Probabilistic integer factorisation using Miller-Rabin primality test and Pollard's rho algorithm

Tariq

Miller-Rabin primality test

Background information

- Built on Fermat's little theorem where $A^{n-1} \equiv 1 \pmod{n}$ holds for prime n but may also hold for composite n (pseudoprime). When Fermat's little theorem holds for all bases A coprime to pseudoprime n, n is called a Carmichael number.
- Therefore, to avoid the Carmichael numbers, a stronger form of testing is used. Strong pseudoprimes n, unlike pseudoprimes, do not hold for all bases A coprime to n.

Method

By Fermat's little theorem,

$$A^{n-1} - 1 \equiv 0 \pmod{n}$$

When testing is necessary (n is odd), $n - 1$ is always even so use difference of squares to factorise:

$$\left(A^{\frac{n-1}{2}} - 1\right)\left(A^{\frac{n-1}{2}} + 1\right) \equiv 0 \pmod{n}$$

Continue this operation until $\frac{n-1}{2^s}$ is odd:

$$\left(A^{\frac{n-1}{2^s}} - 1\right)\left(A^{\frac{n-1}{2^s}} + 1\right)\left(A^{\frac{n-1}{2^{s-1}}} + 1\right)\cdots\left(A^{\frac{n-1}{2}} + 1\right) \equiv 0 \pmod{n}$$

The essence of the check lies in the fact that all primes and some composites (called strong pseudoprimes) will divide one of the LHS elements.

Simplification

Let $\frac{n-1}{2^s} = d$, where d is some odd integer $\Longrightarrow n - 1 = d \cdot 2^s$.
Checking all elements can be simplified to:

Leftmost element:

$$n \mid \left(A^{\frac{n-1}{2^s}} - 1\right)$$
$$\Leftrightarrow A^{\frac{n-1}{2^s}} \equiv 1 \pmod{n}$$
$$\Leftrightarrow A^{d} \equiv 1 \pmod{n}$$

Rest of elements:

$$n \mid \left(A^{\frac{n-1}{2^{s-r}}} + 1\right)$$
$$\Leftrightarrow A^{\frac{n-1}{2^{s-r}}} \equiv -1 \pmod{n}$$
$$\Leftrightarrow A^{\frac{n-1}{2^s} \cdot 2^r} \equiv -1 \pmod{n}$$

$$\Leftrightarrow A^{d \cdot 2^r} \equiv -1 (\text{mod } n)$$

for $0 \leq r < s$

Extra information

- For a 1024-bit number, 6 iterations of Miller-Rabin with random bases will result in an error rate (i.e. mislabeling a composite as a prime) of $< 10^{-40}$. However, in our case, the numbers checked are 64-bit ($< 2^{64}$) and relatively small, meaning a deterministic version of Miller-Rabin can be used to detect all composites.
- If n is composite and is a strong pseudoprime base A, A is called a strong liar. If A determines the compositeness of n, A is called a witness.
- The probability of a false positive can be made arbitrarily small with repeated tests using different bases.
- For any A used, $1 < A < n - 1$ because:

  $$A \equiv 1 (\text{mod } n) \Longrightarrow A^d \equiv 1 (\text{mod } n)$$
  $$A \equiv n - 1 \equiv -1 (\text{mod } n) \Longrightarrow A^{d \cdot 2^0} \equiv A^d \equiv -1 (\text{mod } n)$$

  In both cases, any n passes the check.

Pollard's rho algorithm

Background information

- Is an algorithm used to find non-trivial factors of n in $O\left(\sqrt{\sqrt{n}}\right) = O\left(n^{\frac{1}{4}}\right)$ rather than trial division in $O\left(\sqrt{n}\right)$.

Method

1. A polynomial $f: \mathbb{Z} \to \mathbb{Z}/n\mathbb{Z}$ of degree ≥ 2 is chosen, typically in the form $f(x) = x^2 + c \bmod n$, where $c \in \{1,2,3, \dots\}$. This polynomial is used to generate the pseudorandom sequence $\{x_k\}$.
2. Let $x_0 = 2$ and $x_{k+1} = f(x_k) \Leftrightarrow x_k = f^k(x_0)$.
3. Iterate using two pointers, $T = x_i$ and $H = x_{2i}$.
4. At each step, calculate $p = gcd(|T - H|, n)$.
    a. If $p = 1$, continue iterating.
    b. If $p = n$, increment c and restart the algorithm.
    c. If $1 < p < n$, a non-trivial factor p of n has been found.

How it works

- The algorithm creates a sequence $\{x_k\}$ modulo n that is guaranteed to generate a cycle, and there is an underlying sequence unknown to the algorithm $\{y_k\}$ where $y_k = x_k \bmod p$ and $p \mid n$.
- Pollard's rho algorithm will discover a divisor $p \neq n$ of n if it finds, for $i \neq j$:

  $x_i \equiv x_j \pmod{p} \iff p = \gcd(|x_i - x_j|, n) > 1$

  before it finds $x_i = x_j$ (note $x_i = x_j \Leftrightarrow x_i \equiv x_j \pmod{n}$ because $\{x_k\}$ is modulo n)

  Finding $x_i = x_j$ before a divisor results in $\gcd(0, n) = n$ and so the initial function addend c must be changed.

- It is important to note that the cycle in the sequence $\{y_k\}$ is smaller than the cycle in $\{x_k\}$. The birthday paradox tells us that a collision of a sequence with N possible values is expected in $O\left(\sqrt{N}\right)$. Depending on the values of the two pointers, this means a factor p (when $y_i = y_j$) is usually found before $x_i = x_j$ (where a restart with different function will be necessary), because $\{x_k\}$ and $\{y_k\}$ have n and p possible values respectively, and $O\left(\sqrt{p}\right) \subset O\left(\sqrt{n}\right)$.

Why a polynomial mapping must be used

$\mathbb{Z}/n\mathbb{Z}$, the set of least residues modulo n, is isomorphic (a bijection) to $\mathbb{Z}/p\mathbb{Z} \times \mathbb{Z}/q\mathbb{Z}$ where $n = pq$ (WLOG, let $p \leq \sqrt{n}$) by the Chinese remainder theorem. Each element $x \in \mathbb{Z}/n\mathbb{Z}$ corresponds to a unique pair $(a = x \bmod p, b = x \bmod q)$, where $a \in \mathbb{Z}/p\mathbb{Z}$ and $b \in \mathbb{Z}/q\mathbb{Z}$.

If the mapping $f: \mathbb{Z} \to \mathbb{Z}/n\mathbb{Z}$ is an arbitrary permutation, the search for p will effectively randomly move through $\mathbb{Z}/n\mathbb{Z} \in O(\sqrt{n})$ (because of the birthday paradox) $= O(\text{trial division})$.

However, if the mapping $f: \mathbb{Z} \to \mathbb{Z}/n\mathbb{Z}$ is a polynomial function such that $f(x \bmod n) = f(x) \bmod n$, and because $x_{k+1} = f(x_k) = f((a_k, b_k)) = (f(a_k), f(b_k))$, $\{y_k\}$ is now a determinable sequence where $y_{k+1} = g(y_k) = f(x_k) \bmod p = f(a_k)$, implying the search is now a random search in the domain of $a = \mathbb{Z}/p\mathbb{Z} \in O(\sqrt{p}) = O\left(\sqrt{\sqrt{n}}\right)$.

Extra information

- Floyd's cycle-finding algorithm, implemented as two pointers of different speed, is only used to save memory and is not critical when discussing the logic behind Pollard's rho algorithm.

Our implementation

Program walkthrough

1. Begin by inputting n and creating two data structures: an array and a queue. The array holds the prime factors of n, and the queue is used to perform repeated factorisation in the case that Pollard's rho algorithm outputs a composite factor of n.
2. Trial division of primes $< 10^5$ is performed because it is shown to provide empirical runtime gains. Any prime factors found are stored in the array.
3. The now factorised n is enqueued and the main loop begins. In this loop, the first element e is served, and Miller-Rabin is used to check its primality. If e is prime, it is added to the array of prime factors. If not, Pollard's rho algorithm is used to find a divisor d, and d and $\frac{e}{d}$ are enqueued.
4. Once the queue is empty, it is certain that all prime factors of n have been found and added to the array. The array is returned to the user.

Sample runs and performance analysis

The algorithm was written separately in C and Java to test the difference in performance between the two languages.

C

```
Enter number to factorise (max 64-bit):
378213
378213 = 3*11*73*157
Starting factorisation of first 10^8 numbers...
Factorising the first 10^8 numbers took 301.4294411994 seconds.
```

Factorisation test + time taken to factorise the first 100 000 000 numbers

```
Enter number to factorise (max 64-bit):
56782137892
56782137892 = 2*2*23*617197151
Starting factorisation of first 10^7 numbers...
Factorising the first 10^7 numbers took 23.4294139732 seconds.
```

```
Enter number to factorise (max 64-bit):
567363721
567363721 = 12659*44819
Starting factorisation of first 10^7 numbers...
Factorising the first 10^7 numbers took 23.4294138598 seconds.
```

Factorisation test + time taken to factorise the first 10 000 000 numbers

The algorithm was not written for speed, and I am confident it can be improved further. ~97% of all 32GB of RAM was used when factorising the first 100 000 000 numbers.

Java

```
Enter number to factorise (max 64-bit):
9832671
9832671 = 3*3*3*3*19*6389

Starting factorisation of first 10^7 numbers...
Factorising the first 10^7 numbers took 674.117 seconds.
```

Factorisation test + time taken to factorise the first 10 000 000 numbers

```
Enter number to factorise (max 64-bit):
3978621
3978621 = 3*3*442069

Starting factorisation of first 10^6 numbers...
Factorising the first 10^6 numbers took 51.490 seconds.
```

Factorisation test + time taken to factorise the first 1 000 000 numbers

I expected Java to be slower than C, especially with the use of BigInteger's. I did not, however, expect it to be nearly 30x slower. It should be noted that Eclipse seems to have limited RAM consumption to 40% (12.8GB).