1079. Letter Tile Possibilities

You have `n` `tiles`, where each tile has one letter `tiles[i]` printed on it.

Return *the number of possible non-empty sequences of letters* you can make using the letters printed on those `tiles`.

Example 1:

```
Input: tiles = "AAB"
Output: 8
Explanation: The possible sequences are "A", "B", "AA", "AB", "BA", "AAB", "ABA", "BAA".
```

Example 2:

```
Input: tiles = "AAABBC"
Output: 188
```

Example 3:

```
Input: tiles = "V"
Output: 1
```

Constraints:

- `1 <= tiles.length <= 7`
- `tiles` consists of uppercase English letters.

1079. Letter Tile Possibilities

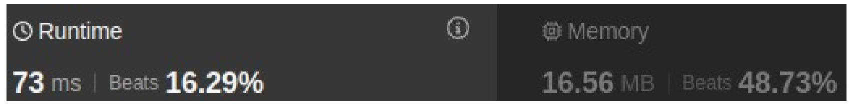
/*

Brute force: Generating n trees: all possible possible words for each level

Time complexity: $O(n!\times\sum_{k=1}^{n}\frac{1}{(n-k)!})$

Space complexity: O(2n+n!)=O(n!)

*/

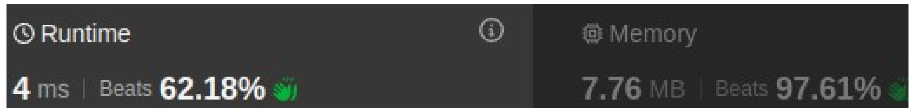



```
class Solution{
  public:
    int numTilePossibilities(std::string tiles) {
      int n=tiles.size();

      // To check if the letter at same index is used or not
      std::vector<int> used(n,0);

     // To store the generating word
      std::string word;

      // To store distinct words
      std::unordered_set<std::string> distinct_words;
```

```
        auto generate=[&](int k,auto& self)->void{
            // If the generated word is of size k
            if(k==0){
                // Save the generated word in the set
                distinct_words.insert(word);
                return;
            }

            // For each letter in the tiles
            for(int i=0;i<n;++i){
                // If the letter at position i is not used
                if(used[i]==0){
                    word.push_back(tiles[i]);      // Add it to the word
                    used[i] = 1;                   // Mark it as used
                    self(k-1,self);                // Do the same to the (k-1) remaining positions in word

                    // If a word of size k is generated
                    used[i]=0;                     // Mark the current letter as not used
                    word.pop_back();               // Reduce the size of the word
                }
            }
        };

        // generate a word of size k from the tiles
        for(int k=1;k<=n;++k){
            generate(k,generate); // create a tree of height k
        }

        return distinct_words.size();
    }
};
```

1079. Letter Tile Possibilities

```
/*
   Brute force: Generating one tree: all possible words
   Time complexity: O(n!)
   Space complexity: O(2n+n!)=O(n!)
*/
```




```
class Solution{
   public:
      int numTilePossibilities(std::string tiles) {
         int n=tiles.size();

         std::vector<int> used(n,0);

         std::string word;

         std::unordered_set<std::string> distinct_words;

         auto generate=[&](int k,auto& self)->void{
            if(k==0) return;
            for(int i=0;i<n;++i){
               if(used[i]==0){
                  word.push_back(tiles[i]);
                  used[i]=1;
                  self(k-1,self);
                  distinct_words.insert(word);
                  used[i]=0;
                  word.pop_back();
               }
            }
         };

         generate(n,generate);  // create a tree of height n


         return distinct_words.size();
      }
};
```

1079. Letter Tile Possibilities

```
/*
   Brute force: Generating one tree: all possible words using frequency array
   Time complexity: O(n!)
   Space complexity: O(n+n!)=O(n!)
*/
class Solution{
   public:
      int numTilePossibilities(std::string tiles) {
         int n=tiles.size();

         std::vector<int> freq(26,0);
         for(auto& c: tiles) freq[c-'A']++;

         std::string word;

         std::unordered_set<std::string> distinct_words;

         auto generate=[&](int k,auto& self)->void{
            if(k==0) return;

            for(int i=0;i<n;++i){
               if(freq[tiles[i]-'A']!=0){
                  word.push_back(tiles[i]);
                  freq[tiles[i]-'A']--;
                  distinct_words.insert(word);
                  self(k-1,self);
                  freq[tiles[i]-'A']++;
                  word.pop_back();
               }
            }
         };

         generate(n,generate);


         return distinct_words.size();
      }
};
```

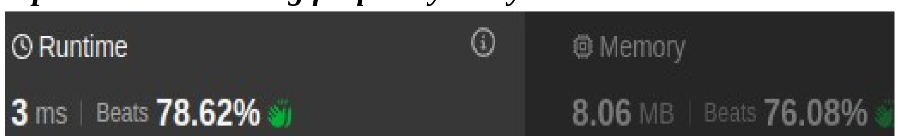
1079. Letter Tile Possibilities

Intuition

Imagine we're playing with Scrabble tiles again, but this time we have the string "AAABBC". We can make an important observation here: what really matters isn't the position of each letter, but rather how many of each letter we have available. Whether we use the first 'A' or the second 'A' doesn't change the sequences we can create - we just need to know we have three 'A's to work with.

This insight leads us to our first key decision: instead of tracking individual letters, we can track the frequency of each letter. Think of it like having separate piles for each letter - three tiles in the 'A' pile, two in the 'B' pile, and one in the 'C' pile. To implement this, we can maintain an array $freq$ where each index represents a letter (0 for 'A', 1 for 'B', etc.), and the value represents how many of that letter we have.

```
/*
   Space optimization counting all possible words using frequency array
   Time complexity: O(n!)
   Space complexity: O(n)
*/
```




```
class Solution{
   public:
      int numTilePossibilities(std::string tiles) {
         int n=tiles.size();

         std::vector<int> freq(26,0);
         for(auto& c: tiles) freq[c-'A']++;

         auto generate=[&](auto& self)->int{
            int ans=0;

            for(int i=0;i<26;++i){
               if(freq[i]==0) continue;
               freq[i]--;
               ans+=1+self(self);
               freq[i]++;
            }
           return ans;
         };

         return generate(generate);
      }
};
```

Time Complexity Analysis: (all above approaches)

The time complexity of this code depends on the number of recursive calls and the branching factor at each step.

1.Number of Recursive Calls:

- At each step, the function iterates over all 26 characters and recursively calls itself if the character's frequency is greater than 0 .
- The total number of recursive calls corresponds to the number of unique sequences that can be formed from the characters in tiles.

2.Worst Case:

- If all characters in tiles are unique, the number of possible sequences is the sum of all permutations of lengths from 1 to n , where n is the length of $tiles$.
- The number of such sequences is $\sum_{k=1}^{n} P(n,k)$, where $P(n,k)$ is the number of permutations of n items taken *k* at a time.
- This sum is approximately $n!$ for large n , since $P(n,k)=\frac{n!}{(n-k)!}$.

3.General Case:

- If there are duplicate characters, the number of unique sequences is reduced due to repeated characters. However, the worst-case time complexity remains exponential because the number of sequences can still be very large.

4.Final Time Complexity:

- The worst-case time complexity is $O(n!)$, where n is the length of the input string $tiles$. This is because the algorithm explores all possible permutations of the characters.

Space Complexity:

1.Recursion Stack: (all above approaches)

- The recursion depth can go up to n , so the space complexity due to the recursion stack is $O(n)$.

Brute force: Generating n trees: all possible possible words for each level

Brute force: Generating one tree: all possible words

- The $used$ array uses $O(n)$ space.
- The unordered_set distinct_words uses $O(n!)$ space

Thus, the space complexity is $O(2n+n!)$.

Brute force: Generating one tree: all possible words using frequency array

- The $freq$ array uses $O(26)=O(1)$ space.
- The unordered_set distinct_words uses $O(n!)$ space

Thus, the space complexity is $O(n+n!)$.

Space optimization using frequency array

- The frequency array $freq$ uses $O(26)=O(1)$ space.

Thus, the space complexity is $O(n)$.

This is an exponential-time algorithm, which is expected given the problem's nature of exploring all possible sequences. For large inputs, the above approaches may not be efficient.

1079. Letter Tile Possibilities

```
/*
   Time optimization using include/exclude
   Time complexity:  O(2n+nlogn+n.2^n)
   Space complexity:  O(n.2^n)
*/
class Solution{
   public:
      int numTilePossibilities(std::string tiles) {
         int n=tiles.size();

         // Preprocess factoriel computations
         std::vector<int> factoriel(n+1,1);
         auto preprocess_factoriel=[&]()->void{
            for(int i=2;i<=n;++i) factoriel[i]=factoriel[i-1]*i;
         };

         preprocess_factoriel();

         // Sort characters to handle duplicates efficiently
         std::sort(tiles.begin(), tiles.end());

         // To store distinct words
         std::unordered_set<std::string> distinct_words;
```

```
        // Generate all possibilities using include/exclude technique
        auto generate=[&](std::string cur_word, int i,auto& self)->int{

            // Count the number of permutations without repetitions of a word
            auto count_permutation=[&]()->int{

                std::vector<int> freq(26,0);
                for(auto& c: cur_word) freq[c-'A']++;

                int num=factoriel[cur_word.size()];
                int denom=1;
                for(int i=0;i<26;++i){
                    denom*=factoriel[freq[i]];
                }
                return num/denom;
            };

            // If we reach the last level
            if(i>=n){
                // If the current word exists, means that we have already computed its number
                // of unique permutations without repetitions
                if(distinct_words.find(cur_word)!=distinct_words.end()) return 0;

                // Otherwise:
                distinct_words.insert(cur_word);    // Add it to the set
                return count_permutation();   // Compute its number of permutations without repetitions
            }

            // If last level not reached
            int include=self(cur_word+tiles[i],i+1,self);   // Include the current letters
            int exclude=self(cur_word,i+1,self);            // Exclude it

           return include+exclude;
        };

        return generate("",0,generate)-1; // -1 for empty string
    }
};
```

Complexity Analysis

Let n be the length of the input string `tiles`.

- Time complexity: $O(2n + nlogn + n.2^n)$

  The time complexity is determined by several components:

  1. initialize the factorial array in $O(n)$
  2. The factorial calculations themselves are $O(n)$ as they iterate from 2 to at most n.
  3. The sorting takes $O(nlogn)$ time.
  4. In the $generate$ function, we create a binary recursion tree where at each position we have two choices (include or exclude), leading to 2^n possible sequences. For each unique sequence, we calculate permutations which involves iterating over the sequence ($O(n)$) and performing factorial calculations ($O(n)$).

  Therefore, the dominant factor is generating and processing all possible sequences, giving us a time complexity of $O(2^n \cdot n)$.

- Space complexity: $O(2^n \cdot n)$

  The space complexity also has multiple components.

  1. The recursion stack can go up to depth n, using $O(n)$ space.
  1. The hash set distinct_words stores unique combinations of characters. In the worst case, with all distinct characters, we could have 2^n different combinations as each character can either be included or excluded. Each sequence in the set can be up to length n. So, the set uses $O(2^n \cdot n)$ space.
  1. The $freq$ array in count_permutation function is constant space $O(1)$ as it's always size 26.

  Thus, the dominant factor is the space needed for storing unique sequences in the `seen` set, making the total space complexity $O(2^n \cdot n)$.