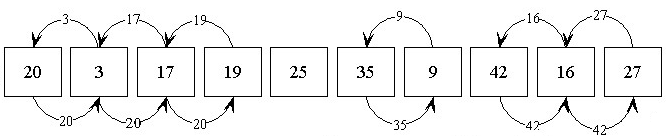
* + [Bubble sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Bubble_sort)
  + [Insertion sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Insertion_sort)
  + [Radix sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Radix_sort)
  + [Heap sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Heapsort)
  + [Merge sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Merge_sort)
  + [Quick sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Quicksort)
  + [Shell sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Shell_sort)
  + [Selection sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Selection_sort)
  + [Timsort](http://en.wikipedia.org/wiki/Sorting_algorithm#Timsort)
  + Comb sort

**Bubble sort :**

A bubble sort, a sorting algorithm that continuously steps through a list, swapping items until they appear in the correct order .

Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements (adjacent elements), and if the first is greater than the second, it swaps them. It continues doing this for the end of the data set and do this again to get a sorted data .



Bubble sort has a worst and average case in timing O(n2) and best case of O(n)

Easier to implement, but slower than Insertion sort, but it is bad in timing of unsorted data, And the time difference in sorting sorted and unsorted data is large.

So it is fast in sorted data but too slow with reversed sorted data.

### Insertion sort:

### Insertion sort does exactly what you would expect: it inserts each element of the array into its proper position, leaving progressively larger stretches of the array sorted. What this means in practice is that the sort iterates down an array, and the part of the array already covered is in order; then, the current element of the array is inserted into the proper position at the head of the array, and the rest of the elements are moved down, using the space just vacated by the element inserted as the final space.  Here is an example: for sorting the array the array 52314 First, 2 is inserted before 5, resulting in 25314 Then, 3 is inserted between 2 and 5, resulting in 23514 Next, one is inserted at the start, 12354 Finally, 4 is inserted between 3 and 5, 12345

Insertion sort is good in small data , but it is not efficient in large data , but still better than the bubble sort, and the algorithm is affected with the input data so it takes O(n) with sorted data while O(n2) with reversed data.

: Heap sort

This sort type is based on the concept of heap, first we build the heap from the data array , then we take the root which is the min/ max value and replace it with the last element in the array and then maintain heap but excluding the last element , and so on.

Heap sort is good since its time is not so long and it doesn't take an extra space other than the array , and it is not affected with input data , the worst and best case are O(nlogn).

.

|  |
| --- |
|  |
| |  |  |  | | --- | --- | --- | | extract maximum element | delete last leaf and write its label to the root | exchange label with the maximum label of its direct descendants | |  |  |  | | (a) | (b) | (c) | |
|  |
|  |
|  |
|  |

|  |
| --- |
|  |
| |  |  | | --- | --- | | exchange label with the maximum label of its direct descendants | heap restored | |  |  | | (d) | (e) | |

 Retrieving the maximum element and restoring the heap

Heap sort is a relatively simple algorithm built upon the heap data structure. A naive implementation requires additional space, but it is possible to do a heap sort in place. Heap sort has guaranteed O(n\*log(n))) performance, though the constant factor is typically a bit higher than for other algorithms such as quicksort. Heap sort is not a stable sort, so the original ordering of equal elements may not be maintained.

Radix sort:

Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. However, the process adopted by this sort method is somewhat counterintuitive, in the sense that the numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so on till the most significant digit.

To appreciate Radix Sort, consider the following analogy: Suppose that we wish to sort a deck of 52 playing cards (the different suits can be given suitable values, for example 1 for Diamonds, 2 for Clubs, 3 for Hearts and 4 for Spades). The 'natural' thing to do would be to first sort the cards according to suits, then sort each of the four seperate piles, and finally combine the four in order. This approach, however, has an inherent disadvantage. When each of the piles is being sorted, the other piles have to be kept aside and kept track of. If, instead, we follow the 'counterintuitive' aproach of first sorting the cards by value, this problem is eliminated. After the first step, the four seperate piles are combined in order and then sorted by suit. If a stable sorting algorithm (i.e. one which resolves a tie by keeping the number obtained first in the input as the first in the output) it can be easily seen that correct final results are obtained.

As has been mentioned, the sorting of numbers proceeds by sorting the least significant to most significant digit. For sorting each of these digit groups, a stable sorting algorithm is needed. Also, the elements in this group to be sorted are in the fixed range of 0 to 9. Both of these characteristics point towards the use of Counting Sort as the sorting algorithm of choice for sorting on each digit (If you haven't read the description on Counting Sort already, please do so now).

The time complexity of the algorithm is as follows: Suppose that the n input numbers have maximum k digits. Then the Counting Sort procedure is called a total of k times. Counting Sort is a linear, or O(n) algorithm. So the entire Radix Sort procedure takes O(kn) time. If the numbers are of finite size, the algorithm runs in O(n) asymptotic time.

A graphical representation of the entire procedure has been provided in the attached applet. This program takes 8 randomly generated 3 digit integers as input and sorts them using Radix Sort.

Best way to sort physical objects

• “algorithm” used to sort punched cards

by hand or by machine

Radix sort is good for cards.

• Sorting cards involves moving them around.

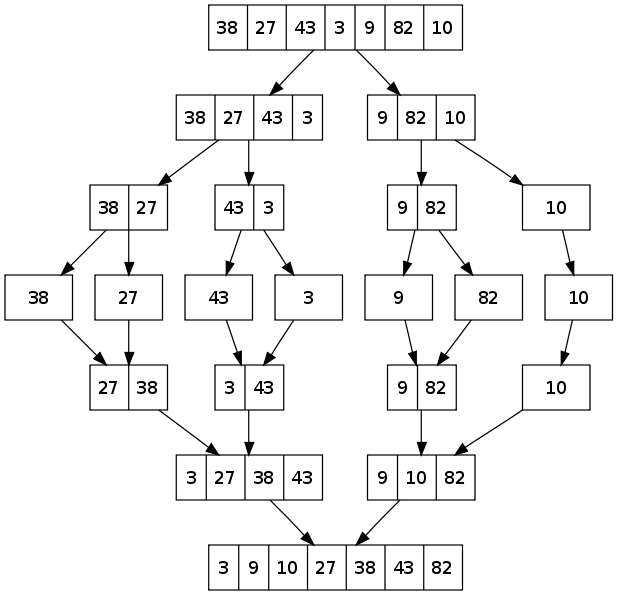
Easy to move cards around in stacks.

Not so good in computer memory.

• Sorting involves copying data.

Each data item must be copied individually.

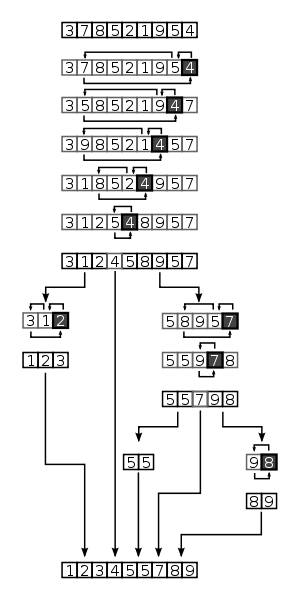
Merge Sort :

[](http://upload.wikimedia.org/wikipedia/commons/e/e6/Merge_sort_algorithm_diagram.svg)

Merge sort works by separating the array into sub arrays and keep separating them until each element is kept alone then the elements are merged together with order so the array is created again but sorted.

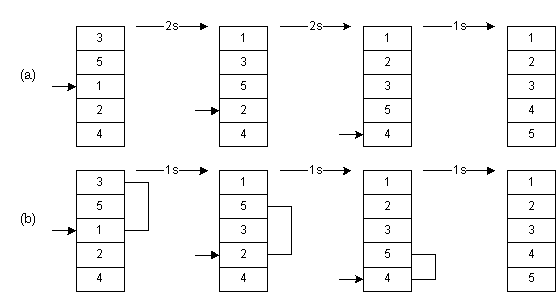
Merge sort is one of the best sort types as its worst case same as the best case O(nlogn)

Merge sort is a fast, stable sorting routine with guaranteed O(n\*log(n)) efficiency. When sorting arrays, merge sort requires additional scratch space proportional to the size of the input array. Merge sort is relatively simple to code and offers performance typically only slightly below that of quicksort:

[](http://upload.wikimedia.org/wikipedia/commons/a/af/Quicksort-diagram.svg)

Quicksort is a relatively simple sorting algorithm using the divide-and-conquer recursive procedure. It is the quickest comparison-based sorting algorithm in practice with an average running time of O(n log(n)). Crucial to quicksort's speed is a balanced partition decided by a well chosen pivot. Quicksort has the advantage of sorting in place, and it works well even in virtual memory environments.

Shell Sort:



It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

### Selection sort :

*Selection sort* is an [in-place](http://en.wikipedia.org/wiki/In-place_algorithm) [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). It has [O](http://en.wikipedia.org/wiki/Big_O_notation)(*n*2) complexity, making it inefficient on large lists, and generally performs worse than the similar[insertion sort](http://en.wikipedia.org/wiki/Insertion_sort). Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than*n* swaps, and thus is useful where swapping is very expensive.

The algorithm works as follows:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Effectively, the list is divided into two parts: the sublist of items already sorted, which is built up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

Here is an example of this sort algorithm sorting five elements:

64 25 12 22 11

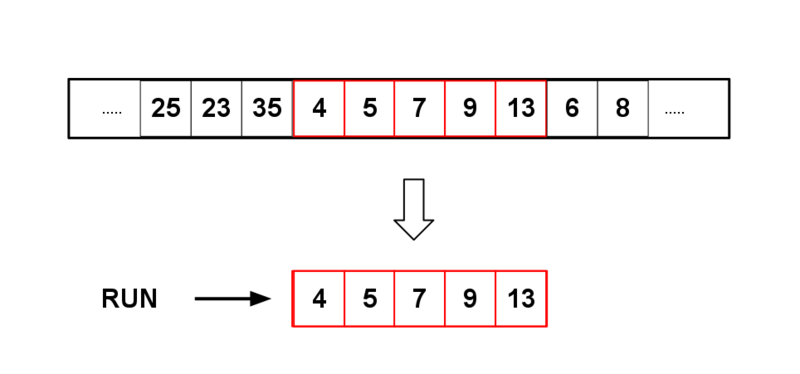
11 25 12 22 64

11 12 25 22 64

11 12 22 25 64

11 12 22 25 64

*Timsort:*

[](http://upload.wikimedia.org/wikipedia/commons/6/63/Selection_of_minrun_by_timsort.png)*Timsort* finds runs in the data, creates runs with insertion sort if necessary, and then uses merge sort to create the final sorted list. It has the same complexity (O(nlogn)) in the average and worst cases, but with pre-sorted data it goes down to O(n).

**comb sort :**

Comb sort is a relatively simplistic [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) originally designed by [Włodzimierz Dobosiewicz](http://en.wikipedia.org/w/index.php?title=W%C5%82odzimierz_Dobosiewicz&action=edit&redlink=1" \o "Włodzimierz Dobosiewicz (page does not exist)) in 1980. Later it was rediscovered and popularized by [Stephen Lacey](http://en.wikipedia.org/w/index.php?title=Stephen_Lacey&action=edit&redlink=1) and [Richard Box](http://en.wikipedia.org/w/index.php?title=Richard_Box&action=edit&redlink=1) with a [Byte Magazine](http://en.wikipedia.org/wiki/Byte_Magazine) [article published in April 1991](http://cs.clackamas.cc.or.us/molatore/cs260Spr03/combsort.htm). Comb sort improves on [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), and rivals algorithms like [Quicksort](http://en.wikipedia.org/wiki/Quicksort" \o "Quicksort) ([visual comparison](http://scripts.franciscocharrua.com/javascript/sort-algorithms/compare.php?comb&quick)). The basic idea is to eliminate *turtles*, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. (*Rabbits*, large values around the beginning of the list, do not pose a problem in bubble sort.).

In bubble sort, when any two elements are compared, they always have a *gap* (distance from each other) of 1. The basic idea of comb sort is that the gap can be much more than one. ([Shell sort](http://en.wikipedia.org/wiki/Shell_sort) is also based on this idea, but it is a modification of [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) rather than bubble sort.)

The gap starts out as the length of the list being sorted divided by the *shrink factor* (generally 1.3; see below), and the list is sorted with that value (rounded down to an integer if needed) for the gap. Then the gap is divided by the shrink factor again, the list is sorted with this new gap, and the process repeats until the gap is 1. At this point, comb sort continues using a gap of 1 until the list is fully sorted. The final stage of the sort is thus equivalent to a bubble sort, but by this time most turtles have been dealt with, so a bubble sort will be efficient.

Conclusion:   
Through my study of the kinds of sort i found that:

For huge data:

For sorted data the insertion sort is very good . Anyway we can look at

the difference between the best case and the last case.

For reversed order data the best sort type is the quick sort.

Quick sort also is the best with random data.

For small data :

Shell sort is the fastest for sorted data.

For reversed order and random data the quick sort is also the fastest.

The radix sort is theoretically from the fastest type of sorts , but it is quite slow .

I see this because the implementation of the linked list as it keep allocating memory locations

and removing another locations , and this takes very long time comparing with the CPU time.

Reference :

1-<https://docs.google.com/viewer?a=v&q=cache:N6rR9LqtoJUJ:www.cse.unr.edu/~bebis/CS477/Lect/InsertionSortBubbleSortSelectionSort.ppt+bubble+sort+algorithm+analysis&hl=ar&gl=ps&pid=bl&srcid=ADGEESgIy4jsySbHCQAn_bAQ2uy7SrgPlokHig1ikwnAV_Z_QGIopob2QSoYmSJIMYAONX-l2Zyugkmw7BVxXmJRa39tLDROhrxd0ASsr1ah9uKRkA0hwW0wUO1mAnDEr_rxW1kLHq5q&sig=AHIEtbQoW2OX4ft0ulAMsU3zuJ716mkn6Q>

2- <http://en.wikipedia.org/wiki/Sorting_algorithm>

3- <http://www.cprogramming.com/tutorial/computersciencetheory/sorting2.html>

4- <http://en.wikipedia.org/wiki/Insertion_sort>

5-<http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/heap/heapen.htm>

6- <http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/radixSort/radix.html>

7-<https://docs.google.com/viewer?a=v&q=cache:46GWBsEazj0J:https://sakai.rutgers.edu/access/content/group/fa48a20a-c32f-4bea-9227-bf5aa7da27ef/Notes/Sorting3.pdf+picture+radix+sort&hl=ar&gl=ps&pid=bl&srcid=ADGEESi8QH9Ompn4dd6cBCWKMuG1EukPQl8lk3mK5Npx_ibJLWfPB9fDmIeNQkl-19eNpT3R65rx64dOwGEJiV4gBDTYV75yrdtSCaFh83qs31m7BaW6JWJBPuBrwznlWVGHxsEcXLix&sig=AHIEtbRCHtQY0LQ56U7WdcGGsQoM_RwQcg>

8- <http://en.wikipedia.org/wiki/Merge_sort>

9- <http://www.cprogramming.com/tutorial/computersciencetheory/mergesort.html>

10- <http://www.cprogramming.com/tutorial/computersciencetheory/quicksort.html>

11-<http://en.wikipedia.org/wiki/Selection_sort>