Data ("state") stored in \( n \) bits: an element of \( \{0, 1\}^n \), often viewed as representing an element of \( \{0, 1, \ldots, 2^n - 1\} \).
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!
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
\textbf{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.
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%.
Data (“state”) stored in $n$ bits:
a non-zero element of $\{0, 1\}^n$,
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 $(a_0, a_1, a_2, \ldots, a_{2^n-1})$ 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 $80\%$.
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!

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 operation:

\( (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6) \) 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:
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 $|a_q|^2 = \Pr |a_r|^2$.

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

“$|0⟩$” in standard notation.
Measurement produces 0.

$(0, 0, 0, 0, 0, 0, 1, 0)$ is

“$|6⟩$” in standard notation.
Measurement produces 6.

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

$(0, 0, 4, 0, 0, 0, 8, 0) = 4|2⟩ + 8|6⟩$: 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

$(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)$ 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, 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 ⊕ 1,
with probability |a_q|^2 / \sum_r |a_r|^2.

(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).
Some examples of 3-qubit states: 

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

"|0⟩" in standard notation.

Measurement produces 0.

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

"|6⟩" in standard notation.

Measurement produces 6.

\((0; 0; 0; 0; 0; 0; -7i; 0)\) = 

\(-7i|6⟩\):

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

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

is measured as \((q_0, q_1, q_2)\),

representing \(q = q_0 + 2q_1 + 4q_2\),

with probability \(|a_q|^2 / \sum_r |a_r|^2\).

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

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

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

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

\((q_0, q_1, q_2) \mapsto (q_0, q_1, q_2 ⊕ 1)\).
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%.

Fast quantum operations, part 1

\(\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)
\end{align*}\)

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

\(\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)
\end{align*}\)

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

\(\begin{align*}
(a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6)
\end{align*}\)

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

\(\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_4, a_5, a_6, a_7, a_0, a_1, a_2, a_3)
\end{align*}\)

is “complementing qubit 2”:

\(\begin{align*}
(q_0, q_1, q_2) \mapsto (q_0, q_1, q_2 \oplus 1).
\end{align*}\)
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

= swapping qubits 0 and 2

  ○ complementing qubit 0

  ○ 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 measured as (q_0 \oplus 1; q_1; q_2),
representing q \oplus 1,
with probability |a_q|^2 = P_r |a_r|^2.

Complementing index bit 0,
complementing qubit 0”.

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

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

Similarly: swapping qubits i, j.

Example with more qubits:

(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).

Example with more qubits:

(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
swapping qubits 0 and 2

- complementing qubit 0
swapping qubits 0 and 2.

Similarly: swapping qubits i, j.

Example with more qubits:

(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto \\
(a_0, a_2, a_4, a_6, a_1, a_5, a_3, a_7)
is a “reversible XOR gate” =
(q_0, q_1, q_2) \mapsto (q_0, q_1, q_0).

Example with more qubits:

(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto \\
(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 a “controlled NOT gate”:
(q_0, q_1, q_2) \mapsto (q_0, q_1, q_2, q_3, \ldots, q_{31}).

Example with more qubits:

(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto \\
(a_0, a_2, a_4, a_6, a_1, a_5, a_3, a_7)
is a “reversible XOR gate” =
(q_0, q_1, q_2) \mapsto (q_0, q_1, q_0).

Example with more qubits:

(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto \\
(a_0, a_2, a_4, a_6, a_1, a_5, a_3, a_7)
is a “controlled NOT gate”:
(q_0, q_1, q_2) \mapsto (q_0, q_1, q_2, q_3, \ldots, q_{31}).
(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).

Similarly: 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.
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 measured as \((q_0 \oplus 1, q_1, q_2)\), representing \(q \oplus 1\), with probability \(|a_q|^2 = P_r|a_r|^2\).

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

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

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

\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_3, a_2, a_4, a_5, a_7, a_6)\)
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:
\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31})
\mapsto (a_0, a_1, a_3, a_2, a_4, a_5, a_7, a_6, a_8, a_9, a_{11}, a_{10}, a_{12}, a_{13}, a_{15}, a_{14}, a_{16}, a_{17}, a_{19}, a_{18}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{27}, a_{26}, a_{28}, a_{29}, a_{31}, a_{30})\).
swapping qubits 0 and 2.

Complement qubit 2:

\[ \text{swapping qubits 0 and 2} \rightarrow (q_2, q_1, q_0). \]

**Example with more qubits:**

- Controlled NOT gate:
  \[ \text{controlled NOT gate} = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \]

- Reversible XOR gate:
  \[ \text{reversible XOR gate} = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \]

- Toffoli gate:
  \[ \text{Toffoli gate} = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \]
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \rightarrow (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)
\rightarrow (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) \rightarrow (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})
\rightarrow (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_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, q_0).

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_{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}).
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:
\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_3, a_2, a_4, a_5, a_7, a_6)\)
\((a_0, a_1, a_3, a_2, a_4, a_5, a_7, a_6) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\)

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:

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

Example with more qubits:

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

The example of a permutation $p$ is:

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

is a "reversible XOR gate" = "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:

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

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

Example with more qubits:

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

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:

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

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

Reversible computation

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

\[
\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) & \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6) \\
\text{is a “Toffoli gate”} &= \text{“controlled controlled NOT gate”:} \\
(q_0, q_1, q_2) & \mapsto (q_0 \oplus q_1 q_2, q_1, q_2).
\end{align*}
\]

Example with more qubits:

\[
\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}) & \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}) \\
& \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{29}, a_{31}, a_{30}).
\end{align*}
\]
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
$(a_{p(0)}, a_{p(1)}, \ldots, a_{p(2^n-1)}) 
\mapsto (a_0, a_1, \ldots, a_{2^n-1})$:
\((a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)\)

\((a_0, a_1, a_2, a_3, a_4, a_5, a_7, a_6)\)
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_{22}, a_{23}, 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
\((a_p(0), a_p(1), \ldots, a_p(2^n - 1))\)
\(\mapsto (a_0, a_1, \ldots, a_{2^n - 1})\):

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

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

The Toffoli gate is a "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:

$$(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{26}, a_{27}, a_{28}, a_{29}, a_{30}, a_{31}) \mapsto (a_1, a_2, a_3, a_4, a_5, a_7, a_6, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{26}, a_{27}, a_{28}, a_{29}, a_{31}, a_{30}).$$
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 $(a_{p(0)}, a_{p(1)}, \ldots, a_{p(2^n-1)}) \mapsto (a_0, a_1, \ldots, a_{2^n-1})$:

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 $(a_5, a_6, a_7) \mapsto (a_5, a_7, a_6) = \text{controlled NOT gate}$:
$(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)$.

Example: Let’s compute $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)$.

Example: Let’s compute $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6)$.
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

\[
\begin{align*}
(a_p(0), a_p(1), \ldots, a_p(2^n-1)) \\
\mapsto (a_0, a_1, \ldots, a_{2^n-1}).
\end{align*}
\]

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 \( q \mapsto q + 1 \mod 8 \):

\[
\begin{align*}
q_0 & \downarrow \downarrow \downarrow \\
& \Rightarrow q_2
\end{align*}
\]

\[
\begin{align*}
q_0 & \downarrow \downarrow \downarrow \\
& \Rightarrow q_1
\end{align*}
\]

\[
\begin{align*}
q_0 & \downarrow \downarrow \downarrow \\
& \Rightarrow q_0 \oplus 1
\end{align*}
\]

\[
\begin{align*}
q_1 & \downarrow \downarrow \downarrow \\
& \Rightarrow q_1 \oplus q_0
\end{align*}
\]

\[
\begin{align*}
q_2 & \downarrow \downarrow \downarrow \\
& \Rightarrow q_2 \oplus \text{c1} = q_1 q_0
\end{align*}
\]
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 $(a_p(0), a_p(1), \ldots, a_p(2^n - 1)) \mapsto (a_0, a_1, \ldots, a_{2^n - 1})$:

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 $q \mapsto q + 1 \text{ mod } 8$.

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

   $c_1 = q_1q_0$

   $q_0 \oplus 1$

   $q_1 \oplus q_0$

   $q_2 \oplus c_1$
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

\[
(a_{p(0)}, a_{p(1)}, \ldots, a_{p(2^n-1)}) \mapsto (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)
\]

1. Build a traditional circuit to compute \( j \mapsto p(j) \)

\[
\text{NOT/XOR/AND gates.}
\]

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 \( q \mapsto q + 1 \text{ mod } 8 \).

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

\[
q_0 \quad q_1 \quad q_2
\]

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

\[
c_1 = q_1 q_0
\]

2. Convert into reversible gates:

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

\[
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3).
\]
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 $(a_p(0); a_p(1); \ldots; a_p(2^n - 1)) \mapsto (a_0; a_1; \ldots; a_{2^n - 1})$:

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 $q \mapsto q + 1 \bmod 8$.

1. Build a traditional circuit:
   - $c_1 = q_1 q_0$
   - $q_0 \oplus 1 q_1 \oplus q_0 q_2 \oplus c_1$

2. Convert into reversible gates.
   - Toffoli for $q_2 \leftarrow q_2 \oplus q_1 q_0$:
     
     $\{ \begin{array}{l}
     (1 - u_2 d) q_2 \mapsto q_2, \\
     (1 - u_2 d) q_0 \mapsto q_0 \\
     (1 - u_2 d) a \mapsto a
     \end{array} \}$

   $\{ \begin{array}{l}
     \{ \begin{array}{l}
     (1 - u_2 d) q_2 \mapsto q_2, \\
     (1 - u_2 d) q_0 \mapsto q_0 \\
     (1 - u_2 d) a \mapsto a
     \end{array} \}$
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 \(q \mapsto q + 1 \mod 8\).

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 $q \mapsto q + 1 \mod 8$.

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 $q_1 \leftarrow q_1 \oplus q_0$:

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

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

\[ q_0 \downarrow \downarrow \rightarrow q_1 = q_1 q_0 \]

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)\)

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 $p$ 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 ⇒ slow.

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

Toffoli for \( q_2 \leftarrow q_2 \oplus q_1 q_0 \):
\[
\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) & \mapsto (a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3).
\end{align*}
\]

Controlled NOT for \( q_1 \leftarrow q_1 \oplus q_0 \):
\[
\begin{align*}
(a_0, a_1, a_2, a_7, a_4, a_5, a_6, a_3) & \mapsto (a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5).
\end{align*}
\]

NOT for \( q_0 \leftarrow q_0 \oplus 1 \):
\[
\begin{align*}
(a_0, a_7, a_2, a_1, a_4, a_3, a_6, a_5) & \mapsto (a_7, a_0, a_1, a_2, a_3, a_4, a_5, a_6).
\end{align*}
\]

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 $q_2 \leftarrow q_2 \oplus q_1q_0$:

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

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 $c_1 \leftarrow q_1q_0$ was merged into $q_2 \leftarrow q_2 \oplus c_1$.

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

For \( q_2 \leftarrow q_2 \oplus q_1 q_0 \):

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

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 \( q_0 \leftarrow q_0 \oplus 1 \):

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

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 \textit{fast} circuits.

Also, it didn’t need extra storage:

circuit operated “in place” after \( c_1 \leftarrow q_1 q_0 \) was merged into \( q_2 \leftarrow q_2 \oplus c_1 \).

Typical circuits aren’t in-place.

Start from any circuit:

inputs \( b_1, b_2, \ldots, b_i, b_{i+1} = 1 \oplus b_f(i+1) b_g(i+1) \);

\( b_{i+2} = 1 \oplus b_f(i+2) b_g(i+2) \);

\[\vdots\]

\( b_T = 1 \oplus b_f(T) b_g(T) \)

specified outputs.
2. Convert into reversible gates.

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

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

For $q_1 \leftarrow q_1 \oplus q_0$:

$$(a_0; a_1; a_2; a_3) \mapsto (a_3; a_6; a_5).$$

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.

Start from any circuit:

inputs $b_1, b_2, \ldots, b_T$:

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

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

$$\vdots$$

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

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 $b_1, b_2, \ldots, b_i$;

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

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

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

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).

Reversible and clean:
after finishing dirty computation, set non-outputs back to 0,
by repeating same operations on non-outputs in reverse order.
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 \( q_1 q_0 \) was merged into \( q_2 \oplus c_1 \).

Didn'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.

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, output).

Clean reversible computation:

(inputs, zeros, zeros) \( \mapsto \) (inputs, zeros, output).
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).
Start from any circuit:

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

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

\[ \vdots \]

\[ b_T = 1 \oplus b_f(T) b_g(T); \]

specified outputs.

Reversible but dirty:

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

\[ b_{i+1} \leftarrow 1 \oplus b_{i+1} \oplus b_f(i+1) b_g(i+1); \]
\[ b_{i+2} \leftarrow 1 \oplus b_{i+2} \oplus b_f(i+2) b_g(i+2); \]

\[ \vdots \]

\[ b_T \leftarrow 1 \oplus b_T \oplus b_f(T) b_g(T). \]

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, zeros) $\mapsto$

(inputs, zeros, 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: (inputs) \rightarrow (inputs, dirt, outputs).

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

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

Given fast circuit for $p$ and fast circuit for $p-1$, build fast reversible circuit for $(x, zeros) \rightarrow (p(x), zeros)$. 
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) \rightarrow 
(inputs, dirt, outputs).

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

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

Given fast circuit for $p$ and fast circuit for $p-1$, build fast reversible circuit for $(x, zeros) \leftrightarrow (p(x), zeros)$.
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) \rightarrow 
(inputs, dirt, outputs).

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

Clean reversible computation:  
(inputs, zeros, zeros) \rightarrow 
(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 \( p^{-1} \), 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:

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

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

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

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

$\left(a_{p(0)}, a_{p(1)}, \ldots, a_{p(2^n-1)}\right)$

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

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 lose time.

Many subtle tradeoffs.

Crude “poly-time” analyses don’t care about this, but serious cryptanalysis is much more precise.
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 \(C^{2^{n+L}} \rightarrow C^{2^{n+L}}\).

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

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

Permutation on first $2^n$ entries is

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

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

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

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 circuit for \( p \)

Fast circuit for \( p^{-1} \),

Fast reversible circuit for \([x; \text{zeros}] \mapsto [p(x); \text{zeros}]\).

Replacing reversible bit operations

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

Permutation on first \( 2^n \) entries is

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

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

First $2^n$ entries is $(p(2^n-1); p(2^n-2); \ldots; p(1); 0; 0; \ldots; 0)$.

Vectors

$2^n$ entries

permutation $2^n-2^n$ 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”:

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

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 $(x; \text{zeros}) \mapsto (p(x); \ldots)$. Replace reversible bit operations with Toffoli gates etc. permuting $C_{2n}^2 + z \rightarrow C_{2n}^2 + z$.

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

Typically prepare vectors supported on first $2n$ entries so don't care how permutation acts on last $2n + z - 2n$ 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”:
$$(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1).$$
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”:
$(a_0, a_1) \mapsto (a_0 + a_1, a_0 - a_1)$. 
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** \( \approx \) 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:

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

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

\(\Pr[\text{output } = q] = \frac{1}{2^n}\).
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.,

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

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:

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

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” \((1, 0, 0, \ldots, 0)\).
Quantum operations, part 2

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

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

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

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

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 \(x \in \{0, 1\}\).
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 \( s \in \{0, 1\} \) 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:

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

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

We don’t have enough data
if \(f\) has many periods.
Assume: only periods are 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 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 $f$?

We don’t have enough data
if $f$ has many periods.
Assume: only periods are 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 \((1, 1, 1, \ldots, 1)\)
produce any output:
\[ \Pr[\text{output} = q] = \frac{1}{2^n} \]

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: only periods are 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

is much, much, much faster.

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

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: only periods are 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 is much, much, much faster.

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:
vector \((1, 1, 1, \ldots, 1, 0, 0, \ldots)\)
with \( 2^n \) entries 1, others 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: only periods are 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 is much, much, much faster.

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
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: only periods are 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 is much, much, much faster.

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

Assume: nonzero \( s \in \{0, 1\}^n \)
\[ f(x) = f(x \oplus s) \]
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: only periods are 0, \( s \).

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

Simon's algorithm

is much, much, much faster.

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 \( s \).
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: only periods are 0, $s$.

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

Simon's algorithm
is much, much, much faster.

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

Apply $n$-fold Hadamard
for reversible $f$ computation:
$1$ in position $(q; 0; 0)$
moves to position $(q; f(q); \ldots)$.

By symmetry,
output is orthogonal to $s$.

Repeat $n + 10$ times.
Use Gaussian elimination
for Gaussian elimination.

Apply $n$-fold Hadamard
for reversible $f$ computation:
$1$ in position $(q, 0, 0)$
moves to position $(q, f(q), 0, \ldots)$.

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

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

Apply Gaussian elimination
for Gaussian elimination.

Gaussian elimination
for Gaussian elimination.
Simon’s algorithm
is much, much, much faster.

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 $s$. 

Simon’s algorithm is much, much, much faster.

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 $s$. 
Simon's algorithm is much, much, much faster. 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: $(1, 1, 1, \ldots; 0, 0, 0, \ldots)$. 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 $s$. 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.
Simon's algorithm is much, much, much faster. Say \( f \) maps \( n \) bits to \( m \) bits, using \( z \) "ancilla" bits for reversibility. Prepare \( n + m + z \) qubits in pure zero state: \( (1, 0, 0, \ldots) \).

Use \( n \)-fold Hadamard to move first \( n \) qubits into uniform superposition: \( (1, 1, 1, \ldots; 0, 0, 0, \ldots) \) 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 \( s \).

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.
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 $s$.

---

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 $2^n$ reversible computations of $f$.
Typically: reversibility overhead is small enough that this easily beats traditional algorithm.
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 $s$.

---

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

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

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 \( s \).

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

\[
b_q = \begin{cases} 
  -a_q & \text{if } f(q) = 0, \\
  a_q & \text{otherwise}.
\end{cases}
\]

This is fast.

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

Repeat steps 1 and 2 about \( \frac{\pi}{4} \cdot 2^{(n/2)} \) 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.

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 steps 1 and 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
\[ b_q = -a_q \text{ if } f(q) = 0, \]
\[ b_q = a_q \text{ otherwise.} \]
This is fast.

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

Repeat steps 1 and 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
\[
\begin{align*}
b_q &= -a_q \text{ if } f(q) = 0, \\
b_q &= a_q \text{ otherwise.}
\end{align*}
\]
This is fast.

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

Repeat steps 1 and 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
\[ b_q = -a_q \text{ if } f(q) = 0, \]
\[ b_q = a_q \text{ otherwise.} \]
This is fast.

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

Repeat steps 1 and 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 $2^{n/2}$ computations of $f$.

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

$\begin{align*}
b_q &= -a_q \text{ if } f(q) = 0, \\
b_q &= a_q \text{ otherwise.}
\end{align*}$

This is fast.

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

Repeat steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after 0 steps:
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 \( 2^n = 2^2 \) 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 steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

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
\[
b_q = -a_q \text{ if } f(q) = 0,
\]
\[
b_q = a_q \text{ otherwise}.
\]
This is fast.

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

Repeat steps 1 and 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 steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Graph of $q \mapsto a_q$

for an example with $n = 12$

after Step 1 + Step 2 + Step 1:
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 steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$.

Graph of $q \mapsto a_q$

for an example with $n = 12$

after $3 \times (\text{Step 1 + Step 2})$:  

![Graph](image_url)
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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Graph of $q \mapsto a_q$ for an example with $n = 12$ after $4 \times (\text{Step 1} + \text{Step 2})$: 

![Graph](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 steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $6 \times (\text{Step 1 + Step 2})$:
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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Graph of $q \mapsto a_q$

for an example with $n = 12$

after $7 \times (\text{Step 1} + \text{Step 2})$: 

![Graph](attachment:graph.png)
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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $8 \times (\text{Step 1} + \text{Step 2})$: 

![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 steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $10 \times (\text{Step 1} + \text{Step 2})$:
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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $11 \times (\text{Step 1 } + \text{ Step 2})$: 

![Graph of $q \mapsto a_q$ for an example with $n = 12$ after $11 \times (\text{Step 1 } + \text{ Step 2})$](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 steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $13 \times (\text{Step 1 + Step 2})$: 
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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$ for an example with $n = 12$ after $14 \times (\text{Step 1} + \text{Step 2})$: 

![Graph](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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Graph of $q \mapsto a_q$

for an example with $n = 12$

after $15 \times (\text{Step 1} + \text{Step 2})$: 

![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 steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$ for an example with $n = 12$ after $17 \times (\text{Step 1} + \text{Step 2})$: 

![Graph of q ↦→ 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $18 \times (\text{Step 1} + \text{Step 2})$: 

\begin{itemize}
  \item[$\vdots$]
\end{itemize}
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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$ for an example with $n = 12$ after $19 \times (\text{Step 1 + Step 2})$: 

![Graph of $q \mapsto a_q$]
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a ← b$ 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $20 \times (\text{Step 1} + \text{Step 2})$: 

\[\begin{array}{c}
\text{Graph of } q \mapsto a_q \\
\text{for an example with } n = 12 \\
after 20 \times (\text{Step 1} + \text{Step 2}): \\
\end{array}\]
Start from uniform superposition over all $n$-bit strings $q$.

Step 1: Set $a \leftarrow b$ where

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

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

This is fast.

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

Repeat steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $30 \times (\text{Step 1} + \text{Step 2})$: 

\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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$ for an example with $n = 12$ after $35 \times (\text{Step 1} + \text{Step 2})$: 

Good moment to stop, measure.
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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$ for an example with $n = 12$ after $40 \times (\text{Step 1} + \text{Step 2})$: 

![Graph of $q \mapsto a_q$ for an example with $n = 12$ after $40 \times (\text{Step 1} + \text{Step 2})$.]
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 steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

Measure the $n$ qubits.

With high probability this finds $s$. 

Graph of $q \mapsto a_q$

for an example with $n = 12$

after $50 \times (\text{Step 1} + \text{Step 2})$: 

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 steps 1 and 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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $70 \times (\text{Step 1 + Step 2})$:
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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $80 \times (\text{Step 1} + \text{Step 2})$: 

![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 steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Graph of $q \mapsto a_q$
for an example with $n = 12$
after $90 \times (\text{Step 1 + Step 2})$: 

\[ 
\begin{array}{c}
\text{Graph of } q \mapsto a_q \\
\text{for an example with } n = 12 \\
after 90 \times (\text{Step 1 + Step 2}):
\end{array}
\]
Start from uniform superposition over all $n$-bit strings $q$.

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

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

Repeat steps 1 and 2 about $0.58 \cdot 2^{0.5n}$ times.

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

Very bad stopping point.
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.

“Grover diffusion”.

- Negate $a$ around its average.
- Also fast.

Repeat steps 1 and 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 $\approx (i=4)^2 0.5n$ iterations.

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 $f(q) = 0$, $b_q = a_q$ otherwise.
This is fast.

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

Repeat steps 1 and 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.

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 $f(q) = 0$, $b_q = a_q$ otherwise. This is fast.

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

Repeat steps 1 and 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.

Graph of $q \mapsto a_q$ for an example with $n = 12$ after $100 \times$ (Step 1 + Step 2):

Very bad stopping point.
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.

$q \mapsto a_q$ 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.

⇒ Probability is $\approx 1$
after $\approx (\pi/4)2^{0.5n}$ iterations.
Graph of $q \mapsto a_q$

Example with $n = 12$

after $100 \times (\text{Step 1} + \text{Step 2})$:

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 $\approx (\pi/4)^{0.5n}$ 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.

⇒ 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

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.

⇒ 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. for roots $q$;
2. 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 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.

Reality: state-of-the-art cryptanalytic algorithms are almost never proven.
Note 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.
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 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.
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.
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.
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? ⇒ 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.