Simons Problem

Simon's problem was the first example of a problem that could be solved faster by quantum computers than by classical computers, since classical computers take exponentially longer to solve. This problem was one of the first to demonstrate the potential advantage of quantum computers over classical computers.

Description

Simon's problem is to analyze an unknown binary function $f: \{0,1\}^n \to \{0,1\}^n$. The function has the special property that there is an unknown bit pattern $s \in \{0,1\}^n$ such that for all x and y in the definition range of the function:

$$f(x) = f(y) \Leftrightarrow x \oplus y = s$$

$\oplus$ represents the addition modulo 2 or the XOR.

The function f delivers the same output value for exactly two different input values. These input values differ by exactly the bit pattern s.

Task

Find the secret bit pattern s.

Classical solution

In order to find the bit pattern s, classical algorithms in the deterministic case require an exponential number of queries of the function f. Note that there exist stochastic approaches that speed up this calculation.

If we consider the function $f: \{0,1\}^n \to \{0,1\}^n$, we need $2^{n-1} + 1$ queries that guarantees us to be able to distinguish between the two cases.

Quantum solution

A quantum computer can solve Simon's problem more efficiently by using the principles of quantum parallelism and quantum interference. The quantum algorithm for Simon's problem works like this:

- **Superposition**: The quantum computer initializes a register of n qubits in a superposition state. To do this, it uses Hadamard gates.
- **Entanglement**: The quantum computer brings the register of n qubits into an entangled state.
- **Queries**: A quantum Fourier transformation is applied to the state. This Fourier transformation incorporates the secret bit pattern s into the entangled qubits.
- **Resolution of the superposition**: The superposition is resolved by applying the Hadamard gates. The results of the entanglement are transferred to the individual qubits.
- **Measurement**: At this point, the qubits "know" the secret bit pattern s, but this detailed information is lost during the measurement process. Measurement gives one bit of the solution. The partial results resulting from multiple measurements must therefore be combined using classical post-processing.

The quantum algorithm itself only requires a single query of the function f to find the bit pattern s.

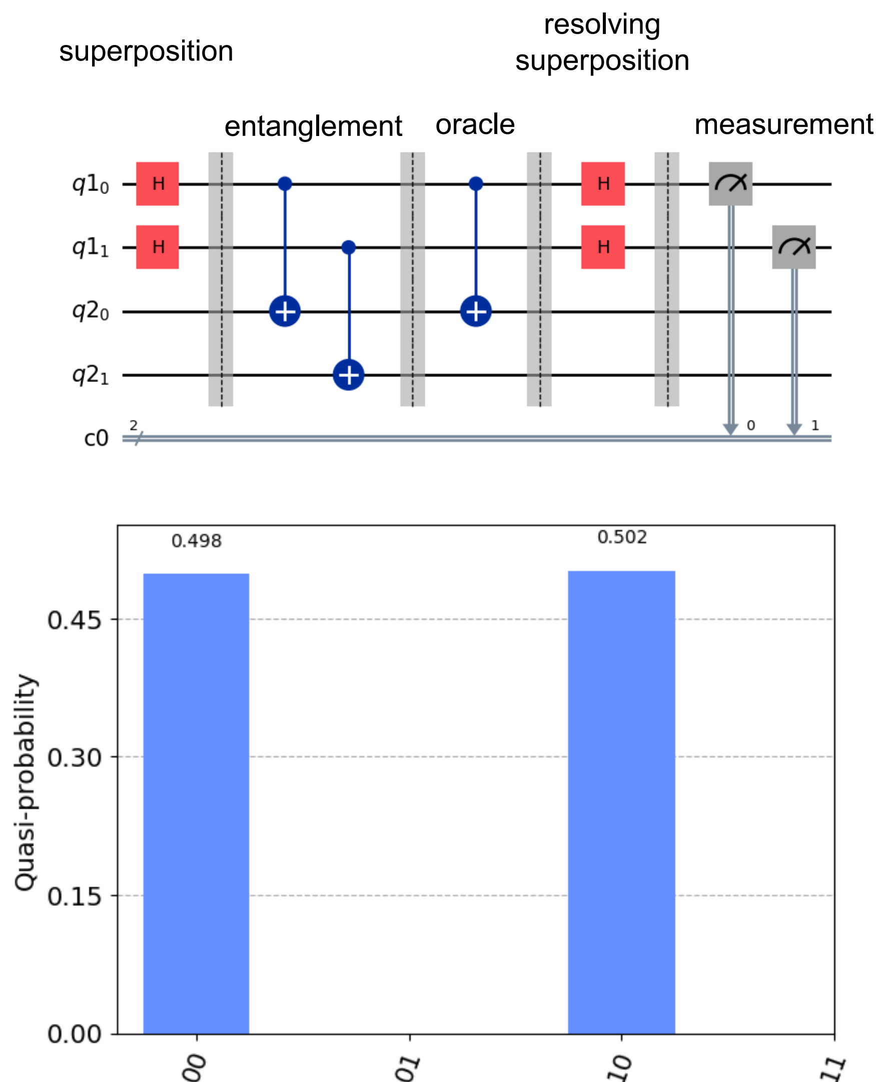
In the subsequent classical post-processing, a further $n - 1$ calculations are required to extract the information from the n qubits, a total of $O(n)$ calculations. This represents an exponential improvement over the classical deterministic method.

Significance

Simon's problem was one of the first examples to show that quantum computers can solve certain problems exponentially faster than classical computers. It paved the way for further developments in quantum computing, including Shor's algorithm and Grover's algorithm. These demonstrate, using other examples, that quantum computers are significantly more efficient at certain tasks.

Example 1

Given is a circuit with the secret string $c = 10$. We use Visual Studio Code to find possible candidates for the secret string.



The results of the simulation are the options 00 and 10. Since 00 is not a valid candidate for c, one can see from the graph that the secret string $c = 10$.

Example 2

A $2{:}1$ function is given with secret string $c = 0101$.

The function explicit:

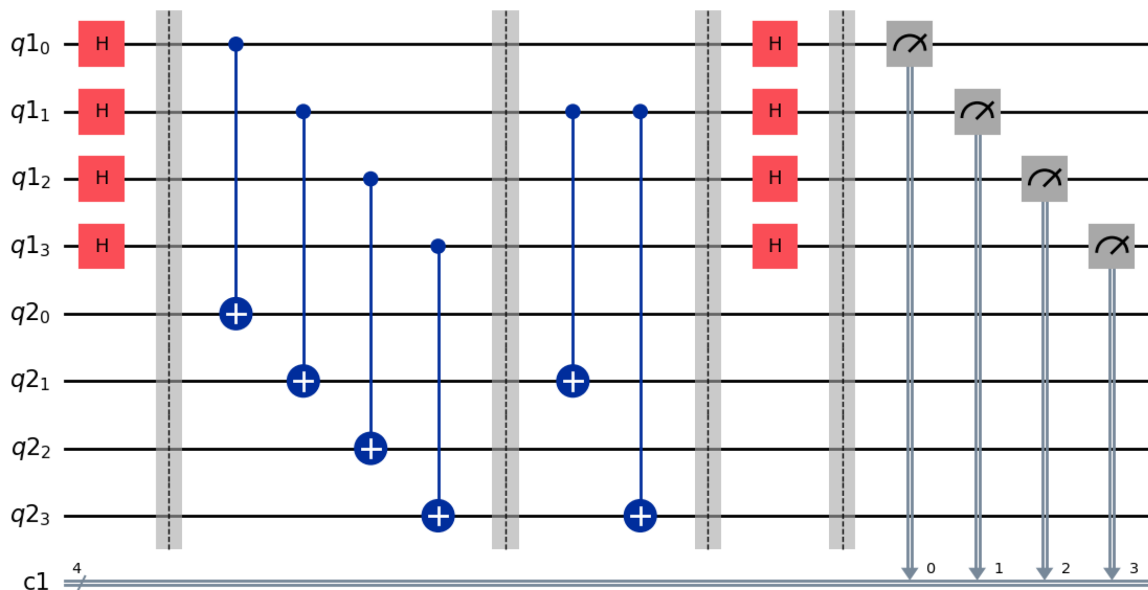
$$f(x){:}\{0\ 1\}^4 \to \{0\ 1\}^4$$

x →	$f(x)$	x →	$f(x)$
$\{0000\}$	$\{1101\}$	$\{1000\}$	$\{1000\}$
$\{0001\}$	$\{0110\}$	$\{1001\}$	$\{1011\}$
$\{0010\}$	$\{0101\}$	$\{1010\}$	$\{1001\}$
$\{0011\}$	$\{1111\}$	$\{1011\}$	$\{0001\}$
$\{0100\}$	$\{0110\}$	$\{1100\}$	$\{1011\}$
$\{0101\}$	$\{1101\}$	$\{1101\}$	$\{1000\}$
$\{0110\}$	$\{1111\}$	$\{1110\}$	$\{0001\}$
$\{0111\}$	$\{0101\}$	$\{1111\}$	$\{1001\}$

The function is $2{:}1$, the constant $c = 0101$. c is often referred as “secret string”.

Circuit:

The circuit is constructed using the secret string $c = 0101$.



Like above holds: After the second row of Hadamards the qubits “know” the solution, but each measurement delivers only one bit of it. What we can use is:

The solution is orthogonal to the bit strings b_x delivered by each measurement.

We remember that $|b\rangle|f(x)\rangle$ comes from two different sources: x_0 and $x_0 \oplus c$ because the function is $2{:}1$.

We calculate the coefficient of $|b\rangle|f(x)\rangle$:

$$\left|\frac{1}{2^n}\left((-1)^{x_0\cdot b}+(-1)^{(x_0\oplus c)\cdot b}\right)\right|^2=\left|\frac{1}{2^n}\left((-1)^{x_0\cdot b}+(-1)^{x_0\cdot b\oplus c\cdot b}\right)\right|^2=$$

$$\left|\frac{1}{2^n}\left((-1)^{x_0\cdot b}+(-1)^{x_0\cdot b}(-1)^{c\cdot b}\right)\right|^2=\left|\frac{1}{2^n}\left((-1)^{x_0\cdot b}\left(1+(-1)^{c\cdot b}\right)\right)\right|^2=;$$

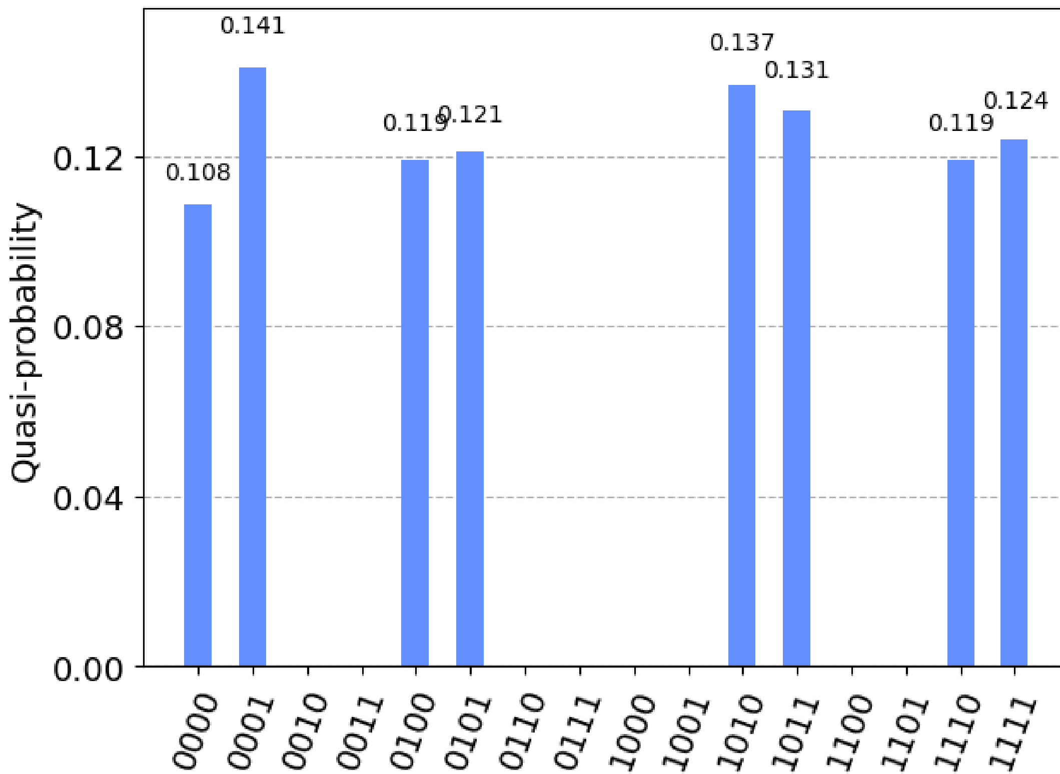
We note that $\left|\left((-1)^{x_0\cdot b}\right)\right|^2$ always gives 1 and proceed:

$$\left|\frac{1}{2^n}\left(1+(-1)^{c\cdot b}\right)\right|^2$$

If the inner product $c\cdot b=0$ we get the probability $\left(\frac{2}{2^n}\right)^2=4^{1-n}$, in our case $\frac{1}{64}$.

If the inner product $c\cdot b=1$ we get 0.

Via Visual Studio we simulate the circuit to find possible bit strings b_x.



We get the candidates: 0000, 0001, 0100, 0101, 1010, 1011, 1110 and 1111.

0000 is no valid option for the secret string because this would state that the function is $1:1$.

Procedure

- We build the scalar products between the bit strings b_x and all possible four-vectors except the vector 0000.
- We look for perpendiculars between the bit strings and the four-vectors. We calculate them via the scalar product $mod2$ giving the result zero.
- We only use bit strings b_x that are linearly independent. Each such bit string reduces the number of possible vectors for the secret string c by half.

We check this explicit with the example above.

Candidate 1: 0001

$(0001)\begin{pmatrix}0\\0\\0\\1\end{pmatrix}=1$	$(0001)\begin{pmatrix}0\\0\\1\\0\end{pmatrix}=0$	$(0001)\begin{pmatrix}0\\0\\1\\1\end{pmatrix}=1$	$(0001)\begin{pmatrix}0\\1\\0\\0\end{pmatrix}=0$
$(0001)\begin{pmatrix}0\\1\\0\\1\end{pmatrix}=1$	$(0001)\begin{pmatrix}0\\1\\1\\0\end{pmatrix}=0$	$(0001)\begin{pmatrix}0\\1\\1\\1\end{pmatrix}=1$	$(0001)\begin{pmatrix}1\\0\\0\\0\end{pmatrix}=0$
$(0001)\begin{pmatrix}1\\0\\0\\1\end{pmatrix}=1$	$(0001)\begin{pmatrix}1\\0\\1\\0\end{pmatrix}=0$	$(0001)\begin{pmatrix}1\\0\\1\\1\end{pmatrix}=1$	$(0001)\begin{pmatrix}1\\1\\0\\0\end{pmatrix}=0$
$(0001)\begin{pmatrix}1\\1\\0\\1\end{pmatrix}=1$	$(0001)\begin{pmatrix}1\\1\\1\\0\end{pmatrix}=0$	$(0001)\begin{pmatrix}1\\1\\1\\1\end{pmatrix}=1$	

Candidate 2: 0100

$(0100)\begin{pmatrix}0\\0\\1\\0\end{pmatrix}=0$	$(0100)\begin{pmatrix}0\\1\\0\\0\end{pmatrix}=1$
$(0100)\begin{pmatrix}0\\1\\1\\0\end{pmatrix}=1$	$(0100)\begin{pmatrix}1\\0\\0\\0\end{pmatrix}=0$
$(0100)\begin{pmatrix}1\\0\\1\\0\end{pmatrix}=0$	$(0100)\begin{pmatrix}1\\1\\0\\0\end{pmatrix}=1$
$(0100)\begin{pmatrix}1\\1\\1\\0\end{pmatrix}=1$	

Candidate 3: 0101

$(0101)\begin{pmatrix}0\\0\\1\\0\end{pmatrix}=0$	$(0101)\begin{pmatrix}1\\0\\0\\0\end{pmatrix}=0$
$(0101)\begin{pmatrix}1\\0\\1\\0\end{pmatrix}=0$	

We see that candidate 3 doesn't reduce the number of possibilities, because candidate 3 is not linearly independent of candidates 1 and 2. In fact, it is the sum of both.

We try candidate 4: 1010

$(1010)\begin{pmatrix}0\\0\\1\\0\end{pmatrix}=1$	$(1010)\begin{pmatrix}1\\0\\0\\0\end{pmatrix}=1$
$(1010)\begin{pmatrix}1\\0\\1\\0\end{pmatrix}=0$	

Candidate 4 gives the solution. We note that visual studio gives the bits in reversed order, so 1010 → 0101 and this is our secret string.

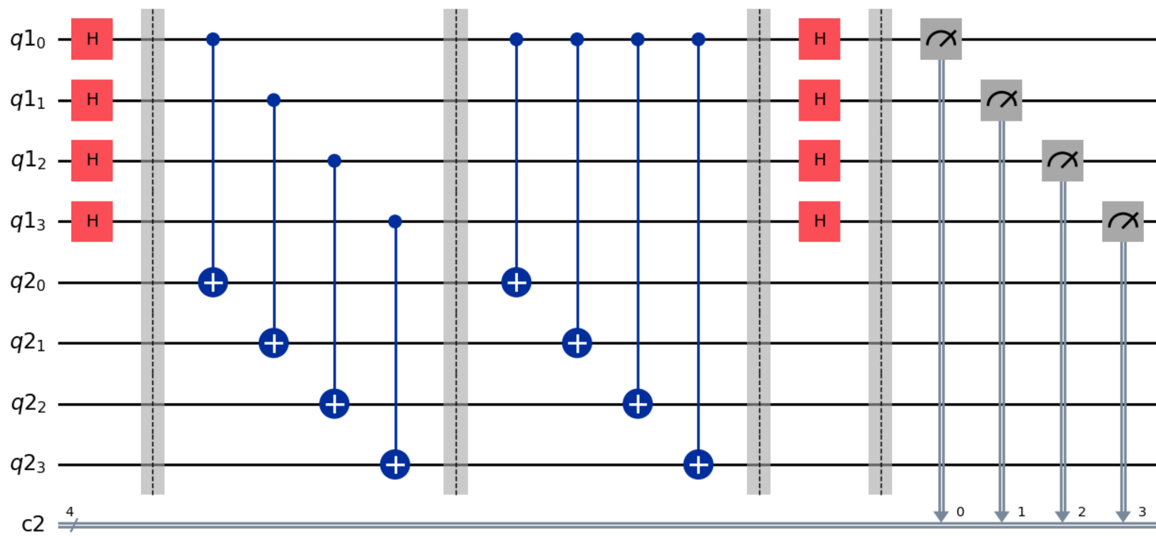
For your convenience the code used:

```
SECRET_WORD = "0101"
REGISTER_SIZE = len(SECRET_WORD)
firstBitPosition = 99

# ================================
# INIT
# ================================
q1 = QuantumRegister(REGISTER_SIZE,'q1')
q2 = QuantumRegister(REGISTER_SIZE,'q2')
c = ClassicalRegister (REGISTER_SIZE)
circuit = QuantumCircuit (q1,q2, c)

circuit.h(q1)
circuit.barrier()

# ================================
# ORACLE GATE
# ================================
circuit.cx(q1,q2)
circuit.barrier()

The circuit with the secret word c=0101
for i in range(len(SECRET_WORD)):
    if (SECRET_WORD[i] == "1") and (firstBitPosition == 99):
        firstBitPosition = i
        circuit.cx(q1[firstBitPosition], q2[i])

    elif (SECRET_WORD[i] == "1") and (firstBitPosition != 99):
        circuit.cx(q1[firstBitPosition], q2[i])
circuit.barrier()
# ================================
# END OF CIRCUIT
# ================================
circuit.h(q1)
circuit.barrier()
circuit.measure(q1,c)

# Calculating the circuit and printout of bx
qasm_sim = Aer.get_backend("qasm_simulator")
job = assemble(circuit,qasm_sim,shots=10)
result = qasm_sim.run(job).result()
counts = result.get_counts()
print(result)
for i in counts:
    print(i)
```

```
# Histogram of measurement results
all_states = [''.join(REGISTER_SIZE) for REGISTER_SIZE in product('01',
repeat=REGISTER_SIZE)]
for state in all_states:
    if state not in counts:
        counts[state] = 0
total_counts = sum(counts.values())
probabilites = {state: count / total_counts for state, count in
counts.items()}
plot_histogram(probabilites)

# prepare arrays for inner product with numpy
state_b = [list(element) for element in counts.keys()]
for i in range(len(state_b)):
    for j in range(len(state_b[i])):
        state_b[i][j] = int(state_b[i][j])
state = [list(element) for element in all_states]
for i in range(len(state)):
    for j in range(len(state[i])):
        state[i][j] = int(state[i][j])

def scalar(x,y_list):
    res_list = []
    print('scalar '+str(x)+' mit liste')
    for y in y_list:
        if (dot(x,y)%2 == 0):
            res_list.append(y)
    return res_list

# Remove vector [0,0,..,0] from state array
# because not possible for the secret string c
for el in state:
    if 1 not in el:
        state.remove(el)

for el in state_b:
    if 1 not in el:
        state_b.remove(el)

for i in state_b:
    if (len(state) <= 1):
        break

    newstate = scalar(i,state)
    print(newstate)
    print('\n')
    state = newstate
```

Example 3

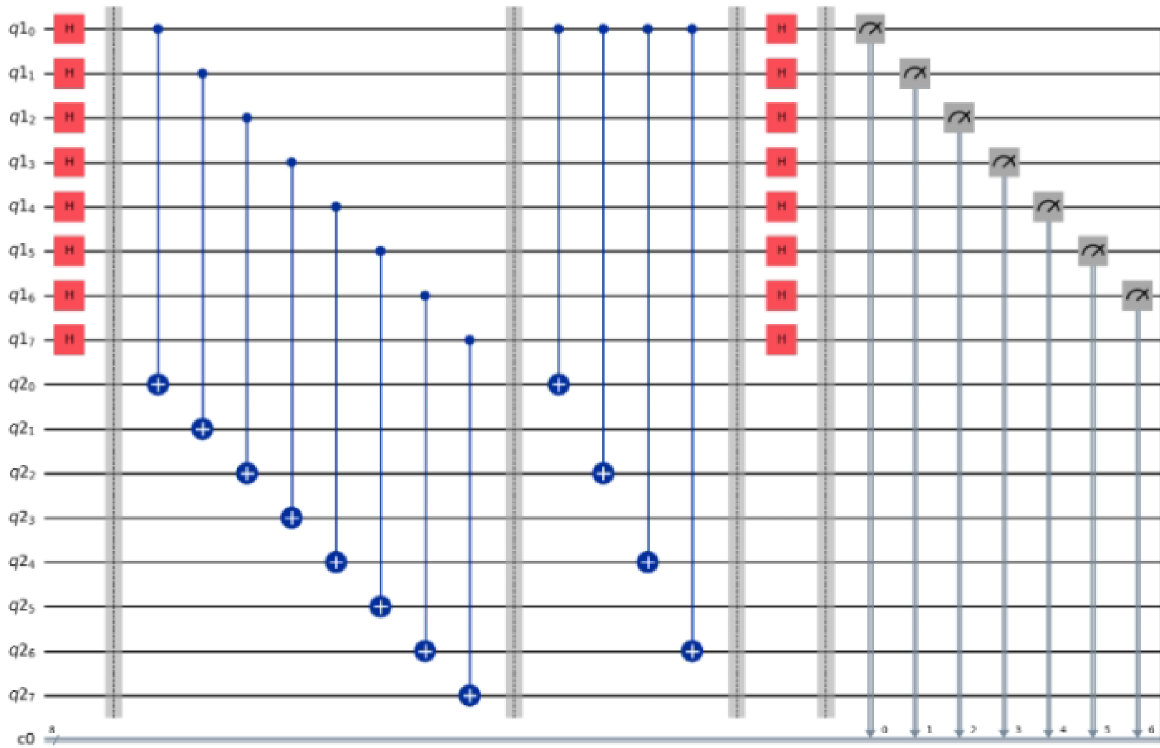
c = 1111




```
Result(backend_name='qasm_simulator', backend_version='0.14.1', qobj_id='59e7b77e-d9a7-457d-bc82-ed8a
0000
1001
0110
0011
1100
0101
scalar [1, 0, 0, 1] mit liste
[[0, 0, 1, 0], [0, 1, 0, 0], [0, 1, 1, 0], [1, 0, 0, 1], [1, 0, 1, 1], [1, 1, 0, 1], [1, 1, 1, 1]]


scalar [0, 1, 1, 0] mit liste
[[0, 1, 1, 0], [1, 0, 0, 1], [1, 1, 1, 1]]


scalar [0, 0, 1, 1] mit liste
[[1, 1, 1, 1]]
```

Example 4

c = 10101010




```
Result(backend_name='qasm_simulator', backend_version='0.14.1', qobj_id='48a216c1-c3ac-4cff-bc4f-bd014b251d1b', job_id='15e0d6fe-
01111111
00000101
00010100
01101001
11111111
01000100
11010101
01011111
11100011
10100010
scalar [0, 1, 1, 1, 1, 1, 1, 1] mit liste
[[0, 0, 0, 0, 0, 0, 1, 1], [0, 0, 0, 0, 0, 1, 0, 1], [0, 0, 0, 0, 0, 1, 1, 0], [0, 0, 0, 0, 1, 0, 0, 1], [0, 0, 0, 0, 1, 0, 1, 0]


scalar [0, 0, 0, 0, 0, 1, 0, 1] mit liste
[[0, 0, 0, 0, 0, 1, 0, 1], [0, 0, 0, 0, 1, 0, 1, 0], [0, 0, 0, 0, 1, 1, 1, 1], [0, 0, 0, 1, 0, 0, 1, 0], [0, 0, 0, 1, 0, 1, 1, 1]


scalar [0, 0, 0, 1, 0, 1, 0, 0] mit liste
[[0, 0, 0, 0, 1, 0, 1, 0], [0, 0, 0, 1, 0, 1, 1, 1], [0, 0, 0, 1, 1, 1, 0, 1], [0, 0, 1, 0, 0, 0, 1, 0], [0, 0, 1, 0, 1, 0, 0, 0]


scalar [0, 1, 1, 0, 1, 0, 0, 1] mit liste
[[0, 0, 0, 1, 1, 1, 0, 1], [0, 0, 1, 0, 1, 0, 0, 0], [0, 0, 1, 1, 0, 1, 0, 1], [0, 1, 0, 0, 1, 0, 0, 0], [0, 1, 0, 1, 0, 1, 0, 1]
...
scalar [0, 1, 0, 1, 1, 1, 1, 1] mit liste
[[0, 1, 0, 1, 0, 1, 0, 1]]
```

Comparison quantum-classic

Classical computing

1. Assumptions and preparatory work:
   - Suppose we have a function $f(x): \{0\ 1\}^n \rightarrow \{0\ 1\}^n$ with property $f(x) = f(y) \Leftrightarrow x \oplus y = s.$
   - Goal: Find the secret bit pattern s.
2. Proceeding:
   - The classical algorithm hast to find pairs of input values x and y that produce the same output: $f(x) = f(y)$.
   - This requires testing $2^{n-1} + 1$ pairs in the worst case.
3. Complexity:
   - This leads to an exponential time complexity, since the number of necessary comparisons grows exponentially with the number of bits n.

Comparison

Quantum run gives how many b_x are needed to form the scalar products.

Regular passes give the maximum number of bit strings needed to be checked for coherence.

n bits	Quantum run	Regular passes
	$n-1$	$2^{n-1}+1$
2	1	3
4	3	9
8	7	129
16	15	32.769
32	31	2.147.483.649
64	64	9.223.372.036.854.775.809
128	127	$1.7 \cdot 10^{38}$
256	255	$5.8 \cdot 10^{76}$
512	511	$6.7 \cdot 10^{153}$

The number of atoms in the universe is estimated to be between 10^{84} and 10^{89}.

I would like to thank my students Tobias Baumann, Niklas Birkler and Robin Faigle that made this paper possible.