# 2273. Find Resultant Array After Removing Anagrams

Easy Array Hash Table String Sorting

## Problem Description

The problem presents us with an array of strings named `words`, where each element is a word composed of lowercase English letters. The main objective is to repeatedly delete words from the array under a specific condition: a word should be deleted if it is an anagram of the word preceding it. The index `i` of the word to be deleted must satisfy the condition that `0 < i < words.length`, meaning that the first word can never be deleted, and we must also have at least two words to proceed with a deletion.

An anagram is defined as a rearrangement of the letters of one word to form another word, with the condition that all original letters are used exactly once. For example, "listen" and "silent" are anagrams, as both contain the same letters in different orders.

This task is to be repeated until there are no more consecutive words that are anagrams of each other. At that point, we should return the final list of words.

## Intuition

The intuition behind the provided solution comes from the definition of anagrams. If two words are anagrams of each other, they will have the same letters in some order. Therefore, by sorting the letters of each word, we can easily compare them to check if they are anagrams.

The implementation uses a list comprehension, which is a concise way to generate a new list based on an existing list. For the current word `w` in `words`, two conditions are checked:

1. If `w` is the first word, it is included because the first word can never be an anagram of a preceding word (as there is no preceding word).
2. If `w` is not the first word, it is included if and only if its sorted form is different from the sorted form of the previous word in the list. This check ensures that `w` is not an anagram of the word immediately before it.

The solution does not require any additional loops or recursive calls because the problem guarantees that the order of deletions does not affect the final list of words that remain. Therefore, sequentially iterating over the list from start to finish is sufficient for finding the solution.

## Solution Approach

The solution utilizes a Python list comprehension to succinctly filter out the unwanted words that are anagrams of their immediate predecessors. Let's dive into the implementation detail:

1. A `for` loop is created by the `enumerate` function, which provides both the index `i` and the word `w` from the `words` list. The use of `enumerate` is essential here as we need to access the previous word to perform the anagram check.
2. The list comprehension iterates over each word (`w`) and its index (`i`) in the list `words`. The condition `i == 0 or sorted(w) != sorted(words[i - 1])` is the heart of this implementation, which serves two purposes:
   - `i == 0`: It ensures that the first word is always included in the final list because there is no word before it to check against for anagrams.
   - `sorted(w) != sorted(words[i - 1])`: For any word that is not first (i.e., when `i` is not `0`), this condition checks if the sorted characters of the current word `w` are different from the sorted characters of the previous word `words[i - 1]`. If they are different, the word `w` is included in the final list.
3. The use of the `sorted` function on strings is a pivotal step since sorting the characters of a string provides a consistent form to compare whether two words are anagrams. If sorting the two words results in identical strings, then those words are anagrams of each other. The condition `sorted(w) != sorted(words[i - 1])` efficiently takes advantage of this.
4. Ultimately, the final list comprises only those words that are not anagrams of the word immediately before them in the array. Consequently, the resulting list is composed of every first word from each potential sequence of anagrams.
5. The list comprehension itself is an efficient way to construct a new list based on the existing `words` list, avoiding the need for additional storage space which would be required if we were appending to a new list in a loop.

This approach ensures a single-pass solution with a time complexity that is mostly dependent on the sorting of individual words, which is `O(n*mlog(m))`, where `n` is the number of words and `m` is the maximum length of a word in the list.

### Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following array of strings:

```
words = ["bat", "tab", "cat"]
```

Now, let's walk step by step through the algorithm using the solution approach provided:

1. We enumerate through `words` using a `for` loop, gaining both the index `i` and the word `w`.
2. For the first word, "bat", `i` is equal to `0`. According to our implementation, we should include the first word in the final list because there's no preceding word to compare with. Hence, "bat" is included in the final list.
3. Moving to the second word, "tab", `i` is now `1`. We sort both "tab" and the previous word "bat", resulting in "abt" for both after sorting. Since they are identical after sorting, they are anagrams. According to our condition, "tab" should not appear in the final list because it is an anagram of the previous word. We move on to the next word without including "tab".
4. The next word is "cat", for which `i` is `2`. We sort "cat" to get "act", and we check it against the sorted previous word we included in the final list, which is "bat" sorted to "abt". Since "act" and "abt" are not equal after sorting, "cat" is not an anagram of the previously included word "bat". Therefore, "cat" should be included in the final list.

Using the list comprehension, which runs through these checks for each word, we get the final list which is:

```
final_words = ["bat", "cat"]
```

After going through the whole array, we've successfully filtered out any words that are anagrams of their immediate predecessors, and the resulting array is returned.

## Solution Implementation

### Python

```python
class Solution:
    def remove_anagrams(self, words: List[str]) -> List[str]:
        # Initialize an empty list to store the non-anagram words
        non_anagrams = []

        # Iterate through the list of words with their indexes
        for index, word in enumerate(words):
            # Append the first word to the result list as there is no previous word to compare
            # With subsequent words, add the word to the result list only if it is not an anagram
            # of the previous word
            if index == 0 or sorted(word) != sorted(words[index - 1]):
                non_anagrams.append(word)

        # Return the list of non-anagram words
        return non_anagrams
```

### Java

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;

class Solution {
    // Method to remove consecutive anagrams from an array of words
    public List<String> removeAnagrams(String[] words) {
        // Initialize a list to store the result
        List<String> result = new ArrayList<>();

        // String to keep track of the previous word (sorted version)
        String previousWordSorted = "";

        // Iterate over each word in the array
        for (String currentWord : words) {
            // Convert the current word to a char array and sort it
            char[] characters = currentWord.toCharArray();
            Arrays.sort(characters);

            // Create a string from the sorted char array
            String sortedCurrentWord = String.valueOf(characters);

            // Check if the sorted current word is different from the previous sorted word
            if (!sortedCurrentWord.equals(previousWordSorted)) {
                // If different, it's not an anagram of the previous word, so add it to the result
                result.add(currentWord);
            }

            // Update the previous word to be the sorted current word for the next iteration
            previousWordSorted = sortedCurrentWord;
        }

        // Return the list of words with consecutive anagrams removed
        return result;
    }
}
```

### C++

```cpp
#include <vector>
#include <string>
#include <algorithm>

// Function to create a signature for a word, which is a sorted string
std::string createWordSignature(const std::string &word) {
    std::string signature = word;
    std::sort(signature.begin(), signature.end()); // Sorts the characters in the word
    return signature; // Return the sorted word as its signature
}

// Function to remove anagrams
std::vector<std::string> removeAnagrams(std::vector<std::string> &words) {
    std::vector<std::string> ans; // Initialize an empty vector to store the non-anagram words
    ans.push_back(words[0]); // Always include the first word in the answer vector

    // Get the signature of the first word
    std::string previousWordSignature = createWordSignature(words[0]);

    // Iterate through each word starting from the second
    for (size_t i = 1; i < words.size(); i++) {
        // For each word starting from the second
        std::string currentWordSignature = createWordSignature(words[i]);

        // Compare the signature of the current word with the previous word's signature
        if (previousWordSignature != currentWordSignature) {
            // If the current word signature is different from the previous
            ans.push_back(words[i]); // Add it to the answer vector
            previousWordSignature = currentWordSignature; // Update the previousWordSignature
        }
    }
    return ans; // Return the vector containing only non-anagrams
}
```

### TypeScript

```typescript
function removeAnagrams(words: string[]): string[] {
    const n = words.length; // Get the length of the words array
    let ans: string[] = []; // Initialize an empty array to store the non-anagram words
    ans.push(words[0]); // Always include the first word in the answer array
    let previousWordSignature = createWordSignature(words[0]).join(''); // Get the signature of the first word

    for (let i = 1; i < n; i++) {
        // For each word starting from the second
        let currentWordSignature = createWordSignature(words[i]).join('');
        if (previousWordSignature !== currentWordSignature) {
            // If the current word signature is different from the previous
            ans.push(words[i]); // Add it to the answer array
            previousWordSignature = currentWordSignature; // Update the signature to the current word's signature
        }
    }
    return ans; // Return the array containing only non-anagrams
}

function createWordSignature(word: string): number[] {
    // Function to create a signature for a word
    let count = new Array(128).fill(0); // Initialize an array with 128 zeroes (assuming ASCII)
    for (let i = 0; i < word.length; i++) {
        // Iterate through each character of the word
        count[word.charCodeAt(i)]++; // Increment the count of the character's ASCII value in the count array
    }
    return count; // Return the count array (word signature)
}
```

```python
class Solution:
    def remove_anagrams(self, words: List[str]) -> List[str]:
        # Initialize an empty list to store the non-anagram words
        non_anagrams = []

        # Iterate through the list of words with their indexes
        for index, word in enumerate(words):
            # Append the first word to the result list as there is no previous word to compare
            # With subsequent words, add the word to the result list only if it is not an anagram
            # of the previous word
            if index == 0 or sorted(word) != sorted(words[index - 1]):
                non_anagrams.append(word)

        # Return the list of non-anagram words
        return non_anagrams
```

## Time and Space Complexity

### Time Complexity

The time complexity of the code primarily depends on two operations: iterating over the list of words and sorting each word to check if it is an anagram of the previous word.

- Iterating over the list is a `O(n)` operation, where `n` is the number of words in the input list.
- For each word, sorting takes `O(klogk)` time, where `k` is the average length of a word.

Thus, the total time complexity of the entire operation can be considered `O(n * klogk)`, as for each word in the list, sorting is performed, and then a comparison is made with the sorted previous word, which is a `O(k)` operation but is negligible relative to the sorting time.

### Space Complexity

The space complexity is determined by the additional space required for the sorted words and the space used by the output list.

- Sorting each word creates a new sorted string, resulting in `O(k)` space for each word (assuming strings are immutable, as in Python). But since this space is reused for each iteration, it is not multiplied by `n`. Therefore, this does not affect the overall space complexity asymptotically.
- The list comprehension builds a new list, and in the worst case, where no anagrams are removed, this takes `O(n)` space.

Hence, the space complexity of the code is `O(n)`, accounting for the space needed to store the output list.