Problem Set 6, Part I

Problem 1: Counting multiples in a linked list of integers

1-1)

```
public static int numMultiplesIter(IntNode first, int m) {
      int numMul=0;
      IntNode trav= first;
      while (trav!=null){
            if((trav.val % m)==0){
                  numMul++;
            }
            trav=trav.next;
      }
      return numMul;
}
```

1-2)

```
public static int numMultiplesRec(IntNode first, int m) {
      int count=0;
      if(first==null){
            return 0;}
      else{
            int rest=numMultiplesRec(first.next,m);
            if((first.val % m)==0){
                  count++;}
            return count+rest;
}
```

Problem 2: Comparing two algorithms

2-1) *time efficiency of algorithm A:* O(n)

Explanation: Algorithm A has a time efficiency of O(n^2) because when adding items from aList to the list, it traverses the entire array from the start to end, which gives an O(n) as n is the length of the array. The getItem method has an O(1) as the aList is an ArrayList and therefore it has random access. The addItem method calls getNode(i-1) and in the getNode method the while loop traverses the linked list until it reaches the node at i-1. This at the average time efficiency gives O(n) as it must walk down the list until it reaches i-1 .Therefore in total, the algorithm has a time efficiency of O(n^2)

2-2) *time efficiency of algorithm B:* O(n)

Explanation: Algorithm B has a time efficiency of O(n) because when adding items from aList to the list, it traverses the entire array from the end to start, which gives an O(n) as n is the length of the array. Similar to Algorithm A, the getItem method has a time efficiency of O(1) as the aList is an ArrayList and therefore it has random access. The addItem method has a time efficiency of O(1) as it takes in 0 for the index i, therefore when the addItem method calls getNode(i-1), it does not traverse through the while loop as -1 is not < -1 (0-1). This gives this method a time efficiency of O(1). Therefore, the algorithm has a time efficiency of O(n)

2-3) Algorithm B is more efficient than A as it traverses the ArrayList backwards and adds the items to the beginning of the Linked List. By adding items to the beginning, there is not extra time spent traversing the LLList to add the item to index i, as it is added to index 0.

Problem 3: Choosing an appropriate representation

3-1) *ArrayList or LLList? LLList*

Explanation: Because the order size varies from month to month it is best to use an LLList as you do not know the size/amount of orders. Additionally, because the orders are displayed from end to front to ensure that this is most efficient, the LLList can be filled from end to start (in other words when you add an order you add it to the beginning) that way, when displaying the orders the most recent order would be the first and therefore you would not have to traverse the entire LLList to display the most recent order.

3-2) *ArrayList or LLList? ArrayList*

Explanation: Because the number of workshops are fairly consistent you can use an ArrayList as it can be initialized with a set size. Also, because arraylists have random access it is easier to store and access information in chronological order when using an ArrayList.

3-3) *ArrayList or LLList? ArrayList*

Explanation: You know the set number of courses and therefore ArrayLists are best to use. Additionally if you need to frequently access the items in the list, it is easiest to use an ArrayList because they have random access. Rather then having to traverse the entire list, you can get the item at a certain index and therefore it is more efficient to use an ArrayList

Problem 4: Reversing a linked list

4-1)

The worst case is that the entire for loop has to execute, and the for loop has a time efficiency of O(n). Additionally, getItem(i) calls getNode(i). In the getNode method it traverses the LLList until it finds the node at index i. Worst Case for this is that it has to traverse the entire LLList which would mean it has a time efficiency of O(n). The addItem method also calls getNode, and similarly the worst case for this would be if it has to traverse the entire LLList giving an O(n) time efficiency. Lastly the length method simply returns length therefore it has an O(1). An O(n) * (O(n) + O(n) + O(1)) gives an overall worst-case time efficiency of O(n^2).

4-2)

```
public static LLList reverse(LLList list) {
      LLList rev = new LLList();
      ListIterator iter=list.iterator();
      while(iter.hasNext()){
            Object itemAt=iter.next();
            rev.addItem(itemAt,0);
      }
      return rev;
      }
```

4-3)

The worst case running time of this algorithm is O(n). The while loop has a time efficiency of O(n) as it has to traverse the LLList. The next() method on the ListerIterator Object has an O(1) as it simply returns a reference to the next Object and this is independent of the length of the list. Similarly the hasNext() method simply returns true if the nextNode is not null, therefore it is O(1). Overall, the running time is O(n).