1.

```
import java.util.Scanner;

class Main {

    public void helper_function() {
        Scanner sc = new Scanner(System.in);

        // Read the number of test cases
        int T = sc.nextInt();

        // Loop through each test case
        for (int t = 0; t < T; t++) {
            // Read the number of cells
            int N = sc.nextInt();

            // Read the edge array
            int[] edge = new int[N];
            for (int i = 0; i < N; i++) {
                edge[i] = sc.nextInt();
            }

            // Find and print the sum of the largest cycle for this test case
            int result = largestSumCycle(N, edge);
            System.out.println(result);
        }

        sc.close();
    }

    public static void main(String[] args) {
        Main m = new Main();
        m.helper_function();
    }

    public static int largestSumCycle(int N, int[] edge) {
        boolean[] visited = new boolean[N];
        boolean[] inStack = new boolean[N];
        int[] stack = new int[N];
        int maxCycleSum = -1;

        for (int i = 0; i < N; i++) {
            if (!visited[i]) {
                int cycleSum = dfs(i, edge, visited, inStack, stack, 0);
                if (cycleSum != -1) {
                    maxCycleSum = Math.max(maxCycleSum, cycleSum);
                }
            }
        }

        return maxCycleSum;
    }

    private static int dfs(int node, int[] edge, boolean[] visited, boolean[] inStack, int[] stack, int
stackIndex) {
        if (visited[node]) {
```

```
            if (inStack[node]) {
                int cycleSum = 0;
                for (int i = stackIndex - 1; i >= 0; i--) {
                    cycleSum += stack[i];
                    if (stack[i] == node) break;
                }
                return cycleSum;
            }
            return -1;
        }

        visited[node] = true;
        inStack[node] = true;
        stack[stackIndex] = node;

        int nextNode = edge[node];
        int cycleSum = -1;
        if (nextNode != -1) {
            cycleSum = dfs(nextNode, edge, visited, inStack, stack, stackIndex + 1);
        }

        inStack[node] = false;
        return cycleSum;
    }
}
```

2.

```
import java.util.*;

class Main {

    public static void main(String[] args) {
        Main m = new Main();
        m.helper_function();
    }

    public void helper_function() {
        Scanner sc = new Scanner(System.in);

        // Read the number of test cases
        int T = sc.nextInt();

        // Loop through each test case
        for (int t = 0; t < T; t++) {
            // Read the number of cells
            int N = sc.nextInt();

            // Read the edge array
            int[] edge = new int[N];
            for (int i = 0; i < N; i++) {
                edge[i] = sc.nextInt();
            }

            // Read the two cells for which the nearest meeting cell is to be found
            int src = sc.nextInt();
            int dest = sc.nextInt();
```

```
            // Find and print the nearest meeting cell for this test case
            int result = nearestMeetingCell(N, edge, src, dest);
            System.out.println(result);
        }

        sc.close();
    }

    public static int nearestMeetingCell(int N, int[] edge, int src, int dest) {
        // Get the distances from src and dest to all other cells
        int[] distFromSrc = getDistances(N, edge, src);
        int[] distFromDest = getDistances(N, edge, dest);

        // Find the nearest meeting cell
        int minDistance = Integer.MAX_VALUE;
        int meetingCell = -1;
        for (int i = 0; i < N; i++) {
            if (distFromSrc[i] != Integer.MAX_VALUE && distFromDest[i] != Integer.MAX_VALUE) {
                int maxDist = Math.max(distFromSrc[i], distFromDest[i]);
                if (maxDist < minDistance) {
                    minDistance = maxDist;
                    meetingCell = i;
                }
            }
        }

        return meetingCell;
    }

    private static int[] getDistances(int N, int[] edge, int start) {
        int[] dist = new int[N];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[start] = 0;

        Queue<Integer> queue = new LinkedList<>();
        queue.offer(start);

        while (!queue.isEmpty()) {
            int node = queue.poll();
            int nextNode = edge[node];
            if (nextNode != -1 && dist[nextNode] == Integer.MAX_VALUE) {
                dist[nextNode] = dist[node] + 1;
                queue.offer(nextNode);
            }
        }

        return dist;
    }
}
```

3.

```
import java.util.Scanner;

class Main {
```

```
    public static void main(String[] args) {
        Main m = new Main();
        m.helper_function();
    }

    public void helper_function() {
        Scanner sc = new Scanner(System.in);

        // Read the number of test cases
        int T = sc.nextInt();

        // Loop through each test case
        for (int t = 0; t < T; t++) {
            // Read the number of cells
            int N = sc.nextInt();

            // Read the edge array
            int[] edge = new int[N];
            for (int i = 0; i < N; i++) {
                edge[i] = sc.nextInt();
            }

            // Find and print the node with the maximum weight for this test case
            int result = maxWeightNode(N, edge);
            System.out.println(result);
        }

        sc.close();
    }

    public static int maxWeightNode(int N, int[] edge) {
        // Array to store the weight of each node
        int[] weights = new int[N];

        // Calculate the weight of each node
        for (int i = 0; i < N; i++) {
            if (edge[i] != -1) {
                weights[edge[i]] += i;
            }
        }

        // Find the node with the maximum weight
        int maxWeight = -1;
        int maxWeightNode = -1;
        for (int i = 0; i < N; i++) {
            if (weights[i] > maxWeight) {
                maxWeight = weights[i];
                maxWeightNode = i;
            }
        }

        return maxWeightNode;
    }
}
```