

# 10\_Sorting and Searching

Quick, Merge, Radix, Bucket

- [Summary](#)
- [QuickSort](#)
- [Merge Sort](#)
- [Radix Sort](#)
- [Bucket Sort](#)

# Summary

## Most Important Sorting Knowledge

As a Technical Artist, focus on:

- 1. **Why Sorting Matters:**
  - GPU performance optimization (batching, transparency sorting).
  - Animation frame ordering.
  - Texture atlas generation.
- 2. **Where Sorting Is Used:**
  - **Draw Call Optimization:** Group by material, shader, or texture.
  - **Transparency Rendering:** Z-depth sorting.
  - **LOD Management:** Sort by distance to the camera.
  - **Animation Playback:** Sort events or keyframes by time.
- 3. **Key Algorithms:**
  - **QuickSort:** Fast general-purpose sorting.
  - **MergeSort:** Stable sort for animations and draw calls.
  - **Radix Sort:** Ultra-fast for IDs, depth sorting, or LOD.
  - **Bucket Sort:** Great for spatial or depth-based sorting.

## Quick Summary Table

Algorithm	Best For	Time Complexity	Stable?
QuickSort	General-purpose, large datasets.	$O(N \log N)$	No
MergeSort	Sorting animations, draw calls stably.	$O(N \log N)$	Yes
Radix Sort	Sorting IDs, LOD distances, fixed data.	$O(N)$	Yes
Bucket Sort	Sorting spatial or depth-based data.	$O(N)$ (best case)	Yes

# Example of Stable vs. Unstable Sort

Imagine sorting a list of objects by **age** (primary key), where objects also have a **name** (secondary property):

**Original List** (Unsorted):

yaml

Copy code

```
[ {Age: 10, Name: "A"}, {Age: 10, Name: "B"}, {Age: 8, Name: "C"} ]
```

## Stable Sort

A stable sorting algorithm **preserves the relative order** of equal elements:

- Both elements with **Age 10** ("A" and "B") keep their original order.

**Result** (After Stable Sort):

yaml

Copy code

```
[ {Age: 8, Name: "C"}, {Age: 10, Name: "A"}, {Age: 10, Name: "B"} ]
```

## Unstable Sort

An unstable sorting algorithm **does not guarantee** the original order of equal elements:

- The relative order of elements with **Age 10** ("A" and "B") may change.

**Result** (After Unstable Sort):

yaml

Copy code

```
[ {Age: 8, Name: "C"}, {Age: 10, Name: "B"}, {Age: 10, Name: "A"} ] <-- Order swapped
```

---

## Why Does Stability Matter?

Stability is critical when:

1. **Sorting Based on Multiple Criteria:**

- You first sort by one key, then by another. Stability ensures the second sort doesn't disrupt the order established by the first.

**Example:** Sorting by **age**, then by **name**:

- First sort: By age → Stable sort keeps relative name order for same ages.
- Second sort: By name → Only elements with equal age are sorted by name.

2. **Preserving Order:**

- In animation systems or rendering pipelines, stability ensures consistent results when sorting objects with identical properties.

3. **Debugging:**

- Stable algorithms provide predictable behavior, making it easier to debug sorting issues.

# QuickSort

## 1. QuickSort

- **Why Important:** QuickSort is one of the fastest general-purpose sorting algorithms for large datasets. It works **in-place** and has an average complexity of  **$O(N \log N)$** .
- **Where It's Useful:**
  - Sorting large datasets like vertex buffers, meshes, or texture indices.
  - Preparing data for **draw calls**: Sorting objects by material, shader, or texture to minimize state changes on the GPU.
  - Sorting elements for **visibility** or **depth-sorting** in transparent objects.

### Example:

- Unity and Unreal **sort renderable objects** back-to-front or front-to-back using QuickSort for transparency or depth optimization.

# Merge Sort

## 2. MergeSort

- **Why Important:** MergeSort is stable (preserves the relative order of equal elements) and handles large datasets efficiently with  **$O(N \log N)$**  time complexity.
- **Where It's Useful:**
  - Sorting animations or frames by timestamps.
  - **Material batching:** Sort objects by material to group draw calls efficiently.
  - **Texture Packing:** Arranging UV data or textures for optimal atlas generation.

### Why Choose MergeSort:

- It's stable, which makes it ideal for sorting when the **relative order** of objects matters (e.g., sorting multiple attributes like texture ID and distance).

# Radix Sort

## 3. Radix Sort

- **Why Important:** Radix Sort is a **non-comparative sorting algorithm** and is faster than QuickSort for integers or fixed-length data types (e.g., IDs, bitmasks). Its complexity is  **$O(N)$**  for small key ranges.
- **Where It's Useful:**
  - Sorting **IDs** for mesh vertices, bones, or texture indices.
  - **Animation Frames:** Sorting animation data with frame indices for playback.
  - Sorting **LOD levels** (Level of Detail) or objects by distance efficiently.

### Why Radix Sort?:

- It's very fast for **fixed-size keys** (integers or floats converted into fixed-size keys), which are common in graphics.

# Bucket Sort

## 4. Bucket Sort

- **Why Important:** Bucket Sort works well when the data is uniformly distributed and falls within a known range.
- **Where It's Useful:**
  - **Depth Sorting:** Sort objects by their Z-depth for transparency.
  - **Light Probes or Particles:** Sorting objects into spatial buckets for rendering optimizations.
  - Sorting **textures or UV coordinates** into regions when packing atlases.

### Why Bucket Sort?:

- Very fast  **$O(N)$**  sorting when data can be grouped into buckets (common in spatial optimizations).