Daily Coding Problem #249

Problem

This problem was asked by Salesforce.

Given an array of integers, find the maximum XOR of any two elements.

Solution

A naive solution would be to loop over each pair of integers and XOR them, keeping track of the maximum found so far. If there are N numbers, this would take $O(N^2)$ time.

We can improve on this by using a trie data structure. If we represent each integer as a k-bit binary number, we can insert it into a trie with its most significant bit at the top and each successive bit one level down. For example, 4, 6, and 7 could be represented in a three-level trie as follows:

```
      1
   0      1
0       0    1
```

Why would we want to do this? Well, once we have constructed such a trie, we can find the maximum XOR product for any given element by going down the trie and always trying to take the path with an opposite bit.

For example, suppose we wanted to find the maximum XOR for 2, which we will represent as `010`, using the trie above. We would use the following procedure:

- The first bit is `0`, so we look for a node on the top level with the value 1. Since

this exists, we make this our current node, and increment our XOR value by `1 << 2`.

- Next, since the second bit is `1`, we want to find a child node with the value `0`. Again, this exists, so we move down to `0`, and increment our XOR value by `1 << 1`.
- Finally, the last bit is `0`, so we look for a child node with a value of `1`. This does not exist, however, so we do not increment our count.

After traversing the trie, we would find the maximum XOR to be `1 << 2 + 1 << 1`, or 6.

These trie operations can be implemented as shown below:

```
class Trie:
    def __init__(self, k):
        self.trie = {}
        self.size = k

    def insert(self, item):
        trie = self.trie

        for i in range(self.size, -1, -1):
            bit = bool(item & (1 << i))
            if bit in trie:
                trie = trie[bit]
            else:
                trie = trie.setdefault(bit, {})

    def find_max_xor(self, item):
        trie = self.trie
        xor = 0

        for i in range(self.size, -1, -1):
            bit = bool(item & (1 << i))
            if (1 - bit) in trie:
                xor |= (1 << i)
                trie = trie[1 - bit]
            else:

                trie = trie[bit]
```

```
        return xor
```

Putting it all together, our solution is to first instantiate a `Trie`, using the maximum bit length to determine the size. Then, we insert the binary representation of each element into the trie. Finally, we loop over each integer to find the maximum XOR that can be generated, updating an XOR counter if the result is the greatest seen so far.

```
def find_max_xor(array):
    k = max(array).bit_length()
    trie = Trie(k)

    for i in array:
        trie.insert(i)

    xor = 0
    for i in array:
        xor = max(xor, trie.find_max_xor(i))

    return xor
```

The complexity of each `insert` and `find_max_xor` operation is `O(k)`, where `k` is the number of bits in the maximum element of the array. Since we must perform these operations for every element, this algorithm takes `O(N * k)` time overall. Similarly, because our trie holds `N` words of size `k`, this uses `O(N * k)` space.
