1418. Display Table of Food Orders in a Restaurant

Medium · Array · Hash Table · String · Ordered Set · Sorting — Leetcode Link

Problem Description

The problem presents a scenario in which we're managing the orders in a restaurant. We have an array called `orders`, where each entry is a list with three elements: the name of the customer, the table number they're sitting at, and the food item they've ordered. Our task is to organize this data into a "display table".

The "display table" is effectively a two-dimensional array where each row represents a table in the restaurant, and each column represents a different food item. The top-left cell is labeled "Table", and the first row lists all the unique food items in alphabetical order, excluding this top-left cell. The first column lists the table numbers in ascending order. The rest of the cells in the table display the count of each food item ordered at each table.

The final output should not include customer names, and should be sorted by table number first, then by food item names alphabetically.

Intuition

To solve this problem, we must aggregate the orders for each table, and then count how many times each food item appears for each table.

Here's the step-by-step approach to do this:

1. **Identify Unique Elements**: We need to collect all unique table numbers and food items. The table numbers will form the rows of our display table, and the food items will form the columns.
2. **Count Ordered Items**: For each order, we count the number of times a food item is ordered per table. This is best done using a data structure like a dictionary or, in the case of Python, a `Counter` which is efficient for this kind of tallying.
3. **Initialize the Display Table**: Create the display table's header by adding "Table" followed by the sorted list of unique food items. This becomes the first row of our final result.
4. **Populate the Display Table**: Iterate through the sorted list of table numbers. For each table, create a new row starting with the table number. Then for each food item, add the count for that item at the current table. If the item hasn't been ordered at the table, the count is 0.
5. **Sort the Data**: Ensure that the table numbers and food items are sorted before creating the display table. Table numbers should be sorted numerically and food items alphabetically.

This approach guarantees that the final data representation matches the required format: a table sorted by table number with each type of food item ordered listed in alphabetical order, showing the quantity ordered at each table.

Solution Approach

The solution utilizes Python's standard library to effectively manage and compute the required output. The process includes:

1. **Data Storage**: Two Python `set` data structures are used to store unique table numbers (`tables`) and unique food items (`foods`) encountered in the list of orders. As sets, they will ignore any duplicate entries automatically.
2. **Counting with Counter**: The `Counter` from the `collections` module in Python is a subclass of the dictionary specifically designed for counting hashable objects. It is used here to tally the occurrences of `food item` per `table number`. The key is designed as a string with the format `'table.food'`, concatenating the table number and food item with a delimiter.
3. **Sorting and Conversion to List**: Once we have all unique table numbers and food items, these are sorted using Python's built-in `sorted` function. They are also converted to lists to prepare for iterating through them to produce the final display table format.
4. **Building the Display Table**: The display table is initialized with its header — a list beginning with `Table` followed by the sorted list of food items. For every table number in the sorted `tables` list, a new row is created starting with the table number in string format. Subsequently, for each food item in the sorted `foods` list, the number of times the food item was ordered at the table is retrieved from the `Counter` and converted to a string before being appended to the current row.
5. **Constructing the Final Output**: The final result (`res`) is a list of lists that mimics the tabular form. It starts with the header row and is followed by one row for each table displaying the counts for each food item.

The pattern followed here is straightforward:

- Use sets to derive uniqueness.
- Use `Counter` for efficient counting of items.
- Convert and sort the sets to list data structures for ordered access.
- Build the desired output in the format of a two-dimensional list, iterating through all table numbers and food items and using the tallies from the `Counter`.

This algorithm is effective as it breaks the problem down into simpler steps, each clearly executed with appropriate Python data structures and library functions. The use of the `Counter` particularly simplifies the problem of tracking and counting the number of food items per table number. The resulting implementation is clean, easy to understand, and performs the required task efficiently.

Example Walkthrough

Let's consider a small set of orders given to us in the following format: `[["David", "3", "Cesar Salad"], ["Alice", "3", "Chicken Sandwich"], ["Alice", "3", "Cesar Salad"], ["David", "2", "Pasta"]]`. Here's how the solution approach will handle this data:

1. **Identify Unique Elements**:
   - Unique table numbers: `{"3", "2"}`
   - Unique food items: `{"Cesar Salad", "Chicken Sandwich", "Pasta"}`
2. **Count Ordered Items**:
   - We count each food item ordered at each table. For example, "Cesar Salad" is ordered twice at table "3". We use a `Counter` to track this, resulting in a dictionary that may look something like `{"3.Cesar Salad": 2, "3.Chicken Sandwich": 1, "2.Pasta": 1}`.
3. **Initialize the Display Table**:
   - The header row is created as `["Table", "Cesar Salad", "Chicken Sandwich", "Pasta"]`, based on the sorted list of unique food items.
4. **Populate the Display Table**:
   - Create a row for table "2": `["2", "0", "0", "1"]` which means table "2" didn't order "Cesar Salad" or "Chicken Sandwich", but ordered "Pasta" once.
   - Create a row for table "3": `["3", "2", "1", "0"]` indicating the counts of "Cesar Salad", "Chicken Sandwich", and "Pasta" respectively.
5. **Constructing the Final Output**:
   - The resulting display table is:
     ```
     [ ["Table", "Cesar Salad", "Chicken Sandwich", "Pasta"],
       ["2", "0", "0", "1"],
       ["3", "2", "1", "0"] ]
     ```
   - This table is in the correct format, showing the counts of food items ordered at each table.

In summary, the example provided highlights the efficiency of the approach in organizing and presenting the data for a restaurant's order management system. Using Python's `Counter`, `set`, and sorting capabilities, we've transformed the list of orders into a display table that's easy to read and sorted accordingly.

Python Solution

```python
from collections import Counter

class Solution:
    def displayTable(self, orders: List[List[str]]) -> List[List[str]]:
        # Initialize sets for tables and food items to record unique items.
        unique_tables = set()
        unique_foods = set()

        # Use a Counter to keep track of the food orders per table.
        food_order_count = Counter()

        # Iterate through each order and update sets and Counter.
        for _, table_number, food_item in orders:
            unique_tables.add(int(table_number))   # Convert table number to int for sorting.
            unique_foods.add(food_item)            # Add the food item.
            food_order_count[f'{table_number}.{food_item}'] += 1  # Increment count.

        # Sort the lists of unique food items and table numbers.
        sorted_foods = sorted(list(unique_foods))
        sorted_tables = sorted(list(unique_tables))

        # Prepare the result table header with 'Table' followed by the sorted food items.
        result = [['Table'] + sorted_foods]

        # Populate the result table with counts per table.
        for table in sorted_tables:
            row = [str(table)]  # Start the row with the table number.
            for food in sorted_foods:
                # Append the string representation of the count of each food item for the table.
                row.append(str(food_order_count[f'{table}.{food}']))
            result.append(row)  # Add the completed row to the result.

        # Return the result table.
        return result

```

Java Solution

```java
class Solution {

    // This method will process a list of orders and display them as a table with food item counts.
    public List<List<String>> displayTable(List<List<String>> orders) {
        // Use TreeSet for automatic sorting
        Set<Integer> tableNumbers = new TreeSet<>();
        Set<String> menuItems = new TreeSet<>();

        // This map holds the concatenation of table number and food item as a key, and their count as a value.
        Map<String, Integer> itemCountMap = new HashMap<>();

        // Processing each order to populate sets and the itemCountMap
        for (List<String> order : orders) {
            int table = Integer.parseInt(order.get(1));
            String foodItem = order.get(2);

            // Add the table number and food item to the respective sets
            tableNumbers.add(table);
            menuItems.add(foodItem);

            // Create a unique key for each table-food pair
            String key = table + "." + foodItem;
            itemCountMap.put(key, itemCountMap.getOrDefault(key, 0) + 1);
        }

        // Prepare the result list, starting with the title row
        List<List<String>> result = new ArrayList<>();
        List<String> headers = new ArrayList<>();

        // Adding "Table" as the first column header
        headers.add("Table");
        // Adding the rest of the food items as headers
        headers.addAll(menuItems);
        result.add(headers);

        // Going through each table number and creating a row for the display table
        for (int tableNumber : tableNumbers) {
            List<String> row = new ArrayList<>();
            // First column of the row is the table number
            row.add(String.valueOf(tableNumber));
            // The rest of the columns are the counts of each food item at this table
            for (String menuItem : menuItems) {
                // Forming the key to get the count from the map
                String key = tableNumber + "." + menuItem;
                // Adding the count to the row; if not present, add "0"
                row.add(String.valueOf(itemCountMap.getOrDefault(key, 0)));
            }
            // Add the row to the result list
            result.add(row);
        }

        // Return the fully formed display table
        return result;
    }
}

```

C++ Solution

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

// Definition for a solution class.
class Solution {
public:
    // Function to display orders in a table format.
    vector<vector<string>> displayTable(vector<vector<string>>& orders) {

        // Use unordered sets to collect unique tables and food items.
        unordered_set<int> tableNumbers;
        unordered_set<string> foodItems;

        // Use a map to keep count of orders for table and food combinations.
        unordered_map<string, int> foodCount;

        // Process orders to fill the collections.
        for (const auto& order : orders) {
            int tableNumber = stoi(order[1]);
            const string& foodItem = order[2];

            // Insert table numbers and food items to respective sets.
            tableNumbers.insert(tableNumber);
            foodItems.insert(foodItem);

            // Increment count of the food item for a specific table.
            ++foodCount[order[1] + "." + foodItem];
        }

        // Convert table number set to a sorted vector.
        vector<int> sortedTables(tableNumbers.begin(), tableNumbers.end());
        sort(sortedTables.begin(), sortedTables.end());

        // Convert food items set to a sorted vector.
        vector<string> sortedFoodItems(foodItems.begin(), foodItems.end());
        sort(sortedFoodItems.begin(), sortedFoodItems.end());

        // Prepare the result variable.
        vector<vector<string>> result;

        // Create title row with "Table" followed by sorted food items.
        vector<string> titleRow {"Table"};
        titleRow.insert(titleRow.end(), sortedFoodItems.begin(), sortedFoodItems.end());

        // Add title row to the result.
        result.push_back(titleRow);

        // Loop over each table and create a row of counts per food item.
        for (int table : sortedTables) {
            vector<string> row;

            // Add table number to the row.
            row.push_back(to_string(table));

            // Loop over each food item and add the count to the row.
            for (const string& foodItem : sortedFoodItems) {
                row.push_back(to_string(foodCount[to_string(table) + "." + foodItem]));
            }

            // Add the row to the result.
            result.push_back(row);
        }

        // Return the filled result.
        return result;
    }
};

```

Typescript Solution

```typescript
// Importing necessary collections from a library equivalent to C++ STL
import { Vector, Set, Map } from 'typescript-collections';

// Function to display orders in a table format.
function displayTable(orders: Array<Array<string>>): Array<Array<string>> {

    // A set to collect unique tables and food item names.
    const tableNumbers: Set<number> = new Set<number>();
    const foodItems: Set<string> = new Set<string>();

    // A map to keep count of the orders by table and food item
    const foodCount: Map<string, number> = new Map<string, number>();

    // Process each order to fill the sets and the map.
    orders.forEach(order => {
        const tableNumber: number = parseInt(order[1]);
        const foodItem: string = order[2];

        // Insert table numbers and food items into respective sets.
        tableNumbers.add(tableNumber);
        foodItems.add(foodItem);

        // Construct a key to uniquely identify a table and food item combination.
        const key: string = `${order[1]}.${foodItem}`;

        // Increment the count for the food item at this table.
        foodCount.set(key, (foodCount.getValue(key) || 0) + 1);
    });

    // Convert the table numbers and food items sets to sorted arrays.
    const sortedTables: Array<number> = Array.from(tableNumbers).sort((a, b) => a - b);
    const sortedFoodItems: Array<string> = Array.from(foodItems).sort();

    // Prepare the result matrix to hold the data.
    const result: Array<Array<string>> = [];

    // Create the title row with "Table" followed by sorted food item names.
    const titleRow: Array<string> = ['Table', ...sortedFoodItems];

    // Add the title row to the result matrix.
    result.push(titleRow);

    // Create a row for each table with counts for each food item.
    sortedTables.forEach(table => {
        const row: Array<string> = [table.toString()];

        // For each food item, retrieve the count and add it to the row.
        sortedFoodItems.forEach(foodItem => {
            const key: string = `${table}.${foodItem}`;
            const count: number = foodCount.getValue(key) || 0;
            row.push(count.toString());
        });

        // Add the completed row to the result matrix.
        result.push(row);
    });

    // Return the completed result matrix.
    return result;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be broken down into several parts:

1. Iterating through the list of orders: This takes $O(N)$ time, where N is the total number of orders.
2. Adding items to and creating the `foods` and `tables` sets: Insertions take $O(1)$ on average, so for N orders, the complexity is $O(N)$.
3. The Counter updates (`mp[f'{table}.{food}'] += 1`) also occur N times, and they take $O(1)$ time each, thus $O(N)$ in total.
4. Sorting the `foods` list takes $O(F \log F)$ time, where F is the number of unique foods.
5. Sorting the `tables` list takes $O(T \log T)$ time, where T is the number of unique tables.
6. Building the `res` list involves a double loop which iterates T times outside and F times inside, leading to $O(T * F)$.

Adding these up, the total time complexity is $O(N) + O(N) + O(N) + O(F \log F) + O(T \log T) + O(T * F)$, which simplifies to $O(N + F \log F + T \log T + T * F)$.

Space Complexity

The space complexity can also be dissected into:

1. The `tables` and `foods` sets, which take $O(T + F)$ space.
2. The `mp` counter, which will store at most N key-value pairs, hence $O(N)$ space.
3. The `res` list, which contains a $T+1$ by F matrix, thus taking $O(T * F)$ space.

Combining these, the overall space complexity is $O(T + F + N + T * F)$. Since N can be at most $T * F$ if every table orders every type of food once, the space complexity simplifies to $O(T * F)$.