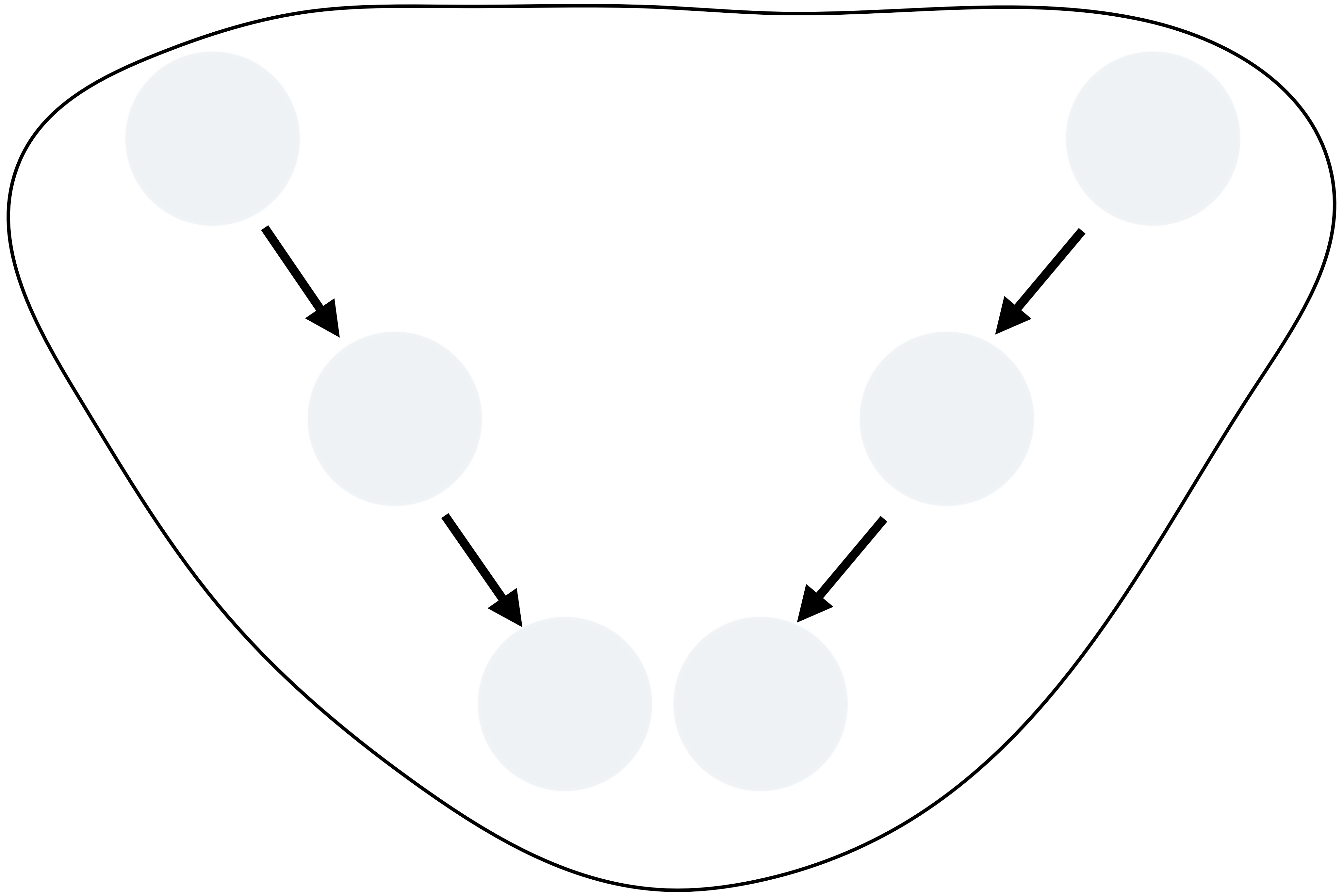
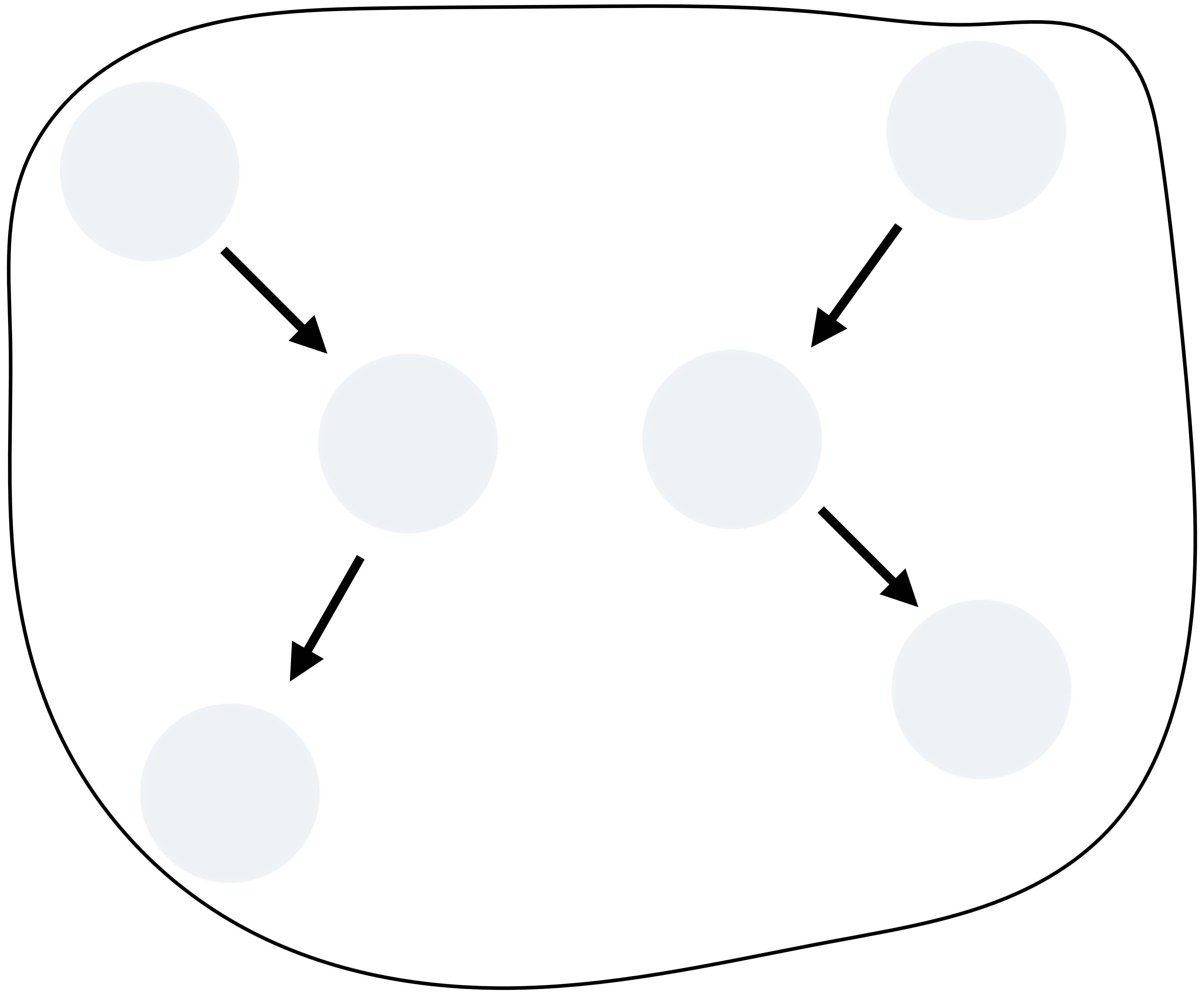
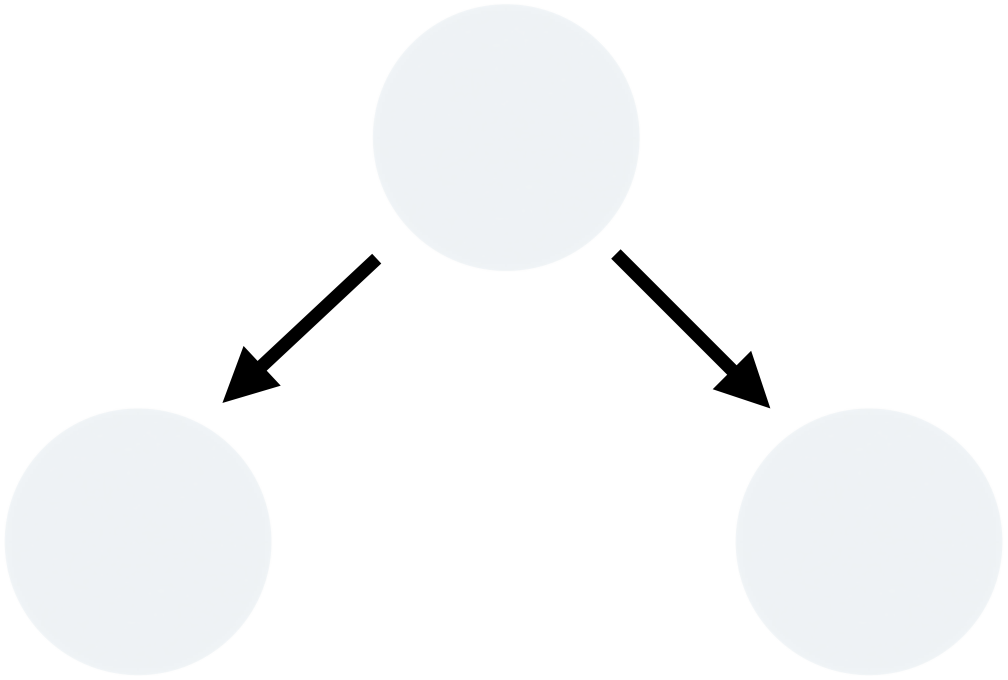
COUNT THE NUMBER OF STRUCTURALLY UNIQUE BINARY TREES POSSIBLE

COUNT THE NUMBER OF STRUCTURALLY UNIQUE BINARY TREES POSSIBLE

SAY YOU HAVE
3 NODES







THESE ARE THE VARIOUS
POSSIBLE SHAPES OF
TREES

NOTE THAT SOME OF
THESE ARE MIRRORS OF
EACH OTHER

COUNT TREES

```java
public static int countTrees(int numNodes) {
    if (numNodes <= 1) {
        return 1;
    }

    int sum = 0;
    for (int i = 1; i <= numNodes; i++) {
        int countLeftTrees = countTrees(i - 1);
        int countRightTrees = countTrees(numNodes - i);

        sum = sum + (countLeftTrees * countRightTrees);
    }

    return sum;
}
```

WHEN THE NUMBER OF NODES IS 1 THERE IS JUST ONE POSSIBLE TREE - THIS IS THE BASE CASE

CONSIDER THAT EVERY NODE CAN BE THE ROOT - THE NODES BEFORE IT WILL BE ON THE LEFT AND THE NODES AFTER IT ON THE RIGHT

NODES ON THE LEFT AND RIGHT FORM THEIR OWN SUBTREES

THIS IS THE NUMBER OF POSSIBLE TREES WITH THIS ROOT - THE COMBINATION OF RIGHT AND LEFT SUBTREES

PRINT ALL NODES WITHIN A RANGE IN A BINARY SEARCH TREE

PRINT ALL NODES WITHIN A RANGE IN A BINARY SEARCH TREE

A RANGE WILL INCLUDE A SUBSET OF NODES IN THE BINARY SEARCH TREE

THIS SUBSET CAN INCLUDE 0 NODES AS WELL

CHECK EVERY NODE TO SEE IF IT'S VALUE IS WITHIN THE RANGE - PRINT IT TO SCREEN IF THE RANGE CONSTRAINTS ARE MET

PRINT NODES WITHIN A RANGE

```java
public static void printRange(Node<Integer> root, int low, int high) {
    if (root == null) {
        return;
    }

    if (low <= root.getData()) {
        printRange(root.getLeftChild(), low, high);
    }

    if (low <= root.getData() && root.getData() <= high) {
        System.out.println(root.getData());
    }

    if (high > root.getData()) {
        printRange(root.getRightChild(), low, high);
    }
}
```

PASS IN THE MIN AND MAX INDICATING THE RANGE WE CARE ABOUT

BASE CASE, NOTHING TO DO FOR A NULL ROOT

IF THE RANGE LOW VALUE IS LESS THAN THE CURRENT NODE, RUN THE OPERATION ON THE LEFT SUBTREE

CHECK THE NODE VALUE TO SEE IF IT'S WITHIN THE RANGE - IF YES, PRINT!

IF THE RANGE HIGH VALUE IS GREATER THAN THE CURRENT NODE, RUN THE OPERATION ON THE RIGHT SUBTREE

CHECK IF A BINARY TREE IS A BINARY *SEARCH* TREE

CHECK IF A BINARY TREE IS A BINARY *SEARCH* TREE

FOR EVERY NODE IN A BINARY SEARCH TREE - THE NODES WITH VALUES <= NODE ARE IN THE LEFT SUBTREE AND NODES WITH VALUES > NODE ARE IN THE RIGHT SUBTREE

CHECK EVERY NODE TO SEE IF THIS CONSTRAINT IS VIOLATED

THIS CAN BE SOLVED ITERATIVELY AND RECURSIVELY

IS BINARY SEARCH TREE?

PASS IN THE MIN AND MAX INDICATING THE RANGE FOR THE SUBTREE

```java
public static boolean isBinarySearchTree(Node<Integer> root, int min, int max) {
    if (root == null) {
        return true;
    }

    if (root.getData() <= min || root.getData() > max) {
        return false;
    }

    return isBinarySearchTree(root.getLeftChild(), min, root.getData())
            && isBinarySearchTree(root.getRightChild(), root.getData(), max);
}
```

A NULL NODE IS A VALID BINARY TREE

IF ANY NODE LIES OUTSIDE THE RANGE THEN THE BST CONSTRAINT HAS BEEN VIOLATED AND WE RETURN FALSE

FOR THE LEFT SUBTREE THE CURRENT NODE'S VALUE SHOULD BE THE MAX

FOR THE RIGHT SUBTREE THE CURRENT NODE'S VALUE SHOULD BE THE MIN

CHECK THE LEFT AND RIGHT SUBTREES TO SEE IF THEY'RE VALID SEARCH TREES - NOTE HOW THE RANGE FOR THE CHECKS CHANGE