SORTING

In computer science, each sorting algorithm is better than the other one in some or the other situation and has its own advantages To measure the performance of each sorting algorithm, the most a prominent factor is runtime that a specific sort uses to put in process adata.
In this study, we will determine the efficiency of the various sorting algorithms according to the time and number of swaps by using randomized trials.

Introduction

In computer science, a sorting algorithm can be elucidated as a well-organized algorithm that lays out elements of a list in a definite order. In brief it arranges a group of items in a specific order. Sorting data has been developed to arrange the array values in numerous ways for a database. For example, sorting will order an array of numbers from lowest to highest or from highest to lowest, or arrange an array of strings into alphabetical order. Typically, it sorts an array into increasing or decreasing order. Frequently simple sorting algorithms comprises two steps which are:-
1) Compare two items
2) Swap two items or copy one item.
It incessantly executes until the data is sorted.

Classification of Sorting Algorithms

System complexity of computational.
Computational complexity in terms of number of swaps.
Memory usage is also a factor in categorizing the sorting algorithms.

Bubble Sort : The algorithm embarks on the beginning of the data set. It contrasts the first two elements, and if the first is greater than the second then it swaps them. It continues the same process for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, rerunning until no swaps have appeared on the last pass.

Insertion Sort: It works by selecting elements from the list one by one and inserting them in their exact position into a new sorted list. The array space can be shared between the new lists and remaining elements, one thing is to be marked that insertion is expensive and requires shifting of all the following elements over by one.

Selection Sort: It is defined as specifically an in-place comparison sort. It has O(n square) complexity, rendering it an inefficient place on large lists, and usually performs worse than similar insertion sort.

Shell Sort: It can do better than bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be elaborated by arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

Quick Sort : Quick sort is a divide and conquer algorithm which counts on a partition operation: for dividing an array, we choose an element, called a pivot. We move all smaller elements before the pivot and then move all greater elements after pivot. We then recursively sort the lesser and greater sub lists together with its modest O(log n) space usage. The most complex issue in quick sort is selecting a good pivot element; consistent poor choices of pivots can result into a drastically slower O(n) performance, but if at each step we select the median as the pivot then it can work in O(n log n)

Merge Sort : It is defined as a comparison-based sorting algorithm. In most of the implementations it is stable, which means that it preserves the input order of equal elements in the sorted output.
Cocktail Sort: This algorithm differs from bubble sort as it sorts in both directions each pass through the list. This sorting algorithm is only marginally more difficult than bubble sort to implement, and solves the problem with so-called turtles in bubble sort.

Working of Sorting Algorithms

Bubble Sort: Swap two adjacent elements if they are out of order. Repeat this process until the array is sorted.

Selection Sort: Select the smallest element in the array, and place it in a correct position. Swap it with the value in the first position. Repeat this process until the array is sorted. (starting at the second position and advancing each time)

Insertion Sort: Scan successive elements for an out-of-order item, then insert the item in the proper place.

Quick Sort: Divide the array into two segments. In the first segment, all the elements are less than or equal to the pivot value. In the second segment, all the elements are greater than or equal to the pivot value. Finally, sort the two segments recursively. Working of Sorting Algorithms

Merge Sort: Begin from the two sorted runs of length 1, merge into a single run of twice the length. Repeat until a single sorted run is left. Merge sort needs N/2 extra buffer. Performance is second place on average, with quite good speed on nearly sorted arrays.

Shell Sort: Sort every nth element in an array using insertion sort. Repeat using smaller N values, until N = 1. On an average, Shell sort is fourth place in speed. Shell sort may sort some distributions slowly.

Assessment Chart

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

Conclusions

Discussion shows that Quick sort is efficient for small as well as large integers. Quick sort is prominently faster in practice than other O (n log n) algorithms. In terms of swapping, the Bubble sort performs the greatest number of swaps because each element will only be compared to adjacent elements and exchanged if they are out of order. Insertion Sort sorts small arrays fast and in contrast big arrays very slowly.