846. Hand of Straights

Medium Greedy Array Hash Table Sorting

Leetcode Link

Problem Description

Alice has a collection of cards, each with a number written on it. To win the game, Alice needs to organize these cards into several groups where each group must have the following properties:

1. Each group contains exactly `groupSize` cards.
2. The `groupSize` cards in each group must be consecutive cards by their numbers.

The challenge is to determine if it is possible for Alice to rearrange her cards to meet the criteria above. We are given an integer array `hand`, which represents the cards Alice has (where each element is the number on the card), and an integer `groupSize`. The goal is to return `true` if Alice can rearrange the cards into the desired groups, or `false` if she cannot.

Intuition

The intuition behind the solution is to count each card and then try to form groups starting with the lowest card value. Here's the thinking process:

1. Count the occurrence of each card value using a `Counter` data structure. This lets us know how many times each card appears in the hand.
2. Sort the `hand` array so that we can process the cards from the smallest to the largest. Sorting is crucial because we want to form groups starting with the smallest cards available.
3. Iterate through the sorted cards. For each card value `v` that still has occurrences left in the counter (i.e., `cnt[v]` is not zero):
   a. Try to make a group of `groupSize` starting from `v`. Check each card value `x` from `v` to `v + groupSize - 1` to ensure they exist (i.e., their count is greater than zero in the counter).
   b. If any card value `x` in this range is not found (i.e., `cnt[x]` is zero), it means we can't form a group starting from `v`, and we return `false`.
   c. If the card is found, decrement the count for that card in the counter since it's now part of a group.
4. After attempting to form groups for all card values, if no problems arise, it means it is possible to rearrange the cards into groups satisfying Alice's conditions, and we return `true`.

By following this process, we can efficiently determine whether or not it is possible to organize Alice's hand into groups of `groupSize` consecutive cards.

Solution Approach

The implementation of the solution leverages a few key ideas in order to check if the cards in Alice's hand can be rearranged into groups of consecutive numbers. Here is a step-by-step breakdown of the solution with reference to the algorithms, data structures, or patterns used:

1. **Use of `Counter` from the collections library**: The solution begins by counting the frequency of each card using the `Counter` data structure. This allows us to keep track of how many copies of each card we have and efficiently update the counts as we form groups.

2. **Sorting the cards**: Before we can form our groups, we sort the array `hand`. This is essential because we need to look at the smallest card first and attempt to group it with the next `groupSize - 1` consecutive card numbers.

3. **Iterating through sorted cards**: We begin a `for` loop through each card value `v` in the sorted hand.

```
for v in sorted(hand):
```

4. **Forming groups by checking card availability**: For each card `v` that is present in the counter (i.e., `cnt[v] > 0`), we look for the next `groupSize` consecutive numbers. We iterate from `v` up to `x + groupSize`, checking if each consecutive card `x` is available.

```
for x in range(v, v + groupSize):
```

5. **Checking and decrementing count**: Each time we find the card `x` is available (i.e., `cnt[x] > 0`), we decrement its count in our counter to indicate that it is now part of a group. If `x` is not available (i.e., `cnt[x] == 0`), we cannot form the required group and return `False`.

```
if cnt[x] == 0:
    return False
cnt[x] -= 1
```

6. **Optimizing by removing zero counts**: Once a card's count reaches zero, we remove it from the counter. This is an optimization step that reduces the size of the counter object as we continue through the cards. It's not strictly necessary for the correctness of the algorithm but can improve performance by reducing the lookup time for the remaining items.

```
if cnt[x] == 0:
    cnt.pop(x)
```

7. **Success condition**: If we successfully iterate through all the card values without finding a group that cannot be formed (that is, we never return `False`), it means that it is possible to rearrange the cards as desired. In that case, the function returns `True`.

By employing this solution approach, we are able to effectively determine the possibility of arranging Alice's cards into the specified groups, exploiting the capabilities of the `Counter` data structure for efficient counting and updating, as well as the ordered processing of the cards based on their values.

Example Walkthrough

Let's say Alice has the following cards in her hand: `[1, 2, 3, 6, 2, 3, 4]`, and the group size she wants to form is `3`. We want to determine if she can organize her cards into groups where each group contains exactly three consecutive cards.

1. **Count the occurrences of each card value using `Counter`:**
   - The count will be `{1: 1, 2: 2, 3: 2, 4: 1, 6: 1}`.
2. **Sort the `hand` array:**
   - The sorted hand is `[1, 2, 2, 3, 3, 4, 6]`.
3. **Iterate through the sorted cards:**
   - Start with the smallest value: `1`.
   - Attempt to find `1, 2, 3` for the first group. All are present, so decrement their counts: `{2: 1, 3: 1, 4: 1, 6: 1}`.
   - The next smallest is `2` (since `1` is no longer there).
   - Attempt to find `2, 3, 4` for the next group. All are present, so decrement their counts: `{3: 0, 4: 0, 6: 1}`.
   - Counter now contains only `{6: 1}`, which is not enough to form a group of three consecutive numbers.
4. **Encountering an issue forming groups:**
   - We are left with the card `6` which cannot form a group with two other consecutive numbers.
   - Therefore, it is *not* possible for Alice to organize her cards into the required groups.

Following the steps outlined in the solution approach, we checked each card and its availability to form groups, updated the counts, and concluded that the arrangement is not possible. We would return `False` in this case.

Python Solution

```python
from collections import Counter

class Solution:
    def is_n_straight_hand(self, hand: List[int], group_size: int) -> bool:
        # Count the frequency of each card in the hand
        card_count = Counter(hand)

        # Sort the hand to form groups in ascending order
        for card_value in sorted(hand):
            # If the current card can start a group
            if card_count[card_value]:
                # Attempt to form a group of the specified size
                for next_card_value in range(card_value, card_value + group_size):
                    # If the next card isn't available, the group can't be formed
                    if card_count[next_card_value] == 0:
                        return False
                    # Decrement the count of the current card forming the group
                    card_count[next_card_value] -= 1
                    # If all instances of this card have been used, remove it from the counter
                    if card_count[next_card_value] == 0:
                        card_count.pop(next_card_value)
        # If all cards can be successfully grouped, return True
        return True
```

Java Solution

```java
class Solution {
    public boolean isNStraightHand(int[] hand, int groupSize) {
        // Creating a map to count the frequency of each card value in the hand
        Map<Integer, Integer> cardCounts = new HashMap<>();
        for (int card : hand) {
            cardCounts.put(card, cardCounts.getOrDefault(card, 0) + 1);
        }

        // Sorting the hand array to ensure we create sequential groups starting from the lowest card
        Arrays.sort(hand);

        // Iterating through sorted hand
        for (int card : hand) {
            // If the card is still in cardCounts, it means it hasn't been grouped yet
            if (cardCounts.containsKey(card)) {
                // Creating a group starting with the current card
                for (int currentCard = card; currentCard < card + groupSize; ++currentCard) {
                    // If the current card does not exist in cardCounts, we can't form a group
                    if (!cardCounts.containsKey(currentCard)) {
                        return false;
                    }
                    // Decrement the count of the current card, as it has been used in the group
                    cardCounts.put(currentCard, cardCounts.get(currentCard) - 1);
                    // If the count goes to zero, remove the card from the map as it is all used up
                    if (cardCounts.get(currentCard) == 0) {
                        cardCounts.remove(currentCard);
                    }
                }
            }
        }

        // If we've formed groups with all cards without returning false, return true
        return true;
    }
}
```

C++ Solution

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

class Solution {
public:
    bool isNStraightHand(vector<int>& hand, int groupSize) {
        // Create a map to count the frequency of each card.
        unordered_map<int, int> cardCounts;
        for (int card : hand) {
            ++cardCounts[card]; // Increment the count for each card.
        }

        // Sort the hand to arrange the cards in ascending order.
        sort(hand.begin(), hand.end());

        // Traverse through the sorted hand.
        for (int card : hand) {
            // If the current card is still in count map (i.e., needed to form a group).
            if (cardCounts.count(card)) {
                // Attempt to create a group starting with the current card.
                for (int nextCard = card; nextCard < card + groupSize; ++nextCard) {
                    // If the next card in the sequence is missing, can't form a group.
                    if (!cardCounts.count(nextCard)) {
                        return false;
                    }
                    // Decrement count for the current card in the sequence.
                    if (--cardCounts[nextCard] == 0) {
                        cardCounts.erase(nextCard); // Remove the card from count map if count reaches zero.
                    }
                }
            }
        }

        // If all cards can be grouped, return true.
        return true;
    }
};
```

Typescript Solution

```typescript
// Import necessary functionalities from the 'lodash' library.
import _ from 'lodash';

// Define a function to check if the hand can be arranged in groups of consecutive cards of groupSize.
// The function accepts a hand (array of numbers) and a groupSize (number).
function isNStraightHand(hand: number[], groupSize: number): boolean {
    // Create a map (an object) to count the frequency of each card.
    const cardCounts: { [key: number]: number } = {};
    hand.forEach(card => {
        cardCounts[card] = (cardCounts[card] || 0) + 1; // Increment the count for each card.
    });

    // Create a sorted copy of the hand to arrange the cards in ascending order.
    const sortedHand = _.sortBy(hand);

    // Traverse through the sorted hand.
    for (const card of sortedHand) {
        // If the count for the current card is more than 0 (i.e., needed to form a group).
        if (cardCounts[card] > 0) {
            // Attempt to create a group starting with the current card.
            for (let nextCard = card; nextCard < card + groupSize; ++nextCard) {
                // If the next card in the sequence is missing or count is 0, can't form a group.
                if (!cardCounts[nextCard]) {
                    return false;
                }
                // Decrement count for the current card in the sequence.
                --cardCounts[nextCard];
            }
        }
    }

    // If all cards can be grouped, return true.
    return true;
}
```

Time and Space Complexity

The given Python function `isNStraightHand` checks whether an array of integers (`hand`) can be partitioned into groups of `groupSize` where each group consists of consecutive integers.

Time Complexity

The time complexity of the function can be broken down as follows:

1. **Counting Elements (Counter):** Constructing a counter for the hand array takes `O(N)` time, with `N` being the length of the `hand` array.

2. **Sorting:** The `sorted(hand)` function has a time complexity of `O(N log N)` since it sorts the array.

3. **Iteration Over Sorted Hand:** The outer loop runs at most `N` times. However, within this loop, there is an inner loop that runs up to `groupSize` times. This results in a total of `O(N * groupSize)` operations in the worst case.

Therefore, the overall worst-case time complexity of the function is `O(N log N + N * groupSize)`. Since `groupSize` is a constant, it can be simplified to `O(N log N)`.

Space Complexity

The space complexity pertains to the additional memory used by the algorithm as the input size grows. For this function:

1. **Counter Space Usage:** The `Counter` used to count instances of elements in `hand` will use `O(N)` space.

2. **Sorted Array Space:** The `sorted()` function creates a new list and thus, requires `O(N)` space as well.

Since both the `Counter` and the sorted list exist simultaneously, the overall space complexity would also be `O(N)`.