

Sorting Techniques

Introduction to Sorting Techniques

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

Why Sorting Algorithms are Important

Sorting algorithms are essential in Computer Science as they simplify complex problems and improve efficiency. They are widely used in searching, databases, divide and conquer strategies, and data structures.

Key Applications:

- Organizing large datasets for easier handling and printing
- Enabling quick access to the k-th smallest or largest elements
- Making binary search possible for fast lookups in sorted data
- Solving advanced problems in both software and algorithm design

Sorting Basics

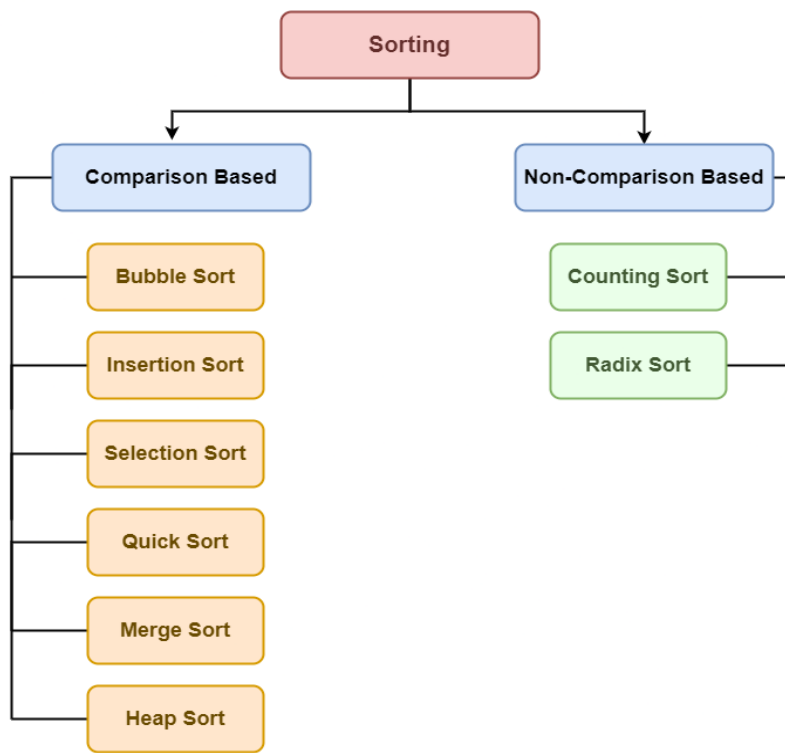
- **In-place Sorting**: An in-place sorting algorithm uses **constant space** for producing the output (modifies the given array only. Examples: Selection Sort, Bubble Sort, Insertion Sort and Heap Sort.
- **Internal Sorting**: Internal Sorting is when all the data is placed in the **main memory** or **internal memory**. In internal sorting, the problem cannot take input beyond allocated memory size.
- **External Sorting** : External Sorting is when all the data that needs to be sorted need not to be placed in memory at a time, the sorting is called external sorting. External Sorting is used for the massive amount of data. For example Merge sort can be used in external sorting as the whole array does not have to be present all the time in memory,
- **Stable sorting**: When two same items appear in the **same order** in sorted data as in the original array called stable sort. Examples: Merge Sort, Insertion Sort, Bubble Sort.
- **Hybrid Sorting**: A sorting algorithm is called Hybrid if it uses more than one standard sorting algorithms to sort the array. The idea is to take advantages of multiple sorting algorithms. For Example IntroSort uses Insertions sort and Quick Sort.

Sorting Techniques

Types of Sorting Techniques

There are various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

1. **Comparison-based:** We compare the elements in a comparison-based sorting algorithm)
2. **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm)



Sorting Algorithms (Comparison Based)

1- Quick Sort

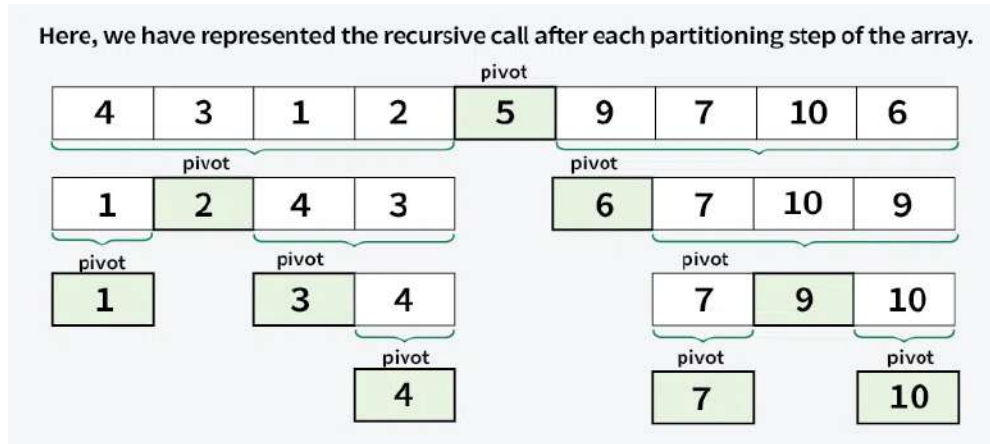
Quick Sort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

There are mainly three steps in the algorithm:

1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).

Sorting Techniques

2. Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right.
3. Recursively Call: Recursively apply the same process to the two partitioned sub-arrays.
4. Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.



Choice of Pivot

There are many different choices for picking pivots.

- Always pick the first (or last) element as a pivot. The below implementation picks the last element as pivot. The problem with this approach is it ends up in the worst case when array is already sorted.
- Pick a random element as a pivot. This is a preferred approach because it does not have a pattern for which the worst case happens.
- Pick the median element as pivot. This is an ideal approach in terms of time complexity as we can find median in linear time and the partition function will always divide the input array into two halves. But it takes more time on average as median finding has high constants.

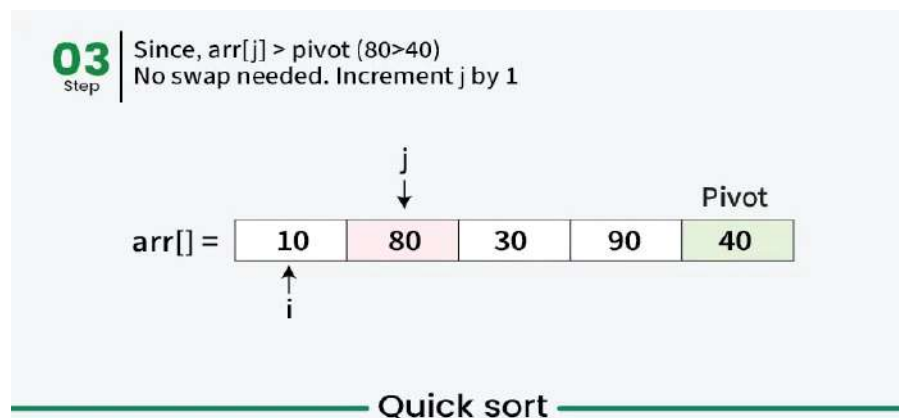
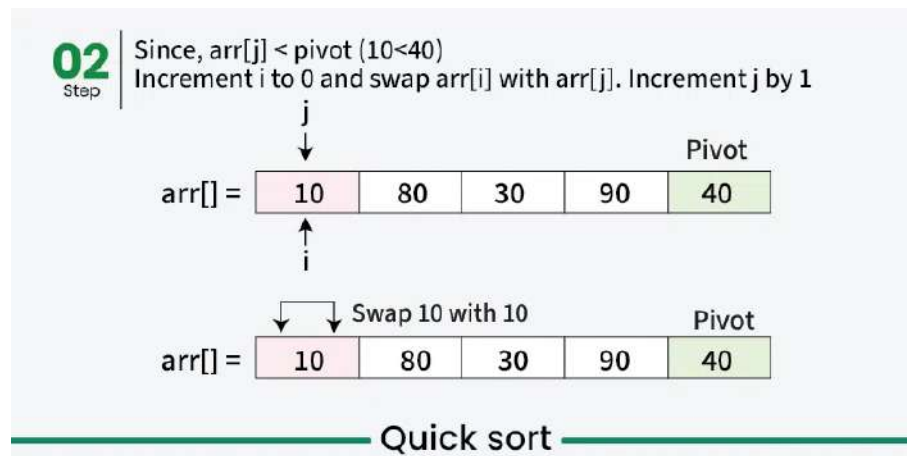
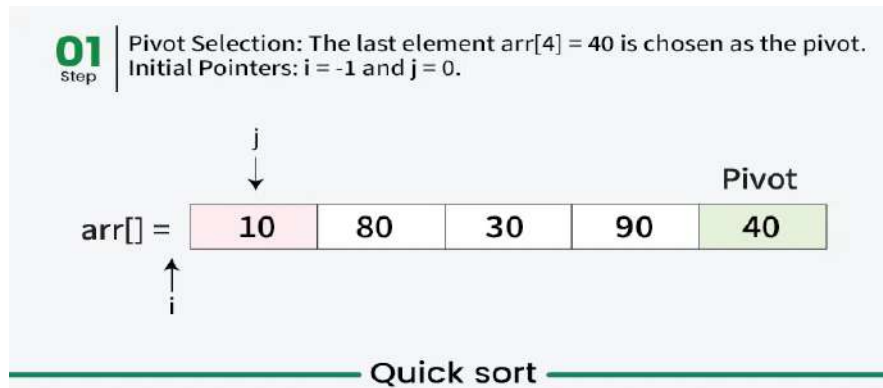
Partition Algorithm

The key process in **quickSort** is a **partition()**. There are three common algorithms to partition. All these algorithms have $O(n)$ time complexity.

1. Naive Partition: Here we create copy of the array. First put all smaller elements and then all greater. Finally we copy the temporary array back to original array. This requires $O(n)$ extra space.

Sorting Techniques

2. **Lomuto Partition:** We have used this partition in this article. This is a simple algorithm, we keep track of index of smaller elements and keep swapping. We have used it here in this article because of its simplicity.
3. **Hoare's Partition:** This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned. Please refer [Hoare's vs Lomuto](#) for details.



Sorting Techniques

04
Step

Since, $\text{arr}[j] < \text{pivot}$ ($30 < 40$)
Increment i by 1 and swap $\text{arr}[i]$ with $\text{arr}[j]$. Increment j by 1

arr[] =

10	80	30	90	40
----	----	----	----	----

Swap 80 with 30

Quick sort

05
Step

Since, $\text{arr}[j] > \text{pivot}$ ($90 > 40$)
No swap needed. Increment j by 1

arr[] =

10	30	80	90	40
----	----	----	----	----

Quick sort

06
Step

Since traversal of j has ended. Now move pivot to its correct position, Swap $\text{arr}[i + 1] = \text{arr}[2]$ with $\text{arr}[4] = 40$.

arr[] =

10	30	80	90	40
----	----	----	----	----

Swap 80 and 40

Partition Index(pi) = 2 Pivot

arr[] =

10	30	40	90	80
----	----	----	----	----

Quick sort

Complexity Analysis of Quick Sort

Time Complexity:

- **Best Case:** ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.

Sorting Techniques

- **Worst Case:** ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Auxiliary Space:

- **Worst-case scenario:** $O(n)$ due to unbalanced partitioning leading to a skewed recursion tree requiring a call stack of size $O(n)$.
- **Best-case scenario:** $O(\log n)$ as a result of balanced partitioning leading to a balanced recursion tree with a call stack of size $O(\log n)$.

Advantages of Quick Sort

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.
- It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.
- Fastest general purpose algorithm for large data when stability is not required.
- It is [tail recursive](#) and hence all the [tail call optimization](#) can be done.

Disadvantages of Quick Sort

- It has a worst-case time complexity of $O(n^2)$, which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Applications of Quick Sort

- Sorting large datasets efficiently in memory.
- Used in library sort functions (like C++ `std::sort` and Java `Arrays.sort` for primitives).
- Arranging records in databases for faster searching.
- Preprocessing step in algorithms requiring sorted input (e.g., binary search, two-pointer techniques).

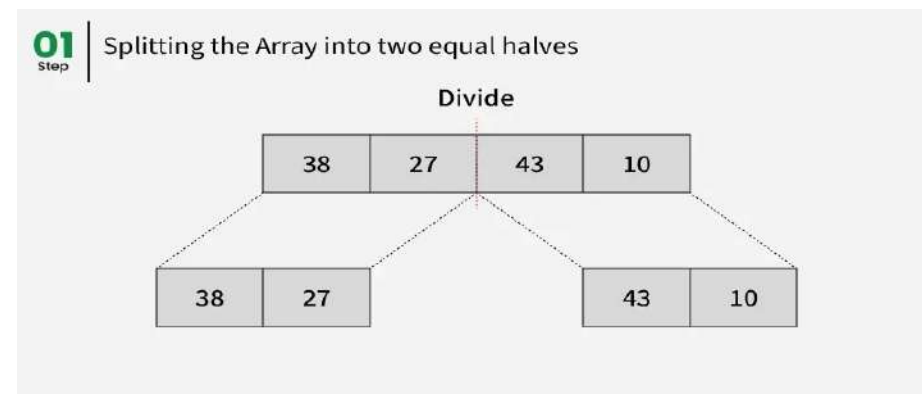
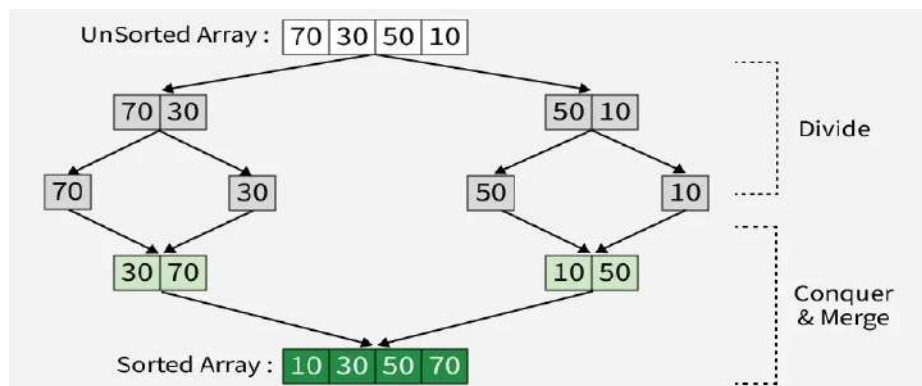
Sorting Techniques

- Finding the kth smallest/largest element using Quickselect (a variant of quicksort).
- Sorting arrays of objects based on multiple keys (custom comparators).
- Data compression algorithms (like Huffman coding preprocessing).
- Graphics and computational geometry (e.g., convex hull algorithms).

2- Merge Sort

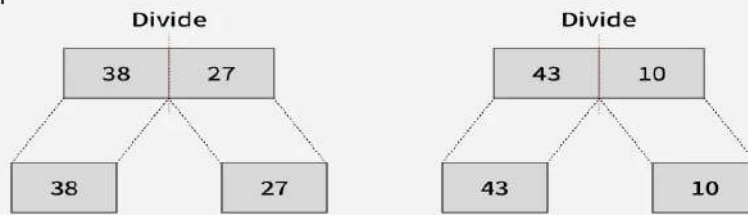
Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the Divide and Conquer approach. It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array.

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.



Sorting Techniques

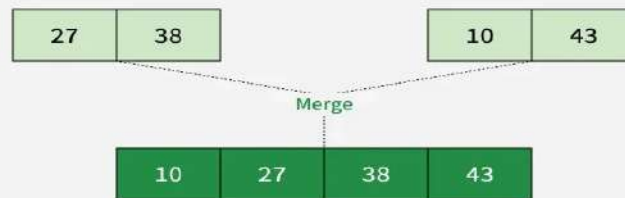
02 Step | Splitting the subarrays into two halves



03 Step | Merging unit length cells into sorted subarrays



04 Step | Merging sorted subarrays into the sorted array



Recurrence Relation of Merge Sort

The recurrence relation of merge sort is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- $T(n)$ Represents the total time taken by the algorithm to sort an array of size n .
- $2T(n/2)$ represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has $n/2$ elements, we have two recursive calls with input size as $(n/2)$.
- $O(n)$ represents the time taken to merge the two sorted halves

Sorting Techniques

Complexity Analysis of Merge Sort

Time Complexity:

- **Best Case:** $O(n \log n)$, When the array is already sorted or nearly sorted.
- **Average Case:** $O(n \log n)$, When the array is randomly ordered.
- **Worst Case:** $O(n \log n)$, When the array is sorted in reverse order.

Auxiliary Space: $O(n)$, Additional space is required for the temporary array used during merging.

Applications of Merge Sort:

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- Used to solve problems like Inversion counting, Count Smaller on Right & Surpasser Count
- Merge Sort and its variations are used in library methods of programming languages.
- It is a preferred algorithm for sorting Linked lists.
- It can be easily parallelized as we can independently sort subarrays and then merge.
- The merge function of merge sort to efficiently solve the problems like union and intersection of two sorted arrays.

Advantages and Disadvantages of Merge Sort

Advantages

- **Stability :** Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of $O(n \log n)$, which means it performs well even on large datasets.
- **Simple to implement:** The divide-and-conquer approach is straightforward.
- **Naturally Parallel :** We independently merge subarrays that makes it suitable for parallel processing.

Sorting Techniques

Disadvantages

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- Merge Sort is **Slower than QuickSort in general** as QuickSort is more cache friendly because it works in-place.

3- Heap Sort

Heap Sort is a comparison-based sorting algorithm based on the **Binary Heap** data structure.

- It is an optimized version of selection sort.
- The algorithm repeatedly finds the maximum (or minimum) element and swaps it with the last (or first) element.
- Using a binary heap allows efficient access to the max (or min) element in $O(\log n)$ time instead of $O(n)$.
- The process is repeated for the remaining elements until the array is sorted.
- Overall, Heap Sort achieves a time complexity of $O(n \log n)$.

Heap Sort Algorithm :

First convert the array into a max heap using heapify, Please note that this happens in-place.

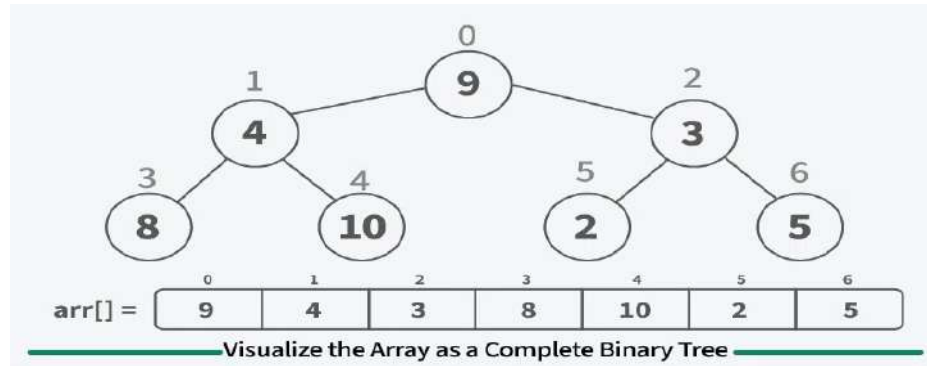
The array elements are re-arranged to follow heap properties. Then one by one delete the root node of the Max-heap and replace it with the last node and heapify. Repeat this process while size of heap is greater than 1.

Detailed Working of Heap Sort

Step 1: Treat the Array as a Complete Binary Tree

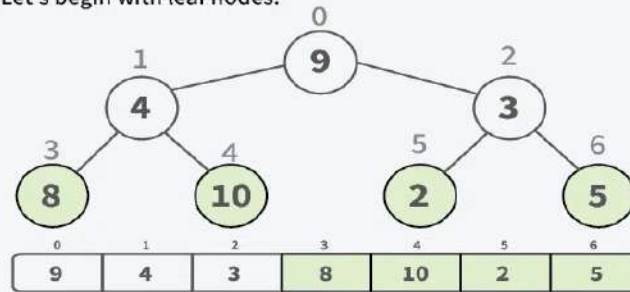
We first need to visualize the array as a complete binary tree. For an array of size n , the root is at index 0, the left child of an element at index i is at $2i + 1$, and the right child is at $2i + 2$.

Sorting Techniques



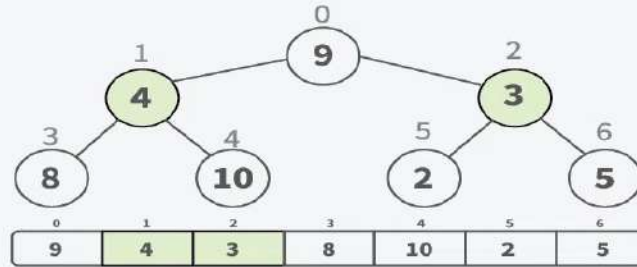
01
Step

Compare each node with its children, ensuring parent nodes are larger. This causes smaller nodes to bubble down and larger nodes to rise to the top. Let's begin with leaf nodes.



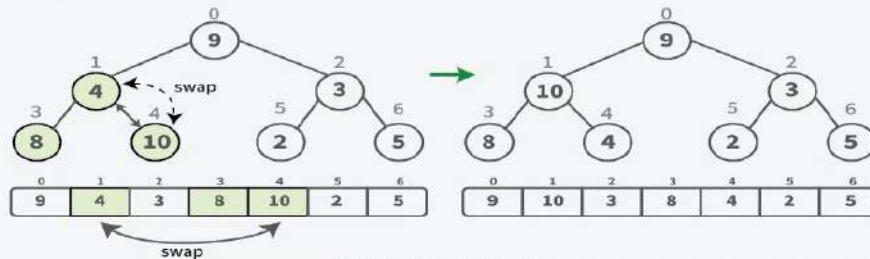
02
Step

Let's look in the next upper level (node 4 and 3)



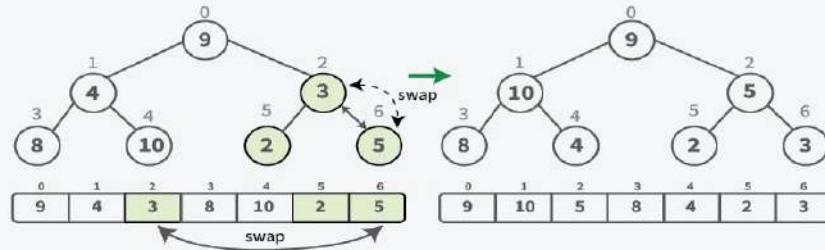
03
Step

Node 4 is smaller than its child node (10), so swap it with the larger child to maintain the property that the parent should be larger than its children.



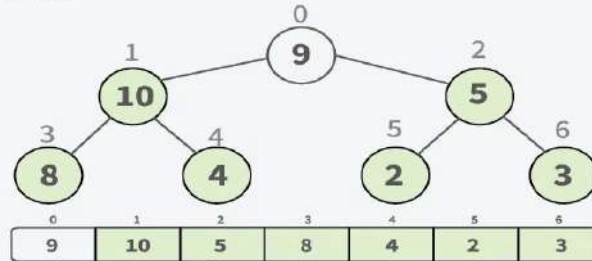
Sorting Techniques

04 Step Check for the next node (3) in current level. Since, it has a larger child, so swap it to ensure the parent has a larger value.



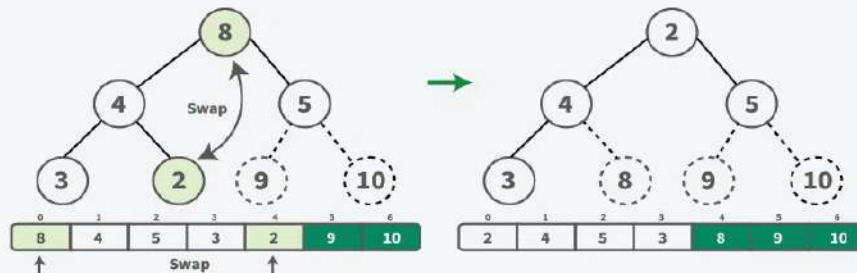
Heapify Binary Tree

05 Step After the completion of current level, we have now two smaller valid max heap



Heapify Binary Tree

06 Step Now, we have a max heap. Swap the maximum element (8) with the last element (2) in the unsorted array, then decrease the heap size by one (ignoring the third last element as it's now sorted).



Remove from Max Heap

Time Complexity: $O(n \log n)$

Auxiliary Space: $O(\log n)$, due to the recursive call stack. However, auxiliary space can be $O(1)$ for iterative implementation.

Important points about Heap Sort

- An in-place algorithm.
- Its typical implementation is not stable but can be made stable
- Typically 2-3 times slower than well-implemented [QuickSort](#). The reason for slowness is a lack of locality of reference.

Sorting Techniques

Advantages of Heap Sort

- **Efficient Time Complexity:** Heap Sort has a guaranteed time complexity of $O(n \log n)$ in all cases, making it suitable for large datasets. The $\log n$ factor comes from the height of the binary heap, ensuring consistent performance.
- **Minimal Memory Usage:** Heap Sort can work in-place with minimal extra memory. Using an iterative `heapify()` avoids additional stack space, so apart from storing the array itself, no extra memory is required.
- **Simplicity:** Heap Sort is relatively easy to understand and implement compared to other efficient sorting algorithms, as it relies on the straightforward binary heap structure without advanced concepts like recursion (if iterative `heapify` is used).

Disadvantages of Heap Sort

- **Costly:** Heap sort is costly as the constants are higher compared to merge sort even if the time complexity is $O(n \log n)$ for both.
- **Unstable:** Heap sort is unstable. It might rearrange the relative order.
- **Inefficient:** Heap Sort is not very efficient because of the high constants in the time complexity.

4- Insertion Sort Algorithm

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- Start with the second element as the first element is assumed to be sorted.
- Compare the second element with the first if the second is smaller then swap them.
- Move to the third element, compare it with the first two, and put it in its correct position
- Repeat until the entire array is sorted.

Sorting Techniques

1st Pass

Key = 11

Compare key with 12. As 12 is in the wrong order, shift 12 to the right to make space for the key.



No element left to compare so insert the key at the beginning.



Insertion Sort

2nd Pass

Key = 13

Compare key with 12, 12 is in the correct order, so no changes take place.



Insertion Sort

3rd Pass

Key = 5

Compare key with all the elements in the sorted subarray starting with 13, if the element is in the wrong order, shift that element to the right.



No element left to compare, so insert key at the beginning.

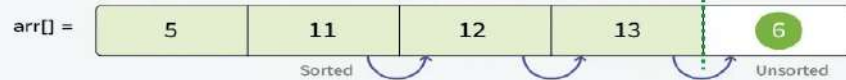


Insertion Sort

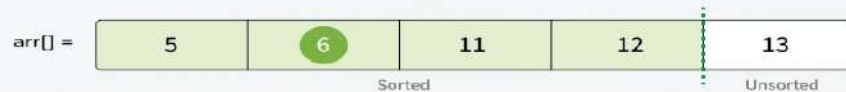
4th Pass

Key = 6

Compare key with all the elements in the sorted subarray starting with 13, if the element is in the wrong order, shift that element to the right.

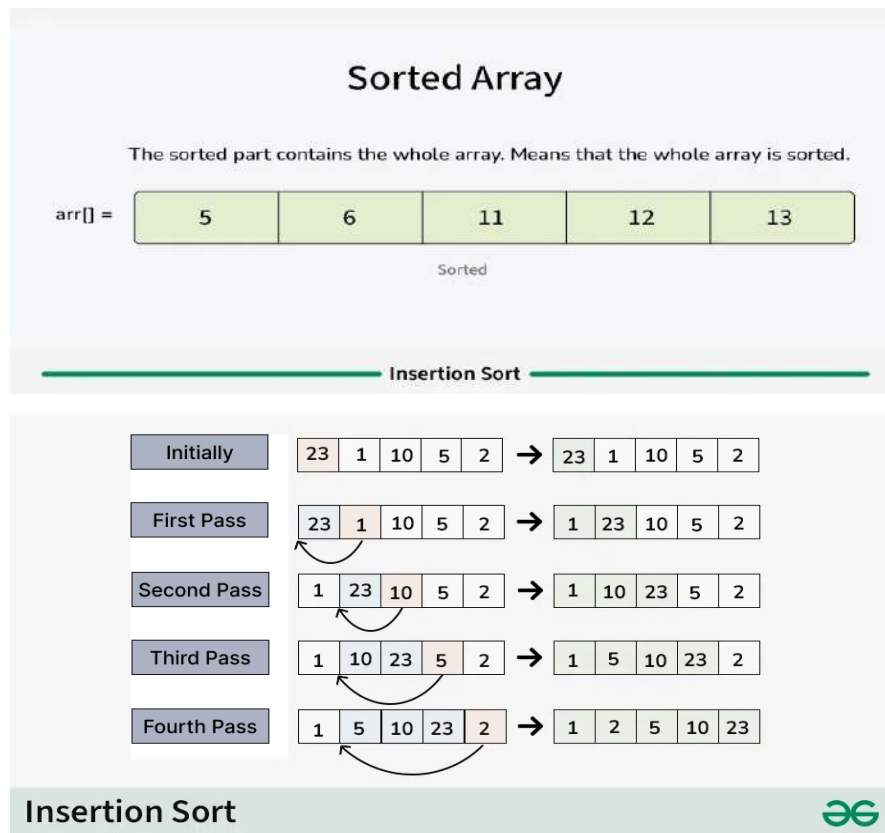


5 is in the correct order, so shifting stops. Insert key after 5.



Insertion Sort

Sorting Techniques



Complexity Analysis of Insertion Sort

Time Complexity

- **Best case: $O(n)$** , If the list is already sorted, where n is the number of elements in the list.
- **Average case: $O(n^2)$** , If the list is randomly ordered
- **Worst case: $O(n^2)$** , If the list is in reverse order

Space Complexity

- **Auxiliary Space:** $O(1)$, Insertion sort requires $O(1)$ additional space, making it a space-efficient sorting algorithm.

Please refer Complexity Analysis of Insertion Sort for details.

Advantages and Disadvantages of Insertion Sort

Advantages

- Simple and easy to implement.
- **Stable** sorting algorithm.

Sorting Techniques

- Efficient for small lists and nearly sorted lists.
- Space-efficient as it is an in-place algorithm.
- Adoptive. the [number of inversions](#) is directly proportional to number of swaps. For example, no swapping happens for a sorted array and it takes $O(n)$ time only.

Disadvantages

- Inefficient for large lists.
- Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

Applications of Insertion Sort

Insertion sort is commonly used in situations where:

- The list is small or nearly sorted.
- Simplicity and stability are important.
- Used as a subroutine in [Bucket Sort](#)
- Can be useful when array is already almost sorted (very few [inversions](#))
- Since Insertion sort is suitable for small sized arrays, it is used in [Hybrid Sorting algorithms](#) along with other efficient algorithms like Quick Sort and Merge Sort. When the subarray size becomes small, we switch to insertion sort in these recursive algorithms. For example [IntroSort](#) and [TimSort](#) use insertions sort.

5- Selection Sort

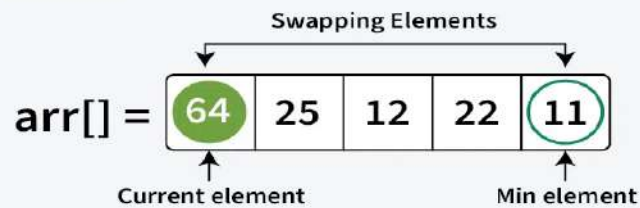
Selection Sort is a comparison-based sorting algorithm. It sorts by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element.

1. Find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then find the smallest among remaining elements (or second smallest) and swap it with the second element.
3. We keep doing this until we get all elements moved to correct position.

Sorting Techniques

01
Step

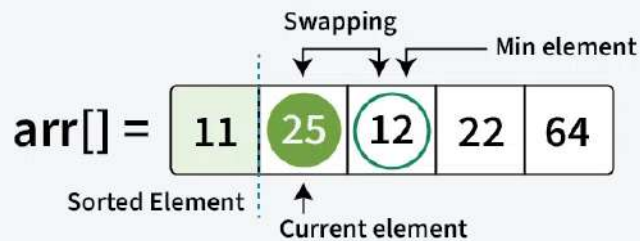
Start from the first element at index 0, find the smallest element in the rest of the array which is unsorted, and swap (11) with current element(64).



Selection Sort Algorithm

02
Step

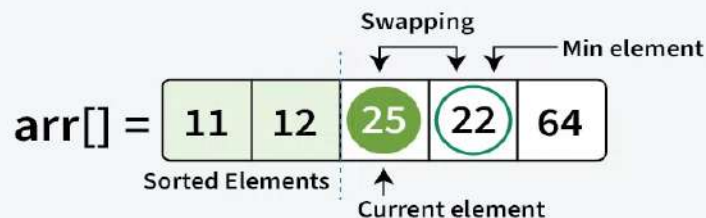
Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).



Selection Sort Algorithm

03
Step

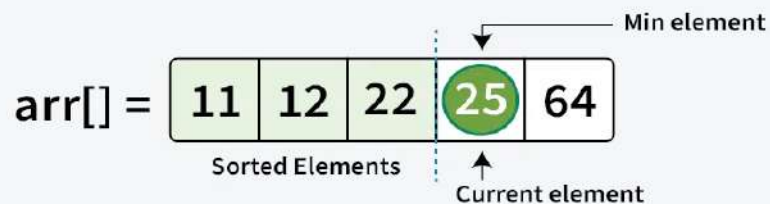
Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).



Selection Sort Algorithm

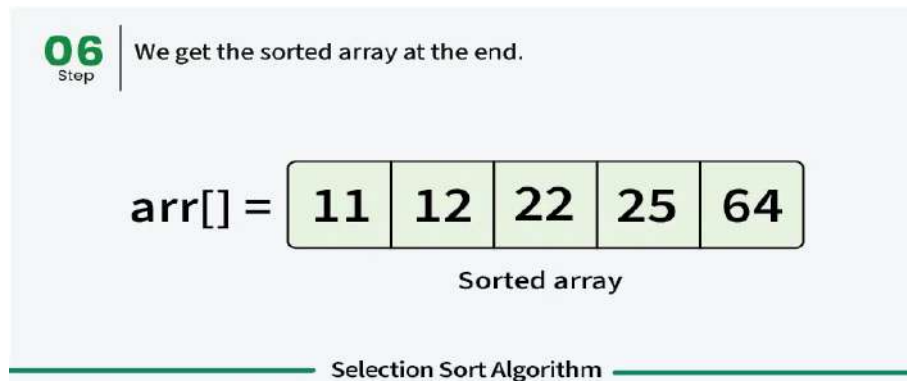
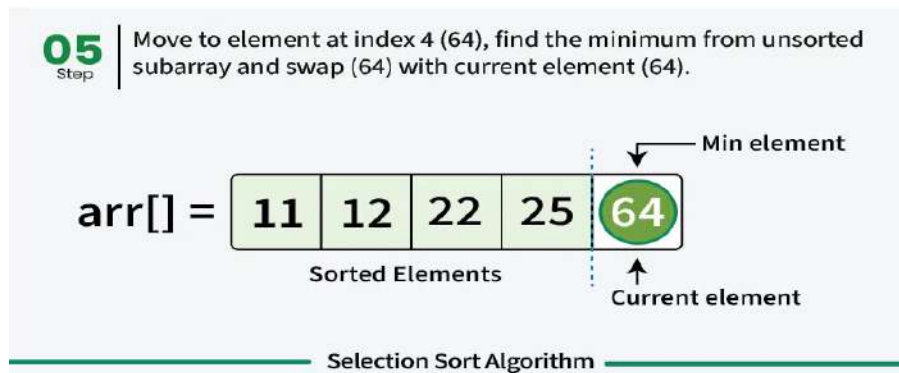
04
Step

Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).



Selection Sort Algorithm

Sorting Techniques



Complexity Analysis of Selection Sort

Time Complexity: $O(n^2)$, as there are two nested loops:

- One loop to select an element of Array one by one = $O(n)$
- Another loop to compare that element with every other Array element = $O(n)$
- Therefore overall complexity = $O(n) * O(n) = O(n*n) = O(n^2)$

Auxiliary Space: $O(1)$ as the only extra memory used is for temporary variables.

Advantages of Selection Sort

- Easy to understand and implement, making it ideal for teaching basic sorting concepts.
- Requires only a constant $O(1)$ extra memory space.
- It requires less number of swaps (or memory writes) compared to many other standard algorithms. Only cycle sort beats it in terms of memory writes. Therefore it can be simple algorithm choice when memory writes are costly.

Sorting Techniques

Disadvantages of the Selection Sort

- Selection sort has a time complexity of $O(n^2)$ makes it slower compared to algorithms like Quick Sort or Merge Sort.
- Does not maintain the relative order of equal elements which means it is not stable.

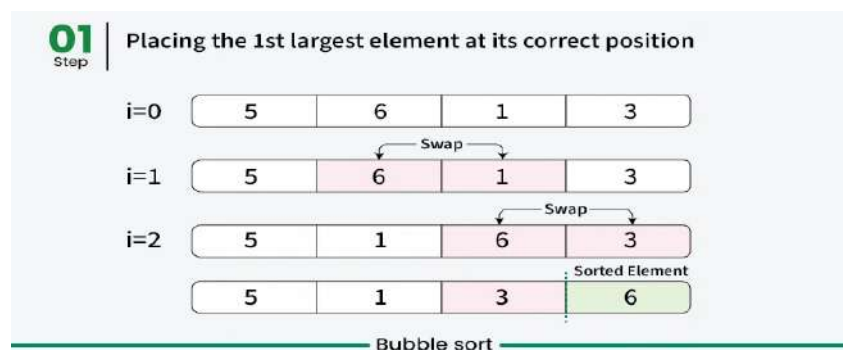
Applications of Selection Sort

- Perfect for teaching fundamental sorting mechanisms and algorithm design.
- Suitable for small lists where the overhead of more complex algorithms isn't justified and memory writing is costly as it requires less memory writes compared to other standard sorting algorithms.
- Heap Sort algorithm is based on Selection Sort.

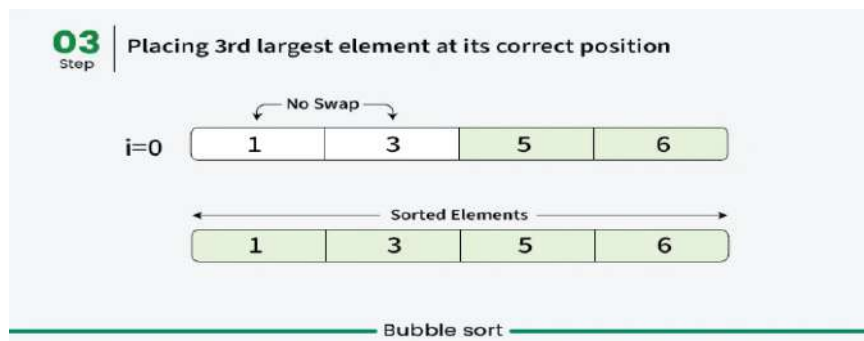
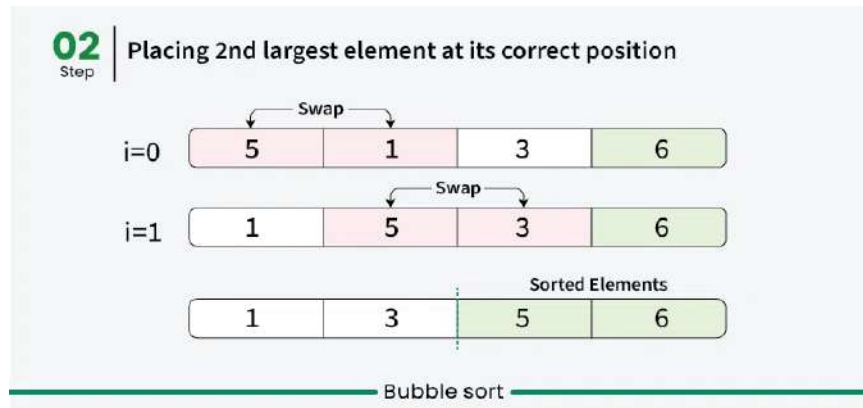
6- Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not efficient for large data sets as its average and worst-case time complexity are quite high.

- Sorts the array using multiple passes. After the first pass, the maximum goes to end (its correct position). Same way, after second pass, the second largest goes to second last position and so on.
- In every pass, process only those that have already not moved to correct position. After k passes, the largest k must have been moved to the last k positions.
- In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.



Sorting Techniques



Complexity Analysis of Bubble Sort:

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of $O(n^2)$ which makes it very slow for large data sets.
- Bubble sort has almost no or limited real world applications. It is mostly used in academics to teach different ways of sorting.

Sorting Techniques

Name	Best Case	Average Case	Worst Case	Memory	Stable	Method Used
Quick Sort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning
Merge Sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging
Heap Sort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection
Insertion Sort	n	n^2	n^2	1	Yes	Insertion
Selection Sort	n^2	n^2	n^2	1	No	Selection
Shell Sort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion
Bubble Sort	n	n^2	n^2	1	Yes	Exchanging

Search Algorithm (Non-Comparison Based)

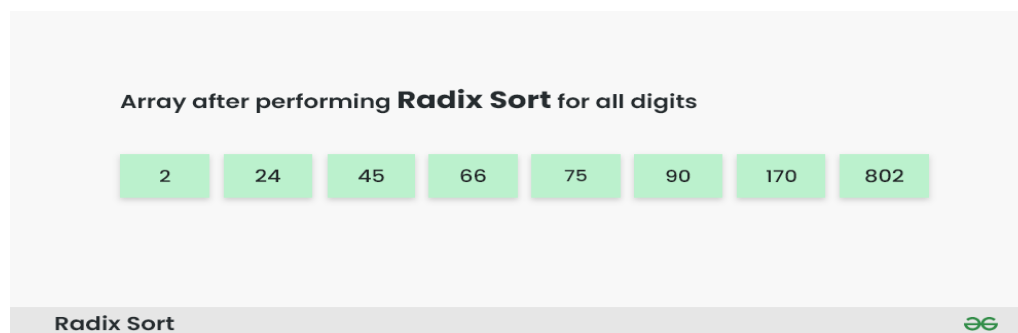
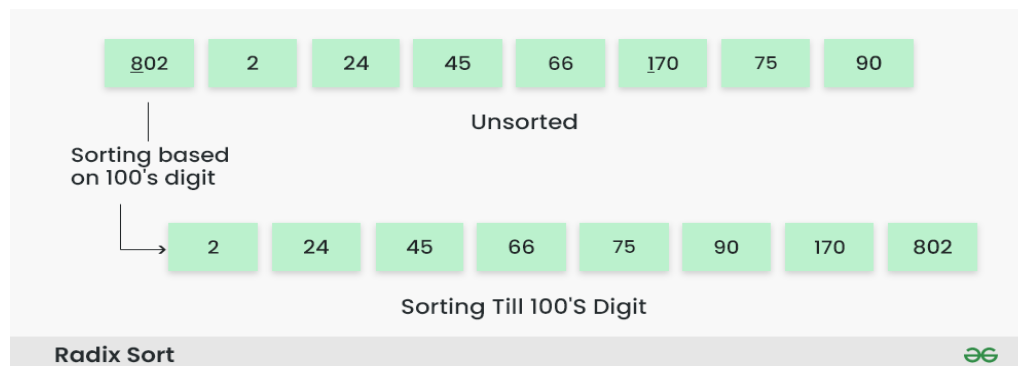
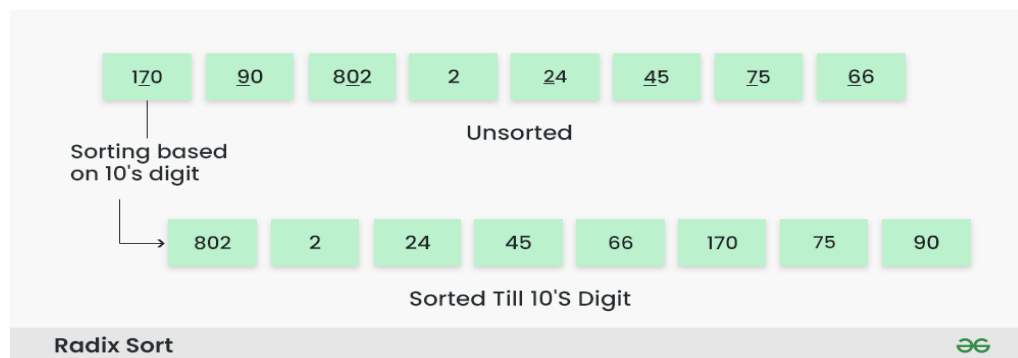
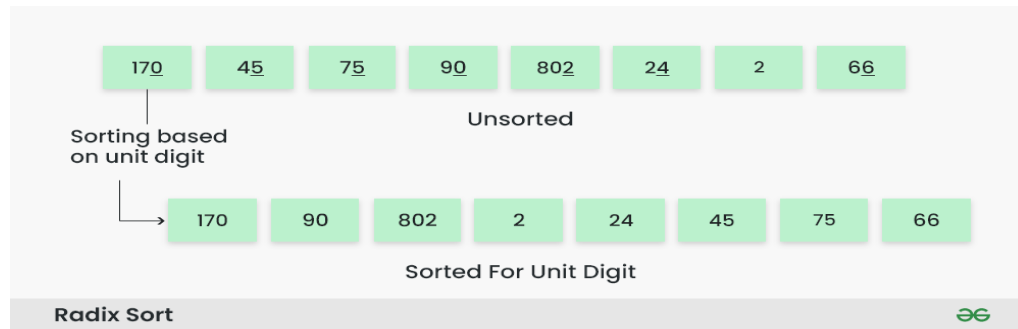
1- Radix Sort

Radix Sort is a linear sorting algorithm (for fixed length digit counts) that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

- It repeatedly distributes the elements into buckets based on each digit's value. This is different from other algorithms like Merge Sort or Quick Sort where we compare elements directly.

Sorting Techniques

- By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, it achieves the final sorted order.
- We use a stable algorithm like [Counting Sort](#) to sort the individual digits so that the overall algorithm remains stable.



Sorting Techniques

Complexity Analysis of Radix Sort:

Time Complexity:

- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by the individual digits which share the same significant position and value. It has a time complexity of $O(d * (n + b))$, where d is the number of digits, n is the number of elements, and b is the base of the number system being used.
- In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets.

Auxiliary Space:

- Radix sort also has a space complexity of $O(n + b)$, where n is the number of elements and b is the base of the number system. This space complexity comes from the need to create buckets for each digit value and to copy the elements back to the original array after each digit has been sorted.