Algorithmic Analysis of an Iterative and Recursive countDistricts Method in Java

public static int countDistrictsIterative(List<Integer> locs, int numCols)   
{  
 ArrayList<Integer> newLocs = (ArrayList<Integer>) locs; 🡺 O(1)  
 int numDistricts = newLocs.size(); 🡺 O(1)  
  
 for(int i = 0; i < newLocs.size(); i++) 🡺 O(n) //It must iterate  
 { // n times.  
 if(newLocs.get(i) - newLocs.get(i+1) == -1) 🡺 O(1)  
 numDistricts = numDistricts - 1; 🡺 O(1)  
 if(*binaryIsMember*(newLocs, newLocs.get(i) + numCols)) 🡺 O(log(n)) // binary  
 numDistricts = numDistricts - 1; 🡺 O(1) // search  
 }  
 return numDistricts;  
}

In this iterative algorithm the runtime complexity is clearly O(n) because of the for- loop. This algorithm will always iterate through the entire array once and only once.

public static int countDistrictsRecursive(List<Integer> locs, int numCols)  
{  
 ArrayList<Integer> newLocs = (ArrayList<Integer>) locs;   
 int numDistricts = 0;  
  
 if(newLocs.size() == 0)  
 numDistricts = 0;  
 else if(newLocs.get(0) - newLocs.get(1) == -1 && *binaryIsMember*(newLocs, newLocs.get(0) + numCols))  
 {  
 newLocs.remove(newLocs.indexOf(newLocs.get(0) + numCols));  
 newLocs.remove(0);  
 numDistricts = numDistricts - 1 + *countDistrictsRecursive*(newLocs, numCols);  
 }  
 else if(newLocs.get(0) - newLocs.get(1) == -1)  
 {  
 newLocs.remove(0);  
 numDistricts = numDistricts - 1 + *countDistrictsRecursive*(newLocs, numCols);  
 }  
 else if(*binaryIsMember*(newLocs, newLocs.get(0) + numCols))  
 {  
 newLocs.remove(newLocs.indexOf(newLocs.get(0) + numCols));  
 numDistricts = numDistricts - 1 + *countDistrictsRecursive*(newLocs, numCols);  
 }  
 else  
 {  
 newLocs.remove(0);  
 numDistricts = numDistricts + *countDistrictsRecursive*(newLocs, numCols);  
 }  
  
 return numDistricts;  
}

Here the runtime complexity is slightly more complicated. Best case is that the locs list is either empty which runs in O(1) or containing every possible index, indices only in one row, or indices only in one column in which case the runtime is O(n/2). The worst case for this algorithm is O(n) when there are no adjacencies to help remove extra indices from the locs list on each successive recursive call. Therefore the algorithm is actually o(n).