Quantum Search algorithm

And Quantum Oracles

The problem

Consider a database of N element.

The goal is to find a certain element x in the database (can be repeated M times).

A classical computer requires $\mathcal{O}(N)$ operations to do so.

With a quantum version we can reduce the complexity drastically, up to $\mathcal{O}\left(\sqrt{N}\right)$.

Follow me and try yourself!

Quantum Oracles

Consider a classical binary function

$$f: \{0,1\}^n \rightarrow \{0,1\}^m$$

A **Quantum Oracle** is a unitary operator associated to *f* such that

$$U_f|x\rangle|q\rangle = |x\rangle|\mathrm{q} \oplus f(x)\rangle$$

The oracle depends on the system (on *f(x)*).

In the definition, *x* is the query register and *q* is the *ancilla* register (or answer register)

Grover Oracle

We want *f(x)* to be 1 if *x* is a solution to the search problem, 0 in any other case *(x* is the binary representation of the index).

Using the ancilla in state $|-\rangle$ it is possible to concatenate X and $M^{\lambda}X$ gates to achieve:

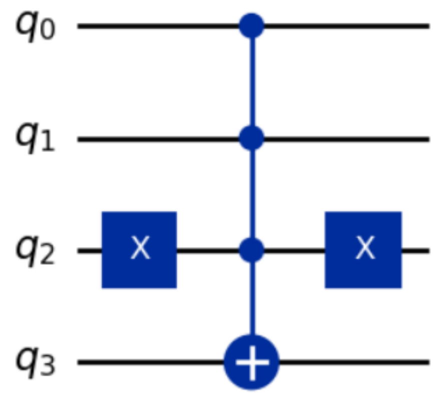
$$U_f|x\rangle|-\rangle = (-1)^{f(x)}|x\rangle|-\rangle$$

In this case, the solution will be marked with a global negative phase.

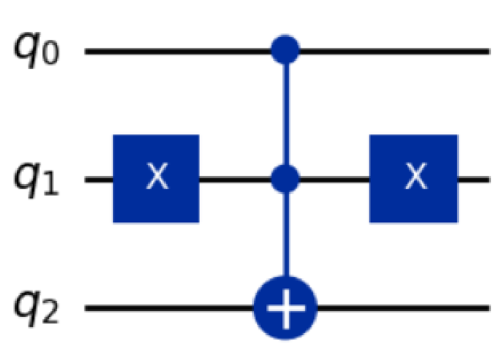
Grover Oracle - Implementation

```
def GroverOracleGate(marked_bitstrings: list[str]) -> Gate:
  if not marked_bitstrings:
        raise ValueError("marked_bitstrings cannot be empty")

  n = len(marked_bitstrings[0])
  if any(len(s) != n for s in marked_bitstrings):
    raise ValueError("All bitstring must have the same lenght")

  qc = QuantumCircuit(n+1, name=f"Oracle_M={len(marked_bitstrings)}")
  oracle_qubit = n

  for target in marked_bitstrings:
    for i, bit in enumerate(reversed(target)):
      if bit == '0':
        qc.x(i)

    qc.mcx(list(range(n)), oracle_qubit)

    for i, bit in enumerate(reversed(target)):
      if bit == '0':
        qc.x(i)

  return qc.to_gate()
```

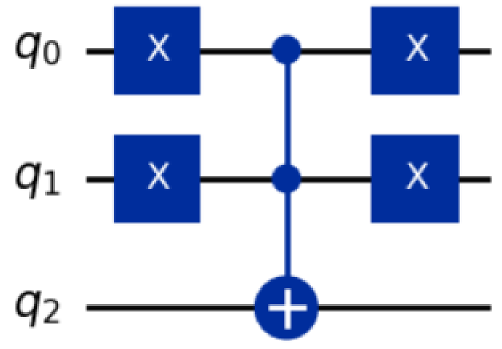
Grover Oracle for target 011



Grover Oracle for target 01

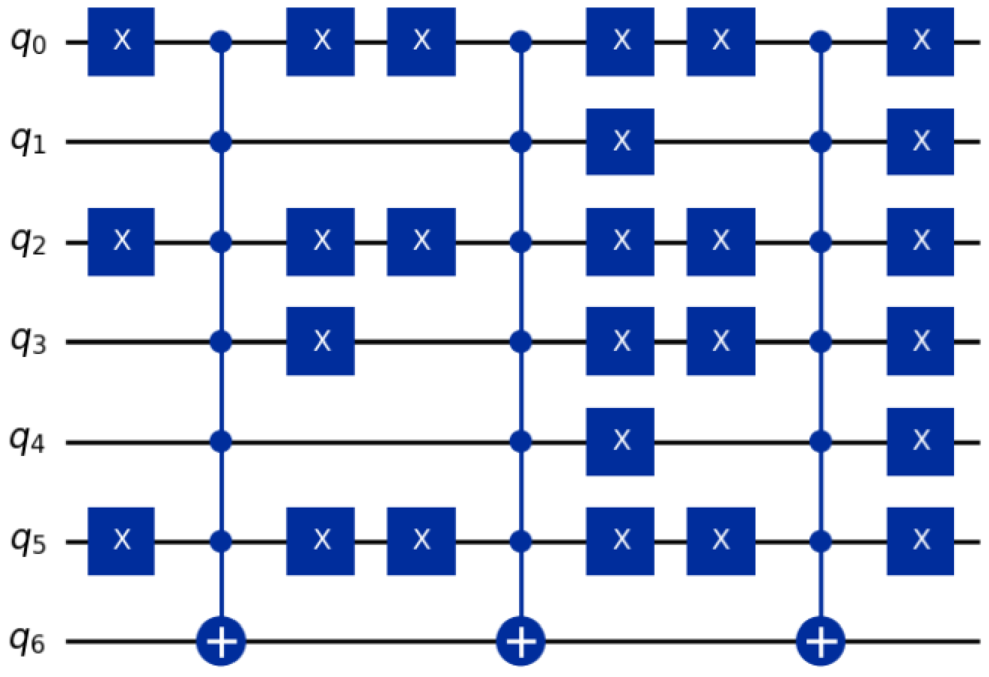


Grover Oracle for target 00

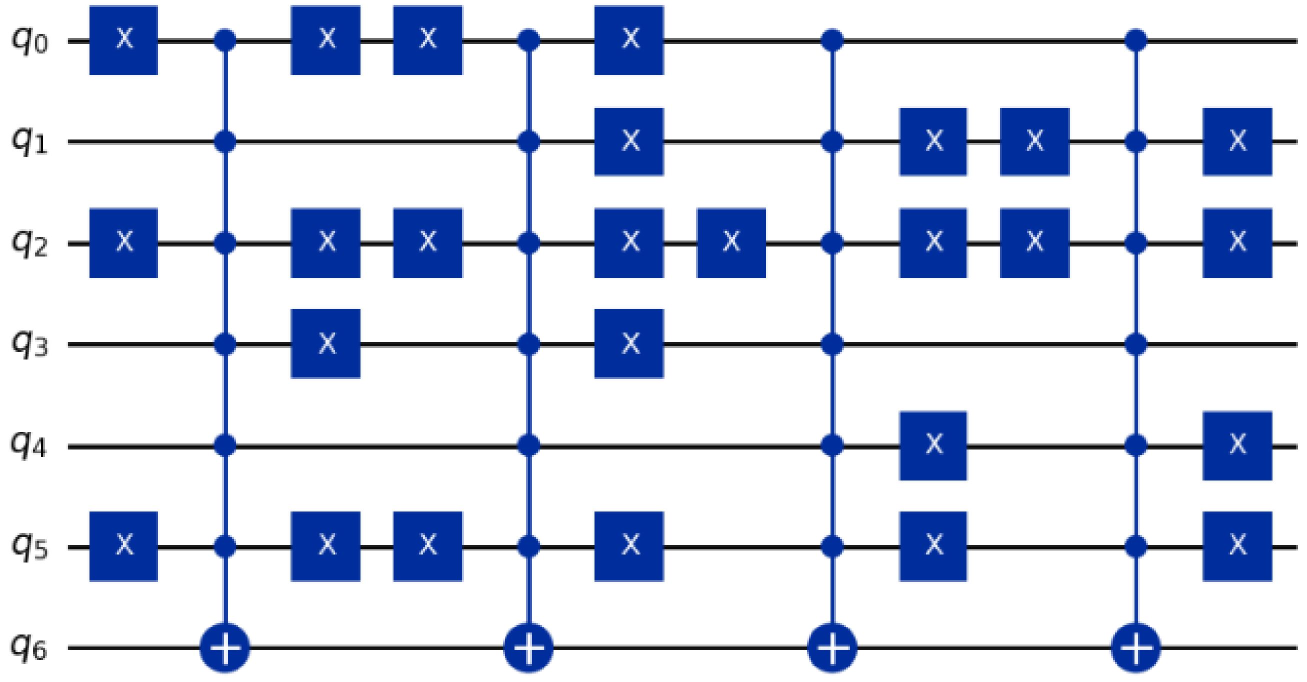


Oracle Gate – Examples with M > 1

Grover Oracle for targets ['011010', '010010', '000000']



Grover Oracle for targets ['011010', '010010', '111001', '001001']



Grover Diffuser

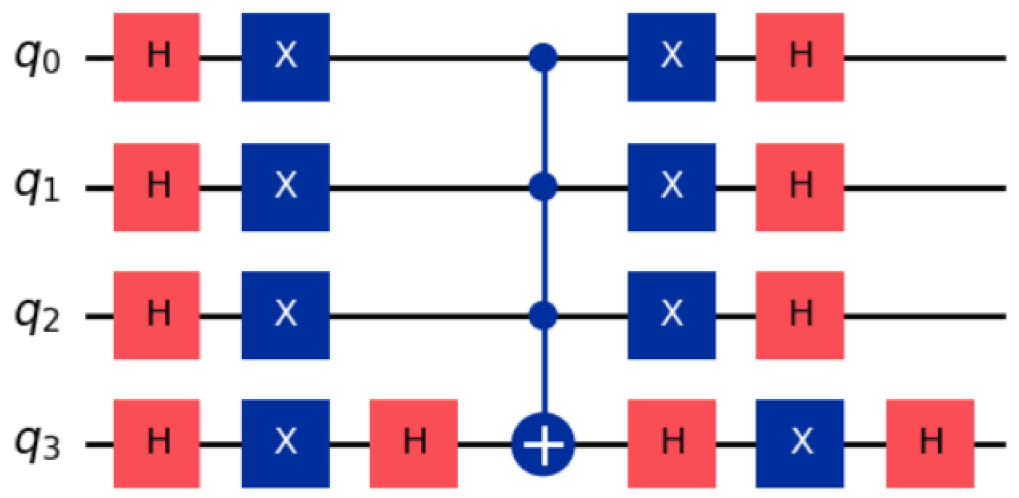
After the Oracle, a **Grover Diffuser gate** is implemented, characterized by the following steps on the *n* query qubits (practically it is analogue to $M^{\lambda}Z$):

1. Apply $H^{\otimes n}$
2. Conditional phase shift $|x\rangle \rightarrow -(-1)^{\delta_{x,0}}|x\rangle$
3. Apply $H^{\otimes n}$

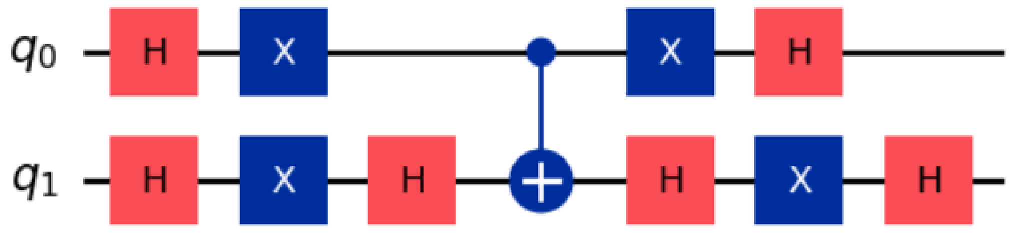
Grover Diffuser - Implementation

```python
def GroverGate(n: int) -> Gate:
    qc = QuantumCircuit(n, name='Grover')

    qc.h(range(n))
    qc.x(range(n))
    qc.h(n-1)
    qc.mcx(list(range(n-1)), n-1)
    qc.h(n-1)
    qc.x(range(n))
    qc.h(range(n))

    return qc.to_gate()
```

Grover Diffuser for n=4



Grover Diffuser for n=2

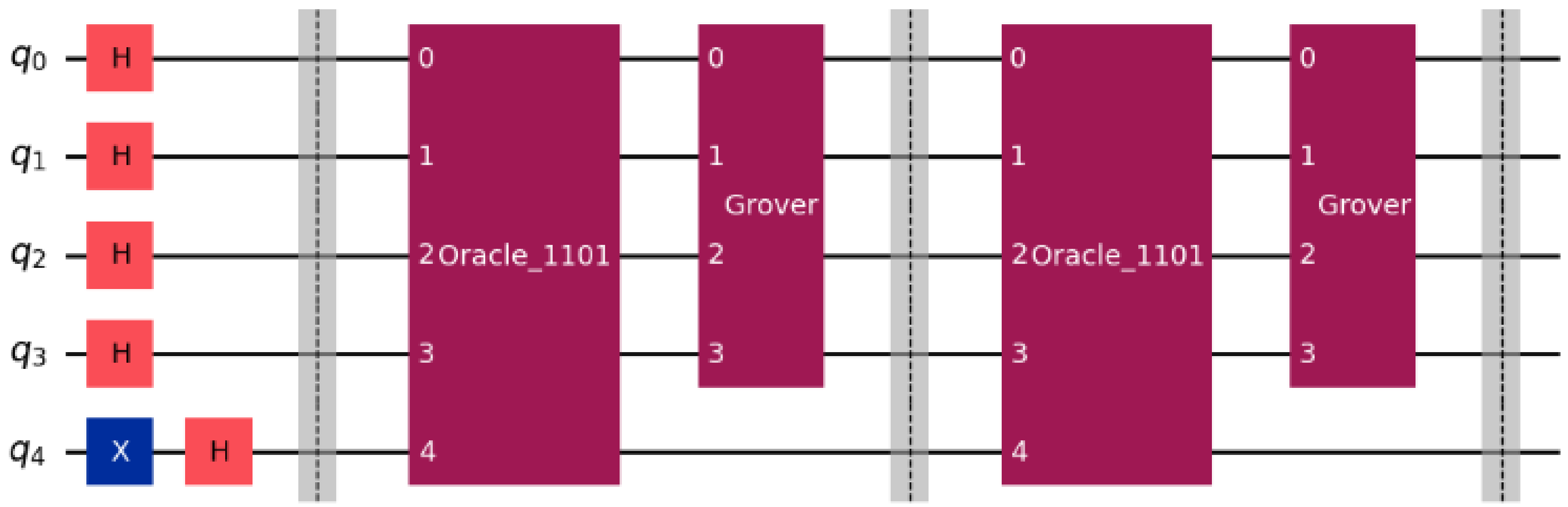


Grover Algorithm

Start with $|0\rangle^{\otimes n+1}$, apply X to the ancilla, then $\mathrm{H}^{\otimes n+1}$

The whole algorithm then consists in the concatenation of the Oracle and the Diffuser, performed for *k* times.

Grover circuit for n=4, target=1101, k=2



Grover Algorithm – Geometric interpretation

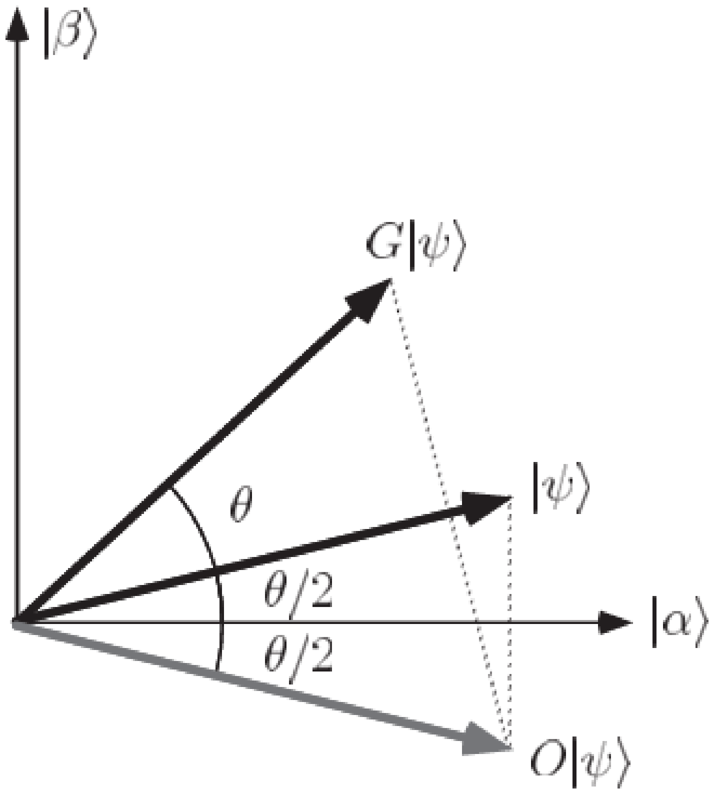
Apply the Grover gate can be seen as a 2D rotation of angle θ in the solution/non-solution plane.

$$|\alpha\rangle = \frac{1}{\sqrt{N-M}} \sum_{x=\text{not sol}} |x\rangle \qquad |\beta\rangle = \frac{1}{\sqrt{M}} \sum_{x=\text{sol}} |x\rangle$$

$$|\psi\rangle = \sqrt{\frac{N-M}{N}}|\alpha\rangle + \sqrt{\frac{M}{N}}|\beta\rangle \qquad \theta = 2 \cdot \arccos\sqrt{\frac{N-M}{M}}$$

$$G = (2|\psi\rangle\langle\psi| - \mathrm{I})U_f$$

$$G^k|\psi\rangle = cos\left(\frac{2k+1}{2}\theta\right)|\alpha\rangle + sin\left(\frac{2k+1}{2}\theta\right)|\beta\rangle$$

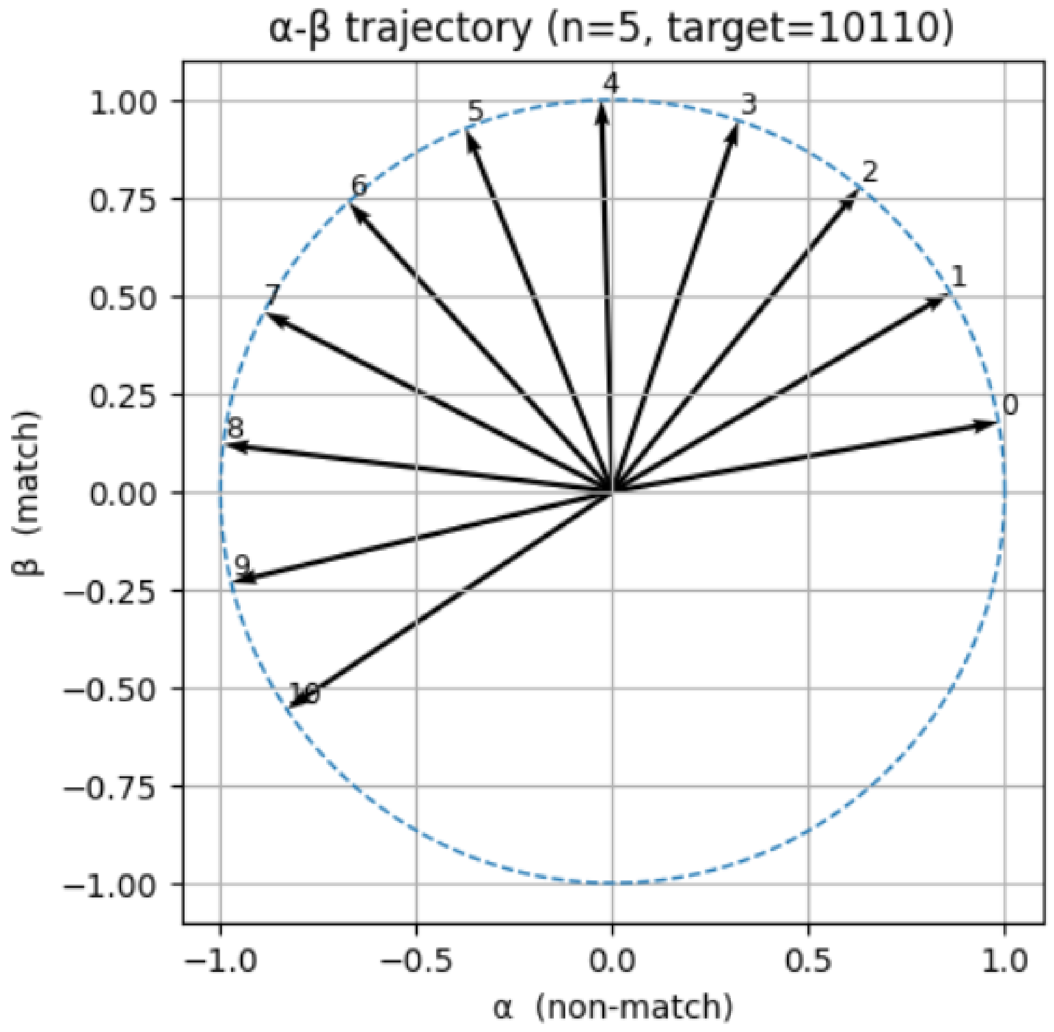


How many times?

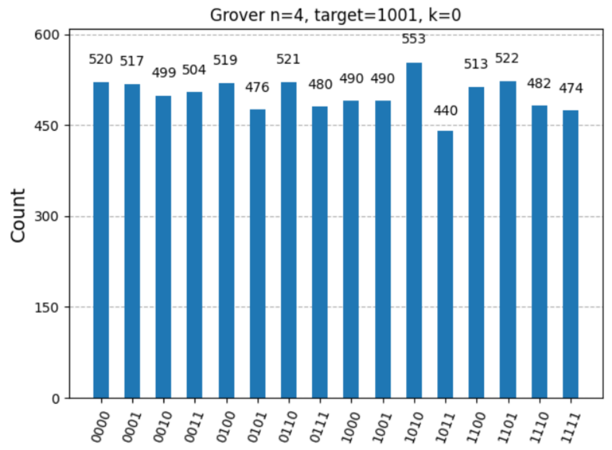
To maximize the success probability, we need $|\psi\rangle \approx |\beta\rangle$, meaning that the initial state must be rotated of $\arccos\sqrt{M/N}$ radians.

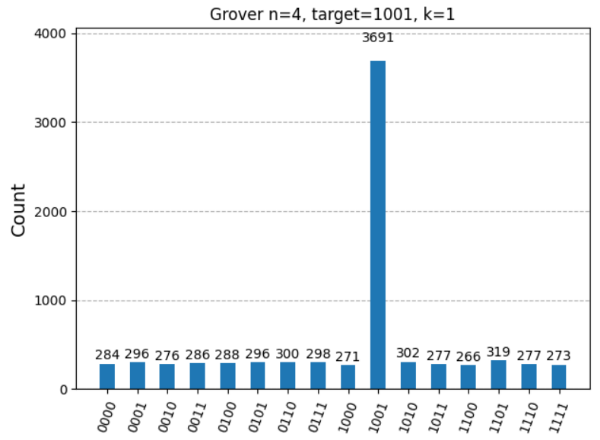
If each step rotates of an angle θ, the Grover gate must be applied *k* times:

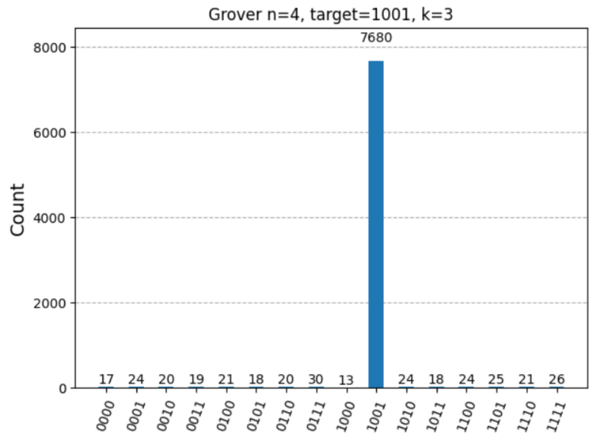
$$k = \mathrm{int}\left(\frac{\arccos\sqrt{\frac{M}{\mathrm{N}}}}{\theta}\right)$$

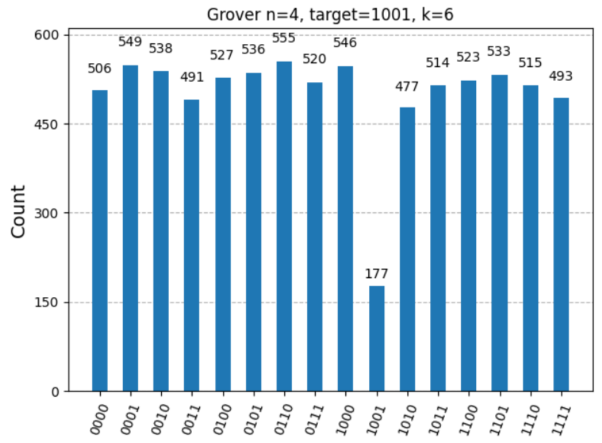


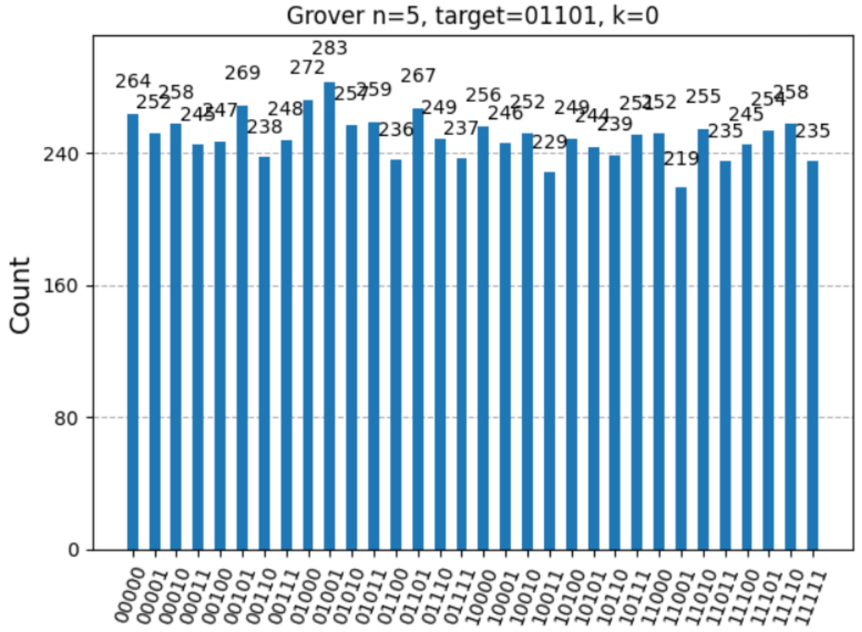
Results – correct picks VS *k*

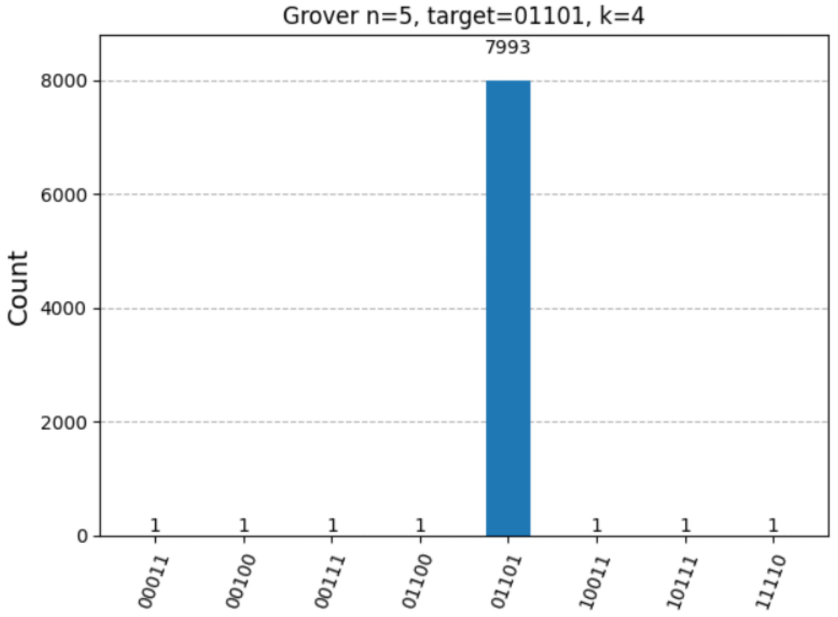


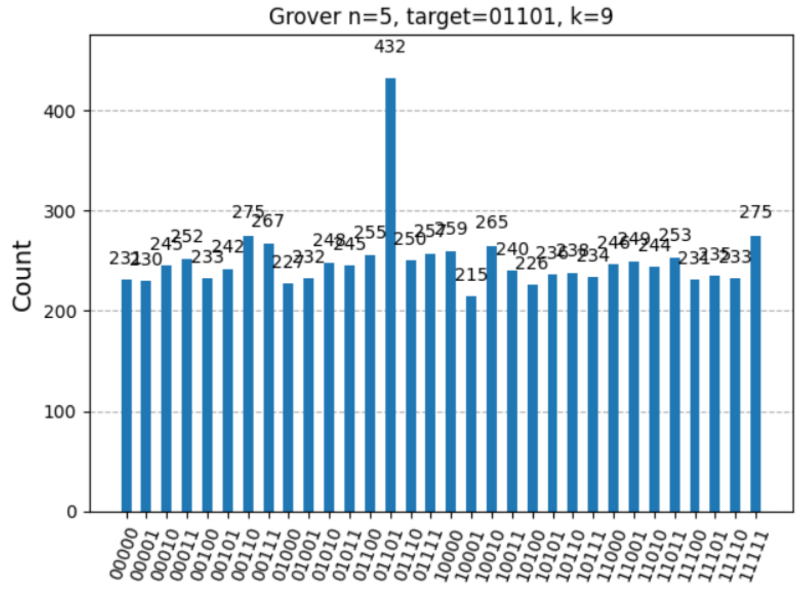




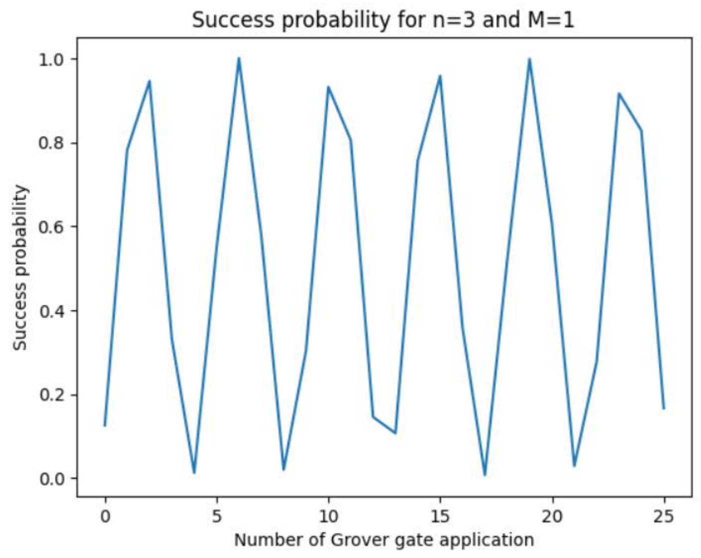


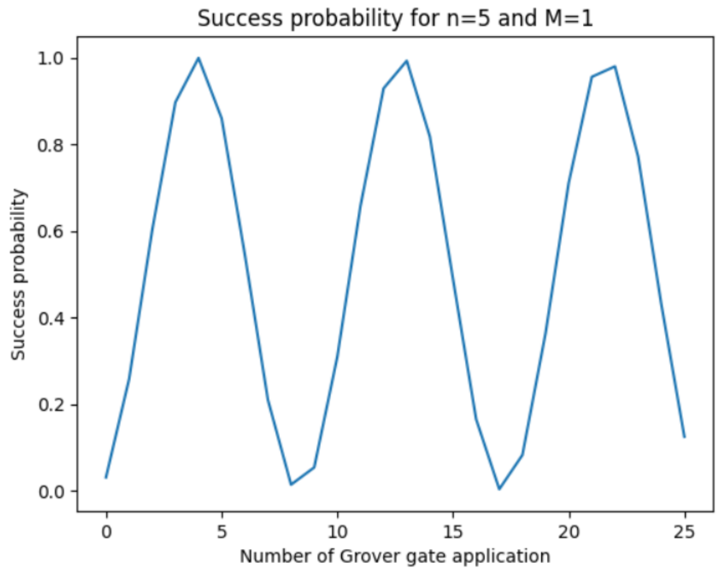


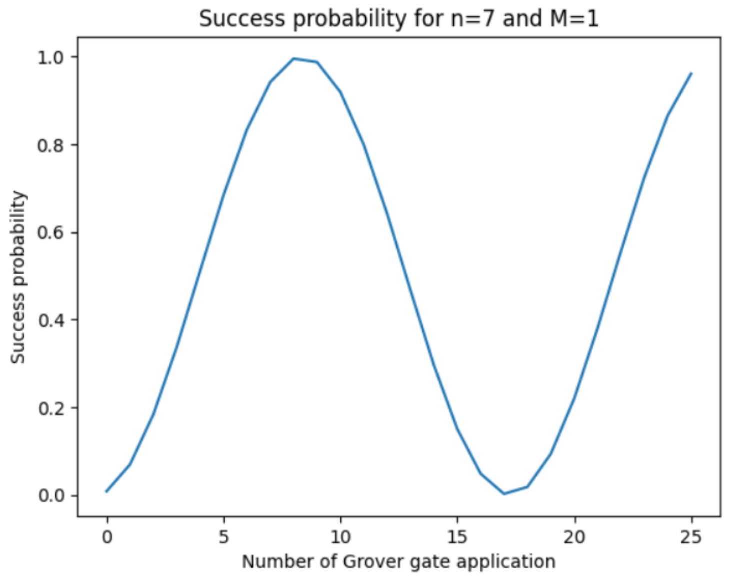


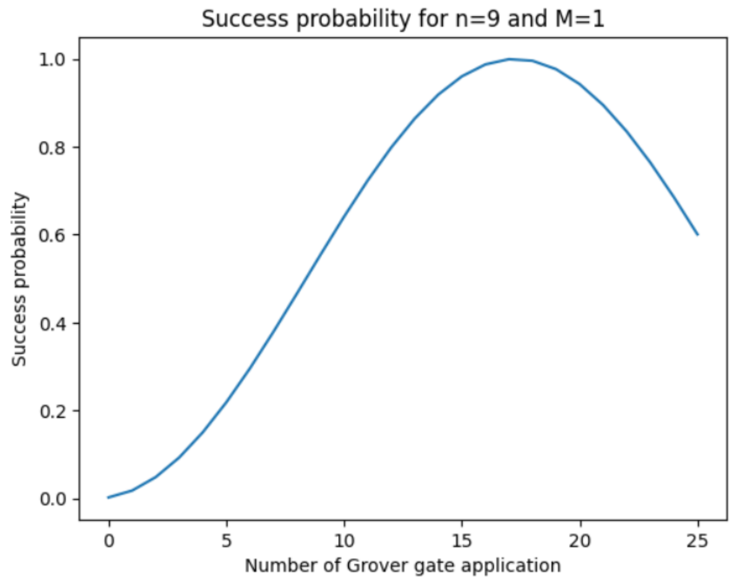


Results – success VS *k*

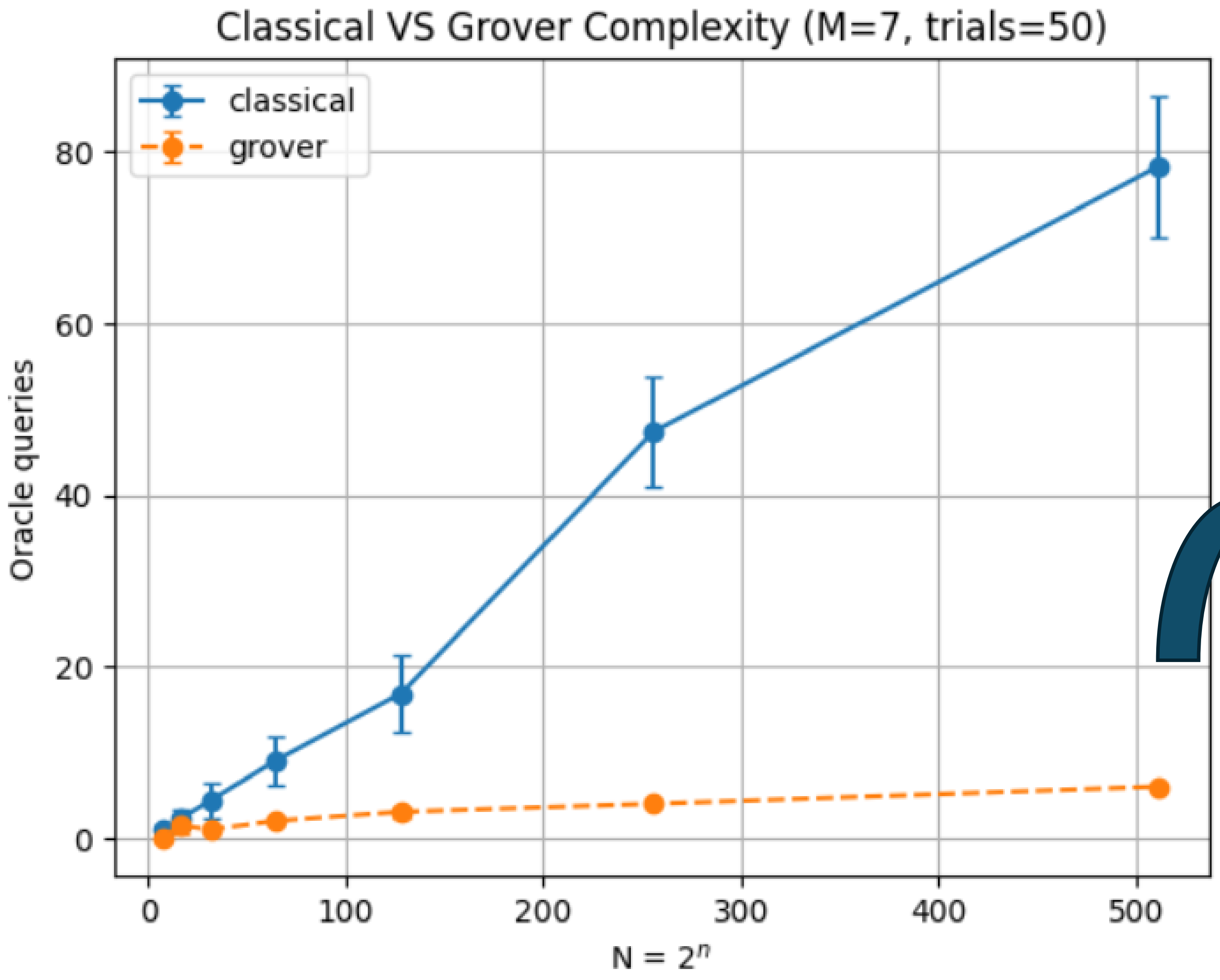


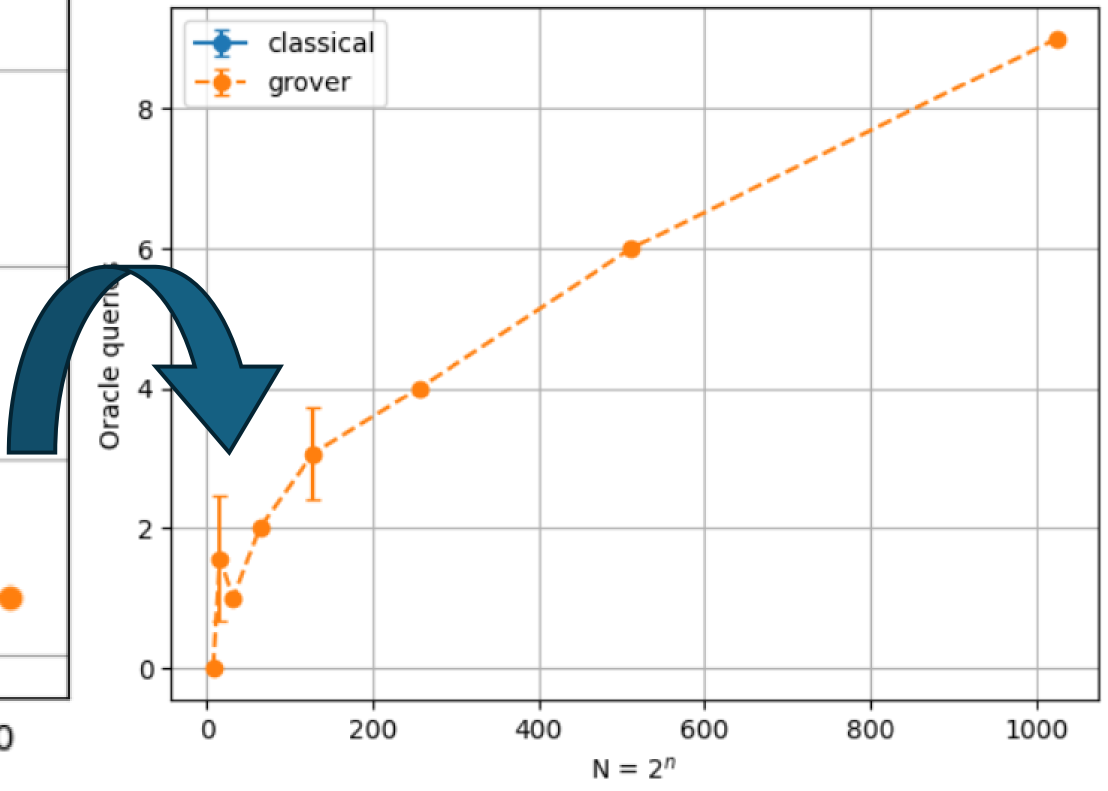






Results – complexity





Thanks for the attention!