Quantum Computing

Chapter 06: Searching and Cryptography

Prachya Boonkwan

Version: January 14, 2026

Sirindhorn International Institute of Technology
Thammasat University, Thailand

Who? Me?

- Nickname: Arm (P'/N' Arm, etc.)
- Born: Aug 1981
- Work
  - Researcher at NECTEC 2005-2024
  - Lecturer at SIIT, Thammasat University 2025-now
- Education
  - B.Eng & M.Eng in Computer Engineering, Kasetsart University, Thailand
  - Obtained Ministry of Science and Technology Scholarship of Thailand in early 2008
  - Did a PhD in Informatics (AI & Computational Linguistics) at University of Edinburgh, UK from 2008 to 2013

Table of Contents

Grover’s Search Algorithm

Information Retrieval

- Information retrieval is the task of identifying and retrieving information sources relevant to an information need
- Indexing: Each document in the dataset is extracted for its keywords (key)
- Search: The user's query is sequentially matched against each document's key; if they match, the solution (correct output value) will be retrieved for the user
- Limitation: Time complexity of unstructured search is $O(n)$, where n is the size of the search space

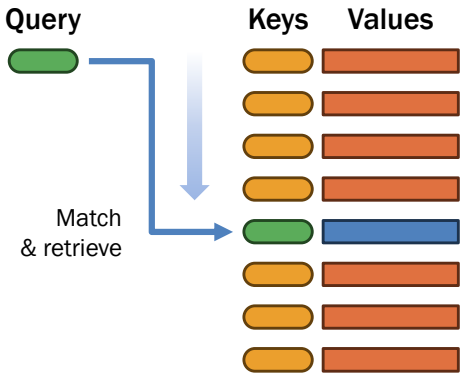


Figure 1: Information retrieval

Quantum Search Algorithm

- Quantum search algorithm is a quantum method for unstructured search, where the solution is retrieved with the highest probability in $O(\sqrt{n})$ time
- Motivation: We can assign a probability distribution of finding the desired solution to the entire search space w.r.t. the query, and gradually amplify the probability of the solution via the oracle function
- Oracle function

$$f(x) = \begin{cases} 1 & x = x_{\text{solution}} \\ 0 & \text{otherwise} \end{cases}$$

- We can construct the oracle operator by sign-flipping the solution y:

$$\begin{aligned} \mathbf{U}_f &= \underbrace{\sum_{x \neq y} |x\rangle\langle x|}_{\text{non-solutions}} - \underbrace{|y\rangle\langle y|}_{\text{solution}} \\ &= \sum_{x \neq y} (-1)^0 |x\rangle\langle x| \\ &\quad + (-1)^1 |y\rangle\langle y| \\ &= \sum_{k=1}^{2^N} (-1)^{f(\mathbb{B}_N(k))} |\mathbb{B}_N(k)\rangle\langle \mathbb{B}_N(k)| \end{aligned}$$

- Note that $\mathbf{U}_f |x\rangle = (-1)^{f(x)} |x\rangle$

Overview of Grover's Search Algorithm

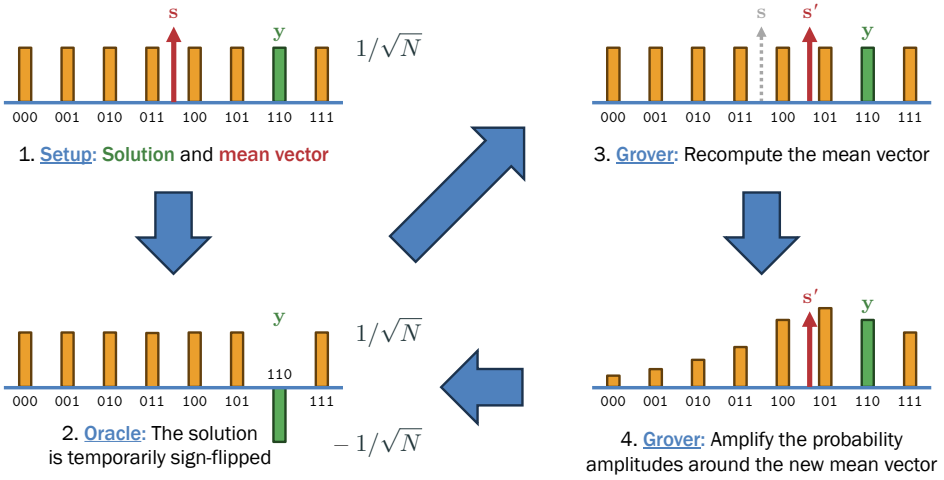


Figure 2: An overview of Grover's search algorithm

Sign-Flipping and Amplitude Amplification

- Step 1: Let $\left|x^{(0)}\right\rangle = |0\rangle^{\otimes K}$ and compute the mean vector

$$\left|s^{(0)}\right\rangle = \mathrm{H}\left|x^{(0)}\right\rangle$$

- Step 2: Temporarily sign-flip the solution

$$\left|q^{(i)}\right\rangle = \mathrm{U}_f\left|s^{(i)}\right\rangle$$

- Step 3: Recompute the mean vector

$$\left|s^{(i+1)}\right\rangle = \mathrm{D}\,\mathrm{H}\left|q^{(i)}\right\rangle$$

- Step 4: Amplify the probability amplitudes

$$\left|x^{(i+1)}\right\rangle = \mathrm{H}\left|s^{(i+1)}\right\rangle$$

- Diffusion operator D is a Householder reflection through the axis $\left|s^{(i)}\right\rangle$:

$$\mathrm{D} = 2|0\rangle\langle 0| - \mathrm{I}$$

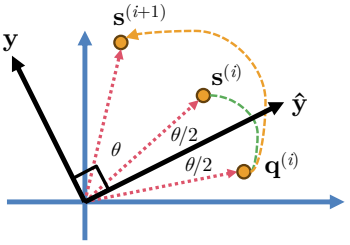


Figure 3: Geometric interpretation of Grover's search algorithm. The vector $\hat{\mathbf{y}} = \sum_{x \neq y} |\mathbb{B}_N(x)\rangle$ is the sum of non-solutions. The angle always increases by θ.

Optimal Number of Iterations

- In each iteration we recompute the mean vector $\left|s^{(i+1)}\right\rangle$ from $\left|s^{(i)}\right\rangle$ by rotating by constant angle θ
- We want to rotate for a certain number of iterations k^* to approximate the solution $\mathbf{y}$, perpendicular to the reflection basis $\hat{\mathbf{y}}$

$$\begin{aligned} k^*\theta + \theta/2 &= \pi/2 \\ k^* &= \left\lfloor \frac{\pi}{4\theta} - \frac{1}{2} \right\rfloor \end{aligned}$$

- Consider a uniform superposition

$$\left|s^{(0)}\right\rangle = \mathbf{H}(\mathbf{y}+\hat{\mathbf{y}}) = \mathbf{y}\sqrt{\frac{1}{n}} + \hat{\mathbf{y}}\sqrt{\frac{n-1}{n}}$$

- Since $\left|s^{(0)}\right\rangle$ is a unit vector of the form $\mathbf{x}\sin\theta + \mathbf{x}_\perp\cos\theta$, we obtain

$$\begin{aligned} \sqrt{1/n} &= \sin\theta \\ &\approx \theta \qquad \text{for small } \theta \end{aligned}$$

- Therefore, we obtain the optimal number of iterations

$$\begin{aligned} k^* &= \left\lfloor \frac{\pi}{4}\sqrt{n} - \frac{1}{2} \right\rfloor \\ &\approx \left\lfloor \frac{\pi}{4}\sqrt{n} \right\rfloor \end{aligned}$$

  where n is the size of the search space
- Going beyond k^* is called overshooting

Grover's Search Algorithm

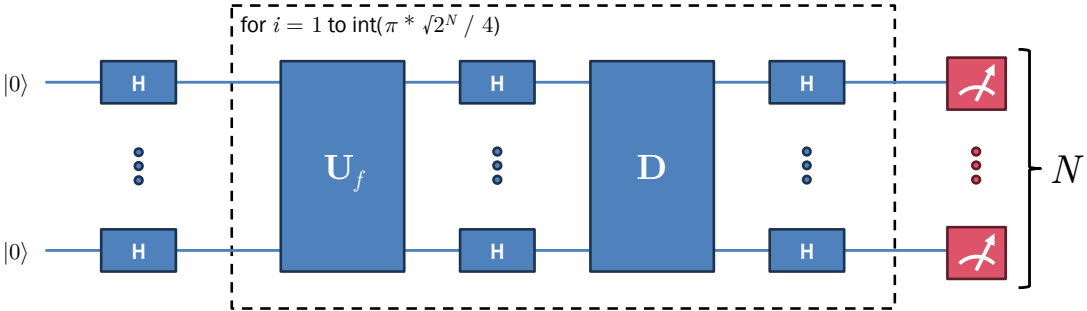


Figure 4: Grover's search algorithm

- PennyLane:
  - Steps 1 and 2 are done by `qml.FlipSign(statevec, wires=...)`
  - Steps 3 and 4 (**H D H**) are done by `qml.templates.GroverOperator(wires=...)`

Grover's Search Algorithm (Actual Implementation)

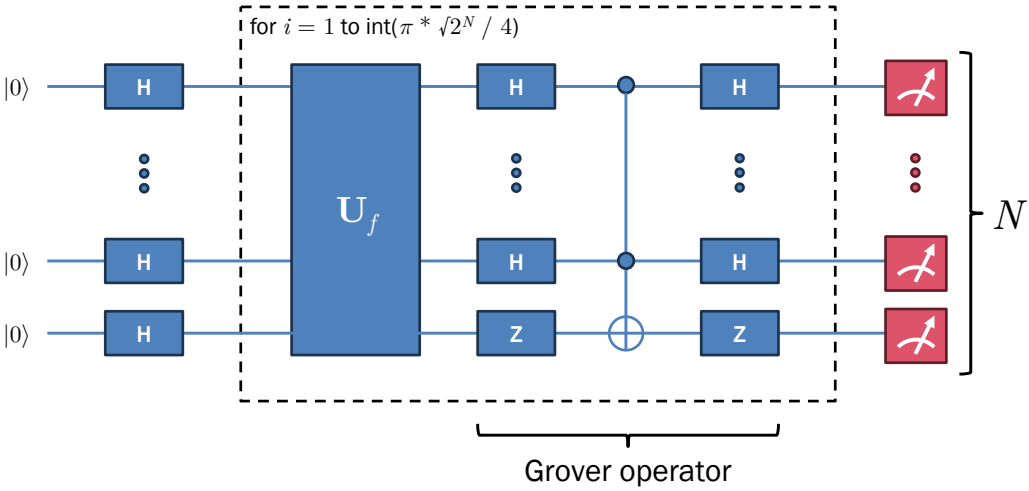


Figure 5: Actual implementation of Grover's search algorithm

Grover's Search Algorithm in PennyLane

```
soln = np.array([1,1,0,1])                                  # Solution: 1101
no_qubits = len(soln)
wires = range(no_qubits)
dev = qml.device('default.qubit', wires=wires)

def oracle():                                               # Treat it as a black box
    qml.FlipSign(soln, wires=wires)

def uniform_superposition(wires):
    for i in wires:
        qml.Hadamard(wires=i)

@qml.qnode(device=dev, shots=1000)
def grover_search():                    # Note that the solution is always hidden in the oracle
    no_iters = int(np.pi / 4 * np.sqrt(2 ** no_qubits))
    uniform_superposition(wires)
    for i in range(no_iters):
        oracle()
        qml.templates.GroverOperator(wires=wires)           # Diffusion operator
    return qml.counts(wires=range(no_qubits))

grover_search()                         # Expected result: {'1101': 1000}
```

Grover's Search Algorithm for Multiple Solutions

- We can incorporate multiple solutions $y_1, \ldots, y_m$ and non-solutions $x_1, \ldots, x_{n-m}$, where $n = 2^N$ is the size of search space:

$$\begin{aligned} \mathbf{U}_f &= \underbrace{\sum_{j=1}^{n-m} \left|x_j\right\rangle \left\langle x_j\right|}_{\text{non-solutions}} - \underbrace{\sum_{j=1}^{m} \left|y_j\right\rangle \left\langle y_j\right|}_{\text{solutions}} \\ &= \sum_{j=1}^{n-m} (-1)^0 \left|x_j\right\rangle \left\langle x_j\right| \\ &\quad + \sum_{j=1}^{m} (-1)^1 \left|y_j\right\rangle \left\langle y_j\right| \\ &= \sum_{k=1}^{n} (-1)^{f(\mathbb{B}_N(k))} |\mathbb{B}_N(k)\rangle \langle \mathbb{B}_N(k)| \end{aligned}$$

- Optimal number of iterations for multiple solutions is derived from the fact that

$$\left|s^{(0)}\right\rangle = \mathrm{H}(\mathbf{y} + \hat{\mathbf{y}}) \quad = \mathbf{y}\sqrt{\frac{m}{n}} + \hat{\mathbf{y}}\sqrt{\frac{n-m}{n}}$$

- Since $\left|s^{(0)}\right\rangle$ is a unit vector of the form $\mathbf{x}\sin\theta + \mathbf{x}_\perp \cos\theta$, we obtain

$$\sqrt{m/n} \;=\; \sin\theta \;\approx\; \theta \quad \text{for small } \theta$$

- The optimal number of iterations is

$$k^* \;=\; \left\lfloor \frac{\pi}{4\theta} - \frac{1}{2} \right\rfloor \;\approx\; \left\lfloor \frac{\pi}{4}\sqrt{\frac{n}{m}} \right\rfloor$$

where m is the number of solutions

Grover's Search Algorithm for Multiple Solutions in PennyLane

```
solns = np.array([ [1,1,0,1], [0,1,0,1] ])                # Solutions: 1101 and 0101
[no_solns, no_qubits] = solns.shape
wires = range(no_qubits)
dev = qml.device('default.qubit', wires=wires)

def oracle():
    for soln in solns:                                    # We sign-flip on each solution
        qml.FlipSign(soln, wires=wires)

def uniform_superposition(wires):
    for i in wires:
        qml.Hadamard(wires=i)

@qml.qnode(device=dev, shots=1000)
def grover_search():                    # Observe the change in the iteration number
    no_iters = int(np.pi / 4 * np.sqrt(2 ** no_qubits / no_solns))
    uniform_superposition(wires)
    for i in range(no_iters):
        oracle()
        qml.templates.GroverOperator(wires=wires)         # Diffusion operator
    return qml.counts(wires=range(no_qubits))

grover_search()                         # Expected result: {'0101': 500, '1101': 500}
```

Exercise 6.1: Grover's search algorithmn

Compute a unitary operator for the oracle functions, whose solutions are as follows.

1. 01
2. 010
3. 110
4. 1101
5. 001 and 100
6. 1010 and 0101
7. 010, 011, and 110
8. all binary representations that consist of one 1 and four 0's

Answer the following questions.

9. Compute the optimal number of iterations k^* for each of the oracle functions in Q.1-Q.8.
10. Compute the rotation angle θ for each of the oracle functions in Q.1-Q.8.
11. Implement PennyLane code for Grover's search algorithm using the oracle functions in Q.1-Q.8.
12. Discuss what will happen when we overshoot in searching with Grover's search algorithm.

Quantum Fourier Transform

Spectral Density

- We can describe a time series $f(t)$ in terms of power distribution according to the frequency components
- Assumption: Time series is a mixture of periodic sinusoid waves (seasonal patterns) of different frequencies ω:

$$\begin{aligned}\Psi(A,\omega,t) &= A\,\mathrm{cis}(-\omega t+\theta)\\ &= \Psi_0\,\mathrm{cis}(-\omega t)\end{aligned}$$

  where $\Psi_0 = A\,\mathrm{cis}(\theta)$ is its complex amplitude, and θ is its phase
- We want to transform a time series into spectra (squared amplitudes $|\Psi_0|^2$) in the frequency domain

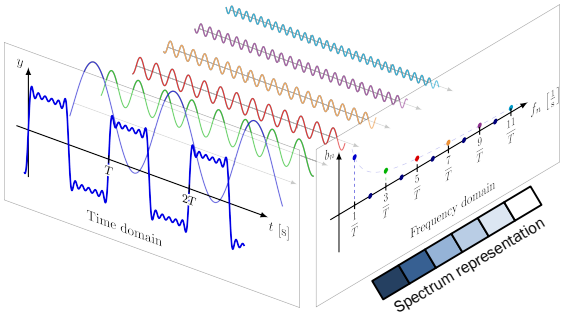


Figure 6: Spectral analysis of a time series [Credit: https://dibsmethodsmeetings.github.io/fourier-transforms/]

Classical Fourier Transform

- Fourier transform of a time series $x(t)$ is a function of ω that reflects how well $x(t)$ aligns with the unit wave of frequency ω:

$$\begin{aligned} F\{x\}(\omega) &= x(t) \odot \Psi(1, \omega, t) \\ &= \int_{-\infty}^{\infty} x(t) \cdot \left[\Psi(1, \omega, t)\right]^* dt \\ &= \int_{-\infty}^{\infty} x(t) \cdot \text{cis}(\omega t) dt \end{aligned}$$

where $\odot$ is the cross-correlation operator:

$$\mathbf{u} \odot \mathbf{v} = \mathbf{u}^\top \mathbf{v}^*$$

- The frequency is usually a multiple of 2π, which is one round of a circle in radian

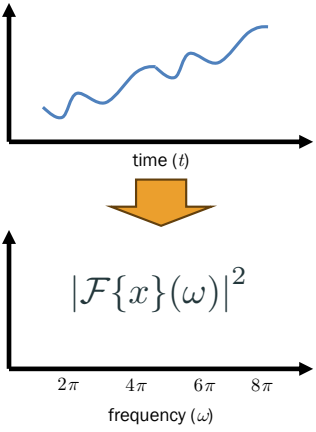


Figure 7: Frequency domain

Classical Fourier Transform

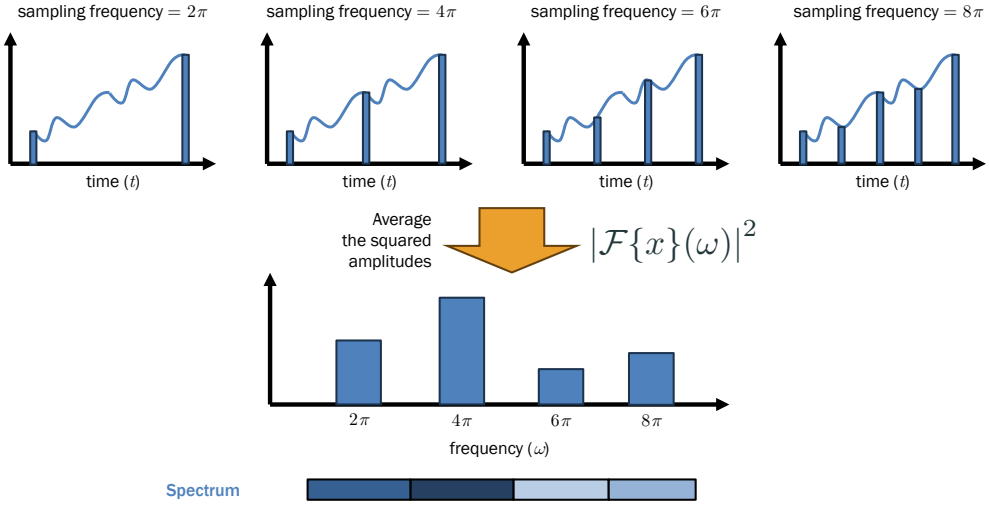
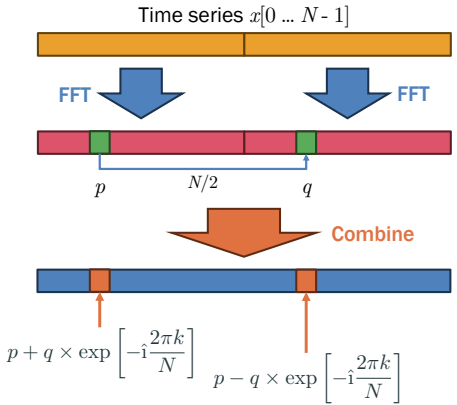


Figure 8: Classical Fourier transform via sampling. Time complexity is $O(n^2)$ due to iterative steps.

Fast Fourier Transform (FFT)

- Popular classical algorithm for discrete Fourier Transform based on recursion
- The length of the time series is in the form of 2^K



- Base case ($N = 1$)

$$\mathcal{F}(x[k]) = x[k]$$

- Recursive case ($N > 1$)

$$p = \mathcal{F}[x[k]]$$

$$q = \mathcal{F}[x[k + \frac{N}{2}]]$$

$$\mathcal{F}[x[k]] = p + q \times \exp\left[-\hat{\imath}\frac{2\pi k}{N}\right]$$

$$\mathcal{F}[x[k + \frac{N}{2}]] = p - q \times \exp\left[-\hat{\imath}\frac{2\pi k}{N}\right]$$

Figure 9: Fast Fourier transform. Time complexity is $O(n \log n)$, where n is the length $\mathbf{x}$.

Quantum Fourier Transform (QFT)

- Discrete Fourier transform converts an input time series $\mathbf{x} = [x_0, \ldots, x_{N-1}]$ into a frequency-domain vector $\mathbf{y}$, where

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j r_N^{-jk}$$

and $r_N = \text{cis}(2\pi/N)$ is the N root of unity

- In quantum computing, we treat a time series as a superposition of data points

$$[x_0, \ldots, x_{2^N-1}] \quad \Rightarrow \quad \frac{1}{\sqrt{N}} \sum_{j=0}^{2^N-1} x_k \, |\mathbb{B}_N(j)\rangle$$

Note that 2^N data points become N qubits

- We consider quantum Fourier transform as state mapping

$$|\mathbb{B}_N(k)\rangle \quad \mapsto \quad \frac{1}{\sqrt{N}} \sum_{j=0}^{2^N-1} r_N^{jk} \, |\mathbb{B}_N(j)\rangle$$

- We define the QFT operator as

$$\mathbf{F} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & r^1 & r^2 & r^3 & \cdots & r^N \\ 1 & r^2 & r^4 & r^6 & \cdots & r^{2N} \\ 1 & r^3 & r^6 & r^9 & \cdots & r^{3N} \\ 1 & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r^N & r^{2N} & r^{3N} & \cdots & r^{NN} \end{bmatrix}$$

where $r_N = \text{cis}(2\pi/N)$

Quantum Algorithm for Fourier Transform

- We can imitate the root of unity r_N^k by the dyadic rational phase gate $\mathbf{R}_k$, where

$$\mathbf{R}_k = \begin{bmatrix} 1 & 0 \\ 0 & \text{cis}(2\pi/2^k) \end{bmatrix}$$

Note that $\mathbf{R}_1 = \mathbf{H}$

- Conditional dyadic rational phase gate

$$\mathbf{CR}_k(a|b) = \begin{bmatrix} 1 & 0 \\ 0 & \text{cis}(2\pi b/2^k) \end{bmatrix} \otimes \mathbf{I}$$

where the first qubit a is the input/output and the second qubit b is the control

- Self-condition: $\mathbf{CR_k}(x|x) = \mathbf{R}_k(x)$

- Fourier block: $\mathbf{FB}_N(x_1)$ maps $|0\rangle$ to $|0\rangle$, and $|1\rangle$ to $\text{cis}\left(\sum_{k=1}^{N} \frac{2\pi x_k}{2^k}\right)|1\rangle$, upon $|x_1\rangle$

$$\mathbf{FB}_N(x_1) = \prod_{k=1}^{N} \mathbf{CR}_k^{(N)}(x_1|x_k)$$

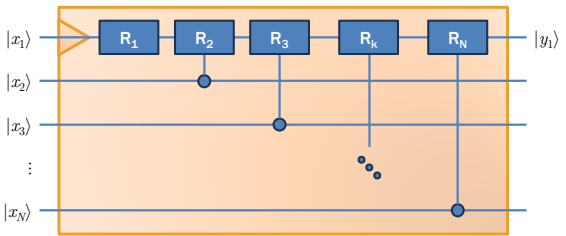


Figure 10: Fourier block $\mathbf{FB}_N(x_1)$

Implementation of Quantum Fourier Transform

- For each qubit $\left|y_j\right\rangle$, we map $|0\rangle \mapsto |0\rangle$, and $|1\rangle \mapsto \mathrm{cis}\left(\sum_{k=1}^{N-j+1} 2\pi x_{j+k}/2^k\right)|1\rangle$

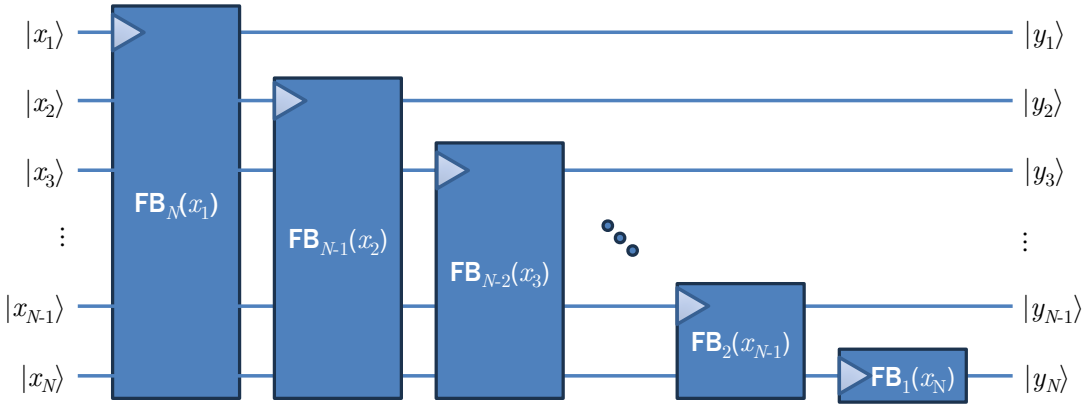


Figure 11: Quantum Fourier transform (**QFT**). Time complexity is $O(N^2)$.

Quantum Fourier Transform in PennyLane

- An implementation of quantum Fourier transform is presented below
- Quantum Fourier transform in PennyLane: `qml.QFT(wires=...)`

```
def fourier_block(target, control_wires):
    no_qubits = len(control_wires) + 1
    qml.Hadamard(wires=target)
    for wire in control_wires:
        qml.ctrl(
            qml.PhaseShift(
                2 * np.pi / np.sqrt(no_qubits),
                wires=target
            )
        )(control=wire)

def qft(wires):
    for i in range(len(wires)):
        fourier_block(wires[i], wires[i+1:])
```

Inverse Quantum Fourier Transform (InvQFT)

- Inverse discrete Fourier transform converts a frequency-domain vector $\mathbf{y} = [y_0, \ldots, y_{N-1}]$ into a time series $\mathbf{x}$

$$x_k = \frac{1}{\sqrt{2}} \sum_{j=0}^{N-1} y_j r_N^{jk}$$

  Note that the phase of r_N is positive

- We consider inverse quantum Fourier transform as state mapping

$$\frac{1}{\sqrt{N}} \sum_{j=0}^{2^N-1} r_N^{jk} |\mathbb{B}_N(j)\rangle \quad \mapsto \quad |\mathbb{B}_N(k)\rangle$$

- We define the InvQFT operator as

$$\mathbf{F}^{-1} = \mathbf{F}^*$$

$$= \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & r^{-1} & r^{-2} & r^{-3} & \cdots & r^{-N} \\ 1 & r^{-2} & r^{-4} & r^{-6} & \cdots & r^{-2N} \\ 1 & r^{-3} & r^{-6} & r^{-9} & \cdots & r^{-3N} \\ 1 & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r^{-N} & r^{-2N} & r^{-3N} & \cdots & r^{-NN} \end{bmatrix}$$

  where $r_N = \mathrm{cis}(2\pi/N)$

- Inverse property:

$$\mathbf{F}^{-1}\,\mathbf{F}\,|x\rangle = \mathbf{F}\,\mathbf{F}^{-1}|x\rangle = |x\rangle$$

Quantum Algorithm for Inverse Fourier Transform

- The inverse of $\mathbf{R}_k$ is $\mathbf{R}_k^*$, because $\mathbf{R}_k$ is a Hermitian matrix,

$$\mathbf{R}_k^* = \begin{bmatrix} 1 & 0 \\ 0 & \mathrm{cis}(-2\pi/2^k) \end{bmatrix}$$

Note that $\mathbf{R}_1^* = \mathbf{H}$

- The inverse of conditional $\mathbf{R}_k$

$$\left[\mathbf{CR}_k(a|b)\right]^* = \begin{bmatrix} 1 & 0 \\ 0 & \mathrm{cis}(-2\pi b/2^k) \end{bmatrix} \otimes \mathbf{I}$$

where the first qubit a is the input/output and the second qubit b is the control

- Inverse Fourier block: $\mathbf{IFB}_N(x_1)$ maps $|0\rangle$ to $|0\rangle$, and $|1\rangle$ to $\mathrm{cis}\left(-\sum_{k=1}^{N} \frac{2\pi x_k}{2^k}\right)|1\rangle$, upon $|x_1\rangle$

$$\mathbf{IFB}_N(x_j) = \prod_{k=1}^{N} \left[\mathbf{CR}_k^{(N)}(x_1|x_k)\right]^*$$

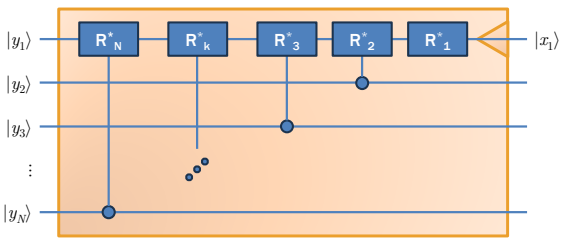


Figure 12: Inverse Fourier block $\mathbf{IFB}_N(x_1)$

Implementation of Inverse Quantum Fourier Transform

- For each qubit $\left|x_j\right\rangle$, we map $|0\rangle \mapsto |0\rangle$, and $|1\rangle \mapsto \text{cis}\left(-\sum_{k=1}^{N-j+1} 2\pi y_{j+k}/2^k\right)|1\rangle$

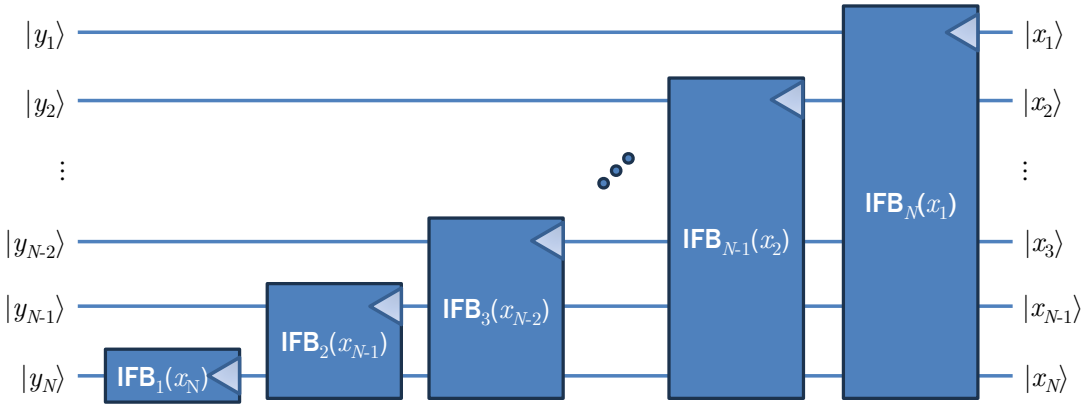


Figure 13: Inverse Quantum Fourier transform ($\mathbf{QFT}^*$)

Inverse Quantum Fourier Transform in PennyLane

- An implementation of quantum Fourier transform is presented below
- Inverse quantum Fourier transform: `qml.adjoint(qml.QFT)(wires=...)`

```
def inverse_fourier_block(target, control_wires):
    no_qubits = len(control_wires) + 1
    qml.Hadamard(wires=target)
    for wire in control_wires:
        qml.ctrl(
            qml.PhaseShift(
                - 2 * np.pi / np.sqrt(no_qubits),
                wires=target
            )
        )(control=wire)

def invqft(wires):
    for i in range(len(wires) - 1, -1, -1):
        inverse_fourier_block(wires[i], wires[i+1:])
```

Critical Comparison

	Classical FFT	QFT
Input	Explicit sequence of $n = 2^K$ items	Superposition of K qubits, whose states represent each item
Speedup paradigm	Divide-and-conquer strategy (recursion)	Quantum parallelism (iteration)
Time complexity	$O(n \log n)$ due to the divide-and-conquer	$O(K^2) = O((\log n)^2)$ due to quantum parallelism
Space complexity	$O(n)$ for all items	$O(K) = O(\log n)$
Output	Fully readable	Partially readable due to measurement collapse
Applications	Standalone data transformer	Subroutine in other quantum algorithms

Table 1: Comparison between classical FFT and QFT

Exercise 6.2: Quantum Fourier Transform

Answer the following questions.

1. Show that the cross-correlation operator has $O(2^K)$ time complexity, when there are 2^K items in the input time series.
2. Show that the classical Fourier transform has $O(2^{2K})$, when there are 2^K items in the input time series.
3. Show that the Fast Fourier Transform algorithm has $O(K2^K)$, when there are 2^K items in the input time series.
4. Show that the quantum Fourier transform algorithm has $O(K^2)$, when there are 2^K items in the input time series.
5. Discuss how we can encode a time series of 2^K items into a mixed state of K qubits.
6. Implement a Python function that converts an input time series of 2^K items into a mixed state of K qubits. Hint: Use `qml.StatePrep(state, wires=...)`.
7. Show that the dyadic rational phase gate $\mathbf{R}_1 = \mathbf{H}$.
8. Show that $\mathbf{IFB}_N\ \mathbf{FB}_N\ |x\rangle = |x\rangle$.
9. Let $\mathbf{F}$ and $\mathbf{F}^*$ be the QFT and InvQFT operators, respectively. Show that $\mathbf{F}^*\mathbf{F} = \mathbf{I}$.

Quantum Phase Estimation

Intuition of Quantum Phase Estimation

- If $|\psi\rangle$ consists of one qubit, we can decompose it into

$$\begin{aligned} |\psi\rangle &= r_1 \operatorname{cis}\theta_1 |0\rangle + r_2 \operatorname{cis}\theta_2 |1\rangle \\ &= \operatorname{cis}\theta_1 \left(r_1 |0\rangle + r_2 \operatorname{cis}(\theta_2 - \theta_1)\right) \end{aligned}$$

  We say $\theta_2 - \theta_1$ is the relative phase and θ_1 is the global phase of $|\psi\rangle$

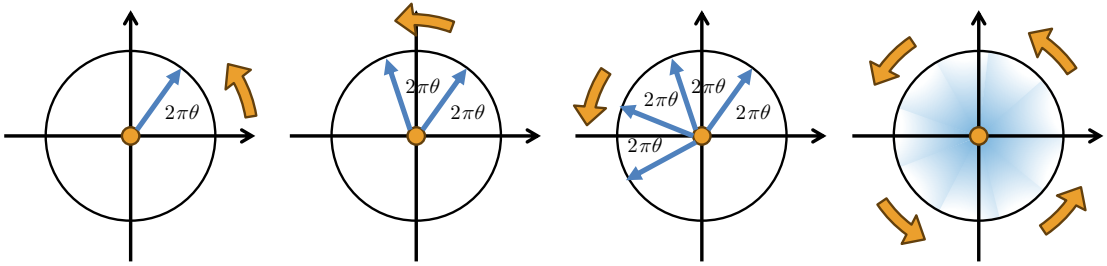
- If a unitary operator $\mathbf{U}$ has the dominant eigenvector $|\psi\rangle$, then

$$\mathbf{U}|\psi\rangle = \operatorname{cis}(2\pi\theta)|\psi\rangle$$

  where $\operatorname{cis}(2\pi\theta)$ is the eigenvalue of $|\psi\rangle$, implying that $2\pi\theta$ is the global phase of $\mathbf{U}$

- Challenge: How can we estimate the phase of any given unitary operator via quantum algorithms?
- Intuition:
  1. We treat a unitary operator $\mathbf{U}$ as a rotary movement of the clock, just like the clock's ticking
  2. This clock rotates with an unknown phase $2\pi\theta$ and we want to approximate it
  3. We let the clock rotate for adequately many times and observe its rotation
  4. With accumulated uniform rotations, a continuous angular gradient emerges
  5. We then look for the interference pattern and extract the phase from its frequency

Intuition of Quantum Phase Estimation



1. Treat a unitary operator as one clock's tick with unknown angle $2\pi\theta$

2. Rotate the clock with this constant angle

3. Keep rotating the clock for many times

4. Observe the interference pattern and extract its phase

Figure 14: Intuition for quantum phase estimation

Phase Estimation via Uniform Angular Rotations

- We imitate k angular rotations of $\mathbf{U}$ with a gate $\mathbf{CU}^k$ controlled by the k-th qubit called an estimation qubit:

$$|0_k x\rangle \mapsto |0_k x\rangle \quad |1_k x\rangle \mapsto |1_k\rangle \otimes \left(\mathbf{U}^k |x\rangle\right)$$

- Since $\mathbf{U}$ can be treated as an angular rotation of phase θ, we also obtain that

$$\mathbf{U}^k |x\rangle = \operatorname{cis}(2\pi\theta k)|x\rangle$$

- We generate adequate angular rotations by a series of exponentials

$$\sum_{k=0}^{N-1} 2^k = \frac{2^N - 1}{2-1} = 2^N - 1$$

- Let N be the granularity degree

$$\begin{aligned}
|y\rangle &= \frac{1}{2^{N/2}} \sum_{k=0}^{2^N-1} \mathbf{CU}^k \left[|\mathbb{B}_N(k)\rangle \otimes |\psi\rangle\right] \\
&= \prod_{k=0}^{N-1} \mathbf{CU}^k \left[|+\rangle^{\otimes N} \otimes |\psi\rangle\right] \\
&= |+\rangle^{\otimes N} \otimes \operatorname{cis}\left(\sum_{k=0}^{2^N-1} 2\pi\theta k\right)|\psi\rangle \\
&= \operatorname{cis}\left(\sum_{k=0}^{2^N-1} 2\pi\theta k\right)|+\rangle^{\otimes N} \otimes |\psi\rangle \\
&= \frac{1}{2^{N/2}} \left(\sum_{k=0}^{2^N-1} \operatorname{cis}(2\pi\theta k)|\mathbb{B}_N(k)\rangle\right) \otimes |\psi\rangle
\end{aligned}$$

Decoding the Phase with Inverse Fourier Transform

- Quantum Fourier transform $\mathbf{QFT}$ for the basis state $\mathbb{B}_N(k)$ is

$$\begin{aligned}&\mathbf{QFT}\,|\mathbb{B}_N(k)\rangle\\ =\;&\frac{1}{2^{N/2}}\sum_{j=0}^{2^N-1}\operatorname{cis}\left(\frac{2\pi jk}{2^N}\right)|\mathbb{B}_N(j)\rangle\end{aligned}$$

- Inverse quantum Fourier transform $\mathbf{QFT}^*$ for the basis state $\mathbb{B}_N(k)$ is

$$\begin{aligned}&\mathbf{QFT}^*\,|\mathbb{B}_N(k)\rangle\\ =\;&\frac{1}{2^{N/2}}\sum_{j=0}^{2^N-1}\operatorname{cis}\left(-\frac{2\pi jk}{2^N}\right)|\mathbb{B}_N(j)\rangle\end{aligned}$$

- We can decode the phase out of $|y_{1:N}\rangle$ by

$$\begin{aligned}|\Psi\rangle \;&=\; \mathbf{QFT}^*\,|y_{1:N}\rangle\\ =\;&\frac{1}{2^{N/2}}\sum_{j=0}^{2^N-1}\sum_{k=0}^{2^N-1}\operatorname{cis}\left(-\frac{2\pi jk}{2^N}\right)\\ &\qquad\times\frac{1}{2^{N/2}}\left(\operatorname{cis}(2\pi\theta k)\,|\mathbb{B}_N(k)\rangle\right)\\ =\;&\frac{1}{2^N}\sum_{j=0}^{2^N-1}\left[\sum_{k=0}^{2^N-1}\operatorname{cis}\frac{2\pi k\left(2^N\theta-j\right)}{2^N}\,|\mathbb{B}_N(k)\rangle\right]\\ =\;&\frac{1}{2^N}\sum_{k=0}^{2^N-1}\left[\sum_{j=0}^{2^N-1}q_{j,k}\right]|\mathbb{B}_N(k)\rangle\end{aligned}$$

where spectrum $q_{j,k}=\operatorname{cis}\frac{2\pi k\left(\left\lfloor 2^N\theta\right\rfloor+2^N\delta-j\right)}{2^N}$

Probability Distribution for Phase Extraction

- Probability for each phase $\mathbb{B}_N(k)$ is

$$P(k) \quad = \quad \frac{1}{2^N} \left| \sum_{j=0}^{2^N-1} q_{j,k} \right|^2$$

- At $k = \lfloor 2^N \theta \rceil$, we extract phase $\theta = k/2^N$;

$$\begin{aligned} P(\lfloor 2^N \theta \rceil) \quad &= \quad \frac{1}{2^N} \left| \sum_{j=0}^{2^N-1} \mathrm{cis}(2\pi k \delta) \right|^2 \\ &= \quad \frac{1}{2^N} \left| \sum_{j=0}^{2^N-1} \exp(2\pi i k \delta) \right|^2 \\ &= \quad \frac{1}{2^N} \left| \frac{1 - \exp(2^N 2\pi i k \delta)}{1 - \exp(2\pi i k \delta)} \right|^2 \quad \geq \frac{4}{\pi^2} \end{aligned}$$

- Summary: We have unitary operator $\mathbf{U}$
  1. Choose the granularity degree N and a random state $|\psi\rangle$
  2. Create a controlled gate $\mathbf{CU}^k$,

$$|0_k x\rangle \mapsto |0_k x\rangle \quad |1_k x\rangle \mapsto |1_k\rangle \otimes \left(\mathbf{U}^k |x\rangle\right)$$

  3. Accumulate angular rotations

$$|y\rangle \quad = \quad \prod_{j=0}^{N-1} \mathbf{CU}^k \left[\left(\mathbf{H}|0\rangle\right)^{\otimes N} \otimes |\psi\rangle \right]$$

  4. Compute the spectrum for each phase

$$|\Psi\rangle \quad = \quad \mathbf{QFT}^* |y_{1:N}\rangle$$

  5. Extract the phase $\theta = k^*/2^N$, where optimal $k^* = \arg\max_k P(\mathbb{B}_N(k)|\Psi)$

Implementation of Quantum Phase Estimation Algorithm

- QPE mapping can be simplified as: $\mathbf{QPE}\left(|0\rangle^{\otimes N} \otimes |\psi\rangle\right) = |\mathbb{B}_N(k^*)\rangle \otimes |\psi\rangle$

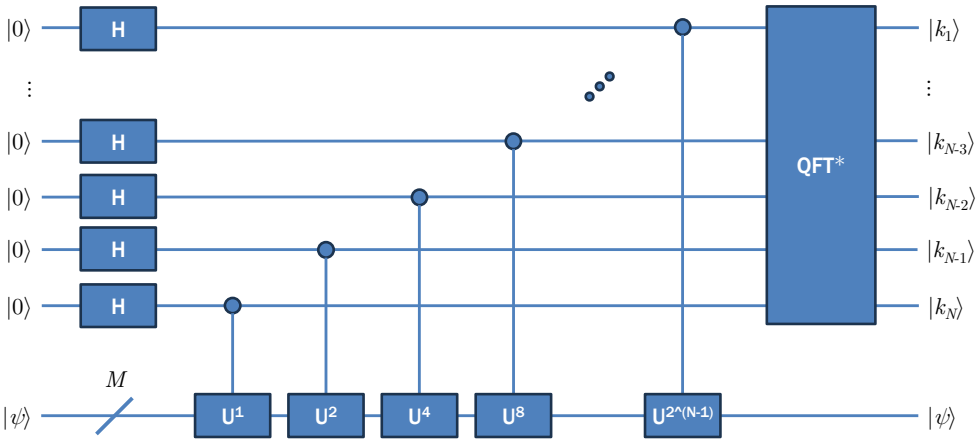


Figure 15: Quantum phase estimation. Time complexity is $O(2^N)$, where N is the granularity degree.

Quantum Phase Estimation in PennyLane

- PennyLane: `qml.QuantumPhaseEstimation(opr, target_wires=..., estimation_wires=...)`

```
# opr = unitary operator U
# target_wires = |\psi>
# estimation_wires = phase estimation wires
def qpe(opr, target_wires, estimation_wires):
    # Prepare the initial state on the target wires
    prep_state(target_wires)

    # Prepare the superposition on the estimation wires
    for est in estimation_wires:
        qml.Hadamard(wires=est)

    # Apply angular rotations
    qml.ControlledSequence(opr(target_wires), control=estimation_wires)

    # Decode the phase by inverse Fourier transform
    qml.adjoint(qml.QFT)(wires=estimation_wires)

    # Return the probability distribution of phases
    return qml.probs(wires=estimation_wires)
```

Practical Strategies for Quantum Phase Estimation

- Challenge: $O(2^N)$ multiplications of $\mathbf{U}$ are required

Strategy	Description	Purpose
Precompute $\mathbf{U}^{2^k}$	If $\mathbf{U}$ is small, we can compute each exponential $\mathbf{U}^{2^k}$ and include them in the QPE circuit	For small circuits and simulation
Transform $\mathbf{U}$ into $\exp(-i\hat{\mathbf{H}})$	Diagonalize $\mathbf{U} = \mathbf{V}\mathbf{D}\mathbf{V}^*$. Then, compute $\hat{\mathbf{H}} = \mathbf{V}(\log \mathbf{D})\mathbf{V}^*$. We obtain $\mathbf{U}^{2^k} = \exp(-i2^k\hat{\mathbf{H}})$.	When $\mathbf{U}$ consists of simple gates
Approximate $\mathbf{U}^{2^k}$	Break $\mathbf{U}$ into manageable exponentials with Trotter-Suzuki decomposition (aka. trotterization)	For large and composite systems

Table 2: Practical solutions to quantum phase estimation

Exercise 6.3: Quantum Phase Estimation

Answer the following questions.

1. What differ the relative phase and the global phase? Explain briefly.
2. Consider the Power Method:

$$\mathbf{U}\left(\mathbf{U}^N|x\rangle\right) \approx \lambda\left(\mathbf{U}^N|x\rangle\right)$$

where $\mathbf{U}$ is a unitary matrix, and N is large enough. $(\mathbf{U}^N|x\rangle, \lambda)$ is the dominant eigen-pair of $\mathbf{U}$. Explain why applying $\mathbf{U}$ for adequately many iterations will result in its dominant eigenvector extracted.

3. Explain why a quantum operator can be eigen-decomposed to its dominant eigen-pair using QPE.
4. Let N be the number of estimation qubits. Explain why the operator $\mathbf{U}$ is rotated for at most 2^N iterations.
5. Explain how a phase θ is extracted from multiple rotations using InvQFT.
6. Discuss why computing $\mathbf{CU}^{2^k}$ is expensive for some unitary operators $\mathbf{U}$.
7. Do the numbers of estimation and target qubits need to be equal to each other? Explain your reasons briefly.

Shor’s Algorithm

Cryptography

- Cryptography is the study and practice of techniques for secure communication in the presence of adversarial behaviors
- The earliest known cryptograph is Caesar ciphers, in which each character is shifted by the position specified in the secret keys

Encryption: $\text{HELLO} \Rightarrow_{e=+3} \text{KHOOR}$

Decryption: $\text{KHOOR} \Rightarrow_{d=-3} \text{HELLO}$

- Two major types of cryptography
  - Symmetric: encryption and decryption keys are identical; e.g. $(a \oplus e) \oplus e = a$
  - Asymmetric: encryption and decryption keys are different; e.g. $(a + e) + d = a$

- Modern cryptography is based on the exponential modulo of prime numbers

$$\text{encrypt}(a|b,p) \quad = \quad (a^b)\,\%p$$

where b is a prime number

- Although we know p, it is still computationally expensive to extract b

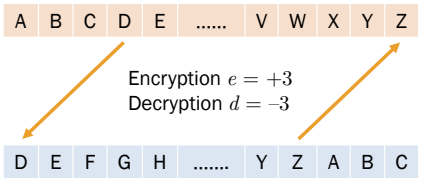


Figure 16: Caesar ciphers with 3 position shifts

Cryptosystem

- Cryptosystem: a suite of cryptographic algorithms needed to implement a particular security service
  - Key generation: The encryption and decryption keys must be generated in pair and then shared only between the sender and the receiver
  - Encryption: An intelligible text is transformed into an unintelligible one using the secret encryption key
  - Decryption: An unintelligible text is transformed into an intelligible one using the secret decryption key
  - Hashing: Data is transformed to a digital signature to evade data tampering via an irreversible one-to-one function

- Definition: A cryptosystem is defined as a quintuple (P, C, K, E, D), where
  - P is a set of plaintexts, a.k.a. plaintext space
  - C is a set of ciphertexts, a.k.a. ciphertext space
  - K is a set of keys, a.k.a. key space
  - $E = \{E_k | k \in K\}$ is a set of encryption functions: $P \mapsto C$
  - $D = \{D_k | k \in K\}$ is a set of decryption functions: $C \mapsto P$
- For any encryption key $e \in K$, there exists a decryption key $d \in K$ such that $D_d(E_e(w)) = w$, for all plaintexts $w \in P$

Exercise 6.4: Asymmetric Cryptography

Let's modify Caesar's ciphers with the exponential modulo of prime numbers.

$$\text{encrypt}(a|b,p) \quad = \quad (a^b)\,\%p$$

where a is the order of character (e.g. A = 0, B = 1, C = 2, etc.), b is the encryption exponent, and p is the modulo. Here, we let $b = 7$ and $p = 187$. Use the above formula to encrypt the following messages character by character.

1. AND
2. HELLO
3. MODIFY
4. COMPLEX

Let's also define a decryption method for our modified Caesar's ciphers by imitating the encryption function.

$$\text{decrypt}(c|d,p) \quad = \quad (c^d)\,\%p$$

where c is the encrypted character, d is the decryption exponent, and p is the modulo. Here, we let $d = 23$ and $p = 187$. Use the above formula to decrypt the following messages.

5. [85, 115, 133]
6. [93, 108, 171, 115]
7. [177, 115, 106, 171, 0]
8. [145, 115, 177, 93, 147, 171]

Classical Prime Factorization

- An integer N is said to be a prime number if N is divisible by only 1 and N
- Prime factorization: to find prime factors of an integer; e.g. $150 = 5 \times 5 \times 3 \times 2$
- "$p|n$" : p is a prime factor of n, e.g. $5|150$
- Prime factorization: $O(2^L L^2)$ time, where L is the length of bitstring n
  1. If $N = 1$, then N is a factor.
  2. Otherwise, pick a random prime a, where $1 < a < \sqrt{N}$.
  3. Compute the greatest common divisor $k = \gcd(N, a)$.
  4. If $k \neq 1$, then we have found a factor. Carry on with factorizing k and N/k recursively.
  5. Otherwise, go back to step 2.

- Euclid's method for finding the greatest common divisor of integers $a, b \geq 0$

$$\gcd(a,b) = \begin{cases} \gcd(b,a) & a < b \\ \gcd(b, a\%b) & b > 0 \\ a & \text{otherwise} \end{cases}$$

where $a\%b$ is the modulo of a by b

$$a\%b = a - b\left\lfloor \frac{a}{b} \right\rfloor$$

- Least common multiple of $a, b \geq 0$ is

$$\text{lcm}(a,b) = \frac{|ab|}{\gcd(a,b)}$$

Coprimality

- Example:

$$\begin{aligned}\gcd(525,252) &= \gcd(252,525\%252)\\ &= \gcd(252,21)\\ &= \gcd(21,252\%21)\\ &= \gcd(21,0)\\ &= 21\end{aligned}$$

$$\begin{aligned}\operatorname{lcm}(525,252) &= \frac{525\times 252}{\gcd(525,252)}\\ &= \frac{525\times 252}{21}\\ &= 6300\end{aligned}$$

- We say that a is coprime with b if

$$\gcd(a,b) = 1$$

- Example: 8 and 9 are coprime, because their GCD is 1 though they are non-prime
- Numbers a and b are coprime if and only if the numbers 2^a-1 and 2^b-1 are coprime

$$\gcd(n^a-1,n^b-1) = n^{\gcd(a,b)}-1$$

- Bézout's identity: For any coprime a,b, there exist integers x,y such that

$$ax+by = \gcd(a,b)$$

Extended Euclid's Method

- The greatest common divisor g of two integers $a, b > 0$ can be written as Bézout's identity:

$$ax + by \quad = \quad g$$

where $(g, x, y) = \text{egcd}(a, 1, 0 \mid b, 0, 1)$ and

$$\begin{aligned} & \text{egcd}(a, x_a, y_a \mid b, x_b, y_b) \\ = & \begin{cases} \text{egcd}(b, x_b, y_b \mid a\%b, & b \neq 0 \text{ and } q = \lfloor \frac{a}{b} \rfloor \\ \quad x_a - q x_b, y_a - q y_b) & \\ (a, x_a, y_a) & \text{otherwise} \end{cases} \end{aligned}$$

- Example:

$$\begin{aligned} & \text{egcd}(525, 1, 0 \mid 252, 0, 1) \\ = \; & \text{egcd}(252, 0, 1 \mid 21, \\ & \quad 1 - 2 \times 0, 0 - 2 \times 1) \\ = \; & \text{egcd}(252, 0, 1 \mid 21, 1, -2) \\ = \; & \text{egcd}(21, 1, -2 \mid 0, \\ & \quad 0 - 12 \times 1, 1 - 12 \times (-2)) \\ = \; & \text{egcd}(21, 1, -2 \mid 0, -12, 25) \\ = \; & (21, 1, -2) \quad \Rightarrow \quad (g, x, y) \end{aligned}$$

- Therefore: $525 \times 1 + 252 \times (-2) = 21$

Classical Prime Factorization in NumPy

```
import numpy as np
import numpy.random as rnd

def prepare_primes(n):
    flags = np.ones(n, dtype=bool)
    flags[0] = flags[1] = False
    for i in range(2, n):
        if flags[i]:
            for j in range(2 * i, n, i):
                flags[j] = False
    return np.arange(n)[flags]

def factorize(n, primes):
    if n == 1: return np.array([n])
    potential_factors = primes[primes <= np.sqrt(n)]
    for a in potential_factors:
        k = np.gcd(n, a)
        if k != 1:
            return np.hstack([ factorize(k, primes), factorize(n // k, primes) ])
    return np.array([n])

primes = prepare_primes(200)
print(factorize(152, primes))    # Expected result: [ 2  2  2 19]
```

Euler's Totient Function

- Euler's totient function $\phi(n)$ is the number of integers $1 \leq k \leq n$ that are coprime to n
- General case: If $n = p_1^{k_1} p_2^{k_2} p_3^{k_3} \ldots p_r^{k_r}$, where each $p_1, \ldots, p_r$ is a prime factor of n, then

$$\phi(n) = \prod_{j=1}^{r} p_j^{k_j - 1}(p_j - 1)$$

- Special case: If n is prime, then

$$\phi(n^k) = n^{k-1}(n-1)$$

- Distributive property: If two integers p, q are coprime, then

$$\phi(pq) = \phi(p) \times \phi(q)$$

- Example: $200 = 2^3 \times 5^2$

$$\begin{aligned} \phi(200) &= \phi(2^3 \times 5^2) \\ &= \phi(2^3) \times \phi(5^2) \\ &= \left[2^2 \times (2-1)\right] \times \left[5^1 \times (5-1)\right] \\ &= 80 \end{aligned}$$

- Intuition: $\phi(n)$ gives the order (size) of the multiplicative group of modulo n
  - Coprimes of 15 are $\{1, 2, 4, 7, 8, 11, 13, 14\}$
  - We then have $\phi(15) = 8$
  - Some numbers are not in this multiplicative group of modulo 15, such as 3, because $x\%15 = 3$ can be reduced to $x\%5 = 1$, which is in a group of modulo 5

Carmichael's Totient Function

- Carmichael's totient function $\lambda(n)$ is the smallest positive integer m such that for every integer a coprime with n,

$$(a^m)\,\%n \quad = \quad 1 \quad \text{always holds}$$

- Recursive definition

$$\lambda(n) \quad = \quad \begin{cases} \phi(n) & n \in \{1,2,4\} \\ & \text{or } n = p_{\text{odd}}^k \\ \frac{1}{2}\phi(n) & n = 2^r \text{ and } r \geq 3 \\ \text{lcm}(\lambda(p_1^{k_1}), \ldots, \lambda(p_r^{k_r})) & n = p_1^{k_1} \ldots p_r^{k_r} \end{cases}$$

where $p_1, \ldots, p_r$ are distinctive primes, prime $p_{\text{odd}} \geq 3$, and $\phi(n)$ is Euler's totient function

- Example: $200 = 2^3 \times 5^2$

$$\begin{aligned} & \lambda(200) \\ = \quad & \lambda(2^3 \times 5^2) \\ = \quad & \text{lcm}(\lambda(2^3), \lambda(5^2)) \\ = \quad & \text{lcm}(\frac{1}{2}\phi(2^3), \lambda(5^2)) \\ = \quad & \text{lcm}(\frac{1}{2} \times 4, \lambda(5^2)) \\ = \quad & \text{lcm}(2, \lambda(5^2)) \\ = \quad & \text{lcm}(2, \phi(5^2)) \\ = \quad & \text{lcm}(2, 20) \\ = \quad & \frac{2 \times 20}{\text{gcd}(2, 20)} \quad = \quad 20 \end{aligned}$$

Exercise 6.5: Prime Numbers and Totient Functions

Compute the GCD and LCM of the following coprime pairs using Euclid's method.

1. 4620 and 78
2. 53482 and 385
3. 148580 and 374
4. 3233230 and 4025

Convert the following coprime pairs into Bézout's identity using the extended Euclid's method.

5. 630 and 66
6. 65065 and 1729
7. 454597 and 2926
8. 8498776 and 7714

Compute Euler's totient function ϕ of the following numbers.

9. $\phi(3^3 \times 5^3 \times 7^4)$
10. $\phi(5^4 \times 6^3 \times 11^2)$
11. $\phi(6^3 \times 10^3 \times 21)$
12. $\phi(2^7 \times 11 \times 17)$

Compute Carmichael's totient function λ of the following numbers.

13. $\lambda(2^6)$
14. $\lambda(5^4)$
15. $\lambda(8^3 \times 7^2)$
16. $\lambda(6^3 \times 21^2)$

Fermat-Euler Theorem

- If n and a are coprime with each other, we obtain that

$$\left(a^{\phi(n)}\right) \% n \quad = \quad 1$$

  where $\phi(n)$ is the order (size) of the multiplicative group of modulo n
- Fermat's little theorem: If p is a prime, then for any integer a,

$$\left(a^{p-1}\right) \% p \quad = \quad 1$$

- Consequence: Suppose we have large coprimes n and a. Although a is kept secret, we may be able to reveal it only if we know n and can estimate $\phi(n)$

- Proof: We know the large coprime n.
  1. Suppose we can estimate $r \approx \phi(n)$.
  2. We guess the coprime a with n, which satisfies the Fermat-Euler theorem

$$\left(a^r - 1\right) \% n \quad = \quad 0$$

  3. If we assume r to be even, we can factorize the LHS into

$$\left[(a^{r/2}+1)(a^{r/2}-1)\right] \% n \quad = \quad 0$$

  4. That means either $a^{r/2}+1$ or $a^{r/2}-1$ is a factor of n. Therefore, a prime factor of n is either

$$\gcd(a^{r/2}+1, n) \quad \text{or} \quad \gcd(a^{r/2}-1, n)$$

Asymmetric Cryptography

- Asymmetric cryptography: cryptographic system that uses pairs of related keys
  1. Generate two large prime numbers p, q.
  2. Compute $n = pq$.
  3. Publish n as the public key (shared among everyone for encryption), and conceal p and q as private keys (kept secret for encryption and decryption).
  4. Encryption and decryption keys depend on the totient $\phi(n) = (p-1)(q-1)$
- Attack of asymmetric cryptography: Recovering the private keys or decrypting a message without legitimate access
- Prime numbers make it hard to attack due to mathematical difficulty

- Math difficulty: It is easy to muliply two large prime numbers $p \times q = n$, but it is intractable to factorize n back to p and q with time complexity $O(n(\log n)^2)$

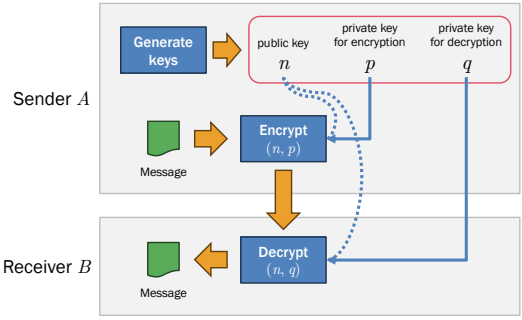


Figure 17: Asymmetric cryptography

Rivest-Shamir-Adleman Cryptosystem (RSA)

- RSA is an asymmetric cryptography system widely used in secure data transmission, e.g. Transport Layer Security (TLS)

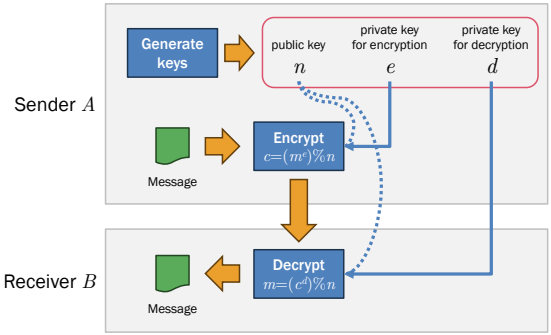


Figure 18: RSA Cryptosystem

- Encryption:
  1. We convert a string M into an integer m.
  2. We compute the encrypted message c:

$$c = (m^e)\,\%n$$

  3. We send m to the receiver.
- Decryption:
  1. We receive the encrypted message c.
  2. We compute the decrypted message m:

$$\begin{aligned} m &= (c^d)\,\%n \\ &= (m^{ed})\,\%n \end{aligned}$$

  3. We convert m back to a string M.

Key Generation of RSA Cryptosystem

1. Choose two large primes p and q, preferably between 2^{1023} and 2^{1024}. They are kept secret.
2. Compute modulus $n = pq$. Use n as a part of the public key.
3. Compute Carmichael's totient function $\lambda(n) = \text{lcm}(\lambda(p), \lambda(q)) = \text{lcm}(p-1, q-1)$. Keep $\lambda(n)$ secret.
4. Choose a coprime e with $\lambda(n)$. Use e as a part of the public key.
5. Compute d in the equation $ed \% \lambda(n) = 1$ by solving $\lambda(n)k + ed = 1$ via the extended Euclid's method. Use d as the private key.
6. Public key is (n, e), while private key is d.

- Demonstration:
  1. We choose two primes $p = 17$ and $q = 11$.
  2. We compute modulus $n = 17 \times 11 = 187$.
  3. We compute Carmichael's totient $\lambda(n) = \text{lcm}(17-1, 11-1) = 80$.
  4. We choose $e = 7$ coprime with $\lambda(n)$.
  5. We solve $\lambda(n) \cdot k + ed = 1$ for d by

$$\begin{aligned} & \text{egcd}(80, 1, 0 \mid 7, 1, 0) \\ = \ & \text{egcd}(7, 0, 1 \mid 3, 1, -11) \\ = \ & \text{egcd}(3, 1, -11 \mid 1, -2, 23) \\ = \ & \text{egcd}(1, -2, 23 \mid 0, 5, -80) \\ = \ & (1, -2, 23) \quad \Rightarrow \quad (g, k, d) \end{aligned}$$

  6. Public key is $(n = 187, e = 7)$, while private key is $d = 23$.

Proof of Correctness

1. The starting point is

$$\lambda(pq) \quad = \quad \mathrm{lcm}(p-1, q-1)$$

2. Since $ed \% \lambda(pq) = 1$, we obtain that $ed - 1 = \mathrm{lcm}(p-1, q-1)$. It means that $ed - 1$ is divisible by $p-1$ and $q-1$. I.e.

$$ed - 1 \quad = \quad h_1(p-1) \quad = \quad h_2(q-1)$$

   for some integers $h_1, h_2 > 0$.

3. Since $\left(m^{ed}\right) \% pq = m \% pq$, we can prove $\left(m^{ed}\right) \% p = m \% p$ and $\left(m^{ed}\right) \% q = m \% q$, separately. Both p and q parts are analogous.

4. The p part
   - Case 1 ($m \% p = 0$): We have that m is a multiple of p. Therefore, m^{ed} is also a multiple of p. Thus, $m^{ed} \% p = m \% p = 0$.
   - Case 2 ($m \% p \neq 0$): We have that $m^{ed} = m^{h_1(p-1)+1} = m \times m^{(p-1)h_1}$. By Fermat's little theorem, we have $m \times \left(m^{(p-1)}\right)^{h_1} \% p = m \times 1^{h_1} = m$.
5. The proof of the q part is analogous to that of the p part.
6. Therefore, we can conclude that

$$\left(m^{ed}\right) \% pq \quad = \quad m \% pq$$

Exercise 6.6: RSA Cryptosystem

1. Suppose we choose two primes $p = 23$ and $q = 17$. Generate the public key (n, e) and the private key d out of them.
2. Discuss why it is intractable to factorize a large number n directly into two multiplicants p and q, although we know that both p and q are large prime numbers.
3. Discuss why Fermat-Euler theorem makes it easier to factorize a large number, which is a multiplication of two large primes.
4. Explain why Fermat's little theorem plays a crucial role in the RSA cryptosystem. [Hint: Consult its proof of correctness.]
5. In practice, text encryption is not done in the character level, but rather in the chunk level. Implement an RSA cryptosystem in NumPy. The encryption and decryption are done in the chunk level of eight ASCII characters, where each text chunk is encoded by:

$$\text{encode}(c_0 c_1 c_2 \dots c_7) \quad = \quad \sum_{k=0}^{7} \left[2^k \times \texttt{ord}(c_k) \right]$$

The system should randomize p and q, and generate (n, e) and d accordingly.

Shor's Algorithm

- Shor's algorithm is a classical-quantum hybrid method for factorizing $n = pq$, where p and q are large prime factors

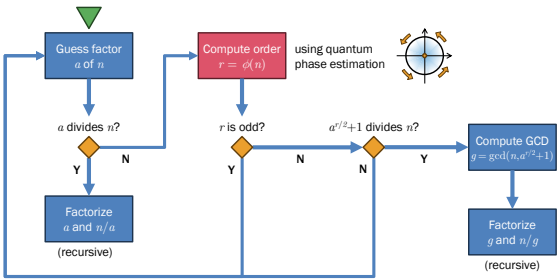


Figure 19: Shor's algorithm for factorizing n. Time complexity is $O(L^2 \log L)$, where L is the length of bitstring n.

- Hybrid algorithm: factorize n
  1. If $n = 1$, then n is a factor.
  2. Otherwise, pick a random prime a, where $1 < a < \sqrt{N}$.
  3. Compute the greatest common divisor $k = \gcd(n, a)$.
  4. If $k \neq 1$, then we have found a factor. Carry on with factorizing k and n/k recursively.
  5. Otherwise, compute the order $r \approx \phi(n)$ using quantum phase estimation.
  6. If r is odd, go back to Step 2.
  7. Otherwise, compute $g = \gcd(n, a^{r/2} + 1)$.
  8. If $g = 1$, go back to Step 2.
  9. Otherwise, we have found a factor. Carry on with factorizing g and n/g recursively.

Estimating the Order with Quantum Phase Estimation

- Motivation: Modulo n with the order r can be seen as a full rotation within nr steps
- We can define a unitary operator $\mathbf{U}_{\%n}(a)$ for modulo n as the following mapping

$$|\mathbb{B}_L(x)\rangle \quad \mapsto \quad |\mathbb{B}_L(a^x \% n)\rangle$$

  Recall that n is the public key and a is obtained from guessing
- Precise definition

$$\mathbf{U}_{\%n}(a) \quad = \quad \sum_{k=0}^{2^L-1} \left|\mathbb{B}_L(a^k \% n)\right\rangle \langle \mathbb{B}_L(k)|$$

- We apply quantum phase estimation on $\mathbf{U}_{\%n}(a)$ to approximate its phase θ, where

$$\theta \quad = \quad \mathrm{QPE}(\mathbf{U}_{\%n}(a))$$

- Since θ is actually a fraction $k^*/2^L$, where k^* is the optimal phase index, we still have to approximate the order r from

$$\frac{k^*}{2^L} \quad \approx \quad \frac{m}{r}$$

  where $0 < m < r$ are coprime
- The approximate of a coprime fraction is called the continued fraction method

Continued Fraction Method

- Continued fraction is a rational estimate of a real number

$$x \approx a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots}}}} = [a_0 | a_1, a_2, a_3, \ldots]$$

  where each of $a_0, a_1, a_2, a_3, \ldots$ is an integer

- Continued fraction method
  1. Start with index $i = 0$ and $q_0 = x$.
  2. Let $a_i = \lfloor q_i \rfloor$.
  3. If $q_i = a_i$ or i hits the limit, then stop.
  4. Otherwise, compute $q_{i+1} = 1/(q_i - a_i)$.
  5. Increase $i = i + 1$.
  6. Go to step 2.

- Example: Approximate a continued fraction of 1.23456 with the limit of 3

$$\begin{array}{lll} (i=0) & q_0 = 1.23456 & a_0 = 1 \\ (i=1) & q_1 = 4.26330 & a_1 = 4 \\ (i=2) & q_2 = 3.79793 & a_2 = 3 \\ (i=3) & q_3 = 1.25324 & a_3 = 1 \end{array}$$

- Here, we obtain three convergents

$$\begin{array}{lll} [1|4] & = 1 + 1/4 & = \frac{5}{4} \\ [1|4,3] & = 1 + 1/(4 + 1/3) & = \frac{16}{13} \\ [1|4,3,1] & = 1 + 1/(4 + 1/(3 + 1/1)) & = \frac{21}{17} \end{array}$$

Estimating the Order with Continued Fraction Method

- Previously, we obtained a fraction $k^*/2^L$ from the quantum phase estimation on $\mathbf{U}_{\%n}(a)$

$$\frac{k^*}{2^L} \quad = \quad \mathbf{QPE}(\mathbf{U}_{\%n}(a))$$

- We extract K convergents of $k^*/2^L$ via the continued fraction method

$$\frac{k^*}{2^L} \quad \approx \quad \frac{a_1}{b_1}, \frac{a_2}{b_2}, \frac{a_3}{b_3}, \cdots, \frac{a_K}{b_K}$$

- We may now be able to treat each b_k as a candidate of the order r

- In the original Shor's algorithm, we compute the least common multiple of $b_1, \ldots, b_K$ and use it as a candidate of the order instead

$$r \quad = \quad \mathrm{lcm}(b_1, b_2, b_3, \ldots, b_K)$$

- Using the LCM guarantees that we will have a large approximate $r \approx \phi(n)$
- In practice: we can do this automatically by converting $k^*/2^N$ into a fraction and then extracting its denominator: `fraction.Fraction(k, 2**N).denominator`

Shor's Algorithm in PennyLane/1

```
import pennylane as qml
from pennylane import numpy as np
from pennylane.numpy import random as rnd
from fractions import Fraction

no_targets, no_counts = 8, 8
no_qubits = no_targets + no_counts
wires = range(no_qubits)
counting_wires = range(no_counts)
target_wires = range(no_counts, no_qubits)

dev = qml.device('default.qubit', wires=wires)

def prepare_primes(n):
    flags = np.ones(n, dtype=bool)
    flags[0] = flags[1] = False
    for i in range(2, n):
        if flags[i]:
            for j in range(2 * i, n, i):
                flags[j] = False
    return np.arange(n)[flags]

primes = prepare_primes(2 ** no_targets)
```

Shor's Algorithm in PennyLane/2

```
def factorize_shor(n, primes):
    if n == 1:
        return np.array([n])

    potential_factors = primes[primes <= np.sqrt(n)]
    for a in potential_factors:

        # Classical factorization
        k = np.gcd(n, a)
        if k != 1:
            return np.hstack([ factorize_shor(k, primes), factorize_shor(n // k, primes) ])

        # Quantum speedup
        r = find_order(n, a)
        if r % 2 == 1: continue

        k = np.gcd(n, a ** (r // 2) - 1)
        if k != 1:
            return np.hstack([ factorize_shor(k, primes), factorize_shor(n // k, primes) ])

    return np.array([n])
```

Shor's Algorithm in PennyLane/3

```
def find_order(n, a):
    probs = order_finding_circuit(n, a)
    idx = int(np.argmax(probs))
    frac = Fraction(idx, 2 ** no_counts)
    r = frac.denominator
    return r

def modulo_matrix(a, power, n):
    dims = 2 ** no_targets
    mat = np.zeros((dims, dims), dtype=np.complex128)
    factor = np.mod(np.power(a, power), n)
    for i_input in range(dims):
        if i < n:
            i_output = (factor * i_input) % n
        else:
            i_output = i_input
        mat[i_output, i_input] = 1.0
    return mat

def apply_power_modulo(a, power, n, ctrl_wires, tgt_wires):
    mat = modulo_matrix(a, power, n)
    qml.ctrl(qml.QubitUnitary(mat, wires=tgt_wires), ctrl_wires)
```

Shor's Algorithm in PennyLane/4

```
@qml.qnode(dev)
def order_finding_circuit(n, a):
    init_state = np.zeros(2 ** no_qubits)
    init_state[0] = 1.0
    qml.StatePrep(init_state, wires=wires)

    for i in counting_wires:
        qml.Hadamard(wires=i)

    for k, ctrl in enumerate(counting_wires):
        power = 2 ** k
        apply_power_modulo(
            a, power, n,
            ctrl_wires=[ctrl], tgt_wires=target_wires
        )

    qml.adjoint(qml.QFT)(wires=counting_wires)

    return qml.probs(wires=counting_wires)

print(factorize_shor(152, primes))
# Expected result: [ 2  2  2 19]
```

Exercise 6.7: Shor's Algorithm

1. Identify the differences between the classical factorization method and Shor's Algorithm.
2. Discuss why knowing the order $r \approx \phi(n)$ helps speed up classical factorization.
3. Explain why the algorithm resorts to classical factorization when r is odd. Hint: Consult Fermat-Euler theorem.
4. Consider the unitary operator $\mathbf{U}_{\%n}(a)$, where

$$\mathbf{U}_{\%n}(a) \quad = \quad \sum_{k=0}^{2^L-1} \left| \mathbb{B}_L(a^k \% n) \right\rangle \left\langle \mathbb{B}_L(k) \right|$$

   Explain why applying this operator on $|+\rangle^{\otimes L}$ is equivalent to 2^L rotations.
5. Explain why the phase θ extracted with the quantum phase estimation algorithm has to be approximated to a fraction m/r, where m and r are coprime, instead of $k^*/2^L$.

Simon’s Algorithm

Simon's Problem

- In cryptography, hashing irreversibly converts an input string into a binary representation
- To avoid collision, hashing is defined as a one-to-one function $\{0,1\}^m \mapsto \{0,1\}^n$, where $m \leq n$
- A function f is NOT one-to-one, if and only if for all $\mathbf{x}, \mathbf{x}'$, there is a non-zero solution $\mathbf{s}$ such that $f(\mathbf{x}) = f(\mathbf{x}')$ and $\mathbf{x}' = \mathbf{x} \oplus \mathbf{s}$
- In classical computing, it is intractable to check the aforementioned condition on all possible pairs $\mathbf{x}, \mathbf{x}'$ due to the $O(2^{N/2})$ time complexity
- Quantum solution: We can solve this problem by testing all input strings $\mathbf{x}$ at the same time and observe the distribution of the outputs $f(\mathbf{x})$

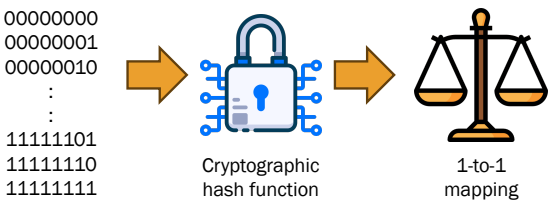


Figure 20: Simon's problem

Simon's Algorithm

- Daniel R. Simon (1994) proposed an efficient quantum algorithm for checking if a function f is one-to-one
- Let's force f to an extreme, where it generates an output hash code in the same length as that of the input
- Let's convert f into a quantum operator $\mathbf{U}_f$ that maps

$$|x_{1:L}\rangle \otimes |0\rangle^{\otimes L} \quad \mapsto \quad |x_{1:L}\rangle \otimes |f(x_{1:L})\rangle$$

- Basic idea: We enumerate all possible inputs $\mathbf{x}$ and solutions $\mathbf{s}$ of f, and measure the distribution of all output hash codes

- First, we enumerate all possible inputs k

$$\begin{aligned} |q_1\rangle &= \mathbf{U}_f \left[\left(\mathbf{H}^{\otimes L} |0\rangle^{\otimes L} \right) \otimes |0\rangle^{\otimes L} \right] \\ &= \frac{1}{2^{L/2}} \sum_{k=0}^{2^L-1} |\mathbb{B}_L(k)\rangle \otimes |f(\mathbb{B}_L(k))\rangle \end{aligned}$$

- We then enumerate all solutions s

$$\begin{aligned} |q_2\rangle &= \left[\mathbf{H}^{\otimes L} \otimes \mathbf{I}^{\otimes L} \right] |q_1\rangle \\ &= \frac{1}{2^{L/2}} \sum_{k=0}^{2^L-1} \left(\sum_{s=0}^{2^L-1} \frac{(-1)^{s \cdot k}}{2^{L/2}} |\mathbb{B}_L(s)\rangle \right) \\ &\qquad \otimes |f(\mathbb{B}_L(k))\rangle \\ &= \sum_{s=0}^{2^L-1} \sum_{k=0}^{2^L-1} \frac{(-1)^{s \cdot k}}{2^L} |\mathbb{B}_L(s)\rangle \otimes |f(\mathbb{B}_L(k))\rangle \end{aligned}$$

Simon's Algorithm (cont'd)

- By the phase kickback technique, we have

$$
\begin{aligned}
& |q_2\rangle \\
= & \sum_{s=0}^{2^L-1} \sum_{k=0}^{2^L-1} \frac{(-1)^{s \cdot k}}{2^L} |\mathbb{B}_L(s)\rangle \otimes |f(\mathbb{B}_L(k))\rangle \\
= & \sum_{s=0}^{2^L-1} \left[|\mathbb{B}_L(s)\rangle \otimes \sum_{k=0}^{2^L-1} \frac{(-1)^{s \cdot k}}{2^L} |f(\mathbb{B}_L(k))\rangle \right]
\end{aligned}
$$

- The probability of solution s is therefore

$$
P_{\text{oracle}}(s) = \left| \sum_{k=0}^{2^L-1} \frac{(-1)^{s \cdot k}}{2^L} |f(\mathbb{B}_L(k))\rangle \right|^2
$$

- Case 1: If f is one-to-one, the probability distribution of all solutions s is uniform

$$
P_{\text{oracle}}(s) = 1/2^L
$$

- Case 2: If f is many-to-one (i.e. non-zero oracle), the probability distribution of all solutions j is not uniform
  - If s is the solution, the range of $f(s)$ will be smaller than 2^L due to many-to-one mapping
  - We deduce mapping $X = \{x_1, \ldots, x_N\}$ to their common image y by

$$
P_{\text{oracle}}(s) = |X|/2^L
$$

Implementation of Simon's Algorithm

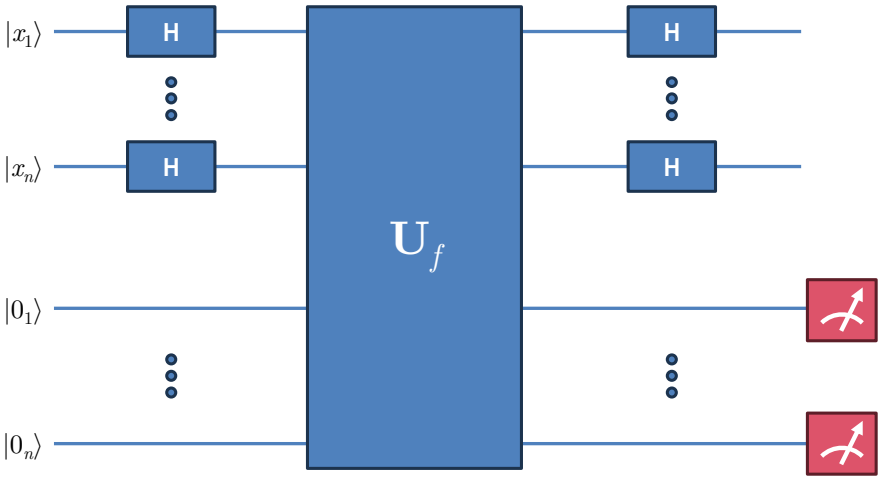


Figure 21: Simon's algorithm

Simon's Algorithm in PennyLane

```
no_qubits = 4
wires = range(2 * no_qubits)
input_wires, output_wires = wires[0 : no_qubits], wires[no_qubits : 2*no_qubits]
dev = qml.device('default.qubit', wires=wires)

def f_mto1(input_wires, output_wires):                  # Black box function (many to one)
    qml.CNOT(wires=[input_wires[3], output_wires[0]])   # Input:  |x0, x1, x2, x3>
    qml.CNOT(wires=[input_wires[1], output_wires[1]])   # Output: |x3, x1, x0 && x1, x1 && x2>
    qml.Toffoli(wires=[input_wires[0], input_wires[1], output_wires[2]])
    qml.Toffoli(wires=[input_wires[1], input_wires[2], output_wires[3]])

@qml.qnode(device=dev, shots=1000)
def simon_algo():                                       # Simon's algorithm
    for i in input_wires:
        qml.Hadamard(wires=i)
    f_mto1(input_wires, output_wires)
    for i in input_wires:
        qml.Hadamard(wires=i)
    return qml.counts(wires=output_wires)

simon_algo()
# Expected result: {'0000': 250, '0100': 62.5, '0101': 62.5, '0110': 62.5, '0111': 62.5,
#                   '1000': 250, '1100': 62.5, '1101': 62.5, '1110': 62.5, '1111': 62.5}
```

Finding the Solution

- Once we obtain the range of f, we discard $|0\rangle^{\otimes L}$ and $|1\rangle^{\otimes L}$ due to the lack of info
- We choose the most likely image (i.e. highest probability) and its linearly independent counterparts: $\mathbf{y}_1, \dots, \mathbf{y}_{L-1}$
- We solve for the solution $\mathbf{s} = [s_1 \dots s_L]^\top$ from a simple XOR equation system

$$\begin{aligned} y_{1,1}s_1 \oplus \dots \oplus y_{1,L}s_L &= 0 \\ &\vdots \\ y_{L-1,1}s_1 \oplus \dots \oplus y_{L-1,L}s_L &= 0 \end{aligned}$$

- If there is a non-zero solution, we select it as the solution $\mathbf{s}$

- Example: From the code previously demonstrated
  - We discard 0000 and 1111 due to no info
  - We then select 1000, 0111, and 0100
  - We solve for $\mathbf{s} = [s_1 \dots s_L]^\top$ from

$$\begin{aligned} (1 \cdot s_1) \oplus (0 \cdot s_2) \oplus (0 \cdot s_3) \oplus (0 \cdot s_4) &= 0 \\ (0 \cdot s_1) \oplus (1 \cdot s_2) \oplus (1 \cdot s_3) \oplus (1 \cdot s_4) &= 0 \\ (0 \cdot s_1) \oplus (1 \cdot s_2) \oplus (0 \cdot s_3) \oplus (0 \cdot s_4) &= 0 \end{aligned}$$

  - We then have $s_1 = 0$, $s_2 = 0$, and $s_3 = s_4$, obtaining $[0,0,0,0]^\top$ or $[0,0,1,1]^\top$
  - We obtain a non-zero solution $\mathbf{s} = [0,0,1,1]^\top$; this is our solution

Exercise 6.8: Simon's Algorithm

Implement the following hash functions using **CNOT**, **CCX**, and other quantum gates in PennyLane. Consult Chapter 4 for Boolean universality of the Toffoli gate.

1. $|x_1, x_2\rangle \mapsto |x_2, x_1\rangle$
2. $|x_1, x_2\rangle \mapsto |\overline{x}_1, \overline{x}_2\rangle$
3. $|x_1, x_2, x_3\rangle \mapsto |x_3, \overline{x}_1, x_1 \wedge x_2\rangle$
4. $|x_1, x_2, x_3\rangle \mapsto |x_1 \wedge x_2, x_1 \vee x_3, x_2 \wedge x_3\rangle$
5. $|x_1, x_2, x_3, x_4\rangle \mapsto |x_4, \overline{x}_3, x_2, \overline{x}_1\rangle$
6. $|x_1, x_2, x_3, x_4\rangle \mapsto$
   $|x_1 \oplus x_2, x_2 \oplus x_3, x_3 \oplus x_4, x_4 \oplus x_1\rangle$
7. $|x_1, x_2, x_3, x_4\rangle \mapsto$
   $|x_1 \vee x_2, x_1 \wedge x_3, x_2 \vee x_3, x_2 \wedge x_4\rangle$

Answer the following questions.

8. Discuss why the hash function of a cryptosystem must be 1-to-1. What issue will we encounter if it is many-to-1?
9. Run Simon's algorithm on the hash functions in Q.1-Q.7 and examine if they are one-to-one.
10. Find the solutions of the hash functions in Q.1-Q.7 with the XOR equation system.
11. Show that the function f is one-to-one if and only if its solution is $\mathbf{0}$ or $\mathbf{1}$.
12. Explain why it is intractable to examine all possible input pairs $\mathbf{x}, \mathbf{x}'$ for any function f, resulting in $O(2^{N/2})$ time complexity.

Conclusion

Conclusion

- Grover's search algorithm relies on sign-flipping (on the hidden solution) and amplitude amplification
- Quantum Fourier transform (QFT) converts a time series (encoded as a superposition of 2^L items) into the frequency domain
- Quantum phase estimation (QPE) extracts the global phase of a unitary operator by rotating it for many times and extracting its frequency
- Shor's algorithm improves the speed of classical prime factorization with QPE on the unitary operator $\mathbf{U}_{\%n}(a,k) = \sum_{k=0}^{2^L-1} \left| a^k \%n \right\rangle \langle k|$
- Simon's algorithm examines a hash function if it is one-to-one by enumerating all possible inputs, applying them on the function, and measuring the probability distribution of the outputs

Questions?

Target Learning Outcomes/1

- Grover's Search Algorithm
  1. Explain the limitation of classical unstructured search ($O(n)$) and how Grover's algorithm provides a quadratic speedup ($O(\sqrt{n})$).
  2. Describe the two core components of the Grover iteration: the oracle function for sign-flipping the solution and the diffusion operator for amplitude amplification.
  3. Calculate the optimal number of iterations to avoid overshooting the solution.
  4. Implement a quantum search routine in PennyLane using `qml.FlipSign` and `qml.templates.GroverOperator`.
- Quantum Fourier Transform (QFT)
  1. Differentiate between the classical Fast Fourier Transform and the Quantum Fourier Transform in terms of input representation and time complexity.
  2. Construct the QFT circuit using Fourier blocks composed of Hadamard gates and conditional dyadic rational phase gates.
  3. Apply the Inverse Quantum Fourier Transform to map frequency-domain vectors back to the time domain.

Target Learning Outcomes/2

- Quantum Phase Estimation (QPE)
  1. Define the concept of a global phase and explain how QPE extracts the phase of a unitary operator given an eigenvector.
  2. Illustrate the QPE workflow: creating a superposition on estimation qubits, applying controlled-power unitary gates, and decoding via InvQFT.
  3. Identify practical strategies for large-scale QPE, such as Trotterization or diagonalization of the operator.
- Shor's Algorithm
  1. Contrast classical prime factorization with the quantum-enhanced speedup.
  2. Describe the hybrid nature of Shor's algorithm, specifically how it uses QPE to solve the order-finding problem.
  3. Relate the mathematical foundations of the RSA cryptosystem (Fermat-Euler Theorem, Carmichael's function) to the vulnerability addressed by Shor's algorithm.

Target Learning Outcomes/3

- Simon's Algorithm
  1. Formulate Simon's problem as the task of determining if a hash function is one-to-one or many-to-one.
  2. Execute the quantum procedure to find the period by measuring output distributions and solving a system of XOR equations.
  3. Evaluate the security implications of hash collisions in cryptographic systems.