**Cocktail Sort (Best: n, Worst: n^2, Average: n^2)**

Cocktail sort is an optimised version of bubble sort. Cocktail sort traverses the array forwards, and once it reaches the end of the array, it traverses backwards. Typically, the cocktail sort is twice as fast as the bubble sort as it eliminates “turtles” which are prevalent in bubble sort. The concept of “turtles” is that smaller elements at the end of the array are harder to move in bubble sort as bubble sort only moves in one direction, effectively only sorting larger numbers in the array efficiently (“rabbits”). Consider if an array is [5, 5, 5, 5, 1], bubble sort would need four passes (n-1). However, the cocktail sort would only need “one pass”, the equivalent to two bubble sort passes. One advantage other than the elimination of “Turtles” is that it ‘remembers’ the last swap done (The elements that have already been sorted). This means that the algorithm will not pass through beyond a limit where the elements have already been sorted. Hence, every iteration/pass will get shorter. It is commonly used as an educational tool to encourage algorithmic thinking.

**Comb Sort (Best: nlogn, Worst: n^2, Average: n^2)**

Comb sort is an algorithm that utilises “gaps” similar to Shell sort which similarly, eliminates the issue of “turtles” like Cocktail shaker sort as it is able to swap between first and last elements. Consider the array [5, 5, 5, 5, 1] again. Comb sort would be able to swap the first and last element, effectively needing one pass to sort. Although, one downside is that the criteria for finishing the sort is having a sorted list. This means that the gap needs to be 1 in the first place in order to verify that a list has been sorted. The most effective “gap” for sorting arrays is a shrink factor of 1.3. Initially, the gap would be the length of the array, but for every pass of the array, the gap would shrink by approximately 1.3. The shrink factor is essential to the run-time complexity of the algorithm. If the shrink factor is too small, it may cause an unnecessary number of passes. If the shrink factor is too large, it may not effectively deal with the issue of “turtles” as the gaps would quickly become small. It is commonly used as an educational tool to encourage algorithmic thinking.

**Shell sort - Ciura’s Gap Sequence (Best: (nlogn)^2\*, Worst: (nlogn)^2\* , Average: (nlogn)^2\*)**

Shell sort optimises insertion sort by utilising gaps, similar to how Comb sort optimises bubble sort by utilising gaps, shrinking at every pass. The basis of shell sort is to reduce the number of exchanges and comparisons between numbers. This means its more efficient than insertion sort as it skips the in-between elements The most optimal gap sequence is empirically proven to be Marcin Ciura’s gap sequence, where the sequence approximates a shrink factor of 2.5 down to 2.25, compared to the original n//2 suggested by the original authors where n is the length of the array as the original time complexity is n^2 [1]. The exact Big O notation of various gap sequences remains an open problem as they are resistant to runtime analysis, and simply only conjectured. Its practical application is used by various coding libraries and compression algorithm software. Shell sort is also used in the Linux kernel.

\*At least better than nlogn^2 as Ciura’s gap sequence exact time complexity remains an open problem

[1] (See <https://upload.wikimedia.org/wikipedia/commons/f/fe/Shell_sort_average_number_of_comparisons_%28English%29.svg>) [Ciura is Ci01]

**Heap sort (Best: nlogn, Worst: nlogn, Average: nlogn)**

Heap sort utilises the heap data structure to sort. A heap data structure is essentially a “sorted binary tree” where it has a root with nodes. Using recursion, we build a max heap where the root is the largest number, with the bottom of the structure being the smallest number. After building the max heap, we then remove the root (the largest number) and swap it with the last element of the heap for that node. We then swap until the largest element is at the root and repeat. Its main advantage is that the worst case is nlogn, unlike quicksort where it is n^2. This good worst time complexity is because the height of the heap (binary tree) is log(n). When building the heap, it takes n/2 elements multiplied by the height (number of levels), which is also nlogn and the sort makes multiple log(n) swaps/comparisons from the root to the leaf. Heap sort draws similarity to selection sort such that it takes the smallest number first. Its application includes priority queues which are used in Djikstra’s shortest path algorithm. It is also utilised in the Linux Kernel.