Bucket Sort Algorithm

Comprehensive Notes & Complexity Analysis

1 Introduction to Bucket Sort

Bucket Sort is a **non-comparison based** sorting algorithm. Unlike algorithms such as Quick Sort, Merge Sort, Bubble Sort, or Selection Sort which sort elements by continuously comparing them with one another, Bucket Sort achieves sorting without direct element-to-element comparison. In this category, it operates alongside other non-comparison sorts like **Radix Sort** and **Counting Sort**.

2 Prerequisites and Constraints

To apply Bucket Sort optimally, the input data must satisfy specific constraints defined prior to execution:

- **Data Type & Range:** The algorithm is specifically built to operate on floating-point numbers distributed within the range $[0.0, 1.0)$.
- **Data Distribution:** For Bucket Sort to execute highly efficiently, the input elements should be **uniformly and independently distributed** across the given range.

3 The Algorithm (Pseudocode)

The fundamental logic involves dividing the interval $[0.0, 1.0)$ into n equal-sized sub-intervals (called "buckets"), and then distributing the n input numbers into their corresponding buckets. Because the elements are uniformly distributed, we expect no single bucket to be overloaded.

Pseudocode: BucketSort(A)

```
Input: Array A containing n floating-point numbers in the range [0.0, 1.0)
Let B[0..n-1] be a new array (acting as our buckets)
1. n = length[A]
2. For i = 0 to n-1:
3.     Make B[i] an empty list
4. For i = 1 to n:
5.     index = floor(n × A[i])
6.     Insert A[i] into list B[index]
7. For i = 0 to n-1:
8.     Sort list B[i] using Insertion Sort
9. Concatenate the lists B[0], B[1], ..., B[n-1] together in order
10. Return the concatenated list
```

4 Step-by-Step Working Mechanism

Consider an input array of length $n = 10$:

$$A = [0.79, 0.13, 0.64, 0.39, 0.20, 0.89, 0.53, 0.42, 0.06]$$

We initialize an array of 10 empty buckets (from index 0 to 9). For each element $A[i]$, we calculate its target bucket index using the multiplier formula:

$$\text{Index} = \lfloor n \times A[i] \rfloor$$

Let us trace the mapping calculation for a few elements:

→ **0.79:** $\lfloor 10 \times 0.79 \rfloor = \lfloor 7.9 \rfloor = 7 \implies$ Insert into **Bucket 7**.

→ **0.13:** $\lfloor 10 \times 0.13 \rfloor = \lfloor 1.3 \rfloor = 1 \implies$ Insert into **Bucket 1**.

→ **0.64:** $\lfloor 10 \times 0.64 \rfloor = \lfloor 6.4 \rfloor = 6 \implies$ Insert into **Bucket 6**.

After processing the entire array, the buckets will be uniformly populated as follows:

Bucket Contents Mapping

- **Bucket 0:** [0.06]
- **Bucket 1:** [0.13]
- **Bucket 2:** [0.20]
- **Bucket 3:** [0.39]
- **Bucket 4:** [0.42]
- **Bucket 5:** [0.53]
- **Bucket 6:** [0.64]
- **Bucket 7:** [0.79]
- **Bucket 8:** [0.89]
- **Bucket 9:** [ ] *(Empty)*

Note: Since every populated bucket contains exactly one element, the internal sorting step (Insertion Sort) triggers no operations. Finally, sequentially concatenating the elements from Bucket 0 to 9 yields the completely sorted array.

5 Time Complexity Analysis

5.1 Best & Average Case: $\mathcal{O}(n)$

The **Best and Average Case** occurs when the input elements are **uniformly distributed**.

- Initializing the n empty buckets takes $\mathcal{O}(n)$ time.
- Calculating the mapped index (multiplication and taking the floor value) and inserting each element takes $\mathcal{O}(1)$ operations per element. For n elements, this entire sequence is $\mathcal{O}(n)$.
- Because the elements are uniformly distributed, each bucket acts as a best-case scenario with a very small, constant number of elements (≈ 1). Sorting $\mathcal{O}(1)$ elements takes $\mathcal{O}(1)$ time. Thus, checking/sorting all n buckets collectively takes $\mathcal{O}(n)$ time.
- Finally, concatenating the lists takes $\mathcal{O}(n)$ time.

Overall Time Complexity: $\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) = \boldsymbol{\mathcal{O}(n)}$.

5.2 Worst Case: $\mathcal{O}(n^2)$

The **Worst Case** occurs when the input elements are **highly clustered** instead of uniformly distributed.
Example Scenario: Consider the input array $A = [0.79, 0.78, 0.795, 0.74]$.

* Applying the bucket index formula $\lfloor 10 \times A[i] \rfloor$ to all these elements yields the exact same index: **7**.
* Consequently, *every single element in the array is dumped into one solitary bucket* (Bucket 7).
* To sort the heavily congested elements within this single bucket, the algorithm falls back to its internal **Insertion Sort** (which works functionally similar to arranging playing cards in a hand).
* Insertion Sort on an array of n elements has a widely known worst-case time complexity of $\mathcal{O}(n^2)$.

Overall Time Complexity: In this tightly clustered scenario, the mapping step proves useless, and Bucket Sort entirely degenerates into Insertion Sort, resulting in a worst-case time complexity of $\boldsymbol{\mathcal{O}(n^2)}$.