



Radix Sort



Counting Sort

Previously, we saw counting sort Algorithm, which was a non-comparison based sorting algorithm.



Non-Comparison Sorting Algorithms

Non-Comparison Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Counting Sort	$O(N + K)$	$O(N + K)$	$O(N + K)$	$O(N+K)$

Non-Comparison Sorting Algorithms

Sorting Algorithm	In-Place	Stable
Counting Sort	No	Yes

Counting Sort: Limitation

What is the limitation of the Counting Sort?



Counting Sort: Limitation

1. Counting sort is used only if the items we want to sort are all **integers**.
2. Counting Sort works best if the **range of integers** to be sorted **is not wide**, i.e., not greater than the size of input array.

Counting Sort: Improvement

Now, can we improve the solution so that we can efficiently sort the following data using non-comparison based sorting algorithm?

181	289	390	121	145	736	514	212
0	1	2	3	4	5	6	7

Counting Sort: Improvement

Instead of sorting the complete number using counting sort, let's first sort the number on the basis of units place, then tens place, then hundreds place using counting sort.

181	289	390	121	145	736	514	212
-----	-----	-----	-----	-----	-----	-----	-----

0

1

2


3

4

5

6

7

1 8 1
hundred  unit
 ten



Counting Sort: Improvement

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

Counting Sort: Improvement

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

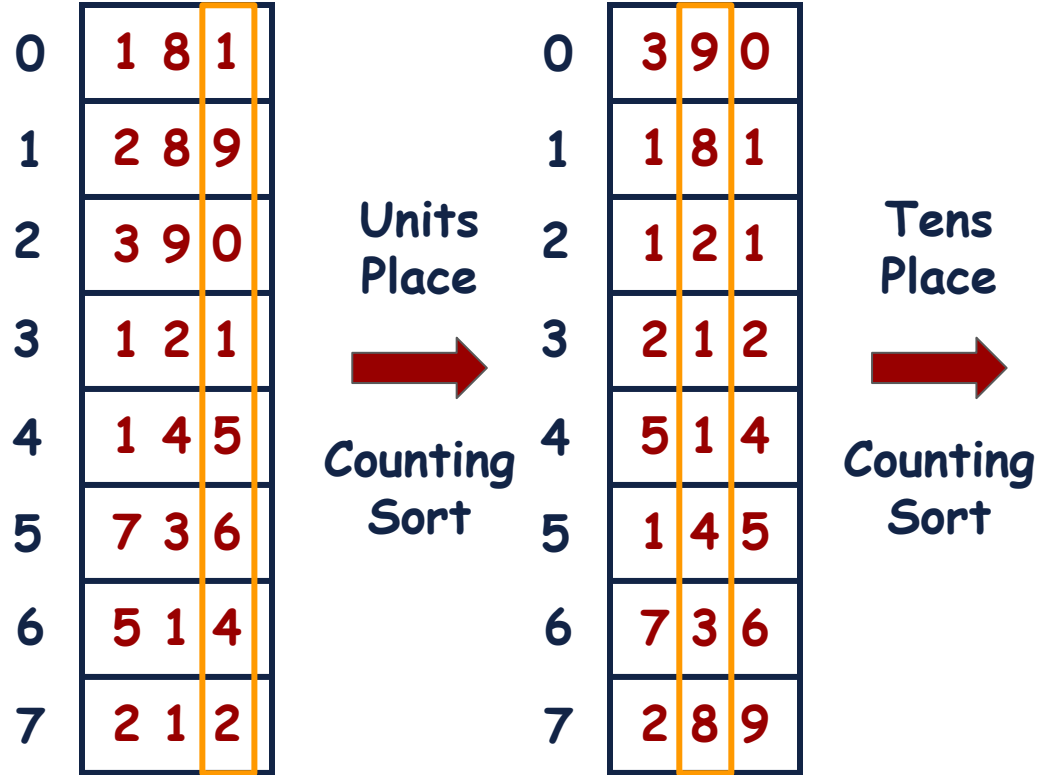
Units
Place

Counting
Sort

Counting Sort: Improvement

0	1 8 1	<div>Units Place</div> <div>→</div> <div>Counting Sort</div>	0	3 9 0
1	2 8 9		1	1 8 1
2	3 9 0		2	1 2 1
3	1 2 1		3	2 1 2
4	1 4 5		4	5 1 4
5	7 3 6		5	1 4 5
6	5 1 4		6	7 3 6
7	2 1 2		7	2 8 9

Counting Sort: Improvement



Counting Sort: Improvement

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

Units
Place

Counting
Sort

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

Tens
Place

Counting
Sort

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

Counting Sort: Improvement

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

Units
Place

Counting
Sort

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

Tens
Place

Counting
Sort

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

Hundreds
Place

Counting
Sort

Counting Sort: Improvement

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

Units
Place

Counting
Sort

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

Tens
Place

Counting
Sort

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

Hundreds
Place

Counting
Sort

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

Counting Sort: Improvement

This sorting includes bases, and Radix is another term for Base.

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

Units
Place

Counting
Sort

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

Tens
Place

Counting
Sort

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

Hundreds
Place

Counting
Sort

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

Radix Sort

Therefore, this algorithm is called Radix Sort.

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

Units
Place

Counting
Sort

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

Tens
Place

Counting
Sort

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

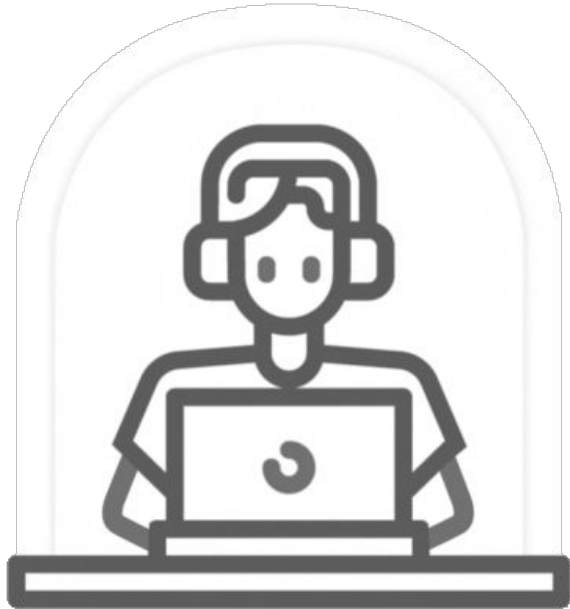
Hundreds
Place

Counting
Sort

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

Radix Sort: Implementation

Now, Let's implement the Radix Sort Algorithm.



FOCUS · MODE

Radix Sort: Implementation

Step 1:

Update the counting sort algorithm, such that we have pass according to which place it has to sort the data.

```
void countingSort(vector<int> &arr, int place)
{
}
```

Radix Sort

```
void countingSort(vector<int> &arr, int place)
{
    vector<int> count(10);
    vector<int> output(arr.size());
    for (int x = 0; x < arr.size(); x++)
    {
        count[(arr[x]/place) % 10]++;
    }
    for (int x = 1; x < count.size(); x++)
    {
        count[x] = count[x - 1] + count[x];
    }
    for (int x = arr.size() - 1; x >= 0; x--)
    {
        int index = count[(arr[x]/place) % 10] - 1;
        count[(arr[x]/place) % 10]--;
        output[index] = arr[x];
    }
    for (int x = 0; x < output.size(); x++)
    {
        arr[x] = output[x];
    }
}
```

Radix Sort: Implementation

Step 2:

Now call the counting sort algorithm according to the number of digits present in the maximum element.

```
main()
{
    vector<int> arr = {181, 289, 390, 121, 145, 736, 514, 212};
    radixSort(arr);
    for (int x = 0; x < arr.size(); x++)
    {
        cout << arr[x] << " ";
    }
}
```

Radix Sort: Implementation

Step 2:

Now call the counting sort algorithm according to the number of digits present in the maximum element.

```
void radixSort(vector<int> &arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int place = 1;
    while ((max / place) > 0)
    {
        countingSort(arr, place);
        place = place * 10;
    }
}
```

Radix Sort: Time Complexity

What is the time complexity of Radix sort?

```
void radixSort(vector<int> &arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int place = 1;
    while((max / place) > 0)
    {
        countingSort(arr, place);
        place = place * 10;
    }
}
```

Radix Sort: Time Complexity

We know the time complexity of Counting Sort is $O(n + k)$.

```
void radixSort(vector<int> &arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int place = 1;
    while((max / place) > 0)
    {
        countingSort(arr, place);
        place = place * 10;
    }
}
```


Radix Sort: Time Complexity

We know the time complexity of Counting Sort is $O(n + k)$.

Now, how many times we are calling the counting sort?

```
void radixSort(vector<int> &arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int place = 1;
    while ((max / place) > 0)
    {
        countingSort(arr, place);
        place = place * 10;
    }
}
```

Radix Sort: Time Complexity

We know the time complexity of Counting Sort is $O(n + k)$.

Number of digits in the max element.

```
void radixSort(vector<int> &arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int place = 1;
    while ((max / place) > 0)
    {
        countingSort(arr, place);
        place = place * 10;
    }
}
```

Radix Sort: Time Complexity

We know the time complexity of Counting Sort is $O(n + k)$.

Number of digits in the max element. Let's call it d .

```
void radixSort(vector<int> &arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int place = 1;
    while ((max / place) > 0)
    {
        countingSort(arr, place);
        place = place * 10;
    }
}
```

Non-Comparison Sorting Algorithms

Non-Comparison Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Counting Sort	$O(N + K)$	$O(N + K)$	$O(N + K)$	$O(N+K)$
Radix Sort	$O(D*(N+K))$	$O(D*(N+K))$	$O(D*(N+K))$	$O(N+K)$

$K = 10$ (Base of the number system used)

Non-Comparison Sorting Algorithms

Sorting Algorithm	In-Place	Stable
Counting Sort	No	Yes
Radix Sort	No	Yes

Counting Sort & Radix Sort

Counting Sort and Radix Sort are mostly used when the numbers to be sorted are whole numbers.

Now, what if we want to sort the decimal numbers?

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
0	1	2	3	4	5	6	7	8	9

Counting Sort & Radix Sort

In such case, it is better to use another non-comparison based sorting algorithm.

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
0	1	2	3	4	5	6	7	8	9



Bucket Sort



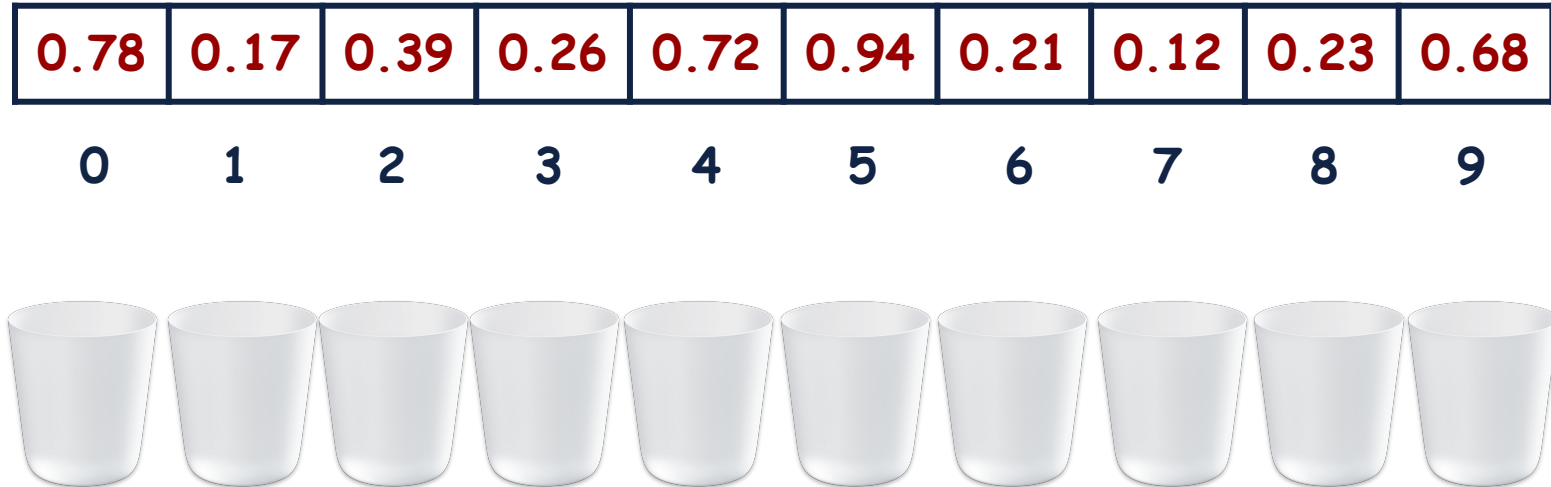
Bucket Sort

Step 1: Create the buckets according to the size of the input array.

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
0	1	2	3	4	5	6	7	8	9

Bucket Sort

Step 1: Create the buckets according to the size of the input array.



Bucket Sort

Step 2: Multiply the elements in the array with the size of the array, take the floor value and place the element in the specific bucket.

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------

0 1 2 3 4 5 6 7 8 9



Bucket Sort

Step 2: Multiply the elements in the array with the size of the array, take the floor value and place the element in the specific bucket.

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------

0 1 2 3 4 5 6 7 8 9

$$0.78 * 10 = \text{floor}(7.8) = 7$$



Bucket Sort

Step 2: Multiply the elements in the array with the size of the array, take the floor value and place the element in the specific bucket.

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------

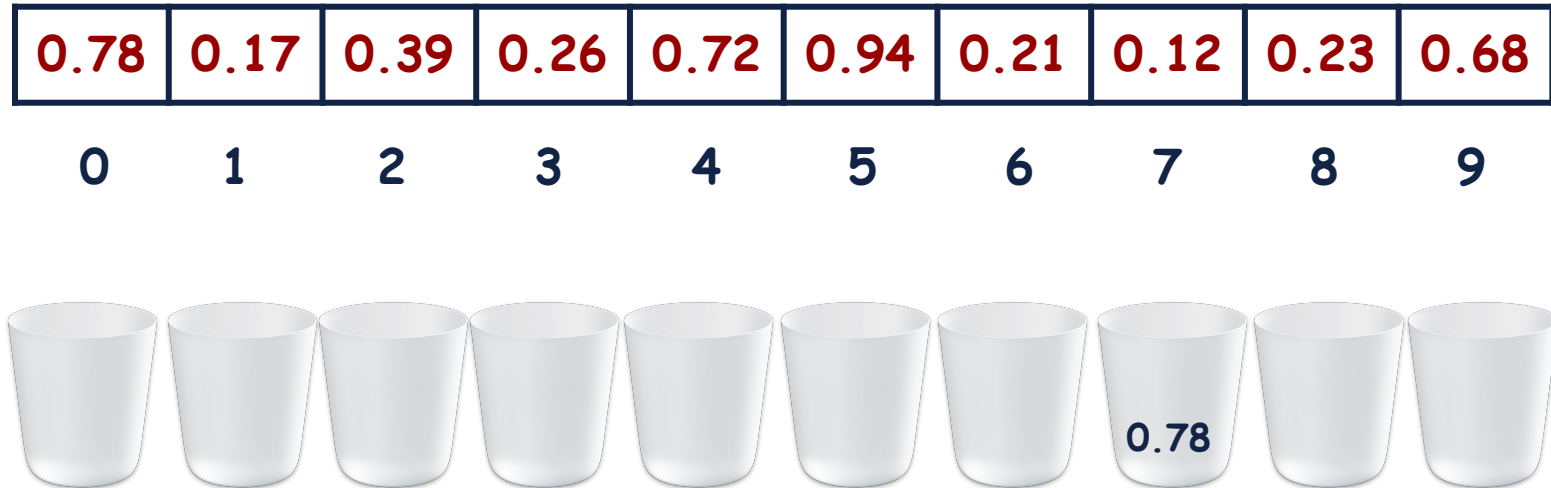
0 1 2 3 4 5 6 7 8 9

$$0.78 * 10 = \text{floor}(7.8) = 7$$



Bucket Sort

Step 3: Repeat the same process for all the elements.



Bucket Sort

Step 3: Repeat the same process for all the elements.



0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------

0 1 2 3 4 5 6 7 8 9

$$0.17 * 10 = \text{floor}(1.7) = 1$$



Bucket Sort

Step 3: Repeat the same process for all the elements.



0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------

0 1 2 3 4 5 6 7 8 9

$$0.39 * 10 = \text{floor}(3.9) = 3$$



Bucket Sort

Step 3: Repeat the same process for all the elements.



0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------

0 1 2 3 4 5 6 7 8 9

$$0.26 * 10 = \text{floor}(2.6) = 2$$



Bucket Sort

Step 3: Repeat the same process for all the elements.



0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------

0 1 2 3 4 5 6 7 8 9

$$0.72 * 10 = \text{floor}(7.2) = 7$$



Bucket Sort

Step 3: Repeat the same process for all the elements.



0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68
------	------	------	------	------	------	------	------	------	------

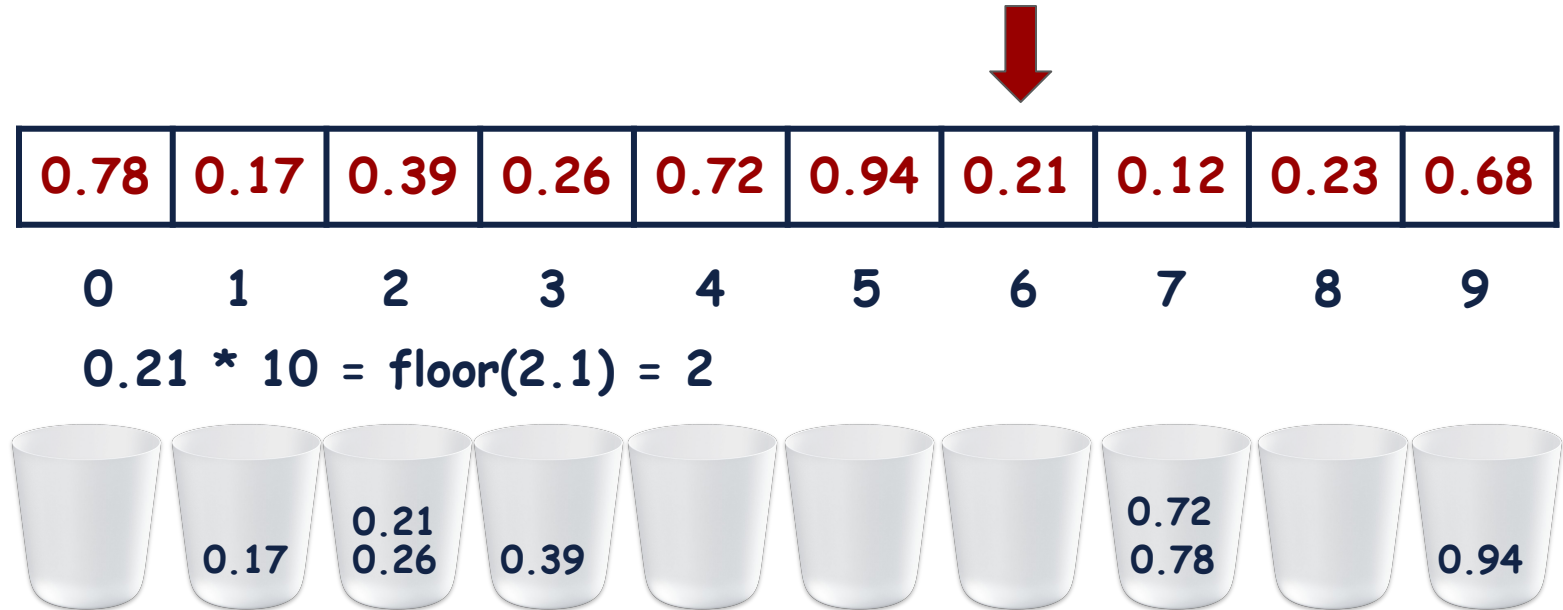
0 1 2 3 4 5 6 7 8 9

$$0.94 * 10 = \text{floor}(9.4) = 9$$



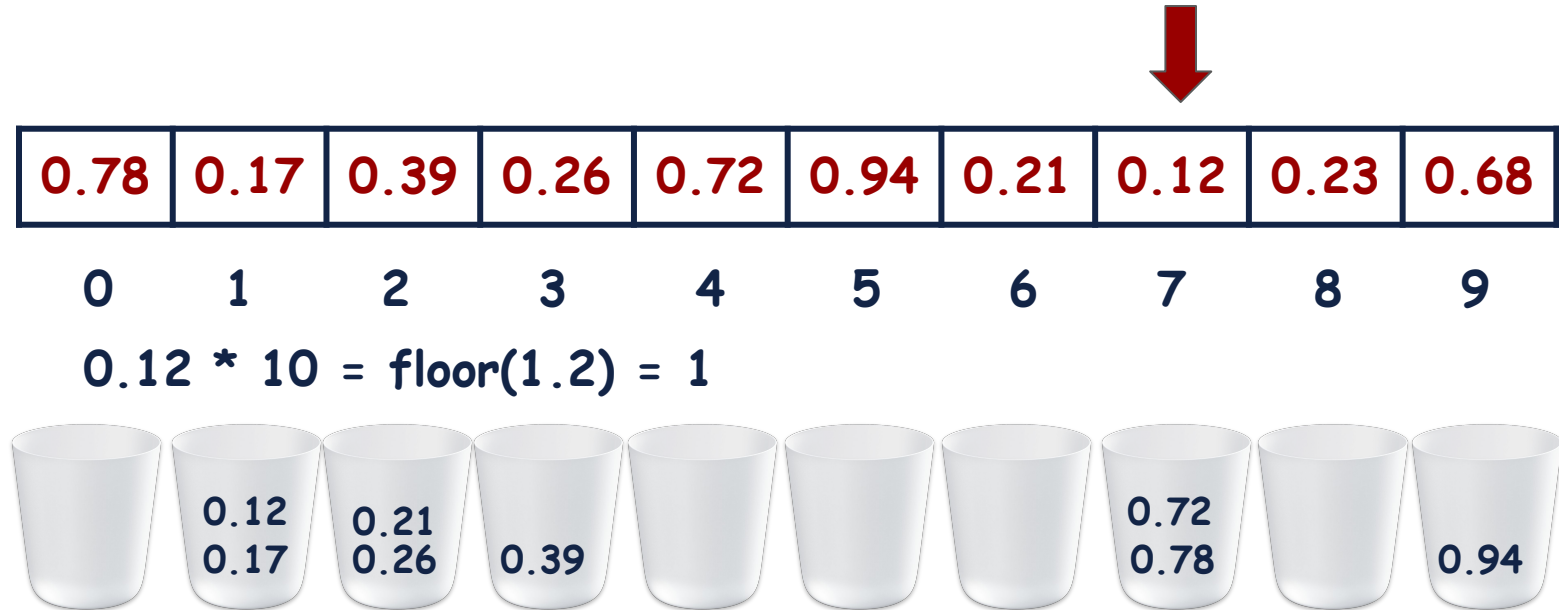
Bucket Sort

Step 3: Repeat the same process for all the elements.



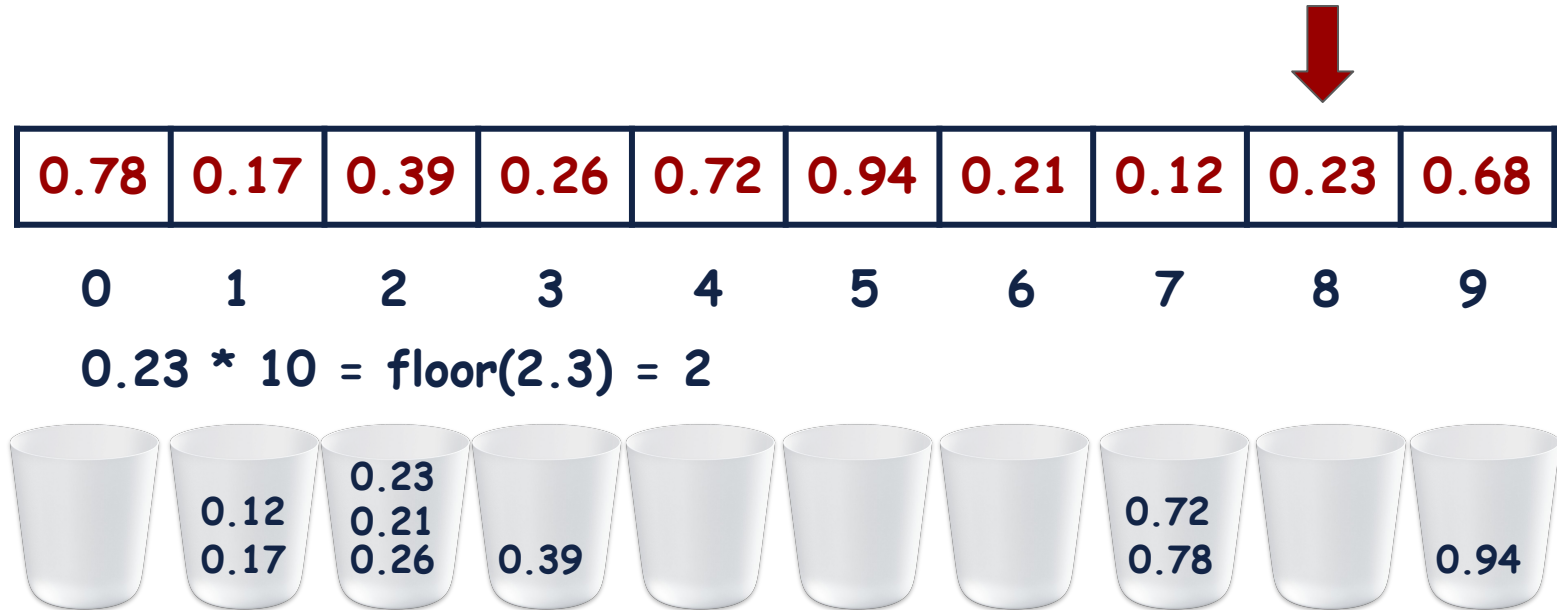
Bucket Sort

Step 3: Repeat the same process for all the elements.



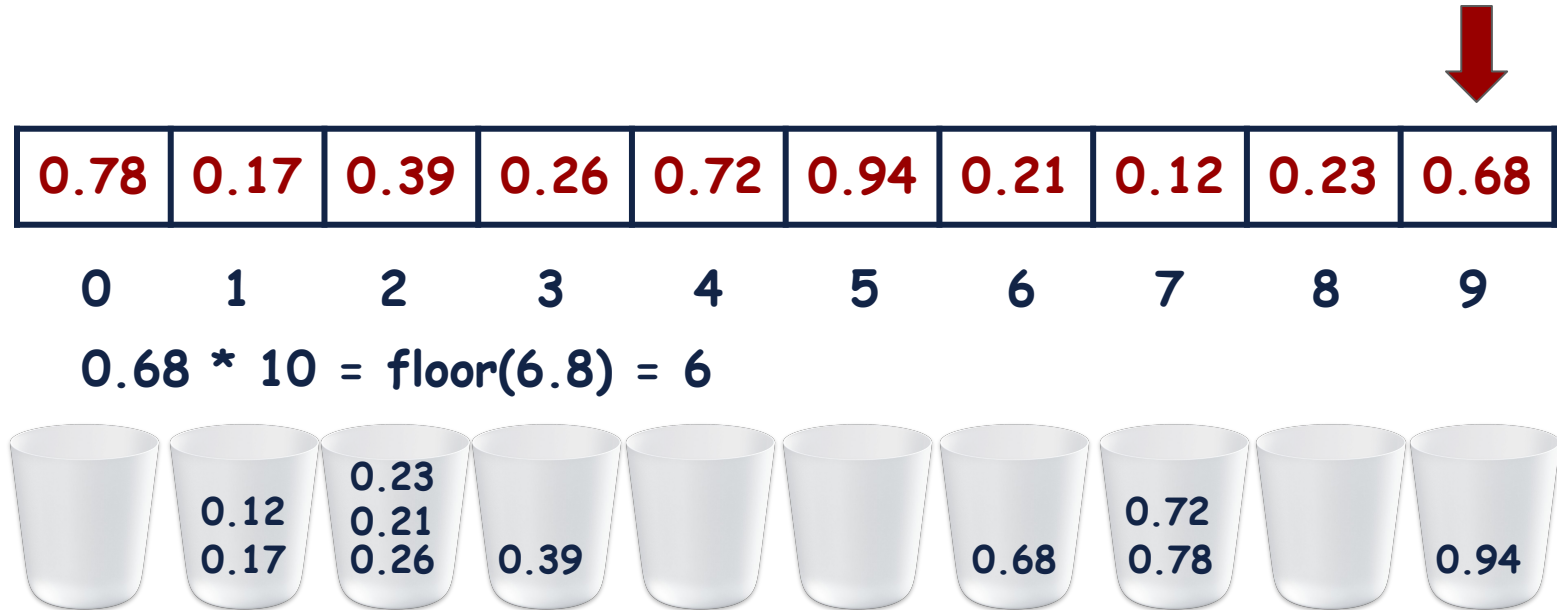
Bucket Sort

Step 3: Repeat the same process for all the elements.



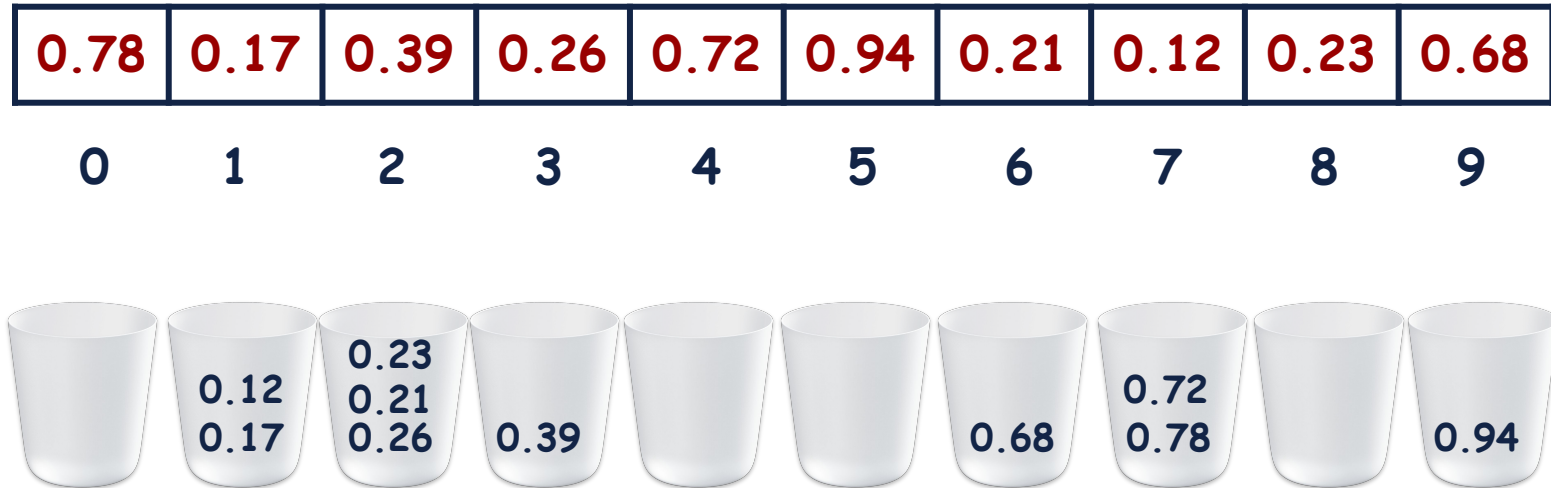
Bucket Sort

Step 3: Repeat the same process for all the elements.



Bucket Sort

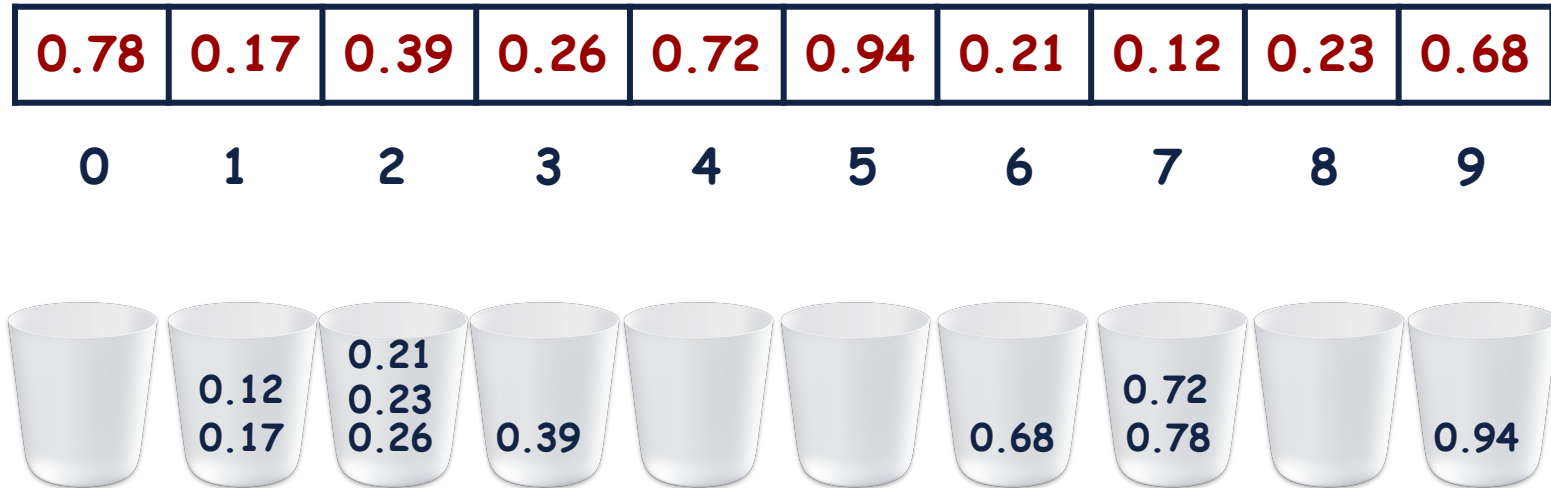
Step 4: Sort all the buckets using any sorting algorithm. In literature, mostly insertion sort is used.



Now, the joke is again on them.

Bucket Sort

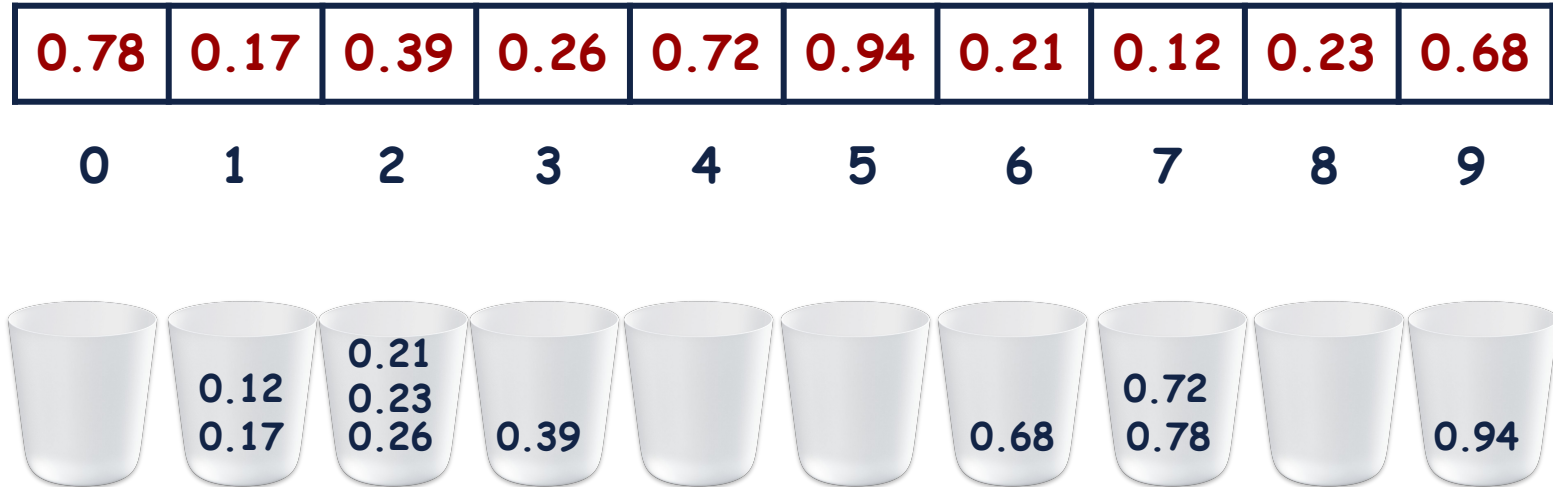
Step 4: Sort all the buckets using any sorting algorithm. In literature, mostly insertion sort is used.



Now, the joke is again on them.

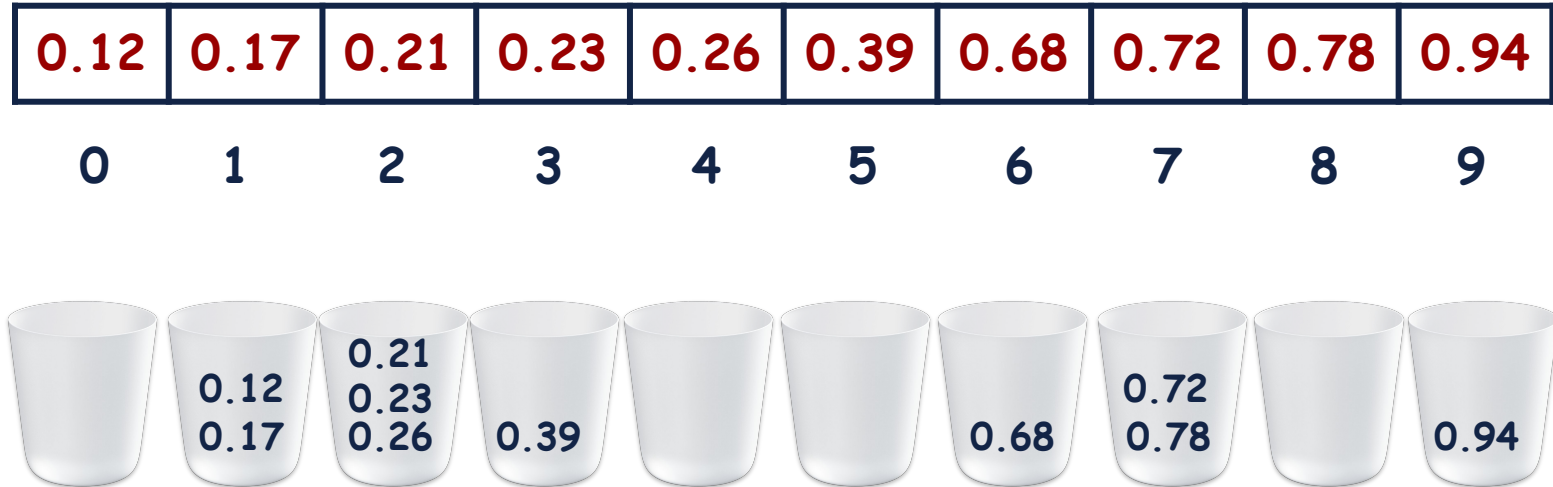
Bucket Sort

Step 5: Now, concatenate the buckets and place the elements back in the input array and the data will be sorted.



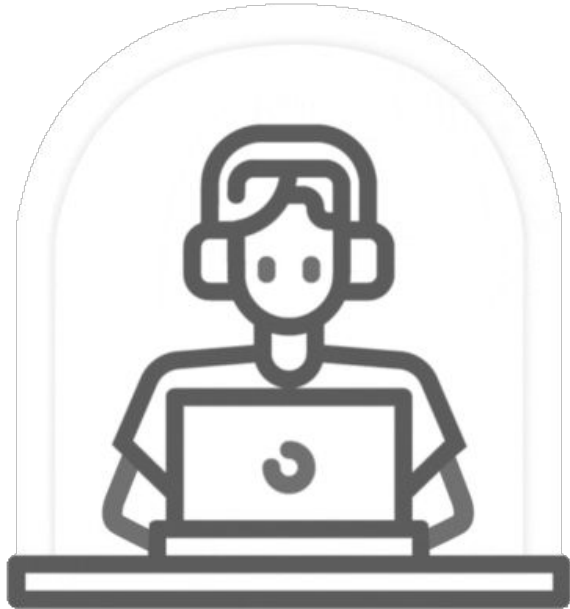
Bucket Sort

Step 5: Now, concatenate the buckets and place the elements back in the input array and the data will be sorted.



|| Bucket Sort: Implementation

Now, Let's implement the Bucket Sort Algorithm.



FOCUS · MODE

Bucket Sort: Implementation

```
main()
{
    vector<float> arr = {0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68};
    bucketSort(arr);
    for(int x = 0; x < arr.size(); x++)
    {
        cout << arr[x] << " ";
    }
}
```

Bucket Sort: Implementation

```
void bucketSort(vector<float> &arr) {  
    vector<vector<float>> bucket(arr.size());  
    for(int x = 0; x < arr.size(); x++)  
    {  
        bucket[int(arr[x] * arr.size())].push_back(arr[x]);  
    }  
    for(int x = 0; x < bucket.size(); x++)  
    {  
        sort(bucket[x].begin(), bucket[x].end());  
    }  
    int index = 0;  
    for(int x = 0; x < bucket.size(); x++)  
    {  
        for(int y = 0; y < bucket[x].size(); y++)  
        {  
            arr[index] = bucket[x][y];  
            index++;  
        }  
    }  
}
```

Bucket Sort: Time Complexity

```
void bucketSort(vector<float> &arr) {  
    vector<vector<float>> bucket(arr.size());  
    for(int x = 0; x < arr.size(); x++)  
    {  
        bucket[int(arr[x] * arr.size())].push_back(arr[x]);  
    }  
    for(int x = 0; x < bucket.size(); x++)  
    {  
        sort(bucket[x].begin(), bucket[x].end());  
    }  
    int index = 0;  
    for(int x = 0; x < bucket.size(); x++)  
    {  
        for(int y = 0; y < bucket[x].size(); y++)  
        {  
            arr[index] = bucket[x][y];  
            index++;  
        }  
    }  
}
```

|| Bucket Sort: Time Complexity

Time Complexity of Bucket Sort depends on the distribution of elements in the buckets.

Bucket Sort: Time Complexity

Time Complexity of Bucket Sort depends on the distribution of elements in the buckets.

If the elements are evenly distributed then the time complexity is $O(n)$.

If the elements are not evenly distributed then the time complexity can reach upto $O(n^2)$.

Non-Comparison Sorting Algorithms

Non-Comparison Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Counting Sort	$O(N + K)$	$O(N + K)$	$O(N + K)$	$O(N+K)$
Radix Sort	$O(D*(N+K))$	$O(D*(N+K))$	$O(D*(N+K))$	$O(N+K)$
Bucket Sort	$O(N)$	$O(N)$	$O(N^2)$	$O(N)$

Non-Comparison Sorting Algorithms

Bhai Radix sort mis ka hi sab sai acha samjaya howa hi

<https://www.geeksforgeeks.org/bucket-sort-2/>

Sorting Algorithm	In-Place	Stable
Counting Sort	No	Yes
Radix Sort	No	Yes
Bucket Sort	No	Depends on the inner Sorting Algorithm used