Gas Cost and Algorithmic Complexity in EVM-Based Smart Contracts

Lucas Kim
Loyola Marymount University
Los Angeles, CA, US

Abstract

This report presents an empirical analysis of the relationship between algorithmic complexity and gas costs in Ethereum Virtual Machine (EVM)-based smart contracts. Through systematic measurement and comparison of different implementation patterns, we demonstrate that computational complexity directly translates to financial costs in blockchain environments. The findings reveal significant gas cost variations—up to 56.26% reduction—when applying algorithmic optimization principles to smart contract design.

1 Introduction

1.1 Problem Statement

In traditional software development, algorithmic complexity analysis focuses primarily on time and space efficiency. However, in blockchain environments, computational complexity has direct financial implications through the gas cost mechanism. Every operation executed on the EVM consumes gas, which must be paid in Ether. This creates a unique paradigm where Big-O complexity notation directly correlates with economic cost.

1.2 Research Significance

The connection between algorithmic complexity and gas costs is fundamental to understanding blockchain economics. Unlike traditional computing environments where computational resources are abstracted, blockchain platforms make computational cost explicit and measurable. This research addresses three critical questions:

(1) How does algorithmic complexity translate into financial cost in Ethereum?
(2) What specific operations cause gas cost spikes?
(3) How can smart contracts be optimized to reduce gas consumption?

1.3 Why Complexity Matters in Blockchain Context

1.3.1 Computational Complexity as Economic Constraint. In the EVM, every computational step incurs a gas cost. This means that:

- An O(n) algorithm will have gas costs that scale linearly with input size
- An O(1) algorithm maintains constant gas cost regardless of input size
- Nested loops ($O(n^2)$) result in quadratic gas cost growth

This direct mapping between complexity and cost is unique to blockchain environments. In traditional systems, developers optimize for performance; in blockchain systems, they must optimize for both performance and economic efficiency.

1.3.2 The Immutability Factor. Smart contracts are immutable once deployed. Unlike traditional software that can be patched or updated, smart contracts require redeployment—at significant cost—to fix inefficiencies. This makes upfront complexity analysis and optimization critical.

1.3.3 Network Resource Constraints. Blockchain networks have limited block space and processing capacity. Inefficient algorithms not only cost more but also consume more network resources, potentially affecting transaction throughput and network scalability.

2 Methodology

2.1 Experimental Design

We implemented four distinct contract patterns to measure gas cost differences:

(1) **Loop Complexity Comparison**: O(n) linear search vs. O(1) constant-time lookup
(2) **Storage vs. Memory Operations**: Persistent storage vs. temporary memory
(3) **Function Modularity**: Code duplication vs. reusable internal functions
(4) **Token Airdrop Case Study**: Naive vs. optimized batch processing

2.2 Measurement Framework

All measurements were conducted using Hardhat's local network, providing accurate gas cost tracking without actual blockchain deployment costs. Each experiment was executed multiple times to ensure consistency, and gas costs were measured using Hardhat's built-in gas reporting capabilities.

2.3 Test Environment

- **Solidity Version**: 0.8.20
- **EVM Target**: Paris
- **Network**: Hardhat Local Network
- **Measurement Tool**: Hardhat Gas Reporter

3 Experimental Results

3.1 Loop Complexity Analysis

Experiment: O(n) vs. O(1) Operations

We compared linear array search against constant-time mapping lookups.

Results:

- **Precomputation Initialization**: 303,191 gas (one-time O(n) cost)
- **O(1) Precomputed Sum Access**: 2,000 gas (constant time)

- **O(n) Array Sum Loop**: Gas cost scales linearly with array size

Analysis:

The precomputation pattern demonstrates a fundamental optimization principle: trading one-time O(n) computation for repeated O(1) access. While the initial computation costs 303,191 gas, subsequent accesses require only 2,000 gas, regardless of array size. For frequently accessed data, this represents a significant cost reduction.

Implications:

- Use mappings instead of arrays when constant-time access is required
- Precompute expensive calculations when values are accessed multiple times
- Consider data access patterns when choosing data structures

3.2 Storage vs. Memory Operations

Experiment: Persistent Storage vs. Temporary Memory

We measured gas costs for processing 10 elements using storage operations versus memory operations.

Results:

- **Storage Operations (10 elements)**: 278,301 gas
- **Memory Operations (10 elements)**: 414,400 gas (includes one-time copy cost)
- **Storage Read Cost**: 2,100 gas per read
- **Memory Read Cost**: 3 gas per read

Analysis:

Storage operations are significantly more expensive than memory operations. A single storage write costs approximately 20,000 gas (first write) or 5,000 gas (subsequent writes), while memory operations cost approximately 3 gas per word. However, the initial cost of copying storage to memory (414,400 gas for 10 elements) must be amortized over subsequent operations.

Critical Finding:

For operations that require multiple iterations over the same data, copying to memory first and then performing operations is more gas-efficient. The break-even point depends on the number of iterations and the size of the data structure.

Implications:

- Minimize storage writes by batching operations
- Copy arrays to memory before iteration when multiple passes are needed
- Use memory for temporary computations
- Write to storage only when persistence is required

3.3 Function Modularity Analysis

Experiment: Code Duplication vs. Modular Design

We compared gas costs for duplicated functions versus modular functions using internal helpers.

Results:

- **Duplicated Functions**: 43,734 gas, 43,712 gas, 43,702 gas (average: 43,716 gas)
- **Modular Functions**: 43,762 gas, 43,717 gas, 43,808 gas (average: 43,762 gas)

Analysis:

Runtime gas costs are nearly identical between duplicated and modular approaches, as internal functions are inlined during compilation. However, modular design significantly reduces contract bytecode size, which affects deployment costs and contract size limits.

Implications:

- Internal functions have no runtime overhead (compiler inlining)
- Modular design improves maintainability without gas cost penalty
- Code duplication increases deployment costs and contract size

3.4 Token Airdrop Case Study

Experiment: Naive vs. Optimized Airdrop Implementation

We compared three approaches to token airdrops:

(1) Naive: Individual transfers in a loop
(2) Optimized: Batch processing with unchecked arithmetic
(3) Chunked: Processing large datasets in batches

Results:

Approach	Recipients	Gas Cost	Gas per Recipient	Efficiency
Naive	3	118,867	39,622	Baseline
Optimized	3	115,368	38,456	2.94% reduction
Naive	10	300,243	30,024	Baseline
Optimized	10	131,336	13,134	56.26% reduction
Chunked	20	331,825	16,591	Scalable

Analysis:

The optimized approach demonstrates significant gas savings, particularly as the number of recipients increases. For 10 recipients, the optimized implementation reduces gas costs by 168,907 gas (56.26% reduction). The chunked approach enables processing arbitrarily large airdrops by breaking them into manageable batches.

Key Optimizations:

(1) **Unchecked Arithmetic**: Saves approximately 20 gas per iteration by skipping overflow checks in safe loops
(2) **Batch Event Emission**: Reduces event emission overhead
(3) **Chunked Processing**: Prevents gas limit issues for large datasets

Implications:

- Algorithmic optimizations scale with input size
- Small optimizations compound significantly for large datasets
- Batch processing is essential for production deployments

4 Theoretical Framework: Complexity to Gas Cost Mapping

4.1 The Computational Cost Model

In the EVM, gas costs follow a predictable pattern based on operation complexity:

Constant Time Operations (O(1)):

- Mapping lookups: 2,100 gas (storage) or 3 gas (memory)
- Arithmetic operations: 3-5 gas
- Function calls: 21 gas base + 3 gas per parameter

Linear Time Operations (O(n)):

- Array iteration: 10-50 gas per iteration + operation costs
- Storage writes in loops: 5,000-20,000 gas per write
- Memory operations in loops: 3 gas per operation

Quadratic Time Operations ($O(n^2)$):

- Nested loops: Product of inner and outer loop costs
- Can quickly exceed block gas limits

4.2 Why This Mapping Matters

4.2.1 Economic Efficiency. Every unit of gas has a monetary value. At current Ethereum prices, inefficient algorithms can cost thousands of dollars in unnecessary fees. For high-frequency operations or contracts with many users, these costs compound significantly.

4.2.2 Network Scalability. Inefficient contracts consume more block space and processing capacity, reducing overall network throughput. This creates negative externalities for all network participants.

4.2.3 User Experience. High gas costs create barriers to entry and reduce the economic viability of decentralized applications. Users may abandon transactions or applications due to prohibitive costs.

4.2.4 Sustainability. As blockchain networks scale, efficient resource utilization becomes critical for long-term sustainability. Complexity-aware design is essential for building scalable decentralized systems.

5 Practical Implications and Recommendations

5.1 Design Principles

Based on our experimental findings, we recommend the following principles for gas-efficient smart contract design:

(1) **Prefer O(1) over O(n)**: Use mappings instead of arrays when constant-time access is required
(2) **Minimize Storage Operations**: Batch memory operations before writing to storage
(3) **Precompute Expensive Calculations**: Trade one-time computation for repeated O(1) access
(4) **Use Unchecked Arithmetic**: In safe loops, unchecked arithmetic saves gas
(5) **Batch Operations**: Reduce transaction overhead by processing multiple items per transaction
(6) **Modularize Code**: Improve maintainability without runtime gas cost penalty

5.2 Optimization Strategies

For Loop Operations:

- Replace linear searches with mapping lookups
- Precompute values accessed multiple times
- Use unchecked arithmetic in safe loops

For Storage Operations:

- Copy arrays to memory before iteration
- Batch storage writes
- Minimize storage reads in loops

For Large Datasets:

- Implement chunked processing
- Use batch operations
- Consider off-chain computation with on-chain verification

5.3 Measurement and Validation

All optimizations should be validated through gas cost measurement. The Hardhat framework provides built-in tools for accurate gas cost tracking, enabling developers to make data-driven optimization decisions.

6 Conclusion

6.1 Key Findings

This research demonstrates that algorithmic complexity directly translates to financial costs in EVM-based smart contracts. Our experimental results show:

(1) **Significant Cost Variations**: Gas costs can vary by over 56% between naive and optimized implementations
(2) **Scalability Impact**: Cost differences compound with input size, making optimization critical for production deployments
(3) **Measurable Improvements**: Systematic application of complexity-aware design principles yields substantial gas savings

6.2 Theoretical Contribution

This work establishes a clear framework for understanding the relationship between computational complexity and economic cost in blockchain environments. Unlike traditional computing, where complexity affects performance, blockchain complexity directly impacts financial cost, creating a unique optimization landscape.

6.3 Practical Significance

The findings have immediate practical applications for:

- **Smart Contract Developers**: Design patterns and optimization strategies
- **DeFi Protocols**: Cost reduction for high-frequency operations
- **Blockchain Researchers**: Understanding of blockchain economics
- **Network Participants**: Awareness of resource consumption patterns

6.4 Future Research Directions

(1) **Extended Complexity Analysis**: Investigation of O(log n) and O(n log n) patterns
(2) **Gas Cost Prediction Models**: Machine learning models for gas cost estimation
(3) **Automated Optimization Tools**: Compiler-level optimizations based on complexity analysis
(4) **Cross-Chain Comparison**: Analysis of complexity-cost relationships in other blockchain platforms

7 References and Data

7.1 Experimental Data

All gas cost measurements were obtained through systematic testing using Hardhat's local network. The complete test suite and measurement scripts are available in the project repository.

7.2 Gas Cost Benchmarks

Operation Type	Gas Cost	Notes
Storage write (first)	20,000	Cold storage access
Storage write (subsequent)	5,000	Warm storage access
Storage read	2,100	Persistent storage
Memory write	3	Temporary storage
Memory read	3	Temporary storage
Function call	21 + 3/param	Base cost + parameters
Loop iteration	10-50	Depends on operations

7.3 Code Repository

The complete implementation, test suite, and measurement scripts are available in the project repository, enabling replication and extension of this research.

Report Generated: Based on empirical measurements conducted using Hardhat framework

Measurement Date: November 2025

Solidity Version: 0.8.20

EVM Target: Paris