1282. Group the People Given the Group Size They Belong To

Medium Array Hash Table

Problem Description

In this problem, we are given `n` people each with a unique ID number from `0` to `n - 1`. These people need to be organized into groups. We are also given an array called `groupSizes` where `groupSizes[i]` is the target size of the group for person `i`. Our task is to form groups in such a way that each person `i` ends up in a group that has the same number of people as specified by `groupSizes[i]`.

To put it simply, if `groupSizes[2] = 4`, person with ID `2` must be in a group that contains exactly 4 individuals. It's important that each person belongs to exactly one group and every person must be in a group. The goal is to find a valid grouping that satisfies the conditions for all people. There could be several possible groupings that are valid, and any of these correct solutions can be returned.

The problem guarantees us that there will be at least one valid way to group all the people.

Intuition

To approach this problem, we need a way to organize people based on their group sizes without losing the information on individual IDs. A good strategy here is to use a hash map (or in Python, a dictionary) to keep track of which IDs need to be in which group sizes.

For each person ID and their corresponding group size, we add the person's ID to the list in the dictionary where the key is their group size. Essentially, we are bucketing person IDs by their group size requirement.

After organizing the people in these buckets, we then construct the actual groups. We know that for each group size, we need to create as many complete groups of that size as possible. Each list associated with a group size might be longer than the group size, which means it could form multiple groups. For example, if we have a group size of 2 and our bucket has IDs `[1, 2, 3, 4]`, then we have enough people to form two groups of two: `[1, 2]` and `[3, 4]`.

The solution code iterates through the dictionary, taking continuous slices of each list that are the size of the corresponding group size, and appending these slices to the final output until all IDs have been grouped.

Solution Approach

The solution is implemented using a `defaultdict` of type `list` from Python's `collections` module. A `defaultdict` is used instead of a regular dictionary to automatically handle the case where an entry for a group size is being accessed for the first time, thus avoiding the need for checking and initializing the empty list manually.

Here is a step-by-step breakdown of the solution approach:

- **Step 1:** Loop over the `groupSizes` array with `enumerate` to get both the index `i` (representing the person's ID) and the value `v` (representing the required group size for this person).

```
for i, v in enumerate(groupSizes):
```

- **Step 2:** Add each person's ID `i` to the list in our dictionary `g` corresponding to their group size `v`. This effectively buckets the IDs into lists where each list contains IDs of people supposed to be in the same group size.

```
g[v].append(i)
```

- **Step 3:** Iterate over each group size `i` and list of IDs `v` in our dictionary `g`.

```
for i, v in g.items():
```

- **Step 4:** For each group size `i` and its corresponding list of IDs `v`, we slice the list of IDs into chunks that are exactly the size of that group. We do this in a list comprehension that loops over a range starting from `0` to `len(v)` stepping by the group size `i`.

```
[v[j : j + i] for j in range(0, len(v), i)]
```

  This slicing step creates the sublists of IDs that form groups of the correct size. The range for the loop ensures that we create as many full groups of size `i` as possible from the available IDs in the list `v`.

- **Step 5:** The inner list comprehension produces lists of the correct group sizes for a single key in the dictionary. The outer list comprehension does this for all keys (group sizes) and concatenates the smaller lists into one larger list of results.

```
return [v[j : j + i] for i, v in g.items() for j in range(0, len(v), i)]
```

The final output is a list of groups, with each group being a list of IDs that satisfies the original group size requirements. The use of a dictionary to categorize IDs by group size and then slicing lists into chunks of the required size provides a clear and efficient approach to solving the problem.

By using this approach, we can group people effectively with a single pass through the `groupSizes` array and then another pass to slice the intermediate results into final groups. This makes the implementation not only clear in its logic but also efficient with a complexity of O(n), where n is the number of people, assuming that dictionary operations take constant time.

Example Walkthrough

Assume we are given the following `groupSizes` array: `[3, 3, 3, 3, 1, 1]`

According to the problem, we have 7 people with IDs `0` through `6`, and the `groupSizes` array indicates each person's desired group size. Let's walk through the algorithm using this array.

Step 1: Loop over the `groupSizes` array to get each person's ID (`i`) and their required group size (`v`). This would look like:

```
for i, v in enumerate([3, 3, 3, 3, 1, 1]):
```

Step 2: Add each person's ID to the `defaultdict` under their corresponding group size key. After this step, our `defaultdict`, named `g`, would look like:

```
{
3: [0, 1, 2, 3, 4],  # IDs 0 to 4 want to be in groups of size 3.
1: [5, 6]            # IDs 5 and 6 want to be in groups of size 1.
}
```

Step 3: Iterate over each entry in the dictionary. Start with the key `3`:

```
for i, v in g.items():
```

In this example, first, `i` would be `3`, and `v` would be `[0, 1, 2, 3, 4]`.

Step 4: Since the list of IDs into chunks of the group size, which is 3 in this case. The list `[0, 1, 2, 3, 4]` will be divided into `[0, 1, 2]` and `[3, 4]`. Since we do not have sufficient people to form another group of size 3, person `4` will not form a complete group and hence will wait for other people of group size `3` from other iterations.

Step 5: The list comprehension inside the `return` statement executes the chunking, and this is done for all group sizes in the dictionary.

The final result from the list comprehension will be `[[0, 1, 2], [3, 4]]` for the key `3`.

Next, we proceed with the key `1`, in which `i` would be `1`, and `v` would be `[5, 6]`.

Performing the same slicing step, we get two groups `[5]` and `[6]`, as both individuals want to be in separate groups of just one person.

Combining all outputs, the solution becomes:

```
[[0, 1, 2], [3, 4], [5], [6]]
```

Each of these sub-arrays represents a group, and as we can see, IDs `0`, `1`, and `2` form a group of size `3`; IDs `3` and `4` are in an incomplete group of size `3` awaiting more members in a full solution; and IDs `5` and `6` each form their own groups of size `1` as required.

This example demonstrates the solution approach and how it effectively groups people according to their desired group sizes while adhering to the provided constraints.

Solution Implementation

Python

```
from collections import defaultdict
from typing import List

class Solution:
    def groupThePeople(self, group_sizes: List[int]) -> List[List[int]]:
        # Dictionary to hold the groups according to their sizes.
        groups = defaultdict(list)

        # Iterate over the list of group sizes with their corresponding indices.
        for idx, size in enumerate(group_sizes):
            # Append the index to the list in the dictionary where key is the size.
            groups[size].append(idx)

        # For each group size (size key) and list of indices (indices list) in the groups,
        # create subgroups by slicing the list into chunks of length equal to the group size.
        result = [indices_list[i : i + size_key] for size_key, indices_list in groups.items()
                  for i in range(0, len(indices_list), size_key)]

        # Return the list of grouped people based on the sizes.
        return result
```

Java

```
import java.util.ArrayList; // Import the ArrayList class
import java.util.Arrays;    // Import the Arrays utility class
import java.util.List;      // Import the List interface

class Solution {
    public List<List<Integer>> groupThePeople(int[] groupSizes) {
        int numPeople = groupSizes.length; // Get the number of people
        List<Integer>[] groupsBySize = new List[numPeople + 1];  // Create an array of lists to hold groups of each size

        // Initialize each list in the array
        Arrays.setAll(groupsBySize, k -> new ArrayList<>());

        // Group people based on their group size
        for (int i = 0; i < numPeople; ++i) {
            groupsBySize[groupSizes[i]].add(i);
        }

        List<List<Integer>> groupedPeople = new ArrayList<>(); // List to hold the final grouped people

        // Process each non-empty group
        for (int size = 0; size < groupsBySize.length; ++size) {
            List<Integer> peopleInGroup = groupsBySize[size]; // Get the list of people in the current group size

            // Subdivide the current group size list into the correct group size
            for (int i = 0; i < peopleInGroup.size(); i += size) {
                // Add the sublist of people who form a group to the final list
                groupedPeople.add(peopleInGroup.subList(i, i + size));
            }
        }

        return groupedPeople; // Return the final grouped people list
    }
}
```

C++

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

class Solution {
public:
    vector<vector<int>> groupThePeople(vector<int>& groupSizes) {
        int numPeople = groupSizes.size();                          // Get the total number of people
        vector<vector<int>> groupsBy2D(numPeople + 1);              // Create a vector of vectors to store groups by their ID
        // Group people based on their group size
        for (int i = 0; i < numPeople; ++i) {
            groupsBy2D[groupSizes[i]].push_back(i);                 // Add person i to the group that corresponds to their group size
        }
        vector<vector<int>> result;                                 // Declare the resulting vector of groups
        // Iterate through each group size
        for (int size = 0; size < groupsBy2D.size(); ++size) {
            // Process current group size if there are groups of 'size'
            for (int i = 0; i < groupsBy2D[size].size(); i += size) {
                result.push_back(groupsBy2D[size].begin() + i, groupsBy2D[size].begin() + i + size); // Create a group of 'size'
                                                                    // Add the group to the result vector
            }
        }
        return result;                                              // Return the result
    }
};
```

TypeScript

```
function groupThePeople(groupSizes: number[]): number[][] {
    // Initialize the result array which will store the subgroups.
    const subGroups: number[][] = [];
    // Create a Map to keep track of groups and their members' indexes.
    const groupMap = new Map<number, number[]>();
    const totalMembers = groupSizes.length;

    // Iterate over each member's group size.
    for (let i = 0; i < totalMembers; ++i) {
        const currentSize = groupSizes[i];

        // Retrieve the current group list from the map or initialize it if it doesn't exist.
        let groupList = groupMap.get(currentSize) ?? [];

        // Add the current member's index to the group list.
        groupList.push(i);

        // Update the map with the new member's index added to the group.
        groupMap.set(currentSize, groupList);

        // If the group list reaches the expected group size, add it to the result array.
        if (groupList.length === currentSize) {
            subGroups.push(groupList);

            // Reset the group in the map as we've completed forming a group of the currentSize.
            groupMap.set(currentSize, []);
        }
    }
    // Return the array of groups formed.
    return subGroups;
}
```

```
from collections import defaultdict
from typing import List

class Solution:
    def groupThePeople(self, group_sizes: List[int]) -> List[List[int]]:
        # Dictionary to hold the groups according to their sizes.
        groups = defaultdict(list)

        # Iterate over the list of group sizes with their corresponding indices.
        for idx, size in enumerate(group_sizes):
            # Append the index to the list in the dictionary where key is the size.
            groups[size].append(idx)

        # For each group size (size key) and list of indices (indices list) in the groups,
        # create subgroups by slicing the list into chunks of length equal to the group size.
        result = [indices_list[i : i + size_key] for size_key, indices_list in groups.items()
                  for i in range(0, len(indices_list), size_key)]

        # Return the list of grouped people based on the sizes.
        return result
```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed as follows:

- The loop `for i, v in enumerate(groupSizes)` iterates over the list `groupSizes`, so it will have a complexity of $O(n)$ where n is the length of `groupSizes`.
- The list comprehension `[v[j : j + i] for i, v in g.items() for j in range(0, len(v), i)]` will iterate over each group `i` in the dictionary `g.items()` and then process chunks of size `i` within each group. Since each element from the `groupSizes` ends up in exactly one chunk, the total number of elements processed through the nested iterations is still $O(n)$.

Overall, the time complexity of the code is $O(n)$ as both the loop and the list comprehension depend linearly on the size of the input `groupSizes`.

Space Complexity

The space complexity can be determined by the additional space used by the data structures in the algorithm:

- A dictionary `g` is populated with potentially n elements (in the worst-case scenario where each group consists of one person), thus it has a complexity of $O(n)$.
- The list comprehension generates a new list that will contain n elements (each person in their respective group), so this also has a complexity of $O(n)$.

Hence, the space complexity of the code is $O(n)$, where n is the number of elements in `groupSizes`.