

## Sorting

### Considerations

- Time
- Space (additional)
- In-Place
  - Space configuration -> small or constant amount of extra storage space
- Stable
  - If 2 objects with equal keys appear in the same order in sorted output as they appear in the input data set
- Simplicity
- Adaptive - run faster in trivial cases
- Comparison vs. Non-comparison

### Selection Sort

- Assume to sort in ascending order
- Each iteration:
  - Repeatedly selecting the minimum from the unsorted portion (linear search)
  - Swapping it with the first element in the unsorted portion
  - Best case:  $\Omega(n^2)$
  - Worst case:  $O(n^2)$
  - Average case:  $\Theta(n^2)$
  - Space:  $O(1)$
  - In-place: Yes
  - Stable: Yes
  - Adaptive: No

### Insertion Sort

- Insert a new element into a proper position of the sorted position
- Each iteration:
  - Compare it with last element in sorted position and swap it if in wrong order
  - Keep searching from the end to the beginning of the sorted portion, until reach a proper position
- Best case:  $\Omega(n)$
- Worst case:  $O(n^2)$
- Average case:  $\Theta(n^2)$
- Space:  $O(1)$
- Stable: Yes
- In-place: Yes
- Adaptive: Yes

## Bubble Sort

- Each iteration:
  - Compare each adjacent pair swap them if in wrong order
  - Unsorted position at the beginning;
  - Sorted position at the end;
  - Unsorted position | Sorted position
- Best case:  $\Omega(n)$
- Worst case:  $O(n^2)$
- Average case:  $\Theta(n^2)$
- Space:  $O(1)$
- Stable: Yes
- In-place: Yes
- Adaptive: Yes

## Shell Sort

- An optimization of insertion sort
- Partition the original array into some subarrays according to the gap size; do insertion sort for each subarray
- Iterate with decreasing gap size until it's equal to 1
- Best case:  $\Omega(n * \log(n))$
- Worst case: open problem ( $O(n * (\log(n))^2)$ )  
With best known gap size sequence,  $O(n^2)$   
with worst case gap size sequence
- Average case: open problem,  $\Theta(n * (\log(n))^2)$   
best known gap size sequence,  $\Theta(n^{3/2})$  with  
most common gap size sequences
- Space:  $O(1)$  gap size
- Stable: No
- In-place: Yes
- Adaptive: Yes

## Merge Sort

- Divide and Conquer Technique
- Divide the problem into several subproblems with smaller instances of the original problem
- Conquer the subproblems by solving them recursively; if the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner
- Combine the solutions of the subproblems into the final solution for the original problem
- Divide: Divide the  $n$ -element sequence into two subsequences of  $n/2$  elements each;
  - Pick the middle position as the partition point
  - Keep dividing until it reaches the base case
- Conquer: Sort the two subsequences recursively using merge sort

- Combine: Merge the two sorted subsequences to produce the sorted answer
- Best case:  $\Theta(n \log(n))$
- Worst case:  $\Theta(n \log(n))$
- Average case:  $\Theta(n \log(n))$
- Space:  $O(n)$
- Stable: Yes
- In-place: No
- Adaptive: No

### Quick Sort

- Apply the divide and conquer technique
- Pick a pivot
- Divide: Partition/rearrange the array into two subarrays according to the value of the pivot
  - Put all elements less than or equal to the pivot before the pivot, and put all elements (greater than the pivot) after the pivot
  - Keep picking a pivot and dividing two subproblems recursively until it reaches the base case
- Conquer: Sort the two subarrays by recursive calls to quick sort
- Combine: No work is needed because the subproblems are already sorted
- How to pick a pivot?
  - Always pick the first element as a pivot
  - Always pick the last element as a pivot
  - Pick random element as a pivot
  - Pick median as the pivot
- How to obtain two subproblems (partition)?  
Key process - pivot selection affects performance
- Best case:  $\Theta(n \log(n))$
- Worst case:  $\Theta(n^2)$
- Average case:  $\Theta(n \log(n))$
- How? Pick the pivot each iteration randomly (randomization)
- Space:  $O(\log(n))$
- Stable: No
- In-place: Yes
- Adaptive: No

### Heap Sort

- Better selection sort
- Uses heap structure rather than linear search to find the next value
- Heap data structure:
  - An array object with a heap property (imagine: a nearly complete binary tree);
  - Each node of the tree corresponds to an element of the array
  - The tree is completely filled on all levels except possibly the lowest which is filled from the left up to a point

- Binary heaps:
  - Min-heap and Max-heap
  - Maintain heap property
- Max-heap property:
  - $A[\text{parent}(i)] \geq A[i]$  (for every node  $i$  other than the root)
- Min-heap property:
  - $A[\text{parent}(i)] \leq A[i]$  (for every node  $i$  other than the root)
- Best case:  $\Theta(n \log(n))$
- Worst case:  $\Theta(n \log(n))$
- Average case:  $\Theta(n \log(n))$
- Space:  $O(1)$
- Stable: No
- In-place: Yes
- Adaptive: No

Selection Sort	Left of current position is sorted. Linear search for next smallest
Insertion Sort	Left of current position is sorted. Current item walks back until in proper place
Bubble Sort	After $n$ sweeps, right most $n$ items sorted Left to right sweep Each time, if $\text{curL} > \text{curR}$ , swap
Shell Sort	Iterated insertion sort with decreasing gap size - increment to skip by when inserting into sorted sublist
Merge Sort	Divide in half (recurse until small) "Sort" by merging two smaller sorted arrays
Quicksort	Randomly pick pivot All $<$ pivot goes to left of pivot, All $>$ pivot to right of pivot Recurse
Heap Sort	Make a heap Do Selection Sort, but with the heap to make finding the largest remaining number fast

Array	Contiguous block of memory
Linked List	Pointer to next node
Stack	Only add/remove from one side (LIFO)
Set	In/Out
Map	Key/Value
Queue	FIFO
Priority Queue	FIFO (can skip)
Tree	Parent/Child or Root/Leaves
Heap	Heap property (parent larger than children)
Hash Table	Hash function generates index from key

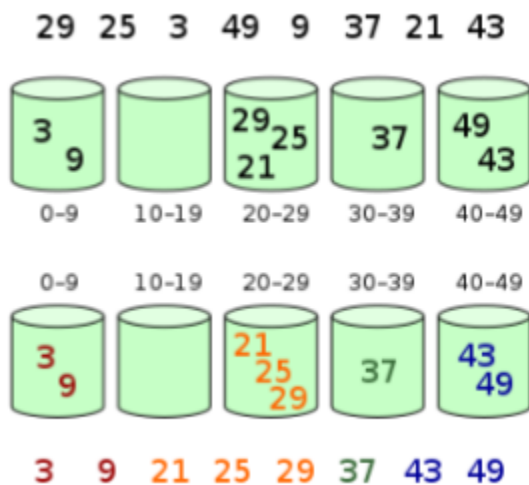
Data needs to be displayed in order	Sorted Array/List, Binary Search Tree
Fastest way to find data (asymptotically)	Hash Table
Slowest way to find data (array)	Linear Search
Mixes up the data order (unordered)	Hash Table
Can quickly find the largest/smallest element (one of these)	Heap
Can quickly find the largest/smallest element (both)	Tree
Find value (sorted): Runs in $\log(n)$ time	Binary Search
Find value (unsorted): Runs in $O(n)$ time	Linear Search
Find value (unordered): Runs in $O(1)$ time (expected/amortized)	Hash Table
Dealing with a complex situation where normal searches don't work (unorganized and non-linear structure (i.e. tree/graph))	Exhaustive Search Algorithm

## Comparison vs. Non-comparison Sorting

- Comparison sorting can be used with any comparable data regardless of the range or distribution of values
- Time complexity of sorting algorithms using only comparisons has a lower bound of  $\Omega(n \log n)$
- Using non-comparison sorting algorithms it is possible to reach  $\Theta(n)$  time complexity for data within a limited range with a particular distribution

## Bucket Sort

- Set up an array of initially empty “buckets”
- **Scatter:** Go over the original array, putting each object in its bucket
- Sort each non-empty bucket
- **Gather:** visit the buckets in order and put all elements back into the original array



## Counting Sort

- An integer sorting algorithm
- It works by determining the position of each key value in the output sequence by counting the number of objects with distinct key values and applying prefix sum to those counts
- Step 1: Find out the maximum element from the given array

### Step 1:

	0	1	2	3	4	5	6	7	max
inputArray	2	5	3	0	2	3	0	3	5

#### Counting Sort



- Step 2: Initialize a countArray[] of length max + 1 with all elements as 0, this array will be used for storing the occurrences of the elements of the input array

### Step 2:

	0	1	2	3	4	5
countArray	0	0	0	0	0	0

#### Counting Sort



- Step 3: In the countArray[], store the count of each unique element of the input array at their respective indices
  - For example: the count of element 2 in the input array is 2. So, store 2 at the index 2 in the countArray[]. Similarly, the count of element 5 in the input array is 1, hence store 1 at index 5 in the countArray[]

### Step 3 :

	0	1	2	3	4	5
<b>countArray</b>	<b>2</b>	<b>0</b>	<b>2</b>	<b>3</b>	<b>0</b>	<b>1</b>

#### Counting Sort



- Step 4: Store the cumulative sum or prefix sum of the elements of the countArray[] by doing  $\text{countArray}[i] = \text{countArray}[i - 1] + \text{countArray}[i]$ . This will help in placing the elements of the input array at the correct index in the output array.

### Step 4 :

	0	1	2	3	4	5
<b>countArray</b>	<b>2</b>	<b>2</b>	<b>4</b>	<b>7</b>	<b>7</b>	<b>8</b>

#### Counting Sort

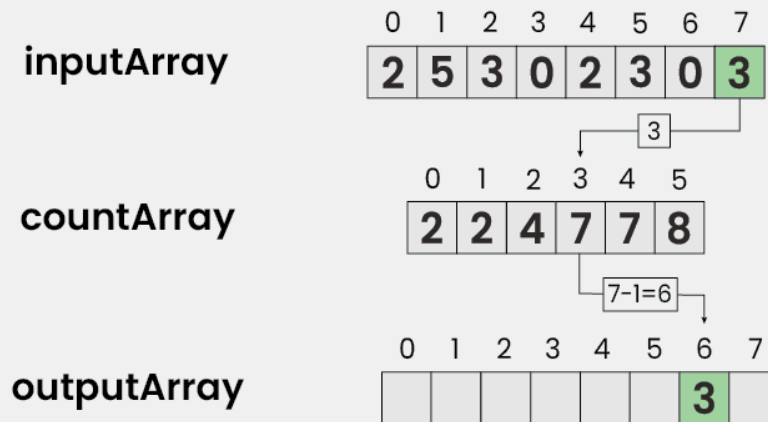


- Step 5: Iterate from end of the input array and because traversing input array from end preserves the order of equal elements, which eventually makes this sorting algorithm stable.
- Update  $\text{outputArray}[\text{countArray}[\text{inputArray}[i]] - 1] = \text{inputArray}[i]$ .



- Also, update  $\text{countArray}[\text{inputArray}[i]] = \text{countArray}[\text{inputArray}[i]] - 1$ .

### Step 5 :

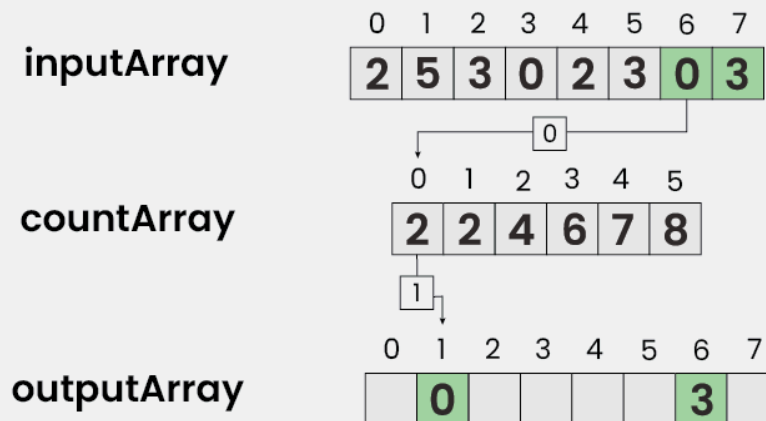


Counting Sort



- Step 6: For  $i = 6$ ,
- Update  $\text{outputArray}[\text{countArray}[\text{inputArray}[6]] - 1] = \text{inputArray}[6]$
- Also, update  $\text{countArray}[\text{inputArray}[6]] = \text{countArray}[\text{inputArray}[6]] - 1$

### Step 6 :

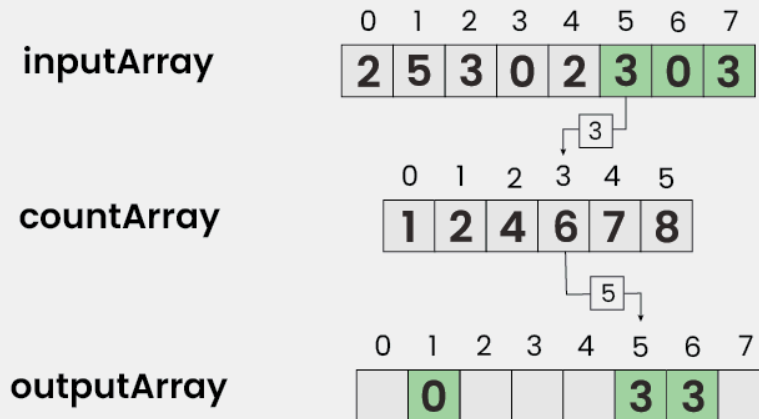


Counting Sort



- Step 7: For  $i = 5$ ,
- Update  $\text{outputArray}[\text{countArray}[\text{inputArray}[5]] - 1] = \text{inputArray}[5]$
- Also, update  $\text{countArray}[\text{inputArray}[5]] = \text{countArray}[\text{inputArray}[5]] - 1$

### Step 7:

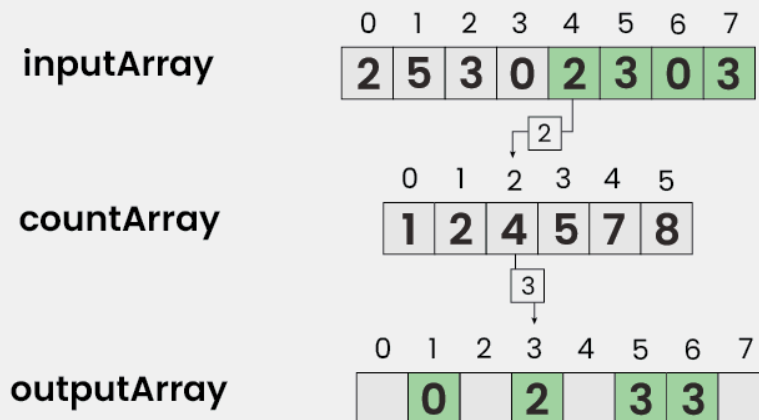


### Counting Sort



- Step 8: For  $i = 4$ ,
- Update  $\text{outputArray}[\text{countArray}[\text{inputArray}[4]] - 1] = \text{inputArray}[4]$
- Also, update  $\text{countArray}[\text{inputArray}[4]] = \text{countArray}[\text{inputArray}[4]] - 1$

### Step 8:

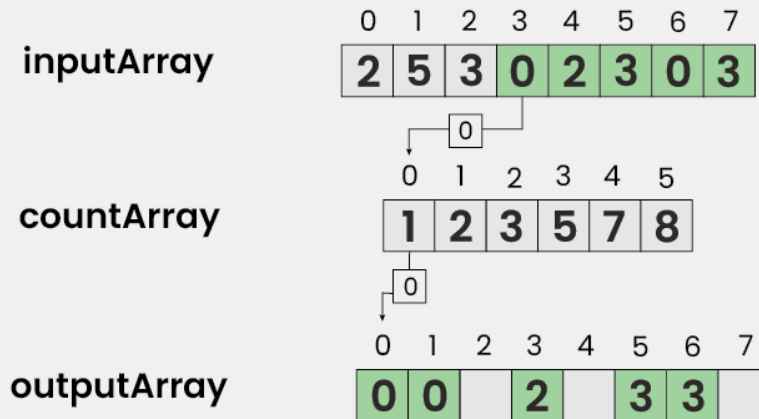


### Counting Sort



- Step 9: For  $i = 3$ ,
- Update  $\text{outputArray}[\text{countArray}[\text{inputArray}[3]] - 1] = \text{inputArray}[3]$
- Also, update  $\text{countArray}[\text{inputArray}[3]] = \text{countArray}[\text{inputArray}[3]] - 1$

### Step 9:

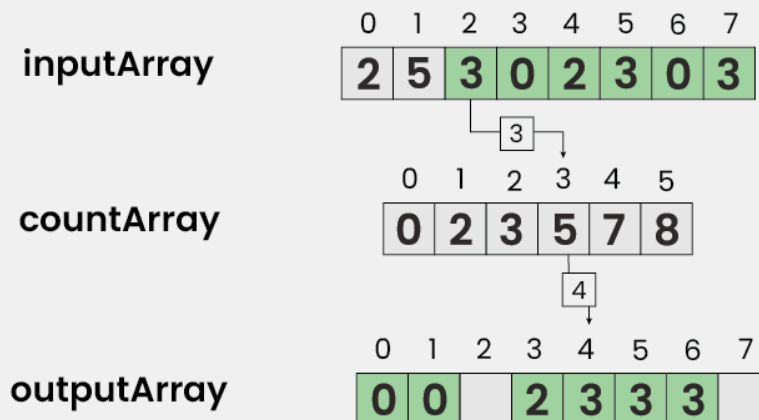


### Counting Sort



- Step 10: For  $i = 2$ ,
- Update  $\text{outputArray}[\text{countArray}[\text{inputArray}[2]] - 1] = \text{inputArray}[2]$
- Also, update  $\text{countArray}[\text{inputArray}[2]] = \text{countArray}[\text{inputArray}[2]] - 1$

### Step 10:

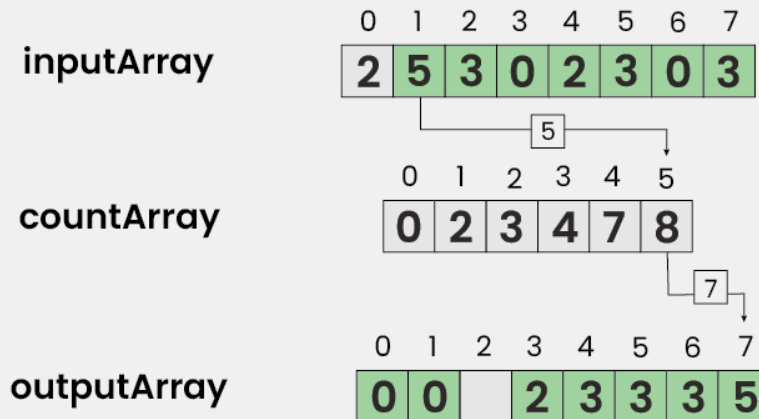


### Counting Sort



- Step 11: For  $i = 1$ ,
- Update  $\text{outputArray}[\text{countArray}[\text{inputArray}[1]] - 1] = \text{inputArray}[1]$
- Also, update  $\text{countArray}[\text{inputArray}[1]] = \text{countArray}[\text{inputArray}[1]] - 1$

### Step 11:

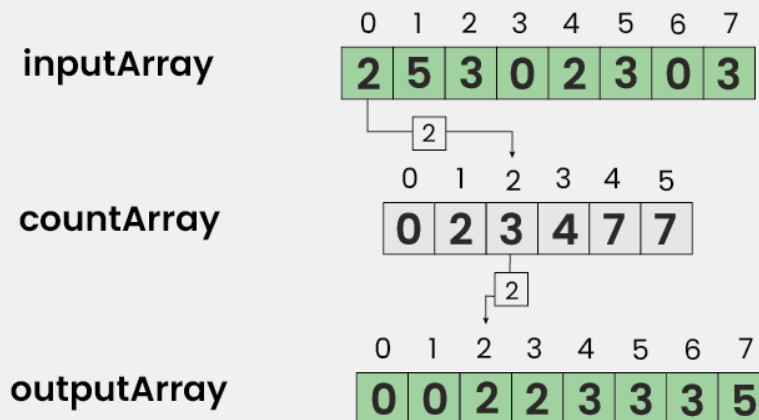


### Counting Sort



- Step 12: For  $i = 0$ ,
- Update  $\text{outputArray}[\text{countArray}[\text{inputArray}[0]] - 1] = \text{inputArray}[0]$
- Also, update  $\text{countArray}[\text{inputArray}[0]] = \text{countArray}[\text{inputArray}[0]] - 1$

### Step 12:



### Counting Sort



#### Counting Sort Algorithm:

- Declare an auxiliary array  $\text{countArray}[]$  of size  $\max(\text{inputArray}) + 1$  and initialize it with 0.
- Traverse array  $\text{inputArray}[]$  and map each element of  $\text{inputArray}[]$  as an index of  $\text{countArray}[]$  array, i.e., execute  $\text{countArray}[\text{inputArray}[i]]++$  for  $0 \leq i < N$ .
- Calculate the prefix sum at every index of array  $\text{inputArray}[]$ .
- Create an array  $\text{outputArray}[]$  of size  $N$ .

- Traverse array `inputArray[]` from end and update `outputArray[ countArray[ inputArray[i] ] - 1] = inputArray[i]`. Also, update `countArray[ inputArray[i] ] = countArray[ inputArray[i] ] - 1`.

## Radix Sort


- Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.

Consider this input

# Array

170	45	75	90	802	24	2	66
-----	----	----	----	-----	----	---	----

Unsorted

**Radix Sort** 

- Step 1: Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.
- Step 2: Sort the elements based on the unit place digits (X=0). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place.
- Sorting based on the unit place:
- Perform counting sort on the array based on the unit place digits.
- The sorted array based on the unit place is [170, 90, 802, 2, 24, 45, 75, 66].


170	45	75	90	802	24	2	66
-----	----	----	----	-----	----	---	----

Unsorted

Sorting based on unit digit

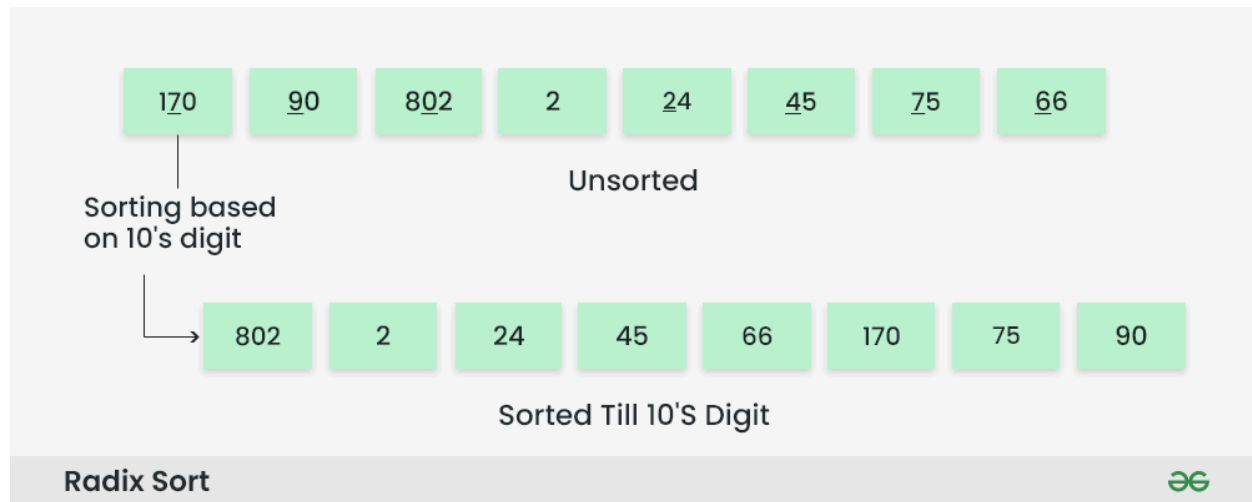
170	90	802	2	24	45	75	66
-----	----	-----	---	----	----	----	----

Sorted For Unit Digit

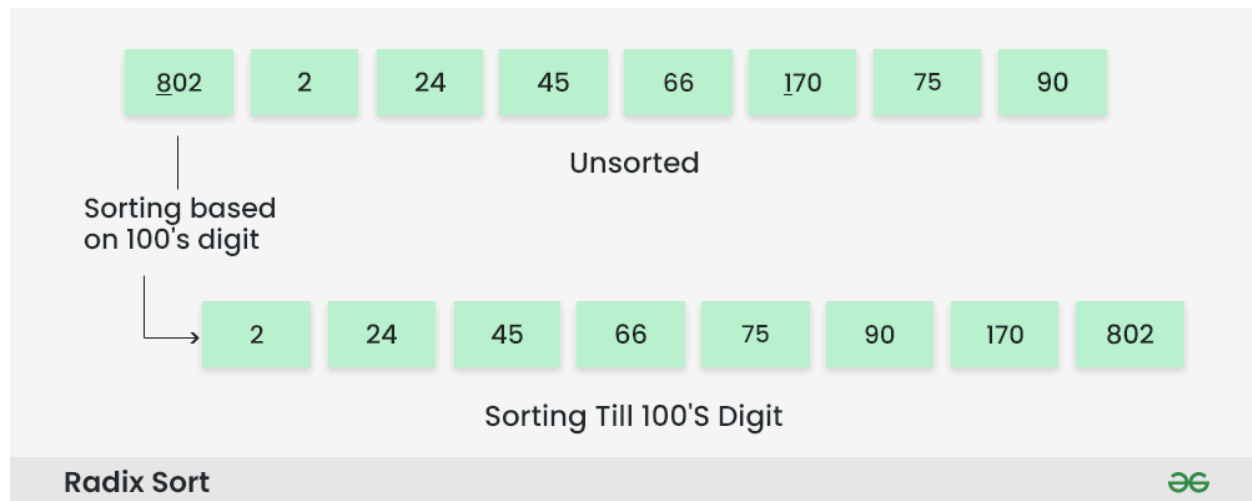
**Radix Sort** 

- Step 3: Sort the elements based on the tens place digits.
- Sorting based on the tens place:
- Perform counting sort on the array based on the tens place digits.

- The sorted array based on the tens place is [802, 2, 24, 45, 66, 170, 75, 90].



- Step 4: Sort the elements based on the hundreds place digits.
- Sorting based on the hundreds place:
- Perform counting sort on the array based on the hundreds place digits.
- The sorted array based on the hundreds place is [2, 24, 45, 66, 75, 90, 170, 802].



- Step 5: The array is now sorted in ascending order.
- The final sorted array using radix sort is [2, 24, 45, 66, 75, 90, 170, 802].