In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and forproducing human-readable output. . More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);

2. The output is a permutation (reordering) of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.[1] Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2006). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

By analyzing an algorithm, we mean to study the performance of an algorithm including the assertion of its correctness and a determination of the cost of its execution. Although a given algorithm is often analyzed in a particular way that is most suitable for such an algorithm, we are more interested in general procedures and techniques that can be used to study the performance of classes of algorithms. To be able to talk about general analysis techniques will not only add to our understanding of the behavior of a class of algorithms but will also, in many cases, lead to useful synthesis procedures. A good example illustrating these points is the various techniques that can be used to analyze a class of sorting algorithms which can be modelled as networks made up of comparator modules. In this paper, we discuss several approaches to such an analysis problem. Moreover, synthesis procedures suggested by these analysis techniques will also be presented.

Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list. When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first component:

(4, 2) (3, 7) (3, 1) (5, 6)

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

(3, 7) (3, 1) (4, 2) (5, 6) (order maintained)

(3, 1) (3, 7) (4, 2) (5, 6) (order changed)

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional [computational cost](http://en.wikipedia.org/wiki/Computational_complexity_theory).

Sorting based on a primary, secondary, tertiary, etc. sort key can be done by any sorting method, taking all sort keys into account in comparisons (in other words, using a single composite sort key). If a sorting method is stable, it is also possible to sort multiple times, each time with one sort key. In that case the keys need to be applied in order of increasing priority.

Example: sorting pairs of numbers as above by second, then first component:

(4, 2) (3, 7) (3, 1) (5, 6) (original)

(3, 1) (4, 2) (5, 6) (3, 7) (after sorting by second component)

(3, 1) (3, 7) (4, 2) (5, 6) (after sorting by first component)

On the other hand:

(3, 7) (3, 1) (4, 2) (5, 6) (after sorting by first component)

(3, 1) (4, 2) (5, 6) (3, 7) (after sorting by second component,

order by first component is disrupted).

In this table, *n* is the number of records to be sorted. The columns "Average" and "Worst" give the time complexity in each case, under the assumption that the length of each key is constant, and that therefore all comparisons, swaps, and other needed operations can proceed in constant time. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself, under the same assumption. These are all [comparison sorts](http://en.wikipedia.org/wiki/Comparison_sort). The run time and the memory of algorithms could be measured using various notations like theta, omega, Big-O, small-o, etc. The memory and the run times below are applicable for all the 5 notations.

| [Comparison sorts](http://en.wikipedia.org/wiki/Comparison_sort) | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Best** | **Average** | **Worst** | **Memory** | **Stable** | **Method** | **Other notes** |
| [Binary tree sort](http://en.wikipedia.org/wiki/Binary_tree_sort) | \mathcal{} n | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} n | Yes | Insertion | When using a [self-balancing binary search tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) |
| [Bogosort](http://en.wikipedia.org/wiki/Bogosort) | \mathcal{} n | \mathcal{} n \cdot n! | \mathcal{} {n \cdot n! \to \infty} | \mathcal{} {1} | No | Luck | Randomly permute the array and check if sorted. |
| [Bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Exchanging | Tiny code size |
| [Cocktail sort](http://en.wikipedia.org/wiki/Cocktail_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Exchanging |  |
| [Comb sort](http://en.wikipedia.org/wiki/Comb_sort) | \mathcal{} n | \mathcal{} n \log n | \mathcal{} n^2 | \mathcal{} {1} | No | Exchanging | Small code size |
| [Cycle sort](http://en.wikipedia.org/wiki/Cycle_sort) | — | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | No | Insertion | In-place with theoretically optimal number of writes |
| [Gnome sort](http://en.wikipedia.org/wiki/Gnome_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Exchanging | Tiny code size |
| [Heapsort](http://en.wikipedia.org/wiki/Heapsort) | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {1} | No | Selection |  |
| [In-place](http://en.wikipedia.org/wiki/In-place) [Merge sort](http://en.wikipedia.org/wiki/Merge_sort) | \mathcal{} - | \mathcal{} - | \mathcal{} {n \left( \log n \right)^2} | \mathcal{} {1} | Yes | Merging | Implemented in Standard Template Library (STL): [[3]](http://www.sgi.com/tech/stl/stable_sort.html); can be implemented as a stable sort based on stable in-place merging: [[4]](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.8381) |
| [Insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Insertion | O(*n* + *d*), where *d* is the number of [inversions](http://en.wikipedia.org/wiki/Permutation_groups#Transpositions.2C_simple_transpositions.2C_inversions_and_sorting) |
| [Introsort](http://en.wikipedia.org/wiki/Introsort) | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} \log n | No | Partitioning & Selection | Used in [SGI](http://en.wikipedia.org/wiki/Silicon_Graphics) [STL](http://en.wikipedia.org/wiki/Standard_Template_Library) implementations |
| [Library sort](http://en.wikipedia.org/wiki/Library_sort) | — | \mathcal{} {n \log n} | \mathcal{} n^2 | \mathcal{} n | Yes | Insertion |  |
| [Merge sort](http://en.wikipedia.org/wiki/Merge_sort) | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | Depends; worst case is  \mathcal{} n | Yes | Merging | Used to sort this table in Firefox [[2]](http://mxr.mozilla.org/seamonkey/source/js/src/jsarray.c). |
| [Patience sorting](http://en.wikipedia.org/wiki/Patience_sorting) | — | — | \mathcal{} n \log n | \mathcal{} n | No | Insertion & Selection | Finds all the [longest increasing subsequences](http://en.wikipedia.org/wiki/Longest_increasing_subsequence) within O(*n* log *n*) |
| [Quicksort](http://en.wikipedia.org/wiki/Quicksort) | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} n^2 | \mathcal{} \log n | Depends | Partitioning | Quicksort is usually done in place with O(log(*n*)) stack space.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] Most implementations are unstable, as stable in-place partitioning is more complex. [Naïve](http://en.wikipedia.org/wiki/Na%C3%AFve_algorithm) variants use an O(*n*) space array to store the partition.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] |
| [Selection sort](http://en.wikipedia.org/wiki/Selection_sort) | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | No | Selection | Stable with O(n) extra space, for example using lists [[5]](http://www.algolist.net/Algorithms/Sorting/Selection_sort). Used to sort this table in Safari or other Webkit web browser [[6]](http://svn.webkit.org/repository/webkit/trunk/Source/JavaScriptCore/runtime/ArrayPrototype.cpp). |
| [Shell sort](http://en.wikipedia.org/wiki/Shell_sort) | \mathcal{} n | \mathcal{} n (\log n)^2  or  \mathcal{} n^{3/2} | Depends on gap sequence; best known is \mathcal{} n (\log n)^2 | \mathcal{} 1 | No | Insertion |  |
| [Smoothsort](http://en.wikipedia.org/wiki/Smoothsort) | \mathcal{} {n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {1} | No | Selection | An [adaptive sort](http://en.wikipedia.org/wiki/Adaptive_sort) - \mathcal{} {n} comparisons when the data is already sorted, and 0 swaps. |
| [Strand sort](http://en.wikipedia.org/wiki/Strand_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} n | Yes | Selection |  |
| [Timsort](http://en.wikipedia.org/wiki/Timsort) | \mathcal{} {n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} n | Yes | Insertion & Merging | \mathcal{} {n} comparisons when the data is already sorted or reverse sorted. |
| [Tournament sort](http://en.wikipedia.org/wiki/Tournament_sort) | — | \mathcal{} n \log n | \mathcal{} n \log n |  |  | Selection |  |

The following table describes [integer sorting](http://en.wikipedia.org/wiki/Integer_sorting) algorithms and other sorting algorithms that are not [comparison sorts](http://en.wikipedia.org/wiki/Comparison_sort). As such, they are not limited by a \Omega\left( {n \log n} \right)lower bound. Complexities below are in terms of *n*, the number of items to be sorted, *k*, the size of each key, and *d*, the digit size used by the implementation. Many of them are based on the assumption that the key size is large enough that all entries have unique key values, and hence that *n* << 2*k*, where << means "much less than."

| Non-comparison sorts | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Best** | **Average** | **Worst** | **Memory** | **Stable** | ***n* << 2*k*** | **Notes** |
| [Pigeonhole sort](http://en.wikipedia.org/wiki/Pigeonhole_sort) | — | \;n + 2^k | \;n + 2^k | \;2^k | Yes | Yes |  |
| [Bucket sort](http://en.wikipedia.org/wiki/Bucket_sort) (uniform keys) | — | \;n+k | \;n^2 \cdot k | \;n \cdot k | Yes | No | Assumes uniform distribution of elements from the domain in the array.[[2]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-clrs-1) |
| [Bucket sort](http://en.wikipedia.org/wiki/Bucket_sort) (integer keys) | — | \;n+r | \;n+r | \;n+r | Yes | Yes | r is the range of numbers to be sorted. If r = \mathcal{O}\left( {n} \right)then Avg RT = \mathcal{O}\left( {n} \right)[[3]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-gt-2) |
| [Counting sort](http://en.wikipedia.org/wiki/Counting_sort) | — | \;n+r | \;n+r | \;n+r | Yes | Yes | r is the range of numbers to be sorted. If r = \mathcal{O}\left( {n} \right)then Avg RT = \mathcal{O}\left( {n} \right)[[2]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-clrs-1) |
| [LSD Radix Sort](http://en.wikipedia.org/wiki/Radix_sort#Least_significant_digit_radix_sorts) | — | \;n \cdot \frac{k}{d} | \;n \cdot \frac{k}{d} | \mathcal{} n | Yes | No | [[3]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-gt-2)[[2]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-clrs-1) |
| [MSD Radix Sort](http://en.wikipedia.org/wiki/Radix_sort#Most_significant_digit_radix_sorts) | — | \;n \cdot \frac{k}{d} | \;n \cdot \frac{k}{d} | \mathcal{} n + \frac{k}{d} \cdot 2^d | Yes | No | Stable version uses an external array of size n to hold all of the bins |
| [MSD Radix Sort](http://en.wikipedia.org/wiki/Radix_sort#Most_significant_digit_radix_sorts) | — | \;n \cdot \frac{k}{d} | \;n \cdot \frac{k}{d} | \frac{k}{d} \cdot 2^d | No | No | In-Place. k / d recursion levels, 2d for count array |
| [Spreadsort](http://en.wikipedia.org/wiki/Spreadsort) | — | \;n \cdot \frac{k}{d} | \;n \cdot \left( {\frac{k}{s} + d} \right) | \;\frac{k}{d} \cdot 2^d | No | No | Asymptotics are based on the assumption that n << 2k, but the algorithm does not require this. |

Sorting algorithms used in computer science are often classified by:

* Computational complexity (worst, average and best behavior) of element comparisons in terms of the size of the list (*n*). For typical sorting algorithms good behavior is O(*n* log *n*) and bad behavior is O(*n*2). (See [Big O notation](http://en.wikipedia.org/wiki/Big_O_notation).) Ideal behavior for a sort is O(*n*), but this is not possible in the average case. [Comparison-based sorting algorithms](http://en.wikipedia.org/wiki/Comparison_sort), which evaluate the elements of the list via an abstract key comparison operation, need at least O(*n* log *n*) comparisons for most inputs.
* [Computational complexity](http://en.wikipedia.org/wiki/Computational_complexity_theory) of swaps (for "in place" algorithms).
* Memory usage (and use of other computer resources). In particular, some sorting algorithms are "[in place](http://en.wikipedia.org/wiki/In-place_algorithm)". Strictly, an in place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(*n*)) additional memory is considered "in place".
* Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
* Stability: [stable sorting algorithms](http://en.wikipedia.org/wiki/Sorting_algorithm#Stability) maintain the relative order of records with equal keys (i.e., values).
* Whether or not they are a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). A comparison sort examines the data only by comparing two elements with a comparison operator.
* General method: insertion, exchange, selection, merging, *etc.*. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.
* Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be [adaptive](http://en.wikipedia.org/wiki/Adaptive_sort).

This project report has been prepared under the valuable suggestion, support and guidance of TOUFIQUE AHAMMAD GAZI, faculty of ELITE POLYTECHNIC INSTITUTE. It was highly rewarding to work with him and we express our heartiest gratitude to him.

Finally, I take this opportunity to thank to Dr. Arun Kumar Dey, Principal of Elite Polytechnic Institute for giving us the scope of carrying out the project work.

Thanking You All……

MD Rameez Raja

Manish Kumar Gupta

Arup Singha

MD Imran Ansari

Sujoya Das

Swarna Basu

Sarita Chaurasia

\* The objective is to take an unordered set of comparable data items and arrange them in order.

\* We will usually sort the data into ascending order sorting into descending order is very similar.

\* Data can be sorted in various ADTs, such as arrays and TREES.

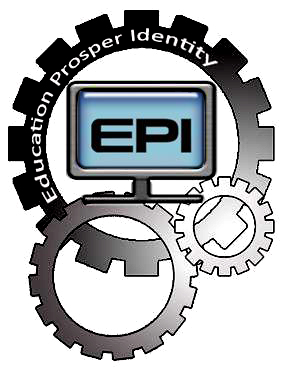
In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1.The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);

2.The output is a permutation (reordering) of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.[1] Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2006). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

ELITE POLYTECHNIC INSTITUTE



DIPLOMA IN COMPUTER SCIENCE & TECHNOLOGY,2010-13

TO WHOME IT MAY CONCERN

This is to certify that Miss. Sujoya Das Regd. No. D101128469 of Computer Science & Technology, batch(2010-13) is a bonafide student of Elite Polytechnic Institute under West Bengal State Council Of Technical Education has carried out a project entitled as “Comparative Study Over Sorting Algorithm” as a partial fulfillment of DCST programme .

It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report. The project report has been approved as it satisfied the academic requirement in respect of project work prescribed for Diploma in Computer Science & Technology.