# 1079. Letter Tile Possibilities

Medium Hash Table String Backtracking Counting

Leetcode Link

## Problem Description

In this problem, you are given `n` tiles, each with a letter printed on it. The problem requires you to determine the total number of distinct non-empty sequences of letters that you can create using these tiles. You may use each tile as many times as it appears. A sequence of letters is a string created by concatenating the tiles in any order, and sequences can be of any length from 1 to the total number of tiles available, provided that the frequency of each tile is not exceeded.

## Intuition

To solve this problem, one efficient approach is to use depth-first search (DFS) along with backtracking. Here's the thought process:

1. Every tile can be included or excluded in a sequence, and you count all sequences as you generate them.
2. To keep track of the tiles you have used, you use the Count of each tile. A `Counter` is ideal since it allows you to count the occurrences of each letter in the string `tiles`.
3. For each letter in the counter, if there's at least one tile of that letter left (i.e., count > 0), you can:
   - Add it to your current sequence (and add one to the answer since adding this letter is a valid sequence by itself),
   - Decrease the count of that tile,
   - Continue the search (recursive DFS call) to consider longer sequences that can be made by adding additional tiles,
   - Increment the count back after the recursive call to backtrack and consider the next letters for the current position in the sequence.
4. This process counts all the valid letter sequences without overusing any single tile, as the counter ensures that you don't use more instances of a letter than are available.
5. Given the recursive nature of the approach, the base case is when there are no more tiles to be placed, at which point the recursion unwinds and accumulates the count of sequences.
6. You initiate the process by calling the `dfs` function with the `Counter` of the tiles, and return the total count of sequences minus one since the empty sequence is not considered a valid sequence but will be counted in the initial call.

By utilizing this approach, the problem is broken down into smaller subproblems, where each subproblem deals with constructing sequences by either including or excluding a particular letter and then recursing on the remaining letters.

## Solution Approach

The provided Python solution makes use of a depth-first search (DFS) to explore all possible sequences of letters that can be made from the given `tiles`. Let's delve deeper into the steps of the implementation:

1. **Counter Data Structure**: The solution first creates a `Counter` from the `collections` module for the `tiles` string. The Counter is a dictionary subclass designed to keep track of the number of occurrences of each element. In this case, it keeps the count of how many times each letter appears in the `tiles`.

2. **Depth-First Search (DFS)**: The `dfs` function is the core part of the solution. It takes a `Counter` (which represents the remaining tiles available for forming sequences) as its argument.

3. **Recursive Exploration**: Within this function, a loop iterates over each distinct letter in the `Counter` along with its count. For each letter:

   - If the letter's count is greater than zero (meaning the letter is available for use), the algorithm progresses with the following steps:
     - Increment the answer (`ans += 1`) since using this single letter is a valid sequence.
     - Reduce the count of this letter in the `Counter` (`cnt[i] -= 1`) to indicate that one instance of the letter has been used.
     - Perform the recursive call to `dfs(cnt)`, which will return the count of all sequences formed by the remaining letters. Add this count to the `ans`.
     - After exploring all sequences with the current letter used, backtrack by restoring the letter's count (`cnt[i] += 1`).
4. **Backtracking**: This ensures that the function explores all the sequences that can be formed when the current letter is not used in further extensions of the current sequence. This makes sure that any one sequence is not counted more than once.

5. **DFS Call and Result**: The initial call to the `dfs` function is made with the `Counter` of the tiles. The `dfs` function will implicitly return `0` when it is called with an empty counter or with a counter where all letters have been used up.

6. **Final Count**: The return from the initial call to `dfs` will be the total number of valid sequences that can be formed. This count includes the empty sequence as well, which is not desired according to the problem statement. Hence, the final result returned is one less than the count returned from `dfs`, effectively excluding the empty sequence.

7. **Time Complexity**: The time complexity of this approach depends on the number of different letters in `tiles` (let's denote this as k) and their counts. It explores all subsets of `tiles`, therefore in the worst case when all letters are unique, the time complexity would be O(2^n), but with repeated letters, the algorithm does not re-explore identical subsets, which improves efficiency.

The solution capitalizes on recursion and backtracking to explore all unique permutations of the tiles. It counts each sequence as it builds them up and backtracks to explore new sequences, ensuring that the final count includes all possible non-empty strings that can be formed.

### Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose the given `tiles` string is "AAB". This means we have 2 'A' tiles and 1 'B' tile. We want to count all distinct non-empty sequences of letters that can be made with these tiles.

#### Initialization

We initialize a `Counter` for the tiles, resulting in `Counter({'A': 2, 'B': 1})`.

#### Start Depth-First Search (DFS)

Now, let's go through the depth-first search process with the Counter.

#### Iteration 1: Choosing 'A'

- We take one 'A' from the Counter, leaving us with `Counter({'A': 1, 'B': 1})`.
- Increase the answer count by 1, as "A" is a valid sequence.
- We recursively call `dfs` with the updated Counter.

#### Inside Recursive Call 1 (With 'A' Used Once)

- We can still use 'A', so we take another 'A'. Now the Counter is `Counter({'A': 0, 'B': 1})`.
- We increment the answer count for sequence "AA".
- Recurse with this Counter.

##### Inside Recursive Call 2 (With 'AA')

- 'A' is no longer available, but 'B' is. We use 'B', leaving Counter empty.
- We increment the answer for "AAB".
- Recurse with an empty Counter, which is the base case and returns.

##### Backtrack to Recursive Call 1

- After the recursion returns, we backtrack, returning the 'B' to the Counter, making it `Counter({'A': 0, 'B': 1})` again.
- We finish exploring with "AA", so we exit and return to the first level of recursion.

#### Iteration 1: Backtracking

- We return the 'A' to the Counter from the first 'A' use, so it's `Counter({'A': 1, 'B': 1})` again.
- Now we consider using 'B' instead of 'A' at this level.

#### Iteration 2: Choosing 'B'

- We take the 'B', leaving us with `Counter({'A': 2})`.
- Increase the answer count for "B".
- Recursively call `dfs` with this Counter.

#### Inside Recursive Call 3 (With 'B' Used)

- Two 'A's are available. Use one 'A' and increase the count for "BA".
- Recurse with `Counter({'A': 1})`.

##### Inside Recursive Call 4 (With 'BA')

- One 'A' left. Increase the answer for "BAA".
- Recurse with an empty Counter, which is the base case and returns.

##### Backtrack to Recursive Call 3

- We restore the 'A' to the Counter, and it's `Counter({'A': 2})` again.
- End of recursion since all possibilities have been considered.

#### Iteration 2: Backtracking

- Place the 'B' back, having now counted "B", "BA", and "BAA" as valid sequences.

#### Final Answer

We have counted "A", "AA", "AAB", "B", "BA", and "BAA". Thus, we have 6 distinct non-empty sequences that can be created, which is our final answer.

#### Finalization

The initial `dfs` call returned the number of valid sequences, including the empty sequence. So, we subtract one from the answer to exclude the empty sequence, which results in a final answer count of 6 - 1 = 5 different non-empty sequences.

#### Conclusion

By going through the above steps, we explore all possible sequences that can be formed with the given tiles, by including or excluding each tile, while ensuring we never exceed the available count of each letter. This depth-first search with backtracking is an efficient method to solve this combinatorial problem.

## Python Solution

```python
from collections import Counter

class Solution:
    def numTilePossibilities(self, tiles: str) -> int:
        # Helper function to perform depth-first search on tile counts.
        def dfs(tile_counter: Counter) -> int:
            combinations_count = 0  # Initialize the count of combinations.

            # Iterate through each tile in the counter.
            for tile, count in tile_counter.items():
                # If there is at least one tile available, use one to form a new sequence.
                if count > 0:
                    combinations_count += 1  # Include this tile as a new possibility.
                    tile_counter[tile] -= 1  # Use one tile.

                    # Recursively count further possibilities by using the recently used tile.
                    combinations_count += dfs(tile_counter)

                    # Undo the choice to backtrack and allow for different combinations.
                    tile_counter[tile] += 1

            # Return the total number of combinations.
            return combinations_count

        # Count the occurrences of each tile.
        tile_counter = Counter(tiles)

        # Start DFS with the count of available tiles to find all possible combinations.
        return dfs(tile_counter)

# Example usage:
# sol = Solution()
# result = sol.numTilePossibilities("AAB")
# print(result)  # Output will be the number of possible sequences that can be formed.

```

## Java Solution

```java
class Solution {
    // Method to calculate the number of possible permutations of the tiles
    public int numTilePossibilities(String tiles) {
        // Array to hold the count of each uppercase letter from A to Z
        int[] count = new int[26];
        // Increment the respective array position for each character in tiles string
        for (char tile : tiles.toCharArray()) {
            count[tile - 'A']++;
        }
        // Start the recursive Depth-First Search (DFS) to calculate permutations
        return dfs(count);
    }

    // Recursive Depth-First Search method to calculate possible permutations
    private int dfs(int[] count) {
        int sum = 0; // Initialize sum to hold number of permutations
        // Iterate over the count array
        for (int i = 0; i < count.length; i++) {
            // If count of a particular character is positive, process it
            if (count[i] > 0) {
                // Increase the sum as we have found a valid character
                sum++;
                // Decrease the count for that character as it is being used
                count[i]--;
                // Further deep dive into DFS with reduced count
                sum += dfs(count);
                // Once DFS is back from recursion, revert the count used for the character
                count[i]++;
            }
        }
        // Return the sum of permutations
        return sum;
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    int numTilePossibilities(string tiles) {
        int count[26] = {}; // Initialize array to store the count of each letter

        // Count the occurrences of each letter in the tiles string
        for (char tile : tiles) {
            ++count[tile - 'A']; // Increase the count for the corresponding letter
        }

        // Define the recursive depth-first search function that calculates the possibilities
        function<int(int* count)> dfs = [&](int* count) -> int {
            int result = 0;
            // Iterate over all possible tile characters
            for (int i = 0; i < 26; ++i) {
                if (count[i] > 0) { // If the tile character is available
                    ++result; // This is a valid possibility by adding one tile
                    --count[i]; // Use one tile of the current character to explore further
                    result += dfs(count); // Explore further and add the result
                    ++count[i]; // Backtrack and add the tile back
                }
            }
            return result; // Return the total possibilities at this recursion level
        };

        // Start the recursion with the initial count and return the result
        return dfs(count);
    }
};

```

## Typescript Solution

```typescript
// Function to calculate the number of possible sequences from a given set of tiles
function numTilePossibilities(tiles: string): number {
    // Create a counter array initialized with zeros for each letter of the alphabet
    const count: number[] = new Array(26).fill(0);
    // Iterate through each tile and increment the corresponding count
    for (const tile of tiles) {
        count[tile.charCodeAt(0) - 'A'.charCodeAt(0)]++;
    }

    // Depth-first search function to explore all combinations
    const dfs = (count: number[]): number => {
        let sum = 0;

        // Loop through the alphabet
        for (let i = 0; i < 26; ++i) {
            // If a tile of the current letter is available, explore further combinations
            if (count[i] > 0) {
                // Increment the sum for the current combination
                sum++;
                // Choose the tile and explore further
                count[i]--;
                // Add the number of combinations from the sub-problem
                sum += dfs(count);
                // Backtrack and return the tile to the pool
                count[i]++;
            }
        }

        // Return the total number of combinations found
        return sum;
    };

    // Start the depth-first search with the initial count array
    return dfs(count);
}

```

## Time and Space Complexity

The given code uses a depth-first search (DFS) strategy with backtracking to generate all unique permutations of the given `tiles` string. Let's analyze both the time complexity and space complexity of the code.

### Time Complexity

The time complexity of this function is determined by the number of recursive calls made to generate all possible sequences, which grows factorially with the number of unique characters in the input string. In the worst case, all characters are unique, which would mean for a string of length `n`, there would be `n!` permutations.

At each level of the recursion, we iterate through all the unique characters left in the counter, decreasing the count for that character and then proceeding with the DFS. The recursion goes as deep as the number of characters in the string, and in each level, it branches based on the number of characters left.

Therefore, the upper bound for the time complexity can be represented as `O(n!)`, where `n` is the length of the `tiles` string. However, because the actual running time depends on the number of unique characters, if we let `k` be the number of unique characters, a more precise representation would be `O(k^n)`, because at each step, we can choose to add any of the remaining `k` characters to the sequence.

### Space Complexity

The space complexity is mainly determined by the call stack used for the recursive DFS calls, and the `Counter` object that maps characters to their counts. The maximum depth of the recursive call stack is `n`, which is the number of characters in the `tiles` string. Therefore, the space complexity for the recursion stack is `O(n)`.

The `Counter` object will have a space complexity of `O(k)`, where `k` is the number of unique characters in the `tiles` string, which is less than or equal to `n`.

Thus, the total space complexity of the algorithm can be considered as `O(n)` because the recursion depth dominates the space used by the `Counter` object.

In conclusion, the time complexity of this code is `O(k^n)` and the space complexity is `O(n)`, where `n` is the total length of the input string and `k` is the number of unique characters.