## horizontal line



**Case Study - Bucket Sort**

21.04.2017

**─**

By-

Rishabh Modi(RA1511003010498)

Utkarsh Sharma(RA1511003010495)

TP - 601

# Table Of Contents

1. Problem Statement 2
2. Problem Solution 2
3. Bucket Sort 3
4. Algorithm And Steps 4
5. Explanation 5
6. Divide And Conquer 5
7. Source Code Solution 6
8. Output 8
9. Diagrammatic Explanation 9
10. Complexity 13
11. Advantages 14
12. Disadvantages 14
13. Conclusion 14
14. References 16

# Problem Statement

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?

# 

# Problem Solution

A simple way is to apply a comparison based sorting algorithm. 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.

Can we sort the array in linear time? Counting sort can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.

The idea is to use bucket sort. Following is bucket algorithm.

# Bucket sort

**Bucket sort**, or **bin sort**, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort, and is a cousin of radix sort in the most to least significant digit flavor. Bucket sort is a generalization of pigeonhole sort. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm.

1. Set up an array of initially empty "buckets".
2. **Scatter**: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. **Gather**: Visit the buckets in order and put all elements back into the original array.

# Algorithm and steps

**bucketSort(arr[], n)**

1) Create n empty buckets (Or lists).

2) Do following for every array element arr[i].

a) Insert arr[i] into bucket[n\*array[i]]

3) Sort individual buckets using insertion sort.

4) Concatenate all sorted buckets.

**Algorithm:**

**function** bucketSort(array, n) **is**

buckets ← new array of n empty lists

**for** i = 0 **to** (length(array)-1) **do**

insert *array[i]* into buckets[msbits(array[i], k)]

**for** i = 0 **to** n - 1 **do**

nextSort(buckets[i]);

**return** the concatenation of buckets[0], ...., buckets[n-1]

### **Explanation**

At first, we define the value of m, which means that all the elements we will introduce for the array will have to be lower than m. Next, we make buckets for the size of m, and we make them null and in the end, we add the elements to the proper buckets. It is not required another type of sorting algorithm, in this example we use bucket sort only, as we use a bucket for each element of the array, this might seem familiar with radix sort.

# Divide and conquer

This is the basic technique behind Bucket Sort.

The name "divide and conquer" is sometimes applied to algorithms that reduce each problem to only one sub-problem, such as the [binary search](https://en.wikipedia.org/wiki/Binary_search" \o "Binary search) algorithm for finding a record in a sorted list (or its analog in [numerical computing](https://en.wikipedia.org/wiki/Numerical_algorithm" \o "Numerical algorithm), the [bisection algorithm](https://en.wikipedia.org/wiki/Bisection_algorithm" \o "Bisection algorithm) for [root finding](https://en.wikipedia.org/wiki/Root-finding_algorithm" \o "Root-finding algorithm)).These algorithms can be implemented more efficiently than general divide-and-conquer algorithms; in particular, if they use [tail recursion](https://en.wikipedia.org/wiki/Tail_recursion" \o "Tail recursion), they can be converted into simple [loops](https://en.wikipedia.org/wiki/Loop_(computing)" \o "Loop (computing)). Under this broad definition, however, every algorithm that uses recursion or loops could be regarded as a "divide and conquer algorithm". Therefore, some authors consider that the name "divide and conquer" should be used only when each problem may generate two or more sub problems.The name **decrease and conquer** has been proposed instead for the single-problem class.

An important application of divide and conquer is in optimization, where if the search space is reduced ("pruned") by a constant factor at each step, the overall algorithm has the same asymptotic complexity as the pruning step, with the constant depending on the pruning factor (by summing the [geometric series](https://en.wikipedia.org/wiki/Geometric_series" \o "Geometric series)); this is known as [prune and search](https://en.wikipedia.org/wiki/Prune_and_search" \o "Prune and search).

# Source Code Solution

// C++ program to sort an array using bucket sort

#include <iostream>

#include <algorithm>

#include <vector>

**using** **namespace** std;

// Function to sort arr[] of size n using bucket sort

**void** bucketSort(**float** arr[], **int** n)

{

// 1) Create n empty buckets

vector<**float**> b[n];

// 2) Put array elements in different buckets

**for** (**int** i=0; i<n; i++)

{

**int** bi = n\*arr[i]; // Index in bucket

b[bi].push\_back(arr[i]);

}

// 3) Sort individual buckets

**for** (**int** i=0; i<n; i++)

sort(b[i].begin(), b[i].end());

// 4) Concatenate all buckets into arr[]

**int** index = 0;

**for** (**int** i = 0; i < n; i++)

**for** (**int** j = 0; j < b[i].size(); j++)

arr[index++] = b[i][j];

}

/\* Driver program to test above function \*/

**int** main()

{

**float** arr[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};

**int** n = **sizeof**(arr)/**sizeof**(arr[0]);

bucketSort(arr, n);

cout << "Sorted array is \n";

**for** (**int** i=0; i<n; i++)

cout << arr[i] << " ";

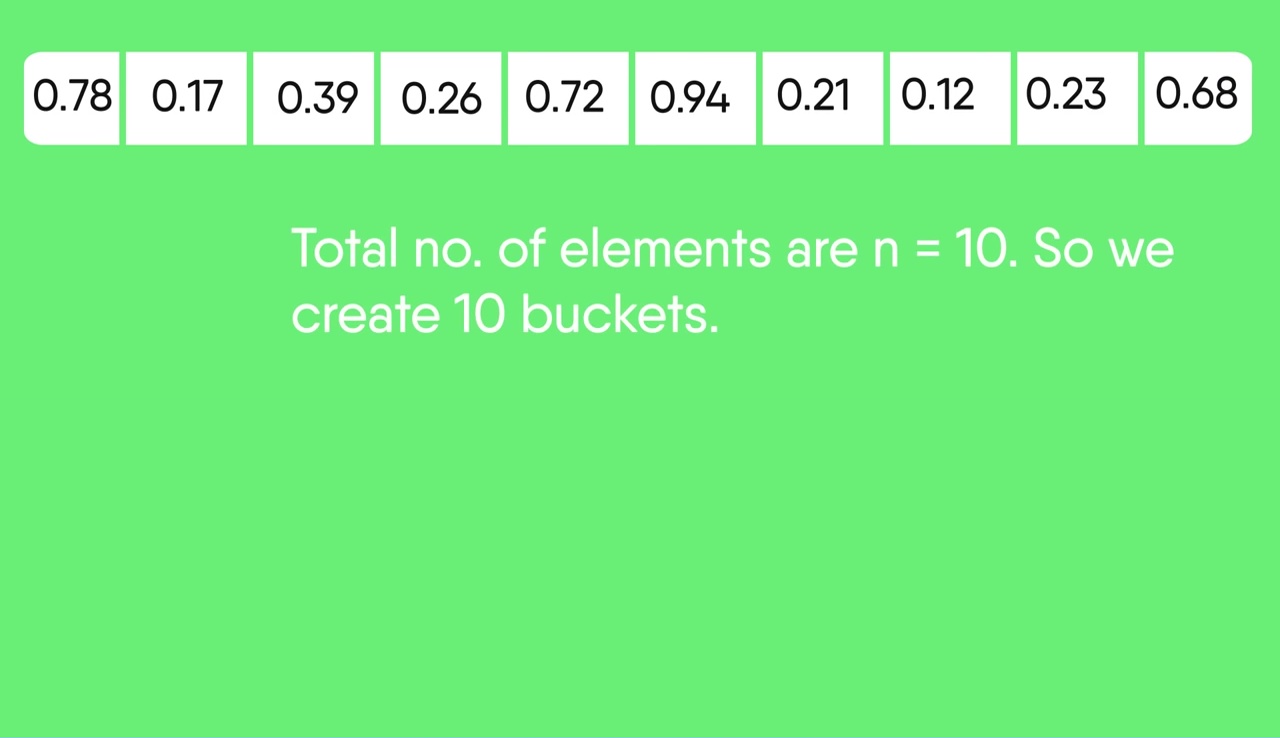
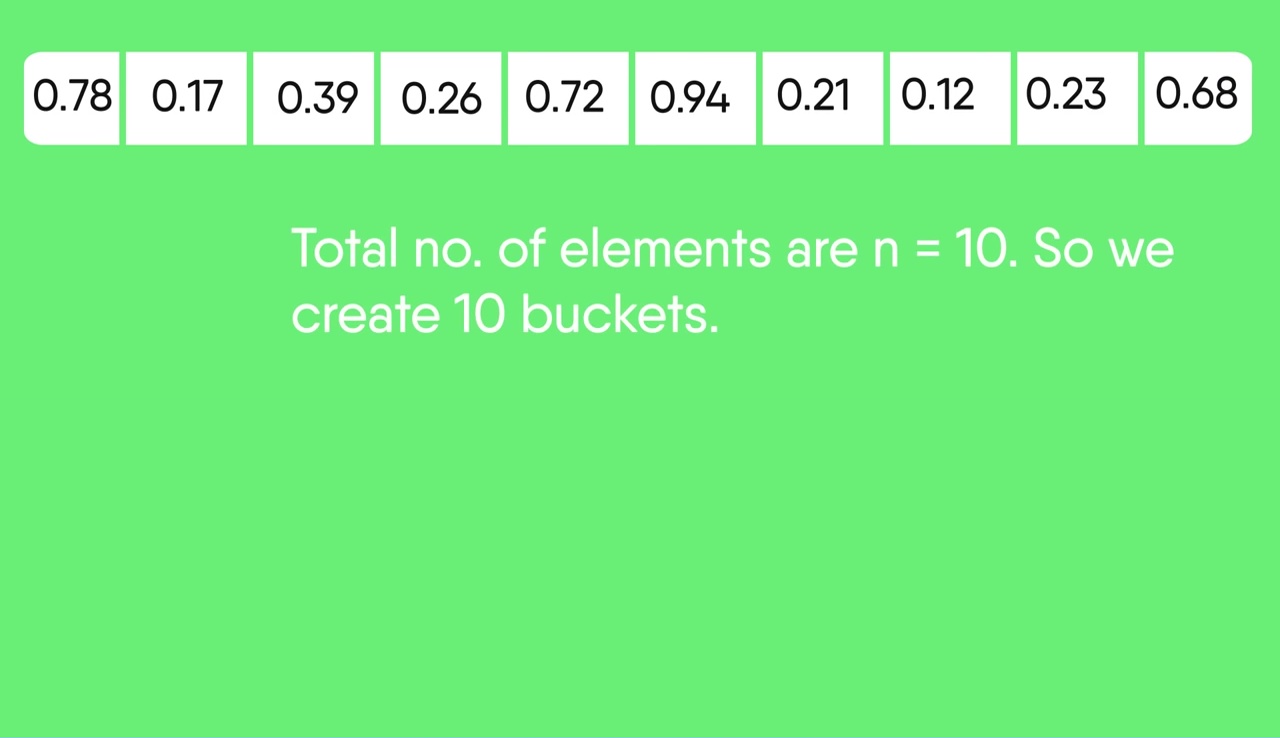
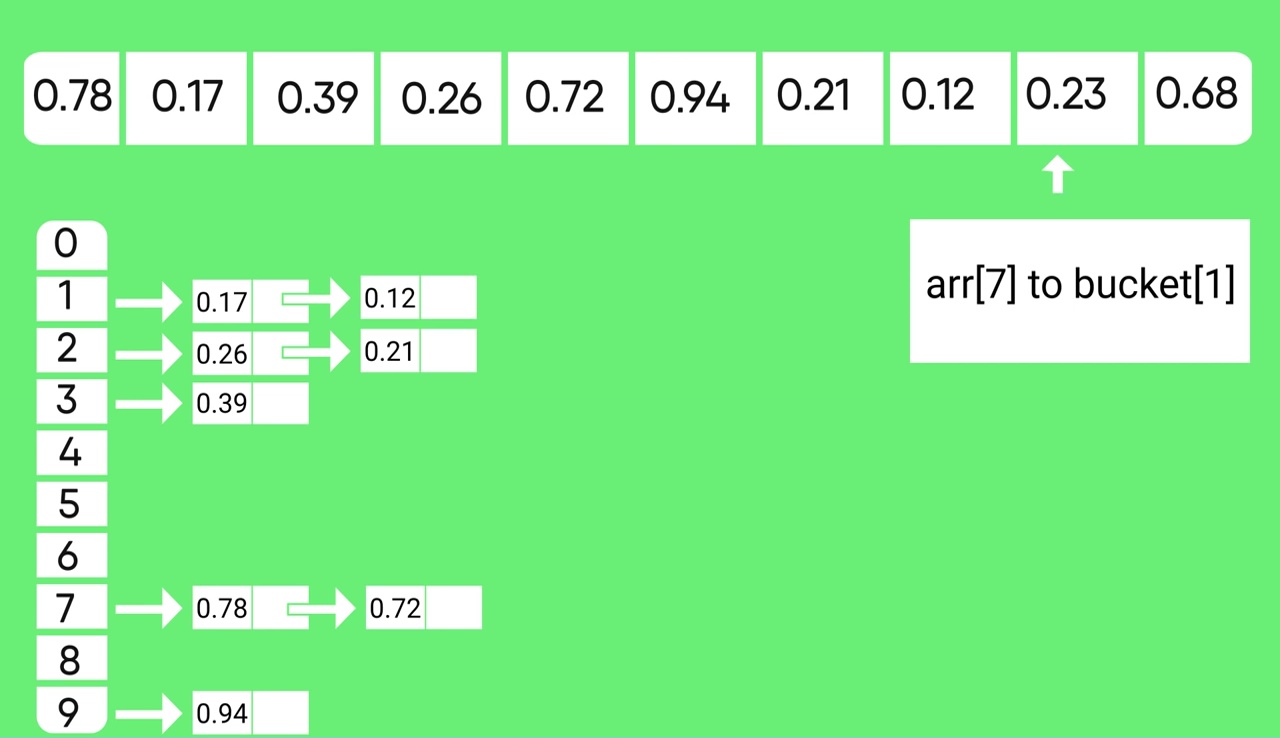
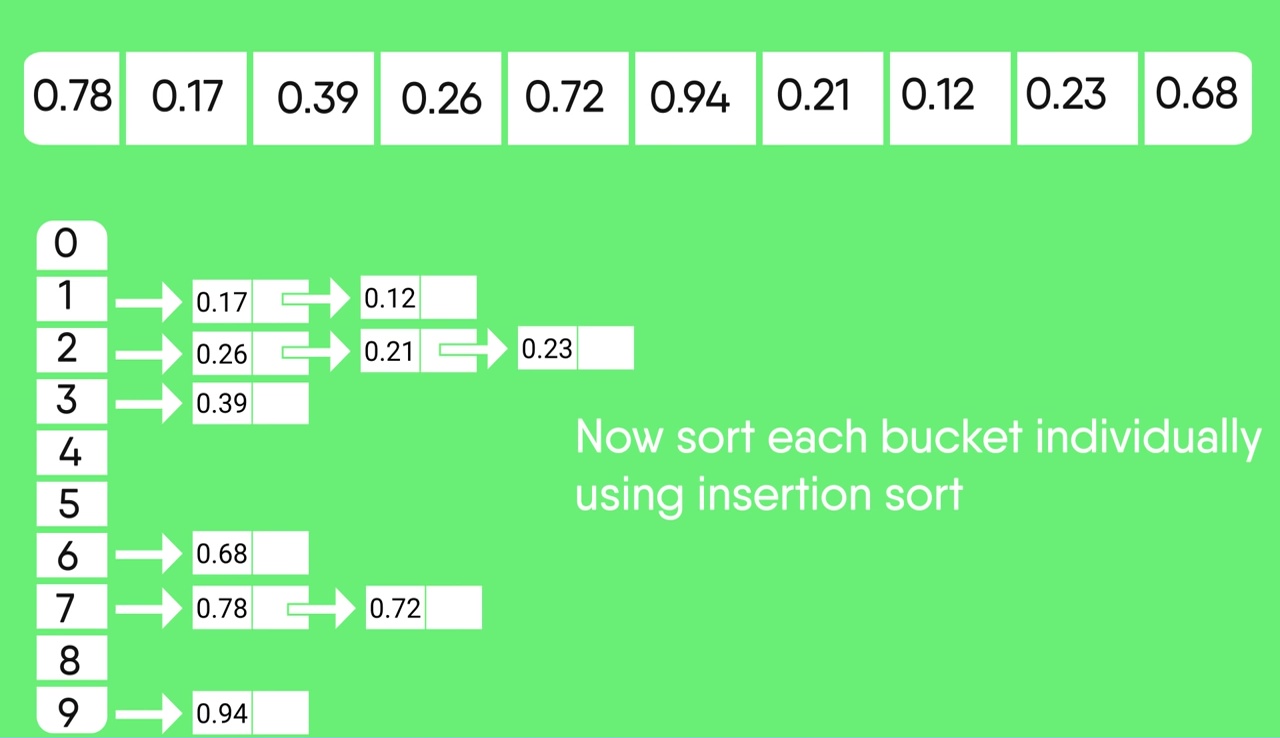
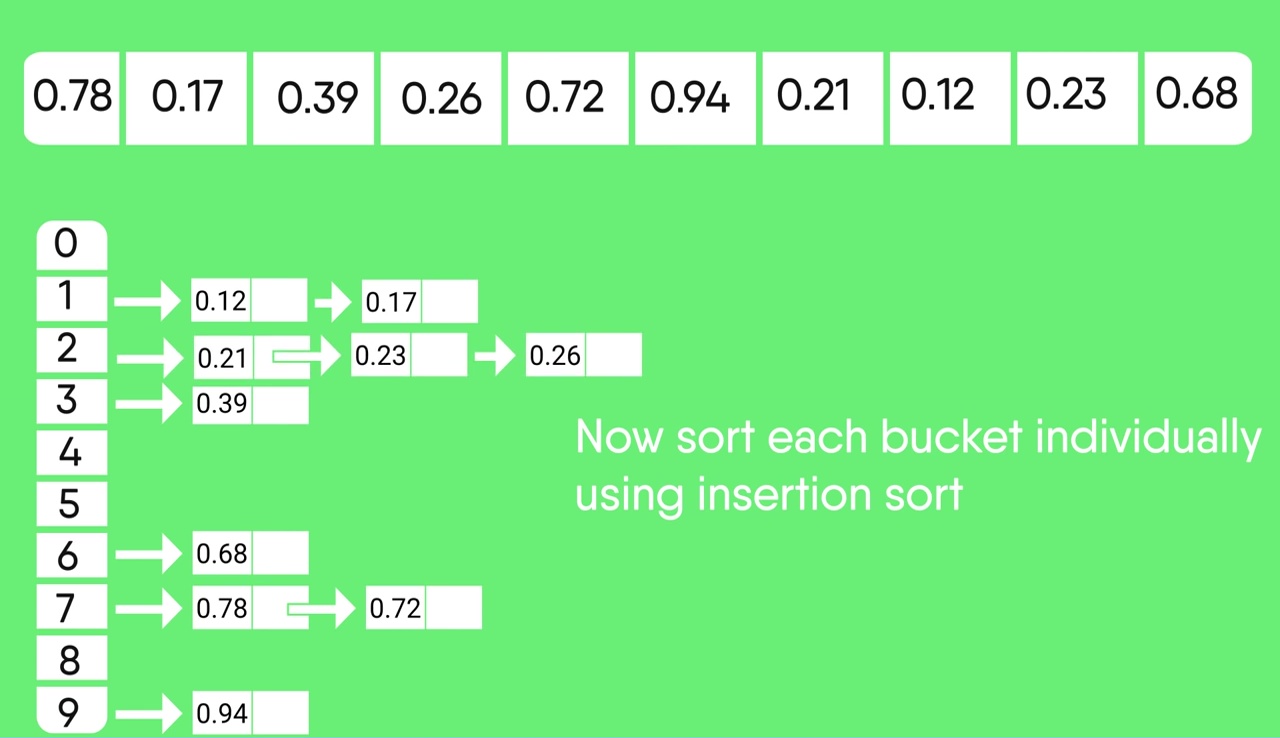
**return** 0;

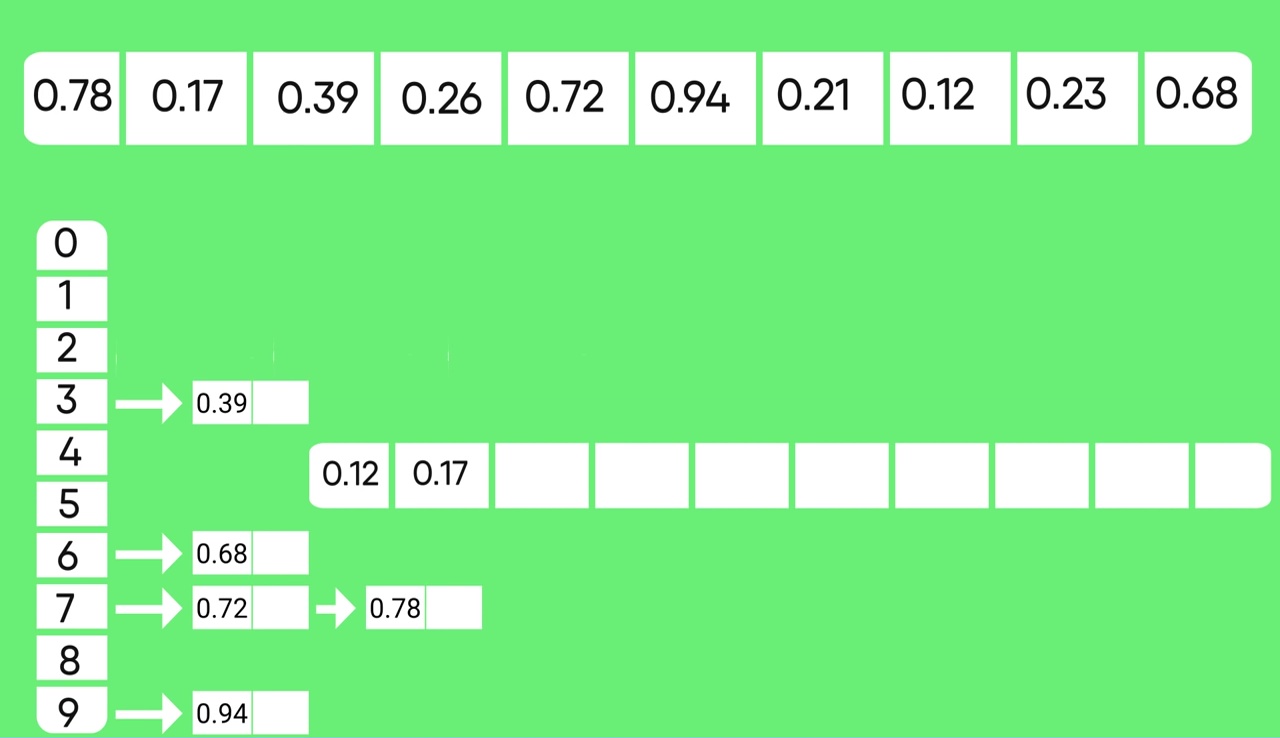
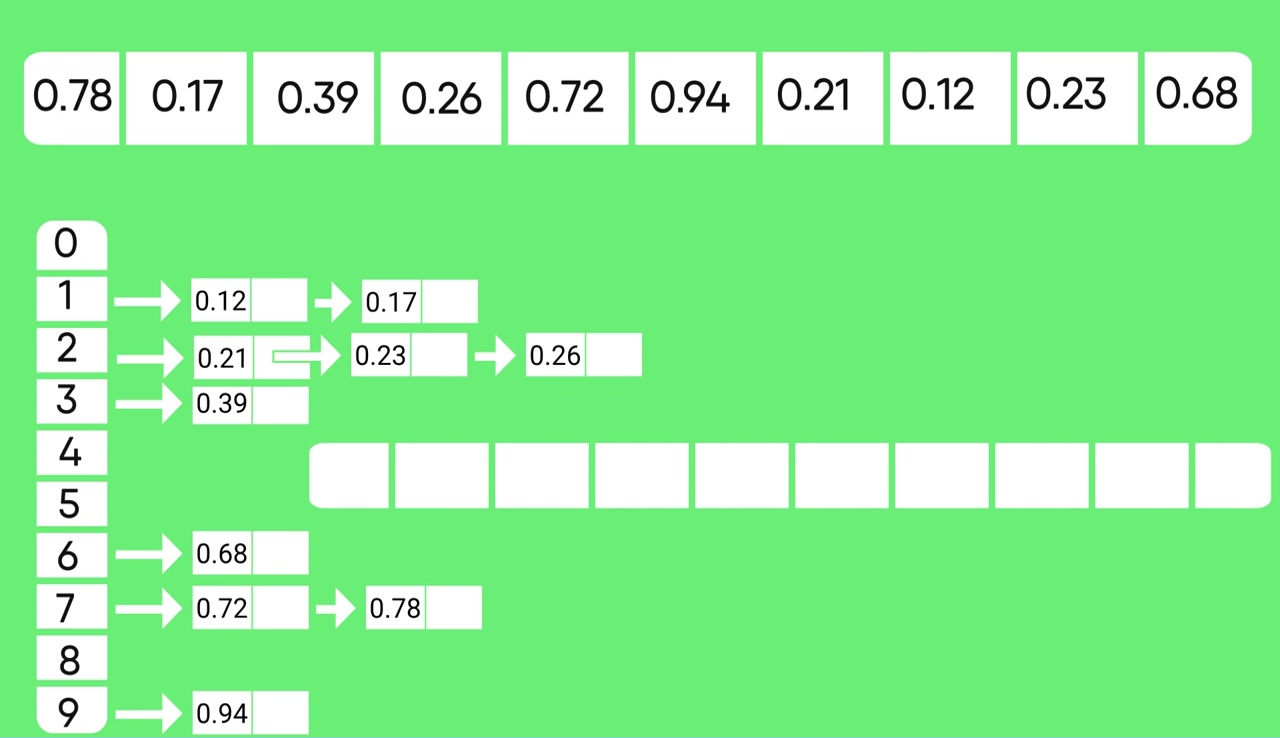
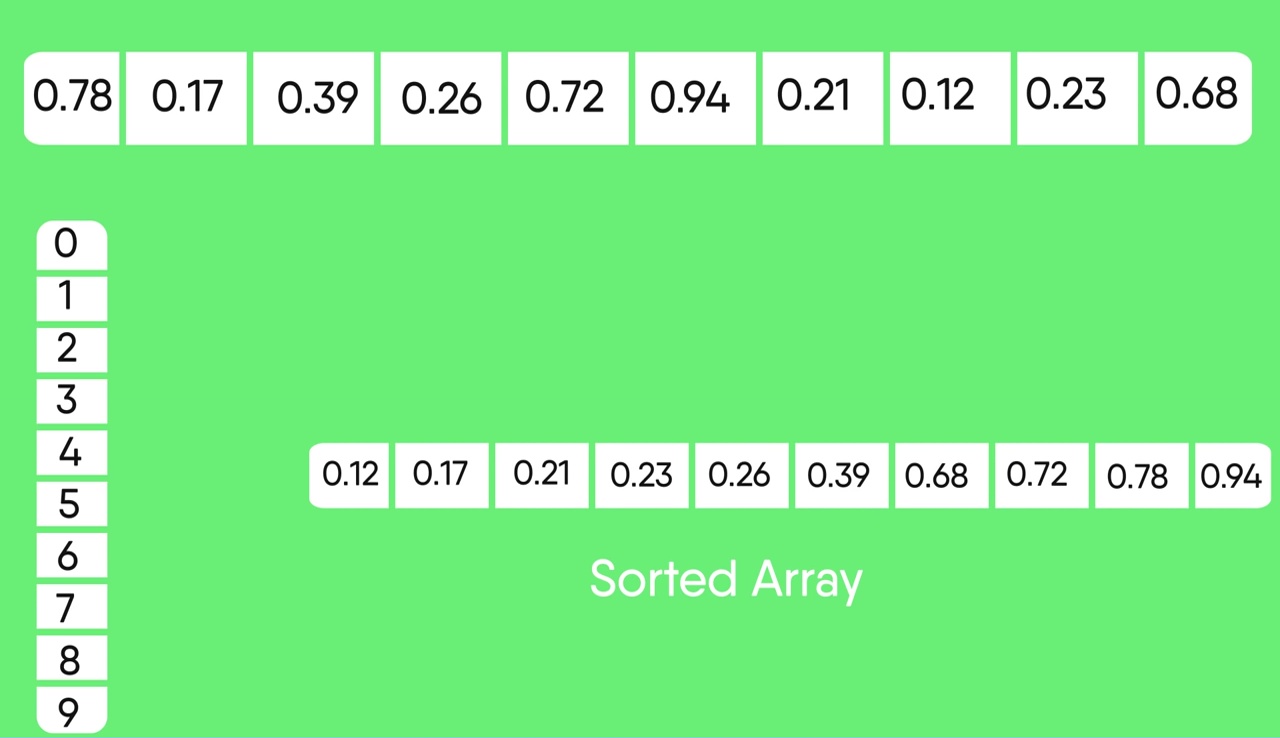
}

Output

Sorted array is

0.1234 0.3434 0.565 0.656 0.665 0.897

Diagrammatic Explanation  
.  
  
  
  


### **Complexity**

Bucket sort can be seen as a generalization of counting sort, in fact, if each bucket has size 1 then bucket sort degenerates to counting sort. The variable bucket size of bucket sort allows it to use O(n) memory instead of O(M) memory, where M is the number of distinct values; in exchange, it gives up counting sort’s O(n + M) worst-case behavior.

The time complexity of bucket sort is: O(n + m) where: m is the range input values, n is the total number of values in the array. Bucket sort beats all other sorting routines in time complexity. It should only be used when the range of input values is small compared with the number of values. In other words, occasions when there are a lot of repeated values in the input. Bucket sort works by counting the number of instances of each input value throughout the array. It then reconstructs the array from this auxiliary data. This implementation has a configurable input range, and will use the least amount of memory possible.

### **Advantages**

* the user knows the range of the elements.
* time complexity is good compared to other algorithms.

### **Disadvantages**

* we are limited to having to know the greatest element.
* extra memory is required.

### **Conclusion:**

If we know the range of the elements, the bucket sort is quite good, with a time complexity of only O(n+k). At the same time, this is a major drawback, since it is a must to know the greatest elements. It is a distribution sort and a cousin of radix sort.

Bucket sort can be seen as a generalization of counting sort; in fact, if each bucket has size 1 then bucket sort degenerates to counting sort. The variable bucket size of bucket sort allows it to use O(*n*) memory instead of O(*M*) memory, where *M* is the number of distinct values; in exchange, it gives up counting sort's O(*n* + *M*) worst-case behavior.

Bucket sort with two buckets is effectively a version of quicksort where the pivot value is always selected to be the middle value of the value range. While this choice is effective for uniformly distributed inputs, other means of choosing the pivot in quicksort such as randomly selected pivots make it more resistant to clustering in the input distribution.

The *n*-way mergesort algorithm also begins by distributing the list into *n* sub lists and sorting each one; however, the sub lists created by mergesort have overlapping value ranges and so cannot be recombined by simple concatenation as in bucket sort. Instead, they must be interleaved by a merge algorithm. However, this added expense is counterbalanced by the simpler scatter phase and the ability to ensure that each sub list is the same size, providing a good worst-case time bound.

Top-down radix sort can be seen as a special case of bucket sort where both the range of values and the number of buckets is constrained to be a power of two. Consequently, each bucket's size is also a power of two, and the procedure can be applied recursively. This approach can accelerate the scatter phase, since we only need to examine a prefix of the bit representation of each element to determine its bucket.

# References

* Corwin, E. and Logar, A. "Sorting in linear time — variations on the bucket sort". *Journal of Computing Sciences in Colleges*, 20, 1, pp.197–202. October 2004.
* [Thomas H. Cormen](https://en.wikipedia.org/wiki/Thomas_H._Cormen" \o "Thomas H. Cormen), [Charles E. Leiserson](https://en.wikipedia.org/wiki/Charles_E._Leiserson" \o "Charles E. Leiserson), [Ronald L. Rivest](https://en.wikipedia.org/wiki/Ronald_L._Rivest" \o "Ronald L. Rivest), and [Clifford Stein](https://en.wikipedia.org/wiki/Clifford_Stein" \o "Clifford Stein). *[Introduction to Algorithms](https://en.wikipedia.org/wiki/Introduction_to_Algorithms" \o "Introduction to Algorithms)*, Second Edition. MIT Press and McGraw-Hill, 2001. [ISBN 0-262-03293-7](https://en.wikipedia.org/wiki/Special:BookSources/0262032937). Section 8.4: Bucket sort, pp.174–177.
* [NIST's Dictionary of Algorithms and Data Structures: histogram sort](https://xlinux.nist.gov/dads/HTML/histogramSort.html)
* **[J](https://en.wikipedia.org/wiki/Bucket_sort" \l "cite_ref-5)** <http://www.rrsd.com/psort/cuj/cuj.htm>
* **www.wiki**Paul E. Black ["Postman's Sort"](https://xlinux.nist.gov/dads/HTML/postmansort.html) from [Dictionary of Algorithms and Data Structures](https://en.wikipedia.org/wiki/Dictionary_of_Algorithms_and_Data_Structures" \o "Dictionary of Algorithms and Data Structures) at [NIST](https://en.wikipedia.org/wiki/National_Institute_of_Standards_and_Technology" \o "National Institute of Standards and Technology).
* Robert Ramey ['"The Postman's Sort"](http://www.rrsd.com/software_development/postmans_sort/cuj/cuj.htm) *C Users Journal* Aug. 1992
* [NIST's Dictionary of Algorithms and Data Structures: bucket sort](https://xlinux.nist.gov/dads/HTML/bucketsort.html)
* **http://www.exforsys.com/tutorials/c-algorithms/bucket-sort.html**
* **http://www.geeksforgeeks.org/bucket-sort-2/**
* **https://en.wikipedia.org/wiki/Bucket\_sort**