Problem 1: Calculate (X^N) % 1,000,000,007

Overview: Efficiently compute the power of a number X raised to an exponent N, modulo 1,000,000,007. This is a common problem in computational mathematics, especially useful in fields like cryptography and number theory.

Algorithm Steps:

1. **Modular Exponentiation**:
    - Use a fast exponentiation algorithm while keeping the results within the modulo 1,000,000,007.
    - This process reduces the complexity from O(N) to O(log N).
2. **Algorithm Explanation**:
    - If N is even, recursively calculate (X^(N/2))^2.
    - If N is odd, return X * (X^(N-1)) % 1,000,000,007.
    - Utilize the property (a*b) % c = ((a % c) * (b % c)) % c.
3. **Handling Large N**:
    - Since N can be as large as 10^9, the algorithm avoids direct multiplication of large numbers which might cause overflow.
4. **Code**:

```
public class PowerModulo {

   private static final long MOD = 1000000007;

   // Function to calculate (x^n) % MOD
   public static long power(long x, long n) {
       // Base case
       if (n == 0) return 1;

       // If n is even
       long halfPower = power(x, n / 2);
       if (n % 2 == 0) {
           return (halfPower * halfPower) % MOD;
       } else {
           // If n is odd
           return (x * (halfPower * halfPower % MOD)) % MOD;
       }
   }

   // Main method to test the function
   public static void main(String[] args) {
       long x = 2; // Base
       long n = 10; // Exponent
       System.out.println("Power of " + x + " raised to " + n + " modulo " + MOD + " is: " + power(x, n));
   }
}
```

Problem 2, 3, 4

Problem 2: Quick Select

Overview: The Quick Select algorithm is an efficient method to find the k-th smallest (or largest) element in an unsorted array. This algorithm is related to the Quick Sort algorithm and has an average time complexity of O(N).

Algorithm Steps:

1. **Partition**:

- Choose a pivot element from the array.
- Rearrange the elements in the array such that elements less than the pivot come before it, and elements greater come after it. This is similar to the partition step in Quick Sort.

2. **Select**:
   - After the partition, check the pivot's position p.
   - If p is the target index k, return the pivot.
   - If p is greater than k, recursively apply Quick Select to the left sub-array.
   - If p is less than k, recursively apply Quick Select to the right sub-array.
3. **Handling Edge Cases**:
   - Ensure the array is non-empty and k is within the array bounds.
   - Handle cases where the array has duplicate values.
4. **Code**

```
public class QuickSelect {

   // Function to find the kth smallest element
   public static int quickSelect(int[] arr, int left, int right, int k) {
       if (left == right) {
           return arr[left];
       }

       int pivotIndex = partition(arr, left, right);

       if (k == pivotIndex) {
           return arr[k];
       } else if (k < pivotIndex) {
           return quickSelect(arr, left, pivotIndex - 1, k);
       } else {
           return quickSelect(arr, pivotIndex + 1, right, k);
       }
   }

   // Partition function used in Quick Select
   private static int partition(int[] arr, int left, int right) {
       int pivot = arr[right];
       int i = left;

       for (int j = left; j < right; j++) {
           if (arr[j] <= pivot) {
               swap(arr, i, j);
               i++;
           }
       }
       swap(arr, i, right);
       return i;
   }

   // Utility method to swap two elements in the array
   private static void swap(int[] arr, int i, int j) {
       int temp = arr[i];
       arr[i] = arr[j];
       arr[j] = temp;
   }

   // Main method to test the quickSelect function
   public static void main(String[] args) {
       int[] array = {12, 3, 5, 7, 4, 19, 26};
```

```
        int k = 3; // Find the 3rd smallest element
        System.out.println("3rd smallest element is: " + quickSelect(array, 0, array.length - 1, k - 1));
    }
}
```

Problem 3: Calculate Square Root with High Precision

Overview: Implement a square root calculation for a positive number with up to six decimal places of precision without using the built-in square root function. The algorithm will use the Babylonian (Heron's) method, an iterative technique for finding the square root.

Algorithm Steps:

1. **Initial Guess**:
    - Start with an initial guess. A good default choice is number / 2.
2. **Iterative Process**:
    - Improve the guess iteratively using the Babylonian formula:
      newGuess = (guess + number / guess) / 2.
    - Repeat this process until the change in guesses is less than a small threshold (e.g., 0.000001 for six decimal places).
3. **Handling Edge Cases**:
    - Ensure the number is positive.
    - Handle the special case when the number is 0 or 1 directly.
4. **Code**

```
public class SquareRootPrecision {

    // Function to calculate the square root of a number with high precision
    public static double sqrt(double number) {
        if (number < 0) {
            throw new IllegalArgumentException("Number must be non-negative");
        }
        if (number == 0 || number == 1) {
            return number;
        }

        double threshold = 0.000001;
        double guess = number / 2;
        double newGuess;

        while (true) {
            newGuess = (guess + number / guess) / 2;
            if (Math.abs(newGuess - guess) < threshold) {
                break;
            }
            guess = newGuess;
        }

        return guess;
    }

    // Main method to test the sqrt function
    public static void main(String[] args) {
        double number = 9; // Example number
        System.out.println("Square root of " + number + " with precision: " + sqrt(number));
    }
}
```

Problem 4: Determine a Valid Course Order Given Prerequisites

Overview: Implement an algorithm to find a valid order to take courses given their prerequisite relationships. The courses and their prerequisites are represented in a 2D array, where a cell [i, j] = 1 indicates that course j is a prerequisite for course i.

Algorithm Steps:

1. **Representing Prerequisites**:
    - Use a 2D array to represent prerequisites. Each row corresponds to a course, and each column represents whether another course is a prerequisite.
2. **Topological Sorting**:
    - Apply a topological sort algorithm to determine a valid order of courses.
    - Use a directed graph where each course is a node, and an edge from j to i exists if course j is a prerequisite for course i.
3. **Detecting Cycles**:
    - Ensure there are no cycles in the graph, as a cycle would mean it's impossible to complete all courses (as per Note 2).
    - Use depth-first search (DFS) or a similar algorithm to detect cycles.
4. **Code**

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class CourseScheduler {

   // Function to find a valid order of courses
   public static List<Integer> findOrder(String[] courseNames, int[][] prerequisites) {
       int numCourses = courseNames.length;
       List<Integer>[] graph = new ArrayList[numCourses];
       int[] inDegree = new int[numCourses];

       // Create graph and in-degree array
       for (int i = 0; i < numCourses; i++) {
           graph[i] = new ArrayList<>();
       }
       for (int i = 0; i < prerequisites.length; i++) {
           for (int j = 0; j < prerequisites[i].length; j++) {
               if (prerequisites[i][j] == 1) {
                   graph[j].add(i);
                   inDegree[i]++;
               }
           }
       }

       // Topological sort using Kahn's algorithm
       Queue<Integer> queue = new LinkedList<>();
       for (int i = 0; i < numCourses; i++) {
           if (inDegree[i] == 0) {
               queue.offer(i);
           }
       }

       List<Integer> order = new ArrayList<>();
       while (!queue.isEmpty()) {
           int course = queue.poll();
           order.add(course);
           for (int nextCourse : graph[course]) {
               inDegree[nextCourse]--;
```

```
                if (inDegree[nextCourse] == 0) {
                    queue.offer(nextCourse);
                }
            }
        }

        if (order.size() == numCourses) {
            return order;
        } else {
            return new ArrayList<>(); // Return empty list if no valid order exists
        }
    }

    // Main method to test the findOrder function
    public static void main(String[] args) {
        String[] courseNames = {"Course 0", "Course 1", "Course 2", "Course 3"};
        int[][] prerequisites = {
            {0, 0, 0, 0},
            {1, 0, 1, 0},
            {0, 0, 0, 1},
            {1, 0, 0, 0}
        };

        List<Integer> order = findOrder(courseNames, prerequisites);
        if (order.isEmpty()) {
            System.out.println("No valid order exists");
        } else {
            System.out.println("A valid order of courses:");
            for (int course : order) {
                System.out.println(courseNames[course]);
            }
        }
    }
}
```