```
package Love_Babbar;

public class Arrays {
    class Arrays {
        // Reverse the array
        class Solution {
            static void rvereseArray(int arr[], int start, int end) {
                int temp;
                while (start < end) {
                    temp = arr[start];
                    arr[start] = arr[end];
                    arr[end] = temp;
                    start++;
                    end--;
                }
            }
        }

        // Find the maximum and minimum element in an array
        class Solution {
            static pair getMinMax(long a[], long n) {
                // Write your code here
                PriorityQueue<Long> pqmin = new PriorityQueue<>();
                PriorityQueue<Long> pqmax = new PriorityQueue<>(Collections.
reverseOrder());
                for (int i = 0; i < n; i++) {
                    pqmin.add(a[i]);
                    pqmax.add(a[i]);
                }
                pair ans = new pair(pqmin.peek(), pqmax.peek());
                return ans;
            }
        }

        // Find the “Kth” max and min element of an array
        class Solution {
            public static int kthSmallest(int[] arr, int l, int r, int k) {
                // Your code here
                PriorityQueue<Integer> pqmax = new PriorityQueue<>(Collections.
reverseOrder());
                for (int i = l; i <= r; i++) {
                    pqmax.add(arr[i]);
                    if (pqmax.size() > k) {
                        pqmax.remove();
                    }
                }
                return pqmax.peek();
            }
```

```
    public static int kthLargest(int[] arr, int l, int r, int k) {
        // Your code here
        PriorityQueue<Integer> pqmin = new PriorityQueue<>();
        for (int i = l; i <= r; i++) {
            pqmin.add(arr[i]);
            if (pqmin.size() > k) {
                pqmin.remove();
            }
        }
        return pqmin.peek();
    }
}

// Given an array which consists of only 0, 1 and 2. Sort the array without
// using any sorting algo
class Solution {
    public static void sort012(int arr[], int n) {
        // code here
        int low = 0, mid = 0, high = n - 1;
        while (mid <= high) {
            if (arr[mid] == 0) {
                swap(arr, low, mid);
                low++;
                mid++;
            } else if (arr[mid] == 1) {
                mid++;
            } else {
                swap(arr, mid, high);
                high--;
            }
        }
    }

    public static void swap(int a[], int l, int r) {
        int temp = a[l];
        a[l] = a[r];
        a[r] = temp;
        return;
    }
}

// Move all the negative elements to one side of the array
// Find the Union and Intersection of the two sorted arrays.
class Solution {
    public static int doUnion(int a[], int n, int b[], int m) {
        // Your code here
        HashSet<Integer> hs = new HashSet<>();
        for (int i = 0; i < n; i++) {
            hs.add(a[i]);
```

```
        }
        for (int i = 0; i < m; i++) {
            hs.add(b[i]);
        }
        return hs.size();
    }

    public static int doInterSection(int a[], int n, int b[], int m) {
        // Your code here
        HashSet<Integer> hs = new HashSet<>();
        int cnt = 0;
        for (int i = 0; i < n; i++) {
            hs.add(a[i]);
        }
        for (int i = 0; i < m; i++) {
            if (hs.contains(b[i]) == true) {
                cnt++;
            }
        }
        return cnt;
    }
}

// Write a program to cyclically rotate an array by one.
class Solution {

    public void rotate(int arr[], int n) {
        int end = arr[n - 1];
        for (int i = n - 1; i > 0; i--) {
            arr[i] = arr[i - 1];
        }
        arr[0] = end;
    }
}

// Find Largest sum contiguous Subarray [V. IMP]
class Solution {

    // arr: input array
    // n: size of array
    // Function to find the sum of contiguous subarray with maximum sum.
    long maxSubarraySum(int arr[], int n) {

        // Your code here
        long max_so_far = Integer.MIN_VALUE;
        long max_ending_here = 0;
        for (int i = 0; i < n; i++) {
            max_ending_here += arr[i];
            if (max_ending_here > max_so_far) {
```

```
            max_so_far = max_ending_here;
        }
        if (max_ending_here < 0) {
            max_ending_here = 0;
        }
    }
    return max_so_far;

  }

}

// Minimize the maximum difference between heights [V.IMP]
class Solution {
  int getMinDiff(int[] arr, int n, int k) {
    // code here
    Arrays.sort(arr);
    int ans = arr[n - 1] - arr[0];
    int tempmin, tempmax;
    tempmin = arr[0];
    tempmax = arr[n - 1];
    for (int i = 1; i < n; i++) {
      if (arr[i] - k < 0) {
        continue;
      }
      tempmin = Math.min(arr[0] + k, arr[i] - k);
      tempmax = Math.max(arr[i - 1] + k, arr[n - 1] - k);
      ans = Math.min(ans, tempmax - tempmin);
    }
    return ans;
  }
}

// Minimum no. of Jumps to reach end of an array
// Find duplicate in an array of N+1 Integers
// Merge 2 sorted arrays without using Extra space.
// Kadane's Algo [V.V.V.V.V IMP]
// Merge Intervals
class Solution {
  class Pair {
    int start;
    int end;

    Pair(int s, int e) {
      start = s;
      end = e;
    }
  }
```

```
    public int[][] overlappedInterval(int[][] Intervals) {
        Arrays.sort(Intervals, (a, b) -> Integer.compare(a[0], b[0]));
        int n = Intervals.length;
        ArrayList<Pair> wrap = new ArrayList<>();
        int prevStart = Intervals[0][0];
        int prevEnd = Intervals[0][1];
        for (int i = 1; i < n; i++) {
            int currStart = Intervals[i][0];
            int currEnd = Intervals[i][1];
            if (currStart <= prevEnd) {
                prevEnd = Math.max(currEnd, prevEnd);
            } else {
                wrap.add(new Pair(prevStart, prevEnd));
                prevStart = currStart;
                prevEnd = currEnd;
            }
        }
        wrap.add(new Pair(prevStart, prevEnd));
        int N = wrap.size();
        int wrapper[][] = new int[N][2];
        for (int i = 0; i < N; i++) {

            wrapper[i][0] = wrap.get(i).start;
            wrapper[i][1] = wrap.get(i).end;

        }
        return wrapper;
    }
}
// Next Permutation
class Solution {
    static List<Integer> nextPermutation(int N, int arr[]) {
        // code here
        // Step 1
        int i_first = -1;
        for (int i = N - 2; i >= 0; i--) {
            if (arr[i] < arr[i + 1]) {
                i_first = i;
                break;
            }
        }
        // Step 2
        int i_second = 0;
        for (int i = N - 1; i >= 0; i--) {
            if (i_first != -1 && arr[i_first] < arr[i]) {
                i_second = i;
                break;
            }
        }
```

```
        // Step 3
        if (i_first != -1) {
            swap(arr, i_first, i_second);
        }

        // Step 4
        reverse(arr, i_first + 1, N - 1);

        List<Integer> ls = new ArrayList<>();
        for (int i = 0; i < N; i++) {
            ls.add(arr[i]);
        }
        return ls;

    }

    static void print(int arr[]) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    static void swap(int arr[], int s, int e) {
        int temp = arr[s];
        arr[s] = arr[e];
        arr[e] = temp;
    }

    static void reverse(int arr[], int s, int e) {
        while (s < e) {
            int temp = arr[s];
            arr[s] = arr[e];
            arr[e] = temp;
            s++;
            e--;
        }
    }
}
// Count Inversion
// Best time to buy and Sell stock
// Find all pairs on integer array whose sum is equal to given number
class Solution {
    int getPairsCount(int[] arr, int n, int k) {
        // code here
        int cnt = 0;
        HashMap<Integer, Integer> hm = new HashMap<>();
        for (int i = 0; i < n; i++) {
            cnt += hm.getOrDefault(k - arr[i], 0);
```

```
            hm.put(arr[i], hm.getOrDefault(arr[i], 0) + 1);
        }
        return cnt;
    }
}
// Find common elements In 3 sorted arrays
class Solution {
    ArrayList<Integer> commonElements(int A[], int B[], int C[], int n1, int n2, int n3) {
        // code here
        ArrayList<Integer> ls = new ArrayList<>();
        int ptr1 = 0, ptr2 = 0, ptr3 = 0;
        while (ptr1 < n1 && ptr2 < n2 && ptr3 < n3) {
            if (A[ptr1] == B[ptr2] && B[ptr2] == C[ptr3]) {
                if (ls.size() > 0 && A[ptr1] == ls.get(ls.size() - 1)) {
                    ptr1++;
                    ptr2++;
                    ptr3++;
                    continue;
                } else
                    ls.add(A[ptr1]);
                ptr1++;
                ptr2++;
                ptr3++;
            } else if (A[ptr1] < B[ptr2]) {
                ptr1++;
            } else if (B[ptr2] < C[ptr3]) {
                ptr2++;
            } else {
                ptr3++;
            }
        }
        return ls;
    }
}
// Rearrange the array in alternating positive and negative items with O(1)
// extra space
// Find if there is any subarray with sum equal to 0
class Solution {
    // Function to check whether there is a subarray present with 0-sum or not.
    static boolean findsum(int arr[], int n) {
        // Your code here
        HashMap<Integer, Integer> hm = new HashMap<>();
        hm.put(0, 1);
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += arr[i];
            if (hm.containsKey(sum)) {
                return true;
            }
```

```
                hm.put(sum, i);
            }
            return false;
        }
    }
    // Find factorial of a large number
    // Find maximum product subarray
    // Find longest consecutive subsequence
    // Given an array of size n and a number k, fin all elements that appear more
    // than " n/k " times. NA
    // Maximum profit by buying and selling a share at most twice
    // Find whether an array is a subset of another array
    // Find the triplet that sum to a given value
    // Trapping Rain water problem
    // Chocolate Distribution problem
    // Smallest Subarray with sum greater than a given value
    // Three way partitioning of an array around a given value
    // Minimum swaps required bring elements less equal K together
    // Minimum no. of operations required to make an array palindrome
    // Median of 2 sorted arrays of equal size
    // Median of 2 sorted arrays of different size
  }
}
```