692. Top K Frequent Words

Medium Trie Hash Table String Bucket Sort Counting Sorting Heap (Priority Queue) Leetcode Link

Problem Description

The problem requires us to find the `k` most frequent strings in an array of strings `words`. This entails two primary tasks: calculating the frequency of each word and then finding the top `k` based on this frequency. The problem adds a layer of complexity by asking for these top `k` strings to be in a specific order. The words should first be sorted based on frequency, with the most frequent words appearing first. If there are multiple words with the same frequency, those words should be ordered lexicographically (like in a dictionary), which means alphabetically in ascending order.

The expected outcome is a list of strings representing the `k` most frequent words in the order described above.

Intuition

The intuition behind the solution is to employ data structures and sorting mechanisms that can efficiently handle the two-level sorting criteria described in the problem statement.

The solution uses the following steps to arrive at the desired result:

1. Count the occurrences of each word using a data structure that maps words to their frequencies. Python's `Counter` class from the `collections` module is well-suited for this and helps us quickly determine the frequency of each word in `words`.
2. Sort the words based on the two criteria using a custom sorting function:
    - First, by the frequency of each word in descending order so that the most frequent words come first.
    - Second, by the word itself in lexicographical order for words with the same frequency. The lexicographical order ensures that if two words have the same frequency, the word that comes first alphabetically will come first in the sorted list.
3. To sort by multiple criteria, we can use a tuple where the first element is the negative frequency (to sort in descending order) and the second element is the word itself (to sort lexicographically as a tiebreaker).
4. Finally, we slice the sorted list to only include the top `k` elements.

Using this approach, we can leverage the powerful built-in sorting available in Python, ensuring that both sorting criteria are met, and only `k` elements are returned as the final result.

Solution Approach

The implementation of the solution takes advantage of Python's libraries and language features to accomplish the task of finding the top `k` most frequent strings. Here's a step-by-step explanation of the methods used:

Step 1: Counting Frequencies

We first use the `Counter` class from Python's `collections` module, which is a specialized dictionary designed for counting hashable objects. It takes an iterable (in this case, the `words` list) and creates a `Counter` object that maps each unique word to its frequency count.

```
cnt = Counter(words)
```

Step 2: Custom Sorting

Once we have the frequency counts, the next step is to sort the items based on our two criteria. Python's `sorted` function is utilized here, which allows for custom sorting logic through its `key` parameter. The `key` parameter accepts a function that returns a tuple containing the sorting criteria for each element in the iterable.

```
sorted(cnt, key=lambda x: (-cnt[x], x))[:k]
```

In the lambda function:

- `x` is each word from the `Counter` keys.
- `-cnt[x]` is the frequency of the word, negated so that the list is sorted in descending order (since Python sorts in ascending order by default).
- `x` is the word itself, which ensures that words with identical frequencies are sorted lexicographically.

The tuple `(-cnt[x], x)` ensures that the sorting happens first by frequency and then, as a tiebreaker, by lexicographical order.

Step 3: Slicing

Lastly, after sorting, we slice the list to only select the first `k` elements, as we're only interested in the top `k` most frequent words.

```
[:k]
```

This represents the final step where the method only returns the first `k` sorted elements, effectively giving us the top `k` most frequent words in the required order.

Combining these steps, the solution applies well-known algorithms for counting (hash table via `Counter`) and sorting (with custom criteria via `sorted`) to efficiently resolve the problem at hand. The use of lambda expressions and list slicing demonstrates the expressive power of Python's syntax, allowing for concise and readable code.

Example Walkthrough

Let's consider the following small example array of strings `words` and a value for `k` to illustrate the solution approach:

`words` = ["apple", "banana", "cherry", "apple", "banana", "apple"] `k` = 2

We want to find the 2 most frequent strings in the `words` array.

Step 1: Counting Frequencies

Using Python's `Counter` from the `collections` module, calculate the frequency of each word.

```
from collections import Counter
cnt = Counter(words)
# cnt = {'apple': 3, 'banana': 2, 'cherry': 1}
```

The `Counter` object now holds the frequency of each word: "apple" appears 3 times, "banana" 2 times, and "cherry" 1 time.

Step 2: Custom Sorting

Now we need to sort the words based on two criteria: the frequency (higher to lower) and lexicographic order. We use a custom sorting function with Python's `sorted`:

```
sorted_words = sorted(cnt, key=lambda x: (-cnt[x], x))
# sorted_words = ['apple', 'banana', 'cherry']
```

Here, `sorted_words` is sorted by frequency in descending order. Since "apple" and "banana" have different frequencies, no need for lexicographical sorting between them. "Cherry" has a lower frequency so comes last.

Step 3: Slicing

Finally, we only want the top `k` elements, so we slice the sorted list:

```
result = sorted_words[:k]
# result = ['apple', 'banana']
```

The result list `['apple', 'banana']` represents the 2 most frequent words in the `words` array, adhering to the frequency then lexicographical ordering rules.

Combining Step 1 through Step 3, the complete code to accomplish finding the `k` most frequent strings would look like this:

```python
from collections import Counter

def k_most_frequent(words, k):
    cnt = Counter(words)
    sorted_words = sorted(cnt, key=lambda x: (-cnt[x], x))
    return sorted_words[:k]

# Usage
words = ["apple", "banana", "cherry", "apple", "banana", "apple"]
k = 2
print(k_most_frequent(words, k))  # Output: ['apple', 'banana']
```

This example walkthrough demonstrates how the combined steps of counting, sorting, and slicing give us a clean solution using Python's powerful built-in features.

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def topKFrequent(self, words: List[str], k: int) -> List[str]:
        # Create a frequency counter for the words
        word_count = Counter(words)

        # Sort the unique words first by frequency in descending order, then alphabetically
        sorted_words = sorted(word_count, key=lambda word: (-word_count[word], word))

        # Return the first k elements of the sorted list which represent the top-k frequent words
        return sorted_words[:k]

```

Java Solution

```java
import java.util.*;

class Solution {
    public List<String> topKFrequent(String[] words, int k) {
        // Map to store the frequency count of each word
        Map<String, Integer> wordCount = new HashMap<>();
        // Calculate the frequency of each word
        for (String word : words) {
            wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
        }

        // Priority Queue to store words based on their frequency and lexicographical order
        PriorityQueue<String> heap = new PriorityQueue<>((word1, word2) -> {
            int frequencyDifference = wordCount.get(word1) - wordCount.get(word2);
            // If frequencies are the same, compare words in reverse lexicographical order
            if (frequencyDifference == 0) {
                return word2.compareTo(word1);
            }
            // Otherwise, order by frequency
            return frequencyDifference;
        });

        // Iterate over the distinct words and add them to the Priority Queue
        // The heap will maintain the top k frequent elements on top
        for (String word : wordCount.keySet()) {
            heap.offer(word);
            // If heap size exceeds k, remove the least frequent/current smallest element
            if (heap.size() > k) {
                heap.poll();
            }
        }

        // LinkedList to store the result in correct order
        LinkedList<String> topKWords = new LinkedList<>();
        // Populating the result list in reverse order since we want the highest frequency on top
        while (!heap.isEmpty()) {
            topKWords.addFirst(heap.poll());
        }

        // Return the list of top k frequent words
        return topKWords;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm> // Needed for sort function

using std::vector;
using std::string;
using std::unordered_map;

class Solution {
public:
    // This function returns the top k frequent words from the vector of strings "words".
    vector<string> topKFrequent(vector<string>& words, int k) {
        // Hash map to store the frequency count for each word.
        unordered_map<string, int> wordCount;
        // Count frequency of each word.
        for (const auto& word : words) {
            ++wordCount[word];
        }

        // Vector to store the unique words for sorting.
        vector<string> uniqueWords;
        // Extract the unique words (keys) from the map.
        for (const auto& pair : wordCount) {
            uniqueWords.emplace_back(pair.first);
        }

        // Custom sort the uniqueWords vector based on the frequency and alphabetical order.
        sort(uniqueWords.begin(), uniqueWords.end(), [&](const string& a, const string& b) {
            // If frequencies are equal, sort by alphabetical order.
            if (wordCount[a] == wordCount[b]) {
                return a < b;
            }
            // Otherwise, sort by frequency in descending order.
            return wordCount[a] > wordCount[b];
        });

        // After sorting, keep only the top k elements in the uniqueWords vector.
        uniqueWords.erase(uniqueWords.begin() + k, uniqueWords.end());

        // Return the top k frequent words.
        return uniqueWords;
    }
};

```

Typescript Solution

```typescript
import { sortBy } from 'lodash';

// Type alias for word and its frequency.
type WordFrequency = { [key: string]: number };

// This function counts the frequency of words in an array and returns an object where keys are words and values are frequencies.
function countWordFrequencies(words: string[]): WordFrequency {
  const wordCount: WordFrequency = {};
  words.forEach(word => {
    wordCount[word] = (wordCount[word] || 0) + 1;
  });
  return wordCount;
}

// This function returns the top k frequent words from the array of strings "words".
function topKFrequent(words: string[], k: number): string[] {
  // Use the countWordFrequencies function to get a frequency count for each word.
  const wordCount: WordFrequency = countWordFrequencies(words);

  // Extract the unique words (keys) from the wordCount object and sort them.
  let uniqueWords: string[] = Object.keys(wordCount);
  uniqueWords = sortBy(uniqueWords, [
    // Sort by frequency in descending order.
    word => -wordCount[word],
    // If frequencies are equal, sort by alphabetical order.
    word => word
  ]);

  // After sorting, keep only the top k elements in uniqueWords array.
  return uniqueWords.slice(0, k);
}

// Insert the necessary imports or dependencies if required.
// Note: The 'sortBy' function is utilized from lodash for convenience and needs to be installed.

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed in a few steps:

1. `Counter(words)` has a time complexity of `O(n)` where n is the number of words, as it requires one pass to count the frequency of each word.
2. `sorted(cnt, key=lambda x: (-cnt[x], x))` has a time complexity of `O(n log n)` on average, as it sorts the list of words by their frequencies, and in case of ties, it sorts them lexicographically. Here, "n" refers to the unique words in the list. 3.[:k]`slicing has a time complexity of`O(k)` because it needs to copy the first k elements from the sorted array.

Overall, the time complexity of the function is dominated by the sorting step, which, in the worst case, is `O(n log n)`.

Space Complexity

For space complexity:

1. The `Counter` uses `O(u)` space where u is the number of unique words in the input list, to store the word counts.
2. The sorted list will also use `O(u)` space.
3. The slice of the top `k` elements will require `O(k)` space, but this does not add to the overall space complexity since `k` is no larger than `u` and so is subsumed by it.

Therefore, the overall space complexity of the function is `O(u)`.