Challenges in quantum algorithms for integer factorization

D. J. Bernstein
University of Illinois at Chicago

Prelude: What is the fastest algorithm to sort an array?

```python
def blindsort(x):
    while not issorted(x):
        permuterandomly(x)
```

def bubblesort(x):
    for j in range(len(x)):
        for i in reversed(range(j)):
            x[i], x[i+1] = (min(x[i], x[i+1]), max(x[i], x[i+1]))

bubblesort takes poly time. Θ(n²) comparisons.
Huge speedup over blindsort!
Is this the end of the story?
Challenges in quantum algorithms for integer factorization

D. J. Bernstein
University of Illinois at Chicago

Prelude: What is the fastest algorithm to sort an array?

```python
def blindsort(x):
    while not issorted(x):
        permuterandomly(x)
```

```python
def bubblesort(x):
    for j in range(len(x)):
        for i in reversed(range(j)):
            x[i],x[i+1] = (min(x[i],x[i+1]),
                            max(x[i],x[i+1]))

bubblesort takes poly time.
Θ(n^2) comparisons.
Huge speedup over blindsort!

Is this the end of the story?
No, still not optimal.
Challenges in quantum algorithms for integer factorization

D. J. Bernstein
University of Illinois at Chicago

Prelude: What is the fastest algorithm to sort an array?

def blindsort(x):
    while not issorted(x):
        permuterandomly(x)

def bubblesort(x):
    for j in range(len(x)):
        for i in reversed(range(j)):
            x[i], x[i+1] = (min(x[i], x[i+1]), max(x[i], x[i+1]))

bubblesort takes poly time.
Θ(n²) comparisons.
Huge speedup over blindsort!

Is this the end of the story?
No, still not optimal.

Analogous: What is the fastest algorithm to factor integers?

Shor’s algorithm takes poly time.
Huge speedup over NFS!

$b^2(\log b)^{1+o(1)}$ qubit operations to factor $b$-bit integer, using standard subroutines for fast integer arithmetic.

Is this the end of the story?
def bubblesort(x):
    for j in range(len(x)):
        for i in reversed(range(j)):
            x[i], x[i+1] = (min(x[i], x[i+1]),
                            max(x[i], x[i+1]))

bubblesort takes poly time.
Θ(n^2) comparisons.

Huge speedup over blindsort!

Is this the end of the story?
No, still not optimal.

Analogous: What is the fastest algorithm to sort an array?

bubble sort takes poly time.
Θ(n^2) comparisons.
Huge speedup over blindsort!

Is this the end of the story?
No, still not optimal.

Analogous: What is the fastest algorithm to factor integers?

Shor’s algorithm takes poly time.
Huge speedup over NFS!

\[ b^2(\log b)^{1+o(1)} \] qubit operations
to factor \( b \)-bit integer,
using standard subroutines
for fast integer arithmetic.

Is this the end of the story?
def bubblesort(x):
    for j in range(len(x)):
        for i in reversed(range(j)):
            x[i], x[i+1] = (min(x[i], x[i+1]),
                            max(x[i], x[i+1]))

bubblesort takes poly time.
Θ(n^2) comparisons.
Huge speedup over blindsort!

Is this the end of the story?
No, still not optimal.

Analogous: What is the fastest algorithm to factor integers?
Shor’s algorithm takes poly time.
Huge speedup over NFS!

\[ b^2 (\log b)^{1+o(1)} \] qubit operations
to factor \( b \)-bit integer,
using standard subroutines for fast integer arithmetic.

Is this the end of the story?
def bubblesort(x):
    for j in range(len(x)):
        for i in reversed(range(j)):
            x[i], x[i+1] = (min(x[i], x[i+1]), max(x[i], x[i+1]))

bubblesort takes poly time.
Θ(\(n^2\)) comparisons.
Huge speedup over blindsort!

Is this the end of the story?
No, still not optimal.

Analogous: What is the fastest algorithm to factor integers?

Shor’s algorithm takes poly time.
Huge speedup over NFS!

\(b^2(\log b)^{1+o(1)}\) qubit operations to factor \(b\)-bit integer,
using standard subroutines for fast integer arithmetic.

Is this the end of the story?
def bubblesort(x):
    for j in range(len(x)):
        for i in reversed(range(j)):
            x[i], x[i+1] = (min(x[i], x[i+1]), max(x[i], x[i+1]))

bubblesort takes poly time. \( \Theta(n^2) \) comparisons.
Huge speedup over blindsort!

Is this the end of the story?
No, still not optimal.

Analogous: What is the fastest algorithm to factor integers?

Shor’s algorithm takes poly time. Huge speedup over NFS!

\( b^2(\log b)^{1+o(1)} \) qubit operations to factor \( b \)-bit integer, using standard subroutines for fast integer arithmetic.

Is this the end of the story?
No, still not optimal.

“Shor’s algorithm: the bubble sort of integer factorization.”
def bubblesort(x):
    for j in range(len(x)):
        for i in reversed(range(j)):
            x[i], x[i+1] = (min(x[i], x[i+1]), max(x[i], x[i+1]))

bubblesort takes poly time.
Θ(n^2) comparisons.
Huge speedup over blindsort!
Is this the end of the story?
No, still not optimal.

Analogous: What is the fastest algorithm to factor integers?
Shor’s algorithm takes poly time.
Huge speedup over NFS!
b^2(\log b)^{1+o(1)} qubit operations to factor b-bit integer,
using standard subroutines for fast integer arithmetic.
Is this the end of the story?
No, still not optimal.
“Shor’s algorithm: the bubble sort of integer factorization.”

A simple exercise to illustrate suboptimality of Shor’s algorithm:
Find a prime divisor of

31415926535897932384626433832795028841971693993751058209749445923078164062862089
98628034825342117067982148086513282306647093844609550582231725359408128481117450
28410270193852110555964462294895493038196442881097566593344612847564823378678316
5271201909146844090122495343014654958537105079227968925892354201995611212902196
08640344181598136297747713099605187072113499999983729780499510597317328160963185
95024459455346908302642522308253344685035261931188171010003137838752886587533208
38142061717766914730359825349042875546873115956286388235378759375195778185778053
2171228806613001922853616035637076646280466842590694035587640247496473028618297455570674
6023648066549911988164756001614524984385233290739414904946016354668049225125205117392984
5071237137869609599465764078951269413639443745530506874105978595977297499725246808459872
7807977156914359976016427394522674635593634568174324156010150308617928168299898472265880
21051141354735739540374200731057853910053761468067491195618146751426912

31415926535897932384626433832795028841971693993751058209749445923078164062862089
98628034825342117067982148086513282306647093844609550582231725359408128481117450
28410270193852110555964462294895493038196442881097566593344612847564823378678316
5271201909146844090122495343014654958537105079227968925892354201995611212902196
08640344181598136297747713099605187072113499999983729780499510597317328160963185
95024459455346908302642522308253344685035261931188171010003137838752886587533208
38142061717766914730359825349042875546873115956286388235378759375195778185778053
2171228806613001922853616035637076646280466842590694035587640247496473028618297455570674
6023648066549911988164756001614524984385233290739414904946016354668049225125205117392984
5071237137869609599465764078951269413639443745530506874105978595977297499725246808459872
7807977156914359976016427394522674635593634568174324156010150308617928168299898472265880
21051141354735739540374200731057853910053761468067491195618146751426912
def bubblesort(x):
    for j in range(len(x)):
        for i in reversed(range(j)):
            x[i], x[i+1] = (min(x[i], x[i+1]), max(x[i], x[i+1]))

bubblesort takes poly time.
Θ(n²) comparisons.
Huge speedup over blindsort!
Is this the end of the story?
No, still not optimal.

Analogous: What is the fastest algorithm to factor integers?
Shor’s algorithm takes poly time.
Huge speedup over NFS!
Is this the end of the story?
No, still not optimal.

“Shor’s algorithm: the bubble sort of integer factorization.”

A simple exercise to illustrate suboptimality of Shor’s algorithm:
Find a prime divisor of $\frac{10^{3009}}{986280348624338327950288419796826034825342117067982148008651328230664.$

$314159265358979323846264338327950288419796826034825342117067982148008651328230664,
284102079193852110555964622948954930381955721019014564856923460334861045432664847488152092096282925490171536367892592036433057270365759591850930921861173189326211489122793818301194912893673362440656643705392171762931767523846748148676940513173637178721468440901224953430146549598530864034185191362977477130996051870721195024459455346908302642522308253344685038142061717766191473035982534904287554568721712268066113001927876611195909216420198682303019520353018529689957736255941396908293311666121782758890750983817546374285836160356570766010471018194295559619846208046684256096491233316770289891320103551764024794673263914199272604269922708612894555707674983850454958858669269956023648066599119988183479775356636986907429016147060061164524919217212477250101444398523223907394143334547762418862518983904946016534666049886273237917860857483322512505117392984896084128488626945604025047123713786960956363419172874677646579946576403795126946839835259570982582262213639443755035068203496252451749399651474105978859972975498930161753928468138499725246808459872736446958486583637632678070771569143599770012966108944169486656016842739452267467688952521385252499543559363561317432141251507606947945109659560101503086179286809208747609178249385416829989487226588048576540142704775551321051141354757395321134271661021359695340374200731057853906219838747487084849801005370614680674919278191197939595206149195618146751429123978490409718649423193


2

Analogous: What is the fastest algorithm to factor integers?

Shor’s algorithm takes poly time.
Huge speedup over NFS!

\[ b^2 \log b \left(1+o(1) \right) \] qubit operations to factor \( b \)-bit integer, using standard subroutines for fast integer arithmetic.

Is this the end of the story?
No, still not optimal.

“Shor’s algorithm: the bubble sort of integer factorization.”

A simple exercise to illustrate suboptimality of Shor’s algorithm:
Find a prime divisor of \( \sqrt[3]{10^{3009}} \).

3

\[ \begin{array}{c}
314159265358979323846264338327950288419716939937510582097494453
96828034825342117067982148086513282306647093846909550582231725
284102701938521055596442294959493039184628190975669539446125
5271201910456485699236403647254648213936077622491412737
748152092096282925491715364367829593600113305453802466251
43305727036575951959028161173819326117930511585047446239796
489122739838130119412983637364406566403086021394946352247371
705392171762931765238467481846766940513200056812714523568027
17363717872146844090122495343014659458537105792729968925882354
08640341815981362977471309965018707211349999937297980499510
950245495543669030262452230285344685053251931818171010003137
3814206171776614730359825349042875546873115956263838253787759
21712286061630139278766111959092162019983905027501606558632
6823030159203530185269899773625299413891249721772823749131515
9058295331166187255889059083175463746493991952506309777707
28583616035637076061047101819429556198946767374434245593779
46208046684259069419233136770288915210475216205966204258038
035576402474964732639141992726042269922798738254716216300931472
02861827945557067493850549588586926995690927210795790329553
60236480665499198813847977535656396907426452782651518417574
0816747060111452491921739217477352014119417536854161361315753
84385232329079414333454776241682621883694856209921222182
90496601653466808498662732379717860874338328796796681454100953
2255205011739298489860412848862946504241965260522106116830
5047123713786906956364371917284767764657579362138908658326459
99465740789152694803983525957908258262052248940772671974872868
13459757453655035566204396526524571493996514314298091606952093722
74105978957797297549899016715793824813826883868472741559518
4972524680845987273646498548653836736226260991246608512438484
7807977156914355977970012961608944169486558584840635423270222582
6016842739452267467889525213852549564666727828964556911635
359553646587432411251507609699475410965906402528879718093145
5600155300861794696902874760919724938958909714940967598562168
168299894722658048575604127404775515323796414512647623436453
2105141347357952311342716610213596595632442254849371871101
4037420073107859960219837474780847849683321445713806751894350
100573576146608708539281911979395062141963487544046374512372
19561814675142691239748940907186494321961567945208

3
Analogous: What is the fastest algorithm to factor integers?

Shor’s algorithm takes poly time. Huge speedup over NFS!

$$b^2(\log b)^{1+o(1)}$$ qubit operations to factor $$b$$-bit integer, using standard subroutines for fast integer arithmetic.

Is this the end of the story? No, still not optimal.

“Shor’s algorithm: the bubble sort of integer factorization.”

A simple exercise to illustrate suboptimality of Shor’s algorithm:

Find a prime divisor of $$\sqrt[10]{3009\pi}$$.
Analogous: What is the fastest algorithm to factor integers? Shor’s algorithm takes poly time. Huge speedup over NFS! 

H1+o(1) qubit operations for b-bit integer, standard subroutines in integer arithmetic.

The end of the story? Not optimal.

“Shor’s algorithm: the bubble sort for fast integer arithmetic.”

A simple exercise to illustrate suboptimality of Shor’s algorithm: Find a prime divisor of $\left[10^{3009}\pi\right]$. 

Important variations in the factorization problem:

• Maybe need one factor.
• Maybe need all factors.
• Maybe factors are large.
• Maybe inputs in superposition.
• Maybe there are many inputs.

Qubits (as even as possible).

Qubits

Area (A)

Qubit operations (gates)

Depth

Time
Analogous: What is the fastest algorithm to factor integers? Shor's algorithm takes poly time. Huge speedup over NFS! 

$2^n (\log_2 b)^{1+o(1)}$ qubit operations to factor $b$-bit integer, using standard subroutines for fast integer arithmetic.

Is this the end of the story? No, still not optimal. "Shor's algorithm: the bubble sort of integer factorization."

A simple exercise to illustrate suboptimality of Shor's algorithm: Find a prime divisor of $10^{3009}$. 

Important variations in the factorization problem: 
• Maybe need one factor. 
• Maybe need all factors. 
• Maybe factors are small. 
• Maybe factors are large. 
• Maybe there are many inputs. 
• Maybe inputs in superposition. 

Important variations in metrics (even assuming perfect devices): 
• Qubits. 
• Area ("$A$", including wire area). 
• Qubit operations ("gates"). 
• Depth. 
• Time ("$T$": latency).
A simple exercise to illustrate suboptimality of Shor’s algorithm: Find a prime divisor of \([10^{3009}\pi]\).
A simple exercise to illustrate suboptimality of Shor’s algorithm:

Find a prime divisor of \[ 10^{3009} \pi \].

Important variations in the factorization problem:

- Maybe need one factor.
- Maybe need all factors.
- Maybe factors are small.
- Maybe factors are large.
- Maybe there are many inputs.
- Maybe inputs in superposition.

Important variations in metrics (even assuming perfect devices):

- Qubits.
- Area ("A", including wire area).
- Qubit operations ("gates").
- Depth.
- Time ("T": latency).
The exercise to illustrate suboptimality of Shor’s algorithm:
Find a prime divisor of \[\left\lfloor 10^{3009}\pi \right\rfloor.\]

Important variations in the factorization problem:
- Maybe need one factor.
- Maybe need all factors.
- Maybe factors are small.
- Maybe factors are large.
- Maybe there are many inputs.
- Maybe inputs in superposition.

Important variations in metrics (even assuming perfect devices):
- Qubits.
- Area (“A”, including wire area).
- Qubit operations (“gates”).
- Depth.
- Time (“T”: latency).

Short-term RSA security
A simple exercise to illustrate suboptimality of Shor’s algorithm:
Find a prime divisor of $10^{3009}$.

Important variations in the factorization problem:
- Maybe need one factor.
- Maybe need all factors.
- Maybe factors are small.
- Maybe factors are large.
- Maybe there are many inputs.
- Maybe inputs in superposition.

Important variations in metrics (even assuming perfect devices):
- Qubits.
- Area (“$A$”, including wire area).
- Qubit operations (“gates”).
- Depth.
- Time (“$T$”: latency).

Short-term RSA security
A simple exercise to illustrate suboptimality of Shor's algorithm:

Find a prime divisor of $\frac{10^{3009}}{\pi}$.

Important variations in the factorization problem:
- Maybe need one factor.
- Maybe need all factors.
- Maybe factors are small.
- Maybe factors are large.
- Maybe there are many inputs.
- Maybe inputs in superposition.

Important variations in metrics (even assuming perfect devices):
- Qubits.
- Area ("A", including wire area).
- Qubit operations ("gates").
- Depth.
- Time ("T": latency).  

Short-term RSA security
Important variations in the factorization problem:
• Maybe need one factor.
• Maybe need all factors.
• Maybe factors are small.
• Maybe factors are large.
• Maybe there are many inputs.
• Maybe inputs in superposition.

Important variations in metrics (even assuming perfect devices):
• Qubits.
• Area (“A”, including wire area).
• Qubit operations (“gates”).
• Depth.
• Time (“T”: latency).

Short-term RSA security
Important variations in the factorization problem:
• Maybe need one factor.
• Maybe need all factors.
• Maybe factors are small.
• Maybe factors are large.
• Maybe there are many inputs.
• Maybe inputs in superposition.

Important variations in metrics (even assuming perfect devices):
• Qubits.
• Area (“A”, including wire area).
• Qubit operations (“gates”).
• Depth.
• Time (“T”: latency).

Short-term RSA security

2003 Beauregard: $2b + 3$ qubits.
::: 2016 Häner–Roetteler–Svore: $2b + 2$ qubits; $64^b (\lg b + O(1))$ Toffoli gates; similar number of CNOT gates; depth $O(b^3)$. 
Important variations in the factorization problem:
• Maybe need one factor.
• Maybe need all factors.
• Maybe factors are small.
• Maybe factors are large.
• Maybe there are many inputs.
• Maybe inputs in superposition.

Important variations in metrics (even assuming perfect devices):
• Qubits.
• Area (“A”, including wire area).
• Qubit operations (“gates”).
• Depth.
• Time (“T”: latency).

Short-term RSA security

2003 Beauregard: 2^b + 3 qubits.
::: 2016 Häner–Roetteler–Svore: 2^b + 2 qubits; 64^b^3 (lg b + O(1)) Toffoli gates; similar number of CNOT gates; depth O(b^3).
Important variations in the factorization problem:
• Maybe need one factor.
• Maybe need all factors.
• Maybe factors are small.
• Maybe factors are large.
• Maybe there are many inputs.
• Maybe inputs in superposition.

Important variations in metrics (even assuming perfect devices):
• Qubits.
• Area ("A", including wire area).
• Qubit operations ("gates").
• Depth.
• Time ("T": latency).

Short-term RSA security

2003 Beauregard: $2b + 3$ qubits.
… 2016 Häner–Roetteler–Svore: $2b + 2$ qubits; $64b^3(lg b + O(1))$ Toffoli gates; similar number of CNOT gates; depth $O(b^3)$. 
Short-term RSA security


2003 Beauregard: $2b + 3$ qubits.

... 2016 Häner–Roetteler–Svore: $2b + 2$ qubits; $64b^3(\lg b + O(1))$ Toffoli gates; similar number of CNOT gates; depth $O(b^3)$. 
Short-term RSA security


2003 Beauregard: $2b + 3$ qubits. . . 2016 Häner–Roetteler–Svore: $2b + 2$ qubits; $64b^3(lg b + O(1))$ Toffoli gates; similar number of CNOT gates; depth $O(b^3)$.

Conventional wisdom: cannot avoid $2b$ qubits for controlled mulmod.

e.g. 4096 qubits for $b = 2048$, very common RSA key size.

So 2048-bit factorization needs 4096 qubits?
Short-term RSA security


2003 Beauregard: $2b + 3$ qubits.

... 2016 Häner–Roetteler–Svore: $2b + 2$ qubits; $64b^3(lg b + O(1))$ Toffoli gates; similar number of CNOT gates; depth $O(b^3)$.

Conventional wisdom:

cannot avoid $2b$ qubits for controlled mulmod.

e.g. 4096 qubits for $b = 2048$, very common RSA key size.

So 2048-bit factorization needs 4096 qubits?

No: NFS uses 0 qubits.
Short-term RSA security


2003 Beauregard: $2b + 3$ qubits.

... 2016 Hänner–Roetteler–Svore: $2b + 2$ qubits; $64b^3(\lg b + O(1))$ Toffoli gates; similar number of CNOT gates; depth $O(b^3)$.

Conventional wisdom: cannot avoid $2b$ qubits for controlled mulmod.

e.g. 4096 qubits for $b = 2048$, very common RSA key size.

So 2048-bit factorization needs 4096 qubits?
No: NFS uses 0 qubits.

NFS takes $L^p + o(1)$ operations with $p = 3\chi + 26\sqrt{13}/3 
\log L = \log 2^b = 3\log\log 2^b$.

Analysis for $b = 2048$ (not easy!): very roughly $2^{112}$ operations.
Short-term RSA security


2003 Beauregard: $2b + 3$ qubits.

... 2016 Häner–Roetteler–Svore: $2b + 2$ qubits; $64b^3(\lg b + O(1))$ Toffoli gates; similar number of CNOT gates; depth $O(b^3)$.

Conventional wisdom:
cannot avoid $2b$ qubits for controlled mulmod.

e.g. 4096 qubits for $b = 2048$, very common RSA key size.

So 2048-bit factorization needs 4096 qubits?
No: NFS uses 0 qubits.

NFS takes $L^{p+o(1)}$ operations with $p = \sqrt[3]{92 + 26\sqrt{13}} = 3 > 1$:

$$\log L = (\log 2^b)^{1/3} = 3(\log \log 2^b)^{2/3} = 3.$$

Analysis for $b = 2048$ (not easy!):
very roughly $2^{112}$ operations.
2003 Beauregard: $2b + 3$ qubits.

... 2016 Häner–Roetteler–Svore: $2b + 2$ qubits; $64b^3(\lg b + O(1))$
Toffoli gates; similar number of
CNOT gates; depth $O(b^3)$.

Conventional wisdom:
cannot avoid $2b$ qubits
for controlled mulmod.

e.g. 4096 qubits for $b = 2048$,
very common RSA key size.

So 2048-bit factorization
needs 4096 qubits?
No: NFS uses 0 qubits.

NFS takes $L^{p+o(1)}$ operations
with $p = \sqrt[3]{92 + 26\sqrt{13}/3} > 3$,
$log L = (\log 2^b)^{1/3}(\log \log 2^b)^{2/3}.$

Analysis for $b = 2048$ (not easy!):
very roughly $2^{112}$ operations.
2003 Beauregard: $2b + 3$ qubits.

... 2016 Häner–Roetteler–Svore: $2b + 2$ qubits; $64b^3(\lg b + O(1))$ Toffoli gates; similar number of CNOT gates; depth $O(b^3)$.

Conventional wisdom:
cannot avoid $2b$ qubits
for controlled mulmod.

e.g. 4096 qubits for $b = 2048$,
very common RSA key size.

So 2048-bit factorization
needs 4096 qubits?
No: NFS uses 0 qubits.

NFS takes $L^{p+o(1)}$ operations
with $p = \sqrt[3]{92 + 26\sqrt{13}/3} > 1.9$,
$log L = (log 2^b)^{1/3}(log log 2^b)^{2/3}$.

Analysis for $b = 2048$ (not easy!):
very roughly $2^{112}$ operations.
2003 Beauregard: $2b + 3$ qubits.

... 2016 Häner–Roetteler–Svore: $2b + 2$ qubits; $64b^3(\lg b + O(1))$
Toffoli gates; similar number of CNOT gates; depth $O(b^3)$.

Conventional wisdom: cannot avoid $2b$ qubits for controlled mulmod.

e.g. 4096 qubits for $b = 2048$, very common RSA key size.

So 2048-bit factorization needs 4096 qubits?
No: NFS uses 0 qubits.

NFS takes $L^{p+o(1)}$ operations with $p = \frac{3}{3} \frac{92 + 26\sqrt{13}}{3} > 1.9$, 
$\log L = (\log 2^b)^{1/3}(\log \log 2^b)^{2/3}$.

Analysis for $b = 2048$ (not easy!): very roughly $2^{112}$ operations.

2017 Bernstein–Biasse–Mosca: $L^{q+o(1)}$ operations with $q = \frac{3}{3} \frac{8}{3} \approx 1.387$, using $b^{2/3+o(1)}$ qubits (and many non-quantum bits).
2003 Beauregard: $2b + 3$ qubits.

\[ \ldots 2016 \text{ Häner–Roetteler–Svore:} \]

$2b + 2$ qubits; $64b^3(\lg b + O(1))$

Toffoli gates; similar number of CNOT gates; depth $O(b^3)$.

Conventional wisdom: cannot avoid $2b$ qubits for controlled mulmod.

e.g. 4096 qubits for $b = 2048$,

very common RSA key size.

So 2048-bit factorization needs 4096 qubits?

No: NFS uses 0 qubits.

NFS takes $L^{p+o(1)}$ operations with $p = 3\sqrt{92 + 26\sqrt{13}}/3 > 1.9$,

$log L = (\log 2^b)^{1/3}(\log \log 2^b)^{2/3}$.

Analysis for $b = 2048$ (not easy!):

very roughly $2^{112}$ operations.

2017 Bernstein–Biasse–Mosca: $L^{q+o(1)}$ operations with $q = 3\sqrt{8/3} \approx 1.387$,

using $b^{2/3+o(1)}$ qubits (and many non-quantum bits).

Open: Analyze for $b = 2048$.

Fewer than 4096 qubits?

Fewer than 2048 qubits?
7

Häner–Roetteler–Svore: 2\text{b} + 2 qubits; 64\text{b}^3(\lg b + O(1))
gates; similar number of gates; depth $O(b^3)$.

Conventional wisdom:
avoid $2\text{b}$ qubits
rolled mulmod.

6 qubits for $b = 2048$,
common RSA key size.

2017 Bernstein–Biasse–Mosca:
$L^{q+o(1)}$ operations
with $q = \sqrt[3]{8/3} \approx 1.387$,
using $b^{2/3+o(1)}$ qubits
(and many non-quantum bits).

Open: Analyze for $b = 2048$.
Fewer than 4096 qubits?
Fewer than 2048 qubits?

8

NFS takes $L^{p+o(1)}$ operations
with $p = \sqrt[3]{92 + 26\sqrt{13}/3} > 1.9$,
$\log L = (\log 2^b)^{1/3}(\log \log 2^b)^{2/3}$.

Analysis for $b = 2048$ (not easy!):
very roughly $2^{112}$ operations.

Counting operations is an
oversimplified cost model: ignores
communication costs, parallelism.
See, e.g., 1981 Brent–Kung AT
theorem for realistic chip model.
NFS takes $L^{p+o(1)}$ operations with $p = \sqrt[3]{92 + 26\sqrt{13}/3} > 1.9$, $\log L = (\log 2^b)^{1/3}(\log \log 2^b)^{2/3}$.

Analysis for $b = 2048$ (not easy!): very roughly $2^{112}$ operations.

2017 Bernstein–Biasse–Mosca: $L^{q+o(1)}$ operations with $q = \sqrt[3]{8/3} \approx 1.387$, using $b^{2/3+o(1)}$ qubits (and many non-quantum bits).

Open: Analyze for $b = 2048$.
Fewer than 4096 qubits?
Fewer than 2048 qubits?

Counting operations is an oversimplified cost model: ignores communication costs.
See, e.g., 1981 Brent–Kung AT theorem for realistic chip model.
NFS takes $L^{p+o(1)}$ operations
with $p = \frac{3}{2} \sqrt{92 + 26\sqrt{13}}/3 > 1.9$, 
$\log L = (\log 2^b)^{1/3}(\log \log 2^b)^{2/3}$.

Analysis for $b = 2048$ (not easy!):
very roughly $2^{112}$ operations.

2017 Bernstein–Biasse–Mosca:
$L^{q+o(1)}$ operations
with $q = \frac{3}{2} \sqrt{8}/3 \approx 1.387$, 
using $b^{2/3+o(1)}$ qubits
(and many non-quantum bits).

Open: Analyze for $b = 2048$.
Fewer than 4096 qubits?
Fewer than 2048 qubits?

Counting operations is an
oversimplified cost model: ignores communication costs, parallelism.
See, e.g., 1981 Brent–Kung AT theorem for realistic chip model.

2003 Beauregard: 2 qubits.
::: 2016 H"aner–Roetteler–Svore: 
2 b + 2 qubits; 64 b
$3 (\log b + O(1))$
Toffoli gates; similar number of 
CNOT gates; depth $O(b^3)$.

Conventional wisdom:
cannot avoid 2b qubits
for controlled mulmod.
e.g. 4096 qubits for 
b = 2048, 
very common RSA key size.
So 2048-bit factorization
needs 4096 qubits?
No: NFS uses 0 qubits.

Counting operations is an
oversimplified cost model: ignores communication costs, parallelism.
See, e.g., 1981 Brent–Kung AT theorem for realistic chip model.
NFS takes $L^{p+o(1)}$ operations with $p = 3^{\sqrt{92 + 26\sqrt{13}}/3} > 1.9$, 
\[ \log L = (\log 2^b)^{1/3}(\log \log 2^b)^{2/3}. \]

Analysis for $b = 2048$ (not easy!): 
very roughly $2^{112}$ operations.

2017 Bernstein–Biasse–Mosca: 
$L^{q+o(1)}$ operations 
with $q = 3^{\sqrt{8/3}} \approx 1.387$, 
using $b^{2/3+o(1)}$ qubits 
(and many non-quantum bits).

Open: Analyze for $b = 2048$. 
Fewer than 4096 qubits? 
Fewer than 2048 qubits?

Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ theorem for realistic chip model.
NFS takes $L^{p+o(1)}$ operations with $p = \frac{3\sqrt{92 + 26\sqrt{13}}}{3} > 1.9$, 
$log L = \left(\log 2^b\right)^{1/3}(\log \log 2^b)^{2/3}$.

Analysis for $b = 2048$ (not easy!): very roughly $2^{112}$ operations.

2017 Bernstein–Biasse–Mosca: 
$L^{q+o(1)}$ operations with $q = \frac{3\sqrt{8/3}}{3} \approx 1.387$, using $b^{2/3+o(1)}$ qubits (and many non-quantum bits).

Open: Analyze for $b = 2048$. Fewer than 4096 qubits? Fewer than 2048 qubits?

Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung AT theorem for realistic chip model.

NFS suffers somewhat from communication costs inside big linear-algebra subroutine.

2001 Bernstein: 
$AT = L^{p'+o(1)}$ with $p' \approx 1.976$.

2017 Bernstein–Biasse–Mosca: 
$AT = L^{q'+o(1)}$ with $q' \approx 1.456$ using $b^{2/3+o(1)}$ qubits.

Open: Analyze for $b = 2048$. 
NFS takes $L^{p + o(1)}$ operations

$$= \sqrt[3]{92 + 26\sqrt{13}}/3 > 1.9,$$

$(\log 2^b)^{1/3} (\log \log 2^b)^{2/3}$. 

for $b = 2048$ (not easy!): roughly $2^{112}$ operations.

Bernstein–Biasse–Mosca:

operations

$$= \sqrt[3]{8/3} \approx 1.387,$$

$3^{1/3} + o(1)$ qubits

using $b^{2/3} + o(1)$ qubits.

Actually have many inputs.

Lower cost for some output?

Lower cost for many outputs?

Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ theorem for realistic chip model.

NFS suffers somewhat from communication costs inside big linear-algebra subroutine.

2001 Bernstein:

$AT = L^{p'} + o(1)$ with $p' \approx 1.976$. 

2017 Bernstein–Biasse–Mosca:

$AT = L^{q'} + o(1)$ with $q' \approx 1.456$ using $b^{2/3} + o(1)$ qubits.

Open: Analyze for $b = 2048$. 

Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ theorem for realistic chip model.

NFS suffers somewhat from communication costs inside big linear-algebra subroutine.

2001 Bernstein: $AT = L^{p'} + o(1)$ with $p' \approx 1.976$. 

2017 Bernstein–Biasse–Mosca: $AT = L^{q'} + o(1)$ with $q' \approx 1.456$ using $b^{2/3} + o(1)$ qubits. Open: Analyze for $b = 2048$. 

Actually have many inputs. Lower cost for some output? Lower cost for many outputs?
Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ theorem for realistic chip model.

NFS suffers somewhat from communication costs inside big linear-algebra subroutine.

2001 Bernstein:
$$AT = L^{p'+o(1)}$$ with $p' \approx 1.976$.

2017 Bernstein–Biasse–Mosca:  
$$AT = L^{q'+o(1)}$$ with $q' \approx 1.456$ using $b^{2/3+o(1)}$ qubits.

Open: Analyze for $b = 2048$. 

Actually have many inputs. Lower cost for some output? Lower cost for many outputs?
Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ theorem for realistic chip model.

NFS suffers somewhat from communication costs inside big linear-algebra subroutine.

2001 Bernstein:

\[ AT = L^{p'} + o(1) \] with $p' \approx 1.976$. 

2017 Bernstein–Biasse–Mosca:

\[ AT = L^{q'} + o(1) \] with $q' \approx 1.456$

using $b^{2/3 + o(1)}$ qubits.

Open: Analyze for $b = 2048$.  

Actually have many inputs. Lower cost for some output? Lower cost for many outputs?
Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ theorem for realistic chip model.

NFS suffers somewhat from communication costs inside big linear-algebra subroutine.

2001 Bernstein:

$$AT = L^{p'} + o(1) \text{ with } p' \approx 1.976.$$ 

2017 Bernstein–Biasse–Mosca:

$$AT = L^{q'} + o(1) \text{ with } q' \approx 1.456$$

using $b^{2/3 + o(1)}$ qubits.

Open: Analyze for $b = 2048$.

Actually have many inputs.
Lower cost for some output?
Lower cost for many outputs?

1993 Coppersmith:

$L^{1.638...+o(1)}$ operations after precomp($b$) involving $L^{2.006...+o(1)}$ operations.
Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ theorem for realistic chip model.

NFS suffers somewhat from communication costs inside big linear-algebra subroutine.

2001 Bernstein:
$AT = L^{p'} + o(1)$ with $p' \approx 1.976$.

2017 Bernstein–Biasse–Mosca:
$AT = L^{q'} + o(1)$ with $q' \approx 1.456$ using $b^{2/3} + o(1)$ qubits.

Open: Analyze for $b = 2048$.

Actually have many inputs. Lower cost for some output? Lower cost for many outputs?

1993 Coppersmith:
$L^{1.638\ldots + o(1)}$ operations after precomp$(b)$ involving $L^{2.006\ldots + o(1)}$ operations.

2014 Bernstein–Lange:
$AT = L^{2.204\ldots + o(1)}$ to factor $L^{0.5 + o(1)}$ inputs; $L^{1.704\ldots + o(1)}$ per input.
Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ theorem for realistic chip model.

NFS suffers somewhat from communication costs inside big linear-algebra subroutine.

2001 Bernstein:

$AT = L^{p'} + o(1)$ with $p' \approx 1.976$.

2017 Bernstein–Biasse–Mosca:

$AT = L^{q'} + o(1)$ with $q' \approx 1.456$ using $b^{2/3} + o(1)$ qubits.

Open: Analyze for $b = 2048$.

Actually have many inputs. Lower cost for some output? Lower cost for many outputs?

1993 Coppersmith:

$L^{1.638...+o(1)}$ operations after precomp($b$) involving $L^{2.006...+o(1)}$ operations.

2014 Bernstein–Lange:

$AT = L^{2.204...+o(1)}$ to factor $L^{0.5+o(1)}$ inputs; $L^{1.704...+o(1)}$ per input.

Open: Any quantum speedups for factoring many integers?
Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ for realistic chip model. NFS suffers somewhat from communication costs inside linear-algebra subroutine.

1993 Coppersmith: $L^{1.638\ldots+o(1)}$ operations after precomp$(b)$ involving $L^{2.006\ldots+o(1)}$ operations.

2014 Bernstein–Lange: $AT = L^{2.204\ldots+o(1)}$ to factor $L^{0.5+o(1)}$ inputs; $L^{1.704\ldots+o(1)}$ per input.

Open: Any quantum speedups for factoring many integers?

Actually have many inputs. Lower cost for some output? Lower cost for many outputs?

You actually have many inputs. Lower cost for some output? Lower cost for many outputs?

Long-term RSA security

Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.

Open: Any quantum speedups for factoring many integers?
Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung $AT$ theorem for realistic chip model.

Actually have many inputs. Lower cost for some output? Lower cost for many outputs?

1993 Coppersmith: $L^{1.638...+o(1)}$ operations after precomp($b$) involving $L^{2.006...+o(1)}$ operations.

2014 Bernstein–Lange: $AT = L^{2.204...+o(1)}$ to factor $L^{0.5+o(1)}$ inputs; $L^{1.704...+o(1)}$ per input.

Open: Any quantum speedups for factoring many integers?

Long-term RSA security

Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.
Counting operations is an oversimplified cost model: ignores communication costs, parallelism. See, e.g., 1981 Brent–Kung AT theorem for realistic chip model.

NFS suffers somewhat from communication costs inside big linear-algebra subroutine.

2001 Bernstein:
\[ AT = L^{p'} + o(1) \]
with \( p' \approx 1 \):

2017 Bernstein–Biasse–Mosca:
\[ AT = L^{q'} + o(1) \]
with \( q' \approx 1 \):

using \( b^2 = 3 + o(1) \) qubits.

Open: Analyze for \( b = 2048 \).

Actually have many inputs. Lower cost for some output? Lower cost for many outputs?

1993 Coppersmith:
\[ L^{1.638...+o(1)} \] operations
after precomp \( (b) \) involving
\[ L^{2.006...+o(1)} \] operations.

2014 Bernstein–Lange:
\[ AT = L^{2.204...+o(1)} \]
to factor \( L^{0.5+o(1)} \) inputs;
\[ L^{1.704...+o(1)} \] per input.

Open: Any quantum speedups for factoring many integers?

Long-term RSA security
Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.
Actually have many inputs.
Lower cost for *some* output?
Lower cost for *many* outputs?

1993 Coppersmith:
$L^{1.638...+o(1)}$ operations
after precomp($b$) involving
$L^{2.006...+o(1)}$ operations.

2014 Bernstein–Lange:
$AT = L^{2.204...+o(1)}$
to factor $L^{0.5+o(1)}$ inputs;
$L^{1.704...+o(1)}$ per input.

Open: Any quantum speedups
for factoring many integers?

---

Long-term RSA security

Long history of advances
in integer factorization.

Long history of RSA users
switching to larger key sizes,
not far beyond broken sizes.
Actually have many inputs.
Lower cost for *some* output?
Lower cost for *many* outputs?

1993 Coppersmith:
\[ L^{1.638\ldots + o(1)} \] operations
after precomp(\(b\)) involving
\[ L^{2.006\ldots + o(1)} \] operations.

2014 Bernstein–Lange:
\[ AT = L^{2.204\ldots + o(1)} \]
to factor \( L^{0.5 + o(1)} \) inputs;
\( L^{1.704\ldots + o(1)} \) per input.

Open: Any quantum speedups for factoring many integers?

---

Long-term RSA security

Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.

“Expert” cryptographers:
“Obviously they won’t react to Shor’s algorithm this way! They’ll switch to codes, lattices, etc. long before quantum computers break RSA-2048! We don’t need to analyze the security of RSA-4096, RSA-8192, RSA-16384, etc..!”
Actually have many inputs.

Cost for *some* output?

Cost for *many* outputs?

1993 Coppersmith:

$$L_1: 638 \cdots + o(1)$$

operations

ecomp($b$) involving $$L_2: 006 \cdots + o(1)$$

operations.

2014 Bernstein–Lange:

$$AT = L_2: 204 \cdots + o(1)$$
to factor $$L_0: 5+ o(1)$$
inputs;

$$L_1: 704 \cdots + o(1)$$
per input.

Open: Any quantum speedups for factoring many integers?

Long-term RSA security

Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.

“Expert” cryptographers:

“Obviously they won’t react to Shor’s algorithm this way! They’ll switch to codes, lattices, etc. long before quantum computers break RSA-2048! We don’t need to analyze the security of RSA-4096, RSA-8192, RSA-16384, etc.!”

We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?
Actually have many inputs.

Lower cost for some output?

Lower cost for many outputs?

1993 Coppersmith:

\[ L_1 : 638 \ldots + o(1) \]

operations after precomp(b) involving \[ L_2 : 006 \ldots + o(1) \] operations.

2014 Bernstein–Lange:

\[ AT = L_2 : 204 \ldots + o(1) \]
to factor \[ L_0 : 5+ o(1) \] inputs;

\[ L_1 : 704 \ldots + o(1) \]
per input.

Open: Any quantum speedups for factoring many integers?

Long-term RSA security

Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.

“Expert” cryptographers:

“Obviously they won’t react to Shor’s algorithm this way! They’ll switch to codes, lattices, etc. long before quantum computers break RSA-2048! We don’t need to analyze the security of RSA-4096, RSA-8192, RSA-16384, etc.!”

We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?
Long-term RSA security

Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.

“Expert” cryptographers: “Obviously they won’t react to Shor’s algorithm this way! They’ll switch to codes, lattices, etc. long before quantum computers break RSA-2048! We don’t need to analyze the security of RSA-4096, RSA-8192, RSA-16384, etc.!”

We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?
Long-term RSA security

Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.

“Expert” cryptographers: “Obviously they won’t react to Shor’s algorithm this way! They’ll switch to codes, lattices, etc. long before quantum computers break RSA-2048! We don’t need to analyze the security of RSA-4096, RSA-8192, RSA-16384, etc.!”

We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?
Long-term RSA security

Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.

“Expert” cryptographers: “Obviously they won’t react to Shor’s algorithm this way! They’ll switch to codes, lattices, etc. long before quantum computers break RSA-2048! We don’t need to analyze the security of RSA-4096, RSA-8192, RSA-16384, etc.!”

We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?

2017 Bernstein–Heninger–Lou–Valenta “Post-quantum RSA” (pqRSA): Generated 1-terabyte RSA key; 2000000 core-hours. Shor’s algorithm: \( >2^{100} \) gates.
Long-term RSA security

Long history of advances in integer factorization.

Long history of RSA users switching to larger key sizes, not far beyond broken sizes.

“Expert” cryptographers: “Obviously they won’t react to Shor’s algorithm this way! They’ll switch to codes, lattices, etc. long before quantum computers break RSA-2048! We don’t need to analyze the security of RSA-4096, RSA-8192, RSA-16384, etc.!”

We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?

2017 Bernstein–Heninger–Lou–Valenta “Post-quantum RSA” (pqRSA): Generated 1-terabyte RSA key; 2000000 core-hours. Shor’s algorithm: \( >2^{100} \) gates.

We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?

2017 Bernstein–Heninger–Lou–Valenta “Post-quantum RSA” (pqRSA): Generated 1-terabyte RSA key; 2000000 core-hours. Shor’s algorithm: \( >2^{100} \) gates.

Long history of advances in integer factorization. Long history of RSA users switching to larger key sizes, not far beyond broken sizes. "Expert" cryptographers: "Obviously they won’t react to Shor’s algorithm this way! They’ll switch to codes, lattices, etc. long before quantum computers break RSA-2048! We don’t need to analyze the security of RSA-4096, RSA-8192, RSA-16384, etc.!

We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?


The secret primes are small: 4096 bits in terabyte key; 1024 bits in gigabyte key. Important time-saver in keygen, signing, decryption.

Is this a weakness?

ECM finds any prime $< y$ using $L^{\sqrt{2}+o(1)}$ multiplications, where $\log L = (\log \log y)^2$. Beats Shor for $\log y < (\log \log \text{modulus})^{2+o(1)}$.

Public ECM record: 274-bit factor of $7^{337}+1$. 
We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?

2017 Bernstein–Heninger–Lou–Valenta “Post-quantum RSA” (pqRSA): Generated 1-terabyte RSA key; 2000000 core-hours. Shor’s algorithm: \( >2^{100} \) gates.


The secret primes are small: 4096 bits in terabyte key; 1024 bits in gigabyte key. Important time-saver in keygen, signing, decryption.

Is this a weakness?

ECM finds any prime \(< y\) using \( L^{\sqrt{2}+o(1)} \) mulmods, where \( \log L = (\log y \log \log y) \). Beats Shor for \( \log y \) below \((\log \log \text{modulus})^{2+o(1)}\).

Public ECM record: 274-bit factor of \( 7^{337} + 1 \).
We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?


The secret primes are small: 4096 bits in terabyte key; 1024 bits in gigabyte key. Important time-saver in keygen, signing, decryption.

Is this a weakness?

ECM finds any prime $< y$ using $L^{\sqrt{2}+o(1)}$ mulmods, where $\log L = (\log y \log \log y)^{1/2}$. Beats Shor for $\log y$ below $(\log \log \text{modulus})^{2+o(1)}$.

Public ECM record: 274-bit factor of $7^{337} + 1$. 

We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?

Bernstein–Heninger–Lou–Valenta “Post-quantum RSA” (pqRSA): Generated 1-terabyte RSA key; 2000000 core-hours. Shor’s algorithm: \( > 2^{100} \) gates.


The secret primes are small:
4096 bits in terabyte key;
1024 bits in gigabyte key.
Important time-saver in keygen, signing, decryption.

Is this a weakness?

ECM finds any prime \( < y \) using \( L^{\sqrt{2} + o(1)} \) mulmods, where \( \log L = (\log \log \log y)^{1/2} \).

Beats Shor for \( \log y \) below \( (\log \log \text{modulus})^{2 + o(1)} \).

Public ECM record:
274-bit factor of \( 7^{337} + 1 \).

Analysis for \( y \approx 2^{1024} \):
\( > 2^{125} \) mulmods, huge depth; \( 2^{33} \)-bit mulmod is slow.

\( 2^{23} \) target primes, but finding just one isn’t enough.
We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?

Bernstein–Heninger–Lou–Valenta “Post-quantum RSA” (pqRSA): Generated 1-terabyte RSA key; 2000000 core-hours.

Shor’s algorithm: $>2^{100}$ gates.


The secret primes are small: 4096 bits in terabyte key; 1024 bits in gigabyte key. Important time-saver in keygen, signing, decryption.

Is this a weakness?

ECM finds any prime $\leq y$ using $L^{\sqrt{2} + o(1)}$ mulmods, where $\log L = (\log y \log \log y)^{1/2}$. Beats Shor for $\log y$ below $(\log \log \text{modulus})^{2 + o(1)}$.

Public ECM record: 274-bit factor of $7^{337} + 1$.

Analysis for $y \approx 2^{1024}$: $>2^{125}$ mulmods, huge depth; and $2^{33}$-bit mulmod is slow.

$2^{23}$ target primes, but finding just one isn’t enough.
We consider possible impact of quantum computers. Shouldn’t we also consider possible impact of users wanting to stick to RSA?


Shor’s algorithm: $2^{100}$ gates.


The secret primes are small: 4096 bits in terabyte key; 1024 bits in gigabyte key. Important time-saver in keygen, signing, decryption.

Is this a weakness?

ECM finds any prime $<y$ using $L^{\sqrt{2+o(1)}}$ mulmods, where $\log L = (\log y \log \log y)^{1/2}$.

Beats Shor for $\log y$ below $(\log \log \text{modulus})^{2+o(1)}$.

Public ECM record: 274-bit factor of $7^{337} + 1$.

Analysis for $y \approx 2^{1024}$: $>2^{125}$ mulmods, huge depth; and $2^{33}$-bit mulmod is slow. $2^{23}$ target primes, but finding just one isn’t enough.
The secret primes are small: 4096 bits in terabyte key; 1024 bits in gigabyte key. Important time-saver in keygen, signing, decryption. Is this a weakness?

ECM finds any prime \( < y \) using \( L^{\sqrt{2} + o(1)} \) mulmods, where \( \log L = (\log y \log \log y)^{1/2} \).
Beats Shor for \( \log y \) below \( (\log \log \text{modulus})^{2 + o(1)} \).

Public ECM record:
274-bit factor of \( 7^{337} + 1 \).

Analysis for \( y \approx 2^{1024} \):
\( > 2^{125} \) mulmods, huge depth; and \( 2^{33} \)-bit mulmod is slow.
2\(^{23} \) target primes, but finding just one isn’t enough.
The secret primes are small:
4096 bits in terabyte key;
1024 bits in gigabyte key.
Important time-saver in
keygen, signing, decryption.

Is this a weakness?

ECM finds any prime $< y$
using $L^{\sqrt{2}+o(1)}$ mulmods,
where $\log L = (\log y \log \log y)^{1/2}$.
Beats Shor for $\log y$ below
$(\log \log \text{modulus})^{2+o(1)}$.

Public ECM record:
274-bit factor of $7^{337} + 1$.

Analysis for $y \approx 2^{1024}$:
$> 2^{125}$ mulmods, huge depth;
and $2^{33}$-bit mulmod is slow.

$2^{23}$ target primes, but
finding just one isn’t enough.

2017 Bernstein–Heninger–Lou–
Valenta: Grover+ECM
finds any prime $< y$
using $L^{1+o(1)}$ mulmods.
The secret primes are small:
4096 bits in terabyte key;
1024 bits in gigabyte key.
Important time-saver in keygen, signing, decryption.

Is this a weakness?

ECM finds any prime \( y \) using
\[
L \sqrt{2 + o(1)} \text{ mulmods,}
\]
where \( \log L = (\log y \log \log y)^{1/2} \).
Beats Shor for \( \log y \) below
\[
(\log \log \text{modulus})^{2 + o(1)}.
\]
Public ECM record: 274-bit factor of \( 7^{337} + 1 \).

Analysis for \( y \approx 2^{1024} \):
\[
> 2^{125} \text{ mulmods, huge depth;}
\]
and \( 2^{33} \)-bit mulmod is slow.

\( 2^{23} \) target primes, but finding just one isn’t enough.

2017 Bernstein–Heninger–Lou–Valenta: Grover+ECM finds any prime \( y \) using
\[
L^{1 + o(1)} \text{ mulmods.}
\]
Seems swamped by overhead.

Open: Better ways for quantum algorithms to find small factors?
The secret primes are small: 4096 bits in terabyte key; 1024 bits in gigabyte key. Important time-saver in signing, decryption.

Is this a weakness?

ECM finds any prime $\leq y$ using $L^{1+o(1)}$ mulmods, where $\log L = (\log y \log \log y)^{1/2}$. Beats Shor for $\log y$ below $(\log \log \text{modulus})^{2+o(1)}$.

Public ECM record: 274-bit factor of $7^{337} + 1$.

Analysis for $y \approx 2^{1024}$: $>2^{125}$ mulmods, huge depth; and $2^{33}$-bit mulmod is slow.

$2^{23}$ target primes, but finding just one isn’t enough.


Open: Better ways for quantum algorithms to find small factors?

Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key. Is a gigabyte key so difficult for Shor’s algorithm to break?
The secret primes are small:
- 4096 bits in terabyte key;
- 1024 bits in gigabyte key.

Important time-saver in keygen, signing, decryption.

Is this a weakness?

ECM finds any prime \(< y\) using
\[ L^{1+o(1)} \text{ mulmods,} \]
where
\[ \log L = (\log y \log \log y) \]
\[ = 2. \]

Beats Shor for \(\log y\) below
\((\log \log \text{modulus})^{2+o(1)}\).

Public ECM record:
274-bit factor of 7
\[ 337 + 1. \]

Analysis for \(y \approx 2^{1024}\):

\(> 2^{125}\) mulmods, huge depth;
and \(2^{33}\)-bit mulmod is slow.

223 target primes, but
finding just one isn’t enough.

2017 Bernstein–Heninger–Lou–Valenta: Grover+ECM
finds any prime \(< y\)
using \(L^{1+o(1)}\) mulmods.

Seems swamped by overhead.

Open: Better ways for quantum algorithms to find small factors?

Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key.

Is a gigabyte key so difficult for Shor’s algorithm to break?
The secret primes are small:
4096 bits in terabyte key;
1024 bits in gigabyte key.
Important time-saver in
keygen, signing, decryption.
Is this a weakness?
ECM finds any prime $< y$
using $L^{1+o(1)}$ mulmods,
where $\log L = (\log y \log \log y)^{1/2}$.
Beats Shor for $\log y$ below
$(\log \log \text{modulus})^{2+o(1)}$.
Public ECM record:
274-bit factor of $7^{337} + 1$.

Analysis for $y \approx 2^{1024}$:
$> 2^{125}$ mulmods, huge depth;
and $2^{33}$-bit mulmod is slow.
$2^{23}$ target primes, but
finding just one isn’t enough.
2017 Bernstein–Heninger–Lou–Valenta: Grover+ECM
finds any prime $< y$
using $L^{1+o(1)}$ mulmods.
Seems swamped by overhead.
Open: Better ways for quantum
algorithms to find small factors?

Minimum security level that
NIST allows for post-quantum
submissions: brute-force/Grover
search for a 128-bit AES key.
Is a gigabyte key so difficult for
Shor’s algorithm to break?
Analysis for $y \approx 2^{1024}$: $>2^{125}$ mulmods, huge depth; and $2^{33}$-bit mulmod is slow.

$2^{23}$ target primes, but finding just one isn’t enough.

2017 Bernstein–Heninger–Lou–Valenta: Grover+ECM finds any prime $<y$ using $L^{1+o(1)}$ mulmods.

Seems swamped by overhead.

Open: Better ways for quantum algorithms to find small factors?

Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key.

Is a gigabyte key so difficult for Shor’s algorithm to break?
Analysis for $y \approx 2^{1024}$:

> $2^{125}$ mulmods, huge depth; and $2^{33}$-bit mulmod is slow.

$2^{23}$ target primes, but finding just one isn’t enough.

2017 Bernstein–Heninger–Lou–Valenta: Grover+ECM finds any prime $< y$ using $L^{1+o(1)}$ mulmods.

Seems swamped by overhead.

Open: Better ways for quantum algorithms to find small factors?

Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key.

Is a gigabyte key so difficult for Shor’s algorithm to break?

$64b^3 \lg b \approx 2^{110}$ for $b = 2^{33}$.

Not totally implausible to argue that Grover’s algorithm could break AES-128 faster than this.
Analysis for $y \approx 2^{1024}$:
$>2^{125}$ mulmods, huge depth; and $2^{33}$-bit mulmod is slow.

$2^{23}$ target primes, but finding just one isn’t enough.

2017 Bernstein–Heninger–Lou–Valenta: Grover+ECM finds any prime $<y$ using $L^{1+o(1)}$ mulmods.

Seems swamped by overhead.

Open: Better ways for quantum algorithms to find small factors?

Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key.

Is a gigabyte key so difficult for Shor’s algorithm to break?

$64b^3 \lg b \approx 2^{110}$ for $b = 2^{33}$.

Not totally implausible to argue that Grover’s algorithm could break AES-128 faster than this.

But Shor’s algorithm can (with more qubits) use faster mulmods.
for $y \approx 2^{1024}$:
mulmods, huge depth; 125-bit mulmod is slow.
get primes, but just one isn’t enough.
Bernstein–Heninger–Lou–
Grover+ECM by prime $< y$
+$o(1)$ mulmods.
swamped by overhead.
Better ways for quantum
algorithms to find small factors?

Minimum security level that
NIST allows for post-quantum
submissions: brute-force/Grover
search for a 128-bit AES key.
Is a gigabyte key so difficult for
Shor’s algorithm to break?
$64b^3 \log b \approx 2^{110}$ for $b = 2^{33}$.
Not totally implausible to argue
that Grover’s algorithm could
break AES-128 faster than this.
But Shor’s algorithm can (with
more qubits) use faster mulmods.

NIST allows submissions to
assume reasonable time limits:
“Plausible values for MAXDEPTH
range from $2^{40}$ logical gates
(approximately the number of
presently envisioned quantum
computing architectures are
expected to serially perform in
a year) through $2^{64}$ logical
gates (the approximate number of gates
that current classical computing
architectures can perform serially
in a decade), to no more than $2^{96}$ logical
gates...
Analysis for $y \approx 2^{1024}$: mulmods, huge depth; and $2^{33}$-bit mulmod is slow.

23 target primes, but finding just one isn’t enough.

Bernstein–Heninger–Lou–Valenta: Grover+ECM finds any prime < $y$ using $L_1 + o(1)$ mulmods.

Seems swamped by overhead.

Open: Better ways for quantum algorithms to find small factors?

Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key.

Is a gigabyte key so difficult for Shor’s algorithm to break?

$64b^3 \log b \approx 2^{110}$ for $b = 2^{33}$.

Not totally implausible to argue that Grover’s algorithm could break AES-128 faster than this.

But Shor’s algorithm can (with more qubits) use faster mulmods.

NIST allows submissions to assume reasonable time limits:

“Plausible values for MAXDEPTH range from $2^{40}$ logical gates (the approximate number of gates presently envisioned quantum computing architectures are expected to serially perform in a year) through $2^{64}$ logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than $2^{96}$ logical gates ...”
Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key.

Is a gigabyte key so difficult for Shor’s algorithm to break?

$$64b^3 \lg b \approx 2^{110} \text{ for } b = 2^{33}.$$ 

Not totally implausible to argue that Grover’s algorithm could break AES-128 faster than this.

But Shor’s algorithm can (with more qubits) use faster mulmods.

NIST allows submissions to assume reasonable time limits:

“Plausible values for MAXDEPTH range from $$2^{40}$$ logical gates (the approximate number of gates presently envisioned quantum computing architectures are expected to serially perform in a year) through $$2^{64}$$ logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than $$2^{96}$$ logical gates …”
Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key.

Is a gigabyte key so difficult for Shor’s algorithm to break?

\[64b^3 \log b \approx 2^{110} \text{ for } b = 2^{33}.\]

Not totally implausible to argue that Grover’s algorithm could break AES-128 faster than this.

But Shor’s algorithm can (with more qubits) use faster mulmods.

NIST allows submissions to assume reasonable time limits:

“Plausible values for MAXDEPTH range from \(2^{40}\) logical gates (the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year) through \(2^{64}\) logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than \(2^{96}\) logical gates . . .”
Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key.

Is a gigabyte key so difficult for Shor's algorithm to break?

$$b \approx 2^{110} \text{ for } b = 2^{33}.$$ 

It's actually implausible to argue that Grover's algorithm could break AES-128 faster than this.

But Shor's algorithm can (with more qubits) use faster mulmods.

NIST allows submissions to assume reasonable time limits:

“Plausible values for MAXDEPTH range from $2^{40}$ logical gates (the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year) through $2^{64}$ logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than $2^{96}$ logical gates . . .”

What is the minimum time for $b$-bit integer multiplication?

Light takes time $\Omega(b^{1}) = 2^{1}$ to cross a $b^{1} = 2 \times b^{1} = 2^{33}$ chip.

1981 Brent–Kung $AT \geq$ small constant $\cdot b^{3} = 2^{10}$, even if wire latency is 0.

(Work around obstacles using faster-than-light communication through long-distance EPR pairs? Haven’t seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)
NIST allows submissions to assume reasonable time limits:

“Plausible values for MAXDEPTH range from $2^{40}$ logical gates (the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year) through $2^{64}$ logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than $2^{96}$ logical gates . . .”

What is the minimum time for $b$-bit integer multiplication?

Light takes time $\Omega(b)$ to cross a $b^{1/2} \times b^{1/2}$ chip.

1981 Brent–Kung $AT \geq$ small constant, even if wire latency is 0.

(Work around obstacles using faster-than-light communication through long-distance EPR pairs? Haven’t seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)
Minimum security level that NIST allows for post-quantum submissions: brute-force/Grover search for a 128-bit AES key.

Is a gigabyte key so difficult for Shor's algorithm to break?

\[
3 \log_2 b \approx 2^{110} \quad \text{for} \quad b = 2^{33}.
\]

Not totally implausible to argue that Grover's algorithm could break AES-128 faster than this. But Shor's algorithm can (with more qubits) use faster mulmods.

NIST allows submissions to assume reasonable time limits:

“Plausible values for MAXDEPTH range from $2^{40}$ logical gates (the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year) through $2^{64}$ logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than $2^{96}$ logical gates . . .”

What is the minimum time for $b$-bit integer multiplication?

Light takes time $\Omega(b^{1/2})$ to cross a $b^{1/2} \times b^{1/2}$ chip.

1981 Brent–Kung $AT$ theorem:

\[
AT \geq \text{small constant} \cdot b^{3/2},
\]

even if wire latency is 0.

(Work around obstacles using faster-than-light communication through long-distance EPR pairs? Haven’t seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)
NIST allows submissions to assume reasonable time limits:

“Plausible values for MAXDEPTH range from $2^{40}$ logical gates (the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year) through $2^{64}$ logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than $2^{96}$ logical gates...”

What is the minimum time for $b$-bit integer multiplication?

Light takes time $\Omega(b^{1/2})$ to cross a $b^{1/2} \times b^{1/2}$ chip.

1981 Brent–Kung $AT$ theorem: $AT \geq$ small constant $\cdot b^{3/2}$, even if wire latency is 0.

(Work around obstacles using faster-than-light communication through long-distance EPR pairs? Haven’t seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)
NIST allows submissions to assume reasonable time limits:

"Plausible values for MAXDEPTH range from $2^{40}$ logical gates (the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year) through $2^{64}$ logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than $2^{96}$ logical gates:"

What is the minimum time for $b$-bit integer multiplication?

Light takes time $\Omega(b^{1/2})$ to cross a $b^{1/2} \times b^{1/2}$ chip.

1981 Brent–Kung $AT$ theorem: $AT \geq$ small constant $\cdot b^{3/2}$, even if wire latency is 0.

(Work around obstacles using faster-than-light communication through long-distance EPR pairs? Haven’t seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)

What is the minimum time for Shor’s algorithm?

Main bottleneck: $a^k \mod N$ for $2b$-bit superposition $e$.

Traditional approach: series of controlled multiplications by $a$ and $1 \mod N$; $a^2 \mod N$ and $1 \mod N$; $a^4 \mod N$; etc.

Can multiply these in parallel, using many more qubits; but hard to parallelize initial computation of $a^2 \mod N$. 
What is the minimum time for $b$-bit integer multiplication?
Light takes time $\Omega(b^{1/2})$ to cross a $b^{1/2} \times b^{1/2}$ chip.

1981 Brent–Kung $AT$ theorem: $AT \geq$ small constant $\cdot b^{3/2}$, even if wire latency is 0.

(Work around obstacles using faster-than-light communication through long-distance EPR pairs? Haven’t seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)
NIST allows submissions to assume reasonable time limits: "Plausible values for MAXDEPTH range from $2^{40}$ logical gates (the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year) through $2^{64}$ logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than $2^{96}$ logical gates.

What is the minimum time for $b$-bit integer multiplication?

Light takes time $\Omega(b^{1/2})$ to cross a $b^{1/2} \times b^{1/2}$ chip.

1981 Brent–Kung $AT$ theorem: $AT \geq$ small constant \cdot $b^{3/2}$, even if wire latency is 0.

(Work around obstacles using faster-than-light communication through long-distance EPR pairs? Haven’t seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)

What is the minimum time for Shor’s algorithm?

Main bottleneck: $a^e \mod N$ for $2b$-bit superposition $e$.

Traditional approach: series of controlled multiplications by $a$ and $1/a \mod N$; $a^2 \mod N$ and $1/a^2 \mod N$; $a^4 \mod N$ and $1/a^4 \mod N$; etc.

Can multiply these in parallel, using many more qubits; but hard to parallelize initial computation of $a^{2^i} \mod N$. 
What is the minimum time for \( b \)-bit integer multiplication?

Light takes time \( \Omega(b^{1/2}) \) to cross a \( b^{1/2} \times b^{1/2} \) chip.

1981 Brent–Kung AT theorem: 
\[ AT \geq \text{small constant} \cdot b^{3/2}, \]
even if wire latency is 0.

(Work around obstacles using faster-than-light communication through long-distance EPR pairs? Haven’t seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)

What is the minimum time for Shor’s algorithm?

Main bottleneck: \( a^e \mod N \) for \( 2b \)-bit superposition \( e \).

Traditional approach: series of controlled multiplications by \( a \) and \( 1/a \mod N \); 
\( a^2 \mod N \) and \( 1/a^2 \mod N \); 
\( a^4 \mod N \) and \( 1/a^4 \mod N \); etc.

Can multiply these in parallel, using many more qubits; 
but hard to parallelize initial computation of \( a^{2^i} \mod N \).
What is the minimum time for integer multiplication?

Light takes time $\Omega(b^{1/2})$ to cross a $b^{1/2} \times b^{1/2}$ chip.

Brent–Kung $AT$ theorem: $AT \geq$ small constant $\cdot b^{3/2}$, even if wire latency is 0.

Round obstacles using faster-than-light communication through long-distance EPR pairs? Haven't seen plausible designs, even if reversible computation avoids FTL impossibility proofs.

What is the minimum time for Shor's algorithm?

Main bottleneck: $a^e \mod N$ for $2b$-bit superposition $e$.

Traditional approach: series of controlled multiplications by $a$ and $1/a \mod N$; $a^2 \mod N$ and $1/a^2 \mod N$; $a^4 \mod N$ and $1/a^4 \mod N$; etc.

Can multiply these in parallel, using many more qubits; but hard to parallelize initial computation of $a^{2^i} \mod N$.

Why gigabyte keys are reasonable:

Big enough to push latency beyond the $2^{64}$ limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer.

More cost to find all primes.
What is the minimum time for $b$-bit integer multiplication?

Light takes time $\Omega(b^{1/2})$ to cross a $b^{1/2} \times b^{1/2}$ chip.

AT theorem: $AT \geq \text{constant} \cdot b^{3/2}$, even if wire latency is 0.

- Obstacles using faster-than-light communication through long-distance EPR pairs?
- Haven't seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)

Why gigabyte keys are reasonable: big enough to push latency beyond the $2^{64}$ limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer. More cost to find all primes.

What is the minimum time for Shor’s algorithm?

Main bottleneck: $a^e \mod N$ for $2b$-bit superposition $e$.

Traditional approach: series of controlled multiplications by $a$ and $1/a \mod N$; $a^2 \mod N$ and $1/a^2 \mod N$; $a^4 \mod N$ and $1/a^4 \mod N$; etc.

Can multiply these in parallel, using many more qubits; but hard to parallelize initial computation of $a^{2^i} \mod N$. 

Can multiply these in parallel, using many more qubits; but hard to parallelize initial computation of $a^{2^i} \mod N$. 

What is the minimum time for $b$-bit integer multiplication?

Light takes time $\Omega(b^{1/2})$ to cross a $b^{1/2} \times b^{1/2}$ chip.

1981 Brent–Kung theorem: $AT \geq$ small constant $\cdot b^3 = 2^3$, even if wire latency is 0.

(Work around obstacles using faster-than-light communication through long-distance EPR pairs? Haven't seen plausible designs, even if reversible computation avoids FTL impossibility proofs.)

What is the minimum time for Shor's algorithm?

Main bottleneck: $a^e \mod N$ for $2b$-bit superposition $e$.

Traditional approach: series of controlled multiplications by $a$ and $1/a \mod N$; $a^2 \mod N$ and $1/a^2 \mod N$; $a^4 \mod N$ and $1/a^4 \mod N$; etc.

Can multiply these in parallel, using many more qubits; but hard to parallelize initial computation of $a^{2^i} \mod N$.

Why gigabyte keys are reasonable: big enough to push latency beyond the $2^{64}$ limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer. More cost to find all primes.)
What is the minimum time for Shor’s algorithm?

Main bottleneck: \( a^e \mod N \) for \( 2^b \)-bit superposition \( e \).

Traditional approach: series of controlled multiplications by \( a \) and \( 1/a \mod N \); 
\( a^2 \mod N \) and \( 1/a^2 \mod N \);  
\( a^4 \mod N \) and \( 1/a^4 \mod N \); etc.

Can multiply these in parallel, using many more qubits;  
but hard to parallelize initial computation of \( a^{2^i} \mod N \).

Why gigabyte keys are reasonable:  
big enough to push latency beyond the \( 2^{64} \) limit,  
under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs.  
These algorithms will take billions of times longer.  
More cost to find all primes.
What is the minimum time for Shor’s algorithm?

Main bottleneck: \(a^e \mod N\) for 2\(b\)-bit superposition \(e\).

Traditional approach: series of controlled multiplications by \(a\) and \(1/a \mod N\);
\(a^2 \mod N\) and \(1/a^2 \mod N\);
\(a^4 \mod N\) and \(1/a^4 \mod N\); etc.

Can multiply these in parallel, using many more qubits;
but hard to parallelize initial computation of \(a^{2^i} \mod N\).

Why gigabyte keys are reasonable:
big enough to push latency beyond the \(2^{64}\) limit,
under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs.
These algorithms will take billions of times longer.
More cost to find all primes.

Open: What is minimum time for integer factorization?
What is the minimum time for Shor’s algorithm?

Main bottleneck: \( a^e \mod N \) for 2-bit superposition \( e \).

Traditional approach: series of controlled multiplications by \( a \) and 1
\( a \mod N; \)
\( N \) and \( 1/a^2 \mod N; \)
\( N \) and \( 1/a^4 \mod N; \) etc.

Multiply these in parallel, using many more qubits;
but hard to parallelize initial computation of \( a^{2^i} \mod N \).

Gigabyte inputs are millions of times larger than 2048-bit inputs.
These algorithms will take billions of times longer.
More cost to find all primes.

Open: What is minimum time for integer factorization?

Why gigabyte keys are reasonable: big enough to push latency beyond the \( 2^{64} \) limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs.

NIST’s middle security level is defined by an AES-192 key.
What is the minimum time for Shor’s algorithm?

Main bottleneck: $a^e \mod N$ for $2^b$-bit superposition $e$.

Traditional approach: series of controlled multiplications by
   
   $a^2 \mod N$;
   $a^4 \mod N$; etc.

   Can multiply these in parallel, using many more qubits;
   hard to parallelize initial computation of $a^{2^i} \mod N$.

   Why gigabyte keys are reasonable:
   big enough to push latency beyond the $2^{64}$ limit,
   under reasonable assumptions.

   Gigabyte inputs are millions of times larger than 2048-bit inputs.

   These algorithms will take billions of times longer.

   More cost to find all primes.

   Open: What is minimum time for integer factorization?

NIST’s middle security level is defined by an AES-192 key.
What is the minimum time for Shor's algorithm?

Main bottleneck: \(a \mod N\) for 2 \(b\)-bit superposition \(e\).

Traditional approach: series of controlled multiplications by \(a = a \mod N\); \(a \mod N\) and \(1 = a \mod N\); \(a \mod N\); etc.

Can multiply these in parallel, using many more qubits; but hard to parallelize initial computation of \(a^2 \mod N\).

Why gigabyte keys are reasonable: big enough to push latency beyond the \(2^{64}\) limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer. More cost to find all primes.

Open: What is minimum time for integer factorization?

NIST's middle security level is defined by an AES-192 key.
Why gigabyte keys are reasonable: big enough to push latency beyond the $2^{64}$ limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer. More cost to find all primes.

Open: What is minimum time for integer factorization?

NIST’s middle security level is defined by an AES-192 key.
Why gigabyte keys are reasonable: big enough to push latency beyond the $2^{64}$ limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer. More cost to find all primes.

Open: What is minimum time for integer factorization?

NIST’s middle security level is defined by an AES-192 key. With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores.
Why gigabyte keys are reasonable:
big enough to push latency beyond the $2^{64}$ limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer. More cost to find all primes.

Open: What is minimum time for integer factorization?

NIST’s middle security level is defined by an AES-192 key.

With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores.

This is nonsense! There is not enough time to broadcast the input to $2^{144}$ parallel computations, and not enough time to collect the results.
Why gigabyte keys are reasonable: big enough to push latency beyond the $2^{64}$ limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer. More cost to find all primes.

Open: What is minimum time for integer factorization?

NIST’s middle security level is defined by an AES-192 key.

With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores.

This is nonsense! There is not enough time to broadcast the input to $2^{144}$ parallel computations, and not enough time to collect the results.

Is NIST implicitly assuming a higher latency limit?
Why gigabyte keys are reasonable:
big enough to push latency beyond the $2^{64}$ limit,
under reasonable assumptions.
Gigabyte inputs are millions of times larger
than 2048-bit inputs.
These algorithms will take billions of times longer.
More cost to find all primes.

Open: What is minimum time for integer factorization?

NIST’s middle security level is defined by an AES-192 key.

With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores.

This is nonsense! There is not enough time to broadcast the input to $2^{144}$ parallel computations, and not enough time to collect the results.

Is NIST implicitly assuming a higher latency limit?

Some improvements to Shor (2017 Bernstein–Biasse–Mosca)

Consider Shor’s algorithm factoring $N = p_1^{e_1} \cdots p_f^{e_f}$. Write $(p_j - 1)p_j^{-1}$ as $2^{t_j}u_j$ with $u_j$ odd.

Unit group is isomorphic to $\mathbb{Z} = 2^{t_1} \times \cdots \times \mathbb{Z} = 2^{t_f} \times \mathbb{Z} = u_1 \times \cdots$. 
Why gigabyte keys are reasonable: big enough to push latency beyond the $2^{64}$ limit, under reasonable assumptions.

Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer. More cost to find all primes.

Open: What is minimum time for integer factorization?

NIST’s middle security level is defined by an AES-192 key. With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores.

This is nonsense! There is not enough time to broadcast the input to $2^{144}$ parallel computations, and not enough time to collect the results.

Is NIST implicitly assuming a higher latency limit?

Some improvements to Shor (2017 Bernstein–Biasse–Mosca)

Consider Shor’s algorithm factoring $N = p_1^{e_1} \cdots p_f^{e_f}$ as $(p_j - 1)p_j^{e_j - 1}$ as $2^t$.

Unit group is isomorphic to $\mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f}$. 

NIST’s middle security level is defined by an AES-192 key. With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores. This is nonsense! There is not enough time to broadcast the input to $2^{144}$ parallel computations, and not enough time to collect the results. Is NIST implicitly assuming a higher latency limit?

Some improvements to Shor (2017 Bernstein–Biasse–Mosca)

Consider Shor’s algorithm factoring $N = p_1^{e_1} \cdots p_f^{e_f}$. Write $(p_j - 1)p_j^{e_j - 1}$ as $2^{t_j}u_j$ with $u_j$ odd.

Unit group is isomorphic to $\mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f} \times \mathbb{Z}/u_1 \cdots \mathbb{Z}/u_f$. 

Why gigabyte keys are reasonable: big enough to push latency beyond the $2^{64}$ limit, under reasonable assumptions. Gigabyte inputs are millions of times larger than 2048-bit inputs. These algorithms will take billions of times longer. More cost to find all primes. Open: What is minimum time for integer factorization?
NIST’s middle security level is defined by an AES-192 key. With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores.

This is nonsense! There is not enough time to broadcast the input to $2^{144}$ parallel computations, and not enough time to collect the results.

Is NIST implicitly assuming a higher latency limit?

Some improvements to Shor (2017 Bernstein–Biasse–Mosca)

Consider Shor’s algorithm factoring $N = p_1^{e_1} \cdots p_f^{e_f}$. Write $(p_j - 1)^{e_j - 1}$ as $2^{t_j} u_j$ with $u_j$ odd.

Unit group is isomorphic to $\mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f} \times \mathbb{Z}/u_1 \times \cdots$. 
NIST’s middle security level is defined by an AES-192 key.

With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores.

This is nonsense! There is not enough time to broadcast the input to $2^{144}$ parallel computations, and not enough time to collect the results.

Is NIST implicitly assuming a higher latency limit?

Some improvements to Shor (2017 Bernstein–Biasse–Mosca)

Consider Shor’s algorithm factoring $N = p_1^{e_1} \cdots p_f^{e_f}$. Write $(p_j - 1)p_j^{e_j - 1}$ as $2^{t_j}u_j$ with $u_j$ odd.

Unit group is isomorphic to $\mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f} \times \mathbb{Z}/u_1 \times \cdots$.

Shor’s algorithm (hopefully) computes order $r$ of random unit. Order $2^{c_j}$ in $\mathbb{Z}/2^{t_j}$ is $2^{t_j}$ with probability $1/2$; $2^{t_j - 1}$ with probability $1/4$; etc.
NIST's middle security level is defined by an AES-192 key. With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores. This is nonsense! There is not enough time to broadcast the input to $2^{144}$ parallel computations, and not enough time to collect the results. Is NIST implicitly assuming a higher latency limit?

Some improvements to Shor (2017 Bernstein–Biasse–Mosca)

Consider Shor's algorithm factoring $N = p_1^{e_1} \cdots p_f^{e_f}$. Write $(p_j - 1)p_j^{e_j - 1}$ as $2^{t_j}u_j$ with $u_j$ odd.

Unit group is isomorphic to $\mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f} \times \mathbb{Z}/u_1 \times \cdots$.

Shor's algorithm (hopefully) computes order $r$ of random unit. Order $2^{c_j}$ in $\mathbb{Z}/2^{t_j}$ is $2^{t_j}$ with probability $1/2$; $2^{t_j - 1}$ with probability $1/4$; etc.
NIST's middle security level is defined by an AES-192 key. With maximum depth $2^{64}$, finding an AES-192 key requires $\approx 2^{144}$ cores. This is nonsense! There is not enough time to broadcast the input to $2^{144}$ parallel computations, and not enough time to collect the results. Is NIST implicitly assuming a higher latency limit?

Some improvements to Shor (2017 Bernstein–Biasse–Mosca)
Consider Shor's algorithm factoring $N = p_1^{e_1} \cdots p_f^{e_f}$. Write $(p_j - 1)p_j^{e_j - 1}$ as $2^{t_j}u_j$ with $u_j$ odd.

Unit group is isomorphic to $\mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f} \times \mathbb{Z}/u_1 \times \cdots$.

Shor's algorithm (hopefully) computes order $r$ of random unit. Order $2^{c_j}$ in $\mathbb{Z}/2^{t_j}$ is $2^{t_j}$ with probability $1/2$; $2^{t_j-1}$ with probability $1/4$; etc.
Some improvements to Shor
(2017 Bernstein–Biasse–Mosca)

Consider Shor’s algorithm factoring $N = p_1^{e_1} \cdots p_f^{e_f}$. Write

$$(p_j - 1)p_j^{e_j - 1}$$

as $2^{t_j}u_j$ with $u_j$ odd.

Unit group is isomorphic to

$$\mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f} \times \mathbb{Z}/u_1 \times \cdots .$$

Shor’s algorithm (hopefully) computes order $r$ of random unit.

Order $2^{c_j}$ in $\mathbb{Z}/2^{t_j}$ is

$2^{t_j}$ with probability $1/2$;

$2^{t_j-1}$ with probability $1/4$; etc.

Shor computes $\gcd\{N, a^{r/2} \}$. Divisible by $p_j$ exactly when

$$c_j < \max\{c_1, \ldots, c_f\}.$$ 

Factorization fails iff all $c_j$ are equal. Chance $\leq 1/2^{f-1}$. 
Some improvements to Shor
(2017 Bernstein–Biasse–Mosca)

Consider Shor’s algorithm
factoring $N = p_1^{e_1} \cdots p_f^{e_f}$. Write
$(p_j - 1)p_j^{e_j - 1}$ as $2^{t_j}u_j$ with $u_j$ odd.

Unit group is isomorphic to
$\mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f} \times \mathbb{Z}/u_1 \times \cdots$.

Shor’s algorithm (hopefully)
computes order $r$ of random unit.
Order $2^{c_j}$ in $\mathbb{Z}/2^{t_j}$ is
$2^{t_j}$ with probability $1/2$;
$2^{t_j-1}$ with probability $1/4$; etc.

Shor computes $\gcd\{N, a^{r/2} - 1\}$.
Divisible by $p_j$ exactly when
$c_j < \max\{c_1, \ldots, c_f\}$.

Factorization fails iff all $c_j$ are
equal. Chance $\leq 1/2^{f-1}$.
Some improvements to Shor  
(2017 Bernstein–Biasse–Mosca)

Consider Shor’s algorithm factoring \( N = p_1^{e_1} \cdots p_f^{e_f} \). Write \((p_j - 1)p_j^{e_j - 1}\) as \(2^t u_j\) with \(u_j\) odd.

Unit group is isomorphic to \(\mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f} \times \mathbb{Z}/u_1 \times \cdots\).

Shor’s algorithm (hopefully) computes order \(r\) of random unit. Order \(2^{c_j}\) in \(\mathbb{Z}/2^{t_j}\) is \(2^{t_j}\) with probability \(1/2\); \(2^{t_j-1}\) with probability \(1/4\); etc.

Shor computes gcd\(\{N, a^{r/2} - 1\}\). Divisible by \(p_j\) exactly when \(c_j < \max\{c_1, \ldots, c_f\}\).

Factorization fails iff all \(c_j\) are equal. Chance \(\leq 1/2^{f-1}\).

More subtle problem: Factorization is likely to split off some of the primes with maximum \(t_j\).

Can iterate Shor’s algorithm enough times to completely factor. Many full-size iterations; many more for adversarial inputs.
Some improvements to Shor (Bernstein–Biasse–Mosca)

Consider Shor's algorithm factoring \( N = p_1^{e_1} \cdots p_f^{e_f} \). Write \( p_j^{e_j - 1} \) as \( 2^{t_j} u_j \) with \( u_j \) odd.

Unit group is isomorphic to \( \mathbb{Z}/2^{t_1} \times \cdots \times \mathbb{Z}/2^{t_f} \times \mathbb{Z}/u_1 \times \cdots \).

Shor's algorithm (hopefully) computes order \( r \) of random unit.

Order \( 2^{c_j} \) in \( \mathbb{Z}/2^{t_j} \) is \( 2^{t_j} \) with probability \( 1/2 \);
\( 2^{t_j} - 1 \) with probability \( 1/4 \); etc.

Can iterate Shor's algorithm enough times to completely factor. Many full-size iterations; many more for adversarial inputs.

Shor computes \( \gcd\{ N, a^{r/2} - 1 \} \).

Divisible by \( p_j \) exactly when \( c_j < \max\{ c_1, \ldots, c_f \} \).

Factorization fails iff all \( c_j \) are equal. Chance \( \leq 1/2^{f-1} \).

More subtle problem:
Factorization is likely to split off some of the primes with maximum \( t_j \).

Better method, inspired by primality testing: compute \( \gcd \) with \( a^{r/2} = 2 + 1, 4 + 1, 8 + 1, \ldots, a^d - 1 \), with odd \( d \).

This splits \( p_j \) according to \( c_j \).
Any two primes have chance \( \geq 1/2 \) of being split.
Factors are around half size.

Much less overhead for recursion.

Also "parallel construction":
Run several times in parallel, giving several factorizations.
Then factor into coprimes.
Some improvements to Shor (2017 Bernstein–Biasse–Mosca)

Consider Shor's algorithm factoring $N = p_1^{e_1} \cdots p_f^{e_f}$. Write $p_j^t_j u_j$ with $u_j$ odd.

Unit group is isomorphic to $\mathbb{Z} = 2^{t_1} \times \cdots \times \mathbb{Z} = 2^{t_f} \times \mathbb{Z}/u_1 \times \cdots$.

Shor's algorithm (hopefully) computes the order $r$ of random unit. This is $2^{t_j}$ with probability $1/2$; $2^{t_j}-1$ with probability $1/4$; etc.

Can iterate Shor's algorithm enough times to completely factor. Many full-size iterations; many more for adversarial inputs.

Better method, inspired by primality testing: compute $\gcd\{N, a^r/2 - 1\}$. Divisible by $p_j$ exactly when $c_j < \max\{c_1, \ldots, c_f\}$.

Factorization fails iff all $c_j$ are equal. Chance $\leq 1/2^{f-1}$.

More subtle problem: Factorization is likely to split off some of the primes with maximum $t_j$.

Any two primes have chance $\geq 1/2$ of being split. Factors are around half size. Much less overhead for recursion.

Also “parallel construction”: Run several times in parallel, giving several factorizations. Then factor into coprimes.
Some improvements to Shor (2017 Bernstein–Biasse–Mosca)

Consider Shor's algorithm factoring \( N = p_1^{e_1} \cdots p_f^{e_f} \). Write \((p_j - 1)p_j^{e_j - 1}j\) as \(2^t u_j\) with \(u_j\) odd.

Unit group is isomorphic to \( \mathbb{Z} = 2^t_1 \times \cdots \times \mathbb{Z} = 2^t_f \times \mathbb{Z} = u_1 \times \cdots \).

Shor's algorithm (hopefully) computes order \( r \) of random unit.

Order \( 2^{c_j} \) in \( \mathbb{Z} = 2^{t_j} \) is \( 2^{t_j} \) with probability \( 1 = 2 \);
\( 2^{t_j} - 1 \) with probability \( 1 = 4 \); etc.

Shor computes \( \gcd\{N, a^{r/2} - 1\} \).
Divisible by \( p_j \) exactly when \( c_j < \max\{c_1, \ldots, c_f\} \).

Factorization fails iff all \( c_j \) are equal. Chance \( \leq 1/2^{f-1} \).

More subtle problem:
Factorization is likely to split off some of the primes with maximum \( t_j \).

Can iterate Shor's algorithm enough times to completely factor. Many full-size iterations; many more for adversarial inputs.

Better method, inspired by primality testing: compute gcd with \( a^{r/2} + 1, a^{r/4} + 1, a^{r/8} + 1, \ldots, a^d + 1, a^d - 1 \), with odd \( d \).

This splits \( p_j \) according to \( c_j \).

Any two primes have chance \( \geq 1/2 \) of being split.

Factors are around half size.

Much less overhead for recursion.

Also “parallel construction”: Run several times in parallel, giving several factorizations. Then factor into coprimes.
Shor computes $\gcd\{N, a^{r/2} - 1\}$. Divisible by $p_j$ exactly when $c_j < \max\{c_1, \ldots, c_f\}$.

Factorization fails iff all $c_j$ are equal. Chance $\leq 1/2^{f-1}$.

More subtle problem:
Factorization is likely to split off some of the primes with maximum $t_j$.

Can iterate Shor's algorithm enough times to completely factor. Many full-size iterations; many more for adversarial inputs.

Better method, inspired by primality testing: compute $\gcd$ with $a^{r/2} + 1$, $a^{r/4} + 1$, $a^{r/8} + 1$, $\ldots$, $a^d + 1$, $a^d - 1$, with odd $d$.

This splits $p_j$ according to $c_j$.
Any two primes have chance $\geq 1/2$ of being split.

Factors are around half size.
Much less overhead for recursion.

Also “parallel construction”:
Run several times in parallel, giving several factorizations.
Then factor into coprimes.
Computes $\gcd\{N, a^{r/2} - 1\}$.
Divisible by $p_j$ exactly when
$\gcd_{\times\{c_1, \ldots, c_f\}}$.
Factorization fails iff all $c_j$ are
equal. Chance $\leq 1/2^{f-1}$.

More subtle problem:
Factorization is likely to
split off some of the
primes with maximum $t_j$.

Can iterate Shor’s algorithm
enough times to completely
factor. Many full-size iterations;
more for adversarial inputs.

Better method, inspired by
primality testing: compute $\gcd$
with $a^{r/2} + 1$, $a^{r/4} + 1$, $a^{r/8} + 1$, 
$\ldots$, $a^d + 1$, $a^d - 1$, with odd $d$.

This splits $p_j$ according to $c_j$.
Any two primes have chance
$\geq 1/2$ of being split.
Factors are around half size.
Much less overhead for recursion.

Also “parallel construction”:
Run several times in parallel,
giving several factorizations.
Then factor into coprimes.

These methods use $>b$ qubits.
Didn’t we claim $b^2 = 3 + o(1)$ qubits?
We actually use Grover’s method
to search for smooth $b$-bit
numbers in NFS.

Oracle for Grover’s method:
factor thoroughly enough
to recognize smooth inputs.

We tweak (improved) Shor to
work in superposition. Careful
with qubit budget for continued
fractions, power detection, etc.
Shor computes gcd \( \{N, a^{r/2} - 1\} \).

Divisible by \( p^j \) exactly when \( c_j \).

Factorization fails iff all \( c_j \) are \( \leq 2^{f-1} \).

Better method, inspired by primality testing: compute gcd with \( a^{r/2} + 1, a^{r/4} + 1, a^{r/8} + 1, \ldots, a^d + 1, a^d - 1 \), with odd \( d \).

This splits \( p_j \) according to \( c_j \).

Any two primes have chance \( \geq 1/2 \) of being split.

Factors are around half size.

Much less overhead for recursion.

Also “parallel construction”: Run several times in parallel, giving several factorizations.

Then factor into coprimes.

Oracle for Grover’s method: factor thoroughly enough to recognize smooth inputs.

We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.
Better method, inspired by primality testing: compute gcd with $a^{r/2} + 1$, $a^{r/4} + 1$, $a^{r/8} + 1$, ..., $a^{d} + 1$, $a^{d} - 1$, with odd $d$.

This splits $p_j$ according to $c_j$. Any two primes have chance $\geq 1/2$ of being split.

Factors are around half size. Much less overhead for recursion.

Also “parallel construction”: Run several times in parallel, giving several factorizations. Then factor into coprimes.

These methods use $>b$ qubits. Didn’t we claim $b^{2/3+o(1)}$ qubits?

We actually use Grover’s method to search for smooth $b^{2/3+o(1)}$-bit numbers in NFS.

Oracle for Grover’s method: factor thoroughly enough to recognize smooth inputs.

We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.
Better method, inspired by primality testing: compute gcd with \( a^{r/2} + 1, a^{r/4} + 1, a^{r/8} + 1, \ldots, a^d + 1, a^d - 1 \), with odd \( d \).

This splits \( p_j \) according to \( c_j \). Any two primes have chance \( \geq 1/2 \) of being split.

Factors are around half size. Much less overhead for recursion.

Also “parallel construction”: Run several times in parallel, giving several factorizations. Then factor into coprimes.

These methods use \( > b \) qubits. Didn’t we claim \( b^{2/3 + o(1)} \) qubits?

We actually use Grover’s method to search for smooth \( b^{2/3 + o(1)} \)-bit numbers in NFS.

Oracle for Grover’s method: factor thoroughly enough to recognize smooth inputs.

We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.
Better method, inspired by primality testing: compute gcd with $a_r = 2^r + 1$, $a_r = 4^r + 1$, $a_r = 8^r + 1$, $\ldots$, $a_d - 1$, with odd $d$. This splits $p_j$ according to $c_j$. Any two primes have chance $\geq 1/2$ of being split. Factors are around half size. Much less overhead for recursion.

"Parallel construction": Run several times in parallel, giving several factorizations. Then factor into coprimes.

These methods use $\geq b$ qubits. Didn’t we claim $b^{2/3+o(1)}$ qubits? We actually use Grover’s method to search for smooth $b^{2/3+o(1)}$-bit numbers in NFS. Oracle for Grover’s method: factor thoroughly enough to recognize smooth inputs. We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.

A different way to improve randomness of factorizations in Shor’s algorithm: replace group $\mathbb{Z}/N^\ast$ with elliptic curve $E(\mathbb{Z}/N)$ for a random elliptic curve $E$. These methods use $\geq b$ qubits. Didn’t we claim $b^{2/3+o(1)}$ qubits? We actually use Grover’s method to search for smooth $b^{2/3+o(1)}$-bit numbers in NFS. Oracle for Grover’s method: factor thoroughly enough to recognize smooth inputs. We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.
Better method, inspired by primality testing: compute gcd with \( a^r = 2^r + 1, \ a^r = 4^r + 1, \ a^r = 8^r + 1, \ \ldots \), with odd \( d \). This splits \( p_j \) according to \( c_j \). Any two primes have chance \( \geq \frac{1}{2} \) of being split. Factors are around half size. Much less overhead for recursion. Also “parallel construction”: Run several times in parallel, giving several factorizations. Then factor into coprimes.

These methods use \( > b \) qubits. Didn’t we claim \( b^{2/3+o(1)} \) qubits? We actually use Grover’s method to search for smooth \( b^{2/3+o(1)} \)-bit numbers in NFS. Oracle for Grover’s method: factor thoroughly enough to recognize smooth inputs.

We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.

A different way to improve randomness of factorizations in Shor’s algorithm: replace group \( (\mathbb{Z}/N)^* \) with \( E(\mathbb{Z}/N) \) for a random elliptic curve \( E \).
Better method, inspired by primality testing: compute gcd with $a_r = 2^{r} + 1$, $a_r = 4^{r} + 1$, $a_r = 8^{r} + 1$, $\ldots$, $a_r = (d-1)$, for odd $d$.

This splits $p_j$ according to $c_j$.

Any two primes have chance $\geq \frac{1}{2}$ of being split.

Factors are around half size.

Much less overhead for recursion.

Also "parallel construction": Run several times in parallel, giving several factorizations. Then factor into coprimes.

These methods use $>b$ qubits.

Didn't we claim $b^{2/3 + o(1)}$ qubits?

We actually use Grover's method to search for smooth $b^{2/3 + o(1)}$-bit numbers in NFS.

Oracle for Grover's method: factor thoroughly enough to recognize smooth inputs.

We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.

A different way to improve randomness of factorizations in Shor's algorithm: replace group $(\mathbb{Z}/N)^*$ with $E(\mathbb{Z}/N)$ for a random elliptic curve $E$. 
These methods use $\geq b$ qubits. Didn’t we claim $b^{2/3+o(1)}$ qubits?
We actually use Grover’s method to search for smooth $b^{2/3+o(1)}$-bit numbers in NFS.

Oracle for Grover’s method: factor thoroughly enough to recognize smooth inputs.

We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.

A different way to improve randomness of factorizations in Shor’s algorithm: replace group $(\mathbb{Z}/N)^*$ with $E(\mathbb{Z}/N)$ for a random elliptic curve $E$. 
These methods use $> b$ qubits. Didn’t we claim $b^{2/3+o(1)}$ qubits? We actually use Grover’s method to search for smooth $b^{2/3+o(1)}$-bit numbers in NFS.

Oracle for Grover’s method: factor thoroughly enough to recognize smooth inputs.

We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.

A different way to improve randomness of factorizations in Shor’s algorithm: replace group $(\mathbb{Z}/N)^\ast$ with $E(\mathbb{Z}/N)$ for a random elliptic curve $E$.

Gal Dor suggests unifying Grover+ECM with Shor: e.g., compute $esP$ on $E(\mathbb{Z}/N)$ where $e$ is superposition of scalars, $s$ is smooth scalar, $E$ is superposition of curves.
These methods use $> b$ qubits. Didn’t we claim $b^{2/3+o(1)}$ qubits?

We actually use Grover’s method to search for smooth $b^{2/3+o(1)}$-bit numbers in NFS.

Oracle for Grover’s method: factor thoroughly enough to recognize smooth inputs.

We tweak (improved) Shor to work in superposition. Careful with qubit budget for continued fractions, power detection, etc.

A different way to improve randomness of factorizations in Shor’s algorithm: replace group $(\mathbb{Z}/N)^*$ with $E(\mathbb{Z}/N)$ for a random elliptic curve $E$.

Gal Dor suggests unifying Grover+ECM with Shor: e.g., compute $esP$ on $E(\mathbb{Z}/N)$ where $e$ is superposition of scalars, $s$ is smooth scalar, $E$ is superposition of curves.

Open: What are minimum costs for this unification?