Merge Lists:

```
SinglyLinkedListNode* mergeLists(SinglyLinkedListNode* head1, SinglyLinkedListNode* head2) {
    SinglyLinkedListNode dummy;
    SinglyLinkedListNode* tail = &dummy;
    dummy.next = NULL;
    while (1) {
        if (head1 == NULL) {
            tail->next = head2;
            break;
        } else if (head2 == NULL) {
            tail->next = head1;
            break;
        }
       if (head1->data <= head2->data) {
            tail->next = head1;
            head1 = head1->next;
        } else {
            tail->next = head2;
            head2 = head2->next;
        }
        tail = tail->next;
    }
    return dummy.next;
}
```

Truck tour:

```
class Result {public static int truckTour(List<List<Integer>> petrolpumps) {
        int n = petrolpumps.size();
        int totalPetrol = 0;
        int totalDistance = 0;
        int currentPetrol = 0;
        int startIndex = 0;
      for (int i = 0; i < n; i++) {
            int petrol = petrolpumps.get(i).get(0);
            int distance = petrolpumps.get(i).get(1);
            totalPetrol += petrol;
            totalDistance += distance;
            currentPetrol += petrol - distance;
               if (currentPetrol < 0) {
               startIndex = i + 1;
               currentPetrol = 0;
            }
        }

        if (totalPetrol < totalDistance) {
            return -1; // Impossible to complete the tour
        }
         return startIndex;
    }
}
```

Poisonous plants:

```
public static int poisonousPlants(List<Integer> p) {
        int n = p.size();
        Stack<int[]> stack = new Stack<>();
        int maxDays = 0;

        for (int i = 0; i < n; i++) {
            int days = 0;
            while (!stack.isEmpty() && stack.peek()[0] >= p.get(i)) {
                days = Math.max(days, stack.pop()[1]);
            }
            if (stack.isEmpty()) {
                days = 0;
            } else {
                days++;
            }
            maxDays = Math.max(maxDays, days);
            stack.push(new int[]{p.get(i), days});
        }

        return maxDays;
    }
}
```

Lowest common ancestor:

```
public static Node lca(Node root, int v1, int v2) {
            while (root != null) {
          if (root.data > v1 && root.data > v2) {
             root = root.left;
          } else if (root.data < v1 && root.data < v2) {
             root = root.right;
          } else {
             break;
          }
       }
       return root;
    }
```

Height of binary tree:

```
if (root == null) {
          return -1; // Return -1 if the tree is empty to ensure the height of an empty tree is -1
       } else {
          int leftHeight = height(root.left);
          int rightHeight = height(root.right);
          return 1 + Math.max(leftHeight, rightHeight); // Height of tree is max of left or right subtree +
1 for the root
}
}
```