Comparative Analysis of Sorting Algorithms

By [T. Aqshin]

1. Introduction

In this paper, we investigate a classical problem in computer science: sorting. The task is to arrange an array of n integers according to a total order which is computed in O(1) time complexity. We use the less than operator (<) as the total order. In this paper, we compare five sorting algorithms: insertion sort, quick sort, heap sort, radix sort, and a hybrid sort.

2. Description of the Algorithms

2.1 Insertion Sort

Insertion sort repeatedly inserts elements into a sorted sequence. The average complexity of insertion sort is O(n^2). The algorithm is stable and in-place.

2.2 Quick Sort

Quick sort is a divide and conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The average time complexity of Quick Sort is O(n log n). The algorithm is not stable but it is in-place.

2.3 Heap Sort

Heap sort is a comparison-based sorting algorithm. It sorts an array by visualizing it as a binary heap. The average time complexity of Heap Sort is O(n log n). The algorithm is not stable but it is in-place.

2.4 Radix Sort

Radix sort is a non-comparative integer sorting algorithm. It sorts numbers digit by digit from least significant digit to most significant. The average time complexity of Radix Sort is O(nk), where n is the number of elements and k is the number of digits. The algorithm is stable and in-place.

2.5 Hybrid Sort

Hybrid sort is a combination of various sorting algorithms. It chooses the most efficient sorting algorithm based on the size of the input array. For small arrays, it uses insertion sort. For medium-sized arrays, it uses heap sort. For large arrays, it uses quick sort.

3. Methodology

The algorithms were implemented in C++ and tested on arrays of varying sizes, from 10 to 900. The results were generated for randomly shuffled arrays of values ranging from -5000 to 5000. Each array size was tested 100 times. The result for each size is the average time it took for each algorithm to sort the input array.

4. Results

The results indicate that the performance of the sorting algorithms varies significantly depending on the size of the input array. For small arrays, insertion sort and hybrid sort (which uses insertion sort for small arrays) performed the best. For medium-sized arrays, heap sort and hybrid sort (which uses heap sort for medium-sized arrays) were the most efficient. For large arrays, quick sort and hybrid sort (which uses quick sort for large arrays) had the best performance.

5. Conclusions

In conclusion, no single sorting algorithm outperforms all others in all scenarios. The efficiency of a sorting algorithm depends on the size and nature of the input array. Therefore, a hybrid approach that adapts to the size of the input array can provide the best performance across a wide range of scenarios. Further research could investigate the impact of different pivot selection strategies in quick sort, or the use of different data structures in heap sort.