Pascal Chatterjee (881216-0938)
Joakim Annebäck (870720-6598)

Integer Factorisation

By Pascal Chatterjee & Joakim Annebäck

The Problem

Integer factorisation is the problem of reducing an integer to its prime factors. An example is the factorisation of 175891579187581657617 into its four prime factors:

$$175891579187581657617 = 3 \times 7 \times 71 \times 117968865987646987$$

Interestingly, though this problem (or its decision variant) is known to be hard, it is not officially NP-complete as no reduction to a known NP-complete problem has yet been found. However, as no polynomial time algorithms have been found, the problem is not thought to be in P either. This makes integer factorisation a likely candidate for being an NP-intermediate class problem - i.e. neither in P nor NP-complete (assuming, of course, that P /= NP).

A Naive Solution

The first algorithm that springs to mind when trying to find factors of *N* is simply to test whether numbers up to $\sqrt{N}$ are divisors. Once we have found a factor, we print it if it is prime, and recursively factorise it otherwise. The primality test can be efficiently implemented using the Miller-Rabin algorithm, but of course the number of divisor-checks is untractable for very large *N*.

Limitations

A limitation of using Miller-Rabin for primality testing is that it will fail for **Carmichael numbers** - those numbers which in the eyes of Miller-Rabin behave just like primes yet are in fact composite.

A Random Approach

Another approach to finding non-trivial factors of *N*, though it is more of a heuristic than an algorithm, is to use chance. Let us assume that *p* is a non-trivial factor of *N*. As we know from the Birthday Problem, it is possible to find two numbers *x* and *y* such that

$$x \equiv y \text{ (mod } p\text{)}$$

with a probability of ½ after $1.177\sqrt{p}$ numbers have been randomly chosen. Once we have such an *x* and *y*, we know that

$$x - y \equiv 0 \text{ (mod } p\text{)} \rightarrow p \mid (x-y)$$

and we knew from before that $p \mid N$. Together this means that

$$p \le gcd(|x-y|, N) \le N$$

i.e. $gcd(|x-y|, N)$ is a non-trivial factor of *N* greater than *p*. At this point we can return this factor, or `failure` if this factor is actually *N* itself.

Cycle finding algorithms

In the approach above, we picked *x* and *y* as random numbers smaller than *N*, hoping we'd find a pair such that $x \equiv y$ (mod some *p*). It turns out we can improve on this selection process, by realising that our selection is a function on the finite set {0... *N-1*}, and $x \equiv y$ (mod some *p*) if a cycle occurs in this function.

Floyd’s Algorithm

Floyd’s algorithm, also known as ‘The Tortoise and the Hare’, is the simplest cycle detection algorithm. It features two pointers, one called the tortoise and one called the hare, that iterate through the values of our function. The hare advances by two steps for every iteration while the tortoise advances by one.

After a while, the sequence will begin to repeat itself, and as the hare is moving faster it will enter this loop first. Eventually the tortoise will enter the loop too, and after a few iterations there must come a point where they land on the same value. When this happens, we can be sure that a loop exists.

Brent’s Algorithm

Richard Brent improved on Floyd’s algorithm by reducing the number of iterations required to detect a loop, and also the number of function calls executed per iteration. In this variant, the tortoise is stationary while the hare advances one step per iteration, but after a certain interval the tortoise jumps to the hare’s position. This interval is initially two steps, and then doubles every time.

Again, the algorithm will terminate when the hare and tortoise land on the same value, but this time the tortoise will not get “left behind” during the journey up the tail of the rho. Once both pointers are in the loop the increasing interval size ensures that the interval eventually exceeds the length of the loop, allowing the hare and the tortoise to have their rendezvous.

Application to Integer Factorising

Pollard’s Rho and “Brent’s rho”

An implementation of Floyd’s cycle algorithm in factorising integers can be found in Pollard’s rho algorithm. Here we want to find a cycle in a function, $f(x)$, that is mapped to and from the set {0...*N-1*}. However, this time, our condition for having found a cycle is not that the tortoise, *x*, and the hare, *y*, point at the same value, it is that $x \equiv y$ (mod some *p*). We check this condition by calculating $gcd(x-y, N)$ - if this is non-trivial (not 1), then *x* is congruent with *y*, so we have not only found a cycle but we have also found a divisor.

What remains is to find a suitable function to iterate through. It turns out that

$$f(x) = x^2 + 1 \,(\text{mod } N)$$

is as good as a random function, while not incurring the overhead of having to generate random numbers all the time.

“Brent’s rho” is the same as Pollard’s rho, but with Floyd’s cycle finding algorithm swapped for Brent’s own.

Optimisations

Randomising some variables

If we consider the rho-shape f iterates through, we can see the loop (or *ring*) containing the values {0... *p-1*}, and the tail containing the values {*p*...*N-1*}, for each *p* that is a divisor of *N*. As divisors always occur *at least* in pairs we can call them *p*,*q*, with $p \leq q$. An optimisation is to try and land on *q* instead of *p* when searching for a divisor of *N*, as when we later divide *N* by *q* we will be left with a smaller number to factorise recursively.

We can try to land on *q* by randomising the starting values for the cycle finding algorithm:

$$x_0 = 0 \leq \textit{random number} \leq N-1$$
$$y_0 = x_0$$

and also randomising the function

$$f(x) = x^2 + c \text{ where } c = 0 \leq \textit{random number} \leq N-1$$

Keeping potential small prime factors in memory

Of course, to save having to do unnecessary work factorising smaller numbers, or any number with small prime factors, we can keep a list of these small prime factors in memory and test whether they divide *N*; if they do, we have found a divisor in constant time.

Parallelising

As our recursive calls to `factor` are independent, and have no side effects, we can run them in parallel using as many threads as the computer has cores. The only issue is that these threads may not be load balanced very well, in the case that one thread has to factorise a large divisor of N while another must factorise a smaller one.

Sieve Algorithms

Quadratic Sieve

When trying to factorise N, the basic aim of the Quadratic Sieve algorithm is, like Fermat's factorisation method, to find x and y such that

$$x^2 \equiv y^2 \ (\text{mod}\ N) \ \text{ where } \ x \not\equiv \pm y \ (\text{mod}\ N)$$

as this implies that

$$x^2 - y^2 \equiv 0 \ (\text{mod}\ N) \rightarrow (x+y)(x-y) \equiv 0 \ (\text{mod}\ N) \rightarrow N \mid (x+y)(x-y)$$

Therefore, $(x+y)$ and $(x-y)$ each contain factors of N, which we can find by calculating $gcd(x+y,\ N)$ and $gcd(x-y,\ N)$.

There is no certainty that the factor we get is a nontrivial factor to N i.e. it is not N or 1. However the probability that it is a nontrivial factor is in practice about ½. If the factor is trivial we simply try again using a different x.

The main problem, of course, is finding two such squares. Luckily, it turns out that we can sometimes *manufacture* a congruence of squares. If we select several candidates, a, and then compute a^2 (mod N), even if a^2 is not a perfect square then a subset of these candidate-products will be.

So how do we find this subset? If we consider every integer to be a vector consisting of the powers of the primes that make it up, the vector of a square integer will consist of even values, or the zero vector modulo 2. Thus, given a set of candidates, or vectors, we just need to find a linear combination that results in the zero vector. This is a problem in linear algebra that can be solved with Gaussian elimination.

Now we just need to find our candidates. We do this by building a *factor base*, consisting of all primes smaller than a certain bound we call *B*, together with a "special prime", -1. We then search for *Q* such that Q^2 (mod *N*) is small and factors in the factor base. If *Q* meets our criteria we add it to our list of candidates, otherwise we discard it and move on.

Finally, we can *sieve* potential *Q*s through the squares of the factor base. It is this process that gives the Quadratic Sieve its name.

General Number Field Sieve

The *General Number Field Sieve, GNFS*, is the fastest algorithm for factorising integers larger than 100 digits. This is a improvement of the QS algorithm. In contrast to QS when it comes to searching for numbers with small prime factors, as done in the *factor base* step in QS, the GNFS has a much more likeliness of finding a factor since it uses smaller numbers. To achieve this improvement GNFS has to do computations and factorisations in *number fields*. As an effect the algorithm gets much more complicated than the simple QS.

Kattis

Performance

Our 'unoptimised' version of **Pollard's rho** factorised 22 integers out of 100.

Our 'optimised' version of **Pollard's rho** factorised **45** out of 100.
(Submission ID 248646) This was our best score.

For some reason, our versions of **Brent's rho** were slower than **Pollard's rho** (scoring around 37). This seems to be an issue with our Java code, because our implementations of **Brent's rho** consistently outperformed **Pollard's rho** in Ruby.

Due to time constraints, we did not implement sieve algorithms which seemed trickier to code efficiently.

Optimisations

Time Out

A very pragmatic optimisation when factoring huge numbers would have been to time out attempts to factor numbers that take too long. We chose not to do this because it doesn’t really have anything to do with the actual problem of factorising integers.