Parallel Spooky Pebbling Makes Regev Factoring More Practical

Gregory D. Kahanamoku-Meyer* MIT
Seyoon Ragavan† MIT
Katherine Van Kirk‡ Harvard

July 2, 2026

Abstract

"Pebble games," an abstraction from classical reversible computing, have found use in the design of quantum circuits for inherently sequential tasks. Gidney showed that allowing Hadamard basis measurements during pebble games can dramatically improve costs—an extension termed "spooky pebble games" because the measurements leave temporary phase errors called ghosts. In this work, we define and study parallel spooky pebble games. Previous work by Blocki, Holman, and Lee (TCC 2022) and Gidney studied the benefits offered by either parallelism or spookiness individually; here we show that these resources can yield impressive gains when used together. First, we show by construction that a line graph of length ℓ can be pebbled in depth 2ℓ (which is exactly optimal) using space $\leq 2.47 \log \ell$. Then, to explore pebbling schemes using even less space, we use a highly optimized A^* search implemented in Julia to find the lowest-depth parallel spooky pebbling possible for a range of concrete line graph lengths ℓ given a constant number of pebbles s.

We show that these techniques can be applied to Regev's factoring algorithm (Journal of the ACM 2025) to significantly reduce the cost of its arithmetic. For example, we find that 4096-bit integers N can be factored in multiplication depth 193, which outperforms the 680 required of previous variants of Regev and the 444 reported by Ekerå and Gärtner for Shor's algorithm (IACR Communications in Cryptology 2025). While the space required for Shor's algorithm is considerably less than any variant of Regev's algorithm including ours, and thus Shor likely remains the best candidate for the first quantum factorization of large integers, our results show that implementations of Regev's algorithm are far from fully optimized, and thus Regev's algorithm may have practical importance in the future. We also believe our pebbling techniques will find applications in quantum cryptanalysis beyond integer factorization, and in quantum circuit compilation more broadly.

*Email: gkm@mit.edu. Supported by U.S. DoE Co-design Center for Quantum Advantage (C^2QA) DE-SC0012704.

†Email: sragavan@mit.edu. Supported by an Akamai Presidential Fellowship, NSF CNS-2154149, a Simons Investigator Award, the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-25-C-0300, and Amazon Research Awards. 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 Defense Advanced Research Projects Agency (DARPA). Supported in part by Jane Street. This work was done in part while the author was visiting the Simons Institute for the Theory of Computing and the Challenge Institute for Quantum Computation at UC Berkeley.

‡Email: kvankirk@g.harvard.edu. Supported by the Fannie and John Hertz Foundation and an NDSEG fellowship.

Contents

1 Introduction **1**
- 1.1 Technical Overview . . . 5
  - 1.1.1 Pebbling Games, Spooky and Parallelized . . . 5
  - 1.1.2 Applying Pebbling Games to Regev Factoring . . . 7

2 Parallel Spooky Pebbling Games **9**
- 2.1 Definitions . . . 9
- 2.2 Previous Results . . . 10
- 2.3 Parallel Spooky Pebbling with Optimal Depth and Logarithmic Space . . . 11
- 2.4 Exactly Optimal Pebbling Schemes via A* Search . . . 15

3 Preliminaries for Our Factoring Application **16**
- 3.1 Classical Algorithms for Lattice Problems . . . 16
- 3.2 Applying SBBP_α to Regev Factoring . . . 17

4 Application: Concrete Resource Estimates for Regev Factoring **18**
- 4.1 From Pebbling to Factoring . . . 19
- 4.2 Setup . . . 21
- 4.3 Results . . . 22
  - 4.3.1 Analysis . . . 23

A Deferred Proofs from Section 2.3 **29**
- A.1 Proof of Proposition 2.14 . . . 29
- A.2 Proof of Lemma 2.15 . . . 30
- A.3 Proof of Lemma 2.16 . . . 30
- A.4 Proof of Lemma 2.18 . . . 31
- A.5 Proof of Lemma 2.19 . . . 32
- A.6 Proof of Lemma 2.20 . . . 32

B Details of A* Search **35**

C Additional Background on Lattice Reduction **36**
- C.1 Heuristic Justification for Assumption 3.1 . . . 36
- C.2 Root-Hermite Factors and Costs for BKZ Lattice Reduction . . . 37
- C.3 Derivation of Equation (4) . . . 37

D Proof of Theorem 4.1 **37**

E Modifying Fibonacci-Based Regev Circuits to Use Parallelism **38**

F Parallelizing Shor's Algorithm **39**

G Detailed Results for Factoring 2048- and 4096-Bit Integers **41**
- G.1 Shor's Algorithm . . . 41
- G.2 Regev's Factoring Algorithm with Fibonacci Arithmetic . . . 41

1 Introduction

How can one efficiently carry out an inherently sequential computation on a quantum computer? It became clear early that quantum computers were unlikely to have a black-box *advantage* in running inherently sequential algorithms [Ozh99]; but surprisingly, due to the reversible nature of quantum computations, sequential computations can actually be *more* challenging for quantum computers than classical ones. More formally, suppose there is some fixed function $H : \{0,1\}^n \to \{0,1\}^n$. Given as inputs a depth ℓ and an initial string $x \in \{0,1\}^n$, we wish to evaluate $H^\ell(x) = H(H \dots (H(x)))$. Tasks of this form are ubiquitous — they are used in cryptography as proofs of work or time-lock puzzles [RSW96, AS15, AT17, ABP17, ABP18]. They also arise when performing exponentiations in a group using repeated squaring (i.e. we have $H(x) = x^2 \bmod N$), which appear for example in Regev's factoring algorithm [Reg25].

While the cost of such a task is straightforwardly ℓ evaluations of H in standard models of classical computation, it becomes significantly more interesting if we insist that all operations be reversible and H is irreversible (or at least, H is not efficiently reversible). In this case, one can still just compute H ℓ times in sequence, but this necessarily uses $O(\ell n)$ space since each iteration of H must be computed out of place. Surprisingly, one can get away with using significantly less space than this without paying prohibitive overheads in the time. This is studied through the abstraction of pebble games, which are played on a directed graph describing the dependencies of data in a computation (here simply a line graph of length ℓ, as each evaluation of H only requires the previous iteration's output as input). The goal is to place a "pebble" on the output vertex, with the rule that pebbles can only be placed on a vertex, or removed from one, if all of that vertex's incoming edges have pebbles on their start vertices (which corresponds to all input data being available for a particular computational step). The space complexity is captured by the maximum number of pebbles used at any one time, and the runtime is captured by the number of times pebbles are placed or removed. For the line graph, it is known, for example, that for any $\epsilon > 0$ we can use $O(\epsilon 2^{1/\epsilon} \cdot \log \ell)$ pebbles to complete the computation in $O(\ell^{1+\epsilon})$ time steps [Ben89, LS90].

An array of work has further improved on this state of affairs by adding two types of twists to pebble games:

- In parallel pebble games [BHL22, BHL25], pebbles may be placed and removed in parallel (as long as the parallel computations do not interfere with each other). The hope is that this allows one to carry out this computation in a depth-efficient manner. In this setting, it has been shown that one can achieve a pebbling depth of $O(\ell)$ using $O(4^{\sqrt{\log \ell}})$ pebbles for the line graph. This pebble count is on the one hand sub-polynomial i.e. $\ell^{o(1)}$, but on the other hand well above being poly-logarithmic. Moreover, it is shown that this is essentially tight in terms of space-depth volume (up to constant factors in the exponent of the number of pebbles).

- In the special case of quantum computation, Gidney [Gid19a] proposed the beautiful notion of a spooky pebble game. The heart of this idea is to exploit the fact that quantum computation can circumvent the need for reversibility with the use of mid-circuit measurements. This loophole does not come for free; the mid-circuit measurements leave a quantum phase that must be corrected later. However, crucially, the phase does not cost any qubits to store, reducing the space cost dramatically.

  Gidney showed that spookiness is a very powerful resource: with just $(1 + o(1)) \log \ell$ pebbles, we can get away with just $(1 + o(1))\ell \log \ell$ pebbling steps for the line graph.[1] Kornerup, Sadun, and Soloveichik [KSS25, Theorem 3.6] generalized this to show that with s pebbles, we only need $O(m\ell)$

[1] Here and throughout this paper, log will denote the base 2 logarithm.

Works	Parallelism?	Spookiness?	Number of pebbles	Pebbling time
[Ben89, LS90]	N	N	$O(\epsilon 2^{1/\epsilon} \cdot \log \ell)$	$O(\ell^{1+\epsilon})$
[Ben89, BHL22]			$O(\sqrt{\log \ell} \cdot 2^{\sqrt{\log \ell}})$	$O(\ell \cdot 2^{\sqrt{\log \ell}})$
[BHL22]	Y	N	$O(4^{\sqrt{\log \ell}})$	$O(\ell)$
[Gid19a, KSS25]	N	Y	$\approx \log \ell$	$\approx \ell \log \ell$
This work	Y	Y	$\approx 2.47 \log \ell$	2ℓ

Table 1: Comparison of our work and previous works on the complexity of reversibly pebbling a line graph of length ℓ. The primary differences between these works are whether they leverage parallelism and/or spookiness as resources. Our work shows that the two resources together in tandem can asymptotically match the best of both worlds, between the space that spookiness allows and the time that parallelism allows.

pebbling steps, provided that $\ell \leq \binom{m+s-2}{s-2} + 1$. Remarkably, they also show that this is tight up to constant factors [KSS25, Theorem 4.13].

The main contribution of our work is to define and thoroughly study — both asymptotically and concretely — parallel spooky pebble games, which capture the benefits that can be reaped by using these two resources (namely, parallelism and spookiness) in tandem. Specifically, we show that we can achieve a pebbling depth of 2ℓ (the best possible) using $\approx 2.47 \log \ell$ pebbles,[2] thus asymptotically getting the best of both worlds between the low pebble count of Gidney [Gid19a] and the low pebbling depth of [BHL22]. By the lower bound shown in [KSS25, Theorem 4.13], this is tight up to constant factors; the number of pebbles must be $\geq \Omega(\log \ell)$ if we want a pebbling depth of $O(\ell)$. We focus on the case of a logarithmic number of pebbles because with $O(1)$ pebbles we cannot asymptotically benefit from parallelism and logarithmically many pebbles are already sufficient for the best possible depth. We provide an overview of these results on pebbling in Table 1. As an additional contribution, we provide code that via A^* search finds the optimal-depth parallel spooky pebbling for any concrete length ℓ and number of pebbles s.[3] We find this numerical search to be useful in practical circumstances, where small optimizations not captured in the asymptotics can reduce the cost for specific values of ℓ and s. Additionally, our code supports weighted cost metrics that allow the user to incorporate knowledge that certain pebbles are more space-intensive to compute than others. This is relevant for our application to factoring (as we will discuss at the end of Section 1.1), and we are optimistic that it will be useful in other applications too.

We defer a further discussion of our results and techniques to Section 1.1, and for now turn our attention to a specific application: quantum factoring via Regev's algorithm, where we find that parallel spooky pebble games can meaningfully improve the algorithm's implementation. This gives us optimism that our results will find applications elsewhere in cryptography and quantum algorithms.

Our target application: concretely efficient Regev factoring. In the decades since Shor's $\mathsf{poly}(n)$ quantum algorithm for factoring n-bit integers was first published [Sho97], a wide array of proposals [BCDP96, VBE96, Sei01, Cop02, CW00, Bea03, TK06, Zal06, EH17, Gid18, HRS17, Gid19b, GE21, CFS25, KY24, Gid25b] have sought to ease the implementation of its algorithmic building blocks such as quantum arithmetic and the

[2] The prefactor is equal to $1/\log \alpha$, where $\alpha \approx 1.32$ is the real solution to $\alpha^3 - \alpha - 1 = 0$. We refer the reader to Theorem 2.12 for a formal statement.

[3] The code is available on GitHub and Zenodo: https://doi.org/10.5281/zenodo.17298960 [KMRV25]

quantum Fourier transform. In some cases, the gate count, qubit count, or depth of the factoring algorithm has been improved by constant or poly(log n) factors through these optimizations. However, viewed at a high level in terms of e.g. the number of n-bit multiplications that must be performed, the gate count of the factoring circuit remained static at $\widetilde{O}(n^2)$ for decades. We note that in these decades there were a few key innovations [Sei01, Zal06, EH17, Gid19b, CFS25] that give constant-factor or even log-factor improvements to the multiplication count and qubit count.

This situation persisted until 2023, when Regev [Reg25] proposed a dramatic higher-dimensional generalization of Shor's algorithm, and showed that such a generalization — when combined with a very careful implementation of the underlying quantum arithmetic — provided a polynomial reduction in the number of quantum n-bit multiplications required to be performed in the factoring circuits, from $O(n)$ to $O(\sqrt{n})$. At the time, the glaring limitation of Regev's circuit was its polynomially higher space complexity compared with that of Shor: while Shor's circuit only required $O(n)$ qubits, Regev's circuit required $O(n^{3/2})$ due to its use of repeated squaring (which is not reversible) for exponentiation modulo the number N to be factored. This issue was resolved by a modification proposed by Ragavan and Vaikuntanathan [RV24] to Regev's circuit, preserving its asymptotic gate count and depth while bringing the qubit usage down to $O(n)$.

However, a closer look reveals a still unsatisfactory picture: despite its asymptotic (in fact, polynomial) improvement over Shor's circuit, even the space-efficient variant of Regev's circuit [Reg25] by [RV24] is concretely much less efficient (when $n = 2048$, say) than state-of-the-art implementations of Shor's algorithm. At a high level, the reason for this is the reversible Fibonacci exponentiation technique proposed by Kaliski [Kal17] and leveraged by [RV24]; each step involves several multiplications of n-bit integers, and moreover requires significant space because it involves storing intermediate values as well as their inverses modulo N. Ragavan [Rag24] proposed constant-factor improvements to the circuit by [RV24] that somewhat mitigate this gap, but the concrete efficiency remains unsatisfactory and this appears to be inherent to Fibonacci-based reversible multiplication techniques. This gap in concrete efficiency was demonstrated in a meticulous study by Ekerå and Gärtner [EG25]. The central motivating question underlying this application of our work is thus the following:

Motivating Question: Does Regev's factoring algorithm offer a more efficient path towards quantumly factoring 2048-bit integers?

We revisit the original repeated squaring proposal by Regev and propose to conserve space and depth using our parallel spooky pebbling techniques. We additionally propose several concretely important optimizations to further cut down the costs of Regev's factoring algorithm, which we sketch in Section 1.1. We also implement our proposed circuits for Regev factoring in detail; in particular we report the depth of the quantum circuits (in terms of multiplication operations mod N, following the previous resource estimation work by [EG25]). We achieve the following guarantees:

- Asymptotically, it matches the depth of [Reg25, RV24, Rag24] and is only a $O(\log n)$ factor worse in the gate and qubit counts, while enjoying much better prefactors. We state this formally in Theorem 4.1.
- Concretely, for $n = 2048$, we can achieve a mod N multiplication depth per run of 157, which significantly improves upon the 580 used by Fibonacci-based implementations of Regev [RV24, Rag24]. This depth also outperforms the 230 reported by [EG25] for Shor's algorithm (although Shor uses much less space, see below for more nuanced comparisons).

  For $n = 4096$, we achieve a mod N multiplication depth of 193, which once again significantly

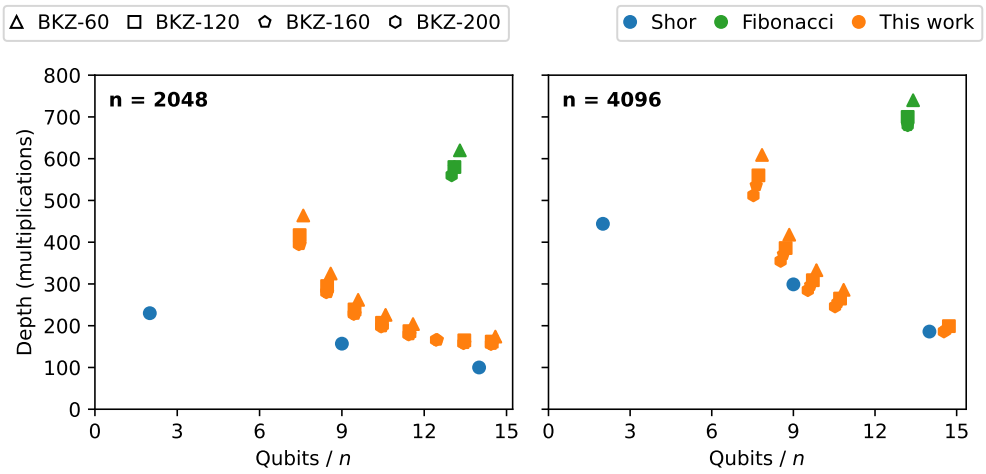

Figure 1: **Depth-space tradeoffs for a single shot of various factoring algorithms.** Note that this configuration of Shor's algorithm requires roughly 10 times fewer shots than the other two constructions, which are based on Regev's algorithm. We include four points for each Regev variant corresponding to the strength of the BKZ lattice reduction used in classical postprocessing (see Section 4 for details). Across the tradeoff curve, our results outperform [RV24] (labeled "Fibonacci") in both space and depth, yielding by far the most efficient implementation of Regev's algorithm currently known. In some regions of parameter space the per-shot cost of our results is comparable to that of Shor's algorithm.

improves on the 680 used by Fibonacci-based implementations of Regev, and outperforms the 444 reported by [EG25] for Shor.

In order to make a fairer comparison between algorithms with variable tradeoffs between time and space, we plot the per-shot time and depth cost of our results against both [RV24] and a parallelized version of Shor (see Appendix F) in Figure 1. For more detailed results see Section 4.3.

Outlook and caveats. The practical takeaway of this work is that the cost of Regev's algorithm can be reduced considerably from the previous state of the art [Reg25, RV24, EG24, EG25], even achieving a per-shot cost that is comparable to Shor's algorithm in some regimes [Sho97, Gid19b, GE21, CFS25, Gid25b][4]. Indeed, on practical input sizes the margin between the two algorithms is narrower than previously thought. Furthermore, we see no inherent obstacle to further optimization that builds on top of these ideas, thus reducing the cost of Regev's algorithm even further. We thus believe that continued study on the concrete efficiency of Regev's algorithm is warranted for at least the following two reasons: (a) Shor's circuit has benefited from decades of optimizations, while Regev's circuit has only existed for two years; and (b) even if we ultimately determine that Regev does not outperform Shor on practically-sized integers, such a determination cannot be made until we are confident that we have the best practical implementation of Regev. Our work makes substantial progress towards this goal, and we are certain that further improvements are possible.

[4]Ekerå and Gärtner [EG25] note that they may be underestimating the benefit offered by windowing for Shor's algorithm in their cost estimates. Regardless of this, we believe the qualitative takeaways of our results — in relation to the concrete potential of variants of Regev's algorithm — still stand.

The circuit depth per shot seems to be particularly promising, which is a metric that could become important in the future: since Regev's algorithm requires running many small circuits and classically combining the results, many intermediate-scale quantum computers could run the algorithm in parallel without any need for quantum communication between them. This would be particularly impactful in streaming settings where it is desired to factor integers quickly as soon as they are received.

We note that in the near future, circuit depth per shot is likely not the most important metric; qubit count and the total number of gates across all runs of the circuit are both important and by these metrics our variants of Regev are still much less efficient than Shor, as we discuss in Section 4.3.1.

Other applications. Understanding the complexity of carrying out inherently sequential computations is also integral to cryptography, where one may want to make use of proof-of-work or time-lock puzzles. Indeed, the motivation of the study by [BHL22, BHL25] of parallel pebbling games was to understand the post-quantum security of such tasks (while also accounting for space usage). These works focus attention on parallelism and do not consider spookiness, due to concerns at the time with the practicality of mid-circuit measurements. However, it has become clear that high-fidelity mid-circuit measurements will be a necessary and integral part of any future quantum computer due to their centrality in quantum error correction; indeed, high-fidelity mid-circuit measurement has already been demonstrated in a number of experiments [DLH+22, KSMM23, ZKL+23, HHB+25].

Organization of the paper. The remainder of this paper is organized as follows:

- In Section 1.1, we provide an overview of our techniques.
- In Section 2, we define and study abstract parallel spooky pebbling games, and prove that in this setting the line graph of length ℓ can be pebbled in optimal depth 2ℓ with at most $2.47 \log \ell$ pebbles. We also present empirical results detailing optimal pebbling depth for some concrete instances of parallel spooky pebbling games.
- In Section 3, we set up some preliminaries that will be useful for our application to Regev's factoring algorithm [Reg25].
- Finally, in Section 4, we put these ingredients together to obtain our results for reimplementing the arithmetic in Regev's factoring circuit.

1.1 Technical Overview

In Section 1.1.1, we provide an overview of the main ideas underlying our parallel spooky pebbling scheme. We then turn to our factoring application in Section 1.1.2, where we discuss how Regev's circuit can be viewed as a pebbling game and additional optimizations specific to Regev factoring that we make in this work.

1.1.1 Pebbling Games, Spooky and Parallelized

Recall that our goal is essentially to compute $H^\ell(x)$ for a function $H : \{0,1\}^n \to \{0,1\}^n$ and string $x \in \{0,1\}^n$. This can be abstracted out as a pebble game on a line graph of length ℓ. As already mentioned, the starting point of Gidney's spooky pebbling proposal [Gid19a] is the following naive idea: after computing $|x\rangle \mapsto |x\rangle\,|H(x)\rangle$, we will be using one extra register. One way to "clean up" the $|x\rangle$ register would simply

be to measure it in the Hadamard basis! We will obtain some classical string $s \in \{0,1\}^n$ and be left with the state $(-1)^{\langle s,x\rangle} |H(x)\rangle$, which now does not use any additional space. Crucially, the probability that we obtain a particular string s is independent of x (because all entries of the Hadamard matrix have the same magnitude), so the absolute amplitudes in our superposition will remain the same; we need only deal with these phases that arise.

Of course, this does not yet solve our problem; this phase will vary across the inputs in our superposition, and will completely destroy the correctness of the underlying algorithm if we need to compute $H^\ell(\cdot)$ in superposition. We thus need to "clean up" this leftover phase at some point, so we will need to recompute x at some point. This gives us access to three operations:

- Pebbling: given a register $|x\rangle$, write $|H(x)\rangle$ in a new register: $|x\rangle |0^n\rangle \mapsto |x\rangle |H(x)\rangle$.
- Ghosting: measure a register $|x\rangle$ in the Hadamard basis. In doing so, we free up n qubits but we create a phase that we will need to clean up later.
- Unpebbling: this is the inverse of pebbling i.e. it computes $|x\rangle |H(x)\rangle \mapsto |x\rangle |0^n\rangle$.

  We implicitly assume that when unpebbling we will also "exorcise" any phases that were accumulated earlier from this stage of the pebbling game i.e. if we have a phase $(-1)^{\langle s,x\rangle}$ and also have $|x\rangle$ available in a register, we can remove this phase e.g. using phase kickback [CEMM98].

This abstract formulation extends the notion of reversible pebbling games [Ben89] from reversible computation to quantum measurement-based uncomputation and was first proposed and christened as "spooky pebbling" by Gidney [Gid19a]. Surprisingly, these operations can be cleverly sequenced to obtain a construction that is more efficient than directly implementing the square-and-multiply algorithm. When computing modular exponents with exponents of bit length ℓ, naive square-and-multiply uses $2\ell - 1$ multiplications ("pebbling steps") and ℓ n-qubit registers ("pebbles"). In comparison, Gidney's proposed pebbling scheme uses $\approx \ell \log \ell$ pebbling steps (which is slightly worse), but manages to do so using only $\approx \log \ell$ pebbles. This already provides a concretely space-efficient and gate-efficient way to implement a sequential function such as H^ℓ.

The main idea underlying Gidney's proposal — which we will also make use of — is the following divide-and-conquer approach. Assume for now that ℓ is a power of 2. The first observation is that with just two pebbles we can march one pebble all the way to position ℓ, while leaving ghosts throughout. We refer to this procedure as "blasting". However, once we have done this, uncomputing the ghosts will be costly. To remedy this, when blasting we will leave "marker pebbles" at a carefully chosen sequence of positions. These markers will provide less costly starting points from which we can uncompute ghosts. With this in mind, we can describe Gidney's pebbling procedure at a high level:

1. Blast all the way to position ℓ, leaving marker pebbles at positions $\ell/2$, $3\ell/4$, $7\ell/8$, ... and ghosts everywhere else.
2. Uncompute everything in the right half (positions $\ell/2$ to ℓ)
3. Recursively blast on the interval $[0, \ell/2]$.
4. Uncompute everything in the left half (positions 0 to $\ell/2$).

We build on Gidney's proposal by defining and analyzing parallel spooky pebbling games, where some of the pebbling steps are run in parallel to further conserve depth. As mentioned earlier, parallelism has

previously been studied as a resource in pebble games [BHL22, BHL25], but without spookiness. We show that spookiness and parallelism together give rise to pebbling games that are more efficient that what was possible using either resource individually.

One natural idea to inject parallelism into Gidney's proposal outlined above would be to run steps 2 and 3 in parallel. This would give a pebbling scheme with optimal depth 2ℓ and $\approx 3\log \ell$ pebbles. We improve on this with a very careful recursive construction based on a Fibonacci-like sequence instead of powers of 2, showing that we can achieve the same depth with just $\approx 2.47 \log \ell$ pebbles (compared with the $O(4^{\sqrt{\log \ell}})$ pebbles used by [BHL22] using only parallelism). The pebbling schemes discussed in this overview are summarized in Table 1.

1.1.2 Applying Pebbling Games to Regev Factoring

Why pebbling? Let us recap the existing approaches to implementing the modular exponentiation in Regev's circuit:

- Using a reversible Fibonacci exponentiation algorithm, as in Ragavan and Vaikuntanathan's optimizations [Kal17, RV24, Rag24]: this incurs significant constant-factor overheads due to the need to work with intermediate registers as well as their inverses modulo N. This barrier seems inherent (see [RV24, "On Kaliski's Work and Quantum-Quantum Multiplication"] for a discussion of this).
- Using the classic square-and-multiply algorithm, as in Regev's original circuit [Reg25]: this is now very efficient in terms of the number of multiplications, but encounters the barrier that squaring modulo N is not a reversible operation and thus must be done out of place. Regev's algorithm performs $O(\sqrt{n})$ squarings mod N and each squaring uses a new n-qubit register, which is the source of the significant $O(n^{3/2})$ qubit requirement.[5] This reversibility barrier is not as inherent; indeed, pebbling games provide a way to circumvent this.

Thus, any approach that could hope to rival the efficiency of Shor's algorithm on 2048-bit integers will likely have to modify the square-and-multiply algorithm to be more space-efficient.

Formulating Regev's circuit as a pebbling game. The bottleneck of Regev's quantum circuit is computing the mapping

$$(z_1, \ldots, z_d) \mapsto \prod_{j=1}^{d} a_j^{z_j} \bmod N,$$

where the a_j's are small bases and the z_j's are non-negative integers $< D$. Regev's square-and-multiply procedure [Reg25] is summarized (in classical terms for now) in Algorithm 1.1. Correctness follows from the fact that at the end of step i^*, we have

$$x = \prod_{j=1}^{d} a_j^{\sum_{i=1}^{i^*} z_{j,i} 2^{\log D - i}} \bmod N,$$

where $z_{j,i}$ is the ith bit of z_j (in big-endian bit order, indexed from 1).

In other words, each iteration of either Step 2a or Step 2b of Algorithm 1.1 constitutes a pebbling step in our pebbling game, which will have length $2 \log D$. We unpack this in more detail in Section 4.1.

[5] We note that Shor's algorithm [Sho97] circumvents this space barrier by essentially offloading the squarings mod N to classical preprocessing and quantumly only doing multiplications mod N. This trick does not appear applicable to Regev; see [RV24, "Space Complexity"] for a discussion of this.

Algorithm 1.1: Regev's square-and-multiply procedure. Note that for simplicity, we write the algorithm here in non-reversible terms.

Data: Indices $z_1, \dots, z_d$
Result: The product $\prod_{j=1}^{d} a_j^{z_j} \bmod N$

1. Initialize $x = 1$.
2. For $i = 1, 2, \dots, \log D$ in that order:
   (a) Update $x \leftarrow x^2 \bmod N$.
   (b) Update $x \leftarrow x \cdot \prod_{j=1}^{d} a_j^{z_{j,i}} \bmod N$.

Amortizing Step 2b across pebbles. As seen in Algorithm 1.1, each iteration of the loop in Step 2 involves two pebbling steps: the squaring mod N in Step 2a and the additional multiplicative update in Step 2b. Neither of these costs is negligible. However, it was observed by Ragavan [Rag24] that one can actually save gates by only carrying out Step 2b for some values j (this is using a technique that bears some high-level similarity to the use of windowing in optimizing Shor's algorithm [Gid19b, GE21]; see [Rag24] for a further comparison of these techniques). Concretely, we will specify some window length w and Step 2 will have the following structure:

- If i is not a multiple of w: update $x \leftarrow x^2 \bmod N$;
- If i is a multiple of w: update $x \leftarrow x^2 \cdot \prod_{j=1}^{d} a_j^{z_{j,[i:i+w)}} \bmod N$.

Here, $z_{j,[i:i+w)}$ is the integer formed from bits i to $i+w$ of z_j, in big-endian bit order. This can be thought of as a batched version of Step 2b.

Ghosting inside step 2b. We optimize further by using measurement based uncomputation to minimize the cost of reversibility inside of the pebble operations. For $\mathsf{Pebble}(i)$ steps from Step 2b when i is a multiple of w, we implement the overall transformation $x \leftarrow x \cdot \prod_{j=1}^{d} a_j^{z_{j,[i:i+w)}}$ by first computing $t_i = \prod_{j=1}^{d} a_j^{z_{j,[i:i+w)}}$ in an ancillary register, and then computing the product of $x \cdot t_i$ into the output. Then, instead of explicitly uncomputing t_i, we use measurement-based uncomputation to "ghost" it. Only during the $\mathsf{Unpebble}(i)$ phase must we explicitly uncompute t_i (because unpebbling must not leave ghosts); during this stage we also fix all the phases associated with previous ghosting of t_i during calls to $\mathsf{Pebble}(i)$.

Computing $\prod_{j=1}^{d} a_j^{z_j,[i:i+w)}$ in practice. This pebbling abstraction implicitly disregards the cost of actually computing $\prod_{j=1}^{d} a_j^{z_j,[i:i+w)}$. Asymptotically, if we set $d = \sqrt{n}$, this is reasonable since this product is typically much smaller than N [Reg25]. However, in practice, we would like to set d to be as large as possible to maximize the benefit we can get from high dimensionality: Ekerå and Gärtner [EG25] and our work both select d to be as large as possible while ensuring that $\prod_{j=1}^{d} a_j^{2^w-1} < 2^n$. In this setting, it follows by construction that the bit length of $\prod_{j=1}^{d} a_j^{z_j,[i:i+w)}$ is no longer negligible.

As discussed in their their study, Ekerå and Gärtner [EG25] chose to disregard the space and gate costs of constructing these trees, which was natural in their context because their finding was that Regev does not perform as well as Shor on cryptographically relevant problem sizes even when biasing the comparison

in Regev's favor by omitting costs such as these. Given our optimism about the algorithms introduced in our work, we choose to also carefully consider these costs.

In this respect, a key point of concern is the space overhead that arises from storing an entire product tree — particularly in our case where we may be constructing multiple product trees in parallel. We state some natural but ultimately limited ways to optimize this issue away, then outline our solution:

- We could save a constant factor in space by uncomputing any right children of nodes in the product tree.
- Even better, when pebbling a particular node, we can ghost each layer of the tree once its parent layer has been computed. Thus we only ever need to store two layers of the tree. However, when unpebbling, we will still need to store the entire tree (or a constant fraction) for the sake of uncomputation.

In our work, we simplify matters by abandoning the product tree, instead classically precomputing $a_j^{z_j 2^i}$ for $i = 0, 1, \ldots, w-1$, and simply multiplying the $d \cdot w$ relevant numbers in sequence. This is comparably efficient when using schoolbook arithmetic, and additionally can essentially be done in place if one uses Gidney's recent low-space classical-quantum adder [Gid25a] for the multiplications.

Not all pebbles are created equal! Finally, we note that our pebble operations corresponding to steps 2a and 2b are qualitatively different in terms of efficiency. The latter step needs to do some additional arithmetic (albeit with small integers), and more significantly needs additional space to store the intermediate value $t = \prod_{j=1}^{d} a_j^{z_{j,[i:i+w)}}$. Although our observation in the above paragraph allows us to shave down this space overhead considerably, it will still not be negligible.

To reflect this, we optimize our choice of pebbling while incorporating a weighted cost metric that captures the fact that pebbles in certain positions require more ancillary space to compute (or uncompute) than others. We include this in our Julia script for finding efficient parallel spooky pebbling games, and are optimistic that the ability to weight different pebbling indices might be useful for other applications as well.

2 Parallel Spooky Pebbling Games

2.1 Definitions

Definition 2.1 (Pebbling Games and States)**.** A pebbling game is specified by a target length ℓ. At any point, the state of the game will be a vector $\mathsf{state} \in \{\mathsf{pebble}, \mathsf{ghost}, \mathsf{empty}\}^{\ell+1}$. We index the elements of state by indices in $[0, \ell]$, and use state_i to denote entry i of state.

The initial and final states of the pebbling game will both be (pebble, empty, empty, ..., empty). (The first entry of the state is a dummy variable which will always be pebble.) We require that one of the intermediate states of the pebbling game has a pebble in position ℓ i.e. $\mathsf{state}_\ell = \mathsf{pebble}$, and the target end state will be $(\mathsf{pebble}, \mathsf{empty}, \ldots, \mathsf{empty})$.

Definition 2.2 (Pebbling Moves)**.** A pebbling move is defined together with an active set, which is a subset of $[0, \ell]$. It can be applied to an input state state (subject to certain restrictions on state) and produces a modified state state'. It can be any one of the following:

- $\mathsf{Pebble}(i)$ for some $i \in [1, \ell]$: this is applicable when $\mathsf{state}_{i-1} = \mathsf{pebble}$ and $\mathsf{state}_i \in \{\mathsf{ghost}, \mathsf{empty}\}$. It will produce state' which is identical to state in all positions except state'_i which will now be pebble. Its active set is $\{i-1, i\}$.

- $\mathsf{Unpebble}(i)$ for some $i \in [1, \ell]$: this is applicable when $\mathsf{state}_{i-1} = \mathsf{state}_i = \mathsf{pebble}$. It will produce state' which is identical to state in all positions except state'_i which will now be empty. Its active set is $\{i-1, i\}$.
- $\mathsf{Ghost}(i)$ for some $i \in [1, \ell]$: this is applicable when $\mathsf{state}_i = \mathsf{pebble}$. It will produce state' which is identical to state in all positions except state'_i which will now be ghost. Its active set is $\{i\}$.

Definition 2.3 (Sequential and Parallel Pebbling Games)**.** In a pebbling game, each time step consists of two phases that are applied in sequence:

- In the first phase of a sequential pebbling game, we can make at most one call to $\mathsf{Pebble}, \mathsf{Unpebble}$. In the first phase of a parallel pebbling game, we can make an arbitrary number of parallel calls to $\mathsf{Pebble}, \mathsf{Unpebble}$, **provided that their active sets are pairwise disjoint.**
- In the second phase, we can make an arbitrary number of parallel calls to Ghost.

Definition 2.4 (Pebbling Cost and Depth)**.** The pebbling cost of a pebbling game is the total number of calls to $\mathsf{Pebble}, \mathsf{Unpebble}$.

We define pebbling depth as the total number of time steps (as defined in Definition 2.3).

Remark 2.5. Note that in the case of a sequential pebbling game, the pebbling cost and pebbling depth will always be equal.

Remark 2.6. We exclude calls to Ghost when calculating pebbling cost, because in typical applications Ghost will be significantly cheaper than the other three operations.

Remark 2.7 (Optimal-Depth Pebbling). It is straightforward to see that a parallel pebbling game of length ℓ must necessarily have depth $\geq 2\ell$. This is because the position of the rightmost pebble/ghost changes by at most ± 1 in each time step. Initially and at the end, this position is 0. At some point this position must be ℓ.

Definition 2.8 (Pebbling Space)**.** Let the states of a pebbling game in sequence be $\mathsf{state}[0], \mathsf{state}[1], \ldots, \mathsf{state}[T]$, where $\mathsf{state}[0], \mathsf{state}[T]$ are the initial and target states defined in Definition 2.1. In other words, for each $i \in [0, T-1]$, $\mathsf{state}[i+1]$ is obtained from $\mathsf{state}[i]$ by applying one time step (as defined in Definition 2.3).

For each $t = 1, \ldots, T$, we define the space usage s_t of time step t as the number of indices $i \in [1, \ell]$ such that at least one of $\mathsf{state}[t-1]_i, \mathsf{state}[t]_i$ is pebble. (Note in particular that this set of indices contains the active sets of all operations performed when going from $\mathsf{state}[t-1]$ to $\mathsf{state}[t]$; this can be verified by inspecting Definition 2.2.) Then we define the pebbling space of the pebbling game to be $\max_{t \in [1,T]} s_t$.

2.2 Previous Results

Sequential spooky pebbling. The study of sequential spooky pebbling was initiated by Gidney [Gid19a] and further explored by Kornerup et al [KSS25]. We refer the reader to [KSS25, Figure 1] for a summary of previous results on this front and state some key results from there for comparison:

Theorem 2.9 (Sequential Spooky Pebbling)**.** *There exist sequential spooky pebbling games achieving any of the following guarantees:*

- *Pebbling space 3 and pebbling cost $O(\ell^2)$ [Gid19a];*
- *Pebbling space $(1 + o(1)) \log \ell$ and pebbling cost $(1 + o(1))\ell \log \ell$ [Gid19a];*
- *Pebbling space s and pebbling cost $O(m\ell)$ provided $\binom{m+s-2}{s-2} \geq \ell$ [KSS25, Lemma 3.5].*

Moreover, the third item here is asymptotically tight: for any s, m such that $n > \binom{m+s-2}{s-2}$, a sequential spooky pebbling of length ℓ must use at least $\Omega(m\ell)$ steps [KSS25, Theorem 4.13].

Parallel (non-spooky) pebbling. The notion of parallel reversible pebbling — which resembles our notion of pebbling but does not allow for ghosting — was previously studied by [BHL22, BHL25] with the motivation of studying the post-quantum security of proof-of-work puzzles. They show the following result:

Theorem 2.10. *There exists a parallel reversible pebbling scheme with pebbling space* $4\sqrt{\log \ell}$ *and pebbling depth* $O(\ell)$ *[BHL22, Lemma 8]. Moreover, if any parallel reversible pebbling scheme achieves pebbling space* s *and pebbling depth* d*, we must have* $sd \geq \ell \cdot \Omega(2^{(\sqrt{2}-o(1))\sqrt{\log \ell}})$ *[BHL25, Theorem 1].*

Lower bounds on parallel spooky pebbling. The aforementioned lower bound by [KSS25, Theorem 4.13] can be readily turned into a lower bound on parallel spooky pebbling games:

Corollary 2.11 (Lower Bound on Parallel Spooky Pebbling)**.** For any s, m such that $\ell > \binom{m+s-2}{s-2}$, a parallel spooky pebbling game of length ℓ must have depth at least $\Omega(m\ell/s)$.

Consequently, for any real constant $c \geq 2$ there exists another constant $c' > 0$ such that at least $c' \log \ell$ pebbles are needed to achieve depth $< c\ell$.

Proof. This is immediate from the observation that with s pebbles, we can only possibly carry out s operations in parallel at a time. Hence any parallel spooky pebbling game can be converted into a sequential spooky pebbling game where the total number of operations is at most $s\times$ the depth of the parallel game.

To see the second statement, let $\mu > 0$ be a universal constant so that the depth must be at least exactly $\frac{m\ell}{s\mu}$. Let $c' > 0$ be any constant such that $\binom{c\mu s+s-2}{s-2} \leq 2^{s/c'}$ for all s sufficiently large. Then suppose for the sake of contradiction that there exists a pebbling game with $s < c' \log \ell$ pebbles and depth $< c\ell$. Then let $m = c\mu s$. Since $s < c' \log \ell$, we have $\ell > 2^{s/c'} \geq \binom{m+s-2}{s-2}$. Consequently, the depth must be at least $\frac{m\ell}{s\mu} = c\ell$, which is a contradiction. □

2.3 Parallel Spooky Pebbling with Optimal Depth and Logarithmic Space

The main result of this section is the following positive result for parallel spooky pebbling. By Corollary 2.11, this is tight up to constant factors. We focus on the case of a logarithmic number of pebbles because with $O(1)$ pebbles we cannot asymptotically benefit from parallelism and as the below theorem establishes, logarithmically many pebbles are already sufficient for the best possible depth.

Theorem 2.12. *Let* $\alpha \approx 1.32$ *be the real solution to* $\alpha^3 - \alpha - 1 = 0$*. Then for any* $\ell \geq 7$*, there exists a parallel spooky pebbling scheme with pebbling space* $\left\lceil \frac{\log \ell - \log 0.94}{\log \alpha} - 3 \right\rceil \leq \lceil 2.47 \log \ell - 2.77 \rceil$ *and pebbling depth exactly* 2ℓ *(which is optimal).*

Before we delve into the details of our pebbling scheme, let us reiterate the high-level ideas that we sketched in Section 1.1. Assume for now that ℓ is a power of 2. The first observation is that with just two pebbles we can march one pebble all the way to position ℓ, while leaving ghosts throughout. We refer to this procedure as "blasting". However, once we have done this, uncomputing the ghosts will be costly. To remedy this, when blasting we will leave "marker pebbles" at a carefully chosen sequence of positions. These markers will provide less costly starting points from which we can uncompute ghosts. With this in mind, we can describe a variant of our pebbling procedure at a high level:

1. Blast all the way to position ℓ, leaving marker pebbles at positions $\ell/2$, $3\ell/4$, $7\ell/8$, ... and ghosts everywhere else.

2. Uncompute everything in the right half (positions $\ell/2$ to ℓ), while recursively blasting on the interval $[0, \ell/2]$ in parallel.

3. Uncompute everything in the left half (positions 0 to $\ell/2$).

It turns out that this achieves depth 2ℓ and pebbling space $\approx 3 \log \ell$. With a more careful and involved variant of the above pebbling game, we can reduce the constant factor in the pebbling space. Roughly, we will place the first marker pebble at a position around $c\ell$ for some $c < 1/2$. Informally, this enables us to get some of the uncomputation in the right half out of the way before we need to allocate additional pebbles to blast in the left half.

Sequence $\{A_k\}$ and its properties. While the above simplified outline of our pebbling scheme worked with ℓ being a power of 2, it will be beneficial for us to work with a different exponentially growing sequence. We define this sequence here and state its properties.

Definition 2.13 (A-sequence)**.** Define the sequence $A_1, A_2, \ldots$ by the base cases $A_1 = 1, A_2 = 2, A_3 = 2$, and the recurrence relation $A_k = A_{k-2} + A_{k-3}$.

For any positive integer n, we let $A^{-1}(n)$ denote the largest index j such that $A_j \le n$. (Note in particular that $A^{-1}(2) = 3$.)

Proposition 2.14 (Properties of $\{A_k\}$)**.** We have:

1. For all $k \ge 2$, $A_{k-1} \le A_k$, with equality if and only if $k = 3$.

2. For any $k \ge 4$, we have:
$$A_k = 2 + \sum_{i=1}^{\lfloor (k-2)/2 \rfloor} A_{k-2i-1}.$$

3. For any $k \ge 6$, we have $A_k = A_{k-1} + A_{k-5}$.

4. For any $k \ge 4$, we have $A_k - A_{k-1} \ge A_{k-1} - A_{k-2}$.

5. For all $k \ge 7$, we have $A_k \ge 0.94\alpha^k$, where
$$\alpha = \frac{(9 - \sqrt{69})^{1/3} + (9 + \sqrt{69})^{1/3}}{18^{1/3}} \approx 1.32$$
is the real root of $\alpha^3 - \alpha - 1 = 0$.

Proof. See Appendix A.1. □

Our parallel spooky pebbling scheme. Our pebbling procedure is presented in Algorithm 2.1. Its two main recursive ingredients are a "blasting" procedure Algorithm 2.2 and an "unblasting" (uncomputation) procedure Algorithm 2.3.

Algorithm 2.1: Optimal-depth parallel pebbling scheme

Data: Pebbling game of length $\ell = A_k$, initialized to state $(\mathsf{pebble}, \mathsf{empty}, \mathsf{empty}, \dots, \mathsf{empty}, \mathsf{empty})$.

Result: A completed pebbling game in the target end state $(\mathsf{pebble}, \mathsf{empty}, \mathsf{empty}, \dots, \mathsf{empty}, \mathsf{empty})$. At some intermediate stage, we will have $\mathsf{state}_\ell = \mathsf{pebble}$.

1. Call $\mathsf{Blast}(0, \ell - 1)$.
2. Apply $\mathsf{Pebble}(\ell)$ followed by $\mathsf{Unpebble}(\ell)$.
3. Call $\mathsf{Unblast}(0, \ell - 1)$.

Algorithm 2.2: $\mathsf{Blast}(i_{\mathsf{start}}, i_{\mathsf{end}})$: a blasting subroutine with "marker pebbles" at A_k

Data: Indices $i_{\mathsf{start}}, i_{\mathsf{end}}$ such that $0 \le i_{\mathsf{start}} < i_{\mathsf{end}} \le \ell - 1$, and $i_{\mathsf{end}} - i_{\mathsf{start}} = A_k - 1$, for some index $k \ge 2$. Initially, we have $\mathsf{state}_{i_{\mathsf{start}}} = \mathsf{pebble}$ and $\mathsf{state}_i \in \{\mathsf{empty}, \mathsf{ghost}\}$ for $i \in [i_{\mathsf{start}} + 1, i_{\mathsf{end}}]$.

Result: At the end of the algorithm, we will have "marker" pebbles at A_k-sized intervals away from the start. Formally, we will have $\mathsf{state}_i = \mathsf{pebble}$ for $i = i_{\mathsf{start}}, i_{\mathsf{start}} + A_{k-3}, i_{\mathsf{start}} + A_{k-3} + A_{k-5}, \dots, i_{\mathsf{end}} - 1, i_{\mathsf{end}}$. All other sites are ghost.

1. Base case: if $i_{\mathsf{end}} - i_{\mathsf{start}} = 1$ (i.e. $k \in \{2, 3\}$), $\mathsf{Pebble}(i_{\mathsf{end}})$ and terminate.
2. If $i_{\mathsf{end}} - i_{\mathsf{start}} > 1$ and $k > 3$, we recursively proceed as follows:
   (a) $\mathsf{Pebble}(i_{\mathsf{start}} + 1)$
   (b) For sites $j \in [i_{\mathsf{start}} + 2, i_{\mathsf{start}} + A_{k-3}]$: $\mathsf{Pebble}(j)$ and then $\mathsf{Ghost}(j - 1)$
   (c) Blast on the remainder of the subgame by recursively calling $\mathsf{Blast}(i_{\mathsf{start}} + A_{k-3}, i_{\mathsf{end}})$.

Analysis of Blast and Unblast. It is fairly straightforward to see from our algorithms that they are valid pebbling games and achieve the claimed input/output behavior. We now turn our attention to establishing the desired efficiency guarantees, initially focusing on Algorithms 2.2 and 2.3. We will address their depth in Lemmas 2.15 and 2.16, and devote the remainder of this subsection (Definition 2.17 to Lemma 2.20) to analyzing their space.

Lemma 2.15. *For all k and $i_{\mathsf{start}}, i_{\mathsf{end}}$ such that $i_{\mathsf{end}} - i_{\mathsf{start}} = A_k - 1$, the algorithm $\mathsf{Blast}(i_{\mathsf{start}}, i_{\mathsf{end}})$ takes exactly $A_k - 1$ time steps.*

Proof. By strong induction on k, see Appendix A.2. □

Lemma 2.16. *For all k and $i_{\mathsf{start}}, i_{\mathsf{end}}$ such that $i_{\mathsf{end}} - i_{\mathsf{start}} = A_k - 1$, the algorithm $\mathsf{Unblast}(i_{\mathsf{start}}, i_{\mathsf{end}})$ takes exactly $A_k - 1$ pebbling time steps.*

Proof. By strong induction on k, see Appendix A.3. □

In the below lemmas, we will account for the space at a specific time step t, as formally defined in Definition 2.8. To do this, we formally define the following quantities:

Definition 2.17 ($b_k(t)$ and $u_k(t)$)**.** Fix a positive integer $k \ge 2$ and consider a pebbling sub-game on a segment $[i_{\mathsf{start}}, i_{\mathsf{end}}]$ such that $i_{\mathsf{end}} - i_{\mathsf{start}} = A_k - 1$.

Algorithm 2.3: $\mathsf{Unblast}(i_{\mathsf{start}}, i_{\mathsf{end}})$: an unblasting subroutine

Data: Indices $i_{\mathsf{start}}, i_{\mathsf{end}}$ such that $0 \le i_{\mathsf{start}} < i_{\mathsf{end}} \le \ell - 1$ and $i_{\mathsf{end}} - i_{\mathsf{start}} = A_k - 1$ for some index $k \ge 2$. Initially we have $\mathsf{state}_{i_{\mathsf{start}}}, \mathsf{state}_{i_{\mathsf{start}}+1}, \dots, \mathsf{state}_{i_{\mathsf{end}}}$ as they would be at the end of $\mathsf{Blast}(i_{\mathsf{start}}, i_{\mathsf{end}})$ (see the specification of Algorithm 2.2 for details).

Result: At the end of the algorithm, we will have $\mathsf{state}_{i_{\mathsf{start}}} = \mathsf{pebble}$ and $\mathsf{state}_i = \mathsf{empty}$ for all i such that $i_{\mathsf{start}} < i \le i_{\mathsf{end}}$.

1. Base case: if $i_{\mathsf{end}} - i_{\mathsf{start}} = 1$ (i.e. $k \in \{2, 3\}$), run $\mathsf{Unpebble}(i_{\mathsf{end}})$ and terminate.
2. If $i_{\mathsf{end}} - i_{\mathsf{start}} > 1$ and $k > 3$, we recursively proceed as follows. Except where stated, all steps are sequential.
   (a) Run the first $A_{k-2} - A_{k-3}$ steps of $\mathsf{Unblast}(i_{\mathsf{start}} + A_{k-3}, i_{\mathsf{end}})$.
   (b) In parallel, run the remaining $A_{k-3} - 1$ steps of $\mathsf{Unblast}(i_{\mathsf{start}} + A_{k-3}, i_{\mathsf{end}})$ as well as a $\mathsf{Blast}(i_{\mathsf{start}}, i_{\mathsf{start}} + A_{k-3} - 1)$.
   (c) Call $\mathsf{Unpebble}(i_{\mathsf{start}} + A_{k-3})$.
   (d) Call $\mathsf{Unblast}(i_{\mathsf{start}}, i_{\mathsf{start}} + A_{k-3} - 1)$.

- During the blasting phase (Algorithm 2.2), we label by time step $t \in [1, A_k - 1]$ the step in which $i_{\mathsf{start}} + t$ is pebbled. We let $b_k(t)$ denote the space usage of time step t, excluding the given pebble at i_{start}.
- During the unblasting phase (Algorithm 2.3), we label by time step $t \in [1, A_k - 1]$ the step in which $i_{\mathsf{end}} - t + 1$ is unpebbled. We let $u_k(t)$ denote the space usage of time step t, excluding the given pebble at i_{start}.

Lemma 2.18. *For any $k \ge 2$ and $t \in [1, A_k - 1]$, we have:*

$$b_k(t) = \left\lceil \frac{k - A^{-1}(A_k - t + 1)}{2} \right\rceil + 1 \text{ pebbles.} \tag{1}$$

Proof. By induction on k, see Appendix A.4. □

Lemma 2.19. *For any integer $k \ge 5$, we have:*

$$u_k(t) = \begin{cases} 1 + u_{k-2}(t), & \text{if } 1 \le t \le A_{k-2} - A_{k-3}, \\ b_{k-3}(t - (A_{k-2} - A_{k-3})) + 1 + u_{k-2}(t), & \text{if } A_{k-2} - A_{k-3} + 1 \le t \\ & \quad \le A_{k-2} - 1, \\ \lfloor (k-1)/2 \rfloor, & \text{if } t = A_{k-2}, \\ u_{k-3}(t - A_{k-2}), & \text{if } A_{k-2} + 1 \le t \le A_k - 1. \end{cases}$$

Proof. This follows from the recursive formulation of Algorithm 2.3, see Appendix A.5. □

Lemma 2.20. *For any $k \geq 6$, we have:*

$$u_k(t) \leq \begin{cases} k-3, & \text{if } 1 \leq t \leq A_k - A_{k-1} + 1, \\ k-4, & \text{if } A_k - A_{k-1} + 2 \leq t \leq A_k - A_{k-2} + 1, \\ k-5, & \text{if } A_k - A_{k-2} + 2 \leq t \leq A_k - A_{k-3} + 1, \\ k-6, & \text{if } A_k - A_{k-3} + 2 \leq t \leq A_k - A_{k-5} + 1, \\ k-r-5, & \text{if } A_k - A_{k-2r+1} + 2 \leq t \\ & \qquad \leq A_k - A_{k-2r-1} + 1, \quad \text{for any } r \in [3, \lfloor (k-3)/2 \rfloor]. \end{cases}$$

Proof. By strong induction on k, making use of the recurrence relation presented in Lemma 2.19. See Appendix A.6 for details. □

Proof of Theorem 2.12. Let $k \geq 7$ be minimal such that $A_k \geq \ell$. It follows from Item 5 of Proposition 2.14 that $k \leq \left\lceil \frac{\log \ell - \log 0.94}{\log \alpha} \right\rceil \leq \lceil 2.47 \log \ell + 0.23 \rceil$. We will run Algorithm 2.1 with this value of k. We associate index i of the length-A_k pebble game with index $i - (A_k - \ell)$ of the length-ℓ pebble game, dropping any operations for which $i < (A_k - \ell)$.

Because the first and last $A_k - \ell$ operations of the length-A_k pebble game operate only on sites with $i < (A_k - \ell)$, the depth of the resulting length-ℓ pebble game is at most $2A_k - 2(A_k - \ell) = 2\ell$ (and therefore it must be exactly 2ℓ). Thus we achieve optimal depth for any ℓ.

To analyze the space, we look at each phase separately:

- During blasting, Lemma 2.18 implies that the number of pebbles is always at most $\lceil (k-3)/2 \rceil + 1 = \lceil (k-1)/2 \rceil < k - 3$.
- When pebbling and unpebbling site ℓ, the number of pebbles is hence at most $\lceil (k+1)/2 \rceil \leq k - 3$.
- When unblasting, Lemma 2.20 implies that the number of pebbles is at most $k - 3$.

Thus the pebbling space is $\leq k - 3 \leq \lceil 2.47 \log \ell - 2.77 \rceil$, as claimed. □

2.4 Exactly Optimal Pebbling Schemes via A* Search

In Section 2 we construct a parallel pebbling scheme that achieves the exactly optimal depth of 2ℓ using at most $\lceil 2.47 \log \ell - 2.77 \rceil$ pebbles. However, there are several reasons that this is not the end of the story, in particular for concrete, practical use of pebble games. These include:

- It is not clear from established lower bounds that the constant factor on this pebble count is tight;
- One may desire to hold the number of pebbles s constant, and find the optimal-depth solution under that constraint (which may be greater than 2ℓ);
- There may be many solutions achieving that same optimal depth for a given s, and it may be desirable to use some other metric (e.g. minimizing total number of pebble operations) to choose among them;
- For practical situations (see Section 4), some extra ancillary space may be needed inside of Pebble(i) and Unpebble(i), and the amount may even vary with i.

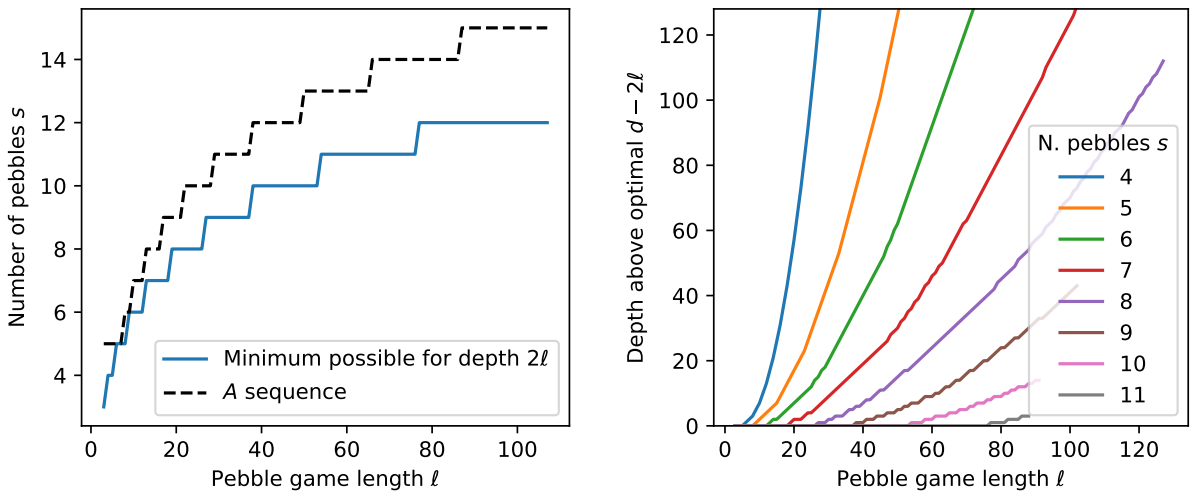

Figure 2: Results of the A^* search for optimal parallel pebble games. In the left figure, we plot the minimum number of pebbles s with which it is possible to achieve the absolute optimal depth 2ℓ, and compare with the space used by the A-sequence construction of Theorem 2.12. In the right figure, we plot the depth as a function of ℓ, for different constant values of the number of pebbles s.

For these reasons, we have implemented in the Julia programming language a highly optimized A* search to find the absolutely optimal pebble game for concrete values of the parameters, such as the number of pebbles s and the pebble game length ℓ. In the left plot of Figure 2, we compare the results output by this search to Theorem 2.12. We find that while the algorithm of Theorem 2.12 is nearly optimal, the constant factor still has room to be improved slightly. In the right plot of Figure 2, we plot the optimal depth achievable for constant values of s, as a function of ℓ. We see that once it is no longer possible to achieve the optimal depth of 2ℓ, the optimal depth for very small numbers of pebbles increases dramatically; meanwhile, for slightly larger numbers of pebbles, the depth remains close to optimal even as the pebble game length ℓ continues to increase. The Julia code is available on GitHub and Zenodo: `https://doi.org/10.5281/zenodo.17298960` [KMRV25].

3 Preliminaries for Our Factoring Application

Let $N < 2^n$ be an n-bit number that we wish to factor. We use $\|\cdot\|$ to denote the ℓ_2 norm of a vector with real entries.

3.1 Classical Algorithms for Lattice Problems

The core of the classical post-processing procedure in Regev's quantum factoring algorithm [Reg25] is an approximate solution to the following lattice problem, which we refer to as the short bounded basis problem (SBBP):

Definition 3.1 (α-approximate Short Bounded Basis Problem)**.** The α- approximate short bounded basis problem SBBP_α is the following task: given as input a basis for a lattice $\mathcal{L} \subset \mathbb{R}^r$ together with a real number $T > 0$, output a list of vectors $z_1, \ldots, z_t \in \mathcal{L}$ (with $t \leq r$) such that both of the following conditions are true:

1. (Shortness) For all $i \in [t]$, we have $\|z_t\| \leq \alpha T$.

2. (Correctness) Any vector in $\mathcal{L}$ of norm $\leq T$ must be an integer linear combination of $z_1, \ldots, z_t$.

Remark 3.2 (Related lattice problems). This problem is closely related to the Short Basis Problem [Ajt99]; in fact, they will be equivalent when $T > \lambda_r(\mathcal{L})$. While it is likely that SBBP_α can generically be reduced to a more standard lattice problem such as $\mathsf{SVP}_{\alpha'}$ (α'-approximate shortest vector), these reductions typically incur concrete losses in the parameters. Our focus is on algorithms that directly solve this problem via lattice basis reduction, following the template laid out by [Reg25, Claim 5.1].

Heuristic algorithms. To aid our concrete resource estimates in Section 4, we will focus on algorithms that work well in practice, rather than those that come with provable guarantees such as LLL [LLL82] and slide reduction [GN08]. One such algorithm is BKZ basis reduction which is analyzed heuristically using the root-Hermite factor as a metric:

Definition 3.3 (Root-Hermite factor)**.** Suppose that a lattice basis reduction algorithm takes as input an arbitrary basis for a lattice $\mathcal{L} \subseteq \mathbb{R}^r$ and outputs a reduced basis $\mathbf{b}_1, \ldots, \mathbf{b}_r$. The root-Hermite factor achieved by this algorithm is the value $\delta_0 > 0$ such that:

$$\|\mathbf{b}_1\| = \delta_0^r (\det \mathcal{L})^{1/r}.$$

We will work with the following heuristic, which we justify in Appendix C.1.

Assumption 3.1. *If a lattice basis reduction algorithm achieves root-Hermite factor δ_0 in dimension r, then it can replace the use of LLL reduction in Regev's algorithm and yield an algorithm for SBBP_α, where:*

$$\alpha = \frac{\sqrt{r}}{2} \cdot \delta_0^{2r}.$$

We will base our analyses on BKZ lattice reduction with block size $\beta = 60, 120, 160, 200$. Looking ahead, we will be applying these algorithms to lattices of dimension in the 300-600 range. Given this, we tabulate estimates of the corresponding root-Hermite factors and costs in 600 dimensions, obtained using the online estimator[6] by Albrecht et al [APS15], in Table 2. These cost estimates indicate that our setups range from being feasible for most academic institutions ($\beta = 60$) to being beyond all known implementations but still not totally inconceivable ($\beta = 200$).

3.2 Applying SBBP_α to Regev Factoring

We refer the reader to [RV24, Section 3] for a high-level overview of Regev's factoring algorithm, here conveying only the core points. The broad structure of the algorithm consists of running a particular quantum circuit several times, and performing classical postprocessing to combine the measurement results in a way that reveals the factors of the input N. The main operation in the quantum circuit is the application of following unitary (here defined by its action on basis states):

$$\mathcal{U}'(\mathbf{a}, N) |\mathbf{z}\rangle |0\rangle = |\mathbf{z}\rangle |\prod_j a_j^{z_j} \bmod N\rangle \tag{2}$$

where $\mathbf{z} \in \mathbb{Z}^n$ and $\mathbf{a} \in \mathbb{Z}_N^n$ is classical. It is the implementation of this unitary to which we apply our pebbling techniques.

With this in mind, let us begin with some notation.

[6] Available at `https://github.com/malb/lattice-estimator`.

- Let d the dimension of the lattice considered by Regev [Reg25], which the algorithm designer is free to select;
- let R denote the radius of the initial quantum superposition over $\mathbb{Z}^d$;
- let $m \geq d + 4$ be the number of times we run and measure the quantum circuit; and
- let $r = d + m$.

For clarity, we also explicitly restate Regev's number-theoretic assumption in the setting of general dimensions d. We work with the variant used by Ekerå and Gärtner [EG24, Assumption 1], which is stronger but comes with the (relatively minor) benefit of allowing one to work with primes rather than squares of primes as the bases.

Conjecture 3.1 (Assumption 1 of [EG24])**.** *For the base integers $a_1, \ldots, a_d$, define the following lattice in d dimensions:*

$$\mathcal{L} = \left\{ (z_1, \ldots, z_d) \in \mathbb{Z}^d \ : \ \prod_{j=1}^{d} a_j^{z_j} \equiv 1 \bmod N \right\}.$$

Then we conjecture that there exists a basis of $\mathcal{L}$ with all vectors of ℓ_2 norm at most $2^{Cn/d}$ for some given constant $C > 0$.

A simple heuristic argument based on the pigeonhole principle and put forth by Regev [Reg25] suggests that taking $C = 1 + \epsilon$ likely suffices. Below, we abstract out the classical post-processing procedure of Regev [Reg25] in terms of SBBP_α.

Lemma 3.4 (Follows from analysis in Section 5 of [Reg25])**.** *Let $\alpha > 0$ be such that the following inequality is satisfied:*

$$\alpha \cdot \sqrt{m+1} \cdot 2^{Cn/d} < \frac{\sqrt{2} \cdot R}{\sqrt{d}} \cdot (4 \cdot 2^n)^{-1/m} / 6. \tag{3}$$

Then, assuming Conjecture 3.1 and given the output from m runs of the quantum circuit as input, it is possible to classically output a nontrivial factor of N with one call to an oracle that solves SBBP_α in $r = d + m$ dimensions, and some additional polynomial-time operations.

Remark 3.5. As shown in [EG24, Appendix A], assuming a stronger variant of Conjecture 3.1 and making some low-cost tweaks to Regev's algorithm allows one to find the multiplicative order of an arbitrary $g \in \mathbb{Z}_N^*$. By [Eke21, Eke24], this then typically suffices to obtain a complete factorization of N rather than just one nontrivial factor.

4 Application: Concrete Resource Estimates for Regev Factoring

Recall from our overview in Section 1.1.2 that we have *asymptotically* efficient circuits implementing Regev's factoring algorithm [Reg25, RV24, Rag24]; however, for cryptographically relevant problem sizes these leave much to be desired in terms of practical costs [EG25]. In this section, we apply our pebbling techniques introduced in Section 2 to make progress towards remedying this problem. We focus on n-bit RSA factoring for $n \in \{2048, 4096\}$.

4.1 From Pebbling to Factoring

Here we explicitly describe how we cast the implementation of Equation (2) as a pebble game. As in Regev's original paper, our task is to compute $(z_1, \dots, z_d) \mapsto \prod_{j=1}^{d} a_j^{z_j} \bmod N$ via repeated squaring. Here, $a_1, \dots, a_d$ are some classically known small bases and the z_j's are in quantum superposition. They will range from 0 to $D-1$, where D is a power of 2. Regev [Reg25] originally proposed to structure this computation as follows:

1. $y_0 \leftarrow 1$
2. for i from 1 to $\log D$
   (a) $y_{2i-1} \leftarrow y_{2i-2}^2 \bmod N$
   (b) $y_{2i} \leftarrow \left(\prod_{j=1}^{d} a_j^{z_{j,i}}\right) y_{2i-1} \bmod N$
3. return $y_{2\log D}$

where $z_{j,i}$ is the ith highest-order bit of z_j (indexed from 1). This algorithm can be interpreted as a line graph of length $2\log D$, where each of the y_* are vertices and the operations $y_{2i-1} \leftarrow y_{2i-2}^2 \bmod N$ and $y_{2i} \leftarrow (\prod_{j=1}^{d} a_j^{z_{j,i}}) y_{2i-1} \bmod N$ correspond to the odd- and even-indexed edges, respectively. Thus we may use pebble games to efficiently compute the final value $y_{\log D} = \prod_j a_j^{z_j} \bmod N$. Indeed, applying our spooky pebbling scheme from Section 2 yields the following theorem:

Theorem 4.1. *There exists a quantum circuit for the unitary $\mathcal{U}$ of Equation (2) which has (n-bit) multiplication depth $4\log D$ and uses at most $1.3(2n + S_\times(n) + o(n))\log\log D$ qubits of space, where $S_\times(n)$ is the number of ancilla qubits used by an n-bit multiplication mod N and we assume $\log\left[\prod_j a_j\right] = o(n)$.*

Proof. See Appendix D. □

Extending on this theoretical analysis, we also numerically compute the multiplication depth for certain concrete problem sizes (see Section 4.3). For practical efficiency in that analysis we make the following further optimizations, alluded to earlier in Section 1.1:

1. We use an amortization trick introduced by [Rag24] that bears some similarity to windowing [Gid19b] to reduce the frequency with which products like $\prod_j a_j^{z_{j,i}}$ need to be multiplied in, reducing the overall length of the pebble game.
2. We immediately measure the final value $y_{2\log D} = \prod_i a_i^{z_i} \bmod N$ when it is produced rather than insisting on keeping it around until the end as in Equation (2). This allows the qubits holding that value to be recycled.
3. We compute each product $\prod_{j=1}^{d} a_j^{z_{i,j}}$ by simply computing each partial product $\prod_{j=1}^{j_{\max}} a_j^{z_{i,j}}$ via a series of in-place classical-quantum multiplies, rather than using a product tree as in [Reg25]. This saves a significant number of qubits in practice and does not asymptotically hurt the gate count when using schoolbook arithmetic. This is because we do not need to store an entire product tree with d leaves. Indeed, using Gidney's recent low-space classical-quantum adder m times [Gid25a], we can multiply an m-qubit quantum value by a classical one in-place using only two total ancilla qubits.
4. In $\mathsf{Pebble}(2i)$, rather than explicitly uncomputing the temporary value $t_i = \prod_{j=1}^{d} a_j^{z_{i,j}}$, we use measurement-based uncomputation to "ghost" it.

5. We make our pebbling strategies *ancilla-aware*—numerically optimizing them in a way that accounts for the fact that different pebble steps (multiplications and squarings) require different numbers of ancilla qubits.

We record our concrete algorithm with these optimizations by explicitly defining the operations Pebble and Unpebble (see Section 2) in Algorithms 4.1 and 4.2 respectively. In these algorithms we assume without loss of generality that the final pebble index is divisible by the window size w. Also, the notation $z_{j,[i:i+w)}$ means "the integer consisting of bits i to $i+w$ of z_j", in highest-bit order as usual.

Finally, we remark that our use of pebbling and spookiness allows us to avoid cumbersome overheads and complications that arise in the Fibonacci-based implementations [RV24,Rag24] of Regev's circuit (which is reversible without relying on mid-circuit measurements). The main such overhead is the need to track the inverses mod N of intermediate quantities used in the algorithm, which we completely circumvent. Indeed, this is the main reason that our methods give concrete improvements over the Fibonacci-based approaches.

Algorithm 4.1: Pebble(i) for Regev factoring

Data: Classical values $\mathbf{a}$, N, window size parameter w, and pebble index $i \in [\log D - w + \log D/w]$.
Quantum registers $\mathbf{z}$, in, and out; at the start, in contains y_{i-1} and out contains 0.
Result: out is set to y_i, and the other registers are unchanged.
if $i \not\equiv 1 \pmod{w+1}$ **then**
  out $\leftarrow$ in^2 mod N
else
  Classically compute $i' = (i-1)w/(w+1)$.
  t $\leftarrow \prod_j a_j^{z_{j,[i':i'+w)}}$
  out $\leftarrow$ t $*$ in mod N
  Ghost(t)
end

Algorithm 4.2: Unpebble(i) for Regev factoring

Data: Classical values $\mathbf{a}$, N, window size parameter w, and pebble index $i \in [\log D - w + \log D/w]$.
Quantum registers $\mathbf{z}$, in, and out; at the start, in contains y_{i-1} and out contains y_i.
Result: out is set to 0, the other registers are unchanged, and all ghosts of intermediate values are removed.
if $i \not\equiv 1 \pmod{w+1}$ **then**
  out $\leftarrow$ out $-$ in^2 mod N
else
  Classically compute $i' = (i-1)w/(w+1)$.
  t $\leftarrow \prod_j a_j^{z_{j,[i':i'+w)}}$
  out $\leftarrow$ out $-$ t $*$ in mod N
  Unghost(t)
  t $\leftarrow$ t $- \prod_j a_j^{z_{j,[i':i'+w]}}$
end

4.2 Setup

Efficiency metrics. We will measure the cost of a quantum circuit in terms of the following metrics:

- We follow [EG25] and estimate the gate count and depth in terms of calls to the atomic operation $|u, v, t, 0^S\rangle \to |u, v, (t + uv) \bmod N, 0^S\rangle$, which we hereby refer to as a "mod N multiplication". In Algorithms 4.1 and 4.2, this means we charge a cost of 1 operation for each operation modifying out; we do not account here for the gates/depth required to compute t.

  We also tabulate the total number of gates across all runs (rather than just one); as expected, we will see that the large number of runs in all variants of Regev make it much worse than Shor by this metric.

- We provide qubit counts, ignoring the small constant number of ancilla qubits used by Gidney's classical-quantum multiplier [Gid25a] when computing t and also the ancilla qubits S for multiplications. We believe that the latter cost will likely be very small in practice [KY24].

  We note that these qubit counts are neglected consistently across our evaluations of Regev with Fibonacci arithmetic in Table 4 and our evaluations of Regev with spooky pebbling in Tables 5 and 6.

Classical post-processing power. As observed by Regev [Reg25], his factoring algorithm and its variants enjoy natural tradeoffs between quantum circuit complexity and classical postprocessing. To incorporate the effect of this into their cost analyses, Ekerå and Gärtner consider three regimes: LLL (which runs in polynomial time), BKZ-200 (which is not practically feasible, but also not too much of an exaggeration), and perfect lattice reduction (which is an extreme case and not even close to practically feasible).

We zoom in to the range of practically feasible (or potentially feasible in the future) algorithms and thus focus on BKZ-β for $\beta = 60, 120, 160, 200$. As explained in Section 3.1, these range from being very feasible with the computational resources typically available to an academic group, to being beyond anything that is currently feasible in practice. We use Albrecht's estimator to estimate the root-Hermite factors achieved by these algorithms, which are tabulated in Table 2. Looking ahead, we will plug these into Assumption 3.1 which we will in turn plug into Lemma 3.4.

Setting parameters in our algorithms. The first parameter that we vary will be the "window size" w that we introduced in Section 1.1.2. Once we do this, we consider any dimension d and number of runs m subject to the following constraints:

- d must be small enough that $\prod_{i=1}^{d} p_i^{2^w-1} \leq 2^n$, where p_i denotes the ith prime. We denote $d_{\max} := d_{\max}(n, w)$ to be the maximal integer d that would satisfy this constraint. This follows [EG25] and is intended to ensure that the cost of computing the product $\mathsf{t} = \prod_j a_j^{z_{j,[i':i'+w)}}$ does not become dominant.

- $m \geq d + 4$ (as required throughout the analysis by [Reg25]).

Recall from [Reg25] that D is the smallest power of 2 greater than or equal to $2\sqrt{d} \cdot R$ i.e. the length of our pebbling game $\log D$ is $\lceil \log(2\sqrt{d} \cdot R) \rceil$. For a particular w and BKZ blocksize β, we will select m, d that minimize $\log D$. It can be derived from Equation (3) and Assumption 3.1 (setting $C \approx 1$ as in Conjecture 3.1) that we require the following (see Appendix C.3 for details):

$$\log D > 2(m + d) \log \delta_0 + \frac{n}{d} + \frac{n}{m}$$

$$+ \tfrac{1}{2}\log(m+d) + \tfrac{1}{2}\log(m+1) + \log\frac{6d}{\sqrt{2}} + \tfrac{2}{m} \tag{4}$$

Remark 4.2 (Including lower-order terms when calculating $\log D$). We remark here that [EG25] ignores the lower-order terms in the above expression and approximates the lower bound on $\log D$ by just $2(m+d)\log\delta_0 + \frac{n}{d} + \frac{n}{m}$. While this is asymptotically a reasonable thing to do, it turns out that the concrete impact on the obtained value of $\log D$ is non-negligible. For example, if we work with $n = 2048$ and BKZ block size $\beta = 200$, and take $d = 222$ and $m = 303$ (as in Table 4), Equation (4) implies that we must take $\log D$ to be at least 44. However, if we were to discard the lower-order terms in the second line we would end up with a lower bound of only 26. Given this gap, we will deviate from [EG25] by including these lower-order terms in our computation of $\log D$. Note that this shifts any comparison between Shor and Regev in the favor of Shor's algorithm.

The final parameter that we vary given $\log D$ is the number of pebbles s in our pebbling game, which roughly scales linearly with the qubit usage of our circuit. Indeed, the qubit count will roughly be $d\log D + sn$. We restrict attention to $s \in [5, 12]$ since each additional pebble adds n qubits.

4.3 Results

We make a comparison of our results to related factoring circuits [RV24, Rag24, EH17] by broadly following the lead of [EG25], but with some important differences. In [EG25] the takeaway was that even with the optimizations of [RV24], Regev's algorithm is outperformed by Shor in both time *and* space simultaneously on practical input sizes (e.g. 2048 bits). Thus, even by a cost metric that ignores space usage (a choice which is favorable to Regev) and only counts total number of multiplications, the authors were able to argue that Shor has better performance in practice. Our parallel spooky pebble games improve the performance of Regev to an extent that the estimates for Shor from [RV24] are no longer clearly better simultaneously in all metrics—indeed, for some settings of parameters, Regev's algorithm with parallel spooky pebbling may outperform in depth but require more space (see below). For this reason, rather more nuance is required to make a fair comparison: how do we balance the cost of space versus depth? Furthermore, how do we account for the fact that Regev's algorithm requires many independent runs of the circuit, while Shor's algorithm can be performed in one shot [Eke21] or by combining the results of several shots, making each shot cheaper [EH17]?

Rather than devise some single metric that attempts to capture this complexity, and then optimize parameters according to that metric, we instead opt to provide cost estimates corresponding to several different settings of parameters for each algorithm, which yield various tradeoffs of depth vs time. First we describe the estimates for previous algorithms to which we compare our work, and note some subtleties about them:

- **Regev's algorithm [Reg25] with the optimizations by [RV24, Rag24].** The concrete costs of these algorithms were studied extensively by [EG25], using the analysis in [Rag24, Theorem 2.4]. Because of our interest in multiplication depth, we make some modifications to this circuit to benefit from parallelism, which we then use to obtain concrete results for comparison with our pebbling results. We describe these modifications and tabulate these results in Appendix G.2.

  As noted in Remark 4.2, we will incorporate lower-order terms in our computation of $\log D$. In other words, we calculate $\log D$ using Equation (4), including the lower-order terms in the second line.

- **Shor's algorithm [Sho97] combined with optimizations by Ekerå and Håstad [EH17, Eke20, Eke23].** These costs have already been calculated and reported by [EG25]; in part, we will use the

results from there without modifications, as tabulated in Table 3. Recall for comparison that this variant of Shor's algorithm uses $\approx 2n$ qubits when factoring an n-bit integer (although smaller is achievable using different methods and at the expense of gate count [Zal06, CFS25, Gid25b]). No known variant of Regev's algorithm, including the present work, achieves such a small space footprint; for comparison, one may wonder what depth could be achieved if this algorithm were allowed to use more space. It is nontrivial to parallelize Shor's algorithm while keeping gate and qubit counts well-controlled; in Appendix F we propose one way of doing so, which we use for the estimates we present.

We note for completeness that [EG25] incorporates windowing into their results with $w = 10$. As they note, this choice of w may be underestimating the benefit offered by windowing for Shor's algorithm. For simplicity, we maintain this choice for our estimates of the cost of this algorithm; we believe the qualitative takeaways of our results — in relation to the concrete potential of variants of Regev's algorithm — stand regardless of this fact.

Our methodology for evaluating our spooky pebbling schemes is as follows: for any n, β, w, we select d, m by iterating over all possible values subject to the constraints stated in "Setting parameters in our algorithms", and minimizing the required value of $\log D$ as dictated by Equation (4). Given $\log D$ and s, we then experimentally (see Section 2.4) find the optimal-depth pebbling game with length $\ell := (w+1)\left\lfloor \frac{\log D - 1}{w} \right\rfloor + 1$ and s pebbles and calculate the resulting depth and total number of mod N multiplications. Note that we pass information about the number of ancilla qubits required to the numerical optimization (see point 5 above). We note that numerical experiments showed that $w < 1$ yielded worse performance (because this is equivalent to turning off windowing), but $w > 2$ yielded worse performance as well because increasing w ends up leading to larger $\log D$, which ultimately increases the length ℓ of the pebble game. We report results for $w = 2$ and several values of s in Tables 5 and 6. See also Figure 1 for a visual depiction of the results, including more values of the time-space tradeoff depending on number of pebbles s.

Remark 4.3 (Choice of s). For most settings of the parameters in Tables 5 and 6, we record results for $s \in \{5, 7, 12\}$, as this gives a representative picture of the tradeoffs between space and depth. $s = 4$ is the only smaller value for which solutions to the pebble game exist at all; the depth and multiplication count of those solutions are very large (in the thousands), so we choose $s = 5$ instead. On the other side, $s = 12$ achieves the absolute optimal depth of 2ℓ for most settings of the parameters, and thus there is no value in using more pebbles. The value $s = 7$ achieves a reasonable tradeoff between the two. We note that for $n = 4096$ and BKZ-$\beta = 60$, we report results for $s = 8$ instead of $s = 12$, as our computational resources were not sufficient to compute the optimal pebbling sequence for $s = 12$ for those parameters.

4.3.1 Analysis

We refer the reader to Tables 3-6 for detailed results comparing these various factoring algorithms.

Comparing depth per shot. For $n = 2048$, Fibonacci-based approaches (Table 4) to the arithmetic in Regev's algorithm can only hope to attain a multiplication depth of 560-620 (depending on the BKZ block size that is used, see Table 4). However when using spooky pebbling with a similar qubit count ($s = 12$), our results requiring a multiplication depth of only 157-175 (Table 5). Using 5 or 7 pebbles yields somewhat worse multiplication depth (396-465 and 229-263, respectively), but still outperforms the Fibonacci variants and has the benefit of using much less space (than the other variants of Regev's algorithm; it still uses more space than Shor). Shor's algorithm as analyzed by [EG25] uses a multiplication depth of 230 if using

windowing and multiple runs. Via the parallelization strategy we describe in Appendix F, this depth can be reduced to 100 when using a similar number of qubits as the $s = 12$ spooky pebble game.

For $n = 4096$, Fibonacci-based approaches end up with a multiplication depth of 680-740, and here spooky pebbling with 8-12 pebbles fares better, requiring a multiplication depth of only 187-200 (for BKZ-120/160/200 using 12 pebbles) or 287 (for BKZ-60 using 8 pebbles) (Table 6). Even with 5 pebbles, the multiplication depth is only 513-610. Meanwhile, Shor's algorithm uses a multiplication depth of 444 as estimated by [EG25]. We note that in the intermediate-space regime roughly corresponding to an $s = 7$ spooky pebble game, parallelized Shor (Appendix F) requires a depth of 299 while our spooky pebble game requires depth 286-334 depending on the choice of BKZ block size. Thus in this intermediate regime the per-shot depth of Shor and Regev is now comparable.

Comparing # of mod N multiplications per run. Even if one were to compare the total number of mod N multiplications as in [EG25] (as opposed to the depth), spooky pebbling still performs quite well; the gap between depth and total # of mod N multiplications per run is small. This is in contrast to the Fibonacci case (Table 4) where there is a gap of a factor of 2. For example, for $n = 2048$ when using spooky pebbling with 12 pebbles, the total number of mod N multiplications per run ranges from 253-295, and for $n = 4096$ it ranges from 326-345. In particular, the total number of operations for $n = 4096$ still outperforms the 444 achieved by Shor when the BKZ block size is 120, 160, or 200.

Comparing # of mod N multiplications across all runs. As is evident from Tables 3-6, if we were to make this comparison counting total mod$N$ multiplications across all runs rather than per one run (which seems to be the important metric for setting error correction parameters, since we need a large fraction of the runs to be error-free), spooky pebbling is about an order of magnitude worse than Shor, but still outperforms the Fibonacci-based results in Table 4 by half an order of magnitude.

Comparing qubit counts. As far as space usage, Shor's algorithm remains the most efficient, requiring only $\approx 2n$ qubits when no parallelization is required (note that this holds for the variants considered in this work; recently it has been shown that reducing the qubit count to $0.5n + o(n)$ is possible [CFS25]). Fibonacci variants of Regev sit on the other extreme, requiring $\geq 13n$ qubits in all cases reported in our work, and likely quite a bit more in practice since these costs disregard the space required to hold the products $\prod_{j=1}^{d} a_j^{z_{j,i}}$. Our variants of Regev require $\approx 7n$-$15n$ qubits, and these costs also account for the space required to hold the intermediate products. Our results hence provide significant concrete improvements over Fibonacci-based implementations of Regev in all metrics, although they remain rather larger than those of Shor.

The effect of more lattice postprocessing power. We finally make a surprising observation that dramatic increases in the classical power required for the lattice postprocessing (corresponding to increasing β in Tables 4-6) have a somewhat marginal concrete effect on the number or depth of mod N multiplications per run. This is with the slight exception of spooky pebbling with $s = 5$, where the constrained space seems to enable more significant benefits from improved classical postprocessing power.

The previous study by [EG25] restricted attention to the extremes of LLL (which is essentially BKZ with block size 2) and BKZ-200, and thus did not arrive at the finding that the performance is qualitatively similar even with intermediate block sizes that are much smaller (and more practicable) than $\beta = 200$.

Acknowledgements. The authors would like to thank Craig Gidney, Isaac Chuang, Mikhail Lukin, Oded Regev, Peter Shor, Noah Stephens-Davidowitz, and Vinod Vaikuntanathan for insightful comments and discussions. We also thank anonymous reviewers for helpful feedback, including the suggestion to consider parallelized variants of Shor's algorithm (Appendix F) in our evaluation.

References

[ABP17] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Depth-Robust Graphs and Their Cumulative Memory Complexity. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, Advances in Cryptology – EUROCRYPT 2017, pages 3–32, Cham, 2017. Springer International Publishing. DOI 10.1007/978-3-319-56617-7_1. 1

[ABP18] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Sustained space complexity. In Jesper Buus Nielsen and Vincent Rijmen, editors, Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II, volume 10821 of Lecture Notes in Computer Science, pages 99–130. Springer, 2018. DOI 10.1007/978-3-319-78375-8_4. 1

[Ajt99] Miklós Ajtai. Generating hard instances of the short basis problem. In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings, volume 1644 of Lecture Notes in Computer Science, pages 1–9. Springer, 1999. DOI 10.1007/3-540-48523-6_1. 17

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. Journal of Mathematical Cryptology, 9(3) pages 169–203, October 2015. DOI 10.1515/jmc-2015-0016. 17, 37

[AS15] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. In Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing, STOC '15, page 595–603, New York, NY, USA, 2015. Association for Computing Machinery. DOI 10.1145/2746539.2746622. 1

[AT17] Joël Alwen and Björn Tackmann. Moderately hard functions: Definition, instantiations, and applications. In Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I, page 493–526, Berlin, Heidelberg, 2017. Springer-Verlag. DOI 10.1007/978-3-319-70500-2_17. 1

[BCDP96] David Beckman, Amalavoyal N Chari, Srikrishna Devabhaktuni, and John Preskill. Efficient networks for quantum factoring. Physical Review A, 54(2) page 1034, 1996. DOI 10.1103/PhysRevA.54.1034. 2

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016, pages 10–24. SIAM, 2016. DOI 10.1137/1.9781611974331.CH2. 37

[Bea03] Stéphane Beauregard. Circuit for Shor's algorithm using 2n+3 qubits. Quantum Inf. Comput., 3(2) pages 175–185, 2003. DOI 10.26421/QIC3.2-8. 2, 40

[Ben89] Charles H. Bennett. Time/Space Trade-Offs for Reversible Computation. SIAM Journal on Computing, 18(4) pages 766–776, August 1989. Publisher: Society for Industrial and Applied Mathematics, DOI 10.1137/0218053. 1, 2, 6

[BHL22] Jeremiah Blocki, Blake Holman, and Seunghoon Lee. The parallel reversible pebbling game: Analyzing the post-quantum security of imhfs. In Eike Kiltz and Vinod Vaikuntanathan, editors, Theory of Cryptography - 20th International Conference, TCC 2022, Chicago, IL, USA, November 7-10, 2022, Proceedings, Part I, volume 13747 of Lecture Notes in Computer Science, pages 52–79. Springer, 2022. DOI 10.1007/978-3-031-22318-1_3. 1, 2, 5, 7, 11

[BHL25] Jeremiah Blocki, Blake Holman, and Seunghoon Lee. The impact of reversibility on parallel pebbling. In Serge Fehr and Pierre-Alain Fouque, editors, Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4-8, 2025, Proceedings, Part I, volume 15601 of Lecture Notes in Computer Science, pages 124–154. Springer, 2025. DOI 10.1007/978-3-031-91107-1_5. 1, 5, 7, 11

[CEMM98] Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. Proc. Roy. Soc. Lond. A, v.454 page 339, 1998. DOI 10.1098/rspa.1998.0164. 6

[CFS25] Clémence Chevignard, Pierre-Alain Fouque, and André Schrottenloher. Reducing the number of qubits in quantum factoring. In Yael Tauman Kalai and Seny F. Kamara, editors, Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2025, Proceedings, Part II, volume 16001 of Lecture Notes in Computer Science, pages 384–415. Springer, 2025. DOI 10.1007/978-3-032-01878-6_13. 2, 3, 4, 23, 24, 40

[Cop02] D. Coppersmith. An approximate Fourier transform useful in quantum factoring, January 2002. DOI 10.48550/arXiv.quant-ph/0201067. 2

[CW00] Richard Cleve and John Watrous. Fast parallel circuits for the quantum Fourier transform. In 41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA, pages 526–536. IEEE Computer Society, 2000. DOI 10.1109/SFCS.2000.892140. 2

[DLH+22] Emma Deist, Yue-Hui Lu, Jacquelyn Ho, Mary Kate Pasha, Johannes Zeiher, Zhenjie Yan, and Dan M. Stamper-Kurn. Mid-Circuit Cavity Measurement in a Neutral Atom Array. Physical Review Letters, 129(20) page 203602, November 2022. DOI 10.1103/PhysRevLett.129.203602. 5

[EG24] Martin Ekerå and Joel Gärtner. Extending Regev's factoring algorithm to compute discrete logarithms. In Markku-Juhani Saarinen and Daniel Smith-Tone, editors, Post-Quantum Cryptography, pages 211–242, Cham, 2024. Springer Nature Switzerland. DOI 10.1007/978-3-031-62746-0_10. 4, 18

[EG25] Martin Ekerå and Joel Gärtner. A high-level comparison of state-of-the-art quantum algorithms for breaking asymmetric cryptography. IACR Communications in Cryptology, 2(1), 2025. DOI 10.62056/ayzojb0kr. 3, 4, 8, 18, 21, 22, 23, 24, 36, 39, 41, 43

[EH17] Martin Ekerå and Johan Håstad. Quantum algorithms for computing short discrete logarithms and factoring RSA integers. In Tanja Lange and Tsuyoshi Takagi, editors, Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings, volume 10346 of Lecture Notes in Computer Science, pages 347–363. Springer, 2017. DOI 10.1007/978-3-319-59879-6_20. 2, 3, 22, 39, 41

[Eke20] Martin Ekerå. On post-processing in the quantum algorithm for computing short discrete logarithms. Des. Codes Cryptogr., 88(11) pages 2313–2335, 2020. DOI 10.1007/S10623-020-00783-2. 22, 41

[Eke21] Martin Ekerå. On completely factoring any integer efficiently in a single run of an order-finding algorithm. Quantum Inf. Process., 20(6) page 205, 2021. DOI 10.1007/S11128-021-03069-1. 18, 22

[Eke23] Martin Ekerå. On the success probability of the quantum algorithm for the short DLP. CoRR, v.abs/2309.01754, 2023. DOI 10.48550/ARXIV.2309.01754. 22, 41

[Eke24] Martin Ekerå. On the success probability of quantum order finding. ACM Trans. Quantum Comput., 5(2) pages 11:1–11:40, 2024. DOI 10.1145/3655026. 18

[GE21] Craig Gidney and Martin Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. Quantum, v.5 page 433, 2021. DOI 10.22331/q-2021-04-15-433. 2, 4, 8

[Gid18] Craig Gidney. Factoring with n+2 clean qubits and n-1 dirty qubits, January 2018. DOI 10.48550/arXiv.1706.07884. 2

[Gid19a] Craig Gidney. Spooky Pebble Games and Irreversible Uncomputation. https://algassert.com/post/1905, 2019. [Accessed 19-02-2025]. 1, 2, 5, 6, 10

[Gid19b] Craig Gidney. Windowed quantum arithmetic, May 2019. DOI 10.48550/arXiv.1905.07682. 2, 3, 4, 8, 19

[Gid25a] Craig Gidney. A Classical-Quantum Adder with Constant Workspace and Linear Gates, July 2025. DOI 10.48550/arXiv.2507.23079. 9, 19, 21

[Gid25b] Craig Gidney. How to factor 2048 bit RSA integers with less than a million noisy qubits, May 2025. DOI 10.48550/arXiv.2505.15917. 2, 4, 23, 40

[GN08] Nicolas Gama and Phong Q. Nguyen. Finding short lattice vectors within Mordell's inequality. In Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing, STOC '08, page 207–216, New York, NY, USA, 2008. Association for Computing Machinery. DOI 10.1145/1374376.1374408. 17, 36

[HHB+25] Daniel Hothem, Jordan Hines, Charles Baldwin, Dan Gresh, Robin Blume-Kohout, and Timothy Proctor. Measuring error rates of mid-circuit measurements. Nature Communications, 16(1) page 5761, July 2025. DOI 10.1038/s41467-025-60923-x. 5

[HRS17] Thomas Häner, Martin Roetteler, and Krysta M. Svore. Factoring using $2n + 2$ qubits with Toffoli based modular multiplication. Quantum Inf. Comput., 17(7&8) pages 673–684, 2017. DOI 10.26421/QIC17.7-8-7. 2

[Kal17] Burton S. Kaliski, Jr. Targeted Fibonacci Exponentiation, November 2017. DOI 10.48550/arXiv.1711.02491. 3, 7

[KMRV25] Gregory D. Kahanamoku-Meyer, Seyoon Ragavan, and Katherine Van Kirk. GregDMeyer/parallel-spooky-pebbling: v0.1, October 2025. DOI 10.5281/zenodo.17298960. 2, 16

[KSMM23] Jin Ming Koh, Shi-Ning Sun, Mario Motta, and Austin J. Minnich. Measurement-induced entanglement phase transition on a superconducting quantum processor with mid-circuit readout. Nature Physics, 19(9) pages 1314–1319, September 2023. DOI 10.1038/s41567-023-02076-6. 5

[KSS25] Niels Kornerup, Jonathan Sadun, and David Soloveichik. Tight Bounds on the Spooky Pebble Game: Recycling Qubits with Measurements. Quantum, v.9 page 1636, February 2025. DOI 10.22331/q-2025-02-18-1636. 1, 2, 10, 11

[KY24] Gregory D. Kahanamoku-Meyer and Norman Y. Yao. Fast quantum integer multiplication with zero ancillas, March 2024. DOI 10.48550/arXiv.2403.18006. 2, 21, 38

[LLL82] Arjen K Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. Mathematische annalen, v.261 pages 515–534, 1982. DOI 10.1007/BF01457454. 17

[LS90] Robert Y. Levine and Alan T. Sherman. A Note on Bennett's Time-Space Tradeoff for Reversible Computation. SIAM Journal on Computing, 19(4) pages 673–677, August 1990. Publisher: Society for Industrial and Applied Mathematics, DOI 10.1137/0219046. 1, 2

[Ozh99] Y. Ozhigov. Quantum computer can not speed up iterated applications of a black box. In Colin P. Williams, editor, Quantum Computing and Quantum Communications, pages 152–159, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. DOI 10.1007/3-540-49208-9_12. 1

[Rag24] Seyoon Ragavan. Regev factoring beyond Fibonacci: Optimizing prefactors. Cryptology ePrint Archive, Paper 2024/636, 2024. https://eprint.iacr.org/2024/636, Online at https://eprint.iacr.org/2024/636. 3, 7, 8, 18, 19, 20, 22, 38, 39, 41, 43

[Reg25] Oded Regev. An efficient quantum factoring algorithm. J. ACM, 72(1) article 10 (13 pages), January 2025. DOI 10.1145/3708471. 1, 3, 4, 5, 7, 8, 16, 17, 18, 19, 21, 22, 36, 37

[RSW96] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, MIT, Weizmann Institute of Science, UC Berkeley, USA, 1996. 1

[RV24] Seyoon Ragavan and Vinod Vaikuntanathan. Space-efficient and noise-robust quantum factoring. In Leonid Reyzin and Douglas Stebila, editors, Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part VI, volume 14925 of Lecture Notes in Computer Science, pages 107–140. Springer, 2024. DOI 10.1007/978-3-031-68391-6_4. 3, 4, 7, 17, 18, 20, 22, 37, 43

[Sei01] Jean-Pierre Seifert. Using fewer qubits in Shor's factorization algorithm via simultaneous Diophantine approximation. In Cryptographers' Track at the RSA Conference, pages 319–327. Springer, 2001. DOI 10.1007/3-540-45353-9_24. 2, 3

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput., 26(5) pages 1484–1509, 1997. DOI 10.1137/S0097539795293172. 2, 4, 7, 22, 39, 41

[TK06] Yasuhiro Takahashi and Noboru Kunihiro. A quantum circuit for Shor's factoring algorithm using $2n + 2$ qubits. Quantum Information & Computation, 6(2) pages 184–192, 2006. DOI 10.1007/s11128-023-04159-y. 2

[VBE96] Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. Physical Review A, 54(1) page 147, 1996. DOI 10.1103/PhysRevA.54.147. 2

[Zal06] Christof Zalka. Shor's algorithm with fewer (pure) qubits, January 2006. DOI 10.48550/arXiv.quant-ph/0601097. 2, 3, 23

[ZKL+23] Daiwei Zhu, Gregory D. Kahanamoku-Meyer, Laura Lewis, Crystal Noel, Or Katz, Bahaa Harraz, Qingfeng Wang, Andrew Risinger, Lei Feng, Debopriyo Biswas, Laird Egan, Alexandru Gheorghiu, Yunseong Nam, Thomas Vidick, Umesh Vazirani, Norman Y. Yao, Marko Cetina, and Christopher Monroe. Interactive cryptographic proofs of quantumness using mid-circuit measurements. Nature Physics, 19(11) pages 1725–1731, November 2023. DOI 10.1038/s41567-023-02162-9. 5

A Deferred Proofs from Section 2.3

A.1 Proof of Proposition 2.14

Proof of Items 1 and 2. We prove all the assertions by strong induction on k. We first address the base cases where $k \leq 5$. We compute $A_1 = 1, A_2 = 2, A_3 = 2, A_4 = 3, A_5 = 4$. The claim about $A_{k-1} \leq A_k$ is now evident, and it remains to check the second assertion:

- When $k = 4$, we have $A_k = 3$ and $\sum_{i=1}^{\lfloor (k-2)/2 \rfloor} A_{k-2i-1} = A_{k-3} = 1$.
- When $k = 5$, we have $A_k = 4$ and $\sum_{i=1}^{\lfloor (k-2)/2 \rfloor} A_{k-2i-1} = A_{k-3} = 2$.

Now for the inductive step. For any $k \geq 6$, we have:

$$\begin{aligned} A_k &= A_{k-2} + A_{k-3} \\ &> A_{k-4} + A_{k-3} \\ &= A_{k-1}. \end{aligned}$$

The inequality is strict because $A_{k-2} \geq A_{k-3} \geq A_{k-4}$ and at least one of these inequalities is strict. Secondly, we have:

$$\begin{aligned} A_k &= A_{k-3} + A_{k-2} \\ &= A_{k-3} + 2 + \sum_{i=1}^{\lfloor (k-4)/2 \rfloor} A_{k-2i-3} \quad \text{(induction hypothesis)} \\ &= A_{k-3} + 2 + \sum_{i=2}^{\lfloor (k-2)/2 \rfloor} A_{k-2i-1} \end{aligned}$$

$$= 2 + \sum_{i=1}^{\lfloor (k-2)/2 \rfloor} A_{k-2i-1}.$$

□

Proof of Item 3. We prove this by induction on k. The base case $k = 6$ can be checked manually: $A_6 = 5 = 4 + 1 = A_5 + A_1$. For the inductive step, consider any $k \geq 7$ and assume the assertion holds for $k - 1$. Then we have that:

$$\begin{aligned} A_k - A_{k-1} &= (A_{k-2} + A_{k-3}) - A_{k-1} \\ &= A_{k-3} - (A_{k-1} - A_{k-2}) \\ &= A_{k-3} - A_{k-6} \\ &= A_{k-5}. \end{aligned}$$

□

Proof of Item 4. For $k \geq 7$, this is immediate from Items 1 and 3. The cases $k = 4, 5, 6$ are straightforward to check manually. □

Proof of Item 5. We prove this by strong induction. For $k = 7, 8, 9$, this can be verified manually. For $k \geq 10$, we have:

$$\begin{aligned} A_k &= A_{k-2} + A_{k-3} \\ &\geq 0.94 \left(\alpha^{k-2} + \alpha^{k-3} \right) \\ &= 0.94 \alpha^k, \end{aligned}$$

by definition of α. □

A.2 Proof of Lemma 2.15

This is straightforward by strong induction on k. For $k = 2, 3$ we have $A_k - 1 = 1$ and indeed we only have one step. For $k > 3$, the first two parts comprise A_{k-3} time steps. The recursive call to $\mathsf{Blast}(i_{\mathsf{start}} + A_{k-3}, i_{\mathsf{end}})$ uses $A_{k-2} - 1$ time steps by the induction hypothesis. In total, this is $A_{k-3} + (A_{k-2} - 1) = A_k - 1$ time steps as desired. □

A.3 Proof of Lemma 2.16

This is again by strong induction on k. For $k = 2, 3$, we have $A_k - 1 = 1$ and only one time step. For $k > 3$, note by the induction hypothesis that $\mathsf{Unblast}(i_{\mathsf{start}} + A_{k-3}, i_{\mathsf{end}})$ uses $A_{k-2} - 1$ time steps and $\mathsf{Unblast}(i_{\mathsf{start}}, i_{\mathsf{start}} + A_{k-3} - 1)$ uses $A_{k-3} - 1$ time steps. Also, by Lemma 2.15, $\mathsf{Blast}(i_{\mathsf{start}}, i_{\mathsf{start}} + A_{k-3} - 1)$ uses $A_{k-3} - 1$ time steps. We have four stages:

1. The first $A_{k-2} - A_{k-3}$ time steps of $\mathsf{Unblast}(i_{\mathsf{start}} + A_{k-3}, i_{\mathsf{end}})$.
2. An additional $A_{k-3} - 1 = (A_{k-2} - 1) - (A_{k-2} - A_{k-3})$ time steps go towards completing $\mathsf{Unblast}(i_{\mathsf{start}} + A_{k-3}, i_{\mathsf{end}})$. This is also the required number of time steps for $\mathsf{Blast}(i_{\mathsf{start}}, i_{\mathsf{start}} + A_{k-3} - 1)$.
3. One time step to unpebble $i_{\mathsf{start}} + A_{k-3}$.

4. The final $A_{k-3} - 1$ time steps to $\mathsf{Unblast}(i_{\mathsf{start}}, i_{\mathsf{start}} + A_{k-3} - 1)$.

In total, this is $(A_{k-2} - A_{k-3}) + (A_{k-3} - 1) + 1 + (A_{k-3} - 1) = A_k - 1$ time steps, which completes the proof. □

A.4 Proof of Lemma 2.18

We do this by induction on k. We first address the base cases where $k = 2, 3$. In this case, we simply have $b_k(1) = 1$. Indeed, we have:

$$\left\lceil \frac{k - A^{-1}(A_k - t + 1)}{2} \right\rceil + 1 = \left\lceil \frac{k - A^{-1}(2)}{2} \right\rceil + 1 = \left\lceil \frac{k-3}{2} \right\rceil + 1 = 1.$$

For clarity, we also manually address the case where $k = 4$. In this case, we have $b_k(1) = 1$ and $b_k(2) = 2$, and indeed we have:

$$\left\lceil \frac{k - A^{-1}(A_k - 1 + 1)}{2} \right\rceil + 1 = \left\lceil \frac{4-4}{2} \right\rceil + 1 = 1 = b_4(1), \text{ and}$$

$$\left\lceil \frac{k - A^{-1}(A_k - 2 + 1)}{2} \right\rceil + 1 = \left\lceil \frac{k-3}{2} \right\rceil + 1 = 2 = b_4(2).$$

Now for the inductive step: assume that $k \geq 5$ and that we have shown the result for $k - 2$. We consider a few different ranges of t:

- For $t = 1$, we have $b_k(t) = 1$, and indeed we have:

$$\left\lceil \frac{k - A^{-1}(A_k - t + 1)}{2} \right\rceil + 1 = \left\lceil \frac{k - A^{-1}(A_k)}{2} \right\rceil + 1 = 1.$$

- For $t \in [2, A_{k-3}]$, we have $b_k(t) = 2$. Moreover, we have $A_k - t + 1 \in [A_{k-2} + 1, A_k - 1] \Rightarrow A^{-1}(A_k - t + 1) \in \{k-2, k-1\}$, so it follows that:

$$\left\lceil \frac{k - A^{-1}(A_k - t + 1)}{2} \right\rceil + 1 = \left\lceil \frac{1 \text{ or } 2}{2} \right\rceil + 1 = 2.$$

- For $t \in [A_{k-3} + 1, A_k - 1]$, we have $b_k(t) = 1 + b_{k-2}(t - A_{k-3})$. The extra pebble comes from the pebble that has been placed at site $i_{\mathsf{start}} + A_{k-3}$. Using the induction hypothesis, it follows that:

$$\begin{aligned} b_k(t) &= 1 + b_{k-2}(t - A_{k-3}) \\ &= 2 + \left\lceil \frac{(k-2) - A^{-1}(A_{k-2} - (t - A_{k-3}) + 1)}{2} \right\rceil \\ &= 2 + \left\lceil \frac{(k-2) - A^{-1}(A_k - t + 1)}{2} \right\rceil \\ &= \left\lceil \frac{k - A^{-1}(A_k - t + 1)}{2} \right\rceil + 1, \end{aligned}$$

  which completes the induction.

□

A.5 Proof of Lemma 2.19

This is straightforward from the recursive formulation of Algorithm 2.3. We address each case in sequence:

- For $t \in [1, A_{k-2} - A_{k-3}]$, the only thing we are doing is unblasting on the segment $[i_{\text{start}} + A_{k-3}, i_{\text{end}}]$. There is one extra pebble sitting idly in position $i_{\text{start}} + A_{k-3}$, which gives us a total pebble usage of $1 + u_{k-2}(t)$.
- For $t \in [A_{k-2} - A_{k-3} + 1, A_{k-2} - 1]$, we additionally commence blasting on the segment $[i_{\text{start}}, i_{\text{start}} + A_{k-3} - 1]$, which contributes an additional $b_{k-3}(t - (A_{k-2} - A_{k-3}))$ pebbles.
- At $t = A_{k-2}$, all we are doing is unpebbling site $i_{\text{start}} + A_{k-3}$. It is apparent from this that the pebbling usage of this step is $b_{k-3}(A_{k-3} - 1) + 1$. Thus we have:

$$\begin{aligned} u_k(t) &= b_{k-3}(A_{k-3} - 1) + 1 \\ &= \left\lceil \frac{(k-3) - A^{-1}(A_{k-3} - (A_{k-3} - 1) + 1)}{2} \right\rceil + 2 \\ &= \left\lceil \frac{(k-3) - A^{-1}(A_{k-3} - (A_{k-3} - 1) + 1)}{2} \right\rceil + 2 \\ &= \left\lceil \frac{k-2}{2} \right\rceil \\ &= \left\lfloor \frac{k-1}{2} \right\rfloor. \end{aligned}$$

- For $t \in [A_{k-2} + 1, A_k - 1]$, all we are doing is unblasting on the segment $[i_{\text{start}}, i_{\text{start}} + A_{k-3} - 1]$. It follows that the pebbling usage of this step is $u_{k-3}(t - A_{k-2})$.

This completes the proof of the lemma. □

A.6 Proof of Lemma 2.20

The cases $k = 6, 7, 8$ may be manually checked. For convenience, we list the relevant numbers below:

- $k = 6$: we have $A_k = 5$ and $u_k([1, 4]) = [3, 3, 2, 1]$.
- $k = 7$: we have $A_k = 7$ and $u_k([1, 6]) = [4, 4, 4, 3, 2, 1]$.
- $k = 8$: we have $A_k = 9$ and $u_k([1, 8]) = [4, 5, 5, 4, 3, 3, 2, 1]$.

Now for the inductive step: assume $k \geq 9$ and the result has been shown for $k - 2$ and $k - 3$. Before we begin, note for simplicity that when $k \geq 5$ and $t \in [A_{k-2} - A_{k-3} + 1, A_{k-2} - 1]$, we have that:

$$\begin{aligned} u_k(t) &= b_{k-3}(t - (A_{k-2} - A_{k-3})) + 1 + u_{k-2}(t) \\ &= \left\lceil \frac{k - 3 - A^{-1}(A_{k-3} - (t - (A_{k-2} - A_{k-3})) + 1)}{2} \right\rceil + 2 + u_{k-2}(t) \\ &= \left\lceil \frac{k + 1 - A^{-1}(A_{k-2} - t + 1)}{2} \right\rceil + u_{k-2}(t). \end{aligned}$$

We now just enumerate through various ranges of t and check each case. Note that by Items 1 and 3 of Proposition 2.14, we have that $A_{k-2} - A_{k-3} = A_{k-7} < A_{k-5} = A_{k-1} - A_k$ (since $k \geq 8$).

- $t \in [1, A_{k-2} - A_{k-3}]$: we have $u_k(t) = 1 + u_{k-2}(t) \leq 1 + ((k-2)-3) = k-4 < k-3$.
- $t = A_{k-2} - A_{k-3} + 1$: we have:

$$\begin{aligned} u_k(t) &= \left\lceil \frac{k+1-A^{-1}(A_{k-2}-t+1)}{2} \right\rceil + u_{k-2}(t) \\ &= \left\lceil \frac{k+1-A^{-1}(A_{k-3})}{2} \right\rceil + u_{k-2}(t) \\ &= 2 + u_{k-2}(t) \text{ (Proposition 2.14, Item 1)} \\ &\leq 2 + ((k-2)-3) \text{ (induction hypothesis)} \\ &= k-3. \end{aligned}$$

- $t \in [A_{k-2} - A_{k-3} + 2, A_k - A_{k-1} + 1]$: note firstly that $A_k - A_{k-1} = A_{k-2} - (A_{k-1} - A_{k-3}) = A_{k-2} - A_{k-4} \Rightarrow A_k - A_{k-1} + 1 = A_{k-2} - (A_{k-4} - 1) \leq A_{k-2} - 3$ since $k \geq 9$. Secondly, we have $A_{k-2} - t + 1 \in [A_{k-4}, A_{k-3} - 1] \Rightarrow A^{-1}(A_{k-2} - t + 1) = k-4$. Thus we have:

$$\begin{aligned} u_k(t) &= \left\lceil \frac{k+1-A^{-1}(A_{k-2}-t+1)}{2} \right\rceil + u_{k-2}(t) \\ &= \left\lceil \frac{k+1-(k-4)}{2} \right\rceil + u_{k-2}(t) \\ &= 3 + u_{k-2}(t) \\ &\leq 3 + ((k-2)-4) \text{ (induction hypothesis)} \\ &= k-3. \end{aligned}$$

- $t \in [A_k - A_{k-1} + 2 = A_{k-2} - A_{k-4} + 2, A_{k-2} - A_{k-5} + 1]$: note that $A_{k-2} - A_{k-5} + 1 \leq A_{k-2} - 2$ since $k \geq 9$. Secondly, we have $A_{k-2} - t + 1 \in [A_{k-5}, A_{k-4} - 1] \Rightarrow A^{-1}(A_{k-2} - t + 1) = k-5$. Thus we have:

$$\begin{aligned} u_k(t) &= \left\lceil \frac{k+1-(k-5)}{2} \right\rceil + u_{k-2}(t) \\ &= 3 + u_{k-2}(t) \\ &\leq 3 + ((k-2)-5) \text{ (induction hypothesis)} \\ &= k-4. \end{aligned}$$

  This bound is sufficient for us since $t \leq A_{k-2} - A_{k-5} + 1 = A_{k-4} + 1 \leq A_{k-3} + 1 = A_k - A_{k-2} + 1$.

- $t \in [A_{k-2} - A_{k-5} + 2, A_{k-2} - A_{k-7} + 1]$: note that $A_{k-2} - A_{k-7} + 1 \leq A_{k-2} - 1$ since $k \geq 9$. Secondly, we have $A_{k-2} - t + 1 \in [A_{k-7}, A_{k-5} - 1] \Rightarrow A^{-1}(A_{k-2} - t + 1) \in \{k-7, k-6\}$. Thus we have:

$$\begin{aligned} u_k(t) &= \left\lceil \frac{k+1-[(k-7) \text{ or } (k-6)]}{2} \right\rceil + u_{k-2}(t) \\ &= 4 + u_{k-2}(t) \\ &\leq 4 + ((k-2)-6) \text{ (induction hypothesis)} \\ &= k-4. \end{aligned}$$

  This bound is sufficient for us since $t \leq A_{k-2} - A_{k-7} + 1 = A_{k-3} + 1 = A_k - A_{k-2} + 1$.

- $t \in [A_{k-2} - A_{(k-2)-2r+1} + 2, A_{k-2} - A_{(k-2)-2r-1} + 1]$ for some $r \in [3, \lfloor (k-5)/2 \rfloor]$. Note that this case only arises if $\lfloor (k-5)/2 \rfloor \geq 3 \Leftrightarrow k \geq 11$. (For $k = 9, 10$, this case may be safely skipped over.)

  In this case, we have that:

$$\begin{aligned} A_{k-2} - A_{(k-2)-2r-1} + 1 &\leq A_{k-2} - A_{(k-2)-2\lfloor (k-5)/2 \rfloor + 1} + 1 \\ &= A_{k-2} - A_{2 \text{ or } 3} + 1 \\ &= A_{k-2} - 1. \end{aligned}$$

  Secondly, we have $A_{k-2} - t + 1 \in [A_{(k-2)-2r-1}, A_{(k-2)-2r+1} - 1] \Rightarrow$
  $A^{-1}(A_{k-2} - t + 1) \in \{(k-2) - 2r - 1, (k-2) - 2r\}$. Thus we have:

$$\begin{aligned} u_k(t) &= \left\lceil \frac{k + 1 - [(k - 2r - 3) \text{ or } (k - 2r - 2)]}{2} \right\rceil + u_{k-2}(t) \\ &= r + 2 + u_{k-2}(t) \\ &\leq r + 2 + ((k-2) - r - 5) \text{ (induction hypothesis)} \\ &= k - 5. \end{aligned}$$

  This bound suffices since $t \leq A_{k-2} - 1 < A_k - A_{k-3} + 1$.

- $t = A_{k-2}$: in this case, we have $u_k(t) = \lfloor (k-1)/2 \rfloor$. Since $t < A_k - A_{k-3} + 1$, it suffices to check that $\lfloor (k-1)/2 \rfloor \leq k - 5$, which holds for any $k \geq 8$.

- $t \in [A_{k-2} + 1 = A_k - A_{k-3} + 1, A_k - A_{k-5} + 1]$: we have

$$\begin{aligned} u_k(t) &= u_{k-3}(t - A_{k-2}) \\ &= u_{k-3}(t' \in [1, A_{k-3} - A_{k-5} + 1]) \\ &\leq (k-3) - 3 \text{ (induction hypothesis)} \\ &= k - 6, \end{aligned}$$

  which is the desired bound.

- $t \in [A_k - A_{k-5} + 2, A_k - A_{k-7} + 1]$, which amounts to the $r = 3$ case: we have

$$\begin{aligned} u_k(t) &= u_{k-3}(t' \in [A_{k-3} - A_{k-5} + 2, A_{k-3} - A_{k-7} + 1]) \\ &\leq (k-3) - 5 \text{ (induction hypothesis)} \\ &= k - r - 5. \end{aligned}$$

At this point, our proof is complete for $k = 9, 10$ since in this case we have $A_k - A_{k-7} + 1 = A_k - 1$. From here on, assume $k \geq 11$. We have two more cases:

- $t \in [A_k - A_{k-7} + 2, A_k - A_{k-9} + 1]$, which amounts to the $r = 4$ case: we have

$$\begin{aligned} u_k(t) &= u_{k-3}(t' \in [A_{k-3} - A_{k-7} + 2, A_{k-3} - A_{k-9} + 1]) \\ &\leq (k-3) - 6 \\ &= k - r - 5. \end{aligned}$$

At this point, our proof is complete for $k = 11, 12$, since in this case we have $A_k - A_{k-9} + 1 = A_k - 1$. From here on, assume $k \geq 13$. We have one last case:

- $t \in [A_k - A_{k-2r+1} + 2, A_k - A_{k-2r-1} + 1]$ for some $r \in [5, \lfloor (k-3)/2 \rfloor]$. Now let $t' = t - A_{k-2} \in [A_{k-3} - A_{k-2r+1} + 2, A_{k-3} - A_{k-2r-1} + 1] \subseteq [A_{k-3} - A_{(k-3)-2(r-2)+1} + 2, A_{k-3} - A_{(k-3)-2(r-1)-1} + 1]$.

  Thus we may take $r' \in \{r-2, r-1\}$ such that $t' \in [A_{k-3} - A_{(k-3)-2r'+1} + 2, A_{k-3} - A_{(k-3)-2r'-1} + 1]$. Moreover, $r' \geq r - 2 \geq 3$. Next we claim that $r' \leq \lfloor (k-6)/2 \rfloor$ (this will enable us to use the induction hypothesis for r' and $k-3$). We address this in two cases:

  - If $r \leq \lfloor (k-4)/2 \rfloor$, this follows from the fact that $r' \leq r - 1 \leq \lfloor (k-6)/2 \rfloor$.
  - If $\lfloor (k-4)/2 \rfloor < r \leq \lfloor (k-3)/2 \rfloor$, this forces k to be odd and $r = (k-3)/2$. If we could show that we could take $r' = r - 2$, then we would be done since $r - 2 \leq (k-7)/2$. This in turn follows from noting that $t' \leq A_{k-3} - 1 = A_{k-3} - A_{(k-3)-2(r-2)-1} + 1$.

  It hence follows from the inductive hypothesis that:

$$\begin{aligned} u_k(t) &= u_{k-3}(t') \\ &\leq (k-3) - r' - 5 \\ &\leq (k-3) - (r-2) - 5 \\ &= k - r - 6, \end{aligned}$$

  as desired.

This exhausts all possible values of t since $A_k - A_{k-2\lfloor (k-3)/2 \rfloor - 1} + 1 = A_k - A_{2 \text{ or } 3} + 1 = A_k - 1$. Thus our induction and the proof of the lemma are complete. □

B Details of A* Search

In Section 2.4, we describe how we use an optimized A* search, implemented in the Julia programming language, to explicitly find the exact optimal parallel pebbling sequences for certain values of ℓ and s. In this Appendix, we describe details of the design of the A* search.

We first must precisely define the problem we are solving:

Given a particular number of pebbles s and pebble game length ℓ, find a series of parallel pebbling steps that has the absolute minimum possible depth.

We cast this as a graph search problem: consider a directed graph whose vertices are all possible configurations of $\leq s$ pebbles (and $\leq \ell$ ghosts) on ℓ indices, and whose edges are valid parallel pebbling steps (which may consist of several moves done in parallel). The goal is to find the shortest path between the "initial" vertex V_i (corresponding to no pebbles present) and the "final" vertex V_f (corresponding to a single pebble at position ℓ and no other pebbles or ghosts present).

This graph search problem initially seems rather hopeless for all but very small ℓ and s. There are $\sum_{s'=0}^{s} \binom{\ell}{s'}$ possible pebble configurations and 2^ℓ possible ghost configurations, so the total number of vertices is $|V| = 2^\ell \cdot \sum_{s'=0}^{s} \binom{\ell}{s'}$. Furthermore the graph has rather high degree: for a vertex whose configuration has s' pebbles, there are $4^{s'}$ valid parallel pebbling moves (on each pebble, apply a pebble operation or not, and afterward ghost that pebble or not). Fortunately we may dramatically reduce the search space by making the following observations, which can be combined into an admissible heuristic for $A*$ search.

First, as we discuss in Section 2, an optimal pebbling provably always begins with an initial "blast" phase where we pebble directly across to position ℓ, leaving some "marker" pebbles along the way. Therefore we can reduce our search to finding the shortest path from *any* vertex V_m that has a pebble at position ℓ, to V_f, since given a particular V_m the steps from V_i to V_m are entirely specified by this rule.

For the remaining search (corresponding to the "unblast" phase), we can make the following observation. $\mathsf{Ghost}(i)$ and $\mathsf{Unpebble}(i)$ have the same effect of removing a pebble, but $\mathsf{Ghost}(i)$ is essentially free. Also, if there are any pebbles with index higher than i that need to be removed, then position i will be pebbled again at some point in the future, at which time any ghosts can be removed. Therefore, the configuration of ghosts in any optimal pebbling is entirely determined by the pebble configuration: there will be ghosts on all positions extending to the highest-indexed position that has a pebble (excluding position ℓ itself), and no ghosts on positions with indices higher than that. This fully removes the factor of 2^ℓ from the number of vertices we must explore.

Next we observe that $\mathsf{Ghost}(i)$ should only ever be applied immediately after $\mathsf{Pebble}(i+1)$ (i.e. after position i is used as input). The proof is straightforward: given a pebbling scheme in which this is not true, simply move $\mathsf{Ghost}(i)$ back in time to the time slot immediately after $\mathsf{Pebble}(i+1)$. This can never make a pebble game less optimal; if anything it opens opportunity for something more optimal, because it frees up a pebble earlier. This rule reduces the in-degree of vertices to $2^{s'}$ from $4^{s'}$, where as above s' is the number of pebbles in the vertex's configuration.

Putting these observations together, the key is to start our search from V_f and work backwards until we encounter the first vertex V_m in which position ℓ is pebbled. Given that, there is one final addition to the A* heuristic which gives a dramatic further speedup. We observe that a configuration whose rightmost pebble (exluding ℓ) is at position $i_{\max}$ must have distance at least $\ell - i_{\max}$ from any vertex V_m, since the rightmost pebble position can change by at most 1 with each parallel pebble step. Including this fact in the A* heuristic leads to a search that is equivalent to looking for solutions of depth 2ℓ first, then solutions of depth $2\ell+1$, etc. The number of valid sequences grows rapidly with how far one is from a depth 2ℓ solution, so this strategy ends up exploring a small number of promising solutions first, before expanding into more numerous possibilities. Furthermore it leads to a nice balance as s is varied: with increased s, the number of vertices grows quickly, but the optimal depth also becomes closer to 2ℓ. In practice we find that this makes it possible to successfully find optimal solutions for a wide range of s.

Finally, we note that when searching for optimal pebblings for the practical application of Regev's factoring algorithm, we include the space cost of the temporary ancillas needed to hold the value t (see Section 4) in the optimization. This space is only required in $\mathsf{Pebble}(i)$ and $\mathsf{Unpebble}(i)$ for which $i \equiv 1 \pmod{w+1}$ leading to nontrivial behavior in the pebbling games, as these steps can only be performed when there is sufficient space to spare.

C Additional Background on Lattice Reduction

C.1 Heuristic Justification for Assumption 3.1

Let us now reason heuristically about how we can use a lattice basis reduction algorithm to solve SBBP in dimension r. Suppose that it achieves root-Hermite factor δ_0 and outputs a size-reduced[7] basis $\mathbf{b}_1, \ldots, \mathbf{b}_r$ with Gram-Schmidt orthogonalization $\mathbf{b}_1^*, \ldots, \mathbf{b}_r^*$. Then, as derived in [EG25, Section 3.3.1], under the Geometric Series Assumption we may approximate $\|\mathbf{b}_i^*\| \approx \|\mathbf{b}_1^*\| / \delta_0^{2(i-1)}$. Following the blueprint from [Reg25, Claim 5.1],

[7] See [GN08, Section 2.2] for a definition of size-reduced bases.

to solve SBBP we will let $\ell \geq 0$ be minimal such that $\|\mathbf{b}^*_{\ell+1}\| > \delta_0^{2(r-1)} T$ and output $\mathbf{b}_1, \ldots, \mathbf{b}_\ell$. Heuristically, we expect that $\|\mathbf{b}^*_m\| > T$ for all $m \geq \ell + 1$, which is enough for correctness. To bound the norm of each $\mathbf{b}_i$ for $i \in [\ell]$, note that we have:

$$\|\mathbf{b}_i\|^2 \leq \sum_{j=1}^{i} \frac{1}{4}\|\mathbf{b}^*_j\|^2 \leq \frac{r}{4} \cdot \delta_0^{4(r-1)} T^2 \Rightarrow \|\mathbf{b}_i\| \leq \frac{\sqrt{r}}{2} \cdot \delta_0^{2r} \cdot T.$$

(The factor of $1/4$ above comes from the assumption that the basis is size-reduced.)

C.2 Root-Hermite Factors and Costs for BKZ Lattice Reduction

Block size	Root-Hermite factor	Estimated CPU clock cycles for dimension 600
60	1.01145310214785	2^{52}
120	1.00843474281592	2^{64}
160	1.00718344897388	2^{76}
200	1.00628260691082	2^{88}

Table 2: Estimated root-Hermite factors and costs (in CPU clock cycles) for BKZ-β for various block sizes β, obtained using Albrecht's estimator [APS15]. Costs are estimated based on [BDGL16].

C.3 Derivation of Equation (4)

Recall from Equation (3) that we require the following. In the below derivation, we will plug in Assumption 3.1 and approximate $C \approx 1$ (where C is as in Conjecture 3.1).

$$\begin{aligned}
\alpha \sqrt{m+1}\, 2^{Cn/d} &< \frac{\sqrt{2}\, R}{\sqrt{d}} \cdot \frac{1}{6} (4 \cdot 2^n)^{-1/m} \\
&\leq \frac{\sqrt{2}\, D}{2d} \cdot \frac{1}{6} (4 \cdot 2^n)^{-1/m} \\
\Leftrightarrow \quad \sqrt{m+d}\, \delta_0^{2(m+d)} \sqrt{m+1}\, 2^{n/d} &< \frac{\sqrt{2}\, D}{d} \cdot \frac{1}{6} (4 \cdot 2^n)^{-1/m} \\
\Leftrightarrow \quad \log D &> 2(m+d) \log \delta_0 + \frac{n}{d} + \frac{n}{m} \\
&\quad + \tfrac{1}{2} \log(m+d) + \tfrac{1}{2} \log(m+1) + \log \frac{6d}{\sqrt{2}} + \tfrac{2}{m}.
\end{aligned}$$

D Proof of Theorem 4.1

The operation $\mathsf{Pebble}(i')$ (in the notation of Section 2), which sets $y_{i'} \leftarrow y_{i'-1}^2 \bmod N$ for odd i' and $y_{i'} \leftarrow \left(\prod_j a_j^{z_{j,i'/2}}\right) y_{i'-1}$ for even i', can be (in either case) implemented by a single out-of-place quantum multiplication. (The value $t = \left(\prod_j a_j^{z_{j,i'/2}}\right)$ can be computed with negligible cost [Reg25, RV24]). Since $\log t \leq \log \left[\prod_j a_j\right] = o(n)$ this second multiplication will actually be cheaper than a general n-bit multiplication; here we count it the same for simplicity. The unpebble operation $\mathsf{Unpebble}(i')$ is equivalent. Thus the multiplication depth of $\mathsf{Pebble}(i')$ and $\mathsf{Unpebble}(i')$ are both 1, and the ancillary space needed is $S_\times(n) + o(n)$.

By Theorem 2.12, a line graph of length ℓ can be pebbled in depth 2ℓ pebble operations using $2.47 \log \ell$ pebbles. Here $\ell = 2 \log D$, so the depth will be $4 \log D$ n-bit multiplications. In terms of space, the maximum number of pebble operations that can occur in parallel is half of the number of pebbles (since each operation involves two pebbles, the input and the output). Thus since one pebble corresponds to n qubits, we use space at most $1.24(2n + S_\times(n) + o(n)) \log \log D$. □

E Modifying Fibonacci-Based Regev Circuits to Use Parallelism

[Rag24, Theorem 2.4] provides a circuit that uses

$$2 \cdot \left(\frac{3r}{s} + 4 \log s + 7 \right) \cdot K \tag{5}$$

mod N multiplications and at least

$$dK\lceil \log(r+1) \rceil + \max(2 \log s + 5, 7)n \tag{6}$$

qubits.

Straightforward optimizations. A closer look reveals that many of the operations in this circuit can be optimized in parallelism and/or space.

- In [Rag24, Lemma 3.5], we need only use n ancilla qubits (rather than $2n$); for typical multiplication circuits [KY24], there is no need to write down a classical constant in a separate register before multiplying it in.
- In [Rag24, Lemma 3.7], the quantum circuit computing $a, a^{-1}, b, b^{-1} \to a, a^{-1}, ab, (ab)^{-1} \pmod N$ can be implemented using 2 parallel blocks of 2 multiplications each and $\approx 2n$ ancilla qubits, rather than 3 sequential multiplications and n ancilla qubits.

  This is because we can compute $a, b \to a, b, ab$ and $a^{-1}, b^{-1} \to a^{-1}, b^{-1}, a^{-1}b^{-1}$ in parallel before uncomputing b using a^{-1}, ab and uncomputing b^{-1} using $a, a^{-1}b^{-1}$ in parallel.
- [Rag24, Lemma 3.8] can be modified in two places: in the first part, we can use a circuit for squaring modulo N without having to write the number down twice. In the second part, we can plug in the aforementioned parallelized version of [Rag24, Lemma 3.7]. This gives a circuit with a depth of $2r/s + 2 \log s$ mod N multiplications, a total of $4r/s + 4 \log s$ total mod N multiplications, and $(2 \log s + 2)n$ ancilla qubits.

Running steps 3a and 3b in parallel. Finally, we can run steps 3a and 3b of [Rag24, Algorithm 4.1] in parallel. In more detail, we will start by running all but two stages of mod N multiplications in [Rag24, Lemma 3.8] as used in step 3a. We will then run step 3b(i) on its own, then run the last two parallel blocks of mod N multiplications in [Rag24, Lemma 3.8] in parallel with the two multiplications in [Rag24, Lemma 3.5] as used in step 3b(ii). Finally, we will run step 3b(iii) on its own. In all cases, we will have a depth of $2r/s + 2 \log s$ mod N multiplications and a total number of $4r/s + 4 \log s + 2$ mod N multiplications. Accounting for the space is more delicate; for clarity, we step through this in cases below.

Case 1: $s = 1$. In this case, we will initially require $4n + o(n)$ ancilla qubits, out of which $2n$ will be restored. After $2(r-1)$ parallel blocks of multiplications and applying 3b(i), we will have (omitting $o(n)$ qubits throughout):

$$\begin{aligned}
&|a\rangle|a^{-1}\rangle|a^{r-1}b\rangle|(a^{r-1}b)^{-1}\rangle|\prod a_i^{r-z_{i,j}}\rangle|0^{3n}\rangle \\
\to &|a\rangle|a^{-1}\rangle|a^{r-1}b\rangle|(a^{r-1}b)^{-1}\rangle|\prod a_i^{r-z_{i,j}}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{-z_{i,j}}\rangle \\
\to &|a\rangle|a^{-1}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{-z_{i,j}}\rangle|0^{3n}\rangle \\
\to &|a\rangle|a^{-1}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{-z_{i,j}}\rangle|\prod a_i^{z_{i,j}}\rangle|0^{2n}\rangle\,.
\end{aligned}$$

Case 2: $s = 2$. In this case, we will initially require $6n + o(n)$ ancilla qubits, out of which $4n$ will be restored. After r parallel blocks of multiplications and applying 3b(i), we will be halfway through the last application of [Rag24, Lemma 3.7] within step 3a. We hence have:

$$\begin{aligned}
&|a\rangle|a^{-1}\rangle|a^2\rangle|a^{-2}\rangle|a^{r-2}b\rangle|(a^{r-2}b)^{-1}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{r-z_{i,j}}\rangle|0^n\rangle \\
\to &|a\rangle|a^{-1}\rangle|a^2\rangle|a^{-2}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{r-z_{i,j}}\rangle|\prod a_i^{-z_{i,j}}\rangle|0^{2n}\rangle \\
\to &|a\rangle|a^{-1}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{-z_{i,j}}\rangle|0^{5n}\rangle \\
\to &|a\rangle|a^{-1}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{-z_{i,j}}\rangle|\prod a_i^{z_{i,j}}\rangle|0^{4n}\rangle\,.
\end{aligned}$$

Case 3: $s \geq 4$. In this case, we require $(2\log s+2)n+o(n)$ clean ancilla qubits, out of which $(2\log s)n+o(n)$ will be restored. After $2r/s + 2\log s - 2$ parallel blocks of multiplications and applying 3b(i), we will have:

$$\begin{aligned}
&|a\rangle|a^{-1}\rangle|a^2\rangle|a^{-2}\rangle|a^4\rangle|a^{-4}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{r-z_{i,j}}\rangle|0^{(2\log s-3)n}\rangle \\
\to &|a\rangle|a^{-1}\rangle|a^2\rangle|a^{-2}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{r-z_{i,j}}\rangle|\prod a_i^{-z_{i,j}}\rangle|0^{(2\log s-2)n}\rangle \\
\to &|a\rangle|a^{-1}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{-z_{i,j}}\rangle|0^{(2\log s+1)n}\rangle \\
\to &|a\rangle|a^{-1}\rangle|a^r b\rangle|(a^r b)^{-1}\rangle|\prod a_i^{-z_{i,j}}\rangle|\prod a_i^{z_{i,j}}\rangle|0^{(2\log s)n}\rangle\,.
\end{aligned}$$

Putting everything together, assuming $s \in \{1, 2\}$. As in all our results in practice as well as those of [EG25], we typically always have $s = 1, 2$. In this case, running 3a and 3b in parallel initially requires $2\log s + 4$ clean ancilla qubits. We claim that the total space usage is then $dK\lceil\log(r+1)\rceil + (2\log s + 8)n$ qubits:

- At the beginning of step 3, this will leave us with $(2\log s + 4)n$ clean ancilla qubits.
- We can run steps 3a and 3b in parallel as per the above. At this point, we will be left with $(2\log s+2)n$ clean ancilla qubits.
- For step 3c, we only need $2n$ clean ancilla qubits, which we have and will restore.
- For step 3d, we only need n ancilla qubits, which we have and will restore.

F Parallelizing Shor's Algorithm

In this appendix we discuss the parallelization of the modular exponentiation in Shor's algorithm [Sho97]. The structure does not meaningfully change when leveraging e.g. the innovations of [EH17], so for simplicity

this section is written in terms of the construction in Shor's original paper. Note that here we do not consider the use of recent innovations using Residue Number Systems to implement quantum factoring [CFS25, Gid25b], whose depth cannot easily be measured in number of n-bit multiplications.

Consider the operation $|x\rangle|0\rangle \to |x\rangle|a^x \bmod N\rangle$, where a and N are classical constants. This is typically implemented via the decomposition

$$a^x \bmod N = \prod_i c_i^{x_i} \bmod N \tag{7}$$

where $c_i = a^{2^i} \bmod N$. As a quantum circuit, this is implemented by starting with the state $|x\rangle|1\rangle$ and then for each i, performing an in-place modular multiplication of the second register by $c_i^{x_i}$. This in-place multiplication is built out of two out-of-place modular multiply-adds using an ancilla register. The first is a multiplication $|z\rangle|0\rangle \to |z\rangle|zc_i^{x_i} \bmod N\rangle$ which yields the desired result in the ancilla register, and the second uses the modular inverse of c_i (precomputed classically) to uncompute the input: $|z\rangle|zc_i^{x_i} \bmod N\rangle \to |z - (c_i^{-1})^{x_i} \cdot zc_i^{x_i} \bmod N\rangle|zc_i^{x_i} \bmod N\rangle = |0\rangle|zc_i^{x_i} \bmod N\rangle$. A swap may be performed to move this value into the original register, but it is not necessary in practice as the first register can now be redefined as the ancilla register for the next iteration.

A particularly nice feature of this construction is that a single qubit can be recycled for the $|x\rangle$ register (see e.g. [Bea03]). The total qubit count of the factoring algorithm is thus roughly $2n$ (plus the workspace required to do an out-of-place multiplication): one qubit for $|x\rangle$, one register for the output, and a second ancilla register.

One may parallelize this process by breaking the bits of x into p chunks $x[i]$ such that $\sum_i x[i] = x$ and then separately computing $a^{x[i]} \bmod N$ in parallel. This yields the state

$$|x\rangle|a^{x[0]} \bmod N\rangle|a^{x[1]} \bmod N\rangle \cdots |a^{x[p-1]} \bmod N\rangle \tag{8}$$

We may then compute $|a^x \bmod N\rangle$ by multiplying these values together, an operation which can be cast as a spooky pebble game on a line graph of length $p-1$ (different geometries for the graph are possible too, since multiplication is associative).

Unfortunately we now run into a bit of a problem: there is no cheap way to uncompute the partial products we produced. Indeed, it seems that if one follows this plan it is necessary to do the entire partial modular exponentiation that produced these registers in reverse to uncompute them, which introduces an extra factor of two to the number of multiplications required. This then increases the space usage, as the need to do the computation in reverse means that qubit recycling can no longer be applied to the $|x\rangle$ register, so an entire register of n qubits is now needed for that.

The good news is that these overheads do not grow with p. If M is the multiplication depth of the original sequential construction, the depth will be $2\lceil M/p \rceil$ multiplications plus the depth of the length $p-1$ spooky pebble game at the end, and the qubit cost will be roughly $(1+2p)n$ qubits plus those needed for the final spooky pebble game (here ignoring ancillas required for the out-of-place multiplications).

Setting $p = 2$ yields no advantage in depth, because the improvement is canceled out by the extra cost of uncomputation. For our estimates in this work, we examine $p = 3$ and $p = 5$. For $p = 3$ the final spooky pebble game is trivial, it uses 2 registers and has depth 3. For $p = 5$ the spooky pebble game is not quite trivial, one can get away with only using 3 extra registers for a multiplication depth of 8. Thus the overall costs are:

- $p = 3$: depth $2\lceil M/3 \rceil + 3$, qubits $\approx 9n$
- $p = 5$: depth $2\lceil M/5 \rceil + 8$, qubits $\approx 14n$

G Detailed Results for Factoring 2048- and 4096-Bit Integers

G.1 Shor's Algorithm

n	Parallelism	Windowing	N. runs	Mod N multiplications			Qubits/n
				Depth	Per run	Overall	
2048	None	No	1	6018	6018	6018	≈ 2
			20	2290	2290	45800	
		$w = 10$	1	602	602	602	
			20	230	230	4600	
	$p = 3$	No	1	4015	12039	12039	≈ 9
			20	1531	4583	91660	
		$w = 10$	1	405	1207	1207	
			20	157	463	9260	
	$p = 5$	No	1	2416	12044	12044	≈ 14
			20	924	4588	91760	
		$w = 10$	1	250	1212	1212	
			20	100	468	9360	
4096	None	No	1	12162	12162	12162	≈ 2
			27	4438	4438	119826	
		$w = 10$	1	1218	1218	1218	
			27	444	444	11988	
	$p = 3$	No	1	8111	24327	24327	≈ 9
			27	2963	8879	239733	
		$w = 10$	1	815	2439	2439	
			27	299	891	24057	
	$p = 5$	No	1	4874	24332	24332	≈ 14
			27	1784	8884	239868	
		$w = 10$	1	496	2444	2444	
			27	186	896	24192	

Table 3: Costs for Shor's algorithm [Sho97] for $n = 2048, 4096$ with optimizations by [EH17, Eke20, Eke23] as evaluated and reported by [EG25], plus parallelism as described in Appendix F. We differentiate along three axes: whether parallelism is used, whether windowing is used [EG25, Table 2] or not [EG25, Table 1], and whether we use one or many runs of the quantum circuit. As noted by [EG25], it is possible that Shor would benefit from even more aggressive windowing than $w = 10$, but here we adhere to the parameter settings used by their work.

G.2 Regev's Factoring Algorithm with Fibonacci Arithmetic

First, we state explicit cost estimates from essentially using [Rag24, Algorithm 4.1] as in [EG25], but with some key modifications to benefit from parallelism which are discussed in Appendix E.

Let $s \leq r$ be positive integers such that s is a power of 2 and s divides r. These are parameters that the algorithm designer is free to choose. Then define the sequence $\{G_k\}$ by $G_0 = 0, G_1 = 1$, and

$G_k = rG_{k-1} + G_{k-2}$ for $k \geq 2$. Let K be maximal such that $G_K \leq D$. We show that the necessary computation can then done with a depth of

$$2 \cdot \left(\frac{2r}{s} + 2 \log s + 4 \right) \cdot K \tag{9}$$

mod N multiplications, a total of

$$2 \cdot \left(\frac{4r}{s} + 4 \log s + 8 \right) \cdot K \tag{10}$$

mod N multiplications, and — assuming[8] $s \in \{1, 2\}$ — a qubit count of

$$\geq dK\lceil \log(r+1) \rceil + (2 \log s + 8)n + o(n), \tag{11}$$

which is a fairly aggressive lower bound because we are including the cost of storing a product tree computing $\prod_{j=1}^{d} a_j^{z_{j,i}}$ in the $o(n)$ term.

With the above modifications in mind, our experimental methodology for evaluating Fibonacci-based approaches to Regev's circuit is straightforward. For any n, β, r, s (such that s is a power of 2 dividing r), we select d, m by iterating over all possible values subject to constraints analogous to those stated in "Setting parameters in our algorithms" (the constraint on d will instead be $\prod_{i=1}^{d} p_i^r \leq 2^n$), and minimizing the required value of $\log D$ as dictated by Equation (4). We then set K to be maximal such that $G_K \leq D$ and calculate and report costs according to Equations (9) and (10).

These comparison points are summarized in Table 4. For the rows corresponding to $(r, s) = (1, 1)$, we report costs exactly as specified above. For the other rows, we search over values of r, s and report the results achieving minimal multiplication depth. In all our settings, it happens that these values are $(r, s) = (4, 2)$.

We observe that — perhaps surprisingly — the concrete gain from increasing BKZ block size is somewhat marginal. This will be apparent in our later spooky pebbling results as well.

[8] We also treat the case of $s \geq 4$ in Appendix E, but we only need these cases for our results.

n	BKZ-β	r, s	d	m	$\log D$	K	Mod N multiplications			Qubits/n
							Depth	Per run	Overall	
2048	60	1, 1	165	227	52	75	900	1800	408600	≥ 14.0
		4, 2	74	210	63	31	620	1240	260400	≥ 13.3
	120	1, 1	177	260	48	69	828	1656	430560	≥ 13.9
		4, 2	75	228	60	29	580	1160	264480	≥ 13.1
	160	1, 1	189	277	46	66	792	1584	438768	≥ 14.0
		4, 2	74	253	59	29	580	1160	293480	≥ 13.1
	200	1, 1	222	303	44	64	768	1536	465408	≥ 14.9
		4, 2	74	278	58	28	560	1120	311360	≥ 13.0
4096	60	1, 1	290	333	66	95	1140	2280	759240	≥ 14.7
		4, 2	129	315	77	37	740	1480	466200	≥ 13.4
	120	1, 1	317	381	60	87	1044	2088	795528	≥ 14.7
		4, 2	128	363	73	35	700	1400	508200	≥ 13.2
	160	1, 1	358	411	57	82	984	1968	808848	≥ 15.1
		4, 2	129	367	71	34	680	1360	499120	≥ 13.2
	200	1, 1	354	434	55	79	948	1896	822864	≥ 14.8
		4, 2	131	427	69	34	680	1360	580720	≥ 13.2

Table 4: Costs for Regev's algorithm for $n = 2048, 4096$ based on the Fibonacci-style arithmetic in [RV24, Rag24]. We differentiate based on BKZ block size β (which corresponds to classical postprocessing power) and whether we set $(r, s) = (1, 1)$ or allow $r > 1$. Our numbers deviate from those of [EG25] for reasons discussed in Appendix G.2. We additionally report **lower bounds** on the qubit usage, ignoring — in addition to qubit costs described at the beginning of Section 4.2 — that we consistently neglect the qubits required to hold the values $\prod_{j=1}^{d} a_j^{z_{j,i}}$. We do not ignore these costs in Tables 5 and 6.

BKZ-β	d	m	$\log D$	s	Mod N multiplications			Qubits/n
					Depth	Per run	Overall	
60	90	203	59	5	465	500	101500	7.6
				7	263	370	75110	9.6
				12	175	295	59885	14.6
120	90	242	56	5	418	451	109142	7.5
				7	240	341	82522	9.5
				12	163	266	64372	14.5
160	94	242	54	5	396	429	103818	7.5
				7	229	326	78892	9.5
				12	157	253	61226	14.5
200	94	255	53	5	396	429	109395	7.4
				7	229	326	83130	9.4
				12	157	253	64515	14.4

Table 5: Costs for Regev's algorithm with parallel spooky pebbling for $n = 2048$. For four different BKZ block sizes β (which corresponds to classical postprocessing power), we present two sets of results corresponding to a tradeoff between qubit count and depth. For all presented results, the window size $w = 2$. Qubit counts here ignore the minor overheads described at the beginning of Section 4.2. However, unlike Table 4, this table does account for the qubits needed to hold $\prod_{j=1}^{d} a_j^{z_{j,i}}$.

BKZ-β	d	m	$\log D$	s	Mod N multiplications			Qubits/n
					Depth	Per run	Overall	
60	162	314	72	5	610	652	204728	7.8
				7	334	475	149150	9.8
				8	287	444	139416	10.8
120	166	371	67	5	561	600	222600	7.7
				7	310	446	165466	9.7
				12	200	381	141351	14.7
160	166	388	65	5	537	575	223100	7.6
				7	298	432	167616	9.6
				12	193	345	133860	14.6
200	162	401	64	5	513	550	220550	7.5
				7	286	419	168019	9.5
				12	187	326	130726	14.5

Table 6: Costs for Regev's algorithm with parallel spooky pebbling for $n = 4096$. For four different BKZ block sizes β (which corresponds to classical postprocessing power), we present two sets of results corresponding to a tradeoff between qubit count and depth. For all presented results, the window size $w = 2$. Qubit counts here ignore the minor overheads described at the beginning of Section 4.2. However, unlike Table 4, this table does account for the qubits needed to hold $\prod_{j=1}^{d} a_j^{z_{j,i}}$.