Q16: Rotate Array by K Places

1. Understand the Problem

- **Paraphrase:** For left rotation: [1,2,3,4,5], k=2 â†' [3,4,5,1,2] For right rotation: [1,2,3,4,5], k=2 â†' [4,5,1,2,3]

---

2. Input, Output, & Constraints

Constraints:

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4
- 0 <= k <= 10^5

---

3. Examples & Edge Cases

Example 1 (Left Rotate:): Input:

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

Output:

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

Example 2 (Right Rotate:): Input:

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

Output:

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

---

4. Approaches

Approach 1: Using Extra Array (Simple)

- **Idea:**
  - For left rotate: Create a new array, place nums[i] at (i - k + n) % n
  - For right rotate: Place nums[i] at (i + k) % n

Java Code:

```
// Left Rotate
public static void leftRotate(int[] nums, int k) {
    int n = nums.length;
    k = k % n;
    int[] res = new int[n];
    for (int i = 0; i < n; i++) {
        res[(i - k + n) % n] = nums[i];
    }
    for (int i = 0; i < n; i++) nums[i] = res[i];
}

// Right Rotate
public static void rightRotate(int[] nums, int k) {
    int n = nums.length;
    k = k % n;
    int[] res = new int[n];
    for (int i = 0; i < n; i++) {
        res[(i + k) % n] = nums[i];
    }
    for (int i = 0; i < n; i++) nums[i] = res[i];
}
```

Complexity:

- Time: O(n)
- Space: O(n) â†' extra array

Approach 2: Using Reverse (Optimal, In-Place)

- **Idea:**
  - Reverse parts of the array to rotate in-place.
  - Left Rotate by k:
  - Reverse first k elements
  - Reverse remaining n-k elements
  - Reverse the whole array
  - Right Rotate by k:
  - Reverse last k elements
  - Reverse first n-k elements
  - Reverse the whole array

Java Code:

```
// Helper to reverse a subarray
private static void reverse(int[] nums, int start, int end) {
    while (start < end) {
        int temp = nums[start];
        nums[start] = nums[end];
        nums[end] = temp;
        start++;
        end--;
    }
}

// Left Rotate
public static void leftRotate(int[] nums, int k) {
    int n = nums.length;
    k = k % n;
    reverse(nums, 0, k - 1);
    reverse(nums, k, n - 1);
    reverse(nums, 0, n - 1);
}

// Right Rotate
public static void rightRotate(int[] nums, int k) {
    int n = nums.length;
    k = k % n;
    reverse(nums, n - k, n - 1);
    reverse(nums, 0, n - k - 1);
    reverse(nums, 0, n - 1);
}
```

Complexity:

- Time: O(n)
- Space: O(1) â†' in-place

---

5. Justification / Proof of Optimality

- Extra array method is intuitive but uses O(n) extra space.
- Reverse method is optimal, in-place, and widely used in interviews.
- Correctly handles k >= n using modulo.

---

6. Variants / Follow-Ups

- Rotate array by one position repeatedly
- Rotate multi-dimensional arrays
- Rotate linked list by k positions
- Rotate circular arrays or arrays with constraints

Q17: Buildings Pattern

1. Understand the Problem

- **Read & Identify:** We are given an array arr of size n. Each element represents the height of a building. We need to print a building-like structure using *.
- **Goal:** Print a 2D pattern where each column represents a building. The number of * in a column equals the value at that index in arr. Columns are tab-separated.
- **Paraphrase:** Imagine a skyline of buildings where height is represented by *. Build the output row by row from top to bottom.

---

2. Input, Output, & Constraints

Constraints:

- 1 â‰¤ n â‰¤ 1000
- 0 â‰¤ arr[i] â‰¤ 1000

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
7
9 3 7 6 2 0 4
```

Output:

```
*
*
*               *
*               *       *
*               *       *
*               *       *                       *
*       *       *       *                       *
*       *       *       *       *               *
*       *       *       *       *               *
```

---

4. Approaches

Approach 1: Iterative Row-by-Row

- **Idea:**
  - Find maxHeight = max(arr)
  - For each row i from maxHeight down to 1:
  - For each column j from 0 to n-1:
  - If arr[j] >= i â†' print *
  - Else â†' print spaces
  - Separate columns with tabs

Java Code:

```
public static void printBuildings(int[] arr) {
    int n = arr.length;
    int maxHeight = 0;
    for (int h : arr) maxHeight = Math.max(maxHeight, h);

    for (int i = maxHeight; i >= 1; i--) {
        for (int j = 0; j < n; j++) {
            if (arr[j] >= i) {
                System.out.print("*\t");
            } else {
                System.out.print("\t");
            }
        }
        System.out.println();
    }
}
```

Complexity:

- Time: O(n * maxHeight) â†' each cell is visited once
- Space: O(1) â†' in-place printing

---

5. Justification / Proof of Optimality

- Correctly prints building heights row-by-row from top to bottom.
- Handles variable heights and tab alignment.
- Efficient for n â‰¤ 1000 and arr[i] â‰¤ 1000.

---

6. Variants / Follow-Ups

- Print rotated skyline (90-degree rotation)
- Print using different symbols or colors for buildings
- Print hollow buildings (outline only)
- Print mirrored buildings (center-aligned)

Q18: Array Adding

1. Understand the Problem

- **Read & Identify:** We are given two arrays, arr1 and arr2, each representing the digits of a number. The arrays may have different sizes.
- **Goal:** Add the two numbers represented by the arrays and return the result as an array of digits.
- **Paraphrase:** Treat arrays as numbers: most significant digit at index 0. Perform addition like elementary school addition, handling carry.

---

2. Constraints

Constraints:

- 0 <= arr1[i], arr2[i] < 10
- Arrays may have different lengths

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
2
2 1
4
1 2 3 4
```

Output:

```
1 2 5 5
```

---

4. Approaches

Approach 1: Simulate Addition from End

- **Idea:**
  - Start from the last index of both arrays
  - Add corresponding digits and carry
  - Store result in a list or array
  - Reverse the result at the end

Java Code:

```
public static int[] addArrays(int[] arr1, int[] arr2) {
    int n1 = arr1.length, n2 = arr2.length;
    int i = n1 - 1, j = n2 - 1;
    int carry = 0;
    java.util.ArrayList<Integer> resList = new java.util.ArrayList<>();

    while (i >= 0 || j >= 0 || carry != 0) {
        int sum = carry;
        if (i >= 0) sum += arr1[i--];
        if (j >= 0) sum += arr2[j--];
        resList.add(sum % 10);
        carry = sum / 10;
    }

    // Reverse the result
    int[] result = new int[resList.size()];
    for (int k = 0; k < resList.size(); k++) {
        result[k] = resList.get(resList.size() - 1 - k);
    }
    return result;
}
```

Complexity:

- Time: O(max(n1, n2)) â†’ each digit processed once
- Space: O(max(n1, n2) + 1) â†’ for result

Approach 2: Convert Arrays to Numbers Using long (Limited)

- **Idea:**
  - Convert arrays to numbers using long
  - Add them
  - Convert the sum back to array

Java Code:

```
static int[] calSum(int a[], int b[], int n, int m) {
    // your code here
    long n1=0,n2=0,res=0;
    for(int i=0;i<n;i++){
        n1=n1*10+(long)a[i];
    }
    for(int i=0;i<m;i++){
        n2=n2*10+(long)b[i];
    }
    res=n1+n2;

    int s=0;
```

```
    long res1=0;
    while(res>0){
      res1=res1*10+res%10;
      res/=10;
      s++;
    }
    int [] z =new int[s];
    for(int i=0;i<s;i++){
      z[i]=(int)(res1%10);
      res1/=10;
    }
    return z;

  }
Limitations:

Fails for large arrays (overflow of long)

Works only if number fits in long
```

Complexity:

- Time: O(n+m+max(n,m)+max(n,m))=O(n+m)
- Space: res1 and other long/int variables â†' O(1) Result array z[] of size s â‰ˆ max(n, m) + 1 â†' O(max(n, m)) âœ… Overall Space Complexity: ð□', (max â□¡ (ð□'› , ð□'š)) O(max(n,m))

---

5. Justification / Proof of Optimality

- Approach 1 is optimal and works for arrays of any size.
- Approach 2 is simpler but limited by long size.
- Both approaches correctly handle carry and array lengths.

---

6. Variants / Follow-Ups

- Subtract two numbers represented by arrays
- Multiply two numbers represented by arrays
- Add multiple arrays of digits
- Handle arrays in reverse order (least significant digit first)

Q19: Array Subtracting

---

1. Understand the Problem

- **Read & Identify:** We are given two arrays, arr1 and arr2, representing digits of two numbers. The arrays may have different sizes.
- **Goal:** Subtract the number represented by arr2 from the number represented by arr1 (arr1 - arr2) and return the result as an array of digits.
- **Paraphrase:** Perform subtraction digit by digit from the end (least significant digit). Handle borrow if needed. Return the resulting number as an array.

---

2. Constraints

- 1 <= n1, n2 <= 10
- 0 <= arr1[i], arr2[i] < 10
- arr1 represents a number greater than or equal to arr2

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
4
1 2 3 4
2
2 1
```

Output:

```
1 2 1 3
```

---

4. Approaches

Approach 1: Digit-by-Digit Subtraction

- **Idea:**
  - Start from the last index of both arrays (least significant digit).
  - Subtract corresponding digits, handling borrow.
  - Store result in a list, reverse at the end, and remove leading zeros.

Java Code:

```java
public static int[] subtractArrays(int[] arr1, int[] arr2) {
    int i = arr1.length - 1;
    int j = arr2.length - 1;
    int borrow = 0;
    java.util.ArrayList<Integer> resList = new java.util.ArrayList<>();
```

```
        while (i >= 0) {
            int diff = arr1[i] - borrow;
            if (j >= 0) diff -= arr2[j--];

            if (diff < 0) {
                diff += 10;
                borrow = 1;
            } else {
                borrow = 0;
            }

            resList.add(diff);
            i--;
        }

        // Remove leading zeros
        while (resList.size() > 1 && resList.get(resList.size() - 1) == 0) {
            resList.remove(resList.size() - 1);
        }

        // Reverse to correct order
        int[] result = new int[resList.size()];
        for (int k = 0; k < resList.size(); k++) {
            result[k] = resList.get(resList.size() - 1 - k);
        }
        return result;
    }
```

Complexity:

- Time: O(n1) â†' each digit processed once
- Space: O(n1) â†' for result

Approach 2: Convert Arrays to Numbers (Limited)

- **Idea:**
  - Convert both arrays to numbers
  - Subtract numbers
  - Convert result back to array
  - Limitation:
  - Works only if numbers fit in int or long.
  - Not recommended for large arrays.

---

5. Justification / Proof of Optimality

- Optimal approach: digit-by-digit subtraction works for any array size.
- Correctly handles borrow and leading zeros.
- Approach 2 is simple but limited by numeric type size.

---

6. Variants / Follow-Ups

- Subtract multiple numbers represented by arrays
- Subtract numbers in reverse order arrays
- Implement array multiplication or division similarly
- Handle negative results if arr2 > arr1

Q20: Maximum Difference Between Two Elements

---

1. Understand the Problem

- **Read & Identify:** We are given an array of positive integers. We need to find the maximum absolute difference between any two elements in the array.
- **Goal:** Find max(|arr[i] - arr[j]|) for all i, j pairs.
- **Paraphrase:** Maximum difference occurs between the largest and smallest element in the array. Thereâ€™s no need to check every pair individually.

---

2. Constraints

- 2 <= n <= 10^6
- 0 < arr[i] <= 10^9
- Elements are positive integers

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
4
16 24 89 35
```

Output:

```
73
Explanation: max(89-16) = 73
```

---

4. Approaches

Approach 1: Optimal Approach (Single Scan)

- **Idea:**
  - Maximum difference is max(arr) - min(arr)
  - Scan array once to find min and max

Java Code:

```
public static long maxDifference(int[] arr) {
    int n = arr.length;
    int minVal = arr[0], maxVal = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] < minVal) minVal = arr[i];
        if (arr[i] > maxVal) maxVal = arr[i];
    }
    return (long)maxVal - (long)minVal;
}
```

Complexity:

- Time: O(n) â†' single pass
- Space: O(1) â†' constant extra space

Approach 2: Brute Force (Check All Pairs)

- **Idea:**
  - Iterate over all pairs (i, j) and compute |arr[i]-arr[j]|
  - Track the maximum

Complexity:

- Time: O(n^2) â†' not feasible for large n
- Space: O(1)

---

5. Justification / Proof of Optimality

- Optimal approach is clearly better: O(n) time, O(1) space, handles large arrays.
- Brute force works for small arrays only and is inefficient.

---

6. Variants / Follow-Ups

- Maximum difference with constraint i < j
- Maximum difference after sorting the array
- Maximum difference for subarrays
- Maximum difference modulo some number

Q21: Shortest Distance Between Two Even Positive Integers

1. Understand the Problem

- **Read & Identify:** We are given an array of integers. We need to find the shortest distance between any two even positive integers.
- **Goal:** Find two even positive integers arr[i] and arr[j] with minimum |i - j|. Return -1 if there is zero or one even positive integer.
- **Paraphrase:** Iterate through the array, track positions of even positive numbers. Calculate distance to the previous even positive number. Keep the minimum distance.

---

2. Constraints

- 2 <= n <= 10^6
- 0 < arr[i] <= 10^9
- Elements are positive integers

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
2
1 2
```

Output:

```
-1
Explanation: Only 1 even positive number.
```

Example 2 (Normal Case): Input:

```
5
2 4 1 6 7
```

Output:

```
```

```
1
Explanation:

Distance(2,4) = 1

Distance(2,6) = 3

Distance(4,6) = 2

Shortest = 1
```

---

4. Approaches

Approach 1: Single Scan (Optimal)

- **Idea:**
  - Keep track of the index of the previous even positive integer.
  - For each even positive integer, compute distance to previous and update minimum.

Java Code:

```java
public static int shortestEvenDistance(int[] arr) {
    int prevIndex = -1;
    int minDist = Integer.MAX_VALUE;

    for (int i = 0; i < arr.length; i++) {
        if (arr[i] > 0 && arr[i] % 2 == 0) { // even positive
            if (prevIndex != -1) {
                minDist = Math.min(minDist, i - prevIndex);
            }
            prevIndex = i;
        }
    }

    return (minDist == Integer.MAX_VALUE) ? -1 : minDist;
}
```

Complexity:

- Time: O(n) â†' single pass
- Space: O(1) â†' only index and minDist

Approach 2: Store All Even Indices (Not Needed)

- **Idea:**
  - Store indices of all even positive numbers in a list
  - Iterate through list to compute distances
  - Limitation:

- Extra space O(k) where k = number of even positive numbers
- Slower in practice for very large n

---

5. Justification / Proof of Optimality

- Optimal approach uses single pass and O(1) space.
- Handles all edge cases: no even numbers, single even number, consecutive even numbers.

---

6. Variants / Follow-Ups

- Shortest distance between odd positive integers
- Shortest distance for all positive numbers divisible by k
- Shortest distance with array in circular form
- Find all pairs with minimum distance instead of just distance

Q22: Sum of Array Except Self

---

1. Understand the Problem

- **Read & Identify:** We are given an array of integers. We need to return a new array such that each element is the sum of all other elements except itself.
- **Goal:** For each index i, calculate sum(arr) - arr[i] and store in result array.
- **Paraphrase:** Compute total sum of array once. Subtract arr[i] from total sum for each index to get output[i]. Avoid nested loops for efficiency.

---

2. Constraints

- 1 <= n <= 5000
- 1 <= arr[i] <= 1000000

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
4
4 3 2 10
```

Output:

```
15 16 17 9
Explanation:

totalSum = 4 + 3 + 2 + 10 = 19

output[0] = 19 - 4 = 15

output[1] = 19 - 3 = 16

output[2] = 19 - 2 = 17

output[3] = 19 - 10 = 9
```

---

4. Approaches

Approach 1: Brute Force

- **Idea:**
  - For each element, iterate through the array and sum all other elements.

Java Code:

```java
public static long[] sumExceptSelfBrute(int[] arr) {
    int n = arr.length;
    long[] result = new long[n];

    for (int i = 0; i < n; i++) {
        long sum = 0;
        for (int j = 0; j < n; j++) {
            if (i != j) sum += arr[j];
        }
        result[i] = sum;
    }
    return result;
}
```

Complexity:

- Time: O(n^2) â†' nested loops
- Space: O(n) â†' result array

Approach 2: Optimal (Total Sum Subtraction)

- **Idea:**
  - Compute total sum of array.
  - For each index, subtract current element from total sum to get output.

Java Code:

```
public static long[] sumExceptSelfOptimal(int[] arr) {
    int n = arr.length;
    long totalSum = 0;
    for (int i = 0; i < n; i++) {
        totalSum += arr[i];
    }

    long[] result = new long[n];
    for (int i = 0; i < n; i++) {
        result[i] = totalSum - arr[i];
    }
    return result;
}
```

Complexity:

- Time: O(n) â†' two passes
- Space: O(n) â†' result array

---

5. Justification / Proof of Optimality

- Optimal approach is better for large n and avoids unnecessary nested loops.
- Brute force is intuitive but inefficient.
- Both approaches handle positive integers correctly.

---

6. Variants / Follow-Ups

- Product of array except self
- Sum except self modulo k
- Arrays containing negative integers
- Sparse arrays with zeros

Q23: Largest Number At Least Twice of Others

---

1. Understand the Problem

- **Read & Identify:** We are given an integer array nums of size n. The largest integer is unique.
- **Goal:** Determine whether the largest number is at least twice as large as every other number in the array. If yes, return the index of the largest number; else return -1.

- **Paraphrase:** Find the largest element and compare it with all other elements. Check if it is â‰¥ 2 Ã— any other element.

---

2. Constraints

- 1 <= n <= 50
- 0 <= nums[i] <= 100
- Largest number in array is unique

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
4
3 6 1 0
```

Output:

```
1
Explanation: 6 â‰¥ 2 Ã— 3, 6 â‰¥ 2 Ã— 1, 6 â‰¥ 2 Ã— 0 â†’ condition satisfied
```

Example 2 (Normal Case): Input:

```
4
1 2 3 4
```

Output:

```
-1
Explanation: 4 < 2 Ã— 3 â†’ condition fails
```

---

4. Approaches

Approach 1: Brute Force

- **Idea:**
  - Find the largest element and its index.
  - Iterate through the array and check if largest >= 2 Ã— arr[i] for all other elements.

Java Code:

```java
public static int largestAtLeastTwiceBrute(int[] nums) {
    int n = nums.length;
    int maxVal = nums[0];
    int maxIndex = 0;

    // Find largest element
    for (int i = 1; i < n; i++) {
        if (nums[i] > maxVal) {
            maxVal = nums[i];
            maxIndex = i;
        }
    }

    // Check condition
    for (int i = 0; i < n; i++) {
        if (i != maxIndex && maxVal < 2 * nums[i]) {
            return -1;
        }
    }

    return maxIndex;
}
```

Complexity:

- Time: O(n) â†’ two passes over array
- Space: O(1) â†’ constant space

Approach 2: Optimal (Single Pass)

- **Idea:**
  - Track both the largest and second largest elements in a single pass.
  - Compare largest with 2 Ã— second largest.

Java Code:

```java
public static int largestAtLeastTwiceOptimal(int[] nums) {
    int n = nums.length;
    int max1 = -1, max2 = -1, maxIndex = -1;

    for (int i = 0; i < n; i++) {
        if (nums[i] > max1) {
            max2 = max1;
            max1 = nums[i];
            maxIndex = i;
        } else if (nums[i] > max2) {
            max2 = nums[i];
        }
    }
```

```
        return (max1 >= 2 * max2) ? maxIndex : -1;
    }
```

Complexity:

- Time: O(n) â†' single pass
- Space: O(1) â†' constant space

---

5. Justification / Proof of Optimality

- Brute force is simple, intuitive, and works well for small n (â‰¤50).
- Optimal approach reduces the comparison step by tracking the second largest element â†' still O(n) but slightly faster and elegant.

---

6. Variants / Follow-Ups

- Largest number at least k times others
- Largest number at least twice for subarrays
- Return all numbers that satisfy condition instead of just largest
- Extend to non-unique largest elements

Q24: Subarray Sum Divisible by k

---

1. Understand the Problem

- **Read & Identify:** We are given an integer array nums and an integer k. We need to find the number of non-empty contiguous subarrays whose sum is divisible by k.
- **Goal:** Count all subarrays nums[i..j] such that (sum(nums[i..j]) % k) == 0.
- **Paraphrase:** Check every possible subarray for divisibility by k. Optimize using prefix sum and modulo properties.

---

2. Constraints

- 1 <= n <= 5000
- -10^4 <= nums[i] <= 10^4
- 2 <= k <= 10^4

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
6 5
4 5 0 -2 -3 1
```

Output:

```
7
Explanation:

Subarrays divisible by 5: [4,5,0,-2,-3,1], [5], [5,0], [5,0,-2,-3], [0],
[0,-2,-3], [-2,-3]
```

Example 2 (Normal Case): Input:

```
4 2
4 5 0 -2
```

Output:

```
4
Explanation:

Subarrays divisible by 2: [4], [0], [0,-2], [-2]
```

---

4. Approaches

Approach 1: Brute Force

- **Idea:**
  - Iterate over all subarrays (i..j)
  - Compute sum and check divisibility by k

Java Code:

```java
public static int subarraySumDivisibleByKBrute(int[] nums, int k) {
    int n = nums.length;
    int count = 0;
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
            sum += nums[j];
            if (sum % k == 0) count++;
```

```
            }
        }
        return count;
    }
```

Complexity:

- Time: O(n^2) â†' nested loops
- Space: O(1) â†' constant space

Approach 2: Optimal (Prefix Sum + HashMap)

- **Idea:**
  - Use prefix sum and modulo property: (sum[j] - sum[i-1]) % k == 0 â†' sum[j] % k == sum[i-1] % k
  - Track frequency of modulo values in a hashmap
  - Count pairs with same modulo

Java Code:

```
import java.util.*;

public static int subarraySumDivisibleByKOptimal(int[] nums, int k) {
    Map<Integer, Integer> modCount = new HashMap<>();
    modCount.put(0, 1); // empty prefix
    int sum = 0;
    int count = 0;

    for (int num : nums) {
        sum += num;
        int mod = ((sum % k) + k) % k; // handle negative numbers
        count += modCount.getOrDefault(mod, 0);
        modCount.put(mod, modCount.getOrDefault(mod, 0) + 1);
    }

    return count;
}
```

Complexity:

- Time: O(n) â†' single pass
- Space: O(k) â†' store modulo frequencies

---

5. Justification / Proof of Optimality

- Brute force is simple but O(nÂ²) â†' inefficient for large n.
- Optimal approach leverages prefix sum + modulo properties â†' O(n) time.
- Correctly handles negative numbers and large arrays.

---

6. Variants / Follow-Ups

- Subarray sum divisible by any number k
- Count subarrays with sum divisible by k and of length â‰¥ m
- Count continuous subarrays with sum exactly equal to k

Q25: Find Geometric Triplets

---

1. Understand the Problem

- **Read & Identify:** We are given a sorted array of integers. We need to find all triplets (a, b, c) such that they form a geometric progression (GP).
- **Goal:** Print all triplets (arr[i], arr[j], arr[k]) where arr[j]/arr[i] == arr[k]/arr[j] â†' common ratio same. Output triplets in lexicographic order (sorted by first, then second, then third element).
- **Paraphrase:** For every possible triplet in the array, check if it forms a geometric progression. Print them sorted automatically since the array is already sorted.

---

2. Constraints

- 1 <= N <= 10
- Array elements are positive integers

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
6
1 2 6 10 18 54
```

Output:

```
2 6 18
6 18 54
```

Example 2 (Normal Case): Input:

```
8
2 8 10 15 16 30 32 64
```

Output:

```
2 8 32
8 16 32
16 32 64
```

---

4. Approaches

Approach 1: Brute Force

- **Idea:**
  - Iterate all triplets (i < j < k) using 3 nested loops
  - Check if arr[j]^2 == arr[i] * arr[k] â†' avoids floating point division
  - Print valid triplets

Java Code:

```
public static void findGPTripletsBrute(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 2; i++) {
        for (int j = i + 1; j < n - 1; j++) {
            for (int k = j + 1; k < n; k++) {
                if ((long)arr[j] * arr[j] == (long)arr[i] * arr[k]) {
                    System.out.println(arr[i] + " " + arr[j] + " " + arr[k]);
                }
            }
        }
    }
}
```

Complexity:

- Time: O(nÂ³) â†' three nested loops
- Space: O(1) â†' no extra space

Approach 2: Optimal (Using Hashing / Map)

- **Idea:**
  - Use a HashSet for fast lookup
  - For each pair (i, j) as first two elements, compute expectedThird = arr[j]*arr[j]/arr[i]
  - Check if expectedThird exists in the array using HashSet

Java Code:

```
import java.util.*;

public static void findGPTripletsOptimal(int[] arr) {
    int n = arr.length;
    Set<Integer> set = new HashSet<>();
    for (int num : arr) set.add(num);

    for (int i = 0; i < n - 2; i++) {
        for (int j = i + 1; j < n - 1; j++) {
            if ((arr[j] * arr[j]) % arr[i] == 0) { // ensure integer division
                int expectedThird = (arr[j] * arr[j]) / arr[i];
                if (set.contains(expectedThird) && expectedThird > arr[j]) {
                    System.out.println(arr[i] + " " + arr[j] + " " +
expectedThird);
                }
            }
        }
    }
}
```

Complexity:

- Time: O(nÂ²) â†' iterate all pairs (i, j)
- Space: O(n) â†' for HashSet lookup

---

5. Justification / Proof of Optimality

- Brute force is simple but O(nÂ³) â†' acceptable since n <= 10.
- Optimal approach reduces complexity using a HashSet lookup for third element â†' O(nÂ²).
- Lexicographic order is naturally satisfied because the array is already sorted.

---

6. Variants / Follow-Ups

- Count number of GP triplets instead of printing
- Triplets for negative numbers or zeros
- Triplets with common ratio = r given, instead of any ratio
- Longest GP subsequence instead of triplets

Q26: Maximum Sum Subarray

---

1. Understand the Problem

- **Read & Identify:** We are given an integer array arr. We need to find the maximum sum of any contiguous subarray.
- **Goal:** Return the sum of the subarray that has the largest sum among all contiguous subarrays.
- **Paraphrase:** Find subarray [i..j] with sum arr[i]+arr[i+1]+...+arr[j] maximized.

---

2. Constraints

- 1 <= n <= 10^4
- -100 <= arr[i] <= 100

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
5
2 3 1 -1 0
```

Output:

```
6
Explanation: Maximum subarray sum = 2 + 3 + 1 = 6
```

Example 2 (Normal Case): Input:

```
8
-2 -3 4 -1 -2 1 5 -3
```

Output:

```
7
Explanation: Maximum subarray sum = 4 + -1 + -2 + 1 + 5 = 7
```

---

4. Approaches

Approach 1: Brute Force

- **Idea:**
  - Check all possible subarrays
  - Calculate sum for each subarray
  - Track maximum sum

Java Code:

```
public static int maxSubarraySumBrute(int[] arr) {
    int n = arr.length;
    int maxSum = Integer.MIN_VALUE;

    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
            sum += arr[j];
            if (sum > maxSum) maxSum = sum;
        }
    }

    return maxSum;
}
```

Complexity:

- Time: O(nÂ²) â†' nested loops
- Space: O(1) â†' constant space

Approach 2: Optimal (Kadaneâ€™s Algorithm)

- **Idea:**
  - Maintain a running sum (currentSum)
  - If currentSum < 0, reset to 0
  - Track maximum sum seen so far

Java Code:

```
public static int maxSubarraySumOptimal(int[] arr) {
    int maxSum = arr[0];
    int currentSum = arr[0];

    for (int i = 1; i < arr.length; i++) {
        currentSum = Math.max(arr[i], currentSum + arr[i]);
        maxSum = Math.max(maxSum, currentSum);
    }

    return maxSum;
}
```

Complexity:

- Time: O(n) â†' single pass
- Space: O(1) â†' constant space

Approach 3: 2b: Optimal with Long (Handling Overflow)

- **Idea:**
  - Use long for sum and maxi to avoid overflow with large integers
  - Same logic as Kadaneâ€™s algorithm: maintain running sum, reset if negative

Java Code:

```
public static int maxSubarraySumOptimalLong(int[] nums) {
    long maxi = Long.MIN_VALUE;
    long sum = 0;

    for (int i = 0; i < nums.length; i++) {
        sum += nums[i];   // safe sum using long
        if (sum > maxi) maxi = sum;

        if (sum < 0) sum = 0;
    }

    return (int) maxi;
}
```

Complexity:

- Time: O(n) â†' single pass
- Space: O(1) â†' constant space, Safe for large integers â†' avoids int overflow

---

5. Justification / Proof of Optimality

- Brute force is simple but O(nÂ²) â†' inefficient for large arrays
- Kadaneâ€™s algorithm works in O(n) and handles negative numbers correctly
- Correctly identifies subarray with maximum sum without storing all subarrays

---

6. Variants / Follow-Ups

- Maximum product subarray
- Maximum sum subarray of length exactly k
- Maximum sum subarray in circular array
- Return start and end indices of maximum subarray

Q27: Divisible Sum Pairs

---

1. Understand the Problem

- **Read & Identify:** We are given an integer array arr and a positive integer k. We need to find all pairs (i, j) with i < j such that (arr[i] + arr[j]) % k == 0.
- **Goal:** Count all valid pairs where the sum is divisible by k.
- **Paraphrase:** Iterate through all possible pairs (i,j) with i < j. Check divisibility and count the pairs. Optimize using modulo frequencies.

---

2. Constraints

- 1 <= n <= 10^3
- 1 <= arr[i] <= 10^6
- 1 <= k <= 10^6

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
6 3
1 3 2 6 1 2
```

Output:

```
5
Explanation:
Valid pairs: (0,2),(0,5),(1,3),(2,4),(4,5)
```

Example 2 (Normal Case): Input:

```
4 5
1 3 2 6
```

Output:

```
1
Explanation:
Valid pair: (1,2)
```

---

4. Approaches

Approach 1: Brute Force

- **Idea:**

- Iterate all pairs (i,j) with i<j
- Check (arr[i]+arr[j]) % k == 0
- Count valid pairs

Java Code:

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

Complexity:

- Time: O(nÂ²) â†' nested loops
- Space: O(1) â†' constant

Approach 2: Optimal (Using Modulo Frequency)

- **Idea:**
  - Use frequency array of mod k values
  - For each element, compute mod = arr[i] % k
  - Number of valid pairs = frequency of (k - mod) % k seen so far

Java Code:

```
public static int divisibleSumPairsOptimal(int[] arr, int k) {
    int[] freq = new int[k];
    int count = 0;

    for (int num : arr) {
        int mod = num % k;
        int complement = (k - mod) % k; // pair sum divisible by k
        count += freq[complement];
        freq[mod]++;
    }

    return count;
}
```

Complexity:

- Time: O(n) â†' single pass

- Space: O(k) â†’ store modulo frequencies

---

5. Justification / Proof of Optimality

- Brute force is simple but O(nÂ²) â†’ acceptable only for small n
- Optimal approach leverages modulo complement counting â†’ O(n) and safe for larger arrays
- Correctly counts all (i,j) with i < j without checking all pairs explicitly

---

6. Variants / Follow-Ups

- Count pairs with sum divisible by any number k
- Find all actual pairs instead of count
- Consider triplets instead of pairs
- Allow negative numbers and handle modulo correctly

Q35: Second Largest Element in an Array

---

1. Understand the Problem

- **Paraphrase:** Traverse the array once (or optimally) to determine the largest and second-largest distinct elements. If only one distinct element exists, return -1.

---

2. Constraints

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4
- Array may contain duplicate elements.

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
[8, 8, 7, 6, 5]
```

Output:

```
7
```

Example 2 (Normal Case): Input:

```
[10, 10, 10, 10, 10]
```

Output:

```
-1
```

---

4. Approaches

Approach 1: Brute Force (Sort + Unique)

- **Idea:**
  - Remove duplicates.
  - Sort array descending.
  - Return second element if exists, else -1.

Java Code:

```java
public static int secondLargestBrute(int[] nums) {
    TreeSet<Integer> set = new TreeSet<>();
    for(int num : nums) set.add(num); // automatically sorts and keeps unique
    if(set.size() < 2) return -1;
    set.pollLast(); // remove largest
    return set.last(); // second largest
}
```

Complexity:

- Time: O(n log n) (due to sorting)
- Space: O(n) (for storing unique elements)

Approach 2: Optimal (Single Pass)

- **Idea:**
  - Track two variables: largest and secondLargest.
  - Traverse the array once.
  - Update largest and secondLargest as you encounter higher values.

Java Code:

```java
public static int secondLargestOptimal(int[] nums) {
    int largest = Integer.MIN_VALUE;
    int secondLargest = Integer.MIN_VALUE;
```

```
        for(int num : nums){
            if(num > largest){
                secondLargest = largest;
                largest = num;
            } else if(num < largest && num > secondLargest){
                secondLargest = num;
            }
        }
        return secondLargest == Integer.MIN_VALUE ? -1 : secondLargest;
    }
```

Complexity:

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

---

5. Justification / Proof of Optimality

- Optimal approach traverses the array once (O(n)) and does not require extra space, making it better than the sorting approach.
- Handles duplicates and negative values correctly.

---

6. Variants / Follow-Ups

- Find third-largest, fourth-largest, etc. â†' can extend with more variables.
- Handle kth largest element using min-heap for large arrays.
- Return indices of the largest and second-largest elements instead of values.

Q36: Remove Duplicates from Sorted Array

---

1. Understand the Problem

- **Read & Identify:** Given a sorted array, remove duplicates in-place so that each unique element appears only once.
- **Goal:** Return the count of unique elements and modify the array so that the first k elements contain the unique values in order.
- **Paraphrase:** Iterate through the array, skip duplicates, and shift unique elements to the front.

---

2. Constraints

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4
- Array is sorted in non-decreasing order.

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
[0, 0, 3, 3, 5, 6]
```

Output:

```
4, Array = [-2, 2, 4, 5, _, _, _, _]
```

Example 2 (Normal Case): Input:

```
[-30, -30, 0, 0, 10, 20, 30, 30]
```

Output:

```
5, Array = [-30, 0, 10, 20, 30, _, _, _]
```

---

4. Approaches

Approach 1: Brute Force (Using Extra Array)

- **Idea:**
  - Traverse the array and add unique elements to a new array.
  - Copy the unique elements back to the original array.

Java Code:

```java
public static int removeDuplicatesBrute(int[] nums) {
    int n = nums.length;
    if(n == 0) return 0;
    int[] temp = new int[n];
    int index = 0;
    temp[index++] = nums[0];
    for(int i = 1; i < n; i++){
        if(nums[i] != nums[i-1]){
            temp[index++] = nums[i];
```

```
            }
        }
        for(int i = 0; i < index; i++){
            nums[i] = temp[i];
        }
        return index;
    }
```

Complexity:

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

Approach 2: Optimal (Two Pointers, In-place)

- **Idea:**
  - Use two pointers: i for iteration, j for placing unique elements.
  - If nums[i] != nums[i-1], copy nums[i] to nums[j].
  - Return j+1 as count of unique elements.

Java Code:

```
public static int removeDuplicatesOptimal(int[] nums) {
    if(nums.length == 0) return 0;
    int j = 0; // pointer for placing unique elements
    for(int i = 1; i < nums.length; i++){
        if(nums[i] != nums[j]){
            j++;
            nums[j] = nums[i];
        }
    }
    return j + 1; // number of unique elements
}
```

Complexity:

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

---

5. Justification / Proof of Optimality

- Optimal approach only traverses the array once and does not require extra space.
- Preserves relative order of unique elements.
- Works for negative numbers, duplicates, and arrays of size 1.

---

6. Variants / Follow-Ups

- Remove duplicates from unsorted array â†' requires HashSet or sorting first.
- Count frequency of unique elements while removing duplicates.
- Return array of unique elements instead of modifying in-place.

Q37: Find Missing Number

1. Understand the Problem

- **Read & Identify:** Given an array of distinct integers of size n containing values from 0 to n inclusive, one number is missing in the range.
- **Paraphrase:** The array has n elements, but the complete range 0..n has n+1 elements. Identify the number not present in the array.

---

2. Constraints

- n == nums.length
- 1 <= n <= 10^4
- 0 <= nums[i] <= n
- All numbers in nums are unique.

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

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

Output:

```
0
```

Example 2 (Normal Case): Input:

```
[0, 2, 3, 1, 4]
```

Output:

```
5
```

Example 3 (Normal Case): Input:

```
[0, 1, 2, 4, 5, 6]
```

Output:

```
3
```

---

4. Approaches

Approach 1: Brute Force (Check Each Number)

- **Idea:**
  - Traverse numbers 0..n.
  - For each number, check if it exists in the array.
  - Return the number which is missing.

Java Code:

```java
public static int missingNumberBrute(int[] nums) {
    int n = nums.length;
    for(int num = 0; num <= n; num++){
        boolean found = false;
        for(int i = 0; i < n; i++){
            if(nums[i] == num){
                found = true;
                break;
            }
        }
        if(!found) return num;
    }
    return -1; // should not happen
}
```

Complexity:

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

Approach 2: Optimal (Sum Formula / XOR)

- **Idea:**

- Idea 1 (Sum Formula):
- Sum of numbers from 0..n = n*(n+1)/2.
- Subtract sum of elements in nums.
- Remaining value = missing number.
- Idea 2 (XOR):
- XOR all numbers 0..n.
- XOR all elements in nums.
- XOR of the two results = missing number (because duplicates cancel out).

Java Code:

```
Java Code (Sum Formula):

public static int missingNumberOptimal(int[] nums) {
    int n = nums.length;
    int sum = n * (n + 1) / 2;
    int arrSum = 0;
    for(int num : nums){
        arrSum += num;
    }
    return sum - arrSum;
}


Java Code (XOR):

public static int missingNumberXOR(int[] nums) {
    int n = nums.length;
    int xor = 0;
    for(int i = 0; i <= n; i++){
        xor ^= i;
    }
    for(int num : nums){
        xor ^= num;
    }
    return xor;
}
```

Complexity:

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

---

5. Justification / Proof of Optimality

- Optimal approaches are O(n) and O(1) extra space.
- Works regardless of array order.
- Sum formula may risk integer overflow for large n, XOR avoids overflow.

---

6. Variants / Follow-Ups

- Array may have duplicates â†' find missing number or repeated numbers.
- Numbers not in range [0, n] â†' adjust formula or XOR range.
- Find all missing numbers in [0, n] if multiple missing.

Q38: Union of Two Sorted Arrays

1. Understand the Problem

- **Paraphrase:** Combine two sorted arrays into one sorted array containing all distinct elements.

---

2. Constraints

- 1 <= n, m <= 10^5
- -10^6 <= arr1[i], arr2[i] <= 10^6
- Arrays are sorted.

---

3. Examples & Edge Cases

Example 1 (Edge Case): Input:

```
arr1 = [], arr2 = [1, 2, 3]
```

Output:

```
0[1, 2, 3]
```

Example 2 (Normal Case): Input:

```
arr1 = [1, 2, 3], arr2 = [2, 3, 4]
```

Output:

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

---

4. Approaches

Approach 1: Brute Force (Merge & Set)

- **Idea:**
  - Merge both arrays into one array.
  - Use a HashSet to remove duplicates.
  - Convert HashSet to sorted array.

Java Code:

```
public static int[] unionBrute(int[] arr1, int[] arr2) {
    Set<Integer> set = new HashSet<>();
    for(int num : arr1) set.add(num);
    for(int num : arr2) set.add(num);
    int[] result = set.stream().sorted().mapToInt(i -> i).toArray();
    return result;
}
```

Complexity:

- Time: O((n + m) log(n + m)) â†' sorting after merge.
- Space: O(n + m) â†' for merged array and HashSet.

Approach 2: Optimal (Two Pointer Technique)

- **Idea:**
  - Use two pointers i and j for arr1 and arr2.
  - Compare elements:
  - If arr1[i] < arr2[j] â†' add arr1[i] and move i.
  - If arr1[i] > arr2[j] â†' add arr2[j] and move j.
  - If equal â†' add one element and move both pointers.
  - Skip duplicates by checking the last element added.
  - Add remaining elements from both arrays.

Java Code:

```
public static List<Integer> unionOptimal(int[] arr1, int[] arr2) {
    List<Integer> union = new ArrayList<>();
    int i = 0, j = 0;
    while(i < arr1.length && j < arr2.length){
        int val;
        if(arr1[i] < arr2[j]){
            val = arr1[i++];
        } else if(arr1[i] > arr2[j]){
            val = arr2[j++];
        } else {
            val = arr1[i];
```

```
            i++; j++;
        }
        if(union.isEmpty() || union.get(union.size() - 1) != val){
            union.add(val);
        }
    }
    while(i < arr1.length){
        if(union.get(union.size() - 1) != arr1[i])
            union.add(arr1[i]);
        i++;
    }
    while(j < arr2.length){
        if(union.get(union.size() - 1) != arr2[j])
            union.add(arr2[j]);
        j++;
    }
    return union;
}
```

Complexity:

- Time: O(n + m) â†' single pass through both arrays.
- Space: O(n + m) â†' for result array.

---

5. Justification / Proof of Optimality

- Optimal approach avoids sorting after merge â†' faster O(n+m)
- Maintains sorted order naturally.
- Handles duplicates efficiently.

---

6. Variants / Follow-Ups

- Intersection of two sorted arrays â†' instead of union.
- k-sorted arrays union â†' merge k arrays using priority queue.
- Unsorted arrays â†' sort first or use HashSet.
- Find union without extra space if arrays are mutable.

Q39: Leaders in an Array

---

1. Understand the Problem

- **Read & Identify:** Given an array of integers, identify elements that are strictly greater than all elements to their right.

- **Goal:** Return a list of all such leaders in the order they appear in the array.
- **Paraphrase:** : A leader is any element that is bigger than all elements on its right, including the last element (which is always a leader).

---

2. Constraints

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

---

3. Examples & Edge Cases

Example 1 (Edge Case): Input:

```
[10]
```

Output:

```
[10]
```

Example 2 (Normal Case): Input:

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

Output:

```
[5, 1, -4, -5]
```

---

4. Approaches

Approach 1: Brute Force

- **Idea:**
  - For each element nums[i], check all elements to its right.
  - If nums[i] is greater than all right elements, add to result.

Java Code:

```java
public static List<Integer> leadersBrute(int[] nums) {
    List<Integer> leaders = new ArrayList<>();
    for (int i = 0; i < nums.length; i++) {
```

```
            boolean isLeader = true;
            for (int j = i + 1; j < nums.length; j++) {
                if (nums[i] <= nums[j]) {
                    isLeader = false;
                    break;
                }
            }
            if (isLeader) leaders.add(nums[i]);
        }
        return leaders;
    }
```

Complexity:

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

Approach 2: Optimal (Right-to-Left Scan)

- **Idea:**
  - Initialize maxFromRight = nums[n-1] â†’ last element is always a leader.
  - Traverse array from right to left:
  - If nums[i] > maxFromRight, then nums[i] is a leader.
  - Update maxFromRight = max(maxFromRight, nums[i]).
  - Reverse result list at the end to maintain original order.

Java Code:

```
public static List<Integer> leadersOptimal(int[] nums) {
    List<Integer> leaders = new ArrayList<>();
    int n = nums.length;
    int maxFromRight = nums[n - 1];
    leaders.add(maxFromRight); // last element is always a leader

    for (int i = n - 2; i >= 0; i--) {
        if (nums[i] > maxFromRight) {
            leaders.add(nums[i]);
            maxFromRight = nums[i];
        }
    }

    Collections.reverse(leaders); // to maintain order of appearance
    return leaders;
}
```

Complexity:

- Time: O(n) â†’ single pass.
- Space: O(n) â†’ for result list.

---

5. Justification / Proof of Optimality

- Optimal approach is linear time and efficient, scanning the array once from right.
- Avoids nested loops â†’ much faster for large arrays (n <= 10^5).
- Always maintains order by reversing the result at the end.

---

6. Variants / Follow-Ups

- Find all leaders except the last element â†’ modified right-to-left scan.
- Find leaders in a circular array â†’ consider array wraps around.
- Find minimum leaders â†’ elements smaller than all right elements.
- Count the number of leaders instead of listing them.

Q40: Rearrange Array Elements by Sign

---

1. Understand the Problem

- **Read & Identify:** Given an array with equal numbers of positive and negative integers, rearrange it such that every consecutive pair has opposite signs.
- **Goal:** Return the rearranged array starting with a positive number while maintaining relative order of positive and negative numbers.
- **Paraphrase:** Interleave positive and negative integers while preserving their original order, starting with a positive integer.

---

2. Constraints

- 2 <= nums.length <= 10^5
- 1 <= |nums[i]| <= 10^4
- Array length is even and positives = negatives.

---

3. Examples & Edge Cases

Example 1 (Edge Case): Input:

```
[1, -1]
```

Output:

```
[1, -1]
```

Example 2 (Normal Case): Input:

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

Output:

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

---

4. Approaches

Approach 1: Brute Force

- **Idea:**
  - Create two separate lists: positives and negatives.
  - Traverse the array and fill the two lists.
  - Interleave elements from positives and negatives into a new array.

Java Code:

```java
public static int[] rearrangeBrute(int[] nums) {
    List<Integer> pos = new ArrayList<>();
    List<Integer> neg = new ArrayList<>();

    for (int num : nums) {
        if (num > 0) pos.add(num);
        else neg.add(num);
    }

    int[] result = new int[nums.length];
    int p = 0, n = 0;

    for (int i = 0; i < nums.length; i++) {
        if (i % 2 == 0) result[i] = pos.get(p++);
        else result[i] = neg.get(n++);
    }

    return result;
}
```

Complexity:

- Time: O(n) â†' single pass for separation + single pass for merging.
- Space: O(n) â†' extra arrays for positive and negative numbers.

Approach 2: Optimal (In-Place Using Extra Indices)

- **Idea:**
  - Use two pointers p and n to track positions of next positive and negative numbers.
  - Traverse the array and place positive and negative numbers alternately in a new array (or can do in-place with rotation logic).
  - Maintains relative order without nested loops.

Java Code:

```
public static int[] rearrangeOptimal(int[] nums) {
    int n = nums.length;
    int[] result = new int[n];
    int p = 0, q = 0; // indices for positives and negatives

    // find first positive index
    for (int i = 0; i < n; i++) {
        if (nums[i] > 0) p++;
        else q++;
    }

    List<Integer> pos = new ArrayList<>();
    List<Integer> neg = new ArrayList<>();

    for (int num : nums) {
        if (num > 0) pos.add(num);
        else neg.add(num);
    }

    int pi = 0, ni = 0;
    for (int i = 0; i < n; i++) {
        if (i % 2 == 0) result[i] = pos.get(pi++);
        else result[i] = neg.get(ni++);
    }

    return result;
}
```

Complexity:

- Time: O(n) â†' single pass.
- Space: O(n) â†' result array needed (in-place is more complex but possible).

---

5. Justification / Proof of Optimality

- Optimal approach ensures O(n) time and preserves relative order of positives and negatives.
- Brute force is easier to implement but uses extra space.
- In-place rearrangement is possible but increases code complexity significantly.

---

6. Variants / Follow-Ups

- Array length odd â†’ more positives or negatives â†’ start with dominant sign.
- Rearrange without extra space â†’ in-place cyclic replacement.
- Rearrange to alternate starting with negative.
- Extend to k-alternating signs (e.g., 2 positives, 1 negative pattern).

Q41: Two Sum

---

1. Understand the Problem

- **Read & Identify:** Given an array of integers nums and a target integer target, find indices of two numbers in nums that sum up to target.
- **Paraphrase:** Find exactly one pair (i, j) such that nums[i] + nums[j] = target

---

2. Constraints

- 2 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4
- -10^5 <= target <= 10^5
- Exactly one solution exists
- Each element used at most once

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
[1,6,2,10,3], target 7
```

Output:

```
[0,1]
```

Example 2 (Normal Case): Input:

```
[1,3,5,-7,6,-3], target 0
```

Output:

```
[1,5]
```

---

4. Approaches

Approach 1: Brute Force

- **Idea:**
  - Check all possible pairs (i,j) where i<j to see if nums[i]+nums[j] == target.

Java Code:

```
public int[] twoSumBrute(int[] nums, int target) {
    int n = nums.length;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            if (nums[i] + nums[j] == target) {
                return new int[]{i, j};
            }
        }
    }
    return new int[]{-1, -1}; // should never reach here
}
```

Complexity:

- Time: O(n^2) â†' two nested loops.
- Space: O(1) â†' no extra space.

Approach 2: Optimal (HashMap)

- **Idea:**
  - Traverse array and store value â†' index in a HashMap.
  - For each element num, check if target - num exists in the map.
  - Return indices immediately when found.

Java Code:

```
import java.util.*;

public int[] twoSumOptimal(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
```

```
                return new int[]{map.get(complement), i};
            }
            map.put(nums[i], i);
        }
        return new int[]{-1, -1}; // should never reach here
    }
```

Complexity:

- Time: O(n) â†' single traversal.
- Space: O(n) â†' for the HashMap.

---

5. Justification / Proof of Optimality

- Brute Force: Simple, but inefficient for large arrays (n^2).
- HashMap Approach: Efficient, single pass, handles negative numbers and duplicates, optimal solution.

---

6. Variants / Follow-Ups

- Find all pairs summing to target (multiple solutions).
- Array is sorted â†' can use two-pointer approach instead of HashMap.
- Return values instead of indices.
- Target sum for more than two numbers â†' generalizes to 3-sum, 4-sum problems.

Q42: Sort an Array of 0's, 1's, and 2's

---

1. Understand the Problem

- **Read & Identify:** Given an array containing only 0, 1, and 2, sort the array in non-decreasing order.
- **Paraphrase:** Arrange all 0â€™s first, followed by 1â€™s, then 2â€™s in the original array.

---

2. Constraints

- 2 <= nums.length <= 10^5
- 1 <= |nums[i]| <= 10^4
- Array length is even and positives = negatives.

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
[1,0,2,1,0]
```

Output:

```
[0,0,1,1,2]
```

---

4. Approaches

Approach 1: Brute Force

- **Idea:**
  - Count number of 0â€™s, 1â€™s, and 2â€™s.
  - Overwrite the array with 0â€™s, 1â€™s, 2â€™s according to counts.

Java Code:

```java
public static void sortColorsCounting(int[] nums) {
    int count0 = 0, count1 = 0, count2 = 0;
    for (int num : nums) {
        if (num == 0) count0++;
        else if (num == 1) count1++;
        else count2++;
    }

    int i = 0;
    while (count0-- > 0) nums[i++] = 0;
    while (count1-- > 0) nums[i++] = 1;
    while (count2-- > 0) nums[i++] = 2;
}
```

Complexity:

- Time: O(n) â†’ single pass to count + single pass to fill.
- Space: O(1) â†’ 3 counters only.

Approach 2: Optimal (Dutch National Flag Algorithm)

- **Idea:**
  - Use three pointers: low, mid, high.
  - low â†’ next position for 0, mid â†’ current element, high â†’ next position for 2.
  - Traverse array with mid:
  - If nums[mid] == 0 â†’ swap with nums[low], low++, mid++
  - If nums[mid] == 1 â†’ mid++
  - If nums[mid] == 2 â†’ swap with nums[high], high--

Java Code:

```
public static void sortColorsDutchFlag(int[] nums) {
    int low = 0, mid = 0, high = nums.length - 1;
    while (mid <= high) {
        if (nums[mid] == 0) {
            int temp = nums[low];
            nums[low] = nums[mid];
            nums[mid] = temp;
            low++;
            mid++;
        } else if (nums[mid] == 1) {
            mid++;
        } else { // nums[mid] == 2
            int temp = nums[mid];
            nums[mid] = nums[high];
            nums[high] = temp;
            high--;
        }
    }
}
```

Complexity:

- Time: O(n) â†' single traversal.
- Space: O(1) â†' in-place.

---

5. Justification / Proof of Optimality

- Dutch National Flag Algorithm ensures single pass and in-place sorting, optimal for large arrays.
- Counting method is simple and uses constant space, but requires two passes.

---

6. Variants / Follow-Ups

- Array contains more than 3 distinct elements â†' can extend with counting sort.
- Sort array in decreasing order instead of non-decreasing.
- Array elements are not consecutive integers â†' use hashmap or quicksort.
- Online streaming input â†' maintain three queues instead of array.

Q43: Move Zeros to End

---

1. Understand the Problem

- **Read & Identify:** Given an array nums, move all the 0s to the end of the array without changing the relative order of non-zero elements.
- **Paraphrase:** Shift all zero elements to the right while keeping non-zero elements in their original sequence.

---

2. Constraints

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4
- Must be in-place (no extra array allowed).

---

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
[0, 1, 4, 0, 5, 2]
```

Output:

```
[1, 4, 5, 2, 0, 0]
```

---

4. Approaches

Approach 1: Brute Force (Shift Zeros Iteratively)

- **Idea:**
  - For each zero, shift all elements on its right one step left and append zero at the end.

Java Code:

```java
public void moveZerosBrute(int[] nums) {
    int n = nums.length;
    for (int i = 0; i < n; i++) {
        if (nums[i] == 0) {
            for (int j = i; j < n - 1; j++) {
                nums[j] = nums[j + 1];
            }
            nums[n - 1] = 0;
        }
    }
}
```

Complexity:

- Time: O(n^2) â†’ shifting elements for each zero.
- Space: O(1) â†’ in-place.

Approach 2: Optimal (Two Pointers)

- **Idea:**
  - Use a pointer pos to track the position to place the next non-zero element.
  - Traverse the array: if element is non-zero, swap it with nums[pos] and increment pos.
  - All zeros naturally move to the end.

Java Code:

```
public void moveZerosOptimal(int[] nums) {
    int pos = 0; // position for the next non-zero
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] != 0) {
            int temp = nums[i];
            nums[i] = nums[pos];
            nums[pos] = temp;
            pos++;
        }
    }
}
```

Complexity:

- Time: O(n) â†’ single traversal.
- Space: O(1) â†’ in-place.

---

5. Justification / Proof of Optimality

- Brute Force: Works but inefficient for large n due to repeated shifting.
- Two-Pointer Approach:
- Only swaps non-zero elements, maintains relative order,
- linear time complexity, minimal swaps â†’ optimal solution.

---

6. Variants / Follow-Ups

- Move all non-zero elements to the start instead.
- Move all zeros to the beginning.
- Count minimum swaps required to move zeros to end.
- Multiple arrays merged â†’ move zeros in combined array.

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 â†' 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:

```java
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) {
            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 â‰¤ n â‰¤ 3000
- 4 Sum: 1 â‰¤ n â‰¤ 200
- K Sum: 1 â‰¤ n â‰¤ 200
- Element range: -10^4 â‰¤ nums[i] â‰¤ 10^4
- Target range: -10^5 â‰¤ target â‰¤ 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 â†’ Time: O(nÂ³), Space: O(3) for recursion stack
- 4 Sum â†’ Time: O(nâ´), Space: O(4) for recursion stack
- K Sum â†’ Time: O(náµ), 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 (kâˆ’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 â†’ Time: O(nÂ²), Space: O(1)
- 4 Sum â†’ Time: O(nÂ³), Space: O(1)
- K Sum â†’ 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 â†' 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 â†' Time: O(nÂ²), Space: O(1)
- 4 Sum â†' Time: O(nÂ³), Space: O(k) recursion
- K Sum â†' 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: Pascalâ€™s Triangle I, II, III

---

1. Problem Understanding

- Pascalâ€™s Triangle I: Return the value at a specific row r and column c (1-indexed).
- Pascalâ€™s Triangle II: Return all values in the r-th row (1-indexed).
- Pascalâ€™s Triangle III: Return the first n rows of Pascalâ€™s Triangle.
- Key differences:
- I â†' Single value lookup.
- II â†' Entire row.
- III â†' 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

- Pascalâ€™s Triangle I â†’ 1 â‰¤ r, c â‰¤ 30, c â‰¤ r
- Pascalâ€™s Triangle II â†’ 1 â‰¤ r â‰¤ 30
- Pascalâ€™s Triangle III â†’ 1 â‰¤ n â‰¤ 30
- All values fit in a 32-bit integer.

---

3. Edge Cases

- c = 1 or c = r â†’ Always 1.
- r = 1 â†’ 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) â†’ Handle gracefully.

---

4. Examples

Example 1:

```
(Pascalâ€™s Triangle I):
Input: r = 4, c = 2 â†’ Output: 3
Explanation: Row 4 â†’ [1, 3, 3, 1] â†’ 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:

```
(Pascalâ€™s Triangle II):
Input: r = 5 â†’ 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:

```
(Pascalâ€™s Triangle III):
Input: n = 4 â†’ 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 Pascalâ€™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 â†' Time: O(rÂ²), Space: O(rÂ²)
- Pascal II â†' Time: O(rÂ²), Space: O(rÂ²)
- Pascal III â†' Time: O(nÂ²), Space: O(nÂ²)

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 â†' Time: O(rÂ²), Space: O(r)
- Pascal II â†' Time: O(rÂ²), Space: O(r)

- Pascal III â†' Time: O(nÂ²), 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 â†' Compute single combination.
- For Pascal II â†' Compute each element using C(r-1, i).
- For Pascal III â†' 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 â†' Time: O(c), Space: O(1)
- Pascal II â†' Time: O(r), Space: O(r)
- Pascal III â†' Time: O(nÂ²), 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(rÂ²) time.
- Optimal / Binomial: Fastest for single value or row, uses formula, minimal space.

---

7. Variants / Follow-Ups

- Pascalâ€™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 â†’ Single element, threshold n/2
- II â†’ Multiple elements, threshold n/3
- n/k â†’ Multiple elements, threshold n/k

---

2. Algorithm

Algorithm 1: Boyerâ€“Moore Voting Algorithm

- The Boyerâ€“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 â€œvote.â€□
    - When two different numbers appear, they cancel each otherâ€™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 â†’ increment its count.
    - Else if thereâ€™s an empty slot (count = 0) â†’ assign this number as a new candidate.
    - Else â†’ 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)**
    - Letâ€™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) â€” single pass for selection + optional verification.
    - Space Complexity: O(1) â€” constant space for fixed k.
- **Why Itâ€™s Efficient**
    - Traditional counting methods (like HashMaps) use O(n) space.
    - Boyerâ€“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 â‰¤ n â‰¤ 10^5
- 2 â‰¤ n for II and n/k
- -10^4 â‰¤ nums[i] â‰¤ 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 â†’ single majority
- No element meets the threshold â†’ 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] â†’ Output: 7
Explanation: 7 appears 5 times in size 9 array â†’ majority

Example 2 (Majority Element II):
Input: nums = [1, 2, 1, 1, 3, 2, 2] â†’ Output: [1, 2]
Explanation: n/3 = 7/3 = 2, elements appearing â‰¥3 times: [1, 2]

Example 3 (Majority Element n/k, k=4):
Input: nums = [1, 2, 2, 3, 2, 1, 1, 4], k = 4 â†’ 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 â†' Time: O(n), Space: O(n)
- Majority Element II â†' Time: O(n), Space: O(n)
- Majority Element n/k â†' 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 â†' middle element is majority.
- For n/3 or n/k â†' count elements while traversing to check frequency.

Java Code:

```java
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 â†’ 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 â†’ generalized version maintaining k-1 candidates.

Steps:

- I â†’ single candidate.

- II â†' two candidates.
- n/k â†' 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 â†’ Time: O(n), Space: O(1)
- Majority Element II â†’ Time: O(n), Space: O(1)
- Majority Element n/k â†’ Time: O(n), Space: O(n)

---

7. Justification / Proof of Optimality

- Brute Force â†’ Simple, works but uses extra space.
- Sorting â†’ Reduces space, slightly slower due to O(n log n)
- Boyer-Moore â†’ Best for I & II, minimal space and linear time.
- n/k generalization â†’ 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 â†’ 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 â†’ 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 â†’ repeating number (A).
- One number is missing â†’ 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 â‰¤ n â‰¤ 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 â†' repeating
- Identify the number not present â†' missing

Steps:

- Create a frequency map of numbers.
- Iterate from 1 to n:
- If count = 2 â†' repeating

- If count = 0 â†’ 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 â†' result = A ^ B
- Find any set bit in XOR â†' divide numbers into 2 groups â†' 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 â†' simple, works, uses extra space
- Math â†' elegant, constant space, careful with overflow
- XOR â†' 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 â€œSingle Numberâ€□ or â€œFind Duplicate Numberâ€□

---

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 â†' 0 inversions
  - Descending array â†' maximum inversions

---

2. Constraints

- 1 â‰¤ nums.length â‰¤ 10^5
- -10^4 â‰¤ nums[i] â‰¤ 10^4

---

3. Edge Cases

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

---

4. Examples

```
Example 1:
Input: [2, 3, 7, 1, 3, 5]
Output: 5
Explanation (inversions):
(0,3) â†’ 2 > 1
(1,3) â†’ 3 > 1
(2,3) â†’ 7 > 1
(2,4) â†’ 7 > 3
(2,5) â†’ 7 > 5

Example 2:
Input: [-10, -5, 6, 11, 15, 17]
Output: 0
Explanation: already sorted â†’ 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] â†' increment count

Java Code:

```java
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(nÂ²)
- 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 â†' 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] â‰¤ right[j] â†' no inversion
- Else â†' inversion count += remaining elements in left array
- Return total inversions

Java Code:

```
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 â†’ simple but slow for large arrays
- Merge Sort â†’ efficient for large arrays, counts inversions during sorting
- Merge Sort is preferred for constraints n â‰¤ 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, itâ€™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 â†’ 0 reverse pairs
- Fully descending array â†’ potentially maximum reverse pairs
- Array with duplicates â†’ count all valid (i, j) pairs
- Single element â†’ 0 reverse pairs

---

4. Examples

```
Example 1:
Input: [6, 4, 1, 2, 7]
Output: 3
Explanation (reverse pairs):
(0, 2) â†’ 6 > 2*1
(0, 3) â†’ 6 > 2*2
(1, 2) â†’ 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] â†' 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(nÂ²)
- 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 â†' easy but slow for large arrays
- Merge Sort â†' 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 â†’ product may reset
- Array contains all negative numbers â†’ even length negative subarray may give max product
- Single element array â†’ product = element itself
- All positive numbers â†’ 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 â†’ 45371*2 = 840

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

Example 3:
Input: [1, -2, 3, 4, -4, -3]
Output: 288
Explanation: Maximum product subarray is [3, 4, -4, -3] â†’ 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 â†' start index
- Iterate j from i to n-1 â†' end index
- Multiply elements from i to j â†' 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;
            for(int j=i;j<n;j++){
                product *= nums[j];
                maxProduct = Math.max(maxProduct, product);
            }
        }
        return maxProduct;
    }
}
```

Complexity (Time & Space):

- Time: O(nÂ²)
- 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 â†' 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:

```
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 â†' prefix *= nums[i], update maxProduct, reset prefix if zero
- Iterate from right to left â†' 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 â†' simple but inefficient, works only for small arrays
- Dynamic Programming â†' optimal, handles negative numbers and zeros
- Prefix/Suffix Scan â†' alternative O(n) approach, slightly simpler to code

---

7. Variants / Follow-Ups

- Maximum sum subarray â†' Kadaneâ€™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 â†' 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 â†' return the non-empty array
- Arrays contain duplicates â†' keep all elements
- Negative numbers â†' ensure correct ordering
- All elements of nums2 are smaller than nums1 â†' insert at the beginning
- All elements of nums2 are larger than nums1 â†' 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 nums1â€™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:

```java
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] â†' nums1[k--] = nums1[i--]
    - Else â†' nums1[k--] = nums2[j--]
- Copy remaining elements of nums2 if any

Java Code:

```java
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 â†’ simple but slower due to sorting
- Two Pointers â†’ optimal and widely used in interviews
- Gap Method â†’ 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
