🏥

Linear Sorting

📘 Linear Sorting — Test Notes

🔷 Distribution Sort

Idea:

- Distribute elements into blocks/buckets using a key function `key(x)`.
- If `x < y ⇒ key(x) < key(y)` (strict), then no sorting is needed inside blocks.
- If `x < y ⇒ key(x) ≤ key(y)` (weak), then we must sort inside blocks.
- Concatenate blocks to get final sorted list.

Process:

1. Find suitable key function.
2. Create empty blocks.
3. Place each element in its block.
4. Sort blocks if needed.
5. Concatenate.

Complexity:

- With strong key: **O(n + k)**, where `k = number of blocks`.
- With weak key: depends on sorting inside blocks (could be O(n log n)).

Properties:

- Stable if we just append to blocks (order preserved).
- Not in-place (extra storage).

Key use: Sorting when we can map values directly into small number of buckets (e.g., integers with small range).

🔷 Counting Sort

Idea:

Optimized distribution sort for integers in a limited range. Instead of storing buckets, just **count frequencies** and then compute positions.

Process:

1. Find max key `k`.
2. Create count array `count[0..k]`, init 0.
3. Count occurrences of each value.
4. Turn counts into cumulative counts (ending indices).
5. Traverse input **right-to-left** → place each element in correct slot of output (this ensures stability).

Complexity:

- **O(n + k)** time, **O(n + k)** space.
- Best/worst/average all the same.

Properties:

- Stable (if traversed right-to-left).
- Not in-place (needs output array).

When to use:

- Integers in a small/known range.
- Subroutine in radix sort.
- Example: sorting exam scores [0..100].

Key details:

- `k = max − min + 1` (range of input).
- Stability is preserved using **right-to-left traversal**.

🔷 Radix Sort

Idea:

Sort **digit by digit** using a stable sort (usually Counting Sort).

- LSD (least significant digit first): bottom-up.
- MSD (most significant digit first): top-down with recursion.

Process (LSD):

1. Find number of digits `d` in max element.
2. For each digit position from LSD → MSD:
    - Use **stable Counting Sort** on that digit.
3. After d passes, array is sorted.

Why it works:

Stability ensures that the ordering from less significant digits is preserved as more significant digits are processed.

Complexity:

- One pass = O(n + k).
- Total = **O(d·(n + k))**.
- If `d` and `k` are constants (e.g., fixed-width integers, base 10/256): **O(n)**.

Properties:

- Stable (must use stable subroutine).
- Not in-place (needs extra storage per pass).

When to use:

- Fixed-width integers, strings, large datasets where comparison sort (O(n log n)) is slower.
- Databases with many tie-break fields.

🔷 Bucket Sort

Idea:

Distribute values into **range-based buckets** (e.g., intervals), then sort each bucket individually.

Process:

1. Decide number of buckets `b`.
2. Distribute elements into buckets using a key function (e.g., bucket = floor(n·x) for [0,1)).
3. Sort each bucket (often insertion sort).
4. Concatenate buckets.

Complexity:

- Distribute: $O(n)$.
- Sort inside buckets: $O(n^2/b)$ on average.
- Total: **$O(n + n^2/b)$**.
- If `b ≈ n` and distribution uniform → expected **$O(n)$**.
- Worst case (all elements in one bucket): $O(n^2)$ with insertion sort (or $O(n \log n)$ with better sort).

Properties:

- Stable if stable sort used inside buckets.
- Not in-place (uses extra buckets).

When to use:

- Input uniformly distributed over a known range (e.g., floats in [0,1)).
- Random real numbers, hash-based grouping.

✅ Quick Comparison

Algorithm	Complexity	Stable	In-place	Best use case
Distribution	$O(n + k)$ (strict key)	Yes	No	When strong key exists
Counting Sort	$O(n + k)$	Yes	No	Integers in small range
Radix Sort	$O(d \cdot (n + k)) \rightarrow O(n)$	Yes	No	Fixed-width ints, strings
Bucket Sort	Expected $O(n)$, worst $O(n^2)$	Depends	No	Uniformly distributed reals