Чехов Данил, МПБ-201

1. Постановка задачи

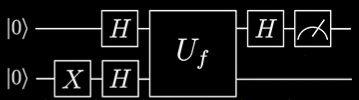
Цель алгоритма Дойча: Определить, является ли функция f: {0,1} -> {0,1} постоянной

(f(0) = f(1)) или сбалансированной (f(0) ≠ f(1)).

Преимущество квантового алгоритма:

Классический подход требует 2 вычисления функции f, чтобы определить её тип, в то время как квантовый алгоритм Дойча решает задачу за 1 вызов функции, используя суперпозицию и интерференцию кубитов.

1. Схема алгоритма



1. Инициализируем два кубита
2. К первому применяем гейт Адамара, ко второму инверсию, а затем гейт Адамара.
3. Применяем оракул U к обоим кубитам
4. Применяем гейт Адамара к первому кубиту
5. Измеряем первый кубит
6. Код на OpenQASM

OPENQASM 2.0;

include "qelib1.inc";

// Объявление квантового регистра (2 кубита) и классического регистра (1 бит)

qreg q[2];

creg c[1];

// Инициализация второго кубита в |1⟩

x q[1];

// Применение гейтов Адамара к обоим кубитам

h q[0];

h q[1];

// Оракул U\_f (f(x) = x) — сбалансированная функция

cx q[0], q[1]; // CNOT: q[1] = q[1] XOR f(q[0])

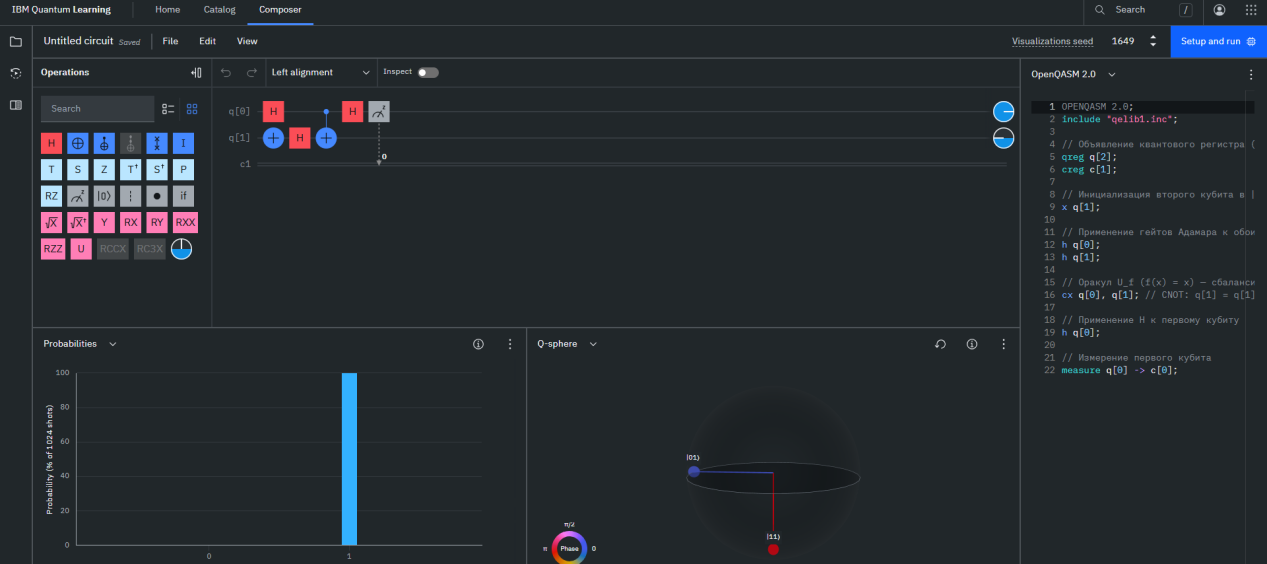
// Применение H к первому кубиту

h q[0];

// Измерение первого кубита

measure q[0] -> c[0];

https://quantum-circuit.com/qconvert



Код на qiskit с учётом шумов:

# === Импорты ===

from qiskit import QuantumCircuit, transpile

from qiskit\_aer import AerSimulator

from qiskit\_ibm\_runtime.fake\_provider import FakeJakartaV2 as FakeJakarta

# === Построение схемы ===

qc = QuantumCircuit(2, 1)

qc.x(1) # подготовка |1⟩ на втором кубите

qc.h([0, 1]) # суперпозиция обоих кубитов

qc.cx(0, 1) # оракул f(x)=x

qc.h(0) # интерференция перед измерением

qc.measure(0, 0)

# === Симуляция с шумом устройства ===

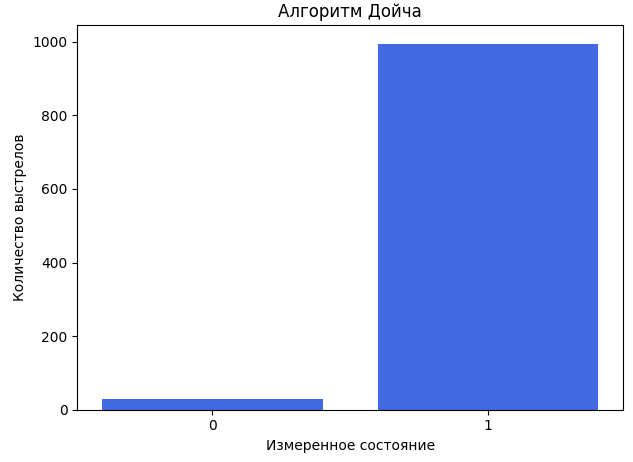
backend = FakeJakarta()

sim = AerSimulator.from\_backend(backend)

t\_qc = transpile(qc, backend=backend, optimization\_level=3)

counts = sim.run(t\_qc, shots=1024).result().get\_counts()

print(counts)



Алгоритм Дойча демонстрирует фундаментальное преимущество квантовых вычислений над классическими, решая задачу определения типа функции

f:{0,1}→{0,1} за один вызов функции, в отличие от классического подхода, требующего двух вычислений. Это достигается за счёт свойств квантовой механики, таких как суперпозиция, интерференция и квантовый параллелизм.

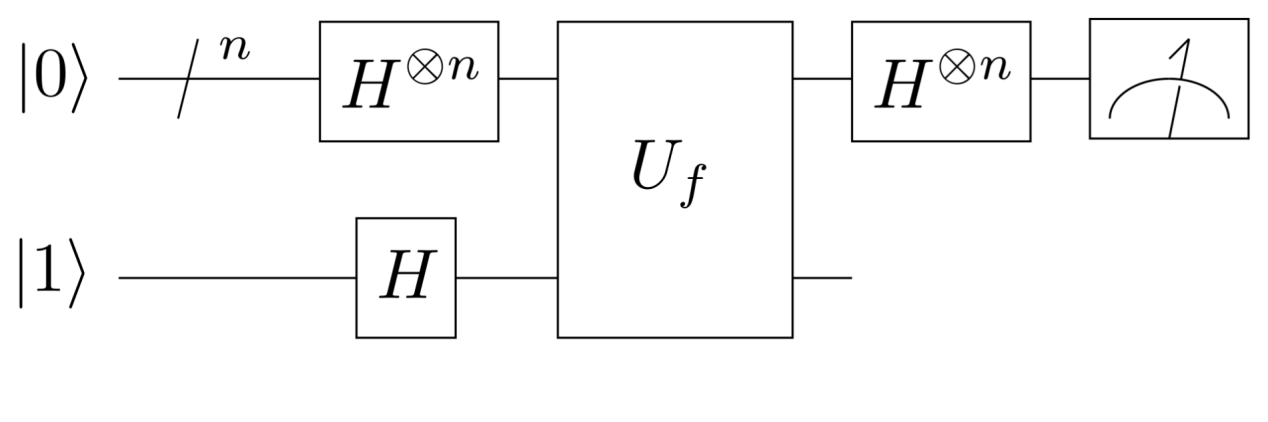
2.Постановка задачи

Цель алгоритма Дойча-Йожи: является ли функция f: {0,1}^n -> {0,1} постоянной (значение f(x) одинаково для всех x) или сбалансированной (значение f(x) равно 0 ровно для половины входов и 1 для другой половины).

Преимущество квантвого алгоритма:

В то время как классический подход требует O(2^n) вычислений функии f, квантовый алгоритм решает задачу за 1 вызов функции, используя квантовый параллелизм и интерференцию.

1. Квантовая схема алгоритма



3)OPENQASM 2.0;

include "qelib1.inc";

qreg q[3]; // 2 входных кубита + 1 вспомогательный

creg c[2]; // Результаты измерения входных кубитов

// Инициализация вспомогательного кубита в |1⟩

x q[2];

h q[2];

// Суперпозиция входных кубитов

h q[0];

h q[1];

// Оракул U\_f (f(x) = x0 ⊕ x1)

cx q[0], q[2];

cx q[1], q[2];

z q[2]; // Фазовый сдвиг для сбалансированной функции

// Интерференция

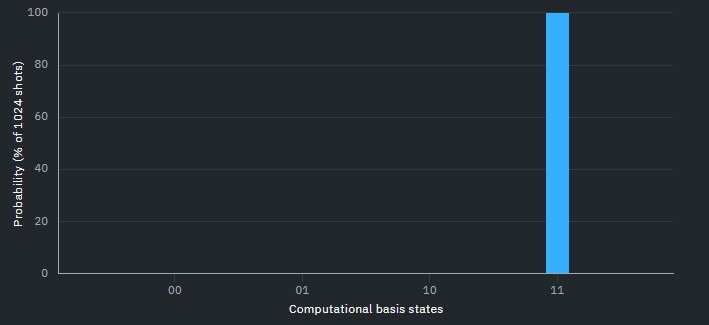
h q[0];

h q[1];

// Измерение входных кубитов

measure q[0] -> c[0];

measure q[1] -> c[1];



Код на qiskit с учётом шумов:

# === Импорты ===

from qiskit import QuantumCircuit, transpile

from qiskit\_aer import AerSimulator

from qiskit\_ibm\_runtime.fake\_provider import FakeJakartaV2 as FakeJakarta

from qiskit.circuit.library import ZGate

# === Построение схемы ===

qc = QuantumCircuit(3, 2)

qc.x(2); qc.h(2) # подготовка |−⟩ на вспомогательном кубите

qc.h([0, 1]) # суперпозиция входных кубитов

qc.cx(0, 2); qc.cx(1, 2) # оракул f(x0,x1)=x0⊕x1

qc.append(ZGate(), [2]) # фазовый сдвиг

qc.h([0, 1]) # интерференция перед измерением

qc.measure([0, 1], [0, 1])

# === Симуляция с шумом устройства ===

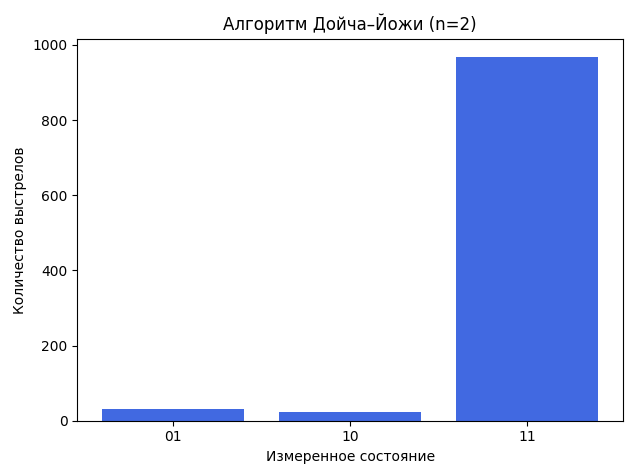
backend = FakeJakarta()

sim = AerSimulator.from\_backend(backend)

t\_qc = transpile(qc, backend=backend, optimization\_level=3)

counts = sim.run(t\_qc, shots=1024).result().get\_counts()

print(counts)



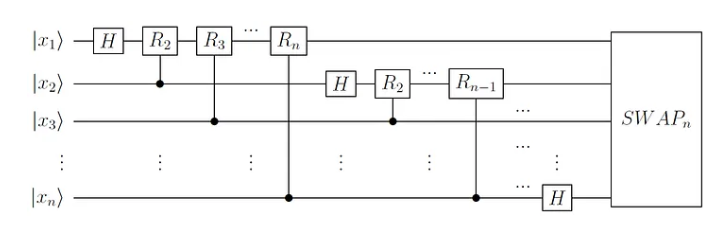
Алгоритм Дойча-Йожи демонстрирует экспоненциальное ускорение, решая задачу определения типа функции для n входов за 1 квантовый запрос (вместо O(2^n) классических вычислений), используя суперпозицию и интерференцию для одновременного анализа всех входов, и служит основой для ключевых квантовых алгоритмов, таких как Шора и Гровера.

1. Постановка задачи:

Цель алгоритма: Выполнить дискретное преобразование Фурье над квантовым состоянием.

Применяется алгоритм в качестве ключевого компонента для факторизации чисел в алгоритме Шора, а также используется в квантовой симуляции и обработке сигналов.

2)Квантовая схема



1. Код

OPENQASM 2.0;

include "qelib1.inc"; // Подключение стандартной библиотеки гейтов

qreg q[3]; // 3 кубита

creg c[3]; // 3 классических бита для измерения

// Применение QFT

// Шаг 1: Гейты Адамара и управляемые фазовые вращения

h q[0];

cu1(pi/2) q[1], q[0]; // Управляемое вращение на π/2 (CPHASE)

cu1(pi/4) q[2], q[0]; // Управляемое вращение на π/4

h q[1];

cu1(pi/2) q[2], q[1]; // Управляемое вращение на π/2

h q[2];

// Шаг 2: SWAP-гейты для коррекции порядка кубитов

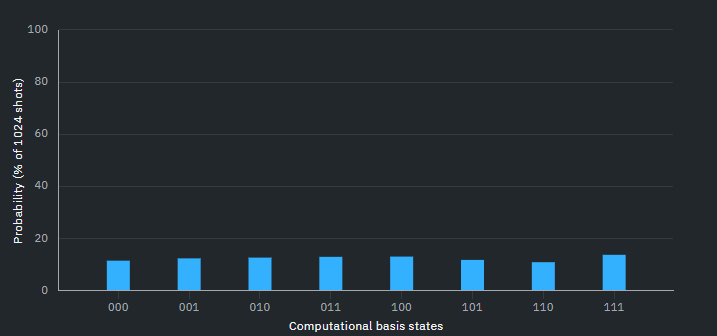
swap q[0], q[2]; // Меняем местами первый и последний кубиты

// Измерение

measure q[0] -> c[0];

measure q[1] -> c[1];

measure q[2] -> c[2];



Код на qiskit с учётом шумов:

import pennylane as qml

import numpy as np

import matplotlib.pyplot as plt

# Параметры шума IBM Brisbane

T1 = 218.04 # мксек

T2 = 127.4 # мксек

gate\_time = 0.15 # мксек

depolarizing\_error = 0.07888 # 7.888%

phase\_damping = 0.01 # 1%

measurement\_error = 0.02003 # 2.003%

def apply\_noisy\_gate(gate, wires, params=None):

"""Применяет гейт с добавлением шума"""

if params is None:

gate(wires=wires)

else:

gate(\*params, wires=wires)

if isinstance(wires, int):

wires = [wires]

for wire in wires:

qml.DepolarizingChannel(depolarizing\_error, wires=wire)

qml.PhaseDamping(phase\_damping, wires=wire)

qml.AmplitudeDamping(gate\_time / T1, wires=wire)

def qft\_rotations(wires):

"""Рекурсивно применяет вращения для QFT"""

if len(wires) == 0:

return

head = wires[0]

tails = wires[1:]

apply\_noisy\_gate(qml.Hadamard, wires=head)

for i, qubit in enumerate(tails, 1):

angle = np.pi / (2 \*\* i)

apply\_noisy\_gate(qml.ControlledPhaseShift, wires=[qubit, head], params=[angle])

qft\_rotations(tails)

def inverse\_qft\_rotations(wires):

"""Обратное QFT преобразование"""

if len(wires) == 0:

return

head = wires[0]

tails = wires[1:]

for i, qubit in enumerate(reversed(tails), 1):

angle = -np.pi / (2 \*\* i)

apply\_noisy\_gate(qml.ControlledPhaseShift, wires=[qubit, head], params=[angle])

apply\_noisy\_gate(qml.Hadamard, wires=head)

inverse\_qft\_rotations(tails)

def swap\_bits(wires):

"""Обмен кубитов для правильного порядка"""

n = len(wires)

for i in range(n // 2):

apply\_noisy\_gate(qml.SWAP, wires=[wires[i], wires[n - i - 1]])

def create\_full\_qft\_test\_circuit(input\_state):

"""Создает схему для проверки QFT туда-обратно"""

n\_qubits = len(input\_state)

dev = qml.device('default.mixed', wires=n\_qubits)

@qml.qnode(dev)

def test\_circuit(state):

# Подготовка состояния

basis\_state = np.zeros(2 \*\* n\_qubits)

idx = int("".join(map(str, state)), 2)

basis\_state[idx] = 1

qml.StatePrep(basis\_state, wires=range(n\_qubits))

# Прямое QFT

qft\_rotations(list(range(n\_qubits)))

swap\_bits(list(range(n\_qubits)))

# Обратное QFT

swap\_bits(list(range(n\_qubits)))

inverse\_qft\_rotations(list(range(n\_qubits)))

# Шум измерения

for wire in range(n\_qubits):

qml.BitFlip(measurement\_error, wires=wire)

return qml.probs(wires=range(n\_qubits))

return test\_circuit

def run\_full\_test(input\_state, runs=100):

"""Запускает полный тест QFT туда-обратно"""

n\_qubits = len(input\_state)

print(f"\n{'=' \* 50}")

print(f"Тест QFT туда-обратно ({runs} раз) для {n\_qubits} кубитов")

print(f"Входное состояние: |{''.join(map(str, input\_state))}⟩")

print("Параметры шума:")

print(f"T1={T1} мксек, T2={T2} мксек")

print(f"Деполяризация: {depolarizing\_error \* 100:.3f}%")

print(f"Ошибка измерения: {measurement\_error \* 100:.3f}%")

# Создаем схему

test\_circuit = create\_full\_qft\_test\_circuit(input\_state)

# Запускаем тест

test\_probs = test\_circuit(input\_state)

print(f"\nВероятности после QFT туда-обратно: {test\_probs}")

# Генерируем результаты измерений

outcomes = np.random.choice(range(2 \*\* n\_qubits), size=runs, p=test\_probs)

counts = {bin(i)[2:].zfill(n\_qubits): np.sum(outcomes == i) for i in range(2 \*\* n\_qubits)}

# Вычисляем точность восстановления

original\_state = "".join(map(str, input\_state))

success\_count = counts.get(original\_state, 0)

success\_rate = success\_count / runs \* 100

print("\nРезультаты измерений:")

for outcome, count in sorted(counts.items()):

print(f"|{outcome}⟩: {count} ({count / runs \* 100:.1f}%)")

print(f"\nТочность восстановления: {success\_rate:.1f}%")

# Визуализация

plt.figure(figsize=(max(8, n\_qubits \* 2), 6))

x = [bin(i)[2:].zfill(n\_qubits) for i in range(2 \*\* n\_qubits)]

y = [counts.get(k, 0) for k in x]

bars = plt.bar(x, y, color='skyblue')

# Подсвечиваем исходное состояние

for i, (outcome, bar) in enumerate(zip(x, bars)):

if outcome == original\_state:

bar.set\_color('green')

plt.text(bar.get\_x() + bar.get\_width() / 2., bar.get\_height(),

f'{bar.get\_height()} (исходное)', ha='center', va='bottom', fontsize=10)

else:

plt.text(bar.get\_x() + bar.get\_width() / 2., bar.get\_height(),

f'{bar.get\_height()}', ha='center', va='bottom', fontsize=10)

plt.xlabel('Состояние')

plt.ylabel('Частота')

plt.title(

f'Восстановление состояния после QFT туда-обратно\nИсходное: |{original\_state}⟩, Точность: {success\_rate:.1f}%')

plt.grid(axis='y', alpha=0.5)

plt.xticks(rotation=45 if n\_qubits > 3 else 0)

plt.tight\_layout()

plt.show()

return success\_rate

# Тестируем для разных состояний

input\_state\_2 = [1, 0] # 2 кубита

input\_state\_3 = [1, 0, 1] # 3 кубита

input\_state\_4 = [1, 0, 1, 0] # 4 кубита

print("Начало тестирования QFT туда-обратно...")

# Запускаем тесты

success\_rates = []

success\_rates.append(run\_full\_test(input\_state\_2))

success\_rates.append(run\_full\_test(input\_state\_3))

success\_rates.append(run\_full\_test(input\_state\_4))

# Сводный график точности

plt.figure(figsize=(8, 5))

x = [2, 3, 4]

plt.bar(x, success\_rates, color=['skyblue', 'lightgreen', 'salmon'])

plt.xlabel('Количество кубитов')

plt.ylabel('Точность восстановления (%)')

plt.title('Точность восстановления состояния после QFT туда-обратно')

plt.xticks(x)

plt.grid(axis='y', alpha=0.5)

for i, rate in enumerate(success\_rates):

plt.text(x[i], rate + 1, f'{rate:.1f}%', ha='center')

plt.tight\_layout()

plt.show()

Тест QFT туда-обратно (100 раз) для 2 кубитов

Входное состояние: |10⟩

Параметры шума:

T1=218.04 мксек, T2=127.4 мксек

Деполяризация: 7.888%

Ошибка измерения: 2.003%

Вероятности после QFT туда-обратно: [0.22180039 0.15845176 0.39932336 0.2204245 ]

Результаты измерений:

|00⟩: 28 (28.0%)

|01⟩: 9 (9.0%)

|10⟩: 37 (37.0%)

|11⟩: 26 (26.0%)

Точность восстановления: 37.0%

==================================================

Тест QFT туда-обратно (100 раз) для 3 кубитов

Входное состояние: |101⟩

Параметры шума:

T1=218.04 мксек, T2=127.4 мксек

Деполяризация: 7.888%

Ошибка измерения: 2.003%

Вероятности после QFT туда-обратно: [0.127648 0.11923509 0.06846658 0.07025787 0.19794365 0.1962554

0.10582817 0.11436524]

Результаты измерений:

|000⟩: 12 (12.0%)

|001⟩: 14 (14.0%)

|010⟩: 4 (4.0%)

|011⟩: 8 (8.0%)

|100⟩: 15 (15.0%)

|101⟩: 20 (20.0%)

|110⟩: 15 (15.0%)

|111⟩: 12 (12.0%)

Точность восстановления: 20.0%

==================================================

Тест QFT туда-обратно (100 раз) для 4 кубитов

Входное состояние: |1010⟩

Параметры шума:

T1=218.04 мксек, T2=127.4 мксек

Деполяризация: 7.888%

Ошибка измерения: 2.003%

Вероятности после QFT туда-обратно: [0.05562896 0.06944134 0.05776363 0.06435075 0.03974185 0.04594274

0.04242961 0.04471266 0.07256824 0.09397682 0.0821019 0.0935444

0.05184676 0.06160577 0.06037824 0.06396633]

Результаты измерений:

|0000⟩: 4 (4.0%)

|0001⟩: 3 (3.0%)

|0010⟩: 10 (10.0%)

|0011⟩: 6 (6.0%)

|0100⟩: 4 (4.0%)

|0101⟩: 7 (7.0%)

|0110⟩: 6 (6.0%)

|0111⟩: 1 (1.0%)

|1000⟩: 4 (4.0%)

|1001⟩: 6 (6.0%)

|1010⟩: 11 (11.0%)

|1011⟩: 11 (11.0%)

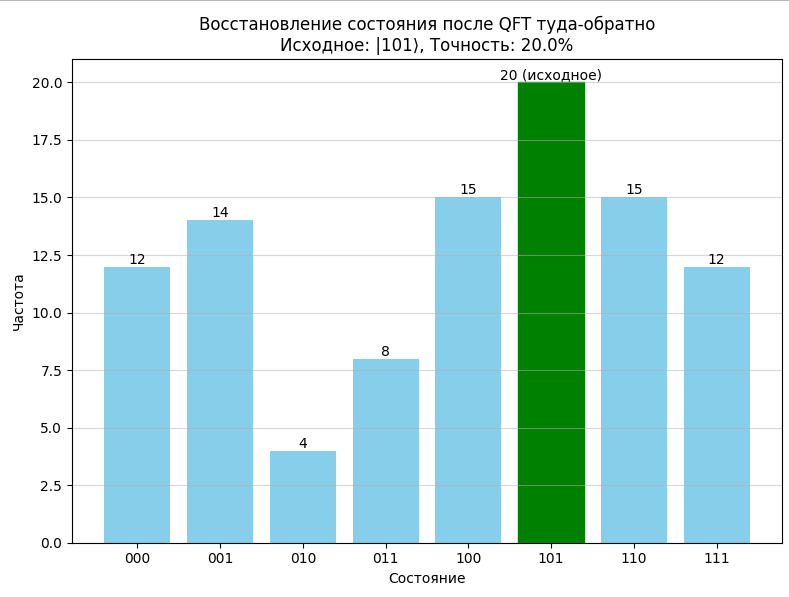
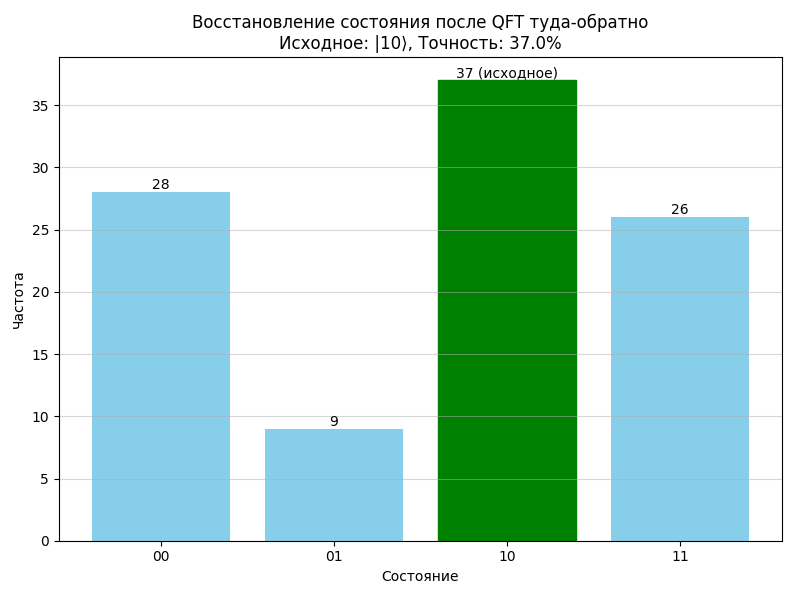
|1100⟩: 5 (5.0%)

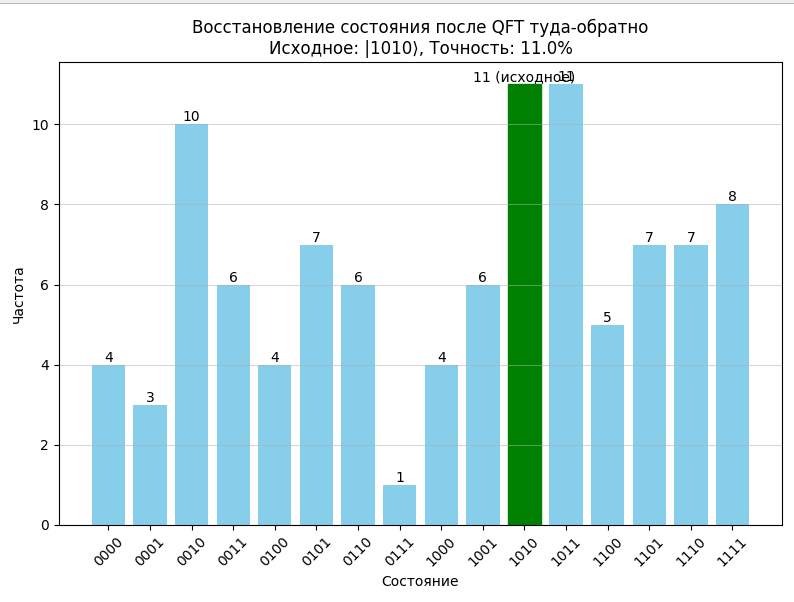
|1101⟩: 7 (7.0%)

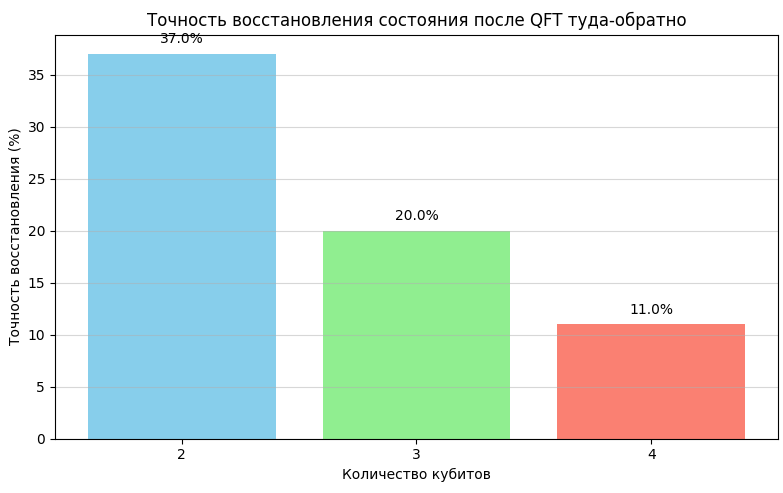
|1110⟩: 7 (7.0%)

|1111⟩: 8 (8.0%)

Точность восстановления: 11.0%







Алгоритм квантового преобразования Фурье (QFT) обеспечивает экспоненциальное ускорение по сравнению с классическим аналогом, являясь ключевым компонентом алгоритмов Шора и квантовой симуляции, но требует устойчивости к шумам и коррекции ошибок для реализации на современных квантовых устройствах.