Project Euler Problem #51

Project Euler 51: Prime Digit Replacements

I'm a fan of Project Euler, the collection of math / programming problems. My goal is to eventually get to 100 problems solved – I'm at 72 right now, so here's another one, or at least my process behind solving it.

1.1 Problem Statement

> By replacing the 1st digit of the 2-digit number *3, it turns out that six of the nine possible values: 13, 23, 43, 53, 73, and 83, are all prime.
>
> By replacing the 3rd and 4th digits of 56**3 with the same digit, this 5-digit number is the first example having seven primes among the ten generated numbers, yielding the family: 56003, 56113, 56333, 56443, 56663, 56773, and 56993. Consequently 56003, being the first member of this family, is the smallest prime with this property.
>
> Find the smallest prime which, by replacing part of the number (not necessarily adjacent digits) with the same digit, is part of an eight prime value family.

1.2 Brute Force Solution

Suppose we have an n digit number, as in the first sentence in the problem statement. Then, we can choose any set of digits, and replace this set in the number with the digits from 1 through 9, yielding 9 new numbers. The brute force solution would take a look at each of the n digit numbers, and check if each is prime.

For an n digit number (of which there are $9 * 10^{n-1}$), we have 2^n subsets of digits we can choose; for each of these subsets, we have a constant 9 or 10 numbers in the family (depending on whether the first digit is included in the set), and we have to check all of these elements for primality. You can imagine this is not ideal, since these numbers grow very quickly. To check all $n = 7$ digit numbers, we would need to do 126 million primality checks. But we can make some improvements.

1.3 Improvements

First, note that we don't want to be checking two numbers in the same family. For the above example, we don't need to check both 56003 and 56113 – we only need to return the smallest member of the family. In an 8 prime value family, the repeated digit is either 0,1, or 2. This already reduces the amount of work we have to do by a factor of 3.

Furthermore, we can use some tricks about divisibility rules for numbers less than 10. How many you want to use is up to you, and of course there are diminishing returns, but here they all are (other than 7; look that one up, it's pretty convoluted and not really worth the effort in my opinion).

Number | Rule |
——— :| ——— :|
2 | Last digit is even |
3 | Digit sum is divisible by 3 |
4 | Last two digits is divisible by 4|
5 | Last digit is 0 or 5|
6 | Divisible by 2 and 3|
8 | Last three digits is divisible by 8|
9 | Digit sum divisible by 9 |
——— | ——— |

Suppose the sum of the digits (ignoring the ones that will be replaced) is d. Then if we replace only one digit, the digit sum of each member of the family will be: $d, d+1, d+2, d+3, ..., d+9$. What happens when we consider each of these modulo, for example 3? How many will give us 0 modulo 3, yielding a composite number?

We place each of the 10 generated digit sums and place it in box depending on its remainder modulo 3; two boxes have 3 elements and one box has 4 elements. By the pigeonhole principle, we have at most 7 prime members of the family. We can calculate the remainder of each of the elements of the family using Python:

```
# If we replace n digits of a number, return how many of the resulting 10 digit sums is 0 m
def digit_sums(n):
  dig_sum = list(map(lambda x: n*x % 3, range(10))) # Calculate the sum of n copies of each
  buckets = [0,0,0]
  for elt in dig_sum:
    buckets[elt] += 1
  return (buckets)

for i in range(1,7):
  print str(i) + ": " + str(digit_sums(i))
```

And our results are here:

```
1: [4, 3, 3]
2: [4, 3, 3]
3: [10, 0, 0]
4: [4, 3, 3]
5: [4, 3, 3]
6: [10, 0, 0]
```

So this tells us that the number of repeated digits is a multiple of 3; and this means that in order to get a composite number, we need the sum of the remaining digits to be something other than a multiple of 3.

It's very unlikely to have a four digit number satisfying our criteria. First, note that the last digit cannot be one that is replaced, since if that were the case we would end up with at least 5 even numbers in the family. Therefore, the first 3 digits are all replaced; since we cannot choose 0, we are left with the following choices:

```
111x, where x 1 or 7
```

It isn't hard to check (even by hand) that there are no results satisfying these criteria.

To sum up, our search is through the following candidates: * 5 or 6 digits, 3 of which are replaced * The final digit is not replaced * If the first digit is replaced, it is not 0 * The sum of the remaining digits is not divisible by 3

1.4 Writing Some Code

We first start with a Python function to generate for a given number and set of elements to be replaced, the number of primes in the family.

```
# Given a number, and an array of digit indices, generate the number of primes in its famil
def generate_family(n, arr):
  family = []
  digs = [x for x in str(n)] # Array of digits
  enum = range(0,10)
  if arr[0] == 0: # You can't end up with a smaller digit number!
    enum = range(1,10)
  # Loop which generates the family
  for i in enum:
    for index in arr:
      digs[index] = str(i);
    family.append(int("".join(digs)))
  # Functional programming! Going right to left: first, map each element to a boolean
```

```
  # value indicating whether it is prime; second, filter for the true elements; last, count
  # the size of the list
  return len(filter(lambda x: x, list(map(lambda x: is_prime(x), family))))
```

And finally, using this function as a black box, we generate a list of n digit candidates by running through all the possible $n-3$ digit substrings (excepting the ones which are mutliples of 3) and for each such substring and a subset of digits to be replaced, find the number of primes in the family. Our program returns the first prime family with 8 or more members.

```
def generate_candidates(n):
  # Calculates all n choose 3 subsets of elements to be replaced.
  subsets = list(itertools.combinations(range(0,n), 3))
  # Calculates all the n-3 digit numbers which are not divisible by 3
  rest = list(filter(lambda x: x%3 != 0, range(10**(n-3), 10**(n-2))))
  for elt in rest:
    digs = [int(x) for x in str(elt)] # Array of digits
    for subset in subsets:
      # Calculate an empty n digit number
      arr = [0] * n
      index = 0
      for num in range(0,n):
        if num in subset:
          arr[num] = 1
        else:
          arr[num] = digs[index]
          index += 1
      num = "".join(map(lambda x: str(x), arr))
      if generate_family(num, subset) > 7:
        return num

print generate_candidates(4) # None
print generate_candidates(5) # None
print generate_candidates(6) # 121313
```

And we are done! Note that there is actually a small inefficiency here: for each combination of the remaining digits, and for every array, we come up with a "dummy number".For example, if our remaining digits read 10, and the array consisted of $0, 1, 2$. Then, we create the dummy variable: 11110. The issue is that we are double counting some "dummy numbers" which can arise as a result of different input combinations of digits to be replaced and remaining digits; for example, when there are lots of 1s. I'm not sure how much more efficient we can be, though.