1. (25 pts)

Suppose you are given two **sorted** linked lists of integers of arbitrary lengths. The difference of the first list with respect to the second is the set of all entries that are in the first list but not in the second, *in the same order as in the original first list.* For example:

```
ptr1 L1: 3->9->12->15->21                    Difference L1-L2 : 9->15->21
ptr2 L2: 2->3->6->12->19->25->81
                                                  9->15->21
```

Implement a NON-RECURSIVE method to compute the difference and return it in a NEW linked list. The original linked lists should not be modified. The new linked list should have a complete new set of Node objects (not shared with the original lists). You may assume that neither of the original lists has any duplicate items, and either of them could be empty.

Your implementation MUST use the efficient $O(m+n)$ algorithm covered in Problem Set 2 on a similar problem: using a pointer per list to track simultaneously, traversing each list exactly once. If you use some other inefficient algorithm, you will get AT MOST HALF the credit. You may implement helper methods if needed.

L1	L2
null	null
null	not null
not null	null

```
public class Node {
   public int data; public Node next;
   public Node(int data, Node next) {
       this.data = data; this.next = next;
   }
}
```

```
// Creates a new linked list consisting of the items that are in the first
// list but not in the second, in the same order as in the first list
// Returns the front of this new linked list
public static Node difference(Node frontL1, Node frontL2) {
        // COMPLETE THIS METHOD - YOU MAY ***NOT*** USE RECURSION
   if(frontL2== null && frontL1 != null){
          return frontL1; X
   }
   if(frontL1 == null || frontL2==null){
          return null;
   }
   Node ptr1 = frontL1;
   Node ptr2= frontL2;
   Node front =null;
while( ptr1!= null ){
if(ptr2 == null){
    front=new Node(ptr1.data,front);
    ptr1=ptr1.next;
}else if(ptr1.data<ptr2.data){
    front=new Node(ptr1.data,front);
      ptr1=ptr1.next;

}else if(ptr2.data < ptr1.data){
      ptr2=ptr2.next;

  }else{
        ptr2= ptr2.next;
        ptr1=ptr1.next;
   }
 }
 return reverse(front);
}

private static Node reverse(Node front){
  Node prev=null;
  Node ptr=front;
  Node second =front.next;
    while (ptr!=null){
      ptr.next = prev;
      prev= ptr;
      ptr= second;
      if(ptr!=null){
       second=second.next
    }
   }
}} return prev;
```

2. (25 pts, 5+5+7+8)

For parts (a) and (b) below:

You are given the following modified sequential search code for searching in an array in which values are in sorted increasing order:

```
public boolean search(int[] A, int target) {
   for (int i=0; i < A.length; i++) {
      if (target == A[i]) return true;
      if (target < A[i]) return false;
   }
   return false;
}
```

a) For an array of length n, work out an exact formula in n (not big O) for the average number of target-to-array item comparisons for successful searches. (Do not count the < comparison in the for loop header.) Assume equal probabilities for matching against any item. You don't need to simplify your answer down to a single term.

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

- comparison for success (==)

Summation of numbers: $\frac{n(n+1)}{2}$

length of array: n

$$\frac{\frac{n(n+1)}{2}}{n} = \frac{\cancel{n}(n+1)}{2\cancel{n}}$$

average number of comparison for successful search: $\frac{n+1}{2}$

b) Repeat the above for the average number of target-to-array item comparisons for failure, assuming equal probabilities for all failure situations.

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

~~The average numb~~

$\times$

Summation of: $\frac{n(n+1)}{2}$

comparison number for failure (<)

length of array: n

$$\frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2}$$

average number of comparisons for failure $\frac{n+1}{2}$

c) In a sorted (ascending) integer array of length n with no duplicates, you are asked to find and print all values in the range x to y, inclusive ($x < y$). Assume both x and y are in the array. Describe the fastest algorithm to accomplish this. (Give concise steps, not Java code.) What is the worst case big O running time of your algorithm if there are k integers within the range? (Assume k is much less than n, and unit time for printing.)

For each element in the array, once it reaches x, we can start printing then till the value is greater than y where we can break.

The worst case big O of this algorith is O(n) because we still have have to go through most of the length of the array n. We end up with O(n) even if we still go through less of the elements.

—4

d) An array of length n holds student scores on an exam. Scores are 0..200, in multiples of 10. Compute the number of students for each possible score (i.e. how many scored 0, 10, 20 ... 200). Describe the fastest algorithm to accomplish this, using extra storage space if needed. What is the worst case big O running time? (You can ignore time taken to allocate any extra storage.)

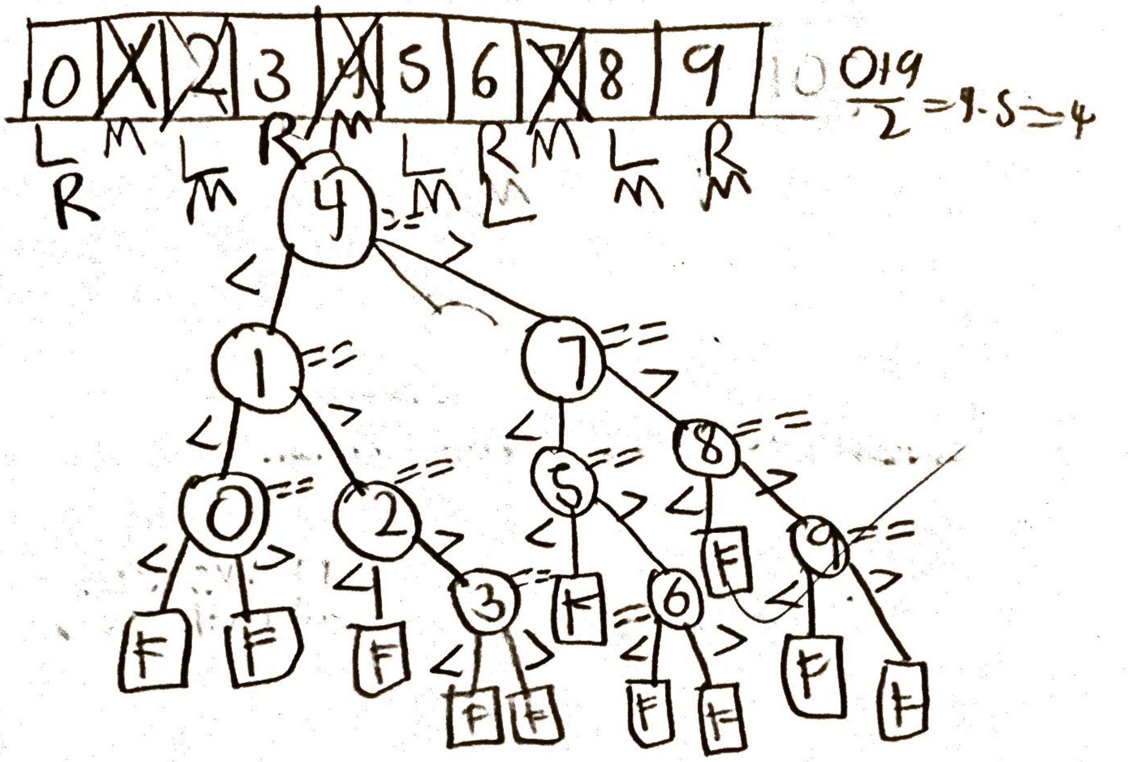
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200

Make 21 counters for each multiple of score (0,10,20..200). For each score (since we know they are multiples of 10) we can match them to its supposed storage and store them into the counter.
The big O is O(n) since we have to go through each element which is a size of length n.

✓

3. (25 pts, 10+7+8)

a) Draw the comparison tree for binary search on a sorted array of length 10. Be sure to include failure nodes, and to mark comparisons on the nodes and branches.



$\frac{0+9}{2} = 4.5 \approx 4$

b) Consider a sorted array of 5 items on which searches are made, with probabilities of match being 0.15, 0.1, 0.2, 0.3, 0.25 for the items, respectively. Would the average number of comparisons for success be better when using binary search on the original sorted array, or using sequential search on an optimal rearrangement of the items? Show your work.

The average number of comparisons is better for sequential search as 2.5 < 3.3

0.3	0.25	0.2	0.15	0.1

sequential on rearragement

$(0.3\times1)+(0.25\times2)+(0.2\times3)+(0.15\times4)+(0.1\times5)$

$0.3 + 0.5 + 0.6 + 0.6 \quad 0.5$

$0.15 \times 4 = 0.60$

$0.15 \times 3 = 0.45$

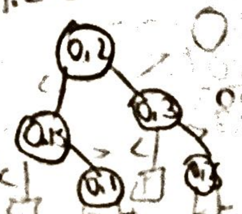
$0.25 \times 5 = 1.25$

$0.6 + 0.8 = 1.4$

$1.4 + 0.6 = 2.0$

$2.0 + 0.5$

2.5 average number of comparisons for sequential



$(0.2\times1)+(0.15\times3)+(0.3\times3)+(0.1\times5)+(0.25\times5) =$

$0.2 + 0.45 = 0.65$

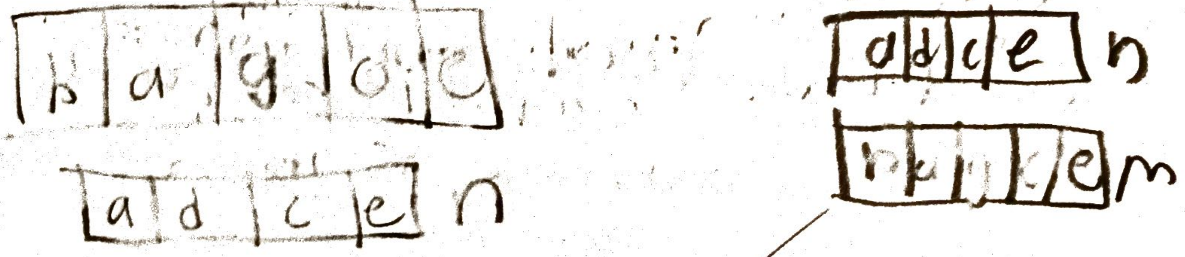
$0.65 + 0.9 = 1.55$

$1.55 + 0.5 = 2.05$

$2.05 + 1.25 = 3.30$

3.3 average number for binary

c) You are asked to find the common names in two unsorted arrays of names of lengths m and n, with no duplicates in either. You may not sort or modify either of the inputs in any way. Describe the scenario for the best case running time, and derive the big O for it.



The scenario best case of this algorithm is $O(\min(m,n)^2)$ as the length of an array are same resulting to comparing each element in min(m,n) length array with the elements in length min(m,n) array