Project Euler – Problems Archives
Solutions Commentary

-by Melwyn Francis Carlo

Problem 53

As the problem clearly states that the first solution is available at $n = 23$, so start from there. ${}_nC_1 = n$, so you may skip $r = 1$. Funny thing is that the values increase from $r = 1$ to $r = \lfloor n \div 2 \rfloor$. After that, the values decrease, and in a symmetric manner. So, if you get k required values from from $r = 1$ to $r = \lfloor n \div 2 \rfloor$, then simply assume $2 \cdot k$ values instead.

Again, if n is an even number, then $\lfloor n \div 2 \rfloor = n \div 2$. In this case, an additional copy of the k value, that was obtained at $r = n \div 2$, will not available throughout the r range.
For odd numbers, there will be two copies of this k:
one at $r = \lfloor n \div 2 \rfloor$, and the other at $r = \lceil n \div 2 \rceil$.

Problem 56

A power value containing the maximum digital sum would imply a power value containing, in majority, the numbers greater than five, in particular, eight and nine. In such a case, the numbers between 90 and 99 come to mind.

Problem 57

An interesting pattern can be observed in the question.

$$\text{Let} \quad \frac{a_1}{b_1} = \frac{3}{2}, \quad \frac{a_2}{b_2} = \frac{7}{5}, \quad \frac{a_3}{b_3} = \frac{17}{12}, \quad \frac{a_4}{b_4} = \frac{41}{29}$$

$$b_2 + b_1 = a_2 \qquad 5 + 2 = 7$$

$$b_2 - b_1 = a_1 \qquad 5 - 2 = 3$$

In other words, $b_n + b_{n-1} = a_n$
and $b_n - b_{n-1} = a_{n-1}$

Now, the knowns are a_{n-1} and b_{n-1},
while the unknowns are a_n and b_n.

Therefore, $b_n = a_{n-1} + b_{n-1}$
and, $a_n = b_n + b_{n-1}$

The initial values are: $a_8 = 1393$
and $b_8 = 985$

Problem 58

I started out by using the prime numbers file used in problem number 3, which contained one million prime numbers. But I got the wrong answer. It turns out that, the problem asks for more than a million prime numbers. So, I took to problem number 7 which relied on the algorithm of Sieves of Eratosthenes. This time, I foolishly set the dynamic array length to one million, thereby only accumulating the prime numbers from number one to a million, and tested against them.
As obvious, it didn't work. Then, I slowly started increasing the dynamic array length, and testing on a trial-and-error basis. At last, it finally worked, when I set the array length to 700 million. Considering that an integer variable is worth four bytes (4B), an integer array of 700 million integers would occupy 2.8 GB (giga-bytes) of space on RAM. That is a lot.

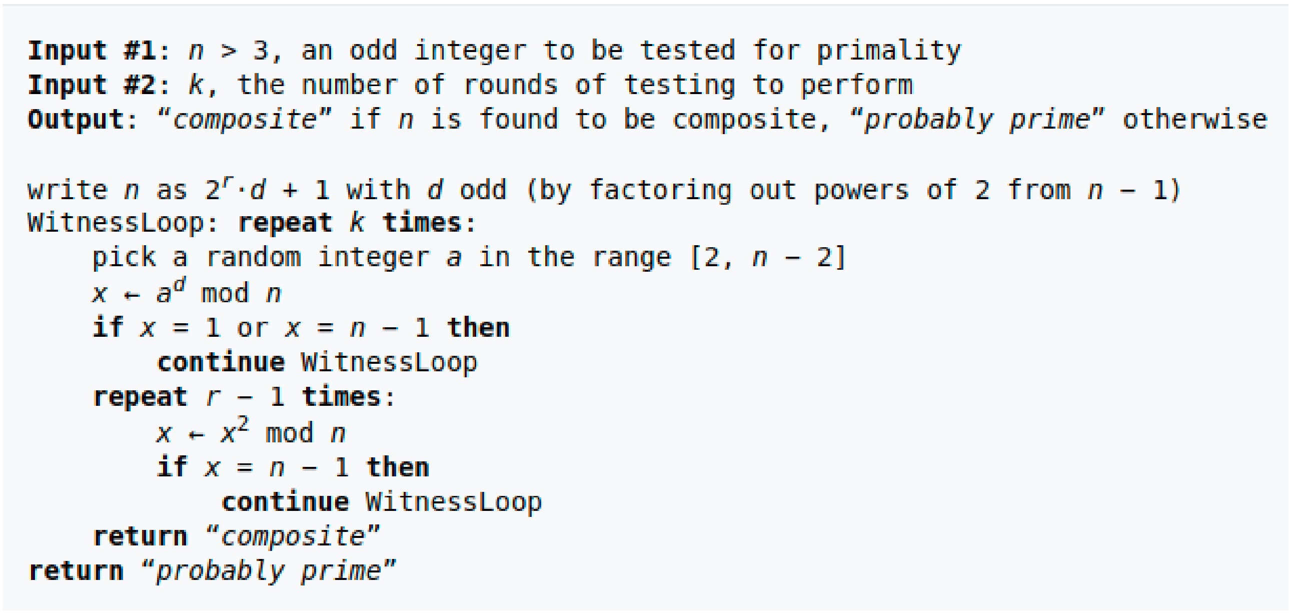
To avoid using that much RAM space, I decided to create my own prime numbers test file for values between one and 700 million. I did this by using the sieve algorithm, but I saved the data in ones and zeroes: one denotes a prime number, zero denotes otherwise. The character position of each binary digit corresponds to the natural number between 1 and 700 million. The C code generating this file has been located in the problem's directory.

I, then, used this file to solve the problem, which used negligible RAM space and wasn't time consuming either (about a couple seconds).

Upon reviewing the works of other programmers, the best suggestion was to rely on a primality test instead. I was new to this concept, so I didn't think of it. The Miller-Rabin primality test was used in this case. Given below, I have created a C version code of the original pseudocode.

The Miller-Rabin test is a probabilistic version of the original Miller test, which is deterministic. The deterministic algorithm is very accurate, but it is also very time consuming. The difference between the two algorithms is minute, pertaining to the WitnessLoop conditions.

Original pseudocode:



```
Input #1: $n > 3$, an odd integer to be tested for primality
Input #2: $k$, the number of rounds of testing to perform
Output: "composite" if $n$ is found to be composite, "probably prime" otherwise

write $n$ as $2^r \cdot d + 1$ with $d$ odd (by factoring out powers of 2 from $n - 1$)
WitnessLoop: repeat $k$ times:
    pick a random integer $a$ in the range $[2, n - 2]$
    $x \leftarrow a^d \bmod n$
    if $x = 1$ or $x = n - 1$ then
        continue WitnessLoop
    repeat $r - 1$ times:
        $x \leftarrow x^2 \bmod n$
        if $x = n - 1$ then
            continue WitnessLoop
    return "composite"
return "probably prime"
```

Reference: *https://en.wikipedia.org/wiki/Miller–Rabin_primality_test#Miller–Rabin_test*

<u>C code:</u>

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

/* This value may be changed as required. */
#define K 10

/*
Input:
        n := number to be tested against
        k := number of iterations
Output:
        0 := composite
        1 := probably prime
*/

int miller_rabin_primality_test (int n)
{
        if (n  <  2)  return 0;

        if (n == 2)  return 1;

        if ((n % 2) == 0)  return 0;


        srand (time (NULL));


        /*
                // An analytical solution for estimating the ‘r’ value.

                // Pick an odd value for ‘d’.
                int d = 3;

                inr r  = (log10 (n – 1)  –  log10 (d)) / log10 (2);
        */


        int d = n – 1;
        int r  = 0;

        while ((d % 2) == 0)
        {
                d /= 2;
                r++;
        }
```

```
    int continue_witness_loop = 0;

    for (int i = 1; i <= k; i++)
    {
        int a = 2 + (rand () % (n – 3));
        int x = pow ((double) a, (double) d) % n;

        if ((x == 1) || (x == (n – 1)))  continue;

        for (int j = 1; j <= (r – 1); j++)
        {
            x = (x * x) % n;

            if (x == (n – 1))
            {
                continue_witness_loop = 1;
                break;
            }
        }

        if (continue_witness_loop)  continue;

        return 0;
    }

    return 1;
}
```

Problem 63

This problem uses the same power function developed in my solution for problem 56. But, the function has been renamed, and its operations have been slightly optimised for this particular problem.

Problem 80

While this wasn't a challenging one, I got stuck nonetheless. On reviewing the forum chat, I realized that I had made the most silly mistake of all: at the 99th decimal place, I hadn't rounded up. So, to correct the error, I calculated upto the 100th decimal place (from i < 99 to i <= 99), and if the final digit was greater than or equal to five, I unit-incremented the 99th decimal digit (100th digit, if you include the integer digit).

But, then again, I failed to get the correct answer. On further review, I had realised that I had to round it up at about the 102nd digit. Upon extending the precision, I managed to obtain the required solution. However, I, just as a few others, blame the question for not being explicit; for one may guess the answer, but never be made to guess the question.

Furthermore, the term 'decimal digits' in the question is misleading to many. They rightly assume it to be the digits following the decimal point, however, according to the author, it includes the integer part of the number as well. I, fortunately, wasn’t given to that trap, as I had tested for the given digital sum of the square root of two against both the scenarios.

Problem 91

Although there exists an even concise solution, using the greatest common divisor function, I was not learned of its application in this particular case. Hence, I chose the simpler, brute force route, instead.

Regardless, the brute force method is not simple unless you are precise enough in marking down the constraints. For that, you must first be able to picture the implausible triangle cases in your head. Note that the triangles that you are imagnining need not be right angled.

Problem 94

The code could perhaps be optimised to run a little faster.

Problem 97

The trick is to solve just the last ten digits, whether it’s addition or multiplication. Calculate digit-wise, from right to left, upto the tenth digit; then, stop. That saves a lot of time.