Efficient sorting is important to optimizing the use of other algorithms such as Binary search and merge algorithms that require sorted lists to work correctly [2,3]. We can not deploy binary search if data is not pre sorted otherwise the search process may get trapped into a blind alley thereby exhibiting worst case complexity.

The time for most sorting algorithms depends on the amount of data or size of the problem. In order to analyze an algorithm, we try to find a relationship showing how the time needed for the algorithm depends on the amount of data. This is called the "complexity" of the algorithm.

Different computers work at different speeds,; there no uniform measurement of how long a computation step will take. Hence analysis of sorting algorithm cannot predict exactly how long it will take on a particular computer.

Analysis of efficiency also depends considerably on the nature of the data. For example, if the original data set already is almost ordered, a sorting algorithm may behave rather differently than if the data set is random is ordered in the reverse direction. So instead of trying to find the time complexity for all legal input permutations, we can choose just one input as a representative input getting the

average-case complexity. (incompressibility method, Via Kolmogorov )

Definitions:

Basically a sorting problem is to arrange a sequence of objects so that the values of their "key" fields are in "non-decreasing" sequence.

Given objects O1, O2, ..., On

With key values K1, K2, ..., Kn respectively,

Arrange the objects in an order Oi1, Oi2, ..., Oin , Such that

Ki1 <= Ki2 <= ... <= Kin.

Analysis of sorting:

Computational complexity (worst, average and best behavior) of element comparisons in terms of the size of the unsorted list is considered for the analysis of the efficiency of sorting algorithm. In general if the size of unsorted list is (n), then for typical sorting algorithms, good behavior is O (n log n) and bad behavior is Ω (n²). The Ideal behavior is O (n).

Theorem 1: For n records, linear sorting will take a complexity O (n log n). If the input sequence is presorted, compared to an unsorted sequence possibly less steps are sufficient. If all you can do is to compare keys, and subsequent actions based on information from key comparison.

Each of sorting algorithms works as follows:

Bubble Sort

y Exchange two adjacent elements if they are out of order. Repeat until array is

sorted.

Selection Sort

y Find the smallest element in the array, and put it in the proper place. Swap it with

the value in the first position. Repeat until 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

y Partition the array into two segments. In the first segment, all elements are less

than or equal to the pivot value. In the second segment, all elements are greater

than or equal to the pivot value. Finally, sort the two segments recursively. Merge Sort

y Start from 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

array.

Shell Sort

y Sort every Nth element in an array using insertion sort. Repeat using smaller N

values, until N = 1. On average, Shell sort is fourth place in speed. Shell sort may

sort some distributions slowly