649. Dota2 Senate

Medium Greedy Queue String Leetcode Link

Problem Description

In the given problem, we are simulating a political power struggle within a senate of a fantasy world known as Dota2. The senate is made up of two parties, Radiant and Dire. Each senator, in turn, has the opportunity to exercise one of two rights—either banning another senator, effectively removing them from the game, or declaring victory for their party if all remaining senators are from their own party.

The outcome we seek to predict is which party will ultimately succeed in passing a change in the Dota2 game by eliminating the opposition's ability to vote. The senators are presented in a string where each character represents a senator from either the Radiant (as 'R') or the Dire (as 'D'). Senators act in the order they appear in the string, and once a senator is banned, they are skipped in subsequent rounds of the process.

Since each senator is smart and employs the best strategy for their party, we need to simulate the rounds of banning to determine which party wins.

Intuition

The intuition behind the solution involves simulating the process using a queue data structure. For simplicity and efficiency, we use a separate queue for each party, recording the positions (indices) of their senators in the senate list. Our goal is to simulate each round where senators ban their immediate opponent senator (if available).

The key insights to arrive at the solution are:

1. Senators will always ban the next senator of the opposite party to maximize their party's chance of victory.
2. Once a senator has made a move, they go to the end of the queue but with an increased index representing their new position in the "virtual" order. This is done to maintain the cyclic nature of the senate arrangement.
3. The process continues until one party's queue is empty, meaning no more senators from that party are left to cast votes or make bans.

By dequeuing the first senator of each party and having them ban the opponent's first senator, we simulate the banning process while keeping track of the new positions. If a Radiant senator acts before a Dire senator, they add to the end of their queue by considering the size of the senate (`senate`), effectively banning the first Dire senator. The same logic applies when a Dire senator acts before a Radiant senator.

The simulation continues until one party has no remaining senators, at which point the surviving party is declared the winner. This approach ensures that we correctly identify which party would win in an ideal scenario where each senator acts optimally for their party's interest.

Solution Approach

The solution uses a greedy algorithm to simulate the senate's round-based decision-making process. A greedy algorithm makes the locally optimal choice at each stage with the hope of finding a global optimum. In this context, the local optimum is for each senator to ban an opposing party senator as early as possible.

We utilize two queues represented by the `deque` data structure from Python's `collections` module:

- `qr` queue stores the indices of the Radiant senators.
- `qd` queue stores the indices of the Dire senators.

The indices allow us to keep track of the order of the senators and their relative positions in the simulated rounds.

Here's the approach step by step, in alignment with the code provided:

1. Iterate through the `senate` string and fill the queues `qr` and `qd` with indices of the senators belonging to the Radiant and Dire parties, respectively.

2. Enter a loop that will run until one of the queues is empty. The condition `while qr and qd:` ensures that the loop continues as long as there are senators from both parties available to take action.

3. In each iteration of the loop:

   - Compare the indices at the front of both `qr` and `qd` which represents the order in which the senators will take action. The senator with the lower index is able to act first.
   - If the first Radiant senator (`qr[0]`) is before the first Dire senator (`qd[0]`), the Radiant senator will ban the Dire senator. The Radiant senator's index is then added back to the `qr` queue, incremented by `n`, which is the length of the `senate` string. This effectively places them at the end of the order for the next round.
   - Similarly, if the Dire senator acts first, they will ban the Radiant senator, and their index (incremented by `n`) will be added back to the `qd` queue.
   - After a senator has acted (either banning or being banned), we remove them from the front of the queue using `popleft()`.
4. After exiting the loop, we check which queue still has senators left, which determines the victorious party. If `qr` is not empty, Radiant wins; otherwise, Dire wins.

5. Finally, return `"Radiant"` if the `qr` queue has senators left, or `"Dire"` if the `qd` queue has senators left.

This approach ensures that each senator acts in the best interest of their party by banning the first available opposition senator and then waiting for their next turn at the end of the senate order.

Example Walkthrough

Let's walk through an example with the senate string `RDD`. We'll simulate the process to see which party comes out victorious.

1. We initialize two queues: `qr` for Radiant and `qd` for Dire. Given the senate string `RDD`, `qr` will initially contain `[0]` because the first senator (index 0) is from the Radiant party, and `qd` will contain `[1, 2]` because the second and third senators (indices 1 and 2) are from the Dire party.

2. Now, we enter the main loop where we process each senator's actions:

   - Both `qr` and `qd` are not empty, so we continue.
   - We compare the front of both queues: `qr[0]` is `0`, and `qd[0]` is `1`. Since the Radiant senator (`R`) at index 0 is the first in line, they act by banning the first Dire senator (`D`) at index 1.
   - We remove the banned Dire senator from queue `qd` by dequeuing it, and then we add the Radiant senator's index incremented by `n` (the length of the senate string) to represent their new virtual position at the end of the next round. Since `n` is 3 here, we add `0 + 3` to `qr`, which becomes `[3]`.
   - The `qd` queue now looks like `[2]`, as the first Dire senator was banned.
3. Continuing with the main loop:

   - Comparing the indices again, we see `qr[0]` is `3` (which is virtually `0` in the next round), and `qd[0]` is `2`. The Dire senator at index 2 acts next, since they are the only one left and thus have the lowest current index.
   - The Dire senator bans the first Radiant senator positioned at virtual index 3 (original index 0 returned to the queue). Since nobody is left in the Radiant queue now, `qd` is decremented by dequeuing the acting senator, and their new indexed position `2 + 3 = 5` is added back to the `qd` queue.
4. We check the queues after the loop iteration:

   - The `qr` queue is empty, which indicates that there are no more Radiant senators to take action.
   - The `qd` queue has one senator left at index 5 (virtually 2).
5. Since `qr` is empty and `qd` still has a senator, the victorious party is Dire.

In this example, the final output is `"Dire"`, and the process demonstrates the greedy approach of banning the opponent's next available senator to ensure the best outcome for one's own party.

Python Solution

```python
from collections import deque

class Solution:
    def predict_party_victory(self, senate: str) -> str:
        # Initialize queues for Radiant and Dire senators' indices
        queue_radiant = deque()
        queue_dire = deque()

        # Populate initial queues with the indices of the Radiant and Dire senators
        for index, senator in enumerate(senate):
            if senator == "R":
                queue_radiant.append(index)
            else:
                queue_dire.append(index)

        # Calculate the length of the senate for future indexing
        n = len(senate)

        # Process the two queues
        while queue_radiant and queue_dire:
            # Take the first senator from each queue and compare their indices
            if queue_radiant[0] < queue_dire[0]:
                # If the Radiant senator comes first, they ban a Dire senator
                # and put themselves at the back of their queue with a new hypothetical index
                queue_radiant.append(queue_radiant[0] + n)
            else:
                # If the Dire senator comes first, they ban a Radiant senator
                # and put themselves at the back of their queue with a new hypothetical index
                queue_dire.append(queue_dire[0] + n)

            # Remove the senators who have exercised their powers
            queue_radiant.popleft()
            queue_dire.popleft()

        # Return the winning party's name based on which queue still has senators
        return "Radiant" if queue_radiant else "Dire"

```

Java Solution

```java
class Solution {
    public String predictPartyVictory(String senate) {
        int totalSenators = senate.length();
        Deque<Integer> radiantQueue = new ArrayDeque<>();
        Deque<Integer> direQueue = new ArrayDeque<>();

        // Populate queues with the indices of 'R' and 'D' senators
        for (int i = 0; i < totalSenators; ++i) {
            if (senate.charAt(i) == 'R') {
                radiantQueue.offer(i);
            } else {
                direQueue.offer(i);
            }
        }

        // Process the queues until one of them is empty
        while (!radiantQueue.isEmpty() && !direQueue.isEmpty()) {
            int radiantIndex = radiantQueue.peek();
            int direIndex = direQueue.peek();

            // The senator with the lower index bans the opposing senator
            if (radiantIndex < direIndex) {
                // The radiant senator bans a dire senator and gets back in line
                radiantQueue.offer(radiantIndex + totalSenators);
            } else {
                // The dire senator bans a radiant senator and gets back in line
                direQueue.offer(direIndex + totalSenators);
            }

            // Remove the senators that have already made a ban
            radiantQueue.poll();
            direQueue.poll();
        }

        // Declare the winner depending on which queue is not empty
        return radiantQueue.isEmpty() ? "Dire" : "Radiant";
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to predict the winner of the senate dispute.
    string predictPartyVictory(string senate) {
        int n = senate.size(); // Get the size of the senate string

        // Queues to store the indices of 'R' and 'D' senators
        queue<int> radiantQueue;
        queue<int> direQueue;

        // Populate the initial queues with the indices of each senator
        for (int i = 0; i < n; ++i) {
            if (senate[i] == 'R') {
                radiantQueue.push(i);
            } else {
                direQueue.push(i);
            }
        }

        // Loop as long as both queues have senators remaining
        while (!radiantQueue.empty() && !direQueue.empty()) {
            int radiantIndex = radiantQueue.front(); // Get the index of the front radiant senator
            int direIndex = direQueue.front(); // Get the index of the front dire senator

            radiantQueue.pop(); // Remove the front radiant senator from the queue
            direQueue.pop(); // Remove the front dire senator from the queue

            // The senator with the smaller index bans the other from the next round
            if (radiantIndex < direIndex) {
                // Radiant senator wins this round and re-enters queue with index increased by n
                radiantQueue.push(radiantIndex + n);
            } else {
                // Dire senator wins this round and re-enters queue with index increased by n
                direQueue.push(direIndex + n);
            }
        }

        // If the radiant queue is empty, Dire wins; otherwise, Radiant wins
        return radiantQueue.empty() ? "Dire" : "Radiant";
    }
};

```

Typescript Solution

```typescript
function predictPartyVictory(senate: string): string {
    // Determine the length of the senate string
    const senateLength = senate.length;
    // Initialize queues to keep track of the indexes of 'R' (Radiant) and 'D' (Dire) senators
    const radiantQueue: number[] = [];
    const direQueue: number[] = [];

    // Populate the queues with the initial positions of the senators
    for (let i = 0; i < senateLength; ++i) {
        if (senate[i] === 'R') {
            radiantQueue.push(i);
        } else {
            direQueue.push(i);
        }
    }

    // Run the simulation until one party has no senators left
    while (radiantQueue.length > 0 && direQueue.length > 0) {
        // Remove the first senator in each queue to simulate a round
        const radiantSenatorIndex = radiantQueue.shift()!;
        const direSenatorIndex = direQueue.shift()!;

        // The senator with the lower index bans the opponent senator from the current round
        // Then, the winning senator gets re-added to the queue for the next round
        if (radiantSenatorIndex < direSenatorIndex) {
            radiantQueue.push(radiantSenatorIndex + senateLength);
        } else {
            direQueue.push(direSenatorIndex + senateLength);
        }
    }

    // After one party has no senators left, return the name of the winning party
    return radiantQueue.length > 0 ? 'Radiant' : 'Dire';
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(N)`, where `N` is the length of the `senate` string.

Reasoning:

- We loop through each character of the string to build the initial `qr` and `qd` queues - this is an `O(N)` operation.
- In the `while` loop, in each iteration, one senator from each party (`R` and `D`) gets 'compared', and one is turned 'inactive' for the current round. Each senator will be dequeued and potentially re-queued once per round. The number of rounds is proportional to the number of senators, because in each round at least one senator is banned from further participation until all senators from the opposing party have been banned.
- Since `qr.append(qr[0] + n)` and `qd.append(qd[0] + n)` just change the index for the next round, the `O(N)` operations within the loop are repeated N times in the worst case (every senator goes once per round). However, due to the nature of the problem, the loop will terminate when one party has no more active senators, so it does not strictly go N rounds.

Space Complexity

The space complexity of the code is `O(N)`, where `N` is the length of the `senate` string.

Reasoning:

- Two queues, `qr` for Radiant senators and `qd` for Dire senators, each can hold at most N elements if all the senators are from one party.
- No other data structures are used that grow with the input size - thus, the dominant term is the space used by the two queues.