Sorting Technique

Bubble Sort :-

1️⃣ Type of Sorting

- **Comparison-based**: sorts by comparing adjacent elements.
- **In-place**: does not require extra memory aside from a few variables.
- **Stable**: preserves relative order of equal elements.

2️⃣ Algorithm Idea

- Repeatedly compare adjacent elements and swap if they are in the wrong order.
- After each pass, the largest unsorted element “bubbles up” to its correct position.
- Continue passes until no swaps are required.

3️⃣ Characteristics

- **Stable** → preserves order of equal elements.
- **In-place** → only constant extra memory needed.
- Simple and easy to implement.
- Can be **adaptive** if optimized: stops early when array is already sorted.

4️⃣ Time Complexity

Let n = number of elements

1. **Worst Case (reverse sorted)** → $O(n^2)$
    - Maximum number of comparisons and swaps.
2. **Average Case (random order)** → $O(n^2)$
    - On average, half the comparisons require swaps.
3. **Best Case (already sorted, with optimization)** → $O(n)$
    - Only one pass required; no swaps detected → algorithm stops early.

Reasoning:

- Two nested loops → outer loop for passes, inner loop for comparisons → gives quadratic complexity in worst and average cases.
- Early stopping reduces best-case to linear.

5️⃣ Space Complexity

- **Auxiliary Space:** O(1)
- Reason: Sorting is done in-place with a temporary variable for swaps.

6️⃣ Stability

- **Yes**
- Equal elements are not swapped, so their relative order remains intact.

7️⃣ Pros and Cons

Pros:

- Simple to implement.
- Stable and in-place.
- Adaptive if optimized.

Cons:

- Inefficient for large arrays ($O(n^2)$).
- Not suitable for practical large-scale applications.

✅ Summary Table (Bubble Sort)

Feature	Value
Type	Comparison-based, In-place
Stability	Stable
Best Case TC	O(n)
Average Case TC	$O(n^2)$
Worst Case TC	$O(n^2)$
Space Complexity	O(1)
Adaptive	Yes (if optimized)
Use	Small arrays, teaching purposes

Selection Sort :-

1️⃣ Type of Sorting

- **Comparison-based**: finds the minimum (or maximum) element in the unsorted portion and places it in the correct position.
- **In-place**: only uses a constant amount of extra memory.
- **Not stable** by default: swapping may change the relative order of equal elements.

2️⃣ Algorithm Idea

- Divide the array into **sorted** and **unsorted** parts.
- Repeatedly select the **minimum element** from the unsorted part and swap it with the first unsorted element.
- Continue until the whole array is sorted.

3️⃣ Characteristics

- **Not stable** → swapping can move equal elements out of order.
- **In-place** → constant extra space.
- Simple and easy to implement.
- **Not adaptive** → always performs the same number of comparisons, even if the array is partially sorted.

4️⃣ Time Complexity

Let n = number of elements

1. **Worst Case (reverse sorted)** → $O(n^2)$
    - Outer loop runs n times, inner loop scans remaining unsorted elements → n(n-1)/2 comparisons.
2. **Average Case (random order)** → $O(n^2)$
    - Same number of comparisons regardless of element arrangement.
3. **Best Case (already sorted)** → $O(n^2)$
    - Even if array is sorted, it still scans for the minimum each pass.

Reasoning:

- **Comparisons:** always n(n-1)/2 → quadratic.
- **Swaps:** exactly n-1 → fewer than Bubble Sort (advantage for large swaps).

5️⃣ Space Complexity

- **Auxiliary Space:** O(1)
- Sorting done in-place using a temporary variable for swapping.

6️⃣ Stability

- **Not stable** by default.
- Example: if two equal elements are swapped, their relative order changes.

7️⃣ Pros and Cons

Pros:

- Simple and easy to implement.
- Performs **minimum number of swaps** (useful if swapping is expensive).
- In-place → low memory usage.

Cons:

- Inefficient for large arrays ($O(n^2)$ comparisons).
- Not stable.
- Not adaptive → does not benefit from partially sorted arrays.

✅ Summary Table (Selection Sort)

Feature	Value
Type	Comparison-based, In-place
Stability	Not stable
Best Case TC	$O(n^2)$
Average Case TC	$O(n^2)$
Worst Case TC	$O(n^2)$
Space Complexity	O(1)
Adaptive	No
Swaps	Minimum (n-1)
Use	Small arrays or when swaps are costly

Insertion Sort :-

1️⃣ Type of Sorting

- **Comparison-based**: elements are compared and shifted to insert them at the correct position.
- **In-place**: only a small amount of extra memory is required.
- **Stable**: preserves the relative order of equal elements.

2️⃣ Algorithm Idea

- Divide the array into **sorted** and **unsorted** parts.
- Pick each element from the unsorted part and **insert it into its correct position** in the sorted part by shifting larger elements.
- Continue until the entire array is sorted.

3️⃣ Characteristics

- **Stable** → equal elements retain their relative order.
- **In-place** → O(1) extra space.
- **Adaptive** → performs efficiently if the array is nearly sorted.
- Simple to implement and efficient for **small datasets**.

4️⃣ Time Complexity

Let n = number of elements

1. **Best Case (already sorted)** → O(n)
    - Only comparisons, no shifting required.
2. **Average Case (random array)** → $O(n^2)$
    - Elements need to be shifted for insertion; roughly $n^2/4$ comparisons and shifts.
3. **Worst Case (reverse sorted)** → $O(n^2)$
    - Every element has to be compared with all elements in the sorted part and shifted → $n(n-1)/2$ comparisons and shifts.

Reasoning:

- Inner loop performs shifting for each element → dominates time complexity.
- Outer loop runs n-1 times → total comparisons and shifts lead to $O(n^2)$ in average/worst cases.

5️⃣ Space Complexity

- **Auxiliary Space:** O(1) → Sorting is in-place using a temporary variable to hold the element being inserted.

6️⃣ Stability

- **Yes**
- Equal elements are never swapped out of order; only shifting occurs.

7️⃣ Pros and Cons

Pros:

- Stable and in-place.
- Adaptive → very efficient for nearly sorted arrays.
- Simple to implement.
- Good for **small arrays**.

Cons:

- Inefficient for large, randomly ordered arrays ($O(n^2)$).
- Not suitable for large-scale datasets.

✅ Summary Table (Insertion Sort)

Feature	Value
Type	Comparison-based, In-place
Stability	Stable
Best Case TC	O(n)
Average Case TC	$O(n^2)$
Worst Case TC	$O(n^2)$
Space Complexity	O(1)
Adaptive	Yes
Use	Small arrays, nearly sorted arrays, teaching purposes

Merge Sort :-

1 Type of Sorting

- **Comparison-based**: elements are compared during the merge step.
- **Divide & Conquer**: splits the array into halves, sorts each half recursively, then merges them.
- **Stable**: preserves the relative order of equal elements.
- **Not in-place**: requires extra memory for merging.

2 Algorithm Idea

- Recursively **divide** the array into two halves until each subarray has one element.
- **Merge** the subarrays back together in sorted order.
- Continue until the entire array is merged and sorted.

3 Characteristics

- **Stable** → equal elements maintain relative order.
- **Not in-place** → requires O(n) extra memory for temporary arrays during merging.
- **Divide & Conquer approach** → very efficient for large datasets.
- Consistently fast regardless of input order.

4 Time Complexity

Let n = number of elements

1. **Best Case** → O(n log n)
2. **Average Case** → O(n log n)
3. **Worst Case** → O(n log n)

Reasoning:

- **Division:** array is recursively split in half → $\log_2(n)$ levels.
- **Merging:** each level requires O(n) comparisons and copying.
- Total = O(n) per level × $\log_2(n)$ levels → **O(n log n)**.

5 Space Complexity

- **Auxiliary Space:** O(n)
- Reason: Temporary arrays are needed to merge sorted subarrays.

6️⃣ Stability

- **Yes**
- During merging, when two elements are equal, the element from the left subarray is copied first → relative order preserved.

7️⃣ Pros and Cons

Pros:

- Stable and reliable.
- **Predictable O(n log n)** time for all cases.
- Efficient for large datasets.
- Works well for **linked lists** (can be done in-place for linked lists).

Cons:

- Requires extra memory → not in-place for arrays.
- Slightly slower than Quick Sort in practice due to memory overhead.

✅ Summary Table (Merge Sort)

Feature	Value
Type	Comparison-based, Divide & Conquer
Stability	Stable
Best Case TC	O(n log n)
Average Case TC	O(n log n)
Worst Case TC	O(n log n)
Space Complexity	O(n)
Adaptive	No
Use	Large datasets, stable sorting, linked lists

Quick Sort :-

1️⃣ Type of Sorting

- **Comparison-based**: elements are compared to a **pivot** during partitioning.
- **Divide & Conquer**: partitions the array into smaller subarrays recursively.
- **Not stable**: swapping may change the relative order of equal elements.
- **In-place**: uses a small amount of extra memory for recursion.

2️⃣ Algorithm Idea

- Choose a **pivot element** (first, last, middle, or median).
- **Partition** the array: elements less than the pivot go left, greater go right.
- Recursively sort the left and right subarrays.
- Combine results → the array is sorted.

3️⃣ Characteristics

- **Not stable** → equal elements may swap positions.
- **In-place** → low memory usage (O(log n) recursion stack).
- **Adaptive** depends on pivot choice → performance can vary.
- **Efficient** for large datasets on average.
- Highly popular in practice due to low overhead.

4️⃣ Time Complexity

Let n = number of elements

1. **Best Case** → O(n log n)
    - Pivot divides array into two nearly equal halves each time.
2. **Average Case** → O(n log n)
    - Random pivot selection usually gives balanced partitions.
3. **Worst Case** → $O(n^2)$
    - Pivot always the smallest or largest element → unbalanced partitions (e.g., already sorted with first/last pivot).

Reasoning:

- **Partitioning:** O(n) comparisons per level.
- **Recursion levels:** $\log_2(n)$ for balanced splits → O(n log n).
- **Worst case:** recursion depth = n → $O(n^2)$.

5️⃣ Space Complexity

- **Auxiliary Space:** O(log n) on average (for recursion stack).
- In-place sorting → no extra arrays needed.

6️⃣ Stability

- **No** → swapping during partitioning may change relative order of equal elements.

7️⃣ Pros and Cons

Pros:

- Fast in practice → low overhead, in-place.
- Average-case O(n log n) → suitable for large datasets.
- Simple implementation and widely used.

Cons:

- Not stable.
- Worst-case $O(n^2)$ if pivot choice is poor.
- Recursive → recursion stack may grow to O(n) in worst case.

✅ Summary Table (Quick Sort)

Feature	Value
Type	Comparison-based, Divide & Conquer
Stability	Not stable
Best Case TC	O(n log n)
Average Case TC	O(n log n)
Worst Case TC	$O(n^2)$
Space Complexity	O(log n)
Adaptive	Yes, with good pivot selection
Use	Large datasets, fast in practice, in-place sorting

Counting Sort :-

1️⃣ Type of Sorting

- **Non-comparison-based**: sorts elements by **counting their occurrences**.
- **Stable**: preserves relative order of equal elements.
- **Not in-place**: requires extra memory for count array.
- Suitable for **integers or small-range discrete values**.

2️⃣ Algorithm Idea

- Find the **maximum value (k)** in the array.
- Create a **count array** of size k+1 to store the frequency of each element.
- Compute the **cumulative frequency** to determine the correct position of each element.
- Place each element in the output array according to the cumulative frequency.

3️⃣ Characteristics

- **Stable** → preserves relative order of equal elements.
- **Not in-place** → requires O(k + n) extra space for count and output arrays.
- Efficient for **integers with small range**.
- **Deterministic** → time complexity depends on n (number of elements) and k (range), not input order.

4️⃣ Time Complexity

Let n = number of elements, k = maximum element value

1. **Best Case:** O(n + k)
2. **Average Case:** O(n + k)
3. **Worst Case:** O(n + k)

Reasoning:

- Counting frequencies → O(n)
- Cumulative frequency calculation → O(k)
- Placing elements in output array → O(n)
- Total = O(n + k)

Note: Unlike comparison sorts, Counting Sort does **not rely on element comparisons**, so it can achieve linear time for small ranges.

5️⃣ Space Complexity

- **Auxiliary Space:** O(n + k)
    - Count array → O(k)
    - Output array → O(n)

6️⃣ Stability

- **Yes**
- When inserting elements into the output array using cumulative frequency, **elements from left to right** maintain their relative order.

7️⃣ Pros and Cons

Pros:

- Linear time for small ranges → very fast.
- Stable → maintains relative order.
- Deterministic → predictable performance.

Cons:

- Not in-place → extra memory needed.
- Not suitable for large ranges → O(k) can be very large.
- Works only for integers (or values that can be mapped to integers).

✅ Summary Table (Counting Sort)

Feature	Value
Type	Non-comparison-based
Stability	Stable
Best Case TC	O(n + k)
Average Case TC	O(n + k)
Worst Case TC	O(n + k)
Space Complexity	O(n + k)
Adaptive	No
Use	Small-range integers, counting problems, linear-time sorting

Radix Sort :-

1 Type of Sorting

- **Non-comparison-based**: sorts elements **digit by digit** rather than by comparing values directly.
- **Stable** → preserves relative order of equal elements.
- **Not in-place** → requires extra memory for intermediate arrays (usually uses Counting Sort as a subroutine).
- Suitable for **integers or fixed-length strings**.

2 Algorithm Idea

- Determine the **maximum number of digits** in the array elements.
- Sort the array **digit by digit** from least significant digit (LSD) to most significant digit (MSD) using a **stable sort** (usually Counting Sort).
- After the last digit sort, the array becomes completely sorted.

3 Characteristics

- **Stable** → uses a stable subroutine (like Counting Sort) for each digit.
- **Not in-place** → auxiliary arrays are used for counting or temporary storage.
- Works efficiently when **range of digits is small** compared to the number of elements.
- **Deterministic** → time complexity depends on n (number of elements) and k (number of digits).

4 Time Complexity

Let n = number of elements, d = number of digits, k = base (e.g., 0–9 for decimal digits)

1. **Best Case:** $O(d \times (n + k))$
2. **Average Case:** $O(d \times (n + k))$
3. **Worst Case:** $O(d \times (n + k))$

Reasoning:

- Each digit is sorted using Counting Sort → $O(n + k)$ per digit.
- Number of digits = d → total time = $O(d \times (n + k))$.

Note: For fixed-length integers, d is usually constant → Radix Sort can run in **linear time O(n)**.

5️⃣ Space Complexity

- **Auxiliary Space:** O(n + k)
    - For Counting Sort on each digit.
    - Temporary arrays needed to store intermediate results.

6️⃣ Stability

- **Yes**
- Stability of the subroutine (Counting Sort) ensures that sorting by each digit does not change the relative order of equal elements.

7️⃣ Pros and Cons

Pros:

- Linear time sorting for small digit numbers → very efficient for large arrays of integers.
- Stable → preserves order of equal elements.
- Deterministic → predictable performance.

Cons:

- Not in-place → uses extra memory.
- Works only for integers or fixed-length strings.
- Efficiency depends on **number of digits**; not suitable for very large numbers with many digits.

✅ Summary Table (Radix Sort)

Feature	Value
Type	Non-comparison-based
Stability	Stable
Best Case TC	O(d × (n + k))
Average Case TC	O(d × (n + k))
Worst Case TC	O(d × (n + k))
Space Complexity	O(n + k)
Adaptive	No
Use	Sorting integers or fixed-length strings efficiently

Bucket Sort :-

1 Type of Sorting

- **Non-comparison-based** (distribution sort): elements are distributed into **buckets** based on their value.
- **Stable** if the sorting used within each bucket is stable (like Insertion Sort).
- **Not in-place** → uses extra memory for buckets.
- Efficient for **uniformly distributed data**, especially floating-point numbers in the range (0,1).

2 Algorithm Idea

- Divide the interval of input values into **k buckets**.
- Distribute each element into its corresponding bucket.
- Sort each bucket individually (usually using Insertion Sort).
- Concatenate all buckets to get the sorted array.

3 Characteristics

- **Stable** → maintains relative order if stable sort is used inside buckets.
- **Not in-place** → needs extra memory for buckets.
- Works best when data is **uniformly distributed**.
- Adaptive for data that is **already partially sorted** within buckets.

4 Time Complexity

Let n = number of elements, k = number of buckets

1. **Best Case (even distribution in buckets)** → O(n + k)
    - Each bucket has roughly n/k elements → sorting inside each bucket is fast (O(n/k) per bucket).
2. **Average Case (uniform distribution)** → O(n + k)
    - Most elements are evenly distributed → sorting inside buckets is efficient.
3. **Worst Case (all elements in one bucket)** → $O(n^2)$
    - All elements fall into a single bucket → sorting that bucket (e.g., with Insertion Sort) takes $O(n^2)$.**Reasoning:**

- Distribution into buckets → O(n)
- Sorting each bucket → depends on elements per bucket
- Concatenating buckets → O(n)

5️⃣ Space Complexity

- **Auxiliary Space:** O(n + k)
    - Buckets store subsets of elements temporarily.

6️⃣ Stability

- **Yes**, if the sorting algorithm used inside buckets is stable.

7️⃣ Pros and Cons

Pros:

- Linear time for uniformly distributed data → very fast.
- Stable if stable sort is used in buckets.
- Adaptive → benefits from partially sorted distributions.

Cons:

- Not in-place → extra memory needed.
- Efficiency depends on **distribution of data**.
- Worst-case $O(n^2)$ if data is skewed.
- Works best for **floating-point numbers in (0,1)** or normalized data.

✅ Summary Table (Bucket Sort)

Feature	Value
Type	Non-comparison-based, distribution sort
Stability	Stable (if stable sort used in buckets)
Best Case TC	O(n + k)
Average Case TC	O(n + k)
Worst Case TC	$O(n^2)$
Space Complexity	O(n + k)
Adaptive	Yes (depends on distribution)
Use	Uniformly distributed numbers, floating-point numbers in (0,1)

Heap Sort :-

Type of Sorting

- **Comparison-based**: sorts elements by using a **binary heap** data structure.
- **In-place**: uses no extra array for sorting aside from a few variables.
- **Not stable**: swapping elements in the heap may change the relative order of equal elements.
- **Divide & Conquer / Selection-based**: repeatedly extracts the maximum (or minimum) from the heap and places it at the correct position.

2️⃣ Algorithm Idea

- Build a **max-heap** from the array (largest element at the root).
- Swap the root with the last element → largest element moves to its correct position.
- Reduce heap size by one and **heapify** the root to restore the heap property.
- Repeat until all elements are sorted.

3️⃣ Characteristics

- **Not stable** → swapping during heapify can change relative order.
- **In-place** → uses constant extra memory.
- **Deterministic** → worst-case time is always O(n log n).
- Works well for **large datasets** and when memory efficiency is important.

4️⃣ Time Complexity

Let n = number of elements

1. **Best Case** → O(n log n)
    - Building heap and heapifying elements still requires log n operations per element.
2. **Average Case** → O(n log n)
    - Each heapify operation takes log n, done for n elements.
3. **Worst Case** → O(n log n)
    - Same as average case → consistent performance.

Reasoning:

- **Building max-heap:** O(n)
- **Heapify during extraction:** O(log n) per element × n elements → O(n log n)
- Total = O(n) + O(n log n) = O(n log n)

5️⃣ Space Complexity

- **Auxiliary Space:** O(1)
- Reason: In-place sorting; no extra arrays needed.

6️⃣ Stability

- **No**
- Heapify operations swap elements arbitrarily → equal elements may change order.

7️⃣ Pros and Cons

Pros:

- In-place → low memory usage.
- Predictable O(n log n) time → good for large datasets.
- Simple implementation using arrays.

Cons:

- Not stable.
- Slower than Quick Sort in practice due to less locality of reference.
- Not adaptive → performance does not improve for partially sorted arrays.

✅ Summary Table (Heap Sort)

Feature	Value
Type	Comparison-based, in-place, selection-like
Stability	Not stable
Best Case TC	O(n log n)
Average Case TC	O(n log n)
Worst Case TC	O(n log n)
Space Complexity	O(1)
Adaptive	No
Use	Large datasets, in-place sorting, predictable performance

Shell Sort :-

1️⃣ Type of Sorting

- **Comparison-based**: generalization of **Insertion Sort**, compares elements that are a gap apart.
- **In-place**: only a small amount of extra memory is needed.
- **Not stable**: swapping distant elements can change the relative order of equal elements.
- Efficient for **medium-sized arrays**.

2️⃣ Algorithm Idea

- Choose a **gap sequence** (e.g., n/2, n/4, …, 1).
- For each gap, perform **gapped insertion sort**: compare elements that are gap apart and swap if needed.
- Reduce the gap until it becomes 1 → final pass is a normal insertion sort.

3️⃣ Characteristics

- **Not stable** → elements far apart may swap.
- **In-place** → only a few variables needed for swapping.
- **Adaptive** → performs better than Insertion Sort on partially sorted arrays.
- Efficiency depends on the **gap sequence** chosen.

4️⃣ Time Complexity

Let n = number of elements

1. **Best Case (nearly sorted array, good gap sequence)** → O(n log n)
2. **Average Case** → depends on gap sequence, typically O(n^(3/2))
3. **Worst Case (original Shell sequence)** → $O(n^2)$

Reasoning:

- The outer loop reduces gaps, inner loop performs gapped insertion sort.
- Fewer comparisons for larger gaps → reduces total shifts compared to normal Insertion Sort.
- Using better sequences (Hibbard, Sedgewick) improves performance to O(n^(4/3))–O(n^(3/2)).

5️⃣ Space Complexity

- **Auxiliary Space:** O(1) → in-place sorting.

6️⃣ Stability

- **No**
- Swapping elements that are far apart can disturb the relative order of equal elements.

7️⃣ Pros and Cons

Pros:

- In-place → low memory usage.
- Adaptive → faster than Bubble or Insertion Sort for medium-sized arrays.
- Simple implementation with gap sequence.

Cons:

- Not stable.
- Time complexity depends on gap sequence.
- Worst-case still $O(n^2)$ with original gap sequence.

✅ Summary Table (Shell Sort)

Feature	Value
Type	Comparison-based, in-place
Stability	Not stable
Best Case TC	O(n log n) (with good gap)
Average Case TC	O(n^(3/2))
Worst Case TC	$O(n^2)$
Space Complexity	O(1)
Adaptive	Yes
Use	Medium-sized arrays, partially sorted arrays

Tim Sort :-

1️⃣ Type of Sorting

- **Hybrid sorting algorithm**: combines **Merge Sort** and **Insertion Sort**.
- **Comparison-based**.
- **Stable**: preserves relative order of equal elements.
- Designed to perform well on **real-world data** (partially ordered arrays).

2️⃣ Algorithm Idea

- Divide the array into **small segments (runs)**.
- Sort each run using **Insertion Sort** (efficient for small arrays).
- Merge runs using **Merge Sort** strategy.
- Repeat until the entire array is merged into a fully sorted array.

3️⃣ Characteristics

- **Stable** → preserves relative order of equal elements.
- **Not in-place** → requires O(n) extra space for merging runs.
- **Adaptive** → extremely efficient for partially sorted arrays.
- Designed for **real-world practical sorting** → used in Python and Java.

4️⃣ Time Complexity

Let n = number of elements, run size = minrun

1. **Best Case (already sorted)** → O(n)
    - Insertion Sort on small runs is linear.
2. **Average Case** → O(n log n)
3. **Worst Case** → O(n log n)
    - Merging runs dominates time complexity.

Reasoning:

- Runs are sorted with Insertion Sort → O(n) for small or nearly sorted runs.
- Merge process → O(n log n) total across all levels.
- Adaptivity gives best-case linear time for nearly sorted data.

5️⃣ Space Complexity

- **Auxiliary Space:** O(n)
- Needed for temporary arrays during merging runs.

6️⃣ Stability

- **Yes**
- Merge step preserves the order of equal elements.

7️⃣ Pros and Cons

Pros:

- Stable.
- Adaptive → very fast on partially sorted arrays.
- Efficient in practice → widely used in standard libraries.
- Handles large datasets well.

Cons:

- Not in-place → extra memory required.
- Slightly more complex than basic sorts like Quick or Merge Sort.

✅ Summary Table (Tim Sort)

Feature	Value
Type	Hybrid (Insertion + Merge), Comparison-based
Stability	Stable
Best Case TC	O(n)
Average Case TC	O(n log n)
Worst Case TC	O(n log n)
Space Complexity	O(n)
Adaptive	Yes
Use	Real-world sorting, partially sorted arrays, large datasets

Cocktail (Shaker) Sort :-

1️⃣ Type of Sorting

- **Comparison-based**, **in-place** sorting algorithm.
- **Stable** → preserves relative order of equal elements.
- Variant of **Bubble Sort**, but sorts in **both directions**.

2️⃣ Algorithm Idea

- Traverse the array **forward**, bubbling the largest element to the end.
- Then traverse **backward**, bubbling the smallest element to the beginning.
- Repeat forward and backward passes until the array is sorted.

3️⃣ Characteristics

- **Stable** → preserves order of equal elements.
- **In-place** → uses only a few extra variables for swapping.
- **Adaptive** → can stop early if no swaps occur during a pass.
- More efficient than Bubble Sort on **partially sorted arrays**.

4️⃣ Time Complexity

Let n = number of elements

1. **Best Case (already sorted)** → O(n)
    - Only one forward and backward pass needed; no swaps.
2. **Average Case** → $O(n^2)$
    - Comparisons and swaps similar to Bubble Sort.
3. **Worst Case (reverse sorted)** → $O(n^2)$
    - Maximum comparisons and swaps required.

Reasoning:

- Two nested loops: outer loop controls passes, inner loop performs comparisons/swaps.
- Forward and backward traversals slightly reduce passes compared to Bubble Sort.

5️⃣ Space Complexity

- **Auxiliary Space:** O(1)
- Sorting is done in-place with a temporary variable for swapping.

6️⃣ Stability

- **Yes**
- Equal elements are not swapped out of order.

7️⃣ Pros and Cons

Pros:

- Stable and in-place.
- Adaptive → better than Bubble Sort on nearly sorted arrays.
- Simple to implement.

Cons:

- Time complexity still $O(n^2)$ for large arrays → inefficient for large datasets.
- Slightly more complex than Bubble Sort but gains minor improvement.

✅ Summary Table (Cocktail Sort / Shaker Sort)

Feature	Value
Type	Comparison-based, in-place
Stability	Stable
Best Case TC	O(n)
Average Case TC	$O(n^2)$
Worst Case TC	$O(n^2)$
Space Complexity	O(1)
Adaptive	Yes
Use	Small or partially sorted arrays

Comb Sort :-

1️⃣ Type of Sorting

- **Comparison-based**, **in-place** sorting algorithm.
- **Improvement over Bubble Sort** → eliminates small values near the end (turtles) faster.
- **Not stable** by default.

2️⃣ Algorithm Idea

- Use a **gap sequence** instead of comparing only adjacent elements.
- Start with a gap = n (array size) and reduce it using a **shrink factor** (commonly 1.3).
- Compare elements that are **gap apart** and swap if needed.
- Repeat until gap = 1 and no swaps occur → array is sorted.

3️⃣ Characteristics

- **Not stable** → swapping distant elements may change relative order of equal elements.
- **In-place** → uses only a few variables.
- Faster than Bubble Sort → handles small values near the end more efficiently.
- Adaptive to some extent → reduces number of swaps for partially sorted arrays.

4️⃣ Time Complexity

Let n = number of elements

1. **Best Case (already sorted)** → O(n log n)
    - Fewer comparisons needed; gap sequence quickly converges.
2. **Average Case** → $O(n^2 / 2^p) \sim O(n^2 / \log n)$
    - Depends on shrink factor; better than Bubble Sort in practice.
3. **Worst Case** → $O(n^2)$
    - If the gap sequence is not optimal, may require many comparisons.

Reasoning:

- Gap reduces the number of small-distance swaps → removes turtles faster.
- Final pass with gap = 1 is like Bubble Sort, but fewer elements are left to swap.

5️⃣ Space Complexity

- **Auxiliary Space:** O(1)
- Sorting done in-place.

6️⃣ Stability

- **No**
- Swapping elements with a large gap may change relative order of equal elements.

7️⃣ Pros and Cons

Pros:

- Faster than Bubble Sort and Cocktail Sort in practice.
- In-place → low memory usage.
- Simple to implement.

Cons:

- Not stable.
- Worst-case $O(n^2)$ → inefficient for large datasets.
- Time complexity depends on shrink factor.

✅ Summary Table (Comb Sort)

Feature	Value
Type	Comparison-based, in-place
Stability	Not stable
Best Case TC	O(n log n)
Average Case TC	~$O(n^2 / \log n)$
Worst Case TC	$O(n^2)$
Space Complexity	O(1)
Adaptive	Somewhat
Use	Small to medium arrays, faster Bubble Sort alternative