Q45: Generate All Permutations

1. Understand the Problem

- **Read & Identify:** A permutation of a set of elements is an arrangement of those elements in a particular order

2. Constraints

- 1 <= nums.length <= 10 (to keep factorial manageable).
- Elements may be distinct or duplicate (if duplicates → handle uniqueness).

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
[1, 2, 3]
```

Output:

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

4. Approaches

Approach 1: Brute Force (Backtracking)

- **Idea:**
  - Try all possibilities recursively by placing each unused element.

Java Code:

```
import java.util.*;

class GeneratePermutations {
    public static List<List<Integer>> permuteBacktracking(int[] nums) {
```

```
        List<List<Integer>> res = new ArrayList<>();
        boolean[] used = new boolean[nums.length];
        backtrack(nums, new ArrayList<>(), used, res);
        return res;
    }

    private static void backtrack(int[] nums, List<Integer> curr, boolean[] used,
List<List<Integer>> res) {
        if (curr.size() == nums.length) {
            res.add(new ArrayList<>(curr));
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (!used[i]) {
                used[i] = true;
                curr.add(nums[i]);
                backtrack(nums, curr, used, res);
                curr.remove(curr.size() - 1);
                used[i] = false;
            }
        }
    }
}
```

Complexity:

- Time: O(n! * n)
- Space: O(n! * n)

Approach 2: Optimal (Lexicographic Ordering via Next Permutation)

- **Idea:**
    - Start with sorted array. Use next permutation repeatedly until no more.

Java Code:

```
class GeneratePermutationsLexico {
    public static List<List<Integer>> permuteLexico(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        Arrays.sort(nums);
        res.add(toList(nums));
        while (nextPermutation(nums)) {
            res.add(toList(nums));
        }
        return res;
    }

    private static boolean nextPermutation(int[] nums) {
        int i = nums.length - 2;
        while (i >= 0 && nums[i] >= nums[i+1]) i--;
        if (i < 0) return false;
        int j = nums.length - 1;
        while (nums[j] <= nums[i]) j--;
        swap(nums, i, j);
```

```
        reverse(nums, i+1, nums.length-1);
        return true;
    }

    private static void swap(int[] nums, int i, int j) {
        int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
    }

    private static void reverse(int[] nums, int l, int r) {
        while (l < r) swap(nums, l++, r--);
    }

    private static List<Integer> toList(int[] nums) {
        List<Integer> list = new ArrayList<>();
        for (int num : nums) list.add(num);
        return list;
    }
}
```

Complexity:

- Time: O(n! * n) (same as brute, but structured)
- Space: O(1)

---

5. Justification / Proof of Optimality

- Backtracking is more intuitive and commonly used.
- Lexicographic method is cleaner if permutations must be in sorted order.
- Both are optimal in terms of complexity (O(n! * n) is unavoidable).

---

6. Variants / Follow-Ups

- Permutations with duplicates (need to skip duplicates).
- String permutations.
- K-th permutation (directly find without generating all).

Q46: Next Permutation

---

1. Understand the Problem

- **Read & Identify:** Given array nums, rearrange into next lexicographically greater permutation.

---

2. Constraints

- 1 <= nums.length <= 100

- 0 <= nums[i] <= 100

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
[1, 2, 3]
```

Output:

```
[1,3,2]
```

---

4. Approaches

Approach 1: Brute Force (Generate All, Pick Next)

- **Idea:**
  - Generate all permutations, sort, and pick the next one.

Complexity:

- Time: O(n! * n)
- Space: O(n!)

Approach 2: Optimal In-Place O(n)

- **Idea:**
  - Find pivot index i where nums[i] < nums[i+1].
  - Find rightmost element j > nums[i].
  - Swap nums[i] and nums[j].
  - Reverse i+1 ... end.

Java Code:

```
class NextPermutation {
    public static void nextPermutation(int[] nums) {
        int i = nums.length - 2;
        while (i >= 0 && nums[i] >= nums[i+1]) i--;

        if (i >= 0) {
            int j = nums.length - 1;
            while (nums[j] <= nums[i]) j--;
            swap(nums, i, j);
        }
        reverse(nums, i+1, nums.length-1);
    }

    private static void swap(int[] nums, int i, int j) {
```

```java
        int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
    }

    private static void reverse(int[] nums, int l, int r) {
        while (l < r) {
            swap(nums, l, r);
            l++;
            r--;
        }
    }
}
```

Complexity:

- Time: O(n)
- Space: O(1)

---

5. Justification / Proof of Optimality

- Brute Force is impractical for large n (n! growth).
- Optimal solution achieves result in linear time, which is best possible.

---

6. Variants / Follow-Ups

- Previous permutation.
- K-th next permutation.
- Next permutation with repeated elements.

Q47: 3 Sum, 4 Sum, K Sum

---

1. Problem Understanding

- These problems are part of the K-Sum family, where we find unique combinations of numbers in an array that sum to a target.
- 3 Sum: Find all unique triplets [nums[i], nums[j], nums[k]] such that their sum is 0.
- 4 Sum: Find all unique quadruplets [a, b, c, d] such that their sum equals a given target.
- K Sum: Generalized problem: find all unique combinations of k numbers that sum to a given target.
- Requirements:
- Use distinct indices.
- Avoid duplicates.
- Output combinations in ascending order.

---

2. Constraints

- Array length varies by problem:
- 3 Sum: 1 ≤ n ≤ 3000
- 4 Sum: 1 ≤ n ≤ 200

- K Sum: 1 ≤ n ≤ 200
- Element range: -10^4 ≤ nums[i] ≤ 10^4
- Target range: -10^5 ≤ target ≤ 10^5
- Only valid unique combinations should be returned.

---

3. Edge Cases

- Arrays smaller than k.
- Arrays with all identical numbers.
- Arrays with no valid combination.
- Arrays with negative and positive numbers.
- Duplicates causing repeated results.
- Large positive/negative targets.

---

4. Examples

Example 1:

```
(3 Sum)
Input: nums = [2, -2, 0, 3, -3, 5]
Output: [[-3, 0, 3], [-2, 0, 2], [-3, -2, 5]]
Explanation: Each triplet sums to 0.
```

Example 2:

```
(4 Sum):
Input: nums = [1, -2, 3, 5, 7, 9], target = 7
Output: [[-2, 1, 3, 5]]
Explanation: -2 + 1 + 3 + 5 = 7.
```

Example 3:

```
(K Sum):
Input: nums = [1, 0, -1, 0, -2, 2], k = 4, target = 0
Output: [[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]
```

---

5. Approaches

Approach 1: Brute Force

Idea:

- Try all possible combinations of elements (3 for 3 Sum, 4 for 4 Sum, k for K Sum) and check if their sum equals the target.

- This is straightforward but computationally expensive.

Steps:

- Use nested loops (3 for 3 Sum, 4 for 4 Sum, recursive for K Sum).
- Check if the combination sums to the target.
- Store unique sets after sorting to avoid duplicates.

Java Code:

```
void kSumBruteForce(int[] nums, int k, int target, List<Integer> temp,
List<List<Integer>> res, int start) {
    if (k == 0 && target == 0) {
        res.add(new ArrayList<>(temp));
        return;
    }
    if (k == 0 || start >= nums.length) return;
    for (int i = start; i < nums.length; i++) {
        temp.add(nums[i]);
        kSumBruteForce(nums, k - 1, target - nums[i], temp, res, i + 1);
        temp.remove(temp.size() - 1);
    }
}

// Example usage:
// 3 Sum
List<List<Integer>> res3 = new ArrayList<>();
kSumBruteForce(nums, 3, 0, new ArrayList<>(), res3, 0);

// 4 Sum
List<List<Integer>> res4 = new ArrayList<>();
kSumBruteForce(nums, 4, target, new ArrayList<>(), res4, 0);

// K Sum
List<List<Integer>> resK = new ArrayList<>();
kSumBruteForce(nums, k, target, new ArrayList<>(), resK, 0);
```

Complexity (Time & Space):

- 3 Sum → Time: $O(n^3)$, Space: O(3) for recursion stack
- 4 Sum → Time: $O(n^4)$, Space: O(4) for recursion stack
- K Sum → Time: $O(n^k)$, Space: O(k) for recursion stack

Approach 2: Better / Improved (Sorting + Two Pointers)

Idea:

- Sort the array and use the two-pointer technique to reduce nested loops by one level.
- Fix (k−2) elements, then use two pointers to find the remaining two.

Steps:

- Sort input array.
- Use nested loops to fix first elements.

- Apply two-pointer technique for remaining two.
- Avoid duplicates.

Java Code:

```
List<List<Integer>> kSumBetter(int[] nums, int start, int k, int target) {
    List<List<Integer>> res = new ArrayList<>();
    if (k == 2) { // Base case: 2 Sum
        int left = start, right = nums.length - 1;
        while (left < right) {
            int sum = nums[left] + nums[right];
            if (sum == target) {
                res.add(Arrays.asList(nums[left], nums[right]));
                while (left < right && nums[left] == nums[left + 1]) left++;
                while (left < right && nums[right] == nums[right - 1]) right--;
                left++; right--;
            } else if (sum < target) left++;
            else right--;
        }
        return res;
    }
    for (int i = start; i < nums.length - k + 1; i++) {
        if (i > start && nums[i] == nums[i - 1]) continue;
        for (List<Integer> subset : kSumBetter(nums, i + 1, k - 1, target - nums[i]))
{
            List<Integer> temp = new ArrayList<>();
            temp.add(nums[i]);
            temp.addAll(subset);
            res.add(temp);
        }
    }
    return res;
}
```

Complexity (Time & Space):

- 3 Sum → Time: $O(n^2)$, Space: O(1)
- 4 Sum → Time: $O(n^3)$, Space: O(1)
- K Sum → Time: O(n^(k-1)), Space: O(k) recursion

Approach 3: Optimal / Recursive Generalized K Sum

Idea:

- Recursively reduce K-Sum into smaller subproblems.
- Base case → 2 Sum using two-pointer method.
- Skip duplicates at all levels.

Steps:

- Sort array.
- Recursively call kSum for k-1 elements.
- Merge current element with results from recursion.
- Return all valid combinations.

Java Code:

```
List<List<Integer>> kSumOptimal(int[] nums, int k, int target) {
    Arrays.sort(nums);
    return kSumHelper(nums, 0, k, target);
}

List<List<Integer>> kSumHelper(int[] nums, int start, int k, int target) {
    List<List<Integer>> res = new ArrayList<>();
    if (k == 2) {
        int left = start, right = nums.length - 1;
        while (left < right) {
            int sum = nums[left] + nums[right];
            if (sum == target) {
                res.add(Arrays.asList(nums[left], nums[right]));
                while (left < right && nums[left] == nums[left + 1]) left++;
                while (left < right && nums[right] == nums[right - 1]) right--;
                left++; right--;
            } else if (sum < target) left++;
            else right--;
        }
        return res;
    }
    for (int i = start; i < nums.length - k + 1; i++) {
        if (i > start && nums[i] == nums[i - 1]) continue;
        for (List<Integer> subset : kSumHelper(nums, i + 1, k - 1, target - nums[i]))
{
            List<Integer> temp = new ArrayList<>();
            temp.add(nums[i]);
            temp.addAll(subset);
            res.add(temp);
        }
    }
    return res;
}
```

Complexity (Time & Space):

- 3 Sum → Time: $O(n^2)$, Space: O(1)
- 4 Sum → Time: $O(n^3)$, Space: O(k) recursion
- K Sum → Time: O(n^(k-1)), Space: O(k) recursion

---

6. Justification / Proof of Optimality

- Brute Force: Simple but exponential, impractical for large n.
- Better / Two Pointer: Reduces nested loops, efficient for 3/4 Sum.
- Optimal Recursive: Elegant, generalizes for any K, avoids code repetition, scalable.

---

7. Variants / Follow-Ups

- 2 Sum (sorted/unsorted)

- 3 Sum Closest / 4 Sum Closest
- Count K-Sum combinations
- Return combinations nearest to target
- Use hash maps for optimized 4 Sum II variant

---

8. Tips & Observations

- Always sort the array first.
- Skip duplicates at every recursion/iteration level.
- Use two-pointer approach efficiently for base 2-Sum.
- Recursion stack grows with K; prune impossible paths early.
- Copy lists when adding to results to prevent mutation.

Q48: Pascal's Triangle I, II, III

---

1. Problem Understanding

- Pascal's Triangle I: Return the value at a specific row r and column c (1-indexed).
- Pascal's Triangle II: Return all values in the r-th row (1-indexed).
- Pascal's Triangle III: Return the first n rows of Pascal's Triangle.
- Key differences:
- I → Single value lookup.
- II → Entire row.
- III → Full triangle up to n rows.
- All variants follow the rule:
- Pascal[r][c] = Pascal[r-1][c-1] + Pascal[r-1][c] with 1-indexed rows and columns.

---

2. Constraints

- Pascal's Triangle I → $1 \le r, c \le 30, c \le r$
- Pascal's Triangle II → $1 \le r \le 30$
- Pascal's Triangle III → $1 \le n \le 30$
- All values fit in a 32-bit integer.

---

3. Edge Cases

- c = 1 or c = r → Always 1.
- r = 1 → Only first element.
- Smallest row (r = 1) or first n rows (n = 1).
- Maximum row (r = 30) to test large numbers.
- Empty output (n = 0 or invalid index) → Handle gracefully.

---

4. Examples

Example 1:

```
(Pascal’s Triangle I):
Input: r = 4, c = 2 → Output: 3
Explanation: Row 4 → [1, 3, 3, 1] → value at column 2 = 3
Row 1:        1
Row 2:       1 1
Row 3:      1 2 1
Row 4:     1 3 3 1
```

Example 2:

```
(Pascal’s Triangle II):
Input: r = 5 → Output: [1, 4, 6, 4, 1]
Row 1:        1
Row 2:       1 1
Row 3:      1 2 1
Row 4:     1 3 3 1
Row 5:    1 4 6 4 1
```

Example 3:

```
(Pascal’s Triangle III):
Input: n = 4 → Output: [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]
Row 1:        1
Row 2:       1 1
Row 3:      1 2 1
Row 4:     1 3 3 1
```

---

5. Approaches

Approach 1: Brute Force

Idea:

- Build Pascal’s Triangle row by row until required value, row, or n rows.

Steps:

- Start from first row [1].
- For each subsequent row, compute elements using current[j] = previous[j-1] + previous[j].
- Store or return required value/row/all rows.

Java Code:

```
// Brute Force: Build Pascal Triangle iteratively
class PascalTriangle {
    // Return single value (I), row (II), or all rows (III)
    public int getValue(int r, int c) {
```

```
        List<List<Integer>> triangle = buildTriangle(r);
        return triangle.get(r-1).get(c-1);
    }

    public List<Integer> getRow(int r) {
        List<List<Integer>> triangle = buildTriangle(r);
        return triangle.get(r-1);
    }

    public List<List<Integer>> getRows(int n) {
        return buildTriangle(n);
    }

    private List<List<Integer>> buildTriangle(int n) {
        List<List<Integer>> triangle = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            List<Integer> row = new ArrayList<>();
            for (int j = 0; j <= i; j++) {
                if (j == 0 || j == i) row.add(1);
                else row.add(triangle.get(i-1).get(j-1) + triangle.get(i-1).get(j));
            }
            triangle.add(row);
        }
        return triangle;
    }
}
```

Complexity (Time & Space):

- Pascal I → Time: $O(r^2)$, Space: $O(r^2)$
- Pascal II → Time: $O(r^2)$, Space: $O(r^2)$
- Pascal III → Time: $O(n^2)$, Space: $O(n^2)$

Approach 2: Better / Improved (Row Optimization)

Idea:

- Instead of storing the entire triangle, generate only required row or value.
- Use previous row to compute current row.

Steps:

- Initialize first row [1].
- Iteratively compute next row using only previous row.
- For single value, stop at required column.
- For row, return last computed row.

Java Code:

```
class PascalTriangleOptimized {
    public int getValue(int r, int c) {
        List<Integer> row = getRow(r);
        return row.get(c-1);
    }
```

```
    public List<Integer> getRow(int r) {
        List<Integer> row = new ArrayList<>();
        row.add(1);
        for (int i = 1; i < r; i++) {
            row.add(0); // expand row
            for (int j = i; j > 0; j--) {
                row.set(j, row.get(j) + row.get(j-1));
            }
        }
        return row;
    }

    public List<List<Integer>> getRows(int n) {
        List<List<Integer>> result = new ArrayList<>();
        for (int i = 1; i <= n; i++) {
            result.add(new ArrayList<>(getRow(i)));
        }
        return result;
    }
}
```

Complexity (Time & Space):

- Pascal I → Time: $O(r^2)$, Space: O(r)
- Pascal II → Time: $O(r^2)$, Space: O(r)
- Pascal III → Time: $O(n^2)$, Space: O(r) per row

Approach 3: Optimal / Most Efficient (Binomial Coefficient)

Idea:

- Use combinatorial formula: C(r-1, c-1) = (r-1)! / ((c-1)! * (r-c)!).
- Compute row or value directly using combination formulas.

Steps:

- For Pascal I → Compute single combination.
- For Pascal II → Compute each element using C(r-1, i).
- For Pascal III → Generate each row using combination formula iteratively.

Java Code:

```
class PascalTriangleOptimal {
    public int getValue(int r, int c) {
        return combination(r-1, c-1);
    }

    public List<Integer> getRow(int r) {
        List<Integer> row = new ArrayList<>();
        int val = 1;
        for (int i = 0; i < r; i++) {
            row.add(val);
            val = val * (r-1-i) / (i+1);
```

```
        }
        return row;
    }

    public List<List<Integer>> getRows(int n) {
        List<List<Integer>> triangle = new ArrayList<>();
        for (int i = 1; i <= n; i++) triangle.add(getRow(i));
        return triangle;
    }

    private int combination(int n, int k) {
        int res = 1;
        for (int i = 0; i < k; i++) res = res * (n-i) / (i+1);
        return res;
    }
}
```

Complexity (Time & Space):

- Pascal I → Time: O(c), Space: O(1)
- Pascal II → Time: O(r), Space: O(r)
- Pascal III → Time: $O(n^2)$, Space: O(r) per row

---

6. Justification / Proof of Optimality

- Brute Force: Simple, builds full triangle, high space usage.
- Better / Optimized: Reduces space for single row/value, still $O(r^2)$ time.
- Optimal / Binomial: Fastest for single value or row, uses formula, minimal space.

---

7. Variants / Follow-Ups

- Pascal's Triangle modulo m
- Generate middle element only
- Print triangle upside-down or in other patterns
- Count paths in grid using combinatorial logic

---

8. Tips & Observations

- First and last elements of any row are always 1.
- Each row can be generated iteratively using previous row (space optimized).
- Binomial coefficient formula avoids building entire triangle.
- Use long type if intermediate factorials may exceed 32-bit integers.

Q49: Majority Element I, II, n/k

1. Problem Understanding

- Majority Element I: Return the element that appears more than n/2 times in the array. Guaranteed to exist.
- Majority Element II: Return all elements appearing more than n/3 times. Output can be in any order.
- Majority Element n/k: Generalized problem: return all elements appearing more than n/k times.
- Key differences:
- I → Single element, threshold n/2
- II → Multiple elements, threshold n/3
- n/k → Multiple elements, threshold n/k

---

2. Algorithm

Algorithm 1: Boyer–Moore Voting Algorithm

- The Boyer–Moore Voting Algorithm is a mathematical elimination approach used to find elements that appear more frequently than a certain threshold (like n/2, n/3, or n/k).
- It relies on the principle of pairing and canceling out occurrences of different elements.
- If one element appears more than n/x times, then there can be at most x - 1 such elements.
- **Intuition**
  - Think of each number as a "vote."
  - When two different numbers appear, they cancel each other's votes.
  - The number(s) that remain after all cancellations are potential majority elements.
  - Finally, a verification step ensures these candidates truly exceed the threshold.
- **How It Works**
  - Maintain counters and candidates (up to k - 1 of them if searching for elements > n/k).
  - Iterate through the array:
  - If the current number matches one of the candidates → increment its count.
  - Else if there's an empty slot (count = 0) → assign this number as a new candidate.
  - Else → decrement all existing counters (as this element "cancels out" votes).
  - After one pass, possible majority elements remain.
  - Verify actual frequencies to confirm valid majorities.
- **Example (Conceptual)**
  - Let's say the array is [a, b, a, c, a, b, a].
  - Every time a meets b or c, one of its votes gets canceled.
  - However, since a appears more than half of the time, it cannot be completely eliminated.
  - The final surviving candidate will be a.
- **Key Observations**
  - For n/2 majority, we track 1 candidate.
  - For n/3 majority, we track 2 candidates.
  - For n/k majority, we track (k - 1) candidates.
  - The algorithm generalizes cleanly with the same elimination principle.
- **Complexity**
  - Time Complexity: O(n) — single pass for selection + optional verification.

- Space Complexity: O(1) — constant space for fixed k.

- **Why It's Efficient**
  - Traditional counting methods (like HashMaps) use O(n) space.
  - Boyer–Moore reduces this to O(1) by keeping only a limited number of potential candidates.
  - It leverages mathematical guarantees about frequency limits to ensure correctness.

- **In Summary**
  - Purpose: Find majority elements (> n/x occurrences)
  - Concept: Pairing and canceling out elements
  - Guarantee: At most (x - 1) candidates
  - Phases:
    - Voting (finding potential candidates)
    - Verification (confirming actual frequencies)
  - Advantages: Linear time, constant space, intuitive logic

---

3. Constraints

- n == nums.length
- 1 ≤ n ≤ 10^5
- 2 ≤ n for II and n/k
- -10^4 ≤ nums[i] ≤ 10^4
- Threshold: n/2, n/3, or n/k depending on variant
- Output: sorted ascending if required (II and n/k)

---

4. Edge Cases

- All elements are the same → single majority
- No element meets the threshold → not possible for I, possible for n/k (return empty)
- Array with negative numbers
- Array of size 1
- Elements appear exactly at the threshold boundary
- Multiple elements qualify in n/k or II

---

5. Examples

```
Example 1 (Majority Element I):
Input: nums = [7, 0, 0, 1, 7, 7, 2, 7, 7] → Output: 7
Explanation: 7 appears 5 times in size 9 array → majority

Example 2 (Majority Element II):
Input: nums = [1, 2, 1, 1, 3, 2, 2] → Output: [1, 2]
Explanation: n/3 = 7/3 = 2, elements appearing ≥3 times: [1, 2]

Example 3 (Majority Element n/k, k=4):
Input: nums = [1, 2, 2, 3, 2, 1, 1, 4], k = 4 → Output: [1, 2]
Explanation: n/k = 8/4 = 2, elements appearing >2 times: [1, 2]
```

6. Approaches

Approach 1: Brute Force

Idea:

- Count frequency of each element and check against threshold.

Steps:

- Use a hash map to count occurrences.
- Check elements appearing more than n/2, n/3, or n/k times.
- Return results.

Java Code:

```
class MajorityElementBrute {
    public int majorityElement(int[] nums) { // n/2
        Map<Integer, Integer> count = new HashMap<>();
        for (int num : nums) count.put(num, count.getOrDefault(num, 0) + 1);
        int n = nums.length;
        for (int key : count.keySet()) if (count.get(key) > n/2) return key;
        return -1; // never occurs
    }

    public List<Integer> majorityElementII(int[] nums) { // n/3
        Map<Integer, Integer> count = new HashMap<>();
        for (int num : nums) count.put(num, count.getOrDefault(num, 0) + 1);
        int n = nums.length;
        List<Integer> res = new ArrayList<>();
        for (int key : count.keySet()) if (count.get(key) > n/3) res.add(key);
        Collections.sort(res);
        return res;
    }

    public List<Integer> majorityElementNK(int[] nums, int k) { // n/k
        Map<Integer, Integer> count = new HashMap<>();
        for (int num : nums) count.put(num, count.getOrDefault(num, 0) + 1);
        int n = nums.length;
        List<Integer> res = new ArrayList<>();
        for (int key : count.keySet()) if (count.get(key) > n/k) res.add(key);
        Collections.sort(res);
        return res;
    }
}
```

Complexity (Time & Space):

- Majority Element I → Time: O(n), Space: O(n)
- Majority Element II → Time: O(n), Space: O(n)
- Majority Element n/k → Time: O(n), Space: O(n)

Approach 2: Better / Improved (Sorting)

Idea:

- Sort array and pick elements at threshold index.

Steps:

- Sort array.
- For n/2 → middle element is majority.
- For n/3 or n/k → count elements while traversing to check frequency.

Java Code:

```
class MajorityElementSort {
    public int majorityElement(int[] nums) { // n/2
        Arrays.sort(nums);
        return nums[nums.length/2];
    }

    public List<Integer> majorityElementII(int[] nums) { // n/3
        Arrays.sort(nums);
        List<Integer> res = new ArrayList<>();
        int n = nums.length, count = 1;
        for (int i=1;i<n;i++){
            if(nums[i]==nums[i-1]) count++;
            else {
                if(count>n/3) res.add(nums[i-1]);
                count=1;
            }
        }
        if(count>n/3) res.add(nums[n-1]);
        return res;
    }

    public List<Integer> majorityElementNK(int[] nums,int k){ // n/k
        Arrays.sort(nums);
        List<Integer> res = new ArrayList<>();
        int n=nums.length, count=1;
        for(int i=1;i<n;i++){
            if(nums[i]==nums[i-1]) count++;
            else{
                if(count>n/k) res.add(nums[i-1]);
                count=1;
            }
        }
        if(count>n/k) res.add(nums[n-1]);
        return res;
    }
}
```

Complexity (Time & Space):

- All variants → Time: O(n log n), Space: O(1) or O(n) depending on sort implementation

Approach 3: Optimal / Most Efficient (Boyer-Moore Voting)

Idea:

- Use Boyer-Moore Voting Algorithm for majority elements.
- For n/k → generalized version maintaining k-1 candidates.

Steps:

- I → single candidate.
- II → two candidates.
- n/k → maintain k-1 candidates and counts.
- Verify candidates against threshold.

Java Code:

```
class MajorityElementOptimal {
    public int majorityElement(int[] nums) { // n/2
        int count=0, candidate=0;
        for(int num: nums){
            if(count==0) candidate=num;
            count += (num==candidate)?1:-1;
        }
        return candidate;
    }

    public List<Integer> majorityElementII(int[] nums){ // n/3
        int n = nums.length;
        int cand1=0,cand2=1,count1=0,count2=0;
        for(int num:nums){
            if(num==cand1) count1++;
            else if(num==cand2) count2++;
            else if(count1==0){cand1=num;count1=1;}
            else if(count2==0){cand2=num;count2=1;}
            else{count1--;count2--;}
        }
        List<Integer> res = new ArrayList<>();
        count1=0; count2=0;
        for(int num:nums){
            if(num==cand1) count1++;
            else if(num==cand2) count2++;
        }
        if(count1>n/3) res.add(cand1);
        if(count2>n/3) res.add(cand2);
        Collections.sort(res);
        return res;
    }

    public List<Integer> majorityElementNK(int[] nums,int k){ // n/k
        Map<Integer,Integer> map = new HashMap<>();
        for(int num:nums){
            map.put(num,map.getOrDefault(num,0)+1);
        }
        int n=nums.length;
        List<Integer> res = new ArrayList<>();
```

```
        for(int key: map.keySet()){
            if(map.get(key)>n/k) res.add(key);
        }
        Collections.sort(res);
        return res;
    }
}
```

Complexity (Time & Space):

- Majority Element I → Time: O(n), Space: O(1)
- Majority Element II → Time: O(n), Space: O(1)
- Majority Element n/k → Time: O(n), Space: O(n)

---

7. Justification / Proof of Optimality

- Brute Force → Simple, works but uses extra space.
- Sorting → Reduces space, slightly slower due to O(n log n)
- Boyer-Moore → Best for I & II, minimal space and linear time.
- n/k generalization → Hash map required for multiple candidates, optimal for small k.

---

8. Variants / Follow-Ups

- Majority Element in a stream
- Find elements appearing more than n/4, n/5 times
- Sliding window majority element
- Top K frequent elements

---

9. Tips & Observations

- Threshold-based problems → think in terms of n/2, n/3, n/k.
- Boyer-Moore is extremely efficient for n/2 and n/3.
- Sorting works universally but costs O(n log n).
- For generalized n/k → maximum of k-1 elements can qualify.
- Always verify candidates for n/k to avoid false positives.

Q50: Find the Repeating and Missing Number

---

1. Problem Understanding

- Given an array of size n with numbers from [1, n].
- One number appears twice → repeating number (A).
- One number is missing → missing number (B).
- Goal: Return [A, B].

- Constraint: Cannot modify the original array.
- Key idea: detect duplicate and missing number without altering array, efficiently.

---

2. Constraints

- n == nums.length
- 1 ≤ n ≤ 10^5
- All numbers in nums are in [1, n]
- Exactly one number repeats
- Exactly one number is missing

---

3. Edge Cases

- Array of minimum size n = 2
- Repeating number at the start or end of array
- Missing number at the start or end of array
- Array with consecutive numbers except for missing/repeating
- Only one element repeats (no other duplicates)

---

4. Examples

```
Example 1:
Input: [3, 5, 4, 1, 1]
Output: [1, 2]
Explanation: 1 repeats, 2 is missing

Example 2:
Input: [1, 2, 3, 6, 7, 5, 7]
Output: [7, 4]
Explanation: 7 repeats, 4 is missing

Example 3:
Input: [6, 5, 7, 1, 8, 6, 4, 3, 2]
Output: [6, 9]
Explanation: 6 repeats, 9 is missing
```

---

5. Approaches

Approach 1: Brute Force (HashMap / Counting)

Idea:

- Count frequency of each number using a HashMap
- Identify the number appearing twice → repeating
- Identify the number not present → missing

Steps:

- Create a frequency map of numbers.
- Iterate from 1 to n:
- If count = 2 → repeating
- If count = 0 → missing

Java Code:

```
class FindRepeatingMissingBrute {
    public int[] findRepeatingMissing(int[] nums) {
        int n = nums.length;
        Map<Integer, Integer> map = new HashMap<>();
        for(int num: nums) map.put(num, map.getOrDefault(num, 0)+1);
        int repeating = -1, missing = -1;
        for(int i=1; i<=n; i++){
            if(!map.containsKey(i)) missing = i;
            else if(map.get(i) == 2) repeating = i;
        }
        return new int[]{repeating, missing};
    }
}
```

Complexity (Time & Space):

- Time: O(n), Space: O(n)

Approach 2: Mathematical / Sum & XOR

Idea:

- Use formulas for sum and sum of squares:
- sum = 1 + 2 + ... + n = n*(n+1)/2
- sumSq = 1^2 + 2^2 + ... + n^2 = n*(n+1)*(2n+1)/6
- Let repeating = A, missing = B
- Observed sum difference: sum(nums) - sum = A - B
- Observed sum of squares difference: sumSq(nums) - sumSq = A^2 - B^2 = (A-B)*(A+B)
- Solve the two equations to find A and B.

Java Code:

```
class FindRepeatingMissingMath {
    public int[] findRepeatingMissing(int[] nums){
        int n = nums.length;
        long sum = n*(n+1)/2;
        long sumSq = n*(n+1)*(2*n+1)/6;
        long sumArr = 0, sumSqArr = 0;
        for(int num: nums){
            sumArr += num;
            sumSqArr += (long)num*num;
        }
        long diff = sumArr - sum; // A - B
        long sumDiff = (sumSqArr - sumSq)/diff; // A + B
        int A = (int)((diff + sumDiff)/2);
```

```
        int B = (int)(sumDiff - A);
        return new int[]{A, B};
    }
}
```

Complexity (Time & Space):

- Time: O(n), Space: O(1)

Approach 3: Optimal / XOR Based

Idea:

- XOR all numbers from 1 to n and all elements in array → result = A ^ B
- Find any set bit in XOR → divide numbers into 2 groups → separate A and B
- Efficient and avoids extra space

Java Code:

```
class FindRepeatingMissingXOR {
    public int[] findRepeatingMissing(int[] nums){
        int n = nums.length;
        int xor = 0;
        for(int num: nums) xor ^= num;
        for(int i=1;i<=n;i++) xor ^= i;

        int setBit = xor & -xor;
        int x=0, y=0;
        for(int num: nums){
            if((num & setBit) != 0) x ^= num;
            else y ^= num;
        }
        for(int i=1;i<=n;i++){
            if((i & setBit)!=0) x ^= i;
            else y ^= i;
        }
        // determine which is repeating
        for(int num: nums){
            if(num==x) return new int[]{x, y};
        }
        return new int[]{y, x};
    }
}
```

Complexity (Time & Space):

- Time: O(n), Space: O(1)

---

6. Justification / Proof of Optimality

- Brute Force → simple, works, uses extra space
- Math → elegant, constant space, careful with overflow

- XOR → optimal, constant space, avoids arithmetic overflow

---

7. Variants / Follow-Ups

- Multiple missing numbers or multiple duplicates
- Arrays with multiple constraints (e.g., numbers in 0 to n-1)
- Similar problems like "Single Number" or "Find Duplicate Number"

---

8. Tips & Observations

- XOR trick works because XOR cancels out identical numbers
- Sum & SumSq method leverages simple algebra
- Always check for integer overflow in sum-of-squares for large n
- Brute force is safe but extra memory heavy

Q51: Count Inversions

---

1. Problem Understanding

- Given an integer array nums, count the number of inversions.
- An inversion is a pair (i, j) such that:
  - i < j
  - nums[i] > nums[j]
- Interpretation: measures how far the array is from being sorted.
  - Sorted array → 0 inversions
  - Descending array → maximum inversions

---

2. Constraints

- 1 ≤ nums.length ≤ 10^5
- -10^4 ≤ nums[i] ≤ 10^4

---

3. Edge Cases

- Already sorted array → 0 inversions
- Fully descending array → maximum inversions
- Array with duplicate numbers → count all valid (i, j) pairs
- Single element array → 0 inversions

---

4. Examples

```
Example 1:
Input: [2, 3, 7, 1, 3, 5]
Output: 5
```

```
Explanation (inversions):
(0,3) → 2 > 1
(1,3) → 3 > 1
(2,3) → 7 > 1
(2,4) → 7 > 3
(2,5) → 7 > 5

Example 2:
Input: [-10, -5, 6, 11, 15, 17]
Output: 0
Explanation: already sorted → no inversions

Example 3:
Input: [9, 5, 4, 2]
Output: 6
Explanation: all possible pairs are inversions
```

---

5. Approaches

Approach 1: Brute Force

Idea:

- Check every pair (i, j) with i < j and count if nums[i] > nums[j].

Steps:

- Initialize count = 0
- Iterate i from 0 to n-2
- Iterate j from i+1 to n-1
- If nums[i] > nums[j] → increment count

Java Code:

```
class CountInversionsBrute {
    public long countInversions(int[] nums){
        int n = nums.length;
        long count = 0;
        for(int i=0;i<n-1;i++){
            for(int j=i+1;j<n;j++){
                if(nums[i] > nums[j]) count++;
            }
        }
        return count;
    }
}
```

Complexity (Time & Space):

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

Approach 2: Merge Sort Based (Optimal)

Idea:

- Use modified merge sort to count inversions while merging.
- If nums[i] > nums[j] in merge step → all remaining elements in left half also form inversions with nums[j].

Steps:

- Implement standard merge sort with a merge function
- During merge:
- If left[i] ≤ right[j] → no inversion
- Else → inversion count += remaining elements in left array
- Return total inversions

Java Code:

```java
class CountInversionsMergeSort {
    public long countInversions(int[] nums){
        return mergeSort(nums, 0, nums.length - 1);
    }

    private long mergeSort(int[] nums, int left, int right){
        long inv = 0;
        if(left < right){
            int mid = left + (right-left)/2;
            inv += mergeSort(nums, left, mid);
            inv += mergeSort(nums, mid+1, right);
            inv += merge(nums, left, mid, right);
        }
        return inv;
    }

    private long merge(int[] nums, 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] = nums[left+i];
        for(int j=0;j<n2;j++) R[j] = nums[mid+1+j];

        int i=0,j=0,k=left;
        long inv = 0;
        while(i<n1 && j<n2){
            if(L[i] <= R[j]){
                nums[k++] = L[i++];
            } else {
                nums[k++] = R[j++];
                inv += (n1 - i); // Remaining elements in left are inversions
            }
        }
        while(i<n1) nums[k++] = L[i++];
        while(j<n2) nums[k++] = R[j++];
        return inv;
```

```
    }
}
```

Complexity (Time & Space):

- Time: O(n log n)
- Space: O(n)

---

6. Justification / Proof of Optimality

- Brute Force → simple but slow for large arrays
- Merge Sort → efficient for large arrays, counts inversions during sorting
- Merge Sort is preferred for constraints n ≤ 10^5

---

7. Variants / Follow-Ups

- Count inversions modulo some number
- Count inversions in a streaming array
- Find k-th inversion pair

---

8. Tips & Observations

- Inversions = minimum number of swaps required to sort array
- Merge sort counting uses divide-and-conquer effectively
- Always check for long type if n is large to avoid overflow

Q52: Reverse Pairs

---

1. Problem Understanding

- Given an integer array nums, count the number of reverse pairs.
- A reverse pair (i, j) satisfies:
  - 0 <= i < j < nums.length
  - nums[i] > 2 * nums[j]
- Conceptually, it's similar to counting inversions but with a multiplicative condition.

---

2. Constraints

- 1 <= nums.length <= 5 * 10^4
- -2^31 <= nums[i] <= 2^31 - 1

---

3. Edge Cases

- Already sorted ascending array → 0 reverse pairs
- Fully descending array → potentially maximum reverse pairs

- Array with duplicates → count all valid (i, j) pairs
- Single element → 0 reverse pairs

---

4. Examples

```
Example 1:
Input: [6, 4, 1, 2, 7]
Output: 3
Explanation (reverse pairs):
(0, 2) → 6 > 2*1
(0, 3) → 6 > 2*2
(1, 2) → 4 > 2*1

Example 2:
Input: [5, 4, 4, 3, 3]
Output: 0
Explanation: no valid pairs

Example 3:
Input: [6, 4, 4, 2, 2]
Output: 2
Explanation: pairs (0, 3) and (0, 4)
```

---

5. Approaches

Approach 1: Brute Force

Idea:

- Check every pair (i, j) with i < j and count if nums[i] > 2 * nums[j].

Steps:

- Initialize count = 0
- Iterate i from 0 to n-2
- Iterate j from i+1 to n-1
- If nums[i] > 2 * nums[j] → increment count

Java Code:

```
class ReversePairsBrute {
    public int reversePairs(int[] nums){
        int n = nums.length;
        int count = 0;
        for(int i=0;i<n-1;i++){
            for(int j=i+1;j<n;j++){
                if((long)nums[i] > 2L * nums[j]) count++;
            }
        }
        return count;
```

```
    }
}
```

Complexity (Time & Space):

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

Approach 2: Merge Sort Based (Optimal)

Idea:

- Similar to inversion count using merge sort
- During merge, for each element in left half, count elements in right half satisfying nums[i] > 2 * nums[j]

Steps:

- Implement modified merge sort
- During merge step, before merging, count valid reverse pairs using two pointers
- Merge arrays normally after counting
- Return total count

Java Code:

```
class ReversePairsMergeSort {
    public int reversePairs(int[] nums){
        return mergeSort(nums, 0, nums.length - 1);
    }

    private int mergeSort(int[] nums, int left, int right){
        if(left >= right) return 0;
        int mid = left + (right-left)/2;
        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid+1, right);

        int j = mid+1;
        for(int i=left;i<=mid;i++){
            while(j<=right && (long)nums[i] > 2L * nums[j]) j++;
            count += (j - mid - 1);
        }

        merge(nums, left, mid, right);
        return count;
    }

    private void merge(int[] nums, 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] = nums[left+i];
        for(int j=0;j<n2;j++) R[j] = nums[mid+1+j];
        int i=0,j=0,k=left;
        while(i<n1 && j<n2){
            if(L[i] <= R[j]) nums[k++] = L[i++];
```

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

Complexity (Time & Space):

- Time: O(n log n)
- Space: O(n)

---

6. Justification / Proof of Optimality

- Brute Force → easy but slow for large arrays
- Merge Sort → efficient, counts reverse pairs while sorting
- Merge Sort is necessary for n <= 5 * 10^4 constraints

---

7. Variants / Follow-Ups

- Count reverse pairs modulo some number
- Count reverse pairs in a streaming array
- Generalized condition: nums[i] > k * nums[j]

---

8. Tips & Observations

- Use long when multiplying by 2 to avoid overflow
- Merge sort counting is similar to counting inversions
- Each merge step counts cross-pairs efficiently

Q53: Maximum Product Subarray

1. Problem Understanding

- Given an integer array nums, find the subarray (contiguous elements) with the maximum product.
- Return the product of elements of that subarray.
- A subarray must contain at least one element.
- Must consider negative numbers and zeros carefully because they can change the sign of the product.

---

2. Constraints

- 1 <= nums.length <= 10^4
- -10 <= nums[i] <= 10
- Product of any prefix or suffix is within -10^9 <= product <= 10^9

---

3. Edge Cases

- Array contains zero → product may reset
- Array contains all negative numbers → even length negative subarray may give max product
- Single element array → product = element itself
- All positive numbers → max product = product of whole array
- Array with mix of negatives, zeros, and positives

---

4. Examples

```
Example 1:
Input: [4, 5, 3, 7, 1, 2]
Output: 840
Explanation: Whole array gives maximum product → 45371*2 = 840

Example 2:
Input: [-5, 0, -2]
Output: 0
Explanation: Maximum product subarray is [0] → 0

Example 3:
Input: [1, -2, 3, 4, -4, -3]
Output: 288
Explanation: Maximum product subarray is [3, 4, -4, -3] → 34-4*-3 = 288
```

---

5. Approaches

Approach 1: Brute Force

Idea:

- Generate all possible subarrays and calculate their product.
- Keep track of maximum product found.

Steps:

- Initialize maxProduct = Integer.MIN_VALUE
- Iterate i from 0 to n-1 → start index
- Iterate j from i to n-1 → end index
- Multiply elements from i to j → calculate product
- Update maxProduct if current product is larger

Java Code:

```
class MaxProductSubarrayBrute {
    public int maxProduct(int[] nums){
        int n = nums.length;
        int maxProduct = Integer.MIN_VALUE;
        for(int i=0;i<n;i++){
            int product = 1;
```

```java
            for(int j=i;j<n;j++){
                product *= nums[j];
                maxProduct = Math.max(maxProduct, product);
            }
        }
        return maxProduct;
    }
}
```

Complexity (Time & Space):

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

Approach 2: Dynamic Programming / Prefix and Suffix Scan

Idea:

- Keep track of maximum and minimum product ending at current index
- Negative numbers may turn minimum into maximum and vice versa

Steps:

- Initialize maxProd = nums[0], minProd = nums[0], result = nums[0]
- Iterate i = 1 to n-1:
- If nums[i] is negative → swap maxProd and minProd
- Update maxProd = max(nums[i], nums[i]*maxProd)
- Update minProd = min(nums[i], nums[i]*minProd)
- Update result = max(result, maxProd)

Java Code:

```java
class MaxProductSubarrayDP {
    public int maxProduct(int[] nums){
        int n = nums.length;
        int maxProd = nums[0], minProd = nums[0], result = nums[0];
        for(int i=1;i<n;i++){
            if(nums[i] < 0){
                int temp = maxProd;
                maxProd = minProd;
                minProd = temp;
            }
            maxProd = Math.max(nums[i], nums[i]*maxProd);
            minProd = Math.min(nums[i], nums[i]*minProd);
            result = Math.max(result, maxProd);
        }
        return result;
    }
}
```

Complexity (Time & Space):

- Time: O(n)

- Space: O(1)

Approach 3: Prefix and Suffix Product Scan

Idea:

- Compute prefix product from left and right
- Max product = maximum among all prefix and suffix products
- Reset product to 1 if zero encountered

Steps:

- Initialize maxProduct = Integer.MIN_VALUE, prefix = 1, suffix = 1
- Iterate from left to right → prefix *= nums[i], update maxProduct, reset prefix if zero
- Iterate from right to left → suffix *= nums[i], update maxProduct, reset suffix if zero

Java Code:

```
class MaxProductSubarrayPrefixSuffix {
    public int maxProduct(int[] nums){
        int n = nums.length;
        int maxProduct = Integer.MIN_VALUE;
        int prefix = 1, suffix = 1;
        for(int i=0;i<n;i++){
            prefix = (prefix==0)? nums[i] : prefix*nums[i];
            suffix = (suffix==0)? nums[n-1-i] : suffix*nums[n-1-i];
            maxProduct = Math.max(maxProduct, Math.max(prefix, suffix));
        }
        return maxProduct;
    }
}
```

Complexity (Time & Space):

- Time: O(n)
- Space: O(1)

---

6. Justification / Proof of Optimality

- Brute Force → simple but inefficient, works only for small arrays
- Dynamic Programming → optimal, handles negative numbers and zeros
- Prefix/Suffix Scan → alternative O(n) approach, slightly simpler to code

---

7. Variants / Follow-Ups

- Maximum sum subarray → Kadane's Algorithm
- Maximum product of k elements in an array
- Maximum product subarray with modulo constraints

---

8. Tips & Observations

- Keep track of both max and min products due to negatives
- Zero splits subarrays → reset product calculation
- Carefully handle integer overflow if product exceeds limits
- DP approach is most widely used in interviews

Q54: Merge Two Sorted Arrays Without Extra Space

---

1. Problem Understanding

- Given two sorted arrays nums1 and nums2.
- Merge them in-place into a single sorted array.
- nums1 has enough space to hold all elements (m + n), first m are valid elements, rest are 0s.
- nums2 has n elements.
- Goal: nums1 should contain the merged sorted array without using extra space.

---

2. Constraints

- n == nums2.length
- m + n == nums1.length
- 0 <= n, m <= 1000
- -10^4 <= nums1[i], nums2[i] <= 10^4
- Both arrays are sorted in non-decreasing order

---

3. Edge Cases

- Either array is empty → return the non-empty array
- Arrays contain duplicates → keep all elements
- Negative numbers → ensure correct ordering
- All elements of nums2 are smaller than nums1 → insert at the beginning
- All elements of nums2 are larger than nums1 → insert at the end

---

4. Examples

```
Example 1:
Input: nums1 = [-5, -2, 4, 5], nums2 = [-3, 1, 8]
Output: [-5, -3, -2, 1, 4, 5, 8]

Example 2:
Input: nums1 = [0, 2, 7, 8], nums2 = [-7, -3, -1]
Output: [-7, -3, -1, 0, 2, 7, 8]

Example 3:
Input: nums1 = [1, 3, 5], nums2 = [2, 4, 6, 7]
Output: [1, 2, 3, 4, 5, 6, 7]
```

5. Approaches

Approach 1: Brute Force (Merge and Sort)

Idea:

- Copy elements from nums2 into nums1's extra space
- Sort the entire nums1

Steps:

- Copy nums2 elements into the last n positions of nums1
- Sort nums1 using Arrays.sort()

Java Code:

```
import java.util.Arrays;

class MergeSortedBrute {
    public void merge(int[] nums1, int m, int[] nums2, int n){
        for(int i=0;i<n;i++) nums1[m+i] = nums2[i];
        Arrays.sort(nums1);
    }
}
```

Complexity (Time & Space):

- Time: O((m+n) log(m+n))
- Space: O(1) (in-place sorting)

Approach 2: Two Pointers from End

Idea:

- Use three pointers from the end to avoid overwriting elements
- Compare largest elements from nums1 and nums2 and fill from the end

Steps:

- Initialize i = m-1 (last valid in nums1), j = n-1 (last in nums2), k = m+n-1 (last in nums1)
- While i >= 0 && j >= 0:
    - If nums1[i] > nums2[j] → nums1[k--] = nums1[i--]
    - Else → nums1[k--] = nums2[j--]
- Copy remaining elements of nums2 if any

Java Code:

```
class MergeSortedTwoPointers {
    public void merge(int[] nums1, int m, int[] nums2, int n){
        int i = m-1, j = n-1, k = m+n-1;
```

```
        while(i>=0 && j>=0){
            if(nums1[i] > nums2[j]) nums1[k--] = nums1[i--];
            else nums1[k--] = nums2[j--];
        }
        while(j>=0) nums1[k--] = nums2[j--];
    }
}
```

Complexity (Time & Space):

- Time: O(m + n)
- Space: O(1)

Approach 3: : Gap Method (Shell Sort Inspired, for strict no extra space)

Idea:

- Treat nums1 and nums2 as a single array
- Use gap method to compare and swap elements at distance gap
- Reduce gap until it becomes 1

Steps:

- Initialize gap = ceil((m+n)/2)
- Compare elements at distance gap in combined arrays
- Swap if out of order
- Reduce gap: gap = ceil(gap/2)
- Repeat until gap = 0

Java Code:

```
class MergeSortedGapMethod {
    public void merge(int[] nums1, int m, int[] nums2, int n){
        int total = m+n;
        int gap = (total+1)/2;
        while(gap>0){
            int i=0, j=i+gap;
            while(j<total){
                int val1 = (i<m)? nums1[i] : nums2[i-m];
                int val2 = (j<m)? nums1[j] : nums2[j-m];
                if(val1 > val2){
                    if(i<m && j<m){
                        int temp = nums1[i]; nums1[i]=nums1[j]; nums1[j]=temp;
                    } else if(i<m && j>=m){
                        int temp = nums1[i]; nums1[i]=nums2[j-m]; nums2[j-m]=temp;
                    } else {
                        int temp = nums2[i-m]; nums2[i-m]=nums2[j-m]; nums2[j-m]=temp;
                    }
                }
                i++; j++;
            }
            if(gap==1) gap=0;
            else gap = (gap+1)/2;
        }
```

```
        for(int i=0;i<n;i++) nums1[m+i] = nums2[i];
    }
}
```

Complexity (Time & Space):

- Time: O((m+n) log(m+n))
- Space: O(1)

---

6. Justification / Proof of Optimality

- Brute Force → simple but slower due to sorting
- Two Pointers → optimal and widely used in interviews
- Gap Method → useful for strict in-place merge without extra space

---

7. Variants / Follow-Ups

- Merge k sorted arrays in-place
- Merge sorted linked lists
- Merge arrays with different data types or custom comparator

---

8. Tips & Observations

- Always merge from the end to avoid overwriting elements in nums1
- Gap method is tricky but reduces extra space constraints
- Two pointer method is most practical for interviews
