Problem Set 6, Part I

Problem 1: Counting unique values

1-1) If all of the objects are unique, because then the second for loop will run just as many times as the first for loop

1-2)

```
for (int i = 0; i < arr.length; i++) { → N
        // does arr[i] also appear somewhere "after"
        // (i.e., to the right) of position i?
        boolean appearsAfter = false; →

        for (int j = i + 1; j < arr.length; j++) { (N - (i+1))
            if (arr[j] == arr[i]) {
                appearsAfter = true;
                break;
            }
        }

        // only count arr[i] if it doesn't appear
        // anywhere to the right of position i
        if (! appearsAfter) {
            count++;
        }
```

O(n) = N(N-(i+1)) = N^2 -Ni+ N → O(n^2), worst case scenario, inner loop doesn't break from arr[j] == arr[i]

1-3) O(n^2) because it will run both loops until it traverses the entire array

1-4) The best case occurs when there are all duplicate numbers, so "arr[j] == arr[i]"...

1-5)...will get called essentially every time, which means that the Big O notation would just be O(n)

Problem 2: Improving the efficiency of an algorithm

2-1)

```
public static int numUnqiue(int[] arr){
        mergeSort(arr);
        int numUnique = 0;
        for(int i = 0; i < arr.length; i++){
            //System.out.println(i+1);
            if(i < arr.length-1 && arr[i] == arr[i+1]){
                //System.out.println("True");
                //System.out.println("arr[i+1] && arr[i]: " + "" +
(arr[i+1]) + "" + "" + arr[i]);
                continue;
            }
            numUnique++;
        }
        return numUnique;

    }
```

2-2) It's when the worst case is for merge sort since it's nlogn which is bigger than O(n), so nlogn

2-3)No, in the best case scenario, it still runs nlogN, so the worst case got better, but the best case got worse, it seems that the sorting allowed us to know where the duplicates are, but the act of sorting seems to have added a big O complexity nlogn in either case

Problem 3: Practice with references

3-1)

Expression	**Address**	**Value**
n	0x128	0x800
n.ch	Ox800	‘e’
n.next	0x802	0x240
n.prev.next	0x182	0x800
n.next.prev	0x246	Ox800
n.next.prev.prev	0x806	Ox180

3-2)

```
m.next = n;
m.prev = n.prev
n.prev = m;
```

3-3)

```
public static void addNexts(DNode tail){
      DNode current = tail;
      Dnode prevNode = tail.prev;
      //Dnode next;
      while(current != null){
            prevNode.next = current;
            current = prevNode
            prevNode = current.prev
      }


}
```

Problem 4: Printing the odd values in a list of integers

4-1)

```
public static void printOddsRecur(IntNode first){
      //prints odd values in the linked list
      if(IntNode.next == null){ //no link at end
            System.out.println("");
      }
      int printOdds = printOddsRecur(first.next)
      if(first.val % 2 == 0){ //even
            System.out.println(first.toString())

      } else{
            System.out.println("");
      }

}
```

4-2)

```
public static void printOddsRecur(IntNode first){
      //prints odd values in the linked list

      while(first.next != null){
            if(first.val % 2 == 0) //even
            {
            System.out.println(first.toString())
            }
            else //odd
            {
            System.out.println("");
            }
      }

}
```