Task-10.1

Code :

```
import java.util.*;
class Solution {
   public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
      List<List<Integer>> result = new ArrayList<>();
      if (nums1.length == 0 || nums2.length == 0 || k == 0) return result;
      PriorityQueue<int[]> minHeap = new PriorityQueue<>(Comparator.comparingInt(a ->
(nums1[a[0]] + nums2[a[1]])));
      for (int i = 0; i < Math.min(k, nums1.length); i++) {
         minHeap.offer(new int[]{i, 0});  // {index in nums1, index in nums2}
      }

      while (k-- > 0 && !minHeap.isEmpty()) {
         int[] pair = minHeap.poll();
         int i = pair[0], j = pair[1];
         result.add(Arrays.asList(nums1[i], nums2[j]));
         if (j + 1 < nums2.length) {
            minHeap.offer(new int[]{i, j + 1});
         }
      }
      return result;
   }

   public static void main(String[] args) {
      Solution sol = new Solution();
      int[] nums1 = {1, 7, 11};
      int[] nums2 = {2, 4, 6};
      int k = 3;

      List<List<Integer>> result = sol.kSmallestPairs(nums1, nums2, k);
      System.out.println(result);  // Output: [[1, 2], [1, 4], [1, 6]]
   }
}
```

10.2 Kth Largest Element in an Array Using Min-Heap
Code:

```
import java.util.*;

class Solution {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int num : nums) {
            minHeap.offer(num); // Insert element

            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }

        return minHeap.peek();
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        int[] nums = {3,2,1,5,6,4};
        int k = 2;
        System.out.println(sol.findKthLargest(nums, k)); // Output: 5
    }
}
```

10.3 K-Closest Points to Origin

Code:

```java
import java.util.*;
class Solution {
    public int[][] kClosest(int[][] points, int k) {
        // Max-Heap to store k closest points
        PriorityQueue<int[]> maxHeap = new PriorityQueue<>(
            (a, b) -> Integer.compare((b[0] * b[0] + b[1] * b[1]), (a[0] * a[0] + a[1] * a[1]))
        );
        // Insert points into the max-heap
        for (int[] point : points) {
            maxHeap.offer(point);

            // If heap size exceeds k, remove the farthest point
            if (maxHeap.size() > k) {
                maxHeap.poll();
            }
        }
        // Convert heap to array
        int[][] result = new int[k][2];
        for (int i = 0; i < k; i++) {
            result[i] = maxHeap.poll();
        }
        return result;
    }
    public static void main(String[] args) {
        Solution sol = new Solution();
        int[][] points = {{3,3},{5,-1},{-2,4}};
        int k = 2;
        int[][] result = sol.kClosest(points, k);
        // Print output
        for (int[] p : result) {
            System.out.println(Arrays.toString(p));
        }
    }
}
```