines are a mathematical model of computation and they have powerful applications in advanced dynamic programming problems such as the *Best Time to Buy and Sell Stock* problems. As shown in (b), if we picture strings that end with different vowels as different states, what we have acquired is actually a map of all possible state transitions.

That said, if we are given the number of strings of length `i` that end in each vowel, like `aCount`, `eCount`, `iCount`, `oCount`, and `uCount`, we can compute the number of strings of length `i + 1` that end in each vowel by simple addition:

```
aCountNew = eCount + iCount + uCount
eCountNew = aCount + iCount
iCountNew = eCount + oCount
oCountNew = iCount
uCountNew = iCount + oCount
```

Starting from here, we have two approaches:

- Bottom-up: We will initialize the number of strings of size `1` to be `1` for each vowel. As the size grows from `1` to `n`, we will iteratively increase the count of strings that end in each vowel according to the rules above.
- Top-down: We can also perform the above idea recursively.

> In fact, we have more than two options. There exist solutions that take $O(logN)$ time, however, they are more advanced and likely fall outside the scope of what you will be expected to know in an interview. As such, they will not be discussed in this article. All the same, we encourage you to learn about them in the discussion section.

---

## Approach 1: Dynamic Programming (Bottom-up)

**Algorithm**

- Initialize five 1D arrays of size `n`, where `aVowelPermutationCount[i]`, `eVowelPermutationCount[i]`, `iVowelPermutationCount[i]`, `oVowelPermutationCount[i]`, and `uVowelPermutationCount[i]` will store the number of strings of length `i` that end in each vowel accordingly.

- Initialize the first element in each of the five arrays to `1`. This is because for each starting vowel there is only one permutation when the length of the string is `1`.

- Iterate the string length, `i`, from `1` to `n`:

  - Follow the rules to count the number of strings that end in each vowel. Take the sum of the last element from each of the five arrays and that will be the answer.

**Implementation**

Java
```java
class Solution {
    public int countVowelPermutation(int n) {

        long[] aVowelPermutationCount = new long[n];
        long[] eVowelPermutationCount = new long[n];
        long[] iVowelPermutationCount = new long[n];
        long[] oVowelPermutationCount = new long[n];
        long[] uVowelPermutationCount = new long[n];

        aVowelPermutationCount[0] = 1L;
        eVowelPermutationCount[0] = 1L;
        iVowelPermutationCount[0] = 1L;
        oVowelPermutationCount[0] = 1L;
        uVowelPermutationCount[0] = 1L;

        int MOD = 1000000007;


        for (int i = 1; i < n; i++) {
            aVowelPermutationCount[i] = (eVowelPermutationCount[i - 1] +
iVowelPermutationCount[i - 1] + uVowelPermutationCount[i - 1]) % MOD;
            eVowelPermutationCount[i] = (aVowelPermutationCount[i - 1] +
iVowelPermutationCount[i - 1]) % MOD;
            iVowelPermutationCount[i] = (eVowelPermutationCount[i - 1] +
oVowelPermutationCount[i - 1]) % MOD;
            oVowelPermutationCount[i] = iVowelPermutationCount[i - 1] % MOD;
            uVowelPermutationCount[i] = (iVowelPermutationCount[i - 1] +
oVowelPermutationCount[i - 1]) % MOD;
        }

        long result = 0L;

        result = (aVowelPermutationCount[n - 1] + eVowelPermutationCount[n - 1] +
iVowelPermutationCount[n - 1] + oVowelPermutationCount[n - 1] + uVowelPermutationCount[n -
1]) % MOD;


        return (int)result;
    }
}
```

**Complexity Analysis**

- Time complexity: $O(N)$ ($N$ equals the input length `n`). This is because iterating from `1` to `n` will take $O(N)$ time. The initializations take constant time. Putting them together gives us $O(N)$ time.
- Space complexity: $O(N)$. This is because we initialized and used five 1D arrays to store the intermediate results.

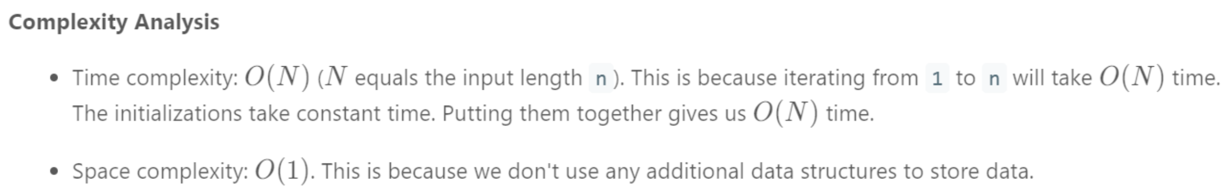
## Approach 2: Dynamic Programming (Bottom-up) with Optimized Space

It is worth noting that, in Approach 1, the `i` th element in each array only depends on the `i - 1` th element in some of the arrays. Therefore, the space complexity can be optimized by using five long variables (instead of 5 arrays of length `n`) to store the counts.

Java:

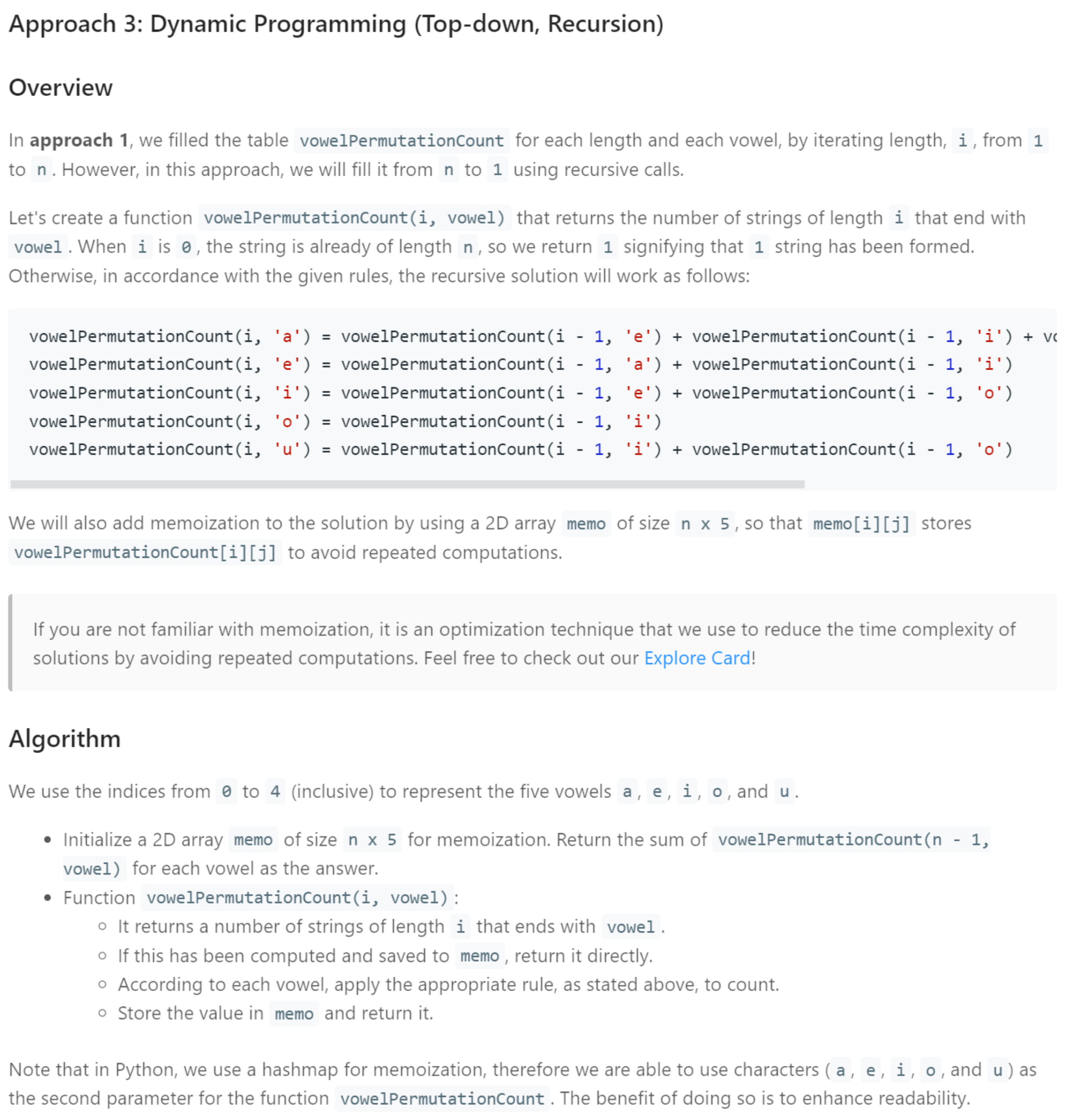
```
class Solution {
    public int countVowelPermutation(int n) {
        long aCount = 1, eCount = 1, iCount = 1, oCount = 1, uCount = 1;
        int MOD = 1000000007;

        for (int i = 1; i < n; i++) {
            long aCountNew = (eCount + iCount + uCount) % MOD;
            long eCountNew = (aCount + iCount) % MOD;
            long iCountNew = (eCount + oCount) % MOD;
            long oCountNew = (iCount) % MOD;
            long uCountNew = (iCount + oCount) % MOD;
            aCount = aCountNew;
            eCount = eCountNew;
            iCount = iCountNew;
            oCount = oCountNew;
            uCount = uCountNew;
        }
        long result = (aCount + eCount + iCount + oCount + uCount)  % MOD;
        return (int)result;
    }
}
```



**Complexity Analysis**

- Time complexity: $O(N)$ ($N$ equals the input length `n`). This is because iterating from `1` to `n` will take $O(N)$ time. The initializations take constant time. Putting them together gives us $O(N)$ time.
- Space complexity: $O(1)$. This is because we don't use any additional data structures to store data.



## Approach 3: Dynamic Programming (Top-down, Recursion)

### Overview

In **approach 1**, we filled the table `vowelPermutationCount` for each length and each vowel, by iterating length, `i`, from `1` to `n`. However, in this approach, we will fill it from `n` to `1` using recursive calls.

Let's create a function `vowelPermutationCount(i, vowel)` that returns the number of strings of length `i` that end with `vowel`. When `i` is `0`, the string is already of length `n`, so we return `1` signifying that `1` string has been formed. Otherwise, in accordance with the given rules, the recursive solution will work as follows:

```
vowelPermutationCount(i, 'a') = vowelPermutationCount(i - 1, 'e') + vowelPermutationCount(i - 1, 'i') + vo
vowelPermutationCount(i, 'e') = vowelPermutationCount(i - 1, 'a') + vowelPermutationCount(i - 1, 'i')
vowelPermutationCount(i, 'i') = vowelPermutationCount(i - 1, 'e') + vowelPermutationCount(i - 1, 'o')
vowelPermutationCount(i, 'o') = vowelPermutationCount(i - 1, 'i')
vowelPermutationCount(i, 'u') = vowelPermutationCount(i - 1, 'i') + vowelPermutationCount(i - 1, 'o')
```

We will also add memoization to the solution by using a 2D array `memo` of size `n x 5`, so that `memo[i][j]` stores `vowelPermutationCount[i][j]` to avoid repeated computations.

> If you are not familiar with memoization, it is an optimization technique that we use to reduce the time complexity of solutions by avoiding repeated computations. Feel free to check out our Explore Card!

### Algorithm

We use the indices from `0` to `4` (inclusive) to represent the five vowels `a`, `e`, `i`, `o`, and `u`.

- Initialize a 2D array `memo` of size `n x 5` for memoization. Return the sum of `vowelPermutationCount(n - 1, vowel)` for each vowel as the answer.
- Function `vowelPermutationCount(i, vowel)`:
  - It returns a number of strings of length `i` that ends with `vowel`.
  - If this has been computed and saved to `memo`, return it directly.
  - According to each vowel, apply the appropriate rule, as stated above, to count.
  - Store the value in `memo` and return it.

Note that in Python, we use a hashmap for memoization, therefore we are able to use characters (`a`, `e`, `i`, `o`, and `u`) as the second parameter for the function `vowelPermutationCount`. The benefit of doing so is to enhance readability.

Java:
```
class Solution {
    private long[][] memo;
    private int MOD = 1000000007;
    public int countVowelPermutation(int n) {
        // each row stands for the length of string
        // each column indicates the vowels
        // specifically, a: 0, e: 1, i: 2, o: 3, u: 4
        memo = new long[n][5];
        long result = 0;
        for (int i = 0; i < 5; i++){
            result = (result + vowelPermutationCount(n - 1, i)) % MOD;
        }
        return (int)result;
    }

    public long vowelPermutationCount(int i, int vowel) {
        if (memo[i][vowel] != 0) return memo[i][vowel];
        if (i == 0) {
            memo[i][vowel] = 1;
        } else {
            if (vowel == 0) {
                memo[i][vowel] = (vowelPermutationCount(i - 1, 1) + vowelPermutationCount(i - 1, 2)
+ vowelPermutationCount(i - 1, 4)) % MOD;
            } else if (vowel == 1) {
                memo[i][vowel] = (vowelPermutationCount(i - 1, 0) + vowelPermutationCount(i - 1, 2))
% MOD;
            } else if (vowel == 2) {
                memo[i][vowel] = (vowelPermutationCount(i - 1, 1) + vowelPermutationCount(i - 1, 3))
% MOD;
            } else if (vowel == 3) {
                memo[i][vowel] = vowelPermutationCount(i - 1, 2);
            } else if (vowel == 4) {
                memo[i][vowel] = (vowelPermutationCount(i - 1, 2) + vowelPermutationCount(i - 1, 3))
% MOD;
            }
        }
        return memo[i][vowel];
    }
}
```

- Time complexity: $O(N)$. This is because there are $N$ recursive calls to each vowel. Therefore, the total number of function calls to `vowelPermutationCount` is $5 \cdot N$, which leads to time complexity of $O(N)$. Initializations will take $O(1)$ time. Putting them together, this solution takes $O(N)$ time.

- Space complexity: $O(N)$. This is because $O(5 \cdot N)$ space is required for memoization. Furthermore, the size of the system stack used by recursion calls will be $O(N)$. Putting them together, this solution uses $O(N)$ space.