2260. Minimum Consecutive Cards to Pick Up

Medium Array Hash Table Sliding Window

Problem Description

In this problem, you're given an array `cards` with each element representing the value of a card. The task is to find the minimum number of consecutive cards you need to pick from the array to get a pair of matching cards (i.e., two cards with the same value). If it is possible to get a matching pair by picking cards consecutively, you should return the minimum number of cards you need to pick. If no matching pairs exist in the sequence of cards, the function should return `-1`.

Intuition

To solve this problem, we can utilize a **HashMap** to keep track of the last seen indices of card values. As we iterate over the array of cards, we check if the current card's value has already been seen before by consulting the HashMap. If it has been seen before, we subtract the previous index where this value was seen from the current index to find the distance between them, which also includes the currently picked card (hence we add 1). We continuously update the minimum distance whenever we find a pair of matching cards.

1. Start by initializing a HashMap `last` to store the last index where each card value was seen.
2. Initialize a variable `ans` to store the answer, initialized to `inf` (infinity), which represents that the initial distance is infinitely large.
3. Iterate over the cards array using an index `i` and the card value `x`: a. If the card `x` is already in the HashMap (`last`), it means a previous card with the same value was seen. Therefore, calculate the distance from the current card to the last card with the same value, which can be done by `i - last[x] + 1`, and update the answer `ans` by the minimum of the current answer and this new distance. b. Update the `last` HashMap to mark the current index `i` as the latest index at which the card value `x` was seen.
4. After the loop, check if `ans` is still `inf`. If it is, that means no matching cards were found, and you should return `-1`. Otherwise, return `ans` as the minimum number of consecutive cards to pick up to have a pair of matching cards.

The strength of this approach lies in its time complexity. Because the HashMap access/update is `O(1)` on average and the iteration of the cards is `O(n)` where `n` is the number of cards, the overall time complexity of this approach is `O(n)` which is efficient.

Solution Approach

The solution uses a simple yet effective algorithm combined with a dictionary (a.k.a. HashMap in other languages) data structure to efficiently track the last occurrence index of card values.

Here's a walkthrough of the code implementation:

- A dictionary named `last` is created to store the last occurrence index of each card value encountered as we iterate over the array.
- A variable named `ans` is initialized with `inf`, which represents infinity. This variable will eventually hold the minimum number of cards required to find a matching pair or stay as infinity if no match is found.
- The code then iterates over each card in the `cards` array using a `for` loop with the index `i` and card value `x`.
  - If the card value `x` is found in the `last` dictionary, this implies that we have encountered this value before, and therefore, we have found a pair of matching cards. The current distance to the last seen matching card is calculated by `i - last[x] + 1`. `+1` is included because both the current card and the last card are part of the set we are considering. We then update `ans` with the minimum of its current value and the newly calculated distance.
  - Whether a match is found or not, the `last` dictionary is updated such that `x` now points to the current index `i`. This operation ensures that the next time `x` is encountered, the distance will be calculated from this point.
- After the loop concludes, the `ans` variable is checked to determine whether it still contains `inf` (meaning no pairs were found). If `ans` is still `inf`, the function returns `-1`, as it is not possible to have matching cards. Otherwise, it returns the value of `ans`, which is the minimum number of consecutive cards needed to pick up to get a matching pair.

The efficiency of the algorithm comes from the use of the `last` dictionary, which allows constant time lookup and update for the indices of card values. This means that the algorithm will perform well, even for large arrays, as the time complexity remains linear (`O(n)`), where `n` is the number of cards.

Example Walkthrough

To illustrate the solution approach, let's consider a small example using the array of cards: `[5, 1, 3, 4, 5, 6, 7, 3]`.

- We start with an empty dictionary `last` and initialize `ans` to `inf`.
- **Iteration 1**: Card value is `5`. Since `5` is not present in the dictionary `last`, we add it to `last` with index `0`: `last[5] = 0`.
- **Iteration 2**: Card value is `1`. It's also not in `last`, thus we add `last[1] = 1`.
- **Iteration 3**: Card value is `3`. We add it to `last`: `last[3] = 2`.
- **Iteration 4**: Card value is `4`. We add `last[4] = 3`.
- **Iteration 5**: Card value is `5`. This time, `5` is already in `last` with the index `0`. We calculate the distance: `i - last[5] + 1 = 5 - 0 + 1 = 6`. We then update `ans` to `6` since `6 < inf`. We also update `last[5]` to the current index: `last[5] = 4`.
- **Iteration 6**: Card value is `6`. We add `last[6] = 5`.
- **Iteration 7**: Card value is `7`. We add `last[7] = 6`.
- **Iteration 8**: Card value is `3`. As `3` is in `last` with the index `2`, we find another pair. We calculate the distance: `i - last[3] + 1 = 8 - 2 + 1 = 7`. We compare this with the current `ans`, which is `6`, and since `7` is larger, we don't update `ans`. Update `last[3] = 7`.

After the iterations, the smallest value in `ans` that was updated is `6`. Therefore, the minimum number of consecutive cards you need to pick up to get a pair of matching cards is `6`.

In contrast, if our cards array was something like `[8, 5, 1, 3, 4]` where no values repeat, at the end of our iteration, the `ans` would still be `inf`, and we would return `-1` since there are no consecutive cards that form a pair.

Solution Implementation

Python

```python
from math import inf

class Solution:
    def minimumCardPickup(self, cards: List[int]) -> int:
        # Create a dictionary to keep track of the last index where each card was seen.
        last_seen = {}
        # Initialize the answer to infinity to represent a large number.
        min_pickup_length = inf

        # Iterate over the list of cards with their indices.
        for index, card_value in enumerate(cards):
            # If the card was seen before, calculate the pickup length.
            if card_value in last_seen:
                # Update the minimum pickup length if a shorter one is found.
                min_pickup_length = min(min_pickup_length, index - last_seen[card_value] + 1)
            # Update the last seen index for the current card.
            last_seen[card_value] = index

        # Return -1 if the answer remains infinity (no pickup found), else return the minimum pickup length.
        return -1 if min_pickup_length == inf else min_pickup_length
```

Java

```java
class Solution {
    public int minimumCardPickup(int[] cards) {
        // Create a map to store the last index of each card value
        Map<Integer, Integer> lastIndexMap = new HashMap<>();
        int numOfCards = cards.length;
        // Initialize the smallest sequence length to maximum possible value
        int minSequenceLength = numOfCards + 1;

        // Iterate through each card in the array
        for (int i = 0; i < numOfCards; ++i) {
            // If the current card has been seen before...
            if (lastIndexMap.containsKey(cards[i])) {
                // Update the smallest sequence length with the minimum between
                // the current smallest sequence length and
                // the length of the current sequence of cards
                minSequenceLength = Math.min(minSequenceLength, i - lastIndexMap.get(cards[i]) + 1);
            }
            // Update the last index for this card value
            lastIndexMap.put(cards[i], i);
        }

        // If no sequence is found (minSequenceLength was not updated), return -1
        // Otherwise, return the smallest sequence length
        return minSequenceLength > numOfCards ? -1 : minSequenceLength;
    }
}
```

C++

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

class Solution {
public:
    // Function to find the minimum number of cards to be picked up in order
    // to get a pair of cards with the same value.
    int minimumCardPickup(vector<int>& cards) {
        unordered_map<int, int> lastIndex; // Stores the last index where each card was seen
        int n = cards.size(); // The number of cards
        int minPickup = n + 1; // Initialize it to an impossible maximum

        // Iterate over the cards
        for (int i = 0; i < n; ++i) {
            // If we have seen cards[i] before, calculate the distance from its last occurrence
            if (lastIndex.count(cards[i])) {
                minPickup = min(minPickup, i - lastIndex[cards[i]] + 1);
            }
            // Update the last seen index of cards[i]
            lastIndex[cards[i]] = i;
        }

        // If minPickup did not change from its initial value, no pair was found; return -1
        // Otherwise, return the minimum number of cards picked up to find a pair
        return minPickup > n ? -1 : minPickup;
    }
};
```

TypeScript

```typescript
function minimumCardPickup(cards: number[]): number {
    // Store the length of the cards array.
    const cardCount = cards.length;
    // Create a new map to store the last occurrence index of each card.
    const lastIndexMap = new Map<number, number>();
    // Initialize the answer to be larger than any possible minimum pickup length.
    let minPickupLength = cardCount + 1;
    // Iterate through the cards.
    for (let index = 0; index < cardCount; ++index) {
        // Check if we have seen the current card before.
        if (lastIndexMap.has(cards[index])) {
            // Update the minimum pickup length if we've found a shorter subarray.
            minPickupLength = Math.min(minPickupLength, index - lastIndexMap.get(cards[index])! + 1);
        }
        // Update the map with the latest index of the current card.
        lastIndexMap.set(cards[index], index);
    }
    // If the answer is still larger than any possible value, return -1 as no valid subarray was found.
    return minPickupLength > cardCount ? -1 : minPickupLength;
}
```


```python
from math import inf

class Solution:
    def minimumCardPickup(self, cards: List[int]) -> int:
        # Create a dictionary to keep track of the last index where each card was seen.
        last_seen = {}
        # Initialize the answer to infinity to represent a large number.
        min_pickup_length = inf

        # Iterate over the list of cards with their indices.
        for index, card_value in enumerate(cards):
            # If the card was seen before, calculate the pickup length.
            if card_value in last_seen:
                # Update the minimum pickup length if a shorter one is found.
                min_pickup_length = min(min_pickup_length, index - last_seen[card_value] + 1)
            # Update the last seen index for the current card.
            last_seen[card_value] = index

        # Return -1 if the answer remains infinity (no pickup found), else return the minimum pickup length.
        return -1 if min_pickup_length == inf else min_pickup_length
```


Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the number of cards. This is because the code iterates through the list of cards exactly once with a single loop (`for i, x in enumerate(cards):`). Within this loop, checking if `x` is in `last` (`if x in last:`) and updating `last[x]` (`last[x] = i`) are both operations that take `O(1)` time on average when using a dictionary in Python.

Space Complexity

The space complexity of the provided code is `O(u)`, where `u` is the number of unique cards. This is because a dictionary `last` is used to store the last index at which each card appears. In the worst-case scenario where all card values are unique, the dictionary would need to store an entry for each card, thus requiring `O(u)` space.