Data Structure and Algorithms

**Insertion sort**

Insertion sort is a simple and efficient sorting algorithm, that creates the final sorted array one element at the time. It is usually implemented when the user has a small data set.

This algorithm isn’t suitable for large data sets as its average and worst-case complexity are of O(n^2) when n is the number of items.

## How does Insertion Sort Work?

We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position.

Insertion Sort

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

Insertion Sort

These values are not in a sorted order.

Insertion Sort

.So, we swap them.

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again, we find 14 and 10 in an unsorted order.

Insertion Sort

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

### **Algorithm**

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Algorithm we have a bigger picture of how this sorting technique works, so we can drive simple steps by which we can achieve insertion sort.

* Step1: - if it is first element, it is already sorted, return 1;
* Step2: - pick next element
* Step3: - Compare with all elements in the sorted sub-list
* Step4: - shift all elements in the sorted sub-list that is greater than the value to be sorted
* Step5: - Insert the value
* Step5: - Repeat until the list is sorted

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison based algorithmic the which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the left most element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

The algorithm is not suitable for large data sets as its average and worst-case complexities are of O(n^2), where n is the number of items.

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.4

Selection Sort

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

Selection Sort

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Now, let us learn some programming aspects of selection sort.

Algorithm

* Step1: - Set MIN to location 0
* Step2: - Search the minimum elements in the list
* Step3: - Swap with value at the location MIN
* Step4: - Increment MIN to point to next element
* Step5: - Repeat until list in sorted

Quick sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.

A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(nLogn) and image.png(n2), respectively.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.

Merge sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

## How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following

−

Unsorted Array

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4

Merge Sort Division

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

Merge Sort Division

We further divide these arrays and we achieve atomic value which can no more be divided.

Merge Sort Division

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

Merge Sort Combine

After the final merging, the list should look like this −

Merge Sort

Now we should learn some programming aspects of merge sorting.

### **Algorithm**

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

* Step1: - if it is only element in the list it is already sorted return.
* Step2: - divide the list recursively into two halves until it can no more be divided
* Step3: - merge the smaller lists into new list in sorted order

Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst-case complexity are of Ο(n2) where **n** is the number of items.

## How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.

Bubble Sort

Bubble sort starts with very first two elements, comparing them to check which one is greater.

Bubble Sort

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

Bubble Sort

We find that 27 is smaller than 33 and these two values must be swapped.

Bubble Sort

The new array should look like this −

Bubble Sort

Next we compare 33 and 35. We find that both are in already sorted positions.

Bubble Sort

Then we move to the next two values, 35 and 10.

Bubble Sort

We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

Bubble Sort

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Bubble Sort

Notice that after each iteration, at least one value moves at the end.

Bubble Sort

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Bubble Sort

Now we should look into some practical aspects of bubble sort.

## Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

The new array should look like this −

Bubble Sort

Next we compare 33 and 35. We find that both are in already sorted positions.

Bubble Sort

Then we move to the next two values, 35 and 10.

Bubble Sort

We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

Bubble Sort

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Bubble Sort

Notice that after each iteration, at least one value moves at the end.

Bubble Sort

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Bubble Sort

Now we should look into some practical aspects of bubble sort.

## Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

Shell Sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on Knuth's formula as −

### **Knuth's Formula**

H = h \* 3 + 1

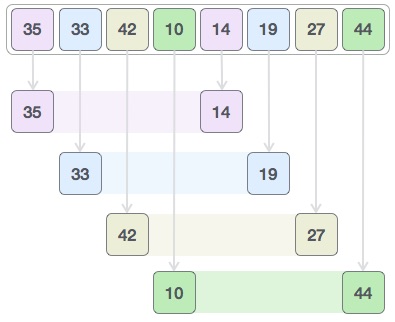
where –

h is interval with initial value 1

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is Ο(n), where n is the number of items. And the worst-case space complexity is O(n).

## How Shell Sort Works?

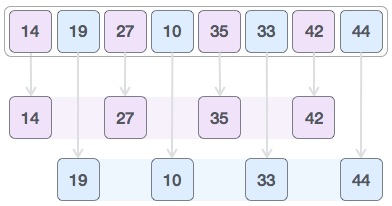
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −

Shell Sort

Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −

Shell Sort

Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −



We see that it required only four swaps to sort the rest of the array.

### **Algorithm**

Following is the algorithm for shell sort.

Fibonacci

Fibonacci series generates the subsequent number by adding two previous numbers. Fibonacci series starts from two numbers − **F0 & F1**. The initial values of F0 & F1 can be taken 0, 1 or 1, 1 respectively.

Fibonacci series satisfies the following conditions −

Fn = Fn-1 + Fn-2

Hence, a Fibonacci series can look like this −

F8 = 0 1 1 2 3 5 8 13

or, this −

F8 = 1 1 2 3 5 8 13 21

For illustration purpose, Fibonacci of F8 is displayed as −

Array Size

2D. Array

Searching in Array

Get Set Max Min on Array

Binary Search in Array

Static vs Dynamic Arrays

Insertion and Appending in Array

Deleting from Array

Linked List

Display Linked List

Count and Sum Linked List

Linear Probing

Chaining

Static Variables in Recursion

Three Recursion

Indirect Recursion

Nested Recursion

Head Recursion

Sum od N natural numbers