Project Euler: 1-100

Øistein Søvik

June 30, 2016
Contents

Project Euler 1: Multiples of 3 and 5 3
Project Euler 2: Even Fibonacci numbers 6
Project Euler 1: Multiples of 3 and 5

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

It is straightforward to iterate over the first 1000 numbers and check for divisibility.

```python
def divisible_by_3_or_5(limit = 1000):
    count = 0
    for num in xrange(1, limit):
        if num%3 == 0 or num%5 == 0:
            count += num
    return count
```

The code above takes on average 12 µs to run. Well under the arbitrary 1 s rule. However we can do better, a small improvement is to allow for different numbers than 3 or 5 and also allow for a customizable range.

```python
def numbers_divisible(divisors=[3, 5], start=0, stop=100):
    count = 0
    for num in xrange(start, stop):
        for d in divisors:
            if num % d == 0:
                count += num
                break
    return count
```

It could seem the above code is quite fast however running `numbers_divisible([3, 5], 0, 10**9)` takes about 40 s to run. This is painfully slow, but also expected since the code runs in \(O(n)\). As we shall see we can make the code run in constant time. The first idea is to find out how many numbers are divisible by 3. In total there are \(\lfloor 1000/3 \rfloor = 333\) numbers below 1000 which are multiples of 3. The sum of these can be found using the sum of the first \(n\) numbers

\[
\sum_{i=0}^{m} i = \frac{m(m+1)}{2}
\]

Using this we now have

\[
\sum_{i=0}^{333} 3i = 3 \left( \frac{333 \cdot (333 + 1)}{2} \right) = 166833 \quad (1)
\]

\[
\sum_{i=0}^{199} 5i = 5 \left( \frac{199 \cdot (199 + 1)}{2} \right) = 99500 \quad (2)
\]

However adding these two will give the wrong answer. That is because we are double counting some values? The numbers we are counting twice are all the numbers divisible by both by 3 and 5. For example 15 is counted in both sums. Therefore we have to subtract all the numbers divisible by both 5 and 3.

\[
\sum_{i=0}^{66} 15i = 15 \left( \frac{66 \cdot (66 + 1)}{2} \right) = 99500 \quad (3)
\]
Evaluating eq. (1) + (2) - (3). Before we try to code this I want to generalize this idea to find the sum of an arbitrary number $k$.

$$\sum_{i=\text{low}}^{\lfloor \text{limit}/k \rfloor} ki = \left( \sum_{i=0}^{\lfloor \text{limit}/k \rfloor} i - \sum_{i=0}^{\text{low}} i \right)$$

$$= \frac{k}{2} \ast (\text{stop} + \text{start}) \ast (\text{stop} - \text{start} + 1), \quad \text{stop} = \lfloor (\text{limit}/k) \rfloor$$

We can now write a function that finds the sum of all multiples of $k$ in a range.

```python
def sum_divisible_by_k(k, start, limit):
  stop = int((limit-1)/float(k))
  return k*(stop+start)*(stop-start+1)/2
```

```python
def divisible_by_3_or_5(divisors=[3, 5], start=0, stop=100):
  total = 0
  for divisor in divisors:
    total += sum_divisible_by_k(divisor, start, stop)
  product = divisors[0]*divisors[1]
  return total - sum_divisible_by_k(product, start, stop)
```

For values up to $10^5$ the improved version runs around 4500 times faster. However as stated this function runs in constant time so a speed comparison is not really necessary. We have now done the silly speed improvements. However can we rewrite the function to allow any number of inputs? The answer to this rhetorical question is yes. However it makes the code a bit more cluttered.

```python
from primefac import primefac
from itertools import combinations
from operator import mul

def sum_divisible_fast(divisors, start, stop):

  def prime_divisors_(divisors):
    proper_divisors = []
    for div in divisors:
      proper_divisors.extend(list(primefac(div)))
    return list(set(prime_divisors))

  proper_divisors = prime_divisors_(divisors)

  total = 0
  for divisor in prime_divisors:
    total += sum_divisible_by_k(divisor, start, stop)

  for i in range(2, len(prime_divisors)+1):
    k = (-1)**(i-1)
    for perm in combinations(prime_divisors, i):
      product = reduce(mul, list(perm))
      total += k*sum_divisible_by_k(product, start, stop)
  return int(total)
```

Let us try to break down what this code does. The first code takes in the divisors $[2, 7, 3, 6, 8, 9, 3]$ and returns a list of the unique primes in the list $[2, 3, 7]$. This is done by importing the `primefac` package. The reason why this is done is that checking numbers divisible by 6 is the same as being divisible by 2 and 3.
total = 0
for divisor in prime_divisors:
    total += sum_divisible_by_k(divisor, start, stop)

This part is the same as before, we are just adding the sum of the multiples of each divisor. The last part is somewhat tricky and is better understood through an example. Take \( [2, 7, 3] \), we first find the numbers multiplied by each factor. As before we have to remove the double counting. We are double counting all multiples of \( 2 \cdot 7 \), \( 7 \cdot 3 \) and \( 2 \cdot 3 \). Finding these pairs is done with the \textit{combinations} module. Sadly removing all of these multiplies takes away too many numbers. The numbers which are a multiple of \( 2 \cdot 7 \cdot 3 \) needs to be added back into the sum. For the general case we switch between adding and subtracting (this is done by the \( k = (-1)^{i-1} \)) until we have gone through the length of the prime divisors.

The running time of this algorithm is \( O(d^{k-1}) \) where \( d \) is the number of divisors. This is also the reason we take time in the beginning to make sure all the divisors are unique primes.
Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

This is one of my favourite problems and as a teaser the solution can be written as

```python
def sum_even_fibonacci(limit):
    a, b = 0, 2
    while b < limit:
        a, b = b, 4 * b + a
    return (a + b - 2) / 4
```

However to understand how this code works we have to go pretty far down the rabbit hole. A warning before we start: this problem is very easy to solve under 1s, and any improvements beyond this is purely for the amusement of the Author.