NUMBER FIELD SIEVES

FEDERICO BONGIORNO

Abstract. Algorithms, development, outcomes and improvements of the project undertaken with `Haskell.org` for the `arithmoi` library under supervision of Andrew Lelechenko.

Introduction

Decomposing integers into prime factors has always been at the core of arithmetic. The ancient Greeks were probably the first ones to study numbers for their own sake and they developed algorithms to effectively deduce properties about them. Among these are the sieve of Eratosthenes and the algorithm of Euclid. Despite these advances, the fastest algorithm to factor a given integer was trial division. In the XVII century, Pierre de Fermat described an alternative approach to factor integers, however the algorithm was still based on trial and error. During the last century, plenty of algorithms to factor integers arose. Among them are the Elliptic Curve Factorisation, the Quadratic Sieve and the General Number Field Sieve. Despite the fact that these algorithms require modern mathematical techniques, the latter two are still based on Fermat's idea and they are the fastest classical algorithms. Indeed, on the 28th of February 2020 , the General Number Field Sieve was used to factor the largest number to date: a 250-digit integer. On the other hand, only the Elliptic Curve Factorisation algorithm was implemented in Haskell. As an initial step, this project developed the Quadratic Sieve in Haskell.

1. Algorithms

Quadratic Sieve. Suppose n is an integer. The quadratic sieve attempts to factor n by finding two integers x and y such that

$$x^2 - y^2 = n$$

Then $(x-y)(x+y) = n$ is, in at least a half of the cases, a non-trivial factorisation of n.

Example 1.1. Let $n = 21$. Note that

$$5^2 - 2^2 = 21$$

so that

$$7 \cdot 3 = (5+2) \cdot (5-2) = 21.$$

To find such integers we compute $y_i = x_i^2 - n$ as x_i runs through the sieving interval and we look for numbers which decompose completely into prime factors, which are less than a fixed bound. These are called smooth numbers. If we can find enough smooth numbers, then we are guaranteed to find a product of y_i's which is a square.

Date: 24th of August 2020.

Indeed, this problem reduces to a linear system of equations over $\mathbb{F}_2$, the finite field with two elements. The factorisation of a smooth number will form the column of a matrix after reduction modulo 2. Finally by taking a suitable product of the x_i 's and y_i's, the end goal is achieved.

The version developed here presents three improvements. Firstly, the sieve uses the multiple polynomials Montgomery method. Instead of mapping x using the polynomial $x^2 - n$, we can use $f(x) = ax^2 + 2bx + c$ where a, b and c are integers such that $b^2 - ac = n$. Then, by completing the square, we get

$$(ax + b)^2 - n = af(x)$$

which reduces the computations to the standard case by replacing $ax + b$ with x and $af(x)$ with y. The advantage of this method is that the numbers generated by f can be made small, and so more likely to be smooth. Another improvement is to sieve with approximate logarithms. Instead of dividing by a prime, we subtract the approximate logarithm of that prime. This is faster than division. A number is then checked for being smooth if, after sieving, the corresponding logged value falls below a given threshold. A third improvement is given by considering a number to be smooth if, after dividing by all the primes in the factor base, the number remaining is itself prime. This may help to find more smooth numbers.

Wiedemann Algorithm. Given a singular square matrix A with coefficients in $\mathbb{F}_2$, the aim of this algorithm is to find a non-zero v such that $Av = 0$. The idea of the algorithm is to estimate the minimal polynomial of A and then use it to infer a solution. This can be achieved as follows. Suppose $p(x)$ is the minimal polynomial of A. Because A is singular,

$$p(x) = x^k q(x)$$

for some $q(x)$ and some $k \geq 1$. Hence

$$A^k q(A) = p(A) = 0$$

gives a non-trivial solution. To estimate $p(x)$, we pick random vectors u and w and compute

$$u^T A^j w$$

as j runs in a fixed interval. Using this data, a variation of Euclid's algorithm for polynomials (Berlekamp-Massey algorithm) gives an estimate of $p(x)$.

2. Development

Weeks 1 - 4. In the first four weeks, I developed a working version of the quadratic sieve employing Gaussian elimination in the linear algebra stage. The sieve was not dividing by higher prime powers and, as a result, it needed a much larger factor base. The algorithm was tested and benchmarked against n_{30} (30-digit integer, see §3). It took `600 s` to factor and used almost `2 GB` of memory. Most of the time was consumed in the linear algebra step.

Difficulties. The main difficulty came from using Haskell's mutable vectors. It was also difficult to mix mutable and immutable types while writing the code for the sieve.

What I learnt. I learnt to write code in Haskell using mutable vectors and many engineering tools. These include Git, GitHub and profiling tools. I also developed a better understanding of Haskell's laziness.

Code. `https://github.com/Bodigrim/arithmoi/pull/202`

Weeks 5 - 8. In the second four weeks, I developed the linear algebra routine to solve sparse binary matrices. The quadratic sieve improved considerably, taking around `60 s` to factor n_{30}. This algorithm also uses a fraction of the memory used in Gaussian elimiantion.

Difficulties. I had problems using random numbers. Because of laziness, if used naively, random numbers may produce the same output throughout the program. It was difficult to use type parameters to implement vectors of given length. GHC expected to know at each step if the parameter was fixed or could vary. I hadn't observed this syntax in any other programming language. Linking the two algorithms together was also not straightforward since rows and columns had to be indexed in the same fashion.

What I learnt. I was not aware of this linear algebra algorithm or, more in general, of the counterintuitive idea to estimate minimal polynomials to solve linear systems. I also learnt about sized vectors and how to operate with generalised abstract datatypes. I had a better understanding of data structures and when to use which.

Code. `https://github.com/Bodigrim/arithmoi/pull/208`

Week 9. I then spent the week after, implementing the multiple polynomial variant. This is an ingenious way to find more smooth numbers in a given interval. This was successful and it cut time and memory significantly. It was taking around `15 s` and `300 MB`.

Difficulties. Choosing a, the leading coefficient of the polynomial is quite easy. Choosing how a varies as multiple sieving blocks are run is much harder. It is desirable to make sure that a is not too far from a fixed values but it has to change at every block. I suspect that the best way to change a is by using random numbers.

What I learnt. I learnt about Haskell's applicative functors and monads while implementing this variant.

Week 10. At this point I realised it was necessary to sieve by higher prime powers. I only changed a few lines of code and it instantly improved the algorithm. It was then taking `6 s`. I also sieved using approximate logarithms. This approach did not require dividing by higher prime powers as the sieving was approximate. Instead, trial division is used on numbers which fall below a certain threshold after the sieve. Its performance was even better than the previous one taking only `1 s`.

Difficulties. Log sieving was actually slower at first. The logarithm of an integer was computed as a double and then floored. Taking logarithm to double precision was the problem and this was solved by calling a function previously developed by

Andrew.

What I learnt. A further speed up was obtained by using an unboxed vector for sieving rather than a boxed one. I learnt about the difference between these two types.

Code. `https://github.com/Bodigrim/arithmoi/pull/210`

Week 11. I then developed the large prime variation, a technique allowing numbers to be considered smooth even if they have one prime factor outside of the factor base. This variant further improved performance. Factoring n_{30} took `0.6 s` and `1 MB` of memory.

Difficulties. At first this variant was very inefficient as I was only picking up a couple of more smooth numbers while investing many resources in finding them. This was due to misunderstanding of the algorithm which was pointed out to me by Andrew.

What I learnt. I became familiar with using maps and better understood how they work.

Week 12. In the last week, I polished and tested the code. I also wrote the documentation and the report.

Difficulties. When testing extensively, I realised there were a few bugs that I hadn't noticed before. It took a surprising amount of time to find them.

What I learnt. I learnt better testing practices. These includes testing for edge cases and writing tests covering different facets of the algorithms. For instance, after developing the linear algebra routine, I only tested for correctedness of the solution. However, in the context of integer factorisation, it is also important for the algorithm to find distinct solutions. This aspect was left untested and the uncovering the related inefficiency later equired several hours of laborious debugging.

Code. `https://github.com/Bodigrim/arithmoi/pull/211`

3. Outcomes

The outcomes of the project are working versions of the quadratic sieve algorithm to factor an integer and the Wiedemann algorithm to solve sparse binary matrices. Here below are the main files in their final versions:

- `https://github.com/Bodigrim/arithmoi/blob/master/Math/NumberTheory/Primes/Factorisation/QuadraticSieve.hs`
- `https://github.com/Bodigrim/arithmoi/blob/master/Math/NumberTheory/Primes/Factorisation/LinearAlgebra.hs`
- `https://github.com/Bodigrim/arithmoi/blob/master/test-suite/Math/NumberTheory/Primes/QuadraticSieveTests.hs`
- `https://github.com/Bodigrim/arithmoi/blob/master/test-suite/Math/NumberTheory/Primes/LinearAlgebraTests.hs`

Here are some performance examples on my machine:

Factoring a 30-digit number n_{30}
Time: `0.599 s`
Memory: `1 MB`

Factoring a 40-digit number n_{40}
Time: `14.213 s`
Memory: `14 MB`

Factoring a 50-digit number n_{50}
Time: `216.649 s`
Memory: `82 MB`

Factoring a 60-digit number n_{60}
Time: `1746.459 s`
Memory: `427 MB`

Solving a matrix of size 1000 and density 0.005
Time: `1.140 s`
Memory: `0 MB`

Solving a matrix of size 10000 and density 0.001
Time: `99.484 s`
Memory: `6 MB`

Solving a matrix of size 20000 and density 0.0005
Time: `485.538 s`
Memory: `61 MB`

4. Improvements

Future improvements are needed, particularly in the linear algebra routine. First of all, experimental evidence suggests that when the matrix is very singular, the solutions output by the algorithm are often the same. This can be problematic. Indeed, given a solution of the matrix, one can infer a factorisation in only a half of the cases. It is desirable to make the solution vary as the initial random seed varies. Another improvement to be carried out is the block variant of the Wiedemann. This improvement may speed up the classical Wiedemann algorithm by up to 64 times.

5. Conclusion

This project ambitiously started with the aim to code efficient versions of both the Quadratic Sieve and the General Number Field Sieve. It then became clear that writing an efficient version of the Quadratic Sieve alone required several small improvements. Experiments showed that these improvements were essential: the algorithm improved by a factor of 1000 in the course of the project. Nonetheless, I am satisifed about the result achieved and the copious amount of knowledge assimilated and I am looking to further improve the algorithm in the coming months.