# Resilience

Mihkel Jõhvik

## Project Euler problem 243

Project Euler is a website dedicated to programming exercises. At the time of writing, there are over 400 exercises of various difficulty listed on the website. Each problem has a numeric answer that can be reached in around 1 minute on modern computers. Once an answer is found, users can visit forums to discuss how they reached the solution.

For every denominator *d*, there are *d-1* proper fractions. With *d = 6*, they are: $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}$ . We define a resilient fraction as one that cannot be cancelled down. These are fractions like $\frac{1}{6}, \frac{5}{6}$ for the denominator 6. We define resilience of a denominator *R(d)* as the ration of its resilient fractions to its proper fractions. Therefore, $R(6)=\frac{2}{5}=0.4$ .

**The problem: Find the smallest denominator that has a resilience smaller than that of $\frac{15499}{94744}$ .**

## Brute-force

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

For every *d*, first calculate its divisors, then see how many values of *d - 1* are also divisible by that number. As no *d-1* is divisible by *d* and no $\frac{1}{d}$ can be cancelled down, we can safely ignore these. The total number of actions is *d + d * n*, where *n* is the number of divisors *d* has.

| Approach | Complexity | Running time |
|---|---|---|
| Brute-force | $d+d*n=O(dn)$ | ∞ (stopped after 72 hours) |

From the code, it's quite clear that we are dealing with a factoring problem: finding the factors of a number. Therefore, it can easily be optimized to shave away a few operations.

## Optimized brute-force

```
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))
```

This is the optimized divisors function. Two changes of note:

**a)** we no longer use a step of 1, instead preferring to test for that straight away.

**b)** instead of doing d comparisons at the outset, we reduce it by doing $\sqrt{d}$ comparisons.

This is effective, because after $\sqrt{d}$ , the values “pivot”.

IE, to increase one would mean decreasing the other.

Therefore, instead of doing *d* comparisons at the outset, we do $\frac{\sqrt{d}}{2}$ instead.

This, however, sadly does not decrease our complexity class. And the running time remained the “same.”

## Co-primes

```
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))
```

Another way of looking at the brute-force problem: we are looking for numbers that are co-primes, meaning numbers that have the largest common denominator of 1. We can use Euclid's algorithm to find the greatest common denominator of every numerator in the set of a number's proper fractions. If the greatest common denominator is one, we have found a resilient fraction. However, the Euclidean algorithm needs at most d steps (worst case) and it's being computed at every iteration. Therefore:

| Approach | Complexity | Running time |
|---|---|---|
| Co-primes | $O(n^2)$ | ∞ (stopped after 43 hours) |

## Euler's totient function

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

def resilience(d):
    return phi(d)/(d-1)
```

Euler's totient function is a function for calculating the number of co-primes of n in some range.

The `factorize` generator function yields prime factors and their exponents of *n*. Which are then used to calculate a product.

To assist in factorization, we have a pre-calculated list of primes up to a billion stored in the system (requires a lot of memory and a 64bit version of Python). The end result:

| Approach | Complexity | Running time |
|---|---|---|
| Totient function | $O(n \log n)$ | 1.5 weeks |

After letting this run for over a week and manually adjusting parameters, the answer is: 892371480

# Comparison

Once the answer was found, we gain access to the problem's forums. There it is stated that the problem is actually related to primorials: numbers that are computed by multiplying prime numbers.

| Approach | Running time |
|---|---|
| Primorials | <10ms |

The answer was a multiple of a primorial: $2*3*5*7*11*13*17*19*23*4$ . This project is a perfect way of demonstrating how throwing huge amounts of CPU cycles at a problem is not a cure-all.