


Approval Sheet

Title of Thesis: Integer Factoring Using Newton's Method in the Dyadics

Name of Candidate: Leslie McAdoo

Master of Sciences, Computer Science 2019

Thesis and Abstract Approved: 

Samuel Lomonaco

Professor

Computer Science and Electrical Engineering

Date Approved: 4-29-2019

ABSTRACT

Title of thesis: **INTEGER FACTORING USING
NEWTON'S METHOD IN THE DYADICS**

Leslie N McAdoo III, Master of Sciences, 2019

Thesis directed by: Professor Samuel Lomonaco
Department of Computer Science

Here we expand upon the work done by Lomonaco (2013) for division of Generic Integers (an abstraction of Dyadic Integers), by extending Newton's Method for the Dyadics to find the roots of functions of Generic Integers, enabling us to compute the quotient of Generic Integers. We use this Division Algorithm as a basis for an implementation of Lomonaco's Boolean Factoring Algorithm, using a Default Representation of Generic Integers. We then implement the Algorithm using a Matrix Representation of Generic Integers, suggested by Lomonaco as a 'Top-Down Approach' that could lead to more efficient storage and usage. This led to a much improved Variant of the Boolean Factoring Algorithm, drastically decreasing the runtime. Finally, we improved the Algorithm further by using a 'Generic Prime,' which uses the Matrix Representation of Generic Integers more intelligently for Factoring Applications. Using Generic Primes for the Boolean Factoring Algorithm, we were able to factor semiprimes with 54 bits, though the time that it took was still not competitive with industry standards.

INTEGER FACTORING USING NEWTON'S METHOD
IN THE DYADICS

by

Leslie N McAdoo III

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Master of Sciences
2019

Advisory Committee:
Professor Samuel Lomonaco, Chair/Advisor
Professor Alan Sherman
Professor Kostas Kalpakis

Acknowledgments

I would like to thank all of the people that have been alongside me for any part of this process, whether it was your encouragement, support, help, patience, proofreading or any number of other things, you made this possible. Especially big thanks go to my wife Emma, my advisor Dr. Samuel Lomonaco, and my committee.

Specifically, I would like to thank Dr. Lomonaco further for his initial ideas for this research, and his guidance throughout this process. One particular piece of advice that proved key to developing greater understanding of the Boolean Factoring Algorithm was the suggestion to use the ‘Top-Down Approach’ to represent Generic Integers (here referred to as the Matrix Representation).

Table of Contents

List of Figures	v
1 Introduction	1
1.1 Overview	1
1.2 Dyadic Integers	1
1.3 Generic Integers	3
1.3.1 Generic Arithmetic	4
1.3.2 Top-Down Approach	5
2 Newton's Method	8
2.1 Newton's Method for Dyadics	8
2.2 Finding Inverses of Dyadic Integers	9
2.3 Division of Dyadic Integers	11
2.3.1 Proof of Correctness	12
2.3.2 Observations	13
3 Factoring Integers with Newton's Method for Generics	17
3.1 Approach	17
3.2 Modifications to the Factoring Algorithm	20
3.2.1 Removing Generic Multiplications	20
3.2.2 Temporary Storage of Boolean Expressions	20
3.2.3 Matrix Representation of Generic Integers	21
3.2.4 Using 'Generic Primes'	21
4 Boolean Factoring Algorithm Performance	25
4.1 Testing Methodology	25
4.2 Algorithm Runtime	27
4.2.1 Theoretical Analysis of Runtime	27
4.2.2 Comparison of the Four Variations of the BF Algorithm	28
4.2.3 Time to Generate the Generic Integer	31
4.3 Algorithm Space Usage	32
5 Conclusions	35

6	Open Questions	37
6.1	Comparing Lopsided Division with Newton's Method	37
6.2	Potential Extension to Quantum Computing	37
	Bibliography	39

List of Figures

4.1	Comparison of the Variations of the BF Algorithm	29
4.2	Semi-log Comparison of the Variations of the BF Algorithm	30
4.3	Comparison of Matrix BF Variants	30
4.4	Time to Generate the Generic Prime	32
4.5	Space Used by the Generic Prime	33
5.1	Space Used by the Generic Integer vs. Bits in the Integer to Factor . .	36

Introduction

Overview

In this paper we expand on the work of Lomonaco [1] in the following manner: We first present Newton's method to find Inverses of Dyadic Integers, and extend this method into one to divide Dyadic Integers. We then apply this method to Generic Integers, and present a Factoring Algorithm with Newton's Method at the core. Four different variants of this Algorithm were implemented, and we analyze the performance of all four variants. Finally we present some open questions and potential directions for further research.

Dyadic Integers

The foundations of the findings for this paper have their roots in the Dyadic Integers (the p -adic Integers for $p = 2$ [2] [3]). The term 'Dyadic Integer' can have various definitions depending on the context in which it is used, but this paper will use the terms and definitions laid out in this section. A full analysis of the Dyadics will not be given.

For the purposes of this paper, the Dyadic Integers, in a naive sense, are the

numbers whose ‘Infinite 2’s Complement’ representation in Binary does not have anything to the right of the decimal point. In this paper, the Dyadic Integers will be denoted by $\mathbb{Z}_{(2)}$, distinct from \mathbb{Z} , the traditional Ring of Rational Integers (or just Integers), and from $\mathbb{Z}_2 = \mathbb{Z}/2\mathbb{Z}$, the Quotient Ring with two Integers isomorphic to \mathbb{F}_2 . Thus, we can think of the Dyadic Integers as an infinite string of Dyadic digits, or ‘bits.’ For convenience, we will think of this string as going from right to left (so the rightmost bit is the ‘first’ bit).

It should be noted that \mathbb{Z} is a subring of $\mathbb{Z}_{(2)}$, since elements in \mathbb{Z} correspond to the elements of $\mathbb{Z}_{(2)}$ whose bits are constant after finitely many bits. Dyadic Integers whose bits are all eventually 0 correspond with non-negative Integers in \mathbb{Z} , and Dyadic Integers whose bits are all eventually 1 correspond to negative Integers in \mathbb{Z} . However, there are Dyadic Integers that have no corresponding Integer in \mathbb{Z} (those whose bits form an infinitely repeating sequence, or those whose sequence of bits has no limit and also does not repeat), so we have $\mathbb{Z} \subset \mathbb{Z}_{(2)}$.

As Lomonaco [1] mentions, the quotient of two odd Dyadic Integers (Dyadic Integers whose first bit is a 1) is also a Dyadic Integer, even if the quotient of the corresponding Integers in \mathbb{Z} is not also in \mathbb{Z} . This fact is especially important for the findings of this paper because it directly implies that all odd Dyadic Integers have an inverse in Dyadic Integers.

It should be noted that since a Dyadic Integer has infinitely many bits, it is often useful to truncate the Dyadic Integer to only a finite number of bits (at the cost of precision). This truncated Dyadic Integer can be thought of as an Integer (in \mathbb{Z}) represented in Binary (more specifically, in 2’s complement). Arithmetic

operations can be performed on a Truncated Dyadic Integer as if it were a Binary Integer, taking care to deal with negative numbers and overflow in 2's complement appropriately.

Generic Integers

Lomonaco [1] presents the idea of a ‘Generic Integer’. On a high level, Generic Integers are an abstraction of Dyadic Integers, where each bit of the Generic Integer, instead of being a 0 or 1, is a Boolean Expression of n variables for some n . Intuitively, this means that for each instantiation of the n Boolean variables (here an instantiation is simply an assignment of the value 0 or 1 to each Boolean variable), of which there are 2^n , the bits of the Generic Integer will take on values of 0 or 1, leaving us with a (potentially infinite) sequence of bits, or a Dyadic Integer. We will refer to these Dyadic Integers yielded by the Generic Integer under the 2^n instantiations of the n Boolean variables as the Integers represented by the Generic Integer (this may lead to some confusion, but in general, we will only be dealing with Generic Integers with a finite length, so the Dyadic Integers represented by this finite sequence of bits correspond to Integers in \mathbb{Z}).

Example 1.1 (Default Odd Generic Integer with 3 bits)

The naive way to construct an Odd Generic Integer with n bits is to simply let the first bit be a 1, and then let the remaining $n - 1$ bits be a different Boolean variable. So to construct an Odd Generic Integer G with 3 bits, we have:

$$G = x_1, x_0, 1$$

Now since we are using 2 Boolean Variables, there exist 2^2 Dyadic Integers that G represents. For instance, under the instantiation $x_0 = 1, x_1 = 0$, we have:

$$G = 0, 1, 1 = 5 \in \mathbb{Z}$$

Intuitively, we constructed G in such a way that it represents the first 2^{n-1} odd Integers, where n is the number of bits of G . It is trivial to confirm that the four Boolean Variable Instantiations yield $G \in \{1, 3, 5, 7\}$. This construction will be referred to as the ‘Default’ Construction (or Representation) of Generic Integers.

Generic Arithmetic

While it makes sense under certain caveats to apply standard arithmetic operations to truncated Dyadic Integers, it does not make as much sense to apply the same operations to Generic Integers (for example, subtraction of Boolean Expressions is not defined, etc.). Lomonaco [1] presents Generic Integer Addition, Negation, and Multiplication, and presents a Division Algorithm (‘Lopsided Division’) for Generic Integers as well, in terms of fundamental Boolean operations, so they make sense for Generic Integers. As one would hope, these arithmetic operations correspond to the same operation between the Integers each Generic Integer represents given a particular instantiation (i.e. to apply an instantiation of the Boolean variables to two Generic Integers and then add the resulting Integers gives the same result as performing Generic Addition on the Generic Integers and applying the instantiation

to the resulting Generic Integer).

Top-Down Approach

Lomonaco [1] details another way to think about Generic Integers. If each bit e_i in a Generic Integer is a Boolean Expression of n different Boolean Variables, then e_i can be expressed as a linear combination of the 2^n Boolean minterms, where the coefficients are drawn from \mathbb{F}_2 , and addition can be thought of as Logical XOR. So $e_i = \sum_{j=0}^{2^n-1} c_{ij} \cdot m_j$, where $c_{ij} \in \{0, 1\}$, and m_j are the Boolean minterms.

For convenience, we will order the minterms in the following way: Let $k \in \mathbb{Z}_{2^n-1}$, $k = \sum_{i=0}^{n-1} 2^i \cdot k_i$ where $k_i \in \{0, 1\}$ (so k_i is the i -th bit in the binary representation of k). Then $m_k = \prod_{i=0}^{2^n-1} x_i^{k_i}$, where $x_i^0 = \overline{x_i}$ and $x_i^1 = x_i$. As an example, the minterms with 2 Boolean variables x_1, x_2 are $\{\overline{x_1} \cdot \overline{x_2}, \overline{x_1} \cdot x_2, x_1 \cdot \overline{x_2}, x_1 \cdot x_2\}$, ordered from m_0 to m_3 . It should be noted that each of these minterms represent a unique instantiation of the n Boolean variables (i.e. there is a unique instantiation of the n Boolean variables that gives $m_i = 1$).

Given this ordering of the minterms, we can redefine the way we construct Generic Integers as follows: Given n Boolean variables, there are 2^n different Integers that it can represent. Denote the 2^n Integers that can be represented with $\{g_0, g_1, \dots, g_{2^n-1}\}$. We seek to have the Generic Integer represent g_i under the instantiation given by the minterm m_i . To accomplish this goal, each bit in the Generic Integer will be the sum of the minterms that correspond to whichever Integers have a 1 in that bit in their binary representation. We can think of this as a binary

matrix where each element in the i -th row is a bit in the binary representation of g_i , and the bits of the Generic Integer are given by the multiplication of the matrix on the left by the row vector of minterms.

Example 1.2 (Matrix Construction of Generic Integers)

We will now construct the same Generic Integer G from Example 1.1 using this Matrix Construction. We note that because there are $n = 2$ Boolean Variables (here x_1, x_2), the $2^n = 4$ minterms are given by $\{\overline{x_1} \cdot \overline{x_2}, \overline{x_1} \cdot x_2, x_1 \cdot \overline{x_2}, x_1 \cdot x_2\}$ as above.

We seek for G to represent the Integers $\{1, 3, 5, 7\}$. Writing these Integers in Binary (adding extra zeroes to the left to make them all the same number of bits) gives us $\{001, 011, 101, 111\}$, so our Binary Matrix is given by:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Left multiplication of this Matrix by the Boolean minterms gives our Generic Integer:

$$\begin{pmatrix} m_0 & m_1 & m_2 & m_3 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} m_2 \oplus m_3 & m_1 \oplus m_3 & m_0 \oplus m_1 \oplus m_2 \oplus m_3 \end{pmatrix} = \begin{pmatrix} x_2 & x_1 & 1 \end{pmatrix}$$

which is what we expect based on the Default Construction.

This way to construct a Generic Integer gives flexibility above the Default Construction, which only allows a Generic Integer with n Boolean Variables to represent the first 2^n odd Integers. Using the Matrix Construction (or ‘Matrix Representation’), the Generic Integer can be adjusted to represent an arbitrary set of 2^n Integers, which will be crucial in our construction of the Generic Prime in §3.2.4.

Newton's Method

Newton's method is a common way of finding roots of polynomials using linear approximations. The use of Newton's method for Integers, or real numbers, is well understood. We sought to use Newton's method initially to find the inverses of Dyadic Integers, with the hope that such a process could be generalized to division of Dyadics, and extended to Generic Integers.

Newton's Method for Dyadics

Given a Function F of a Polynomial $u \in \mathbb{Z}_{(p)}[x]$ (a polynomial whose coefficients lie in the p -adic Integers), with p a prime, we seek an approximate root of the function F ; namely, we seek an approximation of u such that $F(u) = 0$.

Newton's Method is an iterative process, one that begins with an initial approximation, and iteratively improves the approximation until it (hopefully) converges. In the Dyadic Integers, our initial approximation will be the first bit (which can be easily computed using arithmetic (mod 2)), and we will improve our approximation by obtaining another bit at every iteration. We will denote the bit we obtain (on the k -th iteration) by u_k , and we will denote our approximation of u at the beginning of the k -th iteration by $u^{(k)}$. Notice that because we are working with

Dyadic Integers (specifically Truncated Dyadic Integers), and u_k is the k -th bit, we have $u^{(k+1)} = 2^k \cdot u_k + u^{(k)}$. We begin with our initial approximation of $u^{(1)} = u_0$. We also define the function $\Phi_r(x) \equiv x \pmod{r}$, with $0 \leq \Phi_r(x) < r$. With all of the definitions out of the way, we have the formula [4]:

$$u_k = -\frac{\Phi_2\left(\frac{F(u^{(k)})}{2^k}\right)}{\Phi_2(F'(u^{(1)}))} \quad (2.1)$$

Finding Inverses of Dyadic Integers

We would like to use the method described in §2.1 to find a Dyadic Integer, a constant polynomial with dyadic coefficients. It turns out that, without any tinkering, this method works to find inverses of Dyadic Integers. Consider the function $F(u) = a \cdot u - 1$. Clearly the root of this function is $u = a^{-1}$. If we treat u as a Dyadic Polynomial (a trivial Polynomial with only a constant term, or a Dyadic Integer), then we can follow Equation 2.1.

To clearly distinguish between Dyadic Integers and traditional Integers, when writing out a Dyadic Integer, we will use the notation $(\dots b^n b^{n-1} \dots b^1 b^0)_2$ where b^i is the i -th bit (note the subscript 2). For Truncated Dyadic Integers, we will only give the finite number of bits that we have. This is in contrast to traditional Integers, which will be written in base-10 representation, with no subscript.

Example 2.1 (Finding Inverse of the Dyadic Integer)

Let $F(u) = 3u - 1$. To begin, we compute the first bit of u . Let u_0 be the first bit of u . Then $F(u_0) = 3 \cdot u_0 - 1 \equiv 0 \pmod{2}$. Thus we have $u_0 = u^{(1)} = 1$, and we

also have $F'(u) = 3$.

As an aside, we note the quantity $\Phi_2(F'(u^{(1)})) = \Phi_2(3) = 1$ is the denominator for many of these computations. For simplicity, we omit division by 1.

Per Equation 2.1, we seek $u_1 = \Phi_2(\frac{F(u^{(1)})}{2^1}) = \Phi_2(\frac{3 \cdot 1 - 1}{2}) = \Phi_2(\frac{2}{2}) = \Phi_2(1) = 1$

Given u_1 , we can compute $u^{(2)} = u_1 u^{(1)} = (11)_2 = 3$

Similarly, we have $u_2 = \Phi_2(\frac{F(u^{(2)})}{2^2}) = \Phi_2(\frac{3 \cdot 3 - 1}{4}) = \Phi_2(\frac{8}{4}) = \Phi_2(2) = 0$

So $u^{(3)} = u_2 u^{(2)} = (011)_2 = 3$

$$u_3 = \Phi_2(\frac{F(u^{(3)})}{2^3}) = \Phi_2(\frac{3 \cdot 3 - 1}{8}) = \Phi_2(\frac{8}{8}) = \Phi_2(1) = 1$$

$$u^{(4)} = u_3 u^{(3)} = (1011)_2 = 11$$

$$u_4 = \Phi_2(\frac{F(u^{(4)})}{2^4}) = \Phi_2(\frac{3 \cdot 11 - 1}{16}) = \Phi_2(\frac{32}{16}) = \Phi_2(2) = 0$$

$$u^{(5)} = u_4 u^{(4)} = (01011)_2 = 11$$

Here we stop computation, noting two facts: First, we appear to have a pattern in the digits of 3^{-1} in the Dyadics, suggesting (but not proving) that $3^{-1} = (\dots 10101011)_2 = (\overline{1011})_2$. Second, we note the pattern in the last step of computation as the algorithm proceeds. We have $u_1 = \Phi_2(1)$, $u_2 = \Phi_2(2)$, $u_3 = \Phi_2(1)$ and $u_4 = \Phi_2(2)$. We observe that these numbers $(1, 2, 1, 2, \dots)$ seem to form a cycle that is the same as multiplication by $2^{-1} = 2$ in \mathbb{Z}_3 (as $1 \cdot 2 \equiv 2 \pmod{3}$ and $2 \cdot 2 \equiv 1 \pmod{3}$).

We will further explore both of these notes in later sections, as well as formally prove when to stop computation of digits in the inverse of a Dyadic Integer, instead of just hoping that the pattern we observe will continue.

Division of Dyadic Integers

Once we understood how to compute Dyadic Inverses with Newton's Method, it was natural to wonder if such a method could be used to compute division of two Dyadic Integers. If we can compute $\frac{1}{a} = a^{-1}$ in the Dyadics using this method, could we extend it to compute $\frac{b}{a} = b \cdot a^{-1}$? Simple examination of the method in §2.2 suggests that by taking $F(u) = au - b$ (which has a root at $u = b \cdot a^{-1}$), we can find $\frac{b}{a}$ without any extra changes.

Example 2.2 (Division of 15 by 7 with Newton's Method)

As suggested, let $F(u) = 7u - 15$. We note again that $u^{(1)} = u_0 = 1$, that $F'(u) = 7$, and that $\Phi_2(F'(u^{(1)})) = \Phi_2(7) = 1$. We then proceed as expected:

$$u_1 = \Phi_2\left(\frac{F(u^{(1)})}{2^1}\right) = \Phi_2\left(\frac{7 \cdot 1 - 15}{2}\right) = \Phi_2\left(\frac{-8}{2}\right) = \Phi_2(-4) = 0$$

$$u^{(2)} = u_1 u^{(1)} = (01)_2 = 1$$

$$u_2 = \Phi_2\left(\frac{F(u^{(2)})}{2^2}\right) = \Phi_2\left(\frac{7 \cdot 1 - 15}{4}\right) = \Phi_2\left(\frac{-8}{4}\right) = \Phi_2(-2) = 0$$

$$u^{(3)} = u_2 u^{(2)} = (001)_2 = 1$$

$$u_3 = \Phi_2\left(\frac{F(u^{(3)})}{2^3}\right) = \Phi_2\left(\frac{7 \cdot 1 - 15}{8}\right) = \Phi_2\left(\frac{-8}{8}\right) = \Phi_2(-1) = 1$$

$$u^{(4)} = u_3 u^{(3)} = (1001)_2 = 9$$

$$u_4 = \Phi_2\left(\frac{F(u^{(4)})}{2^4}\right) = \Phi_2\left(\frac{7 \cdot 9 - 15}{16}\right) = \Phi_2\left(\frac{48}{16}\right) = \Phi_2(3) = 1$$

$$u^{(5)} = u_4 u^{(4)} = (11001)_2 = 25$$

$$u_5 = \Phi_2\left(\frac{F(u^{(5)})}{2^5}\right) = \Phi_2\left(\frac{7 \cdot 25 - 15}{32}\right) = \Phi_2\left(\frac{160}{32}\right) = \Phi_2(5) = 1$$

$$u^{(6)} = u_5 u^{(5)} = (111001)_2 = 57$$

$$u_6 = \Phi_2\left(\frac{F(u^{(6)})}{2^6}\right) = \Phi_2\left(\frac{7 \cdot 57 - 15}{64}\right) = \Phi_2\left(\frac{384}{64}\right) = \Phi_2(6) = 0$$

$$u^{(7)} = u_6 u^{(6)} = (0111001)_2 = 57$$

$$u_7 = \Phi_2\left(\frac{F(u^{(7)})}{2^7}\right) = \Phi_2\left(\frac{7 \cdot 57 - 15}{128}\right) = \Phi_2\left(\frac{384}{128}\right) = \Phi_2(3) = 1$$

$$u^{(8)} = u_7 u^{(7)} = (10111001)_2 = 185$$

At this point, we see a pattern similar to Example 2.1, where we have a cycle in the last step on computation obtained by the repeated multiplication in \mathbb{Z}_7 by $2^{-1} = 4$ ($3 \cdot 4 \equiv 5 \pmod{7}$, $5 \cdot 4 \equiv 6 \pmod{7}$, and $6 \cdot 4 \equiv 3 \pmod{7}$). This gives us the suggestion that $\frac{15}{7} = 15 \cdot 7^{-1} = \overline{0111001}$ in the Dyadics, which is what we would have gotten had we computed 7^{-1} first and then multiplied by 15. The proof of correctness for these results follows.

Proof of Correctness for Division in the Dyadics with Newton's Method

Definition 2.3 (Definition of Correct Bits in Quotient)

A Quotient $b \cdot a^{-1}$ (alternatively, $\frac{b}{a}$) in the Dyadics has n correct digits iff the first n digits in the expression $\frac{b}{a} \cdot a - b$ are 0.

Algorithm 2.1 Newton's Method for Dyadic Division

Input: a, b , Odd Dyadic Integers

Input: $n \in \mathbb{N}$ the desired number of correct digits of $b \cdot a^{-1}$

$u \leftarrow 1$

for $k \leftarrow 1$ **to** $n - 1$ **do**

$u_k \leftarrow \Phi_2\left(\frac{u \cdot a - b}{2^k}\right)$

if $u_k = 0$ **then**

$u \leftarrow 2^k + u$

end if

end for

return u

Output: $b \cdot a^{-1}$ such that the first n bits are correct

Lemma 2.4 (Algorithm 2.1 is Correct)

To show: The above method for computing division between Odd Dyadic Integers b

and a is correct for at least n bits (after n iterations), by induction on the number of iterations (denoted k). Note that after n iterations, the above method returns $u^{(n)}$, and that for simplicity, the “first” iteration refers to the initial declaration of u (with this caveat, the first bit is returned after the first iteration).

Base Case ($k = 1$): This is trivially true. The algorithm returns $u^{(1)} = 1$. Assuming a and b are both odd, the first bit of $a \cdot u^{(1)} - b = a - b$ is 0.

Inductive Step (Assume true for k): By assumption, we have that the first k bits of the quotient $u^{(k)} = b \cdot a^{-1}$ are correct. The $k + 1^{th}$ iteration of the algorithm examines the last bit of the expression $\frac{u^{(k)} \cdot a - b}{2^k}$ (since we know that $u^{(k)} \cdot a - b$ has 0 in the last k bits, the division by 2^k amounts to just a shift). Now that we have this last bit (u_k), there are two cases to consider. The first case is that $u_k = 0$. In this case, we are done, since $u^{(k)} \cdot a - b$ has 0 in at least the first $k + 1$ bits, and the algorithm returns $u^{(k+1)} = u^{(k)}$. If, however, $u_k = 1$, $u^{k+1} = u_k u^{(k)} = 2^k + u^{(k)}$. Then $u^{(k+1)} \cdot a - b = (u^{(k)} + 2^k) \cdot a - b = (u^{(k)} \cdot a - b) + 2^k \cdot a$. We know, by assumption, that the $k + 1^{th}$ bit of both $u^{(k)} \cdot a - b$ and $2^k \cdot a$ are 1, so when they are summed, the $k + 1^{th}$ bit of their sum is 0. Thus, $u^{(k+1)}$ has $k + 1$ correct digits, and by induction, the algorithm is correct.

□

Observations

Intuition: Intuitively, Algorithm 2.1 works as follows: It maintains an n -bit approximation of the quotient ($u^{(n)}$), and checks the accuracy of the $n + 1^{th}$ bit of

$\frac{b}{a} \cdot a - b$. If this bit is zero, our approximation is good enough, and we can move to the next bit, should we desire. However, if it is not zero, we adjust our quotient to account for this error. Note that this is done in a manner cannot change the first n bits of $\frac{b}{a} \cdot a - b$, so our approximation only gets better over time.

Simplifying Computation: Another observation, hinted at in Example 2.1, deals with the notion of what we are calling the “remainder” in these computations. Given an approximation u of the true quotient of b and a , the remainder is the error associated with that approximation ($a \cdot u - b$). It turns out that we can simplify Algorithm 2.1 to only perform computations on this remainder. Intuitively, we can see from Lemma 2.4 that Algorithm 2.1 re-computes ($a \cdot u - b$) at every step, dividing by a higher power of 2 each time to obtain the next bit in the sequence. However, we note that the remainder after the $i + 1$ -th step is given by

$$\frac{(a \cdot u^{(i+1)} - b)}{2^{i+1}} = \frac{(a \cdot (u_i \cdot 2^i + u^{(i)}) - b)}{2^{i+1}} = \frac{(a \cdot u^{(i)} - b) + a \cdot u_i \cdot 2^i}{2^{i+1}} = \frac{\frac{(a \cdot u^{(i)} - b)}{2^i} + a \cdot u_i}{2}$$

where $\frac{(a \cdot u^{(i)} - b)}{2^i}$ is the remainder after the i -th step.

This observation leads to a simplified, but equivalent (further proof left to the reader), Algorithm, shown in Algorithm 2.2. Especially noteworthy is the fact that Algorithm 2.2 only performs a Generic Addition in each iteration, instead of a Generic Multiplication.

Termination of Newton’s Method in the Dyadics: While Algorithms 2.1 and 2.2 give n correct bits for a given value of n , this by itself is not enough to prove that we can stop after a given iteration and make claims about repeating patterns

Algorithm 2.2 Modified Newton's Method for Dyadic Division

Input: a, b , Odd Dyadic Integers

Input: $n \in \mathbb{N}$ the desired number of correct bits of $b \cdot a^{-1}$

$u \leftarrow 1$

$remainder \leftarrow \frac{a \cdot u - b}{2}$

for $k \leftarrow 1$ **to** $n - 1$ **do**

$u_k \leftarrow remainder \pmod{2}$

$remainder \leftarrow \frac{(remainder + a \cdot u_k \cdot 2^k)}{2}$

$u \leftarrow u_k \cdot 2^k + u$

end for

return u

Output: $b \cdot a^{-1}$ such that the first n bits are correct

of the solution, as we did in Examples 2.1 and 2.2. The intuition of when to stop comes in large part from §2.3.2, and the notion of computation with remainder. In both of the examples, we see that the remainder forms a cycle, and we can see from Algorithm 2.2 that since the remainder for a particular step can be computed only with the remainder from the previous step, once the remainder forms a cycle, it cannot break that cycle. This leads to the following Lemma (proof left to the reader):

Lemma 2.5 (Termination of Newton's Method for Division in the Dyadics)

Given b and a Odd Dyadic Integers, and an approximation u of $b \cdot a^{-1}$, the remainder is given by $a \cdot u - b$. The termination conditions for Newton's Method for Division in the Dyadics are as follows:

1. If the remainder ever reaches 0, the approximation u is exactly correct, and nothing more needs to be done
2. If a cycle is found in the sequence of remainders, the algorithm can be stopped, and the bits of u given by the terms in the cycle will repeat

Equivalence of ‘Lop-Sided’ Division and Newton’s Method in the Dyadics: The division Algorithm defined by Lomonaco [1] (‘Lopsided Division’) defines Division for Dyadic Integers (or Generic Integers) in terms of fundamental Boolean operations. Newton’s Method for Division defines Division in terms of the Arithmetic Operations for Dyadics. Since they are both division algorithms, however, it is natural to wonder about the connection between the two Algorithms. While we give no formal proof of equivalence, a high-level sketch of the similarities and differences of the Algorithms is given below.

The general concept of the Algorithms is the same: Each Algorithm obtains another bit of the quotient with every iteration. However, each Algorithm obtains this bit in what seems to be a very different way. Lopsided Division maintains two different intermediate Dyadic (or Generic) Integers, which combine with the inputs to obtain the next bit of the quotient, performing only logical operations. Newton’s Method instead maintains only one intermediate Integer, performing an arithmetic operation (an addition) to obtain the next bit of the quotient.

At a higher level, we note that the two Division Algorithms approach the ‘carries’ of arithmetic operations differently. Newton’s method allows the carries to propagate fully (i.e. the arithmetic operation has been completed). Lopsided Division, instead of propagating the carries at each step, only propagates the carries as far as is necessary to obtain a bit of the quotient. It then keeps the carries as an intermediate calculation, and combines this intermediate value with the carries generated in the next iteration, never fully letting the carries propagate until the end. It remains an open question which approach is faster (see §6.1 for details).

Factoring Integers with Newton's Method for Generics

Approach

Lomonaco [1] lays out a procedure by which Lopsided Division of an odd Dyadic Integer a by a Generic Integer can lead to a System of Boolean Variables. Intuition would suggest that we could follow a similar procedure using Newton's Method for Division with Generic Integers.

Example 3.1 (Factoring 21 using Newton's Method)

Let $a = 21_{10} = 10101_2 = 7 \cdot 3$, and $x = x_2, x_1, 1$ be a Generic Integer. We observe that since $7_{10} = 111_2$, we expect a system of Boolean Equations with $x_2 = x_1 = 1$ as a solution. (We also expect $x_2 = 0, x_1 = 1$ to be a solution, but more on that later). As an aside, we note that using Newton's Method for Division requires a subtraction, but rather than explicitly defining a subtraction for generics, we simply add the negative (so we would add $-21 = \dots 11101011_2$ in this example).

Using Alg. 2.1, and letting $u_0 = u^{(1)} = 1$, we have the following:

$$u_1 = \Phi_2\left(\frac{u * x + (-21)}{2}\right) = \Phi_2(0, \dots, 1, 1, 1, 1, x_2, \overline{x_2}, \overline{x_2}, x_1) = x_1$$

$$u^{(2)} = x_1, 1$$

$$u_2 = \Phi_2\left(\frac{u * x + (-21)}{4}\right) = \Phi_2(x_1 x_2, \overline{x_1 x_2}, \dots, \overline{x_1 x_2}, \overline{x_1 x_2 \oplus x_1 \oplus x_2}, \overline{x_2}) =$$

$$u^{(3)} = \overline{x_2}, x_1, 1$$

At this point, we will stop the example (because we have enough to find the solution), and focus on how to construct the solution from x and u .

As stated above, we are looking for a system of Boolean equations whose solution yields a factorization of the number a . Intuitively, we need to find some constraints on the system based on the computation that we have already done. Taking a closer look at the computation for u_2 above, we note that the Generic Integer $(x_1x_2, \overline{x_1x_2}, \dots, \overline{x_1x_2}, \overline{x_1x_2 \oplus x_1 \oplus x_2}, \overline{x_2})$ is our most recent approximation of 0 (since it is $a - x \cdot u$, with increasingly better approximations of u). As such, if $u^{(3)}$ were a factor of a , we should have the quantity $a - x \cdot u^{(3)} = 0$, and so all of the remaining bits in our approximation should be zero. And therein lie our constraints on the system. If we can find an assignment for all of the Boolean variables such that the remaining bits in our approximation are 0, we are done, and have found a factor.

Before we use this intuition on the above example, a caveat should be made. Because the algorithm makes use of infinite 2's complement (which has infinitely many bits), and we choose some limit of finite precision, the most significant bit has a rounding error in it. We can avoid this by simply not using the most significant bit in our satisfiability problem.

That caveat aside, if we observe the above Generic Integer, we note that a conceptually simple way to make sure that all bits are 0 is to multiply (logical and) all of the negated expressions together, and find solutions that make that expression

True. Doing that in the example gives the expression x_1x_2 , which has the solution $x_1 = 1, x_2 = 1$. Plugging this solution into x and u gives us $x = (x_2, x_1, 1) = (1, 1, 1) = 111_2 = 7_{10}$ and $u = (\overline{x_2}, x_1, 1) = (0, 1, 1) = 011_2 = 3_{10}$, which is the correct factorization. This algorithm is formalized in Algorithm 3.1.

Algorithm 3.1 Factoring Integer's Using Newton's Method

Input: $a \in \mathbb{Z}$ odd composite

Input: x a Generic Integer

```

 $u \leftarrow 1$ 
 $k \leftarrow 1$ 
 $\Phi \leftarrow \emptyset$ 
while  $\Phi = \emptyset$  do
   $rem \leftarrow a - u \cdot x$ 
   $\Phi \leftarrow \text{satisfy}(rem)$ 
  if  $\Phi = \emptyset$  then
     $u_k \leftarrow \Phi_2(rem)$ 
     $u \leftarrow u + u_k \cdot 2^k$ 
  end if
end while
 $x = \Phi(x)$ 
 $u = \Phi(u)$ 

```

Output: x, u factors of a

Note: Here ‘satisfy(r)’ is a function that returns an instantiation Φ of all the Boolean variables in r such that each bit in the Generic Integer is 0, if such an instantiation exists, and the empty set if none exists.

Number of Variables: An obvious design parameter relates to the number of variables we use to represent the generic divisor x . In general, to factor a number N (that isn't prime) with $\lceil \log_2(N) \rceil = n$ bits, we know that at least one of its factors can be represented by $\lceil \frac{n}{2} \rceil$ bits. Thus, we need only use that many. Certainly the possibility exists for this to be more bits than we need, (for example when the number has a small factor), but to guarantee that we will find a factor, we should use $\lceil \frac{n}{2} \rceil$ bits.

Modifications to the Factoring Algorithm

As expected, factoring large integers using Algorithm 3.1 is slow at best, and computationally infeasible when the integers are large enough. Hoping to reduce the computational barrier for factoring large integers at least in part, several modifications to the original Algorithm were made.

Removing Generic Multiplications

Since the Basis of Algorithm 3.1 is Algorithm 2.1, it follows that we should be able to enhance Algorithm 3.1 in the manner described in §2.3.2. Note that in the case of the Generic Integers, this changes the Algorithm from needing to perform a Generic Multiplication to needed only to perform a Generic Addition (and some shifts). While we didn't emphasize the potential speed increase of such modifications in §2.3.2, when the algorithm is using Generic Integers, the speed increase is significant. A Generic Multiplication is, in a naive sense, a sequence of Generic Additions. To replace a Generic Multiplication with a single Generic Addition greatly speeds up the Algorithm, as shown in §4.

Temporary Storage of Boolean Expressions

One of the problems with Algorithm 3.1, even with multiplications removed, is that when the integer to factor gets large enough, the Boolean expressions used in the middle of the computation become too large to be feasible. When they are large and complicated enough, even conceptually simple operations like Logical XOR become

very costly, and these operations are the foundation of the Algorithm. To combat this, we thought that when an expression became large enough, we could save it into a table, and substitute it with a separate variable. Further computations could then be performed on this simpler variable, and when it came to the satisfiability problem, we could substitute the original expressions back in, and solve from there. However, after many different variations of the re-substitution, it appeared to be too computationally expensive, and the time savings were never realized.

Matrix Representation of Generic Integers

As mentioned in §1.3.2, another way to store Generic Integers is to store a Binary Matrix as the coefficients for the Boolean minterms in each expression. An alternative way of representing this minterm expansion of the Generic Integer is to treat each column of the Binary Matrix as a 2^n bit number. Then instead of doing logical operations on the Boolean Expressions, we can use perform the same logical operations (bitwise) on the number representations, making use of the speed and simplicity of bitwise operations. It is left to the reader to show that bitwise operations on the number representation of the minterm expansion are equivalent to logical operations on the Boolean Expressions.

Using ‘Generic Primes’

Intuitively, this method of Factoring seems similar in a sense to Trial Division, since Division by the Generic Integer represents Division by the all of the Integers

that the Generic Integer can represent. As noted in §1.3.2, however, the Integers that a Generic Integer can represent can be arbitrarily selected. Thinking of this method of Factoring as similar to Trial Division suggests that a easy optimization for the algorithm would be to simply have the Generic Integer (with n Boolean Variables) represent the first 2^n primes.

To do this, we introduce the notion of a so-called ‘Generic Prime,’ alluded to in §1.3.2.

Example 3.2 (Construction of Generic Primes)

Let the number of Boolean Variables $n=2$. Let G_1 be a Generic Odd Integer using the Default Representation, and G_2 be a Generic Prime with with the same number of Boolean Variables.

We note that the Integers represented by G_1 are $\{1, 3, 5, 7\}$, since by the Default Construction we have $G_1 = x_2, x_1, 1$. We seek to have G_2 represent the first $2^n = 4$ primes, $\{3, 5, 7, 11\}$.

To construct G_2 , we note that the minterms for the 2^n Boolean Variables are the same as those in §1.2, $\{\overline{x_1} \cdot \overline{x_2}, \overline{x_1} \cdot x_2, x_1 \cdot \overline{x_2}, x_1 \cdot x_2\}$. The binary matrix is then given by

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

and so the bits of the Generic Integer are given by

$$\begin{aligned}
& \begin{pmatrix} m_0 & m_1 & m_2 & m_3 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \\
&= \begin{pmatrix} m_3 & m_1 \oplus m_2 & m_0 \oplus m_2 \oplus m_3 & m_0 \oplus m_1 \oplus m_2 \oplus m_3 \end{pmatrix} \\
&= \begin{pmatrix} x_1 \cdot x_2 & x_1 \oplus x_2 & \overline{x_1} \cdot \overline{x_2} \oplus x_2 & 1 \end{pmatrix}
\end{aligned}$$

The Generic Prime allows us the ability to factor larger Integers. This is intuitively seen by the fact that there are more bits in the Generic Prime, but more rigorously, because the Generic Prime represents all odd Prime Integers less than or equal to 11, allowing us to factor Integers less than or equal to $11^2 = 121$, whereas by the same logic we can only factor Integers up to 49 with the Generic Odd Integer.

In general, we wish to make a claim about how much additional Factoring Capability using a Generic Prime gives us over using a Generic Odd Integer. We seek to answer the question: ‘Given an Integer N with $O(n)$ bits, how many Integers does a Generic Prime have to represent, as opposed to a Generic Odd Integer, to factor N ?’ If we let $\pi(x) = |\{y : y \in \mathbb{Z}; y \leq x; y \text{ is prime}\}|$ (the number of Prime Integers less than or equal to x), then we can approximate $\pi(x)$ with $x/\ln(x)$ [5]. When trying to factor N with $O(n)$ bits, this means that our Generic Integer needs to represent all possible factors with $O(n/2) = O(n)$ bits. This requires that a

Generic Odd Integer must represent all of the Odd Integers with $O(n)$ bits, which is $O(2^n)$ Integers. However, a Generic Prime need only represent all Primes with $O(n)$ bits, which is given by $\pi(2^n) = 2^n/\ln(2^n) = O(2^n/n)$, so the Generic Prime must represent the first $O(2^n/n)$ Prime Integers, a significant saving over the Generic Odd Integer. This savings will be further explored in §4 and §5.

Boolean Factoring Algorithm Performance

Lomonaco [1] presented the Boolean Factoring Algorithm, which runs in exponential time and is not competitive with the current fastest Factoring Algorithms, but is based on the Lopsided Division Algorithm. We implemented the Boolean Factoring Algorithm using Newton's Method for Division, with some simplifying modifications, and sought to analyze the performance of all four implementations relative to each other.

We implemented and tested all of these variations of the BF Algorithm using Mathematica. The Generic Integers were implemented as lists (either of arbitrary Boolean Expressions or of arbitrary precision Integers, as detailed in §3.2.4)

Testing Methodology

To test the BF Variations, we generated random primes of various lengths (i.e. number of bits), and multiplied them together to form semiprimes of different lengths. Each of these semiprimes was factored with each algorithm three times, and the minimum time was recorded for each semiprime. If multiple semiprimes had the same length, the times for each semiprime of a given length were averaged, so that we could plot a single time for each length of semiprime. While this is

enough to give a rough estimate of the time complexity, not all Integers with the same number of bits take the same amount of time, nor even all semiprimes of the same length (i.e. this methodology is somewhat vulnerable to outliers, especially if only one semiprime of a given length was tested).

For the two Variants that used the Matrix Representation of the Generic Integer, one of the benefits of this Representation is that this Matrix need only be computed once, and then can be used to factor many Integers. When computing this Matrix, the Generic Integer needs to represent all possible options for a Factor of an Integer of a given Length (in general, to factor an Integer with n bits, it is sufficient to find all Factors with $n/2$ bits or fewer). It turns out that for the Generic Prime, the Matrix can be computed to contain the largest Primes necessary to factor an Integer of at most a given size. However, this is not the case for the Generic Odd Integer, because a Generic Odd Integer that is too large could represent the Integer to be factored, which means the process would give no information. To combat this, when factoring Integers of a certain size, we used the smallest Generic Integer that was capable of factoring the Integer (based on the size), so as to compare the two Variants fairly.

Because there is such a drastic difference in the runtime of each variation (as shown by Figure 4.1), it was necessary to institute a cutoff point, after which the algorithm would be terminated, and no time recorded. For this experiment, this cutoff point was 4 hours (i.e. any computation that would have taken longer than 4 hours was terminated before it finished). This was done with the combination of the Timing and the TimingConstrained Functions in Mathematica.

Algorithm Runtime

One of the key metrics for determining viability of any Algorithm, including a Factoring Algorithm, is the runtime of the Algorithm. Of particular note is the Asymptotic Time Complexity of the Algorithm. Lomonaco [1] noted that the BF Algorithm was not competitive with the fastest known Algorithms, but instead ran in time exponential in the bits of the Integer, while the fastest industry standard Algorithms run in sub-exponential time. We sought to conjecture about the asymptotic runtime of the Variations of BF using empirical data.

Theoretical Analysis of Runtime

Before looking at empirical results, we can make some conjectures about the runtimes of each Variant before even seeing empirical data. A quick examination of Algorithm 3.1 shows that the high level operations are the Generic Multiplication (or Addition in the optimized Variants), and the satisfaction problem with the resulting equations, for each iteration of the loop.

In each Variant, the loop runs until it finds a factor, which, intuitively is until we have enough bits of u to represent the smallest factor, which, for an integer N with n bits, is $O(n)$ iterations (that is, $O(n)$ Generic Additions or Multiplications, and $O(n)$ satisfiability problems).

Generic Addition, using the Algorithm presented by Lomonaco [1], for two Generic Integers with $O(n)$ bits each, performs $O(n^2)$ Boolean XOR and AND operations. A Generic Multiplication between the same two Generic Integers performs

$O(n)$ Generic Additions naively, so $O(n^3)$ of each kind of Boolean operation. The size of the operands varies depending on the representation, which is discussed in §4.3, but for now we will state that the size of the operands for the Default Representation is $O(2^n)$, the size for the Matrix Representation of the Generic Odd Integer is also $O(2^n)$, while the size for the Matrix Representation of the Generic Prime is $O(\frac{2^n}{n})$. The satisfiability problem has the Time Complexity of the Size of the Operands, leaving us with theoretical runtimes of $O(n^3 \cdot 2^n)$ (for the Default Representation), $O(n^2 \cdot 2^n)$ (for the Default Representation with Modified Newton's Method and for the Matrix Representation with the Generic Odd Integer), and $O(n \cdot 2^n)$ (for the Matrix Representation of the Generic Prime).

Comparison of the Four Variations of the BF Algorithm

Figure 4.1 shows the plots of runtime for the BF Variants vs. the number of bits in the Integer factored. Immediately two facts become evident: Using the Modified Newton's Method is much faster than the Default Newton's Method, and the Variants using the Matrix Representation of the Generic Integer are much faster than either Variant using the Default Representation. It should be noted that the Runtimes of the Variants that use the Matrix Representation of the Generic Integer are negligible compared to the Runtimes of the other Variants, so the scale of Figure 4.1 makes the plots for the Matrix Representation Variants seem to be 0. Figures 4.2 and 4.3 are more informative for these Variants.

For both Variants using the Default Representation, we can visually confirm

Runtimes for Factoring Algorithms

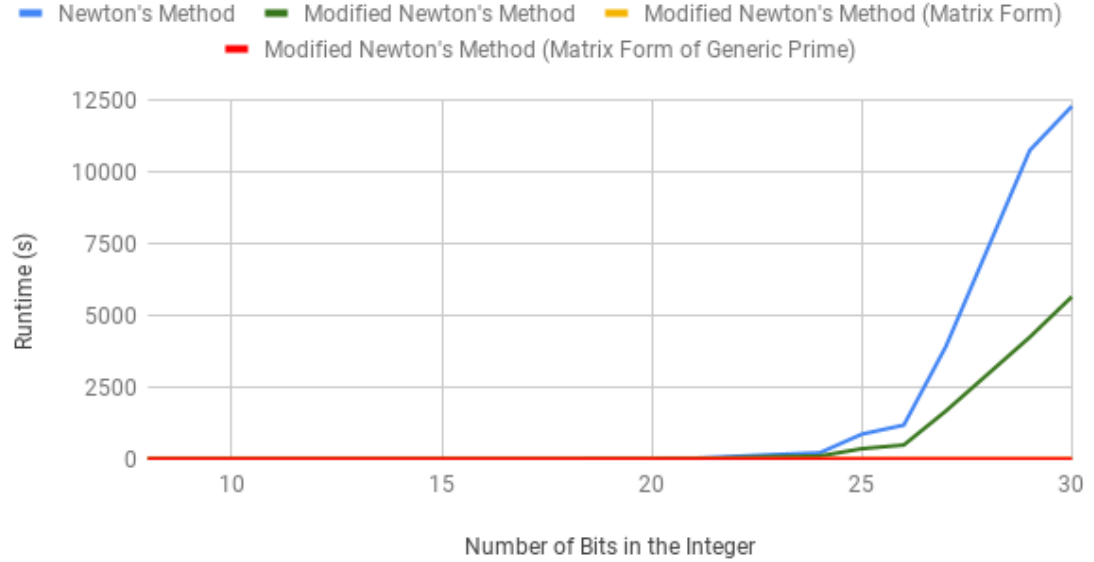


Figure 4.1: Comparison of the Variations of the BF Algorithm

our hypothesis from §4.2.1 that the BF Algorithm runs in Exponential Time. To further confirm our hypothesis, Figure 4.2 shows the runtime of the four Variants on a semi-log plot. All four plots appear to be straight lines on the semi-log plot, another indication that the BF Algorithm runs in Exponential Time.

Since both Variants of the BF Algorithm using the Matrix Representation of the Generic Integer are much faster than the Variants without, we were able to factor much larger Integers using these two Variants than we could using the Default Representation, as shown by Figure 4.3.

We see a similar trend as in Figure 4.1, where both Variants seem to run in exponential time. As can be reasonably expected, we see that using the Matrix Representation of a Generic Prime is much faster than merely using the Matrix Representation of a Generic Odd Integer. Intuitively, this makes sense due to the

Runtimes for Factoring Algorithms

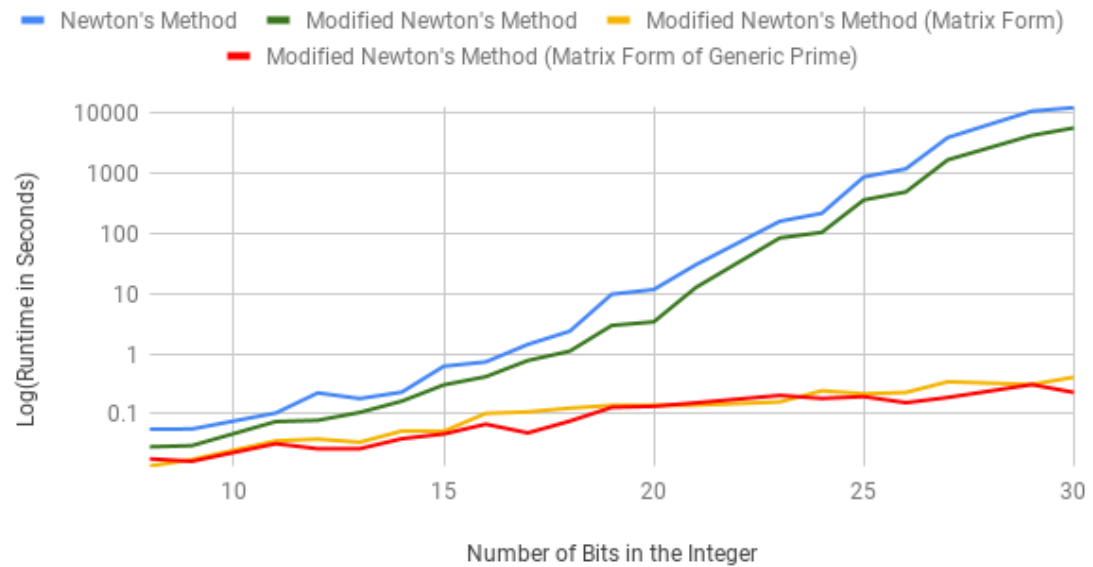


Figure 4.2: Semi-log Comparison of the Variations of the BF Algorithm

Factoring with Matrix Form of Generic Integers

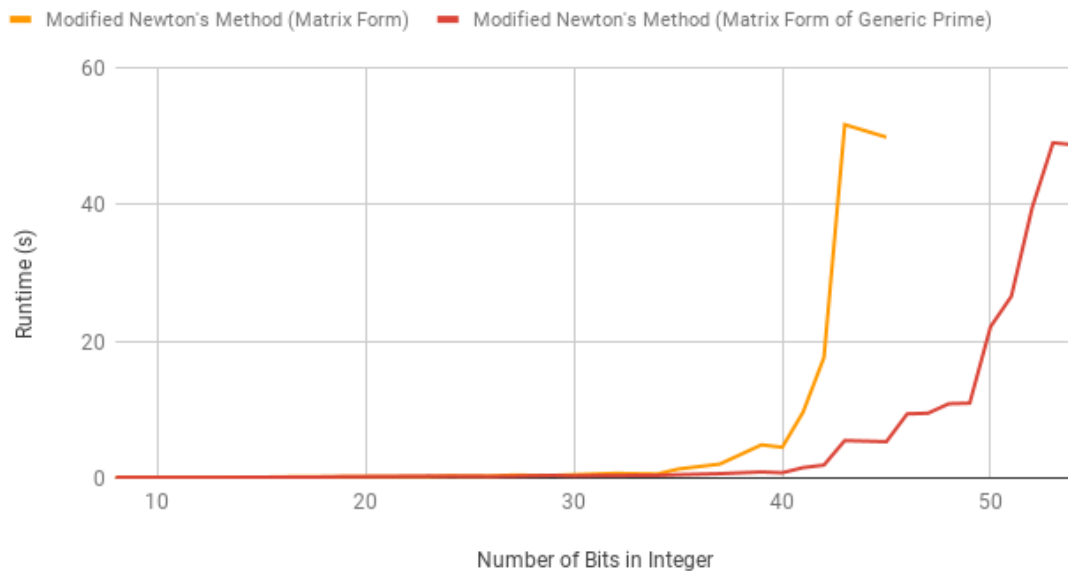


Figure 4.3: Comparison of the BF Algorithm Variations using the Matrix Representation

inefficiency introduced when trying to factor an Integer using non-primes, since all of the calculations involving non-primes give no additional information. §5 compares the runtimes of these two Variants more rigorously.

Time to Generate the Generic Integer

When considering the runtime of the Variants of the BF Algorithm that use the Matrix Representation of the Generic Integer, we need also mention the precomputation cost of Generating the Binary Matrix. The size of the Matrix depends on whether or not the Generic Integer is a Generic Prime (instead of simply a Generic Odd Integer), as well as the size of the Integer being Factored (§4.3 has more details on the Space used by the Matrix), though the Matrix always uses an exponential amount of space. Figure 4.4, unsurprisingly, indicates that it takes an exponential amount of time to generate the Generic Integer (intuitively, we expect generation of a Matrix of exponential size to take exponential time), though it takes much less time to generate the Generic Prime. This observation seems counter intuitive, because the generation of a Generic Prime involves computing a long sequence of Prime numbers, which is much more difficult than generating a list of odd Integers for the Generic Odd Integer. As §4.3 suggests, the Binary Matrix for a Generic Prime needed to Factor a specific Integer is much smaller than the Matrix needed to Factor the same Integer with a Generic Odd Integer.

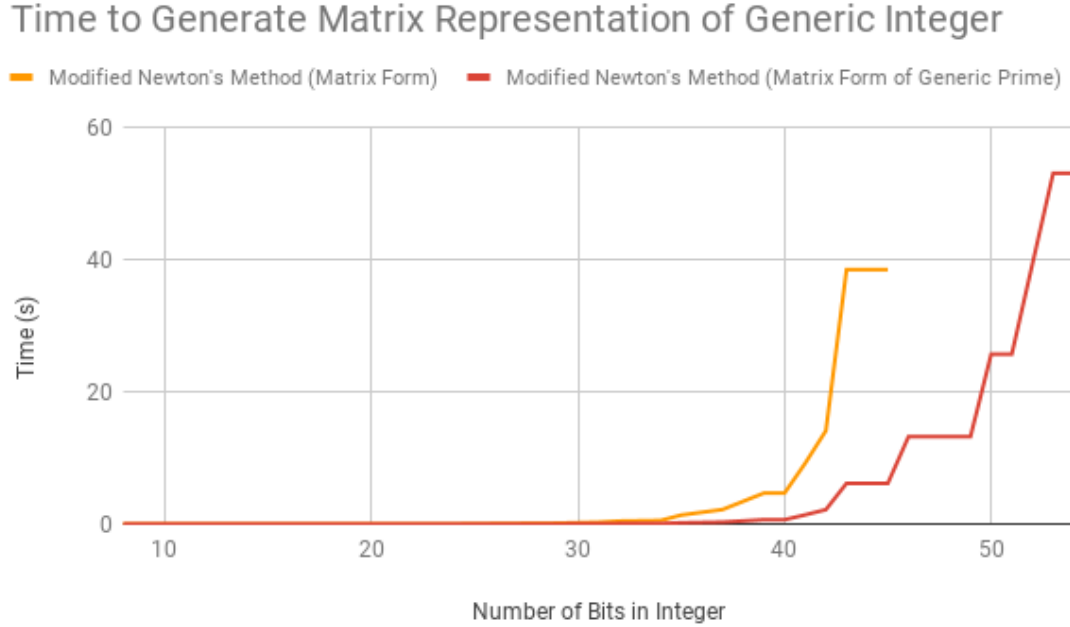


Figure 4.4: Time to Generate the Generic Prime

Algorithm Space Usage

Similar to the way we analyzed the runtime of the Variants of the BF Algorithm, we can analyze the space usage. For simplicity, we will examine the Space Usage of the Variants that use the Default Representation of the Generic Integer separately from the Space Usage of the Variants using the Matrix Representation. We will assume that each Variant is trying to Factor an Integer N with n bits.

For the Variants using the Default Representation, the Algorithm maintains two Generic Integers (u and x), each with $O(n)$ bits. Each of these bits is an arbitrary Boolean expression of $O(n)$ variables, which has $O(2^n)$ terms, suggesting that these Variants have Space Complexity $\in O(n \cdot 2^n)$, or they use exponential Space.

Space Used by Matrix Representation of Generic Integer

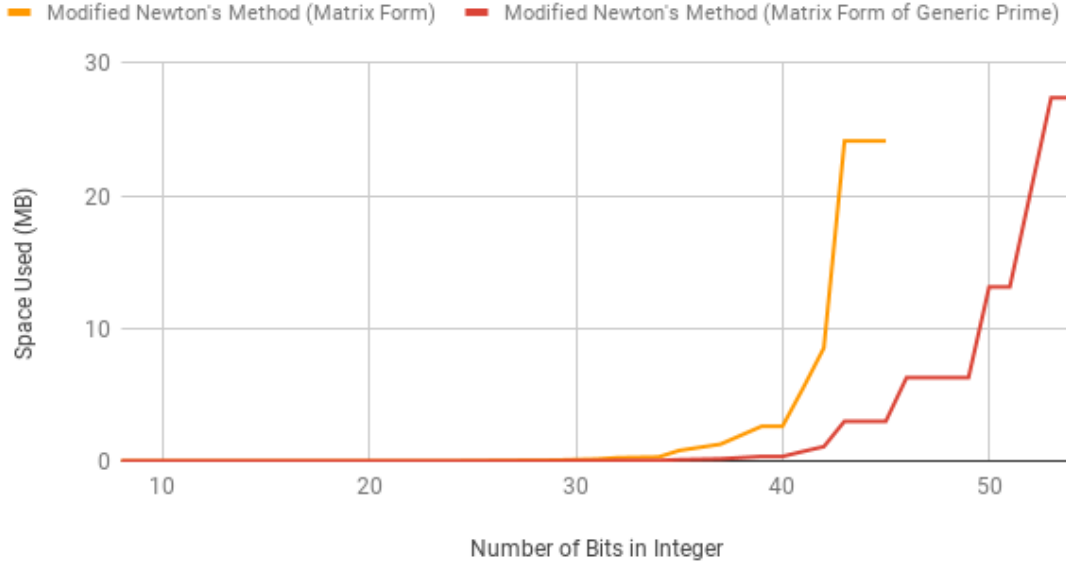


Figure 4.5: Space Used by the Generic Prime

For the Variants that use the Matrix Representation, we see that again, the Algorithm maintains two Generic Integers. Each of these Integers is represented by a Binary Matrix with $O(n)$ columns, since the Generic Integer must have $O(n)$ bits. If the Generic Integer is a Generic Odd Integer, the Binary Matrix must have $O(2^n)$ rows, because it must represent the first $O(2^n)$ odd Integers to have $O(n)$ columns. This means that the Generic Odd Integer Variant of the Algorithm has Space Complexity $\in O(n \cdot 2^n)$. However, if the Generic Integer is a Generic Prime, the Matrix need only have $O(\frac{2^n}{n})$ rows (see §3.2.4 for details), meaning that the Generic Prime Variant of the Algorithm has Space Complexity $\in O(2^n)$.

While we did not measure the maximum memory footprint of any of the Algorithms while they were running, the Variants that use the Matrix Representation of the Generic Integer need the Matrix to be precomputed, and we were able to

measure the Space Usage of this precomputed Matrix. Figure 4.5 shows that both Variants using the Matrix Representation do, in fact, use an exponential amount of space. We also can observe that, as expected, the Generic Prime uses substantially less space than the Generic Odd Integer.

Conclusions

As noted in §4.2, we see that using the Matrix Representation of the Generic Integer for the BF Algorithm is much faster than using the Default Representation, and that using the Generic Prime is faster than using a Generic Odd Integer. This first observation was to be expected, since the operations on the Binary Matrix are simple, incur low overhead, and the operands are of a fixed size, whereas logical operations on an arbitrary Boolean expression can incur high amounts of overhead, and the operands can exponentially increase in size. Similarly, it was to be expected that using a Generic Prime would be faster than using a Generic Odd Integer, since the wasted computations using non-prime Integers are eliminated.

More specifically, we see a similarity in the shapes of Figure 4.3 and Figure 4.5. This suggests that the speedup realized when moving from the Generic Odd Integer to the Generic Prime is simply of function of the size of the Binary Matrix required to factor an Integer of a given size. Figure 5.1 (trendlines are included for clarity) shows just that, that the time to factor a given Integer with the Matrix Representation of a Generic Integer is only a function of the size of the Integer in that larger Integers require larger Matrices. In fact, Figure 5.1 seems to show that there is a linear relationship between the size of the Binary Matrix and the average

time to factor the Integer, which is what we would expect to see based on §4.2.1.

Space Used by Generic Integer vs. Time to Factor

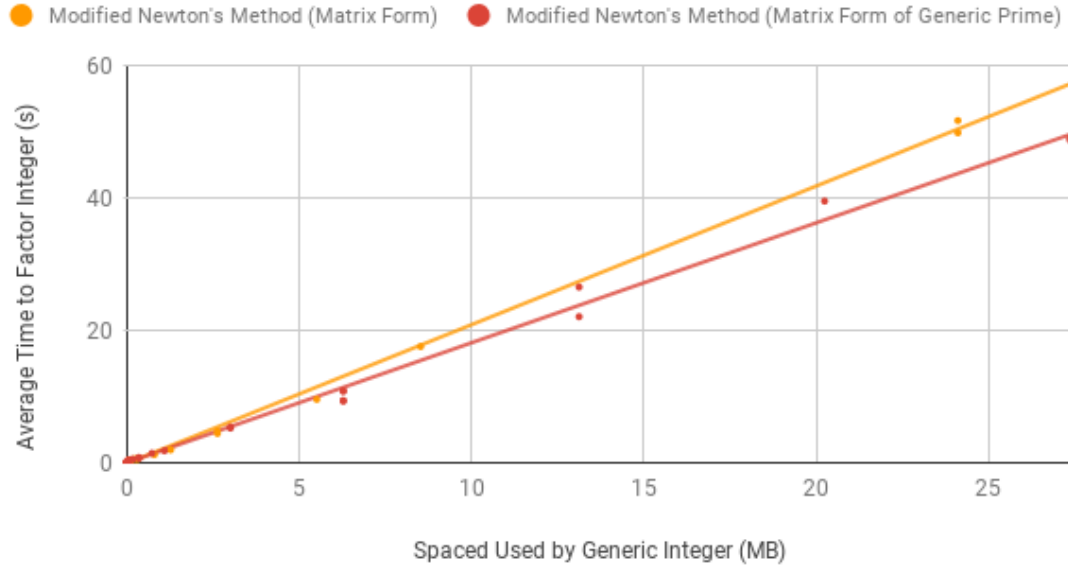


Figure 5.1: Space Used by the Generic Integer vs. Bits in the Integer to Factor

Based on our results, we can state that the BF Algorithm is an Algorithm with exponential Time and Space Complexity. Obviously this means that the BF Algorithm is asymptotically worse than the industry standards (the fastest of which are sub-exponential). For reference, we wanted to see how the fastest BF Variant that we had compared to Mathematica's built-in Factoring (which can reasonably be assumed to be close to the industry standard). For the largest number that we factored (which had 54 bits), we factored it in 48 seconds, while Mathematica's FactorInteger built-in function factored it in less than a millisecond. While there are certainly optimization techniques that could be applied to our implementation, this comparison demonstrates just how far away from the industry standard our implementation is.

Open Questions

Here we lay out some open questions and potential future research directions.

Comparing Lopsided Division with Newton's Method

We see in §2.3.2 a high-level comparison of the Lopsided Division Algorithm with Newton's Method for Division, but it lacks formal rigor to understand all of how the two Algorithms are related, and is missing a comparison from a runtime perspective. Also missing is a theoretical analysis of the complexity of the Lopsided Division Algorithm. It would be interesting to see which Algorithm is faster, if either is significantly so.

Potential Extension to Quantum Computing

While the fastest implementations of the BF Algorithm (the Variants that use the Matrix Representation of the Generic Integer) use an exponential amount of space, it may lead to a Quantum Factoring Algorithm. At a high level, the Binary Matrix that represents the Generic Integer could be represented as a superposition of Quantum states, reducing the space complexity. Since it appears that the runtime of the Algorithm is driven by the amount of space used, reducing the space complexity

could drastically reduce the time complexity. If the application of Newton's method could then be represented as unitary transformations, and a way to observe the result so that it yields a factor, it could lead to an efficient Quantum Factoring Algorithm, one that would amount to a Quantum Parallel Trial Division Algorithm.

Bibliography

- [1] Samuel J. Lomonaco. Symbolic Arithmetic and Integer Factorization. *ArXiv e-prints*, page arXiv:1304.1944, April 2013.
- [2] David Terr, Helena Verrill, and Eric W. Weisstein. “p-adic integer.” From MathWorld—A Wolfram Web Resource. Last visited on 12/6/2018.
- [3] Eric W. Weisstein. “p-adic number.” From MathWorld—A Wolfram Web Resource. Last visited on 12/6/2018.
- [4] Keith O. Geddes, Stephen R. Czapora, and George Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.
- [5] Eric W. Weisstein. “prime number theorem.” From MathWorld—A Wolfram Web Resource. Last visited on 3/15/2019.

