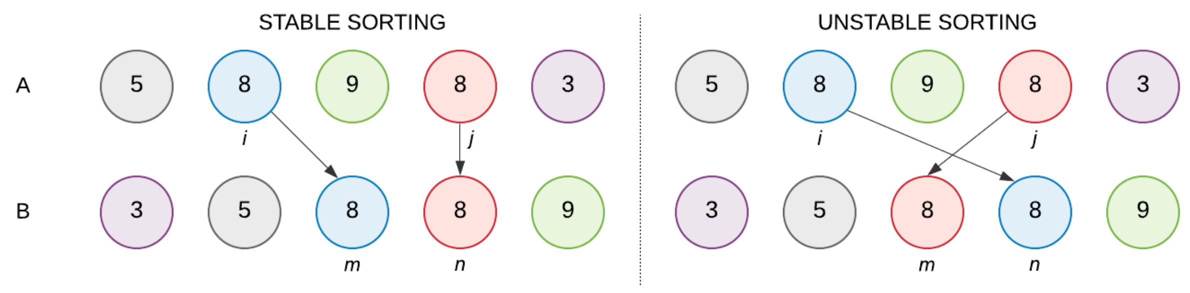
Lecture 5: Linear Sorting Algorithms

September 5, 2023

Stable Sorting

A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input data set.



⇒ When sorting integers, order does not matter. But, the order matters when we are sorting complex data structures.

- array of pairs of integers, e.g. [(1,2), (1,5), (1,6)]. and we are sorting on the first element of the pair then even if the first element is the same, the second element might be different.
- Amazon warehouse scenario, we try to sort all the boxes with their respective sizes. There will be some boxes whose size are the same but the content inside them might be different.

Counting Sort

Counting Sort is a very time efficient algorithm for sorting. Unlike selection sort and merge sort, counting sort is not a comparison based algorithm.
⇒ It avoids comparisons and exploits the O(1) time insertions and lookup in an array.
⇒ It works by determining the positions of each key value in the output sequence by counting the number of objects with distinct key values.

Pseudocode

```
function sort(array, valueEnd){
      create an array called Counts of size valueEnd.
      calculate the frequency of each element in the array
      print all elements linearly as per the Count array
}
```

Implementation

The below implementation will only work for an array which only has unique elements. (This implementation is easier to do analysis on)

1_counting_sort.cpp

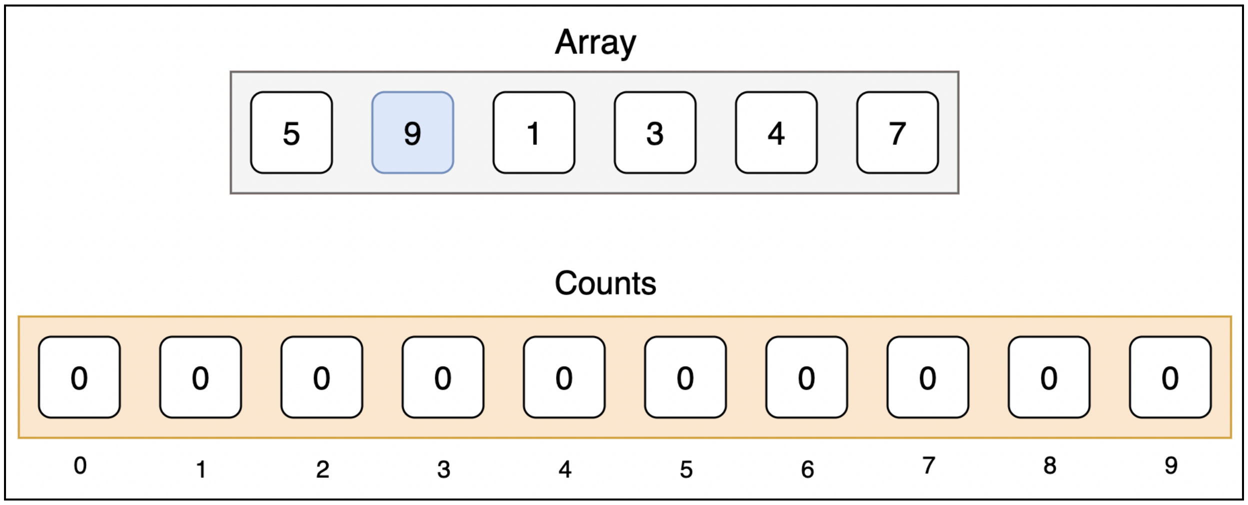
```cpp
vector<int> counting_sort(vector<int> nums, int k){
   // initializing an array of size k, with value 0
   vector<bool> exist(k+1, false);

   // marking which elements exist in array
   for(int i=0; i<nums.size(); i++){
       int currValue = nums[i];
       exist[currValue] = true;
   }

   // finding the sorted order of nums
   vector<int> ans;
   for(int i=0; i<=k; i++){
       if(exist[i])
       ans.push_back(i);
   }
   return ans;
}
```

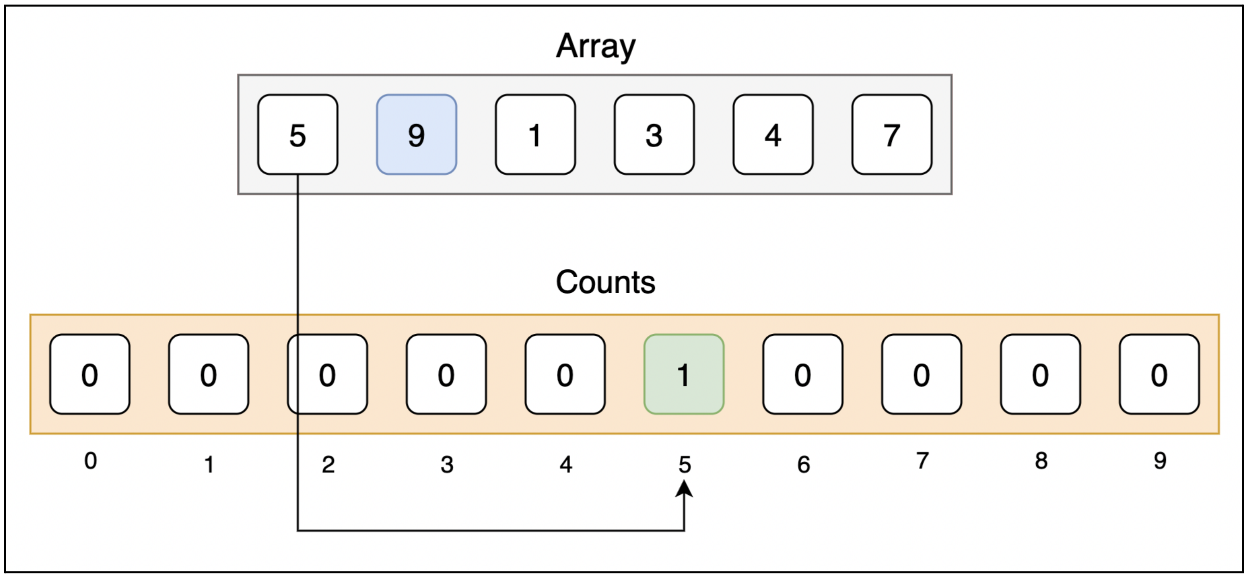
This implementation supports duplicate values in the input array.

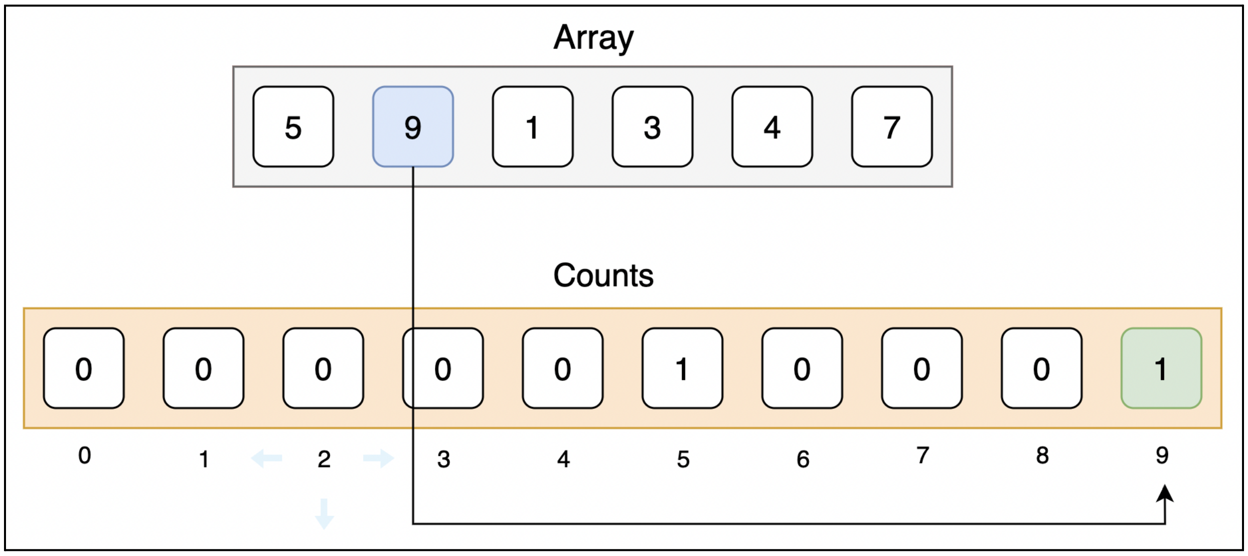
2_counting_sort.cpp

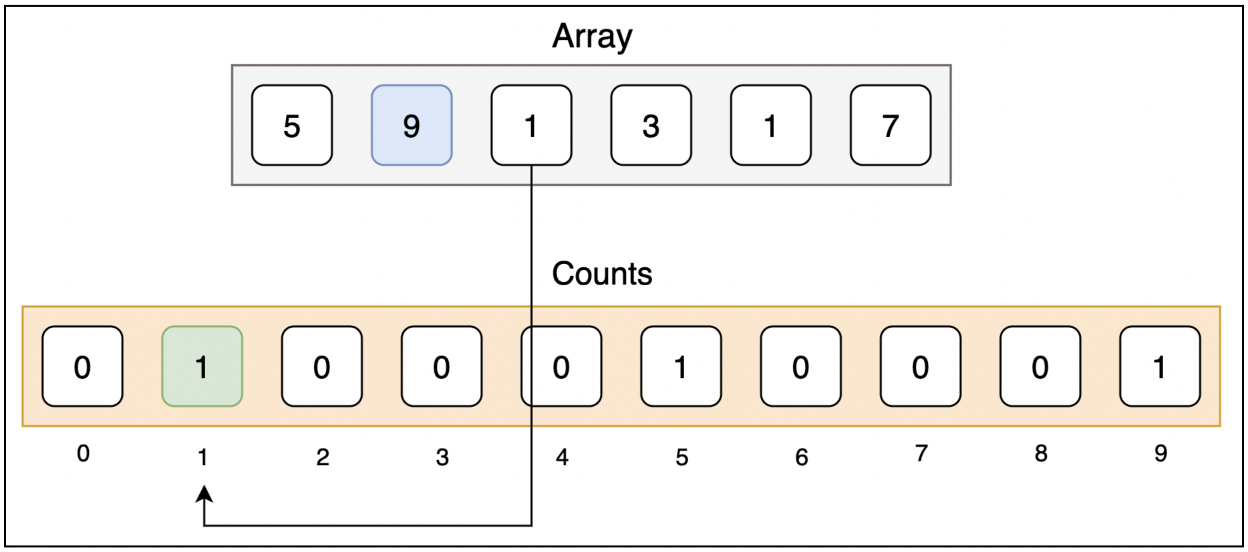
```
vector<int> counting_sort(vector<int> nums, int k){
    // initializing an array of size k, with value 0
    vector<int> counts(k+1, 0);

    // finding frequency of elements in the array
    for(int i=0; i<nums.size(); i++){
        int currValue = nums[i];
        counts[currValue] += 1;
    }

    // finding the sorted order of nums
    vector<int> ans;
    for(int i=0; i<=k; i++){
         while(counts[i] > 0){
            ans.push_back(i);
            counts[i] --;
         }
    }
    return ans;
}
```

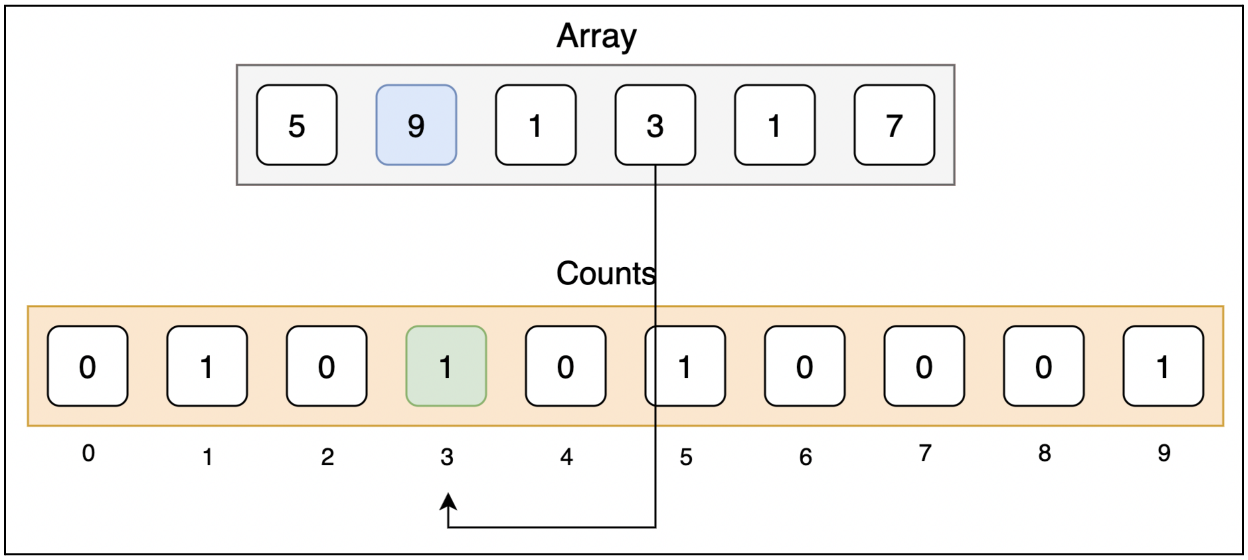
Example and Visualization

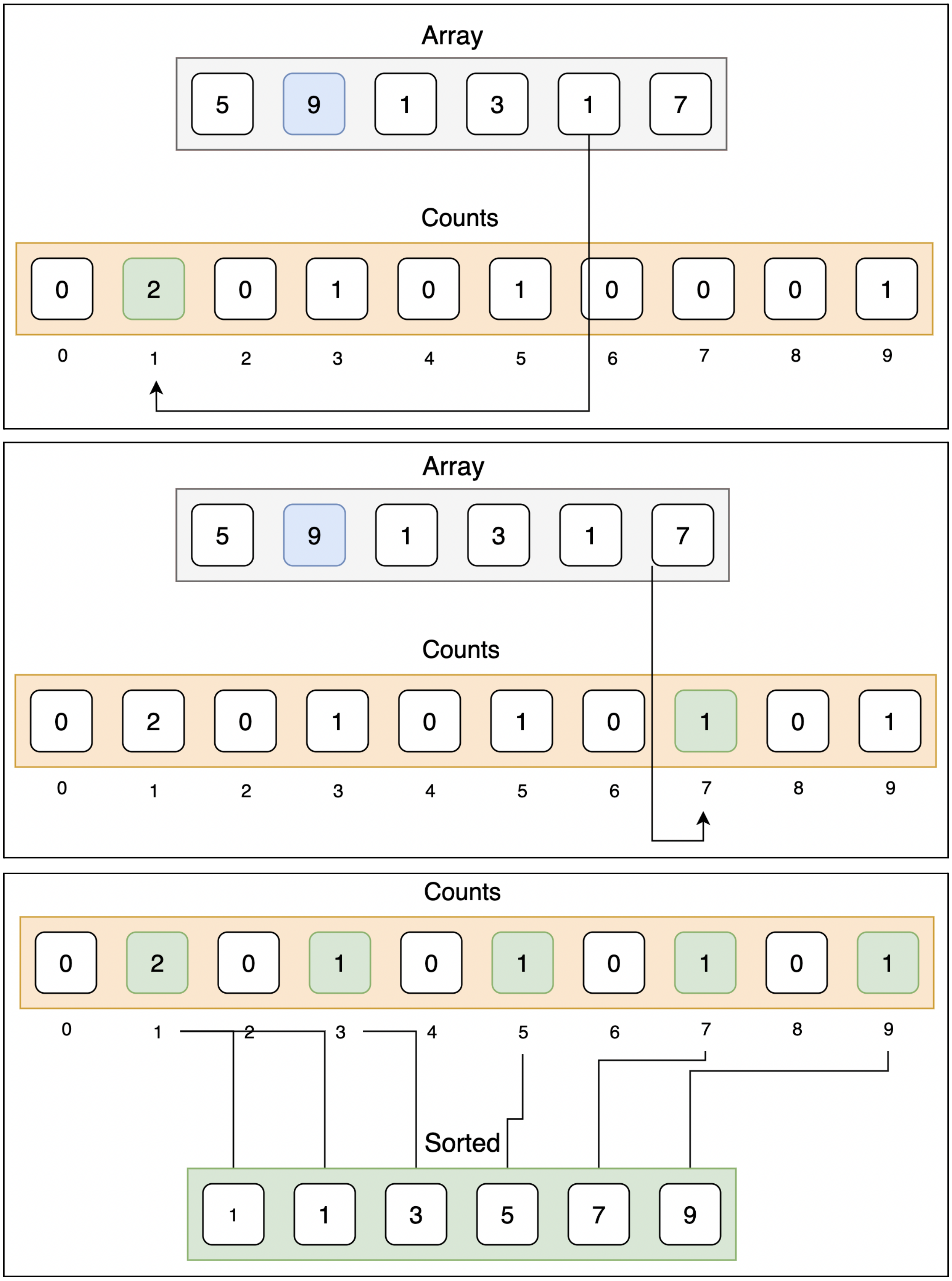





Array
5 9 1 3 4 7
Counts
0 0 0 0 0 1 0 0 0 1
0 1 2 3 4 5 6 7 8 9


Array
5 9 1 3 1 7
Counts
0 1 0 0 0 1 0 0 0 1
0 1 2 3 4 5 6 7 8 9


Array
5 9 1 3 1 7
Counts
0 1 0 1 0 1 0 0 0 1
0 1 2 3 4 5 6 7 8 9


Array
5
9
1
3
1
7
Counts
0
2
0
1
0
1
0
0
0
1
0
1
2
3
4
5
6
7
8
9
Array
5
9
1
3
1
7
Counts
0
2
0
1
0
1
0
1
0
1
0
1
2
3
4
5
6
7
8
9
Counts
0
2
0
1
0
1
0
1
0
1
0
1
2
3
4
5
6
7
8
9
Sorted
1
1
3
5
7
9

Complexity Analysis

Time Complexity

Finding total instruction that counting sort will be running is straightforward.
⇒ Time taken to initialize `counts` array will be k steps.
⇒ Marking all elements that exist in the array will be n steps.
⇒ Finding the sorted order of numbers in the array will be k steps.

Total Instructions: k + n + k

⇒ **Time Complexity: θ(n + k)**

Now, let's do analysis on the value of **k**,

- When n > k, meaning the size of the array is much larger than the maximum element in the array. (for example, size of array is in terms of 10^5, 10^6 but the values inside that array is just in terms of 10^2)
  Time complexity: **θ(n)**
- When k > n, meaning the values inside the array is much larger than the size of the array. (for example, size of array is in terms of 10^2 and the values are in terms of 10^8 and 10^9)
  Time complexity: **θ(k)**

Space Complexity

The algorithm creates two arrays, `counts` and `ans`. But the `ans` array is used to return output, so the only variable that will be counted towards auxiliary space complexity is `counts` array. Which is of size k.

- Input Space Complexity: **θ(n)**
- Auxiliary Space Complexity: **θ(k)**
- Output Space Complexity: **θ(n)**

Radix Sort

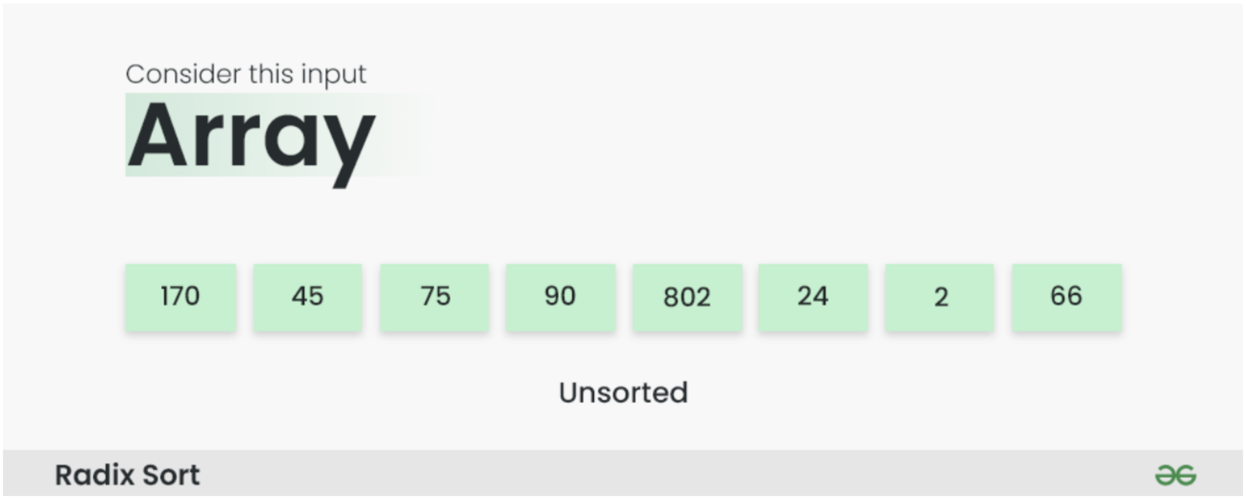
Sorts elements by processing them digit by digit.
⇒ It is an efficient sorting algorithm for integers or strings with fixed-size keys.

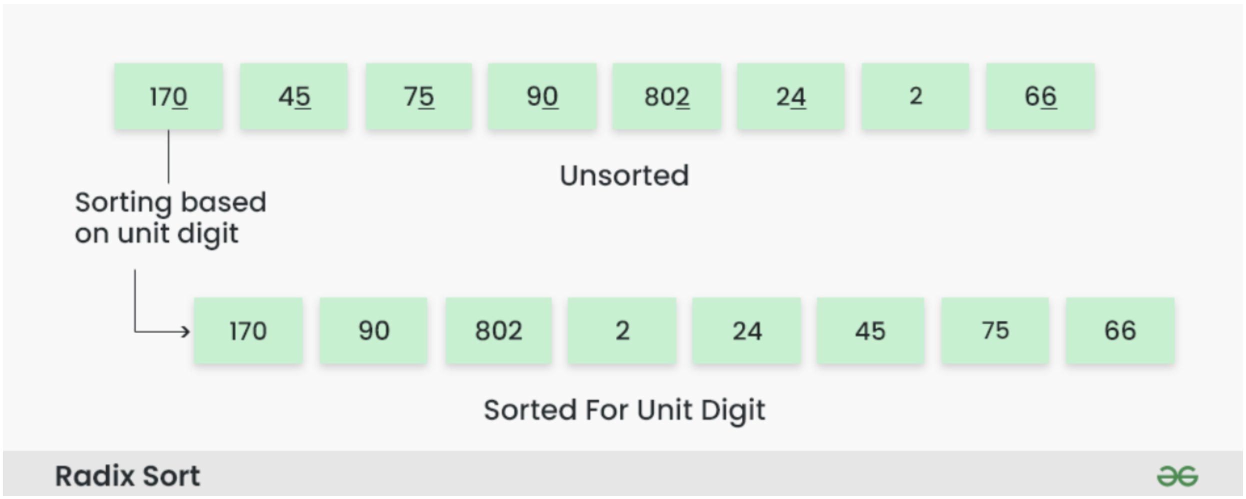
Pseudocode

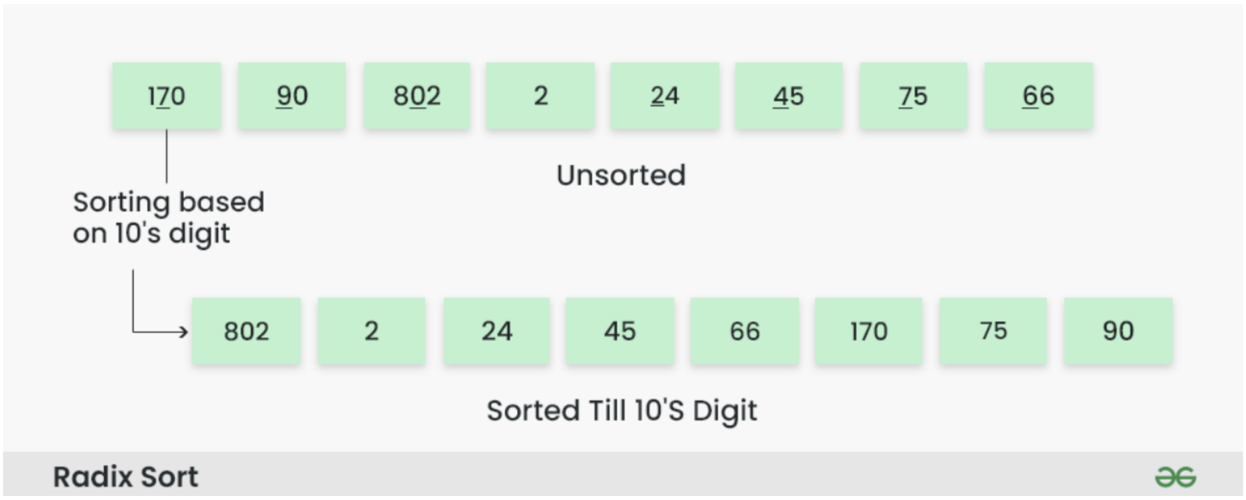
```
function sort(array){
      for d in 0...total_digits:
            count_sort based on digit d
}
```

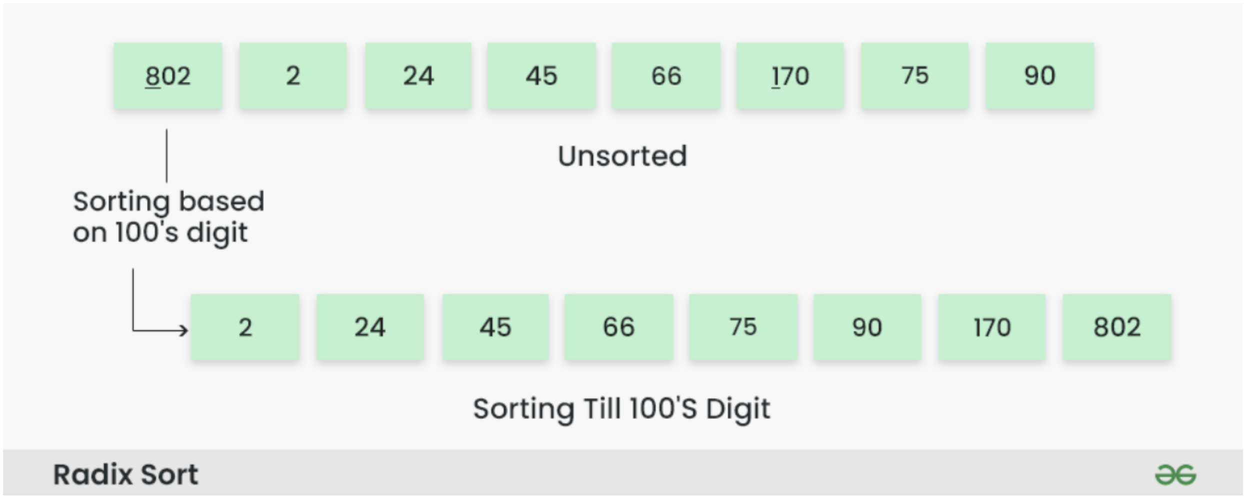
Example and Visualization

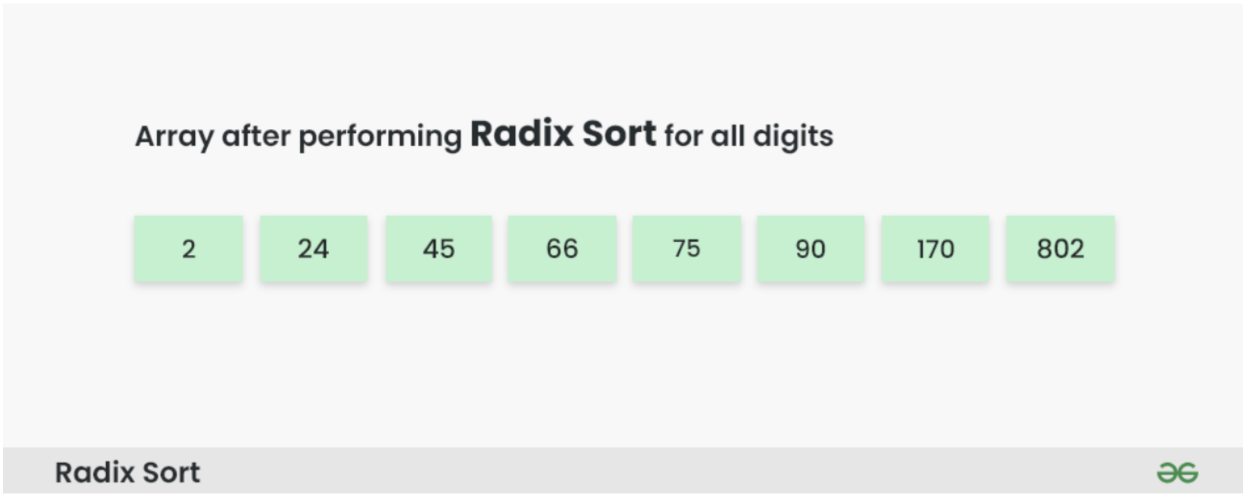
reference: geeksforgeeks.com


Consider this input
Array
170
45
75
90
802
24
2
66
Unsorted
Radix Sort


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


170
90
802
2
24
45
75
66
Unsorted
Sorting based on 10's digit
802
2
24
45
66
170
75
90
Sorted Till 10'S Digit
Radix Sort





Complexity Analysis

Time Complexity

⇒ The count sort algorithm is called the **total number of digits** times.
⇒ Count Sort is always called on only one digit from a whole integer, so k will be in range of 0 to 9. and n might get larger.
We can say count sort will take θ(n) time.

Time Complexity: **θ(n * d)**
where d is the total number of digits.

Space Complexity

⇒ Radix sort uses count sort algorithm, which uses θ(k) auxiliary space complexity. Here in every iteration, k will be 9 (as a digit can only vary from 0 to 9)

- Input Space Complexity: **θ(n)**
- Auxiliary Space Complexity: **θ(1)**
- Output Space Complexity: **θ(n)**

Resources

- **Readings**: Introduction to Algorithms, 4e. Section 8.1, 8.2, 8.3
- Polls