Daily Coding Problem #271

Problem

This problem was asked by Netflix.

Given a sorted list of integers of length `N`, determine if an element `x` is in the list without performing any multiplication, division, or bit-shift operations.

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

Solution

Ordinarily we would use binary search to locate this element. However, the standard implementation of binary search requires us to divide by two in order to find the middle element at each step.

An alternative search technique, called Fibonacci search, was created precisely to get around this limitation. The idea is that Fibonacci numbers are used to find indices to check in the array, and by cleverly updating these indices we can efficiently locate our element.

More specifically, suppose we are trying to find an element `x` in our array. Let `p` and `q` stand for consecutive Fibonacci numbers, where `q` is the smallest Fibonacci number greater than or equal to the size of the array. We compare `x` with `array[p]`, and perform the following logic:

- If `x == array[p]`, we have found our element, and are done.
- If `x < array[p]`, we move `p` and `q` down two indices each, cutting out the upper elements from our search.
- If `x > array[p]`, we move `p` and `q` down one index each, but add an offset of `p` to the next search value.

If we have exhausted our list of Fibonacci numbers, we can be assured that the element is not in our array.

Since this is a bit abstract, let's walk through an example. Suppose our array is `[4, 7, 11, 16, 27, 45, 55, 65, 80, 100]`. Since there are `10` elements in this array, we will need access to the sequence of Fibonacci numbers up to `13`.

We can find this sequence using the following helper function:

```
def get_fib_sequence(n):
    a, b = 0, 1
    sequence = [a]

    while a < n:
        a, b = b, a + b
        sequence.append(a)

    return sequence
```

The Fibonacci numbers we will obtain are `[0, 1, 1, 2, 3, 5, 8, 13]`, so we will initially set our values of `p` and `q` to be `6` and `7`, respectively.

Now suppose we are searching for the element `45`. We will carry out the following steps:

- First, we compare `45` with `array[fibs[p]]`. Since `45 < 80`, we set `p = 4` and `q = 5`.
- Next, we compare `45` with `array[fibs[p]]`. Since `45 > 16`, we set `p = 3` and `q = 4`, and create an offset of `2`.
- Finally, we compare `45` with `array[offset + fibs[p]]`. Since `array[5] == 45`, we have found our element.

```
def fibonacci(array, x):
```

```
    n = len(array)
    fibs = get_fib_sequence(n)

    offset = 0
    p, q = len(fibs) - 2, len(fibs) - 1

    while q > 0:
        index = min(offset + fibs[p], n - 1)
        if x == array[index]:
            return True
        elif x < array[index]:
            p -= 2; q -= 2
        else:
            p -= 1; q -= 1
            offset = index


    return False
```

At each step, this algorithm splits our array into two parts. Since the ratio between successive Fibonacci numbers is less than two-thirds, we can consider the larger part to be no greater than two-thirds the size of the original array. In the worst case, then, if our element always exists in the larger section, we will need to keep searching arrays two-thirds the size of the previous one, using constant operations each time.

Using the master theorem for recurrence relations, we can show this has `O(log N)` complexity as follows:

```
T(N) = T(2N/3) + 1
     = T(4N/9) + 1 + 1
    ...
     = T(1) + k, where (3/2)^k = N
     = T(1) + log(N) / log(3/2)
     = O(log N)
```