Q0: Bubble Sort

1. Problem Understanding

- Goal: Sort an array by repeatedly swapping adjacent elements if they are in the wrong order.
- The largest element "bubbles up" to the end after each full pass.
- After each iteration, one more element at the end becomes sorted.
- **Working Principle**
  - Compare adjacent pairs of elements.
  - If arr[j] > arr[j + 1], swap them.
  - Continue this process for the entire array.
  - After the 1st pass → the largest element reaches the last position.
  - After the 2nd pass → the second-largest is at the second-last position, and so on.
  - Continue until no swaps are needed (array is sorted).
- **Algorithm Steps**
  - Loop i from 0 to n-1
  - Initialize a flag swapped = false
  - Inner loop j from 0 to n - i - 2
    - If arr[j] > arr[j + 1], swap them
    - Set swapped = true
  - If no swaps occurred in the inner loop → break early (array is already sorted)
  - Print or return the sorted array.

---

2. Examples

```
Input: [5, 1, 4, 2, 8]
  Pass 1: [1, 4, 2, 5, 8] → 8 bubbled to end
  Pass 2: [1, 2, 4, 5, 8] → 5 in correct position
  Pass 3: [1, 2, 4, 5, 8] → No swap → stop early
Output: [1, 2, 4, 5, 8]
```

---

3. Approaches

Approach 1:

Java Code:

```java
public class BubbleSort {
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;

            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap elements
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }

            // If no swaps, array is sorted
            if (!swapped) break;
        }
    }
}
```

Complexity (Time & Space):

- Time Complexity:
- Worst Case → $O(n^2)$ (reverse sorted array)
- Best Case → $O(n)$ (already sorted, with swap check)
- Average Case → $O(n^2)$
- Space Complexity: $O(1)$ (in-place sorting)
- Stable: ☑ Yes (equal elements maintain relative order)
- Adaptive: ☑ Yes (can stop early if already sorted)

---

4. Tips & Observations

- Always add a swapped flag for early termination.
- Bubble Sort is mainly for teaching concepts (not used in production).
- Interviewers may ask to optimize or detect early stop condition.
- Understand it well — it helps you grasp in-place sorting and pairwise comparison logic used in many algorithms.

Q1: Selection Sort

---

1. Problem Understanding

- Goal: Sort an array by repeatedly selecting the smallest (or largest) element from the unsorted part and putting it at the beginning.
- At the end of each pass, one element is placed in its correct position.
- Think of it as “selecting” the correct element for each position.
- **Working Principle**
    - Divide the array into two parts: sorted (left) and unsorted (right).
    - Initially, the sorted part is empty, and the unsorted part is the entire array.
    - For each index i:
        - Find the index of the smallest element in the unsorted part.
        - Swap it with the element at index i.
    - Continue until the array is fully sorted.
- **Algorithm Steps**
    - Loop i from 0 to n - 2:
    - Set minIndex = i
    - Loop j from i + 1 to n - 1:
    - If arr[j] < arr[minIndex], update minIndex = j
    - After inner loop, swap arr[i] and arr[minIndex]
    - Repeat until all elements are placed correctly.

---

2. Examples

```
Input: [64, 25, 12, 22, 11]

Pass 1:
Find smallest in [64, 25, 12, 22, 11] → 11
Swap 11 and 64
Array → [11, 25, 12, 22, 64]

Pass 2:
Smallest in [25, 12, 22, 64] → 12
Swap 12 and 25
Array → [11, 12, 25, 22, 64]

Pass 3:
Smallest in [25, 22, 64] → 22
Swap 22 and 25
Array → [11, 12, 22, 25, 64]

Pass 4:
Smallest in [25, 64] → 25
No change
Final: [11, 12, 22, 25, 64]
```

---

3. Approaches

Approach 1:

Java Code:

```
public class SelectionSort {
    public static void selectionSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;

            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap smallest with the first element of unsorted part
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}
```

Complexity (Time & Space):

- Time Complexity:
- Best Case → $O(n^2)$
  - Average Case → $O(n^2)$
  - Worst Case → $O(n^2)$
  - Space Complexity: O(1) (in-place sorting)
- Stable: ❌ No (swapping can break the order of equal elements)
- Adaptive: ❌ No (always performs full passes even if already sorted)

---

4. Tips & Observations

- Ideal for small arrays or when swapping cost > comparison cost.
- Easy to reason about — interviewers often use it to check your understanding of comparison-based sorting.
- Understand why it's not stable — swapping non-adjacent elements breaks order.
- Even if array is sorted, it will still do all passes (non-adaptive).

Q2: Insertion Sort

---

1. Problem Understanding

- Goal: Sort the array by building the sorted portion one element at a time — like how we sort cards in our hand.
- It takes one element from the unsorted part and inserts it into its correct position in the sorted part.
- It's efficient for small or partially sorted arrays.
- **Working Principle**
    - Assume the first element is already sorted.
    - Pick the next element and compare it with elements in the sorted part (to its left).
    - Shift all elements greater than it to the right.
    - Insert the picked element at its correct position.
    - Repeat until the entire array is sorted.
- **Algorithm Steps**
    - Start from index 1 to n - 1 (let the first element be sorted).
    - Store the current element (key = arr[i]).
    - Compare key with elements before it (j = i - 1).
    - While arr[j] > key, shift arr[j] to arr[j + 1].
    - Place the key in the correct position.

---

2. Examples

```
Input: [5, 3, 4, 1, 2]

Step-by-step:

Pass 1 (i=1): key = 3
Compare with 5 → shift 5 → [5, 5, 4, 1, 2]
Insert 3 → [3, 5, 4, 1, 2]

Pass 2 (i=2): key = 4
Compare with 5 → shift 5 → [3, 5, 5, 1, 2]
Compare with 3 → stop → insert 4 → [3, 4, 5, 1, 2]

Pass 3 (i=3): key = 1
Shift 5 → [3, 4, 5, 5, 2]
Shift 4 → [3, 4, 4, 5, 2]
Shift 3 → [3, 3, 4, 5, 2]
Insert 1 → [1, 3, 4, 5, 2]

Pass 4 (i=4): key = 2
Shift 5, 4, 3 → [1, 3, 4, 5, 5], [1, 3, 4, 4, 5], [1, 3, 3, 4, 5]
Insert 2 → [1, 2, 3, 4, 5]

☑ Sorted Array: [1, 2, 3, 4, 5]
```

3. Approaches

Approach 1:

Java Code:

```
public class InsertionSort {
    public static void insertionSort(int[] arr) {
        int n = arr.length;

        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;

            // Move elements greater than key one step ahead
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }

            arr[j + 1] = key;
        }
    }
}
```

Complexity (Time & Space):

- Time Complexity:
  - Best Case → O(n) (already sorted array)
  - Average Case → $O(n^2)$
  - Worst Case → $O(n^2)$ (reverse sorted array)
- Space Complexity: O(1) (in-place)
- Stable: ☑ Yes (does not swap non-adjacent equal elements)
- Adaptive: ☑ Yes (optimized for nearly sorted arrays)

---

4. Tips & Observations

- Excellent for online sorting (can sort data as it arrives).
- Best-case O(n) → very efficient if array is almost sorted.
- Stable: equal elements retain their original order.
- Frequently used in interview warm-ups before moving to Merge/Quick sort.
- Remember: shifting, not swapping — key difference from Bubble/Selection Sort.

Q3: Quick Sort

1. Problem Understanding

- Quick Sort is a divide-and-conquer sorting algorithm.
- Pick a pivot element and partition the array such that:
  - Elements smaller than pivot are on the left
  - Elements larger than pivot are on the right
- Recursively sort left and right subarrays.
- Goal: sort an array in ascending order.

---

2. Constraints

- Array can contain negative and positive integers
- 1 ≤ n ≤ 10^6 (practical limits)
- In-place sorting preferred

---

3. Edge Cases

- Empty array → already sorted
- Single element → already sorted
- All elements equal → pivot selection matters for performance
- Already sorted array → worst-case for naive pivot selection

---

4. Examples

```
Input:

arr = [3, 6, 2, 8, 5]

Output:

[2, 3, 5, 6, 8]
```

---

5. Approaches

Approach 1: Quick Sort using Lomuto Partition

Idea:

- Select last element as pivot.
- Place elements smaller than pivot on left, larger on right.
- Recursively sort left and right subarrays.

Steps:

- If low >= high → return (base case)
- Partition array with pivot
- Recursively quicksort left and right

Java Code:

```
static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int p = partition(arr, low, high);
        quickSort(arr, low, p - 1);
        quickSort(arr, p + 1, high);
    }
}

static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1;
}
Recursion Tree Example (arr = [3, 6, 2]):

[3,6,2]
pivot=2
Left: []      Right: [3,6]
              pivot=6
              Left: [3] Right: []


Sorted result: [2, 3, 6]
```

Complexity (Time & Space):

- Time:
- Average case: O(n log n)
- Worst case (sorted array, bad pivot): O(n^2)
- Space: O(log n) recursion stack

---

6. Justification / Proof of Optimality

- Divides problem recursively → divide-and-conquer
- Works in-place → no extra array needed
- Pivot selection impacts performance → can use random pivot to improve

---

7. Variants / Follow-Ups

- Hoare partition scheme → slightly more efficient in some cases
- Randomized quicksort → choose pivot randomly to avoid worst-case
- 3-way quicksort → handles repeated elements efficiently

---

8. Tips & Observations

- Quick Sort is not stable
- Always check pivot selection for already sorted / reverse sorted arrays
- Small subarrays can be switched to Insertion Sort for efficiency

---

Q4: Merge Sort

1. Problem Understanding

- Merge Sort is a divide-and-conquer sorting algorithm.
- Steps:
  - Divide the array into two halves
  - Recursively sort each half
  - Merge the two sorted halves to get the final sorted array
- Works on any array, including duplicates and negative numbers.

---

2. Constraints

- Array can contain duplicates and negative numbers
- Size: 1 ≤ n ≤ 10^6
- Works on arrays, lists, and linked lists

---

3. Edge Cases

- Empty array → already sorted
- Single element → already sorted
- All elements same → still needs merging

---

4. Examples

```
Input:

arr = [5, 2, 4, 1, 3]


Output:

[1, 2, 3, 4, 5]
```

---

5. Approaches

Approach 1: Quick Sort using Lomuto Partition

Idea:

- Split the array until each subarray has one element
- Merge the sorted subarrays while maintaining order

Java Code:

```java
static void mergeSort(int[] arr, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] L = new int[n1];
    int[] R = new int[n2];

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

```
Recursion Tree Example (arr = [5,2,4,1]):

mergeSort([5,2,4,1])
          /            \
 mergeSort([5,2])   mergeSort([4,1])
     /     \            /     \
 [5]        [2]      [4]      [1]
    \       /          \       /
   merge([5,2])       merge([4,1])
        \               /
        [2,5]        [1,4]
            \        /
         merge([2,5],[1,4])
              [1,2,4,5]
```

Complexity (Time & Space):

- Time: O(n log n) for all cases (best, worst, average)
- Space: O(n) extra space for merging

---

6. Justification / Proof of Optimality

- Always divides array in half → guaranteed log n depth
- Merging is linear → efficient
- Stable sort → preserves relative order of equal elements

---

7. Variants / Follow-Ups

- Iterative Merge Sort (bottom-up)
- Merge Sort on linked lists → can be done in-place with O(1) extra space
- External Merge Sort → for very large datasets on disk

---

8. Tips & Observations

- Very suitable for large datasets
- Good choice when stability is important
- Recursive implementation uses O(log n) stack space

Q67: Array Swaps

1. Problem Understanding

- You're given:
  - An array A of size N.

    - A number X.
- You can swap elements at indices i and j only if
- |i - j| ≥ X.
- Your task is to determine whether it's possible to sort the array in non-decreasing order using such operations (possibly zero).
- Goal:
- Return "YES" if sorting is possible under this constraint, otherwise "NO".

---

2. Constraints

- 1 ≤ X ≤ N ≤ 10^5
- 1 ≤ A[i] ≤ 10^9
- Large N → must be O(N log N) or better.
- The operation constraint limits which indices can interact.

---

3. Edge Cases

- X = 1 → Can swap any two indices → always "YES".
- X = N → No swaps possible → "YES" only if already sorted.
- Array already sorted → "YES", regardless of X.
- X > N/2 → Restricted swaps; some positions can't move freely.
- Duplicate values should not affect the logic.

---

4. Examples

```
Example 1
Input:
3 3
3 2 1
Output:
NO
Explanation:
|i-j| ≥ 3 → no valid swap possible.
Since array isn't sorted → "NO".

Example 2
Input:
5 2
5 1 2 3 4
Output:
YES
Explanation:
You can swap elements with at least distance 2 apart,
which allows rearranging the array to [1, 2, 3, 4, 5].
```

---

5. Approaches

Approach 1: Observation-Based Logic

Idea:

- Depending on X, check how “free” the array elements are to move.
    - When X ≤ N/2:
    - Enough overlapping reachable indices → can rearrange freely → always YES.
    - When X > N/2:
    - Some edges can’t move to the middle → must already be in correct positions.

Steps:

- If X <= N/2: print "YES".
- Else:
    - Copy and sort the array → sortedA.
    - For each index i that cannot be moved freely,
    - i.e. i < X or i > N - X + 1,
    - check if A[i] == sortedA[i].
- If all fixed positions match → "YES", else "NO".

Java Code:

```
public class Solution {
    public static String canBeSorted(int N, int X, int[] A) {
        int[] sorted = A.clone();
        Arrays.sort(sorted);

        // If X <= N / 2, we can always sort
        if (X <= N / 2) return "YES";

        // Otherwise, only check the middle unaffected region
        for (int i = N - X; i < X; i++) {
            if (i >= 0 && i < N && A[i] != sorted[i]) {
                return "NO";
            }
        }
        return "YES";
    }
}
```

Complexity (Time & Space):

- Sorting: O(N log N)
- Checking: O(N)
- Total: O(N log N)
- Space: O(N) (for sorted array)

---

6. Justification / Proof of Optimality

- When X ≤ N/2, segments overlap → full rearrangement possible → always YES.
- When X > N/2, edge elements are fixed because they can't reach beyond |i−j| ≥ X →
- So, only sort possible if fixed parts are already in place.
- This ensures all constraints are respected while still determining sortability.

---

7. Variants / Follow-Ups

- Restricted Swap Distance in Other Forms:
    - |i−j| = X (exact distance swaps only)
    - |i−j| ≤ X (limited local swaps)
- Multi-Segment Sorting Problems:
- Where you can only swap inside certain groups or segments.
- Graph Formulation Variant:
- Each index forms a node, and swap rules define edges — check if all nodes form a connected component to allow full reordering.

---

8. Tips & Observations

- X <= N/2 → Always "YES" — memorize this shortcut ⚡
- Only when X > N/2, we need to compare the edge parts with the sorted version.
- Be careful with 1-based vs 0-based indexing in implementation.
- This problem teaches:
    - How constraints affect sorting ability.
    - How to derive logical shortcuts using symmetry and range reasoning.
- Commonly seen in Codeforces / CodeChef challenges — tests reasoning + array manipulation skills.

Q68: Counting Triplets With Maximum Distance

---

1. Problem Understanding

- You are given:
- An integer N → number of points.
- An array points[] of integers → positions on a number line.
- An integer L → maximum allowed distance.
- You must count the total number of triplets (i, j, k)
- such that the distance between the farthest two points in the triplet
- is ≤ L, i.e.
    - points[k] - points[i] ≤ L (after sorting).

---

2. Constraints

- $1 \le N \le 100$
- $0 \le points[i] \le 1000$
- $1 \le L \le 500$
- Time complexity up to $O(N^2)$ is acceptable.

---

3. Edge Cases

- If $N < 3$ → No triplets possible → return 0.
- All points the same → all triplets valid.
- Very large L → all possible triplets valid.
- Very small L (e.g., 0) → only triplets with same value valid.

---

4. Examples

```
Example 1:
Input
4
2 1 3 4
3
Output
4
Explanation
Sorted array → [1, 2, 3, 4]
Valid triplets (max - min ≤ 3):
{1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}

Example 2:
Input
4
2 1 3 4
2
Output
2
Explanation
Sorted array → [1, 2, 3, 4]
Valid triplets:
{1,2,3}, {2,3,4}
```

---

5. Approaches

Approach 1: Brute Force ($O(N^3)$)

Idea:

- Check every triplet (i, j, k) and see if max - min ≤ L.

Steps:

- Sort the array.
- Loop through all triplets.
- Check condition and count if valid.

Java Code:

```
public static int countTripletsBruteForce(int n, int[] arr, int L) {
    Arrays.sort(arr);
    int count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            for (int k = j + 1; k < n; k++) {
                if (arr[k] - arr[i] <= L) count++;
            }
        }
    }
    return count;
}
```

Complexity (Time & Space):

- Time: $O(N^3)$
- Space: O(1)

Approach 2: Two Pointers / Sliding Window ($O(N^2)$) ☑ (Optimized)

Idea:

- For each starting index i, find the farthest index j
- such that arr[j] - arr[i] ≤ L.
- All elements between them can form valid triplets.

Steps:

- Sort the array.
- Fix i (start).
- Move j until arr[j] - arr[i] > L.
- If there are count = j - i points in range,
    - → triplets = (count - 1) * (count - 2) / 2.

Java Code:

```
public static int countTriplets(int n, int[] arr, int L) {
    Arrays.sort(arr);
    int count = 0;

    for (int i = 0; i < n; i++) {
        int j = i;
```

```
        while (j < n && arr[j] - arr[i] <= L) {
            j++;
        }
        int totalPoints = j - i;
        if (totalPoints >= 3) {
            count += (totalPoints - 1) * (totalPoints - 2) / 2;
        }
    }

    return count;
}
```

Complexity (Time & Space):

- Time: $O(N^2)$
- Space: O(1)

---

6. Justification / Proof of Optimality

- Sorting ensures the difference arr[j] - arr[i] is monotonic,
- allowing an efficient window check.
- The combination formula counts all possible triplets efficiently.
- This approach avoids unnecessary triplet enumeration.

---

7. Variants / Follow-Ups

- Counting Pairs instead of Triplets → similar approach but use count - 1.
- K-sized subsets within a max distance → use combinatorics formula C(count - 1, k - 1).
- 2D or 3D coordinates → use distance formula instead of subtraction.

---

8. Tips & Observations

- Always sort before applying difference-based logic.
- When you fix one point and slide another pointer,
- you often reduce nested loops → $O(N^2)$ → O(N).
- Combinatorial counting (nC2, nC3) is a key trick in such problems.
- For C(n, 3), formula = n*(n-1)*(n-2)/6.
