Journeying *avec* Project Euler

- by Melwyn Francis Carlo

Introduction

This expository article will journey you through the path of Project Euler. The article begins with a background of Project Euler, with hints of personal background included. Next, the aims and scope of the project are discussed, followed by the discussion of a personal approach to solving the problems. An example of this approach is described in detail. Auxiliary tools to supplement the approach have also been described.

Background

Project Euler was started by Colin Hughes, who was fascinated by the world of computing since he was eleven years old, when, in 1983, his parents first gifted him an ORIC-1 micro-computer. The micro-computer got him hooked onto the world of programming and mathematics. However, the first problem, Project Euler Problem #001 was introduced in 2001, when Colin Hughes was in his late twenties.

According to Colin Hughes, “Project Euler exists to encourage, challenge, and develop the skills and enjoyment of anyone with an interest in the fascinating world of mathematics.”

The project is named after Leonard Euler, a Swiss polymath, whose major focus was in the field of mathematics, apart from the physical sciences. His contributions range from number and graph theories, to physics and astronomy.

To know more about the porject, you may refer to the ‘About’ section of the Project Euler website: https://projecteuler.net/about

Problem Archives webpage: https://projecteuler.net/archives

This project consists of some hundreds of mathematical problems to be solved, as listed on the Problems Archives webpage. The problems may be viewed without logging in, however, to submit solutions, you will have to register with the website. The list is numerically ordered, with navigational capabilities to traverse from one list to the next. To view a problem, simply click on the problem title, which will then take you to the intended webpage.

The webpage of a given problem consists of a problem title, a detailed description of the problem along with the final question, and a text box to submit the solution. In case of submission, the text box is replaced by the correct solution.

To navigate to a problem page directly, use the following link, instead:
https://projecteuler.net/problem=N

where the last character, N, is to be replaced by the problem number.

There also exists a minimalist version of the problem page containing just the problem description (and question) in plaintext HTML. To navigate, use the following link:
https://projecteuler.net/minimal=N
As of date (Friday, 23 July, 2021), there are 761 problems in all. Newer problems are added by the author as time goes by. Newer problems used by added every week, then the frequency got reduced to approximately two weeks. This is because of the difficulties involved in producing newer, challenging, and original questions. The problems are ordered in mixed order of difficulty ranging from 5% to 100%. The first 50 problems are fixed at a difficulty rating of 5%. The problems are quite diverse, but with a focus on number theory.

Solving a set of 25 problems avails you a virtual badge, stating that you have completed a new level. The badge, in the form of a PNG image, is automatically generated, and can be embedded into other applications via the following link:
https://projecteuler.net/profile/USERNAME.png

Make sure to replace the term ‘USERNAME’ with your registered username.

The problems may be solved by pen and paper. However, it is intended to be solved using a computer via one of the many available programming languages. Regardless, it is almost impossible to solve all the problems with pen and paper alone, given the functional complexity and iterative lengthiness of the problems.

On registering, you will have to select your intended choice of programming language. However, that must not stop you from trying out a number of different programming languages. I, myself, chose the trio: C, Fortran, and Ada. On my profile, I have mentioned ‘C’ as my language of choice, which will be used for statistical purposes, along with my Indian nationality.

I chose those three programming languages based on my interests in the field of Computer Science and having studied Aerospace Engineering in the past. C is a simple, straighforward, low-level language, used primarily for creating hardware drivers; so, low-level access, and consequently speed and efficiency comes to mind. Fortran is used extensively for scientific computing; a language created primarily for operating on mathematical functions and matrices. Ada is used for designing mission-critical systems, like software for missiles, drones, a commercial airplane’s autopilot system, etc. It gives importance to safety and reliability, at the expense of verbose complexity and certain inefficiencies.

To date, there are about one million users from all over the world coding in about a hundred different programming languages. Just like the mathematical problems themselves, the userbase, too, is quite diverse. The number of users who have solved a particular problem may range from under a million to about a hundred. The high end of the range involve the easy problems, where as the low end imply that the problems are quite challenging to solve. The range is inversely proportional to the difficulty rating.

Aim and Scope

The aim of this project is to solve as many problems as possible throughout one’s lifetime. The solution to each problem must be obtained within the constraints of time: solvable in less than a minute, and within the constraints of space: algorithm operation achievable using a modestly powered computer.

In other words, the solution algorithm for each problem must be as simple, straight-forward, and efficient as possible. So, basically, avoid niche algorithms that rely on accumulating gigabytes of data onto RAM, or on threading processes that utilize a large number of CPU cores. While compiling programs, set the appropriate flags to allow for machine-independent code generation. There is no requirement to solve the problems in an orderly manner. The problems may be solved in order of difficulty, or merely convenience. It must be noted that, after submitting a solution to a given problem, should the solution turn out to be incorrect, you will then have to wait 30 seconds before you are allowed a further submission.

The solutions and algorithms (including readymade code) to solve the problems are available on the internet, should you wish to embark on a strenuous search. However, that is not the aim of Project Euler; merely expediency. Project Euler is a self-journey, a self-quest; and being so, the ride will only be worth it if it is riden truthfully, with diligence and intrigue.

Another important aim of Project Euler is to learn new mathematical concepts; abstract mathematical concepts. So, looking up mathematical concepts, formulae or approaches, in books or on the internet, is not constituted as cheating.

Approach

The solution for a given Project Euler problem must go through the following 8 steps:

1. Studying the underlying mathematical concepts
2. Brainstorming a solution
3. Scribbling down an algorithm
4. Converting the algorithm into code
5. Fixing compiler errors, if any
6. Fixing run-time errors, if any
7. Code optimization
8. Code refactoring

The steps must be followed in that order. It is imperative to first focus on the mathematical aspect of the problem, and then later on the stylistic and efficiency aspects. The steps are explained in detail below. Note that steps 7 and 8 must each be followed by steps 5 and 6.

Step 1: Studying the underlying mathematical concepts

When reading the problem question, one should be able to tell from which mathematical stream does the problem stem. If you are aware of the mathematical concept or formula byheart, then proceed onto the next step. If not, then pick up a mathematical book or go to Google, and look for ideas (NOT solutions) on how to solve similar problems. You must not be tempted to copy-paste the problem question into the search box. Instead, use vague, generic terms to describe the problem.

Step 2: Brainstorming a solution

If the solution is short and straightforward, then you may proceed onto the next step. Otherwise, brainstorm and jot down the steps to be taken to arrive at a solution. It may not necessarily be specific, but like a roadmap. The brainstormed solution may take the form of a written procedure or a graphical flowchart.

Step 3: Scribbling down an algorithm

Based on the steps, prepare an algorithm to view and understand the solution in a way a computer would. The algorithm may be written in the standard algorithm format, or in the form of pseudocode. If you are an advanced programmer, then you may skip this step.

Step 4: Converting the algorithm into code

Convert the algorithm into a specific code written in your preferred programming language.

Step 5: Fixing compiler errors, if any

During compilation, set flags that test against all types of errors and warnings. Also set the programming language version to the latest (ISO) standard. If you encounter any errors, fix them, and repeat the current step until no errors are encountered. At the end of this step, the compilation process must have produced an executable file.

Step 6: Fixing run-time errors, if any

Fixing compiler (or static) errors does not imply that the code is error-free. Run-time (or dynamic) errors cannot be detected by compilers because they are treated as being statically correct. It is only when the code is being run, the functions are being executed, and the variables are being assigned and operated on, that the computer picks up on the errors. In circumstances like these, either the program may stop running mid-way, or the program may output unpredictable, out-of-range, or sheer gibberish results. Just as in the previous step, on encountering errors, fix them, and repeat the current step until no errors are encountered.

Compiler errors are easy to fix because the compiler provides detailed explanation of the errors, including its location within your code, in the form of line and column numbers.

Run-time errors, on the other hand, do not necessarily provide any explanation. So, the impetus to fix the errors rest almost entirely on the programmer’s wit to figure it out.

Run-time errors may not always denote an incorrectly written code. Sometimes, it could denote that the algorithm, or yet still, the brainstormed solution, itself, is incorrect.

A main cue signalling that a run-time error has occurred is that if the submitted solution turns out to be incorrect. Other cues include: change in sign, an out-of-range value, a zero value, an exceedingly large or small value, etc.

If you encounter something known as a ‘segmentation fault’, then it is an error related to the misuse of computer memory, especially when you use arrays in your code. It usually implies that the index of an array is out of bounds. For example, if a 3-element array, indexed from 0 to 2, is referenced to by an index less than zero (< 0) or greater than two (> 2). It could also imply that a statically

initialized array contains a large number of elements, which may not be permissible. In that case, you may dynamically declare and initialize the array variable.

Step 7: Code optimization

Code optimization involves re-writing snippets of code that would bear a positive impact on the execution time of the program.

Some of the examples include:

1. Reducing the number of loop iterations:

Consider finding the greatest something number from 1 to 1000. Being the greatest, it would be reasonable to assume that the number lies somewhere between 500 and 1000. Taking that into account, we would then start our iteration from the number 500 onward, instead of 1.

2. Skipping loop iterations:

Considering looping from number 1 to 100, and finding prime numbers within that range. It would be sensible to skip: all even numbers, numbers whose sum of digits is divisible by 3, and numbers whose last digit is a 5.

3. Breaking early out of loop iterations and functions:

Consider finding the least something number from 1 to 1000. Being the greatest, it would be reasonable to assume that the number lies somewhere between 1 and 500. Regardless, the moment the least number is found, it would be sensible to break out of the loop.

If the above example is embedded within a function, then instead of breaking out of the loop, one could simply break out of the function, that is, return from the function.

4. Displacing constant value operations outside a loop:

Consider looping through the following equation: (i * (2.0 / 10.0)) + (i * 23.46 * 3.5)
where ‘i’ is the ever-changing loop counter variable.

If the above loop iterates from 1 to 1000, then the corresponding equation would be calculated 1000 times. The equation consists of five computational steps: one division, three multiplications, and one addition.

Now, (i * (2.0 / 10.0)) + (i * 23.46 * 3.5) = i * ((2.0 / 10.0) + (23.46 * 3.5))

Let ‘C’ be a constant variable such that, C = ((2.0 / 10.0) + (23.46 * 3.5)) = 0.2 + 82.11 = 82.31

Therefore, (i * (2.0 / 10.0)) + (i * 23.46 * 3.5) = i * C

Placing the constant variable ‘C’ outside the loop reduces the operation to just one computational step, that is, one multiplication.

5. Factoring lengthy computations:

Consider the following equivalent equations: (x+y)^2 = (x^2 + 2xy + y^2)

The left hand side equation involves two computational steps: one addition, followed by either one multiplication or one power function; so, (x+y) * (x+y) OR (x+y)^2 .
However, note that the multiplicative option would rather be preferable, the reasons for which are described in an ordered list below.

Now, the right hand side equation involves five computational steps: three multiplications, followed by two additions.
Hence, the left hand side equation would be the preferred equation to use in the code.

A list of operations, in an ascending order, with respect to the execution time has been given below. The quickest operations are at the top, and the slowest operations are at the bottom of the list.

1. Addition / Subtraction
2. Multiplication
3. Division
4. Power and square root functions
5. Sytem-defined functions
6. Overhead due to user-defined function calls

Using simple ‘IF’ conditional statements and breaking out of the loop or function is much more efficient than using multiple nested ‘IF’ statements.

The code must be optimized by keeping all of this is mind.

Step 8: Code refactoring

Code optimization tended toward efficiency with respect to time. Code refactoring, on the other hand, tends toward efficiency with respect to physical space. Basically, it’s aim is to shorten the code, discard redundancies and repetitions in code.

So, unused variables may be discarded, variables that do change their values over time may be declared as constant, and snippets of code that appear more than two times may be defined as a function and referred to in the main code where needed.

Example: Project Euler – Problems Archives – Problem #001

Title: Multiples of 3 and 5

Problem: 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.

Solution: The tedious way, obviously, is to iterate from 1 to 999 using a 'FOR' or a 'WHILE' loop, as shown in the generic algorithm below.

```
Sum = 0

I = 0

WHILE I <= 999

    IF I Modulo 3 = 0 or I Modulo 5 = 0 THEN
        Sum = Sum + I
    END

    I = I + 1

END

PRINT Sum
```

A quick and efficient way, however, is to make use of explicit non-iterative formulae. First, get the number of multiples of 3 between 1 and 999. Then, get the number of multiples of 5. Interestingly, the total number of multiples is more than expected. That is because certain multiples of 3 coincide with certain multiples of 5. The coinciding numbers, fortunately, also happen to be multiples of 15. Hence, the actual total number of multiples is the uncorrected total number of multiples, minus the number of multiples of 15.

Number of multiples of 3 = ⌊999 / 3⌋ = ⌊333.0⌋ = 333
Number of multiples of 5 = ⌊999 / 5⌋ = ⌊199.8⌋ = 199
Number of multiples of 15 = ⌊999 / 15⌋ = ⌊ 66.6⌋ = 66

Let the number of multiples of 3 be denoted by the letter 'N'. Now, say, the sum of the multiples of 3, is three times the sum of the numbers from 1 to the number of multiples of 3, that is,

Sum of multiples of 3 = 3 * (Summation of i from 1 to N)

Using the summation formula, Sum of multiples of 3 = 3 * (N * (N + 1) / 2)

By backchecking, the sum of multiples of 3 under 10
= 3 + 6 + 9
= 3 * (1 + 2 + 3)
= 3 * (Summation of i from 1 to N)

where, N = ⌊9 / 3⌋ = ⌊3⌋ = 3

As a solution to the problem question,

Sum of multiples of 3 or 5 under 1000 = 3 * (Summation of i from 1 to N1)
+ 5 * (Summation of i from 1 to N2)
- 15 * (Summation of i from 1 to N3)

where, N1, N2, and N3 are the number of multiples of 3, 5, and 15 respectively.

The C, Fortran, and Ada implementations of the above solution are as follows:

```
#include <stdio.h>
#include <math.h>

/* Copyright 2021 Melwyn Francis Carlo */

int main()
{
	int val_1a = floor((1000-1) / 3);
	int val_1b = 3 * ((val_1a * (val_1a + 1)) / 2);
	int val_2a = floor((1000-1) / 5);
	int val_2b = 5 * ((val_2a * (val_2a + 1)) / 2);
	int val_3a = floor((1000-1) / 15);
	int val_3b = 15 * ((val_3a * (val_3a + 1)) / 2);

	printf("%d\n", val_1b + val_2b - val_3b);

	return 0;
}
```

```
PROGRAM F001

! Copyright 2021 Melwyn Francis Carlo

IMPLICIT NONE

INTEGER, PARAMETER :: VAL_1A = FLOOR((1000.0 - 1.0) / 3.0);
INTEGER, PARAMETER :: VAL_1B = 3 * ((VAL_1A * (VAL_1A + 1)) / 2);
INTEGER, PARAMETER :: VAL_2A = FLOOR((1000.0 - 1.0) / 5.0);
INTEGER, PARAMETER :: VAL_2B = 5 * ((VAL_2A * (VAL_2A + 1)) / 2);
INTEGER, PARAMETER :: VAL_3A = FLOOR((1000.0 - 1.0) / 15.0);
INTEGER, PARAMETER :: VAL_3B = 15 * ((VAL_3A * (VAL_3A + 1)) / 2);

PRINT ('(I0)'), VAL_1B + VAL_2B - VAL_3B

END PROGRAM F001
```

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;

--  Copyright 2021 Melwyn Francis Carlo

procedure A001 is

        use Ada.Text_IO;
        use Ada.Integer_Text_IO;

        Val_1A, Val_1B, Val_2A, Val_2B, Val_3A, Val_3B : Float;

begin

        Val_1A := Float'Floor ((1000.0 - 1.0) / 3.0);
        Val_1B := 3.0 * ((Val_1A * (Val_1A + 1.0)) / 2.0);
        Val_2A := Float'Floor ((1000.0 - 1.0) / 5.0);
        Val_2B := 5.0 * ((Val_2A * (Val_2A + 1.0)) / 2.0);
        Val_3A := Float'Floor ((1000.0 - 1.0) / 15.0);
        Val_3B := 15.0 * ((Val_3A * (Val_3A + 1.0)) / 2.0);

        Put (Integer (Val_1B + Val_2B - Val_3B), Width => 0);

end A001;
```

A number of problems have already been solved, which are available for viewing and editing at the following link, which is a Github repository:

https://github.com/melwyncarlo/ProjectEuler

The repository is updated infrequently, with the addition of newer solutions for the unsolved problems. It is best advised that you view the solution after you have created one yourself.
That way you can compare the solutions and learn newer, diverse ways to solve the problems.

Auxiliary Tools

The above mentioned repository also contains a template folder, and a copy of it in the ZIP format for download. The template folder contains template code for the C, Fortran, and Ada programming laguages. The user instructions for utilizing the files in the template folder are described in the READ-ME section of the repository. Moreover, the folder also contains a few supplementary tools to aid in the creation, maintainance, and analysis of the created problems. These supplementary tools are Bash scripts. These scripts may be modified as required, to suit personal needs. The tools included are as follows:

1. make
This script is the most important script. This script accomplishes two roles: make and make clean. ‘make’ basically acts as a makefile, though it also acts as the make command. It compiles all the solution code files in each of the available problem directories. In case of error, it notifies the error, and also opens up the error files for the user to make the necessary corrections. In case of successful compilation, the script creates the ‘run’ script, which will be described next.
‘make clean’ basically cleans up all the executables files generated during compilation from the ‘make’ process. It resets the main project directory to its default state.

2. run
This script is non-existent by default. It is only generated on successful compilation by the above mentioned ‘make’ script. This script runs the executables of all the solution code files from the problem directory as requested by ther user, and displays their output as a list onto the terminal window. If the output from all the solution code files, meaning the various programming languages, produce the same result, then the problem has been correctly and successfully solved in all of the chosen programming languages.

3. iconify
This script adds custom icons to the various files in the available problem directories. The custom icons are obtained from the ‘icons’ directory.

4. Statistics.ods
This is a spreadsheet that contains the following fields: problem title, and completion status, approximate lines of code, and execution time (in seconds) for each programming language.
The problem title column can be filled with the help of the ‘listitles’ script.
The other three columns can be filled with the help of the ‘analyze’ script.

5. analyze
This script reads through the solution code files in each of the available problem directories, and checks for completion. If complete, the script then counts the number of lines of code, and then further acquires the execution time (in seconds) of the code.

6. listitles
This script obtains a list of problem titles for all available problems on the Project Euler website.

7. problemize
This script obtains a list of problem descriptions (questions) for all available problems on the Project Euler website. This script creates or rewrites the ‘Problem’ file in each of the available problem directories. Note that, although the script does its best to remove HTML tags, and to reformat and restructure the text, it is advisable to review the generated text, and make any additional changes by hand, if required.

Conclusion

Project Euler is an excellent past-time hobby for the avid learner. It helps to expand one’s mental faculties, apart from forming an excellent practice tool for mastering various programming languages. However, the solutions must never be copied or viewed; not only is it malpractice, but you would also be depriving yourself of useful knowledge in the fields of mathematics and computing. The project’s background and practical instructions have been described in detail, alongside an example.