Daily Coding Problem #217

Problem

This problem was asked by Oracle.

We say a number is sparse if there are no adjacent ones in its binary representation. For example, `21` (10101) is sparse, but `22` (10110) is not. For a given input `N`, find the smallest sparse number greater than or equal to `N`.

Do this in faster than `O(N log N)` time.

Solution

One way to solve this would be to start with the input number and keep incrementing until we find a number that is sparse. We can do this with the help of a simple `is_sparse` function, which converts our input to binary and traverses the string looking for the substring `11`.

```
def is_sparse(num):
    return '11' not in bin(num)

def next_sparse_number(num):
    while not is_sparse(num):
        num += 1
    return num
```

Since the next sparse number can be `O(N)` away, and each call to `is_sparse` is $O(\log_{10} N)$, this will take `O(N log N)`.

Let's look for a bitwise solution instead. We know that any non-sparse number must contain the substring `011` in its binary representation. (If the binary starts with `11`, we can add a leading `0` without changing the value). So to get rid of these adjacent ones and produce a larger number, we can flip this bit pattern to `100` and turn the trailing digits to `0`.

There is one additional problem, though: what if, when we make this change, we accidentally create another `11` higher up in the bitstring? To efficiently deal with this, we can keep track of the highest bit that we've flipped, so the next time we only have to flip trailing digits until we reach that bit.

```
def next_sparse_number(num):
    # Convert to list of bits, from least to most significant.
    b = []
    while num:
        b.append(num & 1)
        num = num >> 1
    b.append(0)

    # Find the places where 011 exists, and turn them into 100 with trailing zeros.
    highest_zeroed_bit = 0
    for i in range(len(b) - 2):

        if b[i] and b[i + 1] and not b[i + 2]:
            b[i + 2] = 1
            for j in range(i + 1, highest_zeroed_bit - 1, -1):
                b[j] = 0
            highest_zeroed_bit = i + 1

    # Convert back to integer.
    num = 0
    for i in range(len(b)):
        num += b[i] * (1 << i)
    return num
```

This will take `O(log N)`, as we pass through the bits of `N` once, and reset each at most once.