

Sieving Twin Primes

An Implementation in Haskell

Dr. Gergő [U+0091] Á [U+0089] rdi
<http://gergo.erdı.hu/>

1 Sieve of Eratosthenes

To sieve the targeted interval of generators, a set of small prime numbers is needed. These can be easily calculated using the algorithm known as the sieve of Eratosthenes. Since we only need to sieve small primes, there is no need to implement the intricate details described in [1], and instead, we can use the folklore naïve implementation[2]:

```
primes :: [Z]
primes = sieve [2..]
  where sieve (p : ns) = p : sieve [n | n ← ns, p / n]
```

2 Sieving intervals

We generate twin prime candidates using prime numbers from a given interval. The function *sieveInterval* calculates numbers between 2^k and 2^{k+n} that have no small prime divisors.

```
sieveInterval :: Z → Z → [Z]
sieveInterval k n = filter (¬ ∘ hasSmallPrimeDiv) [a..b]
  where a = 2k
        b = 2k+n
        hasSmallPrimeDiv x = any (λp → p ≢ x ∧ p | x) smallprimes
        where smallprimes = take 100 primes
```

3 The Miller-Rabin primality test

To further filter the sieved generators, and to check the actual twin prime candidates, the Miller-Rabin primality test[3] is employed.

For this, first we need to check pseudo-primality:

```
isPseudoPrime n a = isPseudoPrime' n (s, d) a
  where (s, d) = split2 (n - 1)
```

```
split2 n | 2 / n = (0, n)
          | otherwise = let (s, d) = split2 (n ÷ 2)
                          in (s + 1, d)
```

```

isPseudoPrime' n (s, d) a | a ≥ n    = error "a >= n"
                          | otherwise = repeatSquare (s - 1) x

where x = [[ad]]n
      repeatSquare r x | x ≡ 1 ∨ x ≡ n - 1 = TRUE
                      | r > 0           = let x' = [[x2]]n
                                      in repeatSquare (r - 1) x'
                      | otherwise       = FALSE

```

The Miller-Rabin algorithm tests if a given $n \in \mathbb{Z}$ is equal to one or is even (in which case it is obviously not a prime), or finds a witness, $a \in \lambda\mathbb{Z}$, for which n is not a pseudo-prime (in which case n can also not be a prime), or returns with the result "probable prime".

```

data PRIMALITY = ONE | EVEN | WITNESS  $\mathbb{Z}$  | PRIME
deriving (SHOW, EQ)

```

In the actual implementation of the Miller-Rabin primality test, we assume the generalized Riemann conjecture, which allows us to check for witnesses only in $2, \dots, \lfloor 2 \ln^2 n \rfloor$. Using this result[4], we can write a deterministic version of the primality test.

```

millrab ::  $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow$  PRIMALITY
millrab 1 k          = ONE
millrab 2 k          = PRIME
millrab 3 k          = PRIME
millrab n k | 2|n   = EVEN
              | otherwise = combine [mr a | a ← [2..k']]

where k' = minimum [k, n - 1, ⌊2 * (ln n)2⌋]
      (s, d) = split2 (n - 1)

mr a = if isPseudoPrime' n (s, d) a then PRIME else WITNESS a

combine (PRIME : rs) = combine rs
combine []           = PRIME
combine (r : rs)    = r

```

```

millrabDet n = millrab n n
isMillRabPrime p = case millrabDet p of
  PRIME → TRUE
  ⊥     → FALSE

```

4 Generating candidate primes

After using the Miller-Rabin test on the sieved generators, the actual twin prime candidates are generated using the following polynomial (The parameter m is used to set the order of magnitude of generated candidates).

```

type  $\mathcal{G} = \mathbb{Z} \rightarrow \mathbb{Z}$ 
f1 ::  $\mathbb{Z} \rightarrow \mathcal{G}$ 
f1 m x = (h0 + c * x) * 2m - 1
where h0 = 5775
      c   = product (take 6 primes)
      e   = 3299

```

So finally everything is ready to search for twin prime pairs between $f_1(2^k)$ and $f_2(2^{k+n})$:

```
findGenerators :: ℤ → ℤ → [ℤ]
findGenerators k n = filter isMillRabPrime (sieveInterval k n)
```

```
findCandidates :: (ℳ, ℳ) → ℤ → ℤ → [(ℤ, ℤ)]
findCandidates (f1, f2) k n = let gs = findGenerators k n
                                in map (λg → (f1 g, f2 g)) gs
```

5 Sieving twin primes

By this point, we have developed the necessary toolkit to create our twin prime sieve. Since the deterministic Miller-Rabin test has a time complexity in $O(\log^4 n)$, first a probabilistic Miller-Rabin test is done on both numbers of the twin prime pair candidate.

```
both :: (a → ℒ) → (a, a) → ℒ
both p (x, y) = p x ∧ p y
```

```
findPrimePairsProb :: (ℳ, ℳ) → ℤ → ℤ → [(ℤ, ℤ)]
findPrimePairsProb (f1, f2) k n = let ns = findCandidates (f1, f2) k n
                                in filter (both isMillRabPrimeProb) ns
    where isMillRabPrimeProb p = millrab p 2 ≡ PRIME
```

```
findPrimePairs :: (ℳ, ℳ) → ℤ → ℤ → [(ℤ, ℤ)]
findPrimePairs (f1, f2) k n = filter (both isMillRabPrime) (findPrimePairsProb (f1, f2) k n)
```

With this, twin prime sieving is a simple special case of searching for prime pairs in the form $(p, p + 2)$.

```
findTwinPrimes f1 = findPrimePairs (f1, f2)
    where f2 x = f1 x + 2
```

A Results

First runs of the above program actually resulted in disappointment: as soon as the numbers involved hit ten to twelve decimal digits, memory usage started to grow so fast that it filled all available space, resulting in the OS killing the program. It soon became apparent that this problem was caused by the naïve implementation of the $\llbracket x^y \rrbracket_n$ function by first doing the exponentiation, and then taking the remainder. So instead, the final version used the following implementation:

```
llbracket · \rrbracketn :: ℤ → ℤ → ℤ → ℤ
llbracket x1 \rrbracketn      = x mod n
llbracket xy \rrbracketn | 2|y = (llbracket xy÷2 \rrbracketn ↑ 2) mod n
llbracket xy \rrbracketn | 2∤y = (llbracket xy-1 \rrbracketn * x) mod n
```

The actual search for twin primes was done by setting h_0 and c (as in the definition of f_1) and targeting given orders of magnitude by tuning m . Time needed to find the first twin prime pair for a given m , and time needed to check primality of the first number of this pair was measured. Results are shown in figures 1 and 2 compared to the predicted run time in $\Theta(\log^4 p)$.

The largest twin prime pair discovered during testing was $(5775 + 30030 \cdot 261847) \cdot 2^{809} \pm 1 \sim 10^{254}$.

B Possibilities for improvements

The approach presented here lends itself to parallelisation. For starters, the Miller-Rabin testings of candidates are independent from each other, thus, this could be trivially parallelised by a suitably smart Haskell compiler[5].

On a more refined scale, checking for witnesses for a given candidate can also be done in a parallel fashion. This, however, needs some kind of synchronization since once a witness is found, checking for others would be redundant.

A simple change to the code that requires no parallelism but can speed things up is to merge the checking of p and $p + 2$, by first checking if both are pseudo-prime for 2, then if both are for 3, and so on. This way, if the first member of the pair is prime, but the second one is not, this fact is discovered before the full Miller-Rabin check of the first number would be finished.

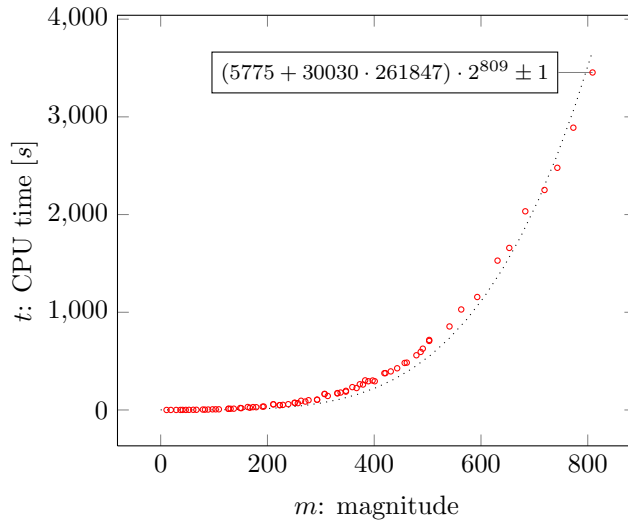


Figure 1: Time needed to find the first twin prime pair of a given magnitude

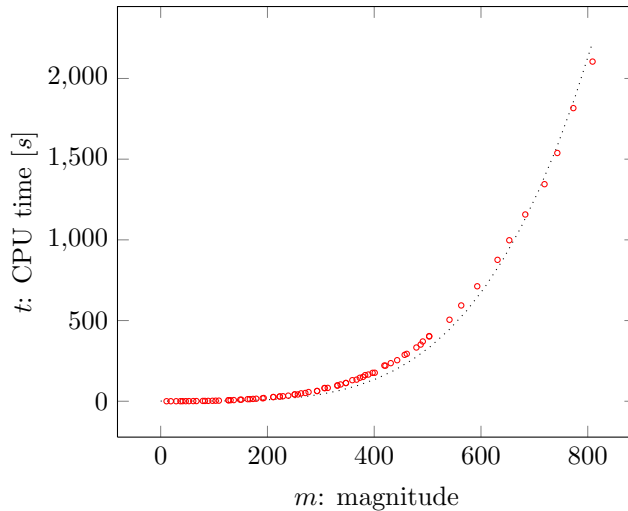


Figure 2: Running time of the deterministic Miller-Rabin primality test

References

- [1] Melissa E. O’Neill. The Genuine Sieve of Eratosthenes. *Journal of Functional Programming*, 19(01):95–106, 2009.
- [2] Lambert Meertens. FUNCTIONAL PEARL Calculating the Sieve of Eratosthenes. *Journal of Functional Programming*, 14(06):759–763, 2004.
- [3] Michael O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12(1):128–138, 1980.
- [4] Gary L. Miller. Riemann’s Hypothesis and tests for primality. In *STOC ’75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 234–239, New York, NY, USA, 1975. ACM.
- [5] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadelphia, USA, May 1996.