# 933. Number of Recent Calls

Easy Design Queue Data Stream

## Problem Description

In this problem, you are tasked with designing a `RecentCounter` class that tracks the number of requests received within a specific time frame. The time frame in question is the last 3000 milliseconds, which equates to the last 3 seconds. This can be likened to monitoring network traffic, where you're interested in understanding the number of events or requests that have occurred in a sliding window of time for performance analysis or rate limiting.

The class should support two operations:

1. `RecentCounter()` Is the constructor that initializes the counter. When a `RecentCounter` object is created, it starts with zero recent requests.
2. `ping(int t)` is a method that is called each time a new request is received. The request comes with a timestamp `t` (in milliseconds), which is strictly increasing with each call. The purpose of `ping` is to add the new request and then return the count of all recent requests within the last 3000 milliseconds, which is the time range from `t - 3000` to `t`, inclusive of both ends of the interval.

## Intuition

The key to solving this problem is to maintain a dynamic list of timestamped requests, ensuring that only those within the last 3000 milliseconds are counted. Because the input guarantees that each `t` in `ping` is increasing, we can use a queue to keep track of the timestamps of the recent requests.

The intuition for using a queue comes from its FIFO (First-In-First-Out) property, which naturally aligns with the progression of time in our problem. When a new request arrives with timestamp `t`, we add it to the end of the queue. Then we need to remove all requests from the front of the queue that are outside the 3000 millisecond window (i.e., older than `t - 3000`).

By doing so, our queue always contains only those requests that are within the 3000 millisecond window. The length of the queue after this cleaning process gives us the number of recent requests, which is what the `ping` method returns.

To implement this idea in code, we utilize the `deque` data structure from Python's `collections` module because it allows for efficient addition and removal of elements from both ends.

The solution is implemented as a class, `RecentCounter`, containing 2 methods: the constructor `__init__` and the method `ping`.

- In the `__init__` method, we initialize a queue (specifically a deque) to store the timestamps of incoming requests. We start with an empty deque since there are no requests yet. A deque is used here because it supports fast appends and pops from both ends.
- The `ping` method is where the main logic is executed:
  1. First, we append the new request's timestamp `t` to the end of the deque with `self.q.append(t)`. This means we are adding the current request to our list of recent requests.
  2. Next, we enter a `while` loop, which will continue to execute as long as the front element of the deque (the oldest request) is less than `t - 3000`. This check is done with `self.q[0] < t - 3000`. Requests that fall outside the last 3000 milliseconds do not belong to the 'recent' category anymore and should be removed.
  3. For each loop iteration, if the condition is met (i.e., the request is too old), we remove the oldest request from the front of the deque with `self.q.popleft()`. Since requests are always timestamped in increasing order, once we find a request that is within the time frame, we can be sure that all subsequent requests are also within it, and there is no need to check the rest of the deque.
  4. Finally, once we have cleaned up all the old requests, the remaining length of the deque represents the count of recent requests. We return this count with `return len(self.q)`.

This implementation ensures that we're always working with a list of requests that have occurred within the last 3000 milliseconds. The time complexity for the `ping` method is O(N), where N is the number of calls to this method. However, because we are only keeping recent requests within the deque, this method is effective and efficient for a large number of requests, as long as the number of concurrent recent requests is reasonable.

### Example Walkthrough

Let's use the solution approach outlined above on a small example to illustrate how the `RecentCounter` class works in practice. For this example, assume we receive a sequence of timestamped requests at the following times (in milliseconds): 1, 100, 3001, 3002.

1. We create a `RecentCounter` object.
2. We call `ping(1)`. The deque now has one element: [1].
   - Since there are no pings older than 3000 milliseconds, we return `len(self.q)` which is 1.
3. Next, we call `ping(100)`. The deque becomes: [1, 100].
   - Both pings are within 3000 milliseconds from the current ping time, so the number of recent requests is 2.
4. Then we call `ping(3001)`. The deque is updated to: [1, 100, 3001].
   - Now, we remove timestamps older than `3001 - 3000` which is 1. After removal, the deque is [100, 3001].
   - The `len(self.q)` now returns 2, since these are the recent pings within the last 3000 milliseconds.
5. Lastly, we call `ping(3002)`. The deque first receives the timestamp 3002: [100, 3001, 3002].
   - We then remove any timestamps older than `3002 - 3000`, which is 2. After removal, the deque is [3001, 3002].
   - We return `len(self.q)`, which is 2, as the count of recent requests.

Throughout this process, the `RecentCounter` maintains a list of timestamps that are within the window of the last 3000 milliseconds. By adding new timestamps to the deque and removing the old ones, we ensure that we always have an accurate count of the recent requests when `ping` is called.

## Solution Implementation

### Python

```python
from collections import deque

class RecentCounter:
    def __init__(self):
        # Initialize a double-ended queue to keep track of ping times
        self.queue = deque()

    def ping(self, t: int) -> int:
        # Add the new ping time to the queue
        self.queue.append(t)

        # Remove ping times that are older than 3000ms from the current time
        while self.queue[0] < t - 3000:
            self.queue.popleft()

        # Return the number of pings within the last 3000ms
        return len(self.queue)

# Example usage:
# recent_counter = RecentCounter()
# count = recent_counter.ping(1)  # Assuming 't' is a timestamp in milliseconds
```

### Java

```java
import java.util.Deque;
import java.util.ArrayDeque;

class RecentCounter {
    // Queue to store the timestamps of the pings.
    private Deque<Integer> queue = new ArrayDeque<>();

    /**
     * Constructor initializes the RecentCounter with an empty queue.
     */
    public RecentCounter() {
    }

    /**
     * Records a new ping, and returns the number of pings in the last 3000 milliseconds.
     *
     * @param t The timestamp of the ping.
     * @return The count of pings in the last 3000 milliseconds.
     */
    public int ping(int t) {
        // Note: Timestamp here is assumed to be in milliseconds.

        // Offer/add the new ping's timestamp to the end of the queue.
        queue.offer(t);

        // Continue removing(polling) pings from the start of the queue if they are older
        // than 3000 milliseconds compared to the current ping's timestamp.
        while (!queue.isEmpty() && queue.peekFirst() < t - 3000) {
            queue.pollFirst();
        }

        // After removing old pings, return the size of the queue,
        // which is the count of recent pings.
        return queue.size();
    }
}

/**
 * The RecentCounter object will be instantiated and called as such:
 * RecentCounter obj = new RecentCounter();
 * int param_1 = obj.ping(t);
 */
```

### C++

```cpp
#include <deque>

class RecentCounter {
private:
    std::deque<int> pings;

public:
    // Constructor of RecentCounter initializes a new instance.
    RecentCounter() {
        // Nothing needed here since the deque 'pings' automatically initializes
    }

    // Method to record a new ping and return the count of pings in the last 3000 milliseconds.
    int ping(int t) {
        // Add the new ping timestamp to the deque
        pings.push_back(t);

        // Remove any pings that are no longer within the past 3000 milliseconds
        while (!pings.empty() && pings.front() < t - 3000) {
            pings.pop_front();
        }

        // Return the count of pings within the past 3000 milliseconds
        return pings.size();
    }
};

/*
 * Use of the RecentCounter class:
 *
 * RecentCounter* obj = new RecentCounter();
 * int param_1 = obj->ping(t);
 * delete obj; // Once done, remember to delete the object to free memory
 */
```

### TypeScript

```typescript
// Array to hold timestamps of recent pings.
let recentPings: number[] = [];

/**
 * Records a ping with the current timestamp and returns the number
 * of pings that occurred in the last 3000 milliseconds.
 * @param {number} t - The current timestamp in milliseconds
 * @returns {number} The number of pings in the last 3000 milliseconds
 */
function ping(t: number): number {
    // Record the new ping by adding the timestamp to the array.
    recentPings.push(t);

    // Remove pings from the array that are older than 3000 milliseconds from the current timestamp.
    while (recentPings.length > 0 && recentPings[0] < t - 3000) {
        recentPings.shift();
    }

    // Return the number of recent pings.
    return recentPings.length;
}
```

```python
from collections import deque

class RecentCounter:
    def __init__(self):
        # Initialize a double-ended queue to keep track of ping times
        self.queue = deque()

    def ping(self, t: int) -> int:
        # Add the new ping time to the queue
        self.queue.append(t)

        # Remove ping times that are older than 3000ms from the current time
        while self.queue[0] < t - 3000:
            self.queue.popleft()

        # Return the number of pings within the last 3000ms
        return len(self.queue)

# Example usage:
# recent_counter = RecentCounter()
# count = recent_counter.ping(1)  # Assuming 't' is a timestamp in milliseconds
```

## Time and Space Complexity

### Time Complexity

The time complexity of the `ping` method is O(N), where N is the number of calls to the method that fall within the 3000 ms window relative to the current call. This is because in the worst-case scenario, all N calls might have occurred within the past 3000 ms, which means the `while` loop will have to run N times to remove all outdated requests (requests older than 3000 ms before the current timestamp `t`). However, it is important to note that each individual call to ping is generally O(1) on average because the while loop doesn't always iterate over all elements. Over time, each element is added once and removed once, leading to an amortized time complexity of O(1).

### Space Complexity

The space complexity is O(N), where N is the number of all pings that are within the 3000 ms time window relative to the current call. This is due to the data structure (a deque) used to store the timestamps of pings. The space used by the deque will grow in direct proportion to the number of elements within the time window.