TABLE OF CONTENTS:

I. GENERAL PROJECT ARCHITECTURE:

i. Overview

The `Sensor Ping Processor` is a simple microservice designed to process pings from different sensors and track their frequency. The system provides two key operations:

1. Accepting a sensor ping.
2. Querying the sensor that had the most pings in the last `k` events.

ii. Components

1. **Controller Layer (`SensorController`)**:
    - Handles API requests for sending pings and fetching sensor frequency data.
    - Two main endpoints: `/ping` (POST) and `/top-frequency-sensor` (GET).
    - Validates input and returns responses to clients.
2. **Service Layer (`SensorService`)**:
    - Manages the core logic of the application.
    - Maintains a frequency map to track how many times each sensor has pinged.
    - It uses an event-indexed data structure to calculate the top-frequency sensor over the last `k` events efficiently.
    - Contains two main methods:
        - `processPing(int sensorID)`: Updates the frequency map and stores the latest ping.
        - `getTopKMaxFreqSensorID(int k)`: Retrieves the sensor with the highest frequency of pings from the last `k` events.

iii. Process Flow

1. **Receiving a Ping**:
    - The event size is incremented.
    - The frequency of the sensor is updated for the current event.
    - A copy of the frequency map is stored at the current event index in `sensorFrequencyIndexedMap`.
2. **Querying the Top-Frequency Sensor**:
    - If the total event size is less than `k`, it returns the sensor with the highest frequency for the total event range.
    - Otherwise, it compares the difference in frequencies, between the current event and the event `k` steps back to find the sensor with the highest frequency in that window.

---

iv. Improvements and Future Enhancements

- **Scaling**: For many sensors and events, consider a distributed caching solution like Redis to store sensor data with timestamps.
- **Database Integration**: Currently, the system is in-memory. Persisting events in a database (e.g., PostgreSQL, MongoDB) would make it more robust in production.We can also go with time series databases such as influxDB, timescale DB to store the data and query it more efficiently.
  For larger ranges (top sensor in the last million pings) we can store data in a data warehouse and use Map-Reduce asynchronously to keep track of such results.

---

vi. Conclusion

This architecture ensures a clean separation of responsibilities and allows for easy modifications. The current system tracks sensor pings and efficiently computes the top-frequency sensor over a sliding window. The data structure design allows for future scaling while ensuring minimal overhead for each new ping.

II. ALGORITHM AND DATA STRUCTURE DESIGN:

i. Approaches:

1.) **Brute Force traversal:** Store the pings in a doubly linked list maintaining the tail node and then traverse the structure till k every time getMaxFreqInKPings is hit, and output the sensorID with the max of frequency.

Time Complexities:
-> Add a ping -> O(1) (maintaining tail node in the list)
-> Query getMaxFreqInKPings -> O(k) k -> 1 -> n(total number of elements in the list) so worst case would be O(n)

So for m queries and n pings the overall get complexity would be -> O(m*n)

Space Complexities:
-> O(n) -> for storing the pings
So for m queries and n pings the overall complexity would be -> O(n)

2.) **Frequency Map of Map:** Store the frequency count at each ping in a map-of-map and then check for the frequency count difference between the latest ping entry and the kth

entry from the last in the map-of-map. Find the max of the difference and output the resultant sensorID.

Time Complexities:

-> Add a ping: O(t)

i. get the latest frequency counts -> O(1) [Map operation]

ii. Copy the previous frequency map -> O(t) [t -> number of unique sensors]

ii. update the frequency count of the 'sensorID' coming in and create a new frequency map entry for that ping in the map of map. -> O(1) [Map operation]

-> Query getMaxFreqInKPings: O(t)

i. Get the frequency counts of latest and kth entry from last in the map of map -> O(1)

ii. traverse the frequency counts of the latest and find the diff from the frequency counts of the kth entry from the last. Calculate and maintain the max of the diff and current maxFreq sensorID -> O(t) Where t is the number of unique sensors

So for 'm' queries and 'n' pings the overall get complexity would be -> O(m*t)

Space Complexities:

-> Add a ping: O(t) where t is the number of unique sensors

So for m queries and n pings the overall complexity would be -> O(n*t)

3.) Modified Segment tree:

We can create a segment tree with nodes as frequencyMap which will store the frequency of sensors. Every time a new ping arrives, we will use the latest pingList to form a segment tree and then query this segment tree for top k elements search.

Time Complexities:

-> Add a ping: O(n)

i. Add sensorID at the end of the pingList -> O(1)

ii. Build segment tree with the latest pingList -> O(n*t) [n -> number of pings in the ping list, t -> number of unique sensors]

And since every time a ping arrives, we need to rebuild the segment tree, the overall complexity would be O(n*n) [Worst case]

-> Query getMaxFreqInKPings: O(log(n))

So for 'm' queries and 'n' pings the overall complexity would be -> O(m*log(n))

Space Complexities:

-> Add a ping: O(n) where n is the number of pings. We can re-build a segment tree every time a new ping arrives.

So for m queries and n pings the overall complexity would be -> O(n)

Approach	**Time Complexity**	**Time Complexity**	**Space Complexity**
	Add a ping (each time)	**Query top K latest**	
Traverse list approach (brute force)	O(1)	O(n*m) n-> number of pings m -> number of queries	O(n) n-> number of pings
Map of Map approach at each ping	O(t) t-> number of unique sensors	O(m*t) m -> number of queries t -> number of unique sensors	O(n*t) n -> number of pings t -> number of unique sensors
Modified Segment Tree	O(n*t) n-> number of pings t -> number of unique sensors	O(m*log(n)) m-> number of queries n -> number of pings	O(n) n-> number of pings

```
NOTE: Count-min-sketch approach would not work here since it keeps the probabilistic track of the latest frequencies of the elements (sensors). Additionally, since it is an approximate data structure, it is not suitable for systems where exact accuracy is required.
```

ii. Assumptions:

The main assumption in the solution is with the growth of the n (number of pings), t (number of unique sensors) would fairly remain small or constant in comparison.

This would allow us to optimize on time complexity while maintain a fairly good space complexity.

iii. Proposed Approach (Final Implementation):

As per the time complexity and space complexity analysis, considering the assumption that t is small in comparison to n, **we have better time complexity in the map-of-map approach** and comparable space complexity with other approaches.

Also,

Additionally, the map-of-map approach gives us the liberty to find individual frequency of the sensors at any past or present point of time relatively easily.

Hence, I went forward with the map-of-map approach.

III. CODE DOCUMENTATION OF FINAL IMPLEMENTATION:

The core functionality of the `SensorService` revolves around:

1. **Tracking sensor pings** by maintaining a history of ping frequencies for each sensor across multiple events.
2. **Querying the most frequent sensor** in a window of the last `k` events.

To achieve this, the service uses a combination of hash maps to store and access the frequency of pings efficiently. This document explains the core data structures, algorithms, and provides a detailed analysis of time and space complexity.

1. `sensorFrequencyIndexedMap` (Map<Integer, Map<Integer, Integer>>):

- This is the main data structure used to track sensor frequencies over time.
- The **outer map** tracks the event size (i.e., the number of pings processed) and is keyed by the event index (e.g., 1, 2, 3, ...).
- The **inner map** tracks the frequency of each sensor for that specific event index, where the key is the sensor ID and the value is the frequency of pings for that sensor.

Example:

```
sensorFrequencyIndexedMap = {
  1: {1: 1},          // After event 1: Sensor 1 pinged once.
  2: {1: 2},          // After event 2: Sensor 1 pinged again.
  3: {1: 2, 2: 1}     // After event 3: Sensor 2 pinged once, Sensor 1
```

```
unchanged.
}
```

2. `eventSize` (int):

- A simple counter that keeps track of how many pings have been processed. It represents the current "event" or "time step" in the system.
- Example: If `eventSize = 3`, it means that 3 pings have been processed so far.

---

i. Core Code analysis:

1. `processPing(int sensorID)`:

- This function processes a ping for a given `sensorID` and updates the `sensorFrequencyIndexedMap` with the frequency of the sensor at the current event.

Steps:

1. Increment `eventSize` to indicate a new ping has been processed.
2. Retrieve the frequency map for the last event from `sensorFrequencyIndexedMap`. If it doesn't exist, create a new map.
3. Increment the frequency of the given `sensorID`.
4. Store the updated frequency map in `sensorFrequencyIndexedMap` at the new `eventSize`.

Time Complexity:

- **Accessing/Updating the inner map**: O(1) on average for a hash map.
- **Copying the previous event's map**: O(t) where `t` is the number of sensors in the previous event.

Thus, the overall time complexity for **processing a ping** is **O(t)**, where `t` is the number of sensors in the previous event.

Example:

If 3 sensors have been pinged so far, adding a new ping would involve copying the map from the previous event and updating it, resulting in O(3) operations.

---

2. `getTopKMaxFreqSensorID(int k)`:

- This function finds the sensor with the highest number of pings over the last `k` events.
- The solution depends on whether the total number of events (`eventSize`) is less than or greater than `k`:
    - **Case 1**: If `eventSize <= k`, it returns the sensor with the highest frequency in all events up to the current event.
    - **Case 2**: If `eventSize > k`, it computes the difference in frequencies between the current event and the event `k` events back, and finds the sensor with the largest frequency increase.

Steps:

1. **Case 1**: Extract the largest frequency sensor from the current event using `extractLargestFreqSensorID()`.
2. **Case 2**: Compute the frequency difference between the current event and the event `k` events back, using `extractLargestFreqSensorIDFromPingDiff()`.

Time Complexity:

- **Case 1**: Scans the frequency map for the latest event, which takes **O(t)** time where `t` is the number of sensors.
- **Case 2**: Scans two frequency maps (for the current and the `k`-th previous event), which also takes **O(t)**.

Thus, the time complexity for querying the top sensor is **O(t)** where `t` is the number of sensors at the current event.

---

3. `extractLargestFreqSensorID(int pingID)`:

- Finds the sensor with the highest frequency for a given event ID (`pingID`).

Steps:

1. Retrieve the sensor frequency map for the specified `pingID`.
2. Iterate over the map to find the sensor with the largest frequency.

Time Complexity:

- The time complexity is **O(t)**, where `t` is the number of sensors in the given event.

Example:

If the frequency map for event 3 is `{1: 2, 2: 1}`, the algorithm will iterate over these 2 entries and return sensor 1.

---

4. `extractLargestFreqSensorIDFromPingDiff(int pingID1, int pingID2)`:

- Compares the sensor frequencies between two events (`pingID1` and `pingID2`) and returns the sensor that has the largest frequency difference.

Steps:

1. Retrieve the sensor frequency maps for both `pingID1` and `pingID2`.
2. Iterate through the map of the most recent event (`pingID1`), calculating the difference in frequency for each sensor compared to the previous event (`pingID2`).
3. Track and return the sensor with the largest frequency difference.

Time Complexity:

- The time complexity is **O(t)** where `t` is the number of sensors in the current event (`pingID1`).

---

ii. Example Walkthrough

Scenario:

- Pings received in order: Sensor 1, Sensor 1, Sensor 2, Sensor 1, Sensor 2
- We want to query the sensor with the most pings in the last `k = 2` events.

Process Flow:

1. **Processing the pings**:

After processing 5 pings, `sensorFrequencyIndexedMap` will look like:

```
{
  1: {1: 1},
  2: {1: 2},
  3: {1: 2, 2: 1},
  4: {1: 3, 2: 1},
  5: {1: 3, 2: 2}
}
```

    -

2. **Querying the most frequent sensor for `k = 3`**:
    - We compare the maps for event 5 (`{1: 3, 2: 2}`) and event 3 (`{1: 2, 2: 1}`).
    - Differences:
        - Sensor 1: `3 - 2 = 1`
        - Sensor 2: `2 - 1 = 1`
    - Both sensors have the same frequency difference, so the algorithm can return either sensor 1 or 2.

---

iii. Overall Time and Space Complexity Analysis

1. Time Complexity:

- **Processing a ping**: O(t), where `t` is the number of sensors in the previous event.
- **Querying the most frequent sensor**: O(t), where `t` is the number of sensors in the current event.
- **Extracting the largest frequency sensor**: O(t), where `t` is the number of sensors in the event.

2. Space Complexity:

- The space complexity is **O(n * t)** where:
    - `n` is the number of events.
    - `t` is the maximum number of sensors processed in any event.
- Each event stores a map of sensor frequencies, and the outer map stores `m` entries (one for each event).

iv. Conclusion:

This algorithm provides efficient processing and querying of sensor pings using hash maps. The time complexity scales linearly with the number of sensors in the most recent event, and the space complexity grows with the number of events and sensors stored.