

PRD 25: Quantum Algorithms and Computational Complexity

Pure Thought AI Challenge 25

Pure Thought AI Challenges Project

January 18, 2026

Abstract

This document presents a comprehensive Product Requirement Document (PRD) for implementing a pure-thought computational challenge. The problem can be tackled using only symbolic mathematics, exact arithmetic, and fresh code—no experimental data or materials databases required until final verification. All results must be accompanied by machine-checkable certificates.

Contents

Domain: Quantum Information Computer Science

Timeline: 6-9 months

Difficulty: High

Prerequisites: Quantum mechanics, linear algebra, complexity theory, graph theory, optimization

0.1 1. Problem Statement

0.1.1 Scientific Context

Quantum computing harnesses the principles of quantum mechanics—superposition, entanglement, and interference—to solve computational problems more efficiently than classical computers. The foundational quantum algorithms, **Grover's search** (1996) and **Shor's factoring** (1994), demonstrated provable quantum speedups over classical algorithms, launching the modern era of quantum information science. Grover's algorithm finds a marked item in an unsorted database of N elements in $O(N)$ queries versus $O(N)$ classically—a quadratic speedup. Shor's algorithm factors integers in polynomial time, threatening RSA cryptography and providing an exponential speedup over the best known classical algorithms.

Quantum walks generalize classical random walks to the quantum setting, providing a powerful framework for designing quantum algorithms. The coined quantum walk uses a "coin" Hilbert space to determine transition directions, while continuous-time quantum walks evolve via unitary operators $\exp(-iHt)$ where H encodes the graph structure. Quantum walks achieve quadratic speedups for problems like element distinctness (Ambainis, 2007) and exponential speedups for certain graph traversal problems (Childs et al., 2003). The **HHL algorithm** (Harrow-Hassidim-Lloyd, 2009) solves linear systems $Ax = b$ in time $O(\log N \text{ poly}())$ where ϵ is the condition number, exponentially faster than classical $O(N)$ algorithms—though caveats apply regarding state preparation and readout.

Variational quantum algorithms, including the **Quantum Approximate Optimization Algorithm (QAOA)** (Farhi et al., 2014) and **Variational Quantum Eigensolver (VQE)**, leverage hybrid quantum-classical optimization to solve combinatorial problems and find ground states of quantum systems. QAOA applies alternating unitary layers controlled by classical optimization of parameters, seeking approximate solutions to NP-hard problems like MaxCut and Max-SAT. While rigorous performance guarantees remain elusive, QAOA shows promise for near-term noisy intermediate-scale quantum (NISQ) devices.

0.1.2 Core Question

Given the quantum circuit model and complexity-theoretic framework:

- Implement canonical quantum algorithms: Grover search, quantum walks, HHL, QAOA
- Analyze query complexity and prove optimality (via polynomial method, adversary bounds)
- Construct oracle separations proving $BQP \neq BPP$ (e.g., Recursive Fourier Sampling)
- Benchmark quantum advantage: when does quantum outperform classical for specific problems?
- Generate certificates: success probabilities, query counts, complexity lower bounds

0.1.3 Why This Matters

- **Cryptographic Impact:** Shor's algorithm threatens RSA, ECC; post-quantum cryptography urgently needed
- **Optimization:** QAOA and VQE promise near-term applications in logistics, drug discovery, materials science
- **Complexity Theory:** Quantum computing provides new tools to understand P vs NP, BQP vs BPP
- **Fundamental Physics:** Computational complexity reflects fundamental limits on information processing in nature
- **Database Search:** Grover's algorithm offers provable speedup for unstructured search, applicable to SAT solving, collision finding

0.1.4 Pure Thought Advantages

- **Exact Simulation:** Small qubit systems (20 qubits) can be simulated exactly using linear algebra
 - **Query Complexity:** Lower bounds proven rigorously via polynomial method, adversary method
 - **Oracle Separations:** BQP ⊈ BPP proven via explicit oracle constructions (no real-world assumptions)
 - **Certificate-Based:** All query complexities, success probabilities, and optimality claims are mathematically provable
 - **Benchmarking:** Compare quantum vs classical on identical problems without hardware noise
-

0.2 2. Mathematical Formulation

0.2.1 Quantum Circuit Model

A **quantum circuit** on n qubits operates on the Hilbert space $H = (\mathbb{C}^2)^n$ with basis states $|x\rangle$ for $x \in \{0, 1\}^n$. Quantum gates are

- **Hadamard:** $H = (1/2)[[1, 1], [1, -1]]$
- **Pauli X, Y, Z:** $x = [[0, 1], [1, 0]], y = [[0, -i], [i, 0]], z = [[1, 0], [0, -1]]$
- **CNOT:** Controlled-NOT flipping target qubit if control is $|1\rangle$
- **Phase gate:** $R_{\phi} = [[1, 0], [0, e^{i\phi}]]$

A quantum algorithm is a sequence of gates followed by measurement in the computational basis, yielding outcome x with probability $|\langle x | \rangle|^2$.

0.2.2 Complexity Classes

- **BPP (Bounded-error Probabilistic Polynomial)**: Classical randomized algorithms with error probability $1/3$
- **BQP (Bounded-error Quantum Polynomial)**: Quantum circuits with $\text{poly}(n)$ gates, error $1/3$
- **NP**: Problems with polynomial-time verifiable certificates
- **P**: Polynomial-time deterministic algorithms

Known relations: P ⊂ BPP ⊂ BQP, P ⊂ NP. **Open:** BQP vs NP (neither contains the other is proven), BPP vs BQP (strongly believed BPP ⊂ BQP).

0.2.3 Query Complexity

Query complexity measures the number of oracle queries $f: \{0,1\}^n \rightarrow \{0,1\}$ required to solve a problem. For Grover search (find x with $f(x) = 1$):

- **Classical lower bound**: $\Omega(\sqrt{N})$ queries (must check $N/2$ items on average)
- **Quantum lower bound**: $\Omega(\sqrt{N})$ queries (Bennett et al., 1997, via hybrid argument)
- **Grover optimal**: $\Theta(\sqrt{N})$ queries, matching lower bound

0.2.4 Grover's Algorithm

Grover operator: $G = (2|0\rangle\langle 0| - I)\otimes O$ where O is the oracle (phase flip on solution) and $|0\rangle = H^n|0\rangle$ is uniform superposition.

Geometric interpretation: G rotates state vector toward $|x\rangle$ in $2D$ subspace spanned by $|0\rangle$ and $|x\rangle$. After k iterations, amplitude of $|x^*\rangle$ is $\sin((2k+1)\pi/N)$ where $\sin = 1/\sqrt{N}$.

Optimal iterations: $k^* = \lceil (N/4) \rceil$ gives success probability $1 - 1/N$.

0.2.5 Quantum Walk Model

A **coined quantum walk** on graph $G = (V,E)$ uses Hilbert space $H_{\text{coin}} \otimes H_{\text{position}}$ where $\dim(H_{\text{coin}}) = \text{maxdegree}(G)$.

Evolution: $U = S(C \otimes I)$ where C is coin operator (often Grover diffusion), S is shift operator moving particle along edges.

Hitting time: Expected time to reach target node. Quantum walks achieve quadratic speedup for many graphs (hypercube, complete graph).

0.2.6 Certificate Specification

A **quantum algorithm certificate** must contain:

- **Circuit description**: Gate sequence, qubit count, depth
- **Success probability**: $P(\text{correct output}) \geq 1 - \epsilon$ with $\epsilon \leq 1/3$
- **Query complexity**: Number of oracle calls, comparison to classical
- **Lower bound proof**: Adversary method, polynomial method, or hybrid argument
- **Complexity class**: BQP, BPP, or other; oracle separation if applicable

- **Numerical simulation:** For 20 qubits, exact amplitudes at each step

0.3 3. Implementation Approach

This is a 6-phase project spanning 6-9 months, implementing canonical quantum algorithms with complexity analysis.

0.3.1 Phase 1: Grover's Algorithm (Months 1-2)

Objective: Implement Grover search, verify $O(N)$ query complexity, prove optimality.

```

1 import numpy as np
2 from typing import Callable, Dict, List
3 from dataclasses import dataclass
4
5 def hadamard_n(n: int) -> np.ndarray:
6     """
7         n-qubit Hadamard gate:  $H^{\otimes n}$ .
8
9     Returns:  $2^n \times 2^n$  unitary matrix.
10    """
11    H1 = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
12    H_n = H1
13    for _ in range(n - 1):
14        H_n = np.kron(H_n, H1)
15    return H_n
16
17 def oracle_matrix(marked_items: List[int], N: int) -> np.ndarray:
18     """
19         Oracle  $O$  that flips phase of marked items:  $|x\rangle = (-1)^{f(x)}|x\rangle$ .
20
21     Args:
22         marked_items: List of indices  $x$  with  $f(x) = 1$ 
23         N: Total number of items ( $N = 2^n$ )
24
25     Returns:  $N \times N$  diagonal matrix with  $-1$  at marked positions.
26    """
27    O = np.eye(N)
28    for x in marked_items:
29        O[x, x] = -1
30    return O
31
32 def grover_operator(oracle: np.ndarray, n: int) -> np.ndarray:
33     """
34         Grover diffusion operator  $G = (2|I - O\rangle\langle I - O|)$ .
35
36          $|0\rangle = H^{\otimes n}|0\rangle = (1/\sqrt{N})\sum_x|x\rangle$  (uniform superposition)
37
38     Returns: Grover operator  $G$ .
39    """
40    N = 2**n
41    H = hadamard_n(n)
42
43    #  $|0\rangle = H|0\rangle$ 
```

```

44     psi = np.zeros(N)
45     psi[0] = 1.0
46     psi = H @ psi
47
48     # Diffusion operator: D = 2| - I
49     D = 2 * np.outer(psi, psi) - np.eye(N)
50
51     # Grover operator: G = D 0
52     G = D @ oracle
53
54     return G
55
56 def grover_search(marked_items: List[int], n: int, verbose: bool =
57     False) -> Dict:
58     """
59     Grover's algorithm: find marked item in O( N ) queries.
60
61     Args:
62         marked_items: List of marked indices (assumed |marked_items| =
63             1 for simplicity)
64         n: Number of qubits (N = 2^n items)
65         verbose: Print iteration details
66
67     Returns: Result dictionary with final state, measurement outcome,
68             success probability.
69     """
70     N = 2**n
71     M = len(marked_items) # Number of solutions
72
73     # Optimal number of iterations
74     theta = np.arcsin(np.sqrt(M / N))
75     k_optimal = int(np.pi / (4 * theta)) if theta > 0 else 0
76
77     # Initial state: uniform superposition
78     H = hadamard_n(n)
79     psi = np.zeros(N)
80     psi[0] = 1.0
81     psi = H @ psi
82
83     # Construct Grover operator
84     O = oracle_matrix(marked_items, N)
85     G = grover_operator(O, n)
86
87     # Apply G^k
88     for k in range(k_optimal):
89         psi = G @ psi
90         if verbose:
91             prob_marked = sum(abs(psi[x])**2 for x in marked_items)
92             print(f"Iteration {k+1}: P(marked) = {prob_marked:.6f}")
93
94     # Measure
95     probabilities = np.abs(psi)**2
96     result = int(np.argmax(probabilities))
97
98     success_prob = sum(probabilities[x] for x in marked_items)

```

```

97     return {
98         'final_state': psi,
99         'measurement_outcome': result,
100        'success_probability': success_prob,
101        'iterations': k_optimal,
102        'oracle_queries': k_optimal,
103        'correct': result in marked_items
104    }
105
106 def verify_grover_optimality(n_range: range) -> Dict:
107     """
108     Verify Grover's N scaling by running for different N.
109
110     Returns: Dictionary mapping N to average queries.
111     """
112     results = {}
113
114     for n in n_range:
115         N = 2**n
116         # Single marked item at random position
117         marked = [np.random.randint(0, N)]
118
119         result = grover_search(marked, n)
120         queries = result['oracle_queries']
121
122         results[N] = {
123             'queries': queries,
124             'sqrt_N': np.sqrt(N),
125             'ratio': queries / np.sqrt(N)
126         }
127
128         print(f"N={N:5d}: {queries:4d} queries,   N ={np.sqrt(N):7.2f}, "
129               f"ratio={queries/np.sqrt(N):.4f}")
130
131     return results
132
133 # Example usage
134 if __name__ == "__main__":
135     # Simple example: N=16, marked item at index 7
136     n = 4
137     marked = [7]
138
139     result = grover_search(marked, n, verbose=True)
140
141     print(f"\nGrover's Algorithm Result:")
142     print(f"  Marked item: {marked[0]}")
143     print(f"  Found: {result['measurement_outcome']}")
144     print(f"  Success probability: {result['success_probability']:.6f}")
145     print(f"  Oracle queries: {result['oracle_queries']}")
146     print(f"  Correct: {result['correct']}")
147
148     # Verify scaling
149     print(f"\nVerifying N scaling:")
150     verify_grover_optimality(range(4, 10))

```

0.3.2 Phase 2: Quantum Walks (Months 2-4)

Objective: Implement coined and continuous-time quantum walks, analyze hitting times.

```

1 import networkx as nx
2 from scipy.linalg import expm
3
4 def grover_diffusion_coin(d: int) -> np.ndarray:
5     """
6         Grover diffusion coin: C = 2| - I where | = (1/ d ) _j | j .
7
8     Args:
9         d: Coin dimension (typically max degree of graph)
10
11    Returns: d      d unitary coin operator.
12    """
13    psi = np.ones(d) / np.sqrt(d)
14    C = 2 * np.outer(psi, psi) - np.eye(d)
15    return C
16
17 def shift_operator(graph: nx.Graph, d_max: int) -> np.ndarray:
18     """
19         Shift operator S for coined quantum walk.
20
21     Maps |j, v      |j, w      where w is j-th neighbor of v.
22
23     Args:
24         graph: NetworkX graph
25         d_max: Maximum degree (coin dimension)
26
27     Returns: Shift operator on (d_max * N)-dimensional Hilbert space.
28     """
29     N = graph.number_of_nodes()
30     dim = d_max * N
31     S = np.zeros((dim, dim), dtype=complex)
32
33     # Relabel nodes to 0, 1, ..., N-1
34     mapping = {node: i for i, node in enumerate(graph.nodes())}
35     G = nx.relabel_nodes(graph, mapping)
36
37     for v in G.nodes():
38         neighbors = list(G.neighbors(v))
39         degree = len(neighbors)
40
41         for j, w in enumerate(neighbors):
42             # |j, v      |j, w
43             # Basis: |coin, position with index = coin + d_max *
44             #         position
45             idx_from = j + d_max * v
46             idx_to = j + d_max * w
47             S[idx_to, idx_from] = 1.0
48
49     return S
50
51 def coined_quantum_walk(graph: nx.Graph, steps: int, start_node: int =
0) -> np.ndarray:

```

```

51 """
52     Coined quantum walk on graph.
53
54     Args:
55         graph: NetworkX graph
56         steps: Number of walk steps
57         start_node: Initial position
58
59     Returns: Probability distribution over nodes after 'steps'.
60 """
61 N = graph.number_of_nodes()
62 d_max = max(dict(graph.degree()).values())
63 dim = d_max * N
64
65 # Coin operator
66 C = grover_diffusion_coin(d_max)
67
68 # Shift operator
69 S = shift_operator(graph, d_max)
70
71 # Walk operator: U = S (C      I_N)
72 U = S @ np.kron(C, np.eye(N))
73
74 # Initial state: |0, start_node (coin state 0, position
75 #                  start_node)
76 psi = np.zeros(dim, dtype=complex)
77 psi[0 + d_max * start_node] = 1.0
78
79 # Evolve for 'steps'
80 for _ in range(steps):
81     psi = U @ psi
82
83 # Measure position (trace over coin space)
84 prob = np.zeros(N)
85 for v in range(N):
86     for j in range(d_max):
87         idx = j + d_max * v
88         prob[v] += abs(psi[idx])**2
89
90 return prob
91
92 def continuous_time_quantum_walk(graph: nx.Graph, t: float, start_node: int = 0) -> np.ndarray:
93     """
94         Continuous-time quantum walk: | (t)      = exp(-iHt)| (0) .
95
96         H is the adjacency matrix (or Laplacian) of the graph.
97
98         Args:
99             graph: NetworkX graph
100            t: Evolution time
101            start_node: Initial position
102
103        Returns: Probability distribution over nodes at time t.
104 """
105 N = graph.number_of_nodes()

```

```

105
106     # Hamiltonian: adjacency matrix
107     A = nx.adjacency_matrix(graph).toarray()
108     H = A.astype(complex)
109
110     # Initial state: |start_node>
111     psi_0 = np.zeros(N, dtype=complex)
112     psi_0[start_node] = 1.0
113
114     # Evolve: |<start_node| (t) = exp(-iHt)|<start_node|(0)
115     U_t = expm(-1j * H * t)
116     psi_t = U_t @ psi_0
117
118     # Probability distribution
119     prob = np.abs(psi_t)**2
120
121     return prob
122
123 def analyze_quantum_walk_speedup(graph: nx.Graph, target_node: int,
124                                 max_steps: int = 100) -> Dict:
124     """
125         Compare quantum vs classical random walk hitting time to target
126         node.
127
127     Returns: Dictionary with hitting times and speedup factor.
128     """
129
130     N = graph.number_of_nodes()
131
132     # Quantum walk: find time to reach target with high probability
133     hitting_time_quantum = None
134     for steps in range(1, max_steps):
135         prob = coined_quantum_walk(graph, steps, start_node=0)
136         if prob[target_node] > 0.5: # Threshold for "hitting"
137             hitting_time_quantum = steps
138             break
139
140     # Classical random walk: expected hitting time
141     # Use eigenvalue analysis or simulation
142     # For simplicity, estimate as N (typical for random graphs)
143     hitting_time_classical = N # Placeholder
144
145     speedup = hitting_time_classical / hitting_time_quantum if
146             hitting_time_quantum else float('inf')
147
148     return {
149         'quantum_hitting_time': hitting_time_quantum,
150         'classical_hitting_time': hitting_time_classical,
151         'speedup': speedup,
152         'graph_size': N
153     }
154
155     # Example: Quantum walk on cycle graph
156     if __name__ == "__main__":
157         # Cycle graph with 16 nodes
158         G = nx.cycle_graph(16)

```

```

158 # Coined quantum walk
159 prob_coined = coined_quantum_walk(G, steps=20, start_node=0)
160 print("Coined quantum walk probability distribution:")
161 print(prob_coined)
162
163 # Continuous-time quantum walk
164 prob_ctqw = continuous_time_quantum_walk(G, t=5.0, start_node=0)
165 print("\nContinuous-time quantum walk probability distribution:")
166 print(prob_ctqw)
167
168 # Analyze speedup
169 speedup_result = analyze_quantum_walk_speedup(G, target_node=8)
170 print(f"\nQuantum walk hitting time:
171     {speedup_result['quantum_hitting_time']}]")
172 print(f"Speedup over classical: {speedup_result['speedup']:.2f}x")

```

0.3.3 Phase 3: HHL Algorithm for Linear Systems (Months 4-5)

Objective: Implement HHL algorithm for solving $Ax = b$, analyze complexity.

```

1 from scipy.linalg import eigh
2
3 def hhl_algorithm_simulation(A: np.ndarray, b: np.ndarray, t: float =
4     1.0,
5                                 epsilon: float = 0.01) -> Dict:
6     """
7         Simulate HHL algorithm for solving Ax = b.
8
9         Algorithm:
10            1. Phase estimation to encode eigenvalues of A in ancilla register
11            2. Controlled rotation to invert eigenvalues: R(  $\lambda_j$  ) where
12                 $\sin(\lambda_j) = 1/\lambda_j$ 
13            3. Uncompute phase estimation
14            4. Post-select on ancilla = |1
15
16        Args:
17            A: Hermitian matrix (N      N), assumed well-conditioned
18            b: Input vector (N-dimensional)
19            t: Evolution time for phase estimation
20            epsilon: Precision parameter
21
22        Returns: Dictionary with solution state | x    and success
23                  probability.
24        """
25
26        N = A.shape[0]
27
28        # Step 1: Eigenvalue decomposition of A
29        eigvals, eigvecs = eigh(A) # A =  $\lambda_j \lambda_j^\dagger$  | u_j u_j^\dagger |
30
31        # Normalize input: | b    =  $\lambda_j \lambda_j^\dagger$  | u_j
32        b_normalized = b / np.linalg.norm(b)
33        betas = eigvecs.T @ b_normalized # Coefficients  $\lambda_j = u_j^\dagger | b$ 
34
35        # Step 2: Simulate controlled rotations
36        # Exact solution: | x    =  $A^{-1}| b = \lambda_j (\lambda_j / \lambda_j^\dagger) | u_j$ 
37        x_state = np.zeros(N, dtype=complex)

```

```

34     success_prob = 0.0
35
36     C = 1.0 # Normalization constant (related to condition number )
37
38     for j in range(N):
39         if abs(eigvals[j]) > epsilon: # Avoid division by near-zero
40             eigenvalues
41             coeff = betas[j] / eigvals[j]
42             x_state += coeff * eigvecs[:, j]
43
44             # Success probability contribution from post-selection
45             # P(ancilla=1) |1/ _j |
46             success_prob += abs(betas[j])**2 / eigvals[j]**2
47
48     # Normalize
49     x_state /= np.linalg.norm(x_state)
50     success_prob /= sum(abs(betas[j])**2 / eigvals[j]**2 for j in
51                         range(N) if abs(eigvals[j]) > epsilon)
52
53     # Classical solution for comparison
54     x_classical = np.linalg.solve(A, b)
55     x_classical /= np.linalg.norm(x_classical)
56
57     # Compare quantum vs classical
58     fidelity = abs(np.vdot(x_state, x_classical))**2
59
60     return {
61         'quantum_solution_state': x_state,
62         'classical_solution': x_classical,
63         'fidelity': fidelity,
64         'success_probability': success_prob,
65         'condition_number': np.linalg.cond(A),
66         'eigenvalues': eigvals
67     }
68
69 def hhl_complexity_analysis(N: int, kappa: float) -> Dict:
70     """
71     Analyze HHL complexity: O(log N poly( , 1/ )).
72
73     Compare to classical Gaussian elimination: O(N ) for sparse,
74     O(N ) for dense.
75
76     Args:
77         N: Matrix dimension
78         kappa: Condition number      = _max / _min
79
80     Returns: Complexity estimates.
81     """
82
83     # Quantum complexity (gate count)
84     # Phase estimation: O(poly(log N, log , log(1/ )))
85     # Hamiltonian simulation: O(poly(log N))
86     quantum_gates = (np.log2(N))**2 * np.log2(kappa)

87
88     # Classical complexity
89     classical_ops_sparse = N**2 # Sparse solver
90     classical_ops_dense = N**3 # Dense Gaussian elimination

```

```

87     # Speedup (caveat: assumes efficient state preparation and readout)
88     speedup_sparse = classical_ops_sparse / quantum_gates
89     speedup_dense = classical_ops_dense / quantum_gates
90
91     return {
92         'quantum_gates': quantum_gates,
93         'classical_ops_sparse': classical_ops_sparse,
94         'classical_ops_dense': classical_ops_dense,
95         'speedup_vs_sparse': speedup_sparse,
96         'speedup_vs_dense': speedup_dense,
97         'caveat': 'Speedup assumes O(polylog N) state preparation and
98                   measurement'
99     }
100
101 # Example: Solve simple linear system
102 if __name__ == "__main__":
103     # Construct well-conditioned Hermitian matrix
104     N = 8
105     A = np.random.randn(N, N)
106     A = (A + A.T) / 2 # Symmetrize
107     A += 5 * np.eye(N) # Ensure positive definite (condition number
108     ~0(1))
109
110     b = np.random.randn(N)
111
112     result = hhl_algorithm_simulation(A, b)
113
114     print("HHL Algorithm Simulation:")
115     print(f"    Fidelity with classical solution:
116           {result['fidelity']:.6f}")
117     print(f"    Success probability: {result['success_probability']:.6f}")
118     print(f"    Condition number : {result['condition_number']:.2f}")
119
120     # Complexity analysis
121     complexity = hhl_complexity_analysis(N=1024, kappa=10.0)
122     print(f"\nComplexity for N=1024,   =10:")
123     print(f"    Quantum gates: {complexity['quantum_gates']:.0f}")
     print(f"    Classical ops (sparse):
           {complexity['classical_ops_sparse']:.0e}")
     print(f"    Speedup vs sparse:
           {complexity['speedup_vs_sparse']:.2e}x")

```

0.3.4 Phase 4: QAOA for Combinatorial Optimization (Months 5-6)

Objective: Implement QAOA for MaxCut, analyze approximation ratio.

```

1  from scipy.optimize import minimize
2  from itertools import combinations
3
4  def maxcut_hamiltonian(graph: nx.Graph) -> np.ndarray:
5      """
6          MaxCut cost Hamiltonian: H_C = - {(i,j) in E} (1 - Z_i Z_j).
7
8          Maximizing cut size      minimizing -H_C.
9

```

```

10     Returns: 2^N      2^N matrix.
11 """
12 N = graph.number_of_nodes()
13 dim = 2**N
14 H_C = np.zeros((dim, dim))
15
16 for i, j in graph.edges():
17     # Z_i Z_j operator
18     Z_i = pauli_z_on_qubit(i, N)
19     Z_j = pauli_z_on_qubit(j, N)
20     ZZ = Z_i @ Z_j
21
22     H_C += 0.5 * (np.eye(dim) - ZZ)
23
24 return H_C
25
26 def pauli_z_on_qubit(k: int, N: int) -> np.ndarray:
27 """
28 Z operator on qubit k in N-qubit system.
29
30 Z = [[1,0],[0,-1]]
31 """
32 Z = np.array([[1, 0], [0, -1]])
33 I = np.eye(2)
34
35 op = I
36 for j in range(N):
37     if j == 0:
38         op = Z if k == 0 else I
39     else:
40         op = np.kron(op, Z if k == j else I)
41
42 return op
43
44 def pauli_x_on_qubit(k: int, N: int) -> np.ndarray:
45 """
46 X operator on qubit k.
47 """
48 X = np.array([[0, 1], [1, 0]])
49 I = np.eye(2)
50
51 op = I
52 for j in range(N):
53     if j == 0:
54         op = X if k == 0 else I
55     else:
56         op = np.kron(op, X if k == j else I)
57
58 return op
59
60 def mixer_hamiltonian(N: int) -> np.ndarray:
61 """
62 Mixer Hamiltonian: H_M = _i X_i.
63
64 Returns: 2^N      2^N matrix.
65 """
66 dim = 2**N
67 H_M = np.zeros((dim, dim))

```

```

66
67     for i in range(N):
68         H_M += pauli_x_on_qubit(i, N)
69
70     return H_M
71
72 def qaoa_circuit(params: np.ndarray, H_C: np.ndarray, H_M: np.ndarray,
73 p: int) -> np.ndarray:
74     """
75     QAOA circuit: | ( , ) = _ {i=1}^p e^{-i \gamma_i H_M} e^{-i \beta_i H_C} |+ ^ n .
76
77     Args:
78         params: Array of 2p parameters [ \gamma_1 , ..., \gamma_p , \beta_1 , ..., \beta_p ]
79         H_C: Cost Hamiltonian
80         H_M: Mixer Hamiltonian
81         p: Number of QAOA layers
82
83     Returns: Final state | ( , ) .
84     """
85
86     N = int(np.log2(H_C.shape[0]))
87     dim = 2**N
88
89     gamma = params[:p]
90     beta = params[p:]
91
92     # Initial state: |+ ^ n = H^n |0
93     psi = np.ones(dim) / np.sqrt(dim)
94
95     # Apply QAOA layers
96     for i in range(p):
97         # Cost layer: e^{-i \gamma_i H_C}
98         U_C = expm(-1j * gamma[i] * H_C)
99         psi = U_C @ psi
100
101         # Mixer layer: e^{-i \beta_i H_M}
102         U_M = expm(-1j * beta[i] * H_M)
103         psi = U_M @ psi
104
105     return psi
106
107 def qaoa_maxcut(graph: nx.Graph, p: int = 1, max_iter: int = 100) ->
108     Dict:
109     """
110     QAOA for MaxCut problem.
111
112     Args:
113         graph: NetworkX graph
114         p: Number of QAOA layers
115         max_iter: Maximum optimization iterations
116
117     Returns: Optimal parameters, state, and approximation ratio.
118     """
119
120     N = graph.number_of_nodes()
121
122     # Construct Hamiltonians

```

```

119     H_C = maxcut_hamiltonian(graph)
120     H_M = mixer_hamiltonian(N)
121
122     # Objective function:      ( , )|H_C| ( , )
123     def objective(params):
124         psi = qaoa_circuit(params, H_C, H_M, p)
125         expectation = np.real(psi.conj() @ H_C @ psi)
126         return -expectation # Minimize - H_C to maximize cut size
127
128     # Optimize
129     init_params = np.random.uniform(0, 2*np.pi, 2*p)
130     result = minimize(objective, init_params, method='COBYLA',
131                        options={'maxiter': max_iter})
132
133     # Extract solution
134     optimal_params = result.x
135     optimal_psi = qaoa_circuit(optimal_params, H_C, H_M, p)
136     qaoa_cut_value = -result.fun
137
138     # Classical MaxCut upper bound (brute force for small graphs)
139     max_cut_classical = maxcut_brute_force(graph)
140
141     # Approximation ratio
142     approx_ratio = qaoa_cut_value / max_cut_classical if
143         max_cut_classical > 0 else 0
144
145     return {
146         'optimal_params': optimal_params,
147         'optimal_state': optimal_psi,
148         'qaoa_cut_value': qaoa_cut_value,
149         'max_cut_classical': max_cut_classical,
150         'approximation_ratio': approx_ratio,
151         'p': p
152     }
153
154     def maxcut_brute_force(graph: nx.Graph) -> float:
155         """
156             Compute maximum cut via brute force enumeration (feasible for N
157             15).
158         """
159         N = graph.number_of_nodes()
160         max_cut = 0
161
162         for partition in range(2***(N-1)): # Only need half due to symmetry
163             cut_size = 0
164             # Decode partition as bitstring
165             S1 = {i for i in range(N) if (partition >> i) & 1}
166             S2 = set(range(N)) - S1
167
168             for i, j in graph.edges():
169                 if (i in S1 and j in S2) or (i in S2 and j in S1):
170                     cut_size += 1
171
172             max_cut = max(max_cut, cut_size)
173
174     return max_cut

```

```

173
174 # Example: QAOA on small random graph
175 if __name__ == "__main__":
176     # Random graph with 6 nodes
177     G = nx.erdos_renyi_graph(6, 0.5, seed=42)
178
179     # Run QAOA with p=1
180     result = qaoa_maxcut(G, p=1)
181
182     print("QAOA for MaxCut:")
183     print(f"    QAOA cut value: {result['qaoa_cut_value']:.4f}")
184     print(f"    Optimal cut (classical):")
185     print(f"        {result['max_cut_classical']:.0f}")
186     print(f"    Approximation ratio: {result['approximation_ratio']:.4f}")
187     print(f"    p={result['p']}")

```

0.3.5 Phase 5: Complexity Analysis and Oracle Separations (Months 6-7)

Objective: Prove query complexity lower bounds, construct BQP vs BPP oracle separations.

```

1 def query_complexity_lower_bound_adversary(N: int) -> Dict:
2     """
3         Prove (N) lower bound for Grover search using adversary method.
4
5         Adversary argument: any quantum algorithm distinguishing between two
6         functions f, g (differing on single input) requires (N)
7         queries.
8
9         Returns: Lower bound certificate.
10    """
11    # Adversary matrix with [x,y] = 1 if f_x(y) != g_x(y)
12    # For Grover: f has solution at x, g has solution at y
13    # Spectral norm || || = N gives lower bound
14
15    lower_bound = np.sqrt(N)
16
17    # Certificate: spectral norm of adversary matrix
18    # (For full proof, construct and compute eigenvalues)
19
20    return {
21        'problem': 'Grover search',
22        'lower_bound': lower_bound,
23        'method': 'Adversary method',
24        'certificate': 'Spectral norm || || = N',
25    }
26
27 def bqp_bpp_oracle_separation() -> Dict:
28     """
29         Construct oracle separation proving BQP^0 ⊈ BPP^0.
30
31         Use Recursive Fourier Sampling (RFS) problem:
32         - Quantum algorithm solves RFS in poly(n) queries
33         - Classical algorithm requires exp(n) queries
34
35         Returns: Oracle construction and complexity bounds.
36     """

```

```

36     # RFS problem: Given oracle access to function f: Z_2^n      Z_2,
37     # output Fourier coefficient f (s) for uniformly random s
38
39     # Quantum algorithm: Hadamard test in O(1) queries
40     quantum_queries = 1
41
42     # Classical algorithm: Must estimate f (s) = (1/2^n) _x
43     # (-1)^{f(x) + s x}
44     # Requires (2^n) samples to distinguish from 0
45     classical_queries_lower_bound = lambda n: 2**(n-1)
46
47     n_example = 10
48
49     return {
50         'problem': 'Recursive Fourier Sampling',
51         'quantum_queries': quantum_queries,
52         'classical_queries_lower_bound':
53             classical_queries_lower_bound(n_example),
54         'separation': 'Exponential',
55         'conclusion': 'BQP^0      BPP^0 for oracle 0 encoding RFS'
56     }
57
58 # Example: Lower bound certificates
59 if __name__ == "__main__":
60     # Grover lower bound
61     lb_grover = query_complexity_lower_bound_adversary(N=1024)
62     print("Grover Search Lower Bound:")
63     print(f"  Problem: {lb_grover['problem']}")
64     print(f"  Lower bound:  ({lb_grover['lower_bound']:.0f}) queries")
65     print(f"  Method: {lb_grover['method']}")

66     # BQP vs BPP separation
67     separation = bqp_bpp_oracle_separation()
68     print(f"\nBQP vs BPP Oracle Separation:")
69     print(f"  Problem: {separation['problem']}")
70     print(f"  Quantum: {separation['quantum_queries']} queries")
71     print(f"  Classical:
72         ({separation['classical_queries_lower_bound']}) queries")
73     print(f"  Separation: {separation['separation']}")
```

0.3.6 Phase 6: Certificate Generation and Export (Months 7-9)

Objective: Generate machine-checkable certificates for all algorithms.

```

1  from dataclasses import dataclass, asdict
2  import json
3  from datetime import datetime
4
5  @dataclass
6  class QuantumAlgorithmCertificate:
7      """Certificate for quantum algorithm performance and correctness."""
8
9      algorithm_name: str
10     problem_size: int    # N or n (number of qubits)
11
12     # Query complexity
```

```

13     quantum_queries: int
14     classical_queries_lower_bound: int
15     speedup: float
16
17     # Success probability
18     success_probability: float
19     error_bound: float
20
21     # Circuit details
22     qubit_count: int
23     gate_count: int
24     circuit_depth: int
25
26     # Verification
27     correctness_verified: bool
28     optimality_proof: str # "Adversary method", "Polynomial method",
29         etc.
30
31     # Metadata
32     timestamp: str
33     simulation_time: float
34
35 def generate_quantum_algorithm_certificate(algorithm_result: Dict,
36                                             algorithm_name: str) ->
37                                             QuantumAlgorithmCertificate:
38     """
39     Generate certificate for quantum algorithm.
40
41     Args:
42         algorithm_result: Output from quantum algorithm simulation
43         algorithm_name: "Grover", "QuantumWalk", "HHL", "QAOA"
44
45     Returns: Certificate object.
46     """
47
48     if algorithm_name == "Grover":
49         N = 2**algorithm_result.get('n', 4)
50         cert = QuantumAlgorithmCertificate(
51             algorithm_name="Grover Search",
52             problem_size=N,
53             quantum_queries=algorithm_result['oracle_queries'],
54             classical_queries_lower_bound=N // 2,
55             speedup=N / (2 * algorithm_result['oracle_queries']),
56             success_probability=algorithm_result['success_probability'],
57             error_bound=1.0 / N,
58             qubit_count=int(np.log2(N)),
59             gate_count=algorithm_result['oracle_queries'] * N, # Rough
60                 estimate
61             circuit_depth=algorithm_result['oracle_queries'],
62             correctness_verified=algorithm_result['correct'],
63             optimality_proof="Adversary method (Bennett et al., 1997)",
64             timestamp=datetime.now().isoformat(),
65             simulation_time=0.0
66         )
67
68     # Add similar branches for QuantumWalk, HHL, QAOA

```

```

66     return cert
67
68 def export_certificate_json(cert: QuantumAlgorithmCertificate,
69     filepath: str):
70     """Export certificate to JSON."""
71     with open(filepath, 'w') as f:
72         json.dump(asdict(cert), f, indent=2)
73
74     print(f"Certificate exported to {filepath}")
75
76 # Example: Full pipeline
77 if __name__ == "__main__":
78     # Run Grover
79     n = 6
80     marked = [42]
81     grover_result = grover_search(marked, n)
82     grover_result['n'] = n
83
84     # Generate certificate
85     cert = generate_quantum_algorithm_certificate(grover_result,
86         "Grover")
87
88     # Export
89     export_certificate_json(cert, "grover_certificate.json")
90
91     print("\nCertificate Summary:")
92     print(f"    Algorithm: {cert.algorithm_name}")
93     print(f"    Quantum queries: {cert.quantum_queries}")
94     print(f"    Classical lower bound:
95         {cert.classical_queries_lower_bound}")
96     print(f"    Speedup: {cert.speedup:.2f}x")
97     print(f"    Success probability: {cert.success_probability:.6f}")

```

0.4 4. Example Starting Prompt

Use this prompt to initialize a long-running AI system for quantum algorithms research:

```

1 You are a quantum algorithm researcher studying computational
2 complexity and quantum advantage.
3 Your task is to implement canonical quantum algorithms (Grover, quantum
4 walks, HHL, QAOA),
5 analyze their query complexity, and prove optimality via adversary and
6 polynomial methods.
7
8 CONTEXT:
9 Quantum computing harnesses superposition and interference to solve
  certain problems faster
than classical computers. Grover's algorithm searches unsorted
  databases in O( N ) queries
versus O(N) classically a provable quadratic speedup. Shor's
  algorithm factors integers in
polynomial time, threatening RSA. Quantum walks generalize random
  walks, achieving speedups

```

```

10 for graph problems. QAOA tackles combinatorial optimization on
11 near-term devices.

12 Complexity theory classifies problems by resources required. BQP
13 (Bounded-error Quantum
14 Polynomial) contains problems solvable by quantum computers in poly(n)
15 time with error 1 /3.
16 BPP is the classical randomized analogue. Oracle separations prove BQP
17 BPP, but relations
18 to NP remain open.

19 OBJECTIVE:
20 Phase 1 (Months 1-2): Implement Grover's algorithm for N=2^n items.
21 Verify O( N ) queries,
22 success probability 1-1/N. Prove optimality via adversary method
23 (spectral norm || ||= N ).

24 Phase 2 (Months 2-4): Implement coined and continuous-time quantum
25 walks on graphs (cycle,
26 hypercube, complete). Analyze hitting times, compare to classical
27 random walks. Identify
28 quadratic speedups.

29 Phase 3 (Months 4-5): Implement HHL algorithm for linear systems Ax=b.
30 Analyze complexity
31 O(log N poly( )), compare to classical O(N ) for sparse, O(N ) for
32 dense. Discuss caveats
33 (state preparation, readout).

34 Phase 4 (Months 5-6): Implement QAOA for MaxCut on random graphs.
35 Optimize parameters ,
36 via classical minimization. Compute approximation ratio vs
37 brute-force solution. Analyze
38 performance vs graph structure.

39 Phase 5 (Months 6-7): Prove query complexity lower bounds using
40 adversary method (Grover ( N )),,
41 polynomial method (collision finding (N^{1/3})). Construct BQP vs
42 BPP oracle separation via
43 Recursive Fourier Sampling.

44 Phase 6 (Months 7-9): Generate machine-checkable certificates for all
45 algorithms:
46 - Query complexity and classical lower bounds
47 - Success probabilities and error bounds
48 - Circuit parameters (qubits, gates, depth)
49 - Optimality proofs (adversary matrix spectral norms)
50 - Export as JSON with exact arithmetic where applicable

51 PURE THOUGHT CONSTRAINTS:
52 - Simulate quantum circuits exactly using numpy linear algebra ( 20
53 qubits)
54 - All complexity claims must have rigorous proofs (adversary,
55 polynomial, hybrid arguments)
56 - Compare quantum vs classical on identical problems without hardware
57 assumptions

```

```

48 - No approximations beyond specified error bounds (e.g., =10
   for QAOA optimization)
49 - Export quantum states, gates, and measurements with full precision
50
51 SUCCESS CRITERIA:
52 - Minimum Viable Result (2-4 months): Grover working with verified N
   scaling, quantum walk
53   on simple graphs, basic QAOA implementation
54 - Strong Result (6-8 months): All algorithms operational, query
   complexity lower bounds proven,
55   HHL analysis complete, QAOA approximation ratios measured
56 - Publication-Quality (9 months): BQP vs BPP oracle separation
   constructed, novel quantum walk
57   applications, comprehensive complexity analysis, comparison with
   theoretical bounds
58
59 START:
60 Begin with Grover's algorithm (Phase 1). Implement oracle, Grover
   operator, and measurement.
61 Verify success probability  $1 - 1/N$  for  $N=16,64,256$ . Plot queries vs
   N to confirm scaling.
62 Prove ( $N$ ) lower bound via adversary method. Export certificate
   with all details.

```

0.5 5. Success Criteria

0.5.1 Minimum Viable Result (MVR) - 2-4 Months

Core Functionality:

- Grover's algorithm: finds marked item in $N/4$ queries with $P(\text{success}) = 1 - 1/N$
- Quantum walk on cycle and hypercube graphs: hitting time measured
- Basic QAOA implementation: MaxCut on 6-8 node graphs
- Certificate generation: query counts, success probabilities

Deliverables:

- `grover.py`: Complete implementation with optimality verification
- `quantumwalk.py` : *Coined and continuous – timewalks*
- `qaoamaxcut.py` : *QAOA with classical parameter optimization*
- `certificates.json`: Query complexity and success probability data

Quality Metrics:

- Grover: $|\text{queries} - N/4| < 2$ for all $N = 2^n$ with $n \leq 10$
- Quantum walk: hitting time on cycle C_N is $O(N)$ vs classical $O(N^2)$
- QAOA: approximation ratio 0.7 for random 3-regular graphs

0.5.2 Strong Result - 6-8 Months

Extended Capabilities:

- HHL algorithm: solve $2^n \times 2^n$ systems with $n \leq 10$, fidelity with classical solution > 0.99

- Query complexity lower bounds: adversary method for Grover ((N)), element distinctness ($(N^{2/3})$)
- Quantum walk speedups: analyze on 10+ graph families (trees, grids, expanders)
- QAOA: test on MaxCut, Max-SAT, graph coloring; approximation ratios documented

Deliverables:

- `hhl.py`: Full HHL with complexity analysis
- `complexity_bounds.py`: Adversary and polynomial method implementations
- `qaoa_suite.py`: QAOA for multiple combinatorial problems
- Research report: "Quantum vs Classical: A Pure Thought Comparison"

Quality Metrics:

- HHL: 100, success probability 0.5, quantum gates $O((\log N)^2)$
- Lower bounds: adversary matrix spectral norms computed exactly (sympy)
- Quantum walk: speedup factor 2 verified on 5 graph families
- QAOA: depth p = 3, approximation ratio documented vs graph size

0.5.3 Publication-Quality Result - 9 Months

Novel Contributions:

- BQP vs BPP oracle separation: full construction of Recursive Fourier Sampling oracle
- New quantum walk application: novel algorithm for graph property testing
- QAOA performance theory: approximation ratio bounds vs graph structure
- Comprehensive database: 1000+ quantum algorithm runs with certificates

Deliverables:

- `oracle_separation.py`: Explicit $BQP^O \neq BPP^O$ construction
- Research paper: "Provable Quantum Advantage: From Grover to Oracle Separations"
- Interactive visualization: Quantum vs classical complexity comparison
- Formal verification: Lean4 proofs for Grover optimality (optional advanced goal)

Quality Metrics:

- Oracle separation: quantum $O(1)$ vs classical (2^n) rigorously proven
- Novel quantum walk: outperforms classical by quadratic factor on new problem
- QAOA theory: approximation ratio bounds proven for specific graph classes
- All certificates verified: success probabilities, query counts, circuit parameters

0.6 6. Verification Protocol

0.6.1 Automated Checks (Run After Every Phase)

```

1  def verify_quantum_algorithm_certificate(cert:
2      QuantumAlgorithmCertificate) -> Dict[str, bool]:
3      """
4          Verify quantum algorithm certificate.
5
6          Returns: Dictionary of Boolean checks.
7      """
8      checks = []
9
10     # 1. Speedup calculation
11     theoretical_speedup = cert.classical_queries_lower_bound /
12         max(cert.quantum_queries, 1)
13     checks['speedup_correct'] = np.isclose(cert.speedup,
14         theoretical_speedup, rtol=0.01)
15
16     # 2. Success probability within bounds
17     checks['success_prob_valid'] = 0.0 <= cert.success_probability
18         <= 1.0
19     checks['error_bound_valid'] = cert.success_probability >= 1 -
20         cert.error_bound
21
22     # 3. Query complexity matches algorithm
23     if cert.algorithm_name == "Grover Search":
24         N = cert.problem_size
25         expected_queries = int(np.pi * np.sqrt(N) / 4)
26         checks['queries_optimal'] = abs(cert.quantum_queries -
27             expected_queries) <= 2
28
29     # 4. Circuit parameters consistent
30     checks['qubit_count_valid'] = cert.qubit_count >=
31         int(np.log2(cert.problem_size))
32     checks['gate_count_positive'] = cert.gate_count > 0
33     checks['depth_reasonable'] = cert.circuit_depth <=
34         cert.gate_count
35
36     return checks
37
38
39 # Example usage
40 cert_example = QuantumAlgorithmCertificate(
41     algorithm_name="Grover Search",
42     problem_size=256,
43     quantum_queries=13,
44     classical_queries_lower_bound=128,
45     speedup=9.85,
46     success_probability=0.996,
47     error_bound=1/256,
48     qubit_count=8,
49     gate_count=256,
50     circuit_depth=13,
51     correctness_verified=True,
52     optimality_proof="Adversary method",
53     timestamp=datetime.now().isoformat(),
54

```

```

45     simulation_time=0.5
46 )
47
48 verification = verify_quantum_algorithm_certificate(cert_example)
49 print("Certificate Verification:")
50 for check, passed in verification.items():
51     status = "    PASS" if passed else "    FAIL"
52     print(f"    {status}: {check}")

```

0.6.2 Cross-Validation Against Known Results

```

1 KNOWN_COMPLEXITY_BOUNDS = {
2     'Grover': {'quantum': lambda N: np.pi * np.sqrt(N) / 4,
3                 'classical': lambda N: N},
4     'ElementDistinctness': {'quantum': lambda N: N**(2/3),
5                             'classical': lambda N: N},
6     'CollisionFinding': {'quantum': lambda N: N**(1/3),
7                           'classical': lambda N: np.sqrt(N)},
8 }
9
10 def cross_validate_complexity(algorithm: str, N: int,
11                                 measured_queries: int):
12     """Compare measured query complexity to theoretical bounds."""
13     if algorithm in KNOWN_COMPLEXITY_BOUNDS:
14         expected = KNOWN_COMPLEXITY_BOUNDS[algorithm]['quantum'](N)
15         error = abs(measured_queries - expected) / expected
16         print(f"{algorithm}: measured={measured_queries},
17               expected={expected:.2f}, error={error:.2%}")
18         assert error < 0.1, f"Query complexity deviates >10% from
19                           theory"

```

0.7 7. Resources and Milestones

0.7.1 Essential References

Foundational Papers:

- L. Grover, "A Fast Quantum Mechanical Algorithm for Database Search", STOC 1996
- P. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring", FOCS 1994
- C. Bennett et al., "Strengths and Weaknesses of Quantum Computing", SIAM J. Comp. 26, 1510 (1997)

Quantum Walks:

- A. Ambainis, "Quantum Walk Algorithm for Element Distinctness", FOCS 2004
- A. Childs et al., "Exponential Algorithmic Speedup by Quantum Walk", STOC 2003

Complexity Theory:

- S. Aaronson, Y. Shi, "Quantum Lower Bounds for the Collision and Element Distinctness Problems", JACM 51, 595 (2004)
- A. Ambainis, "Polynomial Degree and Lower Bounds in Quantum Complexity", CCC 2003

Variational Algorithms:

- E. Farhi, J. Goldstone, S. Gutmann, "A Quantum Approximate Optimization Algorithm", arXiv:1411.4028 (2014)
- A. Peruzzo et al., "A Variational Eigenvalue Solver on a Photonic Quantum Processor", Nat. Commun. 5, 4213 (2014)

Reviews:

- M. Nielsen, I. Chuang, "Quantum Computation and Quantum Information" (Cambridge, 2010) [[Start here](#)]

0.7.2 Software Tools

- **NumPy** (v1.24+): Matrix exponentiation, eigenvalue decomposition
- **SciPy** (scipy.linalg): expm, eigh for quantum evolution
- **NetworkX** (v3.0+): Graph construction for quantum walks
- **Qiskit** (optional): Cross-check against IBM's quantum simulator (for validation)

0.7.3 Common Pitfalls

- **Small-Angle Approximation in Grover:** Near-optimal iterations are crucial; rounding errors can degrade success probability
- **Phase Kickback:** Incorrect oracle implementation can miss phase flip, breaking Grover
- **QAOA Optimization Landscape:** Non-convex, many local minima; use multiple random initializations
- **HHL Caveats:** Exponential speedup requires efficient state preparation (often not achievable in practice)
- **Quantum Walk Encoding:** Shift operator must respect graph structure; incorrect indexing breaks unitarity

0.7.4 Milestone Checklist

Month 2:

- x Grover's algorithm: N=16, 64, 256 with success probability 0.99
- x Query complexity verified: $N/4 + 2$
- x Adversary lower bound: (N) proven via spectral norm

Month 4:

Quantum walks: coined and continuous-time on cycle, hypercube

Hitting time analysis: quantum $O(N)$ vs classical $O(N^2)$ on cycle

HHL algorithm: solving 8×8 systems with fidelity >0.99

Month 6:

QAOA: MaxCut on 10-node graphs, approximation ratio 0.7

Query lower bounds: element distinctness $(N^{2/3})$, collision($N^{1/3}$)

Complexity analysis complete for all algorithms

Month 9:

BQP vs BPP oracle separation: Recursive Fourier Sampling

Novel quantum walk application identified and tested

Comprehensive database: 1000+ algorithm runs with certificates

Research paper draft: "Provable Quantum Advantage via Pure Thought"

End of PRD 25: Quantum Algorithms and Computational Complexity

Pure thought investigation of quantum computational advantage through rigorous implementation and complexity analysis. All speedups proven via adversary and polynomial methods, with machine-checkable certificates.