A Foray into Rust: Euler One

R (Chandra) Chandrasekhar

2021-07-31 | 2023-11-25

As a programmer, I am long in the tooth. I started out with `FORTRAN`, went on to `Forth`, and settled with `C` through three decades or more. Later, it was `MATLAB` and `Octave` for high level computing. For scripting, I used `Perl` or `bash`. `Python`, the current darling of programmers, is an unknown bourne to me.

So why did I choose `Rust` as the new programming language to learn? Rust is *the* emerging programming language, developed at Mozilla [1]. It has been consistently voted the most loved programming language in Stack Overflow Developer Surveys [2]. End-users, such as scientists, are turning to Rust when Python has proven inadequate for some reason [3]. And corporate users include Dropbox, Mozilla, Microsoft, npm, etc. [4].

But there are other, more personal, reasons as well. My previous bet on the future was on `Haskell`. I have tried many times to learn it, almost always giving up in despair, because I was put off by the unfamiliar notation, and its corpus of arcana, like monads, touted by the cognoscenti, as the way to tell the men from the boys. Enough about the why. Now for the how.

To learn Rust, I decided to start by solving Project Euler Problem One—henceforth called *Euler One*, *the problem*, or *the question*—using Rust. This is a chronicle of my first efforts, including false starts, errors, backtracks, etc.

Project Euler Problem One

The statement of the problem is simple and pellucid:

> **Multiples of 3 or 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. [*Emphasis* is mine]

Algorithm for problem solving

The algorithm for problem solving is [5]:

1. Read the question carefully.
2. Understand the question correctly.
3. Answer the question precisely.

The problem asks for *all* the multiples of 3 *or* 5 *below* 1000. I have *emphasized* the words that require careful understanding and thought. Care at this stage of acquaintance with the problem staves off many a careless mistake by nipping it in the bud.

Parsing the question

The word "or"

I have emphasized three words in the problem definition: *all*, *or*, and *below*. The first is obvious. Let us look at the other two.

The phrase "multiples of 3 *or* 5" may be interpreted in two ways. If we think of it as an *inclusive or*, then it means "multiples of $3, multiples of 5, and multiples of both 3 and 5".

If we think of it as an *exclusive or*, then it means "multiples of $3, multiples of 5, but not multiples of both 3 and 5".

Sine the qualification of "but not both" is absent from the rubric, we will assume an inclusive or, i.e., the first interpretation.

The word "below"

The word "below" introduces the mathematical relation $<$ as opposed to $\leq$. This means all multiples of three or five that are less than 1000, but excluding 1000.

The time spent in looking at the question through a magnifying glass is time well spent, because it forces us to assume the mind set of the author who carefully drafted the question. We thereby become acquainted with the possibilities for pitfalls and potholes that could otherwise upend our efforts.

Initial thoughts

The multiples of 3 are those numbers, which when divided by 3, leave a remainder of zero. Likewise the numbers which leave a remainder of zero when divided by 5 are multiples of 5. This implies *integer arithmetic*, and that in turn, could mean we have to *declare* the type of numbers we are using. Floating point division will never do for our problem. But anyway, division is problematic; witness the caveat that the divisor may not be zero in the field of integers, $\mathbb{Z}$.

In terms of division, the `%` operator from other programming languages suggests itself. But is division the most natural way to identify the multiples of a number? Should it not be multiplication instead? It is time to start thinking with a beginner's mind.

We also need a structure like an *array* or *list* where numbers may be appended or inserted until the stopping condition is reached. If we keep a running total, though, we do not need anything else except three receptacles: one for the sum of multiples of three, s_3, another for the sum of multiples of five, s_5, and one more for the sum of multiples of 15, s_{15}. Let us try the latter option first, and leave arrays for a later refinement.

Setting the bounds

We know that $1000 \div 3 = 333$ with a remainder of 1. The largest multiple of 3 less than 1000 is therefore, $333 \times 3 = 999$. The number of multiples of 3, n_3, we will be dealing with is thus 333.

Likewise, $1000 \div 5 = 200$ with a remainder of 0. Since $1,000$ is a multiple of 5, we need the *next lower* multiple of 5 below 1000. That number is $199 \times 5 = 995$, and so, $n_5 = 199$.

With 15, we have $1000 \div 15 = 66$ with a remainder of 10. So, $66 \times 15 = 990$ is the upper bound, and the number of multiples n_{15} is 66.

Because 15 is a multiple of *both* 3 and 5, we need to ensure that we do not add its multiples *twice* in our summations.

Venn diagram representation

I find that viewing a problem pictorially often helps us to grasp it better. In this case, it is not a graph but a Venn diagram that helps. In Figure 1, we use circles A and B to represent multiples of 3 and 5 respectively. The two circles overlap because there exist numbers that are multiples of both 3 and 5: these are the multiples of 15.

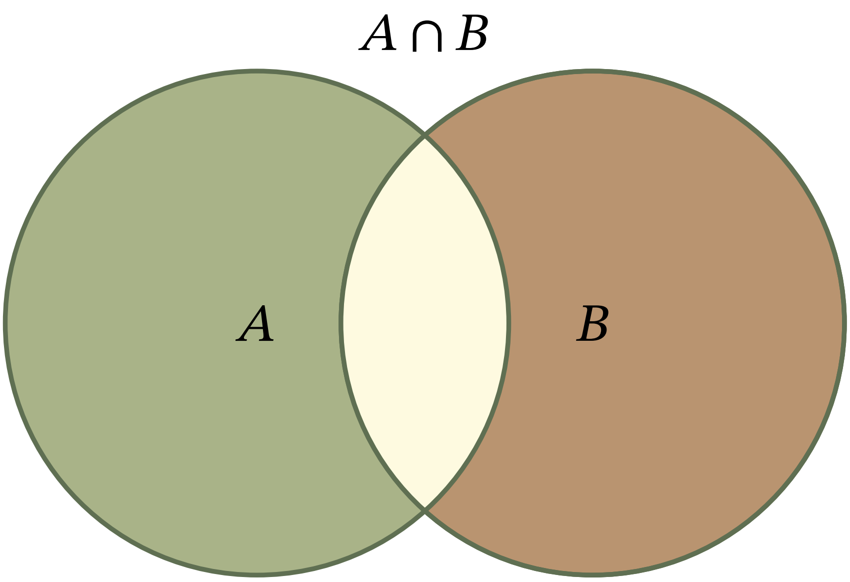

Figure 1: Venn diagram relating multiples of 3, shown as set A, multiples of 5 as set B, and multiples of 15 as their intersection $A \cap B$.

We know from set theory that what we are after is $A \cup B$ or the union of the sets A and B. Also, the number of elements in the sets are related by

$$n(A \cup B) = n(A) + n(B) - n(A \cap B). \tag{1}$$

The expression $n(A)$, for example, denotes the number of (unique) elements in the set A. Equation (1) gives us a convenient way of counting the multiples of 3 or 5, *without double counting the multiples of* 15.

Algorithm and Pseudocode

The most direct algorithm to solve the problem in pseudocode is:

1. Define s_3 as the cumulative sum of the multiples of 3, and initialize it to 0.
2. Define s_5 as the cumulative sum of the multiples of 5, and initialize it to 0.
3. Define s_{15} as the cumulative sum of multiples of 15, and initialize it to 0.
4. Loop through the natural numbers $\mathbb{N}$ from 1 to 333, compute the multiples of 3, one at a time, and add it to s_3.
5. Loop through the natural numbers $\mathbb{N}$ from 1 to 199, compute the multiples of 5, one at a time, and add it to s_5.
6. Loop through the natural numbers $\mathbb{N}$ from 1 to 66, compute the multiples of 15, one at a time, and add it to s_{15}.
7. Evaluate $(s_3 + s_5 - s_{15})$ and present it as the desired result. See Equation (1) for an explanation.

I envisage three independent `for` loops to achieve this. The pseudocode could read:

```
s = s3 = s5 = 0 # initialize variables

for n in [1:333]
do
  s3 = s3 + 3*n
done

for n i [1:199]
do
  s5 = s5 + 5*n
done

for n i [1:66]
do
  s15 = s15 + 15*n
done

print (s3 + s5 -s15)
```

We have implicitly assumed that the `for` loop increment is 1. The mathematics convention for a closed interval is used above to denote that *both* the upper and lower limits are *inclusive*.

First attempt

Let us set to using the syntax of `Rust` and see how the above pseudo code fleshes out. It turns out that `Rust` supports five types of loop and we need the one with the `for` flavour, called the iterator loop.

There is also an example on that web page that is similar to our problem. It uses a `for` loop, but the variable holding the sum is initialized using the `mut` keyword. Let us copy the code fragment and change it to suit our purposes:

```
// Attempt Number 1
  let mut s3 = 0;
  let mut s5 = 0;
  let mut s15 = 0;

  for n in 1..333 {
    s3 += n*3;
  }

  for n in 1..199 {
    s5 += n*5;
  }

  for n in 1..66 {
    s15 += n*15;
  }

  println (s3 + s5 - s15);
```

The above fragment contains numerous errors. So, I needed to backtrack to see an example of the archetypal “Hello World!” program to get the proper invocatory syntax. Languages like `C` and `Java` come with some baggage that needs to be wrapped around the functional code so that it may be rendered into an executable program. `Rust` seems to have borrowed this characteristic from them. Note the use of `s3 += n*3;` which is shorthand for `s3 = s3 + n*3`. The `+=` operator is available in `Rust` but not always in other languages.

Second attempt

My second attempt at the program is now:

```
 // Attempt Number 2
fn main() {
  let mut s3 = 0;
  let mut s5 = 0;
  let mut s15 = 0;

  for n in 1..333 {
    s3 += n*3;
  }

  for n in 1..199 {
```

```
    s5 += n*5;
  }

  for n in 1..66 {
    s15 += n*15;
  }
  println!("{}", s3 + s5 - s15);
  }
```

I have wrapped the whole code fragment with a `main()` function just as in `C`. Moreover, I have learned that `println!` is a macro rather than a function and that it is invoked as shown. This has already disheartened me a bit because something too much like `C` or `Java`—with a lot of clunky statements for simple actions—is a step in the *wrong* direction for an easier-to-use programming language. Let us hope it does not rain pickaxes and shovels when we run the code!

This time, the code was compiled without a murmur. Upon execution, the answer was 232164. Is it correct? Or have we tripped somewhere?

Result with `Octave`

The easiest and laziest way to check the result was to use a naturally vector-based language to verify the above result. I chose `Octave` as it is freely available. Because the natural data structure in `Octave` is a vector or a matrix,, I could type out the whole sequence using the syntax `[start:step:end]` and sum it up to get the sum of multiples. The code was so easy, I could write it without reference to paper:

```
sum([3:3:999]) + sum([5:5:995]) - sum([15:15:990])
```

and this gave a result of 233168. Ouch! it differs from the result using `Rust`. I must also say that, though laconic, `Octave` got the job done with very little fuss or fanfare. Vectorized code is both more powerful and simpler to understand and maintain. The best language for one working with vectors is to use a language that supports them natively.

We must now make third, "repair and maintenance" attempt at the `Rust` code.

Troubleshooting

The `rust` program is so simple that the most likely error must lie with the limits in the `for` loop. Indeed, an experienced programmer would have seen it at once.

Programming languages are notoriously inconsistent on two fronts:

a. Whether they start their indexing with 0 or with 1; and
b. Whether their index ranges are on closed intervals $[a, b]$, or semi-closed intervals $[a, b)$, or $(a, b]$, or open intervals (a, b).

One would have thought that common sense would impel language designers to adopt uniform conventions on these two issues. Unfortunately the authors of programming languages have rather fiercely held philosophical notions, and a divide persists. Thus each foray into a new language must be cautiously done with these two factors in mind. In our case, we need to hark back to the definition of the `..` range operator in `Rust`. The expression `start..end` means that the index variable `i` lies in a semi-closed interval: `start <= i < end`.The `end` parameters in each case need to be increased by one. Our third attempt is shown below:

Third attempt

```
 // Attempt Number 3
fn main() {
  let mut s3 = 0;
  let mut s5 = 0;
  let mut s15 = 0;

  for n in 1..334 {
    s3 += n*3;
  }

  for n in 1..200 {
    s5 += n*5;
  }

  for n in 1..67 {
    s15 += n*15;
  }
  println!("{}", s3 + s5 - s15);
  }
```

This again complied incident-free and the result that popped out was 233168. Bingo! It is the same as what `Octave` gave us. That is a reassuring feeling. The real arbiter of truth, though, is mathematics. What does it say?

The Gold Standard

We are fortunate that in this case, the mathematics is both simple and well known. We are dealing with sums of three arithmetic progressions (AP). The *first term* in an AP is usually denoted a and the *common difference* is denoted by d. The number of terms is usually n. The *last term* is $a_n = a + (n-1)d$, and the sum to n terms is

$$a + a + d + a + 2d + a + 3d + \cdots + a + (n-1)d = \frac{n}{2}(a + a_n) \tag{2}$$

Using this formula, for the multiples of 3, we have $a = 3$, $n = 333$ and $a_n = 999$, giving us

$$s_3 = \frac{333}{2}(3 + 999) = 166833.$$

Likewise, for the multiples of 5, we have $a = 5$, $n = 199$ and $a_n = 995$, yielding

$$s_5 = \frac{195}{2}(3 + 995) = 99500.$$

The required sum, s is therefore $s = s_3 + s_5 = 166833 + 99500 = 266333$. So, the result from `Octave` is correct. Now where did we go wrong in our third attempt with `Rust`?

Fourth attempt

```
 // Fourth attempt
fn main() {
  let mut s3 = 0;
  let mut s5 = 0;

  for n in 1..334 {
    s3 += n*3;
  }

  for n in 1..200 {
    s5 += n*5;
  }

  let s =  s3 + s5;
  println!("{}", s);
  }
```

The result from this last attempt is indeed 266333 and is in accord with both the AP sum formula and the vector sum from `Octave`. We should now hang up our boots and retire. But I want to squeeze a little more juice from the problem.

Vectorizing

The single-line `Octave` program made the solution seem laughably easy. Why? Because the standard data structure in `Octave` is a vector or a matrix. In the context pf `Rust`, we may pose these questions:

1. Does `Rust` have a ready implementation of vectors that may be called upon?
2. Would such an implementation be faster? Less or more error prone?

I had a little peep at the possibilities with `Rust` and realized that being a multi-paradigm language, `Rust` provides many possibilities to accomplish the same task. Moreover, once the simplicity of scalars is left behind with vectors, the knowledge curve is rather steep. So, vectorizing must promise returns commensurate with the learning effort. That was not apparent to me; hence I did not travel that way.

FizzBuzz

The FizzBuzz coding problem is a natural successor to EulerOne. We have already computed the multiples of 3 and the multiples of 5 which are less than 1000. But we did not retain them as separate entities from which we could draw out the multiples of 15 as well. The problem is stated below.

> For any integer from 1 to n, print *Fizz* if it is divisible by 3, *Buzz* if it is divisible by 5, and *FizzBuzz* if it is divisible by 15. [For our purposes, we may set an upper limit as $n < 1000$.]

This is a favourite coding interview problem because it is simple enough to reveal the thought processes behind the programming and the approach to the problem that results from the thought process.

Vectors and set intersections are the easiest way to achieve this, but `Rust` presents a steep climb in knowledge acquisition before even meagre results start trickling in.

Octave implementation of FizzBuzz

In `Octave`, the implementation of `FizzBuzz` is starkly simple. The availability of the set difference as an operation gives us a ready-made solution as shown below. Of course, I have not printed the output, but the vectors named `fizz`, `buzz` and `fizzbuzz` contain the numbers whose elements are associated with these responses. They are not displayed here in good taste.

```
% FizzBuzz
threes = [3:3:999];
fives = [5:5:995];
fifteens = [15:15:990];
fizz = setdiff (threes, fifteens);
buzz = setdiff (fives, fifteens);
fizzbuzz = fifteens;
```

Closing thoughts

`Rust` is not for me. Of that, I am certain. Euler One is suited to vectorization where `Octave` beats `Rust` hands down in simplicity and succinctness. Not all problems are amenable to simplification in `Octave` and I am sure that `Rust` has its niches of exemplary use. The little that I have seen of `Rust`, however, has made me realize that it is not suited to my temperament. Give me `Octave` any day!

https://iambryanhaney.medium.com/another-unreasonable-deep-dive-into-project-euler-problem-1-51a3a841ad67#:~:text=The%20Problem,0%20modulo%203%20or%205

Feedback

Please email me your comments and corrections.

A PDF version of this article is available for download here.

References

[1] —. Rust. Retrieved 1 August 2021 from https://research.mozilla.org/rust/

[2] Jake Goulding. 2020. What is Rust and why is it so popular? Retrieved 1 August 2021 from https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/

[3] Jeffrey M Perkel. 2020. Why scientists are turning to Rust. Despite having a steep learning curve, the programming language offers speed and safety. Retrieved 1 August 2021 from https://www.nature.com/articles/d41586-020-03382-2

[4] Gints Dreimanis. 2020. 9 Companies That Use Rust in Production. Retrieved 1 August 2021 from https://serokell.io/blog/rust-companies

[5] R (Chandra) Chandrasekhar. 2023. Secrets of Academic Success. Timeless Principles for Lifelong Learning. Retrieved 24 November 2023 from https://swanlotus.netlify.app/sas-manuscript/SAS-partial.pdf

