Design & Analysis Algorithm

Assignment 2

AIR UNIVERSITY

<u>Submitted by:</u>

Name:Ahmed Khan
Roll No: 232530
BSCSev-V-A

<u>Submitted to:</u>

Dr faheemullah

Dated: 3 Nov, 2025

AIR UNIVERSITY , ISLAMABAD

Analysis of Sorting Algorithms:

1) Bubble Sort

Idea: Repeatedly sweep through the list, compare adjacent items, and swap them when out of order. Large values “bubble” toward the end of each pass.

Step-by-step process:

1. Start at the beginning of the array.
2. Compare item i and i+1. If i > i+1, swap them.
3. Move to the next pair and repeat until the end — that finishes one pass.
4. After each full pass the largest remaining element is in its final place.
5. Repeat passes until a pass makes no swaps (array is sorted).

Complexities:

- Best: $O(n)$ — if the array is already sorted and you detect “no swaps”.
- Average: $O(n^2)$
- Worst: $O(n^2)$

Space: $O(1)$ extra (in-place).
Stability: Stable (equal elements keep relative order if you don't swap equal ones).
When to use: Simple to implement and useful for tiny arrays or for teaching. Very inefficient for large datasets.

2) Selection Sort

Idea (plain): Repeatedly pick the smallest (or largest) unsorted element and move it to the next position of the sorted prefix.

Step-by-step process

1. For position pos = 0 to n-2:
2. Find the index minIndex of the smallest element in the range pos...n-1.

3. Swap the element at pos with element at minIndex.
4. Increase pos and repeat.

Complexities:

- Best/Average/Worst: $O(n^2)$

Space: O(1).
Stability: Not stable by default (swapping can reorder equal items). It can be made stable but with extra cost (not in-place).
When to use: Predictable behavior, fewer swaps than bubbles. Good when writes are expensive but not for large data because comparisons are still $O(n^2)$.

3) Heap Sort

Idea: Build a binary heap (max-heap for ascending sort), repeatedly remove the root (largest element) and place it at the end of the array.

Step-by-step process

1. Transform the array into a max-heap (heapify) — O(n).
2. For end = n-1 down to 1:
    - Swap heap root (max) with array[end].
    - Reduce heap size by 1.
    - Sift-down the new root to restore heap property.
3. After iterations, array is sorted ascending.

Complexities

- Best/Average/Worst: O(n log n)

Space: O(1) .
Stability: Not stable.
When to use: Reliable O(n log n) worst-case cost — good when worst-case guarantees matter. Slightly slower in practice than tuned quicksort on average due to more memory / cache-unfriendly accesses.

4) Quick Sort

Idea: Choose a pivot element, partition the array into elements less than pivot and greater than pivot, then sort the partitions recursively.

Step-by-step process

1. Choose a pivot (first/last/middle/random/median-of-three).
2. Partition the array into two parts: values ≤ pivot on left, values ≥ pivot on right (partition method variants exist).
3. Recursively apply quicksort to left and right partitions until base case (size ≤ 1).

Complexities

- Best: O(n log n) — balanced partitions.
- Average: O(n log n)
- Worst: $O(n^2)$ — extremely unbalanced partitions (e.g., sorted input with poor pivot choice).

Space: O(log n) average stack for recursion (worst O(n) with bad pivots). In-place partitions are possible.
Stability: Not stable by default.
When to use: Extremely fast in practice for random data; choose pivot strategy to avoid worst-case. Tail recursion elimination or iterative variants reduce stack depth.

5) Counting Sort

Idea (plain): When keys are small integers (or can be mapped to small integers), count how many times each key appears, then compute positions and write output directly — no comparisons.

Step-by-step process

1. Determine the range of input keys (min and max).
2. Create a count array sized for the key range, initialize to 0.
3. For each element, increment its corresponding count.

4. (Optional for stability) Compute prefix sums of count to get final positions.
5. Place each element into the output array at computed positions, decreasing counts as you place (this gives a stable output).
6. Copy output back if needed.

Complexities

- Time: O(n + k) where k = range size (max-min+1).
- Space: O(k + n) if producing output array (or O(k) extra + in-place copy).

Space: Extra arrays for counts and usually output.
Stability: Can be stable when implemented with prefix sums and placing elements accordingly.
When to use: Excellent for integer keys with small range k relative to n. Not suitable if k is huge compared to n (memory explosion).

6) Bucket Sort

Idea: Distribute elements into several buckets according to value ranges, sort each bucket (often with insertion sort), then concatenate buckets.

Step-by-step process

1. Decide number of buckets b and mapping function (e.g., for values in [0,1), bucket index = floor(value*b)).
2. Create b empty buckets.
3. Scan the array, place each element in its bucket.
4. Sort each bucket using another algorithm (insertion sort if buckets are small).
5. Concatenate buckets in order to produce sorted array.

Complexities

- Average: O(n + b + sum cost of bucket sorts). If keys are uniformly distributed and buckets small, average O(n).

- Worst: $O(n^2)$ if all elements land in one bucket and that bucket is sorted with $O(n^2)$ sort.

Space: O(n + b) extra for buckets.
Stability: Can be stable depending on bucket insertion and internal sort.
When to use: Works well when data is uniformly distributed over a range. Useful as a pre-step for floats or when distribution is known. Choice of b and bucket mapping matters.

7) Radix Sort

Idea: Sort numbers by processing digits from least significant to most significant (LSD radix) or the reverse (MSD radix), using a stable intermediate sort (often counting sort) on each digit.

Step-by-step process (LSD variant)

1. Pick a base B (common: 10 for decimal or 256 for bytes).
2. For digit position d = 0 (least significant) to highest:
   - Use a stable counting sort keyed on digit d.
   - After processing all digits, array is sorted.

Complexities

- Time: O(d*(n + B)) where d is number of digits and B is base size. For fixed-width integers d is constant, so linear O(n).
- Space: O(n + B) for counting arrays and output.

Space: Extra arrays for digit-based counting sort — typically O(n + B).
Stability: Stable (requires stable digit-sort).
When to use: Great for large collections of fixed-size integers or strings of bounded length — gives linear time in practice. Choose base B to balance passes and counting-array size (bigger base -> fewer passes, but larger count array). MSD variant useful when keys vary in length (e.g., strings) but more complex.

Comparison table:

Algorithm	Average Time	Worst Time	Space	Stable	In-place	UseCase
Bubble	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes	Teaching, tiny lists
Selection	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes	Minimal writes needed
Heap	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Yes	Need worst-case guarantees
Quick	$O(n \log n)$	$O(n^2)$	$O(\log n)$ avg	No	Yes	Very fast average-case
Counting	$O(n + k)$	$O(n + k)$	$O(k + n)$	Can be	Usually not needed	Small integer range
Bucket	$O(n)$ expected	$O(n^2)$	$O(n + b)$	Can be	Depends	Uniform distribution
Radix	$O(d*(n + B))$	same	$O(n + B)$	Yes	Usually	Fixed-length integers/strings

****************** END ******************************