950. Reveal Cards In Increasing Order

Medium Queue Array Sorting Simulation Leetcode Link

Problem Description

The problem presents a scenario where we have an array called `deck` representing a deck of cards, where each card has a unique integer value. We aim to find an order for this deck such that when we follow a specific process, the cards will be revealed in ascending order.

The process is as follows:

1. Take the top card, reveal it, and remove it from the deck.
2. If there are still cards left in the deck, place the next top card at the bottom of the deck.
3. Repeat steps 1 and 2 until all cards are revealed.

The challenge is to determine the order in which to organize the `deck` initially so that when the above process is followed, the cards are revealed in increasing order.

Intuition

To understand the solution, let's work our way backward. Think about the last two cards that are revealed. For these cards to come out in ascending order, the smaller of the two must be placed at the top of the deck first, followed by the larger card beneath it. This way, during the final iteration of our revealing process, the smaller card is taken and the larger card is placed to the bottom of the deck to be taken in next iteration.

Now, consider the third last card. In order for this card to end up being revealed just before the last two, it must be inserted at the top of the current sequence, pushing the other two cards down one spot, and then cycling the (previously) top card to the bottom of the deck.

By applying this logic repeatedly in reverse, we can construct the initial ordering.

Here is how the thought process translates into our solution approach:

1. Sort the `deck` in reverse order; this ensures we're placing cards in order from highest to lowest.
2. Initialize a deque (double-ended queue) which will allow us to manipulate the order of cards easily.
3. Iterate over each card in reverse sorted order: a. If the deque already has cards, take the bottom card and place it on top (simulating the second step of the revealing process but in reverse). b. Place the current card on top of the deque (simulating that it's the next to be revealed in reverse order of steps).
4. Once all cards have been placed in the deque, we convert it to a list and return that as our solution. This list will yield the cards in ascending order when the described process is applied.

Solution Approach

The solution approach makes use of the following data structures and algorithms:

1. **Sorting**: The very first step involves sorting the array `deck` in reverse order. We do this because we want to place the cards in the deque starting from the highest value to the lowest, effectively building the correct order from the end state back to the start.

2. **Deque (Double-ended queue)**: A deque is used because we need a data structure that allows inserting and deleting elements from both ends efficiently. In Python, `deque` is typically implemented with doubly linked lists, which makes these operations very fast.

3. **Simulating the process in reverse**: The key idea behind the solution is to simulate the revealing process in reverse to construct the initial ordering.

The solution approach step by step:

- First, sort the deck in descending order using the `sorted()` function with `reverse=True`.
- Initialize an empty `deque` from `collections`. This will hold the cards in the order they should be arranged initially.
- Iterate through each card value in the `deck`, starting with the largest value:
  - On each iteration (except the first), since we are simulating the process backward:
    - Remove the card currently at the bottom of the `deque` by using `pop()`.
    - Place that card on top of the `deque` by using `appendleft()`. This simulates moving the next top card to the bottom of the deck in the revealing process, but in reverse.
  - Then, place the current card value on the top of the `deque` using `appendleft()`.
- After iterating through all the cards, convert the `deque` to a list with `list(q)` before returning it.

The `deque` is manipulated using `appendleft()` and `pop()` to ensure that when we "reverse" the steps taken to reveal the cards, the cards would end up in ascending order. Converting the `deque` back into a list gives us the desired initial order of cards.

Example Walkthrough

Let's illustrate the solution approach with a small example deck: `deck = [17, 13, 11, 2, 3, 5, 7]`.

1. First, we sort the `deck` in descending order: `sorted_deck = [17, 13, 11, 7, 5, 3, 2]`.

2. We initialize an empty deque: `q = deque()`.

3. We then iterate over the `sorted_deck` and apply the logic:

   a. Iteration with card 17 (first card, so no deque manipulation):

   - `q` becomes `deque([17])`.

   b. Iteration with card 13:

   - Take card 17 from bottom (`pop`) and move to top (`appendleft`): `q` becomes `deque([17])`.
   - Add card 13 on top (`appendleft`): `q` becomes `deque([13, 17])`.

   c. Iteration with card 11:

   - Take card 13 from bottom and move to top: `q` becomes `deque([13, 17])`.
   - Add card 11 on top: `q` becomes `deque([11, 13, 17])`.

   d. Iteration with card 7:

   - Take card 11 from bottom and move to top: `q` becomes `deque([11, 13, 17])`.
   - Add card 7 on top: `q` becomes `deque([7, 11, 13, 17])`.

   e. Iteration with card 5:

   - Take card 7 from bottom and move to top: `q` becomes `deque([7, 11, 13, 17])`.
   - Add card 5 on top: `q` becomes `deque([5, 7, 11, 13, 17])`.

   f. Iteration with card 3:

   - Take card 5 from bottom and move to top: `q` becomes `deque([5, 7, 11, 13, 17])`.
   - Add card 3 on top: `q` becomes `deque([3, 5, 7, 11, 13, 17])`.

   g. Iteration with card 2:

   - Take card 3 from bottom and move to top: `q` becomes `deque([3, 5, 7, 11, 13, 17])`.
   - Add card 2 on top: `q` becomes `deque([2, 3, 5, 7, 11, 13, 17])`.

4. Finally, we convert `q` to a list to obtain the resulting order for the `deck`: `resulting_deck = list(q) = [2, 3, 5, 7, 11, 13, 17]`.

When we apply the described reveal process to `resulting_deck`, the cards will be revealed in ascending order, matching the original `deck` before we sorted it.

Python Solution

```python
from collections import deque

class Solution:
    def deckRevealedIncreasing(self, deck: List[int]) -> List[int]:
        # Initialize an empty double-ended queue (deque)
        deque_cards = deque()

        # Sort the deck in descending order and iterate over the cards
        for card in sorted(deck, reverse=True):
            # If the deque is not empty, move the last element to the front
            if deque_cards:
                deque_cards.appendleft(deque_cards.pop())

            # Insert the current card to the front of the deque
            deque_cards.appendleft(card)

        # Convert the deque back to a list before returning it
        return list(deque_cards)
```

Java Solution

```java
class Solution {
    public int[] deckRevealedIncreasing(int[] deck) {
        // Initialize a deque to simulate the revealing process
        Deque<Integer> deque = new ArrayDeque<>();

        // Sort the deck in ascending order so that we can reveal them in increasing order
        Arrays.sort(deck);

        // Get the length of the deck
        int deckLength = deck.length;

        // Start from the last card of the sorted deck,
        // which is the maximum card and simulate the reveal process in reverse
        for (int i = deckLength - 1; i >= 0; --i) {
            // If the deque is not empty, move the last card to the front
            // This simulates the "put the last card on the bottom" step in reverse
            if (!deque.isEmpty()) {
                deque.offerFirst(deque.pollLast());
            }
            // Place the current card on top of the deque
            // This simulates the "reveal the top card" step in reverse
            deque.offerFirst(deck[i]);
        }

        // Initialize an array to store the correctly ordered deck
        int[] result = new int[deckLength];

        // Move cards from the deque back to the result array
        for (int i = deckLength - 1; i >= 0; --i) {
            result[i] = deque.pollLast();
        }

        // Return the result array which has the deck ordered
        // to reveal cards in increasing order
        return result;
    }
}
```

C++ Solution

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

class Solution {
public:
    vector<int> deckRevealedIncreasing(vector<int>& deck) {
        // reverse sort the deck so that we get the largest card first
        sort(deck.rbegin(), deck.rend());

        // initialize a double-ended queue to simulate the process
        deque<int> queue;

        // iterate over the sorted deck
        for (int card : deck) {
            // if the queue is not empty, simulate the 'revealing' process:
            // move the last element to the front to mimic the card placement in the final deck
            if (!queue.empty()) {
                queue.push_front(queue.back());
                queue.pop_back();
            }
            // place the current largest card in the front
            queue.push_front(card);
        }

        // convert the deque back to a vector and return it
        return vector<int>(queue.begin(), queue.end());
    }
};
```

Typescript Solution

```typescript
// Function sorts the input array in non-increasing order and
// returns a new array that simulates the deck revealing process
function deckRevealedIncreasing(deck: number[]): number[] {
    // Sort the deck in non-increasing order to get the largest card first
    deck.sort((a, b) => b - a);

    // Initialize a deque structure to simulate the process
    const deque: number[] = [];

    // Iterate over the sorted deck
    for (const card of deck) {
        // If the deque is not empty, simulate the revealing process:
        // Move the last element to the front to mimic the card placement in the final deck
        if (deque.length > 0) {
            deque.unshift(deque.pop()!);
        }
        // Place the current largest card in the front
        deque.unshift(card);
    }

    // Return the deque which now represents the final deck order
    return deque;
}

// Example usage:
// const result = deckRevealedIncreasing([17, 13, 11, 2, 3, 5, 7]);
// console.log(result);
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed based on the operations it performs:

1. `sorted(deck, reverse=True)`: This operation has a time complexity of `O(n log n)`, where `n` is the number of elements in `deck`. Sorting a list is commonly done using algorithms like Timsort in Python, which have this complexity.

2. The for loop iterates over each of the `n` elements in the sorted deck.

   - In each iteration, checking if the queue `q` is not empty is `O(1)`.
   - The `q.pop()` operation (when the queue is not empty) also has a time complexity of `O(1)` because popping from the right end of a deque in Python is done in constant time.
   - The `q.appendleft()` operation has a time complexity of `O(1)` as well, since appending to the left end of a deque is a constant time operation.

However, because these `O(1)` operations are executed for each of the `n` elements of the deck, the loop contributes `O(n)` to the overall time complexity.

When combined, the sorting operation dominates the time complexity, resulting in an overall time complexity of `O(n log n)`.

Space Complexity

The space complexity of the code is determined by the extra data structures used:

1. The sorted list of `deck`: This does not require additional space, as the sort is usually done in-place in Python. Therefore, it is `O(1)`.

2. The deque `q`: In the worst case, it holds all `n` elements of the deck. This results in a space complexity of `O(n)`.

Considering both requirements, the overall space complexity of the code is `O(n)`.