```
public class BinarySearch {
    // Binary Search to find X in sorted array
    class Solution {
        int binarysearch(int arr[], int n, int ele) {
            // code here
            int start = 0;
            int end = n - 1;
            while (start <= end) {
                int mid = (start + end) / 2;
                if (ele == arr[mid])
                    return mid;
                else if (ele > arr[mid])
                    start = mid + 1;
                else
                    end = mid - 1;
            }
            return -1;
        }
    }

    // Implement Lower Bound
    class Solution {

        // Function to find floor of x
        // arr: input array
        // n is the size of array
        static int findFloor(long arr[], int n, long ele) {
            int start = 0;
            int end = arr.length - 1;
            int ans = -1;
            while (start <= end) {
                int mid = (start + end) / 2;
                if (arr[mid] == ele)
                    return mid;
                else if (arr[mid] < ele) {
                    ans = mid;
                    start = mid + 1;
                } else
                    end = mid - 1;
            }
            return ans;
        }

    }

    // Implement Upper Bound
    class Solution {
        Pair getFloorAndCeil(int[] arr, int n, int x) {
            // code here
```

```
        Arrays.sort(arr);
        int floor = getFloor(arr, n, x);
        int ceil = getCeil(arr, n, x);
        return new Pair(floor, ceil);

    }

    int getFloor(int arr[], int n, int ele) {
        int start = 0;
        int end = arr.length - 1;
        int ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (arr[mid] == ele)
                return arr[mid];
            else if (arr[mid] < ele) {
                ans = arr[mid];
                start = mid + 1;
            } else
                end = mid - 1;
        }
        return ans;
    }

    int getCeil(int arr[], int n, int ele) {
        int start = 0;
        int end = arr.length - 1;
        int ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (arr[mid] == ele)
                return arr[mid];
            else if (arr[mid] > ele) {
                ans = arr[mid];
                end = mid - 1;
            } else
                start = mid + 1;
        }
        return ans;
    }
}

// Search Insert Position
class Solution {
    static int searchInsertK(int arr[], int N, int ele) {
        // code here
        int start = 0;
        int end = arr.length - 1;
        int ans = N;
```

```
        while (start <= end) {
            int mid = (start + end) / 2;
            if (arr[mid] == ele)
                return mid;
            else if (arr[mid] > ele) {
                ans = mid;
                end = mid - 1;
            } else
                start = mid + 1;
        }
        return ans;
    }
}

// Check if Input array is sorted
class Solution {
    boolean arraySortedOrNot(int[] arr, int n) {
        // code here
        if (n == 0 || n == 1) {
            return true;
        }
        if (arr[n - 1] >= arr[n - 2]) {
            return arraySortedOrNot(arr, n - 1);
        } else {
            return false;
        }
    }
}
// Find the first or last occurrence of an element
class GFG {
    public int firstOccur(long[] arr, int ele) {
        int start = 0;
        int end = arr.length - 1;
        int ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (arr[mid] == ele) {
                ans = mid;
                end = mid - 1;
            } else if (ele > arr[mid])
                start = mid + 1;
            else
                end = mid - 1;
        }
        return ans;
    }

    public int lastOccur(long[] arr, int ele) {
        int start = 0;
```

```
        int end = arr.length - 1;
        int ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (arr[mid] == ele) {
                ans = mid;
                start = mid + 1;
            } else if (ele > arr[mid])
                start = mid + 1;
            else
                end = mid - 1;
        }
        return ans;
    }

    ArrayList<Long> find(long arr[], int n, int x) {
        // code here
        ArrayList<Long> ls = new ArrayList<>();
        int first = firstOccur(arr, x);
        if (first == -1) {
            ls.add((long) -1);
            ls.add((long) -1);
            return ls;
        }
        int last = lastOccur(arr, x);
        ls.add((long) first);
        ls.add((long) last);
        return ls;
    }
}

// Count occurrences of a number in a sorted array
class Solution {
    int count(int[] arr, int n, int x) {
        // code here
        if (firstOccur(arr, x) == -1) {
            return 0;
        }
        return lastOccur(arr, x) - firstOccur(arr, x) + 1;
    }

    public int firstOccur(int[] arr, int ele) {
        int start = 0;
        int end = arr.length - 1;
        int ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (arr[mid] == ele) {
                ans = mid;
```

```
            end = mid - 1;
        } else if (ele > arr[mid])
            start = mid + 1;
        else
            end = mid - 1;
    }
    return ans;
}

public int lastOccur(int[] arr, int ele) {
    int start = 0;
    int end = arr.length - 1;
    int ans = -1;
    while (start <= end) {
        int mid = (start + end) / 2;
        if (arr[mid] == ele) {
            ans = mid;
            start = mid + 1;
        } else if (ele > arr[mid])
            start = mid + 1;
        else
            end = mid - 1;
    }
    return ans;
}
}

// Find peak element
class Solution {
    public int findPeakElement(int[] nums) {
        int n = nums.length;
        if (n == 1) {
            return 0;
        }
        int start = 0, end = n - 1, ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (mid == 0) {
                return nums[0] >= nums[1] ? 0 : 1;
            } else if (mid == n - 1) {
                return nums[n - 1] >= nums[n - 2] ? n - 1 : n - 2;
            } else if (nums[mid] > nums[mid + 1] && nums[mid] > nums[mid - 1]) {
                return mid;
            } else if (nums[mid] < nums[mid - 1]) {
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }
```

```
        return ans;
    }
}

// Search in Rotated Sorted Array I
class Solution {
    public int search(int[] arr, int target) {
        int start = 0;
        int end = arr.length - 1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (arr[mid] == target)
                return mid;
            else if (arr[start] <= arr[mid]) {
                if (arr[start] <= target && target < arr[mid])
                    end = mid - 1;
                else
                    start = mid + 1;
            }
            // 2nd line
            else {
                // [mid, end] is sorted
                if (arr[mid] < target && target <= arr[end])
                    start = mid + 1;
                else
                    end = mid - 1;
            }
        }

        return -1;
    }
}

// Search in Rotated Sorted Array II
class Solution {
    public boolean search(int[] arr, int target) {
        int start = 0;
        int end = arr.length - 1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (arr[mid] == target) {
                return true;
            } else if (arr[start] < arr[mid]) {
                if (arr[start] <= target && target < arr[mid]) {
                    end = mid - 1;
                } else {
                    start = mid + 1;
                }
            }
```

```
            // 2nd line
            else if (arr[mid] < arr[start]) {
                // [mid, end] is sorted
                if (arr[mid] < target && target <= arr[end]) {
                    start = mid + 1;
                } else {
                    end = mid - 1;
                }
            } else {
                start += 1;
            }
        }

        return false;
    }
}

// Find minimum in Rotated Sorted Array rotated
class Solution {
    public int findMin(int[] arr) {
        int n = arr.length;
        int start = 0;
        int end = n - 1;
        while (start < end) {
            int mid = (start + end) / 2;
            if (arr[mid] < arr[end]) {
                end = mid;
            } else {
                start = mid + 1;
            }
        }

        return arr[start];
    }
}

// Single element in a Sorted Array
class Solution {
    public int singleNonDuplicate(int[] nums) {
        int low = 0, high = nums.length - 2;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (mid % 2 == 0) {
                if (nums[mid] == nums[mid + 1]) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
            } else {
```

```
                if (nums[mid] == nums[mid - 1]) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
            }
        }
        return nums[low];
    }
}
```

// Find kth element of two sorted arrays

```
class Solution {
    public long kthElement(int arr1[], int arr2[], int n, int m, int k) {
        if (m > n) {
            return kthElement(arr2, arr1, m, n, k);
        }

        int low = Math.max(0, k - m), high = Math.min(k, n);

        while (low <= high) {
            int cut1 = (low + high) >> 1;
            int cut2 = k - cut1;
            int l1 = cut1 == 0 ? Integer.MIN_VALUE : arr1[cut1 - 1];
            int l2 = cut2 == 0 ? Integer.MIN_VALUE : arr2[cut2 - 1];
            int r1 = cut1 == n ? Integer.MAX_VALUE : arr1[cut1];
            int r2 = cut2 == m ? Integer.MAX_VALUE : arr2[cut2];

            if (l1 <= r2 && l2 <= r1) {
                return Math.max(l1, l2);
            } else if (l1 > r2) {
                high = cut1 - 1;
            } else {
                low = cut1 + 1;
            }
        }
        return -1;

    }
}
```

// Find out how many times has an array been rotated

```
class Solution {
    int findKRotation(int arr[], int n) {
        // code here
        int start = 0;
        int end = n - 1;
        while (start < end) {
            int mid = (start + end) / 2;
```

```java
            if (arr[mid] < arr[end]) {
                end = mid;
            } else {
                start = mid + 1;
            }
        }

        return start;
    }
}

// Search in a 2 D matrix
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int rows = matrix.length;
        int cols = matrix[0].length;
        int start = 0;
        int end = (rows * cols) - 1;
        while (start <= end) {
            int mid = (start + end) / 2;
            int r = mid / cols;
            int c = mid % cols;
            if (matrix[r][c] == target) {
                return true;
            } else if (matrix[r][c] > target) {
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }
        return false;
    }
}
// Find Peak Element
class Solution {
    public int[] findPeakGrid(int[][] mat) {
        int startCol = 0;
        int endCol = mat[0].length - 1;
        while (startCol <= endCol) {
            int maxRow = 0;
            int midCol = startCol + (endCol - startCol) / 2;
            for (int row = 0; row < mat.length; row++) {
                maxRow = mat[row][midCol] >= mat[maxRow][midCol] ? row : maxRow;
            }
            boolean leftIsBig = midCol - 1 >= startCol && mat[maxRow][midCol - 1] >
mat[maxRow][midCol];
            boolean rightIsBig = midCol + 1 <= endCol && mat[maxRow][midCol + 1] >
mat[maxRow][midCol];
            if (!leftIsBig && !rightIsBig) {
```

```java
            return new int[] { maxRow, midCol };
        } else if (rightIsBig) {
            startCol = midCol + 1;
        } else {
            endCol = midCol - 1;
        }
    }
    return null;
  }
}
// Matrix Median
class Solution {
  public static int countSmallerThanMid(int[] A, int X, int n) {
    int low = 0, high = n - 1;
    while (low <= high) {
      int mid = (low + high) / 2;
      if (A[mid] <= X) {
        low = mid + 1;
      } else {
        high = mid - 1;
      }
    }
    return low;
  }

  int median(int[][] A, int row, int col) {
    int low = 1;
    int high = 1000000000;
    int n = row;
    int m = col;
    while (low <= high) {
      int midi = (low + high) / 2;
      int cnt = 0;
      for (int i = 0; i < n; i++) {
        cnt += countSmallerThanMid(A[i], midi, col);
      }
      if (cnt <= (n * m) / 2) {
        low = midi + 1;
      } else {
        high = midi - 1;
      }
    }
    return low;
  }
}
// Find square root of a number in log time
class Solution {
  long floorSqrt(long x) {
    // Your code here
```

```
        if (x == 0 || x == 1) {
            return x;
        }
        long start = 1, end = x, ans = 0;
        while (start <= end) {
            long mid = (start + end) / 2;
            if (mid * mid == x) {
                return mid;
            }
            if (mid * mid < x) {
                start = mid + 1;
                ans = mid;
            } else {
                end = mid - 1;
            }

        }
        return ans;
    }
}

// Find the Nth root of a number using binary search
class Solution {
    public int NthRoot(int n, int m) {
        // code here
        if (m == 0 || m == 1) {
            return m;
        }
        int start = 1, end = m, ans = 0;
        while (start <= end) {
            int mid = (start + end) / 2;
            int y = (int) Math.pow((double) mid, (double) n);
            if (y == m) {
                return mid;
            } else if (y < m) {
                start = mid + 1;
                ans = mid;
            } else {
                end = mid - 1;
            }

        }
        return -1;
    }
}

// Koko Eating Bananas
class Solution {
    public int minEatingSpeed(int[] piles, int h) {
```

```
        int minpile = Integer.MAX_VALUE;
        int maxpile = Integer.MIN_VALUE;
        int n = piles.length;
        for (int i = 0; i < n; i++) {
            if (piles[i] < minpile) {
                minpile = piles[i];
            }
            if (piles[i] > maxpile) {
                maxpile = piles[i];
            }
        }
        int start = 0, end = maxpile;
        int ans = maxpile;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (check(mid, piles, h)) {
                ans = mid;
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }
        return ans;
    }

    public boolean check(int eat, int arr[], int h) {
        double sum = 0;
        for (int i = 0; i < arr.length; i++) {
            sum += Math.ceil((double) arr[i] / (double) eat);
        }
        return sum <= h;
    }
}

// Minimum days to make M bouquets
class Solution {
    public boolean isPossible(int[] bloomDay, int days, int m, int k) {
        int boques = 0;
        int flower = 0;
        int n = bloomDay.length;
        for (int i = 0; i < n; i++) {
            if (bloomDay[i] <= days) {
                flower++;
            } else
                flower = 0;

            if (flower == k) {
                boques++;
                flower = 0;
```

```
                if (boques == m)
                    return true;
            }
        }
        return false;
    }

    public int minDays(int[] bloomDay, int m, int k) {
        int maxDays = Integer.MIN_VALUE;
        int n = bloomDay.length;

        for (int i = 0; i < n; i++) {
            maxDays = Math.max(maxDays, bloomDay[i]);
        }
        int start = 1;
        int end = maxDays;
        int ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (isPossible(bloomDay, mid, m, k) == true) {
                ans = mid;
                end = mid - 1;
            } else
                start = mid + 1;
        }

        return ans;
    }
}

// Find the smallest Divisor
class Solution {
    public int smallestDivisor(int[] nums, int threshold) {
        int minele = Integer.MAX_VALUE;
        int maxele = Integer.MIN_VALUE;
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            if (nums[i] < minele) {
                minele = nums[i];
            }
            if (nums[i] > maxele) {
                maxele = nums[i];
            }
        }
        int start = 1, end = maxele;
        int ans = maxele;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (check(mid, nums, threshold)) {
```

```
                ans = mid;
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }
        return ans;
    }

    public boolean check(int d, int arr[], int th) {
        int sum = 0;
        for (int i = 0; i < arr.length; i++) {
            sum += Math.ceil((double) arr[i] / (double) d);
        }
        return sum <= th;
    }
}

// Capacity to Ship Packages within D days
class Solution {
    public boolean isPossible(int[] weights, int truckWeight, int days) {
        int currWeight = 0;
        int currDay = 1;
        int n = weights.length;
        for (int i = 0; i < n; i++) {
            currWeight += weights[i];
            if (currWeight > truckWeight) {
                currWeight = weights[i];
                currDay++;
            }
        }
        return (currDay <= days);
    }

    public int shipWithinDays(int[] weights, int days) {
        int maxLoad = Integer.MIN_VALUE;
        int totalLoad = 0;
        int n = weights.length;

        for (int i = 0; i < n; i++) {
            maxLoad = Math.max(maxLoad, weights[i]);
            totalLoad += weights[i];
        }

        int start = maxLoad;
        int end = totalLoad;
        int ans = -1;
        while (start <= end) {
            int mid = (start + end) / 2;
```

```
            if (isPossible(weights, mid, days) == true) {
                ans = mid;
                end = mid - 1;
            } else
                start = mid + 1;
        }
        return ans;
    }
}

// Median of two sorted arrays
// Aggressive Cows
class Solution {
    static boolean isPossible(int a[], int n, int cows, int minDist) {
        int cntCows = 1;
        int lastPlacedCow = a[0];
        for (int i = 1; i < n; i++) {
            if (a[i] - lastPlacedCow >= minDist) {
                cntCows++;
                lastPlacedCow = a[i];
            }
        }
        if (cntCows >= cows)
            return true;
        else
            return false;
    }

    public static void main(String args[]) {
        Arrays.sort(a);
        int low = 1, high = a[n - 1] - a[0];
        while (low <= high) {
            int mid = (low + high) / 2;
            if (isPossible(a, n, cows, mid)) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
       return high;
    }
}

// Book Allocation Problem
public class Solution {
    public int books(ArrayList<Integer> A, int B) {
        if (A.size() < B)
            return -1;
        int low = A.get(0), high = 0;
```

```
        for (int i = 0; i < A.size(); i++) {
            high = high + A.get(i);
            low = Math.min(A.get(i), low);
        }
        int res = -1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (isPossible(A, mid, B)) {
                res = mid;
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
        return low;
    }

    public boolean isPossible(ArrayList<Integer> A, int pages, int B) {
        int sumAllocated = 0, cnt = 0;
        for (int i = 0; i < A.size(); i++) {
            if (sumAllocated + A.get(i) > pages) {
                cnt++;
                sumAllocated = A.get(i);
                if (sumAllocated > pages) {
                    return false;
                }
            } else {
                sumAllocated += A.get(i);
            }
        }
        if (cnt < B) {
            return true;
        }
        return false;
    }
}

// Split array - Largest Sum
class Solution {
    public int splitArray(int[] nums, int k) {

        int start = 0;
        int end = 0;
        int totalSum = 0, maxSum = 0;

        for (int x : nums) {
            totalSum += x;
            maxSum = Math.max(x, maxSum);
        }
```

```
        start = maxSum;
        end = totalSum;

        while (start < end) {
            int mid = start + (end - start) / 2;
            if (f(nums, mid, k)) {
                end = mid;
            } else {
                start = mid + 1;
            }
        }
        return end;
    }

    static boolean f(int[] nums, int target, int k) {

        int maxChunks = 1;
        int sum = 0;

        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];
            if (sum > target) {
                sum = nums[i];
                maxChunks++;
            }
        }
        return maxChunks <= k;
    }
}

// Kth Missing Positive Number
class Solution {
    public int findKthPositive(int[] arr, int k) {
        int low = 0;
        int high = arr.length - 1;
        int res = -1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            int diff = arr[mid] - mid - 1;
            if (diff < k) {
                res = arr[mid] + (k - diff);
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return res == -1 ? k : res;
    }
```

```
    }

    // Minimize Max Distance to Gas Station
    class Solution {
        public static double findSmallestMaxDist(int stations[], int K) {
            // code here
            int n = stations.length;
            double low = 0;
            double high = stations[n - 1] - stations[0];
            double mid = 0;
            while (high - low > (1e-6)) {
                mid = (low + high) / 2.00;
                if (possible(mid, stations, K)) {
                    high = mid;
                } else {
                    low = mid;
                }
            }
            return high;
        }

        public static boolean possible(double mid, int s[], int k) {
            int n = 0;
            for (int i = 0; i < s.length - 1; i++) {
                n += (int) ((s[i + 1] - s[i]) / mid);
            }
            return n <= k;
        }
    }
    // Median of 2 sorted arrays
    class Solution {
        public double findMedianSortedArrays(int[] nums1, int[] nums2) {
            return median(nums1, nums2, nums1.length, nums2.length);
        }

        double median(int arr1[], int arr2[], int m, int n) {
            if (m > n)
                return median(arr2, arr1, n, m);// ensuring that binary search happens on
  minimum size array

            int low = 0, high = m, medianPos = ((m + n) + 1) / 2;
            while (low <= high) {
                int cut1 = (low + high) >> 1;
                int cut2 = medianPos - cut1;

                int l1 = (cut1 == 0) ? Integer.MIN_VALUE : arr1[cut1 - 1];
                int l2 = (cut2 == 0) ? Integer.MIN_VALUE : arr2[cut2 - 1];
                int r1 = (cut1 == m) ? Integer.MAX_VALUE : arr1[cut1];
                int r2 = (cut2 == n) ? Integer.MAX_VALUE : arr2[cut2];
```

```
                if (l1 <= r2 && l2 <= r1) {
                    if ((m + n) % 2 != 0)
                        return Math.max(l1, l2);
                    else
                        return (Math.max(l1, l2) + Math.min(r1, r2)) / 2.0;
                } else if (l1 > r2)
                    high = cut1 - 1;
                else
                    low = cut1 + 1;
            }
            return 0.0;
        }
    }
    // Kth element of 2 sorted arrays

}
```