Daily Coding Problem #289

Problem

This problem was asked by Google.

The game of Nim is played as follows. Starting with three heaps, each containing a variable number of items, two players take turns removing one or more items from a single pile. The player who eventually is forced to take the last stone loses. For example, if the initial heap sizes are 3, 4, and 5, a game could be played as shown below:

```
  A  |  B  |  C
-----------------
  3  |  4  |  5
  3  |  1  |  3
  3  |  1  |  3
  0  |  1  |  3
  0  |  1  |  0
  0  |  0  |  0
```

In other words, to start, the first player takes three items from pile `B`. The second player responds by removing two stones from pile `C`. The game continues in this way until player one takes last stone and loses.

Given a list of non-zero starting values `[a, b, c]`, and assuming optimal play, determine whether the first player has a forced win.

Solution

Since we are dealing with a two-player game, one approach to solve this is to apply a minimax algorithm.

For the base case, we know that if the piles dwindle down to `(0, 0, 0)`, the current player to move is the winner, since the last player must have removed the final stone.

Now let's say you are faced with the heaps `(1, 3, 0)`. There are many possible moves, but the only good one is to remove everything in pile `B`, so that your opponent is forced to take the item in pile `A`. For any other move, there is a response that makes this a losing game. In other words, the value of a given move to player one is equivalent to the value of the best response to player two.

The list of possible moves can be generated by taking between one and all items from each pile. As a result, we can define a recursive solution that enumerates all possible moves, and returns `True` if any of them prevent the opponent from making an optimal move.

```
def update(heaps, pile, items):
    heaps = list(heaps)
    heaps[pile] -= items
    return tuple(heaps)

def get_moves(heaps):
    moves = []

    for pile, count in enumerate(heaps):
        for i in range(1, count + 1):
            moves.append(update(heaps, pile, i))


    return set(moves)

def nim(heaps):
    if heaps == (0, 0, 0):
        return True

    moves = get_moves(heaps)
```

```
    return any([not nim(move) for move in moves])
```

At the start of the game, if each pile has `a`, `b`, and `c` items respectively, there will be `a + b + c` possible moves, which we can denote by `N`. Unfortunately, because of our recursive approach, each subsequent move may only bring the number of items down by one, leading to a run time of `O(N!)`.

We can improve this with memoization. Each time we find the result for a given heap configuration, we store it in a dictionary. Before calculating the value of a given heap, we check this dictionary, and if the key exists we can avoid unnecessary work.

```
def nim(heaps, results={}):
    if tuple(heaps) in results:
        return results[tuple(heaps)]

    if heaps == (0, 0, 0):
        return 1

    moves = get_moves(heaps)
    results[tuple(heaps)] = max([1 - nim(move) for move in moves])

    return results[tuple(heaps)]
```

This solution will run in `O((a * b * c) * N)`, since there are `a * b * c` possible configurations of heaps to compute, and each one will take `O(N)` time to generate moves.

In fact, though, there is a bitwise solution to this game that is only `O(1)`!

Note that the losing state, `(0, 0, 0)`, has an xor product, or "nim-sum", of `0`. Though it is trickier it see, it is also the case that for any given state it is possible to make a move that

turns this product from zero to non-zero, and vice versa. Therefore, we can use the nim-sum after each pair of moves as an invariant.

More concretely, if you are playing Nim, and for a given move your opponent turns the nim-sum from `0` to `3`, you can make a move that turns it back to zero, putting your opponent back in a losing state. As a result, with the exception of one special case, a game is a win for the first player if and only if its nim-sum is nonzero.

```
def nim(heaps):
```

```
    a, b, c = heaps
    if a == b == c == 1:
        return False

    return a ^ b ^ c != 0
```