Sorting is nothing but arranging the data in ascending or descending order. The term sorting came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of sorting came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

## Sorting Efficiency

If you ask me, how will I arrange a deck of shuffled cards in order, I would say, I will start by checking every card, and making the deck as I move on.

It can take me hours to arrange the deck in order, but that's how I will do it.

Well, thank god, computers don't work like this.

Since the beginning of the programming age, computer scientists have been working on solving the problem of sorting by coming up with various different algorithms to sort data.

The two main criterias to judge which algorithm is better than the other have been:

1. Time taken to sort the given data.
2. Memory Space required to do so.

## Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

Although it's easier to understand these sorting techniques, but still we suggest you to first learn about [Space complexity](https://www.studytonight.com/data-structures/space-complexity-of-algorithms), [Time complexity](https://www.studytonight.com/data-structures/time-complexity-of-algorithms) and the [searching algorithms](https://www.studytonight.com/data-structures/search-algorithms), to warm up your brain for sorting algorithms.

# Bubble Sort Algorithm

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

## Implementation

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

| [**→Next**](https://www.javatpoint.com/bucket-sort)[**← Prev**](https://www.javatpoint.com/binary-search) Bubble Sort In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.   1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list. 2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list. 3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.  Algorithm :  * **Step 1**: Repeat Step 2 For i = 0 to N-1 * **Step 2**: Repeat For J = i + 1 to N - I * **Step 3**: IF A[J] > A[i] SWAP A[J] and A[i] [END OF INNER LOOP] [END OF OUTER LOOP * **Step 4**: EXIT  Complexity  | **Scenario** | **Complexity** | | --- | --- | | Space | O(1) | | Worst case running time | O(n2) | | Average case running time | O(n) | | Best case running time | O(n2) | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for n-1 times.

It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

NOTE: If you are not familiar with Sorting in data structure, you should first learn [what is sorting](https://www.studytonight.com/data-structures/introduction-to-sorting) to know about the basics of sorting.

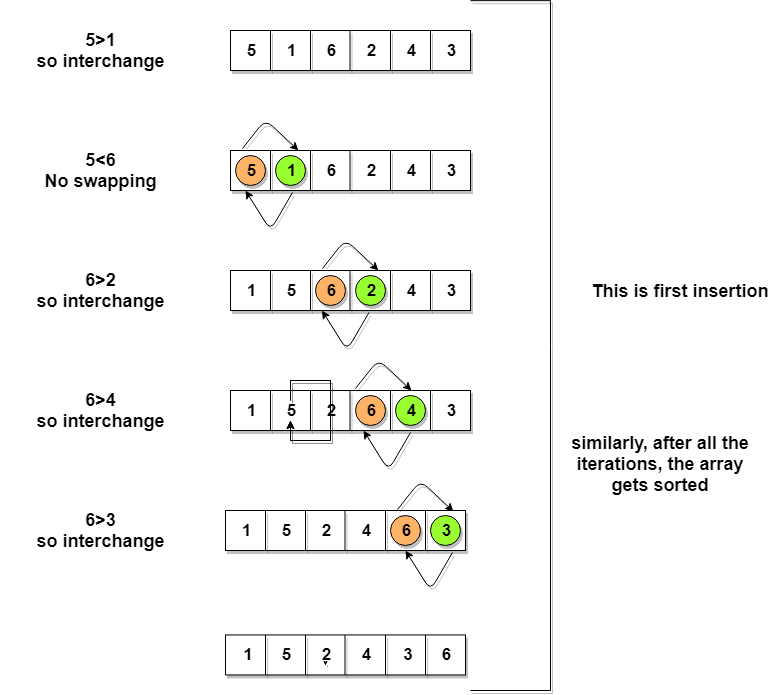
## Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. Repeat Step 1.

Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

Time to write the code for bubble sort:

// below we have a simple C program for bubble sort

#include <stdio.h>

void bubbleSort(int arr[], int n)

{

int i, j, temp;

for(i = 0; i < n; i++)

{

for(j = 0; j < n-i-1; j++)

{

if( arr[j] > arr[j+1])

{

// swap the elements

temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

// print the sorted array

printf("Sorted Array: ");

for(i = 0; i < n; i++)

{

printf("%d ", arr[i]);

}

}

int main()

{

int arr[100], i, n, step, temp;

// ask user for number of elements to be sorted

printf("Enter the number of elements to be sorted: ");

scanf("%d", &n);

// input elements if the array

for(i = 0; i < n; i++)

{

printf("Enter element no. %d: ", i+1);

scanf("%d", &arr[i]);

}

// call the function bubbleSort

bubbleSort(arr, n);

return 0;

}

Although the above logic will sort an unsorted array, still the above algorithm is not efficient because as per the above logic, the outer for loop will keep on executing for 6 iterations even if the array gets sorted after the second iteration.

So, we can clearly optimize our algorithm.

## Complexity Analysis of Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1

Sum = n(n-1)/2

i.e O(n2)

Hence the time complexity of Bubble Sort is O(n2).

The main advantage of Bubble Sort is the simplicity of the algorithm.

The space complexity for Bubble Sort is O(1), because only a single additional memory space is required i.e. for temp variable.

Also, the best case time complexity will be O(n), it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

* Worst Case Time Complexity [ Big-O ]: O(n2)
* Best Case Time Complexity [Big-omega]: O(n)
* Average Time Complexity [Big-theta]: O(n2)
* Space Complexity: O(1)

# Insertion Sort Algorithm

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

## Technique

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, A[j]<=A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

## Complexity

| **Complexity** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Time | Ω(n) | θ(n2) | o(n2) |
| Space |  |  | o(1) |

## 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

## 

Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.

If I give you another card, and ask you to insert the card in just the right position, so that the cards in your hand are still sorted. What will you do?

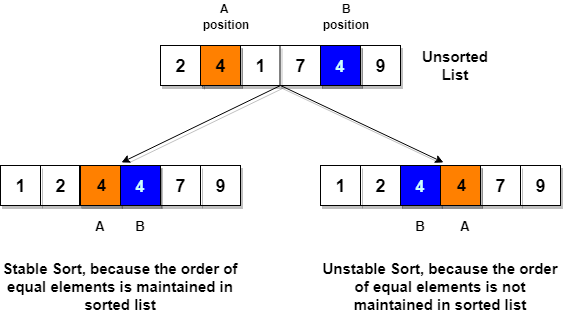
Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will insert the card there.

Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

This is exactly how insertion sort works. It starts from the index 1(not 0), and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted subarray on the left.

Following are some of the important characteristics of Insertion Sort:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a stable sorting technique, as it does not change the relative order of elements which are equal.



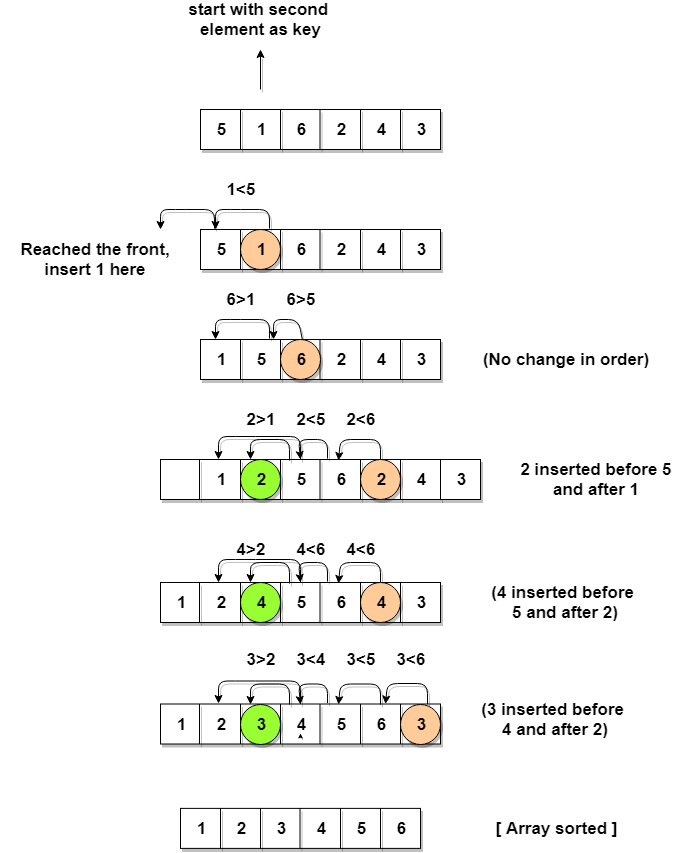
## How Insertion Sort Works?

Following are the steps involved in insertion sort:

1. We start by making the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).
2. We compare the key element with the element(s) before it, in this case, element at index 0:
   * If the key element is less than the first element, we insert the key element before the first element.
   * If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array.



As you can see in the diagram above, after picking a key, we start iterating over the elements to the left of the key.

We continue to move towards left if the elements are greater than the key element and stop when we find the element which is less than the key element.

And, insert the key element after the element which is less than the key element.

### Complexity Analysis of Insertion Sort

As we mentioned above that insertion sort is an efficient sorting algorithm, as it does not run on preset conditions using for loops, but instead it uses one while loop, which avoids extra steps once the array gets sorted.

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n.

Worst Case Time Complexity [ Big-O ]: O(n2)

Best Case Time Complexity [Big-omega]: O(n)

Average Time Complexity [Big-theta]: O(n2)

Space Complexity: O(1)

# Selection Sort Algorithm

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

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

* In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
* In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
* In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

Therefore, by following the above explained process, the elements A[0], A[1], A[2],...., A[n-1] are sorted.

## Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

**A = {10, 2, 3, 90, 43, 56}.**

| **Pass** | **Pos** | **A[0]** | **A[1]** | **A[2]** | **A[3]** | **A[4]** | **A[5]** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 2 | 10 | 3 | 90 | 43 | 56 |
| 2 | 2 | 2 | 3 | 10 | 90 | 43 | 56 |
| 3 | 2 | 2 | 3 | 10 | 90 | 43 | 56 |
| 4 | 4 | 2 | 3 | 10 | 43 | 90 | 56 |
| 5 | 5 | **2** | **3** | **10** | **43** | **56** | **90** |

Sorted A = {2, 3, 10, 43, 56, 90}

## Complexity

| **Complexity** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Time | Ω(n) | θ(n2) | o(n2) |
| Space |  |  | o(1) |

Selection sort is conceptually the most simplest sorting algorithm. This algorithm will first find the smallest element in the array and swap it with the element in the first position, then it will find the second smallest element and swap it with the element in the second position, and it will keep on doing this until the entire array is sorted.

It is called selection sort because it repeatedly selects the next-smallest element and swaps it into the right place.

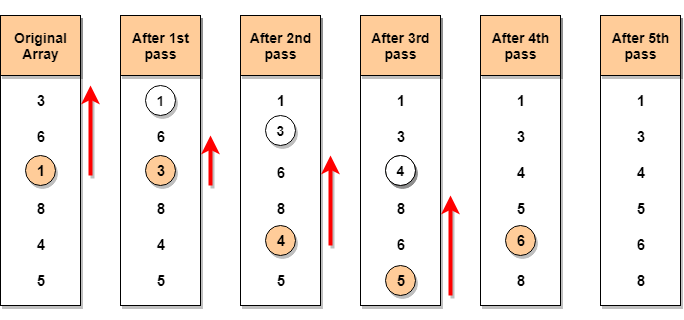
## How Selection Sort Works?

Following are the steps involved in selection sort(for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the second position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Let's consider an array with values {3, 6, 1, 8, 4, 5}

Below, we have a pictorial representation of how selection sort will sort the given array.



In the first pass, the smallest element will be 1, so it will be placed at the first position.

Then leaving the first element, next smallest element will be searched, from the remaining elements. We will get 3 as the smallest, so it will be then placed at the second position.

Then leaving 1 and 3(because they are at the correct position), we will search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

### Finding Smallest Element in a subarray

In selection sort, in the first step, we look for the smallest element in the array and replace it with the element at the first position. This seems doable, isn't it?

Consider that you have an array with following values {3, 6, 1, 8, 4, 5}. Now as per selection sort, we will start from the first element and look for the smallest number in the array, which is 1 and we will find it at the index 2. Once the smallest number is found, it is swapped with the element at the first position.

Well, in the next iteration, we will have to look for the second smallest number in the array. How can we find the second smallest number? This one is tricky?

If you look closely, we already have the smallest number/element at the first position, which is the right position for it and we do not have to move it anywhere now. So we can say, that the first element is sorted, but the elements to the right, starting from index 1 are not.

So, we will now look for the smallest element in the subarray, starting from index 1, to the last index.

Confused? Give it time to sink in.

After we have found the second smallest element and replaced it with element on index 1(which is the second position in the array), we will have the first two positions of the array sorted.

Then we will work on the subarray, starting from index 2 now, and again looking for the smallest element in this subarray.

### Complexity Analysis of Selection Sort

Selection Sort requires two nested for loops to complete itself, one for loop is in the function selectionSort, and inside the first loop we are making a call to another function indexOfMinimum, which has the second(inner) for loop.

Hence for a given input size of n, following will be the time and space complexity for selection sort algorithm:

Worst Case Time Complexity [ Big-O ]: O(n2)

Best Case Time Complexity [Big-omega]: O(n2)

Average Time Complexity [Big-theta]: O(n2)

Space Complexity: O(1)

# Quick Sort Algorithm

# Quick Sort

Quick sort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach. The algorithm processes the array in the following way.

1. Set the first index of the array to left and loc variable. Set the last index of the array to right variable. i.e. left = 0, loc = 0, en d = n - 1, where n is the length of the array.
2. Start from the right of the array and scan the complete array from right to beginning comparing each element of the array with the element pointed by loc.
3. Ensure that, a[loc] is less than a[right].
   1. If this is the case, then continue with the comparison until right becomes equal to the loc.
   2. If a[loc] > a[right], then swap the two values. And go to step 3.
   3. Set, loc = right
4. start from element pointed by left and compare each element in its way with the element pointed by the variable loc. Ensure that a[loc] > a[left]
   1. if this is the case, then continue with the comparison until loc becomes equal to left.
   2. [loc] < a[right], then swap the two values and go to step 2.
   3. Set, loc = left.

## Complexity

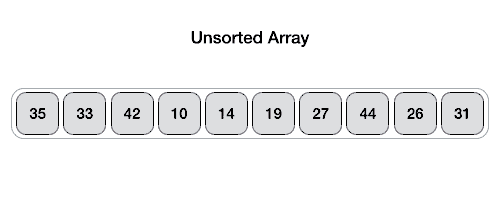
| **Complexity** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Time Complexity | O(n) for 3 way partition or O(n log n) simple partition | O(n log n) | O(n2) |
| Space Complexity |  |  |  |

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(n2), respectively.

## Partition in Quick Sort

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



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

## Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

## Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as −

function partitionFunc(left, right, pivot)

leftPointer = left

rightPointer = right - 1

while True do

while A[++leftPointer] < pivot do

//do-nothing

end while

while rightPointer > 0 && A[--rightPointer] > pivot do

//do-nothing

end while

if leftPointer >= rightPointer

break

else

swap leftPointer,rightPointer

end if

end while

swap leftPointer,right

return leftPointer

end function

## Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

Step 1 − Make the right-most index value pivot

Step 2 − partition the array using pivot value

Step 3 − quicksort left partition recursively

Step 4 − quicksort right partition recursively

## Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm −

procedure quickSort(left, right)

if right-left <= 0

return

else

pivot = A[right]

partition = partitionFunc(left, right, pivot)

quickSort(left,partition-1)

quickSort(partition+1,right)

end if

end procedure

Quick Sort is one of the different [Sorting Technique](https://www.studytonight.com/data-structures/introduction-to-sorting) which is based on the concept of Divide and Conquer, just like [merge sort](https://www.studytonight.com/data-structures/merge-sort). But in quick sort all the heavy lifting(major work) is done while dividing the array into subarrays, while in case of merge sort, all the real work happens during merging the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called partition-exchange sort. This algorithm divides the list into three main parts:

1. Elements less than the Pivot element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as pivot.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as pivot. So after the first pass, the list will be changed like this.

{6 8 17 14 25 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

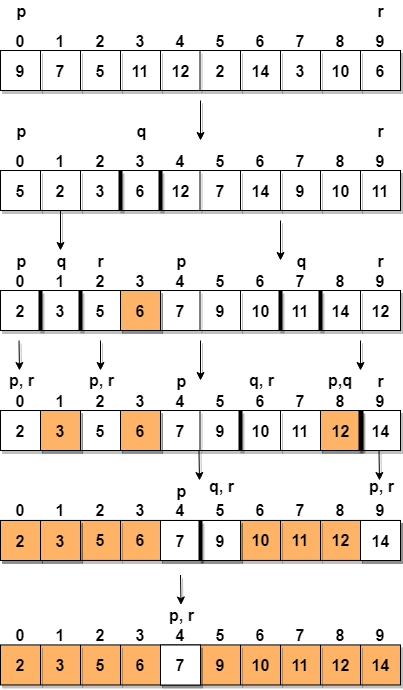
## How Quick Sorting Works?

Following are the steps involved in quick sort algorithm:

1. After selecting an element as pivot, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this partitioning. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the pivot will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the pivot element will be at its final sorted position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of pivot and elements on the right of pivot, and we perform partitioning on them by choosing a pivot in the subarrays.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.



In step 1, we select the last element as the pivot, which is 6 in this case, and call for partitioning, hence re-arranging the array in such a way that 6 will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a pivot for them, in the above diagram, we chose 3 as pivot for the left subarray and 11 as pivot for the right subarray.

And we again call for partitioning.

### Complexity Analysis of Quick Sort

For an array, in which partitioning leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the pivot, hence on the right side.

And if keep on getting unbalanced subarrays, then the running time is the worst case, which is O(n2)

Where as if partitioning leads to almost equal subarrays, then the running time is the best, with time complexity as O(n\*log n).

Worst Case Time Complexity [ Big-O ]: O(n2)

Best Case Time Complexity [Big-omega]: O(n\*log n)

Average Time Complexity [Big-theta]: O(n\*log n)

Space Complexity: O(n\*log n)

As we know now, that if subarrays partitioning produced after partitioning are unbalanced, quick sort will take more time to finish. If someone knows that you pick the last index as pivot all the time, they can intentionally provide you with array which will result in worst-case running time for quick sort.

To avoid this, you can pick random pivot element too. It won't make any difference in the algorithm, as all you need to do is, pick a random element from the array, swap it with element at the last index, make it the pivot and carry on with quick sort.

* Space required by quick sort is very less, only O(n\*log n) additional space is required.
* Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

# Merge Sort Algorithm

# Merge sort

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.

1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

The main idea behind merge sort is that, the short list takes less time to be sorted.

## Complexity

| **Complexity** | **Best case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Time Complexity | O(n log n) | O(n log n) | O(n log n) |
| Space Complexity |  |  | O(n) |

## Example :

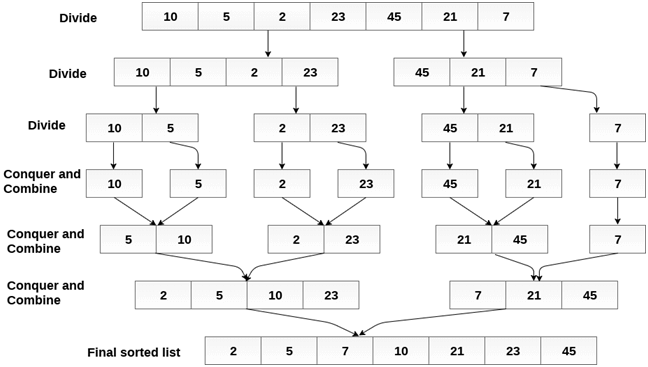
## 

## Learn more

## volume is gedempt

Consider the following array of 7 elements. Sort the array by using merge sort.

1. A = {10, 5, 2, 23, 45, 21, 7}



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

Merge Sort follows the rule of Divide and Conquer to sort a given set of numbers/elements, recursively, hence consuming less time.

Before moving forward with Merge Sort, check these topics out first:

* [Selection Sort](https://www.studytonight.com/data-structures/selection-sorting)
* [Insertion Sort](https://www.studytonight.com/data-structures/insertion-sorting)
* [Space Complexity of Algorithms](https://www.studytonight.com/data-structures/space-complexity-of-algorithms)
* [Time Complexity of Algorithms](https://www.studytonight.com/data-structures/time-complexity-of-algorithms)

In the last two tutorials, we learned about Selection Sort and Insertion Sort, both of which have a worst-case running time of O(n2). As the size of input grows, insertion and selection sort can take a long time to run.

Merge sort , on the other hand, runs in O(n\*log n) time in all the cases.

Before jumping on to, how merge sort works and it's implementation, first lets understand what is the rule of Divide and Conquer?

## Divide and Conquer

If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

Let's take an example, Divide and Rule.

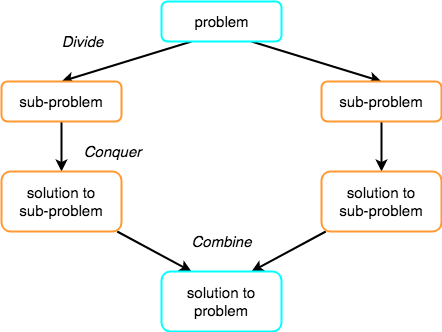
When Britishers came to India, they saw a country with different religions living in harmony, hard working but naive citizens, unity in diversity, and found it difficult to establish their empire. So, they adopted the policy of Divide and Rule. Where the population of India was collectively a one big problem for them, they divided the problem into smaller problems, by instigating rivalries between local kings, making them stand against each other, and this worked very well for them.

Well that was history, and a socio-political policy (Divide and Rule), but the idea here is, if we can somehow divide a problem into smaller sub-problems, it becomes easier to eventually solve the whole problem.

In Merge Sort, the given unsorted array with n elements, is divided into n subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. Divide the problem into multiple small problems.
2. Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. Combine the solutions of the subproblems to find the solution of the actual problem.



## How Merge Sort Works?

As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, sorting a given array.

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two subarrays with 3 elements each.

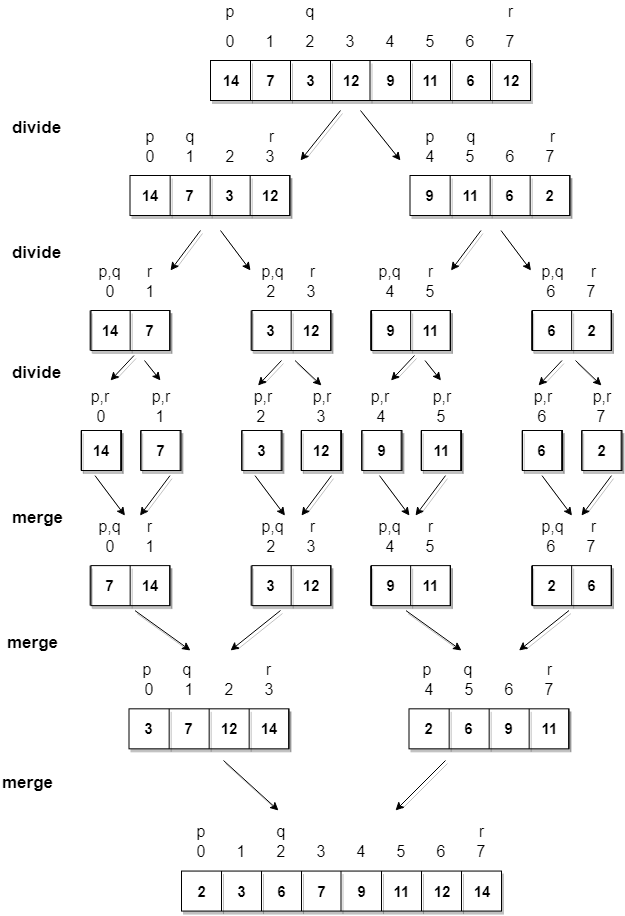
But breaking the orignal array into 2 smaller subarrays is not helping us in sorting the array.

So we will break these subarrays into even smaller subarrays, until we have multiple subarrays with single element in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into subarrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted subarrays, step by step to form one single sorted array.

Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}

Below, we have a pictorial representation of how merge sort will sort the given array.



In merge sort we follow the following steps:

1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
2. Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two subarrays, from p to q and from q + 1 to r index.
3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

## Implementing Merge Sort Algorithm

Below we have a C program implementing merge sort algorithm.

/\*

a[] is the array, p is starting index, that is 0,

and r is the last index of array.

\*/

#include <stdio.h>

// lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

// merge sort function

void mergeSort(int a[], int p, int r)

{

int q;

if(p < r)

{

q = (p + r) / 2;

mergeSort(a, p, q);

mergeSort(a, q+1, r);

merge(a, p, q, r);

}

}

// function to merge the subarrays

void merge(int a[], int p, int q, int r)

{

int b[5]; //same size of a[]

int i, j, k;

k = 0;

i = p;

j = q + 1;

while(i <= q && j <= r)

{

if(a[i] < a[j])

{

b[k++] = a[i++]; // same as b[k]=a[i]; k++; i++;

}

else

{

b[k++] = a[j++];

}

}

while(i <= q)

{

b[k++] = a[i++];

}

while(j <= r)

{

b[k++] = a[j++];

}

for(i=r; i >= p; i--)

{

a[i] = b[--k]; // copying back the sorted list to a[]

}

}

// function to print the array

void printArray(int a[], int size)

{

int i;

for (i=0; i < size; i++)

{

printf("%d ", a[i]);

}

printf("\n");

}

int main()

{

int arr[] = {32, 45, 67, 2, 7};

int len = sizeof(arr)/sizeof(arr[0]);

printf("Given array: \n");

printArray(arr, len);

// calling merge sort

mergeSort(arr, 0, len - 1);

printf("\nSorted array: \n");

printArray(arr, len);

return 0;

}

Given array:

32 45 67 2 7

Sorted array:

2 7 32 45 67

### Complexity Analysis of Merge Sort

Merge Sort is quite fast, and has a time complexity of O(n\*log n). It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

In this section we will understand why the running time for merge sort is O(n\*log n).

As we have already learned in [Binary Search](https://www.studytonight.com/data-structures/binary-search-algorithm) that whenever we divide a number into half in every step, it can be represented using a logarithmic function, which is log n and the number of steps can be represented by log n + 1(at most)

Also, we perform a single step operation to find out the middle of any subarray, i.e. O(1).

And to merge the subarrays, made by dividing the original array of n elements, a running time of O(n) will be required.

Hence the total time for mergeSort function will become n(log n + 1), which gives us a time complexity of O(n\*log n).

Worst Case Time Complexity [ Big-O ]: O(n\*log n)

Best Case Time Complexity [Big-omega]: O(n\*log n)

Average Time Complexity [Big-theta]: O(n\*log n)

Space Complexity: O(n)

* Time complexity of Merge Sort is O(n\*Log n) in all the 3 cases (worst, average and best) as merge sort always divides the array in two halves and takes linear time to merge two halves.
* It requires equal amount of additional space as the unsorted array. Hence its not at all recommended for searching large unsorted arrays.
* It is the best Sorting technique used for sorting [Linked Lists](https://www.studytonight.com/data-structures/introduction-to-linked-list).

# Heap Sort Algorithm

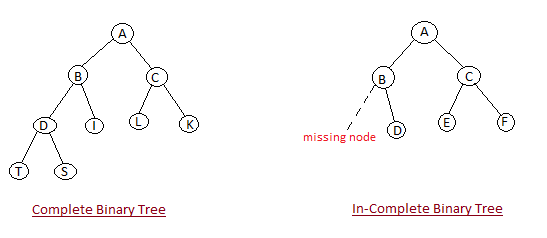
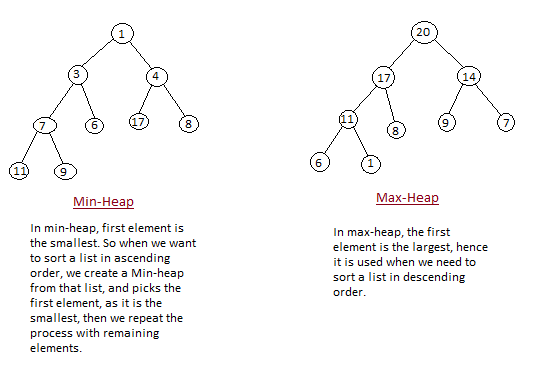
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a Heap data structure from the given array and then utilizing the Heap to sort the array.

You must be wondering, how converting an array of numbers into a heap data structure will help in sorting the array. To understand this, let's start by understanding what is a Heap.

NOTE: If you are not familiar with Sorting in data structure, you should first learn [what is sorting](https://www.studytonight.com/data-structures/introduction-to-sorting) to know about the basics of sorting.

## What is a Heap ?

Heap is a special tree-based data structure, that satisfies the following special heap properties:

1. Shape Property: Heap data structure is always a Complete [Binary Tree](https://www.studytonight.com/data-structures/introduction-to-binary-trees), which means all levels of the tree are fully filled.  
   
2. Heap Property: All nodes are either greater than or equal to or less than or equal to each of its children. If the parent nodes are greater than their child nodes, heap is called a Max-Heap, and if the parent nodes are smaller than their child nodes, heap is called Min-Heap.  
   

## How Heap Sort Works?

Heap sort algorithm is divided into two basic parts:

* Creating a Heap of the unsorted list/array.
* Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

In the below algorithm, initially heapsort() function is called, which calls heapify() to build the heap.

## Implementing Heap Sort Algorithm

Below we have a simple C++ program implementing the Heap sort algorithm.

/\* Below program is written in C++ language \*/

#include <iostream>

using namespace std;

void heapify(int arr[], int n, int i)

{

int largest = i;

int l = 2\*i + 1;

int r = 2\*i + 2;

// if left child is larger than root

if (l < n && arr[l] > arr[largest])

largest = l;

// if right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// if largest is not root

if (largest != i)

{

swap(arr[i], arr[largest]);

// recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n)

{

// build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// one by one extract an element from heap

for (int i=n-1; i>=0; i--)

{

// move current root to end

swap(arr[0], arr[i]);

// call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

/\* function to print array of size n \*/

void printArray(int arr[], int n)

{

for (int i = 0; i < n; i++)

{

cout << arr[i] << " ";

}

cout << "\n";

}

int main()

{

int arr[] = {121, 10, 130, 57, 36, 17};

int n = sizeof(arr)/sizeof(arr[0]);

heapSort(arr, n);

cout << "Sorted array is \n";

printArray(arr, n);

}

### Complexity Analysis of Heap Sort

Worst Case Time Complexity: O(n\*log n)

Best Case Time Complexity: O(n\*log n)

Average Time Complexity: O(n\*log n)

Space Complexity : O(1)

* Heap sort is not a Stable sort, and requires a constant space for sorting a list.
* Heap Sort is very fast and is widely used for sorting.

# Data Structure - Sorting Techniques

Advertisements

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm)

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

* Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
* Dictionary − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

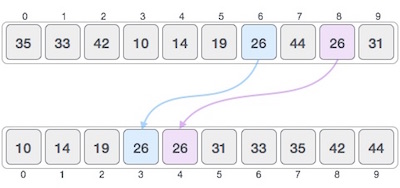
## In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting.

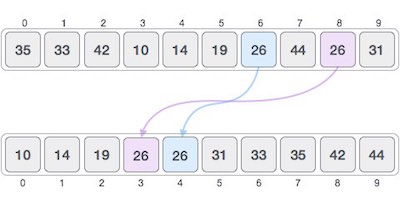
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.

## Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

## Adaptive and Non-Adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

## Important Terms

Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them −

### Increasing Order

A sequence of values is said to be in increasing order, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

### Decreasing Order

A sequence of values is said to be in decreasing order, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

### Non-Increasing Order

A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.

### Non-Decreasing Order

A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.