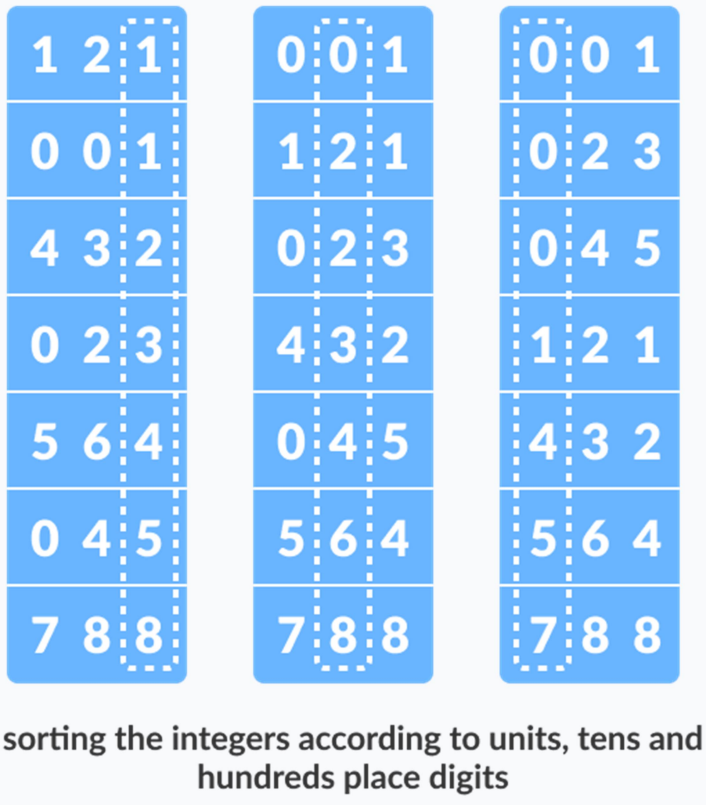
Radix Sort Algorithm

Radix sort is a sorting technique 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.
Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. 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.



sorting the integers according to units, tens and hundreds place digits

Working of Radix Sort

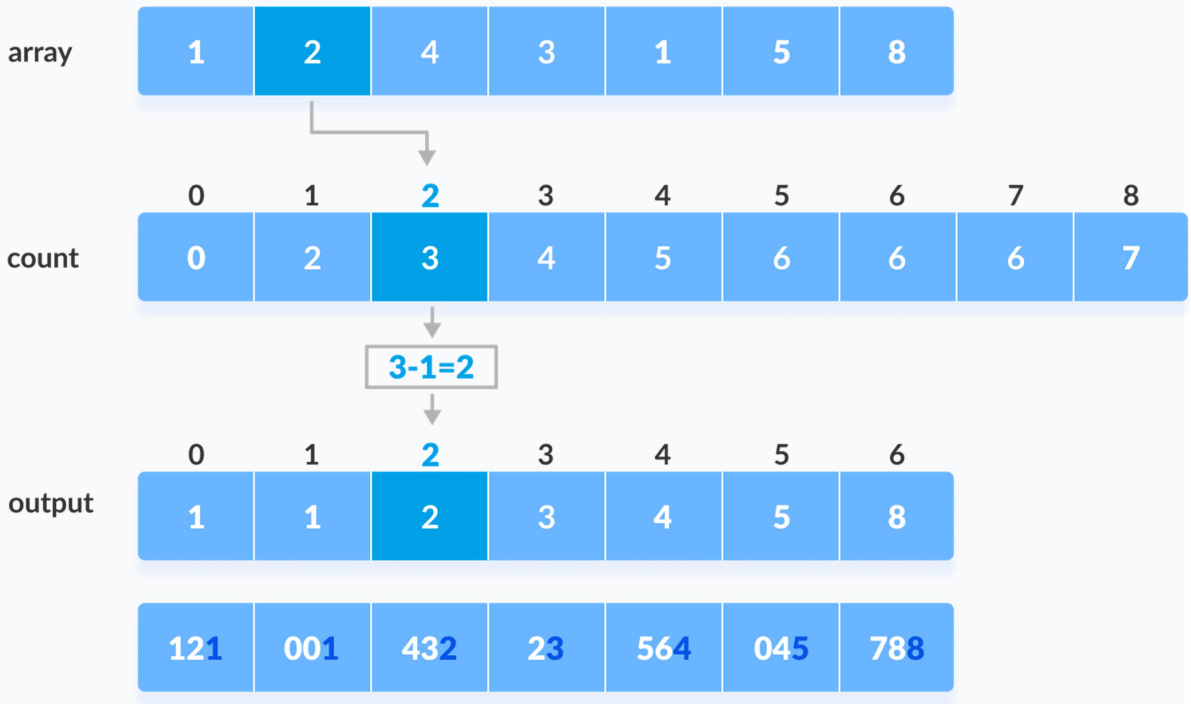
How Radix Sort Works?

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`).

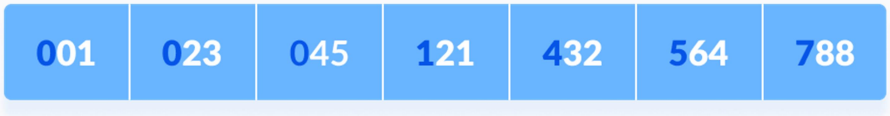


Using counting sort to sort elements based on unit place

3. Now, sort the elements based on digits at tens place.

Sort elements based on tens place

4. Finally, sort the elements based on the digits at hundreds place.



Sort elements based on hundreds place

Radix Sort Algorithm

```
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 `O(d(n+k))`.
Here, `d` is the number cycle and `O(n+k)` is the time complexity of counting sort.
Thus, radix sort has linear time complexity which is better than `O(nlog n)` of comparative sorting algorithms.
If we take very large digit numbers or the number of other bases like 32-bit and 64-bit numbers then it can perform in linear time however the intermediate sort takes large space.

Bucket Sort Algorithm

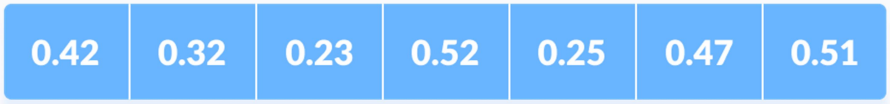
Bucket Sort is a sorting technique that sorts the elements by first dividing the elements into several groups called **buckets**. The elements inside each **bucket** are sorted using any of the suitable sorting algorithms or recursively calling the same algorithm.
Several buckets are created. Each bucket is filled with a specific range of elements. The elements inside the bucket are sorted using any other algorithm. Finally, the elements of the bucket are gathered to get the sorted array.

The process of bucket sort can be understood as **a scatter-gather** approach. The elements are first scattered into buckets then the elements of buckets are sorted. Finally, the elements are gathered in order.

Working of Bucket Sort

How Bucket Sort Works?

1. Suppose, the input array is:



Input array

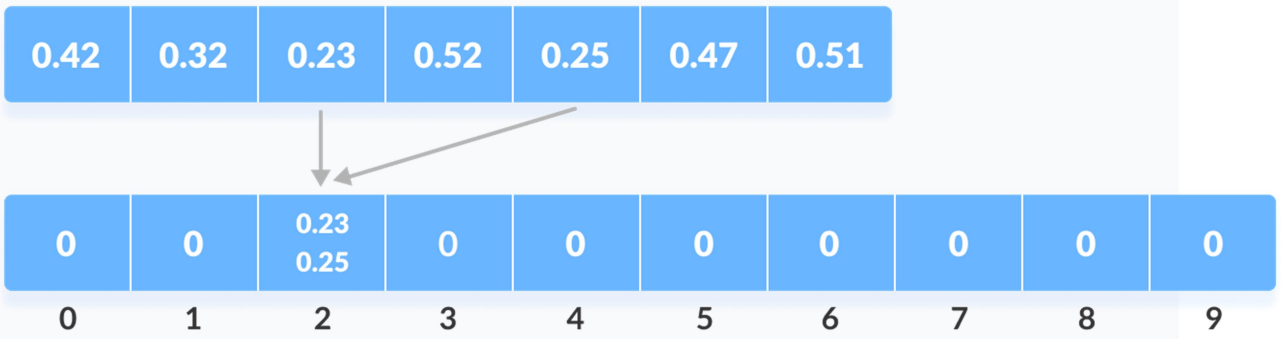
Create an array of size 10. Each slot of this array is used as a bucket for storing elements.

Array in which each position is a bucket

2. Insert elements into the buckets from the array. The elements are inserted according to the range of the bucket.

In our example code, we have buckets each of ranges from 0 to 1, 1 to 2, 2 to 3,..... (n-1) to n.

Suppose, an input element is `.23` is taken. It is multiplied by `size = 10` (ie. `.23*10=2.3`). Then, it is converted into an integer (ie. `2.3≈2`). Finally, .23 is inserted into **bucket-2**.

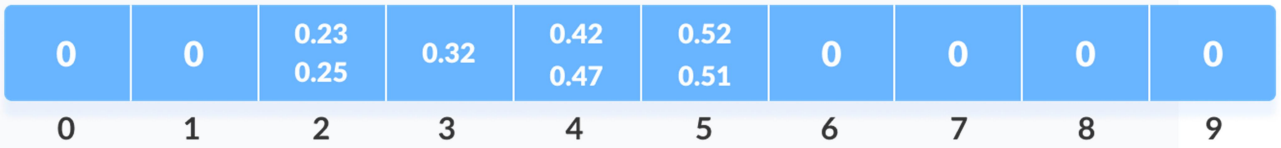


Insert elements into the buckets from the array

Similarly, .25 is also inserted into the same bucket. Everytime, the floor value of the floating point number is taken.

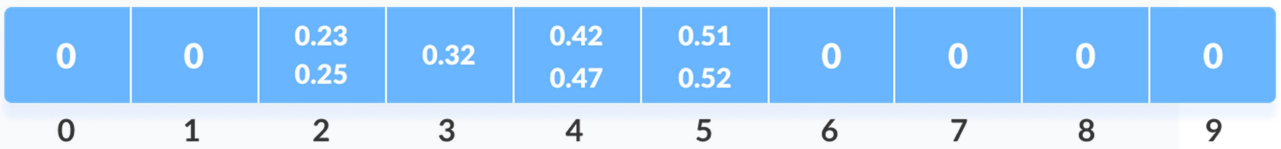
If we take integer numbers as input, we have to divide it by the interval (10 here) to get the floor value.

Similarly, other elements are inserted into their respective buckets.



Insert all the elements into the buckets from the array

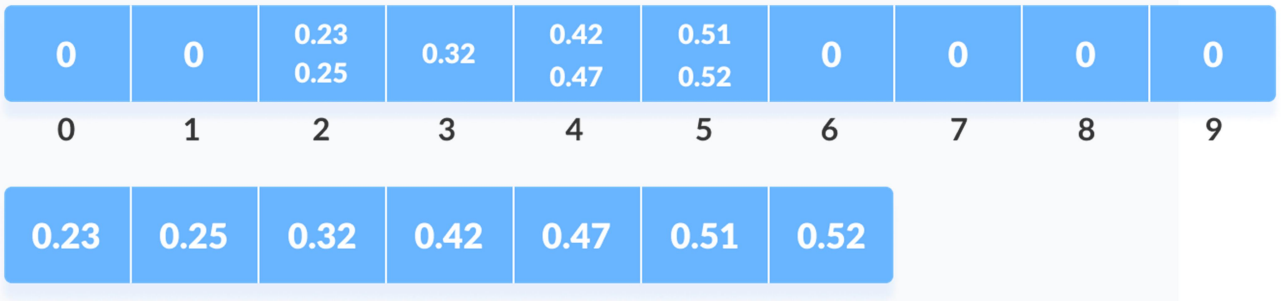
3. The elements of each bucket are sorted using any of the stable sorting algorithms. Here, we have used quicksort (inbuilt function).



Sort the elements in each bucket

4. The elements from each bucket are gathered.

It is done by iterating through the bucket and inserting an individual element into the original array in each cycle. The element from the bucket is erased once it is copied into the original array.



Gather elements from each bucket

Bucket Sort Algorithm

```
bucketSort()
  create N buckets each of which can hold a range of values
  for all the buckets
    initialize each bucket with 0 values
  for all the buckets
    put elements into buckets matching the range
  for all the buckets
    sort elements in each bucket
  gather elements from each bucket
end bucketSort
```

Complexity

- **Worst Case Complexity:** $O(n^2)$
  When there are elements of close range in the array, they are likely to be placed in the same bucket. This may result in some buckets having more number of elements than others.
  It makes the complexity depend on the sorting algorithm used to sort the elements of the bucket.
  The complexity becomes even worse when the elements are in reverse order. If insertion sort is used to sort elements of the bucket, then the time complexity becomes $O(n^2)$.
- **Best Case Complexity:** $O(n+k)$
  It occurs when the elements are uniformly distributed in the buckets with a nearly equal number of elements in each bucket.
  The complexity becomes even better if the elements inside the buckets are already sorted.
  If insertion sort is used to sort elements of a bucket then the overall complexity in the best case will be linear ie. $O(n+k)$. $O(n)$ is the complexity for making the buckets and $O(k)$ is the complexity for sorting the elements of the bucket using algorithms having linear time complexity at the best case.
- **Average Case Complexity:** $O(n)$
  It occurs when the elements are distributed randomly in the array. Even if the elements are not distributed uniformly, bucket sort runs in linear time. It holds true until the sum of the squares of the bucket sizes is linear in the total number of elements.