Daily Coding Problem #242

Problem

This problem was asked by Twitter.

You are given an array of length 24, where each element represents the number of new subscribers during the corresponding hour. Implement a data structure that efficiently supports the following:

- `update(hour: int, value: int)`: Increment the element at index `hour` by `value`.
- `query(start: int, end: int)`: Retrieve the number of subscribers that have signed up between `start` and `end` (inclusive).

You can assume that all values get cleared at the end of the day, and that you will not be asked for `start` and `end` values that wrap around midnight.

Solution

If we look beyond the details, the data structure required here is one that efficiently supports finding the sum of a subarray, and updating individual values in the array. One way of implementing this is by using a binary indexed tree, or Fenwick tree.

To see how this works, suppose the subscribers for an 8-hour range are as follows: `[4, 8, 1, 9, 3, 5, 5, 3]`, and we wanted to sum up number of subscribers from index 1 to index N. A naive solution would require us to go through each element and add it to a running total, which would be `O(N)`. Instead, if we knew in advance some of the subarray sums, we could break our problem apart into precomputed subproblems. In particular, we can use a technique that relies on binary indexing.

We will create a new array of the same size of our subscriber array, and store values in it as follows:

- If the index is odd, simply store the value of `subscribers[i]`.
- If the index is even, store the sum of a range of values up to `i` whose length is a power of two.

This is demonstrated in the diagram below, with `x` representing values of the original array, and dotted lines representing range sums.

```
[1, 2, 3, 4, 5, 6, 7, 8]
 x
 <-->
     x
 <-------->
         x
         <-->
             x
 <-------------------->
```

How does this help us? For any range between `0` and `N - 1`, we can break it apart into these binary ranges, in such a way that we only require `O(log N)` parts.

To make this more concrete, let's look again at our subscriber array, `[4, 8, 1, 9, 3, 5, 5, 3]`. For this array, the binary indexed tree would be `[0, 4, 12, 1, 22, 3, 8, 5, 38]`. As a result, we can calculate `query(0, 6)` using the following steps:

`query(0, 6) = query(0, 3) + query(4, 5) + query(6, 6) = tree[4] + tree[6] + tree[7] = 22 + 8 + 5 = 35`.

Note that if our start index is not `0`, we can transform our problem from `query(a, b)` to `query(0, b + 1) - query(0, a)`, so this is applicable for any range.

To find the indices of our tree to sum up, we can use a clever binary trick: keep decrementing the index by the lowest set of the current index, until the index gets to zero. We can implement this as follows:

```
def query(self, index):
    total = 0
    while index > 0:
```

```
        total += self.tree[index]
        index -= index & -index
    return total
```

Now let's take a look at the `update` operation. Changing the value of the 3rd item in the subscriber array from 1 to 2 would change the values of `tree[3]`, `tree[4]`, and `tree[8]`. Again, we can use the "lowest set bit" trick to increment the appropriate indices:

```
def update(self, index, value):
    while index < len(self.tree):
        self.tree[index] += value
        index += index & -index
```

Putting it all together, the code would look like this:

```
class BIT:
    def __init__(self, nums: int):
        self.tree = [0 for _ in range(len(nums) + 1)]
        for i, num in enumerate(nums):
            self.update(i + 1, num)

    def update(self, index: int, value: int):
        while index < len(self.tree):
            self.tree[index] += value
            index += index & -index

    def query(self, index: int):
        total = 0
        while index > 0:
            total += self.tree[index]
            index -= index & -index
        return total

class Subscribers:
    def __init__(self, nums: int):

        self.bit = BIT(nums)
        self.nums = nums
```

```
    def update(self, hour: int, value: int):
        self.bit.update(hour, value - self.nums[hour])
        self.nums[hour] = value

    def query(self, start: int, end: int):
        return self.bit.query(end + 1) - self.bit.query(start)
```

Because we have decomposed each operation into binary ranges, both `update` and `query` are `O(log N)`.
