university of groningen

Faculty of Science and Engineering

Mathematics and Applied Mathematics

Integer Factorization via Order Finding

Bachelor’s Project Mathematics

February 2024

Student: D.Colella

First supervisor: Steffen Müller

Second assessor: Jaap Top

Abstract
Integer factorization is a challenge that forms the basis of many cryptography systems. These systems rely on the difficulty of factorizing large composites rapidly. The problem of factoring an integer n is closely related to the problem of computing the order of a residue in the ring of integers modulo n.
In this thesis, we consider an algorithm developed by Stange that computes the order of a residue. We discuss approaches to increase the efficiency of Stange's algorithm.
To show the relation between integer factorization and order finding, we present a polynomial time algorithm due to Miller to compute the factorization of n from the order of residues modulo n.
Furthermore, we explore two algorithms that use the order of a residue modulo n to compute a factorization of n, inspired by Shor and Ekerå. Similarly as before, we consider techniques to increase the efficiency of the algorithms and compare their implementations.

Contents

1 Introduction **4**
1.1 Notation 5

2 Index Calculus and Stange's algorithm **6**
2.1 Description of Algorithms 6
2.2 Correctness of Stange's Algorithm and Example 7

3 Improving Stange's Algorithm **10**
3.1 Time Complexity 10
3.2 The Linear Sieve Method 12

4 Polynomial Time Reduction **17**
4.1 Notation and Useful Theorems 17
4.2 Proof of Theorem 3 18

5 Factorization Algorithms **22**
5.1 Shor's Reduction 22
5.2 Ekerå's Reduction 23
5.3 Comparison of the Reductions 24

6 Improving Factorization Reductions **26**
6.1 Single Kernel Algorithm 26
6.2 Using Odd Orders 30
6.3 Employing the Jacobi symbol 33

7 Conclusion **35**

References **36**

8 Appendix **38**
8.1 Subsection 3.1 38
8.2 Subsection 5.3 40
8.3 Subsection 6.1 40
8.4 Subsection 6.2 41
8.5 Subsection 6.3 42

1. Introduction

The *factorization problem* is the task of decomposing an integer into a product of integers. The security of many crypto systems relies on the hardness of the factorization problem. Examples include the EMV, the standard adopted by Visa, Mastercard etc, to secure the chip-and-pin smartcard transactions which is based on the RSA cryptosystem or the SSL/TLS standard commonly used to secure many protocols online [Gau14, Section 1.1]. Multiple algorithms have been developed to factorize integers such as Pollard's rho algorithm, Shor's quantum factoring algorithm, the Elliptic Curve Method or the Number Field Sieve [Gre03, Section 3]
A problem strongly related to the *factorization problem* is the *order problem.*

Definition 1. *Given an integer n and a residue $g \bmod n$, the order problem is the problem of computing the smallest positive integer x such that*

$$g^x = 1 \bmod n.$$

It is known that the *order problem* and the *factorization problem* are closely related. Indeed, either problem can be reduced to solving the other [Mil76, Section "Relative Computational Complexity", Theorem 4]. Moreover, new instances of both problems can be created with ease. Nonetheless, no efficient algorithm has been found to solve these problems [Ste11, Section 11.7.4]. The difficulty in solving these problems is considered large enough that cryptography and security systems are based upon them.
In this thesis, we consider one instance of an algorithm that solves the *order problem.* This algorithm is developed by Stange in [Sta23]. It is strongly inspired by the Index Calculus algorithm, one of the best known algorithm for computing discrete logarithms [DM99, Section 1]. Stange's algorithm utilizes multiplicative relations modulo n to obtain the order of an element in $(\mathbb{Z}/n\mathbb{Z})^\times$. While Stange's algorithm is not the most efficient, it is noteworthy due to its simplicity and connection to the Index Calculus.
We consider improvements to be applied to Stange's algorithm, following Stange's suggestions. We analyze the computational complexity and optimize the values of the parameters of the algorithm accordingly. Furthermore, we consider methods that can decrease the time needed to compute the multiplicative relations.
We then complete Stange's work and formulate algorithms that can solve the *factorization problem* by computing the order of residues using Stange's order finding algorithm. To give a theoretical basis to our work, we demonstrate that a reduction of the *factorization problem* to the *order problem* can be done in an efficient way thanks to an algorithm by Miller [Mil76, Section "Tests For Primality", Definition of A_f].
While Miller's work is of theoretical significance, it is not practical. Therefore, we consider other reductions. Following Stange's suggestion, we consider a famous reduction of the *factorization problem* to the *order problem* popularized by Shor [Sho94, Section 6] which employs residues of even order. This reduction is also notable for its simplicity. We consider a second reduction developed in [Eke21, Section 3.2]. This reduction is notable due to its connection to Miller's work. We then consider approaches that can reduce the number of computations needed to compute the factorization of an integer. These approaches increase the probability of avoiding residues that do not output nontrivial factors.
Throughout this thesis, we implement the algorithms described using the mathematics software Sagemath. Our implementation can be found on https://github.com/Daviderug/Thesis.git.

1.1 Notation

Even integers and perfect powers can be efficiently tested and so are generally not considered within factoring algorithms. To test perfect powers, Bernstein shows that it can be done essentially in linear time [Ber98, Section 1, Theorem 1] Moreover, some may even rely on these assumptions for correctness. For example, Shor's quantum factoring algorithm [Sho94] would fail if the integer had only one prime divisor. Therefore, we employ the following definition within this thesis.

Definition 2. *The integer n is appropriately composite if it is a composite integer that is odd and is not a perfect power.*

Furthermore, within this thesis, we do not concern ourselves with prime factorization. Testing for primality is not considered computationally difficult. Tests such as the Miller-Rabin test can determine primality of an integer efficiently. Therefore, it is sufficient to obtain algorithms that output a nontrivial factorization $n = n_1 \cdot n_2$. The factors n_1, n_2 can be tested for primality with ease and, if found to be composite can be factored further.
Given $n = \prod_{i=1}^{k} p_i^{e_i}$, we make use of the following functions in this thesis:

$$\begin{aligned} \phi(n) &= \prod_{i=1}^{k} p_i^{e_i-1}(p_i - 1) \quad \text{(Euler's totient function)} \\ \lambda(n) &= \mathrm{lcm}_{1\le i\le k}\{p_i^{e_i-1}(p_i - 1)\} \quad \text{(Carmichael function)} \\ \lambda'(n) &= \mathrm{lcm}_{1\le i\le k}\{p_i - 1\} \end{aligned}$$

and for odd n and any integer a

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i} \quad \text{(Jacobi symbol)}$$

We also use the following notation:

- Denote the length of the binary representation of n as $d(n) = \lfloor \log_2(n) \rfloor + 1$.
- Let $h(n) = \max\{\mathrm{ord}_n(2), \ldots, \mathrm{ord}_n(2 \cdot d(n)^2)\}$
- The cost of multiplying two numbers of length $d(n)$ is denoted as $M(n)$. Due to Harvey and Van Der Hoeven [HH21, Theorem 1.1], $M(n)$ has the upper bound $O(d(n)\ln(d(n)))$
- To indicate the highest integer x such that 2^x divides m, we use the standard notation $\nu_2(m) = \max\{x : 2^x | m\}$.
- Denote the index of a residue as $\mathrm{ind}_p(a) = \min\{m : b^m = a \bmod p\}$ for a primitive root b of $(\mathbb{Z}/p\mathbb{Z})^\times$ where p is a prime
- Let $L_n(\alpha, c) = e^{(c+o(1))(\ln n)^\alpha (\ln\ln n)^{1-\alpha}}$ be a function that expresses the asymptotic computational complexity of a subexponential algorithm

2. Index Calculus and Stange's algorithm

2.1 Description of Algorithms

Stange's paper [Sta23] introduces a probabilistic algorithm inspired by the Index Calculus that solves the *order problem* for any integer n. The Index Calculus is a classical algorithm for computing discrete logarithms within $(\mathbb{Z}/n\mathbb{Z})^\times$ for primes n. This algorithm finds a value of x that satisfies the discrete logarithm $y = g^x \bmod n$ where n is a prime integer and g is a primitive root modulo n. Instead, Stange's algorithm computes the order of a residue $g \bmod n$ for any integer n.
Both algorithms attempt to collect N residues $g^{a_i} \bmod n$ that are B-smooth for some parameter B.

Definition 3. *An integer is B-smooth if all of its prime factors $p_1, p_2, \ldots, p_b$ are all less than or equal to B.*

Therefore, the algorithms search for N multiplicative relations of the form

$$g^{a_i} = \prod_{j=1}^{b} p_j^{f_{i,j}} \bmod n$$

for random $a_i \in \mathbb{Z}_{>0}$ and a fixed prime factor base $\{p_1, p_2, \ldots, p_b = B\}$.
These relations are equivalent to the linear relations

$$a_i = \sum_{j=1}^{b} f_{i,j} \log_g(p_j) \bmod \phi(n).$$

While the individual discrete logarithm might not exist for each p_j, assume temporarily that each $p_j \bmod n$ is in the subgroup generated by $g \bmod n$ for the exposition.
The linear relations form the linear system $\mathbf{a} = F \cdot \mathbf{p}$ where $\mathbf{a} = (a_i)_{1\le i\le N}$, the relation matrix is $F = (f_{i,j})_{1\le i\le N, 1\le j\le b}$ and the unknown is $\mathbf{p} = (\log_g p_j)_{1\le j\le b}$.
Generally, we aim to obtain an overdetermined system. Hence, we pick $N = b + c$ for some positive integer c.
The classical Index Calculus algorithm, as described in [Gre03, Subsection 3.5.1], then solves the system and uses the values of $\log_g(p_j)$ to solve the discrete logarithm $y = g^x \bmod n$ where n is a prime integer and g generates the group $(\mathbb{Z}/n\mathbb{Z})^\times$:
computing the multiplicative relation

$$y \cdot g^k = g^x \cdot g^k = \prod_{j=1}^{b} p_j^{e_j} \bmod n$$

for some $k \in \mathbb{Z}_{\ge 0}$ gives rise to the equation

$$x + k = \sum_{j=1}^{b} e_j \log_g(p_j) \bmod \phi(n)$$

which can be solved for x.
On the other hand, Stange's algorithm [Sta23, Section 2, Algorithm 2.2] computes the order of $g \bmod n$ by considering the basis of the right kernel of $F^T \in \mathbb{Q}^{b\times(b+c)}$. The basis is used to compute multiples of the order of n.
To ensure the computations remain in $\mathbb{Z}$ and with the lowest pairwise greatest common divisors possible, the basis has to be scaled to have integer entries with no common factor. We pick c distinct vectors in the basis and denote them $w_1, w_2, \ldots, w_c \in \mathbb{Z}^{b+c}$. We compute the dot product

$$\alpha_t = w_t \cdot \mathbf{a}$$

for $t = 1, 2, \ldots, c$. These α_t are multiples of $\mathrm{ord}_n(g)$. Hence, taking the greatest common divisor of these values gives the order of g with high probability.

Remark: Since the residue $g \bmod n$ is not necessarily a primitive root of n within Stange's algorithm, it is likely that the discrete logarithms $\log_g(p_j)$ do not exist. For the purposes of the algorithm, this is not relevant and does not affect the computation of the order. The discrete logarithms appear only to clarify the connection between the relation matrix F and the vector $\mathbf{a}$ and the value of the discrete logarithm is not computed in any step of the algorithm. Stange [Sta23, Section 1, Remark 1] states that in the case that $\log_g(p_j)$ and $\log_g(p_i)$ do not exist but $\log_g(p_i p_j)$ does, the coefficient of $\log_g(p_i)$ and $\log_g(p_j)$ in the relation matrix will be the same.

2.2 Correctness of Stange's Algorithm and Example

Stange's algorithm is a Monte Carlo algorithm, i.e it is an algorithm that has a probability (generally small and bounded) of an incorrect output. In this case, given an appropriately composite n and a residue $g \bmod n$, the algorithm outputs the order of g with a small probability of outputting a multiple of the order instead. Increasing c increases the probability that the output is exactly the order of g. From the integer n and the residue $g \bmod n$, the relations

$$g^{a_i} = \prod_{j=1}^{b} p_j^{f_{i,j}} \bmod n$$

and the system $\mathbf{a} = F \cdot \mathbf{p}$ are obtained by trial division. This method corresponds to a naive implementation of the Index Calculus. Modern implementations of the algorithm employ more efficient methods to obtain smooth residues. Examples of such methods are considered in Section 3. By the Rank-Nullity Theorem

$$|\text{basis of} \ker(F^T)| = \mathrm{null}(F^T) = b + c - \mathrm{rank}(F^T).$$

Since $\mathrm{rank}(F^T) \leq \min(b, b + c) = b$,

$$|\text{basis of} \ker(F^T)| \geq b + c - b = c.$$

This guarantees that the basis contains at least c vectors $w_1, w_2, \ldots, w_c$.
Note that the vector $\mathbf{a}$ is in the column space of F. In particular,

$$\mathbf{a} = \sum_{j=1}^{b} F_j \mathbf{p}_j \bmod \phi(n)$$

where F_j is the j-th column of the relation matrix F. Each vector $w_t \in \ker(F^T)$ satisfies

$$F_j \cdot w_t = \sum_{i=1}^{b+c} f_{i,j} (w_t)_i = 0$$

for each row j of F^T, i.e. each column j of F. Therefore

$$\mathbf{a} \cdot w_t = \left(\sum_{j=1}^{b} F_j \mathbf{p}_j\right) \cdot w_t = \sum_{j=1}^{b} (F_j \cdot w_t) \mathbf{p}_j = 0 \bmod \phi(n).$$

The formation of the matrix F and vector $\mathbf{a}$ is done within the group $(\mathbb{Z}/n\mathbb{Z})^{\times}$ while the computation of $\ker(F^T)$ and of the dot product $\mathbf{a} \cdot w_t$ is done within the field of rational numbers $\mathbb{Q}$ rather than

$(\mathbb{Z}/n\mathbb{Z})^\times$. This ensures that the relationship $\mathbf{a} \cdot w_t = 0 \bmod \phi(n)$ holds but $\mathbf{a} \cdot w_t \neq 0$ in $\mathbb{Q}$ in most occurrences.
Hence

$$g^{\mathbf{a} \cdot w_t} \bmod n = g^0 = 1 \bmod n$$

Hence, by Lagrange's Theorem $\text{ord}_n(g)$ must divide $\alpha_t = w_t \cdot \mathbf{a}$ for each $t = 1, 2, \dots, c$. The greatest common divisor of all the α_t outputs $\text{ord}_n(g)$ with high probability. In particular, Stange states that the algorithm correctly identifies the order of a given residue with probability of at least 0.999 if $c \geq 9$ assuming Hypothesis 3.1 holds [Sta23, Section 3, Hypothesis 3.1, Theorem 3.2]. This reduces the probability of the algorithm outputting a multiple of the order instead of the exact order to be negligible. Within Stange's implementations, c is taken to be constant 10. Consequently, we take $c = 10$ for all implementations unless otherwise stated.
Stange's algorithm can be summarized in Algorithm 1. To showcase the algorithm, we compute the

Algorithm 1: Stange's order finding algorithm

```
Input  : An appropriately composite $n$ and a residue $g$; integer parameters $b$ and $c$
Output: The multiplicative order of $g \bmod n$
Phase 1: Relation finding
$i \longleftarrow 0$
while $i < b + c$ do
    Draw an integer $a_i$ randomly from the set $\{1, \dots, n\}$ (ensuring that $a_i$ has not been drawn
      previously)
    Compute the smallest positive residue of $g^{a_i} \bmod n$
    if $g^{a_i} = \prod_{j=1}^{b} p_j^{f_{i,j}} \bmod n$ then
        Add $[f_{i,1}, \dots, f_{i,b}]$ as the $i$-th row of $F^T$
        Add $a_i$ as the $i$-th entry of vector $\mathbf{a}$
        $i \longleftarrow i + 1$
Phase 2: Linear algebra
Compute the basis vectors $[w_1, w_2, \dots, w_c]$ of the kernel of $F^T$
Phase 3: GCD computation
for $1 \leq t \leq c$ do
    $\alpha_t \longleftarrow \mathbf{a} \cdot w_t$
return $\gcd(\alpha_1, \alpha_2, \dots, \alpha_c)$
```

order of $g = 2$ with $n = 9983$.
Using $b = 10$ and $c = 10$, we obtain the factor base $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$ and we attempt to find $b + c = 20$ relations. The algorithm finds the relations:

$$\begin{array}{llll}
2^{1225} = 2^2 \cdot 3 \cdot 17^2, & 2^{1558} = 2^4 \cdot 5 \cdot 7 \cdot 17, & 2^{9421} = 2^{10} \cdot 3, & 2^{8119} = 2^3 \cdot 17, \\
2^{9335} = 3 \cdot 7^2 \cdot 19, & 2^{2408} = 2^4 \cdot 5^2 \cdot 17, & 2^{9792} = 2^4 \cdot 19^2, & 2^{2353} = 2 \cdot 3 \cdot 11 \cdot 19, \\
2^{6288} = 2^2 \cdot 5 \cdot 7 \cdot 29, & 2^{6626} = 2^4 \cdot 3^2 \cdot 19, & 2^{4785} = 2 \cdot 3 \cdot 11, & 2^{4887} = 2^3, \\
2^{7762} = 2^3 \cdot 3 \cdot 17, & 2^{7507} = 3^2 \cdot 11 \cdot 29, & 2^{3341} = 2^4 \cdot 11 \cdot 29, & 2^{6537} = 2 \cdot 3^4 \cdot 29, \\
2^{9075} = 2 \cdot 3^2 \cdot 19^2, & 2^{1580} = 17^2, & 2^{3058} = 11 \cdot 13 \cdot 19, & 2^{9395} = 3^3 \cdot 13^2,
\end{array}$$

The transpose of the relation matrix is

$$\begin{pmatrix}
2 & 0 & 2 & 3 & 1 & 4 & 4 & 4 & 0 & 0 & 10 & 4 & 1 & 4 & 0 & 3 & 1 & 3 & 1 & 0 \\
1 & 1 & 0 & 1 & 2 & 0 & 0 & 2 & 2 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 4 & 3 \\
0 & 0 & 1 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\
2 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}$$

where the columns are the exponents of the primes.
Each row in the following matrix is a basis vector of the kernel.

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 1 & -1 & 4 & -2 & -4 & -10 & 1 & -2 \\
0 & 1 & 0 & 0 & 0 & -2 & 1 & 0 & 0 & 0 & 4 & 0 & 2 & -1 & 4 & 1 & -5 & -11 & 1 & -2 \\
0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 4 & 0 & 1 & -1 & 2 & 1 & -2 & -12 & 0 & -1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 2 & -2 & 6 & -1 & -6 & -10 & 2 & -3 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 1 & 1 & 0 & 0 & -2 & -11 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 3 & 0 & 0 & 1 & 0 & 0 & -1 & -12 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 3 & 0 & 0 & -1 & 2 & 0 & -2 & -8 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 & 0 & 2 & 0 & -2 & -16 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 & -4 & 0 & 2 & -4 & -2 & 2
\end{pmatrix}$$

The algorithm computes the following values of α:

$$-34188, \quad -4884, \quad -14652, \quad -19536, \quad -19536,$$
$$-29304, \quad -14652, \quad -4884, \quad -39072, \quad -4884$$

Their greatest common divisor is 4884. We check that $2^{4884} = 1 \bmod 9983$. To determine whether 4884 is the exact order or a multiple of it, further work needs to be done. By trial division of 4884, we can find the factorization $4884 = 2 \cdot 2442$. Computing $2^2 = 4 \neq 1 \bmod 9983$ and $2^{2442} = 2681 \neq 1 \bmod 9983$ shows that the order must be exactly 4884.

3. Improving Stange's Algorithm

Within this section we consider various ways to reduce the running time of Stange's algorithm. The algorithm has a complexity of $L_n(1/2, \beta)$ [Sta23, Section 4] where $L_n(\alpha, c)$ is the function defined in Subsection 1.1. The value of the constant β is not specified by Stange and can be optimized.
In [Sta23, Section 5], Stange offers several implementations to improve the run time. Stange's suggestions also include improving the relation finding phase by using alternative methods to obtain smooth residues.
The computational complexity of obtaining a smooth residue modulo n via trial division is $O(b)$ where b is the number of primes in the factor base. The methods suggested by Stange include the Elliptic Curve Method, the Linear Sieve Method and the Number Field Sieve which have complexity $L_n(1/2, \sqrt{2})$ [Sut21, Section 10.4], $L_n(1/2, 1)$ [COS86, Section 4] and $L_n(1/3, 3^{2/3})$ [Gor93, Section 4] respectively. These methods have lower complexities than trial division and are preferred when computing smooth residues.
In particular, modern implementations of the Index Calculus employ the Gaussian Integers Sieve and the Linear Sieve over large primes [Can+23], both having complexity $L_n(1/2, 1)$ [LO91, Section 4]. Due to the similarities between the Index Calculus and Stange's algorithm, these methods have potential to yield similar improvements in the latter algorithm.
Within this section we consider some of these suggestions and attempt to implement them.

3.1 Time Complexity

Lemma 1. *The time complexity of Stange's algorithm is*

$$O(b^3 + b^2 \cdot u^u)$$

where $u = \log_B(n)$

Proof. Sutherland [Sut21, Section 10.3] shows that the computational running time of the relation finding phase, assuming smoothness testing is done via trial division, is

$$O((b + c) \cdot u^u \cdot b)$$

where u^u is the expected number of random residue g^{a_i} drawn to obtain a B-smooth integer. As previously mentioned, taking $c = 10$ guarantees a high probability of success of the algorithm. We therefore regard c as a negligible constant in the runtime. Hence, given a fixed n and residue g, within the order finding algorithm the relation finding phase is computed in $O((b + c) \cdot u^u \cdot b) = O(b^2 \cdot u^u)$ steps.
As for the linear algebra phase, the computational complexity is

$$O((b + c)^3).$$

Indeed, a basis of the kernel of matrix $F^T \in \mathbb{Q}^{b\times(b+c)}$ can be computed via Gaussian Elimination. Indeed, constructing the block matrix

$$\left[\frac{F^T}{I_{(b+c)}}\right] \in \mathbb{Q}^{(2b+c)\times(b+c)}$$

where $I_{(b+c)}$ is the identity matrix of size $b + c$ and computing the column echelon form of F^T leads to a block matrix

$$\left[\frac{A}{B}\right]$$

where A is the column echelon form of F^T. A zero column of A corresponds to a column of B that form a basis vector of $\ker(F^T)$. Computing such form by Gaussian Elimination requires $O((b+c)^3)$ steps. Hence, the linear algebra step has a time complexity of $O((b+c)^3) = O(b^3)$.
Therefore, the total expected running time of the algorithm is

$$O(b^3 + b^2 \cdot u^u)$$

for $u = \log_B(n)$ $\square$

To improve the performance of Stange's algorithm, we use the value of parameter b that minimizes the expression $b^3 + b^2 \cdot u^u$. Saha [Sah12] approximates the minimum of the expression as

$$u_n \approx \sqrt{\frac{2\ln(n)}{\ln\ln(n)}}.$$

Note that for this choice of u,

$$\frac{u_n}{\ln(n)} \approx \sqrt{\frac{2\ln(n)}{\ln^2(n)\ln\ln(n)}} = 2\sqrt{\frac{1}{2\ln(n)\ln\ln(n)}} =: C_n$$

We approximate b asymptotically using the Prime Number Theorem as stated by Olsen [Ols23, Section 1, Corollary 1.3]

Theorem 2. *Let the number of prime numbers lower than B be $\pi(B)$. Then*

$$\lim_{x\to\infty} \pi(x)\ln(x)/x = 1.$$

We can therefore approximate b as such

$$b \approx \frac{B}{\ln(B)}.$$

Hence, for $u = u_n$ with $B_n = n^{1/u_n}$ we obtain

$$\begin{aligned} b_n \approx B_n/\ln(B_n) &= \frac{u_n}{\ln(n)} n^{1/u_n} = \frac{u_n}{\ln(n)} exp(\ln(n)/u_n) \approx C_n e^{(1/C_n)} \\ &= C_n e^{\frac{\sqrt{2}}{2}\sqrt{\ln(n)\ln\ln(n)}} = L_n(1/2, \sqrt{2}/2) \end{aligned}$$

as the optimal value of the parameter for a fixed n.
Sutherland shows that

$$u_n^{u_n} = L_n(1/2, \sqrt{2}/2)$$

for u_n as expressed above [Sut21, Subsection 10.5]. Therefore

$$b_n^2 u_n^{u_n} = L_n(1/2, \sqrt{2}/2)^2 \cdot L_n(1/2, \sqrt{2}/2) = L_n(1/2, \sqrt{2}/2)^3 = L_n(1/2, 3\sqrt{2}/2)$$

and

$$b^3 = L_n(1/2, \sqrt{2}/2)^3 = L_n(1/2, 3\sqrt{2}/2).$$

Hence, the computational complexity is

$$L_n(1/2, 3\sqrt{2}/2) + L_n(1/2, 3\sqrt{2}/2) = L_n(1/2, 3\sqrt{2}/2)$$

This runtime agrees with Stange's heuristic runtime of $L_n(1/2, \beta)$ [Sta23, Section 4]. Within Stange's paper [Sta23, Section 4], the value of b is optimized to reduce the complexity of relation finding phase, leading to a value of $b_n = L_n(1/2, 1/2)$ and overall computational complexity of the algorithm $L_n(1/2, 9)$. Our optimization takes into account the linear algebra phase as well and leads to a lower computational complexity.
To experimentally confirm that b_n minimizes the running time of Stange's algorithm, we conduct an investigation over the running time of Algorithm 1 as b varies. We consider 20 integers between 500 and 10^8, ensuring that they are appropriately composite. Such integers are

$$I := \{537,\ 765,\ 8585,\ 19053,\ 61453,\ 101371,\ 199989,\ 247251,\ 484957,\ 671015,\ 871933,\\ 907423,\ 1744953,\ 1946725,\ 2177645,\ 2381625,\ 3632763,\ 5001635,\ 7865455,\ 22653803\}.$$

We record and plot the average computing time in seconds of Algorithm 1 for various values of b[1] and $c = 10$ and random residues $g \bmod n$. The plots can be seen in their entirety in the Appendix. We report here a representative sample. For the integers 61453, 871933, 1946725 and 22653803, we obtain the following plots:

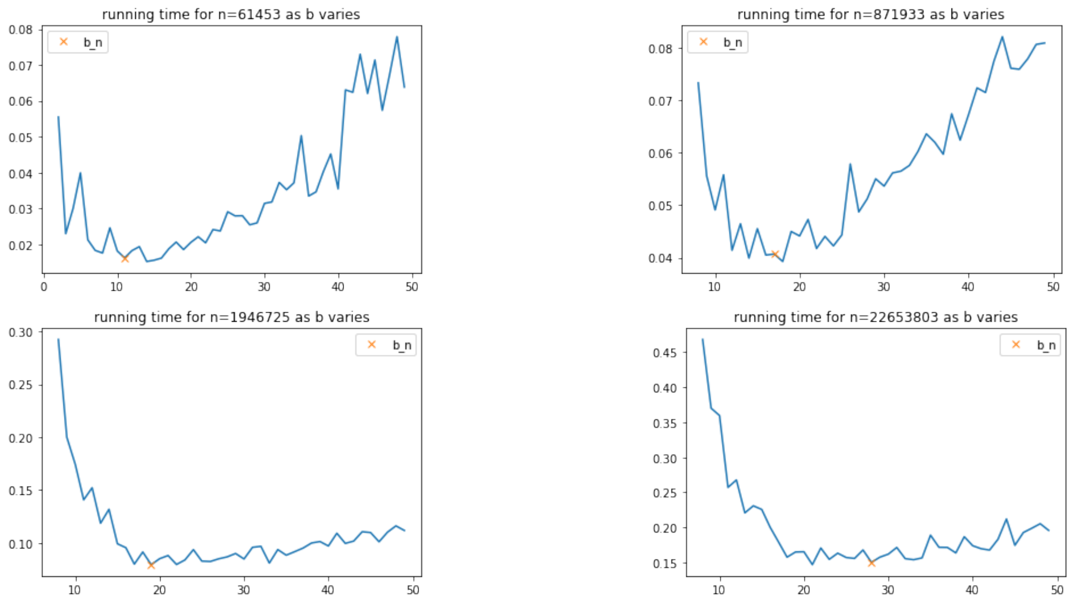


Figure 1. Running time of Stange's algorithm for various values of b for fixed n

Indeed, we see that the approximation gives a good indication of the exact value that minimizes the computational running time.

3.2 The Linear Sieve Method

The implementation of Algorithm 1 uses trial division to check if a residue $g \bmod n$ is B-smooth and compute its factorization. Although this method is simple, it is among the least efficient in finding smooth integers. It has a time complexity of $O(b)$ where b is the number of primes in the factor base [Stu02, Section 3.1.1]. Sieving methods such as the Linear Sieve and the Gaussian Integer Sieve allow for the computation of residues effectively, introduced in [COS86, Section 4 and Section 7]. Within

[1]We limit ourselves to specific intervals for b due to limits in computational power

this thesis we explore the Linear Sieve Method (LSM) due to its simplicity relative to the Gaussian Integer Sieve.

The LSM generates residues in the following way as described in [DM99, Section 2].
Let $H = \lceil \sqrt{n} \rceil$ and $\mathcal{J} = H^2 - n$. Pick c_1 and c_2 to be small integers in $[-M, M]$. Das et al [DM99, Section 2] suggest for M to be such that $2M \approx L_n(1/2, 1/2 + \varepsilon)$ for some small positive ε. We consider the smoothness of the residues

$$\begin{aligned}(H + c_1)(H + c_2) &= H^2 + (c_1 + c_2)H + c_1c_2 \\ &= n + \mathcal{J} + (c_1 + c_2)H + c_1c_2 \\ &= \mathcal{J} + (c_1 + c_2)H + c_1c_2 \bmod n\end{aligned} \tag{1}$$

for various pairs (c_1, c_2), $c_1 \le c_2$ over a factor base S.
Das et al [DM99, Section 2] suggest taking the factor base S to be primes smaller than $L_n(1/2, 1/2)$ and integers $H + c$ for integers c in $[-M, M]$.

To efficiently test these pairs, we use a linear sieve. We fix $c_1 \in [-M, M]$ and create an empty array U of size $|\{c_1, c_1 + 1, \dots, M\}|$. Each position of U represents a possible value of c_2 while each entry of the array is the value of the real logarithms of the small primes in S dividing the residue (1) [COS86, Section 4].
For each prime power p^r where $p \in S$ and r is a small exponent, we attempt to solve the linear equation

$$(H + c_1)(H + c_2) = (H + c_1)c_2 + (\mathcal{J} + c_1H) = 0 \bmod p^r \tag{2}$$

for c_2.
Usual implementations of the LSM suggest using various values of r. Coppersmith et al [COS86, Section 4] suggest taking r to be integers $1, 2, \dots, f$ such that $p^f < L_n(1/2, 1/2)$. However Das et al [DM99, Section 4] recommend using $r = 1$ to decrease computation of a modular inverses which they identify as the costliest operation. We follow this implementation as it allows for a more efficient algorithm. However, this implementation also requires a modification at some later stage which we address subsequently.

If $\gcd(H + c_1, p) = 1$ then $H + c_1$ has an inverse modulo p. Therefore

$$d = -(H + c_1)^{-1}(\mathcal{J} + c_1H) \bmod p$$

is a solution to (2). Then for each integer $c_2 \in [c_1, M]$, check whether $c_2 = d \bmod p$. Every c_2 that satisfies the equality represents a possible pair (c_1, c_2) such that the residue $(H + c_1)(H + c_2)$ is divisible by p. Therefore, we add the real value of $\log_2(p)$ to the entry of U that corresponds to c_2.

Otherwise, $\gcd(H + c_1, p) \ne 1$ means that $\gcd(H + c_1, p) = p$ since p is a prime. Therefore, we compute the exponent $h_1 > 0$ such that

$$p^{h_1} | (H + c_1), \quad p^{h_1+1} \nmid (H + c_1).$$

Similarly, we compute the exponent $h_2 \ge 0$ such that

$$p^{h_2} | (\mathcal{J} + c_1H), \quad p^{h_2+1} \nmid (\mathcal{J} + c_1H).$$

Then for $h = \min\{h_1, h_2\}$,

$$p^h | ((H + c_1)c_2 + (\mathcal{J} + c_1H)), \quad p^{h+1} \nmid ((H + c_1)c_2 + (\mathcal{J} + c_1H))$$

for any value of c_2. Hence, any $c_2 \in [c_1, M]$ gives a pair (c_1, c_2) that solves Equation (2). Therefore, we add the value of $h\log_2(p)$ to all entries of U.

After computing the steps described above for all primes in the factor base, we check which entry of U is "sufficiently close" to the value of real $\log_2(\mathcal{J} + (c_1 + c_2)H + c_1c_2)$. The entries that are "sufficiently close" represent the values of c_2 that, along with the fixed c_2, result in residues of the form as Equation (2) that are smooth over the factor base S. We expect to find $L_n(1/2, 1/2 + 3\varepsilon)$ such residues [DM99, Section 2].
For U to be "sufficiently close" to the value of $\log_2(\mathcal{J} + (c_1 + c_2)H + c_1c_2)$, Das et al [DM99, Section 2] propose to check the absolute value of their difference is smaller than an upper bound 1. Moreover, the choice to only consider $r = 1$ requires an increase of the upper bound. Das et al [DM99, Section 4] ensure that for primes p of less than 200 bits, a bound 2.5 is sufficient. Therefore, in our implementation we use such a bound.
Having found smooth residues , trial division is used to compute the exponents and obtain

$$(H + c_1)(H + c_2) = \prod_{j=1}^{|S|} p_j^{e_j} \bmod n.$$

Das et al [DM99, Section 4] remark that due to the choice of using $r = 1$ and the increased upper bound, the method may output some non smooth residues. Nonetheless, these residues can be filtered out during the trial division phase.

The multiplicative relation for a smooth residue is equivalent to the relation

$$1 = (H + c_1)^{-1}(H + c_2)^{-1} \prod_{j=1}^{|S|} p_j^{e_{i,j}} = \prod_{s \in S} s^{e_s} \bmod n$$

where we express $1 \bmod n$ as a product of elements in the factor base S.
By collecting $|S| + c$ relations of this form, we obtain the matrix $B \in \mathbb{Z}^{(|S|+c)\times|S|}$ where the entries of the i-th row correspond the exponents of each $s \in S$ in the i-th relation.
To compute the order of an element $s \in S$, we consider the relation matrix $B^s \in \mathbb{Z}^{(|S|+c)\times(|S|-1)}$ obtained by removing from B the column $\mathbf{a}^s$ corresponding to the element s. We obtain the linear system

$$-\mathbf{a}^s = B^s\mathbf{p}^s \bmod \phi(n).$$

This system is analogous to the one described in Section 2 and leads to a multiple of the order of s. Similarly as before, the probability of obtaining the exact order depends on the number of basis vectors obtained.

Given the matrix B, suppose the column corresponding to element $s \in S$ is linearly dependent on other columns of B over the rationals. It follows that

$$\text{rank}(B^T) = \text{rank}(B^{s^T})$$

since the column does not contribute to the column space. Therefore, by the Rank-Nullity Theorem,

$$\text{null}(B^{s^T}) = |S| + c - \text{rank}(B^{s^T}) = |S| + c - \text{rank}(B^T) = \text{null}(B^T).$$

Since $\ker(B^T) \subseteq \ker(B^{s^T})$, it must be that $\ker(B^T) = \ker(B^{s^T})$. Therefore, as outlined by Stange [Sta23, Section 3, Hypothesis 3.1], the algorithm outputs $\alpha_i = 0$ for all i. Hence, the algorithm cannot compute the order of s and instead outputs 0.

This occurrence is uncommon when computing multiplicative relations via trial division as discussed by Stange [Sta23, Section 3, Hypothesis 3.1] and confirmed experimentally in our implementations. However, when using the LSM, these occurrences are more common. The residues $(H + c_1)(H + c_2)$ generated by the LSM are more likely to lead to a matrix B that satisfies the condition delineated above for every $s \in S$.
For example, in our implementation, using the LSM for the integer $n = 10001 = 73 \cdot 137$ and $\varepsilon = 0.7$, we obtain a B^T of dimensions (96×110) and $\text{null}(B^T) = 44$. Checking every matrix B^s, we obtain that only residues $s = \pm 1 \bmod n$ have columns that are linearly independent of the other columns of B. Those residues however are trivial cases when computing orders in $(\mathbb{Z}/n\mathbb{Z})^\times$.
In comparison, Stange's algorithm as described in Algorithm 1, given $n = 10001$, $g = 11, b = 10$ and $c = 10$ obtains a relation matrix F of dimensions (20×10). Using the basis vectors of $\ker(F^T)$, the algorithm computes $\text{ord}_n(g) = 1224$. The algorithm necessitates less relations and is able to compute the order while the LSM cannot with our choices in the implementation.
It is of note that the integers that form the relations modulo n generated by the LSM are likely to be smaller than n. By construction, the LSM generates residues of order $O(\sqrt{n})$ [DM99, Section 2]. For integers smaller than n, the multiplicative relations

$$1 = \prod_{s \in S} s^{e_s}$$

hold over $\mathbb{Q}$ rather than exclusively modulo n and so the system

$$-\mathbf{a}^s = B^s \mathbf{p}^s$$

is likely to hold over $\mathbb{Q}$ rather than exclusively modulo $\phi(n)$. In Algorithm 1, the values of a_i are taken randomly in the set $\{1, 2, \dots, n\}$ and so the multiplicative relations are likely to hold modulo $\phi(n)$ but not over the rationals. As more relations hold over $\mathbb{Q}$, it becomes more likely that the vector $\mathbf{a}^s$ is linearly dependent on the rows of B^s over the rationals rather than modulo $\phi(n)$. Thus

$$\alpha_t^s = -\mathbf{a}^s \cdot w_t^s = \Big(\sum_{j=1}^{|S|-1} B_j^s \mathbf{p}_j^s\Big) \cdot w_t^s = \sum_{j=1}^{|S|-1} (B_j^s \cdot w_t^s)\mathbf{p}_j^s = 0,$$

where w_t^s is a vector in the basis of $\ker(B^{s^T})$ and B_j^s is the j-th column of B^s, is likely to hold over $\mathbb{Q}$.

As this is more common in the LSM than in the trial division method, this might be the cause of the failure to compute nontrivial orders.

Stange suggests obtaining more multiplicative relations in the case $\alpha_i = 0$ for all i [Sta23, Section 4]. With the LSM this can be done by increasing the value of ε. This allows for nontrivial orders of residues to be computed if a sufficient value of ε is found. However the sufficient value of ε is not proportional to n. Indeed, using LSM for integers $n = 231 = 3 \times 7 \times 11$ and $n = 235 = 5 \times 47$ and $\varepsilon = 1$, we obtain matrices B^T of dimensions (37×29) with $\text{null}(B^T) = 3$ and dimensions (74×87) with $\text{null}(B^T) = 15$ respectively. While the first matrix considered is not able to compute nontrivial orders, the second one computes the order of multiple residues e.g. $\text{ord}_{235}(2) = 92$.
This is likely due to the number of prime divisors of n. Residues obtained by the LSM are not guaranteed to be in $(\mathbb{Z}/n\mathbb{Z})^\times$ and residues that do not belong is the group must be discarded. Hence, the LSM used for an integer with few prime divisors computes more multiplicative relations than an integer with more prime divisors. Factorization is highly nontrivial and also constitutes the purpose of algorithms developed in Section 5. As such, we cannot account for the number of prime divisors

of n in picking a value of ε as a parameter.

Due to these properties of the LSM that occur when applied to Algorithm 1, the LSM seems to not improve the running time of the algorithm in our implementations despite its lower computational complexity. Because of the similarities between the LSM and the Gaussian Integer Sieve, we expect a similar result when applying the latter to Algorithm 1.

4. Polynomial Time Reduction

It is possible to reduce the *factorization problem* to the *order problem* in polynomial time. To define this relationship more formally we introduce the notion of a Turing machine.

Definition 4. *A Turing machine is an abstract mathematical model that performs operations by reading and writing information from an infinite tape.*

We express the relationship within the following framework.

Definition 5. *Given algorithms f and g we say that f is polynomial time reducible to g, denoted $f \leq_p g$, if there exists a Turing machine which on inputs n and $g(n)$ computes $f(n)$ in $O(d(n)^K)$ steps for some constant K where $d(n)$ is as defined in Subsection 1.1.*
Given algorithms f and g we say that f is polynomial time Turing reducible to g, denoted $f \leq_p^T g$, if there exists a Turing machine with the following properties:

1. *the machine has a distinguished tape on which it calls for values of g, where the cost of calling for $g(m)$ is $d(m) + d(g(m))$ steps*
2. *the machine computes $f(n)$ in $O(d(n)^K)$ steps for some constant K*

The relations $\leq_p$ and $\leq_p^T$ are transitive ones and $f \leq_p g$ implies that $f \leq_p^T g$.
We therefore express the initial claim as such:

Theorem 3. *Suppose the Extended Riemann's Hypothesis holds. Then, there exists a factorization algorithm $\mathcal{F}$ that is polynomial time Turing reducible to a order-finding algorithm $\mathcal{O}$, i.e. $\mathcal{F} \leq_p^T \mathcal{O}$.*

This theorem is taken from [Mil76, Section "Relative Computational Complexity", Theorem 4]. This section concerns itself with showing a proof of this claim.

4.1 Notation and Useful Theorems

The proof follows Miller's proof in [Mil76, Section "Relative Computational Complexity", Theorem 4]. Miller's proof makes use of the Extended Riemann's Hypothesis:

Conjecture 1. *Extended Riemann's Hypothesis (ERH) as stated by [Mil76, Appendix]:*
The zeros of a Dirichlet's L function

$$L(S, \chi_p) = \sum_{n=1}^{\infty} \chi_p(n)/n^S$$

in the critical strip $0 \leq Re(S) \leq 1$ all lie on the line $Re(S) = 1/2$, where χ_p is a Dirichlet character modulo p.

A Dirichlet character modulo p is a function $\chi_p : \mathbb{Z} \to \mathbb{C}$ such that for any $n, m \in \mathbb{Z}$:

1. $\chi_p(n + p) = \chi_p(n)$
2. $\chi_p(nm) = \chi_p(n)\chi_p(m)$
3. $\chi_p(n) \neq 0$ if $\gcd(n, p) = 1$ and $\chi_p(n) = 0$ otherwise

In particular, we consider the Dirichlet character known as the Jacobi symbol

$$\chi_p(n) = \left(\frac{n}{p}\right).$$

The ERH is necessary for the following result proven by [Ank52].

Theorem 4. *Suppose the ERH holds. Then, given primes p and q, if a is the smallest residue modulo p that is not a quadratic residue and b is the smallest residue modulo pq that is not a quadratic residue, then there is a fixed $c \geq 1$ that does not depend on p and q such that $a \leq c \cdot d(p)^2$ and $b \leq c \cdot d(pq)^2$.*

Remark: In [Ank52], the author does not prove an explicit value of c and hence Miller does not indicate what such value would be in his proof. Explicit results have been found since, also relying on the ERH: Lamzouri et al showed that if $p \geq 5$ is a prime, the smallest non quadratic residue modulo p can be bounded by $d(p)^2$ i.e taking $c = 1$ [LLS15, Section 1, Corollary 1.1]. Bach proved that the least non-quadratic residue modulo m for any integer m can be bounded by $2d(m)^2$ [Bac84, Section 6, Theorem 2]. Hence, we assume that $c = 2$.

The following lemma, seen in [Mil76, Appendix]), is also helpful in proving the claim.

Lemma 5. *If q and p are primes and $q|(p-1)$ then a is a q-th residue modulo p, i.e. $a = b^q \bmod p$ for some b, if and only if $q| \operatorname{ind}_p(a)$*

Proof. The lemma is proved using properties of the index.
Suppose $a = c^x$ and $b = c^y$ where c is a primitive root of p and $x = \operatorname{ind}_p(a)$ and $y = \operatorname{ind}_p(b)$.
($\Longrightarrow$) If $a = b^q \bmod p$, then

$$c^x = a = b^q = c^{y \cdot q}$$

Therefore, $x = y \cdot q \bmod \phi(p)$. Hence, $q|(x + \phi(p))$. Since $q|(p-1) = \phi(p)$, we must have that $q|(x + \phi(p) - \phi(p)) = x$.
($\Longleftarrow$) If $q|x$, then

$$\operatorname{ind}_p(a) = q \cdot Q \textit{ for some } Q \in \mathbb{Z}$$

Hence, $a = c^x = c^{q \cdot Q} = (c^Q)^q$ and so a is a q-th residue modulo p. □

We also distinguish two types of integers for the purpose of the proof of the claim. Note that if $n = \prod_{i=1}^{k} p_i^{e_i}$, from the definition of λ', we see that $\nu_2(\lambda'(n)) = \max_{1 \leq i \leq k}\{\nu_2(p_i - 1)\}$.

Definition 6. *An integer n is of type A if $\nu_2(\lambda'(n)) > \nu_2(p_i - 1)$ for at least one p_i and otherwise n is of type B*

We now have all the elements necessary to proceed.

4.2 Proof of Theorem 3

We first discuss a deterministic algorithm $\mathcal{F}$ that obtains a prime factorization of n from the order of sufficiently many elements modulo n obtained by some order-finding algorithm $\mathcal{O}$. Then we show that it satisfies the definition of polynomial time Turing reducible. The algorithm $\mathcal{F}$, inspired by [Mil76, Section "Tests For Primality", Definition of A_f] is so:

1. Compute $h(n) = \max\{\operatorname{ord}_n(2), \ldots, \operatorname{ord}_n(2d(n)^2)\}$ using $\mathcal{O}$ for $d(n)$ as defined in Subsection 1.1
2. For each $a \leq 2d(n)^2$ check whether
   (a) $\gcd(a, n) \neq 1$
   (b) $\gcd(a^{h(n)/2^k} \bmod n - 1, n) \neq 1$ for $a \leq k \leq \nu_2(h(n))$

The algorithm iterates through various values of a. If the condition in step (a) or (b) is true for any a, then the algorithm has found a factor a in the case of step (a) or a factor $\gcd(a^{h(n)/2^k} \bmod n - 1, n)$ in the case of step (b). Otherwise, if the conditions in both steps do not hold for all $a \leq 2d(n)^2$, then the algorithm determines that n is prime.
Assuming the condition in step (a) fails for all $a \leq 2d(n)^2$, we prove the existence of an integer

$a \leq 2d(n)^2$ that satisfies the condition in step (b) by considering composite integers of type A and type B separately as done in [Mil76, Section "Tests For Primality", Lemma 2A].

Suppose n is of type A. Then for some primes p and q that divide n, we have $\nu_2(\lambda'(n)) = \nu_2(p-1) > \nu_2(q-1)$. Take $a \bmod n$ such that a is the smallest residue that is not a quadratic residue modulo p. Since $(q-1)|\lambda'(n)$ and $\nu_2(\lambda'(n)) > \nu_2(q-1)$, we have $(q-1)|\lambda'(n)/2$. Therefore

$$a^{\lambda'(n)/2} = 1 \bmod q$$

We also have that $(p-1)|\lambda'(n)$ and so

$$a^{\lambda'(n)} = (a^{\lambda'(n)/2})^2 = 1 \bmod p.$$

Therefore, $a^{\lambda'(n)/2} = \pm 1 \bmod p$.
However, consider the index $\text{ind}_p(a)$ of $a \bmod p$:

$$a^{\lambda'(n)/2} = (b^{\text{ind}_p(a)})^{\lambda'(n)/2} = 1 \bmod p \iff \text{ord}_p(b) = (p-1)|(\lambda'(n)/2)(\text{ind}_p(a)) \qquad (3)$$

where b is a primitive root modulo p. Since $\nu_2(\lambda'(n)) = \nu_2(p-1) =: t$,

$$\lambda'(n) = 2^t w_1, \quad p - 1 = 2^t w_2$$

for some odd integers w_1 and w_2. Then

$$(2^t w_2)|(2^t w_1/2)(\text{ind}_p(a)) \implies w_2|w_1(\text{ind}_p(a))/2.$$

Since w_2 is odd, $\text{ind}_p(a)/2$ must be an integer and so $\text{ind}_p(a)$ is even. By Lemma 5, this cannot hold when a is not a quadratic residue modulo p.
Hence, $a^{\lambda'(n)/2} = -1 \bmod p$. Note that by Theorem 4, $a \leq 2d(p)^2 \leq 2d(n)^2$.

Suppose n is of type B and consider two primes p and q that divide n. Take $a \bmod n$ such that a is the smallest residue that is not a quadratic residue modulo pq and, without loss of generality, assume that a is a quadratic residue modulo q but not p.
Following similar steps as for integers of type A, we obtain Equation (3) modulo p and modulo q. Lemma 5 can be used for both moduli to show that $a^{\lambda'(n)/2} = -1 \bmod p$ and $a^{\lambda'(n)/2} = 1 \bmod q$. Note that by Theorem 4, $a \leq 2d(pq)^2 \leq 2d(n)^2$.

In both cases, we obtain

$$a^{\lambda'(n)/2} = -1 \bmod p \qquad a^{\lambda'(n)/2} = 1 \bmod q$$

for some $a \leq 2d(n)^2$.
We now use the following Lemma as done in [Mil76, Section "Relative Computational Complexity, Theorem 4]

Lemma 6. *Suppose n is an integer and the functions $\lambda'(n)$ and $h(n)$ are as defined in Subsection 1.1. If* $\gcd(a, n) = 1$ *for all $a < 2d(n)^2$, then*

$$\lambda'(n)|h(n)$$

holds.

Proof. Let $\lambda'(n) = \prod_{j=1}^{v} q_j^{m_j}$ be the prime factorization of $\lambda'(n)$. Then for a fixed j, $q_j^{m_j}|(p-1)$ for some $p|n$. For such q_j and corresponding p, take $a \bmod p$ to be the smallest integer that is not a q_j-th residue modulo p. By Theorem 4, $a \leq 2d(p)^2$ and so $a \leq 2d(n)^2$.
We have that

$$1 = a^{order_p(a)} = b^{\text{ind}_p(a) \cdot order_p(a)} \bmod p$$

for some primitive root b of $(\mathbb{Z}/p\mathbb{Z})^{\times}$ with $\mathrm{ord}_p(b) = (p-1)$. Hence, $q_j^{m_j} \mid \mathrm{ord}_p(b) \mid \mathrm{ind}_p(a) \cdot order_p(a)$. Since $q_j^{m_j} | (p-1)$, we have that $q_j | (p-1)$. Hence, by Lemma 5 $q_j \nmid \mathrm{ind}_p(a)$. Therefore, $q_j^{m_j} | order_p(a)$. Since $a < 2d(n)^2$, we have $\gcd(a,n) = 1$ and so $q_j^{m_j} \mid \mathrm{ord}_n(a) | h(n)$.
Since this holds for any prime power factor $\lambda'(n)$, it also holds that $\lambda'(n) | h(n)$. □

By Lemma 6, $h(n)/\lambda'(n)$ is an integer. As done in [Mil76, Section "Tests For Primality", Lemma 3], consider $k = \nu_2(h(n)/\lambda'(n)) + 1$:

$$\begin{aligned} & 2^{k-1} | (h(n)/\lambda'(n)) \\ \implies & 2^{k-1} \cdot s = (h(n)/\lambda'(n)) \text{ for some integer } s \\ \implies & 2^k \cdot s \cdot \lambda'(n)/2 = h(n) \\ \implies & (\lambda'(n)/2) | (h(n)/2^k) \end{aligned}$$

and so $a^{\lambda'(n)/2} = a^{h(n)/2^k} = 1 \bmod q$.
We also have that

$$a^{h(n)/2^k} = a^{\frac{\lambda'(n)}{2} \frac{h(n)}{\lambda'(n)2^{k-1}}} = (-1)^{\frac{h(n)}{\lambda'(n)2^{k-1}}} = -1 \bmod p$$

since $\frac{h(n)}{\lambda'(n)2^{k-1}}$ is odd.
Therefore, we have that $q | (a^{h(n)/2^k} - 1) \bmod n$ and $p \nmid (a^{h(n)/2^k} - 1 \bmod n)$. Therefore, $\gcd(a^{h(n)/2^k} - 1 \bmod n, n) \notin \{1, n\}$ and so it produces a nontrivial factor of n as seen in [Mil76, Section "Tests For Primality", Proof of Correctness of A_f Case 3].
Hence, if n is composite, algorithm $\mathcal{F}$ finds a residue $a \leq 2d(n)^2$ that outputs a factor as outlined in steps (a) or (b).

Now we proceed with an analysis of the runtime following [Mil76, Section "Relative Computational Complexity, Lemma 5].
Computing $h(n) = \max\{\mathrm{ord}_n(2), \ldots, \mathrm{ord}_n(2d(n)^2)\}$ can be Turing reduced to computing algorithm $\mathcal{O}$ in polynomial time, i.e. $h(n) \leq_p^T \mathcal{O}$. Indeed, as stated in [Mil76, Section "Relative Computational Complexity", Theorem 4] we can define a Turing machine which calls for $\mathrm{ord}_n(2), \ldots, \mathrm{ord}_n(2d(n)^2)$ and computes their maximum, therefore giving $h(n)$ with $2d(n)^2 = O(d(n)^2)$ calls to the tape.
We now show that $\mathcal{F} \leq_p h(n)$.
Since $\mathrm{ord}_n(i) | \lambda(n)$ for all $2 \leq i \leq 2d(n)^2$, then $h(n) | \lambda(n)$ and so $h(n) \leq \lambda(n)$. Since $d(\lambda(n)) = O(d(n)^k)$ by [Mil76, Section "Relative Computational Complexity, Theorem 3], $d(h(n)) \leq d(\lambda(n)) = O(d(n)^k)$.
Step 3 of the algorithm runs in $O(d(n)^{k+3}M(n))$ steps as shown in [Mil76, Section "Tests For Primality", Analysis of Running Time]:

- computing the greatest common divisor of two integers can be done in $O(d(n)^2)$ steps where n is the largest of the two as shown in [Knu97, Section 4.5.2.]
- computing $a^m \bmod n$ can be done in $O(|m|M(n))$ steps as shown in [Knu97, Section 4.3.2.]

  Hence, step (a) can be done in $O(d(n)^2)$ steps. The function $h(n) = \max\{\mathrm{ord}_n(2), \ldots, \mathrm{ord}_n(2d(n)^2)\}$ can be bounded by n from above so computing $a^{h(n)/2^k} \bmod n$ can be bounded by $O(d(n)M(n)$ steps.
  Therefore, one iteration of step (b) can be computed in $O(d(n)M(n) + d(n)^2)$ steps.

  > Note that if an integer is represented in binary form, the number of consecutive zeros to the right of the first non-zero term is the value of $\nu_2(n)$. Indeed, if $n = (n_b \ldots n_2 n_1 n_0)_2$ in binary, then
  >
  > $$n = n_0 + n_1 \cdot 2^1 + n_2 \cdot 2^2 + \cdots + n_b \cdot 2^b.$$

The first non-zero n_i then reveals the highest power of two that can be factored out. Hence, we obtain the bound $\nu_2(h(n)) \leq d(h(n))$.

Therefore, we iterate step (b) over integers $k \in [a, \nu_2(h(n))] \subset [1, d(h(n))]$. Hence, step (b) is computed in $O((d(n)M(n) + d(n)^2)d(h(n)))$.
Multiplication takes at least $O(d(n))$ steps so $M(n)$ is computationally more heavy than $d(n)$. Hence, step (b) is $O(d(n)M(n)d(h(n)))$ steps. Since $d(h(n)) = O(d(n)^k)$, the step can be computed in $O(d(n)^{k+1}M(n))$ steps.
Overall, since step 3 is iterated over $a \leq$, it is computed in $O(d(n)^2 d(n)^{k+1}M(n)) = O(d(n)^{k+3}M(n))$ steps.

Hence, in $O(d(n)^{k+3}M(n))$ steps the algorithm can either determine a factor n' of n or determine that n is prime. We have that $n'|n \implies \lambda'(n')|\lambda'(n)$ and so then $\lambda'(n')|h(n)$.
We repeat the algorithm above replacing n with n'. Therefore, in $O(d(n')^{k+3}M(n'))$ steps we either know n' is prime or that there is a nontrivial factor n'' of n'.

The number of prime factors of n can be bounded by $\log_2(n)$:
if $n = \prod_{i=1}^{k} p_i^{e_i}$ is the prime factorization of n, then

$$\log_2 n = \sum_{i=1}^{k} e_i \log_2 p_i \geq \sum_{i=1}^{k} \log_2 p_i \geq \sum_{i=1}^{k} 1 = k$$

since $e_i \geq 1$ and $p_i \geq 2$.

Therefore, iterating the algorithm at most $d(n)$ times gives all prime factors of n. Thus, we get a prime factorization of n in $O(d(n)^{k+4}M(n))$ steps.
Therefore, $\mathcal{F} \leq_p h(n)$. By transitive properties, we then have that $\mathcal{F}$ is Turing reducible to $\mathcal{O}$ in polynomial time. This finishes the proof of Theorem 3

5. Factorization Algorithms

While Miller's deterministic algorithm shows a strong relation between the *order problem* and the *factorization problem*, implementation of the algorithm is not practical as it requires computing $2d(n)^2$ orders of residues. We look at other reductions that are more efficient. The first reduction is a known reduction of the *factorization problem* to the *order problem.* The second reduction is notable due to the relation to Miller's algorithm described in the previous section.

5.1 Shor's Reduction

Stange's algorithm can be made into a factorization algorithm through the following proposition, inspired by Shor's work on factorization from a quantum order finding algorithm:

Proposition 7. *Given an appropriately composite n, if $x^2 = 1 \bmod n$, and $x \neq \pm 1 \bmod n$ for $x \in (\mathbb{Z}/n\mathbb{Z})^\times$, then* $\gcd(x-1, n)$ *and* $\gcd(x+1, n)$ *are nontrivial factors of n.*

Proof. Since $x^2 - 1 = 0 \bmod n$, it follows that n divides $(x-1)(x+1)$. However $x \neq \pm 1 \bmod n$ so n cannot divide either factor. Hence, $\gcd(x-1, n)$ and $\gcd(x+1, n)$ give nontrivial factors. □

With the inputs n and g, Stange's Algorithm computes the order r of a $g \bmod n$ if successful. If r is even and $g^{r/2} \neq -1 \bmod n$, then Proposition 7 can be applied to $x = g^{r/2}$ to obtain nontrivial factor $\gcd(g^{r/2} - 1, n)$. If the algorithm finds an odd order, then it must restart the computations with a different residue g until if finds an even order. Shor proved that the existence of a residue that satisfies both criteria has probability larger than $1 - 1/2^k$ where k is the number of prime factors of n [Sho94, Section 6].
A summary of this approach is given in Algorithm 2

Algorithm 2: Shor's Reduction

```
Input  : An appropriately composite $n$; integer parameters $b$ and $c$
Output: A factorization of $n$
1 for $2 \leq g \leq n-1$ do
2     if $\gcd(g, n) \neq 1$ then
3         return $\gcd(g, n), n/\gcd(g, n)$
4     Otherwise, $g$ is a unit modulo $n$
5     Use Stange's algorithm to compute the order $r$ of $g \bmod n$ with parameters $b$ and $c$
6     if $r \bmod 2 = 0$ and $g^{r/2} \neq -1 \bmod n$ then
7         return $\gcd(g^{r/2} - 1, n), n/\gcd(g^{r/2} - 1, n)$
```

To showcase the algorithm, we apply it to $n = 9983$ and consider the residue $g = 2$. As seen before, Stange's algorithm outputs order 4884 which is even. Furthermore

$$2^{4884/2} \bmod 9983 = 2^{2442} = 2681 \neq 1 \bmod 9983.$$

Therefore, we apply Proposition 7:

$$\gcd(2681 - 1, 9983) = \gcd(2680, 9983) = 67, \quad 9983/67 = 149$$

We obtain $n = 149 \cdot 67$ as a factorization of n.

5.2 Ekerå's Reduction

In [Eke21], Ekerå proposes a Monte Carlo polynomial-time reduction of the factorization problem to the order problem inspired by Miller's deterministic algorithm seen in Section 4. While Miller's algorithm requires the exact order of multiple residues to compute $h(n)$, Ekerå's algorithm approximates $h(n)$ as defined in Subsection 1.1 with the order of a single residue. Using this intuition we are able to factorize n.
Indeed, Ereká states that the order r of a residue g is likely to be such that $\lambda(n)/r$ is a moderate size product of small prime powers where $\lambda(n)$ is defined in Subsection 1.1. Therefore, it is possible to approximate the value of $\lambda(n)$ or the value of some multiple of $\lambda'(n)$ by multiplying r by small prime powers where $\lambda'(n)$ is defined in Subsection 1.1. Ekerå takes $c \cdot d(n)$ as an upper bound for the prime powers for some integer $c \geq 1$ that can be chosen [Eke21, Section 3.2]. For simplicity, we choose $c = 2$. In approximating a multiple of $\lambda'(n)$, we are approximating $h(n)$. In doing so, the algorithm factorizes n in a single run of the order-finding algorithm.
The algorithm is summarized in Algorithm 3.

Algorithm 3: Ekerå's Reduction

```
Input  : An appropriately composite $n$; integer parameters $b$ and $c$
Output: A factorization of $n$
1  Phase 1: Order finding
2  Choose a random residue $g \bmod n$
3  if $\gcd(g, n) \neq 1$ then
4      return $\gcd(g, n), n/\gcd(g, n)$
5  Otherwise, $g$ is a unit modulo $n$ and so the order $r$ can be computed using Stange's algorithm
     with parameters $b$ and $c$
6  Phase 2: Multiple of the order computation
7  Let $S$ be the set of all primes smaller or equal to $2d(n)$ and let $\eta(q) = \max\{x \in \mathbb{Z} : q^x \leq 2d(n)\}$
8  $R \longleftarrow r \prod_{q \in S} q^{\eta(q)}$
9  Phase 3: GCD computation
10 $z \longleftarrow R/2^{\nu_2(R)}$
11 for $2 \leq x \leq n-1$ do
12     if $\gcd(x, n) \neq 1$ then
13         return $\gcd(x, n), n/\gcd(x, n)$
14     Otherwise, $x$ is a unit modulo $n$
15     $i \longleftarrow 0$
16     while $i \leq \nu_2(R)$ and $x^{2^i z} \neq 1$ do
17         if $\gcd(x^{2^i z} - 1, n) \neq 1$ then
18             return $\gcd(x^{2^i z} - 1, n), n/\gcd(x^{2^i z} - 1, n)$
19         $i \longleftarrow i + 1$
```

The critical difference between this algorithm and Miller's resides in Phase 2. In computing R, we aim to obtain a multiple of $p-1$ for every prime $p|n$. Ekerå proves in [Eke21, Section 3.3.2, Theorem 1] that the probability of success of this algorithm is bounded from below by

$$1 - \left(\frac{C(k, 2)}{2^{n-1}} + \frac{1}{8 \log_2^2(2d(n)^2)} \right)$$

where l is defined as in Algorithm 3 and the integer to be factored is $n = \prod_{i=1}^{k} p_i^{e_i}$.
To demonstrate the algorithm, we apply it to $n = 9983$ and for simplicity assume that the randomly drawn residue is $g = 2$. As seen before, the output from Algorithm 1 is 4884. The value of $2d(n)$ is 28. The set S is $[2, 3, 5, 7, 11, 13, 17, 19, 23]$.
The values of η_{28} obtained are

$\eta_{28}(2) = 4$	$\eta_{28}(7) = 1$	$\eta_{28}(17) = 1$
$\eta_{28}(3) = 3$	$\eta_{28}(11) = 1$	$\eta_{28}(19) = 1$
$\eta_{28}(5) = 2$	$\eta_{28}(13) = 1$	$\eta_{28}(23) = 1$

The value of $\prod_{q \in S} q_i^{\eta(q_i)}$ is therefore 80313433200 and so r is 392250807748800. We decompose r into $r = z \cdot 2^{\nu_2(r)}$ where $z = 6128918871075$ and $2^{\nu_2(r)} = 64 = 2^6$.
We assume that the randomly drawn residue x is 3, which is a unit. The value of $x^z \bmod 9883$ is 7704. This then gives the trivial factor $\gcd(7704 - 1, 9883) = 1$. In the following iteration the value of $x^{z \cdot 2} \bmod 9883$ is 2681. This then gives $\gcd(2681 - 1, 9883) = 67$. This leads to the non-trivial factorization $9883 = 67 \times 149$.

5.3 Comparison of the Reductions

There are advantages and disadvantages to both reductions. Shor's Reduction is simple and straightforward but requires the order to have a specific form. It may necessitate multiple calls to an order finding algorithm. On the other hand, Ekerå's Reduction computes a single order of a residue. However, in obtaining a multiple of the order, the computations of $\eta(q)$ for each $q \in S$ and of the product $r \prod_{q \in S} q^{\eta(q)}$ requires much computational time. In our implementations, we use Stange's algorithm to compute the order of residues modulo n.
Furthermore, the probabilities of success of each reduction differ as the number of prime divisors increases. While the first reduction has increasing probabilities of successfully factoring $n = \prod_{i=1}^{k} p_i^{e_i}$ as the number of prime divisors increases, the opposite is true for the second reduction.
To experimentally compare the efficiency of the reductions, we implement the two reductions and record their respective running times when applied to a set of integers. We look at integers $n \in I$. We plot the average computing time in seconds needed for each reduction to factor each integer $n \in I$ in Figure 2 taking parameters $b = b_n$ for b_n defined as in Subsection 3.1 and $c = 10$.

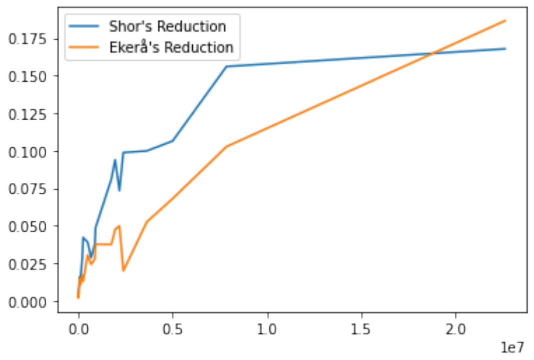


Figure 2. Running times of Shor's Reduction and Ekerå's Reduction

Figure 2 shows that despite differences, the two reductions have comparable runtime. In this comparison we see a heuristic confirmation of Ekerå's intuition that approximating $h(n)$ as a multiple of the order is sufficient to obtain a factorization, therefore making Miller's algorithm practical.

In the coming improvements applied to the algorithms, we refer to the reductions as outlined within this section as the standard form. We use the standard form to compare the improvement obtained.

6. Improving Factorization Reductions

Having established the two reductions we implement to compute a factorization of n, we consider ways to reduce the number of computations needed. Stange suggests computing a multiple of the order rather than the exact order [Sta23, Section 5]. This can be done by computing a single nontrivial α_i rather than several values. This is accomplished by testing for a nontrivial kernel as more relations are found. This can decrease the number of relations needed. Both reductions can be adapted to utilize a multiple of the order to compute a factorization and so this modification may prove to be beneficial.
Another way to improve the runtime is to consider using residues of odd orders. Indeed, Johnston [Joh17, Section 3] proves that Shor's Reduction, as seen in Algorithm 2, can be extended to compute a factorization with residues that do not have even order. While this is only applicable to Shor's Reduction, it is nonetheless of interest.
Furthermore, inspired by Leander's work [Lea22, Theorem 4], we use the Jacobi symbol to avoid residues that do not output nontrivial factors.

6.1 Single Kernel Algorithm

Within Algorithm 1, a great deal of the computational runtime is devoted to the relation finding phase. Attempting to find multiple B-smooth powers of g requires a great number of iterations. While computing the exact order of a residue requires $b + c$ relations, computing a multiple of the order requires less multiplicative relations. In fact, it only requires a single kernel basis vector. To compute such a vector, the algorithm requires less than b relations.
As we see in Algorithm 5 and Algorithm 6 later in this section, both reductions can be adapted to compute a nontrivial factorization using a multiple of the order rather than the exact order, this implementation can decrease the overall running time of the factorization algorithms.
We modify Algorithm 1 to check that the matrix F^T has a non-trivial kernel at every successful relation found. Once it has found one, it returns $\alpha = w \cdot \mathbf{a}$ where w is the basis vector of the kernel. The algorithm is summarized in Algorithm 4.

Algorithm 4: Single Kernel Algorithm (SK)

```
Input  : An appropriately composite $n$ and a residue $g$; integer parameter $b$
Output: A multiple of the multiplicative order of $g \bmod n$
1  Phase 1: Relation finding
2  $i \longleftarrow 0$
3  $L \longleftarrow 0$
4  while $L = 0$ do
5      Draw an integer $a_i$ in the range $\{1, \ldots, n\}$ (ensuring that $a_i$ has not been drawn previously)
6      Compute the smallest positive residue of $g^{a_i} \bmod n$
7      if $g^{a_i} = \prod_{j=1}^{b} p_j^{f_{i,j}}$ then
8          Add $[f_{i,1}, \ldots, f_{i,b}]$ as the $i$-th row of $F^T$
9          Add $a_i$ as the $i$-th entry of vector $\mathbf{a}$
10         $i \longleftarrow i + 1$
11         $L = nullity(F^T)$
12 Phase 2: Linear algebra
13 Compute a nontrivial basis vector $w$ in the kernel of $F^T$
14 $\alpha \longleftarrow \mathbf{a} \cdot w$
15 return $\alpha$
```

While less multiplicative relations are computed, the algorithm computes the nullity of F^T for every relation found, increasing the running time. Nonetheless, the decreased number of relations needed gives an improved running time compared to the original algorithm as seen in Figure 3, obtained by plotting the average running time needed to compute the order of a random residue g modulo the integers $n \in I$ with parameters $b = b_n$ for b_n defined as in Subsection 3.1.

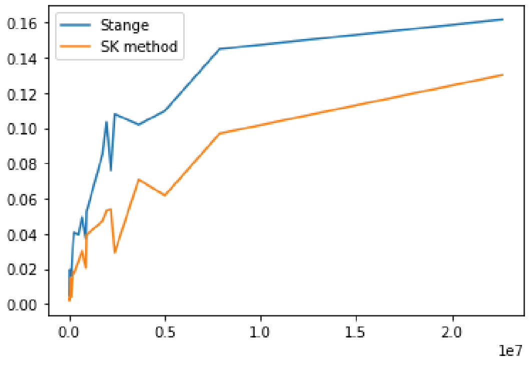


Figure 3. Running times of Algorithm 1 and Algorithm 4

Both reductions presented in Section 5 can be modified to utilize a multiple of the order of $g \bmod n$ to correctly output factorization. We consider the first reduction. We adapt it using the following proposition.

Proposition 8. *Given a residue g, some integer m and even order $r = \mathrm{ord}_n(g)$,*

$$g^{r/2} \not\equiv \pm 1 \bmod n \iff g^{m \cdot r/2^{\nu_2(m)+1}} \not\equiv \pm 1 \bmod n$$

Proof. Consider the order r of a residue g and any integer $m = z \cdot 2^{\nu_2(m)}$.
First note that for any $1 \leq i \leq \nu_2(m)$,

$$g^{r \cdot m/2^i} = (g^r)^{z \cdot 2^{\nu_2(m)-i}} = (1)^{z \cdot 2^{\nu_2(m)-i}} = 1 \bmod n$$

and for $i = \nu_2(m) + 1$,

$$g^{r \cdot m/2^{\nu_2(m)+1}} = (g^{r/2})^z \bmod n.$$

($\Longleftarrow$) If $g^{r/2} = \pm 1 \bmod n$, then $(g^{r/2})^z = (\pm 1)^z = \pm 1 \bmod n$. Hence, by contraposition

$$(g^{r/2})^z \neq \pm 1 \bmod n \implies g^{r/2} \neq \pm 1 \bmod n.$$

($\Longrightarrow$) If $g^{r/2} = y \bmod n$, then $(g^{r/2})^z = y^z \bmod n$.

Claim: If $y \neq \pm 1 \bmod n$ then $y^z \neq \pm 1 \bmod n$
In the case $z = 1$, $y^1 = \pm 1 \bmod n$ directly gives a contradiction.
For $z \neq 1$, note that

$$\mathrm{ord}_n(y) = \mathrm{ord}_n(g^{r/2}) = \frac{\mathrm{ord}_n(g)}{\gcd(\mathrm{ord}_n(g), r/2)} = \frac{r}{\gcd(r, r/2)} = \frac{r}{r/2} = 2$$

and so

$$(-y)^2 = y^2 = 1 \bmod n \implies \mathrm{ord}_n(-y)|2 \implies \mathrm{ord}_n(-y) \in \{1, 2\}$$

However, $\mathrm{ord}_n(-y) = 1$ gives $y = -1$ which is a contradiction.
Therefore, $\mathrm{ord}_n(-y) = 2$ and $\mathrm{ord}_n(y) = 2$.
Hence

$$y^z = 1 \bmod n \implies \mathrm{ord}_n(y)|z \implies 2|z$$

and similarly

$$y^z = -1 \bmod n \implies (-y)^z = -y^z = 1 \bmod n \implies \mathrm{ord}_n(-y)|z \implies 2|z.$$

The integer z is the odd part of m so we have a contradiction in both cases.

Therefore, $(g^{r/2})^z \neq \pm 1 \bmod n$.
Hence

$$g^{r/2} = y \neq \pm 1 \bmod n \implies (g^{r/2})^z \neq \pm 1 \bmod n$$

which concludes the proof. □

We therefore modify Shor's Reduction, as described in Algorithm 2, to be able to use a multiple of the order to compute a nontrivial factorization. This is done by computing $g^{m \cdot r/2^i}$ for $i = 0, 1, 2, \ldots$ until $g^{m \cdot r/2^i} \neq \pm 1$ is reached, revealing that $i = \nu_2(m) + 1$, assuming r is even. Proposition 7 is then applied to obtain a nontrivial factor $\gcd(g^{m \cdot r/2^i} - 1, n)$. This is summarized in Algorithm 5.

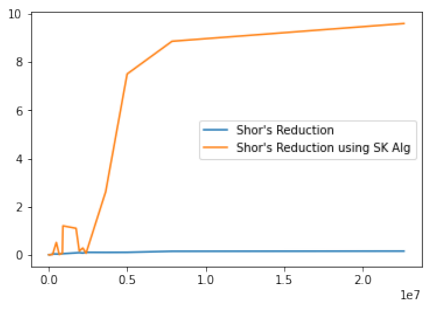
Algorithm 5: Shor's Reduction using Algorithm 4

```
Input  : An appropriately composite $n$; integer parameter $b$
Output: A factorization of $n$
1  for $2 \leq g \leq n - 1$ do
2      if $\gcd(g, n) \neq 1$ then
3          return $\gcd(g, n), n/\gcd(g, n)$
4      Otherwise, $g$ is a unit modulo $n$
5      Use Algorithm 4 to compute a nontrivial multiple $R$ of the order $r$ of $g \bmod n$ with
         parameter $b$
6      if $R \bmod 2 = 0$ then
7          while $g^R = 1 \bmod n$ do
8              $R \longleftarrow R/2$
9          if $g^R \neq \pm 1 \bmod n$ then
10             return $\gcd(g^R - 1, n), n/\gcd(g^R - 1, n)$
```

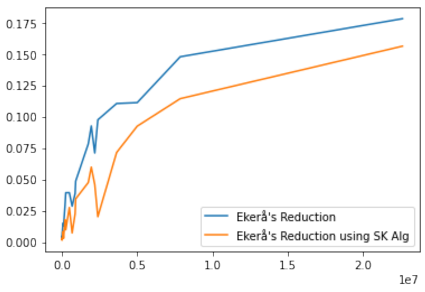
If we consider Ekerå's Reduction, Algorithm 3 approximates $h(n)$ as defined in Subsection 1.1 as multiple of the order of g and produces one by multiplying the order by small prime powers. By computing a multiple of the order, we skip Phase 2 of Algorithm 3 altogether as seen in Algorithm 6.
Figure 4 is obtained by plotting the average running time needed to compute the a nontrivial factorization of integers $n \in I$ with parameters $b = b_n$ for b_n defined as in Subsection 3.1. We see in Figure 4 that this modification does not affect both reductions equally. In fact, it is only an improvement for Ekerå's Reduction. Indeed, within Algorithm 4, while the computation of the multiple of the order requires less time, it is offset by the iterations needed to reach $i = \nu_2(m) + 1$. As for Ekerå's Reduction, computing a multiple of the order removes the need to compute $\eta(q)$ for each $q \in S$. This reduction in computations is beneficial for the running time.

Algorithm 6: Ekerå's Reduction using Algorithm 4

```
Input  : An appropriately composite $n$; integer parameter $b$
Output: A factorization of $n$
1  Phase 1: Order finding
2  Choose a random residue $g \bmod n$
3  if $\gcd(g, n) \neq 1$ then
4      return $\gcd(g, n), n/\gcd(g, n)$
5  Otherwise, $g$ is a unit modulo $n$ and so the multiple $R$ of the order $r$ is be computed using
     Algorithm 4 with parameter $b$.
6  $z \longleftarrow R/2^{\nu_2(R)}$
7  Phase 2: GCD computation
8  for $2 \leq x \leq n-1$ do
9      if $\gcd(x, n) \neq 1$ then
10         return $\gcd(x, n), n/\gcd(x, n)$
11     Otherwise, $x$ is a unit modulo $n$
12     $i \longleftarrow 0$
13     while $i \leq \nu_2(R)$ and $x^{2^i z} \neq 1$ do
14         if $\gcd(x^{2^i z} - 1, n) \neq 1$ then
15             return $\gcd(x^{2^i z} - 1, n), n/\gcd(x^{2^i z} - 1, n)$
16         $i \longleftarrow i + 1$
```



(a) Alg. 4 applied to Shor's Reduction



(b) Alg. 4 applied to Ekerå's Reduction

Figure 4. Running times of Shor's Reduction and Ekerå's Reduction once Alg. 4 has been applied

To quantify such decrease, we consider the computational complexity of Phase 2 of Algorithm 3.

Proposition 9. *Phase 2 of Algorithm 3 has computational complexity $O(d(n))$.*

Proof. Within Phase 2, the algorithm computes the maximal power p^x such that $p^x < 2d(n)$ for all primes in the set S where S is the set of all primes smaller or equal to $2d(n)$. The algorithm then computes the product $r \prod_{p \in S} p^x$.
Assume that p^x is computed naively by multiplying the prime p repeatedly until $p^{x+1} > 2d(n)$ is

obtained[2]. Doing so requires computing

$$O(\lceil \log_p(2d(n)) \rceil)$$

multiplications. Since this is done for each $p \in S$, we must repeat these multiplications $|S|$ times. As this step depends on p, we consider $\lceil \log_2(2d(n)) \rceil$ with $p = 2$ as an upper bound for the number of computations.
The size of S is the number of primes that are smaller or equal to $2d(n)$. Using Theorem 2, we approximate the size of S as

$$\frac{2d(n)}{\ln(2d(n))}.$$

Hence, Phase 2 is computed in at most

$$O(\frac{2d(n)}{\ln(2d(n))} \log_2(2d(n)))$$

multiplications. Using properties of logarithms we obtain that

$$\begin{aligned}\frac{2d(n)}{\ln(2d(n))} \log_2(2d(n)) &= 2d(n) \frac{\log_2(e) \ln(2d(n))}{\ln(2d(n))} \\ &= 2d(n) \log_2(e) = \frac{2d(n)}{\ln(2)}\end{aligned}$$

Hence, Phase 2 has computational complexity $O(\frac{2}{\ln(2)} d(n)) = O(d(n))$. □

Therefore, Algorithm 6 has lower computational complexity than Algorithm 3 by $O(d(n))$.
Due to these results, Algorithm 4 should not be applied to Shor's Reduction as it does not improve the run time while it should be applied to Ekerå's Reduction.

6.2 Using Odd Orders

Proposition 7 relies on finding an even order. This means that residues with odd order must be discarded in Shor's Reduction. However, it is also possible to use residues that have odd orders. Indeed, the proposition can be extended to odd orders as suggested in [Joh17, Section 3]. The results are summarized in Proposition 10.

Proposition 10. *Given an appropriately composite $n = \prod_{i=1}^{k} p_i^{e_i}$, suppose $x \neq 1 \bmod n$ and $x^d = 1 \bmod n$ for an odd prime d that does not divide n. Then $\gcd(x-1, n)$ is a nontrivial factor of n with probability $1 - 1/d^k$.*

Proof. This proof follows [Joh17, Section 3] and [Joh17, Section 4]. Since $x^d = 1 \bmod n$, we can obtain the equality $x^d = 1 \bmod p^e$ for any prime power p^e that divides n. Hence, $\mathrm{ord}_{p^e}(x) | d$ which implies that $\mathrm{ord}_{p^e}(x)$ is either d or 1 since d is prime.
Let A be the product of the prime powers for which $\mathrm{ord}_{p^e}(x) = 1$, i.e. $x = 1 \bmod p^e$, and B be the product of the prime powers q^l for which $\mathrm{ord}_{q^l}(x) = d$, i.e. $x = m \neq 1 \bmod q^l$. We then have that $n = A \cdot B$ for A and B coprime with $x = 1 \bmod A$ and $x = m \neq 1 \bmod B$.
The Chinese Remainder Theorem ensures that the map $\varphi : \mathbb{Z}/n\mathbb{Z} \longrightarrow (\mathbb{Z}/A\mathbb{Z}) \times (\mathbb{Z}/B\mathbb{Z})$ where

$$\varphi(x \bmod n) = [x \bmod A, x \bmod B]$$

[2]Other methods, such as square-and-multiply algorithms compute exponentiation more efficiently. For simplicity we do not consider those in our complexity analysis.

is an isomorphism and so

$$x \bmod n \mapsto [1 \bmod A, m \bmod B] \implies x - 1 \bmod n \mapsto [0 \bmod A, m - 1 \bmod B].$$

Hence, $A|(x-1)$ and $B \nmid (x-1)$. Therefore

$$\gcd(x-1, n) = \gcd(x-1, A) \cdot \gcd(x-1, B) = A \cdot \gcd(x-1, B).$$

Suppose q^l is a prime power such that $q^l|(x-1)$ and $q^l|B$. Since $q^l|(x-1)$, $x = 1 \bmod q^l$ holds. However by construction of B, it cannot be that $x = 1 \bmod q^l$. Hence, $\gcd(x-1 \bmod n, B)$ cannot be divisible by any prime power and so $x - 1 \bmod n$ and B must be coprime. Hence

$$\gcd(x-1, n) = A.$$

We now consider the probability that the algorithm outputs a trivial factorization, i.e. $A = n$ or $A = 1$.

If $A = n$, then that means that $x = 1 \bmod p_i^{e_i}$ for each i. By the Chinese Remainder Theorem, this has the solution $x = 1 \bmod n$ up to congruence. However, $x \neq 1 \bmod n$ and so $A = n$ does not occur.

If $A = 1$, we then have $B = n$ and so $x \bmod p_i^{e_i}$ has order d for all $i = 1, 2, \ldots, k$. Hence, $d|\phi(p_i^{e_i}) = p_i^{e_i-1}(p-1)$ for each i. The integer d is prime so $d|p_i^{e_i-1}$ or $d|(p-1)$. If $d|p_i^{e_i-1}$ then $d = p_i|n$ which is a contradiction.
Then it must be that $d|(p_i - 1)$ for all i. For $d > 2$, $d|(p_i - 1)$ has probability $1/d$ for each i. Since the events $d|(p_i - 1)$ for each i are mutually independent, the probability that $A = 1$ is given by the product of the probability that $d|(p_i - 1)$ for each $i = 1, 2, \ldots, k$. Hence, $A = 1$ occurs with probability $1/d^k$.

Hence, $\gcd(x-1, n)$ has probability $1/d^k$ of being a trivial factor of $n = \prod_{i=1}^k p_i^{e_i}$. □

Proposition 10 can be used to reduce the *factorization problem* to the *order problem*. Indeed, if we find an odd order $\mathrm{ord}_n(g) = r$ and a prime factor d of the order r, then $x = g^{r/d}$ satisfies the requirements of Proposition 10 as long as $d \nmid n$. If that were the case, however, we would have a factor of n. Otherwise we use the prime divisors d of r to compute the factor $\gcd(g^{r/d} - 1, n)$ with a probability that tends to one as d or the number of prime divisors of n increase.
Implementing this proposition for all prime divisors of the orders requires computing these primes. This is however impractical. Grosshans et al suggest in [Gro+17, Section IV.B.] to check by trial division whether the order is divisible by small primes $p \in \{2, 3, 5\}$. We therefore modify Shor's Reduction as described in Algorithm 7.
We compare the performance of Shor's Reduction once this variant has been applied in Figure 5 over the integers I with parameters $b = b_n$ for b_n defined as in Subsection 3.1 and $c = 10$. We indeed see that there is a great decrease in the running time. In particular, integers that are divisible by the primes $p \in \{2, 3, 5\}$ have a near zero runtime due to the construction of the algorithm. Since Proposition 10 can only be applied if $p \nmid n$, we must factor out such integers before attempting to compute the order of residues modulo n. As such, integers that have $p \in \{2, 3, 5\}$ as prime divisors are factorized immediately. A more fair comparison of the two reductions is done by only considering integers that are not divisible by such values of p.
Therefore, we compare the efficiency of this variant over integers

$$I_1 := \{86129, 112157, 187493, 802879, 1438051, 1553143, 2444393, 6044779, 33713471, 49010591\}$$

that do not have $2, 3$ and 5 as divisors as done in Figure 6 with the same parameter values as before. Nonetheless, we see an improvement as allowing for odd order to be used for factoring reduces the number of calls to the order finding algorithm needed as expected.

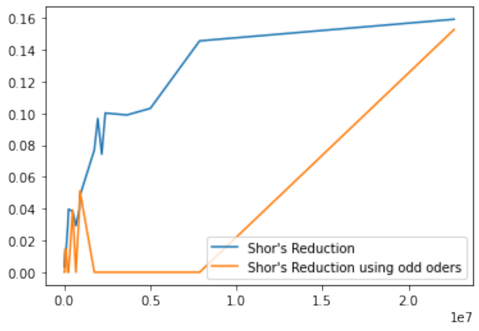


Figure 5. Alg. 7 applied to Shor's Reduction over Integers I

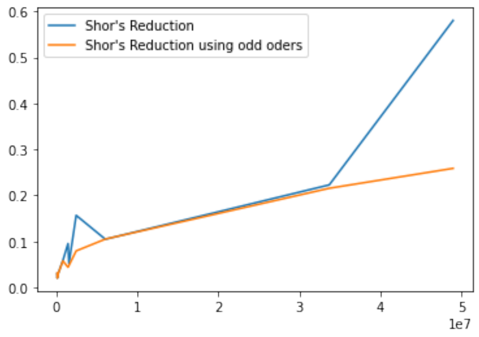


Figure 6. Alg. 7 applied to Shor's Reduction over integers I_1

Algorithm 7: Shor's Reduction using Proposition 7

```
Input  : An appropriately composite $n$; integer parameters $b$ and $c$
Output: A factorization of $n$
for $p \in \{2, 3, 5\}$ do
    if $p | n$ then
        return $p, n/p$
for $2 \leq g \leq n - 1$ do
    if $\gcd(g, n) \neq 1$ then
        return $\gcd(g, n), n/\gcd(g, n)$
    Otherwise, $g$ is a unit modulo $n$
    Use Stange's algorithm to compute the order $r$ of $g \bmod n$ with parameter $b$ and $c$
    if $r \bmod 2 = 0$ and $g^{r/2} \neq -1 \bmod n$ then
        return $\gcd(g^{r/2} - 1, n), n/\gcd(g^{r/2} - 1, n)$
    else if $r \bmod 3 = 0$ and $g^{r/3} \neq -1 \bmod n$ then
        return $\gcd(g^{r/3} - 1, n), n/\gcd(g^{r/3} - 1, n)$
    else if $r \bmod 5 = 0$ and $g^{r/5} \neq -1 \bmod n$ then
        return $\gcd(g^{r/5} - 1, n), n/\gcd(g^{r/5} - 1, n)$
```

6.3 Employing the Jacobi symbol

The Jacobi symbol is an extension of the Legendre symbol to composite odd integers. Leander's work [Lea22, Theorem 4] shows that for an integer $n = pq$ with p and q primes and for $g \bmod n$ with negative Jacobi symbol, i.e. $\left(\frac{g}{n}\right) = -1$, the probability that the order of g is even and

$$g^{\mathrm{ord}_b(g)/2} \neq -1 \bmod n$$

is at least $\frac{3}{4}$.
While the integers we consider are not of the form required, we can nonetheless use the Jacobi symbol to decrease the running time of both reductions. Indeed, due to [Coh93, Section 4, Chapter 1, Algorithm 1.4.12], computing the Jacobi symbol can be done without computing prime factors, with complexity analogous to Euclid's algorithm. This can be done using various properties of the Jacobi symbol such as the law of quadratic reciprocity. Therefore, using the Jacobi symbol to avoid computations can be advantageous.
Indeed, it can be used in Shor's Reduction to decrease the calls to the order computing function. To avoid computing the order of a residue that is revealed to be odd and therefore not suitable, we use the following proposition:

Proposition 11. *If n is appropriately composite and $a \in (\mathbb{Z}/n\mathbb{Z})^\times$,*

$$\left(\frac{a}{n}\right) = -1 \implies \mathrm{ord}_n(a) \text{ is even}$$

Proof. Let $r = \mathrm{ord}_n(a)$ and suppose r is odd. Then

$$1 = \left(\frac{1}{n}\right) = \left(\frac{a^r}{n}\right) = \left(\frac{a}{n}\right)^r.$$

Since r is odd we must have

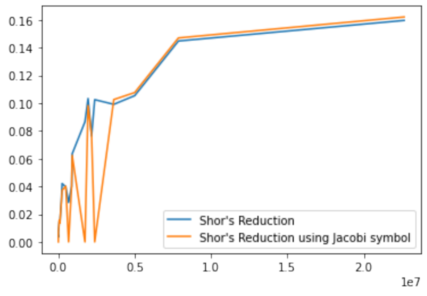
$$1 = \left(\frac{a}{n}\right)^r = \left(\frac{a}{n}\right)$$

and so $\left(\frac{a}{n}\right) = 1$. By contraposition, the proposition holds. □

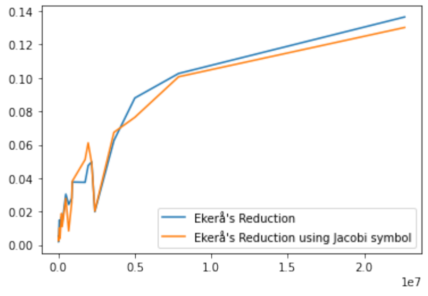
Therefore, by computing the Jacobi symbol $\left(\frac{a}{n}\right)$, we ensure that the order is even without the need to compute it. To implement this idea in Shor's Reduction, after we draw a residue $g \bmod n$, we compute the Jacobi symbol. If we find that $\left(\frac{g}{n}\right) = 1$, we draw another residue. Otherwise, we proceed with computing the order and follow the rest of the algorithm as usual.
As for Ekerå's Reduction, as described in Algorithm 3, the proof of Theorem 3 shows that the algorithm searches for the smallest quadratic nonresidue modulo some prime $p|n$. To guarantee only such residues are considered, we filter out quadratic residues modulo n. Indeed, for $n = \prod_{i=1}^{k} p_i^{e_i}$,

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i} = -1 \implies \left(\frac{a}{p_i}\right) = -1 \text{ for at least one } p|n.$$

By doing so we guarantee that the residues considered are quadratic nonresidues for at least one prime divisor of n. To implement this idea in Ekerå's Reduction, we compute the Jacobi symbol of each residue $x \bmod n$ drawn in "GCD computation" phase.



(a) Alg. 4 applied to Shor's Reduction



(b) Alg. 4 applied to Ekerå's Reduction

Figure 7. Running times of Reductions 1 and Ekerå's Reduction once Alg. 4 has been applied

We apply this approach to both reductions over the integers I and compare the run times to each of the standard reductions in Figure 7 with parameters $b = b_n$ for b_n defined as in Subsection 3.1 and $c = 10$. Within Shor's Reduction in Figure 7a, we observe that for some integers the algorithm has a runtime very close to zero.
Indeed, by avoiding negative Jacobi symbol residues the algorithm is more likely to draw residues g that are not in $(\mathbb{Z}/n\mathbb{Z})^\times$ and so computes factors $\gcd(g, n)$, $n/\gcd(g, n)$. This is done without computing the order of a residue. Since this factorization relies on finding small factors of n essentially at random, it does not apply for all integers. For integers that do not obtain a factorization this way, we see that using the Jacobi symbol allows to decrease the run time occasionally.
On the other hand, Figure 7b shows that, while not drastically changing the behaviour of the runtimes, avoiding positive Jacobi symbol residues allows for consistent small decreases in the runtime.

7. Conclusion

In this thesis, we consider Stange's algorithm [Sta23, Section 2, Algorithm 2.2] to solve the *order problem*. This algorithm, strongly inspired by the Index Calculus algorithm, searches for residues that are B-smooth to form multiplicative relations modulo n over a factor base. These relations are then used to obtain a relation matrix and compute the order of a residue in $(\mathbb{Z}/n\mathbb{Z})^\times$.
To reduce the running time of Stange's algorithm, we consider values of the parameter b which represents the number of primes in the factor base. The computational complexity of the algorithm is given by $O(b^3 + b^2 u^u)$ where $u = \log_B(n)$. We find that $b = L_n(1/2, \sqrt{2}/2)$ minimizes the expression. Therefore, we find that the minimal complexity of the algorithm is $L_n(1/2, 3\sqrt{2}/2)$.
Following Stange's suggestion [Sta23, Section 5], we also consider using a linear sieve to efficiently compute smooth residues. As the LSM is commonly used in the Index Calculus, it has potential to be useful within Stange's algorithm [Can+23]. However, they are likely to form a relation matrix that cannot output a nontrivial order. Due to this characteristic of the LSM, it is not recommended to combine it with Stange's algorithm. Further research is needed to determine the theoretical reason of this behaviour and whether the Gaussian Integer Sieve, an additional sieving method commonly used in the Index Calculus, leads to similar results.
Miller's algorithm [Mil76, Section "Tests For Primality", Definition of A_f] demonstrates that the *factorization problem* is polynomial time Turing reducible to the *order problem*. Despite its relevance for the purpose of proving theoretical reduction, Miller's algorithm is not implemented as it requires computing the order of many residues.
More efficient reductions are considered. Shor's Reduction utilizes a proposition inspired by Shor. To obtain nontrivial factors of an integer n, it requires the order of a residue to be even. As a consequence, it may require multiple calls to Stange's order finding algorithm. On the other hand, Ekerå's Reduction [Eke21], strongly inspired by Miller's algorithm, only requires a single call to Stange's order finding algorithm. However the subsequent computations to obtain nontrivial factors are more involved than those in Shor's Reduction. Nonetheless, the two reductions result in comparable running times in our implementations.
To reduce the number of multiplicative relations needed, we considered utilizing a multiple of the order rather than the exact order as suggested by Stange [Sta23, Section 5]. This method decreases the number of relations computed as it only requires a nontrivial kernel of F^T. While this method decreases the running time of Stange's algorithm, it does not affect both factorization reductions equally. As the reductions are modified to compute a factorization from a multiple of the order, Shor's Reduction increases in computational complexity while Ekerå's Reduction decreases.
We examine an extension of Shor's Reduction that allows for residues with odd orders to also output a factorization suggested by [Joh17, Section 3]. In particular we consider orders that are divisible by $p \in \{2, 3, 5\}$. This allows for a factorization to be found with a reduced number of calls to the order finding algorithm. This extension decreases the running time as expected, especially for integers that are divisible by such small primes.
Finally, inspired by [Lea22, Theorem 4], we employ the Jacobi symbol. By computing the Jacobi symbol of a residue, which can be done efficiently, it is possible to determine whether the residue can produce a factorization within both Shor's Reduction and Ekerå's Reduction. Indeed, both reductions can be improved upon if we only consider residues with negative Jacobi symbol value.

A possible avenue for further research to improve on these algorithms is to consider other methods to obtain smooth relations. Methods such as the Elliptic Curve Method, the Number Field Sieve and Gaussian Integers Sieve are briefly discussed in this thesis but not fully explored. They may lead to more efficient implementation of Stange's order finding algorithm.

References

[Ank52] Nesmith Ankeny. “The least quadratic non residue”. In: *Annals of Mathematics* 55 (1952), pp. 65–72. URL: https://api.semanticscholar.org/CorpusID:120759136.

[Bac84] Carl Eric Bach. “Analytic Methods in the Analysis and Design of Number-Theoretic Algorithms”. AAI8512746. PhD thesis. 1984.

[Ber98] DJ Bernstein. “Detecting perfect powers in essentially linear time”. In: *Mathematics Of Computation* 67.223 (July 1998), pp. 1253–1283. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-98-00952-1.

[Can+23] John Cannon et al. *Handbook Of Magma Functions*. 2.28. Available at https://magma.maths.usyd.edu.au/magma/handbook/text/214, Accessed on 06/01/2024. Computational Algebra Group, School of Mathematics and Statistics, University of Sydney. July 2023.

[Coh93] Henri Cohen. “A course in computational algebraic number theory”. In: *Graduate texts in mathematics*. Springer Berlin, Heidelberg, 1993. URL: https://api.semanticscholar.org/CorpusID:118037646.

[COS86] Don Coppersmith, Andrew M. Odlzyko, and Richard Schroeppel. “Discrete Logarithms In GF(p)”. In: *Algorithmica* 1.1–4 (Jan. 1986), pp. 1–15. ISSN: 0178-4617. DOI: 10.1007/BF01840433. URL: https://doi.org/10.1007/BF01840433.

[DM99] Abhijit Das and C. E. Veni Madhavan. “Performance Comparison of Linear Sieve and Cubic Sieve Algorithms for Discrete Logarithms over Prime Fields”. In: *Algorithms and Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 295–306. ISBN: 978-3-540-46632-1.

[Eke21] Martin Ekerå. “On completely factoring any integer efficiently in a single run of an order-finding algorithm”. In: *Quantum Information Processing* 20.6 (June 2021), p. 205. ISSN: 1573-1332. DOI: 10.1007/s11128-021-03069-1. URL: https://doi.org/10.1007/s11128-021-03069-1.

[Gau14] Pierrick Gaudry. *Integer factorization and discrete logarithm problems*. University Lecture Notes. 2014. URL: https://cristal.univ-lille.fr/jncf2014/files/lecture-notes/gaudry.pdf.

[Gor93] Daniel M. Gordon. “Discrete Logarithms in GF(P) Using the Number Field Sieve”. In: *SIAM Journal on Discrete Mathematics* 6.1 (1993), pp. 124–138. DOI: 10.1137/0406010. eprint: https://doi.org/10.1137/0406010. URL: https://doi.org/10.1137/0406010.

[Gre03] Jonathan Gregg. “On Factoring Integers and Evaluating Discrete Logarithms”. In: Harvard College Cambridge, Massachusetts, 2003. URL: https://api.semanticscholar.org/CorpusID:2064410.

[Gro+17] Frédéric Grosshans et al. *Factoring Safe Semiprimes with a Single Quantum Query*. 2017. arXiv: 1511.04385 [quant-ph].

[HH21] David Harvey and Joris van der Hoeven. “Integer multiplication in time $O(n\log n)$”. In: *Annals of Mathematics* 193.2 (2021), pp. 563–617. DOI: 10.4007/annals.2021.193.2.4. URL: https://doi.org/10.4007/annals.2021.193.2.4.

[Joh17] Anna M. Johnston. “Shor’s Algorithm and Factoring: Don’t Throw Away the Odd Orders”. In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 83. URL: https://api.semanticscholar.org/CorpusID:12911205.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896842.

[Lea22] Gregor Leander. *Improving the Success Probability for Shor’s Factoring Algorithm*. 2022. arXiv: quant-ph/0208183 [quant-ph].

[LLS15] Youness Lamzouri, Xiannan Li, and Kannan Soundararajan. “Conditional bounds for the least quadratic non-residue and related problems”. In: *Mathematics of Computation* 84.295 (July 2015), pp. 2391–2412.

[LO91] B. A. LaMacchia and A. M. Odlyzko. "Computation of discrete logarithms in prime fields". In: *Designs, Codes and Cryptography* 1.1 (May 1991), pp. 47–62. ISSN: 1573-7586. DOI: 10.1007/BF00123958. URL: https://doi.org/10.1007/BF00123958.

[Mil76] Gary L. Miller. "Riemann's hypothesis and tests for primality". PhD thesis. 1976, pp. 300–317. DOI: https://doi.org/10.1016/S0022-0000(76)80043-8. URL: https://www.sciencedirect.com/science/article/pii/S0022000076800438.

[Ols23] Jan-Fredrik Olsen. "An Operator Theoretic Approach to the Prime Number Theorem". English. In: *Journal Of Mathematical Physics Analysis Geometry* 19.1 (2023), pp. 172–177. ISSN: 1812-9471. DOI: 10.15407/mag19.01.172.

[Sah12] Chandan Saha. *IISC Computational Number Theory and Algebra Lecture Notes 21: The Index Calculus method.* Accessed on 10/01/2024. 2012. URL: https://www.csa.iisc.ac.in/~chandan/courses/CNT/notes/lec21.pdf.

[Sho94] Peter W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124–134. URL: https://api.semanticscholar.org/CorpusID:15291489.

[Sta23] Katherine E. Stange. "Factoring using multiplicative relations modulo n: a subexponential algorithm inspired by the index calculus". In: *Mathematical Cryptology* 3.2 (Oct. 2023), pp. 2–10. URL: https://journals.flvc.org/mathcryptology/article/view/134295.

[Ste11] G.Krantz Steven. *The Proof is in the Pudding. SubtitleThe Changing Nature of Mathematical Proof.* Springer New York, NY, 2011. DOI: https://doi.org/10.1007/978-0-387-48744-1.

[Stu02] Chris Studholme. *UToronto Research Paper: The Discrete Log Problem.* Accessed on 10/01/2024. 2002. URL: http://www.cs.toronto.edu/~cvs/dlog/research_paper.pdf.

[Sut21] Andrew Sutherland. *MIT Mathematics 18.783, Elliptic Curves Lecture Notes 11: Index calculus, smooth numbers, and factoring integers.* Accessed on 10/01/2024. 2021. URL: https://math.mit.edu/classes/18.783/2021/LectureNotes10.pdf.

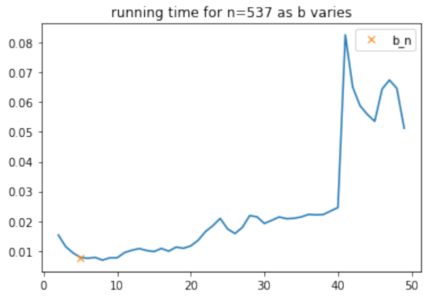
8. Appendix

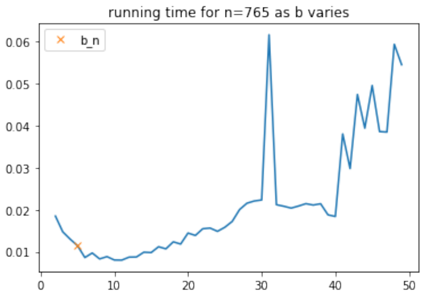
For the integers

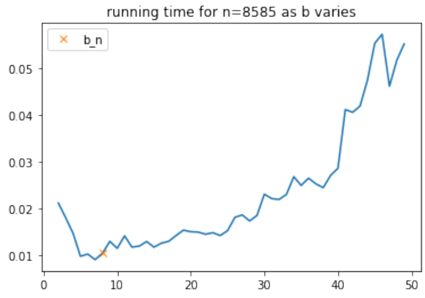
$$[537,\ 765,\ 8585,\ 19053,\ 61453,\ 101371,\ 199989,\ 247251,\ 484957,\ 671015,\ 871933,\ 907423,\ 1744953,\ 1946725,\ 2177645,\ 2381625,\ 3632763,\ 5001635,\ 7865455,\ 22653803].$$

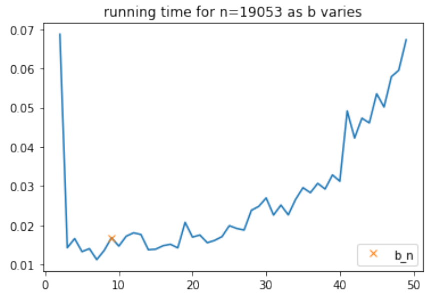
the following graphs are plotted.

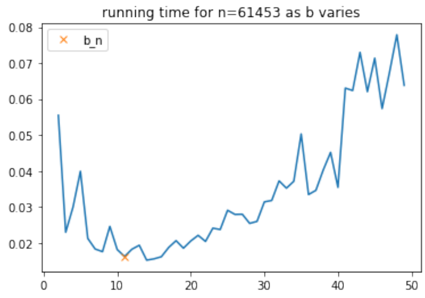
8.1 Subsection 3.1

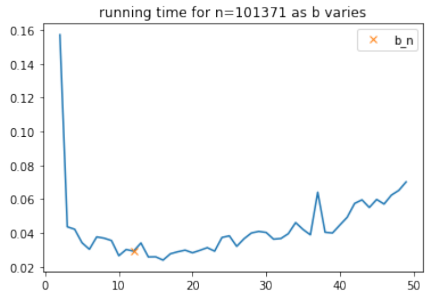


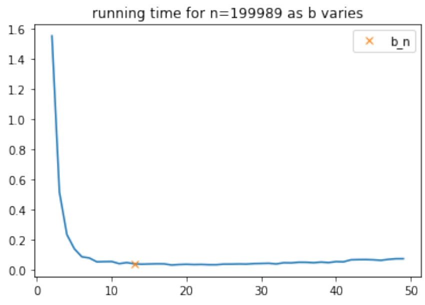


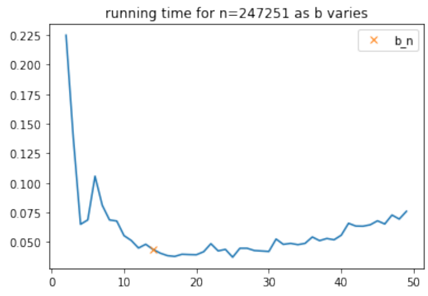


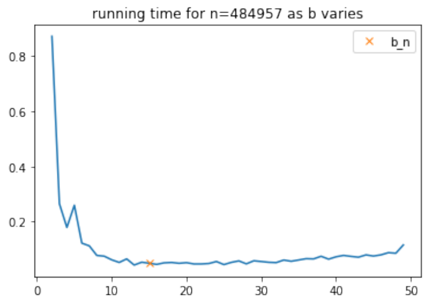


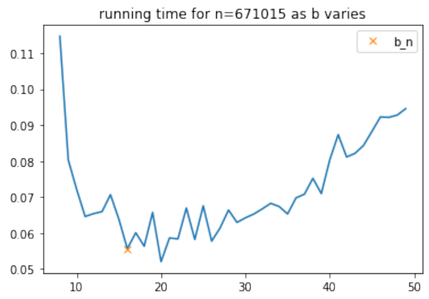


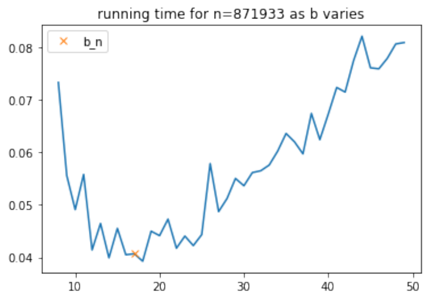


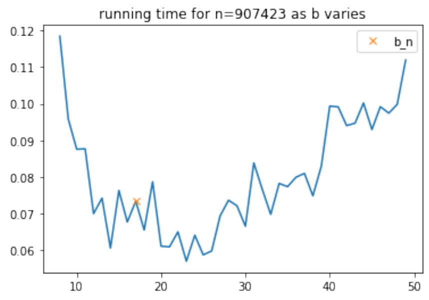


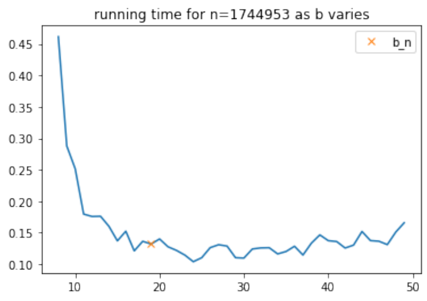


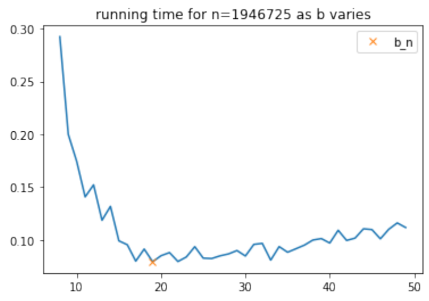

running time for n=484957 as b varies
b_n

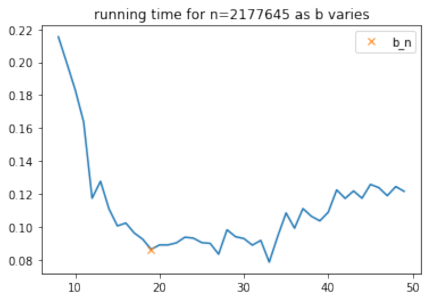

running time for n=671015 as b varies
b_n

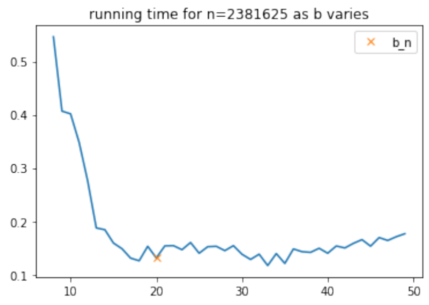

running time for n=871933 as b varies
b_n

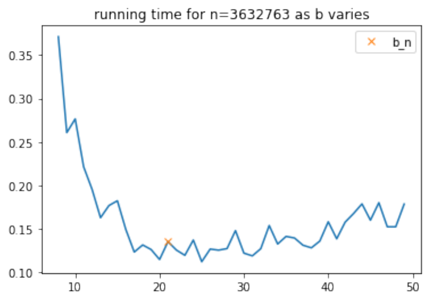

running time for n=907423 as b varies
b_n

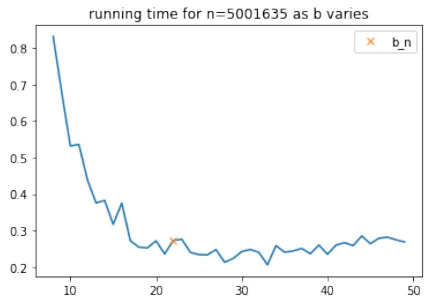

running time for n=1744953 as b varies
b_n

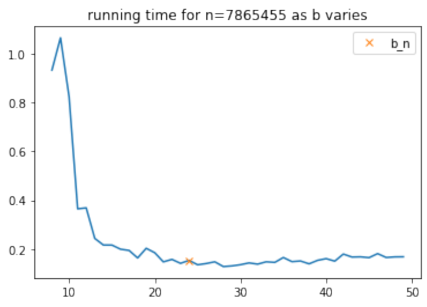

running time for n=1946725 as b varies
b_n


running time for n=2177645 as b varies
b_n


running time for n=2381625 as b varies
b_n


running time for n=3632763 as b varies
b_n


running time for n=5001635 as b varies
b_n



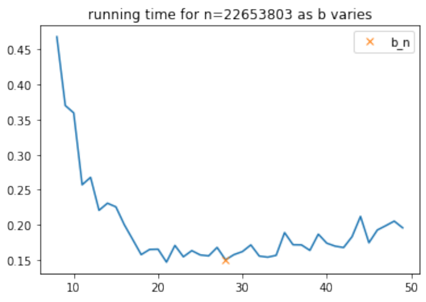


Figure 10. Running time of Stange's algorithm for various values of b for fixed n

8.2 Subsection 5.3

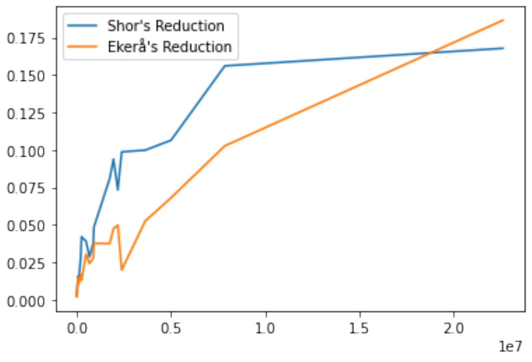


Figure 11. Running times of Shor's Reduction and Ekerå's Reduction

8.3 Subsection 6.1

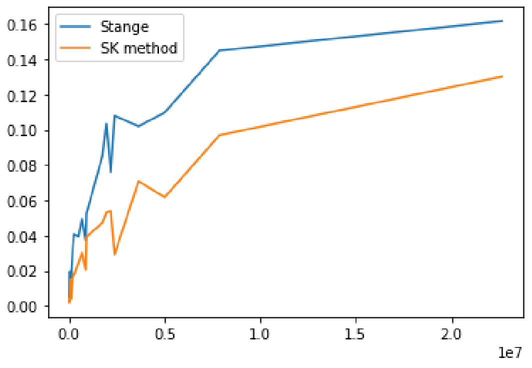
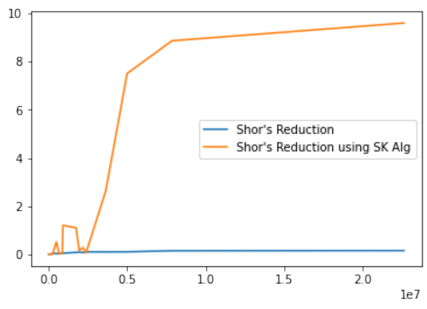
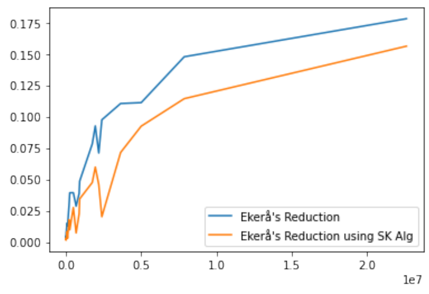


Figure 12. Running times of Algorithm 1 and Algorithm 4



(a) Alg. 4 applied to Shor's Reduction



(b) Alg. 4 applied to Ekerå's Reduction

Figure 13. Running times of Shor's Reduction and Ekerå's Reduction once Alg. 4 has been applied

8.4 Subsection 6.2

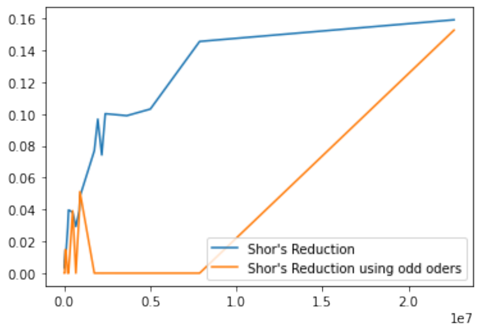


Figure 14. Alg. 7 applied to Shor's Reduction over integers I

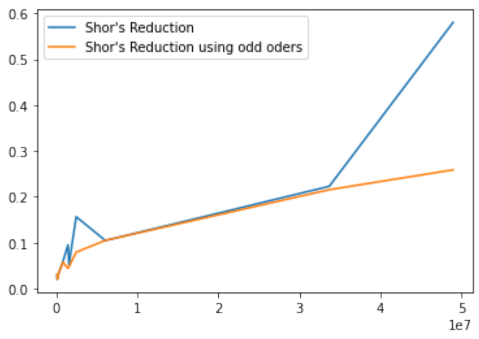
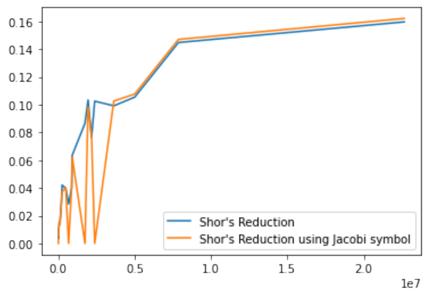
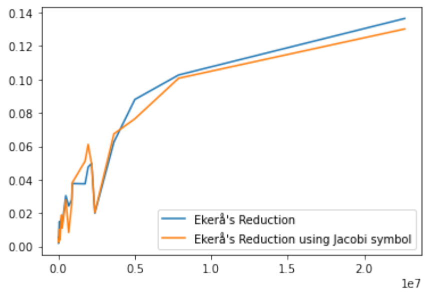


Figure 15. Alg. 7 applied to Shor's Reduction over integers I_1

8.5 Subsection 6.3



(a) Alg. 4 applied to Shor's Reduction



(b) Alg. 4 applied to Ekerå's Reduction

Figure 16. Running times of Shor's Reduction and Ekerå's Reduction once Alg. 4 has been applied