<https://leetcode.ca/2016-11-26-362-Design-Hit-Counter/>

Design a hit counter which counts the number of hits received in the **past 5 minutes** (i.e., the past 300 seconds).

Your system should accept a timestamp parameter (in seconds granularity), and you may assume that calls are being made to the system in chronological order (i.e., timestamp is monotonically increasing). Several hits may arrive roughly at the same time.

Implement the **HitCounter** class:

**HitCounter()** Initializes the object of the hit counter system.

**void hit(int timestamp)** Records a hit that happened at timestamp (in seconds). Several hits may happen at the same timestamp.

**int getHits(int timestamp)** Returns the number of hits in the past 5 minutes from timestamp (i.e., the past 300 seconds).

**Example 1:**

Input

["HitCounter", "hit", "hit", "hit", "getHits", "hit", "getHits", "getHits"]

[[], [1], [2], [3], [4], [300], [300], [301]]

Output

[null, null, null, null, 3, null, 4, 3]

**Explanation**

HitCounter hitCounter = new HitCounter();

hitCounter.hit(1); // hit at timestamp 1.

hitCounter.hit(2); // hit at timestamp 2.

hitCounter.hit(3); // hit at timestamp 3.

hitCounter.getHits(4); // get hits at timestamp 4, return 3.

hitCounter.hit(300); // hit at timestamp 300.

hitCounter.getHits(300); // get hits at timestamp 300, return 4.

hitCounter.getHits(301); // get hits at timestamp 301, return 3.

**Constraints:**

1 <= timestamp <= 2 \* 10^9

All the calls are being made to the system in chronological order (i.e., timestamp is monotonically increasing).

At most 300 calls will be made to hit and getHits.

**Follow up:**

What if the number of hits per second could be huge? Does your design scale?

**Attempt 1: 2024-12-05**

**Solution 1: Queue (10 min)**

**Algorithm**

Use a queue, add the current timestamp to the queue every time you click, and then when you need to get the number of clicks, we start from the beginning of the queue,

If the time stamp at the beginning is beyond 5 minutes, delete it until the time stamp at the beginning stops within 5 minutes

Then the number of elements returned to the queue is the requested number of clicks

class HitCounter {

private Queue<Integer> hits;

/\*\* Initialize your data structure here. \*/

public HitCounter() {

hits = new LinkedList<>();

}

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

public void hit(int timestamp) {

hits.add(timestamp);

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

public int getHits(int timestamp) {

// Remove hits older than 5 minutes (300 seconds)

while (!hits.isEmpty() && hits.peek() <= timestamp - 300) {

hits.poll();

}

return hits.size();

}

}

hit(int timestamp):

Time Complexity: O(1), as adding to the queue is constant time.

Space Complexity: O(1), additional space.

getHits(int timestamp):

Time Complexity: O(N), where N is the number of timestamps older than 300 seconds.

In practice, N is small because we only maintain recent hits.

Space Complexity: O(H), where H is the number of hits in the past 300 seconds.

**Follow Up**

**What if the number of hits per second could be huge? Does your design scale?**

**Refer to**

<https://leetcode.ca/2016-11-26-362-Design-Hit-Counter/>

**Follow-up question**

Two one-dimensional arrays times[] and hits[] with a size of 300 are defined, which are used to store timestamps and hits respectively.

In the click function, take the time stamp to 300, and then check whether the previously saved time stamp in this position is the same as the current time stamp. It also means that it is the same time stamp, then the corresponding number of clicks will increase by 1, if not the same, it means that five minutes have passed, then reset the corresponding number of clicks to 1.

Then when returning the number of hits, we need to traverse the times array to find all the positions within 5 minutes, and then add up the number of clicks in the corresponding positions in hits.

**Need further clarify:**

real-time getHits() ?

batch off-line process, then getHits() ?

1st thought is map-reduce, distribute to multiple hosts, so when getHits() need a broadcast to collect all data

Possible options:

use a lock

equal distribution each machine works independently to count its own users from the past minute. When we request the global number, we just need to add all counters together.

import

java.util.ArrayDeque;

public class Design\_Hit\_Counter {

public class HitCounter {

private ArrayDeque<Integer> queue; // @note: ArrayDeque has no capacity restrictions

// Why is ArrayDeque better than LinkedList?

// If you need add/remove of the both ends, ArrayDeque is significantly better than a linked list

// https://stackoverflow.com/questions/6163166/why-is-arraydeque-better-than-linkedlist

/\*\* Initialize your data structure here. \*/

public HitCounter() {

queue = new ArrayDeque<Integer>();

}

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

public void hit(int timestamp) {

queue.offer(timestamp);

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

// Time Complexity : O(n)

public int getHits(int timestamp) {

int startTime = timestamp - 300;

// remove all hits over 300 seconds old

while(!queue.isEmpty() && queue.peek() <= startTime) {

queue.poll();

}

return queue.size();

}

}

public class HitCounterFollowUp {

private int[] times;

private int[] hits;

/\*\* Initialize your data structure here. \*/

public HitCounterFollowUp() {

int[] times = new int[300];

int[] hits = new int[300];

}

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

public void hit(int timestamp) {

int idx = timestamp % 300;

if (times[idx] != timestamp) {

times[idx] = timestamp; // update with most recent timestamp

hits[idx] = 1;

} else {

++hits[idx];

}

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

// Time Complexity : O(n)

public int getHits(int timestamp) {

int res = 0;

for (int i = 0; i < 300; ++i) {

if (timestamp - times[i] < 300) { // values in times[] not ordered at all

res += hits[i];

}

}

return res;

}

}

/\*\*

\* Your HitCounter object will be instantiated and called as such:

\* HitCounter obj = new HitCounter();

\* obj.hit(timestamp);

\* int param\_2 = obj.getHits(timestamp);

\*/

}

############

class HitCounter {

private Map<Integer, Integer> counter;

/\*\* Initialize your data structure here. \*/

public HitCounter() {

counter = new HashMap<>();

}

/\*\*

Record a hit.

@param timestamp - The current timestamp (in seconds granularity).

\*/

public void hit(int timestamp) {

counter.put(timestamp, counter.getOrDefault(timestamp, 0) + 1);

}

/\*\*

Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity).

\*/

public int getHits(int timestamp) {

int hits = 0;

for (Map.Entry<Integer, Integer> entry : counter.entrySet()) {

if (entry.getKey() + 300 > timestamp) {

hits += entry.getValue();

}

}

return hits;

}

}

/\*\*

\* Your HitCounter object will be instantiated and called as such:

\* HitCounter obj = new HitCounter();

\* obj.hit(timestamp);

\* int param\_2 = obj.getHits(timestamp);

\*/

**Refer to chatGPT**

**What if the number of hits per second could be huge? Does your design scale?**

If the number of hits per second could be huge, the queue-based solution may not scale well, as it would require storing potentially millions of timestamps. Instead, we can use a **bucket-based design** to optimize for both time and space.

**Bucket-Based Design**

**Key Idea:**

Use an array of fixed size (300) to store the counts of hits in each second within the past 300 seconds.

Use modular arithmetic to map timestamps to the array indices, avoiding the need to store each individual timestamp.

class HitCounter {

private int[] times; // Array to store the timestamp for each bucket

private int[] hits; // Array to store the count of hits for each bucket

/\*\* Initialize your data structure here. \*/

public HitCounter() {

times = new int[300]; // 300 buckets for 300 seconds

hits = new int[300];

}

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

public void hit(int timestamp) {

int index = timestamp % 300; // Find the bucket for this timestamp

if (times[index] != timestamp) {

// Reset the bucket if it belongs to an older time,

// update with most recent timestamp

times[index] = timestamp;

hits[index] = 1;

} else {

// Increment the hit count for the current bucket

hits[index]++;

}

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

public int getHits(int timestamp) {

int totalHits = 0;

for (int i = 0; i < 300; i++) {

// Only consider buckets within the 300-second window

if (timestamp - times[i] < 300) {

totalHits += hits[i];

}

}

return totalHits;

}

}

**Explanation:**

Bucket Structure:

**Each bucket represents one second within the last 300 seconds.**

**times[i] stores the latest timestamp when this bucket was updated.**

**hits[i] stores the count of hits for that second.**

Resetting Old Buckets:

**If a bucket's timestamp (times[i]) is older than 300 seconds compared to the current timestamp, it is reset when accessed.**

Modular Indexing:

The array index is calculated using timestamp % 300, ensuring a rolling reuse of buckets.

**Complexity:**

**hit(int timestamp):**

Time Complexity: O(1), as it directly updates the corresponding bucket.

Space Complexity: O(300), for the arrays.

**getHits(int timestamp):**

Time Complexity: O(300), as it iterates through all buckets to calculate the total hits.

Space Complexity: O(300), where H is the number of hits in the past 300 seconds.

**Scalability:**

Handles High Hit Volume:

Instead of storing individual hits, it only stores a count for each second, drastically reducing space usage.

It scales well with a large number of hits per second.

Fixed Space:

The space requirement remains constant at O(300), regardless of the number of hits.

**Limitations:**

Precision:

This design assumes hits occur uniformly across seconds. If extremely precise hit data is required (e.g., per millisecond), this approach might not suffice.

Query Performance:

While O(300) is efficient for most cases, it might not be ideal for systems requiring sub-millisecond query times. Further optimization (e.g., aggregating counts) might be needed.

This bucket-based approach is a practical and scalable solution for the follow-up question.