Bucket sort

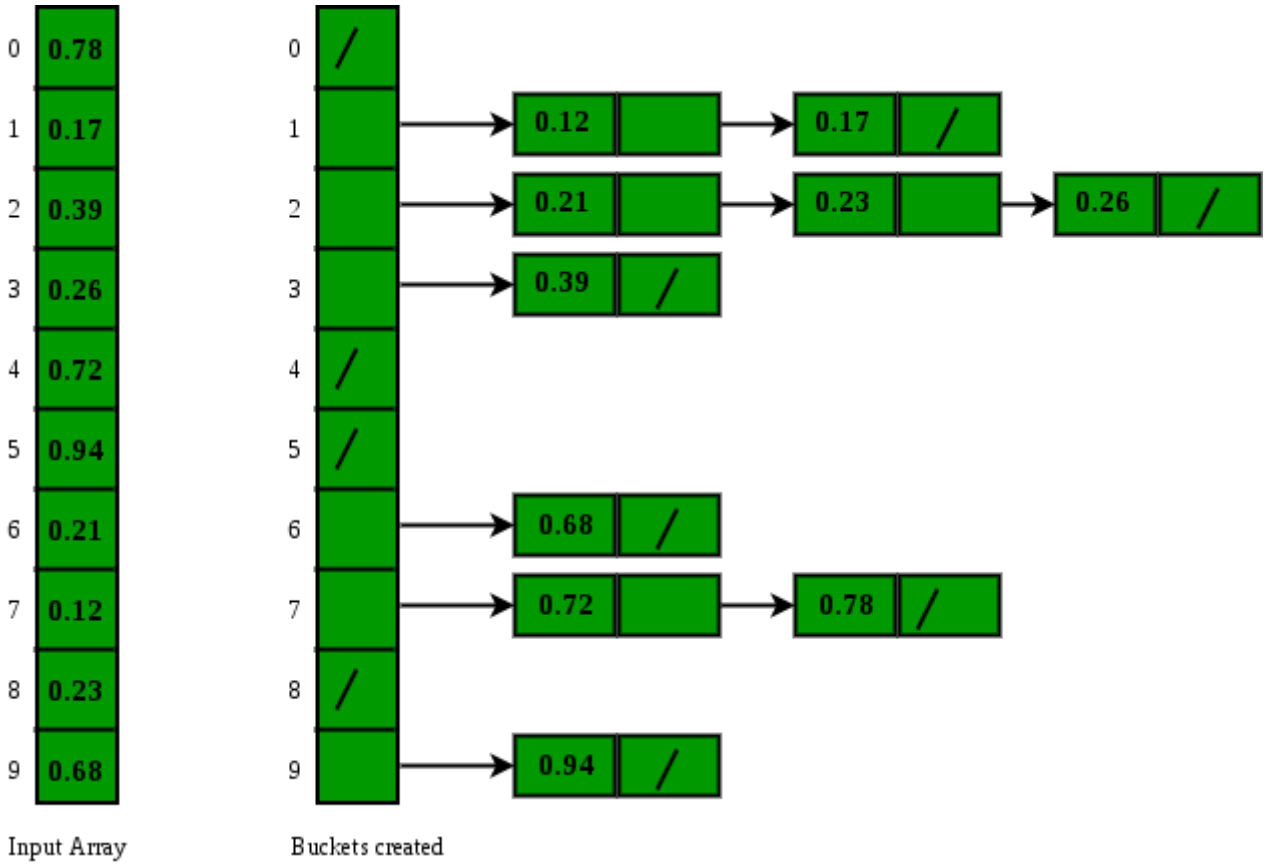
Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem. Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval [0,1). Bucket sort divides the interval [0,1) into n equal-sized sub-intervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0,1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.

Algorithm:

```
BUCKET-SORT(A)
1 let B[0 .. n-1] be a new array
2 n = A.length
3 for i = 0 to n - 1
4   make B[i] an empty list                    ...(step 1)
5 for i = 1 to n                               ...(step 2)
6   insert A[i] into list B[ floor(n*A[i])]
7 for i = 0 to n - 1
8   sort list B[i] with insertion sort / counting sort          ...(step 3)
9 concatenate the lists B[0], B[1], ... B[n - 1] together in order  ...(step 4)
```



Time Complexity

If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket. Step 4 also takes O(n) time as there will be n items in all buckets.The difficult step is to analyze step 3.