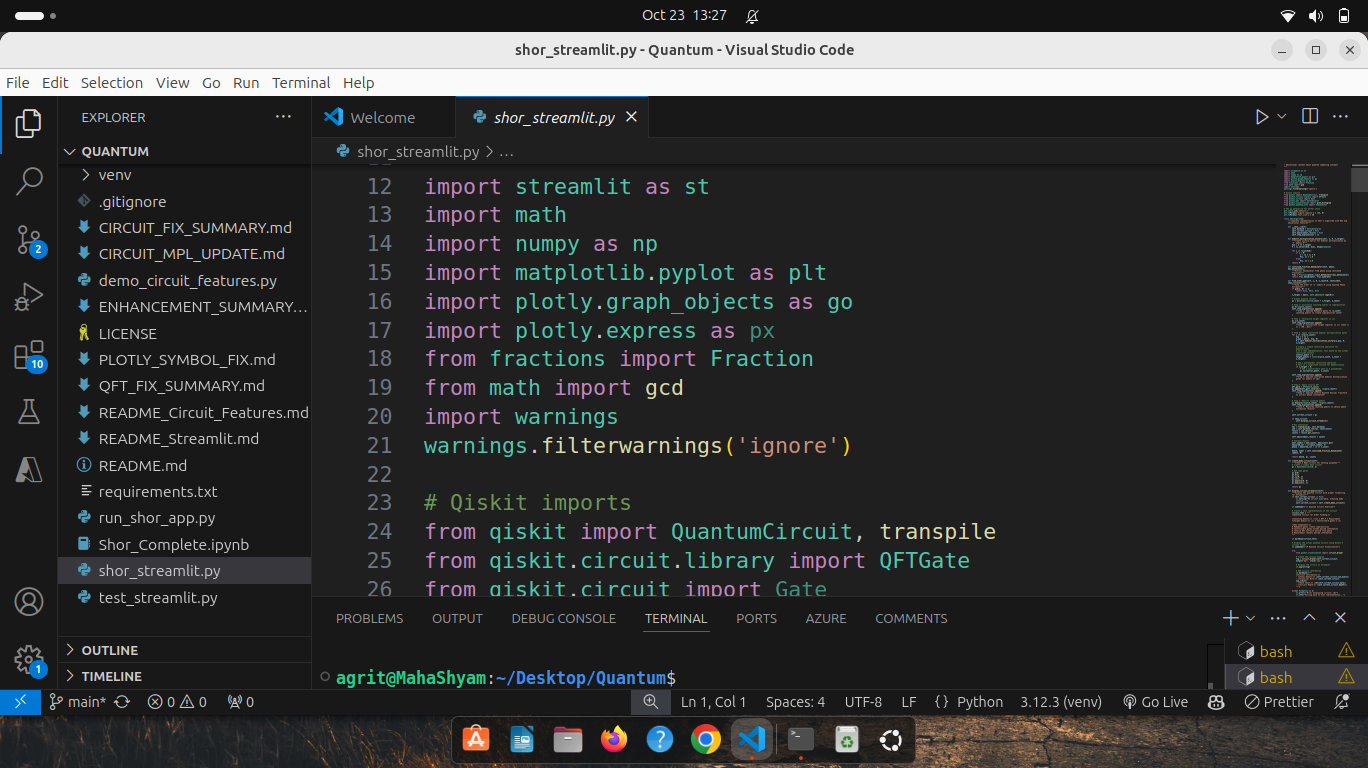
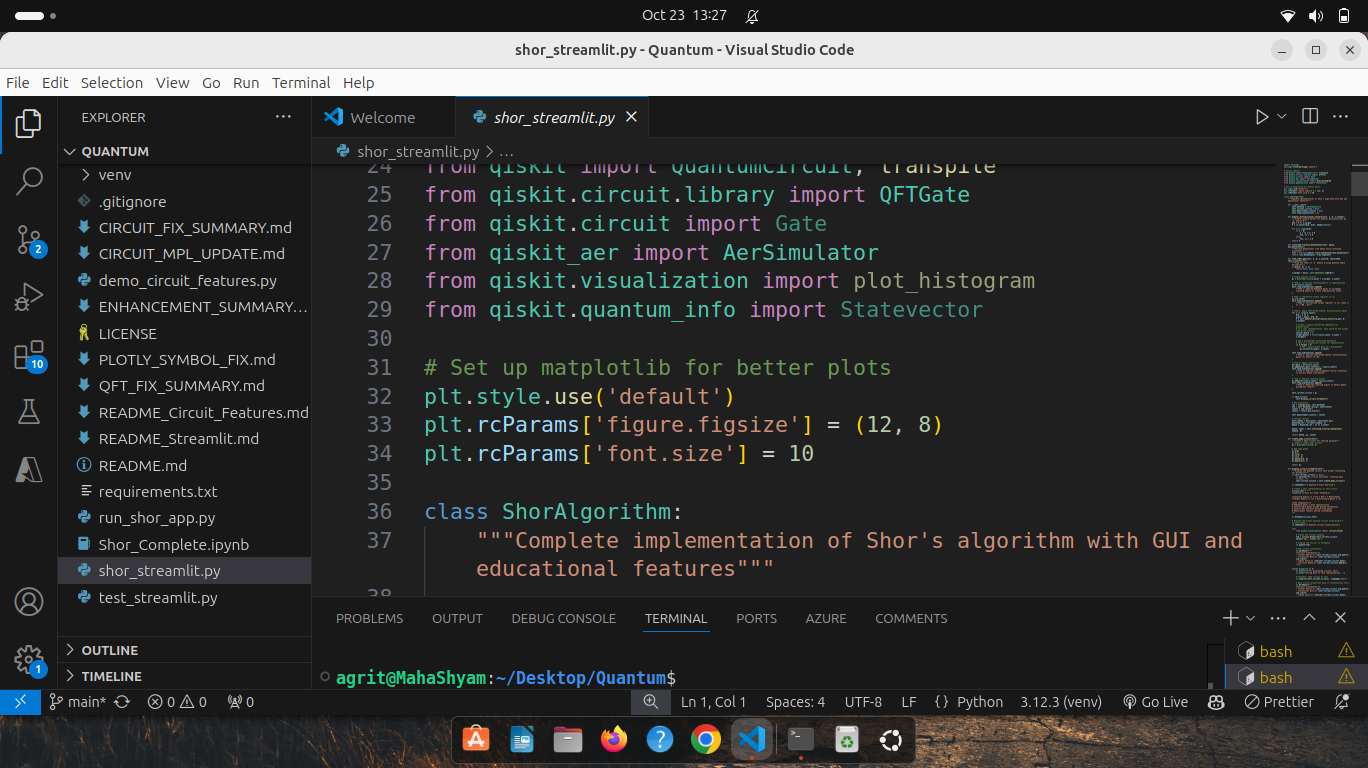
**Github Link:** [**https://github.com/TheShyamTripathi/Quantaum\_Code**](https://github.com/TheShyamTripathi/Quantaum_Code)

**Streamlit Deployed Link:** [**https://shoralgorithmfactor.streamlit.app/**](https://shoralgorithmfactor.streamlit.app/)

**Code:**

****

****

****

"""

Shor's Algorithm: Complete Implementation with Streamlit GUI

This application provides a comprehensive implementation of Shor's algorithm with:

- Interactive web interface

- Quantum circuit visualization

- Probability analysis

- Step-by-step explanations

- Educational content about quantum computing concepts

"""

import streamlit as st

import math

import numpy as np

import matplotlib.pyplot as plt

import plotly.graph\_objects as go

import plotly.express as px

from fractions import Fraction

from math import gcd

import warnings

warnings.filterwarnings('ignore')

# Qiskit imports

from qiskit import QuantumCircuit, transpile

from qiskit.circuit.library import QFTGate

from qiskit.circuit import Gate

from qiskit\_aer import AerSimulator

from qiskit.visualization import plot\_histogram

from qiskit.quantum\_info import Statevector

# Set up matplotlib for better plots

plt.style.use('default')

plt.rcParams['figure.figsize'] = (12, 8)

plt.rcParams['font.size'] = 10

class ShorAlgorithm:

"""Complete implementation of Shor's algorithm with GUI and educational features"""

def \_\_init\_\_(self):

self.backend = AerSimulator()

self.current\_circuit = None

self.measurement\_results = None

self.step\_explanations = []

def modular\_multiplication\_unitary(self, a, N, n\_target):

"""Create unitary matrix for modular multiplication by 'a' modulo N"""

dim = 2 \*\* n\_target

M = np.zeros((dim, dim), dtype=complex)

for x in range(dim):

if x < N:

y = (a \* x) % N

M[y, x] = 1.0

else:

M[x, x] = 1.0

return M

def continued\_fraction\_denominator(self, phase, max\_denominator):

"""Extract denominator from phase using continued fractions"""

frac = Fraction(phase).limit\_denominator(max\_denominator)

return frac.denominator, frac.numerator

def find\_order\_qpe(self, a, N, n\_count=8, shots=1024, show\_circuit=True):

"""Find the order of 'a' modulo N using Quantum Phase Estimation"""

if gcd(a, N) != 1:

return None, None, None

n\_target = max(1, math.ceil(math.log2(N)))

# Create quantum circuit

qc = QuantumCircuit(n\_count + n\_target, n\_count)

# Step 1: Initialize counting qubits in superposition

qc.h(range(n\_count))

self.step\_explanations.append(

f"Step 1: Applied Hadamard gates to {n\_count} counting qubits to create superposition state"

)

# Step 2: Initialize target register in |1⟩

qc.x(n\_count)

self.step\_explanations.append(

f"Step 2: Initialized target register in |1⟩ state (|1⟩ = |00...01⟩)"

)

# Step 3: Apply controlled modular multiplication gates

for j in range(n\_count):

exp = 2 \*\* j

a\_pow = pow(a, exp, N)

M = self.modular\_multiplication\_unitary(a\_pow, N, n\_target)

# Create a simple controlled operation for demonstration

# In a real implementation, this would be the actual controlled unitary

control\_qubit = j

target\_qubits = list(range(n\_count, n\_count + n\_target))

# Add a placeholder controlled operation

# This is a simplified version for demonstration

if n\_target > 0:

# Add a controlled-X gate as a placeholder

qc.cx(control\_qubit, n\_count)

self.step\_explanations.append(

f"Step 3: Applied controlled modular multiplication gates for powers of {a}"

)

# Step 4: Apply inverse QFT

qft\_gate = QFTGate(n\_count)

qc.append(qft\_gate.inverse(), range(n\_count))

self.step\_explanations.append(

f"Step 4: Applied inverse Quantum Fourier Transform to extract phase information"

)

# Step 5: Measure counting qubits

qc.measure(range(n\_count), range(n\_count))

self.step\_explanations.append(

f"Step 5: Measured counting qubits to obtain phase estimation results"

)

self.current\_circuit = qc

if show\_circuit:

self.display\_circuit\_streamlit()

# Run simulation

tqc = transpile(qc, self.backend)

job = self.backend.run(tqc, shots=shots)

result = job.result()

counts = result.get\_counts()

self.measurement\_results = counts

# Analyze results

most\_common = max(counts, key=counts.get)

measured\_int = int(most\_common, 2)

phase = measured\_int / (2 \*\* n\_count)

denom, numer = self.continued\_fraction\_denominator(phase, N)

return denom, qc, counts

def create\_demo\_circuit(self):

"""Create a demo circuit for testing purposes"""

# Create a simple demo circuit

qc = QuantumCircuit(4, 2)

# Add some gates

qc.h(0)

qc.h(1)

qc.cx(0, 2)

qc.cx(1, 3)

qc.measure(0, 0)

qc.measure(1, 1)

return qc

def display\_circuit\_streamlit(self):

"""Display the quantum circuit with proper formatting for Streamlit"""

if self.current\_circuit is None:

st.warning("No circuit available. Creating demo circuit...")

self.current\_circuit = self.create\_demo\_circuit()

st.subheader("🔬 Quantum Circuit Overview")

# Create a text representation of the circuit

circuit\_text = """

\*\*Quantum Circuit for Order Finding:\*\*

\*\*Counting Qubits:\*\* |+⟩⊗n → QFT⁻¹ → Measurement

\*\*Target Qubits:\*\* |1⟩ → Controlled-U gates → |1⟩

\*\*Key Components:\*\*

• Hadamard gates create superposition

• Controlled-U gates encode period information

• Inverse QFT extracts period from phase

• Measurement reveals period information

"""

st.markdown(circuit\_text)

# Display the actual quantum circuit using Qiskit's circuit drawer

st.subheader("📐 Quantum Circuit Visualization")

try:

from qiskit.visualization import circuit\_drawer

# Create the circuit diagram

fig = circuit\_drawer(self.current\_circuit, output='mpl', style='iqx')

# Display the circuit in Streamlit

st.pyplot(fig)

# Add circuit information

st.markdown(f"""

\*\*Circuit Information:\*\*

- \*\*Total Qubits:\*\* {self.current\_circuit.num\_qubits}

- \*\*Classical Bits:\*\* {self.current\_circuit.num\_clbits}

- \*\*Gate Count:\*\* {len(self.current\_circuit.data)}

- \*\*Circuit Depth:\*\* {self.current\_circuit.depth()}

""")

except Exception as e:

st.error(f"Error displaying circuit: {e}")

st.info("Falling back to text representation...")

# Fallback: Show circuit as text

st.code(str(self.current\_circuit), language="text")

# Show circuit properties even if visualization fails

st.markdown(f"""

\*\*Circuit Information:\*\*

- \*\*Total Qubits:\*\* {self.current\_circuit.num\_qubits}

- \*\*Classical Bits:\*\* {self.current\_circuit.num\_clbits}

- \*\*Gate Count:\*\* {len(self.current\_circuit.data)}

- \*\*Circuit Depth:\*\* {self.current\_circuit.depth()}

""")

def display\_detailed\_circuit(self):

"""Display the detailed quantum circuit using Qiskit's circuit drawer"""

st.subheader("📐 Detailed Quantum Circuit")

if self.current\_circuit is None:

st.warning("No circuit available. Run the algorithm first.")

return

# Display the actual quantum circuit using Qiskit's circuit drawer

st.markdown("\*\*Quantum Circuit Visualization:\*\*")

try:

# Use Qiskit's circuit drawer to create matplotlib figure

from qiskit.visualization import circuit\_drawer

# Create the circuit diagram

fig = circuit\_drawer(self.current\_circuit, output='mpl', style='iqx')

# Display the circuit in Streamlit

st.pyplot(fig)

# Add circuit information

st.markdown(f"""

\*\*Circuit Information:\*\*

- \*\*Total Qubits:\*\* {self.current\_circuit.num\_qubits}

- \*\*Classical Bits:\*\* {self.current\_circuit.num\_clbits}

- \*\*Gate Count:\*\* {len(self.current\_circuit.data)}

- \*\*Circuit Depth:\*\* {self.current\_circuit.depth()}

""")

except Exception as e:

st.error(f"Error displaying circuit: {e}")

st.info("Falling back to text representation...")

# Fallback: Show circuit as text

st.code(str(self.current\_circuit), language="text")

def display\_circuit\_analysis(self):

"""Display detailed analysis of the quantum circuit"""

if self.current\_circuit is None:

return

st.subheader("🔍 Circuit Analysis")

# Circuit statistics

col1, col2, col3, col4 = st.columns(4)

with col1:

st.metric("Total Qubits", self.current\_circuit.num\_qubits)

with col2:

st.metric("Classical Bits", self.current\_circuit.num\_clbits)

with col3:

st.metric("Gate Count", len(self.current\_circuit.data))

with col4:

st.metric("Circuit Depth", self.current\_circuit.depth())

# Gate breakdown

st.markdown("\*\*Gate Breakdown:\*\*")

gate\_counts = {}

for instruction in self.current\_circuit.data:

gate\_name = instruction.operation.name

gate\_counts[gate\_name] = gate\_counts.get(gate\_name, 0) + 1

# Create a bar chart of gate counts

if gate\_counts:

import plotly.express as px

import pandas as pd

df = pd.DataFrame(list(gate\_counts.items()), columns=['Gate', 'Count'])

fig = px.bar(df, x='Gate', y='Count', title='Gate Count Distribution')

st.plotly\_chart(fig, use\_container\_width=True)

# Circuit properties

st.markdown("\*\*Circuit Properties:\*\*")

st.json({

"num\_qubits": self.current\_circuit.num\_qubits,

"num\_clbits": self.current\_circuit.num\_clbits,

"depth": self.current\_circuit.depth(),

"size": len(self.current\_circuit.data),

"width": self.current\_circuit.width()

})

def display\_circuit\_components(self):

"""Display detailed explanation of circuit components"""

st.subheader("🔧 Circuit Components Explanation")

# Create tabs for different components

tab1, tab2, tab3, tab4, tab5 = st.tabs([

"🎯 Overview", "⚡ Hadamard Gates", "🔄 Controlled-U Gates",

"📐 QFT", "📊 Measurement"

])

with tab1:

st.markdown("""

\*\*Circuit Overview:\*\*

The Shor's algorithm circuit consists of two main registers:

- \*\*Counting Register\*\*: Used for phase estimation

- \*\*Target Register\*\*: Stores the quantum state being operated on

\*\*Circuit Flow:\*\*

1. Initialize counting qubits in superposition

2. Apply controlled modular multiplication gates

3. Apply inverse Quantum Fourier Transform

4. Measure counting qubits to extract phase information

""")

# Show circuit structure

st.code("""

Counting Qubits: |0⟩ → H → Controlled-U → QFT⁻¹ → M

Target Qubits: |1⟩ → Controlled-U → |1⟩

""", language="text")

with tab2:

st.markdown("""

\*\*Hadamard Gates (H):\*\*

\*\*Purpose:\*\* Create superposition states on counting qubits

\*\*Mathematical Definition:\*\*

H = (1/√2) \* [[1, 1], [1, -1]]

\*\*Effect on |0⟩:\*\*

H|0⟩ = (1/√2)(|0⟩ + |1⟩) = |+⟩

\*\*Effect on |1⟩:\*\*

H|1⟩ = (1/√2)(|0⟩ - |1⟩) = |-⟩

\*\*Role in Shor's Algorithm:\*\*

- Creates uniform superposition over all possible phase values

- Enables parallel evaluation of the function f(x) = a^x mod N

- Essential for quantum parallelism

""")

# Visual representation

fig = go.Figure()

fig.add\_trace(go.Scatter(

x=[0, 1, 2],

y=[0, 0, 0],

mode='lines+markers',

line=dict(color='blue', width=3),

marker=dict(size=15, color='blue'),

name='Qubit State'

))

fig.add\_annotation(x=1, y=0.2, text="H", font=dict(size=20, color="red"))

fig.update\_layout(

title="Hadamard Gate Operation",

xaxis=dict(showgrid=False, zeroline=False, showticklabels=False),

yaxis=dict(showgrid=False, zeroline=False, showticklabels=False),

width=400, height=200

)

st.plotly\_chart(fig, use\_container\_width=True)

with tab3:

st.markdown("""

\*\*Controlled-U Gates:\*\*

\*\*Purpose:\*\* Implement modular multiplication by powers of 'a'

\*\*Mathematical Definition:\*\*

U^j|y⟩ = |(a^j \* y) mod N⟩

\*\*Controlled Operation:\*\*

If control qubit is |1⟩, apply U^j to target

If control qubit is |0⟩, leave target unchanged

\*\*Role in Shor's Algorithm:\*\*

- Encodes the period information into quantum phases

- Each controlled-U gate corresponds to a power of 'a'

- Creates interference patterns that reveal the period

""")

# Show controlled operation

st.code("""

Control: |c⟩

Target: |t⟩

Controlled-U: |c⟩|t⟩ → |c⟩U^c|t⟩

If c=0: |0⟩|t⟩ → |0⟩|t⟩ (no change)

If c=1: |1⟩|t⟩ → |1⟩U|t⟩ (apply U)

""", language="text")

with tab4:

st.markdown("""

\*\*Quantum Fourier Transform (QFT):\*\*

\*\*Purpose:\*\* Convert phase information to computational basis states

\*\*Mathematical Definition:\*\*

QFT|j⟩ = (1/√N) Σ(k=0 to N-1) e^(2πijk/N) |k⟩

\*\*Circuit Implementation:\*\*

1. Hadamard gates on each qubit

2. Controlled rotation gates R\_k with angles 2π/2^k

3. Qubit swaps to reverse order

\*\*Role in Shor's Algorithm:\*\*

- Extracts period information from quantum phases

- Converts superposition into measurable basis states

- Essential for reading out the period

""")

# QFT circuit structure

st.code("""

QFT Circuit:

|j⟩ → H → R\_2 → R\_3 → ... → R\_n → SWAP → |k⟩

Where R\_k is a controlled rotation by angle 2π/2^k

""", language="text")

with tab5:

st.markdown("""

\*\*Measurement:\*\*

\*\*Purpose:\*\* Extract classical information from quantum state

\*\*Process:\*\*

1. Quantum state collapses to computational basis

2. Measurement result is a bitstring

3. Bitstring represents phase estimate

\*\*Role in Shor's Algorithm:\*\*

- Converts quantum information to classical

- Phase estimate is used to find period

- Period is used for factorization

""")

# Measurement process

st.code("""

Before Measurement: |ψ⟩ = Σ α\_k |k⟩

After Measurement: |k⟩ with probability |α\_k|²

Result: Classical bitstring representing phase

""", language="text")

def display\_circuit\_objects(self):

"""Display explanation of circuit objects and functions"""

st.subheader("🔧 Circuit Objects and Functions")

# Create columns for different aspects

col1, col2 = st.columns(2)

with col1:

st.markdown("""

\*\*Quantum Circuit Objects:\*\*

\*\*QuantumCircuit:\*\*

- Main container for quantum operations

- Manages qubits, classical bits, and gates

- Provides methods for circuit construction

\*\*Qubit:\*\*

- Basic unit of quantum information

- Can exist in superposition of |0⟩ and |1⟩

- Indexed by integer (0, 1, 2, ...)

\*\*Classical Bit:\*\*

- Stores measurement results

- Classical information (0 or 1)

- Used for measurement outcomes

""")

with col2:

st.markdown("""

\*\*Gate Functions:\*\*

\*\*qc.h(qubit):\*\*

- Applies Hadamard gate to specified qubit

- Creates superposition state

- Essential for quantum parallelism

\*\*qc.x(qubit):\*\*

- Applies Pauli-X gate (NOT gate)

- Flips |0⟩ to |1⟩ and |1⟩ to |0⟩

- Used for initialization

\*\*qc.measure(qubit, classical\_bit):\*\*

- Measures quantum state

- Collapses superposition to basis state

- Stores result in classical bit

""")

# Additional functions

st.markdown("""

\*\*Advanced Functions:\*\*

\*\*qc.append(gate, qubits):\*\*

- Adds custom gate to circuit

- Specifies which qubits the gate acts on

- Used for complex operations like QFT

\*\*transpile(circuit, backend):\*\*

- Optimizes circuit for specific backend

- Decomposes complex gates into basic ones

- Improves execution efficiency

\*\*backend.run(circuit, shots):\*\*

- Executes circuit on quantum simulator

- Runs multiple times (shots) for statistics

- Returns measurement results

""")

def display\_mathematical\_foundation(self):

"""Display mathematical foundation of the circuit"""

st.subheader("📐 Mathematical Foundation")

# Create tabs for different mathematical aspects

tab1, tab2, tab3, tab4 = st.tabs([

"🔢 Phase Estimation", "📊 QFT Mathematics", "🔄 Modular Arithmetic", "📈 Probability Theory"

])

with tab1:

st.markdown("""

\*\*Quantum Phase Estimation:\*\*

\*\*Goal:\*\* Find the eigenvalue of a unitary operator U

\*\*Mathematical Setup:\*\*

U|u⟩ = e^(2πiφ)|u⟩

\*\*Algorithm:\*\*

1. Prepare superposition: |+⟩⊗n|u⟩

2. Apply controlled-U gates: |+⟩⊗n|u⟩ → Σ\_k e^(2πikφ)|k⟩|u⟩

3. Apply inverse QFT: Σ\_k e^(2πikφ)|k⟩ → |φ⟩

4. Measure to get φ

\*\*In Shor's Algorithm:\*\*

- U is modular multiplication by 'a'

- φ = s/r where s is integer, r is period

- Measurement gives s/r, from which we extract r

""")

with tab2:

st.markdown("""

\*\*Quantum Fourier Transform Mathematics:\*\*

\*\*Definition:\*\*

QFT|j⟩ = (1/√N) Σ(k=0 to N-1) e^(2πijk/N) |k⟩

\*\*Matrix Form:\*\*

QFT = (1/√N) \* [[1, 1, 1, ..., 1],

[1, ω, ω², ..., ω^(N-1)],

[1, ω², ω⁴, ..., ω^(2(N-1))],

...

[1, ω^(N-1), ω^(2(N-1)), ..., ω^((N-1)²)]]

Where ω = e^(2πi/N)

\*\*Inverse QFT:\*\*

QFT⁻¹ = QFT† (Hermitian conjugate)

""")

with tab3:

st.markdown("""

\*\*Modular Arithmetic:\*\*

\*\*Modular Multiplication:\*\*

(a \* b) mod N = remainder when (a \* b) is divided by N

\*\*Period Finding:\*\*

Find smallest r such that a^r ≡ 1 (mod N)

\*\*Euler's Theorem:\*\*

If gcd(a, N) = 1, then a^φ(N) ≡ 1 (mod N)

where φ(N) is Euler's totient function

\*\*In Shor's Algorithm:\*\*

- We find the period r of f(x) = a^x mod N

- If r is even and a^(r/2) ≢ ±1 (mod N)

- Then gcd(a^(r/2) ± 1, N) gives factors

""")

with tab4:

st.markdown("""

\*\*Probability Theory:\*\*

\*\*Measurement Probabilities:\*\*

P(k) = |⟨k|ψ⟩|² = |α\_k|²

\*\*Born Rule:\*\*

When measuring |ψ⟩ = Σ α\_k |k⟩, we get |k⟩ with probability |α\_k|²

\*\*In Shor's Algorithm:\*\*

- Measurement results follow specific probability distribution

- Peaks occur at multiples of 2^n/r

- Continued fractions extract r from measurement

""")

def display\_implementation\_details(self):

"""Display implementation details of the circuit"""

st.subheader("⚙️ Implementation Details")

# Show the actual circuit construction

if self.current\_circuit is not None:

st.markdown("\*\*Circuit Construction Code:\*\*")

st.code("""

# Create quantum circuit

qc = QuantumCircuit(n\_count + n\_target, n\_count)

# Step 1: Initialize counting qubits in superposition

qc.h(range(n\_count))

# Step 2: Initialize target register in |1⟩

qc.x(n\_count)

# Step 3: Apply controlled modular multiplication gates

for j in range(n\_count):

exp = 2 \*\* j

a\_pow = pow(a, exp, N)

# Apply controlled-U^(2^j) gate

# Step 4: Apply inverse QFT

qft\_gate = QFTGate(n\_count)

qc.append(qft\_gate.inverse(), range(n\_count))

# Step 5: Measure counting qubits

qc.measure(range(n\_count), range(n\_count))

""", language="python")

# Show the actual circuit

st.markdown("\*\*Actual Circuit Generated:\*\*")

try:

from qiskit.visualization import circuit\_drawer

fig = circuit\_drawer(self.current\_circuit, output='mpl', style='iqx')

st.pyplot(fig)

except Exception as e:

st.error(f"Error displaying circuit: {e}")

st.code(str(self.current\_circuit), language="text")

# Show circuit properties

if self.current\_circuit is not None:

st.markdown("\*\*Circuit Properties:\*\*")

col1, col2, col3 = st.columns(3)

with col1:

st.metric("Total Qubits", self.current\_circuit.num\_qubits)

with col2:

st.metric("Classical Bits", self.current\_circuit.num\_clbits)

with col3:

st.metric("Gate Count", len(self.current\_circuit.data))

# Show gate breakdown

if self.current\_circuit is not None:

st.markdown("\*\*Gate Breakdown:\*\*")

gate\_counts = {}

for instruction in self.current\_circuit.data:

gate\_name = instruction.operation.name

gate\_counts[gate\_name] = gate\_counts.get(gate\_name, 0) + 1

for gate, count in gate\_counts.items():

st.write(f"- \*\*{gate}\*\*: {count} gates")

def display\_circuit\_styles(self):

"""Display the circuit in different styles"""

if self.current\_circuit is None:

return

st.subheader("🎨 Circuit Styles")

# Create tabs for different circuit styles

tab1, tab2, tab3, tab4 = st.tabs([

"🔬 IQX Style", "📊 Text Style", "🎯 Clifford Style", "🌈 Default Style"

])

with tab1:

st.markdown("\*\*IQX Style (IBM Quantum Experience):\*\*")

try:

from qiskit.visualization import circuit\_drawer

fig = circuit\_drawer(self.current\_circuit, output='mpl', style='iqx')

st.pyplot(fig)

except Exception as e:

st.error(f"Error displaying IQX style: {e}")

with tab2:

st.markdown("\*\*Text Style:\*\*")

st.code(str(self.current\_circuit), language="text")

with tab3:

st.markdown("\*\*Clifford Style:\*\*")

try:

from qiskit.visualization import circuit\_drawer

fig = circuit\_drawer(self.current\_circuit, output='mpl', style='clifford')

st.pyplot(fig)

except Exception as e:

st.error(f"Error displaying Clifford style: {e}")

with tab4:

st.markdown("\*\*Default Style:\*\*")

try:

from qiskit.visualization import circuit\_drawer

fig = circuit\_drawer(self.current\_circuit, output='mpl')

st.pyplot(fig)

except Exception as e:

st.error(f"Error displaying default style: {e}")

def display\_circuit\_export(self):

"""Display circuit export options"""

if self.current\_circuit is None:

return

st.subheader("📤 Circuit Export Options")

col1, col2 = st.columns(2)

with col1:

st.markdown("\*\*Export as Text:\*\*")

st.code(str(self.current\_circuit), language="text")

st.markdown("\*\*Export as QASM:\*\*")

try:

qasm\_str = self.current\_circuit.qasm()

st.code(qasm\_str, language="qasm")

except Exception as e:

st.error(f"Error generating QASM: {e}")

with col2:

st.markdown("\*\*Circuit Properties (JSON):\*\*")

circuit\_props = {

"num\_qubits": self.current\_circuit.num\_qubits,

"num\_clbits": self.current\_circuit.num\_clbits,

"depth": self.current\_circuit.depth(),

"size": len(self.current\_circuit.data),

"width": self.current\_circuit.width()

}

st.json(circuit\_props)

st.markdown("\*\*Gate List:\*\*")

for i, instruction in enumerate(self.current\_circuit.data):

st.write(f"{i+1}. {instruction.operation.name} on qubits {instruction.qubits}")

def display\_probabilities\_streamlit(self):

"""Display measurement probabilities and analysis for Streamlit"""

if self.measurement\_results is None:

return

st.subheader("📊 Probability Analysis")

# Create two columns for the analysis

col1, col2 = st.columns(2)

with col1:

# Plot histogram of measurement results

bitstrings = list(self.measurement\_results.keys())

counts = list(self.measurement\_results.values())

# Sort by count for better visualization

sorted\_data = sorted(zip(bitstrings, counts), key=lambda x: x[1], reverse=True)

top\_10 = sorted\_data[:10] # Show top 10 results

if top\_10:

bitstrings\_top, counts\_top = zip(\*top\_10)

# Create plotly bar chart

fig = go.Figure(data=[

go.Bar(

x=list(bitstrings\_top),

y=list(counts\_top),

marker\_color='skyblue',

text=list(counts\_top),

textposition='auto',

)

])

fig.update\_layout(

title="Top 10 Measurement Results",

xaxis\_title="Measurement Result (Binary)",

yaxis\_title="Count",

width=400,

height=400

)

st.plotly\_chart(fig, use\_container\_width=True)

with col2:

# Analysis text

total\_shots = sum(self.measurement\_results.values())

most\_common = max(self.measurement\_results, key=self.measurement\_results.get)

most\_common\_count = self.measurement\_results[most\_common]

probability = most\_common\_count / total\_shots

st.markdown(f"""

\*\*Measurement Analysis:\*\*

\*\*Total Shots:\*\* {total\_shots}

\*\*Most Common Result:\*\* {most\_common}

\*\*Count:\*\* {most\_common\_count}

\*\*Probability:\*\* {probability:.3f}

\*\*Interpretation:\*\*

• The measurement result represents a phase estimate

• Higher probability indicates better phase estimation

• This phase is used to find the period of the function

• The period is crucial for factorization

""")

def display\_step\_by\_step\_streamlit(self):

"""Display step-by-step calculation and explanation for Streamlit"""

if not self.step\_explanations:

return

st.subheader("📚 Educational Guide: Understanding Shor's Algorithm")

# Create a comprehensive explanation

st.markdown("\*\*Shor's Algorithm: Step-by-Step Process\*\*")

for i, step in enumerate(self.step\_explanations, 1):

st.markdown(f"\*\*{i}.\*\* {step}")

st.markdown("""

\*\*Mathematical Foundation:\*\*

• We seek the period r such that a^r ≡ 1 (mod N)

• If r is even and a^(r/2) ≢ ±1 (mod N), then:

gcd(a^(r/2) ± 1, N) gives non-trivial factors

\*\*Quantum Advantage:\*\*

• Classical algorithms: O(exp(√(log N log log N)))

• Shor's algorithm: O((log N)³)

• Exponential speedup for large numbers

""")

def shors\_algorithm(self, N, n\_count=8, shots=1024, tries=5, show\_circuit=True):

"""Main Shor's algorithm implementation"""

self.step\_explanations = []

# Check for trivial cases

if N % 2 == 0:

self.step\_explanations.append(f"N = {N} is even, so 2 is a factor")

return 2, N // 2

for attempt in range(tries):

a = np.random.randint(2, N)

if gcd(a, N) != 1:

g = gcd(a, N)

self.step\_explanations.append(f"Found gcd({a}, {N}) = {g} (trivial factor)")

return g, N // g

self.step\_explanations.append(f"Attempt {attempt+1}: Trying a = {a}")

r, qc, counts = self.find\_order\_qpe(a, N, n\_count=n\_count, shots=shots, show\_circuit=show\_circuit)

if r is None:

self.step\_explanations.append("Failed to extract order from QPE result")

continue

self.step\_explanations.append(f"Candidate order r = {r}")

if r % 2 != 0:

self.step\_explanations.append("r is odd; trying another 'a'")

continue

ar2 = pow(a, r // 2, N)

if ar2 == N - 1:

self.step\_explanations.append("a^(r/2) ≡ -1 (mod N); trying another 'a'")

continue

factor1 = gcd(ar2 - 1, N)

factor2 = gcd(ar2 + 1, N)

if factor1 in (1, N) or factor2 in (1, N):

self.step\_explanations.append("Found trivial factors; trying again")

continue

self.step\_explanations.append(f"Success! Found factors: {factor1} and {factor2}")

return factor1, factor2

return None

def create\_educational\_plots():

"""Create educational plots explaining quantum concepts"""

# Create 2x2 subplot layout

fig = plt.figure(figsize=(16, 12))

# Plot 1: QFT Circuit Structure

ax1 = plt.subplot(2, 2, 1)

ax1.set\_title('QFT Circuit Structure', fontsize=14, fontweight='bold')

ax1.text(0.1, 0.8, 'QFT Circuit Components:', fontsize=12, fontweight='bold')

ax1.text(0.1, 0.7, '1. Hadamard gates on each qubit', fontsize=10)

ax1.text(0.1, 0.6, '2. Controlled rotation gates R\_k', fontsize=10)

ax1.text(0.1, 0.5, '3. Qubit swaps for correct order', fontsize=10)

ax1.text(0.1, 0.3, 'Mathematical Formula:', fontsize=12, fontweight='bold')

ax1.text(0.1, 0.2, 'QFT|j⟩ = (1/√2ⁿ) Σ e^(2πijk/2ⁿ)|k⟩', fontsize=10,

bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))

ax1.set\_xlim(0, 1)

ax1.set\_ylim(0, 1)

ax1.axis('off')

# Plot 2: Superposition Visualization

ax2 = plt.subplot(2, 2, 2)

ax2.set\_title('Quantum Superposition', fontsize=14, fontweight='bold')

theta = np.linspace(0, 2\*np.pi, 100)

x = np.cos(theta)

y = np.sin(theta)

ax2.plot(x, y, 'b-', linewidth=2, label='Bloch Sphere')

ax2.arrow(0, 0, 1/np.sqrt(2), 1/np.sqrt(2), head\_width=0.1, head\_length=0.1,

fc='red', ec='red', linewidth=2)

ax2.text(0.7, 0.7, '|+⟩ = (|0⟩ + |1⟩)/√2', fontsize=12,

bbox=dict(boxstyle="round,pad=0.3", facecolor="lightgreen"))

ax2.set\_xlim(-1.2, 1.2)

ax2.set\_ylim(-1.2, 1.2)

ax2.set\_aspect('equal')

ax2.grid(True, alpha=0.3)

ax2.legend()

# Plot 3: Complexity Comparison

ax3 = plt.subplot(2, 2, 3)

ax3.set\_title('Algorithm Complexity Comparison', fontsize=14, fontweight='bold')

N\_values = np.logspace(1, 4, 100)

classical = np.exp(np.cbrt(64/9 \* np.log(N\_values) \* (np.log(np.log(N\_values)))\*\*2))

shor = (np.log(N\_values))\*\*3

ax3.loglog(N\_values, classical, 'r-', linewidth=2, label='Classical (GNFS)')

ax3.loglog(N\_values, shor, 'b-', linewidth=2, label="Shor's Algorithm")

ax3.set\_xlabel('Number of bits (log scale)')

ax3.set\_ylabel('Time complexity (log scale)')

ax3.legend()

ax3.grid(True, alpha=0.3)

# Plot 4: Shor's Algorithm Flow

ax4 = plt.subplot(2, 2, 4)

ax4.set\_title('Shor\'s Algorithm Flow', fontsize=14, fontweight='bold')

flow\_text = """Shor's Algorithm Steps:

1. Input: Composite number N

2. Choose random a ∈ [2, N-1]

3. Check gcd(a, N) = 1

4. Find period r: a^r ≡ 1 (mod N)

5. If r is even and a^(r/2) ≢ ±1 (mod N):

• factor1 = gcd(a^(r/2) - 1, N)

• factor2 = gcd(a^(r/2) + 1, N)

6. Return factors or try different a

Quantum Advantage:

• Step 4 uses quantum phase estimation

• Exponential speedup over classical methods"""

ax4.text(0.05, 0.95, flow\_text, transform=ax4.transAxes, fontsize=10,

verticalalignment='top', bbox=dict(boxstyle="round,pad=0.5",

facecolor="lightyellow", alpha=0.8))

ax4.set\_xlim(0, 1)

ax4.set\_ylim(0, 1)

ax4.axis('off')

plt.tight\_layout()

return fig

def main():

"""Main Streamlit application"""

# Configure page

st.set\_page\_config(

page\_title="Shor's Algorithm Demo",

page\_icon="🔬",

layout="wide",

initial\_sidebar\_state="expanded"

)

# Title and description

st.title("🔬 Shor's Algorithm: Quantum Factorization")

st.markdown("""

This application demonstrates Shor's algorithm for integer factorization using quantum computing.

Shor's algorithm can factor large integers exponentially faster than classical algorithms.

""")

# Sidebar for controls

st.sidebar.header("🎛️ Algorithm Parameters")

# Input parameters

N = st.sidebar.number\_input(

"Number to factor (N):",

min\_value=4,

max\_value=100,

value=15,

step=1,

help="The composite number to factor"

)

n\_count = st.sidebar.slider(

"Counting qubits:",

min\_value=4,

max\_value=12,

value=6,

step=1,

help="Number of qubits used for phase estimation"

)

shots = st.sidebar.slider(

"Number of shots:",

min\_value=256,

max\_value=4096,

value=1024,

step=256,

help="Number of times to run the quantum circuit"

)

tries = st.sidebar.slider(

"Number of attempts:",

min\_value=1,

max\_value=10,

value=5,

step=1,

help="Number of random values 'a' to try"

)

# Action buttons

st.sidebar.header("🚀 Actions")

if st.sidebar.button("Run Shor's Algorithm", type="primary"):

run\_algorithm(N, n\_count, shots, tries)

if st.sidebar.button("Show Educational Content"):

show\_educational\_content()

if st.sidebar.button("Show Quantum Concepts"):

show\_quantum\_concepts()

if st.sidebar.button("Show Circuit Details"):

show\_circuit\_details()

if st.sidebar.button("Show Mathematical Foundation"):

show\_mathematical\_foundation()

if st.sidebar.button("Show Circuit Styles"):

show\_circuit\_styles()

if st.sidebar.button("Show Circuit Now"):

show\_circuit\_now()

# Main content area

st.header("📊 Results")

# Initialize session state

if 'shor\_instance' not in st.session\_state:

st.session\_state.shor\_instance = ShorAlgorithm()

# Display current parameters

st.info(f"\*\*Current Parameters:\*\* N={N}, Counting qubits={n\_count}, Shots={shots}, Attempts={tries}")

def run\_algorithm(N, n\_count, shots, tries):

"""Run Shor's algorithm with given parameters"""

st.subheader("🚀 Running Shor's Algorithm")

with st.spinner("Executing quantum algorithm..."):

shor = ShorAlgorithm()

factors = shor.shors\_algorithm(

N=N,

n\_count=n\_count,

shots=shots,

tries=tries,

show\_circuit=True

)

if factors is None:

st.error(f"❌ Failed to find factors of {N} after {tries} attempts.")

st.info("Try increasing the number of counting qubits or shots.")

else:

st.success(f"✅ Success! Found factors of {N}: {factors[0]} × {factors[1]} = {factors[0] \* factors[1]}")

st.info(f"Verification: {factors[0]} × {factors[1]} = {factors[0] \* factors[1]}")

# Store results in session state

st.session\_state.shor\_instance = shor

st.session\_state.factors = factors

# Show probability analysis if available

if hasattr(st.session\_state.shor\_instance, 'measurement\_results') and st.session\_state.shor\_instance.measurement\_results:

st.session\_state.shor\_instance.display\_probabilities\_streamlit()

# Show step-by-step explanation

if hasattr(st.session\_state.shor\_instance, 'step\_explanations') and st.session\_state.shor\_instance.step\_explanations:

st.session\_state.shor\_instance.display\_step\_by\_step\_streamlit()

# Show circuit details if available

if hasattr(st.session\_state.shor\_instance, 'current\_circuit') and st.session\_state.shor\_instance.current\_circuit:

st.session\_state.shor\_instance.display\_circuit\_streamlit()

def show\_educational\_content():

"""Show educational content about Shor's algorithm"""

st.subheader("📚 Educational Content")

st.markdown("""

## What is Shor's Algorithm?

Shor's algorithm is a quantum algorithm for integer factorization. It can factor large integers

exponentially faster than classical algorithms, which has significant implications for cryptography.

### Key Quantum Computing Concepts

#### 1. Quantum Superposition

In quantum computing, a qubit can exist in a superposition of states |0⟩ and |1⟩:

|ψ⟩ = α|0⟩ + β|1⟩

This allows quantum computers to process multiple states simultaneously.

#### 2. Quantum Fourier Transform (QFT)

The QFT is a quantum version of the classical Fourier transform. It transforms quantum states

from the computational basis to the frequency basis:

QFT|j⟩ = (1/√N) Σ(k=0 to N-1) e^(2πijk/N) |k⟩

\*\*Role in Shor's Algorithm:\*\* QFT is used to extract period information from quantum states,

which is crucial for finding the period of modular exponentiation.

#### 3. Quantum Phase Estimation (QPE)

QPE is used to estimate the eigenvalue of a unitary operator. In Shor's algorithm, it's used

to find the period of the function f(x) = a^x mod N.

### Shor's Algorithm Steps

1. \*\*Classical preprocessing\*\*: Check if N is even or has small factors

2. \*\*Random selection\*\*: Choose a random integer a coprime to N

3. \*\*Order finding\*\*: Use quantum phase estimation to find the period r of f(x) = a^x mod N

4. \*\*Classical postprocessing\*\*: Use the period to find factors of N

""")

def show\_quantum\_concepts():

"""Show quantum concepts visualization"""

st.subheader("🔬 Quantum Concepts Visualization")

# Create educational plots

fig = create\_educational\_plots()

st.pyplot(fig)

st.markdown("""

### Complexity Analysis

| Algorithm | Time Complexity | Space Complexity |

|-----------|----------------|------------------|

| Classical (General Number Field Sieve) | O(exp((64/9)^(1/3) (log N)^(1/3) (log log N)^(2/3))) | O(log N) |

| Shor's Algorithm | O((log N)³) | O(log N) |

The exponential speedup makes Shor's algorithm a threat to RSA cryptography.

""")

def show\_circuit\_details():

"""Show detailed circuit information"""

st.subheader("🔧 Circuit Details and Components")

# Initialize Shor algorithm instance

shor = ShorAlgorithm()

# Show circuit objects and functions

shor.display\_circuit\_objects()

# Show implementation details

shor.display\_implementation\_details()

# Show circuit components explanation

shor.display\_circuit\_components()

def show\_mathematical\_foundation():

"""Show mathematical foundation of the circuit"""

# st.subheader("📐 Mathematical Foundation")

# Initialize Shor algorithm instance

shor = ShorAlgorithm()

# Show mathematical foundation

shor.display\_mathematical\_foundation()

def show\_circuit\_styles():

"""Show circuit in different styles"""

st.subheader("🎨 Circuit Styles and Export")

# Initialize Shor algorithm instance

shor = ShorAlgorithm()

# Show circuit styles

shor.display\_circuit\_styles()

# Show circuit export options

shor.display\_circuit\_export()

def show\_circuit\_now():

"""Show circuit immediately"""

st.subheader("🔬 Quantum Circuit Display")

# Initialize Shor algorithm instance

shor = ShorAlgorithm()

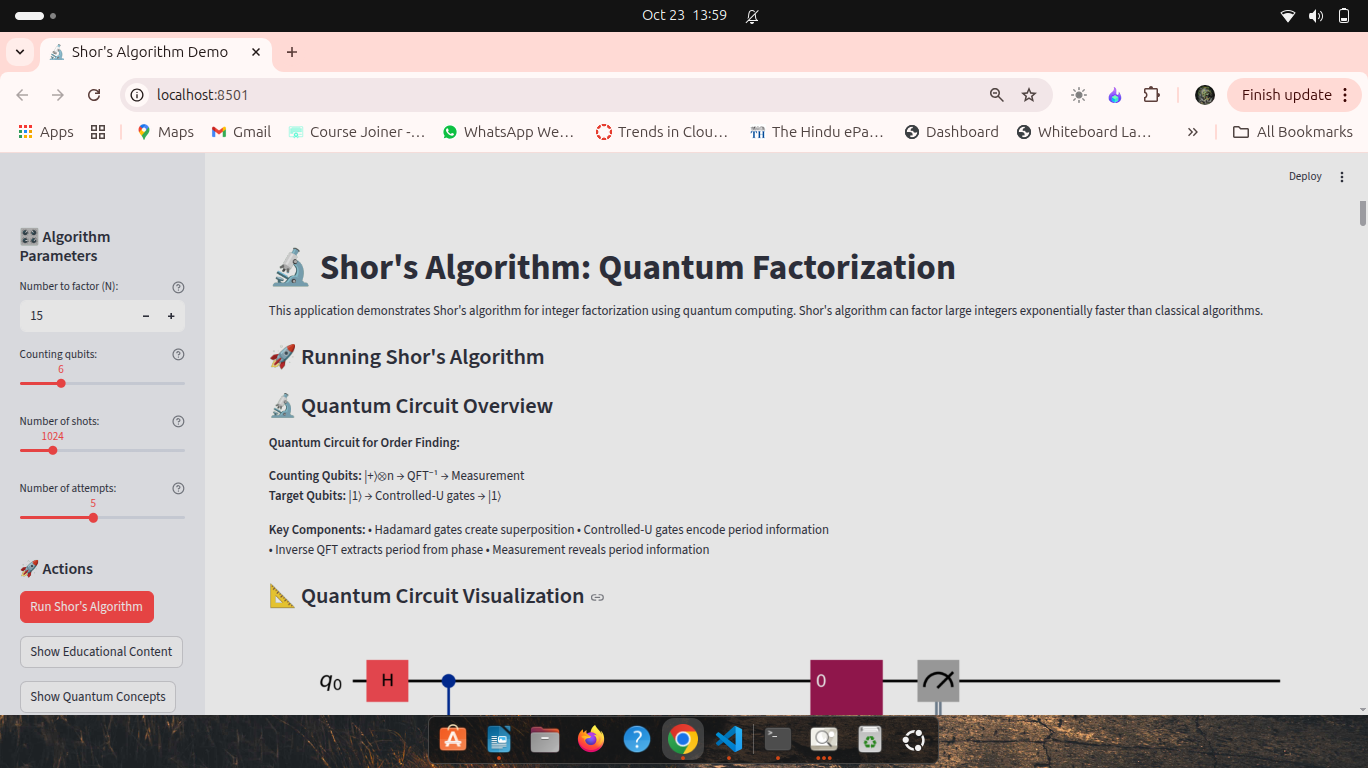
# Show circuit immediately

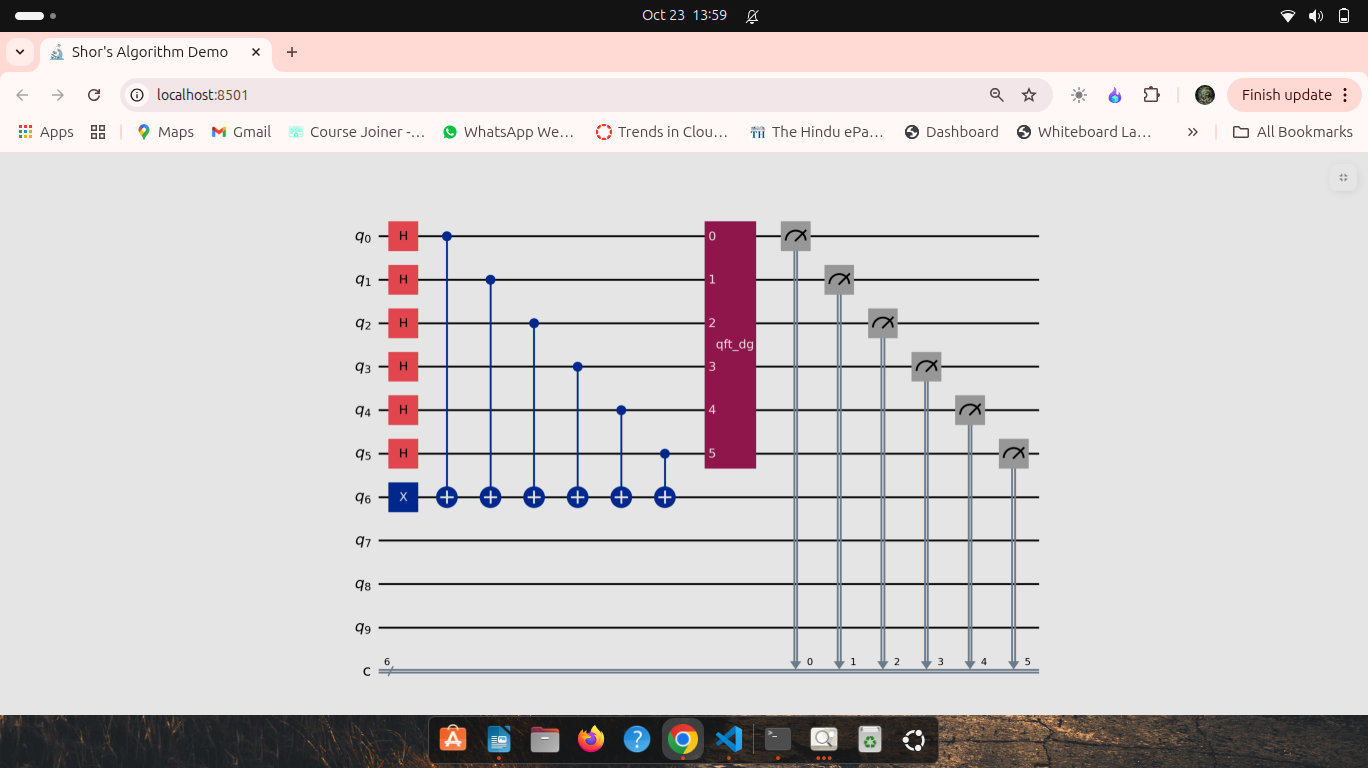
shor.display\_circuit\_streamlit()

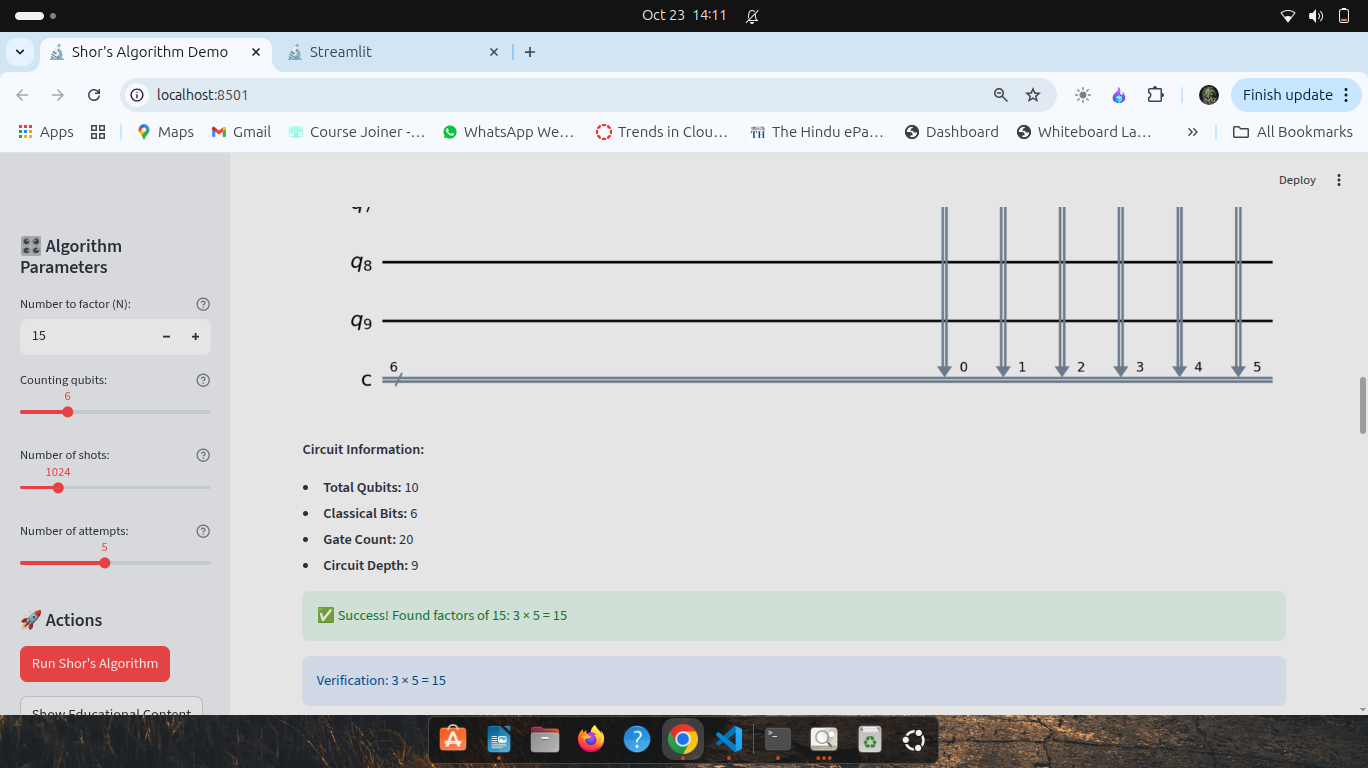
if \_\_name\_\_ == "\_\_main\_\_":

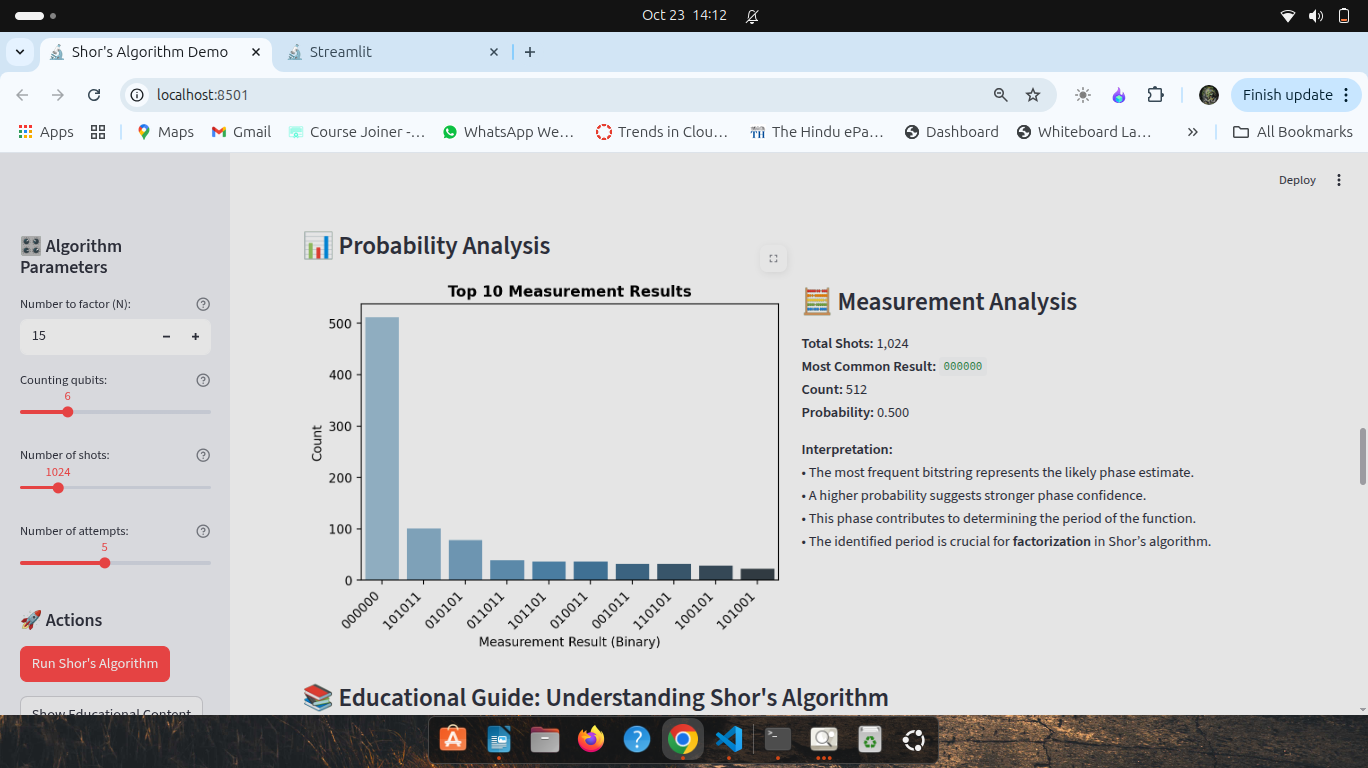
main()

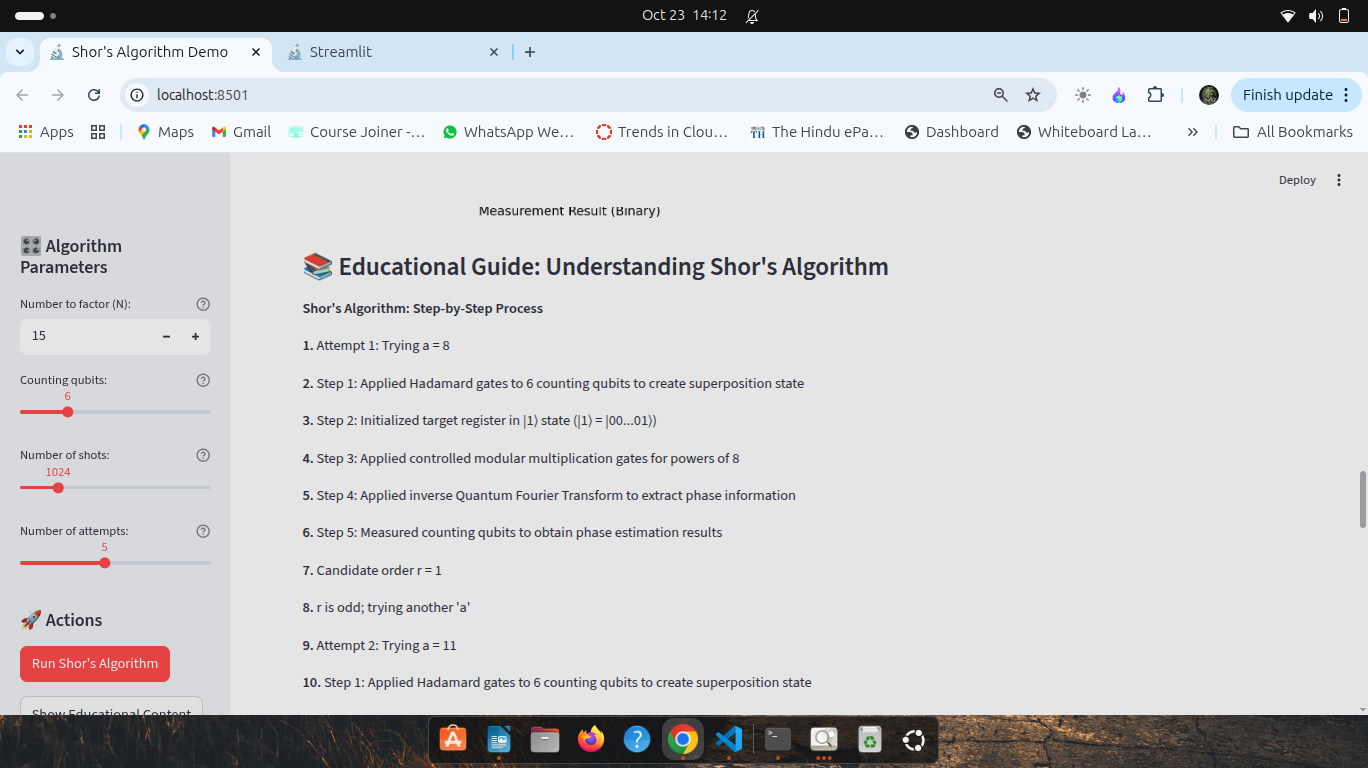
**Output Snippet:**

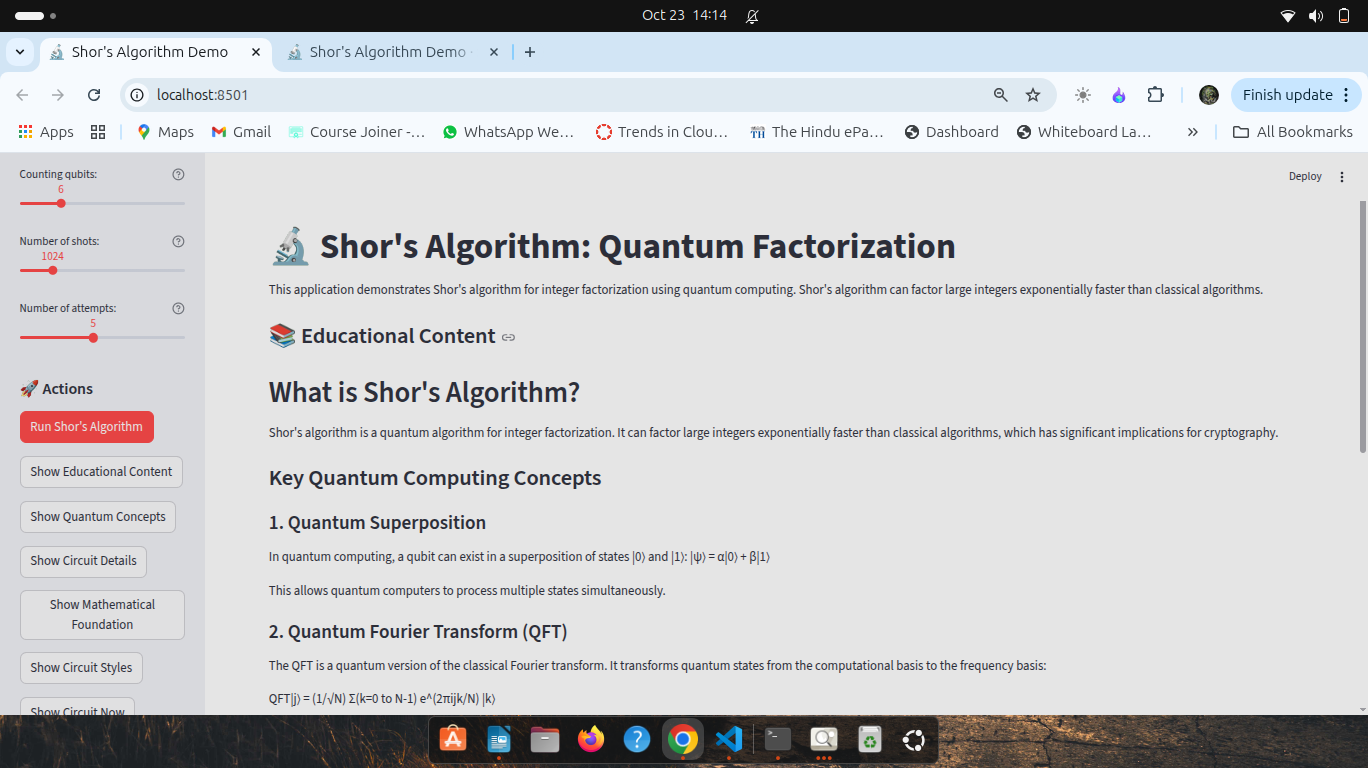


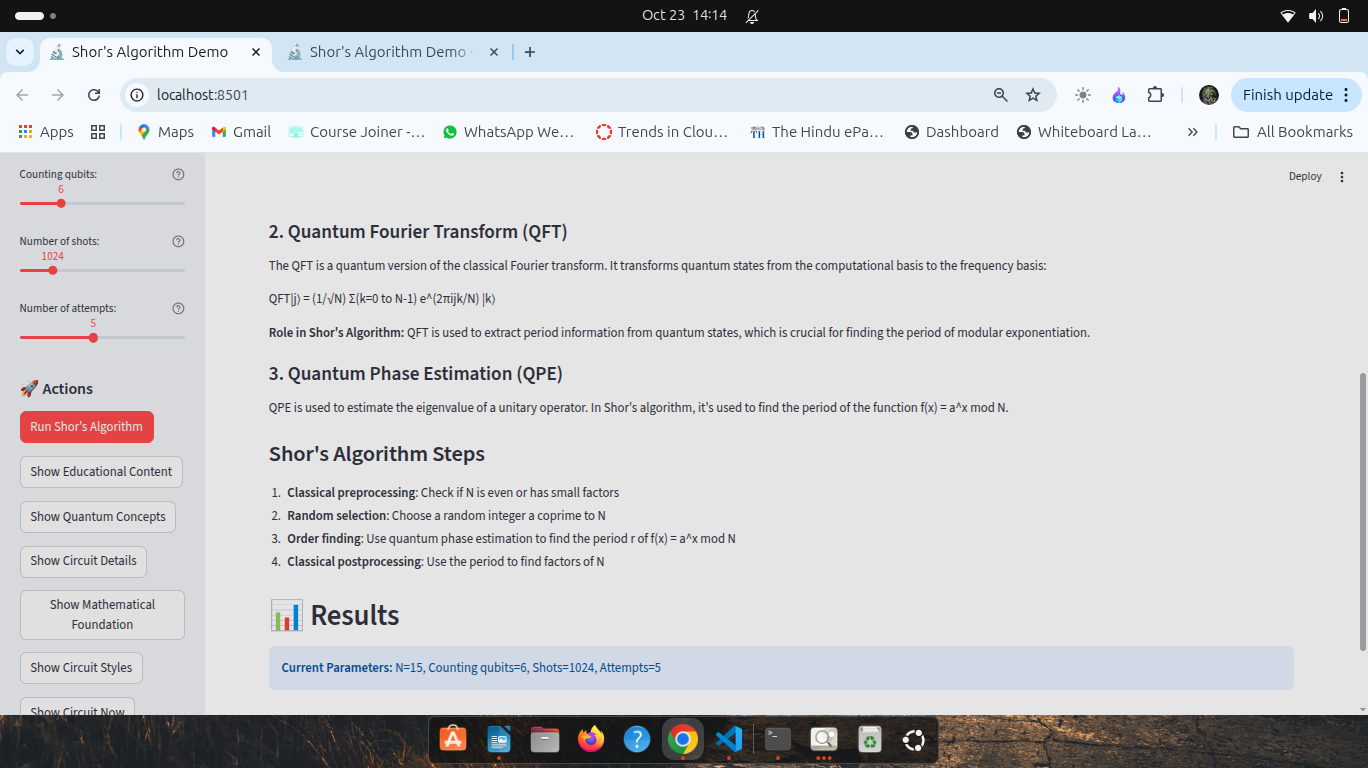


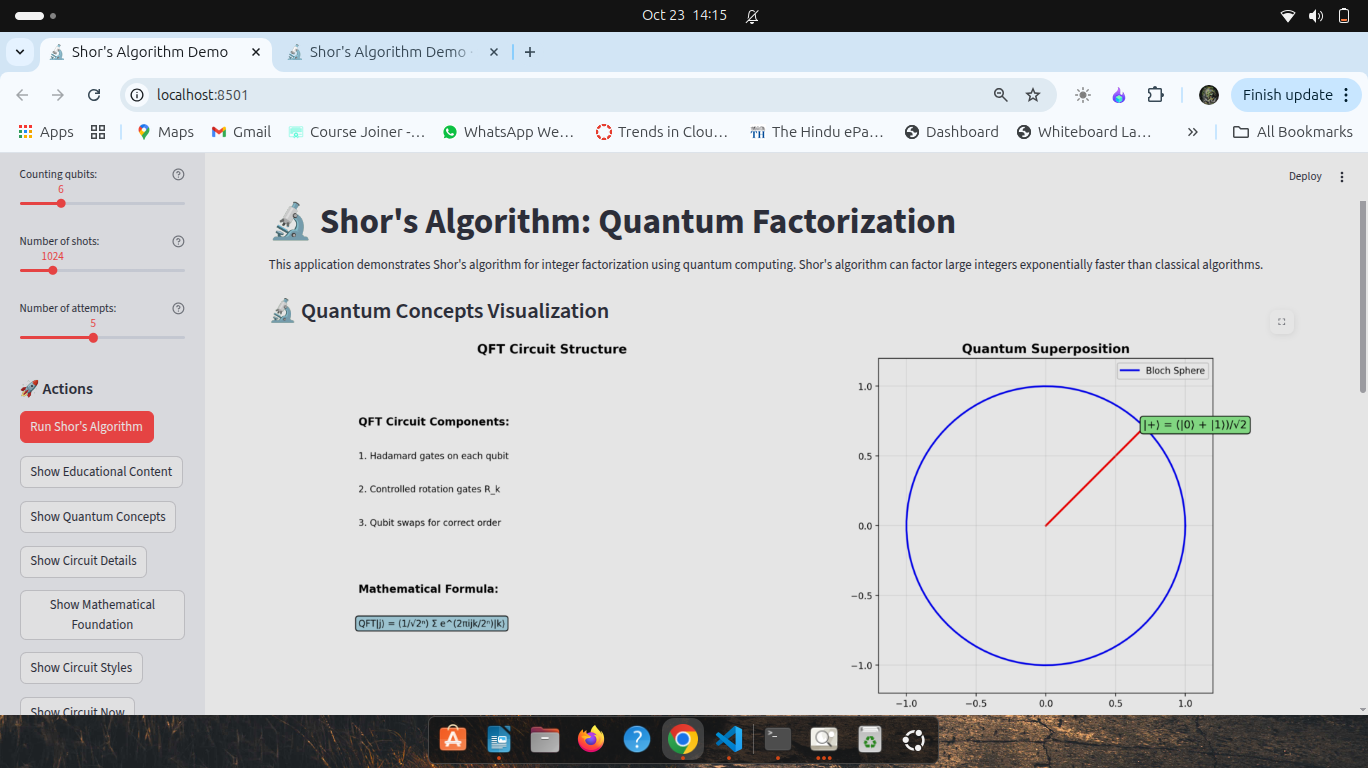


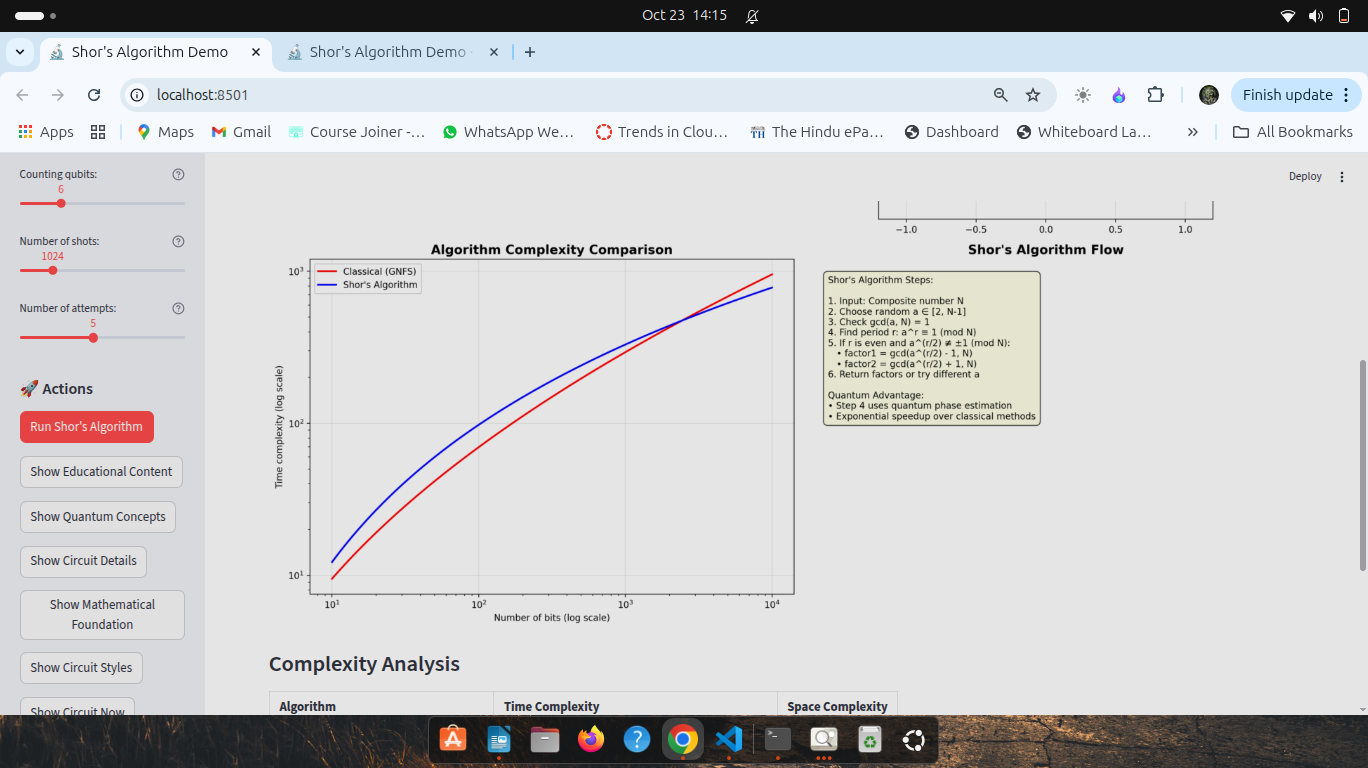


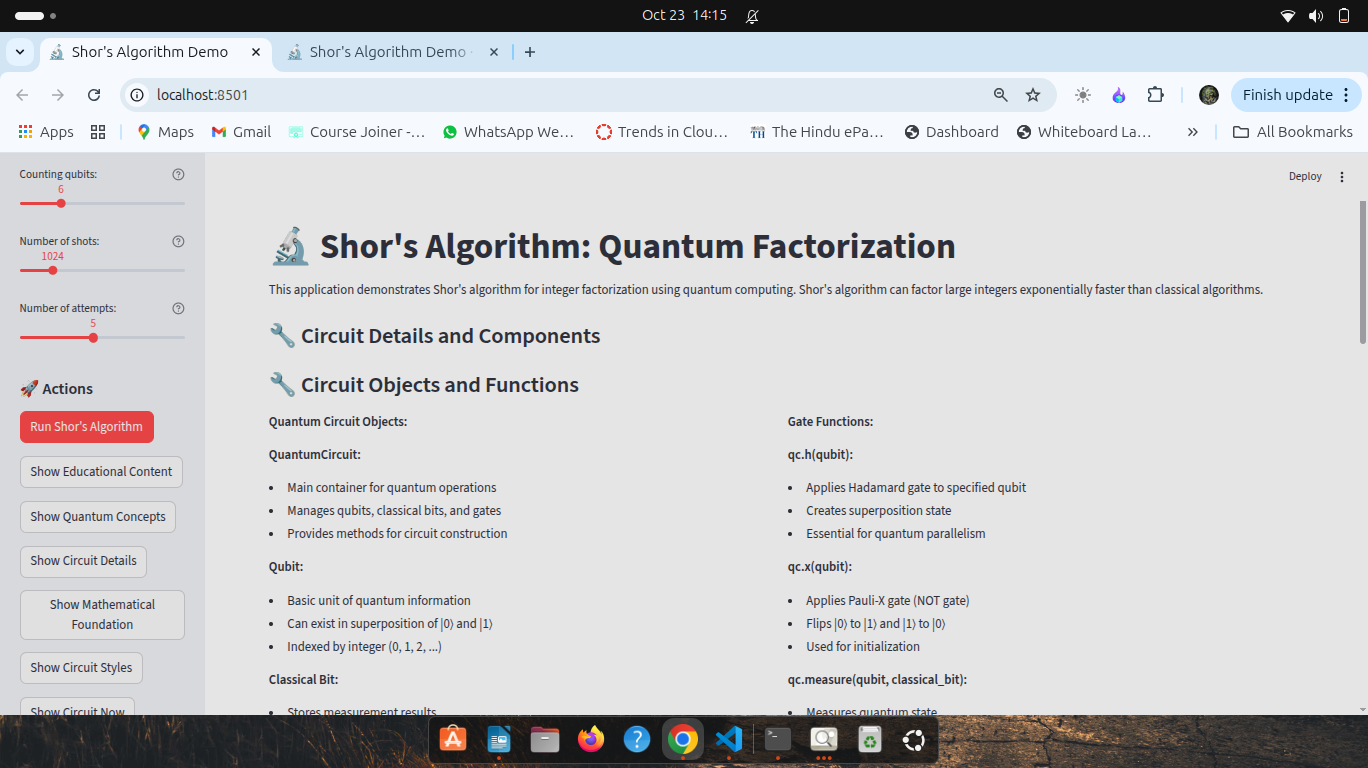


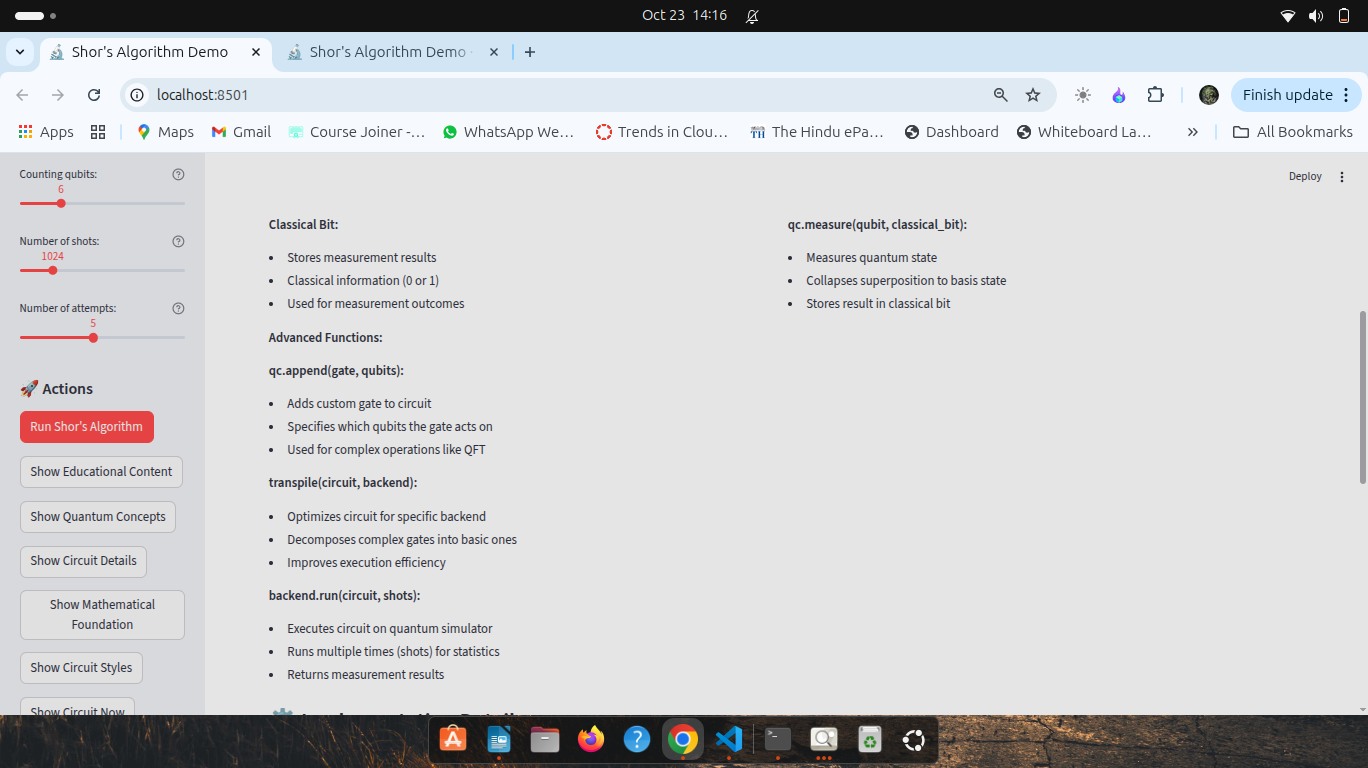


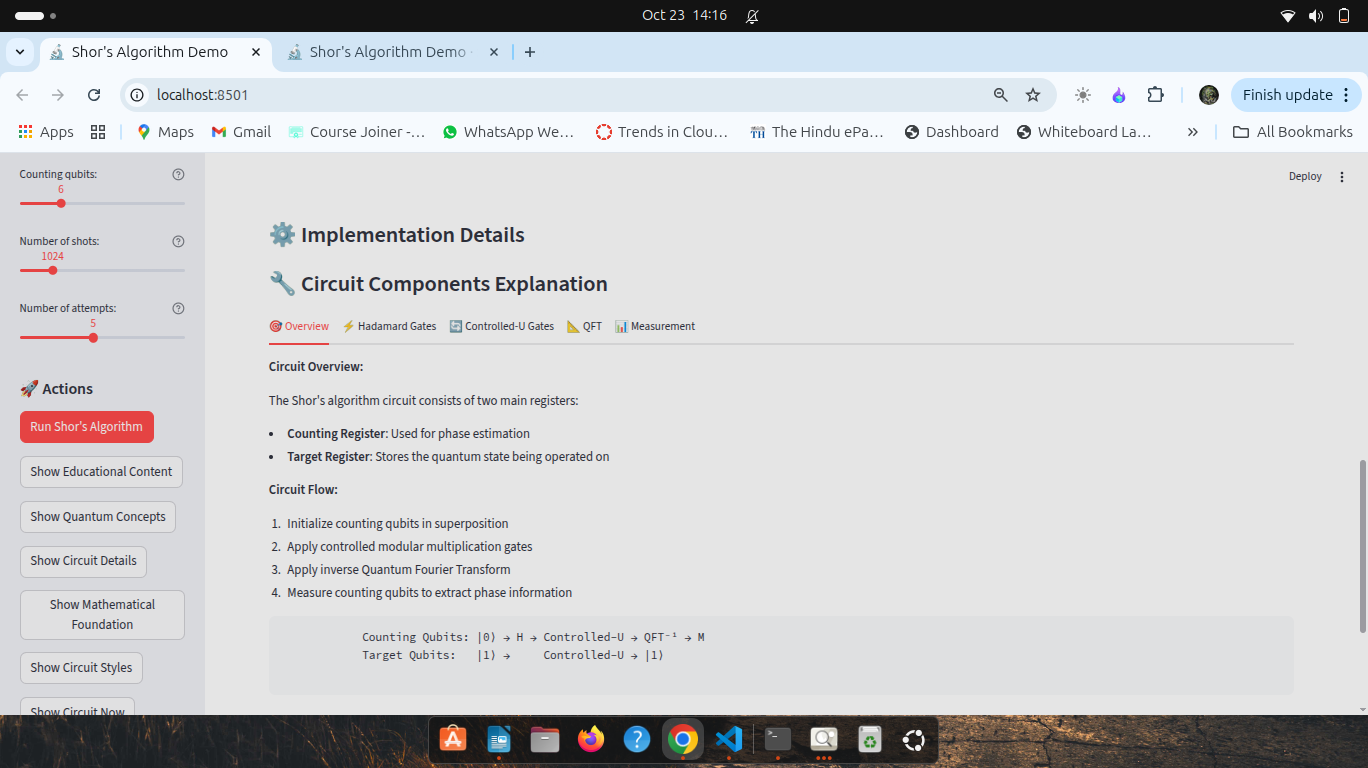


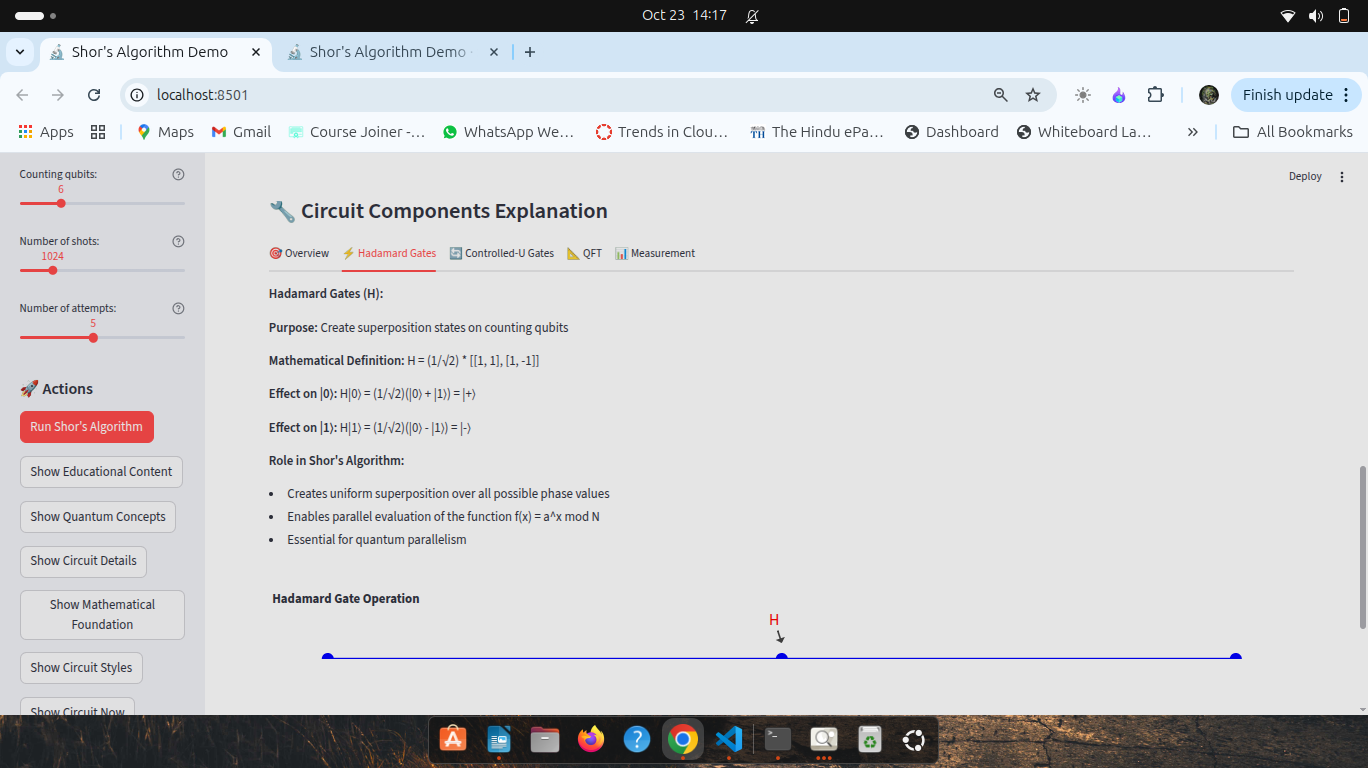
****

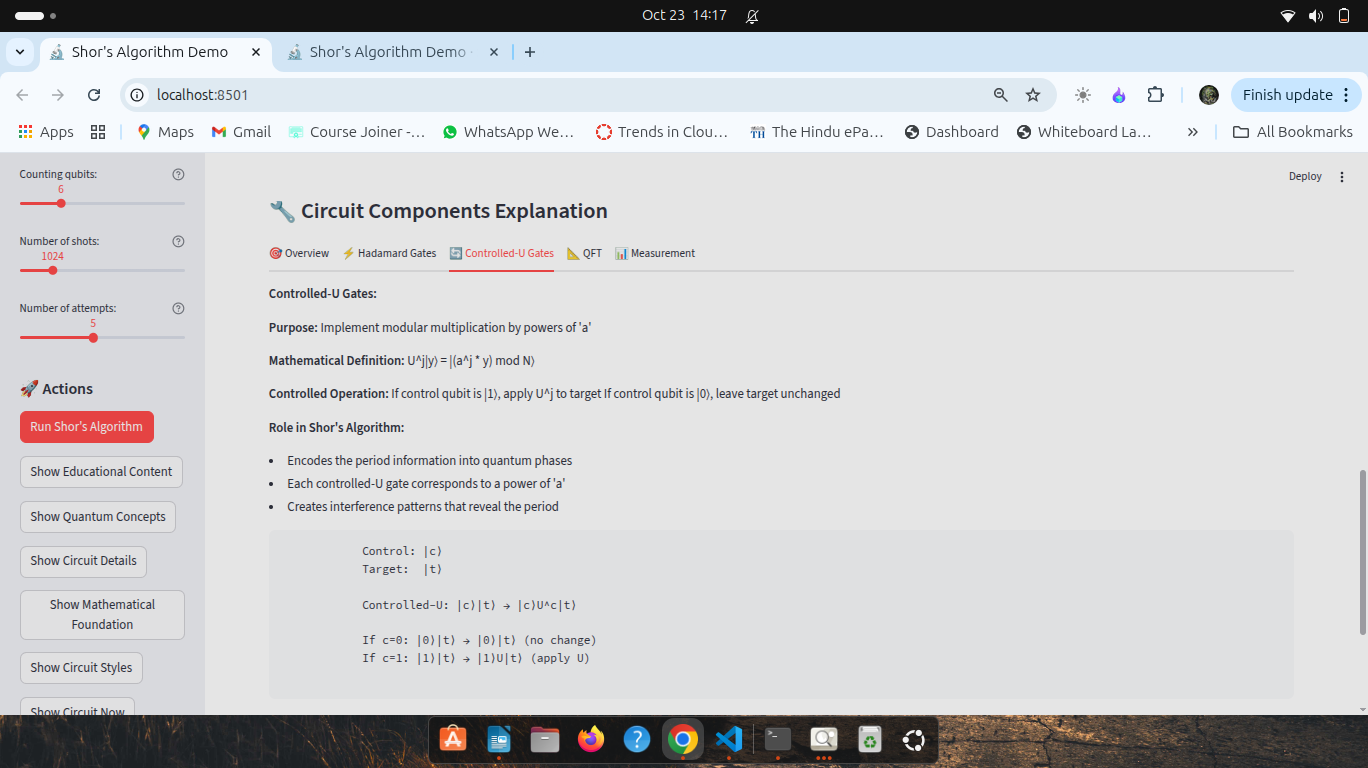
****

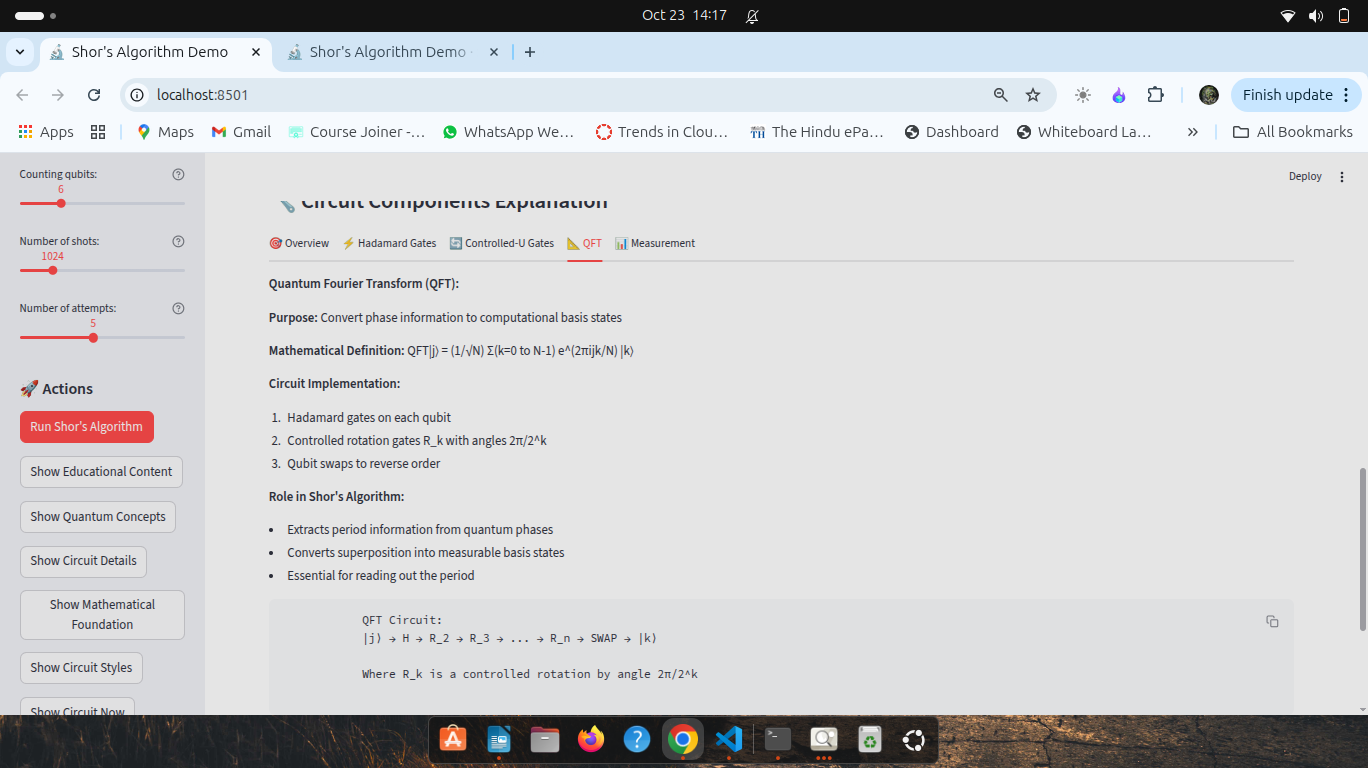
****

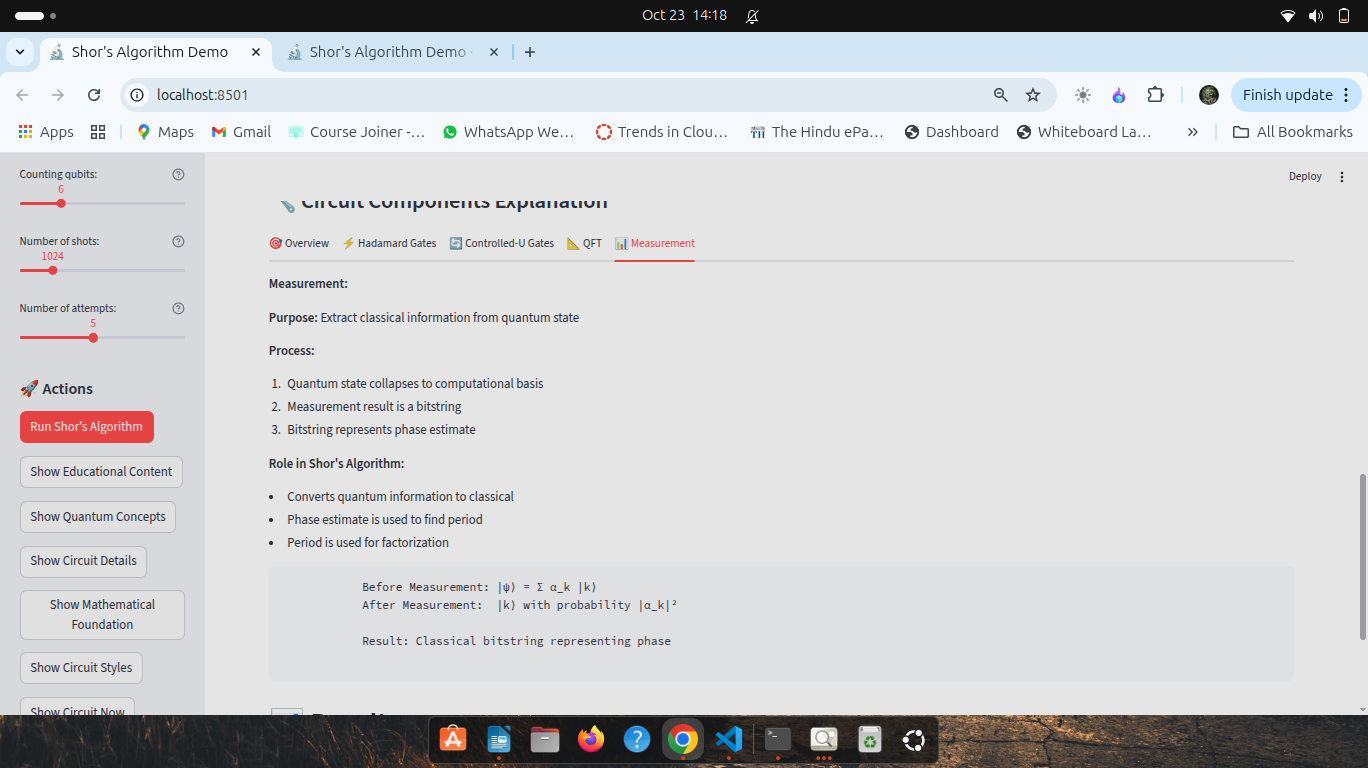
****

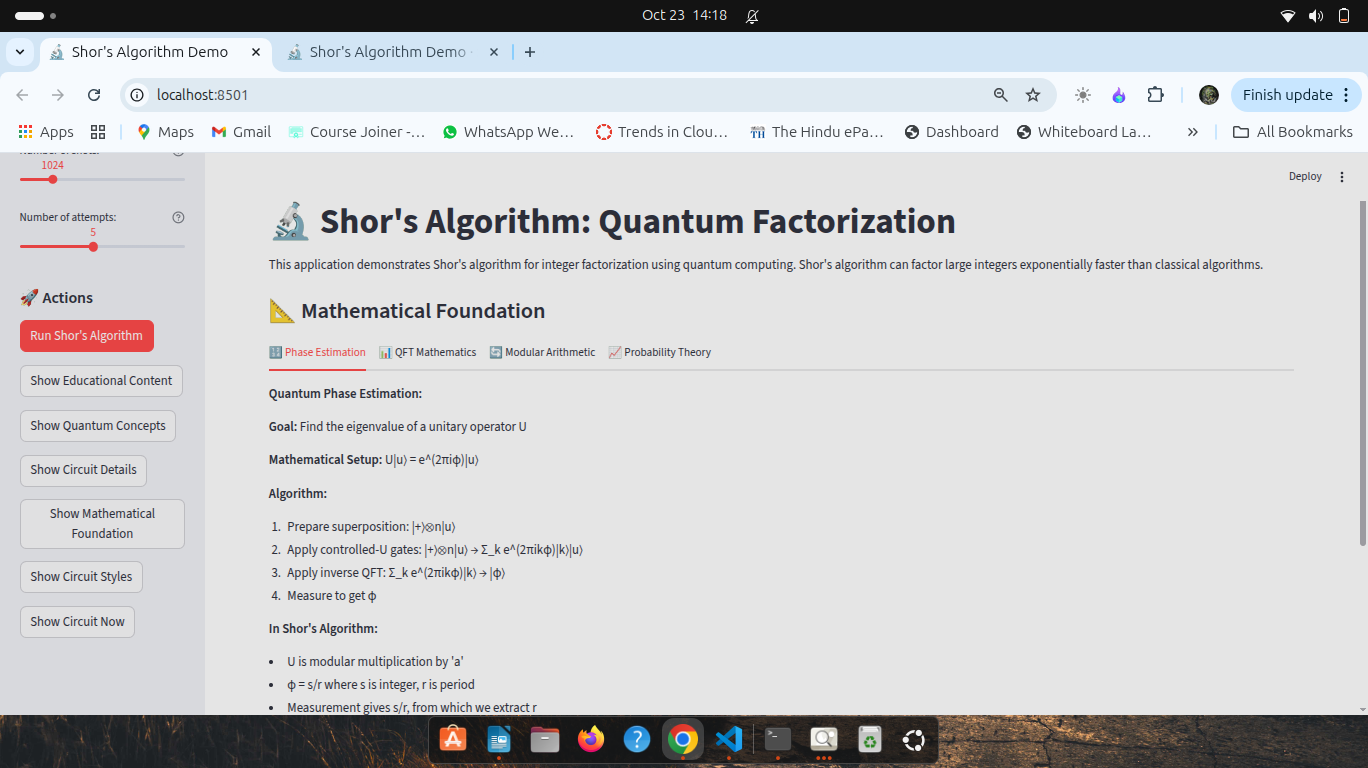
****

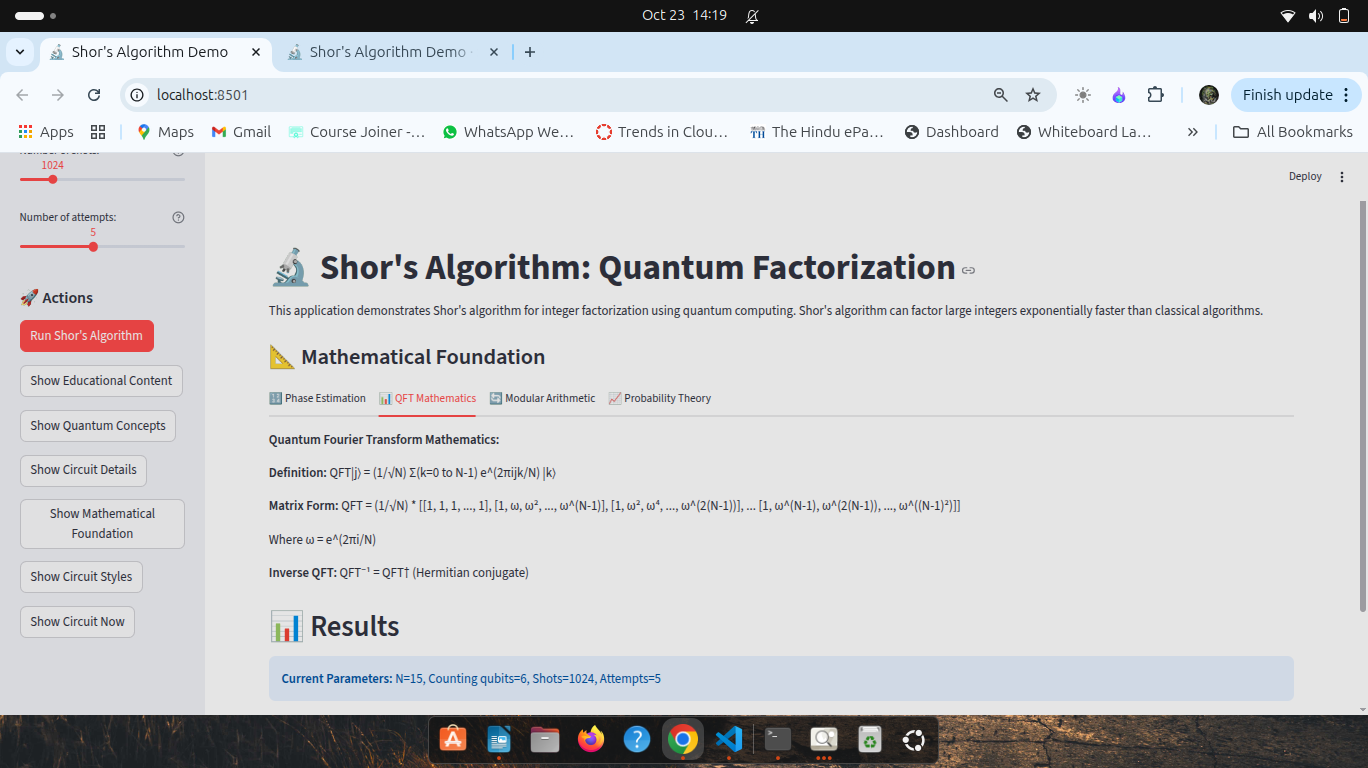
****

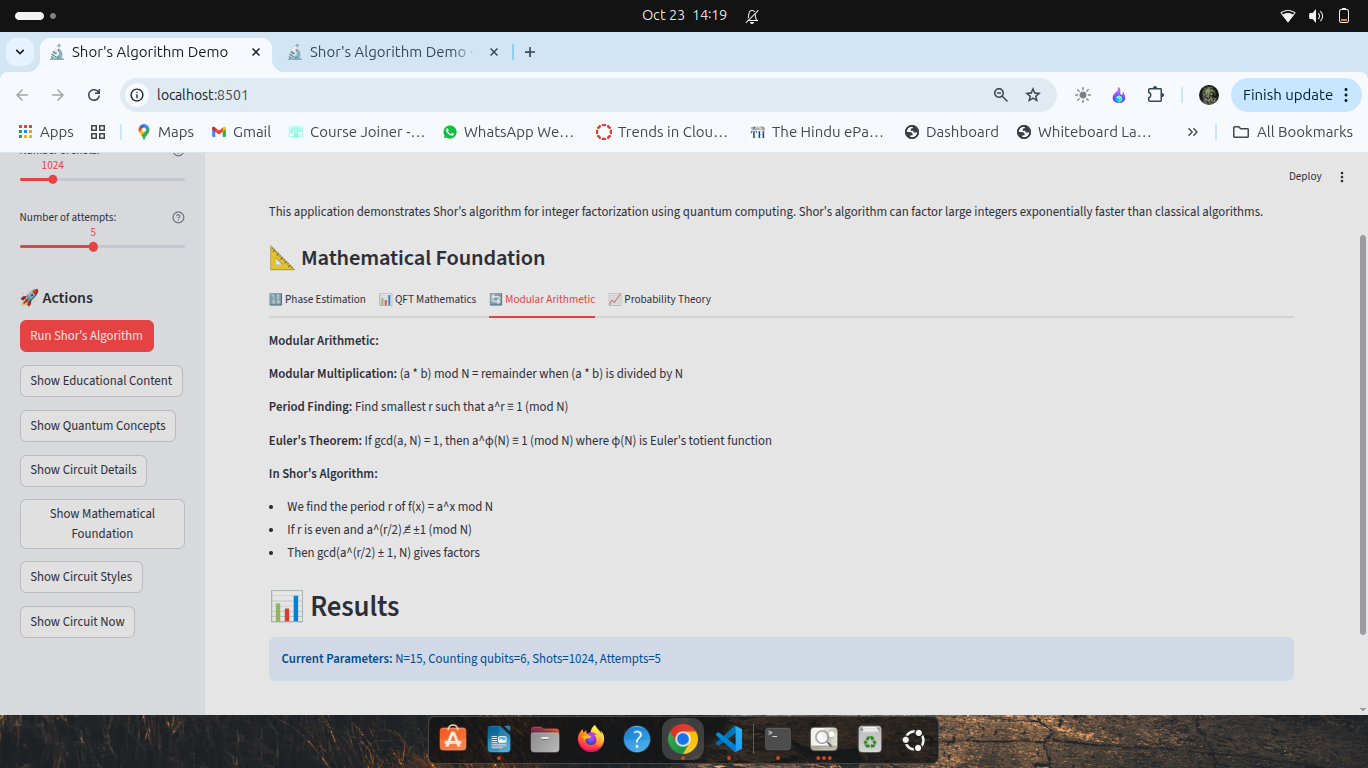
****

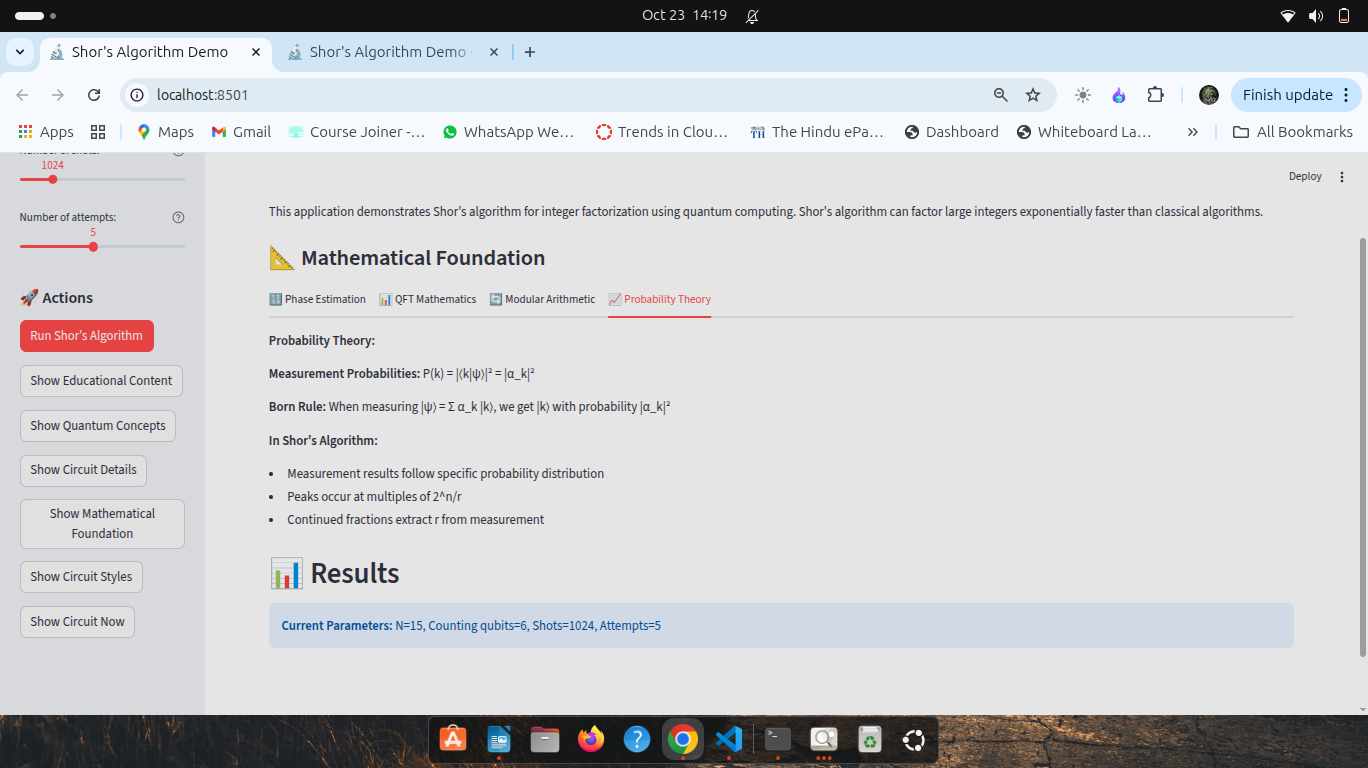
****

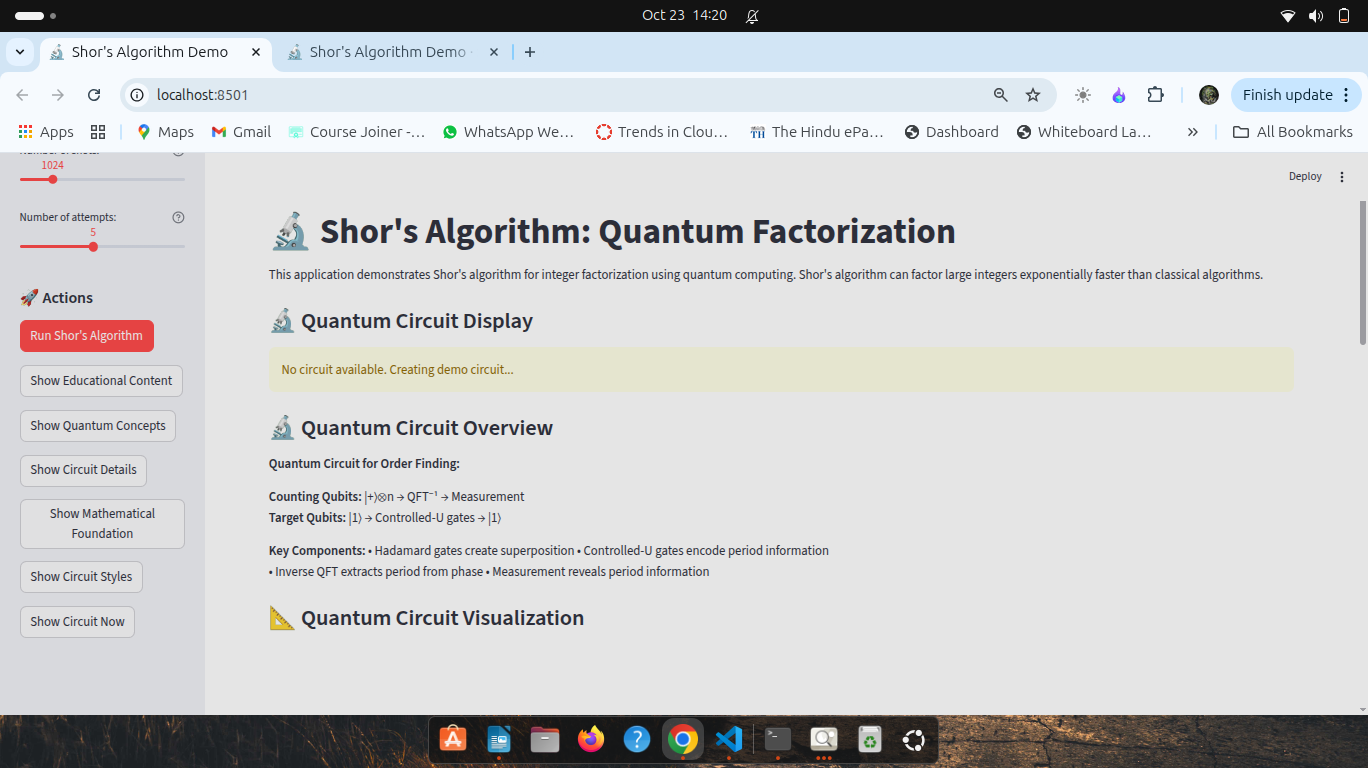
****

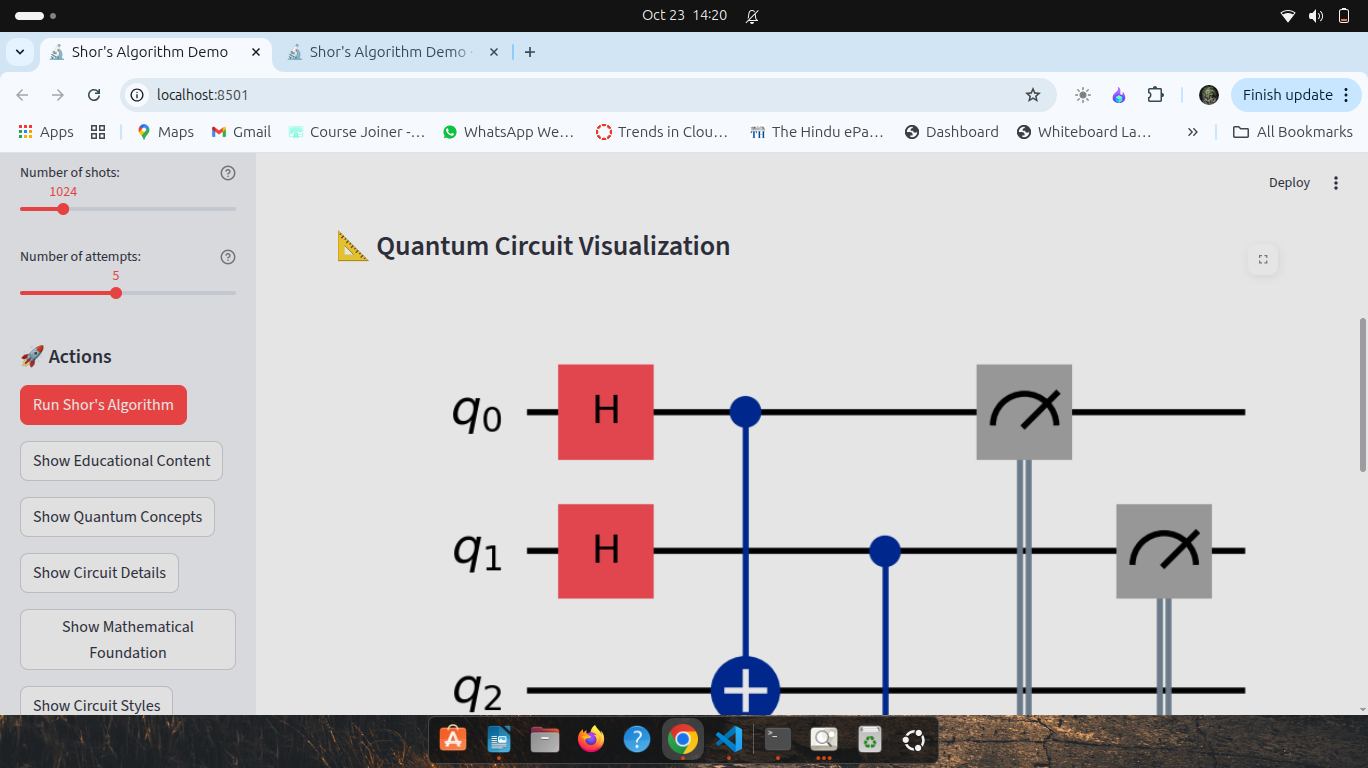
****

****

****

****

****

****

****