2363. Merge Similar Items

Easy Array Hash Table Ordered Set Sorting

Problem Description

You are provided with two 2D integer arrays called `items1` and `items2`. Each of these arrays contains pairs representing individual items, where the first number in the pair is the unique value of the item, and the second number is the weight of the item. The primary task is to merge these two sets of items together. When merging, if an item with the same value exists in both sets, you should add their weights together. Once merged, you should return a new 2D array containing pairs of items. Each pair should contain a unique value from any of the original arrays and the total weight for that value. Make sure to return the merged array in ascending order based on the item values.

Intuition

The solution focuses on efficiently combining two sets of items and accumulating the weights for items with the same value. The central idea is to use a data structure that allows easy updates of weights when the same value is encountered and also supports retrieval of items in a sorted manner based on their values. This can be achieved using a hash map that keeps track of the sum of weights for each unique value.

Here's a step-by-step approach on how we arrive at the solution:

1. Use a Counter (a special dictionary for counting hashable objects) from Python's collections module. It conveniently handles the addition of weights for us whenever we encounter an item with a value that's already in the Counter.
2. Iterate over both `items1` and `items2` (using `chain` from the `itertools` module to avoid writing two loops or concatenating the lists) and for each item, add its weight to the corresponding value in the Counter.
3. After processing all items, we have a Counter where the keys are the unique values and the values are the corresponding total weights. The remaining task is to sort this Counter by the item values.
4. Return the sorted items as a list of lists, which matches the expected output format.

By following these steps, we merge the two arrays and aggregate the weights for any duplicates, providing us with the correct output.

Solution Approach

The implementation of the solution is straightforward and utilizes Python's built-in libraries and data structures. Below is a step-by-step breakdown of the approach:

1. **Counter from `collections`**: In Python, the `Counter` class is specifically designed to count hashable objects. It is essentially a dictionary subclass that helps to count the occurrences of each element. In the given solution, we use it not just to count, but to sum up the weights of items based on their unique values.
2. **`chain` from `itertools`**: The `chain` function is used to iterate over multiple iterables as if they were a single iterable. In this problem, we treat `items1` and `items2` as one continuous iterable so that we can apply the same action to both without duplicating code.
3. **Iteration and accumulation**: We iterate over each item from `chain(items1, items2)`. For each value-weight pair `(v, w)`, we add the weight `w` to the count of `v` in our Counter. If `v` is not yet in the Counter, it's added with the starting weight `w`.
4. **Sorting the result**: Once we have the Counter with all values and their collective weights, we need to sort this data. However, a Counter doesn't maintain sorted order, so we convert the Counter into a list of items using `cnt.items()`. Each item is a tuple of `(value, total_weight)`. We then sort this list of tuples by the first element in each tuple, which is the value, as required for the output.
5. **Returning the result**: After sorting, the items are in the form of a list of tuples. Since the problem specifies that the result should be a 2D integer array, we return the sorted list without any further transformation, as the list of tuples is interpreted as a 2D list.

Here's how the code achieves this:

```
class Solution:
    def mergeSimilarItems(self, items1: List[List[int]], items2: List[List[int]]) -> List[List[int]]:
        cnt = Counter()
        # Chain both item lists and accumulate the weights in Counter
        for v, w in chain(items1, items2):
            cnt[v] += w
        # Sort by value and return the (value, total_weight) pairs as a list
        return sorted(cnt.items())
```

By relying on Counter and chain, the solution merges the two item lists effectively, with minimal lines of code and high readability.kpi

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose we're given the following two lists of items:

- `items1` is `[[1, 3], [4, 5], [6, 7]]`
- `items2` is `[[1, 1], [3, 3], [4, 1]]`

We want to merge these lists by combining the weights of items that have the same value and sort them by value.

Using the steps provided:

1. Create a Counter. Since Python's Counter isn't explicitly mentioned in our example code, we'll start by simply creating an empty one: `cnt = Counter()`.
2. Now, we chain `items1` and `items2` and begin to iterate over them. As we encounter each item, we add its weight to the corresponding value in the Counter:
   - For the first pair `[1, 3]` from `items1`, `cnt[1] = 3`.
   - Next, `[4, 5]` from `items1`, `cnt[4] = 5`.
   - Then `[6, 7]` from `items1`, `cnt[6] = 7`.
   - Now we move to `items2`. The first pair is `[1, 1]`. Since `1` is already in `cnt`, we add to its weight: `cnt[1] = 3 + 1 = 4`.
   - Next, `[3, 3]` from `items2`, `cnt[3] = 3` since `3` is not yet in `cnt`.
   - Finally, `[4, 1]` from `items2`, where `4` is already in `cnt`, so we update: `cnt[4] = 5 + 1 = 6`.
3. After iterating, we have a Counter with the following contents: `Counter({1: 4, 4: 6, 6: 7, 3: 3})`.
4. We need to sort this by item values. We convert the Counter to a list of items with `cnt.items()` which gives us: `[(1, 4), (4, 6), (6, 7), (3, 3)]`. When we sort this list, we get: `[(1, 4), (3, 3), (4, 6), (6, 7)]`.
5. Finally, we return the sorted list of tuples. This is exactly the output that the problem is expecting, which is a 2D list (or array in other programming languages) with item values and their total weights: `[[1, 4], [3, 3], [4, 6], [6, 7]]`.

So, the merged and sorted list is `[[1, 4], [3, 3], [4, 6], [6, 7]]`. This small example demonstrates how the provided solution approach effectively combines and sorts the item pairs from two different lists.

Solution Implementation

Python

```
from collections import Counter
from itertools import chain

class Solution:
    def mergeSimilarItems(self, items1, items2):
        # Create a Counter object to keep track of item weights
        item_weights = Counter()

        # Iterate over the combined items from both lists
        for value, weight in chain(items1, items2):
            # Sum the weights of items with the same value
            item_weights[value] += weight

        # Return a sorted list of tuples, where each tuple consists of an item value and its total weight
        # The list is sorted by item values
        return sorted(item_weights.items())
```

Java

```
class Solution {
    public List<List<Integer>> mergeSimilarItems(int[][] items1, int[][] items2) {
        // Array to count the quantities of each item where index represents the item value
        int[] itemCounts = new int[1010];

        // Adding quantities of items from items1 to itemCounts
        for (int[] item : items1) {
            int value = item[0]; // Item value
            int quantity = item[1]; // Item quantity
            itemCounts[value] += quantity;
        }

        // Adding quantities of items from items2 to itemCounts
        for (int[] item : items2) {
            int value = item[0]; // Item value
            int quantity = item[1]; // Item quantity
            itemCounts[value] += quantity;
        }

        // List to store the merged items as {value, quantity}
        List<List<Integer>> mergedItems = new ArrayList<>();

        // Iterating through itemCounts to add non-zero quantities to mergedItems
        for (int value = 0; value < itemCounts.length; ++value) {
            if (itemCounts[value] > 0) {
                // Creating a list of item value and quantity and adding it to the result
                mergedItems.add(List.of(value, itemCounts[value]));
            }
        }

        // Return the list of merged items
        return mergedItems;
    }
}
```

C++

```
#include <vector>
using namespace std;

class Solution {
public:
    vector<vector<int>> mergeSimilarItems(vector<vector<int>>& items1, vector<vector<int>>& items2) {
        // Initialize a counter array to store frequencies for items up to index 1000
        int itemFrequency[1010] = {};

        // Iterate through first list of items, increment the frequency of each item
        for (const auto& item : items1) {
            itemFrequency[item[0]] += item[1];
        }

        // Iterate through second list of items, increment the frequency of each item
        for (const auto& item : items2) {
            itemFrequency[item[0]] += item[1];
        }

        // Declare a vector to hold the merged items
        vector<vector<int>> mergedItems;

        // Iterate through the frequency array
        for (int i = 0; i < 1010; ++i) {
            // If the current item has a non-zero frequency, add it to the merged list
            if (itemFrequency[i]) {
                mergedItems.push_back({i, itemFrequency[i]});
            }
        }

        // Return the merged list of items
        return mergedItems;
    }
};
```

TypeScript

```
function mergeSimilarItems(items1: number[][], items2: number[][]): number[][] {
    // Create an array to keep track of the combined weights. Initialize all counts to 0.
    // The index represents the item's value, and the value at each index represents the weight.
    const itemWeights = new Array(1001).fill(0);

    // Iterate over the first array of items to add their weights to the itemWeights array
    for (const [value, weight] of items1) {
        itemWeights[value] += weight;
    }

    // Iterate over the second array of items to add their weights to the itemWeights array
    for (const [value, weight] of items2) {
        itemWeights[value] += weight;
    }

    // Convert the itemWeights array into tuples (value, weight), filter out items with a weight of 0,
    // and return the result.
    return itemWeights.map((weight, value) => [value, weight]) // Convert to tuples
                      .filter(([value, weight]) => weight !== 0); // Filter out items with zero weight
}
```

```
from collections import Counter
from itertools import chain

class Solution:
    def mergeSimilarItems(self, items1, items2):
        # Create a Counter object to keep track of item weights
        item_weights = Counter()

        # Iterate over the combined items from both lists
        for value, weight in chain(items1, items2):
            # Sum the weights of items with the same value
            item_weights[value] += weight

        # Return a sorted list of tuples, where each tuple consists of an item value and its total weight
        # The list is sorted by item values
        return sorted(item_weights.items())
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed in the following steps:

1. **Chain Iteration**: The `chain` function from `itertools` takes two lists `items1` and `items2` and creates an iterator that will iterate over both. The iteration itself takes `O(N + M)` time where `N` is the length of `items1` and `M` is the length of `items2`.
2. **Counter Update**: For each `(v, w)` pair in the combined iterator, the counter's value is updated, which is an `O(1)` operation. Since this happens for each element in the combined lists, it contributes to `O(N + M)` time complexity.
3. **Sorting**: The `sorted` function sorts the items in the counter based on keys. In Python, sorting is implemented with the Timsort algorithm which has a time complexity of `O(X * log(X))`, where `X` is the number of unique keys in the `cnt` Counter. In the worst case, each item could be unique, leading to `X = N + M`.

Combining these, the overall time complexity is `O(N + M) + O(N + M) + O((N + M) * log(N + M))` which simplifies to `O((N + M) * log(N + M))`.

Space Complexity

The space complexity of the given code can be considered as follows:

1. **Counter Space**: The `Counter` stores unique keys and their counts. In the worst case, every `(v, w)` pair might have a unique `v`, which means it could have up to `N + M` entries. Thus, it requires `O(N + M)` space.
2. **Output List**: The space required to hold the sorted items is also `O(N + M)` because every key-count pair from the counter is converted into a list `(v, w)` in the final output.

Therefore, the overall space complexity is `O(N + M)`.