A low-resource quantum factoring algorithm

Daniel J. Bernstein[1,2], Jean-François Biasse[3], and Michele Mosca[4,5,6]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, NL
djb@cr.yp.to
[3] University of South Florida
Department of Mathematics and Statistics
biasse@usf.edu
[4] Institute for Quantum Computing and Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada
mmosca@uwaterloo.ca
[5] Perimeter Institute for Theoretical Physics, Waterloo, Ontario, Canada
[6] Canadian Institute for Advanced Research, Toronto, Ontario, Canada

Abstract. In this paper, we present a factoring algorithm that, assuming standard heuristics, uses just $(\log N)^{2/3+o(1)}$ qubits to factor an integer N in time $L^{q+o(1)}$ where $L = \exp((\log N)^{1/3}(\log\log N)^{2/3})$ and $q = \sqrt[3]{8/3} \approx 1.387$. For comparison, the lowest asymptotic time complexity for known pre-quantum factoring algorithms, assuming standard heuristics, is $L^{p+o(1)}$ where $p > 1.9$. The new time complexity is asymptotically worse than Shor's algorithm, but the qubit requirements are asymptotically better, so it may be possible to physically implement it sooner.

1 Introduction

The two main families of public-key primitives in widespread use today rely on the presumed hardness of the RSA problem [22] or the discrete-logarithm

Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was supported by the Commission of the European Communities through the Horizon 2020 program under project 645622 (PQCRYPTO), by the U.S. National Science Foundation under grants 1018836 and 1314919, by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005, by NIST under grant 60NANB17D, by the Simons Foundation under grant 430128; by NSERC; by CFI; and by ORF. IQC and the Perimeter Institute are supported in part by the Government of Canada and the Province of Ontario. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: d4969875ec8996389d6dd1271c032a204f7bbc42. Date: 2017.04.19.

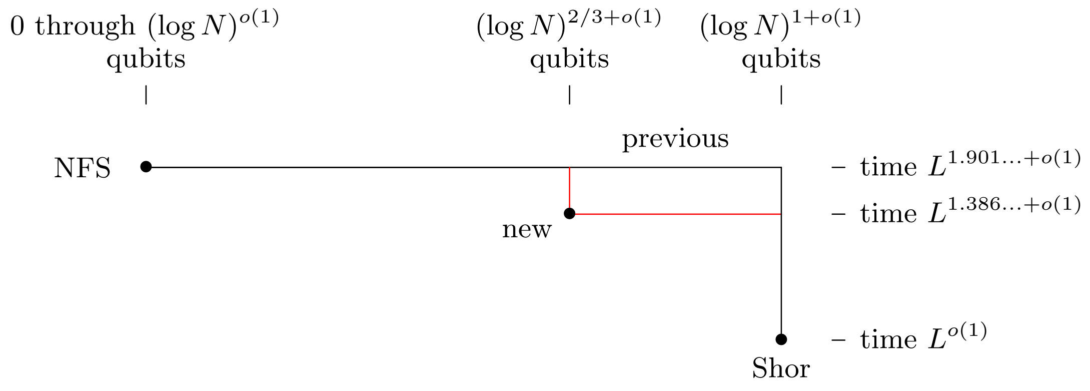
0 through $(\log N)^{o(1)}$ qubits
$(\log N)^{2/3+o(1)}$ qubits
$(\log N)^{1+o(1)}$ qubits
previous
NFS
new
Shor
time $L^{1.901...+o(1)}$
time $L^{1.386...+o(1)}$
time $L^{o(1)}$

Fig. 1. Tradeoffs between factorization time and number of logical qubits; i.e., evolution of time as more and more qubits become available.

problem [13] respectively. Shor's algorithm [25] provides an efficient solution to both the factorization problem and the discrete-logarithm problem, thus breaking these primitives, assuming that the attacker has a large general-purpose quantum computer.

Shor's algorithm has motivated a new field of research, post-quantum cryptography, consisting of cryptographic primitives designed to resist quantum attacks. It is clear that the main public-key primitives will have to be replaced before the practical realization of large-scale quantum computers. However, the precise time line remains an important open question that can have significant economic consequences. In particular, the community needs to predict the point in time when quantum computers will threaten commonly deployed RSA key sizes, whether through Shor's algorithm or any other quantum factoring algorithm.

An obvious obstruction to the implementation of Shor's algorithm is the number of qubits necessary to run it. The number of qubits used by Shor's algorithm is $\Theta(\log N)$, where N is the integer being factored; i.e., the number of qubits grows linearly with the number of bits in N. There has been some effort to reduce the Θ constant; see, e.g., [27], [4], [24], [3], and [26].

1.1. Contributions of this paper. We present a factoring algorithm that, assuming standard heuristics, uses a *sublinear* number of qubits, specifically $(\log N)^{2/3+o(1)}$ qubits, to factor N in time $L^{q+o(1)}$ where $q = \sqrt[3]{8/3} \approx 1.387$ and $L = \exp((\log N)^{1/3}(\log \log N)^{2/3})$.

To put this in perspective: The lowest asymptotic time complexity for known pre-quantum (0-qubit) factoring algorithms, assuming standard heuristics, is $L^{p+o(1)}$ where $p = \sqrt[3]{92 + 26\sqrt{13}}/3 \approx 1.902$. This exponent p is from a 1993 algorithm by Coppersmith [12], slightly improving upon the exponent $\sqrt[3]{64/9} \approx 1.923$ from [18].

The new time complexity is asymptotically worse than Shor's algorithm, but the qubit requirements are asymptotically better, so it may be possible to physically implement the new algorithm sooner than Shor's algorithm.

The fact that we use fewer qubits than Shor's algorithm *for all sufficiently large key sizes* does not answer the question of whether we use fewer qubits than Shor's algorithm to break, e.g., common 2048-bit RSA keys. Optimization of exact qubit requirements for this algorithm is a challenging open problem.

1.2. Discrete logarithms. The same idea can also be used for multiplicative-group discrete logarithms. (On the other hand, the idea has no obvious impact upon the number of qubits needed for elliptic-curve discrete logarithms.)

Specifically, the idea of NFS has been adapted to solving discrete-logarithm problems in the multiplicative group of any prime field. See [14, 23] for early work and [2] for the latest optimizations.

The first stage of these algorithms computes discrete logarithms of many small numbers in the field. The best pre-quantum complexity known for this stage is $L^{p+o(1)}$. Here $p \approx 1.902$ as before, and the N used in defining L is replaced by the number of elements of the field. The idea of our factoring algorithm adapts straightforwardly to this context, reducing the cost of the first stage to $L^{q+o(1)}$, where $q \approx 1.387$ as before.

The second stage deduces the discrete logarithm of the target. This stage takes time $L^{d+o(1)}$ where $d \approx 1.232$. If many discrete-logarithm problems are posed for the same field then this second stage is the bottleneck (since the first stage is reused for all targets), and we have not found a way to speed up this stage using sublinear quantum resources. On the other hand, if there are relatively few targets then the first stage is the bottleneck.

There is a fast-moving literature (see, e.g., [20]) on pre-quantum techniques to solve discrete-logarithm problems in the multiplicative group of *extension fields*. We expect our approach to combine productively with these techniques, but we have not attempted to analyze the details or the resulting costs.

1.3. Notes on fault tolerance. Our primary cost metrics are time and the number of *logical* qubits. Beware, however, that an improved tradeoff between these metrics does not guarantee an improved tradeoff between time and the number of *physical* qubits.

In what Gottesman calls the "standard version" (see [15]) of the threshold theorem for fault-tolerant quantum computing, a logical circuit using Q qubits and containing T gates is converted into a fault-tolerant circuit using $Q(\log QT)^{O(1)}$ physical qubits. This bound is too weak to say anything useful about our algorithm: for us $\log T$ is $(\log N)^{1/3+o(1)}$, so all the bound says is that the resulting fault-tolerant circuit uses $(\log N)^{O(1)}$ physical qubits.

Gottesman in [15] introduced a different approach to fault-tolerant quantum computing, encoding Q logical qubits as just $O(Q)$ physical qubits, without much overhead in the number of qubit operations. However, Gottesman's analysis is focused on the case that T is in $Q^{O(1)}$. While extending the analysis to larger T may yield useful results in terms of quantum overhead, it is important to note that Gottesman explicitly disregards the cost of pre-quantum computation (for decoding error-correcting codes), while we take all computations into account.

To factor in time $L^{q+o(1)}$ with a sublinear number of physical qubits, it would be enough to encode Q logical qubits as, e.g., $Q^{1.49+o(1)}$ physical qubits, with time overhead at most $\exp(Q^{0.49+o(1)})$ and with logical error rate at most $1/\exp(Q^{0.51+o(1)})$. We leave this as another challenge.

1.4. Notation. We use the standard abbreviation "$\mathbb{Z}/M$" for the quotient $\mathbb{Z}/M\mathbb{Z}$.

2 Factoring integers with NFS

The number-field sieve (NFS) is a factoring method introduced by Pollard [21] and subsequently improved by many authors. NFS produced the $L^{p+o(1)}$ asymptotic speed record mentioned above; it was also used for the latest RSA factorization record, the successful factorization of a 768-bit RSA modulus [17]. Our $L^{q+o(1)}$ algorithm, described in Section 3, uses quantum techniques to accelerate the relation-collection step in NFS.

This section gives a high-level description of NFS. For simplicity we restrict attention to the version of NFS introduced by Buhler, Lenstra, and Pomerance in [10], without the subsequent multi-field improvement [12] from Coppersmith (which does not seem to produce a better exponent in our context).

NFS begins as follows. Assume that N is an odd positive integer. Compute $m = \lfloor N^{1/d} \rfloor$; here $d \geq 2$ is an integer parameter optimized below. Assume that $N > 2^{d^2}$; then $N < 2m^d$ by [10, Proposition 3.2]. Write N in base m as $m^d + c_{d-1}m^{d-1} + \cdots + c_1 m + c_0$ where each of $c_{d-1}, \ldots, c_1, c_0$ is between 0 and $m-1$. Define $f = X^d + c_{d-1}X^{d-1} + \cdots + c_1 X + c_0 \in \mathbb{Z}[X]$, so that $f(m) = N$. Check whether f is irreducible; if not then the factorization of f immediately reveals a nontrivial factorization of N, as noted in [10, Section 3].

Let α be a root of f, and let ϕ be the ring homomorphism $\sum_i a_i \alpha^i \mapsto \sum_i a_i m^i$ from $\mathbb{Z}[\alpha]$ to $\mathbb{Z}/N$. Find, as explained below, a nontrivial set S of pairs (a, b) such that the following two properties hold simultaneously:

$$\text{"rational side":} \qquad \prod_{(a,b)\in S} (a + bm) \text{ is a square } X^2 \text{ in } \mathbb{Z},$$

$$\text{"algebraic side":} \qquad f'(\alpha)^2 \prod_{(a,b)\in S} (a + b\alpha) \text{ is a square } \beta^2 \text{ in } \mathbb{Z}[\alpha].$$

Then compute $Y = \phi(\beta)$. Note that $Y^2 = \phi(\beta^2) = \phi(f'(\alpha))^2 \prod \phi(a + b\alpha) = f'(m)^2 \prod (a + bm) = (f'(m)X)^2$ in $\mathbb{Z}/N$ since $\phi(a + b\alpha) = a + bm$ in $\mathbb{Z}/N$. Check whether $\gcd\{N, Y - f'(m)X\}$ is a nontrivial factor of N.

NFS actually produces many sets S at negligible extra cost, leading to many such potential factorizations. Conjecturally every odd positive integer N is factored by this procedure into products of prime powers.

2.1. Finding squares on the rational side. Consider first the simpler problem of finding S such that $\prod_{(a,b)\in S}(a + bm)$ is a square. NFS handles this as follows.

Define an integer as "y-smooth" when it is not divisible by any primes $>y$. Here the "smoothness bound" y is a parameter optimized below.

Find many y-smooth integers of the form $a+bm$, and combine these y-smooth integers to form a square. More specifically, search through the space

$$U = \{(a, b) : \quad a, b \in \mathbb{Z}, \quad \gcd\{a, b\} = 1, \quad |a| \le u, \quad 0 < b \le u\},$$

where u is another parameter optimized below. For each $(a, b) \in U$ such that $a + bm$ is y-smooth, factor $a + bm$ as $(-1)^{e_0} p_1^{e_1} \cdots p_B^{e_B}$ where $p_1 < \cdots < p_B$ are the primes $\le y$, and compute the exponent vector

$$e(a, b) = (e_0 \bmod 2, \ldots, e_B \bmod 2) \in \mathbb{F}_2^{B+1}.$$

If there are at least $B + 2$ such pairs (a_i, b_i) then the vectors $e(a_i, b_i)$ must have a nontrivial linear dependency: linear algebra reveals bits $x_i \in \mathbb{F}_2$, not all zero, such that $\sum_i x_i e(a_i, b_i) = 0$ in $\mathbb{F}_2^{B+1}$, which directly yields a square $\prod_{i:x_i \ne 0}(a_i + b_i m)$.

2.2. Finding squares on the algebraic side. The search for S such that $f'(\alpha)^2 \prod_{(a,b)\in S}(a + b\alpha)$ is a square is handled similarly.

Define $g(a, b) = (-b)^d f(-a/b) = a^d - c_{d-1}a^{d-1}b + \cdots + c_1 a(-b)^{d-1} + c_0(-b)^d$. Search for pairs (a, b) in the same space U such that $g(a, b)$ is y-smooth.

There is a standard definition of an exponent vector $e'(a, b) \in \mathbb{F}_2^{B'+B''}$ for any such pair (a, b). This vector has the following properties: if $f'(\alpha)^2 \prod_{(a,b)\in S}(a+b\alpha)$ is a square then $\sum_{(a,b)\in S} e'(a, b) = 0$; conversely, if $\sum_{(a,b)\in S} e'(a, b) = 0$ then $f'(\alpha)^2 \prod_{(a,b)\in S}(a + b\alpha)$ is a square, assuming standard heuristics; the vector length $B' + B''$, like $B + 1$, is approximately $y/\log y$; and e' is not difficult to compute. See [10, Sections 5 and 8] for the detailed definition of e', involving ideals and quadratic characters of $\mathbb{Z}[\alpha]$; the point of $g(a, b)$ is that $\mathcal{N}(a + b\alpha) = g(a, b)$, where $\mathcal{N}$ is the norm map from $\mathbb{Z}[\alpha]$ to $\mathbb{Z}$.

2.3. Overall algorithm. Algorithm 1 combines all of these steps. It searches through U for pairs (a, b) such that both $a + bm$ and $g(a, b)$ are y-smooth, i.e., such that $(a + bm)g(a, b)$ is y-smooth. If there are enough such pairs (a, b) then linear algebra finds a nontrivial linear dependency between the vectors $(e(a, b), e'(a, b)) \in \mathbb{F}_2^{B+1+B'+B''}$, i.e., a set S of pairs (a, b) such that both $\prod_{(a,b)\in S}(a + bm)$ and $f'(\alpha)^2 \prod_{(a,b)\in S}(a + b\alpha)$ are squares.

By generating some further pairs (a, b) one obtains more linear dependencies, obtaining further sets S as noted above. For simplicity we omit this refinement from the algorithm statement.

3 Accelerating NFS using quantum search

The main loop in Algorithm 1 searches for y-smooth integers $(a + bm)g(a, b)$, where (a, b) ranges through a set U of size $u^{2+o(1)}$. If the number of y-smooth integers $(a+bm)g(a, b)$ is at least $B+2+B'+B''$ then the algorithm is guaranteed

Algorithm 1 Conventional NFS

```
Input: Odd positive integer $N$ and parameters $d, y, u$ with $N > 2^{d^2}$.
Output: A divisor of $N$ (conjecturally often nontrivial when $N$ is not a prime power).
 1: Compute $m = \lfloor N^{1/d} \rfloor$.
 2: Write $N$ in base $m$ as $m^d + c_{d-1}m^{d-1} + \cdots + c_1 m + c_0$.
 3: Define $f = X^d + c_{d-1}X^{d-1} + \cdots + c_1 X + c_0 \in \mathbb{Z}[X]$.
 4: If $f$ has a proper factor $h$ in $\mathbb{Z}[X]$, return $h(m)$.
 5: Define $g(a, b) = a^d - c_{d-1}a^{d-1}b + \cdots + c_1 a(-b)^{d-1} + c_0(-b)^d$.
 6: for each $(a, b) \in \mathbb{Z} \times \mathbb{Z}$ with $\gcd\{a, b\} = 1$, $|a| \le u$, $0 < b \le u$ do
 7:     if $a + bm$ and $g(a, b)$ are $y$-smooth then
 8:         Compute the vector $(e(a, b), e'(a, b)) \in \mathbb{F}_2^{B+1+B'+B''}$.
 9:     end if
10: end for
11: If these vectors are linearly independent, return 1.
12: Find a nonempty subset $S$ of $\{(a, b)\}$ where the corresponding vectors have sum 0.
13: Compute $X = \sqrt{\prod_{(a,b)\in S}(a + bm)}$ and $\beta = \sqrt{f'(\alpha)^2 \prod_{(a,b)\in S}(a + \alpha b)}$.
14: return $\gcd\{N, \phi(\beta) - f'(m)X\}$.
```

to find a linear dependency, and conjecturally has a good chance of factoring N. This cutoff $B + 2 + B' + B''$ is in $y^{1+o(1)}$, and standard parameter choices are tuned so that there are in fact this many y-smooth values.

Algorithm 2 uses Grover's algorithm for the same search. Other steps of the algorithm remain unchanged. In this section we analyze the impact of this speedup upon the overall complexity of NFS.

The main appeal of this algorithm, compared to Shor's algorithm, is as follows. When NFS parameters are optimized, the number of bits in $(a+bm)g(a, b)$ is at most $(\log N)^{2/3+o(1)}$. With careful attention to reversible algorithm design (see Sections 4, 5, and 6) we fit the entire algorithm into $(\log N)^{2/3+o(1)}$ qubits. This is asymptotically *sublinear* in the length of N.

Note that our optimization here is for time. We would not be surprised if allowing a somewhat larger exponent of L in the time allows a constant-factor improvement in the number of qubits, but establishing this requires solving the challenging qubit-optimization problem mentioned in Section 1.

3.1. Complexity analysis. The following analysis shows, under the same heuristics used for previous NFS analyses, that the optimal time exponent q for this algorithm is $\sqrt[3]{8/3}$. As in the conventional NFS analysis by Buhler, Lenstra, and Pomerance [10], we choose

- $y \in L^{\beta+o(1)}$,
- $u \in L^{\epsilon+o(1)}$, and
- $d \in (\delta + o(1))(\log N)^{1/3}(\log\log N)^{-1/3}$,

where β, ϵ, δ are positive real numbers and $L = \exp((\log N)^{1/3}(\log\log N)^{2/3})$. Conventional NFS takes $\epsilon = \beta$, but we end up with ϵ larger than β; specifically, our optimization will produce $\beta = \sqrt[3]{1/3}$, $\epsilon = \sqrt[3]{9/8}$, and $\delta = \sqrt[3]{8/3}$.

Algorithm 2 New: NFS accelerated using quantum search

```
Input: Odd positive integer $N$ and parameters $d, y, u$ with $N > 2^{d^2}$.
Output: A divisor of $N$ (conjecturally often nontrivial when $N$ is not a prime power).
 1: Compute $m = \lfloor N^{1/d} \rfloor$.
 2: Write $N$ in base $m$ as $m^d + c_{d-1}m^{d-1} + \cdots + c_1 m + c_0$.
 3: Define $f = X^d + c_{d-1}X^{d-1} + \cdots + c_1 X + c_0 \in \mathbb{Z}[X]$.
 4: If $f$ has a proper factor $h$ in $\mathbb{Z}[X]$, return $h(m)$.
 5: Define $g(a, b) = a^d - c_{d-1}a^{d-1}b + \cdots + c_1 a(-b)^{d-1} + c_0(-b)^d$.
 6: Use Grover's algorithm to search for all $(a, b) \in \mathbb{Z} \times \mathbb{Z}$ with $\gcd\{a, b\} = 1$, $|a| \le u$,
    $0 < b \le u$ such that $a + bm$ and $g(a, b)$ are $y$-smooth.
 7: for each such $(a, b)$ do
 8:    Compute the vector $(e(a, b), e'(a, b)) \in \mathbb{F}_2^{B+1+B'+B''}$.
 9: end for
10: If these vectors are linearly independent, return 1.
11: Find a nonempty subset $S$ of $\{(a, b)\}$ where the corresponding vectors have sum 0.
12: Compute $X = \sqrt{\prod_{(a,b)\in S}(a + bm)}$ and $\beta = \sqrt{f'(\alpha)^2 \prod_{(a,b)\in S}(a + \alpha b)}$.
13: return $\gcd\{N, \phi(\beta) - f'(m)X\}$.
```

The quantities $a+bm$ and $g(a,b)$ that we wish to be smooth are bounded in absolute value by, respectively, $u + uN^{1/d} \le 2uN^{1/d}$ and $(d+1)N^{1/d}u^d$. Their product is thus bounded by $x = 2(d+1)N^{2/d}u^{d+1}$. Note that

$$\begin{aligned}\log x &= \log(2(d+1)) + \frac{2}{d}\log N + (d+1)\log u \\ &\in \left(\frac{2}{\delta} + \delta\epsilon + o(1)\right)(\log N)^{2/3}(\log\log N)^{1/3}.\end{aligned}$$

A uniform random integer in $[1, x]$ has y-smoothness probability $v^{-v(1+o(1))}$, where

$$v = \frac{\log x}{\log y} \in \frac{1}{\beta}\left(\frac{2}{\delta} + \delta\epsilon + o(1)\right)(\log N)^{1/3}(\log\log N)^{-1/3}.$$

We have $\log v \in (1/3 + o(1))\log\log N$ so this smoothness probability is

$$\exp(-(1+o(1))v\log v) = \exp\left(-\frac{1}{3\beta}\left(\frac{2}{\delta} + \delta\epsilon + o(1)\right)(\log N)^{1/3}(\log\log N)^{2/3}\right),$$

i.e., $L^{-(2/\delta+\delta\epsilon+o(1))/3\beta}$. We heuristically assume that the same asymptotic holds for the smoothness probability of the products $(a + bm)g(a, b)$.

The search space has size $u^{2+o(1)} = L^{2\epsilon+o(1)}$ and needs to contain $y^{1+o(1)} = L^{\beta+o(1)}$ smooth products. We thus need $2\epsilon - (2/\delta + \delta\epsilon)/3\beta \ge \beta$ for the algorithm to work as $N \to \infty$; i.e., we need $2 > \delta/3\beta$ and $\epsilon \ge (\beta + 2/3\beta\delta)/(2 - \delta/3\beta)$.

There is no point in taking ϵ larger than this cutoff, so we assume from now on that $\epsilon = (\beta + 2/3\beta\delta)/(2 - \delta/3\beta)$. (In this equality case we also need to take a large enough $o(1)$ for u to ensure enough smooth products, but this affects only the $o(1)$ in the final complexity.) The smoothness probability is now $L^{-2\epsilon+\beta+o(1)}$.

The conventional pre-quantum complexity analysis continues by saying that searching $L^{2\epsilon+o(1)}$ integers takes time $L^{2\epsilon+o(1)}$. We instead search with Grover's algorithm. Specifically, we partition the search space U in any systematic fashion into $L^{\beta+o(1)}$ parts, each of size $L^{2\epsilon-\beta+o(1)}$, each (with overwhelming probability) producing $L^{o(1)}$ smooth values. Grover's algorithm takes time $L^{\epsilon-\beta/2+o(1)}$ to search each part, for total time just $L^{\epsilon+\beta/2+o(1)}$.

Linear algebra takes time $L^{2\beta+o(1)}$. The pre-quantum-search exponent 2ϵ is balanced against 2β when $\epsilon = \beta$, i.e., $\beta^2 - \beta\delta/3 - 2/3\delta = 0$, forcing $\beta = (\delta + \sqrt{\delta^2 + 24/\delta})/6$ since $\delta - \sqrt{\delta^2 - 24/\delta}$ is negative. It is now a simple calculus exercise to see that taking $\delta = \sqrt[3]{3}$ produces the minimum $\beta = \sqrt[3]{8/9}$, satisfying the requirement $2 > \delta/3\beta$, and thus total time $L^{\sqrt[3]{64/9}+o(1)}$, roughly $L^{1.923}$.

Our quantum-search exponent $\epsilon+\beta/2$ is balanced against 2β when $\epsilon = 3\beta/2$, i.e., $\beta^2-\beta\delta/4-1/3\delta = 0$, forcing $\beta = (\delta+\sqrt{\delta^2 + 64/3\delta})/8$. This time the calculus exercise produces $\delta = \sqrt[3]{8/3}$ and the minimum $\beta = \sqrt[3]{1/3}$, again satisfying $2 > \delta/3\beta$, and thus total time $L^{\sqrt[3]{8/3}+o(1)}$, roughly $L^{1.387}$.

Note that a more realistic cost model for two-dimensional NFS circuits was used in [7], assigning a higher cost $L^{2.5\beta+o(1)}$ to linear algebra and ending up with exponent approximately 1.976 for conventional NFS. An analogous analysis of our algorithm ends up with exponent approximately 1.456.

4 A quantum relation search

This section presents an algorithm to find a λ-bit string s such that $F(s)$ is y-smooth. If many such strings exist then the algorithm makes a random choice; if no such string exists then the algorithm fails.

We assume that $F(s)$ is an integer between $-x$ and x for each λ-bit string s. We also assume that $\log y \in \Theta(\lambda)$; that $\log x \in (\log y)^{2+o(1)}$; and that the function F is computable by a reversible $(\log x)^{1+o(1)}$-bit circuit in time $2^{o(\lambda)}$.

Our time budget for the search algorithm is $2^{(0.5+o(1))\lambda}$. Our qubit budget is $(\log x)^{1+o(1)} = \lambda^{2+o(1)}$.

4.1. ECM as a subroutine. The usual pre-quantum approach is as follows. Lenstra's elliptic-curve method (ECM) [19], assuming standard heuristics and again assuming $\log x \in (\log y)^{2+o(1)}$, takes time $\exp((\log y)^{1/2+o(1)})$ and space $O(\log x)$ to find all primes $\le y$ dividing a nonzero input integer in $[-x, x]$. By trial division, within the same space, one sees whether the integer is y-smooth.

Generic techniques due to Bennett [5] convert any algorithm taking time T and space S into a reversible algorithm taking time $T^{1+\epsilon}$ and space $O(S \log T)$. For us $T^{1+\epsilon} \in y^{o(1)} = 2^{o(\lambda)}$ and $S \log T \in (\log x)(\log y)^{1/2+o(1)} = (\log x)^{5/4+o(1)}$. Applying Grover's algorithm then takes time $2^{(0.5+o(1))\lambda}$ using $(\log x)^{5/4+o(1)}$ qubits. This is beyond our budget. (The NFS application takes time $L^{q+o(1)}$ using $(\log N)^{5/6+o(1)}$ qubits, which meets our goal of sublinearity but is not as strong as we would like.)

4.2. Shor as a subroutine. To do better we replace the ECM subroutine with Shor's factoring method. We emphasize that here Shor is being applied only to integers between $-x$ and x; these are asymptotically much smaller than N.

Recall that, to find y-smooth integers $F(s)$, Grover's search algorithm uses a quantum circuit $U_{F,y}$ such that

- $U_{F,y}|s\rangle = -|s\rangle$ if $F(s)$ is y-smooth.
- $U_{F,y}|s\rangle = |s\rangle$ if $F(s)$ is not y-smooth.

This circuit does not need to be derived from a pre-quantum circuit; it can carry out quantum computations, such as Shor's algorithm. The main challenge is to minimize the number of qubits used to compute $U_{F,y}$, while staying within a $2^{o(\lambda)}$ time bound. Grover's algorithm then takes time $2^{(0.5+o(1))\lambda}$.

Section 5 explains how to apply Shor's algorithm to a superposition of odd positive integers, factoring with significant probability each integer that is not a power of a prime. Section 6 explains how to use Shor's algorithm repeatedly to recognize y-smooth integers.

4.3. Application to NFS. For our NFS application in Section 3, we choose an even integer λ so that $2^\lambda \in L^{2\epsilon-\beta+o(1)}$. We map λ-bit strings s to pairs (a, b) in a straightforward way, choosing a range of $2^{\lambda/2}$ consecutive integers a within $[-u, u]$ and a range of $2^{\lambda/2}$ consecutive integers b within $[1, u]$. We define x and y as in the previous section, and we define $F(s)$ as $(a + bm)g(a, b)$. The assumptions of this section are satisfied.

The algorithm in this section finds a, b in these ranges such that $(a+bm)g(a, b)$ is y-smooth. The algorithm takes time $2^{(0.5+o(1))\lambda} = L^{\epsilon-\beta/2+o(1)}$ and works with $(\log x)^{1+o(1)} = (\log N)^{2/3+o(1)}$ qubits as desired. Repeating the same algorithm $L^{o(1)}$ times finds all such pairs (a, b) with overwhelming probability. (This is overkill: Section 3 needs enough pairs to find a nontrivial linear dependency but does not need to find *all* pairs.) Repeating for $L^{\beta+o(1)}$ ranges covers all pairs (a, b) in the set U defined in the previous section. That set omits pairs (a, b) with $\gcd\{a, b\} > 1$; we simply discard such pairs.

5 Shor's factorization method in superposition

The conventional view is that Shor's algorithm is applied to *one* odd positive integer $M \in [1, x]$, obtaining a (hopefully nontrivial) divisor M_1 of M. We instead factor a *superposition* of inputs M, obtaining a *superposition* of divisors M_1 of M. This changes costs: in particular, Shor's original algorithm uses $(\log x)^{2+o(1)}$ qubits when it is run in superposition.

This section reviews the relevant features of Shor's algorithm, and explains a variant of the algorithm that fits into $(\log x)^{1+o(1)}$ qubits even when run in superposition. This section also explains a further variant (applicable to both the conventional case and the superposition case) that often finds more factors at the same cost.

5.1. Review of Shor's algorithm. Shor starts with some a coprime to M and precomputes $a^2 \bmod M$, $a^4 \bmod M$, $a^8 \bmod M$, etc., along with their inverses.

Shor then carries out a quantum computation, ending with a measurement, yielding an approximation $z/2^m$ of a fraction of the form k/r, where r is the order of a modulo M.

If m is chosen large enough, at least twice the number of bits of M, then, with probability $\Omega(1/\log\log M)$, the largest denominator below M in the continued fraction of $z/2^m$ will be exactly r. "Probability" here implicitly assumes that the random variable a is uniformly distributed in $(\mathbb{Z}/M)^*$.

One can switch to more reliable methods of finding r, improving the probability to $\Omega(1)$ at constant overhead, as discussed in, e.g., [25] and [11]. For us Shor's original method is adequate: the $\log\log$ factor is subsumed by $(\log x)^{o(1)}$.

Usually r is even. Shor then finishes by computing $M_1 = \gcd\{M, a^{r/2} - 1\}$.

5.2. Shor in superposition without many qubits. Using the same method to factor a superposition of inputs M means that we also need to execute the selection of a, the precomputation of $a^2 \bmod M$ etc., the continued-fraction computation, and the computation of $\gcd\{M, a^{r/2} - 1\}$ in superposition. We need to be careful here to fit these computations into our space budget, just $(\log x)^{1+o(1)}$ qubits.

As an example of what can go wrong, consider the seemingly trivial first step in typical statements of Shor's algorithm, namely generating an integer a uniformly at random between 1 and $M - 1$ (assuming $M > 1$). The textbook implementation of this step is rejection sampling: generate $b = \lceil \log_2 x \rceil$ random bits; interpret those bits as an integer R between 0 and $2^b - 1$; restart if $R \geq (M-1)\lfloor 2^b/(M-1) \rfloor$; compute $a = 1 + (R \bmod M - 1)$. The restart happens with probability $<1/2$, so on average <2 values of R are required.

The obvious way to handle a superposition of M is to choose in advance how many R's to generate, and to choose this number to be large, so that failures cannot be expected to occur. In the context of NFS, generating $(\log N)^{1/3+o(1)}$ values of R is adequate, for a total of $(\log N)^{1+o(1)}$ random bits. This might not sound like a problem, but storing this number of qubits is beyond our budget.

We instead generate one value of R and define $a = 1 + (R \bmod M - 1)$, skipping the rejection step. This cannot reduce the success probability of Shor's algorithm by more than a factor 2. We could bring this factor arbitrarily close to 1 by generating a few more bits in R, but a factor 2 is already not a problem for us: it is subsumed by $(\log x)^{o(1)}$ at the level of detail of our analysis.

Furthermore, there is no reason for us to store R in superposition: we use one R for all choices of M, so we do not need to spend qubits storing it. We do vary R across the multiple Shor calls explained in Section 6, so that each M is overwhelmingly likely to be factored; i.e., the function of M defined by our choice of the sequence of R is overwhelmingly likely to equal the desired function of M, recognizing whether or not M is y-smooth.

There is a more serious problem with the precomputation in Shor's algorithm: Shor uses a quadratic number, $(\log x)^{2+o(1)}$, of bits to store the sequence $a^2 \bmod M$, $a^4 \bmod M$, etc. This precomputation is important for Shor's method of computing $a^e \bmod M$ with e in superposition: namely, Shor uses the ith bit of

e to control a multiplication by $a^i \bmod M$ and then to control a multiplication by $1/a^i \bmod M$ (used to erase the previous temporary value).

We use, instead of Shor's strategy, a conventional pre-quantum "square and multiply" exponentiation algorithm taking time $(\log x)^{O(1)}$ and space $O(\log x)$. Bennett's generic conversion then produces a reversible algorithm taking time $(\log x)^{O(1)}$ and space $O(\log x \log\log x)$, i.e., space $(\log x)^{1+o(1)}$ as desired.

Finally, standard pre-quantum algorithms take time $(\log x)^{O(1)}$ to compute continued fractions, m-bit powers modulo M, gcd, and inverses mod M, all in space $O(\log x)$. Again generic conversion produces reversible algorithms for the same computations taking time $(\log x)^{O(1)}$ and space $O(\log x \log\log x)$.

5.3. Further factorization for free. We point out an easy tweak to Shor's algorithm that, starting from r, often finds more factors of an odd integer M in the same time (and space), and that is also much more reliable at separating any particular prime divisors of M from each other. (See also [16] for some other post-r tweaks that do not provide the same reliability but that sometimes help.)

Understanding this tweak requires a review of the probability that r produces a nontrivial divisor $M_1 = \gcd\bigl\{M, a^{r/2} - 1\bigr\}$ of M. Assume that M has prime factorization $p_1^{e_1} p_2^{e_2} \cdots p_f^{e_f}$, and write $(p_j - 1)p_j^{e_j - 1}$ as $2^{t_j} u_j$ where u_j is odd. By assumption M is odd so each $t_j \ge 1$. The group $(\mathbb{Z}/M)^*$ is isomorphic to the product of the groups $\mathbb{Z}/2^{t_1}, \mathbb{Z}/2^{t_2}, \ldots, \mathbb{Z}/2^{t_f}, \mathbb{Z}/u_1, \ldots, \mathbb{Z}/u_f$; choosing a uniform random element $a \in (\mathbb{Z}/M)^*$ is equivalent to choosing independent uniform random elements $x_1, x_2, \ldots, x_f, y_1, y_2, \ldots, y_f$ of these groups.

Write the order of x_j as 2^{c_j}, and write the order of y_j as d_j. The order r of a is then $2^{\max\{c_1,\ldots,c_f\}} d$, where d is an odd integer, namely $\operatorname{lcm}\{d_1, \ldots, d_f\}$. Note that c_j is t_j with probability $1/2$; $t_j - 1$ with probability $1/4$; and so on through 1 and 0, each with probability $1/2^{t_j}$. For any particular value of c_1, the chance that $c_2 = c_1$ is at most $1/2$, and similarly for c_3 etc., so the chance of all of $c_1, \ldots, c_f$ being identical is at most $1/2^{f-1}$.

Assume from now on that $c_1, \ldots, c_f$ are not identical. Then $\max\{c_1, \ldots, c_f\} > 0$ so r is even. By construction $a^r = 1$ in $(\mathbb{Z}/M)^*$, so $a^r = 1$ in $(\mathbb{Z}/p_j^{e_j})^*$, so $a^{r/2} = \pm 1$ in $(\mathbb{Z}/p_j^{e_j})^*$. The case $+1$ occurs exactly when $(r/2)x_j = 0$ in $\mathbb{Z}/2^{t_j}$, i.e., when $c_j < \max\{c_1, \ldots, c_f\}$; so $M_1 = \gcd\bigl\{M, a^{r/2} - 1\bigr\}$ is divisible by p_j exactly when $c_j < \max\{c_1, \ldots, c_f\}$.

In other words, computing M_1 splits the prime divisors p_j of M into two nonempty classes: those for which $c_j < \max\{c_1, \ldots, c_f\}$, and those for which $c_j = \max\{c_1, \ldots, c_f\}$. Hence M is nontrivially factored into M_1 and M/M_1.

Our tweak (inspired by "strong probable prime" tests [1]) is to compute

$$\begin{aligned}&\gcd\{M, a^{r/2} + 1\}, \gcd\{M, a^{r/4} + 1\}, \gcd\{M, a^{r/8} + 1\}, \ldots,\\ &\gcd\bigl\{M, a^d + 1\bigr\}, \gcd\bigl\{M, a^d - 1\bigr\}.\end{aligned}$$

These divisors of M have product exactly M and fit into essentially the same space as M. This splits the prime divisors into more classes, namely those for which $c_j = \max\{c_1, \ldots, c_f\}$, those for which $c_j = \max\{c_1, \ldots, c_f\} - 1$, those for which $c_j = \max\{c_1, \ldots, c_f\} - 2$, and so on, ending with those for which $c_j = 0$.

Shor's algorithm is unlikely to split p_i from p_j when t_i and t_j are significantly below $\max\{t_1, \ldots, t_f\}$. For example, if $f = 3$ and $(t_1, t_2, t_3) = (20, 3, 2)$, then c_1 is almost always larger than c_2 and c_3, so Shor's algorithm will almost always factor M into $p_1^{e_1}$ and $p_2^{e_2} p_3^{e_3}$. For our tweak, each pair (i, j) with $i \neq j$ has chance at least 50% of being split, since $c_i = c_j$ with probability at most 50%.

We also briefly mention a different way to avoid biases towards particular primes, namely to change the group used in Shor's algorithm from $(\mathbb{Z}/N)^*$ to a randomly selected elliptic-curve group $E(\mathbb{Z}/N)$. This is analogous to the change in [19] from the $p - 1$ factorization method to ECM.

6 Recognizing smooth integers

We present two constructions of fast quantum circuits $U_{F,y}$ usable in Section 4. Recall that the job of $U_{F,y}$, given a superposition of inputs s, is to recognize for each s whether $F(s) \in \{-x, \ldots, x\}$ is y-smooth.

6.1. Parallel construction. Starting with $M = F(s)$, declare non-smoothness if $M = 0$. Otherwise replace M by its absolute value, and remove all powers of 2 from M. From now on, M is an odd positive integer.

Use the tweaked version of Shor's algorithm presented in Section 5.3 to obtain a factorization of M into various divisors. Repeat this t times, where $t \in (\log x)^{o(1)}$ is a parameter chosen below, obtaining t factorizations of M. This consumes $(\log x)^{1+o(1)}$ qubits.

Use the algorithm of [8] to factor all the divisors, and thus also M, into coprimes. Use the algorithm of [6] (or, alternatively, [9]) to write each of the coprimes as a maximal power of a root. Note that if all the roots are $\leq y$ then M has been proven to be y-smooth; save one bit indicating whether this is the case. These algorithms take time and space $(\log x)^{1+o(1)}$, so reversible versions take time $(\log x)^{1+\epsilon+o(1)}$ and space $(\log x)^{1+o(1)}$.

If M is in fact y-smooth but this algorithm fails to prove it, then one of the roots is $>y$, and thus contains two distinct prime divisors p, q of M. This means that all t factorizations of M failed to split p from q.

There are at most $\log_2 M \leq \log_2 x$ prime divisors of M, and thus fewer than $(\log_2 x)^2$ pairs of distinct prime divisors. Fix a pair (p, q). Recall that each run of Shor's algorithm has probability $\Omega(1/\log\log x)$ of finding r. For the tweaked version, given that r is found, there is conditional probability $\geq 1/2$ of splitting p from q, and thus probability $\Omega(1/\log\log x)$ of splitting p from q. The probability of a failed split after t repetitions is thus $1/\exp(\Omega(t/\log\log x))$. Now let (p, q) vary: the total probability is below $(\log_2 x)^2/\exp(\Omega(t/\log\log x))$. We choose t just large enough for this probability bound to be below $1/4$; then $t \in (\log\log x)^{2+o(1)}$, so $t \in (\log x)^{o(1)}$ as claimed above.

We now uncompute everything except for the qubit indicating whether M was proven y-smooth. We then reuse the same temporary storage to repeat the entire procedure T times, accumulating T independent proof qubits. Together these qubits reliably indicate whether M is y-smooth; the failure probability is at most $1/4^T$. We take T as, e.g., $(\log N)^{1/2+o(1)}$, consuming only $(\log N)^{1/2+o(1)}$

qubits and reducing the failure probability to $1/\exp((\log N)^{1/2+o(1)})$. This failure probability can safely be ignored, since all our computations take time at most $\exp((\log N)^{1/3+o(1)})$.

6.2. Serial construction. As an alternative approach, we apply Shor's algorithm *serially.* First we use Shor's algorithm to split M into two factors, then we use Shor's algorithm to split the largest factor that remains, etc.

Like the parallel approach, this serial approach runs Shor's algorithm on a superposition of odd positive integers M, as explained in Section 5, after reducing to the odd case. Unlike the parallel approach, this serial approach does not need the tweak from Section 5.3: it is enough here to have a significant probability of splitting M whenever M is not a power of a prime. This serial approach also does not need factorization into coprimes as a subroutine.

As in the parallel approach, factoring M into factors $\le y$ proves that M is y-smooth; and it is sufficient to achieve, e.g., probability $3/4$ of finding a proof when a proof exists, since an outer loop can then amplify the proof-finding probability. An advantage of the serial approach is that this outer loop is unnecessary: we simply repeat Shor's algorithm enough times (see below) that every y-smooth input integer will, with overwhelming probability, be factored into factors $\le y$. The parallel approach could not afford the qubits for so many repetitions.

This serial approach requires care at three points. First, Shor's algorithm—as we have stated it—has no chance of factoring powers of primes. If the largest factor that remains is (e.g.) p^2, where p is prime, then Shor's algorithm will repeatedly try and fail to factor p^2. Postprocessing the list of factors to find powers, as in the parallel construction, will split p^2, but if the list also contains a product $qr > y$ then the overall algorithm will not recognize M as smooth.

To avoid this case we incorporate power detection into each run of Shor's algorithm. As noted above, there are pre-quantum power-detection algorithms that take time and space $(\log x)^{1+o(1)}$, so reversible versions take time $(\log x)^{1+\epsilon+o(1)}$ and space $(\log x)^{1+o(1)}$.

(Whether this case needs to be avoided is a different question. It seems unlikely that prime powers larger than y are common, and it seems unlikely that throwing away smooth numbers with such factors noticeably affects the performance of NFS. But we prefer to have subroutines that always work, so that such questions do not need to be asked.)

Second, we need to ensure that we have run Shor's algorithm enough times. A y-smooth positive integer $M \le x$ will have many factors: at least $(\log M)/\log y$, and perhaps as many as $\log_2 x$. A product $\le y$ does not need to be factored further, but Shor's algorithm will need to succeed many times before the largest factor is so small.

We maintain a list of integers >1 whose product is M. Initially this list contains simply M (unless $M = 1$, in which case the list is empty). An iteration of the algorithm uses power detection, followed by Shor's algorithm, to try to split the largest element of the list into ≥ 2 factors, each factor being >1. We define the iteration to be successful if either (1) this splitting succeeds—this happens fewer than $\log_2 x$ times, since each splitting increases the list size—or (2) the largest element of the list is prime.

Each iteration succeeds with probability $\Omega(1/\log\log x)$. Specifically: If the largest element of the list is prime, then the iteration succeeds by definition. If the largest element of the list is a square, cube, etc., then power detection succeeds. Otherwise Shor's algorithm succeeds with probability $\Omega(1/\log\log x)$.

We run t iterations. We choose t to guarantee that, except with probability $O(1/x)$, there are at least $\log_2 x$ successful iterations—which cannot all be splittings, so the largest element of the list must be prime, and then this largest element is $\leq y$ if and only if M is y-smooth. Concretely, we choose t with a $\Theta(\log\log x)$ factor accounting for the success probability of each iteration, a $\log_2 x$ factor for the number of successful iterations desired, and a further constant factor to be able to apply a Chernoff-type bound on the overall failure probability; see Appendix A. Note that $t \in (\log x)^{1+o(1)}$.

Third, we need to record enough information for each iteration to be reversible, and we need to do this while still fitting into $(\log x)^{1+o(1)}$ qubits.

Along with the list of factors of M, we keep a journal of actions taken by the iterations. Each iteration produces exactly one journal entry. An iteration that splits the ith list entry into two factors, replacing it by one factor at position i in the list and one factor added to the end of the list, records a journal entry $(2, i)$. More generally, an iteration that splits the ith list entry into $k \geq 2$ factors (e.g., splitting p^3 into p, p, p) records a journal entry (k, i). Reversing this iteration means multiplying the last $k - 1$ entries of the list into the ith entry of the list, removing those $k - 1$ entries, and removing the journal entry. An iteration that does not split the list records a journal entry $(0, 0)$.

The list always has at most $\log_2 x$ entries, so recording a journal entry takes $O(\log\log x)$ bits. The total number of journal entries is $t \in (\log x)^{1+o(1)}$, so the total journal size is $(\log x)^{1+o(1)}$.

References

1. M. M. Artjuhov. Certain criteria for primality of numbers connected with the little Fermat theorem. *Acta Arithmetica*, 12:355–364, 1966.
2. Razvan Barbulescu. *Algorithms of discrete logarithm in finite fields.* Thesis, Université de Lorraine, December 2013. `https://tel.archives-ouvertes.fr/tel-00925228`.
3. Stéphane Beauregard. Circuit for Shor's algorithm using $2n + 3$ qubits. *Quantum Information & Computation*, 3(2):175–185, 2003.
4. David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. Efficient networks for quantum factoring. *Phys. Rev. A*, 54:1034–1063, Aug 1996.
5. Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, 1989.
6. Daniel J. Bernstein. Detecting perfect powers in essentially linear time. *Math. Comput.*, 67(223):1253–1283, 1998.
7. Daniel J. Bernstein. Circuits for integer factorization: a proposal, 2001. `https://cr.yp.to/papers.html#nfscircuit`.

8. Daniel J. Bernstein. Factoring into coprimes in essentially linear time. *J. Algorithms*, 54(1):1–30, 2005.
9. Daniel J. Bernstein, Hendrik W. Lenstra Jr., and Jonathan Pila. Detecting perfect powers by factoring into coprimes. *Math. Comput.*, 76(257):385–388, 2007.
10. Joe P. Buhler, Hendrik W. Lenstra, Jr., and Carl Pomerance. Factoring integers with the number field sieve. In Arjen K. Lenstra and Hendrik W. Lenstra, Jr., editors, *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 50–94. Springer Berlin Heidelberg, 1993.
11. Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 454. The Royal Society, 1998.
12. Don Coppersmith. Modifications to the number field sieve. *J. Cryptology*, 6(3):169–180, 1993.
13. Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Society*, 22(6):644–654, 1976.
14. Dan Gordon. Discrete logarithms in GF(p) using the number field sieve. *SIAM J. Discrete Math*, 6:124–138, 1993.
15. Daniel Gottesman. Fault-tolerant quantum computation with constant overhead. *Quantum Information & Computation*, 14(15-16):1338–1372, 2014. https://arxiv.org/pdf/1310.2984.
16. Frédéric Grosshans, Thomas Lawson, François Morain, and Benjamin Smith. Factoring safe semiprimes with a single quantum query, 2015. http://arxiv.org/abs/1511.04385.
17. Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*. Springer, 2010.
18. Arjen K. Lenstra, Hendrik W. Lenstra, Jr., Mark S. Manasse, and John M. Pollard. The number field sieve. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on theory of computing*, pages 564–572, New York, NY, USA, 1990. ACM.
19. Hendrik W. Lenstra, Jr. Factoring integers with elliptic curves. *Ann. of Math. (2)*, 126(3):649–673, 1987.
20. Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. Proceedings of Mycrypt 2016, to appear, 2016. https://eprint.iacr.org/2016/1102.
21. John M. Pollard. Factoring with cubic integers. In Arjen K. Lenstra and Hendrik W. Lenstra, Jr., editors, *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 4–10. Springer Berlin Heidelberg, 1993.
22. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
23. Oliver Schirokauer. Discrete logarithms and local units. *Philosophical Transactions: Physical Sciences and Engineering*, 345:409–423, 1993.
24. Jean-Pierre Seifert. Using fewer qubits in Shor's factorization algorithm via simultaneous Diophantine approximation. In *Proceedings of the 2001 Conference on*

Topics in Cryptology: The Cryptographer's Track at RSA, CT-RSA 2001, pages 319–327, London, UK, UK, 2001. Springer-Verlag.

25. Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.
26. Yasuhiro Takahashi and Noboru Kunihiro. A quantum circuit for Shor's factoring algorithm using $2n+2$ qubits. *Quantum Info. Comput.*, 6(2):184–192, March 2006.
27. Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. *Phys. Rev. A*, 54:147–153, Jul 1996.

A Number of iterations for the serial construction

This appendix justifies the claim in Section 6.2 that choosing a sufficiently large $t \in O(\log x \log\log x)$ produces at least $\log_2 x$ successful iterations except with probability $O(1/x)$.

We abstract and generalize the situation in Section 6.2 as follows. The algorithm state evolves through t iterations: from state S_0, to state $S_1 = A_1(S_0)$, to state $S_2 = A_2(S_1)$, and so on through state $S_t = A_t(S_{t-1})$. We are given a positive real number p and a guarantee that each iteration is successful with probability *at least* p. Our objective is to put an upper bound on the chance that there are fewer than s successful iterations.

More formally: Fix a finite set X. (The algorithm state at each moment will be an element of this set.) Also fix a function $f : X \to \mathbb{Z}$. (For each algorithm state S, the value $f(S)$ is the total number of successes that occurred as part of producing this algorithm state; we augment the algorithm state if necessary to count the number of successes.) Finally, fix a positive real number p.

Let A be a random function from X to X. (This will be what the algorithm does in one iteration.) Note that "random" does not mean "uniform random"; we are not assuming that A is uniformly distributed over the space of functions from X to X.

Define A as **admissible** if the following two conditions are both satisfied:

- $f(A(S)) - f(S) \in \{0, 1\}$ for each $S \in X$.
- $q(A, S) \geq p$ for each $S \in X$, where $q(A, S)$ by definition is the probability that $f(A(S)) - f(S) = 1$.

(In other words, no matter what the starting state S is, an A iteration starting from S has probability at least p of being successful.)

Let t be a positive integer. (This is the number of iterations in the algorithm.) Let $A_1, A_2, \ldots, A_t$ be independent admissible random functions from X to X. (A_i is what the algorithm does in the ith iteration; concretely, these functions are independent if the coin flips used in the ith iteration of the algorithm are independent of the coin flips used in the jth iteration whenever $i \neq j$.)

Let S_0 be a random element of X. (This is the initial algorithm state.) Note that again we are not assuming a uniform distribution. Recursively define $S_i = A_i(S_{i-1})$ for each $i \in \{1, 2, \ldots, t\}$. ($S_i$ is the state of the algorithm after i iterations.)

Proposition 1. *Let s be a positive real number. Assume that $tp \geq s$. Then $f(S_t) - f(S_0) \leq s$ with probability at most* $\exp(-(1 - s/tp)^2 tp/2)$.

In other words, except with probability at most $\exp(-(1-s/tp)^2 tp/2)$, there are more than s successes in t iterations.

In the application in Section 6.2, we take $p \in \Omega(1/\log\log x)$ to match the analysis of Shor's algorithm; we take $s = \log_2 x$; and we compute t as an integer between, say, $(4\log_2 x)/p$ and $(4\log_2 x)/p + 1.5$. (Taking a gap noticeably larger than 1 means that we can compute t from a low-precision approximation to $(4\log_2 x)/p$.) Then $t \in O(\log x \log\log x)$. The condition $tp \geq s$ in the proposition is satisfied, so there are more than $\log_2 x$ successes in t iterations, except with probability at most $\exp(-(1-s/tp)^2 tp/2)$. The quantity tp is at least $4\log_2 x$, and the quantity $1 - s/tp$ is at least $3/4$, so $(1-s/tp)^2 tp/2 \geq (9/8)\log_2 x > 1.6\log x$, so the probability is below $1/x^{1.6}$.

Proof. Chernoff's bound says that if $v_1, v_2, \ldots, v_t$ are *independent* random elements of $\{0,1\}$, with probabilities $\mu_1, \mu_2, \ldots, \mu_t$ respectively of being 1 and with $\mu = \mu_1 + \mu_2 + \cdots + \mu_t$, then the probability that $v_1 + v_2 + \cdots + v_t \leq \delta\mu$ is at most $\exp(-(1-\delta)^2\mu/2)$ for $0 < \delta \leq 1$.

It is tempting at this point to define $v_i = f(S_i) - f(S_{i-1})$, but then there is no reason for $v_1, v_2, \ldots, v_t$ to be independent.

Instead flip independent coins $c_1, c_2, \ldots, c_t$, where $c_i = 1$ with probability $p/q(A_i, S_{i-1})$ and $c_i = 0$ otherwise. Define $v_i = c_i(f(S_i) - f(S_{i-1}))$.

By assumption $S_i = A_i(S_{i-1})$, so $f(S_i) - f(S_{i-1}) = f(A_i(S_{i-1})) - f(S_{i-1})$. This difference is 1 with probability exactly $q(A_i, S_{i-1})$, and 0 otherwise. Hence v_i is 1 with probability exactly p, and 0 otherwise. This is also true conditioned upon $v_1, v_2, \ldots, v_{i-1}$, since A_i and c_i are independent of $v_1, v_2, \ldots, v_{i-1}$; hence $v_1, v_2, \ldots, v_t$ are independent.

Now substitute $\mu_1 = \mu_2 = \cdots = \mu_t = p$, $\mu = tp$, and $\delta = s/tp$ in Chernoff's bound: we have $0 < \delta \leq 1$ (since $0 < s \leq tp$), so the probability that $v_1 + v_2 + \cdots + v_t \leq s$ is at most $\exp(-(1-s/tp)^2 tp/2)$.

Note that $v_i \leq f(S_i) - f(S_{i-1})$, so $v_1 + v_2 + \cdots + v_t \leq f(S_t) - f(S_0)$. Hence the probability that $f(S_t) - f(S_0) \leq s$ is at most $\exp(-(1-s/tp)^2 tp/2)$ as claimed. $\square$