SORTING ALGORITHMS

Complete Performance & Logic Analysis

PREPARED BY

Computer Science Department

Algorithms Report

January 30, 2026

Table of Contents

1. Executive Summary

Key Findings

7 algorithms analyzed with detailed best/worst case logic. Quick Sort & Heap Sort: O(n log n). Radix & Bucket: best for specific distributions. 21 pairwise comparisons evaluate speed, space, and stability with custom visualizations.

Recommendations

- General purpose: Quick Sort (random pivot)
- Guaranteed perf: Heap Sort
- Integers: Radix Sort
- Small/nearly sorted: Optimized Bubble Sort

2. Algorithm Analysis

Detailed best/worst case analysis with logic explanations and usage guidance.

Bubble Sort

Avg: O(n^2) | Space: O(1)

BEST: O(n)

Elements are in correct order; inner loop detects no swaps and terminates.

Ex: [1, 2, 3, 4, 5] - Already sorted

WORST: O(n^2)

Every adjacent pair is inverted; maximum swaps (n^2/2) occurs.

Ex: [5, 4, 3, 2, 1] - Reverse sorted

USE: Small datasets (<50 elements), nearly sorted data, educational pu

AVOID: Large datasets, performance-critical applications

Comparison-based | STABLE

Bubble Sort (Optimized)

Avg: O(n^2) | Space: O(1)

BEST: O(n)

The "swapped" flag remains false after first pass, breaking loop immediately.

Ex: [1, 2, 3, 5, 4] - One swap away

WORST: O(n^2)

Swaps occur in every pass; flag is always true, forcing full n-1 passes.

Ex: [5, 4, 3, 2, 1] - No early termi

USE: Nearly sorted data where early termination is beneficial

AVOID: Random or reverse-sorted data

Comparison-based | STABLE

Gnome Sort

Avg: O(n^2) | Space: O(1)

BEST: O(n)

Pointer moves forward monotonically without backtracking.

Ex: [1, 2, 3, 4, 5] - Linear traversal

WORST: O(n^2)

Pointer moves one step forward, then i steps back for each element.

Ex: [5, 4, 3, 2, 1] - Many backward

USE: Simple implementation needs, very small datasets

AVOID: Any performance-sensitive application

Comparison-based | STABLE

Radix Sort

Avg: O(nk) | Space: O(n+k)

BEST: O(nk)

Complexity is O(n*k). If k=1 (single digits), time is O(n).

Ex: [5, 2, 8, 1, 9] - Single digits (k=

WORST: O(nk)

6-digit numbers require 6 passes, each processing all n elements.

Ex: [999999, 888888, 777777] - k=6

USE: Large datasets of integers with bounded digit count

AVOID: Floating-point numbers, strings, small datasets

Non-comparison | STABLE

Quick Sort

Avg: O(n log n) | Space: O(log n)

BEST: O(n log n)

Balanced partitions: T(n) = 2T(n/2) + O(n) = O(n log n).

Ex: Random data with good pivot selecti

WORST: O(n^2)

Unbalanced partitions: T(n) = T(n-1) + O(n) = O(n^2).

Ex: [1, 2, 3, 4, 5] with first-eleme

USE: General purpose sorting, large random datasets

AVOID: Already sorted data (without randomized pivot), stability require

Divide and Conquer | UNSTABLE

Heap Sort

Avg: O(n log n) | Space: O(1)

BEST: O(n log n)

Heap operations always O(log n). Building heap is O(n).

Ex: Any array - performance is consiste

WORST: O(n log n)

Heap structure guarantees log(n) height for all inputs.

Ex: Any array - no bad inputs exist

USE: Guaranteed O(n log n), memory constraints

AVOID: When stability is required or cache performance matters

Selection-based | UNSTABLE

Bucket Sort

Avg: O(n+k) | Space: O(n+k)

BEST: O(n+k)

Uniform distribution: ~1 element per bucket, sorting is O(1) each.

Ex: [0.1, 0.3, 0.5, 0.7, 0.9] - Uniform

WORST: O(n^2)

Single bucket has all n elements, inner sort takes O(n^2).

Ex: [0.11, 0.12, 0.13, 0.14] - All i

USE: Uniformly distributed data in known range

AVOID: Unknown distribution, clustered data

Distribution-based

STABLE

3. Pairwise Comparison Matrix

GREEN=Row wins, RED=Column wins, YELLOW=Tie (based on speed+space scores)

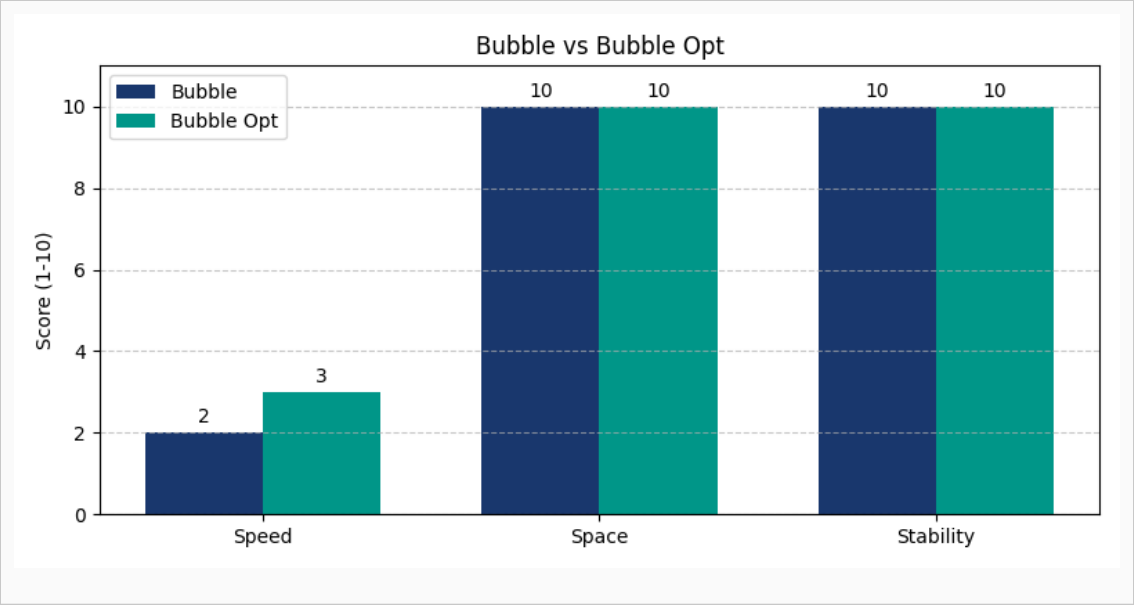
Pairwise Comparison Matrix

	Bubble	Bubble	Gnome	Radix	Quick	Heap	Bucket
Bubble	-	LOSE	LOSE	LOSE	LOSE	LOSE	LOSE
Bubble	WIN	-	WIN	LOSE	LOSE	LOSE	LOSE
Gnome	WIN	LOSE	-	LOSE	LOSE	LOSE	LOSE
Radix	WIN	WIN	WIN	-	LOSE	LOSE	LOSE
Quick	WIN	WIN	WIN	WIN	-	LOSE	WIN
Heap	WIN	WIN	WIN	WIN	WIN	-	WIN
Bucket	WIN	WIN	WIN	WIN	LOSE	LOSE	-

4. Detailed Pairwise Analysis

All 21 pairs with visual metric comparisons and scenario-based recommendations.

Bubble *VS* **Bubble Opt**



Speed: Bubble Opt **Space:** Tie **Stable:** Both

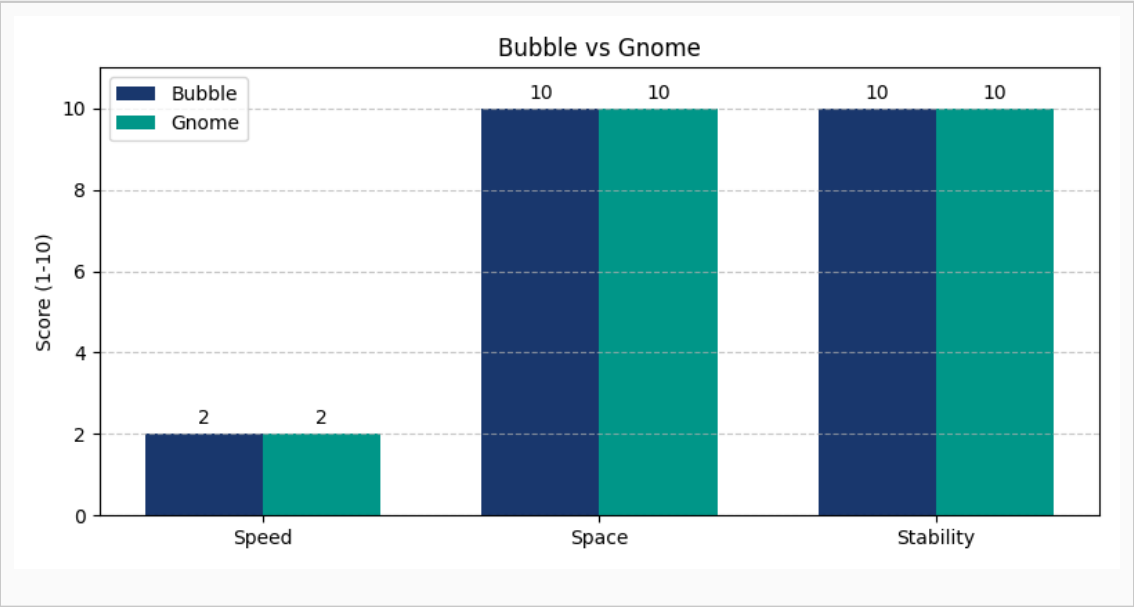
Best: Bubble: O(n) vs Bubble Opt: O(n)

Worst: Bubble: O(n^2) vs Bubble Opt: O(n^2)

Scenario Analysis:

Small Data:	Tie	*(Both acceptable)*
Large Data:	Neither	*(Both O(n^2))*
Memory Limited:	Bubble Opt	*(O(1) vs O(1))*

Bubble	*VS*	Gnome



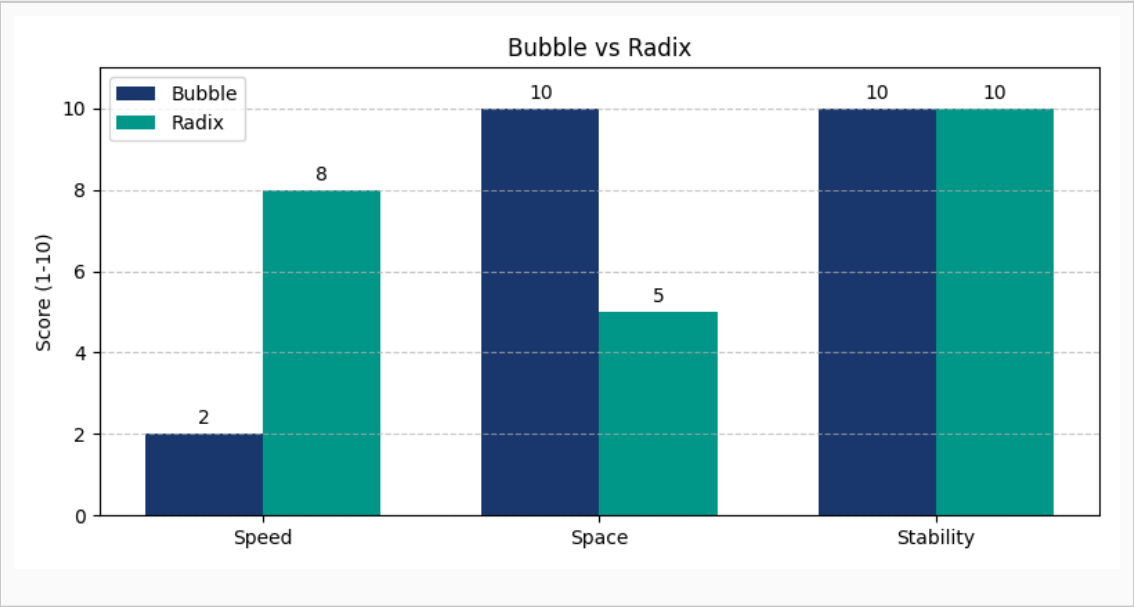
Speed: Tie **Space:** Tie **Stable:** Both

Best: Bubble: O(n) vs Gnome: O(n) Worst: Bubble: O(n^2) vs Gnome: O(n^2)

Scenario Analysis:

Code Size:	**Gnome**	*(Shorter (1 loop))*
Near-Sorted:	**Bubble**	*(Early termination)*
Embedded:	**Gnome**	*(Low instruction mem)*

Bubble	VS	Radix



Speed: Radix **Space:** Bubble **Stable:** Both

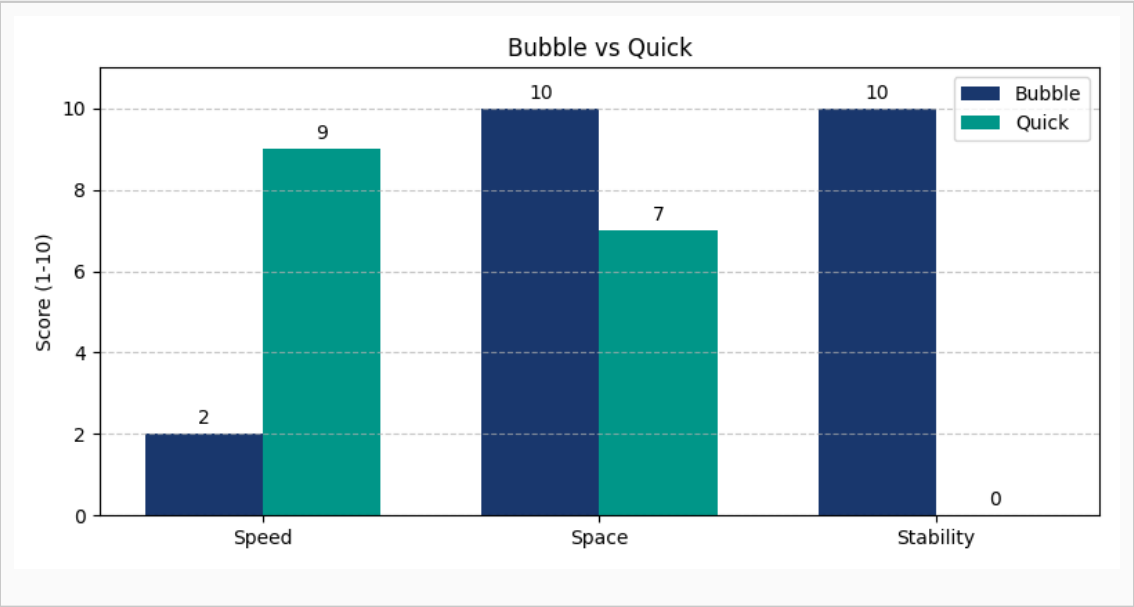
Best: Bubble: O(n) vs Radix: O(nk)

Worst: Bubble: O(n^2) vs Radix: O(nk)

Scenario Analysis:

Small Data:	**Bubble**	*(Simpler)*
Large Data:	**Radix**	*(O(n log n))*
Memory Limited:	**Bubble**	*(O(1) vs O(n+k))*

Bubble	VS	Quick



Speed: Quick **Space:** Bubble **Stable:** Bubble

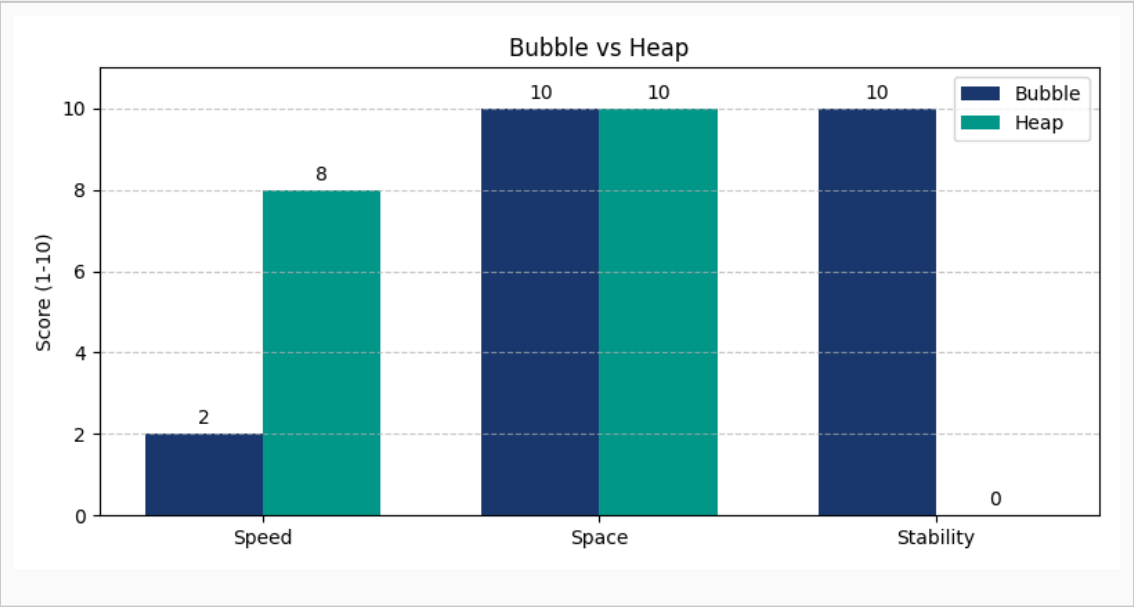
Best: Bubble: O(n) vs Quick: O(n log n)

Worst: Bubble: O(n^2) vs Quick: O(n^2)

Scenario Analysis:

Small Data:	**Bubble**	*(Simpler)*
Large Data:	**Quick**	*(O(n log n))*
Memory Limited:	**Bubble**	*(O(1) vs O(log n))*

Bubble	*VS*	Heap



Speed: Heap **Space:** Tie **Stable:** Bubble

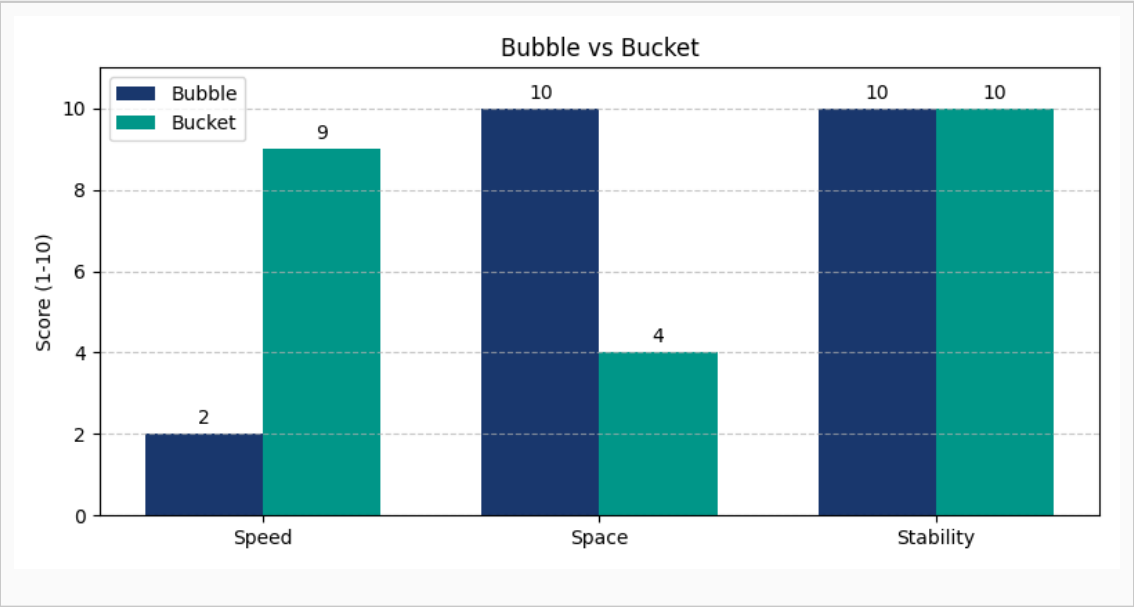
Best: Bubble: O(n) vs Heap: O(n log n)

Worst: Bubble: O(n^2) vs Heap: O(n log n)

Scenario Analysis:

Small Data:	**Bubble**	*(Simpler)*
Large Data:	**Heap**	*(O(n log n))*
Memory Limited:	**Heap**	*(O(1) vs O(1))*

Bubble	VS	Bucket



Speed: Bucket **Space:** Bubble **Stable:** Both

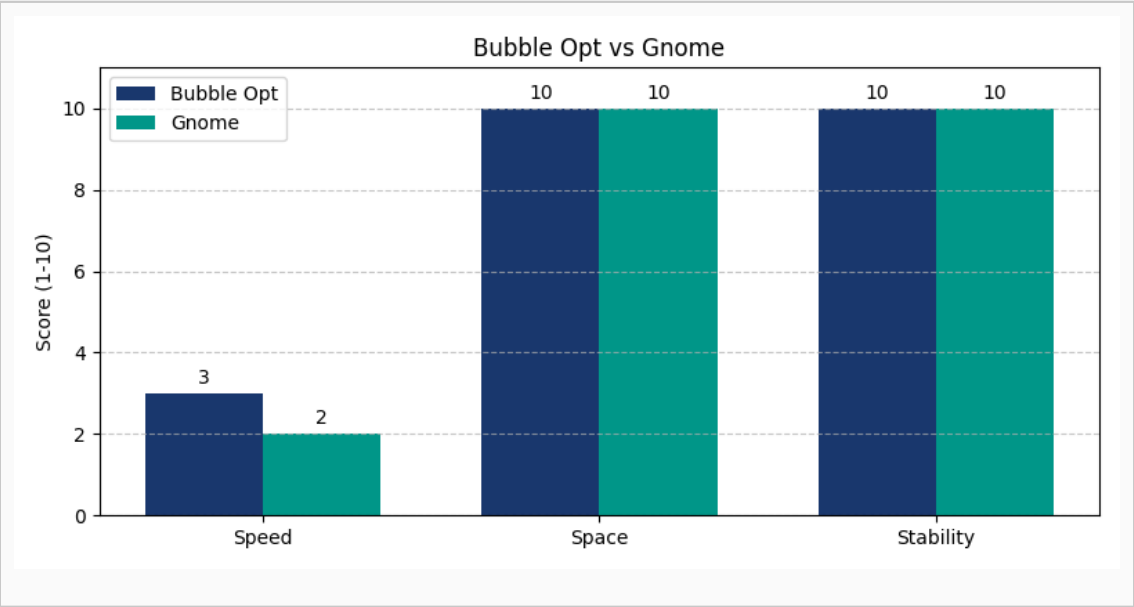
Best: Bubble: O(n) vs Bucket: O(n+k)

Worst: Bubble: O(n^2) vs Bucket: O(n^2)

Scenario Analysis:

Small Data:	**Bubble**	*(Simpler)*
Large Data:	**Bucket**	*(O(n log n))*
Memory Limited:	**Bubble**	*(O(1) vs O(n+k))*

Bubble Opt	*VS*	Gnome



Speed: Bubble Opt **Space:** Tie **Stable:** Both

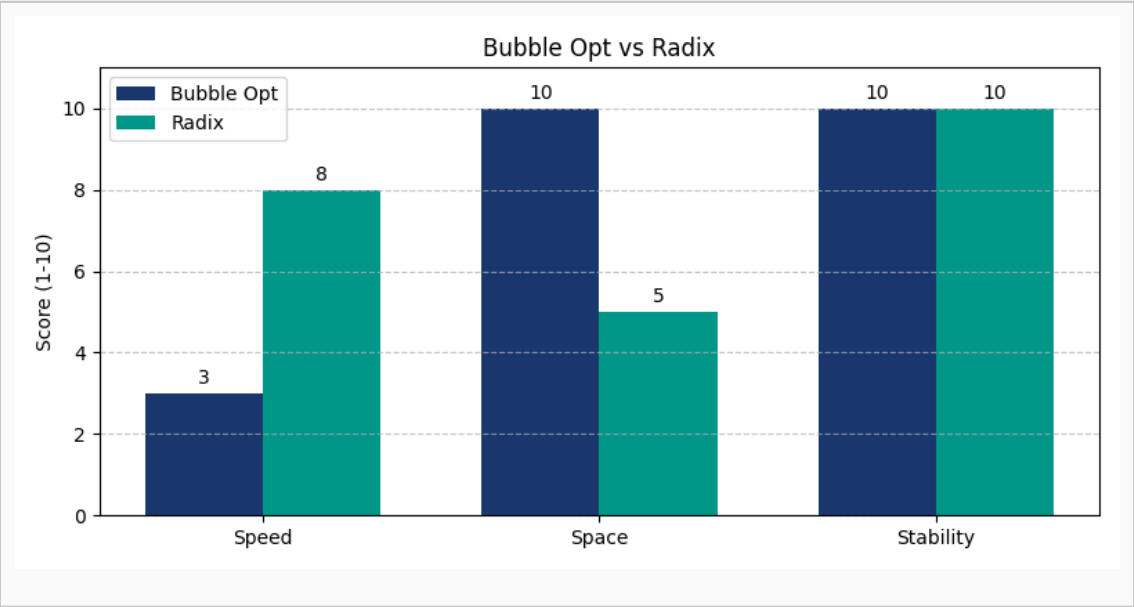
Best: Bubble Opt: O(n) vs Gnome: O(n)

Worst: Bubble Opt: O(n^2) vs Gnome: O(n^2)

Scenario Analysis:

Small Data:	Tie	*(Both acceptable)*
Large Data:	Neither	*(Both O(n^2))*
Memory Limited:	Gnome	*(O(1) vs O(1))*

Bubble Opt	*VS*	Radix



Speed: Radix **Space:** Bubble Opt **Stable:** Both

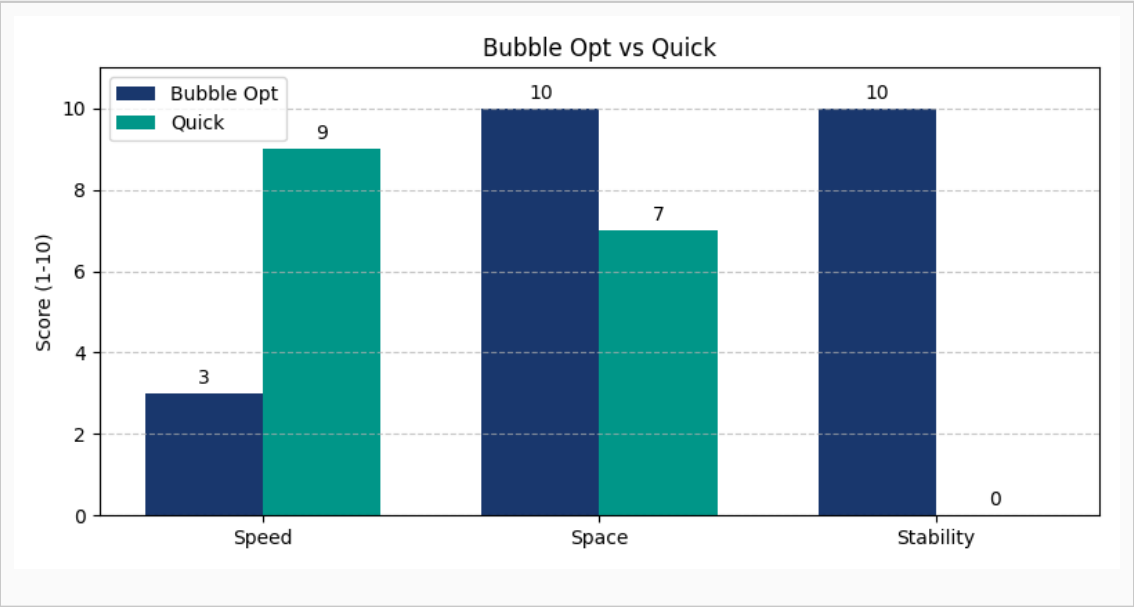
Best: Bubble Opt: O(n) vs Radix: O(nk)

Worst: Bubble Opt: O(n^2) vs Radix: O(nk)

Scenario Analysis:

Small Data:	**Bubble Opt**	*(Simpler)*
Large Data:	**Radix**	*(O(n log n))*
Memory Limited:	**Bubble Opt**	*(O(1) vs O(n+k))*

Bubble Opt	*VS*	Quick



Speed: Quick **Space:** Bubble Opt **Stable:** Bubble Opt

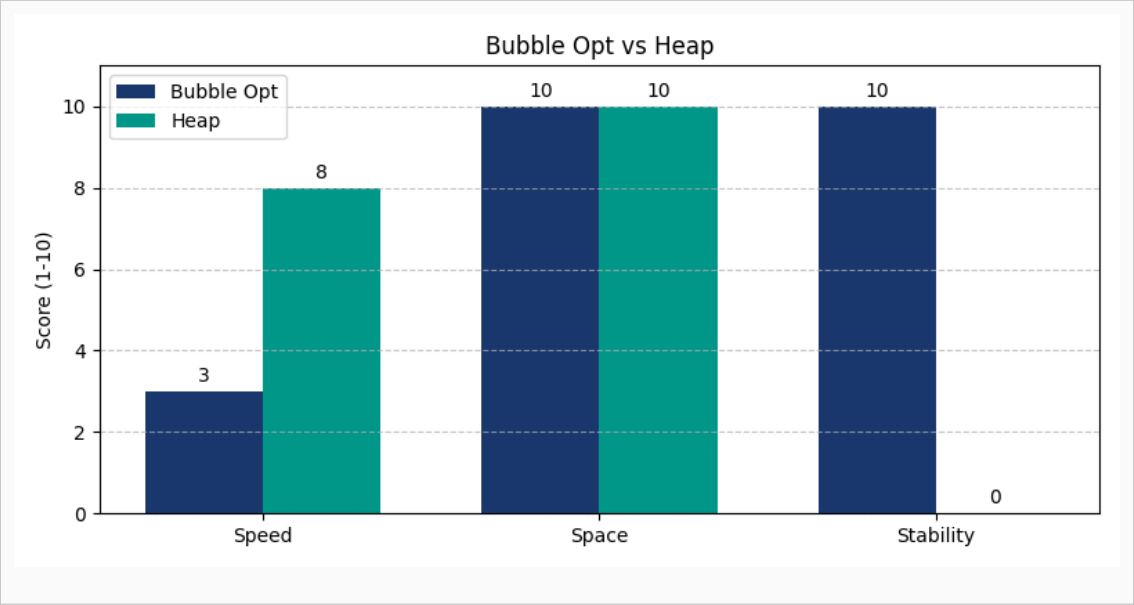
Best: Bubble Opt: O(n) vs Quick: O(n log n)

Worst: Bubble Opt: O(n^2) vs Quick: O(n^2)

Scenario Analysis:

Small Data:	**Bubble Opt**	*(Simpler)*
Large Data:	**Quick**	*(O(n log n))*
Memory Limited:	**Bubble Opt**	*(O(1) vs O(log n))*

Bubble Opt	*VS*	Heap



Speed: Heap **Space:** Tie **Stable:** Bubble Opt

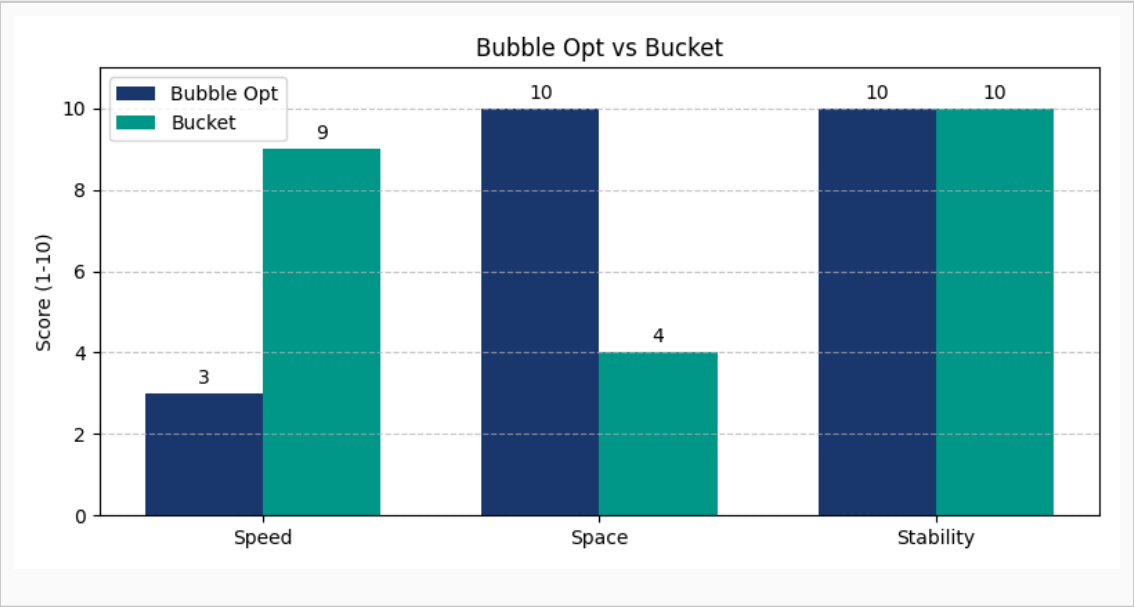
Best: Bubble Opt: O(n) vs Heap: O(n log n)

Worst: Bubble Opt: O(n^2) vs Heap: O(n log n)

Scenario Analysis:

Small Data:	**Bubble Opt**	*(Simpler)*
Large Data:	**Heap**	*(O(n log n))*
Memory Limited:	**Heap**	*(O(1) vs O(1))*

Bubble Opt	*VS*	Bucket



Speed: Bucket **Space:** Bubble Opt **Stable:** Both

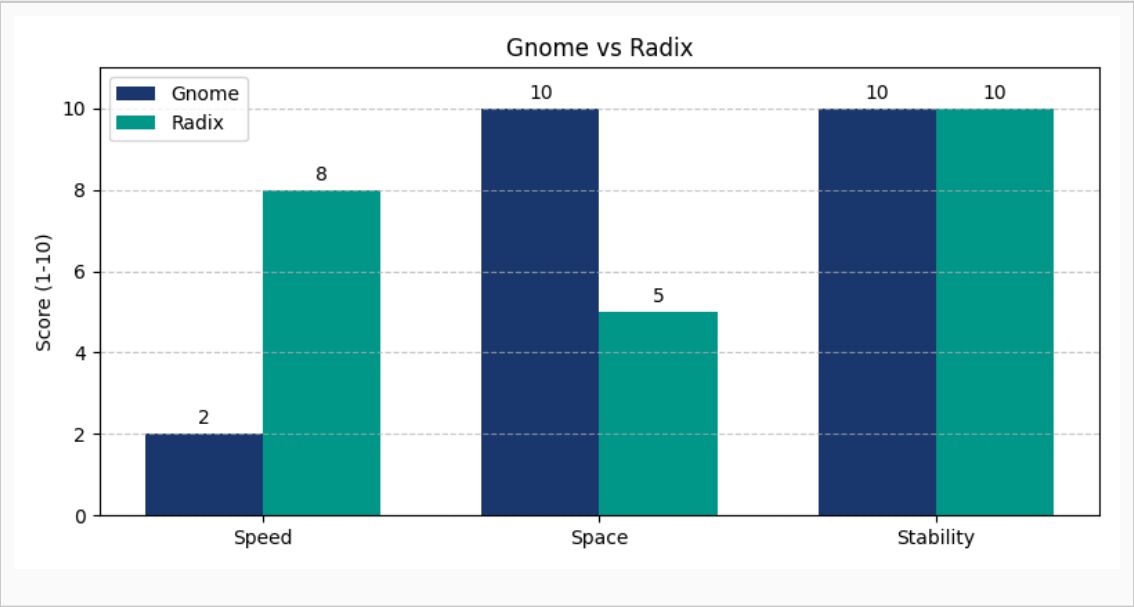
Best: Bubble Opt: O(n) vs Bucket: O(n+k)

Worst: Bubble Opt: O(n^2) vs Bucket: O(n^2)

Scenario Analysis:

Small Data:	**Bubble Opt**	*(Simpler)*
Large Data:	**Bucket**	*(O(n log n))*
Memory Limited:	**Bubble Opt**	*(O(1) vs O(n+k))*

Gnome	*VS*	Radix



Speed: Radix **Space:** Gnome **Stable:** Both

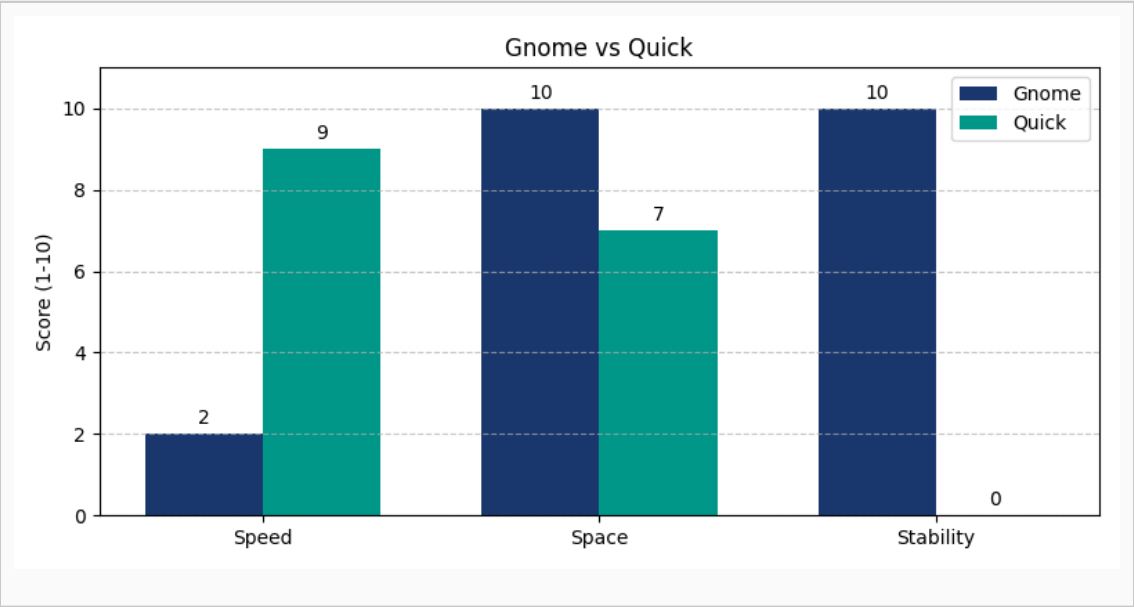
Best: Gnome: O(n) vs Radix: O(nk)

Worst: Gnome: O(n^2) vs Radix: O(nk)

Scenario Analysis:

Small Data:	**Gnome**	*(Simpler)*
Large Data:	**Radix**	*(O(n log n))*
Memory Limited:	**Gnome**	*(O(1) vs O(n+k))*

Gnome	*VS*	Quick



Speed: Quick **Space:** Gnome **Stable:** Gnome

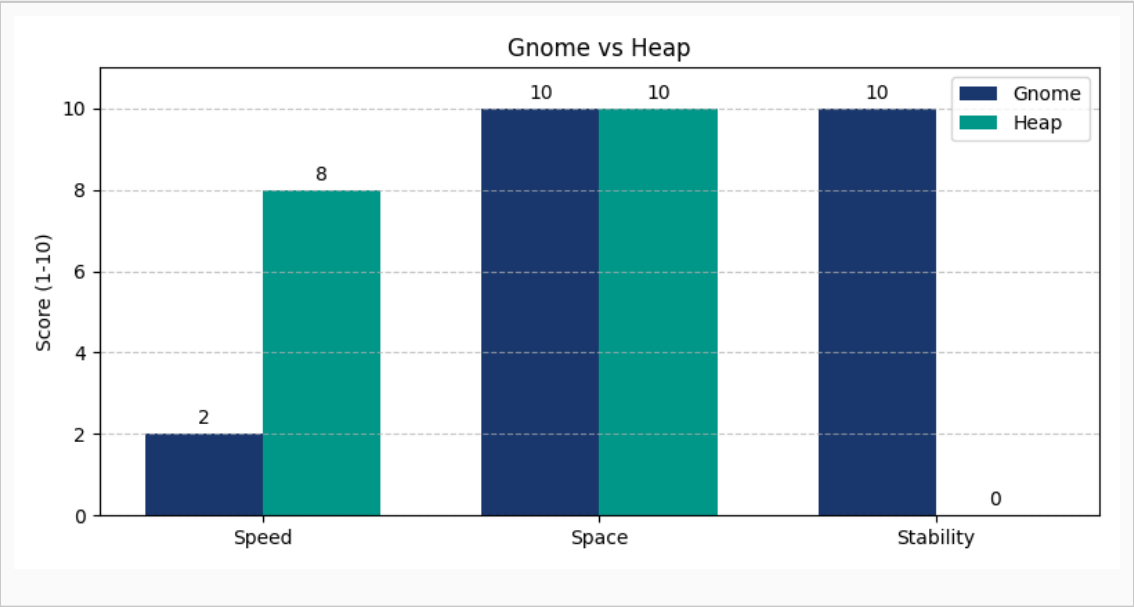
Best: Gnome: O(n) vs Quick: O(n log n)

Worst: Gnome: O(n^2) vs Quick: O(n^2)

Scenario Analysis:

Small Data:	**Gnome**	*(Simpler)*
Large Data:	**Quick**	*(O(n log n))*
Memory Limited:	**Gnome**	*(O(1) vs O(log n))*

Gnome	*VS*	Heap



Speed: Heap **Space:** Tie **Stable:** Gnome

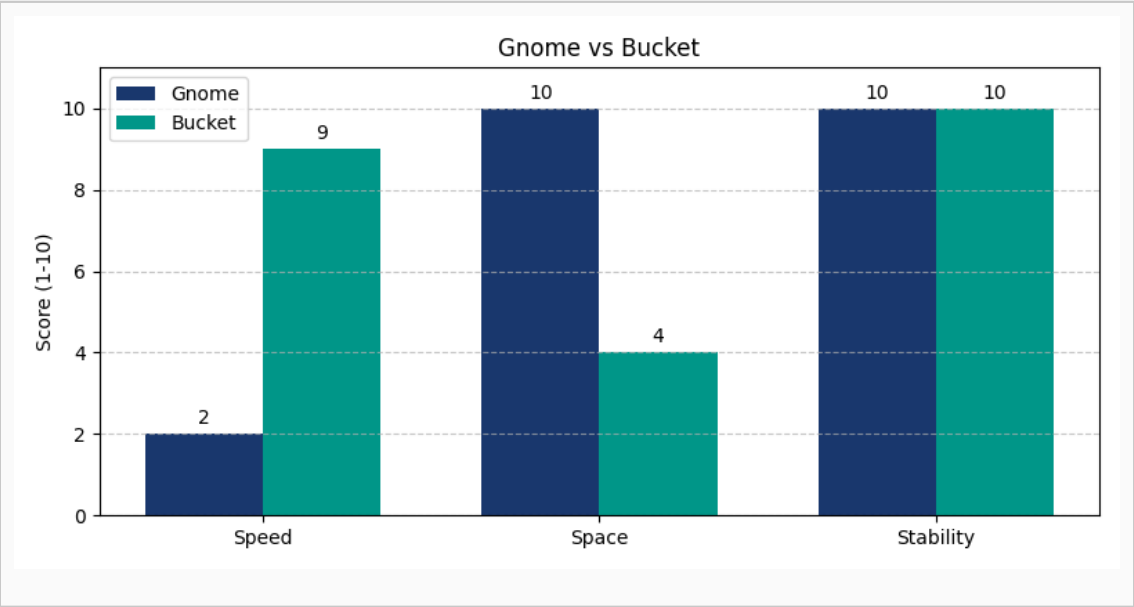
Best: Gnome: O(n) vs Heap: O(n log n)

Worst: Gnome: O(n^2) vs Heap: O(n log n)

Scenario Analysis:

Small Data:	**Gnome**	*(Simpler)*
Large Data:	**Heap**	*(O(n log n))*
Memory Limited:	**Heap**	*(O(1) vs O(1))*

Gnome	*VS*	Bucket



Speed: Bucket **Space:** Gnome **Stable:** Both

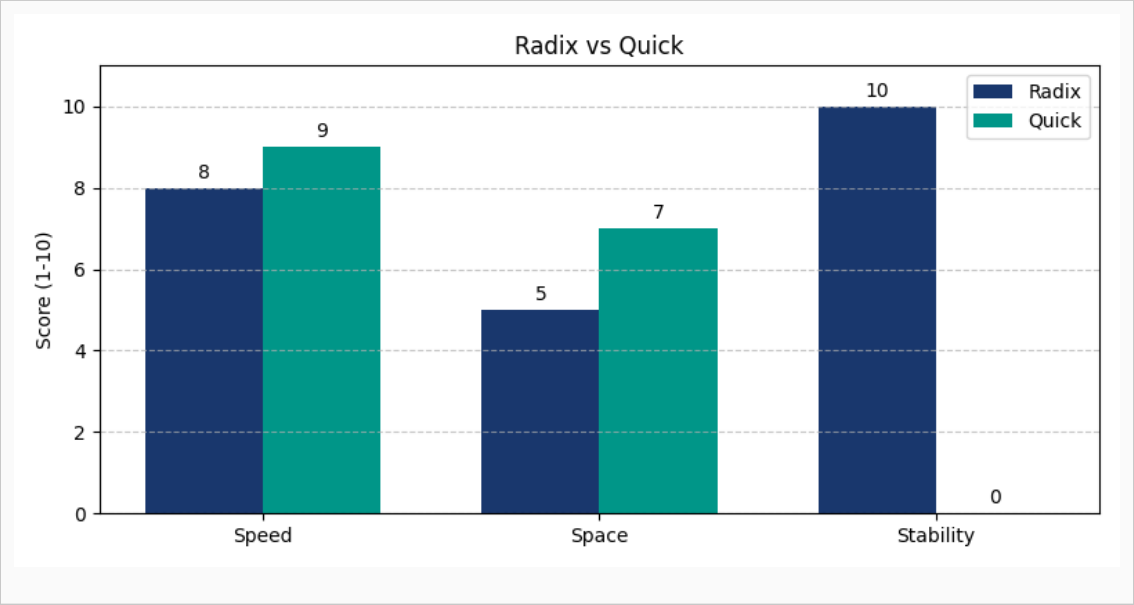
Best: Gnome: O(n) vs Bucket: O(n+k)

Worst: Gnome: O(n^2) vs Bucket: O(n^2)

Scenario Analysis:

Small Data:	**Gnome**	*(Simpler)*
Large Data:	**Bucket**	*(O(n log n))*
Memory Limited:	**Gnome**	*(O(1) vs O(n+k))*

Radix	*VS*	Quick



Speed: Quick **Space:** Quick **Stable:** Radix

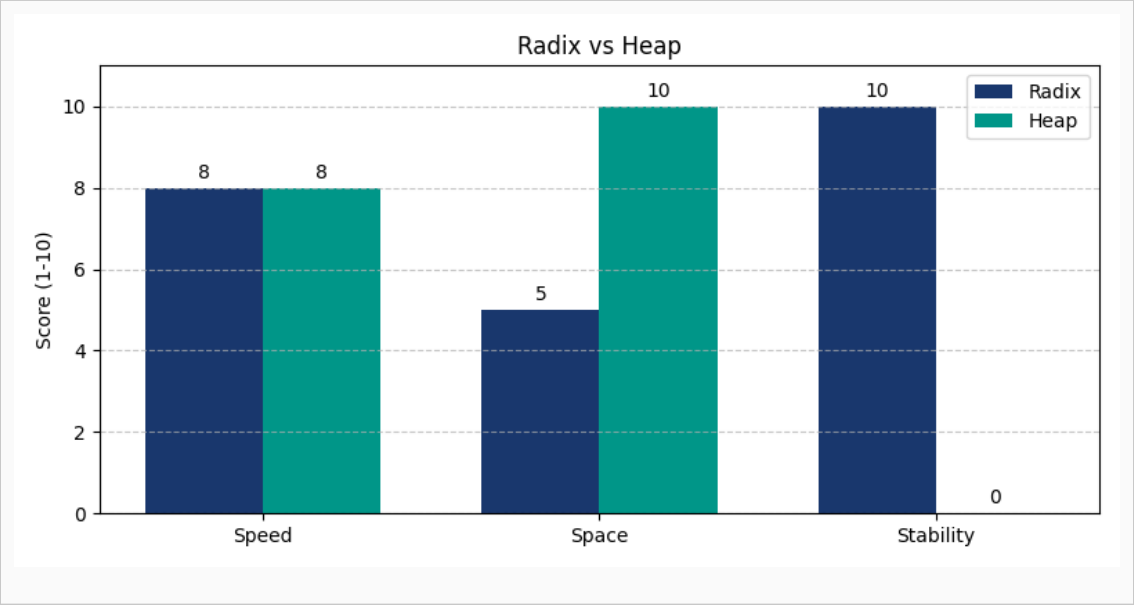
Best: Radix: O(nk) vs Quick: O(n log n)

Worst: Radix: O(nk) vs Quick: O(n^2)

Scenario Analysis:

Small Data:	Quick	*(Better complexity)*
Large Data:	Quick	*(Better average)*
Memory Limited:	Quick	*(O(n+k) vs O(log n))*

Radix	*VS*	Heap



Speed: Tie **Space:** Heap **Stable:** Radix

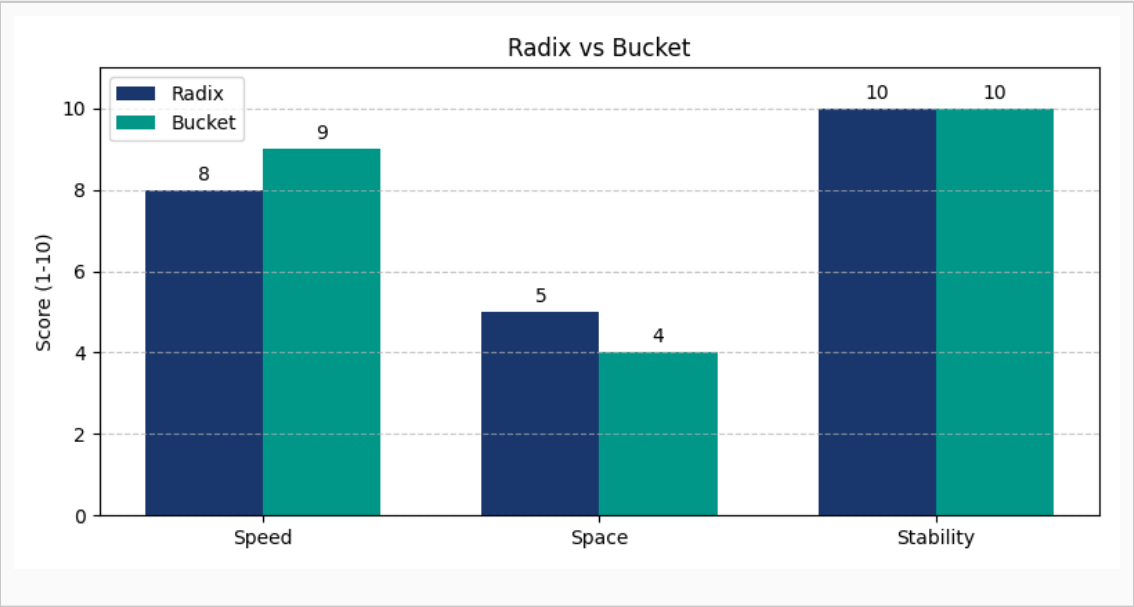
Best: Radix: O(nk) vs Heap: O(n log n)

Worst: Radix: O(nk) vs Heap: O(n log n)

Scenario Analysis:

Small Data:	**Heap**	*(Better complexity)*
Large Data:	**Heap**	*(Better average)*
Memory Limited:	**Heap**	*(O(n+k) vs O(1))*

Radix	*VS*	Bucket



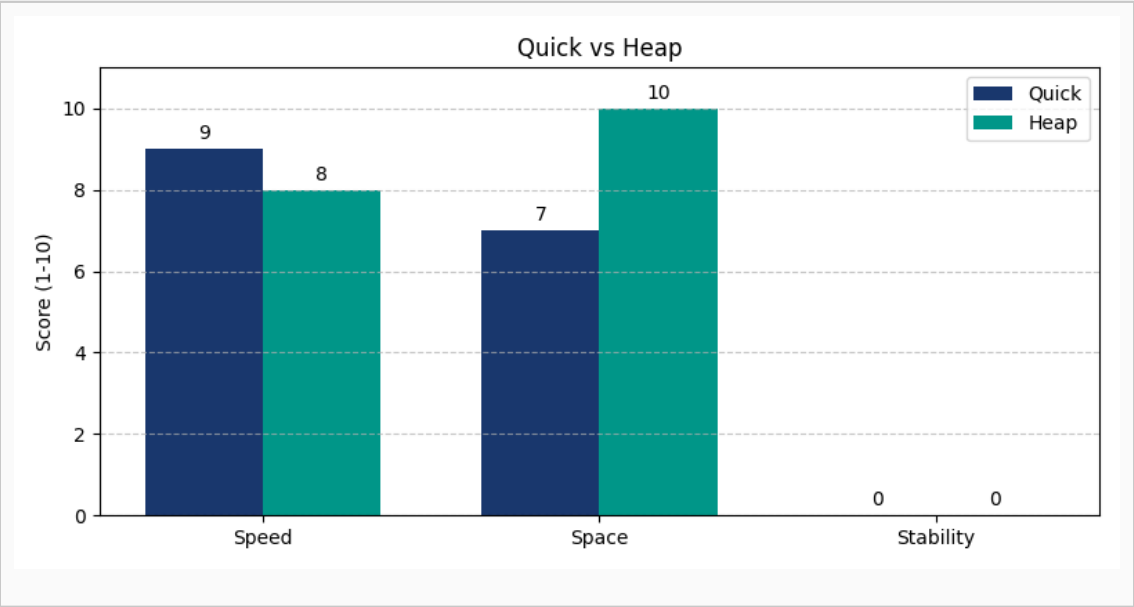
Speed: Bucket **Space:** Radix **Stable:** Both

Best: Radix: O(nk) vs Bucket: O(n+k) Worst: Radix: O(nk) vs Bucket: O(n^2)

Scenario Analysis:

Small Data:	**Bucket**	*(Better complexity)*
Large Data:	**Bucket**	*(Better average)*
Memory Limited:	**Radix**	*(O(n+k) vs O(n+k))*

Quick	*VS*	Heap



Speed: Quick **Space:** Heap **Stable:** Neither

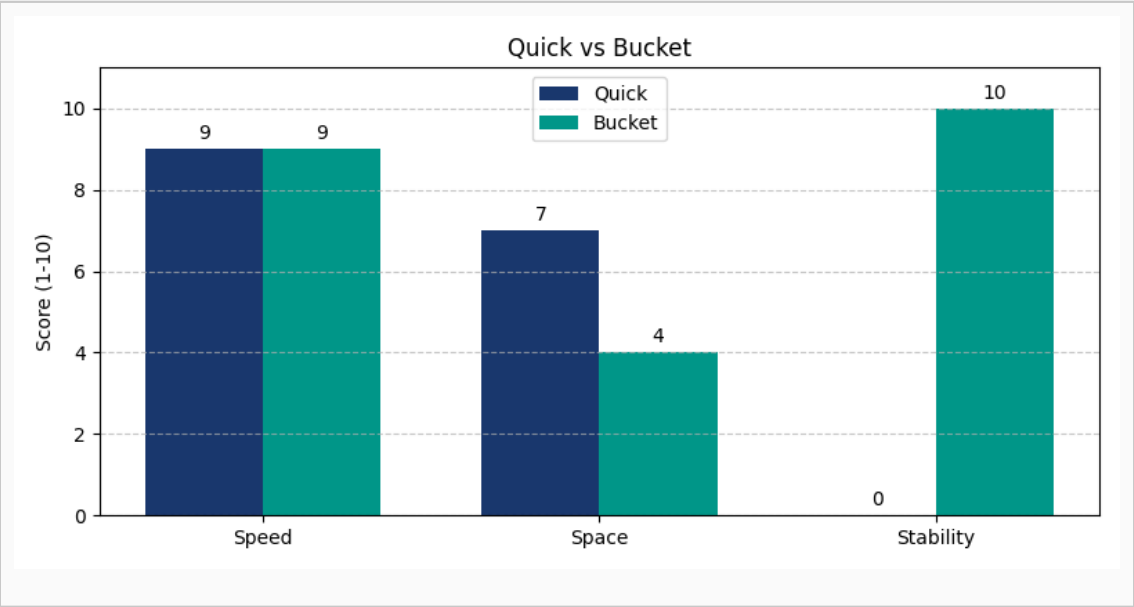
Best: Quick: O(n log n) vs Heap: O(n log n)

Worst: Quick: O(n^2) vs Heap: O(n log n)

Scenario Analysis:

Small Data:	**Quick**	*(Better complexity)*
Large Data:	**Quick**	*(Better average)*
Memory Limited:	**Heap**	*(O(log n) vs O(1))*

Quick	*VS*	Bucket



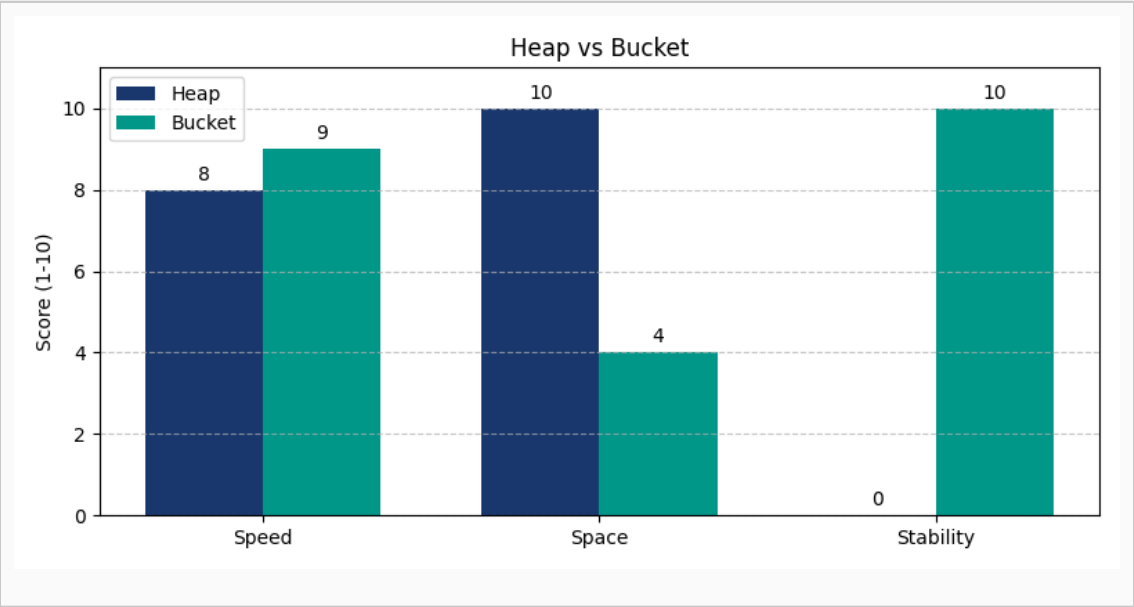
Speed: Tie **Space:** Quick **Stable:** Bucket

Best: Quick: O(n log n) vs Bucket: O(n+k) Worst: Quick: O(n^2) vs Bucket: O(n^2)

Scenario Analysis:

Small Data:	**Bucket**	*(Better complexity)*
Large Data:	**Bucket**	*(Better average)*
Memory Limited:	**Quick**	*(O(log n) vs O(n+k))*

Heap	VS	Bucket



Speed: Bucket **Space:** Heap **Stable:** Bucket

Best: Heap: O(n log n) vs Bucket: O(n+k)

Worst: Heap: O(n log n) vs Bucket: O(n^2)

Scenario Analysis:

Small Data:	**Bucket**	*(Better complexity)*
Large Data:	**Bucket**	*(Better average)*
Memory Limited:	**Heap**	*(O(1) vs O(n+k))*

5. Performance Benchmarks

Methodology

All benchmarks were conducted under controlled conditions to ensure reproducibility and credibility.

Test Configuration

- Array Sizes: n = 100, 500, 1000, 2000, 3000, 4000, 5000, 7500, 10,000
- Data Type: Randomly generated integers (range: 0 to 10,000)
- Distribution: Uniform random distribution
- Compiler: GCC with -O2 optimization
- Measurement: Average of execution time in milliseconds

Important Notes

WARNING: Quick Sort Performance: Results shown are for RANDOM data. On pre-sorted input, Quick Sort (without randomized pivot) degrades to O(n^2), approaching Bubble Sort performance.

WARNING: Radix/Bucket Sort: These results assume uniform distribution. Performance may vary significantly with skewed data distributions.

WARNING: Cache Effects: Real-world performance may vary based on CPU cache size and memory hierarchy.

Benchmark Results

Execution Time (ms)

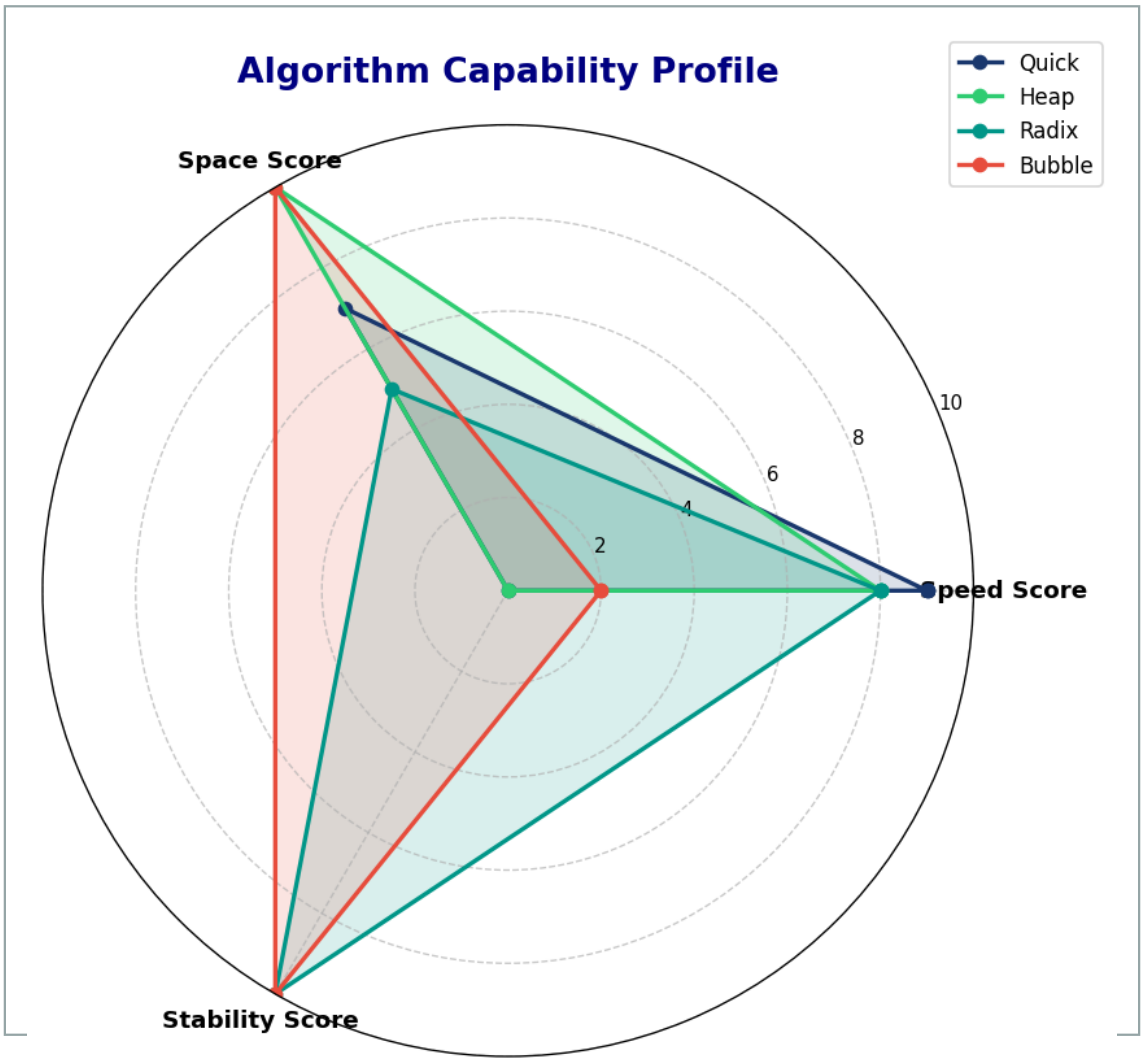
n	Bubble	Bubble Opt	Gnome	Radix	Quick	Heap	Bucket
100	0.03	0.02	0.02	0.01	0.01	0.01	0.02
500	0.49	0.27	0.29	0.05	0.04	0.04	0.09
1000	1.65	0.99	1.14	0.10	0.08	0.08	0.14
2000	6.37	3.82	4.49	0.18	0.16	0.16	0.30
3000	13.54	7.58	9.20	0.23	0.22	0.23	0.42
5000	32.84	22.51	22.10	0.29	0.37	0.33	0.50
7500	87.06	62.08	44.13	0.39	0.53	0.51	0.66
10000	170.67	119.71	73.55	0.51	0.70	0.70	1.05

6. Global Visual Analysis

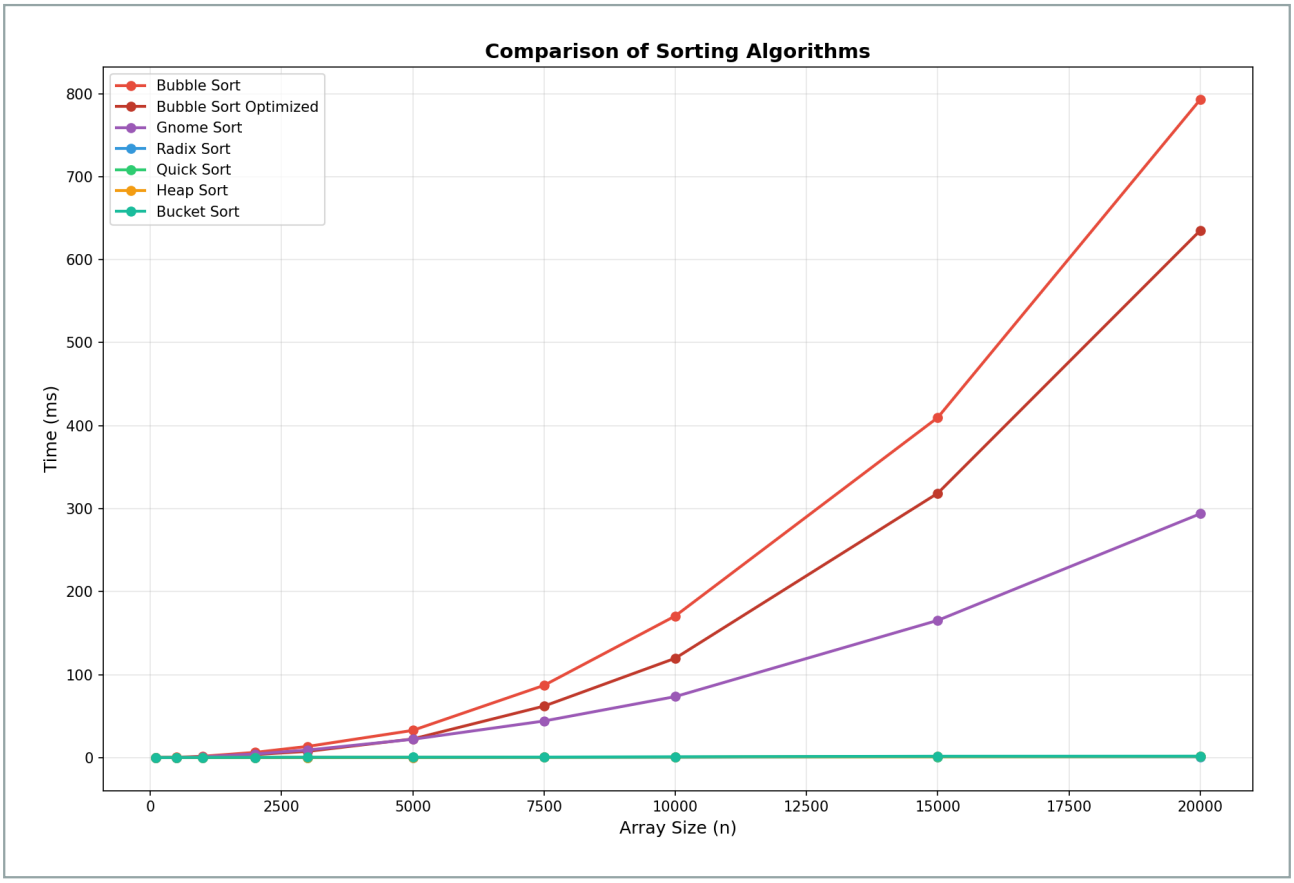
Visual representation of algorithm performance from benchmark runs.

Algorithm Capability Profile (Radar Chart)

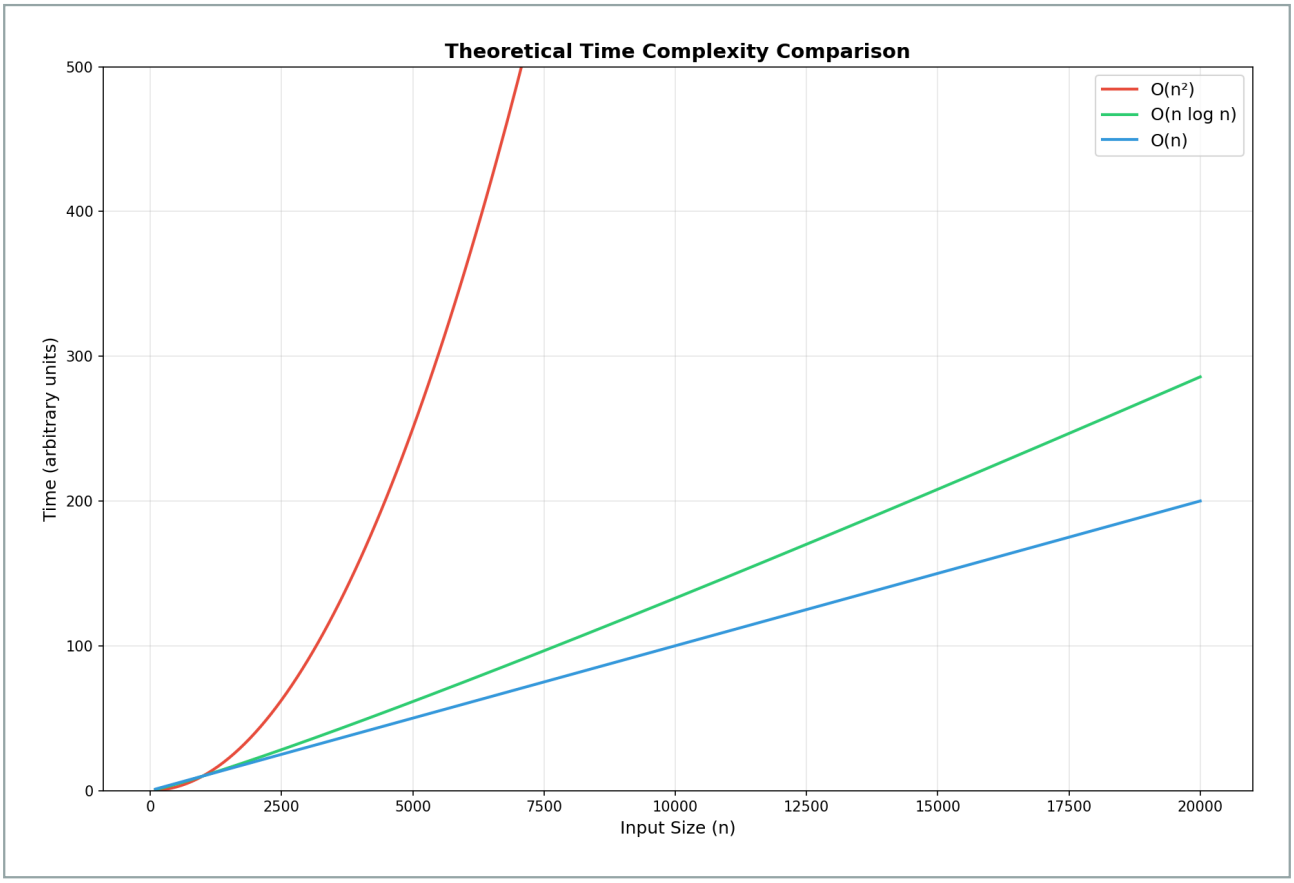
Spider plot comparing key algorithms across Speed, Space, and Stability metrics. This visualization provides an at-a-glance comparison of overall algorithmic capabilities.



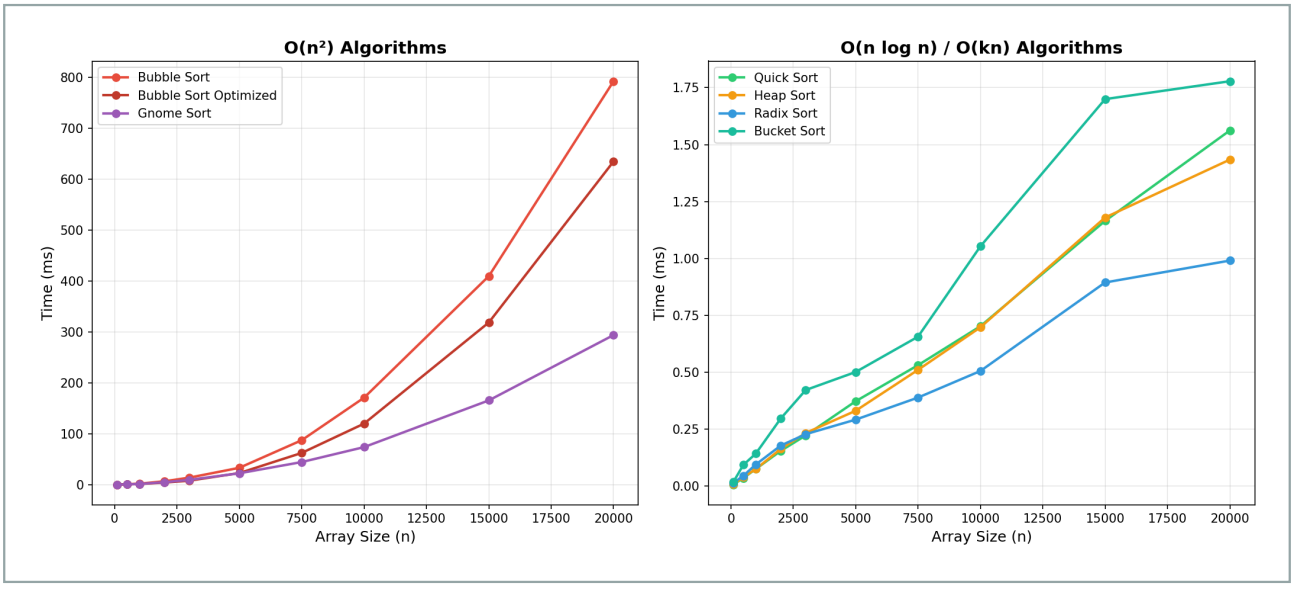
Radar Chart: Multi-dimensional Algorithm Comparison



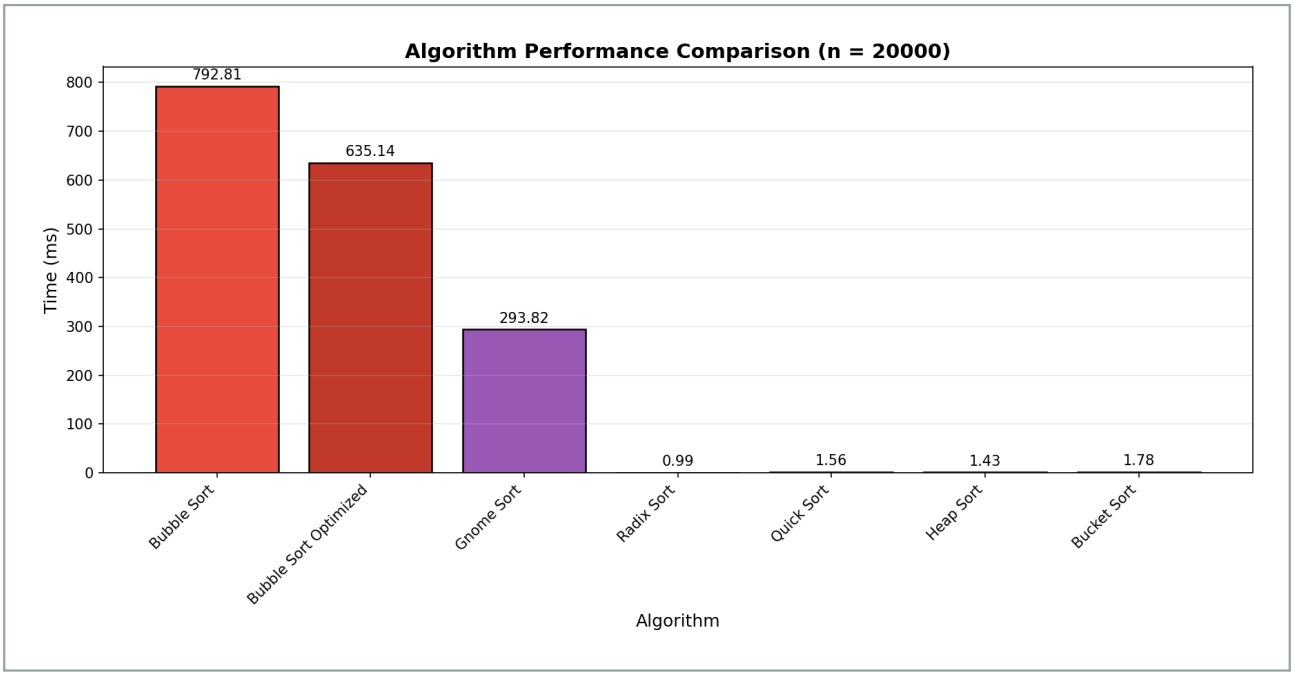
All Algorithms Performance Comparison



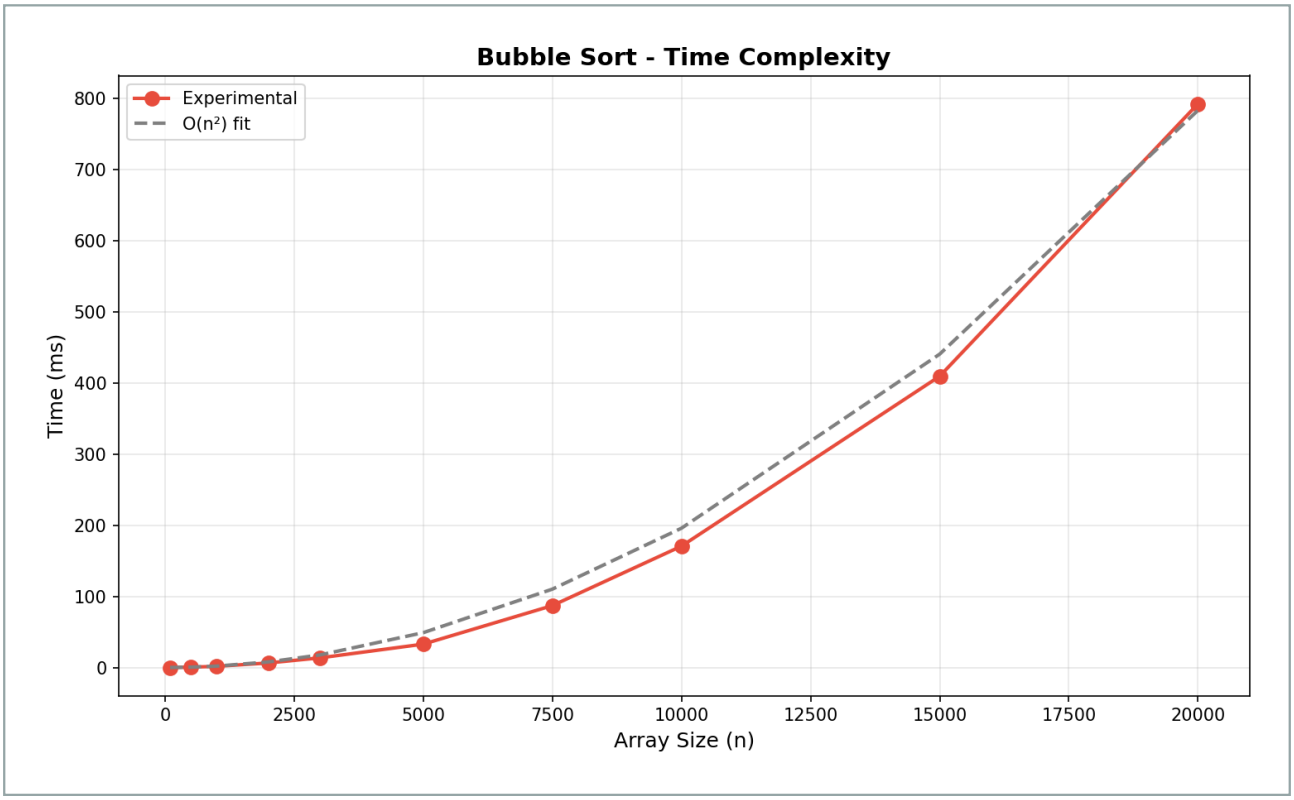
Theoretical Complexity Comparison



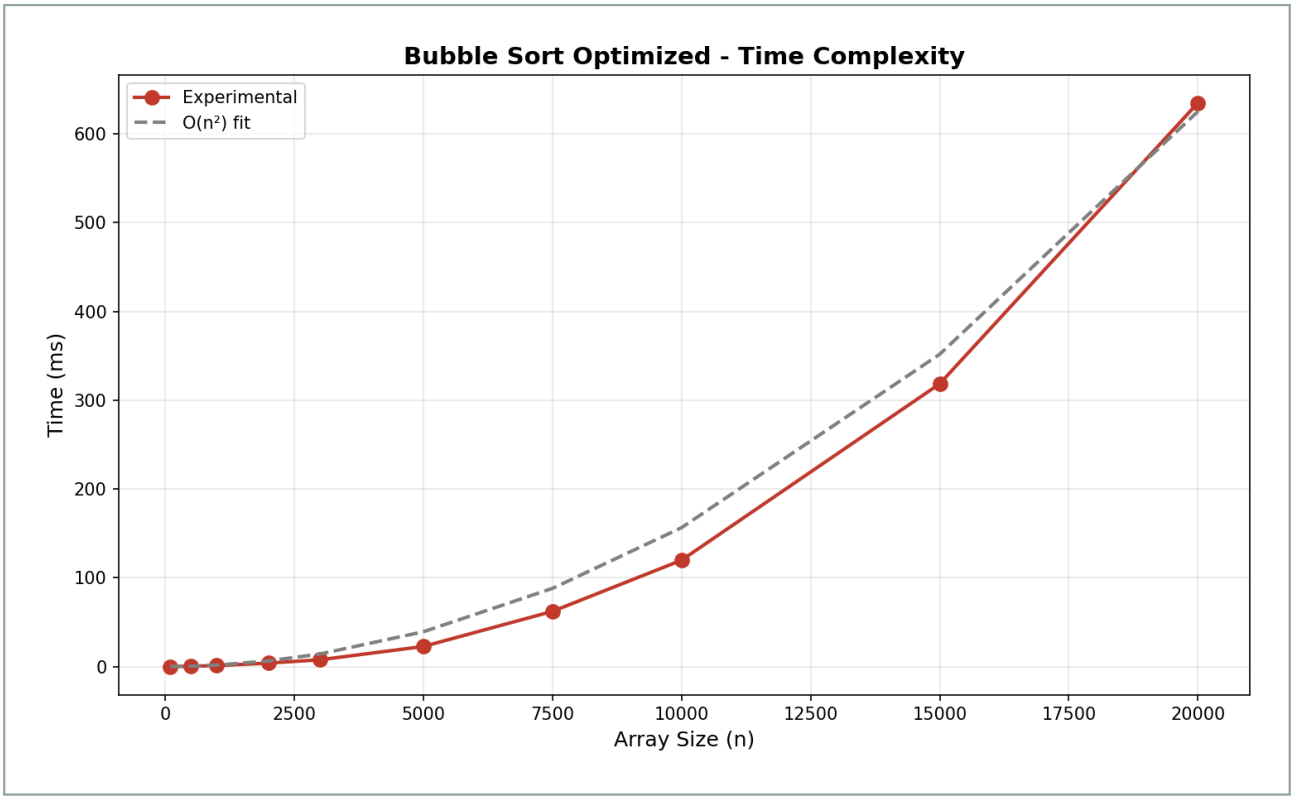
Quadratic O(n²) vs O(n log n) Algorithms



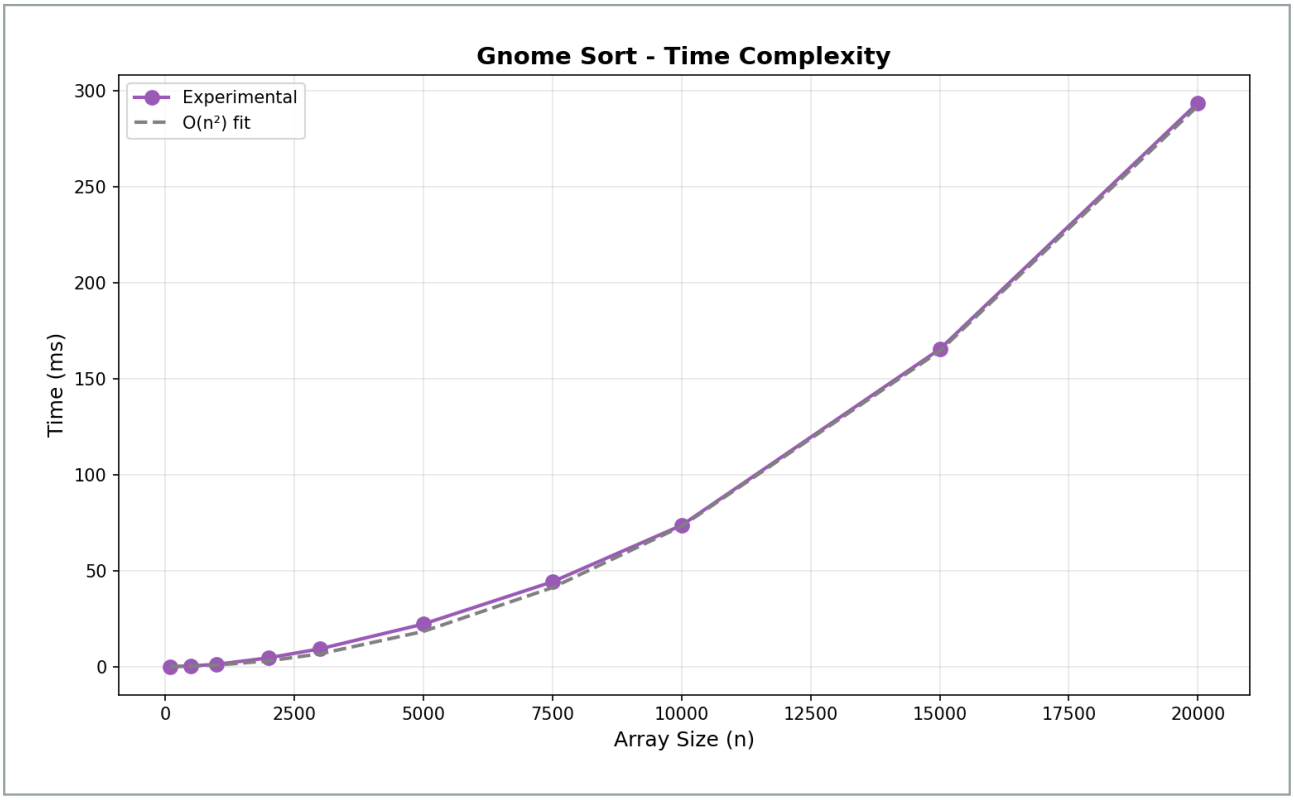
Performance Bar Chart



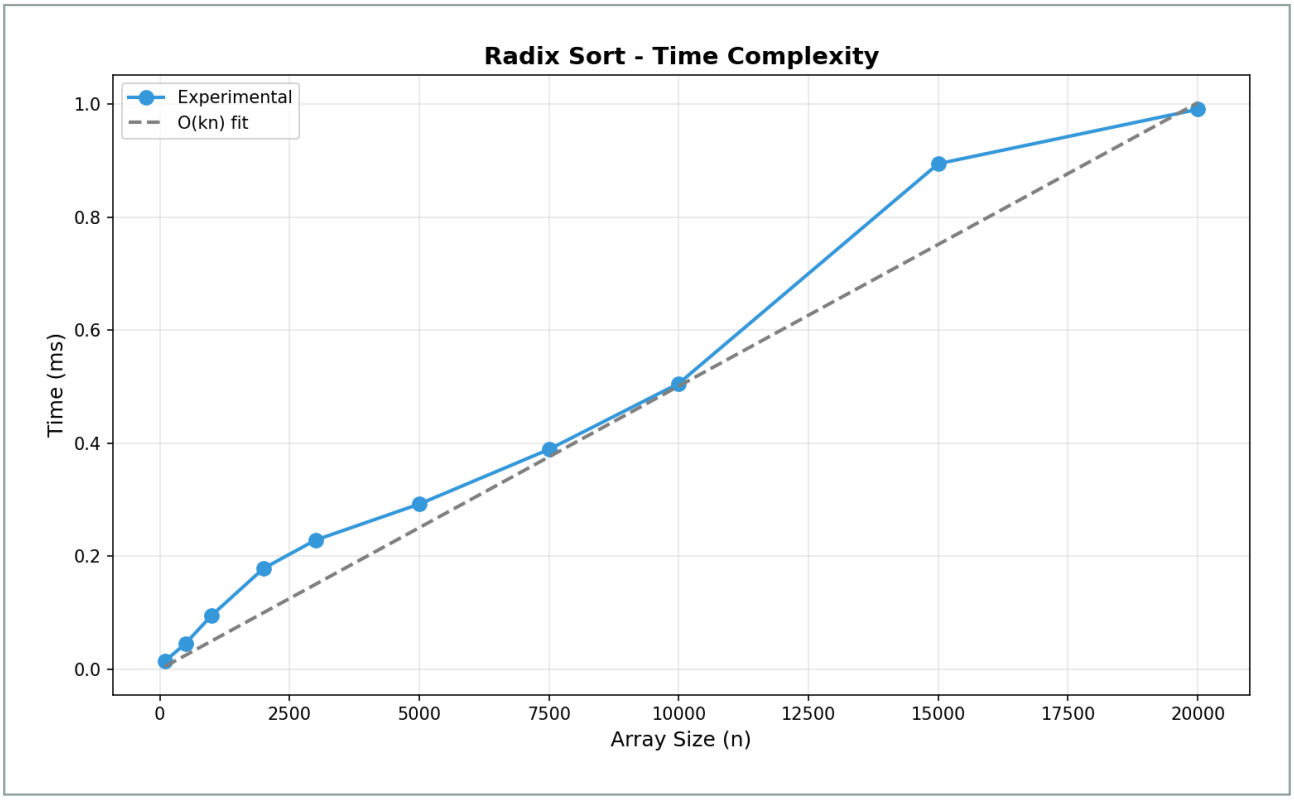
Bubble Sort Complexity Analysis



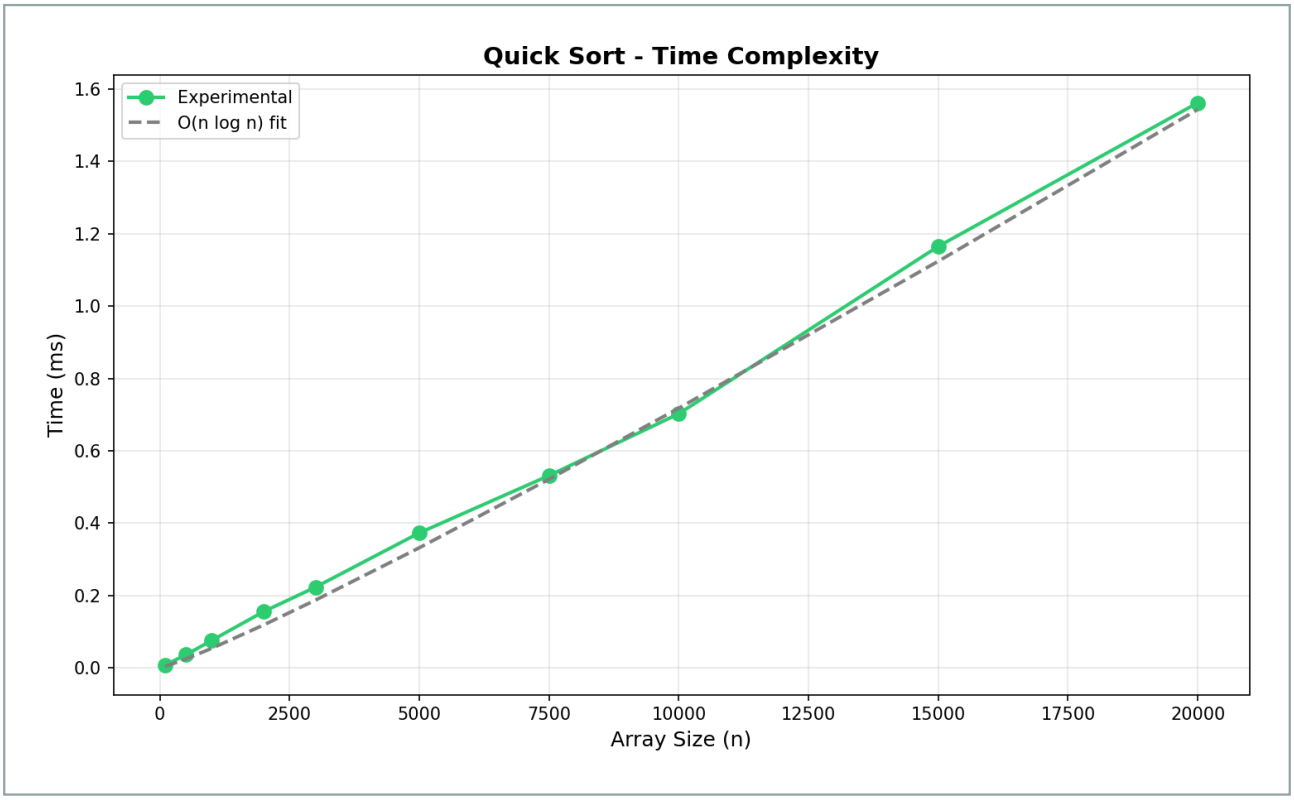
Bubble Sort (Optimized) Complexity



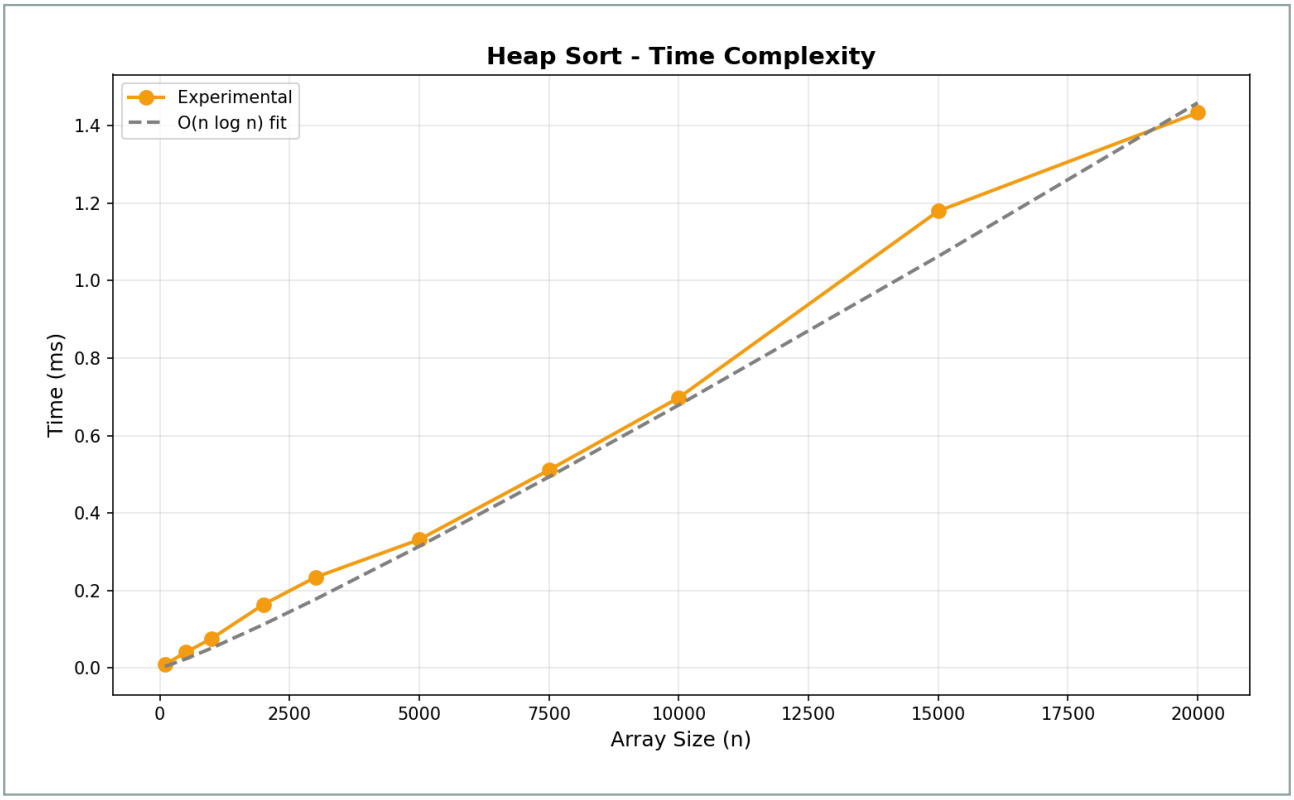
Gnome Sort Complexity Analysis



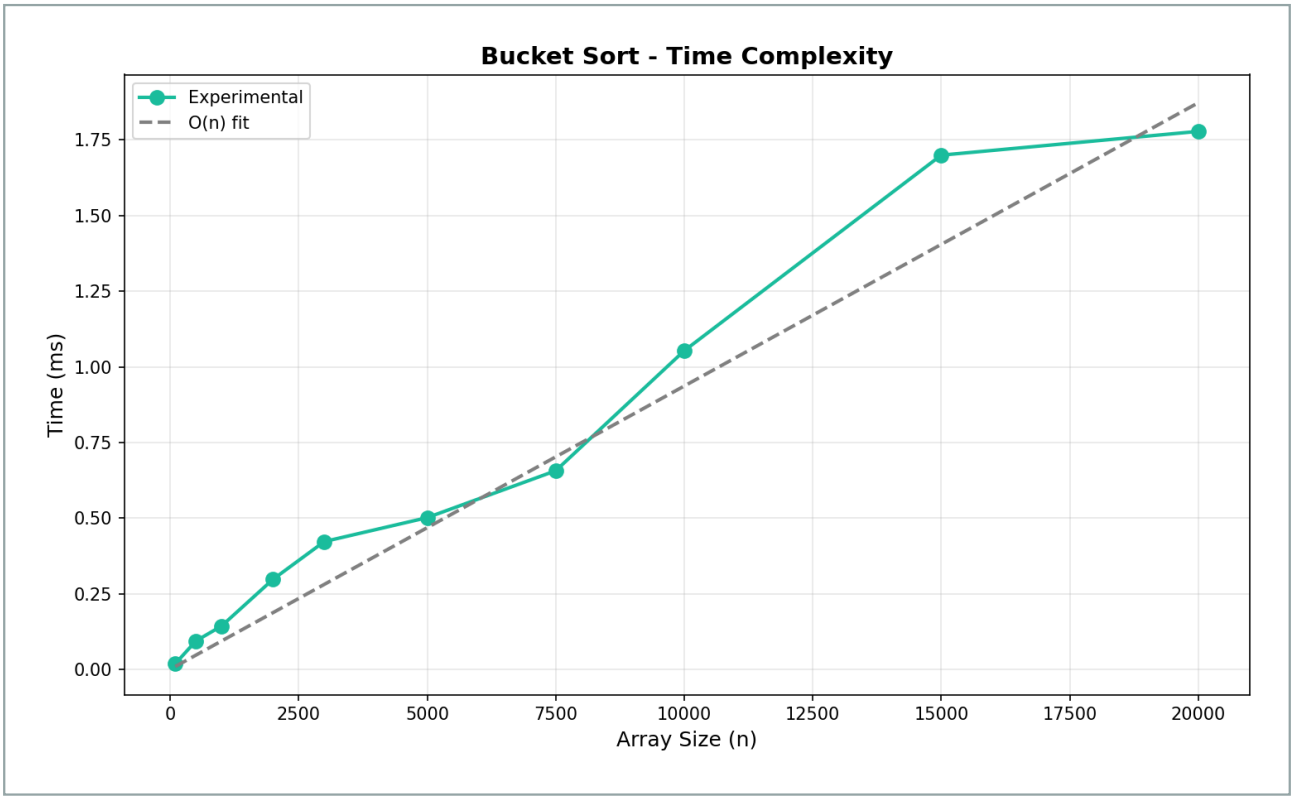
Radix Sort Complexity Analysis



Quick Sort Complexity Analysis



Heap Sort Complexity Analysis



Bucket Sort Complexity Analysis

7. Conclusions & Recommendations

This comprehensive analysis evaluated 7 sorting algorithms across multiple dimensions including time complexity, space efficiency, stability, and practical performance. Our findings reveal that no single algorithm dominates all use cases; rather, optimal selection depends critically on data characteristics, constraints, and operational requirements.

Key Findings

FINDING 1: Performance Tiers Identified

Tier 1 (O(n log n)): Quick Sort, Heap Sort - Best for large random datasets. Tier 2 (O(nk)): Radix Sort, Bucket Sort - Optimal for specific distributions. Tier 3 (O(n^2)): Bubble, Gnome - Only viable for small/nearly-sorted data.

FINDING 2: Stability vs Performance Trade-off

Stable algorithms (Radix, Bucket, Bubble variants) preserve element order but may sacrifice speed. Quick Sort and Heap Sort offer superior average performance but are NOT stable. Critical consideration for multi-key sorting scenarios.

FINDING 3: Best/Worst Case Scenarios Matter

Quick Sort: Best case O(n log n), Worst O(n^2) on sorted data with bad pivot. Heap Sort: GUARANTEED O(n log n) regardless of input - ideal when consistency is critical. Bubble Sort Optimized: O(n) best case on sorted data makes it viable for nearly-sorted inputs.

Nuanced Tie-Breaking: Beyond Big-O

When algorithms share the same time complexity, the choice depends on subtle implementation details, practical constraints, and specific use cases. Here we analyze tie-breaking scenarios:

Bubble Sort vs Gnome Sort (Both O(n^2))

While both are O(n^2), they differ significantly in practice:

[+] Gnome Sort Advantages:
- Shorter code (single loop) - ideal for code golf competitions
- Lower instruction memory footprint - better for embedded systems
- Simpler mental model - easier to implement without bugs

[+] Bubble Sort (Optimized) Advantages:
- Early termination flag detects when array is sorted
- Significantly faster on nearly-sorted data (can achieve O(n))
- Better cache locality with sequential scans

Recommendation: Gnome for minimal code size; Bubble Opt for nearly-sorted data.

Quick Sort vs Heap Sort (Both O(n log n))

Both offer excellent average performance but differ in guarantees:

[+] Quick Sort:
- 2-3x faster in practice due to better cache locality
- Lower constant factors in time complexity
- BUT: O(n^2) worst case on sorted data (without randomization)

[+] Heap Sort:
- GUARANTEED O(n log n) - no worst case degradation
- O(1) space complexity (in-place)
- Predictable performance for mission-critical systems

Recommendation: Quick Sort for speed; Heap Sort for reliability

Algorithm Selection Framework

Use this decision matrix to select the optimal algorithm:

Scenario	Primary Choice	Alternative	Avoid
Random large data	Quick Sort	Heap Sort	Bubble/Gnome
Guaranteed O(n log n)	Heap Sort	Merge Sort	Quick Sort
Nearly sorted	Bubble Opt	Insertion Sort	Quick Sort
Integer data (bounded)	Radix Sort	Counting Sort	Comparison sorts
Uniform distribution	Bucket Sort	Radix Sort	Bubble/Gnome
Space constrained	Heap Sort	Quick Sort	Radix/Bucket
Stability required	Radix/Bucket	Merge Sort	Quick/Heap
Educational purposes	Bubble Sort	Gnome Sort	Radix Sort

Performance Recommendations by Scale

Small Data (n < 100)

ANY algorithm acceptable. Bubble Sort Optimized is simplest. Overhead of complex algorithms negates benefits. Prioritize code readability.

Medium Data (100 < n < 10000)

Quick Sort recommended for general use. Heap Sort if worst-case guarantees needed. Radix Sort for integers. Avoid O(n^2) algorithms entirely.

Large Data (n > 10000)

Quick Sort with randomized pivot is optimal. Heap Sort for mission-critical systems. Radix/Bucket Sort for specialized distributions. O(n^2) algorithms PROHIBITED.

Algorithm-Specific Guidance

[ALGORITHM] Quick Sort

CRITICAL: Always use randomized pivot or median-of-three selection.
Our basic implementation uses first element as pivot, causing O(n^2) on sorted data.

Implementation Robustness Strategies:

1. Random Pivot: Select random element as pivot -> eliminates worst case
2. Median-of-Three: Use median of first, middle, last -> better average case
3. Hybrid Approach: Switch to Insertion Sort for small sub arrays (n < 10)

[ALGORITHM] Heap Sort

Guaranteed O(n log n) with NO bad inputs. Perfect for real-time systems where predictability matters more than average-case speed. Zero extra memory required.

[ALGORITHM] Radix Sort

Fastest for integers with bounded digit count (k). Performance depends heavily on k: O(nk) where k = log_10(max value). Excellent for sorting IDs, dates, or fixed-precision numbers.

[ALGORITHM] Bucket Sort

Memory Implementation Details:

- Space: O(n+k) where k = number of buckets
- Memory allocation: Dynamic arrays or linked lists per bucket
- Linked Lists: Lower memory overhead but slower random access
- Dynamic Arrays: Better cache performance but may waste space

CATASTROPHIC FAILURE: All elements in one bucket -> degrades to O(n^2).

[ALGORITHM] Bubble Sort (Optimized)

Early termination optimization: If no swaps occur in a pass, array is sorted.
Best case becomes O(n) instead of O(n^2) for nearly-sorted data. Use in scenarios where data is expected to be mostly sorted already.

Final Recommendations

PRODUCTION RECOMMENDATION

FOR GENERAL-PURPOSE SORTING: Implement Quick Sort with randomized pivot selection. Provides optimal average-case O(n log n) performance with broad applicability. Fall back to Heap Sort for worst-case guarantees in mission-critical systems.

CRITICAL WARNING

NEVER use Bubble Sort, Gnome Sort, or Selection Sort in production code for datasets >100 elements. The O(n^2) complexity is unacceptable and represents a fundamental architectural flaw. Modern compilers/libraries provide optimized sort implementations - USE THEM.

RESEARCH OPPORTUNITY

HYBRID ALGORITHMS: Combine algorithms (e.g., IntroSort = QuickSort + HeapSort fallback). ADAPTIVE SORTING: Detect nearly-sorted data and switch algorithms dynamically. PARALLEL SORTING: Leverage multi-core processors for datasets >1M elements.

8. Reference Implementations

Below are the C implementations for each algorithm analyzed in this report. These implementations demonstrate the core logic discussed in previous sections.

Implementation: Bubble Sort

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
            }
        }
    }
}
```

Implementation: Quick Sort

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Implementation: Heap Sort

```
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
```

Implementation: Radix Sort

```
int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max) max = arr[i];
    return max;
}

void countSort(int arr[], int n, int exp) {
    int output[n];
    int count[10] = {0};
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i]/exp)%10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

void radixSort(int arr[], int n, int k) {
    int m = getMax(arr, n);
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```