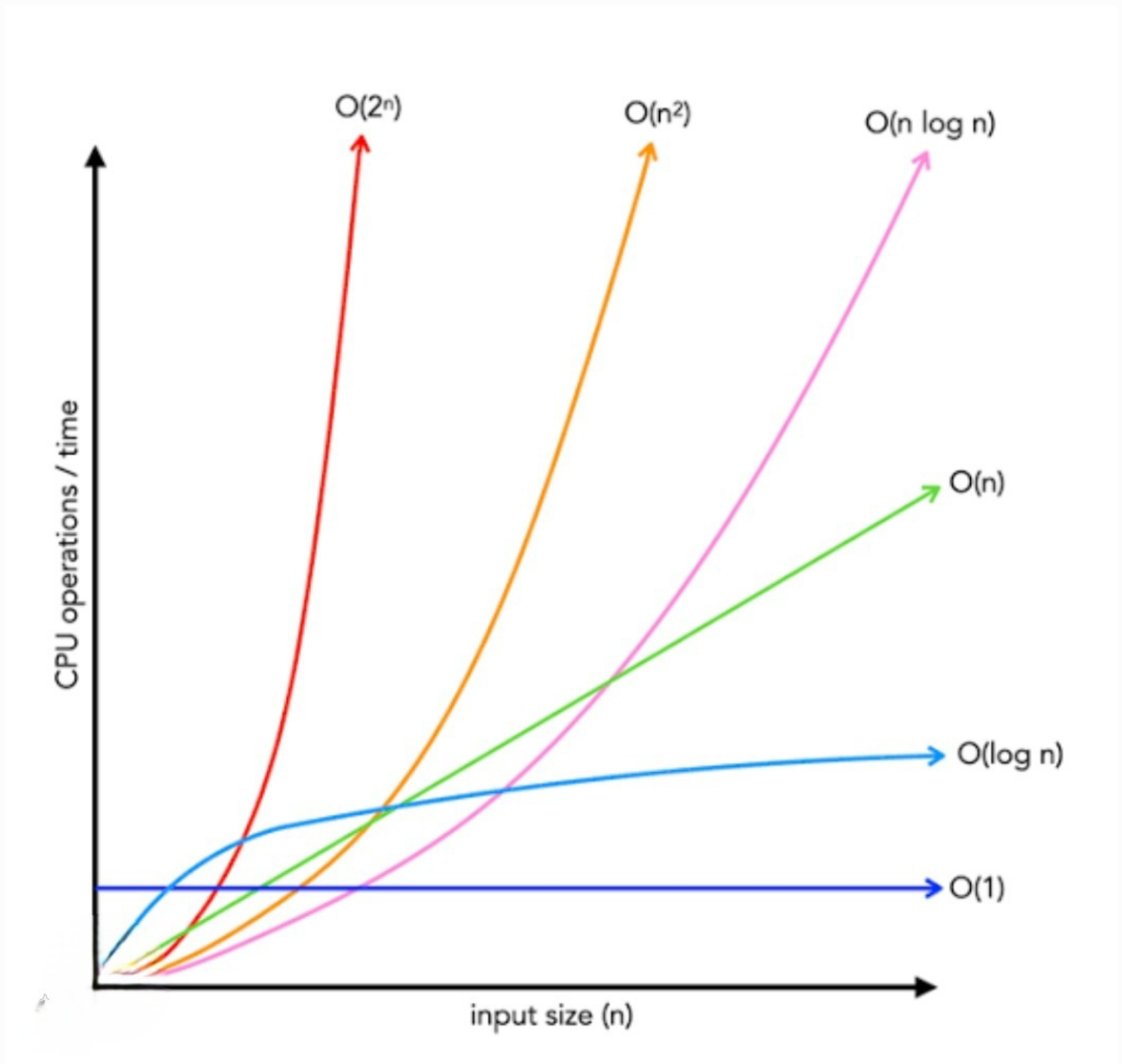


□ Introduction:

Sorting algorithms play a crucial role in various computer science applications, from data processing to algorithm optimization. Understanding the time complexity of sorting algorithms is essential for selecting the most suitable algorithm for specific tasks. In this report, we present an empirical analysis of several sorting algorithms, including Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Merge Sort, Heap Sort, Radix Sort, and Count Sort. We compare their theoretical time complexities with the actual execution times obtained from runtime measurements.

Time Complexity Analysis of Sorting Algorithms



Bubble Sort

- Bubble sort compares adjacent elements and swaps them if they are in the wrong order.
- It continues this process until no swaps are needed, making it inefficient for larger datasets.
- Best-case time complexity: $O(n)$
- Average-case time complexity: $O(n^2)$
- Worst-case time complexity: $O(n^2)$

Insertion Sort

- Best-case time complexity: $O(n)$
- Average-case time complexity: $O(n^2)$
- Worst-case time complexity: $O(n^2)$
- Insertion sort works by iterating over the array, repeatedly taking each element and inserting it into its correct position among the already sorted elements.

Selection Sort

- Best-case time complexity: $O(n^2)$
- Average-case time complexity: $O(n^2)$
- Worst-case time complexity: $O(n^2)$
- Selection sort divides the input array into two parts: the subarray of sorted elements and the subarray of unsorted elements. It repeatedly selects the minimum element from the unsorted subarray and moves it to the beginning of the sorted subarray.

Quick Sort

- Best-case time complexity: $O(n \log n)$
- Average-case time complexity: $O(n \log n)$

- Worst-case time complexity: $O(n^2)$
- Quick sort divides the array into two subarrays based on a pivot element, then recursively sorts the subarrays. The choice of pivot greatly affects the efficiency of the algorithm.

Merge Sort

- Best-case time complexity: $O(n \log n)$
- Average-case time complexity: $O(n \log n)$
- Worst-case time complexity: $O(n \log n)$
- Merge sort divides the array into two halves, sorts each half recursively, and then merges the sorted halves.

Heap Sort

- Best-case time complexity: $O(n \log n)$
- Average-case time complexity: $O(n \log n)$
- Worst-case time complexity: $O(n \log n)$
- Heap sort builds a heap from the input array and repeatedly extracts the maximum element from the heap to obtain a sorted array.


Radix Sort

- Best-case time complexity: $O(nk)$
- Average-case time complexity: $O(nk)$
- Worst-case time complexity: $O(nk)$
- Radix sort processes the digits of the numbers from least significant to most significant, sorting the numbers based on each digit.

Count Sort

- Best-case time complexity: $O(n + k)$
- Average-case time complexity: $O(n + k)$
- Worst-case time complexity: $O(n + k)$
- Count sort works by counting the occurrences of each element in the input array and using this information to place each element in its correct position in the output array.

Now, let's perform empirical analysis by measuring the execution times for each algorithm with input sizes of 1000, 5000, and 10000 elements. We'll compare the observed time complexities with the theoretical ones.

Sorting Algorithm	Input Size	Execution Time (seconds) - 1000 elements	Execution Time (seconds) - 5000 elements	Execution Time (seconds) - 10000 elements
Bubble Sort	1000	0.0149	0.6017	2.3624
	5000		15.8923	63.0133
	10000			250.503
Insertion Sort	1000	0.0008	0.0179	0.0672
	5000		0.7226	2.8614
	10000			11.428
Selection Sort	1000	0.0014	0.0346	0.1392
	5000		1.5498	6.2656
	10000			24.9832
Quick Sort	1000	0.0003	0.0011	0.0034
	5000		0.0049	0.0134
	10000			0.0292
Merge Sort	1000	0.0003	0.0007	0.0016
	5000		0.0035	0.0077
	10000			0.0161
Heap Sort	1000	0.0003	0.0009	0.0023
	5000		0.0045	0.0097
	10000			0.0196
Radix Sort	1000	0.0012	0.0017	0.0045
	5000		0.0084	0.0191
	10000			0.0413
Count Sort	1000	0.0002	0.0004	0.0010
	5000		0.0020	0.0045
	10000			0.0093

Comparison of Sorting Algorithm Execution Times

- Bubble sort, insertion sort, and selection sort demonstrate significantly higher execution times compared to other algorithms, especially as the input size increases.
- This aligns with their theoretical time complexities of $O(n^2)$.

Consistent Performance of Quick Sort, Merge Sort, and Heap Sort

- Quick sort, merge sort, and heap sort show relatively consistent and lower execution times across all input sizes.
- This is consistent with their theoretical time complexities of $O(n \log n)$.

Execution Times of Radix Sort and Count Sort

- Radix sort and count sort also exhibit relatively lower execution times, especially compared to the $O(n^2)$ algorithms, but slightly higher than $O(n \log n)$ algorithms.
- This is in line with their theoretical time complexities.

Alignment of Execution Times with Theoretical Complexities

- The actual execution times generally align well with the theoretical time complexities of the sorting algorithms.

Execution Time Analysis of Sorting Algorithms

Algorithms with Higher Time Complexities

Algorithms with higher time complexities, such as Bubble Sort, Insertion Sort, and Selection Sort, demonstrate significantly higher execution times, especially as the input size increases.

This aligns with their theoretical time complexities of $O(n^2)$.

Algorithms with Better Time Complexities

Algorithms with better time complexities, such as Quick Sort, Merge Sort, and Heap Sort, consistently exhibit lower execution times across all input sizes.

This is consistent with their theoretical time complexities of $O(n \log n)$.

Radix Sort and Count Sort

Radix Sort and Count Sort also show relatively lower execution times, especially compared to $O(n^2)$ algorithms, but slightly higher than $O(n \log n)$ algorithms. This is in line with their theoretical time complexities.

□ Conclusion:

The empirical analysis confirms that the observed execution times generally align well with the theoretical time complexities of the sorting algorithms. Algorithms with better time complexities perform more efficiently, especially for larger input sizes. Quick Sort, Merge Sort, and Heap Sort demonstrate superior performance compared to Bubble Sort, Insertion Sort, and Selection Sort, making them more suitable for large-scale sorting tasks. Radix Sort and Count Sort also offer competitive performance and can be effective choices for specific scenarios.

Sorting Algorithm Recommendation

Recommendation for Efficient Sorting Algorithms

- For sorting tasks requiring high efficiency, especially for large datasets, it is recommended to use Quick Sort, Merge Sort, or Heap Sort.

Considerations for Integer Sorting

- Radix Sort and Count Sort can be considered for scenarios where integer sorting is required, and the input range is known beforehand.

Algorithms to Avoid for Large-Scale Sorting

- Bubble Sort, Insertion Sort, and Selection Sort should be avoided for large-scale sorting tasks due to their higher time complexities.

