Introduction to quantum algorithms and introduction to code-based cryptography

Daniel J. Bernstein
University of Illinois at Chicago & Technische Universiteit Eindhoven

Data ("state") stored in $n$ bits: an element of $\{0, 1\}^n$, often viewed as representing an element of $\{0, 1, \ldots, 2^n - 1\}$. 
Introduction to quantum algorithms and introduction to code-based cryptography

Daniel J. Bernstein
University of Illinois at Chicago & Technische Universiteit Eindhoven

Data ("state") stored in $n$ bits: an element of $\{0, 1\}^n$, often viewed as representing an element of $\{0, 1, \ldots , 2^n - 1\}$.

State stored in $n$ qubits: a nonzero element of $\mathcal{C}^{2^n}$. Retrieving this vector is tough!
Introduction to quantum algorithms and introduction to code-based cryptography

Daniel J. Bernstein
University of Illinois at Chicago & Technische Universiteit Eindhoven

Data (“state”) stored in $n$ bits: an element of $\{0, 1\}^n$, often viewed as representing an element of $\{0, 1, \ldots, 2^n - 1\}$.

State stored in $n$ qubits: a nonzero element of $\mathbb{C}^{2^n}$.

Retrieving this vector is tough!

If $n$ qubits have state $(a_0, a_1, \ldots, a_{2^n-1})$ then measuring the qubits produces an element of $\{0, 1, \ldots, 2^n - 1\}$ and destroys the state.

Measurement produces element $q$ with probability $|a_q|^2 / \sum_r |a_r|^2$. 
Data ("state") stored in \(n\) bits:
an element of \(\{0, 1\}^n\),
often viewed as representing
an element of \(\{0, 1, \ldots, 2^n - 1\}\).

State stored in \(n\) qubits:
a nonzero element of \(\mathbb{C}^{2^n}\).

Retrieving this vector is tough!

If \(n\) qubits have state
\((a_0, a_1, \ldots, a_{2^n-1})\) then

**measuring** the qubits produces
an element of \(\{0, 1, \ldots, 2^n - 1\}\)
and destroys the state.

Measurement produces element \(q\)
with probability \(|a_q|^2 / \sum_r |a_r|^2\).
Data ("state") stored in \( n \) bits:
an element of \( \{0, 1\}^n \),
often viewed as representing
an element of \( \{0, 1, \ldots, 2^n - 1\} \).

State stored in \( n \) qubits:
a nonzero element of \( \mathbb{C}^{2^n} \).
Retrieving this vector is tough!

If \( n \) qubits have state
\((a_0, a_1, \ldots, a_{2^n - 1})\) then
**measuring** the qubits produces
an element of \( \{0, 1, \ldots, 2^n - 1\} \)
and destroys the state.
Measurement produces element \( q \)
with probability \( |a_q|^2 / \sum_r |a_r|^2 \).

Some examples of 3-qubit states:
\((1, 0, 0, 0, 0, 0, 0, 0)\) is
"\(|0\rangle\)" in standard notation.
Measurement produces 0.
Data ("state") stored in $n$ bits: an element of $\{0, 1\}^n$, often viewed as representing an element of $\{0, 1, \ldots, 2^n - 1\}$.

State stored in $n$ qubits: a nonzero element of $\mathbb{C}^{2^n}$.

Retrieving this vector is tough!

If $n$ qubits have state $(a_0, a_1, \ldots, a_{2^n-1})$ then measuring the qubits produces an element of $\{0, 1, \ldots, 2^n - 1\}$ and destroys the state.

Measurement produces element $q$ with probability $|a_q|^2 / \sum_r |a_r|^2$.

Some examples of 3-qubit states:

$(1, 0, 0, 0, 0, 0, 0, 0)$ is "$|0\rangle$" in standard notation. Measurement produces 0.

$(1, 0, 0, 0, 0, 0, 0, 0)$ is "$|0\rangle$" in standard notation. Measurement produces 0.
Data ("state") stored in $n$ bits:
an element of $\{0, 1\}^n$,
often viewed as representing
an element of $\{0, 1, \ldots, 2^n - 1\}$.

State stored in $n$ qubits:
a nonzero element of $\mathbb{C}^{2^n}$.
Retrieving this vector is tough!

If $n$ qubits have state
$(a_0, a_1, \ldots, a_{2^n-1})$ then
**measuring** the qubits produces
an element of $\{0, 1, \ldots, 2^n - 1\}$
and destroys the state.

Measurement produces element $q$
with probability $|a_q|^2 / \sum_r |a_r|^2$.

Some examples of 3-qubit states:
$(1, 0, 0, 0, 0, 0, 0, 0)$ is
"$|0\rangle$" in standard notation.
Measurement produces 0.
Data (“state”) stored in $n$ bits: an element of $\{0, 1\}^n$, often viewed as representing an element of $\{0, 1, \ldots, 2^n - 1\}$.

State stored in $n$ qubits: a nonzero element of $\mathbb{C}^{2^n}$. Retrieving this vector is tough!

If $n$ qubits have state $(a_0, a_1, \ldots, a_{2^n-1})$ then **measuring** the qubits produces an element of $\{0, 1, \ldots, 2^n - 1\}$ and destroys the state.

Measurement produces element $q$ with probability $|a_q|^2 / \sum_r |a_r|^2$.

Some examples of 3-qubit states:

$(1, 0, 0, 0, 0, 0, 0, 0)$ is “$|0\rangle$” in standard notation. Measurement produces 0.

$(0, 0, 0, 0, 0, 0, 1, 0)$ is “$|6\rangle$” in standard notation. Measurement produces 6.
Data ("state") stored in $n$ bits:
an element of $\{0, 1\}^n$,
often viewed as representing
an element of $\{0, 1, \ldots, 2^n - 1\}$.

State stored in $n$ qubits:
a nonzero element of $\mathbb{C}^{2^n}$.
Retrieving this vector is tough!

If $n$ qubits have state
$(a_0, a_1, \ldots, a_{2^n-1})$ then
**measuring** the qubits produces
an element of $\{0, 1, \ldots, 2^n - 1\}$
and destroys the state.
Measurement produces element $q$
with probability $|a_q|^2/\sum_r |a_r|^2$.

Some examples of 3-qubit states:
$(1, 0, 0, 0, 0, 0, 0, 0)$ is
"$|0\rangle$" in standard notation.
Measurement produces 0.

$(0, 0, 0, 0, 0, 0, 1, 0)$ is
"$|6\rangle$" in standard notation.
Measurement produces 6.

$(0, 0, 0, 0, 0, 0, -7i, 0) = -7i|6\rangle$:
Measurement produces 6.
Data ("state") stored in $n$ bits:
an element of $\{0, 1\}^n$, 
often viewed as representing
an element of $\{0, 1, \ldots, 2^n - 1\}$.

State stored in $n$ qubits:
a nonzero element of $\mathbb{C}^{2^n}$.
Retrieving this vector is tough!

If $n$ qubits have state
$(a_0, a_1, \ldots, a_{2^n-1})$ then
**measuring** the qubits produces
an element of $\{0, 1, \ldots, 2^n - 1\}$
and destroys the state.
Measurement produces element $q$
with probability $|a_q|^2 / \sum_r |a_r|^2$.

Some examples of 3-qubit states:
$(1, 0, 0, 0, 0, 0, 0, 0)$ is
"$|0\rangle$" in standard notation.
Measurement produces 0.

$(0, 0, 0, 0, 0, 0, 1, 0)$ is
"$|6\rangle$" in standard notation.
Measurement produces 6.

$(0, 0, 0, 0, 0, 0, -7i, 0) = -7i|6\rangle$:
Measurement produces 6.

$(0, 0, 4, 0, 0, 0, 8, 0) = 4|2\rangle + 8|6\rangle$:
Measurement produces
2 with probability 20%,
6 with probability 80%.
Some examples of 3-qubit states:

- \((1, 0, 0, 0, 0, 0, 0, 0)\) is \(\ket{0}\) in standard notation. Measurement produces 0.

- \((0, 0, 0, 0, 0, 0, 1, 0)\) is \(\ket{6}\) in standard notation. Measurement produces 6.

- \((0, 0, 0, 0, 0, 0, -7i, 0)\) = \(-7i \ket{6}\): Measurement produces 6.

- \((0, 0, 4, 0, 0, 0, 8, 0)\) = \(4 \ket{2} + 8 \ket{6}\): Measurement produces 2 with probability 20%, 6 with probability 80%.
Data ("state") stored in $n$ bits:
- an element of $\{0,1\}^n$, often viewed as representing an element of $\{0,1,\ldots,2^n-1\}$.

State stored in $n$ qubits:
- a nonzero element of $\mathbb{C}^{2^n}$.

Retrieving this vector is tough!

If $(a_0,a_1,\ldots,a_{2^n-1})$ then measuring the qubits produces an element of $\{0,1,\ldots,2^n-1\}$ and destroys the state.

Measurement produces element $q$ with probability $|a_q|^2 / \sum_r |a_r|^2$.

Some examples of 3-qubit states:
- $(1,0,0,0,0,0,0,0)$ is "$|0\rangle$" in standard notation.
  Measurement produces 0.
- $(0,0,0,0,0,1,0)$ is "$|6\rangle$" in standard notation.
  Measurement produces 6.
- $(0,0,0,0,0,0,-7i,0) = -7i|6\rangle$:
  Measurement produces 6.
- $(0,0,4,0,0,0,8,0) = 4|2\rangle + 8|6\rangle$:
  Measurement produces
  2 with probability 20%,
  6 with probability 80%.

Fast quantum operations, part 1

$(a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7) \mapsto (a_1,a_0,a_3,a_2,a_5,a_4,a_7,a_6)$ is complementing index bit 0, hence "complementing qubit 0".
Data (“state”) stored in \( n \) bits:

- An element of \( \{0, 1\}^n \), often viewed as representing an element of \( \{0, 1, \ldots, 2^n - 1\} \).

State stored in \( n \) qubits:

- An element of \( \mathbb{C}^{2^n} \).

Retrieving this vector is tough!

If \( n \) qubits have state \((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\) then measuring the qubits produces an element of \( \{0, 1, \ldots, 2^n - 1\} \) and destroys the state.

\[ |a_q|^2 = \Pr |a_r|^2. \]

Some examples of 3-qubit states:

- \((1, 0, 0, 0, 0, 0, 0, 0)\) is “\(|0\rangle\)” in standard notation. Measurement produces 0.

- \((0, 0, 0, 0, 0, 0, 1, 0)\) is “\(|6\rangle\)” in standard notation. Measurement produces 6.

- \((0, 0, 0, 0, 0, 0, -7i, 0) = -7i|6\rangle:\) Measurement produces 6.

- \((0, 0, 4, 0, 0, 0, 8, 0) = 4|2\rangle + 8|6\rangle:\) Measurement produces 2 with probability 20%, 6 with probability 80%.

Fast quantum operations, part 1:

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\] is complementing index bit 0, hence “complementing qubit 0”.
Some examples of 3-qubit states:

$(1, 0, 0, 0, 0, 0, 0, 0)$ is “$|0\rangle$” in standard notation.
Measurement produces 0.

$(0, 0, 0, 0, 0, 0, 1, 0)$ is “$|6\rangle$” in standard notation.
Measurement produces 6.

$(0, 0, 0, 0, 0, 0, -7i, 0) = -7i |6\rangle$:
Measurement produces 6.

$(0, 0, 4, 0, 0, 0, 8, 0) = 4 |2\rangle + 8 |6\rangle$:
Measurement produces 2 with probability 20%,
6 with probability 80%.

Fast quantum operations, part 1

$(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) ↦→ (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)$ is complementing index bit 0, hence “complementing qubit 0”.
Some examples of 3-qubit states:

\((1, 0, 0, 0, 0, 0, 0, 0)\) is “\(|0\rangle\)” in standard notation. Measurement produces 0.

\((0, 0, 0, 0, 0, 0, 1, 0)\) is “\(|6\rangle\)” in standard notation. Measurement produces 6.

\((0, 0, 0, 0, 0, 0, 0, -7i) = -7i|6\rangle\):
Measurement produces 6.

\((0, 0, 4, 0, 0, 0, 8, 0) = 4|2\rangle + 8|6\rangle\):
Measurement produces 2 with probability 20%,
6 with probability 80%.

Fast quantum operations, part 1

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\)
is complementing index bit 0, hence “complementing qubit 0”.

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\) is measured as \((q_0, q_1, q_2)\),
representing \(q = q_0 + 2q_1 + 4q_2\),
with probability \(|a_q|^2 / \sum_r |a_r|^2\).

\((a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\) is measured as \((q_0 \oplus 1, q_1, q_2)\),
representing \(q \oplus 1\),
with probability \(|a_q|^2 / \sum_r |a_r|^2\).
Examples of 3-qubit states:

(0, 0, 0, 0, 0) is standard notation. Measurement produces 0.

(0, 0, 0, 1, 0) is standard notation. Measurement produces 6.

(0, 0, 0, −7i, 0) = −7i|6⟩: Measurement produces 6.

(0, 0, 0, 8, 0) = 4|2⟩ + 8|6⟩: Measurement produces probability 20%, probability 80%.

Fast quantum operations, part 1

(a₀, a₁, a₂, a₃, a₄, a₅, a₆, a₇) ↦→

(a₁, a₀, a₃, a₂, a₅, a₄, a₇, a₆)
is complementing index bit 0, hence “complementing qubit 0”.

(a₀, a₁, a₂, a₃, a₄, a₅, a₆, a₇)
is measured as (q₀, q₁, q₂), representing \( q = q₀ + 2q₁ + 4q₂ \), with probability \( |a_q|^2 / \sum_r |a_r|^2 \).

(a₁, a₀, a₃, a₂, a₅, a₄, a₇, a₆)
is measured as (q₀ ⊕ 1, q₁, q₂), representing \( q \oplus 1 \), with probability \( |a_q|^2 / \sum_r |a_r|^2 \).
3-qubit states:

\((1; 0; 0; 0; 0; 0; 0; 0)\) is

\(|0\rangle\) in standard notation.
Measurement produces 0.

\((0; 0; 0; 0; 0; 0; 1; 0)\) is

\(|6\rangle\) in standard notation.
Measurement produces 6.

\((0; 0; 0; 0; 0; 0; -7i; 0) = -7i|6\rangle:\)

is measured as \((q_0, q_1, q_2)\),
representing \(q = q_0 + 2q_1 + 4q_2\),
with probability \(|a_q|^2 / \sum_r |a_r|^2\).

\((0; 0; 4; 0; 0; 0; 8; 0) = 4|2\rangle + 8|6\rangle:\)

is measured as \((q_0 \oplus 1, q_1, q_2)\),
representing \(q \oplus 1\),
with probability \(|a_q|^2 / \sum_r |a_r|^2\).

Fast quantum operations, part 1

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\)
is complementing index bit 0,

hence “complementing qubit 0”.

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\)
is measured as \((q_0, q_1, q_2)\),
representing \(q = q_0 + 2q_1 + 4q_2\),
with probability \(|a_q|^2 / \sum_r |a_r|^2\).

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_4, a_5, a_6, a_7, a_0, a_1, a_2, a_3)\)
is “complementing qubit 2”:

\((q_0, q_1, q_2) \mapsto (q_0, q_1, q_2 \oplus 1)\).
Some examples of 3-qubit states:

(1; 0; 0; 0; 0; 0; 0; 0) is \(|0\rangle\) in standard notation.
Measurement produces 0.

(0; 0; 0; 0; 0; 0; 1; 0) is \(|6\rangle\) in standard notation.
Measurement produces 6.

(0; 0; 0; 0; 0; 0; -7i; 0) = -7i \(|6\rangle\):
Measurement produces 6.

(0; 0; 4; 0; 0; 0; 8; 0) = 4 \(|2\rangle\) + 8 \(|6\rangle\):
Measurement produces 2 with probability 20%,
6 with probability 80%.

---

\[
\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) &\mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6) \\
\text{is complementing index bit 0,} \\
&\text{hence “complementing qubit 0”}.
\end{align*}
\]

\[
\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) &\mapsto (a_4, a_5, a_6, a_7, a_0, a_1, a_2, a_3) \\
\text{is “complementing qubit 2”:} \\
(q_0, q_1, q_2) &\mapsto (q_0, q_1, q_2 \oplus 1).
\end{align*}
\]

\[
\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) &\mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \\
\text{is measured as} (q_0, q_1, q_2), \\
&\text{representing} q = q_0 + 2q_1 + 4q_2, \\
&\text{with probability} |a_q|^2 / \sum_r |a_r|^2.
\end{align*}
\]

\[
\begin{align*}
(a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6) &\mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6) \\
\text{is measured as} (q_0 \oplus 1, q_1, q_2), \\
&\text{representing} q \oplus 1, \\
&\text{with probability} |a_q|^2 / \sum_r |a_r|^2.
\end{align*}
\]
Fast quantum operations, part 1

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\)
is complementing index bit 0, hence “complementing qubit 0”.

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\)
is measured as \((q_0, q_1, q_2)\), representing \(q = q_0 + 2q_1 + 4q_2\),
with probability \(|a_q|^2 / \sum_r |a_r|^2\).

\((a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\)
is measured as \((q_0 \oplus 1, q_1, q_2)\), representing \(q \oplus 1\),
with probability \(|a_q|^2 / \sum_r |a_r|^2\).

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_4, a_5, a_6, a_7, a_0, a_1, a_2, a_3)\)
is “complementing qubit 2”:

\((q_0, q_1, q_2) \mapsto (q_0, q_1, q_2 \oplus 1)\).
Fast quantum operations, part 1

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\)

is complementing index bit 0, hence “complementing qubit 0”.

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\) is measured as \((q_0, q_1, q_2)\), representing \(q = q_0 + 2q_1 + 4q_2\), with probability \(|a_q|^2/\sum_r |a_r|^2\).

\((a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\) is measured as \((q_0 \oplus 1, q_1, q_2)\), representing \(q \oplus 1\), with probability \(|a_q|^2/\sum_r |a_r|^2\).

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\) \mapsto \((a_4, a_5, a_6, a_7, a_0, a_1, a_2, a_3)\)
is “complementing qubit 2”:

\((q_0, q_1, q_2) \mapsto (q_0, q_1, q_2 \oplus 1)\).

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\) \mapsto \((a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7)\)
is “swapping qubits 0 and 2”:

\((q_0, q_1, q_2) \mapsto (q_2, q_1, q_0)\).
Fast quantum operations, part 1

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\)

is complementing index bit 0, hence “complementing qubit 0”.

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\)

is measured as \((q_0, q_1, q_2)\), representing \(q = q_0 + 2q_1 + 4q_2\),
with probability \(|a_q|^2 / \sum_r |a_r|^2\).

\((a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\)

is measured as \((q_0 \oplus 1, q_1, q_2)\),
representing \(q \oplus 1\),
with probability \(|a_q|^2 / \sum_r |a_r|^2\).

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_4, a_5, a_6, a_7, a_0, a_1, a_2, a_3)\)
is “complementing qubit 2”:

\((q_0, q_1, q_2) \mapsto (q_0, q_1, q_2 \oplus 1)\).

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7)\)
is “swapping qubits 0 and 2”:

\((q_0, q_1, q_2) \mapsto (q_2, q_1, q_0)\).

Complementing qubit 2

\[= \text{swapping qubits 0 and 2} \]

\(\circ \text{complementing qubit 0}\)
\(\circ \text{swapping qubits 0 and 2}\).

Similarly: swapping qubits \(i, j\).
Quantum operations, part 1

(\(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7\)) \rightarrow
(\(a_3, a_2, a_5, a_4, a_7, a_6\))
is complementing index bit 0,
complementing qubit 0”.

(\(a_0, a_2, a_3, a_4, a_5, a_6, a_7\))
measured as (\(q_0, q_1, q_2\)),
setting \(q = q_0 + 2q_1 + 4q_2\),
probability \(|a_q|^2 / \sum_r |a_r|^2\).

(\(a_3, a_2, a_5, a_4, a_7, a_6\))
measured as (\(q_0 \oplus 1, q_1, q_2\)),
setting \(q \oplus 1\),
probability \(|a_q|^2 / \sum_r |a_r|^2\).

Complementing qubit 2
= swapping qubits 0 and 2
  \(\circ\) complementing qubit 0
  \(\circ\) swapping qubits 0 and 2.

Similarly: swapping qubits \(i, j\).
Fast quantum operations, part 1

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\]
is complementing index bit 0, hence “complementing qubit 0”.

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_4, a_7, a_6)\]
is measured as \((q_0, q_1, q_2)\), representing \(q = q_0 + 2q_1 + 4q_2\), with probability \(|a_q|^2 / \sum_r |a_r|^2\).

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_4, a_5, a_6, a_7, a_0, a_1, a_2, a_3)\]
is “complementing qubit 2”:
\[(q_0, q_1, q_2) \mapsto (q_0, q_1, q_2 \oplus 1)\].

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7)\]
is “swapping qubits 0 and 2”:
\[(q_0, q_1, q_2) \mapsto (q_2, q_1, q_0)\].

Complementing qubit 2
\[= \text{swapping qubits 0 and 2}\]
\[\circ \text{complementing qubit 0}\]
\[\circ \text{swapping qubits 0 and 2}.\]

Similarly: swapping qubits \(i, j\).

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_3, a_2, a_4, a_5, a_6, a_7)\]
is a “reversible XOR gate” = “controlled NOT gate”:
\[(q_0, q_1, q_2) \mapsto (q_0 \oplus q_1, q_1, q_2)\].
Fast quantum operations, part 1

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)\]

is complementing index bit 0, hence “complementing qubit 0”.

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7)\]

is measured as \((q_0, q_1, q_2)\), representing
\[q = q_0 + 2q_1 + 4q_2,\]
with probability \(|a_q|^2\).

Complementing qubit 2

\[= \text{swapping qubits 0 and 2} \]

\[\circ \text{ complementing qubit 0}\]

\[\circ \text{ swapping qubits 0 and 2.}\]

Similarly: swapping qubits \(i, j\).
\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_4, a_5, a_6, a_7, a_0, a_1, a_2, a_3)\]
is “complementing qubit 2”:
\[(q_0, q_1, q_2) \mapsto (q_0, q_1, q_2 \oplus 1).\]

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7)\]
is “swapping qubits 0 and 2”:
\[(q_0, q_1, q_2) \mapsto (q_2, q_1, q_0).\]

Complementing qubit 2

\[
\text{= swapping qubits 0 and 2} \\
\text{  \circ complementing qubit 0} \\
\text{  \circ swapping qubits 0 and 2.}
\]

Similarly: swapping qubits \(i, j\).
(a₀, a₁, a₂, a₃, a₄, a₅, a₆, a₇) ↦
(a₄, a₅, a₆, a₇, a₀, a₁, a₂, a₃)
is “complementing qubit 2”:
(q₀, q₁, q₂) ↦ (q₀, q₁, q₂ ⊕ 1).

(a₀, a₁, a₂, a₃, a₄, a₅, a₆, a₇) ↦
(a₀, a₄, a₂, a₆, a₁, a₅, a₃, a₇)
is “swapping qubits 0 and 2”:
(q₀, q₁, q₂) ↦ (q₂, q₁, q₀).

Complementing qubit 2
= swapping qubits 0 and 2
  • complementing qubit 0
  • swapping qubits 0 and 2.

Similarly: swapping qubits i, j.

(a₀, a₁, a₂, a₃, a₄, a₅, a₆, a₇) ↦
(a₀, a₁, a₃, a₂, a₄, a₅, a₆, a₇)
is a “reversible XOR gate” =
“controlled NOT gate”:
(q₀, q₁, q₂) ↦ (q₀ ⊕ q₁, q₁, q₂).

Example with more qubits:
(a₀, a₁, a₂, a₃, a₄, a₅, a₆, a₇, a₈, a₉, a₁₀, a₁₁, a₁₂, a₁₃, a₁₄, a₁₅,
a₁₆, a₁₇, a₁₈, a₁₉, a₂₀, a₂₁, a₂₂, a₂₃, a₂₄, a₂₅, a₂₆, a₂₇, a₂₈, a₂₉, a₃₀, a₃₁)
↦ (a₀, a₁, a₃, a₂, a₄, a₅, a₇, a₆,
a₈, a₉, a₁₁, a₁₀, a₁₂, a₁₃, a₁₅, a₁₄,
a₁₆, a₁₇, a₁ₙ, a₁₈, a₂₀, a₂₁, a₂₂, a₂₃, a₂₄, a₂₅, a₂₇, a₂₆, a₂₈, a₂₉, a₃₁, a₃₀).
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)

is a “reversible XOR gate” = “controlled NOT gate”:
(q_0, q_1, q_2) \mapsto (q_0 \oplus q_1, q_1, q_2).

Example with more qubits:
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31})
\mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}).
is "complementing qubit 2":

\[(q_0, q_1, q_2) \mapsto (q_0, q_1 \oplus 1, q_2).\]

Complementing qubit 2 = swapping qubits 0 and 2:

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_4, a_2, a_6, a_1, a_5, a_7, a_3).\]

is a "reversible XOR gate" = "controlled NOT gate":

\[(q_0, q_1) \mapsto (q_0 \oplus q_1, q_1).\]

is a "Toffoli gate" = "controlled controlled NOT gate":

\[(q_0, q_1, q_2) \mapsto (q_0 \oplus q_1 q_2, q_1, q_2).\]

Example with more qubits:

\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6).\]
\( (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_3, a_2, a_4, a_5, a_7, a_6) \)
is a “reversible XOR gate” = “controlled NOT gate”:
\((q_0, q_1, q_2) \mapsto (q_0 \oplus q_1, q_1, q_2)\).

Example with more qubits:
\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}) \mapsto (a_0, a_1, a_3, a_2, a_4, a_5, a_7, a_6, a_8, a_9, a_{11}, a_{10}, a_{12}, a_{13}, a_{15}, a_{14}, a_{16}, a_{17}, a_{19}, a_{18}, a_{20}, a_{21}, a_{23}, a_{22}, a_{24}, a_{25}, a_{27}, a_{26}, a_{28}, a_{29}, a_{31}, a_{30})\).

\( (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6) \)
is a “Toffoli gate” = “controlled controlled NOT gate”:
\((q_0, q_1, q_2) \mapsto (q_0 \oplus q_1 q_2, q_1, q_2)\).
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto
(a_0, a_1, a_3, a_2, a_4, a_5, a_7, a_6)
is a “reversible XOR gate” =
“controlled NOT gate”:
(q_0, q_1, q_2) \mapsto (q_0 \oplus q_1, q_1, q_2).

Example with more qubits:
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31})
\mapsto
(a_0, a_1, a_3, a_2, a_4, a_5, a_7, a_6, a_8, a_9, a_{11}, a_{10}, a_{12}, a_{13}, a_{15}, a_{14}, a_{16}, a_{17}, a_{18}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{27}, a_{26}, a_{28}, a_{29}, a_{31}, a_{30}).

(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto
(a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6)
is a “Toffoli gate” =
“controlled controlled NOT gate”:
(q_0, q_1, q_2) \mapsto (q_0 \oplus q_1 q_2, q_1, q_2).
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) ↦→ (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)
is a “reversible XOR gate” =
“controlled NOT gate”:
(q_0, q_1, q_2) ↦→ (q_0 ⊕ q_1, q_1, q_2).

Example with more qubits:
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}) ↦→ (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}).

(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) ↦→ (a_0, a_1, a_3, a_2, a_4, a_5, a_7, a_6)
is a “Toffoli gate” =
“controlled controlled NOT gate”:
(q_0, q_1, q_2) ↦→ (q_0 ⊕ q_1 q_2, q_1, q_2).

Example with more qubits:
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}) ↦→ (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}).
Reversible computation

Say \( p \) is a permutation of \( \{0, 1, \ldots, 2^n - 1\} \).

General strategy to compose these fast ancilla-free gate to obtain \( (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \):

\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_3, a_2, a_4, a_5, a_7, a_6)
\]

"reversible XOR gate" = "controlled NOT gate":

\[
(q_0, q_1, q_2) \mapsto (q_0 \oplus q_1, q_1, q_2).
\]

Example with more qubits:

\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_10, a_11, a_12, a_13, a_14, a_15, a_16, a_17, a_18, a_19, a_20, a_21, a_22, a_23, a_24, a_25, a_26, a_27, a_28, a_29, a_30, a_31) \mapsto (a_1, a_3, a_2, a_4, a_5, a_7, a_6, a_10, a_11, a_12, a_13, a_14, a_15, a_16, a_17, a_18, a_19, a_20, a_21, a_22, a_23, a_24, a_25, a_26, a_27, a_28, a_29, a_30, a_31)
\]

\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_10, a_11, a_12, a_13, a_14, a_15, a_16, a_17, a_18, a_19, a_20, a_21, a_22, a_23, a_24, a_25, a_26, a_27, a_28, a_29, a_30, a_31) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_10, a_11, a_12, a_13, a_15, a_14, a_16, a_17, a_18, a_19, a_20, a_21, a_22, a_23, a_24, a_25, a_26, a_27, a_28, a_29, a_31, a_30).
\]
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6)

is a “Toffoli gate” = “controlled controlled NOT gate”:

(q_0, q_1, q_2) \mapsto (q_0 \oplus q_1 q_2, q_1, q_2).

Example with more qubits:

(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31})

\mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}).

Reversible computation

Say p is a permutation of \{0, 1, \ldots, 2^n - 1\}. General strategy to compose these fast quantum operations to obtain index permutation:

(\mathbf{a}^\top_0, \mathbf{a}^\top_1, \mathbf{a}^\top_2, \ldots, \mathbf{a}^\top_{2^n - 1}) \mapsto (\mathbf{a}^p_{\mathbf{p}^{-1}(0)}, \mathbf{a}^p_{\mathbf{p}^{-1}(1)}, \ldots, \mathbf{a}^p_{\mathbf{p}^{-1}(2^n - 1)}).
Reversible computation

Say $p$ is a permutation of $\{0, 1, \ldots, 2^n - 1\}$.

General strategy to compose these fast quantum operations to obtain index permutation:

$$(a_0, a_1, \ldots, a_{2^n - 1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n - 1)}).$$
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6)
is a “Toffoli gate” = “controlled controlled NOT gate”:
(q_0, q_1, q_2) \mapsto (q_0 \oplus q_1 q_2, q_1, q_2).

Example with more qubits:
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31})
\mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{15}, a_{14}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{23}, a_{22}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{31}, a_{30}).

Reversible computation
Say \(p\) is a permutation of \(\{0, 1, \ldots, 2^n - 1\}\).

General strategy to compose these fast quantum operations to obtain index permutation
(a_0, a_1, \ldots, a_{2^n - 1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n - 1)}):
(a₀, a₁, a₂, a₃, a₄, a₅, a₆, a₇) ↦→ (a₀, a₁, a₂, a₃, a₄, a₅, a₇, a₆)
is a “Toffoli gate” = “controlled controlled NOT gate”:
(q₀, q₁, q₂) ↦→ (q₀ ⊕ q₁q₂, q₁, q₂).

Example with more qubits:
(a₀, a₁, a₂, a₃, a₄, a₅, a₆, a₇, a₈, a₉, a₁₀, a₁₁, a₁₂, a₁₃, a₁₄, a₁₅,
a₁₆, a₁₇, a₁₈, a₁₉, a₂₀, a₂₁, a₂₂, a₂₃, a₂₄, a₂₅, a₂₆, a₂₇, a₂₈, a₂₉, a₃₀, a₃₁)  
⇒ (a₀, a₁, a₂, a₃, a₄, a₅, a₇, a₆, a₈, a₉, a₁₀, a₁₁, a₁₂, a₁₃, a₁₅, a₁₄, a₁₆, a₁₇, a₁₈, a₁₉, a₂₀, a₂₁, a₂₃, a₂₂, a₂₄, a₂₅, a₂₆, a₂₇, a₂₈, a₂₉, a₃₁, a₃₀).

Reversible computation
Say p is a permutation of \{0, 1, \ldots, 2^n - 1\}.

General strategy to compose these fast quantum operations to obtain index permutation
(a₀, a₁, \ldots, a_{2^n-1}) ↦→ (a⁻¹₁(0), a⁻¹₁(1), \ldots, a⁻¹₁(2^n-1)):

1. Build a traditional circuit to compute \( j \mapsto p(j) \) using NOT/XOR/AND gates.
2. Convert into reversible gates: e.g., convert AND into Toffoli.
Reversible computation

Say \( p \) is a permutation of \( \{0, 1, \ldots, 2^n - 1\} \).

General strategy to compose these fast quantum operations to obtain index permutation

\[
(a_0, a_1, \ldots, a_{2^n-1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n-1)})
\]

1. Build a traditional circuit to compute \( j \mapsto p(j) \) using NOT/XOR/AND gates.

2. Convert into reversible gates: e.g., convert AND into Toffoli.

Example: Let's compute

\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)
\]

permutation

\[
q_0 \mapsto q_0 + 1 \mod 8.
\]

1. Build a traditional circuit to compute \( q_0 \mapsto q_0 + 1 \mod 8 \).

2. Convert into reversible gates: e.g., convert AND into Toffoli.
Reversible computation

Say \( p \) is a permutation of \( \{0, 1, \ldots, 2^n - 1\} \).

General strategy to compose these fast quantum operations to obtain index permutation
\[
(a_0, a_1, \ldots, a_{2^n-1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n-1)});
\]

1. Build a traditional circuit to compute \( j \mapsto p(j) \) using NOT/XOR/AND gates.
2. Convert into reversible gates: e.g., convert AND into Toffoli.

Example: Let’s compute
\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5);
\]

permutation \( q \mapsto q + 1 \mod 8 \).

1. Build a traditional circuit to compute \( q \mapsto q + 1 \mod 8 \).

\[
\begin{align*}
q_0 & \quad q_1 \\
q_0 \oplus 1 & \quad q_1 \oplus c_1 \\
& \quad c_1 = q_1 q_0
\end{align*}
\]
Reversible computation

Say \( p \) is a permutation of \( \{0, 1, \ldots, 2^n - 1\} \).

General strategy to compose these fast quantum operations to obtain index permutation:

1. Build a traditional circuit to compute \( j \mapsto p(j) \).
2. Convert into reversible gates: e.g., convert AND into Toffoli.

Example: Let's compute
\[
\begin{align*}
( a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7 ) &\mapsto \\
( a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6 )
\end{align*}
\]

\[a_0, a_1, \ldots, a_{2^n - 1} \mapsto a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n - 1)}\]
Reversible computation

Say $p$ is a permutation of $\{0, 1, \ldots, 2^n - 1\}$.

General strategy to compose these fast quantum operations to obtain index permutation
\[(a_0, a_1, \ldots, a_{2^n - 1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n - 1)})\]:

1. Build a traditional circuit to compute $j \mapsto p(j)$ using NOT/XOR/AND gates.

2. Convert into reversible gates: e.g., convert AND into Toffoli.

Example: Let’s compute
\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)\]; permutation $q \mapsto q + 1 \mod 8$.

1. Build a traditional circuit to compute $q \mapsto q + 1 \mod 8$. 

\[
\begin{align*}
  q_0 &\downarrow & q_1 &\downarrow & q_2 \\
  q_0 &\oplus 1 \quad q_1 &\oplus q_0 \quad q_2 &\oplus c_1 \\
  c_1 &= q_1 q_0
\end{align*}
\]
Reversible computation

Say \( p \) is a permutation of \( \{ 0, 1, \ldots, 2^n - 1 \} \).

A general strategy to compose these fast quantum operations to obtain index permutation

\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6);
\]

permutation \( q \mapsto q + 1 \) mod 8.

1. Build a traditional circuit to compute \( j \mapsto p(j) \) using NOT/XOR/AND gates.

2. Convert into reversible gates: e.g., convert AND into Toffoli.

Example: Let's compute

\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6);
\]

permutation \( q \mapsto q + 1 \) mod 8.

1. Build a traditional circuit to compute \( q \mapsto q + 1 \) mod 8.

2. Convert into reversible gates.

Toffoli for \( q_2 \leftarrow q_2 \oplus q_1 q_0 \):

\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3).
\]

Example:

\[
\begin{align*}
q_0 & \quad q_1 & \quad q_2 \\
\downarrow & \quad \downarrow & \quad \downarrow \\
q_0 \oplus 1 & \quad q_1 \oplus q_0 & \quad q_2 \oplus c_1 \\
\end{align*}
\]

\[
c_1 = q_1 q_0
\]
Reversible computation

Say $p$ is a permutation of $\{0, 1, \ldots, 2^n - 1\}$.

General strategy to compose these fast quantum operations to obtain index permutation $(a_0, a_1, a_2, \ldots, a_{2^n - 1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n - 1)})$:

1. Build a traditional circuit to compute $j \mapsto p(j)$ using NOT/XOR/AND gates.
2. Convert into reversible gates: e.g., convert AND into Toffoli.

Example: Let's compute $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)$; permutation $q \mapsto q + 1 \mod 8$.

1. Build a traditional circuit to compute $q \mapsto q + 1 \mod 8$.

2. Convert into reversible gates: Toffoli for $q_2 \leftarrow q_2 \oplus q_1 q_0$:

$(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3)$. 

![Diagram of quantum circuit](image)
Example: Let’s compute
\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6);\]
permutation \(q \mapsto q + 1 \text{ mod } 8.\)

1. Build a traditional circuit
to compute \(q \mapsto q + 1 \text{ mod } 8.\)

\[
\begin{align*}
q_0 & \quad q_1 & \quad q_2 \\
\downarrow & \quad \downarrow & \quad \downarrow \\
q_0 \oplus 1 & \quad q_1 \oplus q_0 & \quad q_2 \oplus c_1 \\
\end{align*}
\]

\[c_1 = q_1 q_0\]

2. Convert into reversible gates.
Toffoli for \(q_2 \leftarrow q_2 \oplus q_1 q_0:\)
\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3).\]
Example: Let’s compute
\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)\);
permutation \(q \mapsto q + 1 \text{ mod } 8\).

1. Build a traditional circuit
to compute \(q \mapsto q + 1 \text{ mod } 8\).

2. Convert into reversible gates.
Toffoli for \(q_2 \leftarrow q_2 \oplus q_1 q_0\):
\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3)\).
Example: Let’s compute 
\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6);\]
permutation \(q \mapsto q + 1 \mod 8\).

1. Build a traditional circuit to compute \(q \mapsto q + 1 \mod 8\).

\[
\begin{array}{c}
q_0 \\
\downarrow
\end{array} 
\begin{array}{c}
q_1 \\
\downarrow
\end{array} 
\begin{array}{c}
q_2 \\
\downarrow
\end{array} 
\begin{array}{c}
c_1 = q_1 q_0 \\
\downarrow
\end{array} 
\begin{array}{c}
q_0 \oplus 1 \\
\downarrow
\end{array} 
\begin{array}{c}
q_1 \oplus q_0 \\
\downarrow
\end{array} 
\begin{array}{c}
q_2 \oplus c_1 \\
\downarrow
\end{array}
\]

2. Convert into reversible gates.

Toffoli for \(q_2 \leftarrow q_2 \oplus q_1 q_0\):
\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_3).\]

Controlled NOT for \(q_1 \leftarrow q_1 \oplus q_0\):
\[(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_3) \mapsto (a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5).\]
Example: Let’s compute
\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)\);
permutation \(q \mapsto q + 1 \mod 8\).

1. Build a traditional circuit to compute \(q \mapsto q + 1 \mod 8\).

2. Convert into reversible gates.

   Toffoli for \(q_2 \leftarrow q_2 \oplus q_1 q_0\):
   \((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3)\).

   Controlled NOT for \(q_1 \leftarrow q_1 \oplus q_0\):
   \((a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3) \mapsto (a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5)\).

   NOT for \(q_0 \leftarrow q_0 \oplus 1\):
   \((a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)\).
Example: Let's compute
\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)\);
A permutation \(q \mapsto q + 1 \mod 8\).

1. Build a traditional circuit to compute \(q \mapsto q + 1 \mod 8\).

2. Convert into reversible gates.

   - Toffoli for \(q_2 \leftarrow q_2 \oplus q_1 q_0\):
     \((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3)\).

   - Controlled NOT for \(q_1 \leftarrow q_1 \oplus q_0\):
     \((a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3) \mapsto (a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5)\).

   - NOT for \(q_0 \leftarrow q_0 \oplus 1\):
     \((a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)\).

This permutation example was deceptively easy.
It didn't need many operations.
For large \(n\), most permutations \(p\) need many operations ⇒ slow.
Really want fast circuits.
1. Build a traditional circuit to compute 
\( q \mapsto q + 1 \mod 8 \).

2. Convert into reversible gates.

Toffoli for \( q_2 \leftarrow q_2 \oplus q_1 q_0 \):
\[
( a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7 ) \mapsto ( a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3 ).
\]

Controlled NOT for \( q_1 \leftarrow q_1 \oplus q_0 \):
\[
( a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3 ) \mapsto ( a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5 ).
\]

NOT for \( q_0 \leftarrow q_0 \oplus 1 \):
\[
( a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5 ) \mapsto ( a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6 ).
\]

This permutation example was deceptively easy.
It didn’t need many operations.
For large \( n \), most permutations need many operations ⇒ slow.
Really want fast circuits.
2. Convert into reversible gates.

Toffoli for \( q_2 \leftarrow q_2 \oplus q_1 q_0 \):
\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto
(a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3).
\]

Controlled NOT for \( q_1 \leftarrow q_1 \oplus q_0 \):
\[
(a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3) \mapsto
(a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5).
\]

NOT for \( q_0 \leftarrow q_0 \oplus 1 \):
\[
(a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5) \mapsto
(a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6).
\]

This permutation example was deceptively easy.

It didn’t need many operations.

For large \( n \), most permutations need many operations \( \Rightarrow \) slow.

Really want fast circuits.
2. Convert into reversible gates.

Toffoli for $q_2 \leftarrow q_2 \oplus q_1 q_0$:

$$(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3).$$

Controlled NOT for $q_1 \leftarrow q_1 \oplus q_0$:

$$(a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3) \mapsto (a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5).$$

NOT for $q_0 \leftarrow q_0 \oplus 1$:

$$(a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6).$$

This permutation example was deceptively easy.

It didn’t need many operations.

For large $n$, most permutations $p$ need many operations $\Rightarrow$ slow.

Really want fast circuits.
2. Convert into reversible gates.

Toffoli for $q_2 \leftarrow q_2 \oplus q_1 q_0$:
$(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3)$.

Controlled NOT for $q_1 \leftarrow q_1 \oplus q_0$:
$(a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3) \mapsto (a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5)$.

NOT for $q_0 \leftarrow q_0 \oplus 1$:
$(a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)$.

This permutation example was deceptively easy.

It didn’t need many operations.

For large $n$, most permutations $p$ need many operations $\Rightarrow$ slow.

Really want fast circuits.

Also, it didn’t need extra storage: circuit operated “in place” after computation $c_1 \leftarrow q_1 q_0$ was merged into $q_2 \leftarrow q_2 \oplus c_1$.

Typical circuits aren’t in-place.
Convert into reversible gates.

For \( q_2 \leftarrow q_2 \oplus q_1 q_0 \):
\[
(a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_2, a_7, a_4, a_5, a_6, a_3).
\]

Need NOT for \( q_1 \leftarrow q_1 \oplus q_0 \):
\[
(a_2, a_7, a_4, a_5, a_6, a_3) \mapsto (a_2, a_1, a_4, a_3, a_6, a_5).
\]

For \( q_0 \leftarrow q_0 \oplus 1 \):
\[
(a_2, a_1, a_4, a_3, a_6, a_5) \mapsto (a_1, a_2, a_3, a_4, a_5, a_6).
\]

This permutation example was deceptively easy.

It didn’t need many operations.

For large \( n \), most permutations \( p \) need many operations \( \Rightarrow \) slow.

Really want fast circuits.

Also, it didn’t need extra storage: circuit operated “in place” after computation \( c_1 \leftarrow q_1 q_0 \) was merged into \( q_2 \leftarrow q_2 \oplus c_1 \).

Typical circuits aren’t in-place.

Start from any circuit:

inputs \( b_1 ; b_2 ; \ldots ; b_i ; b_{i+1} = 1 \oplus b_{f(i+1)} b_{g(i+1)} ; b_{i+2} = 1 \oplus b_{f(i+2)} b_{g(i+2)} ; \ldots ; b_T = 1 \oplus b_{f(T)} b_{g(T)} \); specified outputs.
2. Convert into reversible gates.

Toffoli for $q_2 \leftarrow q_2 \oplus q_1 q_0$:

$\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
a_4 \\
a_5 \\
a_6 \\
a_7 \\
\end{pmatrix} \mapsto \begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
a_7 \\
a_4 \\
a_5 \\
a_6 \\
a_3 \\
\end{pmatrix}$.

Controlled NOT for $q_1 \leftarrow q_1 \oplus q_0$:

$\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
a_4 \\
a_5 \\
a_6 \\
a_7 \\
\end{pmatrix} \mapsto \begin{pmatrix}
a_0 \\
a_7 \\
a_2 \\
a_1 \\
a_4 \\
a_3 \\
a_6 \\
a_5 \\
\end{pmatrix}$.

NOT for $q_0 \leftarrow q_0 \oplus 1$:

$\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
a_4 \\
a_5 \\
a_6 \\
a_7 \\
\end{pmatrix} \mapsto \begin{pmatrix}
a_7 \\
a_0 \\
a_1 \\
a_2 \\
\end{pmatrix}$.

This permutation example was deceptively easy.

It didn’t need many operations.

For large $n$, most permutations $p$ need many operations $\Rightarrow$ slow.

Really want fast circuits.

Also, it didn’t need extra storage: circuit operated “in place” after computation $c_1 \leftarrow q_1 q_0$ was merged into $q_2 \leftarrow q_2 \oplus c_1$.

Typical circuits aren’t in-place.

Start from any circuit: inputs $b_1, b_2, \ldots, b_i, b_{i+1} = 1 \oplus b_{f(i+1)} b_{g(i+1)}; b_{i+2} = 1 \oplus b_{f(i+2)} b_{g(i+2)}; \ldots; b_T = 1 \oplus b_{f(T)} b_{g(T)}$; specified outputs.
This permutation example was deceptively easy.

It didn’t need many operations.

For large $n$, most permutations $p$ need many operations $\Rightarrow$ slow.

Really want fast circuits.

Also, it didn’t need extra storage:
circuit operated “in place” after computation $c_1 \leftarrow q_1 q_0$ was merged into $q_2 \leftarrow q_2 \oplus c_1$.

Typical circuits aren’t in-place.

Start from any circuit:
inputs $b_1, b_2, \ldots, b_i$;

$b_{i+1} = 1 \oplus b_{f(i+1)} b_{g(i+1)}$;

$b_{i+2} = 1 \oplus b_{f(i+2)} b_{g(i+2)}$;

$\ldots$

$b_T = 1 \oplus b_{f(T)} b_{g(T)}$; specified outputs.
This permutation example was deceptively easy.

It didn’t need many operations.

For large $n$, most permutations $p$ need many operations $\Rightarrow$ slow.

Really want fast circuits.

Also, it didn’t need extra storage: circuit operated “in place” after computation $c_1 \leftarrow q_1 q_0$ was merged into $q_2 \leftarrow q_2 \oplus c_1$.

Typical circuits aren’t in-place.

---

Start from any circuit:

inputs $b_1, b_2, \ldots, b_i$;

\[
b_{i+1} = 1 \oplus b_{f(i+1)} b_{g(i+1)}; \\
b_{i+2} = 1 \oplus b_{f(i+2)} b_{g(i+2)}; \\
\ldots \\
b_T = 1 \oplus b_{f(T)} b_{g(T)};
\]

specified outputs.
This permutation example was deceptively easy.

It didn’t need many operations.

For large $n$, most permutations $p$ need many operations $\Rightarrow$ slow.

Really want fast circuits.

Also, it didn’t need extra storage: circuit operated “in place” after computation $c_1 \leftarrow q_1 q_0$ was merged into $q_2 \leftarrow q_2 \oplus c_1$.

Typical circuits aren’t in-place.

Start from any circuit:
inputs $b_1, b_2, \ldots, b_i$;

$b_{i+1} = 1 \oplus b_{f(i+1)} b_{g(i+1)}$;

$b_{i+2} = 1 \oplus b_{f(i+2)} b_{g(i+2)}$;

$\ldots$

$b_T = 1 \oplus b_{f(T)} b_{g(T)}$;

specified outputs.

Reversible but dirty:
inputs $b_1, b_2, \ldots, b_T$;

$b_{i+1} \leftarrow 1 \oplus b_{i+1} \oplus b_{f(i+1)} b_{g(i+1)}$;

$b_{i+2} \leftarrow 1 \oplus b_{i+2} \oplus b_{f(i+2)} b_{g(i+2)}$;

$\ldots$

$b_T \leftarrow 1 \oplus b_T \oplus b_{f(T)} b_{g(T)}$.

Same outputs if all of $b_{i+1}, \ldots, b_T$ started as 0.
This permutation example was deceptively easy. It didn’t need many operations. For large $n$, most permutations $p$ need many operations $\Rightarrow$ slow. Really want fast circuits.

Also, it didn’t need extra storage: circuit operated “in place” after computation $c_1 \leftarrow q_1 q_0$ was merged into $q_2 \leftarrow q_2 \oplus c_1$.

Circuits aren’t in-place.

Start from any circuit:
inputs $b_1, b_2, \ldots, b_i$;

\[
\begin{align*}
  b_{i+1} &= 1 \oplus b_f(i+1) b_g(i+1); \\
  b_{i+2} &= 1 \oplus b_f(i+2) b_g(i+2); \\
  \vdots \\
  b_T &= 1 \oplus b_f(T) b_g(T);
\end{align*}
\]

specified outputs.

Reversible but dirty:
inputs $b_1, b_2, \ldots, b_T$;

\[
\begin{align*}
  b_{i+1} &\leftarrow 1 \oplus b_{i+1} \oplus b_f(i+1) b_g(i+1); \\
  b_{i+2} &\leftarrow 1 \oplus b_{i+2} \oplus b_f(i+2) b_g(i+2); \\
  \vdots \\
  b_T &\leftarrow 1 \oplus b_T \oplus b_f(T) b_g(T).
\end{align*}
\]

Same outputs if all of $b_{i+1}, \ldots, b_T$ started as 0.

Reversible and clean: after finishing dirty computation, set non-outputs back to 0, by repeating same operations on non-outputs in reverse order.

Original computation:
(inputs) $\mapsto$ (inputs, dirt, outputs).

Dirty reversible computation:
(inputs, zeros) $\mapsto$ (inputs, dirt, outputs).

Clean reversible computation:
(inputs, zeros, zeros) $\mapsto$ (inputs, zeros, outputs).

Here is the full text representation as if you were reading it naturally:

This permutation example was deceptively easy. It didn’t need many operations. For large $n$, most permutations $p$ need many operations $\Rightarrow$ slow. Really want fast circuits.

Also, it didn’t need extra storage: circuit operated “in place” after computation $c_1 \leftarrow q_1 q_0$ was merged into $q_2 \leftarrow q_2 \oplus c_1$.

Circuits aren’t in-place.

Start from any circuit:
inputs $b_1, b_2, \ldots, b_i$;

\[
\begin{align*}
  b_{i+1} &= 1 \oplus b_f(i+1) b_g(i+1); \\
  b_{i+2} &= 1 \oplus b_f(i+2) b_g(i+2); \\
  \vdots \\
  b_T &= 1 \oplus b_f(T) b_g(T);
\end{align*}
\]

specified outputs.

Reversible but dirty:
inputs $b_1, b_2, \ldots, b_T$;

\[
\begin{align*}
  b_{i+1} &\leftarrow 1 \oplus b_{i+1} \oplus b_f(i+1) b_g(i+1); \\
  b_{i+2} &\leftarrow 1 \oplus b_{i+2} \oplus b_f(i+2) b_g(i+2); \\
  \vdots \\
  b_T &\leftarrow 1 \oplus b_T \oplus b_f(T) b_g(T).
\end{align*}
\]

Same outputs if all of $b_{i+1}, \ldots, b_T$ started as 0.

Reversible and clean: after finishing dirty computation, set non-outputs back to 0, by repeating same operations on non-outputs in reverse order.

Original computation:
(inputs) $\mapsto$ (inputs, dirt, outputs).

Dirty reversible computation:
(inputs, zeros) $\mapsto$ (inputs, dirt, outputs).

Clean reversible computation:
(inputs, zeros, zeros) $\mapsto$ (inputs, zeros, outputs).
This permutation example was deceptively easy. It didn't need many operations. For large $n$, most permutations $p$ need many operations $\Rightarrow$ slow. Really want fast circuits. Also, it didn't need extra storage: circuit operated "in place" after computation $c_1 \leftarrow q_1 q_0$ was merged into $q_2 \oplus c_1$. Aren't in-place.

Start from any circuit:
inputs $b_1, b_2, \ldots, b_i$;

\[
b_{i+1} = 1 \oplus b_{f(i+1)} b_{g(i+1)};
\]

\[
b_{i+2} = 1 \oplus b_{f(i+2)} b_{g(i+2)};
\]

\[\cdots\]

\[
b_T = 1 \oplus b_{f(T)} b_{g(T)};
\]

specified outputs.

Reversible but dirty:
inputs $b_1, b_2, \ldots, b_T$;

\[
b_{i+1} \leftarrow 1 \oplus b_{i+1} \oplus b_{f(i+1)} b_{g(i+1)};
\]

\[
b_{i+2} \leftarrow 1 \oplus b_{i+2} \oplus b_{f(i+2)} b_{g(i+2)};
\]

\[\cdots\]

\[
b_T \leftarrow 1 \oplus b_T \oplus b_{f(T)} b_{g(T)}.
\]

Same outputs if all of $b_{i+1}, \ldots, b_T$ started as 0.

Reversible and clean: after finishing dirty computation, set non-outputs back to 0 by repeating same operations on non-outputs in reverse order.

Original computation:
(inputs) $\mapsto$ (inputs, dirt, outputs).

Dirty reversible computation:
(inputs, zeros, zeros) $\mapsto$ (inputs, dirt, outputs).

Clean reversible computation:
(inputs, zeros, outputs) $\mapsto$ (inputs, zeros, outputs).
This permutation example was deceptively easy. It didn’t need many operations. For large \( n \), most permutations \( p \) need many operations ⇒ slow. Really want fast circuits.

Also, it didn’t need extra storage: circuit operated “in place” after computation \( c_1 ← q_1 q_0 \) was merged into \( q_2 ← q_2 ⊕ c_1 \).

Typical circuits aren’t in-place.

Start from any circuit:
inputs \( b_1, b_2, \ldots, b_i \);
\( b_{i+1} = 1 ⊕ b_f(i+1) b_g(i+1) \);
\( b_{i+2} = 1 ⊕ b_f(i+2) b_g(i+2) \);
\ldots
\( b_T = 1 ⊕ b_f(T) b_g(T) \);
specified outputs.

Reversible but dirty:
inputs \( b_1, b_2, \ldots, b_T \);
\( b_{i+1} ← 1 ⊕ b_{i+1} ⊕ b_f(i+1) b_g(i+1) \);
\( b_{i+2} ← 1 ⊕ b_{i+2} ⊕ b_f(i+2) b_g(i+2) \);
\ldots
\( b_T ← 1 ⊕ b_T ⊕ b_f(T) b_g(T) \).
Same outputs if all of \( b_{i+1}, \ldots, b_T \) started as 0.

Reversible and clean: after finishing dirty computation, set non-outputs back to 0, by repeating same operations on non-outputs in reverse order.

Original computation:
(inputs) \( → \) (inputs; dirt; outputs).

Dirty reversible computation:
(inputs; zeros; zeros) \( → \) (inputs; dirt; outputs).

Clean reversible computation:
(inputs; zeros; zeros) \( → \) (inputs; zeros; outputs).
Start from any circuit:
inputs $b_1, b_2, \ldots, b_i$;

\begin{align*}
b_{i+1} &= 1 \oplus b_f(i+1) b_g(i+1) \\
b_{i+2} &= 1 \oplus b_f(i+2) b_g(i+2) \\
\vdots \\
b_T &= 1 \oplus b_f(T) b_g(T);
\end{align*}
specified outputs.

Reversible but dirty:
inputs $b_1, b_2, \ldots, b_T$;

\begin{align*}
b_{i+1} &\leftarrow 1 \oplus b_{i+1} \oplus b_f(i+1) b_g(i+1) \\
b_{i+2} &\leftarrow 1 \oplus b_{i+2} \oplus b_f(i+2) b_g(i+2) \\
\vdots \\
b_T &\leftarrow 1 \oplus b_T \oplus b_f(T) b_g(T).
\end{align*}
Same outputs if all of $b_{i+1}, \ldots, b_T$ started as 0.

Reversible and clean:
after finishing dirty computation, set non-outputs back to 0, by repeating same operations on non-outputs in reverse order.

Original computation:
(inputs) $\mapsto$
(inputs, dirt, outputs).

Dirty reversible computation:
(inputs, zeros, zeros) $\mapsto$
(inputs, dirt, outputs).

Clean reversible computation:
(inputs, zeros, zeros) $\mapsto$
(inputs, zeros, outputs).
From any circuit:
\[ b_1, b_2, \ldots, b_i; \]
\[ 1 \oplus b_f(i+1) b_g(i+1); \]
\[ 1 \oplus b_f(i+2) b_g(i+2); \]
\[ \oplus b_f(T) b_g(T); \]
all outputs.

Reversible and clean:
after finishing dirty computation, set non-outputs back to 0, by repeating same operations on non-outputs in reverse order.

Original computation:
\((\text{inputs}) \mapsto (\text{inputs}, \text{dirt}, \text{outputs})\).

Dirty reversible computation:
\((\text{inputs}, \text{zeros}, \text{zeros}) \mapsto (\text{inputs}, \text{dirt}, \text{outputs})\).

Clean reversible computation:
\((\text{inputs}, \text{zeros}, \text{zeros}) \mapsto (\text{inputs}, \text{zeros}, \text{outputs})\).

Given fast circuit for \( p \) and fast circuit for \( p - 1 \), build fast reversible circuit for
\((x, \text{zeros}) \mapsto (p(x), \text{zeros})\).
Start from any circuit:

\[ b_i; b_{i+1}; \ldots; b_{i+2}; \ldots; b_T; \]

\[ b_{i+1} = 1 \oplus b_f(i+1) b_g(i+1); \]
\[ b_{i+2} = 1 \oplus b_f(i+2) b_g(i+2); \]
\[ \ldots \]
\[ b_T = 1 \oplus b_f(T) b_g(T). \]

Reversible and clean:
after finishing dirty computation, set non-outputs back to 0, by repeating same operations on non-outputs in reverse order.

Original computation:
\[(\text{inputs}) \mapsto (\text{inputs, dirt, outputs}).\]

Dirty reversible computation:
\[(\text{inputs, zeros, zeros}) \mapsto (\text{inputs, dirt, outputs}).\]

Clean reversible computation:
\[(\text{inputs, zeros, zeros}) \mapsto (\text{inputs, zeros, outputs}).\]

Given fast circuit for \( p \) and fast circuit for \( p - 1 \), build fast reversible circuit:
\[(x, \text{zeros}) \mapsto (p(x), \text{zeros}).\]
Reversible and clean:
after finishing dirty computation,
set non-outputs back to 0,
by repeating same operations
on non-outputs in reverse order.

Original computation:
(inputs) $\mapsto$ (inputs, dirt, outputs).

Dirty reversible computation:
(inputs, zeros, zeros) $\mapsto$ (inputs, dirt, outputs).

Clean reversible computation:
(inputs, zeros, zeros) $\mapsto$ (inputs, zeros, outputs).

Given fast circuit for $p$ and fast circuit for $p^{-1}$,
built fast reversible circuit for
$(x, \text{zeros}) \mapsto (p(x), \text{zeros})$. 

\[ b_{i+1} = 1 \oplus b_f(i+1) b_g(i+1); \]
\[ b_{i+2} = 1 \oplus b_f(i+2) b_g(i+2); \]
\[ \vdots \]
\[ b_T = 1 \oplus b_f(T) b_g(T). \]
Reversible and clean: after finishing dirty computation, set non-outputs back to 0, by repeating same operations on non-outputs in reverse order.

Original computation:
(inputs) \mapsto (inputs, dirt, outputs).

Dirty reversible computation:
(inputs, zeros, zeros) \mapsto (inputs, dirt, outputs).

Clean reversible computation:
(inputs, zeros, zeros) \mapsto (inputs, zeros, outputs).

Given fast circuit for $p$ and fast circuit for $p^{-1}$, build fast reversible circuit for $(x, zeros) \mapsto (p(x), zeros)$. 
Reversible and clean:
after finishing dirty computation,
set non-outputs back to 0,
by repeating same operations
on non-outputs in reverse order.

Original computation:
\[(inputs) \mapsto (inputs; dirt; outputs).\]

Dirty reversible computation:
\[(inputs; zeros; zeros) \mapsto (inputs; dirt; outputs).\]

Clean reversible computation:
\[(inputs; zeros; zeros) \mapsto (inputs; zeros; outputs).\]

Given fast circuit for \(p\) and fast circuit for \(p^{-1}\),
built fast reversible circuit for
\[(x, zeros) \mapsto (p(x), zeros).\]

Replace reversible bit operations
with Toffoli gates etc.

Permuation on first \(2^n\) entries is
\[(a_0, a_1, \ldots, a_{2^n-1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n-1)}).\]

Typically prepare vectors
supported on first \(2^n\) entries
so don’t care how permutation
acts on last \(2^{n+z} - 2^n\) entries.
Reversible and clean: after finishing dirty computation, set non-outputs back to 0, by repeating same operations on non-outputs in reverse order.

Original computation:
\[(\text{inputs}) \mapsto (\text{inputs}; \text{dirt}; \text{outputs}).\]

Dirty reversible computation:
\[(\text{inputs}; \text{zeros}; \text{zeros}) \mapsto (\text{inputs}; \text{dirt}; \text{outputs}).\]

Clean reversible computation:
\[(\text{inputs}; \text{zeros}; \text{zeros}) \mapsto (\text{inputs}; \text{zeros}; \text{outputs}).\]

Given fast circuit for \(p\) and fast circuit for \(p^{-1}\), build fast reversible circuit for \((x; \text{zeros}) \mapsto (p(x); \text{zeros})\).

Replace reversible bit operations with Toffoli gates etc. permuting \(C_{2^{n+z}} \rightarrow C_{2^{n+z}}\). Permutation on first \(2^n\) entries is \((a_0, a_1, \ldots, a_{2^n-1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n-1)})\).

Typically prepare vectors supported on first \(2^n\) entries so don’t care how permutation acts on last \(2^{n+z} - 2^n\) entries.

Warning: Number of qubits \(\approx\) number of bit operations in original \(p; p^{-1}\) circuits. This can be much larger than number of bits stored in the original circuits.
Given fast circuit for $p$ and fast circuit for $p^{-1}$, build fast reversible circuit for $(x, \text{zeros}) \mapsto (p(x), \text{zeros})$.

Replace reversible bit operations with Toffoli gates etc.

Permuting $\mathbb{C}^{2^{n+z}} \to \mathbb{C}^{2^{n+z}}$.

Permutation on first $2^n$ entries is $(a_0, a_1, \ldots, a_{2^n-1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n-1)})$.

Typically prepare vectors supported on first $2^n$ entries so don’t care how permutation acts on last $2^{n+z} - 2^n$ entries.

Warning: Number of qubits $\approx$ number of bit operations in original $p, p^{-1}$ circuits.

This can be much larger than number of bits stored in the original circuits.
Given fast circuit for $p$ and fast circuit for $p^{-1}$, build fast reversible circuit for $(x, \text{zeros}) \mapsto (p(x), \text{zeros})$.

Replace reversible bit operations with Toffoli gates etc.

Permuting $\mathbb{C}^{2^{n+z}} \rightarrow \mathbb{C}^{2^{n+z}}$.

Permutation on first $2^n$ entries is

$$(a_0, a_1, \ldots, a_{2^n-1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n-1)}).$$

Typically prepare vectors supported on first $2^n$ entries so don’t care how permutation acts on last $2^{n+z} - 2^n$ entries.

Warning: Number of qubits $\approx$ number of bit operations in original $p, p^{-1}$ circuits.

This can be much larger than number of bits stored in the original circuits.
Given fast circuit for $p$ and fast circuit for $p^{-1}$, build fast reversible circuit for $(x, \text{zeros}) \mapsto (p(x), \text{zeros})$.

Replace reversible bit operations with Toffoli gates etc. permuting $C^{2n+z} \rightarrow C^{2n+z}$.

Permutation on first $2^n$ entries is
$$(a_0, a_1, \ldots, a_{2^n-1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n-1)})$$.

Typically prepare vectors supported on first $2^n$ entries so don’t care how permutation acts on last $2^{n+z} - 2^n$ entries.

Warning: Number of qubits $\approx$ number of bit operations in original $p, p^{-1}$ circuits. This can be much larger than number of bits stored in the original circuits.
Given fast circuit for $p$ and fast circuit for $p^{-1}$, build fast reversible circuit for $(x, \text{zeros}) \mapsto (p(x), \text{zeros})$.

Replace reversible bit operations with Toffoli gates etc. permuting $C^{2^{n+z}} \rightarrow C^{2^{n+z}}$.

Permutation on first $2^n$ entries is $(a_0, a_1, \ldots, a_{2^n-1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n-1)})$.

Typically prepare vectors supported on first $2^n$ entries so don’t care how permutation acts on last $2^{n+z} - 2^n$ entries.

Warning: Number of qubits $\approx$ number of bit operations in original $p, p^{-1}$ circuits.

This can be much larger than number of bits stored in the original circuits.

Many useful techniques to compress into fewer qubits, but often these lose time.

Many subtle tradeoffs.
Given fast circuit for $p$ and fast circuit for $p^{-1}$, build fast reversible circuit for $(x, \text{zeros}) \mapsto (p(x), \text{zeros})$.

Replace reversible bit operations with Toffoli gates etc.
permuting $C^{2^n+2^z} \rightarrow C^{2^n+2^z}$.

Permutation on first $2^n$ entries is
$(a_0, a_1, \ldots, a_{2^n-1}) \mapsto (a_{p^{-1}(0)}, a_{p^{-1}(1)}, \ldots, a_{p^{-1}(2^n-1)})$.

Typically prepare vectors supported on first $2^n$ entries so don’t care how permutation acts on last $2^{n+z} - 2^n$ entries.

Warning: Number of qubits $\approx$ number of bit operations in original $p, p^{-1}$ circuits.

This can be much larger than number of bits stored in the original circuits.

Many useful techniques to compress into fewer qubits, but often these lose time. Many subtle tradeoffs.

Crude “poly-time” analyses don’t care about this, but serious cryptanalysis is much more precise.
Given fast circuit for $p$ and fast circuit for $p^{-1}$, build fast reversible circuit for $(x; \text{zeros}) \mapsto (p(x); \text{zeros})$.

Replace reversible bit operations with Toffoli gates etc.

Permuting $C^{2n+z} \rightarrow C^{2n+z}$.

Permutation on first $2^n$ entries is $(a_0; a_1; \ldots; a_{2^n-1}) \mapsto (a_{p^{-1}(0)}; a_{p^{-1}(1)}; \ldots; a_{p^{-1}(2^n-1)})$.

Typically prepare vectors supported on first $2^n$ entries so don't care how permutation acts on last $2^{n+z} - 2^n$ entries.

Warning: Number of qubits $\approx$ number of bit operations in original $p, p^{-1}$ circuits.

This can be much larger than number of bits stored in the original circuits.

Many useful techniques to compress into fewer qubits, but often these lose time.

Many subtle tradeoffs.

Crude “poly-time” analyses don’t care about this, but serious cryptanalysis is much more precise.

Fast quantum operations, part 2

“Hadamard”:

$$(a_0; a_1) \mapsto (a_0 + a_1; a_0 - a_1).$$
Given fast circuit for \( p \) and fast circuit for \( p^{-1} \), build fast reversible circuit for \((x; \text{zeros}) \mapsto (p(x); \text{zeros})\).

Replace reversible bit operations with Toffoli gates etc.

Permuting \( C^{2n+z} \rightarrow C^{2n+z} \).

First \( 2^n \) entries is
\[ (a_0; a_1; \ldots; a_{2^n-1}) \]

Vectors supported on first \( 2^n \) entries so don’t care how permutation acts on last \( 2^n + z - 2^n \) entries.

Warning: Number of qubits \( \approx \) number of bit operations in original \( p, p^{-1} \) circuits.

This can be much larger than number of bits stored in the original circuits.

Many useful techniques to compress into fewer qubits, but often these lose time.

Many subtle tradeoffs.

Crude “poly-time” analyses don’t care about this, but serious cryptanalysis is much more precise.
Warning: Number of qubits \( \approx \) number of bit operations in original \( p, p^{-1} \) circuits.

This can be much larger than number of bits stored in the original circuits.

Many useful techniques to compress into fewer qubits, but often these lose time.

Many subtle tradeoffs.

Crude “poly-time” analyses don’t care about this, but serious cryptanalysis is much more precise.

Fast quantum operations, part 2

“Hadamard”:

\[(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1)\]
Warning: Number of qubits \( \approx \) number of bit operations in original \( p, p^{-1} \) circuits.

This can be much larger than number of bits stored in the original circuits.

Many useful techniques to compress into fewer qubits, but often these lose time. Many subtle tradeoffs.

Crude “poly-time” analyses don’t care about this, but serious cryptanalysis is much more precise.

---

Fast quantum operations, part 2

“Hadamard”: \((a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1)\).
Warning: Number of **qubits** \( \approx \) number of **bit operations** in original \( p, p^{-1} \) circuits.

This can be much larger than number of **bits stored** in the original circuits.

Many useful techniques to compress into fewer qubits, but often these lose time. Many subtle tradeoffs.

Crude “poly-time” analyses don’t care about this, but serious cryptanalysis is much more precise.

---

Fast quantum operations, part 2

“Hadamard”:

\[
(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1).
\]

\[
(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3).
\]
Warning: Number of **qubits**
≈ number of **bit operations**
in original $p, p^{-1}$ circuits.

This can be much larger
than number of **bits stored**
in the original circuits.

Many useful techniques
to compress into fewer qubits,
but often these lose time.
Many subtle tradeoffs.

Crude “poly-time” analyses
don’t care about this,
but serious cryptanalysis
is much more precise.

**Fast quantum operations, part 2**

“Hadamard”:
$(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1)$.

$(a_0, a_1, a_2, a_3) \mapsto$
$(a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3)$.

Same for qubit 1:
$(a_0, a_1, a_2, a_3) \mapsto$
$(a_0 + a_2, a_1 + a_3, a_0 - a_2, a_1 - a_3)$. 
Warning: Number of qubits \(\approx\) number of bit operations in original \(p, p^{-1}\) circuits.

This can be much larger than number of bits stored in the original circuits.

Many useful techniques to compress into fewer qubits, but often these lose time. Many subtle tradeoffs.

Crude “poly-time” analyses don’t care about this, but serious cryptanalysis is much more precise.

Fast quantum operations, part 2

“Hadamard”:
\((a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1)\).

\((a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3)\).

Same for qubit 1:
\((a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1, a_1 + a_3, a_0 - a_2, a_1 - a_3)\).

Qubit 0 and then qubit 1:
\((a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3) \mapsto (a_0 + a_1 + a_2 + a_3, a_0 - a_1 + a_2 - a_3, a_0 + a_1 - a_2 - a_3, a_0 - a_1 - a_2 + a_3)\).
Number of qubits ≈ number of bit operations in original $p, p^{-1}$ circuits. This can be much larger than number of bits stored in the original circuits. Many useful techniques to compress into fewer qubits, but often these lose time. Many subtle tradeoffs. Crude "poly-time" analyses don't care about this, but serious cryptanalysis is much more precise.

Fast quantum operations, part 2

"Hadamard":

$$(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1).$$

$$(a_0, a_1, a_2, a_3) \mapsto$$

$$(a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3).$$

Same for qubit 1:

$$(a_0, a_1, a_2, a_3) \mapsto$$

$$(a_0 + a_2, a_1 + a_3, a_0 - a_2, a_1 - a_3).$$

Qubit 0 and then qubit 1:

$$(a_0, a_1, a_2, a_3) \mapsto$$

$$(a_0 + a_1 + a_2 + a_3, a_0 - a_1 + a_2 - a_3, a_0 + a_1 - a_2 - a_3, a_0 - a_1 - a_2 + a_3).$$

Repeat $n$ times: e.g.,

$$(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1).$$

Measuring $(1, 0, 0, \ldots, 0)$ always produces 0.

Measuring $(1, 1, 1, \ldots, 1)$ can produce any output:

$$\Pr[\text{output } = q] = 1 = 2^n.$$
Fast quantum operations, part 2

“Hadamard”:
\[(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1).\]
\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3).\]

Same for qubit 1:
\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_2, a_1 + a_3, a_0 - a_2, a_1 - a_3).\]

Qubit 0 and then qubit 1:
\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1 + a_2 + a_3, a_0 - a_1 + a_2 - a_3, a_0 + a_1 - a_2 - a_3, a_0 - a_1 - a_2 + a_3).\]
Fast quantum operations, part 2

“Hadamard”:
\[(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1).\]
\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3).\]

Same for qubit 1:
\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_2, a_1 + a_3, a_0 - a_2, a_1 - a_3).\]

Qubit 0 and then qubit 1:
\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3) \mapsto (a_0 + a_1 + a_2 + a_3, a_0 - a_1 + a_2 - a_3, a_0 + a_1 - a_2 - a_3, a_0 - a_1 - a_2 + a_3).\]

Repeat \(n\) times: e.g.,
\[(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1),\]
Measuring \((1, 0, 0, \ldots, 0)\) always produces 0.
Measuring \((1, 1, 1, \ldots, 1)\)
can produce any output:
\[\Pr[\text{output} = q] = 1/2^n.\]
Fast quantum operations, part 2

“Hadamard”:

\[(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1)\].

\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3)\].

Same for qubit 1:

\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_2, a_1 + a_3, a_0 - a_2, a_1 - a_3)\].

Qubit 0 and then qubit 1:

\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1 + a_2 + a_3, a_0 - a_1 + a_2 - a_3, a_0 + a_1 - a_2 - a_3, a_0 - a_1 - a_2 + a_3)\].

Repeat \(n\) times: e.g.,

\[(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1)\].

Measuring \((1, 0, 0, \ldots, 0)\) always produces 0.

Measuring \((1, 1, 1, \ldots, 1)\) can produce any output:

\[\Pr[\text{output} = q] = 1/2^n\].
Fast quantum operations, part 2

“Hadamard”:
\[(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1).\]
\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1, a_0 - a_1, a_2 + a_3, a_2 - a_3).\]

Same for qubit 1:
\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_2, a_1 + a_3, a_0 - a_2, a_1 - a_3).\]

Qubit 0 and then qubit 1:
\[(a_0, a_1, a_2, a_3) \mapsto (a_0 + a_1 + a_2 + a_3, a_0 - a_1 + a_2 - a_3, a_0 + a_1 - a_2 - a_3, a_0 - a_1 - a_2 + a_3).\]

Repeat \(n\) times: e.g.,
\[(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1).\]
Measuring \((1, 0, 0, \ldots, 0)\)
always produces 0.
Measuring \((1, 1, 1, \ldots, 1)\)
can produce any output:
\[\Pr[\text{output} = q] = \frac{1}{2^n}.\]
Aside from “normalization”
(irrelevant to measurement),
have \(\text{Hadamard} = \text{Hadamard}^{-1}\),
so easily work backwards
from “uniform superposition”
\((1, 1, 1, \ldots, 1)\) to “pure state”
\((1, 0, 0, \ldots, 0)\).
Quantum operations, part 2

"Hadamard":

$(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1)$.

$(a_2, a_3) \mapsto (a_1 + a_3, a_0 - a_2, a_1 - a_3)$.

For qubit 1:

$(a_2, a_3) \mapsto (a_0 + a_1 + a_3, a_0 - a_2, a_1 - a_3)$.

And then qubit 1:

$(a_2, a_3) \mapsto (a_0 - a_1 + a_2, a_2 + a_3, a_0 - a_1 + a_2 - a_3, a_0 + a_2 - a_3, a_0 - a_1 - a_2 + a_3)$.

Repeat $n$ times: e.g.,

$(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1)$.

Measuring $(1, 0, 0, \ldots, 0)$ always produces 0.

Measuring $(1, 1, 1, \ldots, 1)$ can produce any output:

$\Pr[\text{output } = q] = 1/2^n$.

Aside from "normalization" (irrelevant to measurement), have Hadamard = Hadamard$^{-1}$, so easily work backwards from "uniform superposition" $(1, 1, 1, \ldots, 1)$ to "pure state" $(1, 0, 0, \ldots, 0)$.

Simon's algorithm

Assume: nonzero $s \in \{0, 1\}$ satisfies $f(x) = f(x \oplus s)$ for every $x \in \{0, 1\}^n$.

Can we find this period $s$, given a fast circuit for $f$?
Fast quantum operations, part 2

Hadamard:
\[(a_0; a_1) \mapsto (a_0 + a_1; a_0 - a_1).\]

\[(a_0; a_1; a_2; a_3) \mapsto (a_0 + a_1 + a_2 + a_3; a_0 - a_1 + a_2 - a_3; a_0 - a_1 - a_2 + a_3; a_0 + a_1 - a_2 - a_3).\]

Repeat \(n\) times: e.g.,
\[(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1).\]

Measuring \((1, 0, 0, \ldots, 0)\) always produces 0.

Measuring \((1, 1, 1, \ldots, 1)\) can produce any output:
\[
\Pr[\text{output} = q] = \frac{1}{2^n}.
\]

Aside from “normalization” (irrelevant to measurement), have Hadamard = Hadamard\(^{-1}\), so easily work backwards from “uniform superposition” \((1, 1, 1, \ldots, 1)\) to “pure state” \((1, 0, 0, \ldots, 0)\).

Simon’s algorithm

Assume: nonzero \(s \in \{0, 1\}\) satisfies \(f(x) = f(x \oplus s)\) for every \(x \in \{0, 1\}^n\).

Can we find this period \(s\), given a fast circuit for \(f\)?
Fast quantum operations, part 2

"Hadamard":
( a_0 ; a_1 ) → ( a_0 + a_1 ; a_0 − a_1 ).
( a_0 ; a_1 ; a_2 ; a_3 ) →
( a_0 + a_1 + a_2 + a_3 ; a_0 − a_1 + a_2 − a_3 ,
a_0 + a_1 − a_2 − a_3 ; a_0 − a_1 − a_2 + a_3 ).

Repeat n times: e.g.,
(1, 0, 0, . . . , 0) → (1, 1, 1, . . . , 1).
Measuring (1, 0, 0, . . . , 0) always produces 0.
Measuring (1, 1, 1, . . . , 1) can produce any output:
Pr[output = q] = 1/2^n.

Aside from “normalization” (irrelevant to measurement),
have Hadamard = Hadamard^{−1},
so easily work backwards from “uniform superposition”
(1, 1, 1, . . . , 1) to “pure state”
(1, 0, 0, . . . , 0).

Simon’s algorithm

Assume: nonzero s ∈{0, 1} satisfies f(x) = f(x ⊕ s)
for every x ∈{0, 1}^n.
Can we find this period s,
given a fast circuit for f?
Repeat \( n \) times: e.g.,
\[(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1).\]

Measuring \((1, 0, 0, \ldots, 0)\)
always produces 0.

Measuring \((1, 1, 1, \ldots, 1)\)
can produce any output:
\[\Pr[\text{output} = q] = 1/2^n.\]

Aside from “normalization”
(irrelevant to measurement),
have Hadamard = Hadamard\(^{-1}\),
so easily work backwards
from “uniform superposition”
\((1, 1, 1, \ldots, 1)\) to “pure state”
\((1, 0, 0, \ldots, 0)\).

Simon’s algorithm

Assume: nonzero \( s \in \{0, 1\}^n \)
satisfies \( f(x) = f(x \oplus s) \)
for every \( x \in \{0, 1\}^n \).
Can we find this period \( s \),
given a fast circuit for \( f \)?
Repeat $n$ times: e.g.,
$(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1)$.

Measuring $(1, 0, 0, \ldots, 0)$
always produces 0.

Measuring $(1, 1, 1, \ldots, 1)$
can produce any output:
$\Pr[\text{output} = q] = 1/2^n$.

Aside from “normalization”
(irrelevant to measurement),
have $\text{Hadamard} = \text{Hadamard}^{-1}$,
so easily work backwards
from “uniform superposition”
$(1, 1, 1, \ldots, 1)$ to “pure state”
$(1, 0, 0, \ldots, 0)$.

Simon’s algorithm
Assume: nonzero $s \in \{0, 1\}^n$
satisfies $f(x) = f(x \oplus s)$
for every $x \in \{0, 1\}^n$.
Can we find this period $s$,
given a fast circuit for $f$?
We don’t have enough data
if $f$ has many periods.
Assume: $\{\text{periods}\} = \{0, s\}$.
Repeat $n$ times: e.g.,
$(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1)$.

Measuring $(1, 0, 0, \ldots, 0)$
always produces 0.

Measuring $(1, 1, 1, \ldots, 1)$
can produce any output:
$\Pr[\text{output } = q] = 1/2^n$.

Aside from “normalization”
(irrelevant to measurement),
have $\text{Hadamard} = \text{Hadamard}^{-1}$,
so easily work backwards
from “uniform superposition”
$(1, 1, 1, \ldots, 1)$ to “pure state”
$(1, 0, 0, \ldots, 0)$.

Simon’s algorithm

Assume: nonzero $s \in \{0, 1\}^n$
satisfies $f(x) = f(x \oplus s)$
for every $x \in \{0, 1\}^n$.
Can we find this period $s$,
given a fast circuit for $f$?

We don’t have enough data
if $f$ has many periods.
Assume: $\{\text{periods}\} = \{0, s\}$.

Traditional solution:
Compute $f$ for many inputs,
sort, analyze collisions.
Success probability is very low
until $\#\text{inputs}$ approaches $2^{n/2}$.
Repeat \( n \) times: e.g.,
\[(1, 0, 0, \ldots, 0) \mapsto (1, 1, 1, \ldots, 1).\]

Measuring \((1, 0, 0, \ldots, 0)\)
produces 0.

Measuring \((1, 1, 1, \ldots, 1)\)
produces any output:
\[
\text{Pr}[\text{output} = q] = \frac{1}{2^n}.
\]

Aside from “normalization” (irrelevant to measurement),

\[\text{Hadamard} = \text{Hadamard}^{-1},\]
so easily work backwards
from “uniform superposition” \((1, 1, 1, \ldots, 1)\) to “pure state” \((1, 0, 0, \ldots, 0)\).

Simon’s algorithm

Assume: nonzero \( s \in \{0, 1\}^n \)
satisfies \( f(x) = f(x \oplus s) \)
for every \( x \in \{0, 1\}^n \).

Can we find this period \( s \),
given a fast circuit for \( f \)?

We don’t have enough data
if \( f \) has many periods.

Assume: \( \{\text{periods}\} = \{0, s\} \).

Traditional solution:
Compute \( f \) for many inputs,
sort, analyze collisions.
Success probability is very low
until \( \# \text{inputs} \) approaches \( 2^{n/2} \).

Simon’s algorithm uses
far fewer qubit operations
if \( n \) is large and
reversibility overhead is low.
Repeat \( n \) times: e.g.,
\[(1, 1, 1, \ldots, 1) \]
\[\mapsto (1, 1, 1, \ldots, 1) \]

output: 
\[1/2^n.\]

Aside from “normalization” (irrelevant to measurement),
Hadamard \(-1\),
work backwards
from “uniform superposition”
(1; 1; 1; \ldots; 1)
to “pure state”
(1; 0; 0; \ldots; 0).

Simon’s algorithm
Assume: nonzero \( s \in \{0, 1\}^n \)
satisfies \( f(x) = f(x \oplus s) \)
for every \( x \in \{0, 1\}^n \).
Can we find this period \( s \),
given a fast circuit for \( f \)?
We don’t have enough data
if \( f \) has many periods.
Assume: \{periods\} = \{0, s\}.

Traditional solution:
Compute \( f \) for many inputs,
sort, analyze collisions.
Success probability is very low
until \#inputs approaches \( 2^n/2 \).
Simon’s algorithm

Assume: nonzero \( s \in \{0, 1\}^n \)
satisfies \( f(x) = f(x \oplus s) \)
for every \( x \in \{0, 1\}^n \).
Can we find this period \( s \),
given a fast circuit for \( f \)?

We don’t have enough data
if \( f \) has many periods.
Assume: \( \{\text{periods}\} = \{0, s\} \).

Traditional solution:
Compute \( f \) for many inputs,
sort, analyze collisions.
Success probability is very low
until \#inputs approaches \( 2^{n/2} \).
Simon’s algorithm

Assume: nonzero \( s \in \{0, 1\}^n \) satisfies \( f(x) = f(x \oplus s) \)
for every \( x \in \{0, 1\}^n \).
Can we find this period \( s \), given a fast circuit for \( f \)?

We don’t have enough data if \( f \) has many periods.
Assume: \( \{\text{periods}\} = \{0, s\} \).

Traditional solution:
Compute \( f \) for many inputs, sort, analyze collisions.
Success probability is very low until \#inputs approaches \( 2^{n/2} \).

Simon’s algorithm uses far fewer qubit operations if \( n \) is large and
reversibility overhead is low.
Simon’s algorithm

Assume: nonzero \( s \in \{0, 1\}^n \)
satisfies \( f(x) = f(x \oplus s) \)
for every \( x \in \{0, 1\}^n \).
Can we find this period \( s \),
given a fast circuit for \( f \)?

We don’t have enough data
if \( f \) has many periods.
Assume: \( \{\text{periods}\} = \{0, s\} \).

Traditional solution:
Compute \( f \) for many inputs,
sort, analyze collisions.
Success probability is very low
until \#inputs approaches \( 2^{n/2} \).

Simon’s algorithm uses
far fewer qubit operations
if \( n \) is large and
reversibility overhead is low.

Say \( f \) maps \( n \) bits to \( m \) bits using
\( z \) “ancilla” bits for reversibility.

Prepare \( n + m + z \) qubits
in pure zero state:
vector \((1, 0, 0, \ldots)\).
Simon’s algorithm

Assume: nonzero $s \in \{0, 1\}^n$
satisfies $f(x) = f(x \oplus s)$
for every $x \in \{0, 1\}^n$.
Can we find this period $s$, given a fast circuit for $f$?

We don’t have enough data
if $f$ has many periods.
Assume: \{periods\} = \{0, s\}.

Traditional solution:
Compute $f$ for many inputs,
sort, analyze collisions.
Success probability is very low
until \#inputs approaches $2^n/2$.

Simon’s algorithm uses
far fewer qubit operations
if $n$ is large and
reversibility overhead is low.

Say $f$ maps $n$ bits to $m$ bits using
$z$ “ancilla” bits for reversibility.

Prepare $n + m + z$ qubits
in pure zero state:
vector $(1, 0, 0, \ldots)$.

Use $n$-fold Hadamard
to move first $n$ qubits
into uniform superposition:
$(1, 1, 1, \ldots, 1, 0, 0, \ldots)$
with $2^n$ entries 1, others 0.
Simon’s algorithm uses far fewer qubit operations if $n$ is large and reversibility overhead is low.

Say $f$ maps $n$ bits to $m$ bits using $z$ “ancilla” bits for reversibility.

Prepare $n + m + z$ qubits in pure zero state: vector $(1, 0, 0, \ldots)$.

Use $n$-fold Hadamard to move first $n$ qubits into uniform superposition: $(1, 1, 1, \ldots, 1, 0, 0, \ldots)$ with $2^n$ entries 1, others 0.
Simon's algorithm uses far fewer qubit operations if \( n \) is large and reversibility overhead is low.

Say \( f \) maps \( n \) bits to \( m \) bits using \( z \) “ancilla” bits for reversibility.

Prepare \( n + m + z \) qubits in pure zero state:

\[
\text{vector } (1, 0, 0, \ldots).
\]

Use \( n \)-fold Hadamard to move first \( n \) qubits into uniform superposition:

\[
(1, 1, 1, \ldots, 1, 0, 0, \ldots)
\]

with \( 2^n \) entries 1, others 0.

Apply fast vector permutation for reversible \( f \) computation:

1 in position \((q, 0, 0)\) moves to position \((q, f(q), 0)\).

Note symmetry between 1 at \((q, f(q), 0)\) and 1 at \((q \oplus s, f(q), 0)\).
Simon’s algorithm uses far fewer qubit operations if $n$ is large and reversibility overhead is low.

Say $f$ maps $n$ bits to $m$ bits using $z$ “ancilla” bits for reversibility.

Prepare $n + m + z$ qubits in pure zero state:
vector $(1, 0, 0, \ldots)$.

Use $n$-fold Hadamard to move first $n$ qubits into uniform superposition:
$(1, 1, 1, \ldots, 1, 0, 0, \ldots)$ with $2^n$ entries 1, others 0.

Apply fast vector permutation for reversible $f$ computation:
1 in position $(q, 0, 0)$ moves to position $(q, f(q), 0)$.

Note symmetry between 1 at $(q, f(q), 0)$ and 1 at $(q \oplus s, f(q), 0)$. 

Simon’s algorithm uses far fewer qubit operations if \( n \) is large and reversibility overhead is low.

Say \( f \) maps \( n \) bits to \( m \) bits using \( z \) “ancilla” bits for reversibility.

Prepare \( n + m + z \) qubits in pure zero state: vector \((1, 0, 0, \ldots)\).

Use \( n \)-fold Hadamard to move first \( n \) qubits into uniform superposition:
\((1, 1, 1, \ldots, 1, 0, 0, \ldots)\) with \(2^n\) entries 1, others 0.

Apply fast vector permutation for reversible \( f \) computation:
1 in position \((q, 0, 0)\) moves to position \((q, f(q), 0)\).

Note symmetry between 1 at \((q, f(q), 0)\) and 1 at \((q \oplus s, f(q), 0)\).
Simon’s algorithm uses far fewer qubit operations if $n$ is large and reversibility overhead is low.

Say $f$ maps $n$ bits to $m$ bits using $z$ “ancilla” bits for reversibility.

Prepare $n + m + z$ qubits in pure zero state: vector $(1, 0, 0, \ldots)$.

Use $n$-fold Hadamard to move first $n$ qubits into uniform superposition: $(1, 1, 1, \ldots, 1, 0, 0, \ldots)$ with $2^n$ entries 1, others 0.

Apply fast vector permutation for reversible $f$ computation: 1 in position $(q, 0, 0)$ moves to position $(q, f(q), 0)$.

Note symmetry between 1 at $(q, f(q), 0)$ and 1 at $(q \oplus s, f(q), 0)$.

Apply $n$-fold Hadamard.
Simon’s algorithm uses far fewer qubit operations if \( n \) is large and reversibility overhead is low.

Say \( f \) maps \( n \) bits to \( m \) bits using \( z \) “ancilla” bits for reversibility.

Prepare \( n + m + z \) qubits in pure zero state: vector \((1, 0, 0, \ldots)\).

Use \( n \)-fold Hadamard to move first \( n \) qubits into uniform superposition: \((1, 1, 1, \ldots, 1, 0, 0, \ldots)\) with \( 2^n \) entries 1, others 0.

Apply fast vector permutation for reversible \( f \) computation: 1 in position \((q, 0, 0)\) moves to position \((q, f(q), 0)\).

Note symmetry between 1 at \((q, f(q), 0)\) and 1 at \((q \oplus s, f(q), 0)\).

Apply \( n \)-fold Hadamard.

Measure. By symmetry, output is orthogonal to \( s \).
Simon’s algorithm uses far fewer qubit operations if $n$ is large and reversibility overhead is low.

Say $f$ maps $n$ bits to $m$ bits using $z$ “ancilla” bits for reversibility.

Prepare $n + m + z$ qubits in pure zero state: vector $(1, 0, 0, \ldots)$.

Use $n$-fold Hadamard to move first $n$ qubits into uniform superposition: $(1, 1, 1, \ldots, 1, 0, 0, \ldots)$ with $2^n$ entries 1, others 0.

Apply fast vector permutation for reversible $f$ computation: 1 in position $(q, 0, 0)$ moves to position $(q, f(q), 0)$.

Note symmetry between 1 at $(q, f(q), 0)$ and 1 at $(q \oplus s, f(q), 0)$.

Apply $n$-fold Hadamard.

Measure. By symmetry, output is orthogonal to $s$.

Repeat $n + 10$ times.

Use Gaussian elimination to (probably) find $s$. 

Applying fast vector permutation for reversible $f$ computation: 1 in position $(q, 0, 0)$ moves to position $(q, f(q), 0)$.

Note symmetry between 1 at $(q, f(q), 0)$ and 1 at $(q \oplus s, f(q), 0)$.

Apply $n$-fold Hadamard.

Measure. By symmetry, output is orthogonal to $s$.

Repeat $n + 10$ times.

Use Gaussian elimination to (probably) find $s$. 
Simon's algorithm uses far fewer qubit operations if \( n \) is large and reversibility overhead is low.

Say \( f \) maps \( n \) bits to \( m \) bits using \( z \) "ancilla" bits for reversibility.

Prepare \( n + m + z \) qubits in pure zero state:

\[
\begin{pmatrix} 1, 0, 0, \ldots \end{pmatrix}.
\]

Apply \( n \)-fold Hadamard to move first \( n \) qubits into uniform superposition:

\[
\begin{pmatrix} 1; 1; 1; \ldots; 1; 0; 0; \ldots \end{pmatrix}
\] with \( 2^n \) entries 1, others 0.

Apply fast vector permutation for reversible \( f \) computation:

1 in position \((q, 0, 0)\)

moves to position \((q, f(q), 0)\).

Note symmetry between

1 at \((q, f(q), 0)\) and
1 at \((q \oplus s, f(q), 0)\).

Apply \( n \)-fold Hadamard.

Measure. By symmetry, output is orthogonal to \( s \).

Repeat \( n + 10 \) times.

Use Gaussian elimination to (probably) find \( s \).

---

Example:

\[
\begin{align*}
f(0) &= 4, \\
f(1) &= 7, \\
f(2) &= 2, \\
f(3) &= 3, \\
f(4) &= 7, \\
f(5) &= 4, \\
f(6) &= 3, \\
f(7) &= 2.
\end{align*}
\]
Simon's algorithm uses far fewer qubit operations if $n$ is large and reversibility overhead is low.

Say $f$ maps $n$ bits to $m$ bits using $z$ "ancilla" bits for reversibility.

Prepare $n + m + z$ qubits in pure zero state:

\[ \text{vector } (1, 0, 0, \ldots) \]

Use $n$-fold Hadamard to move first $n$ qubits into uniform superposition:

\[ (1, 1, 1, \ldots, 1, 0, 0, \ldots) \]

with $2^n$ entries 1, others 0.

Apply fast vector permutation for reversible $f$ computation:
1 in position $(q, 0, 0)$ moves to position $(q, f(q), 0)$.

Note symmetry between 1 at $(q, f(q), 0)$ and 1 at $(q \oplus s, f(q), 0)$.

Apply $n$-fold Hadamard.

Measure. By symmetry, output is orthogonal to $s$.

Repeat $n + 10$ times.

Use Gaussian elimination to (probably) find $s$.

Example, 3 bits to 3 bits:

- $f(0) = 4$
- $f(1) = 7$
- $f(2) = 2$
- $f(3) = 3$
- $f(4) = 7$
- $f(5) = 4$
- $f(6) = 3$
- $f(7) = 2$
Simon's algorithm uses far fewer qubit operations if $n$ is large and reversibility overhead is low. Say $f$ maps $n$ bits to $m$ bits using $z$ "ancilla" bits for reversibility. Prepare $n + m + z$ qubits in pure zero state: vector $(1; 0; 0; \ldots)$. Use $n$-fold Hadamard to move first $n$ qubits into uniform superposition: $(1; 1; 1; \ldots; 1; 0; 0; \ldots)$ with $2^n$ entries 1, others 0.

Apply fast vector permutation for reversible $f$ computation: 1 in position $(q; 0; 0)$ moves to position $(q; f(q); 0)$. Note symmetry between 1 at $(q, f(q), 0)$ and 1 at $(q \oplus s, f(q), 0)$.

Apply $n$-fold Hadamard.

Measure. By symmetry, output is orthogonal to $s$.

Repeat $n + 10$ times. Use Gaussian elimination to (probably) find $s$.

Example, 3 bits to 3 bits:

- $f(0) = 4$
- $f(1) = 7$
- $f(2) = 2$
- $f(3) = 3$
- $f(4) = 7$
- $f(5) = 4$
- $f(6) = 3$
- $f(7) = 2$
Apply fast vector permutation for reversible $f$ computation:
1 in position $(q, 0, 0)$
moves to position $(q, f(q), 0)$.

Note symmetry between
1 at $(q, f(q), 0)$ and
1 at $(q \oplus s, f(q), 0)$.

Apply $n$-fold Hadamard.

Measure. By symmetry,
output is orthogonal to $s$.

Repeat $n + 10$ times.
Use Gaussian elimination
to (probably) find $s$.

Example, 3 bits to 3 bits:

\[
\begin{align*}
  f(0) &= 4. \\
  f(1) &= 7. \\
  f(2) &= 2. \\
  f(3) &= 3. \\
  f(4) &= 7. \\
  f(5) &= 4. \\
  f(6) &= 3. \\
  f(7) &= 2.
\end{align*}
\]
Apply fast vector permutation for reversible $f$ computation:
1 in position $(q, 0, 0)$ moves to position $(q, f(q), 0)$.

Note symmetry between 1 at $(q, f(q), 0)$ and 1 at $(q \oplus s, f(q), 0)$.

Apply $n$-fold Hadamard.

Measure. By symmetry, output is orthogonal to $s$.

Repeat $n + 10$ times.
Use Gaussian elimination to (probably) find $s$.

Example, 3 bits to 3 bits:

$f(0) = 4.$
$f(1) = 7.$
$f(2) = 2.$
$f(3) = 3.$
$f(4) = 7.$
$f(5) = 4.$
$f(6) = 3.$
$f(7) = 2.$
Apply fast vector permutation for reversible $f$ computation:
1 in position $(q, 0, 0)$ moves to position $(q, f(q), 0)$.

Note symmetry between 1 at $(q, f(q), 0)$ and 1 at $(q \oplus s, f(q), 0)$.

Apply $n$-fold Hadamard.

Measure. By symmetry, output is orthogonal to $s$.

Repeat $n + 10$ times.
Use Gaussian elimination to (probably) find $s$.

Example, 3 bits to 3 bits:

\[
\begin{align*}
&f(0) = 4. \\
&f(1) = 7. \\
&f(2) = 2. \\
&f(3) = 3. \\
&f(4) = 7. \\
&f(5) = 4. \\
&f(6) = 3. \\
&f(7) = 2.
\end{align*}
\]

Complete table shows that $f(x) = f(x \oplus 5)$ for all $x$.

Let's watch Simon's algorithm for $f$, using 6 qubits.
Apply fast vector permutation for reversible $f$ computation:

1 in position $(q, 0, 0)$ moves to position $(q, f(q), 0)$.

By symmetry between $f(q), 0$ and $\oplus s, f(q), 0$.

Apply $n$-fold Hadamard.

Measure. By symmetry, output is orthogonal to $s$.

Repeat $n + 10$ times.

Use Gaussian elimination to (probably) find $s$.

Example, 3 bits to 3 bits:

$f(0) = 4.$
$f(1) = 7.$
$f(2) = 2.$
$f(3) = 3.$
$f(4) = 7.$
$f(5) = 4.$
$f(6) = 3.$
$f(7) = 2.$

Complete table shows that $f(x) = f(x \oplus 5)$ for all $x$.

Let's watch Simon's algorithm for $f$, using 6 qubits.

Step 1. Set up pure zero state:

Example, 3 bits to 3 bits:

$f(0) = 4.$
$f(1) = 7.$
$f(2) = 2.$
$f(3) = 3.$
$f(4) = 7.$
$f(5) = 4.$
$f(6) = 3.$
$f(7) = 2.$

Complete table shows that $f(x) = f(x \oplus 5)$ for all $x$.

Let's watch Simon's algorithm for $f$, using 6 qubits.
Apply fast vector permutation for reversible computation: 
1 in position $(q; 0; 0)$ moves to position $(q; f(q); 0)$. 

Note symmetry between 1 at $(q; f(q); 0)$ and 1 at $(q \oplus s; f(q); 0)$. 

Apply $n$-fold Hadamard. 

Measure. By symmetry, output is orthogonal to $s$. 

Repeat $n + 10$ times. 

Use Gaussian elimination to (probably) find $s$. 

---

**Example, 3 bits to 3 bits:**

$f(0) = 4.$

$f(1) = 7.$

$f(2) = 2.$

$f(3) = 3.$

$f(4) = 7.$

$f(5) = 4.$

$f(6) = 3.$

$f(7) = 2.$

Complete table shows that $f(x) = f(x \oplus 5)$ for all $x$. 

Let's watch Simon's algorithm for $f$, using 6 qubits.
Apply fast vector permutation for reversible \( f \) computation:
1 in position \((q; 0; 0)\) moves to position \((q; f(q); 0)\).

Note symmetry between 1 at \((q; f(q); 0)\) and 1 at \((q \oplus s; f(q); 0)\).

Apply \( n \)-fold Hadamard.
Measure. By symmetry, output is orthogonal to \( s \).
Repeat \( n + 10 \) times.
Use Gaussian elimination to (probably) find \( s \).

Example, 3 bits to 3 bits:
\[
\begin{align*}
f(0) &= 4. \\
f(1) &= 7. \\
f(2) &= 2. \\
f(3) &= 3. \\
f(4) &= 7. \\
f(5) &= 4. \\
f(6) &= 3. \\
f(7) &= 2.
\end{align*}
\]

Complete table shows that \( f(x) = f(x \oplus 5) \) for all \( x \).

Let’s watch Simon’s algorithm for \( f \), using 6 qubits.

Step 1. Set up pure zero state:
\[
1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0.
\]
Example, 3 bits to 3 bits:

\[ f(0) = 4. \]
\[ f(1) = 7. \]
\[ f(2) = 2. \]
\[ f(3) = 3. \]
\[ f(4) = 7. \]
\[ f(5) = 4. \]
\[ f(6) = 3. \]
\[ f(7) = 2. \]

Complete table shows that
\[ f(x) = f(x \oplus 5) \] for all \( x \).

Let’s watch Simon’s algorithm for \( f \), using 6 qubits.
Example, 3 bits to 3 bits:

\[
\begin{align*}
    f(0) &= 4. \\
    f(1) &= 7. \\
    f(2) &= 2. \\
    f(3) &= 3. \\
    f(4) &= 7. \\
    f(5) &= 4. \\
    f(6) &= 3. \\
    f(7) &= 2.
\end{align*}
\]

Complete table shows that

\[
f(x) = f(x \oplus 5) \text{ for all } x.
\]

Let’s watch Simon’s algorithm for \( f \), using 6 qubits.

Step 2. Hadamard on qubit 0:

\[
\begin{array}{cccccccc}
    1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Example, 3 bits to 3 bits:

\[
\begin{align*}
  f(0) &= 4. \\
  f(1) &= 7. \\
  f(2) &= 2. \\
  f(3) &= 3. \\
  f(4) &= 7. \\
  f(5) &= 4. \\
  f(6) &= 3. \\
  f(7) &= 2.
\end{align*}
\]

Complete table shows that

\[
f(x) = f(x \oplus 5) \text{ for all } x.
\]

Let's watch Simon's algorithm for \( f \), using 6 qubits.

Step 3. Hadamard on qubit 1:

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Example, 3 bits to 3 bits:

\[ f(0) = 4. \]
\[ f(1) = 7. \]
\[ f(2) = 2. \]
\[ f(3) = 3. \]
\[ f(4) = 7. \]
\[ f(5) = 4. \]
\[ f(6) = 3. \]
\[ f(7) = 2. \]

Complete table shows that 
\[ f(x) = f(x \oplus 5) \] for all \( x \).

Let’s watch Simon’s algorithm for \( f \), using 6 qubits.

Step 4. Hadamard on qubit 2:

\[ 1, 1, 1, 1, 1, 1, 1, 1, \]
\[ 0, 0, 0, 0, 0, 0, 0, 0, \]
\[ 0, 0, 0, 0, 0, 0, 0, 0, \]
\[ 0, 0, 0, 0, 0, 0, 0, 0, \]
\[ 0, 0, 0, 0, 0, 0, 0, 0, \]
\[ 0, 0, 0, 0, 0, 0, 0, 0, \]
Example, 3 bits to 3 bits:

\[ f(0) = 4. \]
\[ f(1) = 7. \]
\[ f(2) = 2. \]
\[ f(3) = 3. \]
\[ f(4) = 7. \]
\[ f(5) = 4. \]
\[ f(6) = 3. \]
\[ f(7) = 2. \]

Complete table shows that
\[ f(x) = f(x \oplus 5) \] for all \( x \).

Let’s watch Simon’s algorithm
for \( f \), using 6 qubits.
Example, 3 bits to 3 bits:

\[
\begin{align*}
  f(0) &= 4. \\
  f(1) &= 7. \\
  f(2) &= 2. \\
  f(3) &= 3. \\
  f(4) &= 7. \\
  f(5) &= 4. \\
  f(6) &= 3. \\
  f(7) &= 2. \\
\end{align*}
\]

Complete table shows that
\[
  f(x) = f(x \oplus 5) \text{ for all } x.
\]

Let's watch Simon's algorithm for \( f \), using 6 qubits.

Step 6. Hadamard on qubit 0:

\[
\begin{align*}
  0, 0, 0, 0, 0, 0, 0, \\
  0, 0, 0, 0, 0, 0, 0, \\
  0, 0, 1, 1, 0, 0, 1, \bar{1}, \\
  0, 0, 1, \bar{1}, 0, 0, 1, 1, \\
  1, 1, 0, 0, 1, \bar{1}, 0, 0, \\
  0, 0, 0, 0, 0, 0, 0, 0, \\
  0, 0, 0, 0, 0, 0, 0, 0, \\
  1, \bar{1}, 0, 0, 1, 1, 0, 0.
\end{align*}
\]

Notation: \( \bar{1} = -1 \).
Example, 3 bits to 3 bits:

\[ f(0) = 4. \]
\[ f(1) = 7. \]
\[ f(2) = 2. \]
\[ f(3) = 3. \]
\[ f(4) = 7. \]
\[ f(5) = 4. \]
\[ f(6) = 3. \]
\[ f(7) = 2. \]

Complete table shows that
\[ f(x) = f(x \oplus 5) \] for all \( x \).

Let’s watch Simon’s algorithm for \( f \), using 6 qubits.

Step 7. Hadamard on qubit 1:

\[
\begin{align*}
0, 0, 0, 0, 0, 0, 0, 0, \\
0, 0, 0, 0, 0, 0, 0, 0, \\
1, 1, \overline{1}, \overline{1}, 1, 1, \overline{1}, \overline{1}, \\
1, \overline{1}, \overline{1}, 1, 1, 1, \overline{1}, \overline{1}, \\
1, 1, 1, 1, 1, \overline{1}, 1, \overline{1}, \\
0, 0, 0, 0, 0, 0, 0, 0, \\
0, 0, 0, 0, 0, 0, 0, 0, \\
1, \overline{1}, \overline{1}, \overline{1}, 1, 1, 1, 1.
\end{align*}
\]
Example, 3 bits to 3 bits:

\[ f(0) = 4. \]
\[ f(1) = 7. \]
\[ f(2) = 2. \]
\[ f(3) = 3. \]
\[ f(4) = 7. \]
\[ f(5) = 4. \]
\[ f(6) = 3. \]
\[ f(7) = 2. \]

Complete table shows that
\[ f(x) = f(x \oplus 5) \text{ for all } x. \]

Let’s watch Simon’s algorithm for \( f \), using 6 qubits.

Step 8. Hadamard on qubit 2:

\[
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
2, 0, 2, 0, 0, 2, 0, 2,
2, 0, 2, 0, 0, 2, 0, 2,
2, 0, 2, 0, 0, 2, 0, 2,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
2, 0, 2, 0, 0, 2, 0, 2.
\]
Example, 3 bits to 3 bits:

\[
\begin{align*}
    f(0) &= 4. \\
    f(1) &= 7. \\
    f(2) &= 2. \\
    f(3) &= 3. \\
    f(4) &= 7. \\
    f(5) &= 4. \\
    f(6) &= 3. \\
    f(7) &= 2. \\
\end{align*}
\]

Complete table shows that
\[ f(x) = f(x \oplus 5) \] for all \( x \).

Let’s watch Simon’s algorithm for \( f \), using 6 qubits.

**Step 8. Hadamard on qubit 2:**

\[
\begin{align*}
    0, 0, 0, 0, 0, 0, 0, 0, \\
    0, 0, 0, 0, 0, 0, 0, 0, \\
    2, 0, \bar{2}, 0, 0, \bar{2}, 0, \bar{2}, \\
    2, 0, \bar{2}, 0, 0, \bar{2}, 0, 2, \\
    2, 0, 2, 0, 0, 2, 0, 2, \\
    0, 0, 0, 0, 0, 0, 0, 0, \\
    0, 0, 0, 0, 0, 0, 0, 0, \\
    2, 0, 2, 0, 0, \bar{2}, 0, \bar{2}.
\end{align*}
\]

**Step 9. Measure.**

First 3 qubits are uniform random vector orthogonal to 101: i.e., 000, 010, 101, or 111.
Example, 3 bits to 3 bits:

\[ f(0) = 4. \]
\[ f(1) = 7. \]
\[ f(2) = 2. \]
\[ f(3) = 3. \]
\[ f(4) = 7. \]
\[ f(5) = 4. \]
\[ f(6) = 3. \]
\[ f(7) = 2. \]

The table shows that \( f(x \oplus 5) \) for all \( x \).

Watch Simon's algorithm using 6 qubits.

Step 8. Hadamard on qubit 2:

\[
\begin{align*}
0, & 0, 0, 0, 0, 0, 0, 0, \\
0, & 0, 0, 0, 0, 0, 0, 0, \\
2, & 0, 2, 0, 0, 2, 0, 2, \\
2, & 0, 2, 0, 0, 2, 0, 2, \\
2, & 0, 2, 0, 0, 2, 0, 2, \\
0, & 0, 0, 0, 0, 0, 0, 0, \\
0, & 0, 0, 0, 0, 0, 0, 0, \\
2, & 0, 2, 0, 0, 2, 0, 2.
\end{align*}
\]

Step 9. Measure.
First 3 qubits are uniform random vector orthogonal to 101: i.e., 000, 010, 101, or 111.

Grover's algorithm

Assume: unique \( s \in \{0, 1\}^n \) has \( f(s) \).

Traditional algorithm to find \( s \):
compute \( f \) for many inputs,
hope to find output 0.
Success probability is very low
until \#inputs approaches \( 2^n \).
Example, 3 bits to 3 bits:

\( f(0) = 4. \)

\( f(1) = 7. \)

\( f(2) = 2. \)

\( f(3) = 3. \)

\( f(4) = 7. \)

\( f(5) = 4. \)

\( f(6) = 3. \)

\( f(7) = 2. \)

Complete table shows that \( f(x) = f(x \oplus 5) \) for all \( x \).

Let's watch Simon's algorithm for \( f \), using 6 qubits.

Step 8. Hadamard on qubit 2:

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & \bar{2} & 0 & 0 & \bar{2} & 0 & \bar{2} \\
2 & 0 & \bar{2} & 0 & 0 & \bar{2} & 0 & 2 \\
2 & 0 & 2 & 0 & 0 & 2 & 0 & 2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 2 & 0 & 0 & \bar{2} & 0 & \bar{2}
\end{array}
\]

Step 9. Measure.

First 3 qubits are uniform random vector orthogonal to 101: i.e., 000, 010, 101, or 111.

Grover’s algorithm

Assume: unique \( s \in \{0, 1\}^n \) has \( f(s) = 0. \)

Traditional algorithm to find \( s \):

compute \( f \) for many inputs,

hope to find output 0.

Success probability is very low

until \( \# \) inputs approaches \( 2^n \).
Example, 3 bits to 3 bits:

\[ f(0) = 4. \]
\[ f(1) = 7. \]

\[ 4 < 3 < 7 < 2 < 3 < 2 < 4 < 2. \]

\[ f(2) = 2. \]
\[ f(3) = 3. \]
\[ f(4) = 7. \]
\[ f(5) = 4. \]
\[ f(6) = 3. \]
\[ f(7) = 2. \]

Complete table shows that
\[ f(x) = f(x \oplus 5) \]
for all \( x \).

Let's watch Simon's algorithm for
\( f \), using 6 qubits.

**Grover’s algorithm**

Assume: unique \( s \in \{0, 1\}^n \) has \( f(s) = 0 \).

Traditional algorithm to find \( s \):
compute \( f \) for many inputs,
hope to find output 0.
Success probability is very low
until \#inputs approaches \( 2^n \).
Step 8. Hadamard on qubit 2:

0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0,
2, 0, 2, 0, 0, 2, 0,
2, 0, 2, 0, 0, 2, 0,
0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0,
2, 0, 2, 0, 0, 2, 0.

Step 9. Measure.

First 3 qubits are uniform random vector orthogonal to 101: i.e., 000, 010, 101, or 111.

Grover’s algorithm

Assume: unique \( s \in \{0, 1\}^n \) has \( f(s) = 0 \).

Traditional algorithm to find \( s \): compute \( f \) for many inputs, hope to find output 0. Success probability is very low until #inputs approaches \( 2^n \).
Step 8. Hadamard on qubit 2:

0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
2, 0, 2, 0, 0, 2, 0, 2,
2, 0, 2, 0, 0, 2, 0, 2,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
2, 0, 2, 0, 0, 2, 0, 2.

Step 9. Measure.

First 3 qubits are uniform random vector orthogonal to 101: i.e., 000, 010, 101, or 111.

Grover’s algorithm

Assume: unique \( s \in \{0, 1\}^n \) has \( f(s) = 0 \).

Traditional algorithm to find \( s \): compute \( f \) for many inputs, hope to find output 0. Success probability is very low until \( \# \) inputs approaches \( 2^n \).

Grover’s algorithm takes only \( 2^{n/2} \) reversible computations of \( f \).

Typically: reversibility overhead is small enough that this easily beats traditional algorithm.
Grover’s algorithm

Assume: unique \( s \in \{0, 1\}^n \)
has \( f(s) = 0 \).

Traditional algorithm to find \( s \):
compute \( f \) for many inputs,
hope to find output 0.
Success probability is very low
until \#inputs approaches \( 2^n \).

Grover’s algorithm takes only \( 2^{n/2} \)
reversible computations of \( f \).

Typically: reversibility overhead
is small enough that this
easily beats traditional algorithm.

Measure.
With high probability this finds \( s \).

Start from uniform superposition
over all \( n \)-bit strings \( q \).

Step 1:

\[
\begin{align*}
\text{set } a &\leftarrow b \\
\text{where } b_q &= -a_q \\
\text{if } f(q) = 0, \\
\text{otherwise.}
\end{align*}
\]

This is fast.

Step 2:

Negate \( a \) around its average.
This is also fast.

Repeat Step 1 + Step 2
about \( 0.58 \cdot 2^{0.5 \cdot n} \)
times.

Measure the \( n \) qubits.
With high probability this finds \( s \).
Grover's algorithm

Assume: unique $s \in \{0, 1\}^n$ has $f(s) = 0$.

Traditional algorithm to find $s$: compute $f$ for many inputs, hope to find output 0.
Success probability is very low until $\#$inputs approaches $2^n$.

Grover's algorithm takes only $2^{n/2}$ reversible computations of $f$.

Typically: reversibility overhead is small enough that this easily beats traditional algorithm.

Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$

$$b_q = a_q \text{ otherwise}.$$ 

This is fast.

Step 2: "Grover diffusion". Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Step 8. Hadamard on qubit 2:

0; 0; 0; 0; 0; 0; 0; 0;
0; 0; 0; 0; 0; 0; 0; 0;
2; 0; 2; 0; 0; 2; 0; ...
2; 0; 2:

Step 9. Measure.
First 3 qubits are uniform random vector orthogonal to 101: i.e.,
000, 010, 101, or 111.
Grover's algorithm
Assume: unique $s \in \{0, 1\}^n$ has $f(s) = 0$.

Traditional algorithm to find $s$: compute $f$ for many inputs, hope to find output 0.
Success probability is very low until $\#$inputs approaches $2^n$.

Grover's algorithm takes only $2^{n/2}$ reversible computations of $f$.
Typically: reversibility overhead is small enough that this easily beats traditional algorithm.

Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where $b_q = -a_q$ if $f(q) = 0$, $b_q = a_q$ otherwise.
This is fast.

Step 2: “Grover diffusion”. Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Grover’s algorithm

Assume: unique $s \in \{0, 1\}^n$ has $f(s) = 0$.

Traditional algorithm to find $s$:
compute $f$ for many inputs,
hope to find output 0.
Success probability is very low until \#inputs approaches $2^n$.

Grover’s algorithm takes only $2^{n/2}$ reversible computations of $f$.
Typically: reversibility overhead is small enough that this easily beats traditional algorithm.

Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where
$\quad b_q = -a_q$ if $f(q) = 0$,
$\quad b_q = a_q$ otherwise.
This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2
about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 
Grover's algorithm

Assume: unique \( s \in \{0, 1\}^n \) has \( f(s) = 0 \).

Traditional algorithm to find \( s \): compute \( f \) for many inputs, hope to find output \( 0 \).
Success probability is very low until \( \# \text{inputs} \) approaches \( 2^n \).

Grover's algorithm takes only \( 2^{n/2} \) reversible computations of \( f \).

Typically: reversibility overhead is small enough that this beats traditional algorithm.

Start from uniform superposition over all \( n \)-bit strings \( q \).

Step 1: Set \( a \leftarrow b \) where
\( b_q = -a_q \) if \( f(q) = 0 \),
\( b_q = a_q \) otherwise.
This is fast.

Step 2: “Grover diffusion”. Negate \( a \) around its average.
This is also fast.

Repeat Step 1 + Step 2 about \( 0.58 \cdot 2^{0.5n} \) times.

Measure the \( n \) qubits.
With high probability this finds \( s \).
Grover's algorithm

Assume: unique \( s \in \{0, 1\}^n \)

has \( f(s) = 0 \).

Traditional algorithm to find \( s \):
compute \( f \) for many inputs, hope to find output 0.
Success probability is very low until \( \#\text{inputs} \) approaches \( 2^n \).

Grover's algorithm takes only \( 2^n \) reversible computations of \( f \).
Typically: reversibility overhead is small enough that this easily beats traditional algorithm.

Start from uniform superposition over all \( n \)-bit strings \( q \).

Step 1: Set \( a \leftarrow b \) where
\[
  b_q = -a_q \text{ if } f(q) = 0, \quad b_q = a_q \text{ otherwise.}
\]
This is fast.

Step 2: “Grover diffusion”.
Negate \( a \) around its average.
This is also fast.

Repeat Step 1 + Step 2 about \( 0.58 \cdot 2^{0.5n} \) times.

Measure the \( n \) qubits.
With high probability this finds \( s \).
Grover's algorithm

Assume: unique $s \in \{0; 1\}^n$ has $f(s) = 0$.

Traditional algorithm to find $s$:
compute $f$ for many inputs, hope to find output 0.
Success probability is very low until $\#\text{inputs} \approx 2^n$.

Grover's algorithm takes only $2^n$ reversible computations of $f$.
Typically: reversibility overhead is small enough that this easily beats traditional algorithm.

Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$b_q = -a_q$ if $f(q) = 0$,
$b_q = a_q$ otherwise.

This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$.
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$

$$b_q = a_q \text{ otherwise.}$$

This is fast.

Step 2: “Grover diffusion”. Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after 0 steps:
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

- $b_q = -a_q$ if $f(q) = 0$,
- $b_q = a_q$ otherwise.

This is fast.

Step 2: “Grover diffusion”.

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after Step 1:
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where
\[ b_q = -a_q \text{ if } f(q) = 0, \]
\[ b_q = a_q \text{ otherwise}. \]
This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 
Start from uniform superposition over all \( n \)-bit strings \( q \).

Step 1: Set \( a \leftarrow b \) where
\[
b_q = -a_q \text{ if } f(q) = 0, \\
b_q = a_q \text{ otherwise.}
\]
This is fast.

Step 2: “Grover diffusion”. Negate \( a \) around its average. This is also fast.

Repeat Step 1 + Step 2 about \( 0.58 \cdot 2^{0.5n} \) times.

Measure the \( n \) qubits. With high probability this finds \( s \).
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$
$$b_q = a_q \text{ otherwise.}$$

This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $2 \times (\text{Step 1} + \text{Step 2})$:
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$

$$b_q = a_q \text{ otherwise.}$$

This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average. This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $3 \times (\text{Step 1} + \text{Step 2})$: 

\begin{center}
\begin{tikzpicture}
\end{tikzpicture}
\end{center}
Start from uniform superposition over all \( n \)-bit strings \( q \).

Step 1: Set \( a \leftarrow b \) where
\[
b_q = -a_q \text{ if } f(q) = 0,
b_q = a_q \text{ otherwise.}
\]
This is fast.

Step 2: “Grover diffusion”. Negate \( a \) around its average.
This is also fast.

Repeat Step 1 + Step 2 about \( 0.58 \cdot 2^{0.5n} \) times.

Measure the \( n \) qubits.
With high probability this finds \( s \).
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

- $b_q = -a_q$ if $f(q) = 0$,
- $b_q = a_q$ otherwise.

This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$
for an example with $n = 12$
after $5 \times (\text{Step 1} + \text{Step 2})$: 

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{graph.png}
\caption{Normalized graph of $q \mapsto a_q$
for an example with $n = 12$
after $5 \times (\text{Step 1} + \text{Step 2})$.}
\end{figure}
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where
\[ b_q = -a_q \text{ if } f(q) = 0, \]
\[ b_q = a_q \text{ otherwise}. \]
This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $6 \times (\text{Step 1} + \text{Step 2})$: 

-1.0
-0.5
0.0
0.5
1.0
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

\[ b_q = -a_q \text{ if } f(q) = 0, \]
\[ b_q = a_q \text{ otherwise.} \]

This is fast.

Step 2: “Grover diffusion”.

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $7 \times (\text{Step 1 + Step 2})$: 

-1.0
-0.5
0.0
0.5
1.0
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$
$$b_q = a_q \text{ otherwise.}$$

This is fast.

Step 2: “Grover diffusion”.

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $8 \times (\text{Step 1 + Step 2})$: 

---

![Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $8 \times (\text{Step 1 + Step 2})$]
Start from uniform superposition over all \( n \)-bit strings \( q \).

Step 1: Set \( a \leftarrow b \) where
\[
b_q = -a_q \text{ if } f(q) = 0, \\
b_q = a_q \text{ otherwise.}
\]
This is fast.

Step 2: “Grover diffusion”.
Negate \( a \) around its average.
This is also fast.

Repeat Step 1 + Step 2 about \( 0.58 \cdot 2^{0.5n} \) times.

Measure the \( n \) qubits.
With high probability this finds \( s \).
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$b_q = -a_q$ if $f(q) = 0$,
$b_q = a_q$ otherwise.

This is fast.

Step 2: “Grover diffusion”.

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $10 \times (\text{Step 1} + \text{Step 2})$: 

```
-1.0
-0.5
0.0
0.5
1.0
```
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$

$$b_q = a_q \text{ otherwise.}$$

This is fast.

Step 2: “Grover diffusion”.

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $11 \times (\text{Step 1 + Step 2})$:
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where
\[ b_q = -a_q \text{ if } f(q) = 0, \]
\[ b_q = a_q \text{ otherwise}. \]
This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $12 \times (\text{Step 1} + \text{Step 2})$: 

\[ \begin{array}{c|c|c|c|c|c} 
  & -1.0 & -0.5 & 0.0 & 0.5 & 1.0 \\
\hline
0.0 & & & & & \\
\hline
1.0 & & & & & \\
\end{array} \]
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$
\begin{align*}
    b_q &= -a_q & \text{if } f(q) = 0, \\
    b_q &= a_q & \text{otherwise.}
\end{align*}
$$

This is fast.

Step 2: “Grover diffusion”.

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

| Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $13 \times (\text{Step 1} + \text{Step 2}):$ | }
Start from uniform superposition over all \( n \)-bit strings \( q \).

Step 1: Set \( a \leftarrow b \) where
\[
b_q = -a_q \quad \text{if} \quad f(q) = 0, \\
b_q = a_q \quad \text{otherwise}.
\]
This is fast.

Step 2: “Grover diffusion”. Negate \( a \) around its average. This is also fast.

Repeat Step 1 + Step 2 about \( 0.58 \cdot 2^{0.5n} \) times.

Measure the \( n \) qubits.
With high probability this finds \( s \).
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where $b_q = -a_q$ if $f(q) = 0$, $b_q = a_q$ otherwise. This is fast.

Step 2: “Grover diffusion”. Negate $a$ around its average. This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits. With high probability this finds $s$. Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $15 \times (\text{Step 1} + \text{Step 2})$: 

\[\begin{array}{c}
-1.0 \\
-0.5 \\
0.0 \\
0.5 \\
1.0
\end{array}\]
Start from uniform superposition over all \( n \)-bit strings \( q \).

Step 1: Set \( a \leftarrow b \) where

\[
b_q = -a_q \text{ if } f(q) = 0, \quad b_q = a_q \text{ otherwise.}
\]

This is fast.

Step 2: “Grover diffusion”.
Negate \( a \) around its average.
This is also fast.

Repeat Step 1 + Step 2 about \( 0.58 \cdot 2^{0.5n} \) times.

Measure the \( n \) qubits.
With high probability this finds \( s \).

Normalized graph of \( q \mapsto a_q \) for an example with \( n = 12 \) after \( 16 \times (\text{Step 1} + \text{Step 2}) \):
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

\[ b_q = -a_q \text{ if } f(q) = 0, \]
\[ b_q = a_q \text{ otherwise}. \]

This is fast.

Step 2: “Grover diffusion”. Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $17 \times (\text{Step 1} + \text{Step 2})$: 

![Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $17 \times (\text{Step 1} + \text{Step 2})$]
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$

$$b_q = a_q \text{ otherwise.}$$

This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $18 \times (\text{Step 1} + \text{Step 2})$: 

![Normalized graph of $q \mapsto a_q$](image)
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where
$$b_q = -a_q \text{ if } f(q) = 0,$$
$$b_q = a_q \text{ otherwise.}$$
This is fast.

Step 2: “Grover diffusion”. Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$.
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

\[
b_q = -a_q \text{ if } f(q) = 0,
\]
\[
b_q = a_q \text{ otherwise}.
\]

This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $20 \times (\text{Step 1} + \text{Step 2})$: 

\[
\begin{array}{c}
\text{Normalized graph of } q \mapsto a_q \\
\text{for an example with } n = 12 \\
after 20 \times (\text{Step 1} + \text{Step 2}): \\
\end{array}
\]
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$b_q = -a_q$ if $f(q) = 0$,
$b_q = a_q$ otherwise.

This is fast.

Step 2: “Grover diffusion”.

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

\[ b_q = -a_q \text{ if } f(q) = 0, \]

\[ b_q = a_q \text{ otherwise.} \]

This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$
for an example with $n = 12$
after $30 \times (\text{Step 1 + Step 2})$: 

\begin{center}
\begin{tikzpicture}
\end{tikzpicture}
\end{center}
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where
\[ b_q = -a_q \text{ if } f(q) = 0, \]
\[ b_q = a_q \text{ otherwise.} \]
This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$.

Good moment to stop, measure.

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $35 \times (\text{Step 1} + \text{Step 2})$:
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$

$$b_q = a_q \text{ otherwise}.$$  

This is fast.

Step 2: “Grover diffusion”. 

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$.  

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $40 \times (\text{Step 1} + \text{Step 2})$: 

\begin{axis}[
    xmin=-1.0, xmax=1.0,
    ymin=-1.0, ymax=1.0,
    xtick={-1.0,-0.5,0.0,0.5,1.0},
    ytick={-1.0,-0.5,0.0,0.5,1.0},
] 
\end{axis}
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

- $b_q = -a_q$ if $f(q) = 0$,
- $b_q = a_q$ otherwise.

This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $45 \times (\text{Step 1} + \text{Step 2})$: 

\begin{center}
\begin{tikzpicture}
\begin{axis}[
    xmin=-1.0, xmax=1.0,
    ymin=-1.0, ymax=1.0,
    xtick={-1.0,-0.5,0.0,0.5,1.0},
    ytick={-1.0,-0.5,0.0,0.5,1.0},
    grid=major,
    xlabel=$q$,
    ylabel=$a_q$,
]
\end{axis}
\end{tikzpicture}
\end{center}
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where $b_q = -a_q$ if $f(q) = 0$, $b_q = a_q$ otherwise. This is fast.

Step 2: “Grover diffusion”. Negate $a$ around its average. This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits. With high probability this finds $s$.

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $50 \times (\text{Step 1 + Step 2})$: Traditional stopping point.
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$
$$b_q = a_q \text{ otherwise.}$$

This is fast.

Step 2: “Grover diffusion”. Negate $a$ around its average. This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

---

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $60 \times (\text{Step 1} + \text{Step 2})$: 

![Normalized graph of $q \mapsto a_q$](image_url)
Start from uniform superposition over all \( n \)-bit strings \( q \).

Step 1: Set \( a \leftarrow b \) where 
\[
b_q = -a_q \text{ if } f(q) = 0,
\]
\[
b_q = a_q \text{ otherwise.}
\]
This is fast.

Step 2: “Grover diffusion”. 
Negate \( a \) around its average.
This is also fast.

Repeat Step 1 + Step 2 about \( 0.58 \cdot 2^{0.5n} \) times.

Measure the \( n \) qubits.
With high probability this finds \( s \).

Normalized graph of \( q \mapsto a_q \) for an example with \( n = 12 \) after \( 70 \times \) (Step 1 + Step 2):
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$b_q = -a_q \text{ if } f(q) = 0,$$
$$b_q = a_q \text{ otherwise.}$$

This is fast.

Step 2: “Grover diffusion”. Negate $a$ around its average. This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits. With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $80 \times (\text{Step 1} + \text{Step 2})$: 

-1.0
-0.5
0.0
0.5
1.0
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a ← b$ where

- $b_q = -a_q$ if $f(q) = 0$,
- $b_q = a_q$ otherwise.

This is fast.

Step 2: “Grover diffusion”.

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $90 \times (\text{Step 1} + \text{Step 2})$: 

![Normalized graph](image)
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where
\[ b_q = -a_q \text{ if } f(q) = 0, \]
\[ b_q = a_q \text{ otherwise.} \]
This is fast.

Step 2: “Grover diffusion”.
Negate $a$ around its average.
This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.
With high probability this finds $s$.

Very bad stopping point.
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

- $b_q = -a_q$ if $f(q) = 0$, otherwise.

This is fast.

“Grover diffusion”.
Negate $a$ around its average. Also fast.

Step 1 + Step 2 act linearly on this vector.

Easily compute eigenvalues and powers of this linear map to understand evolution of state of Grover’s algorithm.

⇒ Probability is $\approx 1$ after $\approx (\frac{\pi}{4})^2 \cdot 58 \cdot 2^{0.5n}$ iterations.

Measure the $n$ qubits.

With high probability this finds $s$. 
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where $b_q = -a_q$ if $f(q) = 0$, $b_q = a_q$ otherwise. This is fast.

Step 2: “Grover diffusion”. Negate $a$ around its average. This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits. With high probability this finds $s$.

$q \mapsto a_q$ is completely described by a vector of two numbers (with fixed multiplicities):

1. $a_q$ for roots $q$;
2. $a_q$ for non-roots $q$.

Step 1 + Step 2 act linearly on this vector.

Easily compute eigenvalues and powers of this linear map to understand evolution of state of Grover’s algorithm.

$\Rightarrow$ Probability is $\approx 1$ after $\approx (\pi/4)2^{0.5n}$ iterations.
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

$$ b_q = -a_q \quad \text{if} \quad f(q) = 0, $$

$$ b_q = a_q \quad \text{otherwise}. $$

This is fast.

Step 2: "Grover diffusion".

Negate $a$ around its average.

This is also fast.

Repeat Step 1 + Step 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$.

$q \mapsto a_q$ is completely described by a vector of two numbers (with fixed multiplicities):

(1) $a_q$ for roots $q$;
(2) $a_q$ for non-roots $q$.

Step 1 + Step 2 act linearly on this vector.

Easily compute eigenvalues and powers of this linear map to understand evolution of state of Grover’s algorithm.

$\Rightarrow$ Probability is $\approx 1$ after $\approx (\pi/4)2^{0.5n}$ iterations.

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $100 \times (\text{Step 1} + \text{Step 2})$:

Very bad stopping point.

Normalized graph of $q \mapsto a_q$ for an example with $n = 12$ after $100 \times (\text{Step 1} + \text{Step 2})$:
Normalized graph of $q \mapsto a_q$
for an example with $n = 12$
after $100 \times (\text{Step 1} + \text{Step 2})$:

$q \mapsto a_q$ is completely described
by a vector of two numbers
(with fixed multiplicities):
(1) $a_q$ for roots $q$;
(2) $a_q$ for non-roots $q$.

Step 1 + Step 2
act linearly on this vector.

Easily compute eigenvalues
and powers of this linear map
to understand evolution
of state of Grover’s algorithm.

$\Rightarrow$ Probability is $\approx 1$
after $\approx (\pi/4)2^{0.5n}$ iterations.

Very bad stopping point.
Normalized graph of $q \mapsto a_q$

Example with $n = 12$

$10 \times \text{(Step 1 + Step 2)}$:

Very bad stopping point.

$q \mapsto a_q$ is completely described by a vector of two numbers (with fixed multiplicities):

1. $a_q$ for roots $q$;
2. $a_q$ for non-roots $q$.

Step 1 + Step 2 act linearly on this vector.

Easily compute eigenvalues and powers of this linear map to understand evolution of state of Grover’s algorithm.

$\Rightarrow$ Probability is $\approx 1$ after $\approx (\pi/4)2^{0.5n}$ iterations.

Notes on provability

Textbook algorithm analysis:

Proof of correctness

New algorithm

↑ ↑

↓ ↓

Proof of run time

Mislead students into thinking that best algorithm = best proven algorithm.
$q \mapsto a_q$ is completely described by a vector of two numbers (with fixed multiplicities):

1. $a_q$ for roots $q$;
2. $a_q$ for non-roots $q$.

Step 1 + Step 2 act linearly on this vector.

Easily compute eigenvalues and powers of this linear map to understand evolution of state of Grover’s algorithm.

$\Rightarrow$ Probability is $\approx 1$ after $\approx (\pi/4)2^{0.5n}$ iterations.
$q \mapsto a_q$ is completely described by a vector of two numbers (with fixed multiplicities):

1. $a_q$ for roots $q$;
2. $a_q$ for non-roots $q$.

Step 1 + Step 2 act linearly on this vector.

Easily compute eigenvalues and powers of this linear map to understand evolution of state of Grover’s algorithm.

$\Rightarrow$ Probability is $\approx 1$ after $\approx (\pi/4)2^{0.5n}$ iterations.

Notes on provability

Textbook algorithm analysis:

Proof of correctness
New algorithm
Proof of run time

Mislead students into thinking that best algorithm = best proven algorithm.
$q \mapsto a_q$ is completely described by a vector of two numbers (with fixed multiplicities):

1. $a_q$ for roots $q$;
2. $a_q$ for non-roots $q$.

Step 1 + Step 2 act linearly on this vector.

Easily compute eigenvalues and powers of this linear map to understand evolution of state of Grover’s algorithm.

$\Rightarrow$ Probability is $\approx 1$ after $\approx (\pi/4)2^{0.5n}$ iterations.

Notes on provability
Textbook algorithm analysis:

Proof of correctness
New algorithm
Proof of run time

Mislead students into thinking that best algorithm = best proven algorithm.
is completely described by a vector of two numbers (with fixed multiplicities):
(1) \( a_q \) for roots \( q \);
(2) \( a_q \) for non-roots \( q \).

Step 1 + Step 2 act linearly on this vector.

Easily compute eigenvalues of this linear map to understand evolution of Grover’s algorithm.

⇒ Probability is \( \approx 1 \)
after \( \approx (\pi/4)2^{0.5n} \) iterations.

Notes on provability

Textbook algorithm analysis:

Proof of correctness

New algorithm

Proof of run time

Mislead students into thinking that best algorithm = best proven algorithm.

Reality: state-of-the-art cryptanalytic algorithms are almost never proven.
q \mapsto a_q is completely described by a vector of two numbers (with fixed multiplicities):
(1) $a_q$ for roots $q$;
(2) $a_q$ for non-roots $q$.

Step 1 + Step 2 acts linearly on this vector.

Easily compute eigenvalues and powers of this linear map to understand evolution of state of Grover's algorithm.

$\Rightarrow$ Probability is $\approx 1$ after $\approx (\frac{1}{4})^20_5n$ iterations.

Notes on provability

Textbook algorithm analysis:

Proof of correctness

New algorithm

Proof of run time

Mislead students into thinking that best algorithm =

best proven algorithm.
Notes on provability

Textbook algorithm analysis:

Proof of correctness

New algorithm

Proof of run time

Mislead students into thinking that best algorithm = best proven algorithm.

Reality: state-of-the-art cryptanalytic algorithms are almost never proven.
Notes on provability

Textbook algorithm analysis:

- Proof of correctness
- New algorithm
- Proof of run time

Mislead students into thinking that best algorithm = best proven algorithm.

Reality: state-of-the-art cryptanalytic algorithms are almost never proven.
Notes on provability

Textbook algorithm analysis:

Proof of correctness

↓

New algorithm

↓

Proof of run time

Mislead students into thinking that best algorithm = best proven algorithm.

Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: “Work harder, find proofs!”
Notes on provability

Textbook algorithm analysis:

Proof of correctness

New algorithm

Proof of run time

Mislead students into thinking that best algorithm = best \textit{proven} algorithm.

Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: “Work harder, find proofs!”

Consensus of the experts: proofs probably do not exist for most of these algorithms. So demanding proofs is silly.
Notes on provability

Textbook algorithm analysis:

- Proof of correctness
- New algorithm
- Proof of run time

Mislead students into thinking that best algorithm = best *proven* algorithm.

Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: “Work harder, find proofs!”

Consensus of the experts: proofs probably do not *exist* for most of these algorithms. So demanding proofs is silly.

Without proofs, how do we analyze correctness+speed?

Answer: Real algorithm analysis relies critically on heuristics and *computer experiments*. 
Notes on provability

Textbook algorithm analysis:

Proof of correctness

New algorithm

↑ ↑

↓ ↓

Proof of run time

Mislead students into thinking that best algorithm = best proven algorithm.

Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: “Work harder, find proofs!”

Consensus of the experts: proofs probably do not exist for most of these algorithms. So demanding proofs is silly.

Without proofs, how do we analyze correctness + speed?

Answer: Real algorithm analysis relies critically on heuristics and computer experiments.

What about quantum algorithms?
Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.
Notes on provability

Textbook algorithm analysis:
Proof of correctness
New algorithm
Proof of run time
Mislead students into thinking that best algorithm = best proven algorithm.

Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response:
“Work harder, find proofs!”

Consensus of the experts:
proofs probably do not exist for most of these algorithms.
So demanding proofs is silly.

Without proofs, how do we analyze correctness + speed?
Answer: Real algorithm analysis relies critically on heuristics and computer experiments.

What about quantum algorithms?
Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.
Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: “Work harder, find proofs!”

Consensus of the experts: proofs probably do not exist for most of these algorithms. So demanding proofs is silly.

Without proofs, how do we analyze correctness+speed? Answer: Real algorithm analysis relies critically on heuristics and computer experiments.

What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.
Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: “Work harder, find proofs!”

Consensus of the experts: proofs probably do not exist for most of these algorithms. So demanding proofs is silly.

Without proofs, how do we analyze correctness+speed? Answer: Real algorithm analysis relies critically on heuristics and computer experiments.

What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.
Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: “Work harder, find proofs!”

Consensus of the experts: proofs probably do not exist for most of these algorithms. So demanding proofs is silly.

Without proofs, how do we analyze correctness + speed?
Answer: Real algorithm analysis relies critically on heuristics and computer experiments.

What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer? ⇒ Huge extrapolation errors.
Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: “Work harder, find proofs!”

Consensus of the experts: proofs probably do not exist for most of these algorithms. So demanding proofs is silly.

Without proofs, how do we analyze correctness + speed? Answer: Real algorithm analysis relies critically on heuristics and computer experiments.

What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer? ⇒ Huge extrapolation errors.
Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response:
“Work harder, find proofs!”

Consensus of the experts: proofs probably do not exist for most of these algorithms. So demanding proofs is silly.

Without proofs, how do we analyze correctness+speed? Answer: Real algorithm analysis relies critically on heuristics and computer experiments.

What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer? \(\Rightarrow\) Huge extrapolation errors.


3. Fast \textit{trapdoor simulation}. Simulator (like prover) knows more than the algorithm does. Tung Chou has implemented this, found errors in two publications.
state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: “Work harder, find proofs!”

Consensus of the experts: Proofs probably do not exist for most of these algorithms. Demanding proofs is silly.

Without proofs, how do we analyze correctness + speed?

Answer: Real algorithm analysis relies critically on heuristics and computer experiments.

What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer? ⇒ Huge extrapolation errors.


3. Fast trapdoor simulation.

Simulator (like prover) knows more than the algorithm does. Tung Chou has implemented this, found errors in two publications.

Post-quantum cryptography

Grover’s algorithm finds a 128-bit AES key using $2^{64}$ quantum AES evaluations.
Reality: state-of-the-art cryptanalytic algorithms are almost never proven.

Ignorant response: "Work harder, find proofs!"

Consensus of the experts: proofs probably do not exist for most of these algorithms. Demanding proofs is silly.

How do we analyze correctness+speed?
Algorithm analysis relies critically on heuristics and computer experiments.

What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer? ⇒ Huge extrapolation errors.
3. Fast trapdoor simulation. Simulator (like prover) knows more than the algorithm does. Tung Chou has implemented this, found errors in two publications.

Post-quantum cryptography
Grover's algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.
What about quantum algorithms? Want to analyze, optimize quantum algorithms *today* to figure out safe crypto against *future* quantum attack.

1. Simulate *tiny* q. computer? ⇒ Huge extrapolation errors.


3. Fast **trapdoor simulation**. Simulator (like prover) knows more than the algorithm does. Tung Chou has implemented this, found errors in two publications.

Post-quantum cryptography
Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.
What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer? ⇒ Huge extrapolation errors.


3. Fast trapdoor simulation. Simulator (like prover) knows more than the algorithm does. Tung Chou has implemented this, found errors in two publications.

Post-quantum cryptography

Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.
What about quantum algorithms?
Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer?
⇒ Huge extrapolation errors.


3. Fast trapdoor simulation. Simulator (like prover) knows more than the algorithm does.
Tung Chou has implemented this, found errors in two publications.

Post-quantum cryptography
Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management:
Assume that this is feasible—or will be feasible in, e.g., 2025.
What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer?⇒ Huge extrapolation errors.


3. Fast trapdoor simulation. Simulator (like prover) knows more than the algorithm does. Tung Chou has implemented this, found errors in two publications.

Post-quantum cryptography
Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management: Assume that this is feasible—or will be feasible in, e.g., 2025. “AES-128 is dead.”
What about quantum algorithms? Want to analyze, optimize quantum algorithms \textit{today} to figure out safe crypto against \textit{future} quantum attack.

1. Simulate \textit{tiny} q. computer? \implies \text{Huge extrapolation errors.}


3. Fast \textbf{trapdoor simulation}. Simulator (like prover) knows more than the algorithm does. Tung Chou has implemented this, found errors in two publications.

Post-quantum cryptography
Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management:
Assume that this is feasible—or will be feasible in, e.g., 2025. “AES-128 is dead.”

Fix: Switch to AES-256.
What about quantum algorithms? Want to analyze, optimize quantum algorithms *today* to figure out safe crypto against *future* quantum attack.

1. Simulate *tiny* q. computer? ⇒ Huge extrapolation errors.
3. Fast **trapdoor simulation**. Simulator (like prover) knows more than the algorithm does. Tung Chou has implemented this, found errors in two publications.

---

**Post-quantum cryptography**

Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management: Assume that this is feasible—or will be feasible in, e.g., 2025. “AES-128 is dead.”

Fix: Switch to AES-256.

AES-256 has 14 rounds. Maybe 12 rounds are enough for $2^{128}$ post-quantum security? Maybe 10 rounds are enough?
What about quantum algorithms? Want to analyze, optimize quantum algorithms *today* to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer? ⇒ Huge extrapolation errors.
3. Fast **trapdoor simulation**. Simulator (like prover) knows more than the algorithm does. Tung Chou has implemented this, found errors in two publications.

**Post-quantum cryptography**

Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management: Assume that this is feasible—or will be feasible in, e.g., 2025. “AES-128 is dead.”

Fix: Switch to AES-256.

AES-256 has 14 rounds.

Maybe 12 rounds are enough for $2^{128}$ post-quantum security?

Maybe 10 rounds are enough?

Shor’s algorithm (similar to Simon’s algorithm) factors RSA modulus $N$ by finding period of $x \mapsto 2^x \mod N$.

Number of qubit operations $\approx$ number of bit operations to compute.
What about quantum algorithms? Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer?⇒Huge extrapolation errors.
3. Fast trapdoor simulation.
   Simulator (like prover) knows more than the algorithm does.
   Tung Chou has implemented this, found errors in two publications.

**Post-quantum cryptography**

Grover's algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management: Assume that this is feasible—or will be feasible in, e.g., 2025. “AES-128 is dead.”

Fix: Switch to AES-256.

AES-256 has 14 rounds.

Maybe 12 rounds are enough for $2^{128}$ post-quantum security?
Maybe 10 rounds are enough?

Shor’s algorithm (similar to Simon’s) factors RSA modulus by finding period of $x \mapsto 2^x \mod N$. Number of qubit operations $\approx$ number of bit operations to compute $2^x \mod N$. 
What about quantum algorithms?
Want to analyze, optimize quantum algorithms today to figure out safe crypto against future quantum attack.

1. Simulate tiny q. computer? ⇒ Huge extrapolation errors.
3. Fast trapdoor simulation. Simulator (like prover) knows more than the algorithm does.

Tung Chou has implemented this, found errors in two publications.

Post-quantum cryptography

Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management:
Assume that this is feasible—or will be feasible in, e.g., 2025. “AES-128 is dead.”

Fix: Switch to AES-256.

AES-256 has 14 rounds.
Maybe 12 rounds are enough for $2^{128}$ post-quantum security?
Maybe 10 rounds are enough?

Shor’s algorithm
(similar to Simon’s algorithm)
factors RSA modulus $N$ by finding period of $x \mapsto 2^x \mod N$.

Number of qubit operations $\approx$ number of bit operations to compute $2^x \mod N$. 
Post-quantum cryptography

Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management:
Assume that this is feasible—or will be feasible in, e.g., 2025.
“AES-128 is dead.”

Fix: Switch to AES-256.

AES-256 has 14 rounds.
Maybe 12 rounds are enough for $2^{128}$ post-quantum security?
Maybe 10 rounds are enough?

Shor’s algorithm (similar to Simon’s algorithm) factors RSA modulus $N$ by finding period of $x \mapsto 2^x \mod N$.

Number of qubit operations \( \approx \) number of bit operations to compute $2^x \mod N$. 
Post-quantum cryptography

Grover’s algorithm finds
128-bit AES key using
$2^{64}$ quantum AES evaluations.

Sensible risk management:
Assume that this is feasible—or will be feasible in, e.g., 2025.
“AES-128 is dead.”

Fix: Switch to AES-256.

AES-256 has 14 rounds.
Maybe 12 rounds are enough
for $2^{128}$ post-quantum security?
Maybe 10 rounds are enough?

Shor’s algorithm
(similar to Simon’s algorithm)
factors RSA modulus $N$ by
finding period of $x \mapsto 2^x \mod N$.

Number of qubit operations
$\approx$ number of bit operations
to compute $2^x \mod N$.
$\approx 2^{64}$ qubit operations
when $N$ is around 1 gigabyte.
Post-quantum cryptography

Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management:
Assume that this is feasible—or will be feasible in, e.g., 2025.
“AES-128 is dead.”

Fix: Switch to AES-256.

AES-256 has 14 rounds.
Maybe 12 rounds are enough for $2^{128}$ post-quantum security?
Maybe 10 rounds are enough?

Shor’s algorithm (similar to Simon’s algorithm) factors RSA modulus $N$ by finding period of $x \mapsto 2^x \mod N$.

Number of qubit operations $\approx$ number of bit operations to compute $2^x \mod N$.
$\approx 2^{64}$ qubit operations when $N$ is around 1 gigabyte.

Shor also finds $\log_g h$ by finding period of $(x, y) \mapsto g^x h^y$. 
Post-quantum cryptography

Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management:
Assume that this is feasible—or will be feasible in, e.g., 2025.
“AES-128 is dead.”

Fix: Switch to AES-256.

AES-256 has 14 rounds.
Maybe 12 rounds are enough for $2^{128}$ post-quantum security?
Maybe 10 rounds are enough?

Shor’s algorithm
(similar to Simon’s algorithm)
factors RSA modulus $N$ by finding period of $x \mapsto 2^x \bmod N$.

Number of qubit operations
$\approx$ number of bit operations
to compute $2^x \bmod N$.

$\approx 2^{64}$ qubit operations
when $N$ is around 1 gigabyte.

Shor also finds $\log g \, h$ by
finding period of $(x, y) \mapsto g^x h^y$.

“RSA is dead. ECC is dead.”
Post-quantum cryptography

Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management:
Assume that this is feasible—or will be feasible in, e.g., 2025.
“AES-128 is dead.”

Fix: Switch to AES-256.

AES-256 has 14 rounds.
Maybe 12 rounds are enough for $2^{128}$ post-quantum security?
Maybe 10 rounds are enough?

Shor’s algorithm (similar to Simon’s algorithm) factors RSA modulus $N$ by finding period of $x \mapsto 2^x \mod N$.

Number of qubit operations $\approx$ number of bit operations to compute $2^x \mod N$.

$\approx 2^{64}$ qubit operations when $N$ is around 1 gigabyte.

Shor also finds $\log_g h$ by finding period of $(x, y) \mapsto g^x h^y$.

“RSA is dead. ECC is dead.”
But some systems seem safe.
Post-quantum cryptography

Grover's algorithm finds
128-bit AES key using
$2^{64}$ quantum AES evaluations.

Sensible risk management:
Assume that this is feasible—
or will be feasible in, e.g., 2025.
AES-128 is dead.

Fix: Switch to AES-256.

AES-256 has 14 rounds.

Maybe 12 rounds are enough
for $2^{128}$ post-quantum security?

Maybe 10 rounds are enough?

Shor's algorithm
(similar to Simon's algorithm)
factors RSA modulus $N$ by
finding period of $x \mapsto 2^x \mod N$.

Number of qubit operations
$\approx$ number of bit operations
to compute $2^x \mod N$.

$\approx 2^{64}$ qubit operations
when $N$ is around 1 gigabyte.

Shor also finds $\log g \cdot h$ by
finding period of $(x,y) \mapsto g^x h^y$.

"RSA is dead. ECC is dead."

But some systems seem safe.

Hash-based signatures.
Example: 1979 Merkle hash-tree public-key signature system.

Code-based cryptography.
Example: 1978 McEliece hidden-Goppa-code public-key encryption system.

Lattice-based cryptography.
Example: 1998 "NTRU".

Multivariate-quadratic-equations cryptography.
Example: 1996 Patarin "HFE$^\neg$" public-key signature system.
Post-quantum cryptography

Grover’s algorithm finds 128-bit AES key using $2^{64}$ quantum AES evaluations.

Sensible risk management: Assume that this is feasible—or will be feasible in, e.g., 2025.

“AES-128 is dead.” Fix: Switch to AES-256.

AES-256 has 14 rounds.

Maybe 12 rounds are enough for $2^{128}$ post-quantum security?

Maybe 10 rounds are enough?

Shor’s algorithm (similar to Simon’s algorithm) factors RSA modulus $N$ by finding period of $x \rightarrow 2^x \mod N$.

Number of qubit operations $\approx$ number of bit operations to compute $2^x \mod N$.

$\approx 2^{64}$ qubit operations when $N$ is around 1 gigabyte.

Shor also finds $\log g h$ by finding period of $(x, y) \rightarrow g^x h^y$.

“RSA is dead. ECC is dead.”

But some systems seem safe.

Hash-based signatures.
Example: 1979 Merkle hash-tree public-key signature system.

Code-based cryptography.
Example: 1978 McEliece hidden-Goppa-code public-key encryption system.

Lattice-based cryptography.
Example: 1998 “NTRU” public-key encryption system.

Multivariate-quadratic-equations cryptography.
Example: 1996 Patarin “HFE-v−” public-key signature system.
Shor’s algorithm
(similar to Simon’s algorithm)
factors RSA modulus $N$ by
finding period of $x \mapsto 2^x \mod N$.

Number of qubit operations
$\approx$ number of bit operations
to compute $2^x \mod N$.

$\approx 2^{64}$ qubit operations
when $N$ is around 1 gigabyte.

Shor also finds $\log g h$
by
finding period of $(x, y) \mapsto g^x h^y$.

“RSA is dead. ECC is dead.”
But some systems seem safe.

Hash-based signatures.
Example: 1979 Merkle hash-tree
classical public-key signature system.

Code-based cryptography.
Example: 1978 McEliece
hidden-Goppa-code
classical public-key encryption system.

Lattice-based cryptography.
Example: 1998 “NTRU”.

Multivariate-quadratic-equations cryptography.
Example:
1996 Patarin “HFE$^v$−”
classical public-key signature system.
Shor’s algorithm (similar to Simon’s algorithm) factors RSA modulus \( N \) by finding period of \( x \mapsto 2^x \mod N \).

Number of qubit operations \( \approx \) number of bit operations to compute \( 2^x \mod N \).

\( \approx 2^{64} \) qubit operations when \( N \) is around 1 gigabyte.

Shor also finds \( \log g h \) by finding period of \( (x, y) \mapsto g^x h^y \).

“RSA is dead. ECC is dead.” But some systems seem safe.

---

**Hash-based signatures.**
Example: 1979 Merkle hash-tree public-key signature system.

**Code-based cryptography.**
Example: 1978 McEliece hidden-Goppa-code public-key encryption system.

**Lattice-based cryptography.**
Example: 1998 “NTRU”.

**Multivariate-quadratic-equations cryptography.**
Example: 1996 Patarin “HFEv−” public-key signature system.
Shor's algorithm (similar to Simon's algorithm) factors RSA modulus $N$ by finding period of $x \mapsto 2^x \mod N$. Number of qubit operations is around 1 gigabyte. To finds $\log g h$ by finding period of $(x, y) \mapsto g^x h^y$. "RSA is dead. ECC is dead." But some systems seem safe.

Hash-based signatures. Example: 1979 Merkle hash-tree public-key signature system.


Lattice-based cryptography. Example: 1998 "NTRU".

Shor's algorithm similar to Simon's algorithm factors RSA modulus $N$ by finding period of $x \mapsto 2^x \mod N$. Number of qubit operations $\approx$ number of bit operations to compute $2^x \mod N$. $\approx 2^{64}$ qubit operations when $N$ is around 1 gigabyte. Shor also finds $\log g \cdot h$ by finding period of $(x, y) \mapsto g^x h^y$. “RSA is dead. ECC is dead.” But some systems seem safe.

---

**Hash-based signatures.**
Example: 1979 Merkle hash-tree public-key signature system.

**Code-based cryptography.**
Example: 1978 McEliece hidden-Goppa-code public-key encryption system.

**Lattice-based cryptography.**
Example: 1998 “NTRU”.

**Multivariate-quadratic-equations cryptography.**
Example: 1996 Patarin “HFE $v^-$” public-key signature system.
Shor's algorithm
(similar to Simon's algorithm)
factors RSA modulus $N$ by
finding period of $x \mapsto 2^x \mod N$.

Number of qubit operations
$\approx$ number of bit operations
to compute $2^x \mod N$.
$\approx 2^{64}$ qubit operations
when $N$ is around 1 gigabyte.

Shor also finds $\log g \cdot h$ by
finding period of $(x; y) \mapsto g^x h^y$.

"RSA is dead. ECC is dead."
But some systems seem safe.

Hash-based signatures.
Example: 1979 Merkle hash-tree public-key signature system.

Code-based cryptography.
Example: 1978 McEliece hidden-Goppa-code public-key encryption system.

Lattice-based cryptography.
Example: 1998 “NTRU”.

Multivariate-quadratic-equations cryptography.
Example: 1996 Patarin “HFE$^v$−” public-key signature system.
Hash-based signatures.
Example: 1979 Merkle hash-tree public-key signature system.

Code-based cryptography.
Example: 1978 McEliece hidden-Goppa-code public-key encryption system.

Lattice-based cryptography.
Example: 1998 “NTRU”.

Multivariate-quadratic-equations cryptography.
Example: 1996 Patarin “HFE$^-$” public-key signature system.
Hash-based signatures.
Example: 1979 Merkle hash-tree public-key signature system.

Code-based cryptography.
Example: 1978 McEliece Goppa-code public-key encryption system.

Lattice-based cryptography.
Example: 1998 “NTRU”.

Multivariate-quadratic-equations cryptography.
Example: 1996 Patarin “HFE” public-key signature system.

The 1978 McEliece cryptosystem (with 1986 Niederreiter speedup)
Receiver’s public key: “random”
500 × 1024 matrix $K$ over $F_2$.
Specifies linear $F_{1024^2} \rightarrow F_{500^2}$.
Hash-based signatures.
Example: 1979 Merkle hash-tree public-key signature system.

Code-based cryptography.
Example: 1978 McEliece hidden-Goppa-code public-key encryption system.

Lattice-based cryptography.
Example: 1998 “NTRU”.

Multivariate-quadratic-equations cryptography.
Example: 1996 Patarin “HFE$^{−}$” public-key signature system.

The 1978 McEliece cryptosystem (with 1986 Niederreiter speedup)
Receiver’s public key: “random” 500 × 1024 matrix $K$ over $\mathbb{F}_2$.
Specifies linear $\mathbb{F}_{2^{1024}} \rightarrow \mathbb{F}_{2^{500}}$. 
Hash-based signatures.
Example: 1979 Merkle hash-tree public-key signature system.

Code-based cryptography.
Example: 1978 McEliece hidden-Goppa-code public-key encryption system.

Lattice-based cryptography.
Example: 1998 "NTRU".

Multivariate-quadratic-equations cryptography.
Example: 1996 Patarin "HFE v − " public-key signature system.

The 1978 McEliece cryptosystem (with 1986 Niederreiter speedup)
Receiver’s public key: “random” 500 × 1024 matrix $K$ over $\mathbb{F}_2$.
Specifies linear $\mathbb{F}_2^{1024} \rightarrow \mathbb{F}_2^{500}$. 
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)
Receiver’s public key: “random”
500 × 1024 matrix $K$ over $\mathbb{F}_2$.
Specifies linear $\mathbb{F}_2^{1024} \to \mathbb{F}_2^{500}$. 
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)

Receiver’s public key: “random”
500 × 1024 matrix $K$ over $F_2$.
Specifies linear $F_2^{1024} \rightarrow F_2^{500}$.

Messages suitable for encryption:
1024-bit strings of weight 50.
$\{ e \in F_2^{1024} : \# \{ i : e_i = 1 \} = 50 \}$. 
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)

Receiver’s public key: “random”
500 × 1024 matrix $K$ over $F_2$.
Specifies linear $F_2^{1024} \rightarrow F_2^{500}$.

Messages suitable for encryption:
1024-bit strings of weight 50.
$\{ e \in F_2^{1024} : \# \{ i : e_i = 1 \} = 50 \}$.

Encryption of $e$ is $Ke \in F_2^{500}$.
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)

Receiver’s public key: “random”
500 × 1024 matrix $K$ over $\mathbb{F}_2$.
Specifies linear $\mathbb{F}_2^{1024} \rightarrow \mathbb{F}_2^{500}$.

Messages suitable for encryption:
1024-bit strings of weight 50.
$\{e \in \mathbb{F}_2^{1024} : \#\{i : e_i = 1\} = 50\}$.

Encryption of $e$ is $Ke \in \mathbb{F}_2^{500}$.

“Padding”: Choose random $e$;
send $Ke$; use SHA-256($e, Ke$) as AES-256-GCM key to encrypt
actual message of any length.
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)

Receiver’s public key: “random”
500 × 1024 matrix $K$ over $\mathbb{F}_2$.
Specifies linear $\mathbb{F}_2^{1024} \rightarrow \mathbb{F}_2^{500}$.

Messages suitable for encryption:
1024-bit strings of weight 50.
\[ \{ e \in \mathbb{F}_2^{1024} : \#\{ i : e_i = 1 \} = 50 \}. \]

Encryption of $e$ is $Ke \in \mathbb{F}_2^{500}$.

“Padding”: Choose random $e$; send $Ke$; use SHA-256($e, Ke$) as AES-256-GCM key to encrypt actual message of any length.

Attacker, by linear algebra, easily works backwards from $Ke$ to some $v \in \mathbb{F}_2^{1024}$ such that $Kv = Ke$. 
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)

Receiver’s public key: “random”
$500 \times 1024$ matrix $K$ over $\mathbb{F}_2$.
Specifies linear $\mathbb{F}_2^{1024} \rightarrow \mathbb{F}_2^{500}$.

Messages suitable for encryption:
1024-bit strings of weight 50.
$\{ e \in \mathbb{F}_2^{1024} : \# \{i : e_i = 1\} = 50 \}$.

Encryption of $e$ is $K e \in \mathbb{F}_2^{500}$.

“Padding”: Choose random $e$;
send $Ke$; use SHA-256($e, Ke$) as
AES-256-GCM key to encrypt
actual message of any length.

Attacker, by linear algebra,
easily works backwards from $Ke$ to some $v \in \mathbb{F}_2^{1024}$ such that $K v = Ke$. 
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)
Receiver’s public key: “random”
500 × 1024 matrix $K$ over $\mathbb{F}_2$.
Specifies linear $\mathbb{F}_2^{1024} \rightarrow \mathbb{F}_2^{500}$.

Messages suitable for encryption:
1024-bit strings of weight 50.
$\{e \in \mathbb{F}_2^{1024} : \#\{i : e_i = 1\} = 50\}$.

Encryption of $e$ is $K e \in \mathbb{F}_2^{500}$.

“Padding”: Choose random $e$;
send $K e$; use SHA-256($e, K e$) as
AES-256-GCM key to encrypt
actual message of any length.

Attacker, by linear algebra,
easily works backwards
from $K e$ to some $v \in \mathbb{F}_2^{1024}$
such that $K v = K e$. 
The 1978 McEliece cryptosystem (with 1986 Niederreiter speedup) Receiver’s public key: “random” 500 × 1024 matrix $K$ over $F_2$. Specifies linear $F_2^{1024} \rightarrow F_2^{500}$. Messages suitable for encryption: 1024-bit strings of weight 50. \{e \in F_2^{1024} : \# \{i : e_i = 1\} = 50\}. Encryption of $e$ is $Ke \in F_2^{500}$. “Padding”: Choose random $e$; send $Ke$; use SHA-256($e, Ke$) as AES-256-GCM key to encrypt actual message of any length.

Attacker, by linear algebra, easily works backwards from $Ke$ to some $v \in F_2^{1024}$ such that $Kv = Ke$. 
The 1978 McEliece cryptosystem (with 1986 Niederreiter speedup)

Receiver’s public key: “random”
500 × 1024 matrix $K$ over $\mathbb{F}_2$.
Specifies linear $\mathbb{F}_2^{1024} \to \mathbb{F}_2^{500}$.

Messages suitable for encryption:
1024-bit strings of weight 50.
$\{ e \in \mathbb{F}_2^{1024} : \# \{ i : e_i = 1 \} = 50 \}$.

Encryption of $e$ is $Ke \in \mathbb{F}_2^{500}$.

“Padding”: Choose random $e$; send $Ke$; use SHA-256($e, Ke$) as AES-256-GCM key to encrypt actual message of any length.

Attacker, by linear algebra, easily works backwards from $Ke$ to some $v \in \mathbb{F}_2^{1024}$ such that $Kv = Ke$.

i.e. Attacker finds some element $v \in e + \text{Ker } K$.

Note that $\# \text{Ker } K \geq 2^{524}$.

Attacker wants to decode $v$: to find element of $\text{Ker } K$ at distance only 50 from $v$.
Presumably unique, revealing $e$. 
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)
Receiver’s public key: “random”
500 × 1024 matrix $K$ over $\mathbb{F}_2$.
Specifies linear $\mathbb{F}_2^{1024} \rightarrow \mathbb{F}_2^{500}$.
Messages suitable for encryption:
1024-bit strings of weight 50.
$\{ e \in \mathbb{F}_2^{1024} : \# \{ i : e_i = 1 \} = 50 \}$.
Encryption of $e$ is $Ke \in \mathbb{F}_2^{500}$.
“Padding”: Choose random $e$; send $Ke$; use SHA-256($e$, $Ke$) as AES-256-GCM key to encrypt actual message of any length.

Attacker, by linear algebra, easily works backwards from $Ke$ to some $v \in \mathbb{F}_2^{1024}$ such that $Kv = Ke$.
i.e. Attacker finds some element $v \in e + \text{Ker } K$.
Note that $\# \text{Ker } K \geq 2^{524}$.
Attacker wants to decode $v$: to find element of Ker $K$ at distance only 50 from $v$.
Presumably unique, revealing $e$.
But decoding isn’t easy!
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)
Receiver’s public key: “random”
500 × 1024 matrix $K$ over $\mathbb{F}_2$.
Messages suitable for encryption:
strings of weight 50.
\{ $e$ \in $\mathbb{F}_{1024}^2$ : \# \{ $i$ : $e_i = 1$ \} = 50 \}.
Attacker, by linear algebra,
easily works backwards
from $Ke$ to some $v \in \mathbb{F}_2^{1024}$
such that $Kv = Ke$.
i.e. Attacker finds some
element $v \in e + \text{Ker} K$.
Note that \# Ker $K \geq 2^{524}$.
Attacker wants to decode $v$:
to find element of Ker $K$
at distance only 50 from $v$.
Presumably unique, revealing $e$.
But decoding isn’t easy!

Information-set decoding
Choose random size-500 subset $S \subseteq \{1, 2, 3, \ldots, 1024\}$.
For typical $K$ : Good chance
that $\mathbb{F}_2^S \rightarrow \mathbb{F}_{1024}^2 \xrightarrow{K} \mathbb{F}_2^{500}$
is invertible.

"Padding": Choose random $e$;
send $Ke$; use SHA-256($e, Ke$) as
AES-256-GCM key to encrypt
actual message of any length.
The 1978 McEliece cryptosystem
(with 1986 Niederreiter speedup)
Receiver's public key: “random”
500 × 1024 matrix K
over \( \mathbb{F}_2 \).
Specifies linear \( \mathbb{F}_{1024} \rightarrow \mathbb{F}_{500}^2 \).
Messages suitable for encryption:
1024-bit strings of weight 50.
\[ \{ e \in \mathbb{F}_{1024}^2 : \# \{ i : e_i = 1 \} = 50 \} \]
Encryption of \( e \) is \( Ke \in \mathbb{F}_{500}^2 \).
“Padding”: Choose random \( e \);
send \( Ke \); use SHA-256(\( e;Ke \)) as
AES-256-GCM key to encrypt
actual message of any length.

Attacker, by linear algebra,
easily works backwards
from \( Ke \) to some \( v \in \mathbb{F}_2^{1024} \)
such that \( Kv = Ke \).
i.e. Attacker finds some
element \( v \in e + \text{Ker } K \).
Note that \( \# \text{Ker } K \geq 2^{524} \).
Attacker wants to decode \( v \):
to find element of \( \text{Ker } K \)
at distance only 50 from \( v \).
Presumably unique, revealing \( e \).
But decoding isn’t easy!

Information-set decoding
Choose random size-500 subset
\( S \subseteq \{ 1, 2, 3, \ldots, 1024 \} \).
For typical \( K \): Good chance
that \( \mathbb{F}_2^S \rightarrow \mathbb{F}_2^{1024} \rightarrow \mathbb{F}_2^{500} \)
is invertible.
Attacker, by linear algebra, easily works backwards from $Ke$ to some $v \in \mathbb{F}_2^{1024}$ such that $Kv = Ke$.

i.e. Attacker finds some element $v \in e + \text{Ker } K$.

Note that $\# \text{Ker } K \geq 2^{524}$.

Attacker wants to decode $v$: to find element of $\text{Ker } K$ at distance only 50 from $v$.

Presumably unique, revealing $e$.

But decoding isn’t easy!

Information-set decoding
Choose random size-500 subset $S \subseteq \{1, 2, 3, \ldots, 1024\}$.

For typical $K$: Good chance that $\mathbb{F}_2^S \rightarrow \mathbb{F}_2^{1024} \rightarrow \mathbb{F}_2^{500}$ is invertible.
Attacker, by linear algebra, easily works backwards from $K e$ to some $v \in \mathbb{F}_2^{1024}$ such that $K v = K e$.

i.e. Attacker finds some element $v \in e + \text{Ker } K$.

Note that $\# \text{Ker } K \geq 2^{524}$.

Attacker wants to decode $v$: to find element of $\text{Ker } K$ at distance only 50 from $v$.

Presumably unique, revealing $e$.

But decoding isn’t easy!

Information-set decoding

Choose random size-500 subset $S \subseteq \{1, 2, 3, \ldots, 1024\}$.

For typical $K$: Good chance that $\mathbb{F}_2^S \rightarrow \mathbb{F}_2^{1024} \xrightarrow{K} \mathbb{F}_2^{500}$ is invertible.
Attacker, by linear algebra, easily works backwards from $Ke$ to some $v \in F_2^{1024}$ such that $Kv = Ke$.

i.e. Attacker finds some element $v \in e + \text{Ker } K$.

Note that $\# \text{Ker } K \geq 2^{524}$.

Attacker wants to decode $v$:

to find element of Ker $K$

at distance only 50 from $v$.

Presumably unique, revealing $e$.

But decoding isn’t easy!

---

**Information-set decoding**

Choose random size-500 subset $S \subseteq \{1, 2, 3, \ldots, 1024\}$.

For typical $K$: Good chance that $F_2^S \hookrightarrow F_2^{1024} \xrightarrow{K} F_2^{500}$ is invertible.

Hope $e \in F_2^S$; chance $\approx 2^{-53}$.

Apply inverse map to $Ke$,

revealing $e$ if $e \in F_2^S$. 

Attacker, by linear algebra, easily works backwards from $Ke$ to some $v \in F_{2}^{1024}$ such that $Kv = Ke$.

i.e. Attacker finds some element $v \in e + \text{Ker } K$.
Note that $\# \text{Ker } K \geq 2^{524}$.

Attacker wants to decode $v$: to find element of $\text{Ker } K$ at distance only 50 from $v$.
Presumably unique, revealing $e$.
But decoding isn’t easy!

Information-set decoding
Choose random size-500 subset $S \subseteq \{1, 2, 3, \ldots, 1024\}$.

For typical $K$: Good chance that $F_{2}^{S} \rightarrow F_{2}^{1024} \xrightarrow{K} F_{2}^{500}$ is invertible.

Hope $e \in F_{2}^{S}$; chance $\approx 2^{-53}$.
Apply inverse map to $Ke$, revealing $e$ if $e \in F_{2}^{S}$.
If $e \notin F_{2}^{S}$, try again.
$\approx 2^{80}$ bit operations in total.
Attacker, by linear algebra, easily works backwards from $Ke$ to some $v \in F_2^{1024}$ such that $Kv = Ke$.
i.e. Attacker finds some element $v \in e + \ker K$.
Note that $\#\ker K \geq 2^{524}$.
Attacker wants to decode $v$:
to find element of $\ker K$
at distance only 50 from $v$.
Presumably unique, revealing $e$.
But decoding isn’t easy!

Information-set decoding
Choose random size-500 subset $S \subseteq \{1, 2, 3, \ldots, 1024\}$.
For typical $K$: Good chance that $F_2^S \rightarrow F_2^{1024} \xrightarrow{K} F_2^{500}$
is invertible.
Hope $e \in F_2^S$; chance $\approx 2^{-53}$.
Apply inverse map to $Ke$,
revealing $e$ if $e \in F_2^S$.
If $e \notin F_2^S$, try again.
$\approx 2^{80}$ bit operations in total.
Bad estimate by McEliece: $\approx 2^{64}$. 
Attacker, by linear algebra, works backwards from some \( v \in \mathbb{F}_2^{1024} \) such that \( K v = K e \).

Attacker finds some \( v \in e + \text{Ker } K \).

Note that \( \# \text{Ker } K \geq 2^{524} \).

Attacker wants to decode \( v \): find element of Ker \( K \) at distance only 50 from \( v \). Presumably unique, revealing \( e \).

But decoding isn’t easy!

**Information-set decoding**

Choose random size-500 subset \( S \subseteq \{1, 2, 3, \ldots, 1024\} \).

For typical \( K \): Good chance that \( \mathbb{F}_2^S \hookrightarrow \mathbb{F}_2^{1024} \xrightarrow{K} \mathbb{F}_2^{500} \) is invertible.

Hope \( e \in \mathbb{F}_2^S \); chance \( \approx 2^{-53} \).

Apply inverse map to \( K e \), revealing \( e \) if \( e \in \mathbb{F}_2^S \).

If \( e \not\in \mathbb{F}_2^S \), try again.

\( \approx 2^{80} \) bit operations in total.

Bad estimate by McEliece: \( \approx 2^{64} \).

Analyzing and optimizing attacks:

1962 Prange.
1981 Omura.
1988 Lee–Brickell.
1988 Leon.
1989 Krouk.
1989 Stern.
1989 Dumer.
1990 Coffey–Goodman.
1990 van Tilburg.
1991 Dumer.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
Attacker, by linear algebra, works backwards from $K e$ to some $v \in F_{1024}^2$ such that $K v = Ke$.

i.e. Attacker finds some $v \in e + \ker K$.

Note that $\# \ker K \geq 2^{524}$.

Attacker wants to decode $v$ to find element of $\ker K$ at distance only 50 from $v$.

Presumably unique, revealing $e$.

But decoding isn't easy!

### Information-set decoding

Choose random size-500 subset $S \subseteq \{1, 2, 3, \ldots, 1024\}$.

For typical $K$: Good chance that $F_2^S \hookrightarrow F_2^{1024} \xrightarrow{K} F_2^{500}$ is invertible.

Hope $e \in F_2^S$; chance $\approx 2^{-53}$.

Apply inverse map to $K e$, revealing $e$ if $e \in F_2^S$.

If $e \notin F_2^S$, try again.

$\approx 2^{80}$ bit operations in total.

Bad estimate by McEliece: $\approx 2^{64}$.

Analyzing and optimizing attacks:

- 1962 Prange.
- 1981 Omura.
- 1988 Leon.
- 1989 Krouk.
- 1989 Stern.
- 1989 Dumer.
- 1990 Coffey–Goodman.
- 1990 van Tilburg.
- 1993 Chabanne–Courteau.
- 1993 Chabaud.
- 1994 van Tilburg.
- 1994 Canteaut–Chabanne.
- 1998 Canteaut–Chabaud.
Information-set decoding

Choose random size-500 subset $S \subseteq \{1, 2, 3, \ldots, 1024\}$.

For typical $K$: Good chance that $\mathbb{F}_2^S \rightarrow \mathbb{F}_2^{1024} \xrightarrow{K} \mathbb{F}_2^{500}$ is invertible.

Hope $e \in \mathbb{F}_2^S$; chance $\approx 2^{-53}$.

Apply inverse map to $Ke$, revealing $e$ if $e \in \mathbb{F}_2^S$.

If $e \notin \mathbb{F}_2^S$, try again.

$\approx 2^{80}$ bit operations in total.

Bad estimate by McEliece: $\approx 2^{64}$.

Analyzing and optimizing attacks:

- 1962 Prange.
- 1981 Omura.
- 1988 Leon.
- 1989 Krouk.
- 1989 Stern.
- 1989 Dumer.
- 1990 Coffey–Goodman.
- 1990 van Tilburg.
- 1991 Dumer.
- 1993 Chabanne–Courteau.
- 1993 Chabaud.
- 1994 van Tilburg.
- 1994 Canteaut–Chabanne.
- 1998 Canteaut–Chabaud.
Information-set decoding

Choose random size-500 subset $S \subseteq \{1, 2, 3, \ldots, 1024\}$.

For typical $K$: Good chance that $F_2^S \leftrightarrow F_2^{1024} \xrightarrow{K} F_2^{500}$ is invertible.

Hope $e \in F_2^S$; chance $\approx 2^{-53}$.

Apply inverse map to $Ke$, revealing $e$ if $e \in F_2^S$.

If $e \notin F_2^S$, try again.

$\approx 2^{80}$ bit operations in total.

Bad estimate by McEliece: $\approx 2^{64}$.

Analyzing and optimizing attacks:

1989 Dumer.
1990 Coffey–Goodman.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.
Information-set decoding

Choose random size-500 subset \( S \subseteq \{1, 2, 3, \ldots, 1024\} \).

Typical \( K \): Good chance that \( F_S^2 \rightarrow F_{1024}^2 \rightarrow K \rightarrow F_{500}^2 \) is invertible.

If \( e \in F_S^2 \); chance \( \approx 2^{-53} \).

Apply inverse map to \( Ke \), revealing \( e \) if \( e \in F_S^2 \).

If \( e \notin F_S^2 \), try again.

\( \approx 2^{80} \) bit operations in total.

Bad estimate by McEliece: \( \approx 2^{64} \).

Analyzing and optimizing attacks:


2008 Bernstein–Lange–Peters:

more speedups; \( \approx 2^{60} \) cycles; attack actually carried out.


2009 Finiasz–Sendrier.

2010 Bernstein–Lange–Peters.

2011 May–Meurer–Thomae.


2015 May–Ozerov.
Information-set decoding
Choose random size-500 subset $S \subseteq \{1; 2; 3; \cdots; 1024\}$.

For typical $K$:

Good chance

that $F^{500} \rightarrow F^{1024}$, $K \rightarrow F^{500}$

is invertible.

Hope $e \in F^{500}$; chance $\approx 2^{-53}$.

Apply inverse map to $Ke$;

revealing $e$ if $e \in F^{500}$.

If $e = \in F^{500}$, try again.

$\approx 2^{80}$ bit operations in total.

Bad estimate by McEliece: $\approx 2^{64}$.

Analyzing and optimizing attacks:

1989 Dumer.
1990 Coffey–Goodman.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.

2008 Bernstein–Lange–Peters:

more speedups; $\approx 2^{60}$ cycles; attack actually carried out.

2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2015 May–Ozerov.
Information-set decoding

Choose random size-500 subset \( S \subseteq \{1; 2; 3; \ldots; 1024\} \).

For typical \( K \): Good chance that \( F_{S, 2} \rightarrow F_{1024, 2} \rightarrow F_{500, 2} \) is invertible.

Hope \( e \in F_{S, 2} \); chance \( \approx 2^{-53} \).

Apply inverse map to \( Ke \), revealing \( e \) if \( e \in F_{S, 2} \).

If \( e = \in F_{S, 2} \), try again.

\( \approx 2^{80} \) bit operations in total.

Bad estimate by McEliece: \( \approx 2^{64} \).

Analyzing and optimizing attacks:
1989 Dumer.
1990 Coffey–Goodman.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.
2008 Bernstein–Lange–Peters:
more speedups; \( \approx 2^{60} \) cycles;
attack actually carried out.
2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2015 May–Ozerov.
Analyzing and optimizing attacks:
1989 Dumer.
1990 Coffey–Goodman.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.

2008 Bernstein–Lange–Peters:
more speedups; \( \approx 2^{60} \) cycles;
attack actually carried out.


2009 Finiasz–Sendrier.

2010 Bernstein–Lange–Peters.

2011 May–Meurer–Thomae.


2015 May–Ozerov.
Analyzing and optimizing attacks:
1989 Dumer.
1990 Coffey–Goodman.
1990 van Tilburg.
1991 Dumer.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.
2008 Bernstein–Lange–Peters: more speedups; \( \approx 2^{60} \) cycles; attack \textbf{actually carried out}.
2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2015 May–Ozerov.

Modern McEliece
Easily rescue system by using a larger public key: "random" \((n/2) \times n\) matrix \(K\) over \(\mathbb{F}_2\). e.g., 1800 \times 3600.
Analyzing and optimizing attacks:

1989 Dumer.
1990 Coffey–Goodman.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.

2008 Bernstein–Lange–Peters:
more speedups; $\approx 2^{60}$ cycles;
attack actually carried out.

2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2015 May–Ozerov.

Modern McEliece

Easily rescue system by using
a larger public key: "random"
$(n/2) \times n$ matrix $K$ over $F_2$.

E.g., $1800 \times 3600$. 
Analyzing and optimizing attacks:

1989 Dumer.
1990 Coffey–Goodman.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.

2008 Bernstein–Lange–Peters: more speedups; \( \approx 2^{60} \) cycles; attack **actually carried out**.

2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2015 May–Ozerov.

Modern McEliece

Easily rescue system by using a larger public key: “random” \((n/2) \times n\) matrix \(K\) over \(F_2\), e.g., \(1800 \times 3600\).
2008 Bernstein–Lange–Peters: more speedups; $\approx 2^{60}$ cycles; attack actually carried out.


2009 Finiasz–Sendrier.

2010 Bernstein–Lange–Peters.

2011 May–Meurer–Thomae.


2015 May–Ozerov.

Modern McEliece

Easily rescue system by using a larger public key: “random” $(n/2) \times n$ matrix $K$ over $\mathbb{F}_2$. e.g., $1800 \times 3600$. 

Modern McEliece
2008 Bernstein–Lange–Peters: more speedups; $\approx 2^{60}$ cycles; attack **actually carried out**.

2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2015 May–Ozerov.

---

**Modern McEliece**

Easily rescue system by using a larger public key: “random” $(n/2) \times n$ matrix $K$ over $F_2$. e.g., $1800 \times 3600$.

Larger weight $w \approx n/(2 \lg n)$. e.g. $e \in F_2^{3600}$ of weight 150.
2008 Bernstein–Lange–Peters: more speedups; \( \approx 2^{60} \) cycles; attack **actually carried out**.


2009 Finiasz–Sendrier.

2010 Bernstein–Lange–Peters.

2011 May–Meurer–Thomae.


2015 May–Ozerov.

---

**Modern McEliece**

Easily rescue system by using a larger public key: “random” \((n/2) \times n\) matrix \(K\) over \(\mathbb{F}_2\).

E.g., \(1800 \times 3600\).

Larger weight \(w \approx n/(2 \log n)\).

E.g. \(e \in \mathbb{F}_2^{3600}\) of weight 150.

1962 attack cost: \(2^{(1+o(1))w}\).
Modern McEliece

Easily rescue system by using a larger public key: “random” \((n/2) \times n\) matrix \(K\) over \(\mathbb{F}_2\). e.g., \(1800 \times 3600\).

Larger weight \(w \approx n/(2 \log n)\).
e.g. \(e \in \mathbb{F}_2^{3600}\) of weight 150.

1962 attack cost: \(2^{(1+o(1))w}\).

After extensive research,
2015 attack cost: \(2^{(1+o(1))w}\).
2008 Bernstein–Lange–Peters: more speedups; $\approx 2^{60}$ cycles; attack **actually carried out**.


2009 Finiasz–Sendrier.

2010 Bernstein–Lange–Peters.

2011 May–Meurer–Thomae.


2015 May–Ozerov.

---

**Modern McEliece**

Easily rescue system by using a larger public key: “random” $(n/2) \times n$ matrix $K$ over $\mathbb{F}_2$.

*e.g.*, $1800 \times 3600$.

Larger weight $w \approx n/(2 \lg n)$.

*e.g.* $e \in \mathbb{F}_2^{3600}$ of weight $150$.

1962 attack cost: $2^{(1+o(1))w}$.

After extensive research, 2015 attack cost: $2^{(1+o(1))w}$.

Post-quantum: $2^{(0.5+o(1))w}$.

*e.g.* $\approx 2^{26}$ Grover iterations to search $2^{53}$ choices of $S$. 