### Non-comparison Sorting Algorithms

* Non-comparison sorting methods are not based on the comparison of two elements on less, greater or equal.

[Counting Sort](https://www.happycoders.eu/algorithms/counting-sort/) – as the name suggests – counts elements. For example, to sort an array of numbers from 1 to 10,

Pass 1: Count how often the 1 occurs, how often the 2 occurs, etc. up to the 10.

Pass 2: Write down the 1 as often as it occurs, starting from the left, then the 2 as often as it occurs, and so on until the 10.

Application :

* This technique is usually used only for small number types like byte, char, or short, or if the range of numbers to be sorted is known (e.g., ints between 0 and 150).

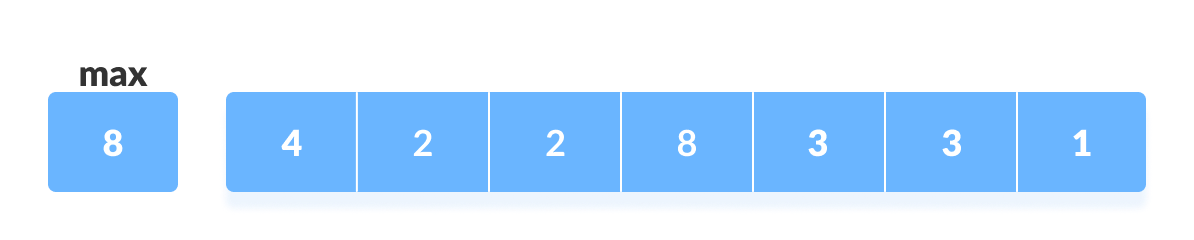
Not Inplace:

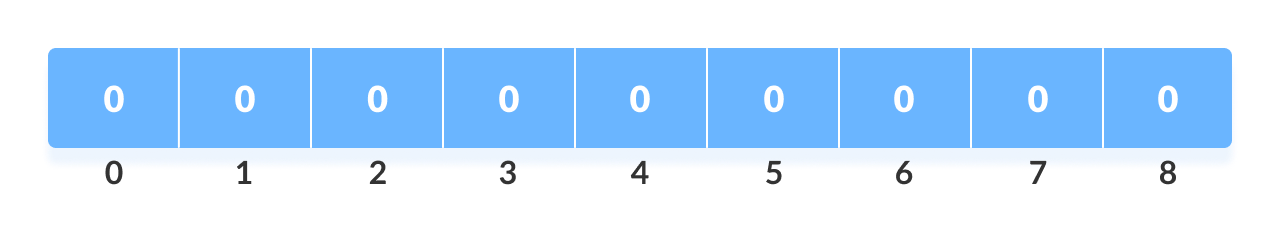
* The reason for this is that, to count the elements, we need an additional array corresponding to the size of the number range.

# Counting Sort Algorithm (Self Study)

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

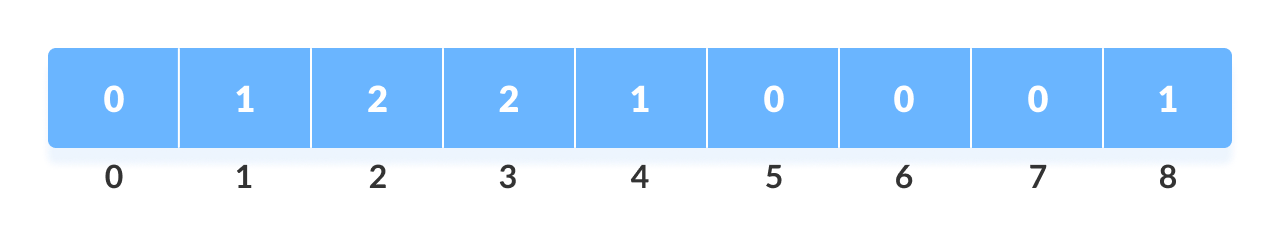
## How Counting Sort Works?

1. Find out the maximum element (let it be max) from the given array.
2. Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.

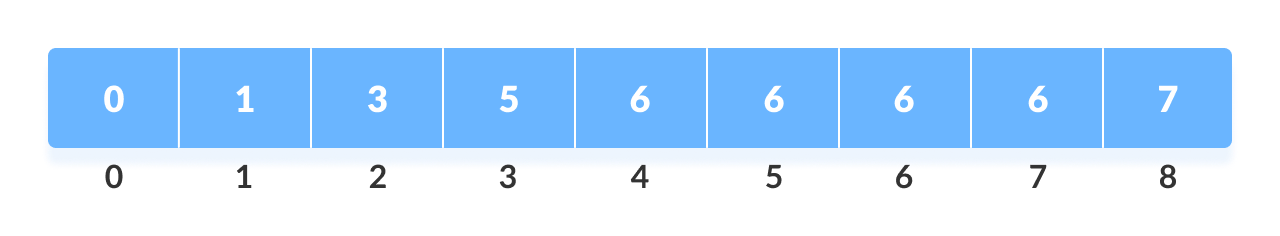


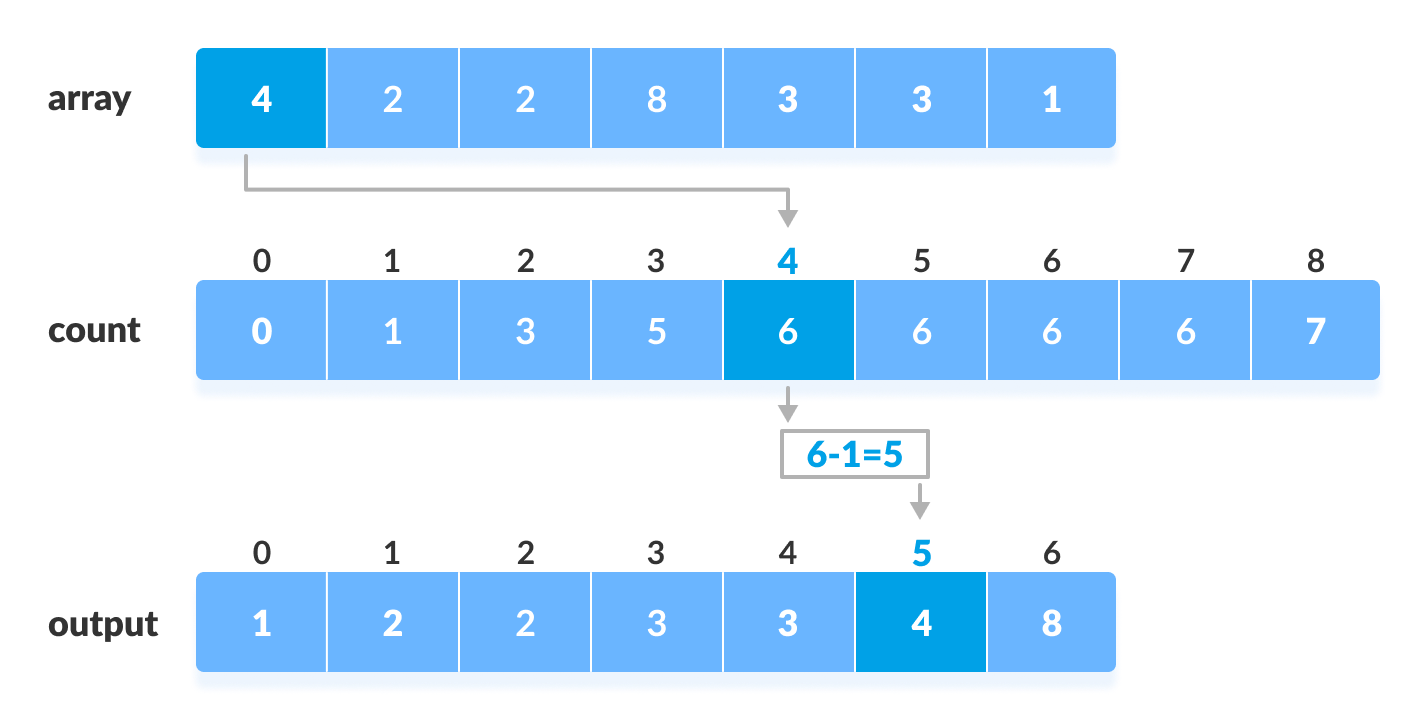
1. Store the count of each element at their respective index in count array

* For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position.



1. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.



1. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.Counting sort
2. After placing each element at its correct position, decrease its count by one.

## Counting Sort Algorithm (Reference…. Can convert into proper structured algorithm as discussed in class)

countingSort(array, size)

set max = find largest element in array

initialize count array with all zeros

for j=0 to size

find the total count of each unique element and

store the count at jth index in count array

for I = 1 to max

find the cumulative sum and store it in count array itself

for j = size down to 1

restore the elements to array

decrease count of each element restored by 1

## Complexity

**Time Complexities:**

There are mainly four main loops. (Finding the greatest value can be done outside the function.)

|  |  |
| --- | --- |
| for-loop | time of counting |
| 1st | O(max) |
| 2nd | O(size) |
| 3rd | O(max) |
| 4th | O(size) |

Overall complexity = O(max)+O(size)+O(max)+O(size) = O(max+size)

* **Worst Case Complexity:** O(n+k)
* **Best Case Complexity:** O(n+k)
* **Average Case Complexity:** O(n+k)

Note : There is no comparison between any elements, so it is better than comparison based sorting techniques. But, it is bad if the integers are very large because the array of that size should be made.

**Space Complexity:**

The space complexity of Counting Sort is O(max). Larger the range of elements, larger is the space complexity.

**Radix Sort**

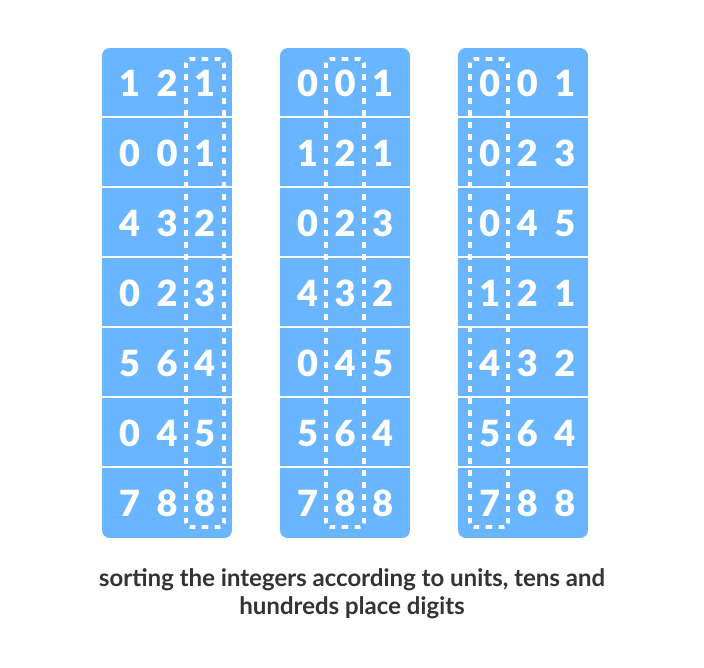
Main logic: sorts the elements by first grouping the individual digits of the same **place value**.

Then, sort the elements according to their increasing/decreasing order.

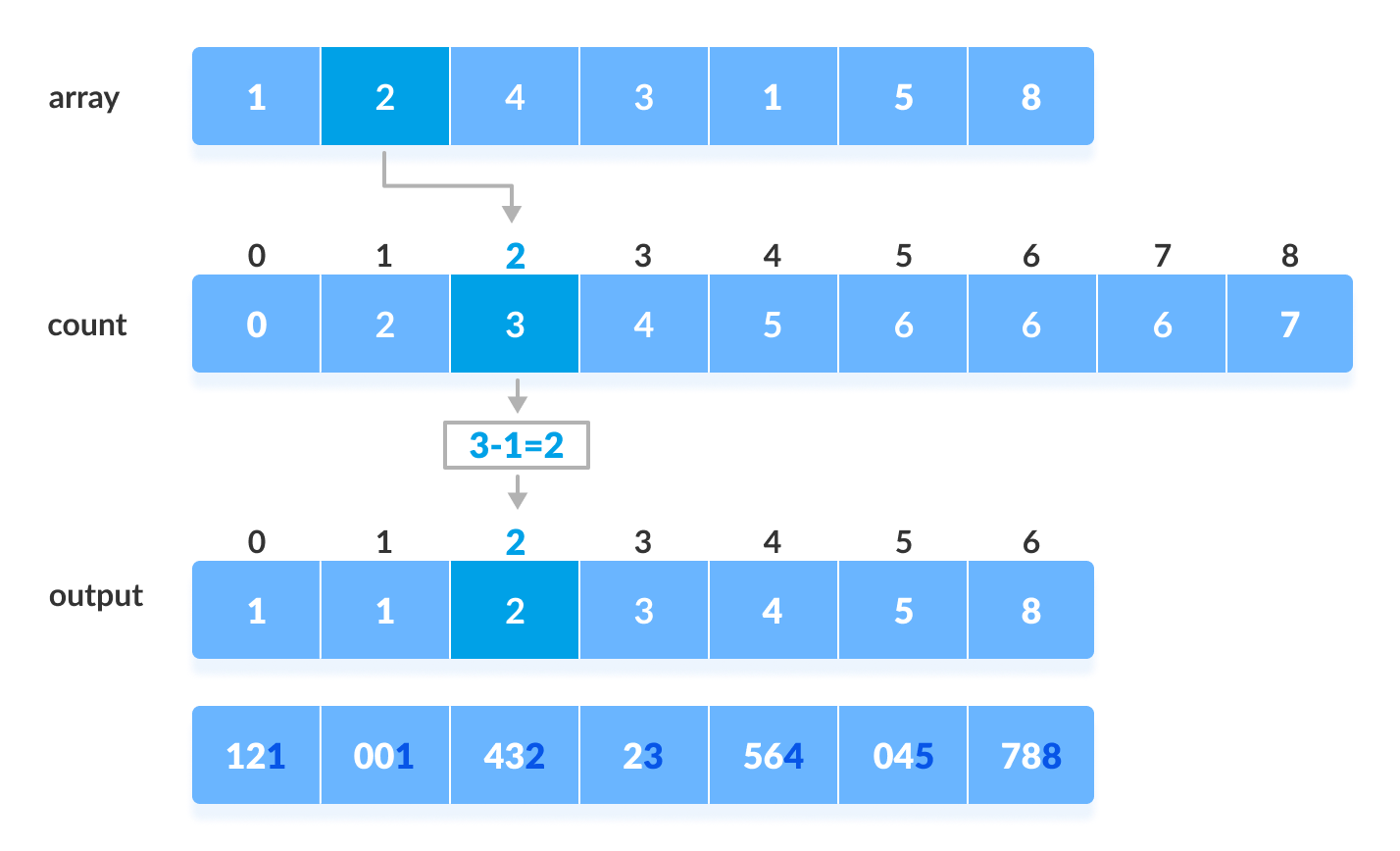
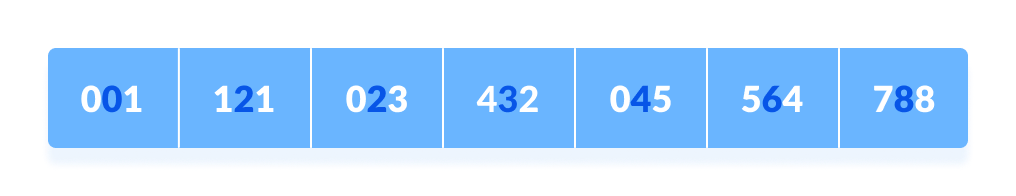
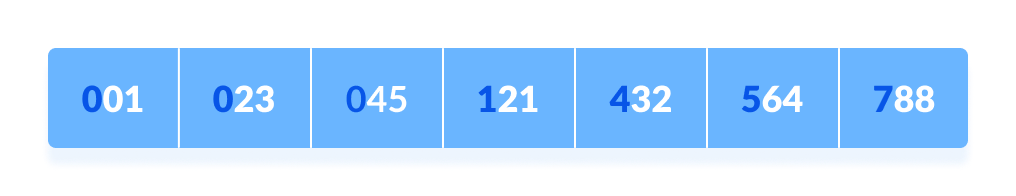
E.g. For array of n elements.

* First, we will sort elements based on the value of the unit place Sorting can be done using Count Sort).
* Then, we will sort elements based on the value of the tenth place.
* This process goes on until the last significant place.

Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.



**Working of Radix Sort:**

1. Find the largest element in the array, i.e. max. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements.  
     
   In this array [121, 432, 564, 23, 1, 45, 788], we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).
2. Now, go through each significant place one by one.  
     
   Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this.  
     
   Sort the elements based on the unit place digits (X=0).
3. Now, sort the elements based on digits at tens place.
4. Finally, sort the elements based on the digits at hundreds place.

## Radix Sort Algorithm (Can be converted into stepwise algorithm as discussed in the class)

radixSort(array)

d <- maximum number of digits in the largest element

create d buckets of size 0-9

for i <- 0 to d

sort the elements according to ith place digits using countingSort

countingSort(array, d)

max <- find largest element among dth place elements

initialize count array with all zeros

for j <- 0 to size

find the total count of each unique digit in dth place of elements and

store the count at jth index in count array

for i <- 1 to max

find the cumulative sum and store it in count array itself

for j <- size down to 1

restore the elements to array

decrease count of each element restored by 1

Complexity

Since radix sort is a non-comparative algorithm, it has advantages over comparative sorting algorithms.

For the radix sort that uses counting sort as an intermediate stable sort, the time complexity is (d \* complexity of Counting Sort)  = O(d(n+k)).

## Radix Sort Applications

Radix sort is implemented in

* places where there are numbers in large ranges.

Summary of sorting:

| Algorithm | Time best case | Time avg. case | Time worst case | Space | Stable |
| --- | --- | --- | --- | --- | --- |
| [Insertion Sort](https://www.happycoders.eu/algorithms/insertion-sort/) | O(n) | O(n²) | O(n²) | O(1) | Yes |
| [Selection Sort](https://www.happycoders.eu/algorithms/selection-sort/) | O(n²) | O(n²) | O(n²) | O(1) | No |
| [Bubble Sort](https://www.happycoders.eu/algorithms/bubble-sort/) | O(n) | O(n²) | O(n²) | O(1) | Yes |
| [Quicksort](https://www.happycoders.eu/algorithms/quicksort/) | O(n log n) | O(n log n) | O(n²) | O(log n) | No |
| [Merge Sort](https://www.happycoders.eu/algorithms/merge-sort/) | O(n log n) | O(n log n) | O(n log n) | O(n) | Yes |
| [Heapsort](https://www.happycoders.eu/algorithms/heapsort/) | O(n log n) | O(n log n) | O(n log n) | O(1) | No |
| [Counting Sort](https://www.happycoders.eu/algorithms/counting-sort/) | O(n + k) | O(n + k) | O(n + k) | O(n + k) | Yes |

(The variable k in Counting Sort stands for keys – the number of possible values).

Some important comments:

1. Based on Number of Swaps or Inversion This is the number of times the algorithm swaps elements to sort the input.  Selection Sort  requires the minimum number of swaps.
2. Based on Number of Comparisons This is the number of times the algorithm compares elements to sort the input. Using [Big-O notation](https://guide.freecodecamp.org/computer-science/notation/big-o-notation/), the sorting algorithm examples listed above require at least  O(nlogn) comparisons in the best case and  O(n^2)  comparisons in the worst case for most of the outputs.
3. Based on Recursion or Non-Recursion Some sorting algorithms, such as  Quick Sort , use recursive techniques to sort the input. Other sorting algorithms, such as  Selection Sort  or  Insertion Sort , use non-recursive techniques. Finally, some sorting algorithm, such as  Merge Sort , make use of both recursive as well as non-recursive techniques to sort the input.
4. Based on Stability Sorting algorithms are said to be  stable  if the algorithm maintains the relative order of elements with equal keys. In other words, two equivalent elements remain in the same order in the sorted output as they were in the input.
5. Insertion sort ,  Merge Sort , and  Bubble Sort  are stable
6. Heap Sort  and  Quick Sort  are not stable
7. Based on Extra Space Requirement Sorting algorithms are said to be  in place  if they require a constant  O(1)  extra space for sorting.
8. Insertion sort  and  Quick-sort  are  in place  sort as we move the elements about the pivot and do not actually use a separate array which is NOT the case in merge sort where the size of the input must be allocated beforehand to store the output during the sort.
9. Merge Sort  is an example of  out place  sort as it require extra memory space for its operations.