Resilience

Project Euler problem 243

Mihkel Jõhvik
University of Tartu
a72091@ut.ee

Abstract

This is a project report on the solution of Project Euler's problem 243. We start out with a brute-force approach and see how various optimizations impact the speed of the algorithm.

Introduction

ProjectEuler.net is a website dedicated to mathematical programming exercises [1]. Registered members have access to over 400 tasks and registration is free. Each exercise has a numeric answer and can be solved using any programming language. While not a requirement, the style guide states that the running time of a program that computes the correct answer should roughly 1 minute. Thus, brute-force solutions are accepted, but users are encouraged to optimize their algorithms. Answers are accepted by typing in a control code (a *captcha*) and the solution. The following project attempts to solve Problem 243, first by brute force and then by applying different optimizations to the algorithm.

Problem 243

The problem statement is as follows: "Find the smallest denominator d, having a resilience $R(d) < \frac{15499}{94744}$."

Resilience of a denominator is defined as the ratio of proper fractions that are resilient to the overall proper fractions. Resilient fraction is a fraction that cannot be reduced further, like $\frac{2}{5}$. A proper fraction of a denominator is any fraction that's numerator is less than the denominator. For example, given the denominator 6, the proper fractions would be:

$\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}$ the resilient fractions would be $\frac{1}{6}, \frac{5}{6}$ and the resilience of the denominator would be $\frac{2}{5}$ [2].

At the time of writing 3878 people have solved the problem.

Brute-force approach

The brute-force approach is as-follows:
We iterate over every $d > 1$ and calculate its resilience by finding all its resilient proper factions by first calculating all its divisors, except 1 and iterating over all the proper fractions. The Python code:

```
def divisors(n):
    div = [n]
    d = 2
    while d < n:
        if n % d == 0:
            div.append(d)
        d += 1
    return div

def resilience(d):
    n = d - 1
    resilient = 0
    div = divisors(d)
    while n > 0:
        for i in div:
            if n % i == 0:
                resilient += 1
                break
        n -= 1
    return float(((d-1)-resilient)/(d-1))
```

There are no optimizations in this code and, thanks to

this, has a complexity of $O(d)$ thanks to first doing *d* comparisons in the divisors function and then *d* * *n* in the resilience function (where *n* is the number of divisors *d* has), which is quite slow at higher values. The algorithm was allowed to run for 72 hours without reaching a conclusion, before it was terminated in favor of faster options.

However, as the divisors function is a number factorization problem (finding all the factors of a number), we can decrease the running time of the algorithm quite significantly.

Optimized brute-force approach

The following is the product of a few simple optimizations of the code:

```
def divisors(n):
    div = []
    d = 2
    if n % d == 0:
        div.append(d)
    d += 1
    while d < sqrt(n):
        if n % d == 0:
            div.append(d)
            div.append(int(n/d))
        d += 2
    return div

def resilience(d):
    n = d - 1
    resilient = d - 1
    div = divisors(d)
    while n > 1:
        for i in div:
            if n % i == 0:
                resilient -= 1
                break
        n -= 1
    return float(resilient/(d-1))
```

We first skip all paired numbers and focus on the odd numbers. Secondly, we reduce the number of comparisons at the outset from *d* to $\sqrt{d}$ by using the fact that factors "pivot" after the square root. Meaning that after *n*n*, every new factor must have an equivalent smaller factor. Thus, the total decrease of time in the divisor function was $\frac{\sqrt{d}}{2}$.

Co-prime approach

The above approaches have tried to find the solution by counting how many factors each consecutive denominator has, then dividing all the numerators of its prime factors to determine whether any of them are, in fact, resilient.

Another way of applying the brute-force approach, is finding co-primes, numbers that have have the greatest common denominator of 1, of *d* in the range $1 < n < d$.

One way of doing it, is by applying the Euclidean algorithm:

```
def gcd(k,l):
    ma = max(k,l)
    mi = min(k,l)
    while mi != 0:
        temp = ma % mi
        ma = mi
        mi = temp
    return ma

def resilience(d):
    n = d - 1
    resilient = d - 1
    while n > 1:
        if gcd(n,d) != 1:
            resilient -= 1
        n -= 1
    return float(resilient/(d-1))
```

However, as the Euclidean algorithm has an upper bound of *d* and it's computed on every iteration of *d - 1*, the complexity of the above algorithm is $O(n^2)$. This algorithm was run concurrently with the other, improved brute-force approach (on separate cores of the same machine) and was stopped once it was unable to reach an answer within 43 hours.

It turns out that there is a function very fittingly called Euler's totient function that applies the co-prime calculation in a very similar way to the one above, answering the question "How many numbers in a given range are co-primes of *n*?" The function is defined as $\varphi(n)=n\prod_{p/n}(1-\frac{1}{p})$ and the product is over the distinct prime numbers dividing *n* [3]. An implementation of the totient function is as follows:

```
def totient(n):
    result = 1
    for i in ((prime-1) * prime ** (exponent-1)
              for prime, exponent in
factorize(n)):
        result *= i
    return result

def resilience(n):
    return totient(n)/(n-1)
```

The runtime complexity of this is *O(n*log*n)*, using prime factorization with a list of previously computed primes (this requires 64-bit Python). After letting the program run for roughly 1.5 weeks, manually restarting it and using multiple instances of the program with a shared pre-computed list of primes, the answer was finally found: 892371480.

Comparison to other solutions

After finding the solution and gaining access to the forums of Problem 243, it was found that the problem relates to

primorials: numbers that are computed by multiplying only prime factors. The result was a multiple of a primorial: $2*3*5*7*11*13*17*19*23*4$. The a program that calculates the first few primordial numbers and computes their resilience has a running time of < 10 ms.

Conclusion

The approach taken in this project was to brute-force the solution, which took roughly 1.5 weeks and an increasingly massive amount of memory. While the approach was not elegant, it did provide a result, albeit a far-cry from the style guide's 1 minute restriction.

The project helps demonstrate how simply throwing massive amounts of CPU cycles at a project is in no way a substitute for a good analysis of the problem and a smart algorithm.

References

[1] http://projecteuler.net/about 27.05.2013

[2] http://projecteuler.net/problem=243 27.05.2013

[3] Long, Calvin T. 1972. *Elementary Introduction to Number Theory*

[4] http://projecteuler.net/thread=243 27.05.2013 (requires solving the problem)