Daily Coding Problem #283

Problem

This problem was asked by Google.

A regular number in mathematics is defined as one which evenly divides some power of 60. Equivalently, we can say that a regular number is one whose only prime divisors are 2, 3, and 5.

These numbers have had many applications, from helping ancient Babylonians keep time to tuning instruments according to the diatonic scale.

Given an integer N, write a program that returns, in order, the first N regular numbers.

Solution

A naive solution would be to first generate all powers of 2, 3, and 5 up to some stopping point, and then find every product we can obtain from multiplying one power from each group. We can then sort these products and take the first N to find our solution.

```
def regular_numbers(n):
    twos = [2 ** i for i in range(n)]
    threes = [3 ** i for i in range(n)]
    fives = [5 ** i for i in range(n)]
```

```
    solution = set()
    for two in twos:
        for three in threes:
            for five in fives:
                solution.add(two * three * five)

    return sorted(solution)[:n]
```

Since there are `N` integers in each prime group, our solution set will contain $O(N^3)$ numbers. As a result, the sorting process will take $O(N^3 \log N)$ time.

An alternative is to create a generative solution by using a heap. Note that for any regular number x, we can generate three additional regular numbers by calculating 2x, 3x, and 5x. Conversely, it must be the case that any regular number must be twice, three times, or five times some other regular number.

To implement this, we can initialize a min-heap starting with the value 1. Each time we pop a value x from the heap, we yield it, then push 2x, 3x, and 5x onto the heap. We can continue this process until we have yielded `N` integers.
One point to consider is that, for example, the number 6 will be pushed to the heap twice, once for being a multiple of two and once for being a multiple of three. To avoid yielding this value twice, we maintain a variable for the last number popped, and only process a value if it is greater than this variable.

```
import heapq

def regular_numbers(n):
    solution = [1]
    last = 0; count = 0

    while count < n:
        x = heapq.heappop(solution)
        if x > last:
            yield x
            last = x; count += 1
            heapq.heappush(solution, 2 * x)
            heapq.heappush(solution, 3 * x)
            heapq.heappush(solution, 5 * x)
```

Each pop and push operation will take $O(\log N)$ time. Since we will consider at most the

Each pop and push operation will take `O(log N)` time. Since we will consider at most the first `N` multiples of 2, 3, and 5, there will be `O(N)` of these operations, leading to an `O(N log N)` runtime.

Can we do better than this?

In fact, we can, by using an algorithm attributed to Dijkstra. As discussed above, each regular number must be a multiple of 2, 3, or 5. Suppose we store our solution in a sorted array. Then at each step in our program, we will keep track of three counters, `i2`, `i3`, and `i5`, to represent the smallest indices of the array such that multiplying 2, 3, or 5 by the corresponding index would create a previously unseen regular number.

To make this more clear, here is what the indices would look like as we generate the first few numbers.

```
        Solution         |  m2 |  m3 |  m5
-----------------------------------------
[1, 0, 0, 0, 0, 0, 0, 0] |  1  |  0  |  0
[1, 2, 0, 0, 0, 0, 0, 0] |  1  |  1  |  0
[1, 2, 3, 0, 0, 0, 0, 0] |  2  |  1  |  0
[1, 2, 3, 4, 5, 0, 0, 0] |  2  |  1  |  1
[1, 2, 3, 4, 5, 6, 0, 0] |  3  |  2  |  1
[1, 2, 3, 4, 5, 6, 8, 0] |  4  |  2  |  1
[1, 2, 3, 4, 5, 6, 8, 9] |  4  |  3  |  1
```

Let's look at the last row of the table above, where we have just added 9 to our solution. To generate the next regular number, we must either multiply 2 by `solution[4]`, or 3 by `solution[3]`, or 5 by `solution[1]`. The key idea here is that at any given point, there should only be three numbers to choose from to find the next number.

```
def regular_numbers(n):
    solution = [1] * n
    i2, i3, i5 = 0, 0, 0

    for i in range(1, n):
        m = min(2 * solution[i2], 3 * solution[i3], 5 * solution[i5])
        solution[i] = m

        if m % 2 == 0:
            i2 += 1
        if m % 3 == 0:
            i3 += 1
```

```
            if m % 5 == 0:
                i5 += 1

        return solution
```

For each value between one and `N`, we only need to perform a single `min` operation and increment up to three counters, so this algorithm will run in `O(N)` time and space.
