Project Euler with SIAM

at ASU WEST — lead by David Pratt, October 23, 2019

Introduction

Project Euler[1] is a member-run website that serves as a collection of recreational mathematical and computer programming problems that require skills in both mathematics and programming to be solved efficiently. *Efficiency* is at the core of every Project Euler problem, and we will see that it comes easiest when we apply mathematical thinking.

The goal of this series is to ignite a flame of fascination towards programming while simultaneously extinguishing any aversions to it. We will accomplish our goal by reasoning about these problems in a mathematical context before writing any code. Let's jump in.

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.*

A naive non-mathematical approach would check every number from 0 to 999 for divisibility by 3 or 5. This is a solution written in Python that uses this approach:

```
total = 0
for n in range(1000):
    if n % 3 == 0 or n % 5 == 0:
        total += n
print(total)
```

But we are mathematicians, and writing code should not be our first method of attack. We already know what numbers are multiples of 3 or 5, so there is no reason to test *any* number.

The sum of multiples of 3 less than 1000 is

$$s_3 = 3 + 6 + \cdots + 996 + 999.$$

Seeing the sum written out, you may be tempted to instead write

$$s_3 = 3(1 + 2 + \cdots + 332 + 333).$$

[1] www.projecteuler.net

This is a good temptation to give into, as you may recognize that the sum of numbers from 1 to n is easily computable. This leaves us with

$$s_3 = 3\left(\frac{333 \cdot 334}{2}\right).$$

Applying the same logic to the sum of multiples of 5 less than 1000, we see that

$$s_5 = 5\left(\frac{199 \cdot 200}{2}\right).$$

If we were to say the final answer is $s_3 + s_5$, we would be missing a key detail[2]. The multiples of 15 are present in both s_3 and s_5. Since we only want to include the multiples of 15 a single time, our final answer is $s_3 + s_5 - s_{15}$.

To program a solution, it would be in our best interest to define a function $s(N, m)$ that gives the sum of multiples of m less than N. We can translate our mathematical thinking above into Python code like this:

```
1  def s(N, m=1):
2          # calculates the sum of multiples of m less than N
3          N = N - 1  # we want numbers *less than* N
4          n = N // m
5          '''
6          s(1000, 3) = 3 * (1 + 2 + ... + 332 + 333) = 3 * (333*334) / 2
7             ^    ^    ^                               ^     ^       ^
8             N    m    m                               n     m    n*(n+1)
9          '''
10         return m * n * (n + 1) // 2
11
12 print(s(1000, 3) + s(1000, 5) - s(1000, 15))
```

Either way, we came to the same answer as the naive solution – so what's different? The difference is in the *time complexity* of the two solutions.

We say that the naive algorithm has time complexity $\mathcal{O}(n)$ where n is the upper bound to the multiples of 3 and 5 (in our case, $n = 1000$). Time complexity $\mathcal{O}(n)$ is often called *linear* time complexity because the runtime scales with n.

The second solution has time complexity $\mathcal{O}(1)$, or *constant* time complexity. This is to say that no matter what the upper bound is, we expect the algorithm to run in the same amount of time.

Using the magic of IPython[3] we can time each solution for a number of inputs. The runtimes are collected in Table 1. Notice how the runtime of the naive solution increases by a factor of 10 whenever n is increased by a factor of 10,

[2] Namely, the inclusion-exclusion principle.

[3] IPython is available in the Anaconda distribution of Python or via `pip install ipython`. Timings can be done easily using the `%timeit` magic function.

n	naive	optimal	$\frac{\text{naive}}{\text{optimal}}$
10^3	77μs	628ns	122.6
10^4	825μs	651ns	1267.3
10^5	8.04ms	708ns	11355.9
10^6	91.1ms	803ns	113449.6

Table 1: Runtime comparison

but the optimal solution remains at nearly[4] the same runtime regardless of how n changes. This is exactly what it means for an algorithm to run in *linear* time or *constant* time.

Even with this relatively simple example, we see the importance of crafting efficient algorithms. Project Euler demands that your algorithm be efficient, else your computer will be calculating the answer for 100 years instead of 100ms. If you find time complexity interesting, you may consider MAT 421 or ACO 201 to be fulfilling courses.

Project Euler #7

10,001st prime *By listing the first six prime numbers: 2, 3, 5, 7, 11, and 13, we can see that the 6th prime is 13. What is the 10,001st prime number?*

It looks like we are going to need to develop an algorithm that can determine the primality of a number. There are a myriad of algorithms[5], all of varying time complexity and *complexity*, but we are going to develop an algorithm that improves upon the naive *trial-division* method.

The trial-division method is exactly what you expect. To test the primality of an integer n, we test if any integer $1 < k < n$ divides n evenly. If such a k exists, n is not prime, else n is prime. Here is an implementation of trial-division in Python:

```
1  def isprime_td(n):
2      if n == 1:
3          return False
4
5      for i in range(2, n):
6          if n % i == 0:
7              return False
8      return True
```

Let's try to optimize the trial division algorithm.

[4]The reason why the time increases is because multiplication and division do not run in constant time (therefore, the optimal algorithm does not *technically* run in constant time).

[5]www.wikipedia.org/wiki/Primality_test

A simple observation is that we only need to find a prime factor of n to determine that n is not prime. Well, apart from 2, primes are always odd, so we just have to test division by 2 and subsequent odd numbers. In Python:

```
def isprime_odd(n):
    if n == 1:
        return False

    if n % 2 == 0 and n != 2:
        return False

    # test only odd numbers
    for i in range(3, n, 2):
        if n % i == 0:
            return False

    return True
```

This cut our computation time in half, but we can do better. If n has a prime factor $b \geq \sqrt{n}$, then n has a prime factor $a \leq \sqrt{n}$ such that $ab = n$ (prove it to yourself). This means that we don't have to test all odd numbers from 3 to n, but rather the odd numbers from 3 to $\sqrt{n}$. In Python:

```
def isprime_root(n):
    if n == 1:
        return False

    if n % 2 == 0 and n != 2:
        return False

    # test only odd numbers between 3 and sqrt(n)
    for i in range(3, 1+int(sqrt(n)), 2):
        if n % i == 0:
            return False

    return True
```

Let's take a step back so we can see the improvement from algorithm to algorithm. In the naive trial division algorithm, we test at most $n-2$ numbers. In the next algorithm, we test at most $\lfloor n/2 \rfloor$ numbers. And in the next algorithm where we use the square root optimization, we test at most $1+\lfloor \sqrt{n}/2 \rfloor$ numbers.

The progress is very apparent if we try to test $n = 1{,}000{,}003$, the closest prime to 1 million.

Algorithm	time for $n = 1{,}000{,}003$
naive	70.8 ms
odd	35.9 ms
root	34.8 μs

There is one more optimization we can make. Consider numbers of the form $6k$. Are these numbers prime? What about numbers of the form $6k+1$, $6k+2$, $6k+3$, $6k+4$, $6k+5$? In Python:

```
def isprime(n):
    if n == 1:
        return False

    if n % 2 == 0 and n != 2:
        return False
    if n % 3 == 0 and n != 3:
        return False

    for i in range(6, 2+int(sqrt(n)), 6):
        if n % (i-1) == 0 or n % (i+1) == 0:
            return False

    return True
```

This function is a few microseconds faster than the `isprime_root` function.

Now we can finally answer the question posed to us.

```
prime_count = 0
n = 1

while prime_count < 10001:
    n += 1
    if isprime(n):
        prime_count += 1

print(n)
```

Project Euler #10

Summation of primes *The sum of primes below 10 is* $2+3+5+7=17$. *Find the sum of all the primes below two million.*

Seeing as we just developed a primality test, we may be tempted to use that in an attempt to answer this question. Think about how much work is required to test every number less than two million. Can you recognize the fault in this method and develop an algorithm that does not have this flaw?