Algorithm Reviewer

WHAT IS SORTING?

Sorting is arranging data in a specific order — usually ascending or descending. Sorting helps in searching, organizing, and presenting data efficiently.

There are two main categories:

1. **Comparison-Based Sorts** → compare two elements at a time (e.g., Bubble, Selection, Insertion, Merge, Quick, etc.)
2. **Non-Comparison Sorts** → use counting or digit/position instead of direct comparison (e.g., Counting, Radix, Bucket Sort)

COMPARISON SORTS

- These sorts rely on comparing elements to determine their order.

1. Selection Sort

Nature/Idea:

- Finds the smallest (or largest) element and places it at the correct position, one by one.
- Think of it like "picking the smallest card" in each round.

Steps:

1. Find the minimum value in the array.
2. Swap it with the first unsorted element.
3. Move the boundary of sorted/unsorted part.
4. Repeat until all are sorted.

Time Complexity:

- Best = Worst = Average = $O(n^2)$
- Space = O(1)

Technique to Ace It:
Remember: **Selection = "Find & Swap"**
You always select the smallest, then swap it forward.

2. Insertion Sort

Nature/Idea:

- Builds the sorted array one item at a time.
- Like sorting playing cards — you take one card and insert it in the right place among the sorted ones.

Steps:

1. Start with the 2nd element (1st is sorted by default).
2. Compare it backward until you find the right position.
3. Insert it there.
4. Continue for all elements.

Time Complexity:

- Best = O(n) (already sorted)
- Worst = $O(n^2)$
- Space = O(1)

Technique to Ace It:
Think: **Insert each element in the right place among previous ones.**
Works best for **small or nearly sorted data**.

3. Merge Sort

Nature/Idea:

- Divide and conquer algorithm.
- Divides array into halves, sorts them recursively, then merges them.

Steps:

1. Split the array into halves until single-element lists.
2. Merge them back while sorting each pair.

Time Complexity:

- Best = Average = Worst = O(n log n)
- Space = O(n)

Technique to Ace It:
Visualize: **Split → Sort → Merge.**
Perfect for **large datasets** and **stable sorting**.

NON-COMPARISON SORTS

- These do not compare elements directly but use numeric properties (like digit or frequency).

Counting Sort

- is a non-comparison-based sorting algorithm that works by counting how many times each element appears in the input array.
  Instead of comparing elements (like Bubble or Merge Sort), it simply counts and places elements in order based on their frequency.

It's perfect for sorting integers or data with a limited known range (like student scores, ages, or small IDs).

Nature/Idea:

- Works on integer data within a known range.
- Counts how many times each number appears.

Steps:

1. Create a count array for each unique element.
2. Accumulate counts (prefix sum).
3. Place elements into their sorted positions.

Time Complexity:

- O(n + k), where k = range of values
- Space = O(k)

Technique to Ace It:
Perfect for sorting **small-range integers** quickly.
Not for negative or floating values.

Radix Sort

Nature/Idea:

- Sorts numbers digit by digit — from least significant digit (LSD) to most (MSD).
- Uses **Counting Sort** as a subroutine for each digit.

Steps:

1. Sort by 1's place (least significant digit).
2. Sort by 10's place, 100's place, etc.

3. Combine results after all digits.

Time Complexity:

- O(n × k) (where k = number of digits)

Technique to Ace It:
Remember: **Radix = digit by digit sorting.** Works best for **fixed-length integers or strings**.

Bucket Sort

Nature/Idea:

- Divides elements into several "buckets" (groups).
- Each bucket is sorted individually, then combined.

Steps:

1. Scatter: Put elements into buckets based on value.
2. Sort each bucket (using another algorithm).
3. Gather: Merge all sorted buckets.

Time Complexity:

- Best = O(n + k)
- Worst = O(n^2) (if data unevenly distributed)

Technique to Ace It:
Imagine: **Sorting by groups (buckets)** — often used for **floating numbers between 0 and 1**.

SPECIAL SORTING USING LAMBDA AND TUPLES

🔹 Tuple Sort

Nature/Idea:

- Python allows sorting based on multiple values (tuples).
- Example: sort by name, then by score.

```python
students = [("John", 85), ("Alice", 90), ("Bob", 85)]
students.sort(key=lambda x: (x[1], x[0]))
print(students)
# [('Bob', 85), ('John', 85), ('Alice', 90)]
```

Explanation:

- lambda x: (x[1], x[0]) → means sort by 2nd item first, then 1st item if tie.

Technique to Ace It:
Think of tuple as **multi-level sorting** (like Excel: sort by column B, then A).

Lambda Function in Sorting

Nature/Idea:

- Anonymous function that defines a custom sorting rule.
- Used inside sorted() or .sort() to customize order.

Example 1: Sort by string length

```python
words = ["apple", "banana", "fig"]
sorted(words, key=lambda x: len(x))
# ['fig', 'apple', 'banana']
```

Example 2: Sort descending by number

```python
nums = [3, 1, 4]
sorted(nums, key=lambda x: -x)
# [4, 3, 1]
```

Technique to Ace It:
Remember:
👉 key= decides *how to sort*
👉 lambda gives the *rule*

Bucket Sort – The Organized Distributor

🌱 Nature / Introduction

Bucket Sort is a non-comparison-based sorting algorithm.
It's inspired by how people sort objects by category or range, like putting coins into bins based on their value (₱1, ₱5, ₱10, etc.).

It works best for uniformly distributed data—for example, numbers between 0 and 1, or grades between 0–100 that are evenly spread out.

How It Works (Step-by-Step)

Think of buckets as small containers or groups that hold elements with similar ranges.

- **Step 1: Create Buckets**

Divide the input range into several **intervals (buckets)**.

Example: If your numbers are between 0 and 1
→ Create 10 buckets for [0.0–0.1), [0.1–0.2), ... [0.9–1.0)

- **Step 2: Scatter (Distribution)**

Put each element into its corresponding bucket depending on its range.

- **Step 3: Sort Each Bucket**

Sort the individual buckets — often using **Insertion Sort** because each bucket is small and nearly sorted.

- **Step 4: Gather (Concatenation)**

Combine all the sorted buckets into one final sorted list.

```python
def bucket_sort(arr):
    n = len(arr)
    buckets = [[] for _ in range(n)]

    # Step 1 & 2: Distribute elements into buckets
    for num in arr:
        index = int(num * n)   # assuming numbers are between 0
        buckets[index].append(num)

    # Step 3: Sort each bucket
    for bucket in buckets:
        bucket.sort()

    # Step 4: Concatenate results
    sorted_arr = []
    for bucket in buckets:
        sorted_arr.extend(bucket)

    return sorted_arr

data = [0.23, 0.15, 0.89, 0.45, 0.67, 0.38]
print(bucket_sort(data))
```

Output:

```csharp
[0.15, 0.23, 0.38, 0.45, 0.67, 0.89]
```