ProjectEuler Archive 1: Simple Summation

YarrowForAlgernon

August 2025

Contents

1 Challenge Description

Challenge Name: Multiples of 3 or 5

Challenge Description:

> *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.*

2 Naive solution

2.1 The First Thing That Comes To Mind

The first thought that comes to mind is to iterate over every integer between 1 and 1000 and test if they are a multiple of 3 or 5. If they are, we can add them to the sum of all previous multiples and then check the next number, until we reach 1000.

Here is some python code that calculates exactly this:

```
def sumOfAllMultiplesUnderN(multiple: int, N: int = 1000) -> int:
    sumOfMultiples = 0
    for integer in range(1, N):
        if (integer % multiple == 0):
            sumOfMultiples += integer
    return sumOfMultiples


if __name__ == "__main__":
    sumOf3 = sumOfAllMultiplesUnderN(3)
    sumOf5 = sumOfAllMultiplesUnderN(5)
    finalSum = (sumOf3 + sumOf5)
    print(finalSum)
```

There is a problem with this implementation, however, and that is the number '15'. 15 is divisible by both 3 and 5, and thus all subsequent multiples of 15 have this property.

If we run the code as shown above, we will get a result that will be higher than in reality, because all occurrences of multiples of 15 have been counted twice, once in all multiples of 3 and again in all multiples of 5.

We can correct this error using the Principle of Inclusion and Exclusion (PIE). Since we have counted all multiples of 15 twice, if we subtract from the 'finalSum' variable in the code above the sum of all multiples of 15 below 1000, then it will be as if we had only counted all multiples of 15 once.

We can also add some additional code to measure the runtime of the algorithm in nanoseconds:

2.2 The Algorithm

```
from time import time_ns

```

```
def sumOfAllMultiplesUnderN(multiple: int, N: int = 1000) -> int:
    sumOfMultiples = 0
    for integer in range(1, N):
        if (integer % multiple == 0):
            sumOfMultiples += integer
    return sumOfMultiples


if __name__ == "__main__":
    startTime = time_ns()
    sumOf3 = sumOfAllMultiplesUnderN(3)
    sumOf5 = sumOfAllMultiplesUnderN(5)
    sumOf15 = sumOfAllMultiplesUnderN(15)
    finalSum = (sumOf3 + sumOf5) - sumOf15 # application of PIE
    stopTime = time_ns()
    print(finalSum)
    print("Time taken in nanoseconds: " + str(stopTime - startTime))
```

Note: we only measure the runtime of the algorithm and not the runtime of the print statements, since they incur unnecessary overhead and aren't part of the algorithm.

The big O notation of this algorithm is O(n) since we naively iterate over n numbers. The average runtime of this algorithm on my 10th gen i-3 was 171642 nanoseconds or roughly 0.17 milliseconds.

3 Gauss' Trick

3.1 Explanation Of Maths

There exists a far more efficient solution than having to sum every multiple between 1 and 1000.

The solution is to turn the whole process into a modified version of Gauss' trick. Gauss' trick is a way to very quickly compute summations, through the equation:

Theorem 3.1 (Gauss Summation).

$$\sum_{i=1}^{n} i = \frac{n}{2} * (n+1) \tag{3.1}$$

Where n is the upper bound of the summation.

This formula is possible as every pair of the highest of and lowest number, then the next highest and next lowest, etc will all add up to $n + 1$. For example, the summation of every integer from 1 to 100, (1 + 2 + 3 ...), can be paired as follows:

(100 + 1) + (99 + 2) + (98 + 3) + ...

Since every pair is equivalent to 101, AKA the term $n + 1$ in Theorem 3.1, if we multiply 101 by how many pairs there are, we will come up with the total sum.

The total number of pairs that can be formed is half of n, AKA $n/2$ in Theorem 3.1. Thus, by multiplying these two values together (number of pairs * value of each pair) we get Theorem 3.1.

However, our summation counts in multiples of 3 and 5. This isn't the neat form of $1 + 2 + 3 + \dots$ but instead $3 + 5 + 6 + 9 + ...$

In problem-solving, when you encounter a complex problem, you usually reduce it to a simpler problem and solve that first. In this case, let's only consider multiples of 3:

```
3 + 6 + 9 + ...

Then, let's only consider multiples of 5:

5 + 10 + ...

And if we add them together:

3 + 5 + 6 + 9 + 10 + ...

It's our original summation!
```

Now, we just have to put each of these individual summations in the form of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, which is quite easy. Since each of the above summations count in multiples, we can simply factor out those multiples. So our final expression to calculate the answer should look like this:

$$3\sum_{i=1}^{k} i + 5\sum_{i=1}^{k} i - 15\sum_{i=1}^{k} i \tag{3.2}$$

Where k is the quotient of n divided by the constant multiplying the summations.

We have to take into account that n isn't our original upper bound of 999 (first integer below 1000) so we replace it with k.

So our final general equation to calculate the summation of any multiple is:

Corollary 3.1.1.

$$m\sum_{i=1}^{k} i = m\frac{k(k+1)}{2} \tag{3.3}$$

Where m is our multiple of choice and k is the quotient of n divided by m.

We can now use Corollary 3.1.1 to make an algorithm that can calculate the solution using Gauss Summation.

3.2 The Algorithm

```
from time import time_ns

def SumOfMultiplesUnderN(multiple: int, N: int=999) -> float:
    quotient = N // multiple
    sumOfMultiples = multiple * (0.5 * quotient * (quotient+1))
    return sumOfMultiples

if __name__ == "__main__":
    startTime = time_ns()
    finalSum = SumOfMultiplesUnderN(3) + SumOfMultiplesUnderN(5) -
    SumOfMultiplesUnderN(15)
    stopTime = time_ns()
    print(finalSum)
    print("Time taken in nano seconds: " + str(stopTime - startTime))
```

The big O notation for this algorithm is O(1), since the number of calculations for each summation don't scale with the size of our input. The average runtime of this algorithm on my 10th gen i-3 is 6462 nanoseconds, or 0.006 milliseconds, over 20x faster than our original algorithm.