Java Interview Questions — Part 2

Array Programs, Searching & Sorting, Matrix Programs (Q41 – Q70)

Prepared for Shammi Bharti

Contents

Section A — Array Programs (Q41 to Q60)

Q41. Reverse Array

What is being asked: Print an array's elements in reverse order. Example: `[1,2,3,4]` -> `[4,3,2,1]`.

Logic: Loop from the last index down to the first index and print/store each element.

```
public class ReverseArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int[] reversed = new int[arr.length];

        for (int i = 0; i < arr.length; i++) {
            reversed[i] = arr[arr.length - 1 - i];
        }

        for (int num : reversed) {
            System.out.print(num + " ");
        }
    }
}
```

Output: `5 4 3 2 1`

Q42. Reverse Array In-Place

What is being asked: Same as Q41, but without creating a new array — modify the original array directly.

Logic: Use the two-pointer technique: `left` starts at index 0, `right` starts at the last index. Swap them and move inward until they meet (same idea as reversing a string in-place).

```
public class ReverseArrayInPlace {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int left = 0, right = arr.length - 1;

        while (left < right) {
            int temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
            left++;
            right--;
        }

        for (int num : arr) {
```

```
            System.out.print(num + " ");
        }
    }
}
```

Q43. Find Duplicate Elements in Array

What is being asked: Find which elements appear more than once in an array.

Logic: Use a `HashSet`. Loop through the array — if `set.add(num)` returns `false`, it means that number was already in the set, so it is a duplicate.

```
import java.util.HashSet;

public class FindDuplicates {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 5, 1};
        HashSet<Integer> seen = new HashSet<>();
        HashSet<Integer> duplicates = new HashSet<>();

        for (int num : arr) {
            if (!seen.add(num)) {
                duplicates.add(num);
            }
        }

        System.out.println("Duplicate elements: " + duplicates);
    }
}
```

Output: `Duplicate elements: [1, 2]`

Q44. Remove Duplicates from Array

What is being asked: Keep only the first occurrence of every element, in their original order.

Logic: Use a `LinkedHashSet<Integer>` (keeps insertion order, removes duplicates automatically).

```
import java.util.LinkedHashSet;

public class RemoveDuplicatesArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 1, 5};
        LinkedHashSet<Integer> result = new LinkedHashSet<>();
```

```
        for (int num : arr) {
            result.add(num);
        }

        System.out.println("After removing duplicates: " + result);
    }
}
```

Output: `After removing duplicates: [1, 2, 3, 4, 5]`

Q45. Remove Duplicates and Sort Array

What is being asked: Same as Q44, but also sort the result in ascending order.

Logic: A `TreeSet<Integer>` automatically removes duplicates **and** keeps elements sorted — it does both jobs at once.

```
import java.util.TreeSet;

public class RemoveDuplicatesAndSort {
    public static void main(String[] args) {
        int[] arr = {5, 2, 3, 2, 4, 1, 5};
        TreeSet<Integer> result = new TreeSet<>();

        for (int num : arr) {
            result.add(num);
        }

        System.out.println("Sorted, no duplicates: " + result);
    }
}
```

Output: `Sorted, no duplicates: [1, 2, 3, 4, 5]`

Q46. Sort Array in Ascending Order

Logic: Use the built-in `Arrays.sort()` method — it sorts in ascending order by default.

```
import java.util.Arrays;

public class SortAscending {
    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 1, 9};
        Arrays.sort(arr);
        System.out.println(Arrays.toString(arr));
    }
```

```
}
```

Output: `[1, 2, 5, 8, 9]`

Q47. Sort Array in Descending Order

Logic: Java's `Arrays.sort()` only works ascending for primitive `int[]`. The simplest trick is to first sort ascending, then reverse it. (For `Integer[]` objects you could also use `Collections.reverseOrder()`.)

```
import java.util.Arrays;
import java.util.Collections;

public class SortDescending {
    public static void main(String[] args) {
        Integer[] arr = {5, 2, 8, 1, 9};
        Arrays.sort(arr, Collections.reverseOrder());
        System.out.println(Arrays.toString(arr));
    }
}
```

Output: `[9, 8, 5, 2, 1]`

Q48. Find Maximum Element in Array

Logic: Assume the first element is the maximum so far. Loop through the rest, and whenever a bigger element is found, update the maximum.

```
public class FindMaximum {
    public static void main(String[] args) {
        int[] arr = {3, 7, 2, 9, 4};
        int max = arr[0];

        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }

        System.out.println("Maximum: " + max);
    }
}
```

Q49. Find Minimum Element in Array

Logic: Same idea as Q48, but keep updating whenever a smaller element is found.

```
public class FindMinimum {
    public static void main(String[] args) {
        int[] arr = {3, 7, 2, 9, 4};
        int min = arr[0];

        for (int i = 1; i < arr.length; i++) {
            if (arr[i] < min) {
                min = arr[i];
            }
        }

        System.out.println("Minimum: " + min);
    }
}
```

Q50. Find Second Largest Element

Logic: Keep two variables: `largest` and `secondLargest`. While looping, if the current number is bigger than `largest`, push the old `largest` down into `secondLargest` before updating. Else if it is bigger than `secondLargest` (but not `largest`), update `secondLargest` only.

```
public class SecondLargest {
    public static void main(String[] args) {
        int[] arr = {3, 7, 2, 9, 4};
        int largest = Integer.MIN_VALUE;
        int secondLargest = Integer.MIN_VALUE;

        for (int num : arr) {
            if (num > largest) {
                secondLargest = largest;
                largest = num;
            } else if (num > secondLargest && num != largest) {
                secondLargest = num;
            }
        }

        System.out.println("Second Largest: " + secondLargest);
    }
}
```

Output: `Second Largest: 7`

Q51. Find Third Largest Element

Logic: Same idea as Q50, but track three variables: `first`, `second`, `third`. Every new number is compared against all three in order, shifting the others down when it is bigger.

```
public class ThirdLargest {
    public static void main(String[] args) {
        int[] arr = {3, 7, 2, 9, 4, 9, 12};
        int first = Integer.MIN_VALUE;
        int second = Integer.MIN_VALUE;
        int third = Integer.MIN_VALUE;

        for (int num : arr) {
            if (num > first) {
                third = second;
                second = first;
                first = num;
            } else if (num > second && num != first) {
                third = second;
                second = num;
            } else if (num > third && num != second && num != first) {
                third = num;
            }
        }

        System.out.println("Third Largest: " + third);
    }
}
```

Q52. Find Missing Number in Array

What is being asked: An array contains numbers from 1 to `n`, but one number is missing. Find it. Example: `[1,2,4,5]` with `n=5` is missing 3.

Logic: The sum of numbers from 1 to `n` is `n*(n+1)/2` (a well-known formula). Subtract the actual sum of the array from this expected sum — the difference is the missing number.

```
public class MissingNumber {
    public static void main(String[] args) {
        int[] arr = {1, 2, 4, 5};
        int n = 5;   // numbers are supposed to range from 1 to n

        int expectedSum = n * (n + 1) / 2;
        int actualSum = 0;
        for (int num : arr) {
            actualSum += num;
```

```
        }

        System.out.println("Missing Number: " + (expectedSum - actualSum));
    }
}
```

Output: `Missing Number: 3`

Q53. Move All Zeros to End

What is being asked: Move all zeros in the array to the end, while keeping the relative order of non-zero elements. Example: `[0,1,0,3,12]` -> `[1,3,12,0,0]`.

Logic: Keep a pointer `insertPos` starting at 0. Loop through the array; whenever a non-zero number is found, place it at `insertPos` and increase `insertPos`. After the loop, fill everything from `insertPos` to the end with zeros.

```
public class MoveZerosToEnd {
    public static void main(String[] args) {
        int[] arr = {0, 1, 0, 3, 12};
        int insertPos = 0;

        for (int num : arr) {
            if (num != 0) {
                arr[insertPos] = num;
                insertPos++;
            }
        }

        while (insertPos < arr.length) {
            arr[insertPos] = 0;
            insertPos++;
        }

        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Output: `1 3 12 0 0`

Q54. Rotate Array by K Positions

What is being asked: Shift every element to the right by k positions, wrapping around. Example: `[1,2,3,4,5]` rotated by 2 -> `[4,5,1,2,3]`.

Logic: Use a temporary array. Every element at index `i` moves to its new position `(i + k) % arr.length` (the `%` handles the wrap-around).

```
public class RotateArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int k = 2;
        int n = arr.length;
        int[] rotated = new int[n];

        for (int i = 0; i < n; i++) {
            rotated[(i + k) % n] = arr[i];
        }

        for (int num : rotated) {
            System.out.print(num + " ");
        }
    }
}
```

Output: `4 5 1 2 3`

Q55. Merge Two Sorted Arrays

What is being asked: Combine two already-sorted arrays into one sorted array.

Logic: Use two pointers, `i` for the first array and `j` for the second array. Compare `arr1[i]` and `arr2[j]`, copy the smaller one into the result, and move that pointer forward. Once one array finishes, copy the remaining elements of the other directly.

```
public class MergeSortedArrays {
    public static void main(String[] args) {
        int[] arr1 = {1, 3, 5};
        int[] arr2 = {2, 4, 6};
        int[] merged = new int[arr1.length + arr2.length];

        int i = 0, j = 0, k = 0;
        while (i < arr1.length && j < arr2.length) {
            if (arr1[i] <= arr2[j]) {
                merged[k++] = arr1[i++];
            } else {
                merged[k++] = arr2[j++];
            }
        }
        while (i < arr1.length) merged[k++] = arr1[i++];
        while (j < arr2.length) merged[k++] = arr2[j++];

        for (int num : merged) {
```

```
            System.out.print(num + " ");
        }
    }
}
```

Output: `1 2 3 4 5 6`

Q56. Find Intersection of Two Arrays

What is being asked: Find elements that are present in **both** arrays.

Logic: Put all elements of the first array into a `HashSet`. Loop through the second array, and any element that is also in the set is part of the intersection.

```
import java.util.HashSet;

public class ArrayIntersection {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
        int[] arr2 = {3, 4, 5, 6, 7};

        HashSet<Integer> set1 = new HashSet<>();
        for (int num : arr1) set1.add(num);

        HashSet<Integer> result = new HashSet<>();
        for (int num : arr2) {
            if (set1.contains(num)) {
                result.add(num);
            }
        }

        System.out.println("Intersection: " + result);
    }
}
```

Output: `Intersection: [3, 4, 5]`

Q57. Find Union of Two Arrays

What is being asked: Combine all unique elements present in either array.

Logic: Add all elements of both arrays into one `HashSet` — a set automatically removes duplicates, giving the union directly.

```
import java.util.HashSet;

public class ArrayUnion {
    public static void main(String[] args) {
```

```
        int[] arr1 = {1, 2, 3};
        int[] arr2 = {3, 4, 5};

        HashSet<Integer> result = new HashSet<>();
        for (int num : arr1) result.add(num);
        for (int num : arr2) result.add(num);

        System.out.println("Union: " + result);
    }
}
```

Output: `Union: [1, 2, 3, 4, 5]`

Q58. Add Each Odd and Even Pair in Array

What is being asked: Pick out odd-indexed/even-valued style pairings and add them — a common variation is to separately sum all odd numbers and all even numbers in the array.

Logic: Loop through the array once. If a number is even, add it to `evenSum`; if odd, add it to `oddSum`.

```
public class OddEvenPairSum {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6};
        int evenSum = 0, oddSum = 0;

        for (int num : arr) {
            if (num % 2 == 0) {
                evenSum += num;
            } else {
                oddSum += num;
            }
        }

        System.out.println("Sum of even numbers: " + evenSum);
        System.out.println("Sum of odd numbers: " + oddSum);
    }
}
```

Q59. Two Sum Problem

What is being asked: Given an array and a target value, find two numbers that add up to the target, and return their indices. Example: `[2,7,11,15]`, target `9` -> indices `[0,1]` because `2+7=9`.

Logic: Use a `HashMap<value, index>`. For every number, calculate `complement = target - num`. If `complement` is already in the map, you found your pair. Otherwise, add the current number and its index to the map and keep going. This is an `O(n)` solution (much faster than checking every pair).

```
import java.util.HashMap;

public class TwoSum {
    public static void main(String[] args) {
        int[] arr = {2, 7, 11, 15};
        int target = 9;
        HashMap<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < arr.length; i++) {
            int complement = target - arr[i];
            if (map.containsKey(complement)) {
            System.out.println("Indices: " + map.get(complement) + ", " + i);
                break;
            }
            map.put(arr[i], i);
        }
    }
}
```

Output: `Indices: 0, 1`

Q60. Kadane's Algorithm (Maximum Subarray Sum)

What is being asked: Find the largest possible sum of any contiguous (continuous) part of the array. Example: `[-2,1,-3,4,-1,2,1,-5,4]` -> maximum subarray sum is `6` (from `[4,-1,2,1]`).

Logic — Kadane's Algorithm:

- Keep two variables: `currentSum` (sum of the subarray ending at the current position) and `maxSum` (the best answer found so far).
- For each number: if `currentSum` becomes negative, reset it to the current number (starting fresh is better than carrying a negative sum forward). Otherwise, add the current number to `currentSum`.
- After each step, update `maxSum` if `currentSum` is bigger.

```
public class KadanesAlgorithm {
    public static void main(String[] args) {
        int[] arr = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        int currentSum = arr[0];
        int maxSum = arr[0];

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

```
            maxSum = Math.max(maxSum, currentSum);
        }

        System.out.println("Maximum Subarray Sum: " + maxSum);
    }
}
```

Output: Maximum Subarray Sum: 6

Section B — Searching & Sorting (Q61 to Q67)

Q61. Binary Search

What is being asked: Quickly find an element in a **sorted** array by repeatedly dividing the search range in half.

Logic:

- Keep `low = 0` and `high = arr.length - 1`.
- Check the middle element. If it equals the target, you found it.
- If the middle element is smaller than the target, the answer must be on the right side, so move `low` to `mid + 1`.
- If it is bigger, move `high` to `mid - 1`.
- Repeat until `low > high` (not found) or the target is found.

```
public class BinarySearch {
    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 8, 10, 12, 14};
        int target = 10;
        int low = 0, high = arr.length - 1;
        int result = -1;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] == target) {
                result = mid;
                break;
            } else if (arr[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }

        System.out.println("Element found at index: " + result);
    }
}
```

Output: `Element found at index: 4`

Why it matters in interviews: Binary Search runs in `O(log n)` time, much faster than a normal loop search which is `O(n)`. This is a very common interview question.

Q62. Bubble Sort

What is being asked: Sort an array by repeatedly swapping adjacent elements if they are in the wrong order.

Logic: Compare each pair of neighbouring elements; if the left one is bigger than the right one, swap them. After each full pass, the largest unsorted element "bubbles up" to its correct position at the end.

```
public class BubbleSort {
    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 1, 9};

        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }

        for (int num : arr) System.out.print(num + " ");
    }
}
```

Time Complexity: `O(n^2)` — simple but slow for large arrays.

Q63. Insertion Sort

What is being asked: Build the sorted array one element at a time, like sorting playing cards in your hand.

Logic: Take each element (`key`) one by one. Compare it with the elements before it, and shift the bigger ones one step to the right, until you find the correct spot to "insert" the key.

```
public class InsertionSort {
    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 1, 9};

        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }

        for (int num : arr) System.out.print(num + " ");
```

```
    }
}
```

Time Complexity: `O(n^2)` worst case, but fast on nearly-sorted data.

Q64. Merge Sort

What is being asked: A faster sorting algorithm that uses the **divide and conquer** strategy.

Logic:

- Divide the array into two halves repeatedly until each part has only one element (a single element is always "sorted").
- Merge the smaller sorted halves back together in the correct order (same merging idea as Q55, Merge Two Sorted Arrays).

```
public class MergeSort {
    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 1, 9, 3};
        mergeSort(arr, 0, arr.length - 1);
        for (int num : arr) System.out.print(num + " ");
    }

    static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    static void merge(int[] arr, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        int i = left, j = mid + 1, k = 0;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) temp[k++] = arr[i++];
            else temp[k++] = arr[j++];
        }
        while (i <= mid) temp[k++] = arr[i++];
        while (j <= right) temp[k++] = arr[j++];

        for (int m = 0; m < temp.length; m++) {
            arr[left + m] = temp[m];
        }
    }
```

```
}
```

Time Complexity: `O(n log n)` — much better than Bubble/Insertion sort for large data.

Q65. Quick Sort

What is being asked: Another divide-and-conquer sorting algorithm, usually the fastest in practice.

Logic:

- Pick a `pivot` element (commonly the last element).
- Rearrange the array so everything smaller than the pivot goes to its left, and everything bigger goes to its right (this is called "partitioning").
- Recursively apply the same process to the left part and the right part.

```
public class QuickSort {
    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 1, 9, 3};
        quickSort(arr, 0, arr.length - 1);
        for (int num : arr) System.out.print(num + " ");
    }

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

    static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
            }
        }
        int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
        return i + 1;
    }
}
```

Time Complexity: `O(n log n)` on average.

Q66. Counting Sort

What is being asked: A non-comparison sorting algorithm that works very fast when the range of values is small and known.

Logic:

- Create a `count[]` array, where `count[i]` will store how many times the value `i` appears.
- Loop through the input and increase the count for each value.
- Loop through the `count[]` array in order and rebuild the sorted output.

```
public class CountingSort {
    public static void main(String[] args) {
        int[] arr = {4, 2, 2, 8, 3, 3, 1};
        int max = 8;   // largest possible value in the array
        int[] count = new int[max + 1];

        for (int num : arr) {
            count[num]++;
        }

        int index = 0;
        for (int i = 0; i <= max; i++) {
            while (count[i] > 0) {
                arr[index++] = i;
                count[i]--;
            }
        }

        for (int num : arr) System.out.print(num + " ");
    }
}
```

Time Complexity: `O(n + k)` where `k` is the range of values — very fast, but only practical when `k` is not too large.

Q67. Bucket Sort

What is being asked: Divide elements into a number of "buckets," sort each bucket individually, then combine them — works well for data spread evenly over a range (commonly used for decimal numbers between 0 and 1).

Logic:

- Create a number of empty buckets (lists).
- Put each element into the bucket that matches its range/value.

- Sort each bucket individually (a simple sort like `Collections.sort()` works fine since each bucket is small).
- Concatenate all the buckets in order to get the final sorted result.

```
import java.util.*;

public class BucketSort {
    public static void main(String[] args) {
        double[] arr = {0.42, 0.32, 0.23, 0.52, 0.25, 0.78};
        int n = arr.length;
        List<List<Double>> buckets = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            buckets.add(new ArrayList<>());
        }

        // Put each element into its bucket
        for (double num : arr) {
            int bucketIndex = (int) (num * n);
            buckets.get(bucketIndex).add(num);
        }

        // Sort each bucket and merge
        int index = 0;
        for (List<Double> bucket : buckets) {
            Collections.sort(bucket);
            for (double num : bucket) {
                arr[index++] = num;
            }
        }

        for (double num : arr) System.out.print(num + " ");
    }
}
```

Section C — Matrix Programs (Q68 to Q70)

Q68. Matrix Addition

What is being asked: Add two matrices of the same size, element by element.

Logic: Loop through every row and column. The value at `result[i][j]` is `matrixA[i][j] + matrixB[i][j]`.

```
public class MatrixAddition {
    public static void main(String[] args) {
        int[][] a = {{1, 2}, {3, 4}};
        int[][] b = {{5, 6}, {7, 8}};
        int rows = a.length, cols = a[0].length;
        int[][] result = new int[rows][cols];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = a[i][j] + b[i][j];
            }
        }

        for (int[] row : result) {
            for (int val : row) System.out.print(val + " ");
            System.out.println();
        }
    }
}
```

Output:

```
6 8
10 12
```

Q69. Matrix Multiplication

What is being asked: Multiply two matrices using standard matrix multiplication rules (rows of the first matrix multiplied with columns of the second).

Logic: For each cell `result[i][j]`, take the dot product of row `i` from matrix A and column `j` from matrix B — multiply corresponding elements and add them up. This needs three nested loops: one for the row, one for the column, and one to do the sum of products.

```
public class MatrixMultiplication {
    public static void main(String[] args) {
        int[][] a = {{1, 2}, {3, 4}};
        int[][] b = {{5, 6}, {7, 8}};
        int rowsA = a.length, colsA = a[0].length, colsB = b[0].length;
```

```
        int[][] result = new int[rowsA][colsB];

        for (int i = 0; i < rowsA; i++) {
            for (int j = 0; j < colsB; j++) {
                int sum = 0;
                for (int k = 0; k < colsA; k++) {
                    sum += a[i][k] * b[k][j];
                }
                result[i][j] = sum;
            }
        }

        for (int[] row : result) {
            for (int val : row) System.out.print(val + " ");
            System.out.println();
        }
    }
}
```

Output:

```
19 22
43 50
```

Q70. Matrix Transpose

What is being asked: Flip a matrix over its diagonal — rows become columns and columns become rows. Example: a 2x3 matrix becomes a 3x2 matrix.

Logic: Create a new matrix where `transpose[j][i] = original[i][j]` for every cell.

```
public class MatrixTranspose {
    public static void main(String[] args) {
        int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
        int rows = matrix.length, cols = matrix[0].length;
        int[][] transpose = new int[cols][rows];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                transpose[j][i] = matrix[i][j];
            }
        }

        for (int[] row : transpose) {
            for (int val : row) System.out.print(val + " ");
            System.out.println();
        }
    }
```

```
}
```

Output:

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

End of Part 2. Part 3 covers Linked List, Stack & Queue, Binary Tree, and Graph Problems.