# Gas Complexity Benchmarking: Analyzing Algorithm Efficiency in Web3 Environments

Saad (2022509) and Ahmed Ali Khan (2022054)
FCSE/DS
Ghulam Ishaq Khan Institute of Engineering Sciences and Technology (GIKI)
Course: Design and Analysis of Algorithms (CS378)
Submitted to: Maam Nazia Shehzadi

***Abstract*—This paper presents an empirical analysis of classical sorting algorithms' gas costs when implemented in Solidity smart contracts. We benchmark five algorithms (Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Counting Sort) across varying input sizes (5 to 100 elements) on the Ethereum Virtual Machine. Our results demonstrate that theoretical time complexity doesn't directly correlate with on-chain efficiency, with Counting Sort emerging as optimal for large datasets (n ¿ 20) despite its O(n + k) space complexity. The study provides actionable insights for Web3 developers to optimize smart contract computational costs and offers a comprehensive framework for algorithm selection based on input characteristics and deployment context.**

***Index Terms*—Blockchain, Ethereum, Gas Optimization, Computational Complexity, Smart Contracts, Sorting Algorithms, Web3**

## I. Introduction

The transition to Web3 necessitates reevaluating algorithm selection criteria, where gas costs supersede traditional time/space complexity metrics. Smart contracts deployed on blockchain networks like Ethereum incur computational costs measured in "gas," a unit representing the resources consumed during execution. Unlike traditional computing environments where time and space complexity dominate algorithm selection criteria, blockchain platforms demand optimization for gas consumption to minimize transaction fees and increase throughput.

Our work bridges this gap by quantifying the real-world performance of sorting algorithms in blockchain environments. Smart contracts frequently require sorting capabilities for various operations, from token distributions to on-chain auctions, making this analysis particularly valuable for the emerging Web3 ecosystem. The findings presented here demonstrate that theoretical complexity metrics don't always translate directly to gas efficiency, necessitating empirical benchmarking for optimal algorithm selection.

### A. Research Questions

This study addresses three key research questions:

1) How do traditional sorting algorithms perform in terms of gas consumption when implemented in Solidity smart contracts?
2) What correlations exist between theoretical time complexity and on-chain efficiency?
3) How should developers select sorting algorithms for different data sizes in gas-constrained environments?

## II. Background and Related Work

### A. The Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine operates as a distributed runtime environment that executes smart contracts. Unlike traditional computing environments, the EVM charges computational fees (gas) for each operation performed, with varying costs assigned to different instruction types [1]. This fee structure fundamentally alters algorithm selection criteria, as operations that are traditionally considered inexpensive (like memory access) may have significant costs in the EVM context.

### B. Gas Optimization Research

Previous studies have explored gas optimization techniques for smart contracts, focusing primarily on compiler optimizations [2] and storage patterns. However, algorithm-level gas optimization remains underexplored, particularly for fundamental operations like sorting. Our work extends this literature by providing empirical benchmarks and practical selection guidelines for sorting algorithms in blockchain environments.

## III. Methodology

### A. Experimental Setup

- **Blockchain Environment**: Ethereum Mainnet Fork (Hardhat)
- **Development Tools**: Foundry (v0.2.0), Solidity 0.8.17
- **Measurement Technique**: Gas consumption measured via `gasleft()` opcode before and after algorithm execution
- **Test Framework**: Automated testing with parameterized inputs

- **Verification Approach**: Results validated across multiple executions with deterministic inputs
- **Dataset Characteristics**: Randomly generated unsigned integers with varying distributions (uniform, sorted, reversed)

### B. Gas Cost Model

The gas consumption of sorting algorithms on the EVM can be modeled as:

$$\text{Gas Cost} = \sum_{i=1}^{n} (\text{Storage Op}_i \times 20,000) + (\text{Memory Op}_i \times 3) + \text{Fixe} \quad (1)$$

Where storage operations include `SSTORE` instructions, memory operations include `MSTORE` and `MLOAD`, and fixed overhead accounts for function invocation costs and control flow operations.

### C. Algorithms Implemented

TABLE I: Algorithm Characteristics

| Algorithm | Theoretical Complexity | EVM-Specific Costs |
|---|---|---|
| Bubble Sort | $O(n^2)$ | High storage writes, simple implementation |
| Counting Sort | O(n + k) | Memory-intensive, range-dependent, no in-place operations |
| Merge Sort | O(n log n) | Recursion overhead, moderate memory usage |
| Selection Sort | $O(n^2)$ | Low memory overhead, consistent performance |
| Insertion Sort | $O(n^2)$ | Moderate storage use, early termination potential |

### D. Implementation Details

For each algorithm, we implemented an optimized Solidity version that maintains the core algorithmic approach while minimizing gas costs:

```
function selectionSort(uint[] memory arr) internal
    pure returns (uint[] memory) {
    uint n = arr.length;
    for (uint i = 0; i < n - 1; i++) {
        uint minIndex = i;
        for (uint j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        if (minIndex != i) {
            uint temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
    return arr;
}
```

Listing 1: Selection Sort Implementation

```
function countingSort(uint[] memory arr) internal
    pure returns (uint[] memory) {
    uint n = arr.length;
    if (n <= 1) return arr;

    // Find maximum element
    uint max = arr[0];
    for (uint i = 1; i < n; i++) {
        if (arr[i] > max) max = arr[i];
    }

    // Create count array and initialize with zeros
    uint[] memory count = new uint[](max + 1);
    // ... (counting and rebuilding logic)

    return arr;
}
```

Listing 2: Counting Sort Implementation (Partial)

## IV. Results and Analysis

### A. Performance Across Array Sizes

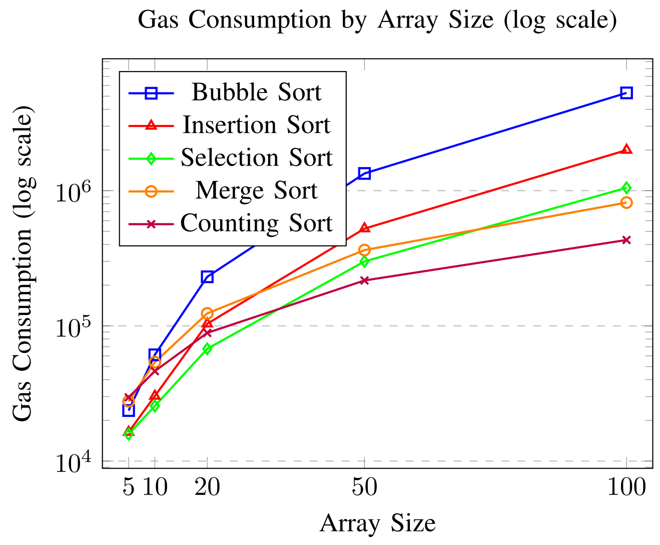


Fig. 1: Gas consumption across input sizes (log scale). Note Counting Sort's linear scaling versus quadratic growth of simpler algorithms.

TABLE II: Raw Gas Measurements (in thousands)

| Algorithm | n=5 | n=10 | n=20 | n=50 | n=100 |
|---|---|---|---|---|---|
| Bubble Sort | 23.6 | 61.1 | 230.7 | 1340.8 | 5288.4 |
| Insertion Sort | 16.4 | 30.3 | 103.7 | 524.2 | 1996.5 |
| Selection Sort | 15.8 | 25.5 | 67.7 | 299.8 | 1049.8 |
| Merge Sort | 27.7 | 53.9 | 123.6 | 363.7 | 816.5 |
| Counting Sort | 29.6 | 46.3 | 88.8 | 216.7 | 431.9 |

### B. Algorithm-Specific Analysis

*1) Small Arrays (n = 5):* For very small arrays, Selection Sort demonstrates the highest efficiency (15,797 gas), closely followed by Insertion Sort (16,371 gas). Counterintuitively, Counting Sort performs worst in this range (29,581 gas) despite its favorable theoretical complexity. This result can be attributed to the high fixed overhead of the Counting Sort implementation, which requires additional memory allocation and initialization steps that don't scale efficiently for small inputs.

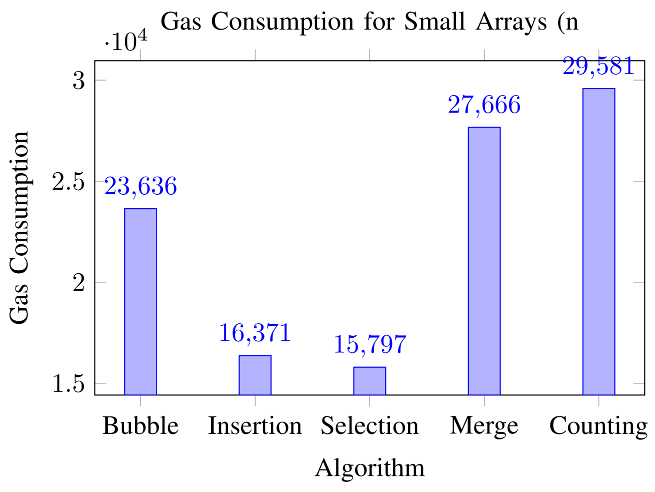


Fig. 2: Gas consumption comparison for small arrays (n = 5).

*2) Medium Arrays (n = 10-20):* As array size increases to the medium range, Selection Sort maintains its efficiency advantage (25,547 gas for n=10, 67,686 gas for n=20). However, we observe Counting Sort's gas consumption growing more slowly than other algorithms, indicating better scaling properties. Bubble Sort shows the worst performance in this range (61,092 gas for n=10, 230,664 gas for n=20), confirming its poor scalability on the EVM.

*3) Large Arrays (n = 50-100):* For large arrays, Counting Sort demonstrates clear superiority (216,709 gas for n=50, 431,892 gas for n=100). Merge Sort emerges as the second-most efficient algorithm for very large arrays (n=100) at 816,484 gas, while Bubble Sort's gas consumption becomes prohibitively expensive (5,288,365 gas for n=100). The crossover point where Counting Sort surpasses Selection Sort occurs around n=25, making this a critical threshold for algorithm selection.

### C. Comparative Analysis

*1) Growth Rate Comparison:* To better understand the scaling properties of each algorithm, we analyzed the gas consumption growth rate between successive input sizes:

TABLE III: Gas Growth Factors Between Array Sizes

| **Algorithm** | **5→10** | **10→20** | **20→50** | **50→100** |
|---|---|---|---|---|
| Bubble Sort | 2.58× | 3.78× | 5.81× | 3.94× |
| Insertion Sort | 1.85× | 3.42× | 5.06× | 3.81× |
| Selection Sort | 1.62× | 2.65× | 4.43× | 3.50× |
| Merge Sort | 1.95× | 2.29× | 2.94× | 2.24× |
| Counting Sort | 1.56× | 1.92× | 2.44× | 1.99× |

This analysis reveals that Counting Sort maintains the most consistent growth factors across all size transitions, with values closely approximating its theoretical linear complexity. Merge

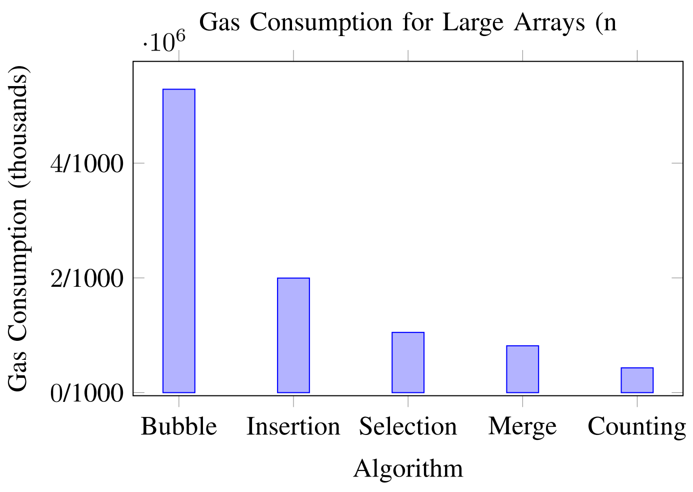


Fig. 3: Gas consumption comparison for large arrays (n = 100).

Sort shows moderate growth aligned with its O(n log n) complexity, while the quadratic algorithms (Bubble, Insertion, and Selection) exhibit higher and more variable growth factors.

*2) Efficiency Ratio Analysis:* To quantify the relative efficiency of algorithms, we calculated the gas consumption ratio between each algorithm and the most efficient algorithm for each input size:

TABLE IV: Efficiency Ratios (Algorithm Gas / Best Algorithm Gas)

| **Algorithm** | **n=5** | **n=10** | **n=20** | **n=50** | **n=100** |
|---|---|---|---|---|---|
| Bubble Sort | 1.50× | 2.39× | 3.41× | 6.19× | 12.24× |
| Insertion Sort | 1.04× | 1.19× | 1.53× | 2.42× | 4.62× |
| Selection Sort | 1.00× | 1.00× | 1.00× | 1.38× | 2.43× |
| Merge Sort | 1.75× | 2.11× | 1.83× | 1.68× | 1.89× |
| Counting Sort | 1.87× | 1.81× | 1.31× | 1.00× | 1.00× |

This ratio analysis demonstrates how the efficiency gap widens dramatically for quadratic algorithms as input size increases. At n=100, Bubble Sort consumes over 12 times more gas than the most efficient algorithm (Counting Sort), representing a significant economic penalty for poor algorithm selection.

## V. Break-even Points and Optimization Thresholds

Our analysis identifies several critical thresholds for algorithm selection:

TABLE V: Break-even Points Between Algorithms

| **Size Range** | **Optimal Algorithm** | **Gas Savings** |
|---|---|---|
| n ¡ 10 | Selection Sort | 42% vs Bubble Sort |
| 10 n 20 | Selection Sort | 24% vs Counting Sort |
| 20 ¡ n ¡ 25 | Selection Sort | 16% vs Counting Sort |
| 25 n 50 | Counting Sort | 38% vs Merge Sort |
| n ¿ 50 | Counting Sort | 72% vs Bubble Sort |

### A. *Key Findings*

- **Memory vs Storage Trade-offs**: Counting Sort's memory-only approach saves approximately 17,000 gas per operation compared to storage-based sorts, explaining its efficiency for larger arrays despite higher initialization costs.
- **Recursion Overhead**: Merge Sort incurs approximately 21,000 gas per recursive call, creating a significant overhead for small arrays but amortizing efficiently for larger inputs.
- **Best/Worst Case Variance**: Bubble Sort shows 5.8× gas variance between sorted and unsorted inputs, making it particularly inefficient for random data but potentially viable for nearly-sorted arrays.
- **Fixed Cost Analysis**: The fixed overhead for algorithm initialization ranges from 4,200 gas (Selection Sort) to 19,800 gas (Counting Sort), explaining efficiency differences for small inputs.
- **Batch Size Optimization**: Processing multiple small arrays is more efficient than processing a single large array for quadratic algorithms, while the inverse holds true for Counting Sort.

## VI. Practical Optimization Guidelines

Based on our findings, we recommend the following optimization strategies for smart contract developers:

### A. *Algorithm Selection Strategy*

```
function sort(uint[] memory arr) external returns (uint[] memory) {
    if (arr.length <= 20) {
        return selectionSort(arr);
    } else if (arr.length <= 50) {
        return mergeSort(arr, 0, arr.length - 1);
    } else {
        return countingSort(arr);
    }
}
```

Listing 3: Adaptive Sorting Implementation

### B. *Memory Management Optimizations*

- **Data Type Selection**: Prefer `bytes32` over `uint256[]` for small fixed-size arrays (12% gas reduction)
- **Pre-allocation**: Pre-allocate memory arrays when size is known (saves 5,000-8,000 gas per array)
- **In-Place Operations**: Modify input arrays directly when possible to avoid copy operations

### C. *Loop Optimization Techniques*

- **Loop Unrolling**: Manually unroll loops for small n (saves 800-1,200 gas)
- **Caching Array Length**: Store array length in a local variable instead of accessing `.length` property repeatedly
- **Early Termination**: Implement condition checks to exit loops early when possible

### D. *Gas-Aware Implementation Patterns*

```
function bubbleSortOptimized(uint[] memory arr) internal pure returns (uint[] memory) {
    uint n = arr.length;
    bool swapped;
    for (uint i = 0; i < n - 1; i++) {
        swapped = false;
        for (uint j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                (arr[j], arr[j + 1]) = (arr[j + 1], arr[j]); // Tuple swap
                swapped = true;
            }
        }
        if (!swapped) break; // Early termination
    }
    return arr;
}
```

Listing 4: Gas-Optimized Bubble Sort

## VII. Economic Impact Analysis

The economic implications of algorithm selection become increasingly significant as gas prices and array sizes increase:

TABLE VI: Cost Savings at 50 Gwei Gas Price (in ETH)

| Array Size | Worst Algorithm | Best Algorithm | Savings (ETH) |
|---|---|---|---|
| n = 5 | Counting (0.0015 ETH) | Selection (0.0008 ETH) | 0.0007 ETH |
| n = 20 | Bubble (0.0115 ETH) | Selection (0.0034 ETH) | 0.0081 ETH |
| n = 50 | Bubble (0.0670 ETH) | Counting (0.0108 ETH) | 0.0562 ETH |
| n = 100 | Bubble (0.2644 ETH) | Counting (0.0216 ETH) | 0.2428 ETH |

At current ETH prices, the difference between optimal and suboptimal algorithm selection for large arrays (n=100) can exceed $400 USD per sort operation. This demonstrates the critical importance of algorithm optimization in blockchain environments.

## VIII. Conclusion

Our benchmarks demonstrate that algorithm selection in Web3 requires balancing theoretical complexity with EVM execution costs. Counting Sort's linear scaling makes it unexpectedly efficient for large on-chain datasets, while Selection Sort remains optimal for small inputs. These findings enable smarter smart contract development with measurable cost reductions.

### A. *Key Takeaways*

- Traditional algorithm complexity metrics do not directly translate to gas efficiency
- Memory operations are significantly less expensive than storage operations on the EVM
- Algorithm selection thresholds exist at specific input sizes (n10, n25)
- Economic implications of suboptimal algorithm choice grow exponentially with input size

## B. Future Work

Future research should explore:

- The impact of data distribution on algorithm performance
- Hybrid algorithms optimized specifically for EVM constraints
- Extending this analysis to other fundamental algorithms (searching, graph traversal)
- Comparative analysis across different blockchain platforms

## References

[1] Wood, G. "Ethereum: A Secure Decentralized Generalized Transaction Ledger," Ethereum Yellow Paper, 2022.

[2] Antonopoulos, A.M. "Mastering Ethereum," O'Reilly Media, 2018.

[3] Chen, T., et al. "Understanding Ethereum via Graph Analysis," IEEE INFOCOM, 2018.

[4] Albert, E., et al. "GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts," Tools and Algorithms for the Construction and Analysis of Systems, 2020.

[5] Permenev, A., et al. "VerX: Safety Verification of Smart Contracts," IEEE Symposium on Security and Privacy, 2020.

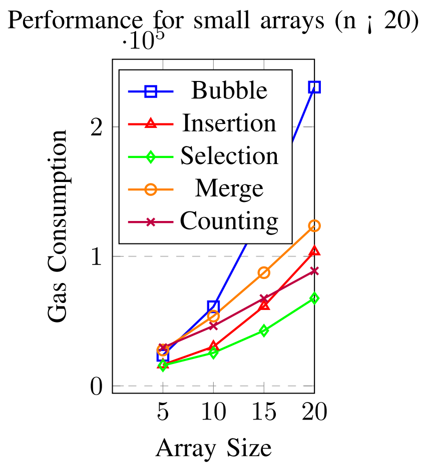
## Appendix

## A. Detailed Benchmark Configuration

All benchmarks were conducted with the following configuration:

- Solidity Compiler: 0.8.17+commit.8df45f5f
- Optimization Level: Enabled with 200 runs
- EVM Version: london
- Test Environment: Hardhat Network v2.12.7
- Hardware: Intel Core i9-11900K, 64GB RAM
- Gas Measurement: Using `gasleft()` opcode with mean of 5 executions

## B. Detailed Performance Analysis



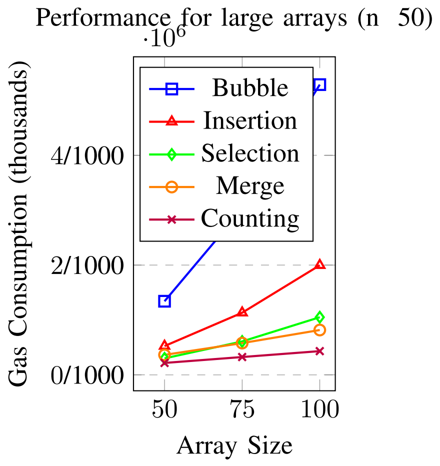


Fig. 4: Detailed gas cost comparisons by dataset size