The Jacobi Factoring Circuit

A Case Study on Factoring in Sublinear Quantum Space

Gregory D. Meyer*, Seyoon Ragavan*, Vinod Vaikuntanathan*, Katherine Van Kirk†
*MIT, †Harvard




Roadmap

- Part I: a birds-eye view of space cost in quantum period finding

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
- Things I won’t say much about:

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
- Things I won’t say much about:
  - Most works on improving Shor’s algorithm before ~ 2024

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
- Things I won't say much about:
  - Most works on improving Shor's algorithm before ~ 2024
  - Regev's factoring algorithm and subsequent variants

Roadmap

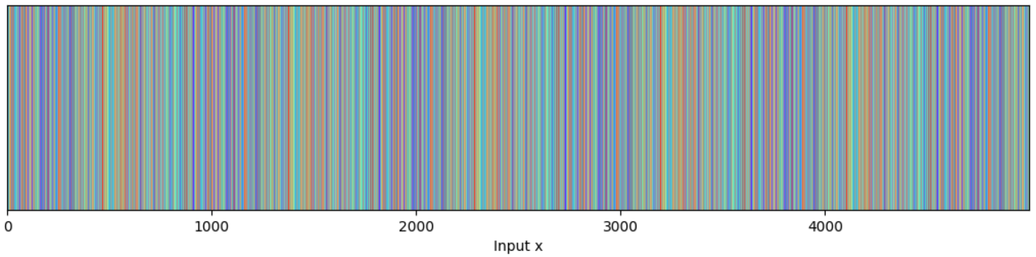
- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
- Things I won't say much about:
  - Most works on improving Shor's algorithm before ~ 2024
  - Regev's factoring algorithm and subsequent variants
  - Cost metrics other than qubit count (gates, depth, etc.) and techniques for improving these

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
  - Trick 1: shortening the period
  - Trick 2: compressing the output and ancilla qubits

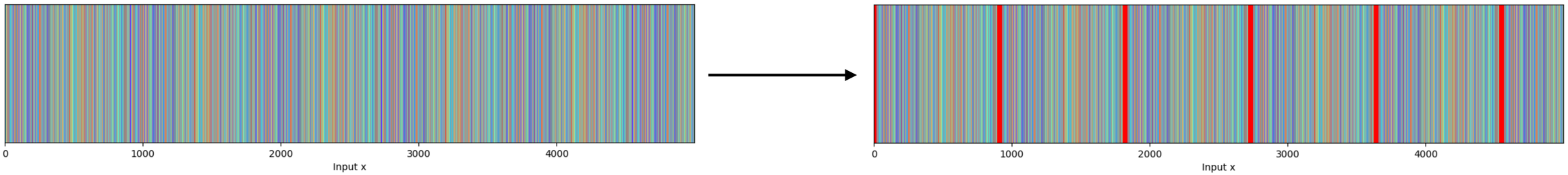
The Star of the Show: Quantum Period Finding

- Strictly periodic function $f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$ with unknown period T
  - $x \equiv y \pmod{T} \Leftrightarrow f(x) = f(y)$



The Star of the Show: Quantum Period Finding

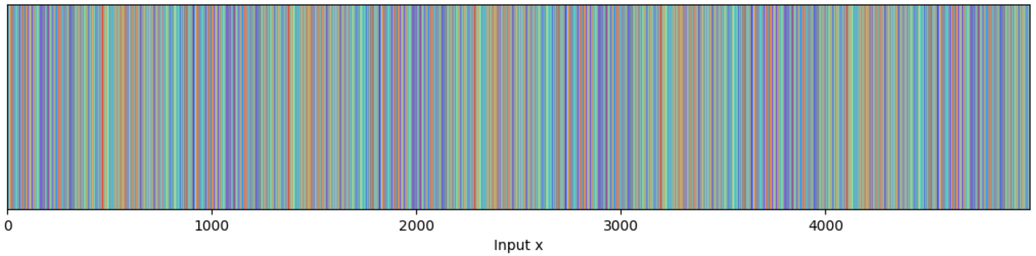
- Strictly periodic function $f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$ with unknown period T
  - $x \equiv y \pmod{T} \Leftrightarrow f(x) = f(y)$
- Informal theorem statement: can quantumly recover T using essentially only the gates/space needed to compute $f(x)$ for $|x| \leq T^2$



Shor's Period-Finding Algorithm

- The algorithm:

1. Prepare the superposition $\sum_{x=1}^{T^2} |x\rangle |f(x)\rangle$

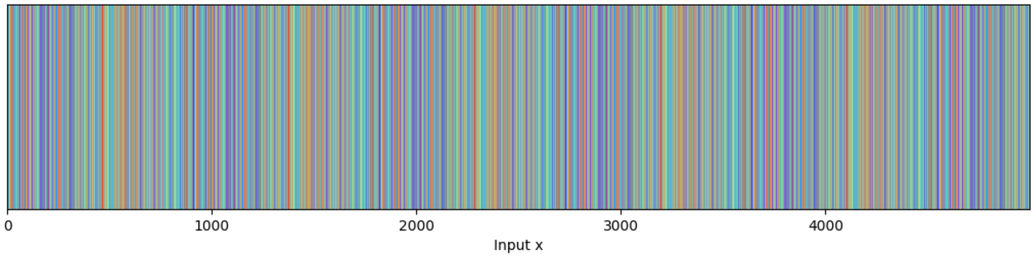


Shor's Period-Finding Algorithm

- The algorithm:

1. Prepare the superposition $\sum_{x=1}^{T^2} |x\rangle |f(x)\rangle$

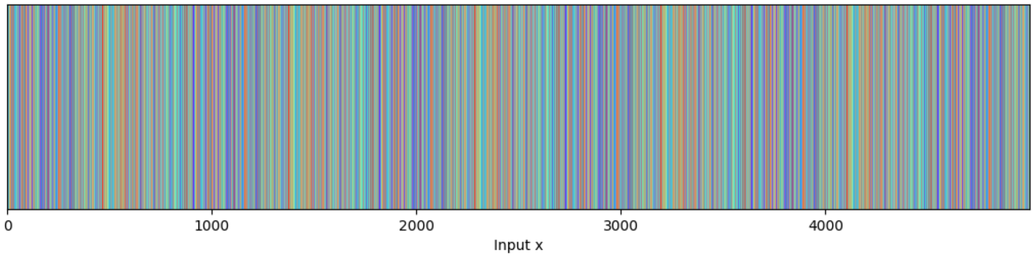
   - Signal has period $T \rightarrow$ frequency $1/T$



Shor's Period-Finding Algorithm

- The algorithm:

1. Prepare the superposition $\sum_{x=1}^{T^2} |x\rangle |f(x)\rangle$
   - Signal has period $T \rightarrow$ frequency $1/T$
2. Apply a quantum Fourier transform to the first register and measure $\rightarrow$ recover a sample $\frac{a}{T}$, where a is **uniformly** sampled from $\{0,1,\ldots,T-1\}$

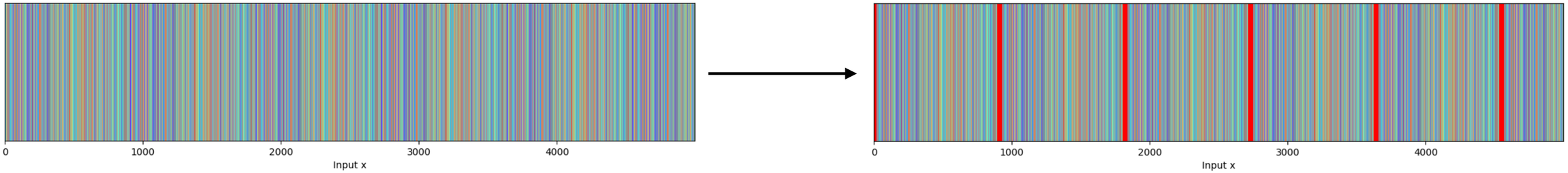


Shor's Period-Finding Algorithm

- The algorithm:

1. Prepare the superposition $\sum_{x=1}^{T^2} |x\rangle |f(x)\rangle$
   - Signal has period $T \rightarrow$ frequency $1/T$
2. Apply a quantum Fourier transform to the first register and measure $\rightarrow$ recover a sample $\frac{a}{T}$, where a is **uniformly** sampled from $\{0,1,\ldots,T-1\}$

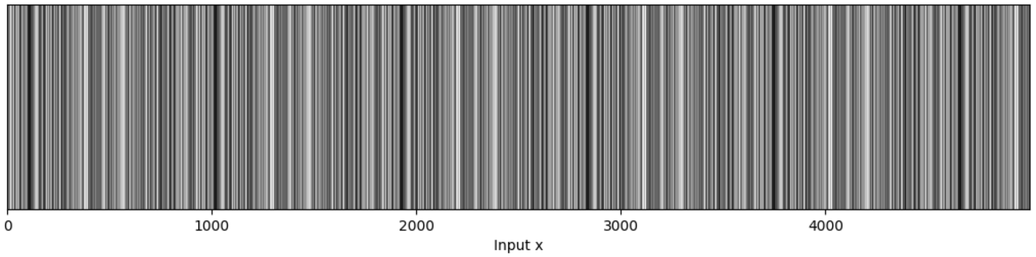
- With decent probability: $\gcd(a, T) = 1 \rightarrow$ learn T



General Quantum Period Finding

Hales-Hallgren ’98, May-Schlieper ’22

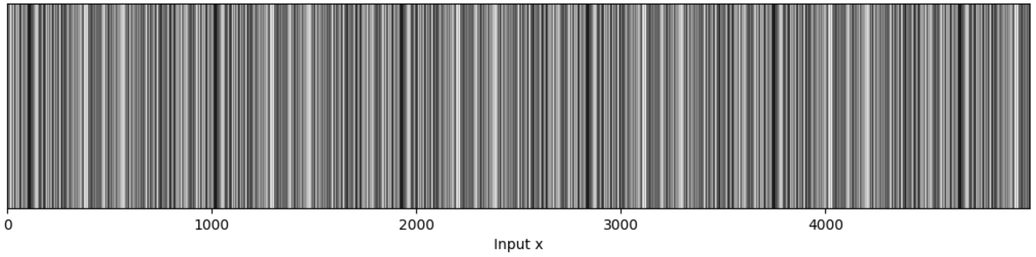
- ~~Strictly~~ periodic function $f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$ with unknown period T
  - $x \equiv y \pmod{T} \Rightarrow f(x) = f(y)$



General Quantum Period Finding

Hales-Hallgren '98, May-Schlieper '22

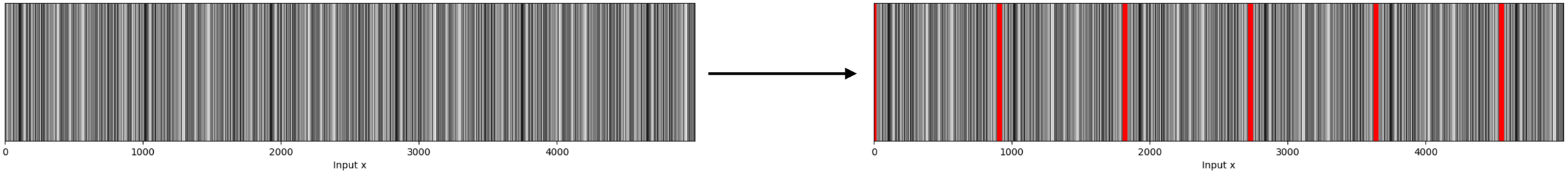
- ~~Strictly~~ periodic function $f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$ with unknown period T
  - $x \equiv y \pmod{T} \Rightarrow f(x) = f(y)$
- Linearity of the Fourier transform → the same algorithm still outputs a **(not necessarily uniform) random** multiple of $1/T$



General Quantum Period Finding

Hales-Hallgren '98, May-Schlieper '22

- ~~Strictly~~ periodic function $f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$ with unknown period T
  - $x \equiv y \pmod{T} \Rightarrow f(x) = f(y)$
- Linearity of the Fourier transform → the same algorithm still outputs a **(not necessarily uniform) random** multiple of $1/T$
- Informal theorem statement: for "reasonable" f, this is still sufficient to recover T



Aside: What is a Reasonable f?

Hales-Hallgren '98, May-Schlieper '22

- Obvious failure mode: constant f
  - More generally: when f has a smaller period $T/d \to \hat{f}$ concentrates on values a/T where a is a multiple of d

Aside: What is a Reasonable f?

Hales-Hallgren '98, May-Schlieper '22

- Obvious failure mode: constant f
  - More generally: when f has a smaller period $T/d \to \hat{f}$ concentrates on values a/T where a is a multiple of d
- As long as this is not the case, we can succeed after taking enough samples!

Aside: What is a Reasonable f?

Hales-Hallgren '98, May-Schlieper '22

- Obvious failure mode: constant f
  - More generally: when f has a smaller period $T/d \to \hat{f}$ concentrates on values a/T where a is a multiple of d
- As long as this is not the case, we can succeed after taking enough samples!
  - Example: random periodic f (by Chernoff bound)

Qubit Cost of Period Finding

- Input qubits: the length of the initial superposition

Cheatsheet

$$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$$

$$f(x+T) = f(x) \forall x$$

Qubit Cost of Period Finding

- Input qubits: the length of the initial superposition
  - Length $T^2 \Rightarrow \sim 2 \log T$ qubits

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \forall x$

Qubit Cost of Period Finding

- Input qubits: the length of the initial superposition
  - Length $T^2 \Rightarrow \sim 2 \log T$ qubits
- Output qubits: exactly ℓ_{out}

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x+T) = f(x) \forall x$

Qubit Cost of Period Finding

- Input qubits: the length of the initial superposition
  - Length $T^2 \Rightarrow \sim 2 \log T$ qubits
- Output qubits: exactly ℓ_{out}
- Ancilla qubits: any extra workspace we need to compute $f(x)$ for $x \leq T^2$

Cheatsheet

$$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$$

$$f(x+T) = f(x) \forall x$$

Qubit Count of Shor's Original Algorithm

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \forall x$

Qubit Count of Shor's Original Algorithm

- Recall: Shor for factoring an n-bit number N runs period finding with the function $f(x) = a^x \bmod N$

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \forall x$

Qubit Count of Shor's Original Algorithm

- Recall: Shor for factoring an n-bit number N runs period finding with the function $f(x) = a^x \bmod N$
  - Period $T \approx N$

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \forall x$

Qubit Count of Shor's Original Algorithm

- Recall: Shor for factoring an n-bit number N runs period finding with the function $f(x) = a^x \bmod N$
  - Period $T \approx N$
  - Output is an element of $\mathbb{Z}_N$, so $\ell_{\text{out}} = n$

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \forall x$

Qubit Count of Shor's Original Algorithm

- Recall: Shor for factoring an n-bit number N runs period finding with the function $f(x) = a^x \bmod N$
  - Period $T \approx N$
  - Output is an element of $\mathbb{Z}_N$, so $\ell_{\text{out}} = n$
- Costs:
  - Input qubits: $\sim 2 \log T \sim 2n$

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \forall x$

Qubit Count of Shor's Original Algorithm

- Recall: Shor for factoring an n-bit number N runs period finding with the function $f(x) = a^x \bmod N$
  - Period $T \approx N$
  - Output is an element of $\mathbb{Z}_N$, so $\ell_{\text{out}} = n$
- Costs:
  - Input qubits: $\sim 2 \log T \sim 2n$
  - Output qubits: $\ell_{\text{out}} = n$

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \forall x$

Qubit Count of Shor's Original Algorithm

- Recall: Shor for factoring an n-bit number N runs period finding with the function $f(x) = a^x \bmod N$
  - Period $T \approx N$
  - Output is an element of $\mathbb{Z}_N$, so $\ell_{\text{out}} = n$
- Costs:
  - Input qubits: $\sim 2 \log T \sim 2n$
  - Output qubits: $\ell_{\text{out}} = n$
  - Ancilla qubits: turns out to be n

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \forall x$

Qubit Count of Shor's Original Algorithm

- Recall: Shor for factoring an n-bit number N runs period finding with the function $f(x) = a^x \bmod N$
  - Period $T \approx N$
  - Output is an element of $\mathbb{Z}_N$, so $\ell_{\text{out}} = n$
- Costs:
  - Input qubits: $\sim 2 \log T \sim 2n$
  - Output qubits: $\ell_{\text{out}} = n$
  - Ancilla qubits: turns out to be n
- Total: $4n$

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \forall x$

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
  - Trick 1: shortening the period
  - Trick 2: compressing the output and ancilla qubits

Saving Input Qubits with Qubit Recycling/Semiclassical QFT

Griffiths-Niu ’95

- Definition: f is one-pass streamable if we can reversibly compute $f(x)$ by streaming through x one bit at a time

Saving Input Qubits with Qubit Recycling/Semiclassical QFT

Griffiths-Niu ’95

- Definition: f is one-pass streamable if we can reversibly compute $f(x)$ by streaming through x one bit at a time
  - No additional junk state allowed!

Saving Input Qubits with Qubit Recycling/Semiclassical QFT

Griffiths-Niu ’95

- Definition: f is one-pass streamable if we can reversibly compute $f(x)$ by streaming through x one bit at a time
  - No additional junk state allowed!
  - Can only look through x once

Saving Input Qubits with Qubit Recycling/Semiclassical QFT

Griffiths-Niu ’95

- Definition: f is one-pass streamable if we can reversibly compute $f(x)$ by streaming through x one bit at a time
  - No additional junk state allowed!
  - Can only look through x once

Saving Input Qubits with Qubit Recycling/Semiclassical QFT

Griffiths-Niu ’95

- Definition: f is one-pass streamable if we can reversibly compute $f(x)$ by streaming through x one bit at a time
  - No additional junk state allowed!
  - Can only look through x once
- Informal theorem statement: if f is one-pass streamable, we don’t need to pay any qubits at all to store the input!

Saving Input Qubits with Qubit Recycling/Semiclassical QFT

Griffiths-Niu '95

- Definition: f is one-pass streamable if we can reversibly compute $f(x)$ by streaming through x one bit at a time
  - No additional junk state allowed!
  - Can only look through x once
- Informal theorem statement: if f is one-pass streamable, we don't need to pay any qubits at all to store the input!
- Idea: push the measurement gates at the end of Shor's algorithm earlier in the circuit

Qubit Recycling for Shor's Algorithm

- $f(x) = a^x \bmod N$ is actually one-pass streamable!

Qubit Recycling for Shor's Algorithm

- $f(x) = a^x \bmod N$ is actually one-pass streamable!
- Idea: $x = \sum_i b_i 2^i \Rightarrow f(x) = \prod_i \left(a^{2^i}\right)^{b_i} \bmod N$, so we just have to do a multiplication based on each bit of x

Qubit Recycling for Shor's Algorithm

- $f(x) = a^x \bmod N$ is actually one-pass streamable!
- Idea: $x = \sum_i b_i 2^i \Rightarrow f(x) = \prod_i \left(a^{2^i}\right)^{b_i} \bmod N$, so we just have to do a multiplication based on each bit of x
- So with qubit recycling, we save $2 \log T \sim 2n$ qubits, and our qubit cost is down to $\sim 2n$

Qubit Recycling for Shor's Algorithm

- $f(x) = a^x \bmod N$ is actually one-pass streamable!
- Idea: $x = \sum_i b_i 2^i \Rightarrow f(x) = \prod_i \left(a^{2^i}\right)^{b_i} \bmod N$, so we just have to do a multiplication based on each bit of x
- So with qubit recycling, we save $2 \log T \sim 2n$ qubits, and our qubit cost is down to $\sim 2n$
- State of the art (ignoring an additive $O(1)$) from the earlier 2000's to 2024

Saving Input Qubits When You Can't Recycle

Seifert '01, Ekerå-Håstad '17

- Before: superposition extends to T^2, and can solve period finding (with decent probability) with one run of the quantum circuit

Saving Input Qubits When You Can't Recycle

Seifert '01, Ekerå-Håstad '17

- Before: superposition extends to T^2, and can solve period finding (with decent probability) with one run of the quantum circuit
- Different tradeoff: superposition extends to just $T^{1+\epsilon}$, and can solve period finding with $1 + 1/\epsilon$ runs of the quantum circuit

Saving Input Qubits When You Can't Recycle

Seifert '01, Ekerå-Håstad '17

- Before: superposition extends to T^2, and can solve period finding (with decent probability) with one run of the quantum circuit
- Different tradeoff: superposition extends to just $T^{1+\epsilon}$, and can solve period finding with $1 + 1/\epsilon$ runs of the quantum circuit
  - Outputs need to be postprocessed using a lattice algorithm in $1 + 1/\epsilon$ dimensions

Saving Input Qubits When You Can't Recycle

Seifert '01, Ekerå-Håstad '17

- Before: superposition extends to T^2, and can solve period finding (with decent probability) with one run of the quantum circuit
- Different tradeoff: superposition extends to just $T^{1+\epsilon}$, and can solve period finding with $1 + 1/\epsilon$ runs of the quantum circuit
  - Outputs need to be postprocessed using a lattice algorithm in $1 + 1/\epsilon$ dimensions
- Input qubits: reduced from $2 \log T$ to $(1 + \epsilon)\log T$, so roughly halved

Qubit Cost of Period Finding, Refined

- Input qubits:
  - If f is one-pass streamable: 1 qubit

Cheatsheet

$$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$$

$$f(x+T) = f(x) \, \forall x$$

Qubit Cost of Period Finding, Refined

- Input qubits:
  - If f is one-pass streamable: 1 qubit
  - Otherwise:
    - Naive: $\sim 2 \log T$ qubits

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x + T) = f(x) \, \forall x$

Qubit Cost of Period Finding, Refined

- Input qubits:
  - If f is one-pass streamable: 1 qubit
  - Otherwise:
    - Naive: $\sim 2 \log T$ qubits
    - Using multiple runs and lattice postprocessing: $\sim (1+\epsilon)\log T$ qubits

Cheatsheet

$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$

$f(x+T) = f(x) \, \forall x$

Qubit Cost of Period Finding, Refined

- Input qubits:
  - If f is one-pass streamable: 1 qubit
  - Otherwise:
    - Naive: $\sim 2 \log T$ qubits
    - Using multiple runs and lattice postprocessing: $\sim (1+\epsilon) \log T$ qubits
- Output qubits: exactly ℓ_{out}
- Ancilla qubits: any extra workspace we need to compute $f(x)$ for $x \leq T^2$ or $T^{1+\epsilon}$

Cheatsheet

$$f : \mathbb{Z} \to \{0,1\}^{\ell_{\text{out}}}$$

$$f(x+T) = f(x) \, \forall x$$

Saving Output and Ancilla Qubits

May-Schlieper '22, Chevignard-Fouque-Schrottenloher '25

- Shor's function $a^x \bmod N$ has the attractive feature that it is strictly periodic

Saving Output and Ancilla Qubits

May-Schlieper '22, Chevignard-Fouque-Schrottenloher '25

- Shor's function $a^x \bmod N$ has the attractive feature that it is strictly periodic
- Saving output qubits: forget strict periodicity and compress the output
  - Example: $(a^x \bmod N) \bmod 2^{20}$

Saving Output and Ancilla Qubits

May-Schlieper '22, Chevignard-Fouque-Schrottenloher '25

- Shor's function $a^x \bmod N$ has the attractive feature that it is strictly periodic
- Saving output qubits: forget strict periodicity and compress the output
  - Example: $(a^x \bmod N) \bmod 2^{20}$
- Saving ancilla qubits: need to be able to compute the compressed output in a cleverer way than just computing the original output

Saving Output and Ancilla Qubits

May-Schlieper '22, Chevignard-Fouque-Schrottenloher '25

- Shor's function $a^x \bmod N$ has the attractive feature that it is strictly periodic
- Saving output qubits: forget strict periodicity and compress the output
  - Example: $(a^x \bmod N) \bmod 2^{20}$
- Saving ancilla qubits: need to be able to compute the compressed output in a cleverer way than just computing the original output
  - Breakthrough idea by Chevignard-Fouque-Schrottenloher '25: this can actually be done for the above example using residue number systems!

Aside: Challenges with Inverting Mod p

Proos-Zalka '03, Khattar et al. '25, Chevignard et al. '26, Schrottenloher '26

- Bottleneck in Shor's algorithm applied to elliptic-curve discrete log: actually computing the elliptic curve group operation

Aside: Challenges with Inverting Mod p

Proos-Zalka '03, Khattar et al. '25, Chevignard et al. '26, Schrottenloher '26

- Bottleneck in Shor's algorithm applied to elliptic-curve discrete log: actually computing the elliptic curve group operation
  - Over $\mathbb{F}_p$: requires one to compute $x \mapsto x^{-1} \bmod p$ in superposition at least once

Aside: Challenges with Inverting Mod p

Proos-Zalka '03, Khattar et al. '25, Chevignard et al. '26, Schrottenloher '26

- Bottleneck in Shor's algorithm applied to elliptic-curve discrete log: actually computing the elliptic curve group operation
  - Over $\mathbb{F}_p$: requires one to compute $x \mapsto x^{-1} \bmod p$ in superposition at least once
  - Standard approach for this is the extended Euclidean algorithm, which used to be a nightmare to implement quantumly (Proos-Zalka '03)

Aside: Challenges with Inverting Mod p

Proos-Zalka '03, Khattar et al. '25, Chevignard et al. '26, Schrottenloher '26

- Bottleneck in Shor's algorithm applied to elliptic-curve discrete log: actually computing the elliptic curve group operation
  - Over $\mathbb{F}_p$: requires one to compute $x \mapsto x^{-1} \bmod p$ in superposition at least once
  - Standard approach for this is the extended Euclidean algorithm, which used to be a nightmare to implement quantumly (Proos-Zalka '03)
- Two recent solutions:
  - Chevignard-Fouque-Schrottenloher '26: sidestep the issue of inversion entirely with a Legendre symbol trick

Aside: Challenges with Inverting Mod p

Proos-Zalka '03, Khattar et al. '25, Chevignard et al. '26, Schrottenloher '26

- Bottleneck in Shor's algorithm applied to elliptic-curve discrete log: actually computing the elliptic curve group operation
  - Over $\mathbb{F}_p$: requires one to compute $x \mapsto x^{-1} \bmod p$ in superposition at least once
  - Standard approach for this is the extended Euclidean algorithm, which used to be a nightmare to implement quantumly (Proos-Zalka '03)
- Two recent solutions:
  - Chevignard-Fouque-Schrottenloher '26: sidestep the issue of inversion entirely with a Legendre symbol trick
  - Khattar et al. '25, Schrottenloher '26: bite the bullet and run extended Euclidean, but in a space-efficient way ("dialog technique")

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
  - Trick 1: shortening the period
  - Trick 2: compressing the output and ancilla qubits

Recent Successes for Factoring and ECDLP

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						

Recent Successes for Factoring and ECDLP

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			

Recent Successes for Factoring and ECDLP

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			
Babbush et al. ’26	EC-256: 1200 qubits	Not public					

Recent Successes for Factoring and ECDLP

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			
Babbush et al. ’26	EC-256: 1200 qubits	Not public					
Schrottenloher ’26	EC-256: 1192 qubits						

Recent Successes for Factoring and ECDLP

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			
Babbush et al. ’26	EC-256: 1200 qubits	Not public					
Schrottenloher ’26	EC-256: 1192 qubits						
Jacobi factoring (Meyer-R-Vaikuntanathan-Van Kirk ’25)	*Factoring P^2 Q where Q has 500 bits with ~ 1000 qubits*						

Jacobi Factoring

- Input: a positive integer N with the promise that it is of the form P^2Q for large primes P and Q

Jacobi Factoring

- Input: a positive integer N with the promise that it is of the form P^2Q for large primes P and Q
  - $n = \lfloor \log_2 N \rfloor + 1$ is the bit length of N as usual

Jacobi Factoring

- Input: a positive integer N with the promise that it is of the form P^2Q for large primes P and Q
  - $n = \lfloor \log_2 N \rfloor + 1$ is the bit length of N as usual
  - $m = \lfloor \log_2 Q \rfloor + 1$ is the bit length of Q

Jacobi Factoring

- Input: a positive integer N with the promise that it is of the form P^2Q for large primes P and Q
  - $n = \lfloor \log_2 N \rfloor + 1$ is the bit length of N as usual
  - $m = \lfloor \log_2 Q \rfloor + 1$ is the bit length of Q
- Theorem (our work): there is a quantum circuit for recovering P, Q with $O(m)$ qubits!

Jacobi Factoring

- Input: a positive integer N with the promise that it is of the form P^2Q for large primes P and Q
  - $n = \lfloor \log_2 N \rfloor + 1$ is the bit length of N as usual
  - $m = \lfloor \log_2 Q \rfloor + 1$ is the bit length of Q
- Theorem (our work): there is a quantum circuit for recovering P, Q with $O(m)$ qubits!
  - Additional selling point: if optimising asymptotics, we can get just $\tilde{O}(n)$ gates at the expense of $\tilde{O}(m)$ qubits (Shor and Regev both need a total of $\tilde{O}(n^2)$ gates)

Aside: Setting Parameters

How should we set $m = \log Q$ relative to n?

$$N = P^2 Q$$

Our $O(m)$ qubit count doesn't help

Aside: Setting Parameters

How should we set $m = \log Q$ relative to n?

$$N = P^2 Q$$

Classical algorithms could exploit this structure to run faster than general NFS

- General NFS: $\exp(\tilde{O}(n^{1/3}))$
- Lenstra ECM/Mulder '24: $\exp(\tilde{O}(m^{1/2}))$

Our $O(m)$ qubit count doesn't help

Aside: Setting Parameters

How should we set $m = \log Q$ relative to n?

$$N = P^2 Q$$

Classical algorithms could exploit this structure to run faster than general NFS

- General NFS: $\exp(\tilde{O}(n^{1/3}))$
- Lenstra ECM/Mulder '24: $\exp(\tilde{O}(m^{1/2}))$

Set $m = \tilde{O}(n^{2/3})$

Our $O(m)$ qubit count doesn't help

Jacobi Factoring: Summary

- Input: a positive integer N with the promise that it is of the form P^2Q for large primes P and Q
  - $n = \lfloor \log_2 N \rfloor + 1$ is the bit length of N as usual
  - $m = \lfloor \log_2 Q \rfloor + 1$ is the bit length of Q, and $m = \tilde{O}(n^{2/3})$
- Theorem (our work): there is a quantum circuit for recovering P, Q with $\tilde{O}(n^{2/3})$ qubits and $\tilde{O}(n)$ gates
  - Classically, this problem is still plausibly as hard as factoring an n-bit RSA integer

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
  - Trick 1: shortening the period
  - Trick 2: compressing the output and ancilla qubits

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
  - Trick 1: shortening the period
  - Trick 2: compressing the output and ancilla qubits

Why We Need a Different Function

- Shor's algorithm and all improvements: work with the function $f(x) = a^x \bmod N$ (or a compressed variant) with period $T \approx N$

Why We Need a Different Function

- Shor's algorithm and all improvements: work with the function $f(x) = a^x \bmod N$ (or a compressed variant) with period $T \approx N$
- Currently stuck at $\Omega(n)$ qubits

Why We Need a Different Function

- Shor's algorithm and all improvements: work with the function $f(x) = a^x \bmod N$ (or a compressed variant) with period $T \approx N$
- Currently stuck at $\Omega(n)$ qubits
  - If using qubit recycling: no output compression so need n qubits for output

Why We Need a Different Function

- Shor's algorithm and all improvements: work with the function $f(x) = a^x \bmod N$ (or a compressed variant) with period $T \approx N$
- Currently stuck at $\Omega(n)$ qubits
  - If using qubit recycling: no output compression so need n qubits for output
  - Otherwise: typically need $\log T \sim n$ qubits for input

Why We Need a Different Function

- Shor's algorithm and all improvements: work with the function $f(x) = a^x \bmod N$ (or a compressed variant) with period $T \approx N$
- Currently stuck at $\Omega(n)$ qubits
  - If using qubit recycling: no output compression so need n qubits for output
  - Otherwise: typically need $\log T \sim n$ qubits for input
    - Chevignard-Fouque-Schrottenloher manages to get $\approx n/2$ qubits by exploiting RSA structure ($N = PQ$) to reduce to discrete log (EH17)

Why We Need a Different Function

- Shor's algorithm and all improvements: work with the function $f(x) = a^x \bmod N$ (or a compressed variant) with period $T \approx N$
- Currently stuck at $\Omega(n)$ qubits
  - If using qubit recycling: no output compression so need n qubits for output
  - Otherwise: typically need $\log T \sim n$ qubits for input
    - Chevignard-Fouque-Schrottenloher manages to get $\approx n/2$ qubits by exploiting RSA structure ($N = PQ$) to reduce to discrete log (EH17)

Goal to improve on Shor: find a different function $j(x)$ which:

- Also has periodicity that enables us to factor N;

Why We Need a Different Function

- Shor's algorithm and all improvements: work with the function $f(x) = a^x \bmod N$ (or a compressed variant) with period $T \approx N$
- Currently stuck at $\Omega(n)$ qubits
  - If using qubit recycling: no output compression so need n qubits for output
  - Otherwise: typically need $\log T \sim n$ qubits for input
    - Chevignard-Fouque-Schrottenloher manages to get $\approx n/2$ qubits by exploiting RSA structure ($N = PQ$) to reduce to discrete log (EH17)

Goal to improve on Shor: find a different function $j(x)$ which:

- Also has periodicity that enables us to factor N;
- Has a shorter period; and

Why We Need a Different Function

- Shor's algorithm and all improvements: work with the function $f(x) = a^x \bmod N$ (or a compressed variant) with period $T \approx N$
- Currently stuck at $\Omega(n)$ qubits
  - If using qubit recycling: no output compression so need n qubits for output
  - Otherwise: typically need $\log T \sim n$ qubits for input
    - Chevignard-Fouque-Schrottenloher manages to get $\approx n/2$ qubits by exploiting RSA structure ($N = PQ$) to reduce to discrete log (EH17)

Goal to improve on Shor: find a different function $j(x)$ which:

- Also has periodicity that enables us to factor N;
- Has a shorter period; and
- Is easier to implement than $a^x \bmod N$

Preliminary: The Legendre Symbol

- a is a *quadratic residue* modulo an odd prime p if there exists integer x such that $a \equiv x^2 \pmod{p}$

Preliminary: The Legendre Symbol

- a is a *quadratic residue* modulo an odd prime p if there exists integer x such that $a \equiv x^2 \pmod{p}$
- Legendre symbol essentially indicates whether this is the case:

$$\left(\frac{a}{p}\right) = \begin{cases} 1, \text{ if } a \text{ is a nonzero quadratic residue modulo } p; \text{ and} \\ -1, \text{ if } a \text{ is not a quadratic residue modulo } p; \text{ and} \\ 0, \text{ if } a \text{ divisible by } p\,. \end{cases}$$

Preliminary: The Jacobi Symbol

- Essentially generalises the Legendre symbol to odd composite moduli

Preliminary: The Jacobi Symbol

- Essentially generalises the Legendre symbol to odd composite moduli
- For $N = p_1^{\alpha_1} \ldots p_r^{\alpha_r}$, define:

$$\left(\frac{a}{N}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \ldots \left(\frac{a}{p_r}\right)^{\alpha_r}$$

Preliminary: The Jacobi Symbol

- Essentially generalises the Legendre symbol to odd composite moduli
- For $N = p_1^{\alpha_1} \ldots p_r^{\alpha_r}$, define:

$$\left(\frac{a}{N}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \ldots \left(\frac{a}{p_r}\right)^{\alpha_r}$$

- Note for intuition: the quadratic residue characterisation does **not** carry over from the Legendre symbol

Preliminary: The Jacobi Symbol

- Essentially generalises the Legendre symbol to odd composite moduli
- For $N = p_1^{\alpha_1} \ldots p_r^{\alpha_r}$, define:

$$\left(\frac{a}{N}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \ldots \left(\frac{a}{p_r}\right)^{\alpha_r}$$

- Note for intuition: the quadratic residue characterisation does **not** carry over from the Legendre symbol
  - Could have $\left(\frac{a}{N}\right) = 1$ without a being a quadratic residue modulo N

Preliminary: The Jacobi Symbol

- Essentially generalises the Legendre symbol to odd composite moduli
- For $N = p_1^{\alpha_1} \ldots p_r^{\alpha_r}$, define:

$$\left(\frac{a}{N}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \ldots \left(\frac{a}{p_r}\right)^{\alpha_r}$$

- Useful property: $a \equiv b \pmod{N} \Rightarrow \left(\frac{a}{N}\right) = \left(\frac{b}{N}\right)$

Preliminary: The Jacobi Symbol

- Essentially generalises the Legendre symbol to odd composite moduli
- For $N = p_1^{\alpha_1} \ldots p_r^{\alpha_r}$, define:

$$\left(\frac{a}{N}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \ldots \left(\frac{a}{p_r}\right)^{\alpha_r}$$

- Useful property: $a \equiv b \pmod{N} \Rightarrow \left(\frac{a}{N}\right) = \left(\frac{b}{N}\right)$
- Theorem (from Euclid to Schönhage 1971): can compute $\left(\frac{a}{N}\right)$ efficiently without knowing the factorisation of N — in fact, in time $\tilde{O}(\log N)$

Period Finding Beyond Shor!

Goal to improve on Shor: find a different function $j(x)$ which:

- Also has periodicity that enables us to factor N;
- Has a shorter period; and
- Is easier to implement than $a^x \bmod N$ ✔

The function $j(x)$ that we're looking for is the Jacobi symbol $j(x) = \left(\frac{x}{N}\right)$!

Factoring from Jacobi Symbol Periodicity

- For RSA integers ($N = PQ$): product of two periodic functions with smaller periods but itself only has period N

$$\left(\frac{a}{N}\right) = \left(\frac{a}{P}\right)\left(\frac{a}{Q}\right)$$

Factoring from Jacobi Symbol Periodicity

- For RSA integers ($N = PQ$): product of two periodic functions with smaller periods but itself only has period N

$$\left(\frac{a}{N}\right) = \left(\frac{a}{P}\right)\left(\frac{a}{Q}\right)$$

- What about $N = P^2Q$?

$$\left(\frac{a}{N}\right) = \left(\frac{a}{P}\right)^2\left(\frac{a}{Q}\right) = \left(\frac{a}{Q}\right), \text{ which is periodic}^* \text{ with period } Q!$$

* modulo minor technical caveats; could have $\left(\frac{a}{P}\right) = 0$ for a tiny fraction of inputs a

Period Finding Beyond Shor!

Goal to improve on Shor: find a different function $j(x)$ which:

- Also has periodicity that enables us to factor $N = P^2Q$; ✔
- Has a shorter period; and ✔ (just Q rather than N)
- Is easier to implement than $a^x \bmod N$ ✔

The function $j(x)$ that we're looking for is the Jacobi symbol $j(x) = \left(\frac{x}{N}\right)$!

Where We Are

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			
Babbush et al. ’26	EC-256: 1200 qubits	Not public					
Schrottenloher ’26	EC-256: 1192 qubits						
Jacobi factoring (Meyer-R-Vaikuntanathan-Van Kirk ’25)	*Factoring P^2 Q where Q has 500 bits with ~ 1000 qubits*			✔		✔	

Where We Are

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			
Babbush et al. ’26	EC-256: 1200 qubits	Not public					
Schrottenloher ’26	EC-256: 1192 qubits						
Jacobi factoring (Meyer-R-Vaikuntanathan-Van Kirk ’25)	*Factoring P^2 Q where Q has 500 bits with ~ 1000 qubits*		✔	✔		✔	

Roadmap

- Part I: a birds-eye view of space cost in quantum period finding
- Part II: a bag of tricks for saving space
- Part III: an overview of recent successes in applying these tricks to RSA and elliptic-curve discrete log
- Part IV: Jacobi factoring as a case study on some of these tricks
  - Trick 1: shortening the period
  - Trick 2: compressing the output and ancilla qubits

Output Compression is Easy

- Happy accident: the Jacobi symbol $j(x) = \left(\frac{x}{N}\right)$ is already ± 1, so the output is just one bit!
- What about the ancilla qubits?

30000 Foot View: Quantum Streaming

- Goal: compute $\left(\frac{a}{N}\right)$
  - Small quantum input $a \leq Q^{1+\epsilon}$

30000 Foot View: Quantum Streaming

- Goal: compute $\left(\frac{a}{N}\right)$
  - Small quantum input $a \leq Q^{1+\epsilon}$
  - Large classical input N

30000 Foot View: Quantum Streaming

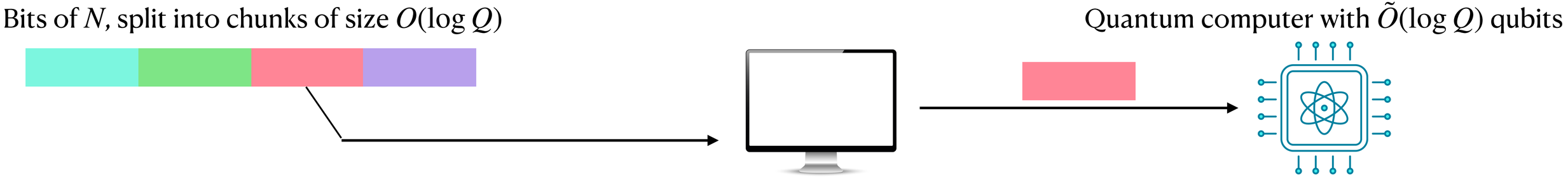
- Goal: compute $\left(\frac{a}{N}\right)$
  - Small quantum input $a \leq Q^{1+\epsilon}$
  - Large classical input N
- How could we compute this without ever writing down all of N quantumly?

30000 Foot View: Quantum Streaming

- Goal: compute $\left(\frac{a}{N}\right)$
  - Small quantum input $a \leq Q^{1+\epsilon}$
  - Large classical input N
- How could we compute this without ever writing down all of N quantumly?
- **Idea:** N is classically known → could quantumly "stream" through bits of N to save space

30000 Foot View: Quantum Streaming

- Goal: compute $\left(\frac{a}{N}\right)$
  - Small quantum input $a \leq Q^{1+\epsilon}$
  - Large classical input N
- How could we compute this without ever writing down all of N quantumly?
- **<u>Idea:</u>** N is classically known → could quantumly "stream" through bits of N to save space

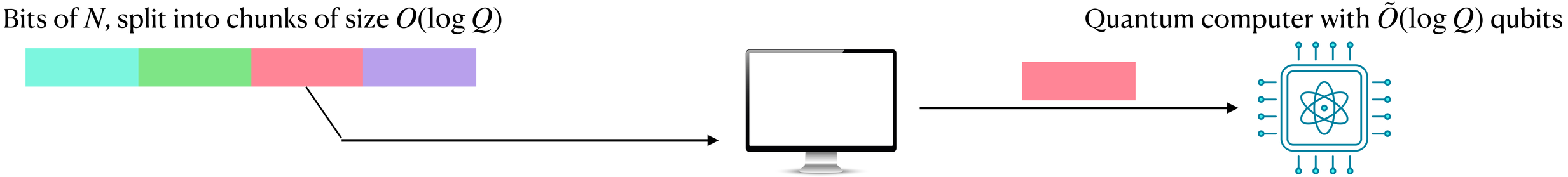


30000 Foot View: Quantum Streaming

- Goal: compute $\left(\frac{a}{N}\right)$
  - Small quantum input $a \leq Q^{1+\epsilon}$
  - Large classical input N
- How could we compute this symbol streaming N quantumly?
- **Idea:** N is classically known → could quantumly "stream" through bits of N to save space

But quantum streaming is just a hope.

Why does the Jacobi symbol lend itself to streaming?



Aside: Algorithms Computing the Jacobi Symbol

ft. Euclid, 2000 years ago

Jacobi Properties

- Periodicity: $\left(\frac{a}{b}\right) = \left(\frac{a \bmod b}{b}\right)$
- Reciprocity:* $\left(\frac{a}{b}\right) = (-1)^{f(a,b)}\left(\frac{b}{a}\right)$

  for a very simple f

* modulo minor technical caveats; requires a, b odd

Aside: Algorithms Computing the Jacobi Symbol

ft. Euclid, 2000 years ago

Jacobi Properties

- Periodicity: $\left(\frac{a}{b}\right) = \left(\frac{a \bmod b}{b}\right)$
- Reciprocity:* $\left(\frac{a}{b}\right) = (-1)^{f(a,b)}\left(\frac{b}{a}\right)$

  for a very simple f

$$f(a,b) = \begin{cases} 0, \text{ if } a \equiv 1 \pmod 4 \text{ or } b \equiv 1 \pmod 4 \\ 1, \text{ if } a \equiv b \equiv 3 \pmod 4 \end{cases}$$

* modulo minor technical caveats; requires a, b odd

Aside: Algorithms Computing the Jacobi Symbol

ft. Euclid, 2000 years ago

Jacobi Properties

- Periodicity: $\left(\frac{a}{b}\right) = \left(\frac{a \bmod b}{b}\right)$
- Reciprocity:* $\left(\frac{a}{b}\right) = (-1)^{f(a,b)}\left(\frac{b}{a}\right)$

  for a very simple f

Greatest Common Divisor (GCD) Properties

- Periodicity: $\gcd(a, b) = \gcd(a \bmod b, b)$
- Reciprocity: $\gcd(a, b) = \gcd(b, a)$

* modulo minor technical caveats; requires a, b odd

Aside: Algorithms Computing the Jacobi Symbol

ft. Euclid, 2000 years ago

Jacobi Properties

- Periodicity: $\left(\frac{a}{b}\right) = \left(\frac{a \bmod b}{b}\right)$
- Reciprocity:* $\left(\frac{a}{b}\right) = (-1)^{f(a,b)}\left(\frac{b}{a}\right)$

  for a very simple f

Greatest Common Divisor (GCD) Properties

- Periodicity: $\gcd(a, b) = \gcd(a \bmod b, b)$
- Reciprocity: $\gcd(a, b) = \gcd(b, a)$

Extended Euclidean algorithm solves both these problems!

* modulo minor technical caveats; requires a, b odd

Aside: Algorithms Computing the Jacobi Symbol

ft. Euclid, 2000 years ago

- Extended Euclidean recursion: if $a < b$, swap a, b. Else, update $a \leftarrow a \bmod b$.
- Standard runtime: $O(\log a \log b)$

Aside: Algorithms Computing the Jacobi Symbol

ft. Euclid, 2000 years ago

- Extended Euclidean recursion: if $a < b$, swap a, b. Else, update $a \leftarrow a \bmod b$.
- Standard runtime: $O(\log a \log b)$
- Schönhage 1971: complicated (and little-known!) divide-and-conquer algorithm that outputs the "transcript" or "dialog" of extended Euclidean in $\tilde{O}(\log a + \log b)$ time

Aside: Algorithms Computing the Jacobi Symbol

ft. Euclid, 2000 years ago

- Extended Euclidean recursion: if $a < b$, swap a, b. Else, update $a \leftarrow a \bmod b$.
- Standard runtime: $O(\log a \log b)$
- Schönhage 1971: complicated (and little-known!) divide-and-conquer algorithm that outputs the "transcript" or "dialog" of extended Euclidean in $\tilde{O}(\log a + \log b)$ time
- Recent works (Khattar et al. '25, Schrottenloher '26) propose concretely efficient quantum implementations of extended Euclidean!

Computing the Jacobi Symbol

It's all about $N \bmod a$

- Our task: compute $\left(\frac{a}{N}\right)$ for $a \leq Q^{1+\epsilon}$

Computing the Jacobi Symbol

It's all about $N \bmod a$

- Our task: compute $\left(\frac{a}{N}\right)$ for $a \leq Q^{1+\epsilon}$
- First step of Euclidean algorithm:

$$\left(\frac{a}{N}\right) = (-1)^{f(a,N)}\left(\frac{N}{a}\right) = (-1)^{f(a,N)}\left(\frac{N \bmod a}{a}\right)$$

Computing the Jacobi Symbol

It's all about $N \bmod a$

- Our task: compute $\left(\frac{a}{N}\right)$ for $a \leq Q^{1+\epsilon}$
- First step of Euclidean algorithm:

$$\left(\frac{a}{N}\right) = (-1)^{f(a,N)}\left(\frac{N}{a}\right) = (-1)^{f(a,N)}\left(\frac{N \bmod a}{a}\right)$$

- Example:

$$\left(\frac{0101}{1001101010100010010110010111010011010}\right) \overset{swap}{\rightarrow} \left(\frac{1001101010100010010110010111010011010}{0101}\right) \overset{mod}{=} \left(\frac{0010}{0101}\right)$$

Computing the Jacobi Symbol

It's all about $N \bmod a$

- Our task: compute $\left(\frac{a}{N}\right)$ for $a \leq Q^{1+\epsilon}$
- First step of Euclidean algorithm:

$$\left(\frac{a}{N}\right) = (-1)^{f(a,N)}\left(\frac{N}{a}\right) = (-1)^{f(a,N)}\left(\frac{N \bmod a}{a}\right)$$

- Example:

$$\left(\frac{0101}{1001101010100010010110010111010011010}\right) \overset{swap}{\rightarrow} \left(\frac{1001101010100010010110010111010011010}{0101}\right) \overset{mod}{=} \left(\frac{0010}{0101}\right)$$

Both inputs are small ($< Q^{1+\epsilon}$) after just one step!

Computing the Jacobi Symbol

It's all about $N \bmod a$

- Our task: compute $\left(\frac{a}{N}\right)$ for $a \leq Q^{1+\epsilon}$
- First step of Euclidean algorithm:

$$\left(\frac{a}{N}\right) = (-1)^{f(a,N)}\left(\frac{N}{a}\right) = (-1)^{f(a,N)}\left(\frac{N \bmod a}{a}\right)$$

- Example:

$$\left(\frac{0101}{1001101010100010010110010111010011010}\right) \overset{swap}{\rightarrow} \left(\frac{1001101010100010010110010111010011010}{0101}\right) \overset{mod}{=} \left(\frac{0010}{0101}\right)$$

Both inputs are small ($< Q^{1+\epsilon}$) after just one step!

- Two step procedure:
  1. Use quantum streaming → compute $N \bmod a$ (this step is basically long division)

Computing the Jacobi Symbol

It's all about $N \bmod a$

- Our task: compute $\left(\frac{a}{N}\right)$ for $a \leq Q^{1+\epsilon}$
- First step of Euclidean algorithm:

$$\left(\frac{a}{N}\right) = (-1)^{f(a,N)}\left(\frac{N}{a}\right) = (-1)^{f(a,N)}\left(\frac{N \bmod a}{a}\right)$$

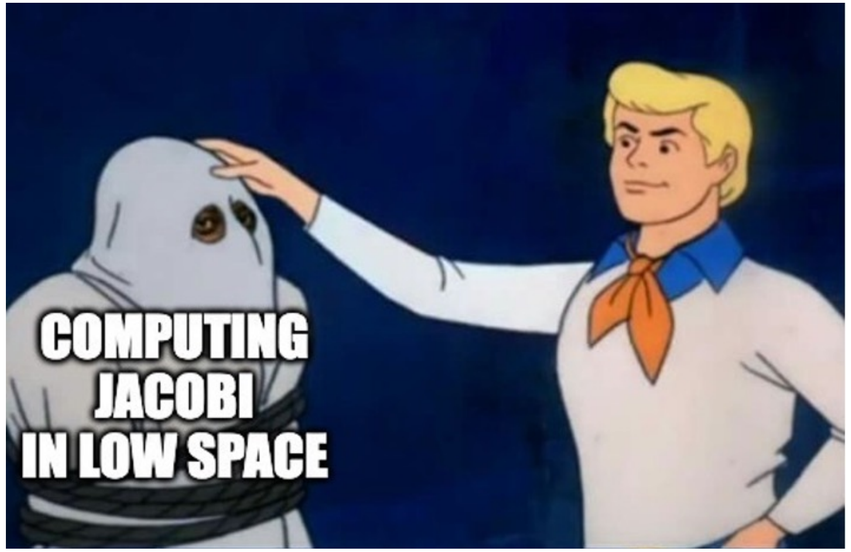
- Example:

$$\left(\frac{0101}{10011010101000100101100101110101 1010}\right) \overset{swap}{\rightarrow} \left(\frac{10011010101000100101100101110101 1010}{0101}\right) \overset{mod}{=} \left(\frac{0010}{0101}\right)$$

Both inputs are small ($< Q^{1+\epsilon}$) after just one step!

- Two step procedure:
  1. Use quantum streaming → compute $N \bmod a$ (this step is basically long division)
  2. Now can finish in $\tilde{O}(\log Q)$ gates/space using Schönhage or recent implementations of Euclid

Streaming for Jacobi



Streaming for Jacobi

Streaming for Jacobi

← The only place where we need quantum streaming!

We made it!

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			
Babbush et al. ’26	EC-256: 1200 qubits	Not public					
Schrottenloher ’26	EC-256: 1192 qubits						
Jacobi factoring (Meyer-R-Vaikuntanathan-Van Kirk ’25)	*Factoring P^2 Q where Q has 500 bits with ~ 1000 qubits*		✔	✔		✔	

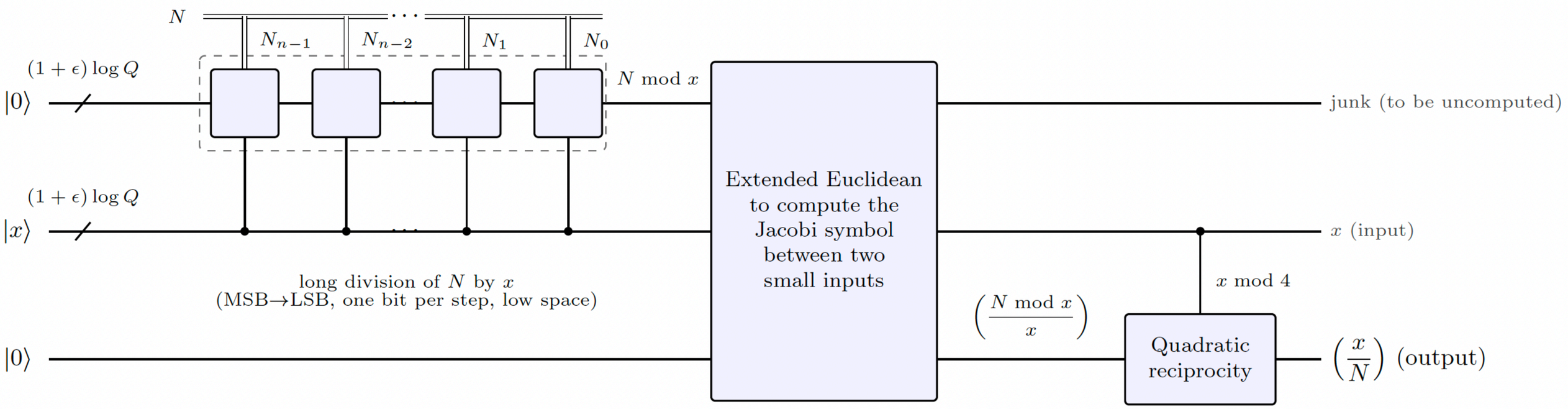
We made it!

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			
Babbush et al. ’26	EC-256: 1200 qubits	Not public					
Schrottenloher ’26	EC-256: 1192 qubits						
Jacobi factoring (Meyer-R-Vaikuntanathan-Van Kirk ’25)	*Factoring P^2 Q where Q has 500 bits with ~ 1000 qubits*		✔	✔	✔	✔	

We made it!

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			
Babbush et al. ’26	EC-256: 1200 qubits	Not public					
Schrottenloher ’26	EC-256: 1192 qubits						
Jacobi factoring (Meyer-R-Vaikuntanathan-Van Kirk ’25)	*Factoring P^2 Q where Q has 500 bits with ~ 1000 qubits*		✔	✔	✔	✔	✔

Jacobi Circuit Schematic



Open Questions

- Better classical algorithms for factoring P^2Q when Q is small?

Open Questions

- Better classical algorithms for factoring P^2Q when Q is small?
- What other integers N can we factor using this algorithm (e.g. using number theory magic)?

Open Questions

- Better classical algorithms for factoring P^2Q when Q is small?
- What other integers N can we factor using this algorithm (e.g. using number theory magic)?
  - Current generalisation: can **completely** factor $N = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_r^{\alpha_r}$ for distinct $\alpha_1, \ldots, \alpha_r$

Open Questions

- Better classical algorithms for factoring P^2Q when Q is small?
- What other integers N can we factor using this algorithm (e.g. using number theory magic)?
  - Current generalisation: can **completely** factor $N = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_r^{\alpha_r}$ for distinct $\alpha_1, \ldots, \alpha_r$
- Other quantum factoring algorithms exploiting special structure in N? (Many such algorithms in the classical world)

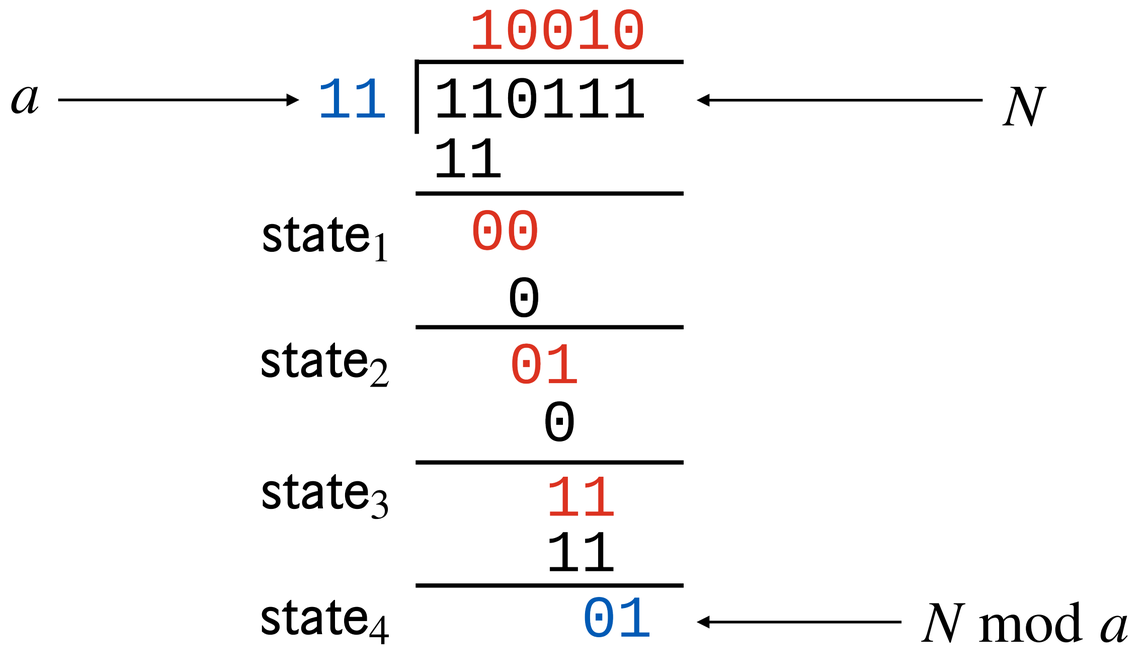
Thank you! Questions?

Authors	Result	Tricks for saving input qubits			Output + ancilla compression	Arithmetic ingredients	
		Qubit recycling	Multiple runs	Shorter period		Legendre symbol	Euclidean algorithm
Shor + qubit recycling	RSA-2048: 4100 qubits						
Chevignard-Fouque-Schrottenloher, Gidney ’25	RSA-2048: 1399 qubits EC-256: 1193 qubits			Via EH17 reduction to dlog			
Babbush et al. ’26	EC-256: 1200 qubits	Not public					
Schrottenloher ’26	EC-256: 1192 qubits						
Jacobi factoring (Meyer-R-Vaikuntanathan-Van Kirk ’25)	*Factoring P^2 Q where Q has 500 bits with ~ 1000 qubits*						

Bonus Slides

Our Construction, In Detail

A Natural Attempt: Long Division

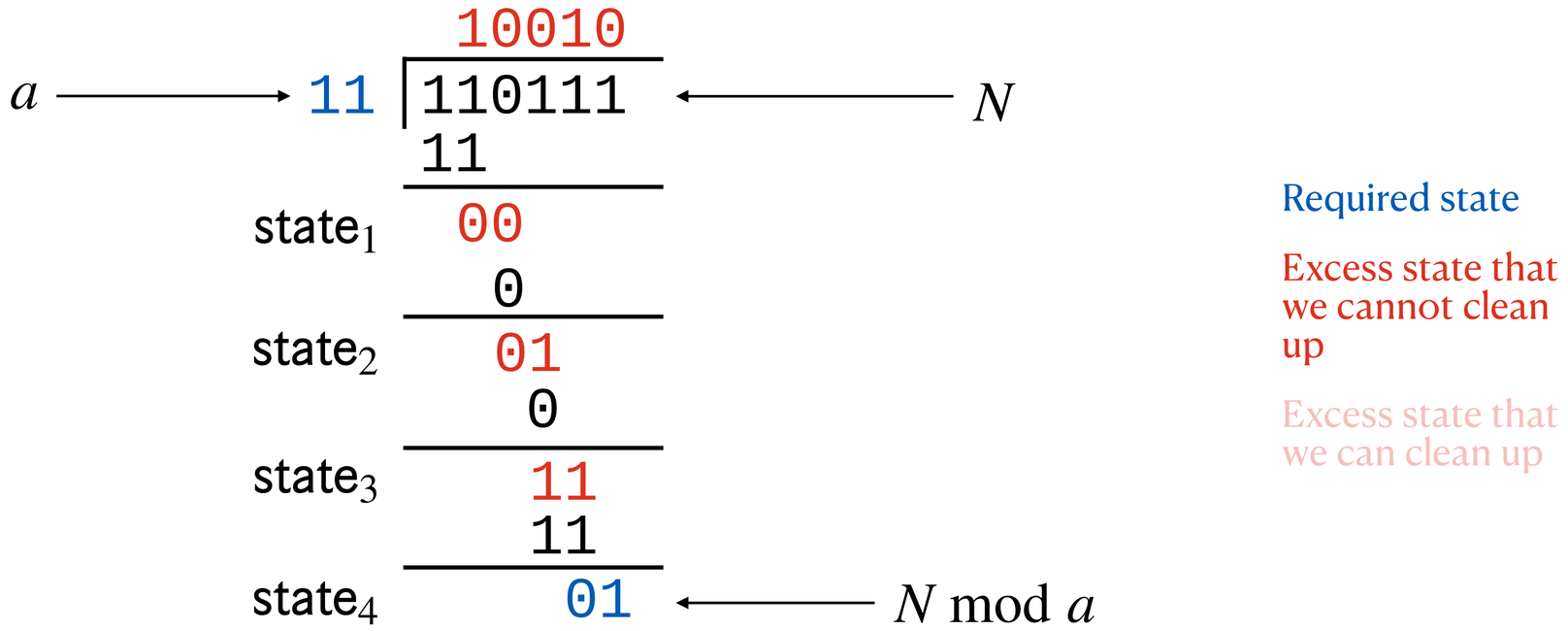


Required state

Excess state that we cannot clean up

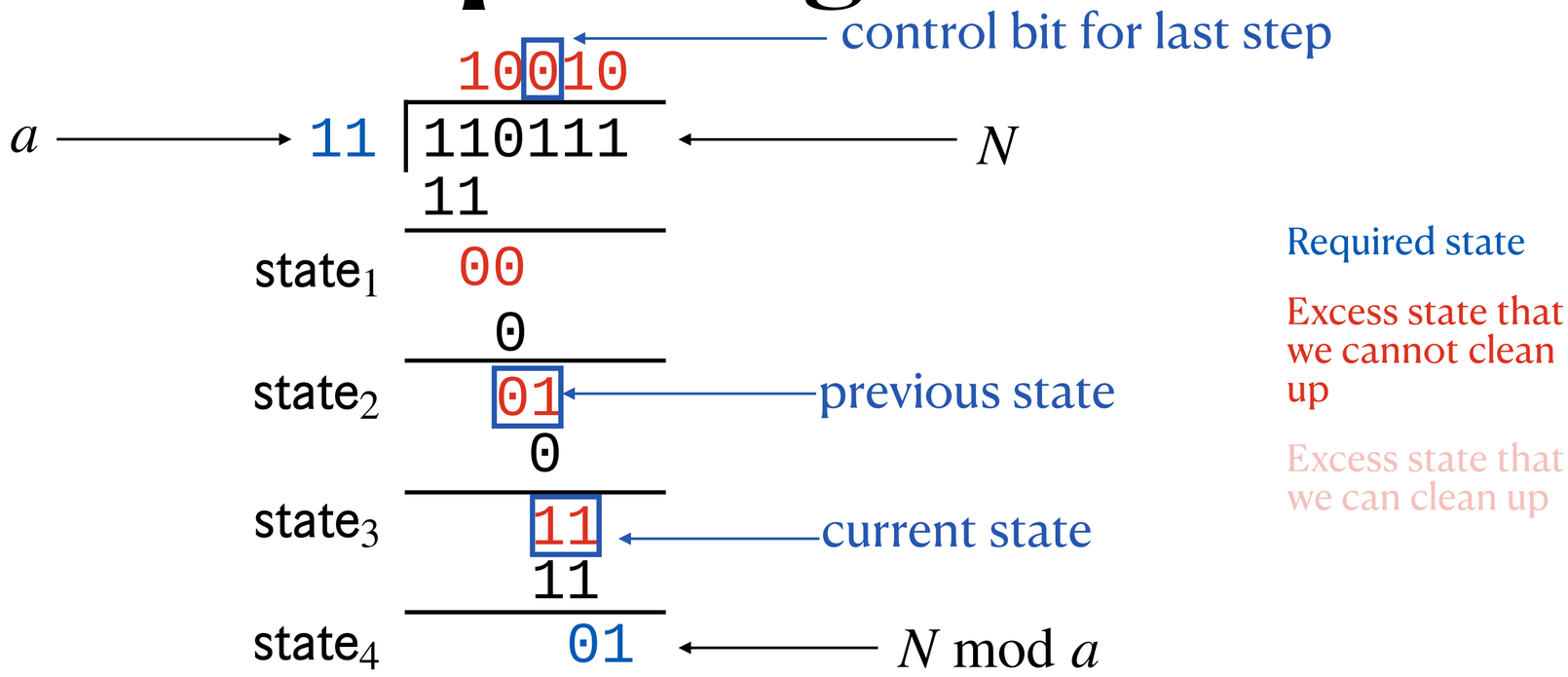
Excess state that we can clean up

A Natural Attempt: Long Division



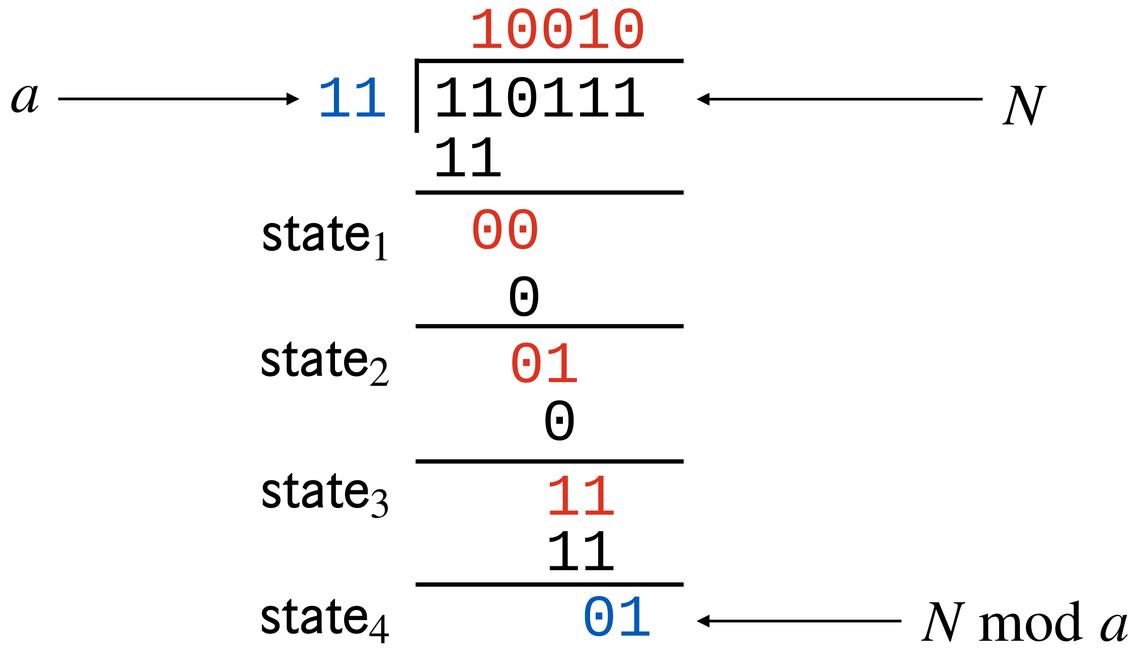
- Pro: only need to look at $O(m)$ bits of N at a time, and each state_t is compact

A Natural Attempt: Long Division



- Pro: only need to look at $O(m)$ bits of N at a time, and each state_t is compact
- Con: no reversibility → end up using $O(n)$ qubits anyway

Long Division: A Bird's Eye View

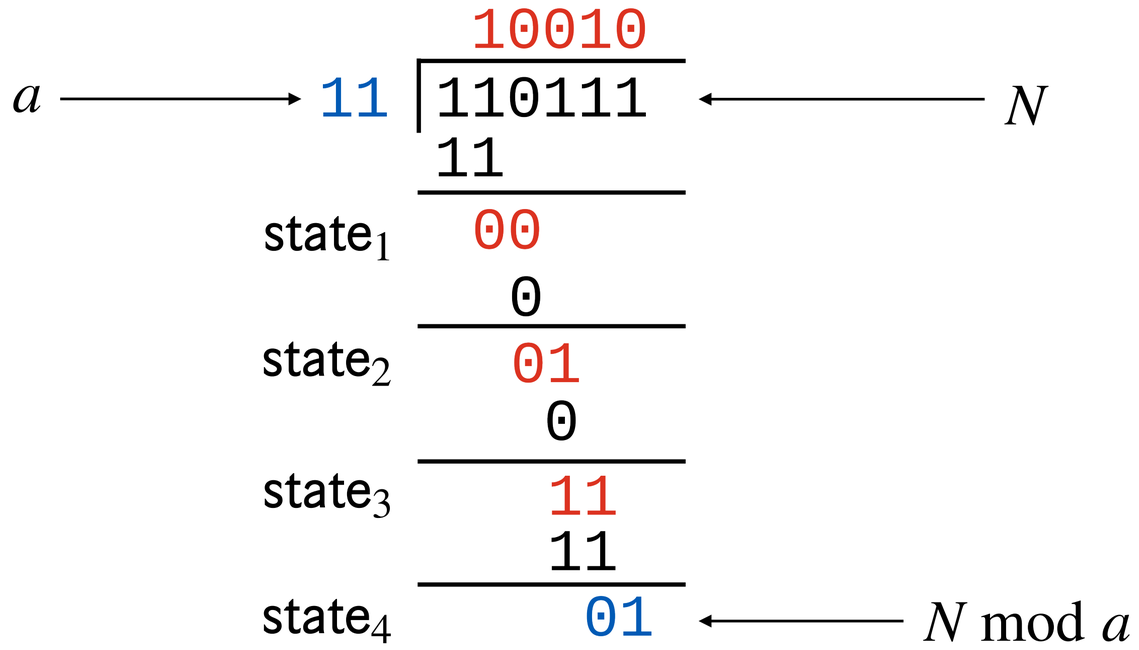


Required state

Excess state that we cannot clean up

Excess state that we can clean up

Long Division: A Bird's Eye View



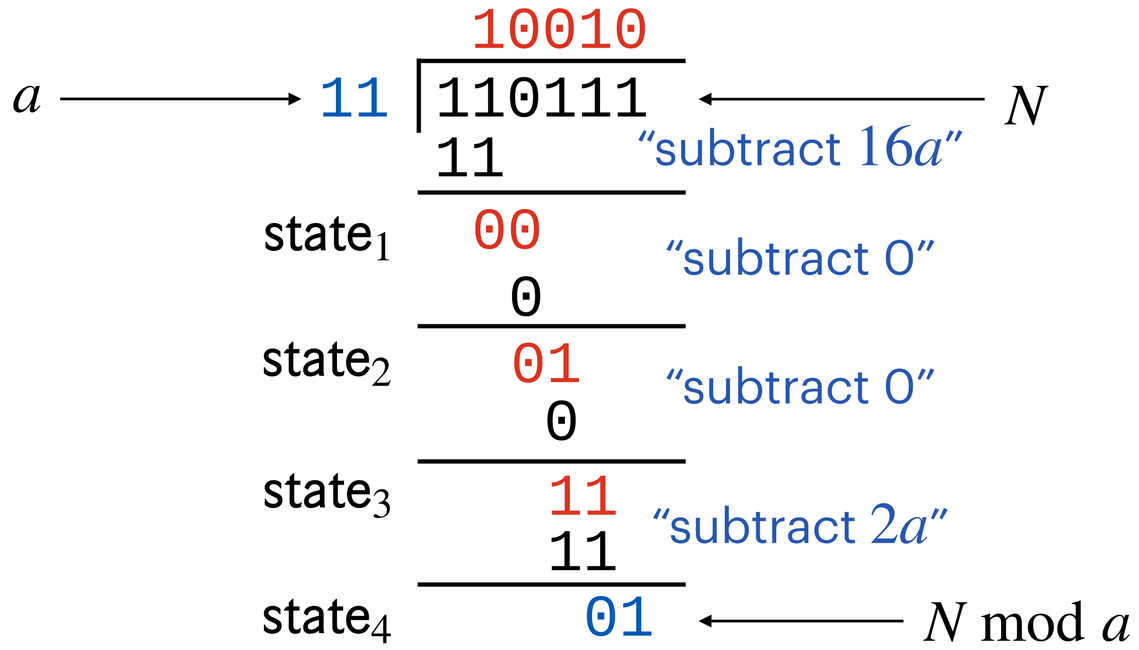
Required state

Excess state that we cannot clean up

Excess state that we can clean up

- Goal of long division: find k such that $N \approx ka$, and output $N - ka$

Long Division: A Bird's Eye View



Required state

Excess state that we cannot clean up

Excess state that we can clean up

- Goal of long division: find k such that $N \approx ka$, and output $N - ka$
- Does this by subtracting $2^r a$ from state, starting with large r (most significant bit of k) and going down to $r = 0$ (least significant bit of k)

Our Idea: "Backwards Long Division"

Long division: initialise state $= N$ and subtract $2^r a$ from state,
starting from large r (MSB of k) and going down to small r (LSB of k)

Our Idea: "Backwards Long Division"

Long division: initialise state $= N$ and subtract $2^r a$ from state,
starting from large r (MSB of k) and going down to small r (LSB of k)

Multiple of a from long division: `110110`

N: `110111`

Our Idea: "Backwards Long Division"

Long division: initialise state $= N$ and subtract $2^r a$ from state,
starting from large r (MSB of k) and going down to small r (LSB of k)

Multiple of a from long division: `110110`
N: `110111`
From backwards long division: `100111`

Backwards long division: initialise state $= 0$ and **add $2^r a$ to state,**
starting from small r (LSB of k) and going up to large r (MSB of k)

Our Idea: "Backwards Long Division"

Long division: initialise state $= N$ and subtract $2^r a$ from state,
starting from large r (MSB of k) and going down to small r (LSB of k)

Multiple of a from long division: `110110`
N: `110111`
From backwards long division: `100111`

Backwards long division: initialise state $= 0$ and **add $2^r a$ to state,**
starting from small r (LSB of k) and going up to large r (MSB of k)

Key observation: it is very easy at time t to decide based on state
whether we just added $2^t a$; simply check whether state $\geq 2^t a$

Our Idea: "Backwards Long Division"

Long division: initialise state $= N$ and subtract $2^r a$ from state, starting from large r (MSB of k) and going down to small r (LSB of k)

Multiple of a from long division: 110110

N: 110111

From backwards long division: 100111

Backwards long division: initialise state $= 0$ and **add $2^r a$ to state, starting from small r (LSB of k) and going up to large r (MSB of k)**

After backwards long division: once we have a multiple of a that matches N in the least significant bits, computing $N \bmod a$ is straightforward!

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** state $\geq 2^t a$

Our Idea: "Backwards Long Division"

Long division: initialise state $= N$ and subtract $2^r a$ from state, starting from large r (MSB of k) and going down to small r (LSB of k)

Multiple of a from long division: 110110

N: 110111

From backwards long division: 100111

Backwards long division: initialise state $= 0$ and **add $2^r a$ to state, starting from small r (LSB of k) and going up to large r (MSB of k)**

After backwards long division: once we have a multiple of a that matches N in the least significant bits, computing N mod a is straightforward!

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** state $\geq 2^t a$

Implementing Backwards Long Division

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** state $\geq 2^t a$

Implementing Backwards Long Division

- Instead of constructing k from MSB to LSB, let's construct it from LSB to MSB

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** state $\geq 2^t a$

Implementing Backwards Long Division

- Instead of constructing k from MSB to LSB, let's construct it from LSB to MSB
- Algorithm (assume for simplicity that a is odd):
  - Initialise $\mathsf{state} = 0$ (state will eventually be our multiple ka)

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** $\mathsf{state} \geq 2^t a$

Implementing Backwards Long Division

- Instead of constructing k from MSB to LSB, let's construct it from LSB to MSB
- Algorithm (assume for simplicity that a is odd):
  - Initialise $\mathsf{state} = 0$ (state will eventually be our multiple ka)
  - At time t, add $2^t a$ to state if 

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** $\mathsf{state} \geq 2^t a$

Implementing Backwards Long Division

- Instead of constructing k from MSB to LSB, let's construct it from LSB to MSB
- Algorithm (assume for simplicity that a is odd):
  - Initialise $\mathsf{state} = 0$ (state will eventually be our multiple ka)
  - At time t, add $2^t a$ to state if 

  - Stop when N and state ($= ka$) are "close" in some sense 

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** $\mathsf{state} \geq 2^t a$

Implementing Backwards Long Division

- Instead of constructing k from MSB to LSB, let's construct it from LSB to MSB
- Algorithm (assume for simplicity that a is odd):
  - Initialise state $= 0$ (state will eventually be our multiple ka)
  - At time t, add $2^t a$ to state if N and state differ in the tth least significant bit
  - Stop when N and state ($= ka$) are "close" in some sense

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** state $\geq 2^t a$

Implementing Backwards Long Division

- Instead of constructing k from MSB to LSB, let's construct it from LSB to MSB
- Algorithm (assume for simplicity that a is odd):
  - Initialise state $= 0$ (state will eventually be our multiple ka)
  - At time t, add $2^t a$ to state if N and state differ in the tth least significant bit
  - Stop when N and state ($= ka$) are "close" in some sense 

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** state $\geq 2^t a$

Implementing Backwards Long Division

- Instead of constructing k from MSB to LSB, let's construct it from LSB to MSB
- Algorithm (assume for simplicity that a is odd):
  - Initialise state $= 0$ (state will eventually be our multiple ka)
  - At time t, add $2^t a$ to state if N and state differ in the tth least significant bit
  - Stop when N and state ($= ka$) agree on the $n - m$ least significant bits

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** state $\geq 2^t a$

Implementing Backwards Long Division

- Instead of constructing k from MSB to LSB, let's construct it from LSB to MSB
- Algorithm (assume for simplicity that a is odd):
  - Initialise state $= 0$ (state will eventually be our multiple ka)
  - At time t, add $2^t a$ to state if N and state differ in the tth least significant bit
  - Stop when N and state ($= ka$) agree on the $n - m$ least significant bits
    - Equivalently: $N \equiv ka \pmod{2^{n-m}}$

Key observation: it is very easy at time t to decide based on state **whether we just added $2^t a$; simply check whether** state $\geq 2^t a$

Tracing Through Our Algorithm

Recall:

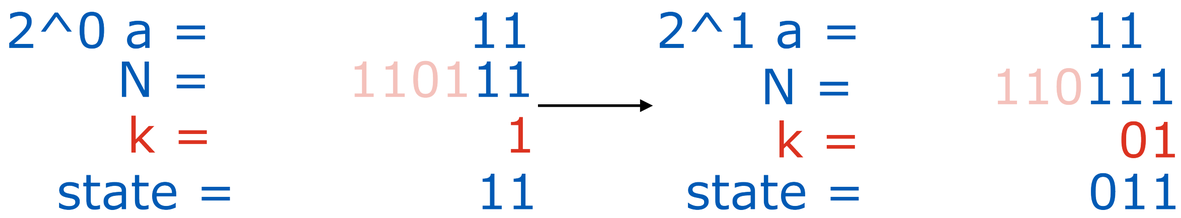
state = ka

```
2^0 a =          11
    N =      110111
    k =           1
state =          11
```

Tracing Through Our Algorithm

Recall:

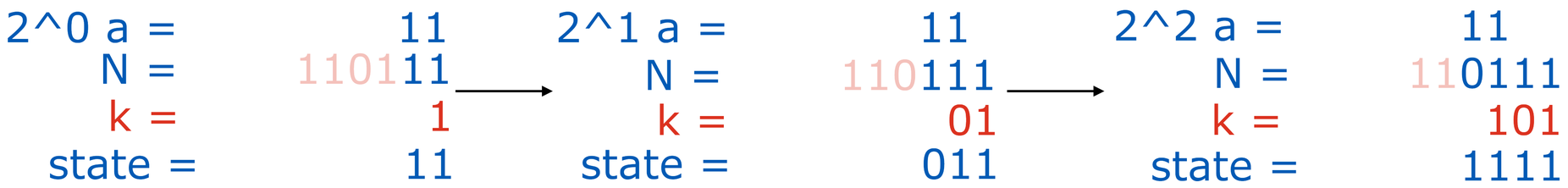
state = ka



Tracing Through Our Algorithm

Recall:

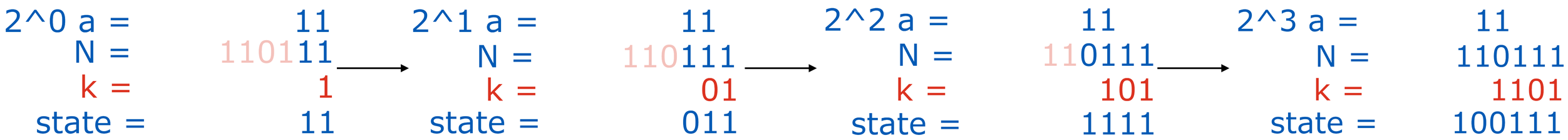
state = ka



Tracing Through Our Algorithm

Recall:

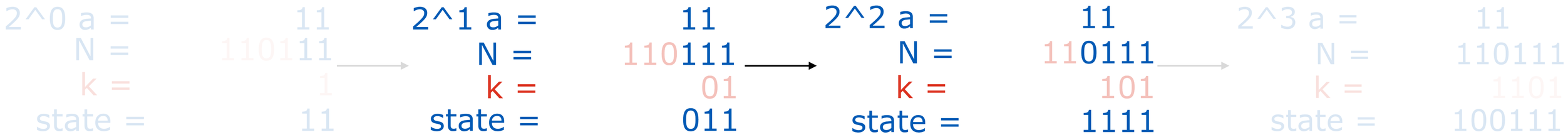
state = ka



Tracing Through Our Algorithm

Recall:

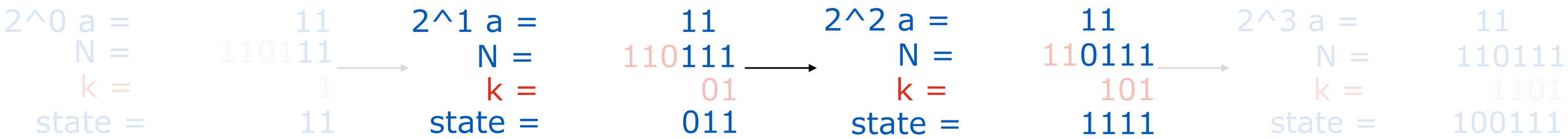
state = ka



Tracing Through Our Algorithm

Recall:

state $= ka$

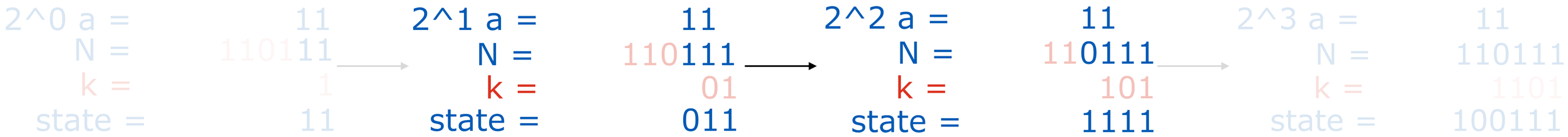


- Our earlier observation: a simple comparison between state and $2^t a$ suffices for uncomputation at each step

Tracing Through Our Algorithm

Recall:

state $= ka$

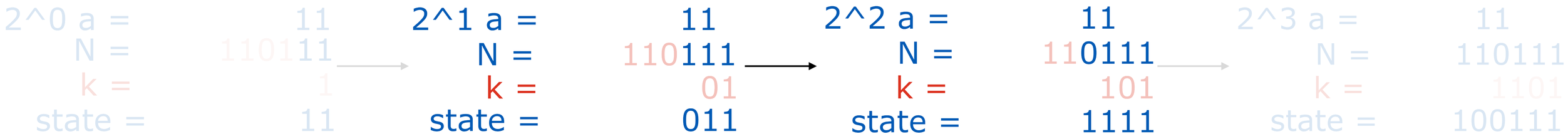


- Our earlier observation: a simple comparison between state and $2^t a$ suffices for uncomputation at each step
- Second observation:
  - The trailing bits of state and N are classically known; and

Tracing Through Our Algorithm

Recall:

state $= ka$

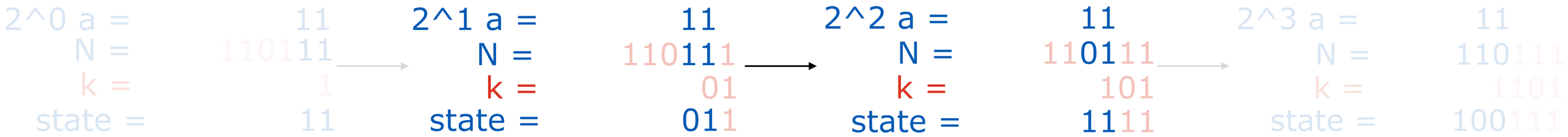


- Our earlier observation: a simple comparison between state and $2^t a$ suffices for uncomputation at each step
- Second observation:
  - The trailing bits of state and N are classically known; and
  - We only need the leading-order bits of state to make the above comparison with $2^t a$

Tracing Through Our Algorithm

Recall:

state $= ka$



- Our earlier observation: a simple comparison between state and $2^t a$ suffices for uncomputation at each step
- Second observation:
  - The trailing bits of state and N are classically known; and
  - We only need the leading-order bits of state to make the above comparison with $2^t a$

This now gets us down to $O(m)$ space!

Pushing Down the Gates and Depth

- Idea: instead of setting one bit of k at a time, we will set m bits at a time

Pushing Down the Gates and Depth

- Idea: instead of setting one bit of k at a time, we will set m bits at a time
  - Enables us to benefit from fast integer multiplication algorithms (for m-bit inputs)

Pushing Down the Gates and Depth

- Idea: instead of setting one bit of k at a time, we will set m bits at a time
  - Enables us to benefit from fast integer multiplication algorithms (for m-bit inputs)
  - To decide what multiple of a to add at each step: start by computing $a^{-1} \bmod 2^m$

Pushing Down the Gates and Depth

- Idea: instead of setting one bit of k at a time, we will set m bits at a time
  - Enables us to benefit from fast integer multiplication algorithms (for m-bit inputs)
  - To decide what multiple of a to add at each step: start by computing $a^{-1} \bmod 2^m$

Two options to compute $a^{-1} \bmod 2^m$:

1. (Obnoxious; concretely inefficient) Use Schönhage to achieve this in $\tilde{O}(m)$ gates/space/depth

Pushing Down the Gates and Depth

- Idea: instead of setting one bit of k at a time, we will set m bits at a time
  - Enables us to benefit from fast integer multiplication algorithms (for m-bit inputs)
  - To decide what multiple of a to add at each step: start by computing $a^{-1} \bmod 2^m$

Two options to compute $a^{-1} \bmod 2^m$:

1. (Obnoxious; concretely inefficient) Use Schönhage to achieve this in $\tilde{O}(m)$ gates/space/depth
2. (Using the 2^m modulus; can be made concretely efficient) Recurse down to computing $a^{-1} \bmod 2^{m/2}$
   a. Recursion step just does some multiplications and bit shifts on m-bit integers

Pushing Down the Gates and Depth

- Idea: instead of setting one bit of k at a time, we will set m bits at a time
  - Enables us to benefit from fast integer multiplication algorithms (for m-bit inputs)
  - To decide what multiple of a to add at each step: start by computing $a^{-1} \bmod 2^m$
- Space usage: still $\tilde{O}(m)$

Pushing Down the Gates and Depth

- Idea: instead of setting one bit of k at a time, we will set m bits at a time
  - Enables us to benefit from fast integer multiplication algorithms (for m-bit inputs)
  - To decide what multiple of a to add at each step: start by computing $a^{-1} \bmod 2^m$
- Space usage: still $\tilde{O}(m)$
- Computational work: $O(n/m)$ multiplications of m-bit integers

Pushing Down the Gates and Depth

- Idea: instead of setting one bit of k at a time, we will set m bits at a time
  - Enables us to benefit from fast integer multiplication algorithms (for m-bit inputs)
  - To decide what multiple of a to add at each step: start by computing $a^{-1} \bmod 2^m$
- Space usage: still $\tilde{O}(m)$
- Computational work: $O(n/m)$ multiplications of m-bit integers
  - Gates: $O(n/m) \times \tilde{O}(m) = \tilde{O}(n)$
  - Depth: $O(n/m) \times \tilde{O}(1) = \tilde{O}(n/m)$

From Backwards Long Division to $N \bmod a$

From Backwards Long Division to $N \bmod a$

- The example we just worked out:
  - $N = 55, a = 3$ $(n = 6, m = 2)$

simplified presentation based on an observation by Daniel J. Bernstein

From Backwards Long Division to $N \bmod a$

- The example we just worked out:
  - $N = 55, a = 3$ $(n = 6, m = 2)$
  - End up with $ka = \text{state} = 39 = 13 \times 3 \equiv N \pmod{2^{n-m} = 16}$

simplified presentation based on an observation by Daniel J. Bernstein

From Backwards Long Division to $N \bmod a$

- The example we just worked out:
  - $N = 55, a = 3$ $(n = 6, m = 2)$
  - End up with $ka = \mathsf{state} = 39 = 13 \times 3 \equiv N \pmod{2^{n-m} = 16}$
- Now define $N' = \dfrac{N - \mathsf{state}}{2^{n-m}} = \dfrac{N - ka}{2^{n-m}}$ (informally, this is the "remainder")

simplified presentation based on an observation by Daniel J. Bernstein

From Backwards Long Division to $N \bmod a$

- The example we just worked out:
  - $N = 55, a = 3$ $(n = 6, m = 2)$
  - End up with $ka = \mathsf{state} = 39 = 13 \times 3 \equiv N \pmod{2^{n-m} = 16}$
- Now define $N' = \dfrac{N - \mathsf{state}}{2^{n-m}} = \dfrac{N - ka}{2^{n-m}}$ (informally, this is the "remainder")
- Finally: $N \equiv N' \cdot \left(2^{n-m} \bmod a\right) \pmod{a}$, and the RHS is easily computable with $\tilde{O}(m)$ gates!

simplified presentation based on an observation by Daniel J. Bernstein

From Backwards Long Division to $N \bmod a$

- The example we just worked out:
  - $N = 55, a = 3$ $(n = 6, m = 2)$
  - End up with $ka = \mathsf{state} = 39 = 13 \times 3 \equiv N \pmod{2^{n-m} = 16}$
- Now define $N' = \dfrac{N - \mathsf{state}}{2^{n-m}} = \dfrac{N - ka}{2^{n-m}}$ (informally, this is the "remainder")
- Finally: $N \equiv N' \cdot \left(2^{n-m} \bmod a\right) \pmod{a}$, and the RHS is easily computable with $\tilde{O}(m)$ gates!

That's it! We computed $N \bmod a$!

simplified presentation based on an observation by Daniel J. Bernstein