**Sorting**

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

**For example**: The below list of characters is sorted in increasing order of their ASCII values. That is, the character with lesser ASCII value will be placed first than the character with higher ASCII value.

(INPUT) z a h r a =🡺 a a h r z (OUTPUT)

# 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.

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

## Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows −

*procedure bubbleSort( list : array of items )*

*loop = list.count;*

*for i = 0 to loop-1 do:*

*swapped = false*

*for j = 0 to loop-1 do:*

*/\* compare the adjacent elements \*/*

*if list[j] > list[j+1] then*

*/\* swap them \*/*

*swap( list[j], list[j+1] )*

*swapped = true*

*end if*

*end for*

*/\*if no number was swapped that means*

*array is sorted now, break the loop.\*/*

*if(not swapped) then*

*break*

*end if*

*end for*

*end procedure return list*

**Selection Sort**

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in 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 leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n2), where **n** is the number of items.

## How Selection Sort Works?

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.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

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

**Step 1** − Set MIN to location 0

**Step 2** − Search the minimum element in the list

**Step 3** − Swap with value at location MIN

**Step 4** − Increment MIN to point to next element

**Step 5** − Repeat until list is sorted

### Pseudocode

*procedure selection sort*

*list : array of items*

*n : size of list*

*for i = 1 to n - 1*

*/\* set current element as minimum\*/*

*min = i*

*/\* check the element to be minimum \*/*

*for j = i+1 to n*

*if list[j] < list[min] then*

*min = j;*

*end if*

*end for*

*/\* swap the minimum element with the current element\*/*

*if indexMin != i then*

*swap list[min] and list[i]*

*end if*

*end for*

*end procedure*

**Insertion Sort**

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). 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 Insertion Sort Works?

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.

**Step 1** − If it is the first element, it is already sorted. return 1;

**Step 2** − Pick next element

**Step 3** − Compare with all elements in the sorted sub-list

**Step 4** − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

**Step 5** − Insert the value

**Step 6** − Repeat until list is sorted

## Pseudocode

*procedure insertionSort( A : array of items )*

*int holePosition*

*int valueToInsert*

*for i = 1 to length(A) inclusive do:*

*/\* select value to be inserted \*/*

*valueToInsert = A[i]*

*holePosition = i*

*/\*locate hole position for the element to be inserted \*/*

*while holePosition > 0 and A[holePosition-1] > valueToInsert do:*

*A[holePosition] = A[holePosition-1]*

*holePosition = holePosition -1*

*end while*

*/\* insert the number at hole position \*/*

*A[holePosition] = valueToInsert*

*end for*

*end procedure*

**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

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

Merge Sort Combine

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.

**Step 1** − if it is only one element in the list it is already sorted, return.

**Step 2** − divide the list recursively into two halves until it can no more be divided.

**Step 3** − merge the smaller lists into new list in sorted order.

### Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

*procedure mergesort( var a as array )*

*if ( n == 1 ) return a*

*var l1 as array = a[0] ... a[n/2]*

*var l2 as array = a[n/2+1] ... a[n]*

*l1 = mergesort( l1 )*

*l2 = mergesort( l2 )*

*return merge( l1, l2 )*

*end procedure*

*procedure merge( var a as array, var b as array )*

*var c as array*

*while ( a and b have elements )*

*if ( a[0] > b[0] )*

*add b[0] to the end of c*

*remove b[0] from b*

*else*

*add a[0] to the end of c*

*remove a[0] from a*

*end if*

*end while*

*while ( a has elements )*

*add a[0] to the end of c*

*remove a[0] from a*

*end while*

*while ( b has elements )*

*add b[0] to the end of c*

*remove b[0] from b*

*end while*

*return c*

*end procedure*

**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

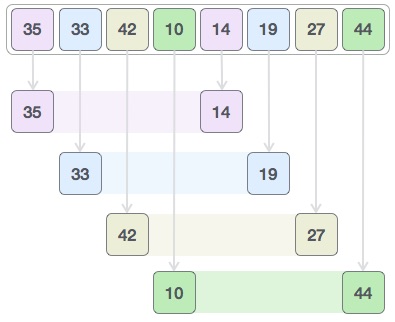
where −

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?

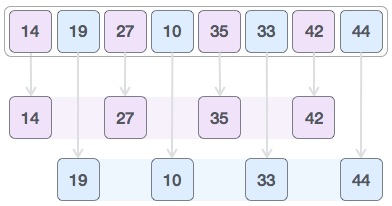
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values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

**Step 1** − Initialize the value of *h*

**Step 2** − Divide the list into smaller sub-list of equal interval *h*

**Step 3** − Sort these sub-lists using **insertion sort**

**Step 3** − Repeat until complete list is sorted

## Pseudocode

Following is the pseudocode for shell sort.

procedure shellSort()

A : array of items

/\* calculate interval\*/

while interval < A.length /3 do:

interval = interval \* 3 + 1

end while

while interval > 0 do:

for outer = interval; outer < A.length; outer ++ do:

/\* select value to be inserted \*/

valueToInsert = A[outer]

inner = outer;

/\*shift element towards right\*/

while inner > interval -1 && A[inner - interval] >= valueToInsert do:

A[inner] = A[inner - interval]

inner = inner - interval

end while

/\* insert the number at hole position \*/

A[inner] = valueToInsert

end for

/\* calculate interval\*/

interval = (interval -1) /3;

end while

end procedure

**Quick Sort**

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

**Pseudo Code for recursive QuickSort function :**

*/\* low --> Starting index, high --> Ending index \*/*

*quickSort(arr[], low, high)*

*{*

*if (low < high)*

*{*

*/\* pi is partitioning index, arr[pi] is now*

*at right place \*/*

*pi = partition(arr, low, high);*

*quickSort(arr, low, pi - 1); // Before pi*

*quickSort(arr, pi + 1, high); // After pi*

*}*

*[](https://www.geeksforgeeks.org/wp-content/uploads/gq/2014/01/QuickSort2.png)}*

**Partition Algorithm**  
There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

*/\* low --> Starting index, high --> Ending index \*/*

*quickSort(arr[], low, high)*

*{*

*if (low < high)*

*{*

*/\* pi is partitioning index, arr[pi] is now*

*at right place \*/*

*pi = partition(arr, low, high);*

*quickSort(arr, low, pi - 1); // Before pi*

*quickSort(arr, pi + 1, high); // After pi*

*}*

*}*

**Pseudo code for partition()**

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

*partition (arr[], low, high)*

*{*

*// pivot (Element to be placed at right position)*

*pivot = arr[high];*

*i = (low - 1) // Index of smaller element*

*for (j = low; j <= high- 1; j++)*

*{*

*// If current element is smaller than the pivot*

*if (arr[j] < pivot)*

*{*

*i++; // increment index of smaller element*

*swap arr[i] and arr[j]*

*}*

*}*

*swap arr[i + 1] and arr[high])*

*return (i + 1)*

*}*

**Illustration of partition() :**

*arr[] = {10, 80, 30, 90, 40, 50, 70}*

*Indexes: 0 1 2 3 4 5 6*

*low = 0, high = 6, pivot = arr[h] = 70*

*Initialize index of smaller element,* ***i = -1***

*Traverse elements from j = low to high-1*

***j = 0*** *: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*

***i = 0***

*arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j*

*// are same*

***j = 1*** *: Since arr[j] > pivot, do nothing*

*// No change in i and arr[]*

***j = 2*** *: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*

***i = 1***

*arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30*

***j = 3*** *: Since arr[j] > pivot, do nothing*

*// No change in i and arr[]*

***j = 4*** *: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*

***i = 2***

*arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped*

***j = 5*** *: Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]*

***i = 3***

*arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped*

We come out of loop because j is now equal to high-1.

**Finally we place pivot at correct position by swapping**

**arr[i+1] and arr[high] (or pivot)**

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after

it.

**Searching**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a **search algorithm** is any [algorithm](https://en.wikipedia.org/wiki/Algorithm) which solves the [search problem](https://en.wikipedia.org/wiki/Search_problem), namely, to retrieve information stored within some data structure, or calculated in the [search space](https://en.wikipedia.org/wiki/Feasible_region) of a [problem domain](https://en.wikipedia.org/w/index.php?title=Problem_domain&action=edit&redlink=1), either with [discrete or continuous values](https://en.wikipedia.org/wiki/Continuous_or_discrete_variable).

**Linear Search**

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



## Algorithm

Linear Search ( Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

## Pseudocode

procedure linear\_search (list, value)

for each item in the list

if match item == value

return the item's location

end if

end for

end procedure

**Binary Search**

Binary search is a fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

## How Binary Search Works?

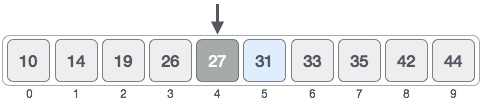
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0 ) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again.

low = mid + 1

mid = low + (high - low) / 2

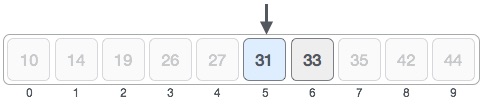
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

## Pseudocode

The pseudocode of binary search algorithms should look like this −

Procedure binary\_search

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exists.

set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

if A[midPoint] < x

set lowerBound = midPoint + 1

if A[midPoint] > x

set upperBound = midPoint - 1

if A[midPoint] = x

EXIT: x found at location midPoint

end while

end procedure