Bucket Sort

Varun Kumar

July 5, 2025

1. Logic

Bucket Sort distributes the elements into a number of buckets. Each bucket is then sorted individually (often using Insertion Sort or another algorithm), and finally, all the buckets are concatenated to form the sorted array.

Key Idea

Distribute elements into several buckets, sort each bucket, and concatenate them to get the final sorted array.

2. Number of Comparisons and Shifts

Let n be the number of elements.

Worst Case (All in One Bucket)

- Comparisons: $O(n^2)$ (if sorted using Insertion Sort inside each bucket)
- Shifts: Depends on internal sort

Best Case (Uniform Distribution)

- Comparisons: $O(n)$
- Shifts: $O(n)$

3. Optimal Behavior

Bucket sort works best when:

- Input is uniformly distributed over a known range.
- Number of buckets is well chosen.

4. Pseudocode

```
function bucketSort(arr):
    n = length(arr)
    create n empty buckets

    for each element in arr:
        insert it into the appropriate bucket

    for each bucket:
        sort the bucket

    concatenate all buckets into arr
```

Stability Note

Bucket Sort is not inherently stable. Stability depends on the sorting algorithm used inside each bucket.

5. Example Walkthrough

Given: `[0.42, 0.32, 0.23, 0.52, 0.25, 0.47]`

Step 1: Bucket Distribution

- Bucket 0: [0.23, 0.25]
- Bucket 1: [0.32]
- Bucket 2: []
- Bucket 3: [0.42, 0.47]

- Bucket 4: [0.52]

Step 2: Sort Each Bucket

- Bucket 0: [0.23, 0.25]
- Bucket 3: [0.42, 0.47]

Step 3: Concatenate

`[0.23, 0.25, 0.32, 0.42, 0.47, 0.52]`

6. Python Code with Explanation

```
def bucket_sort(arr):
    n = len(arr)
    if n == 0:
        return arr

    # Create n empty buckets
    buckets = [[] for _ in range(n)]

    # Insert elements into buckets
    for num in arr:
        index = int(n * num)
        buckets[index].append(num)

    # Sort each bucket
    for bucket in buckets:
        bucket.sort()

    # Concatenate all buckets
    sorted_arr = []
    for bucket in buckets:
        sorted_arr.extend(bucket)

    return sorted_arr
```

7. Is Bucket Sort Stable?

Stability of Bucket Sort

Bucket Sort is **not inherently stable**. Its stability depends on the sorting algorithm used inside each bucket.

Explanation: If a *stable* sorting algorithm (like Insertion Sort) is used within the buckets, then Bucket Sort will be stable. However, if an *unstable* sort is used, then the overall algorithm will also be unstable.

8. How Insertion Sort Ensures Stability

Insertion Sort compares elements using the `>` operator. When two elements are equal, it does **not swap** or move them unnecessarily. This means their original relative order is preserved.

For example, consider the list:

$$[(4, \mathrm{A}), (4, \mathrm{B}), (3, \mathrm{C})]$$

After sorting:

$$[(3, \mathrm{C}), (4, \mathrm{A}), (4, \mathrm{B})]$$

Elements with the same key `4` (`A` and `B`) remain in the same order as they appeared in the input.

Therefore, if Insertion Sort is used inside each bucket, the overall Bucket Sort becomes stable.

9. Is Bucket Sort In-place?

Answer: No, Bucket Sort is not in-place.

Explanation: Bucket Sort creates multiple auxiliary lists (buckets) to store elements during the sorting process. These additional data structures require extra memory proportional to the number of input elements and buckets.

- It uses $O(n + k)$ extra space, where k is the number of buckets.
- Since it does not sort within the original array without extra space, it is not considered in-place.

10. Can Bucket Sort Be Made In-place?

Answer: Theoretically possible, but not practical.

Explanation: Bucket Sort requires auxiliary space to create multiple buckets. To make it in-place, we would need to:

- Divide the input array into segments (buckets) without extra space.
- Sort each segment in-place.
- Carefully rearrange elements to simulate bucket behavior.

However, this approach:

- Complicates implementation.
- Risks losing the linear time benefit.
- Often requires pointer juggling or index remapping.

Conclusion: Although an in-place variant is possible with complex logic and trade-offs, the standard Bucket Sort is not in-place and is preferred with auxiliary space for simplicity and speed.

11. Python Code: Bucket Sort using Insertion Sort

```
# Insertion Sort function for individual buckets
def insertion_sort(bucket):
    for i in range(1, len(bucket)):
        key = bucket[i]
        j = i - 1
        while j >= 0 and bucket[j] > key:
            bucket[j + 1] = bucket[j]
            j -= 1
        bucket[j + 1] = key

# Bucket Sort using Insertion Sort in each bucket
def bucket_sort(arr):
    n = len(arr)
    if n == 0:
        return arr

    # Create n empty buckets
    buckets = [[] for _ in range(n)]

    # Distribute elements into appropriate buckets
    for num in arr:
        index = int(n * num) # Assumes input in range [0, 1)
        buckets[index].append(num)

    # Sort each bucket using insertion sort
    for bucket in buckets:
        insertion_sort(bucket)

    # Concatenate all sorted buckets
    sorted_arr = []
    for bucket in buckets:
        sorted_arr.extend(bucket)

    return sorted_arr
```

12. Time & Space Complexity and Its Properties

Case	Complexity	Property	Value
Best Case	$O(n+k)$	Stable	Yes (if insertion sort)
Average Case	$O(n+k)$	In-place	No
Worst Case	$O(n^2)$	Adaptive	No
Space Complexity	$O(n+k)$ (extra space)	Recursive	No

13. Final Summary (10 Key Points)

1. Bucket Sort is a **distribution-based** sorting algorithm.
2. It divides the input into multiple **buckets** and sorts each bucket individually.
3. Ideal for inputs that are **uniformly distributed** over a known range.
4. Each bucket can be sorted using any sorting algorithm (e.g., Insertion Sort).
5. The overall time complexity is $O(n+k)$ in the average and best cases.
6. Worst-case time is $O(n^2)$ when all elements fall into the same bucket.
7. **Stability** depends on the sorting method used inside each bucket.
8. It is **not in-place** due to additional memory for buckets.
9. It is **not adaptive** — does not take advantage of existing order.
10. Bucket Sort is efficient for large inputs with uniform distribution, but not general-purpose.