# Daily Coding Problem #132

## Problem

This question was asked by Riot Games.

Design and implement a HitCounter class that keeps track of requests (or hits). It should support the following operations:

- `record(timestamp)`: records a hit that happened at `timestamp`
- `total()`: returns the total number of hits recorded
- `range(lower, upper)`: returns the number of hits that occurred between timestamps `lower` and `upper` (inclusive)

Follow-up: What if our system has limited memory?

## Solution

Let's first assume the timestamps in Unix time, that is, integers.

We can naively create a HitCounter class by simply using an unsorted list to store all the hits, and implement range by querying over each hit one by one:

```
class HitCounter:
    def __init__(self):
        self.hits = []

    def record(self, timestamp):
        self.hits.append(timestamp)

    def total(self):
        return len(self.hits)

    def range(self, lower, upper):
        count = 0
        for hit in self.hits:
            if lower <= hit <= upper:
                count += 1
        return count
```

Here, `record()` and `total()` would take constant time, but `range()` would take O(n) time.

One tradeoff we could make here is to use a sorted list or binary search tree to keep track of the hits. That way, `range()` would now take O(lg n) time, but so would `record()`.

We'll use python's `bisect` library to maintain sortedness:

```
import bisect

class HitCounter:
    def __init__(self):
        self.hits = []

    def record(self, timestamp):
        bisect.insort_left(self.hits, timestamp)
```

```
    def total(self):
        return len(self.hits)

    def range(self, lower, upper):
        left = bisect.bisect_left(self.hits, lower)
        right = bisect.bisect_right(self.hits, upper)
        return right - left
```

This will still take up a lot of space, though -- one element for each timestamp.

To address the follow-up question, we can make several possible trade-offs.

One possible trade-off would be to sacrifice accuracy for memory by grouping together timestamps in a coarser granularity, such as minute or even hours. That means we'll lose some accuracy around the borders but we'd be using up to a constant factor less space.

For our solution, we'll keep track of each group in a tuple where the first item is the timestamp in minutes and the second is the number of hits occurring within that minute. We'll sort the tuples by minute for O(lg n) record:

```
import bisect
from math import floor

class HitCounter:
    def __init__(self):
        self.counter = 0
        self.hits = [] # (timestamp in minutes, # of times)

    def record(self, timestamp):
        self.counter += 1

        minute = floor(timestamp / 60)
        i = bisect.bisect_left([hit[0] for hit in self.hits], minute)

        if i < len(self.hits) and self.hits[i][0] == minute:
            self.hits[i] = (minute, self.hits[i][1] + 1)
```

```
            else:
                self.hits.insert(i, (minute, 1))

    def total(self):
        return self.counter

    def range(self, lower, upper):
        lower_minute = floor(lower / 60)
        upper_minute = floor(upper / 60)
        lower_i = bisect.bisect_left([hit[0] for hit in self.hits],
lower_minute)
        upper_i = bisect.bisect_right([hit[0] for hit in self.hits],
upper_minute)

        return sum(self.hits[i][1] for i in range(lower_i, upper_i))
```

---