

# Durr-Hoyer Algorithm Implementation

## Finding the Maximum using 4 Qubits

The Durr-Hoyer algorithm is a quantum algorithm for finding the minimum/maximum element in an unstructured database. It provides a quadratic speedup compared to classical algorithms.

This implementation uses:

- **4 qubits** for searching
- **Oracle:** X gates + Hadamard + MCX + Hadamard (to implement multi-controlled Z)
- **Diffusion Operator:** Hadamard  $\rightarrow$  X  $\rightarrow$  Multi-controlled Z  $\rightarrow$  X  $\rightarrow$  Hadamard

```
In [12]: from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, transpile
from qiskit_aer import AerSimulator
import numpy as np
from math import pi, sqrt
import matplotlib.pyplot as plt
```

## Step 1: Define the Oracle

The oracle uses X gates to mark the target state (maximum element). For this example, we mark state  $|1111\rangle$  (15 in decimal) as the maximum.

```
In [13]: def oracle(qc, qubits, target_state=15):
        """
        Oracle that marks the target state using X gates and multi-controlled Z gate.
        Multi-controlled Z is implemented as Hadamard - MCX (multi-controlled X) - Hada

        Args:
            qc: QuantumCircuit
            qubits: list of qubit indices
            target_state: the state to mark (default: 15 = |1111>)
        """
        # Convert target state to binary representation
        binary = format(target_state, f'0{len(qubits)}b')

        # Apply X gates to qubits where target state has 0
        for i, bit in enumerate(binary):
            qubit_index = len(qubits) - 1 - i
            if bit == '0':
                qc.x(qubits[qubit_index])

        # Apply multi-controlled Z gate
        if len(qubits) > 1:
            qc.h(qubits[-1])
```

```

        qc.mcx(qubits[:-1], qubits[-1])
        qc.h(qubits[-1])
    else:
        qc.z(qubits[0])

    # Undo the X gates
    for i, bit in enumerate(binary):
        qubit_index = len(qubits) - 1 - i
        if bit == '0':
            qc.x(qubits[qubit_index])

print("Oracle function defined.")

```

Oracle function defined.

## Step 2: Define the Diffusion Operator

The diffusion operator amplifies the marked state:

1. Apply Hadamard gates to all qubits
2. Apply X gates to all qubits
3. Apply multi-controlled Z gate
4. Apply X gates to all qubits
5. Apply Hadamard gates to all qubits

```

In [14]: def diffusion_operator(qc, qubits):
    """
    Diffusion operator for amplitude amplification.
    Structure: H → X → Multi-controlled Z → X → H

    Args:
        qc: QuantumCircuit
        qubits: list of qubit indices
    """
    # Step 1: Apply Hadamard gates
    for qubit in qubits:
        qc.h(qubit)

    # Step 2: Apply X gates
    for qubit in qubits:
        qc.x(qubit)

    # Step 3: Apply multi-controlled Z gate using H-MCX-H decomposition
    if len(qubits) > 1:
        qc.h(qubits[-1])
        qc.mcx(qubits[:-1], qubits[-1])
        qc.h(qubits[-1])
    else:
        qc.z(qubits[0])

    # Step 4: Apply X gates
    for qubit in qubits:
        qc.x(qubit)

```

```

# Step 5: Apply Hadamard gates
for qubit in qubits:
    qc.h(qubit)

print("Diffusion operator function defined.")

```

Diffusion operator function defined.

## Step 3: Build the Complete Durr-Hoyer Circuit

```

In [15]: def durr_hoyer_circuit(num_qubits=4, target_state=15, num_iterations=None):
    """
    Construct the complete Durr-Hoyer circuit.

    Args:
        num_qubits: number of qubits (default: 4)
        target_state: the maximum element to find
        num_iterations: number of Grover iterations (None = optimal)

    Returns:
        QuantumCircuit with measurements
    """
    # Create quantum and classical registers
    qr = QuantumRegister(num_qubits, 'q')
    cr = ClassicalRegister(num_qubits, 'c')
    qc = QuantumCircuit(qr, cr)

    # Step 1: Initialize with Hadamard gates (superposition)
    qc.h(qr)
    qc.barrier()

    # Calculate optimal number of iterations if not specified
    if num_iterations is None:
        num_iterations = int(np.pi / 4 * np.sqrt(2**num_qubits))

    qc.barrier()

    # Step 2: Apply Durr-Hoyer iterations (oracle + diffusion)
    for i in range(num_iterations):
        # Apply oracle to mark target state
        oracle(qc, list(qr), target_state)
        qc.barrier()

        # Apply diffusion operator for amplitude amplification
        diffusion_operator(qc, list(qr))
        qc.barrier()

    # Step 3: Measure all qubits
    qc.measure(qr, cr)

    return qc

# Build the circuit
num_qubits = 4

```

```
target_state = 15 # |1111> - maximum value for 4 qubits

circuit = durr_hoyer_circuit(num_qubits=num_qubits, target_state=target_state, num_i
print(f"Durr-Hoyer circuit created with {num_qubits} qubits")
print(f"Target state (maximum): {target_state} ({format(target_state, '04b')}))")
```

Durr-Hoyer circuit created with 4 qubits

Target state (maximum): 15 (|1111>)

## Step 4: Display the Circuit

```
In [16]: # Display the circuit
print(circuit)
```



## Step 5: Execute the Circuit

```
In [17]: # Data array (replace or modify as needed)
data = [5,12,27,8,2,19,10,33,7,14,22,6,3,18,1,30]

# Validate data length is a power of two and determine number of qubits
N = len(data)
if N & (N - 1) != 0:
    raise ValueError("Data length must be a power of two for this simple implementa
num_qubits = int(np.log2(N))

# Determine target index (index of the maximum element) and value
target_value = max(data)
target_state = data.index(target_value)

# Build the circuit using the computed num_qubits and target_state
circuit = durr_hoyer_circuit(num_qubits=num_qubits, target_state=target_state, num_

print(f"Durr-Hoyer circuit created with {num_qubits} qubits")
print(f"Target (maximum): index {target_state} -> value {target_value} ({format(ta

# Use AER simulator
simulator = AerSimulator()

# Transpile & execute
transpiled_circuit = transpile(circuit, simulator)
job = simulator.run(transpiled_circuit, shots=1000)
result = job.result()

# Get measurement results
counts = result.get_counts(transpiled_circuit)

print("\nMeasurement Results:")
print("="*50)

# Print counts with index and corresponding value from the array
sorted_counts_print = sorted(counts.items(), key=lambda x: int(x[0], 2))
for b, c in sorted_counts_print:
    idx = int(b, 2)
    val = data[idx]
    pct = (c / 1000) * 100
    print(f"State |{b}> (index {idx}, value {val}): {c} shots ({pct:.1f}%)"
```

Durr-Hoyer circuit created with 4 qubits  
 Target (maximum): index 7 -> value 33 ( $|0111\rangle$ )

Measurement Results:

```
=====
State |0000> (index 0, value 5): 1 shots (0.1%)
State |0001> (index 1, value 12): 5 shots (0.5%)
State |0010> (index 2, value 27): 4 shots (0.4%)
State |0011> (index 3, value 8): 3 shots (0.3%)
State |0100> (index 4, value 2): 1 shots (0.1%)
State |0101> (index 5, value 19): 3 shots (0.3%)
State |0110> (index 6, value 10): 3 shots (0.3%)
State |0111> (index 7, value 33): 957 shots (95.7%)
State |1000> (index 8, value 7): 3 shots (0.3%)
State |1001> (index 9, value 14): 2 shots (0.2%)
State |1010> (index 10, value 22): 5 shots (0.5%)
State |1011> (index 11, value 6): 1 shots (0.1%)
State |1100> (index 12, value 3): 1 shots (0.1%)
State |1101> (index 13, value 18): 5 shots (0.5%)
State |1110> (index 14, value 1): 4 shots (0.4%)
State |1111> (index 15, value 30): 2 shots (0.2%)
```

## Step 6: Visualize the Results

```
In [18]: # Sort counts for plotting
sorted_counts = sorted(counts.items(), key=lambda x: int(x[0], 2))
bitstrings = [item[0] for item in sorted_counts]
values = [item[1] for item in sorted_counts]

# Create x-tick labels as "index:value"
index_labels = [f"{int(bs,2)}:{data[int(bs,2)]}" for bs in bitstrings]

# Create the bar plot
plt.figure(figsize=(14, 6))
bars = plt.bar(range(len(bitstrings)), values, color='steelblue', edgecolor='navy')

# Highlight the target state
target_bit = format(target_state, f'0{num_qubits}b')
if target_bit in bitstrings:
    target_index = bitstrings.index(target_bit)
    bars[target_index].set_color('green')
    bars[target_index].set_edgecolor('darkgreen')

plt.xlabel('State (index:value)', fontsize=12)
plt.ylabel('Measurement Counts', fontsize=12)
plt.title('Durr-Hoyer Algorithm: Finding Maximum (index:value)', fontsize=14, fontw
plt.xticks(range(len(bitstrings)), index_labels, rotation=45)
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()

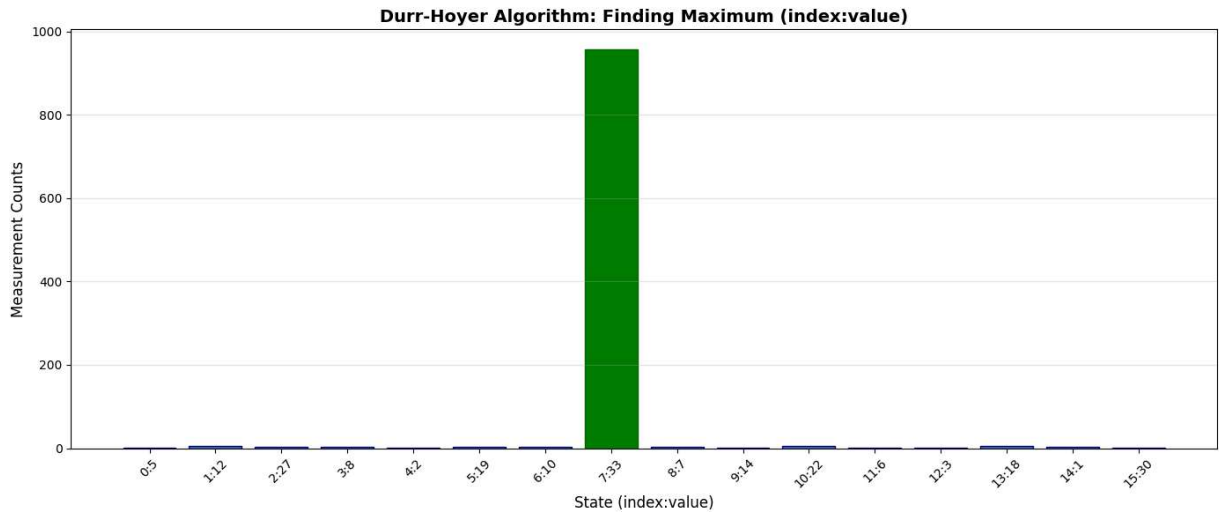
# Print summary
max_state = max(counts, key=counts.get)
max_index = int(max_state, 2)
max_value = data[max_index]
```

```

max_count = counts[max_state]

print("\n" + "="*50)
print("ALGORITHM SUMMARY")
print("="*50)
print(f"Target Maximum: index {target_state} -> value {target_value} ({format(target_value, 'd')})")
print(f"Most Probable Result: index {max_index} -> value {max_value} ({format(max_value, 'd')})")
print(f"Success Rate: {(max_count/1000)*100:.1f}%")
print(f"Algorithm correctly identified the maximum: {max_index == target_state}")

```



```

=====
ALGORITHM SUMMARY
=====
Target Maximum: index 7 -> value 33 (|0111>)
Most Probable Result: index 7 -> value 33 (|0111>)
Success Rate: 95.7%
Algorithm correctly identified the maximum: True

```

## Step 7: Circuit Depth and Statistics

```

In [19]: print(f"\nCircuit Statistics:")
print(f"="*50)
print(f"Number of Qubits: {circuit.num_qubits}")
print(f"Circuit Depth: {circuit.depth()}")
print(f"Number of Instructions: {len(circuit)}")
print(f"Number of Barriers: {circuit.count_ops().get('barrier', 0)}")

```

Circuit Statistics:

```

=====
Number of Qubits: 4
Circuit Depth: 38
Number of Instructions: 88
Number of Barriers: 8

```