Problem Set 6, Part I

Andy Vo

Problem 1: Printing the odd values in a linked list of integers

1-1)

```
public static void printOddsRecursive(IntNode first) {
    if (first == null) {
        return;
    } else {
        if (first.val % 2 == 1) {
            System.out.println(first.val);
            printOddsRecursive(first.next);
        } else {
            printOddsRecursive(first.next);
}
```

1-2)

```
public static void printOddsIterative(IntNode first) {
    if (first == null) {
        return;
    }
    IntNode trav = first;
    while (trav != null) {
        if (trav.val % 2 == 1) {
            System.out.println(trav.val);
        }
        trav = trav.next;
    }
}
```

Problem 2: Comparing two algorithms

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

Explanation: This algorithm is O(n) linear time because we iterate through one for loop "n" times. Furthermore, we do not need to worry about getItem because we are pulling from an ArrayList, which has random access, or O(1) constant time. Lastly, our addItem method for algorithm A is adding to the beginning of the LLList, and so we do not need to iterate through the entirety of the linked list to add the node.

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

Explanation: This algorithm is quadratic time as a result of the following: the algorithm iterates through a for loop "n" times, and within each iteration of the for loop, we use the addItem method to add at the end of the linked list. As a result, we must traverse through the linked list "n" times to add the node. Note, however, like algorithm A, aList.getItem(i) grabs an item from an array that has constant time O(1). Together, we have O(n^2) which can be written as $O(n^2)$.

2-3)

Algorithm A is more efficient than algorithm B because of its time complexity. Both allocate the same amount of memory for variables such as "head" and "length". Both also iterate through the "for" loop the same amount of "n" times. What they differ in is where each algorithm adds its respective new Node object using the addItem method. In algorithm A, we add each new node to the beginning since we iterate from the end of the array. This is the most optimal input for the addItem method because we don't have to iterate through the entirety of the linked list to add another node.

Problem 3: Choosing an appropriate representation

3-1) *ArrayList or LLList?* ArrayList.

Explanation: For this specific scenario, the ArrayList would be a better application to serve the needs of this user. A couple of key hints point in the direction of ArrayList. To begin with, the user states that they don't tend to change the types of tools they sell and as a result, items are "seldom added or removed from the list," which is great for arrays since we don't have to worry about shifting often. In the event that someone does purchase a product, we will have to remove an item from the list of that type of product. This is also best suited for an ArrayList because we can simply remove the last item of the list in constant O(1) time and keep track of how many are left because of random access compared to the traversal of linked lists. Speaking of which, we are told that the "product's position in the list is also its product ID" and so ArrayList would be best here not only because of random access but also because the position of an object is related to its index in an array. Thus, by random access, we can modify contents at O(1) time.

3-2) *ArrayList or LLList?* LLList.

Explanation: For this specific scenario, the LLList would be a better application to serve the needs of this user. The critical clue for this scenario is that the "number of tweets can vary widely from week to week in a way that is hard to predict." With that in mind, accurately preallocating memory for the week's number of tweets is near impossible and it would be very inefficient to shift tweets once the array is full. Thus, it is best to use an LLList since it can never be full. Displaying tweets in reverse chronological order would not be difficult if we use a doubly linked list and carry a reference to the most recently added tweet (node) to the list. From there, we can traverse "backward" through the LLList.

3-3) *ArrayList or LLList?* ArrayList.

Explanation: For this specific scenario, the ArrayList would be a better application to serve the needs of this user. We are told that the number of events per month is rather consistent, which allows us to preallocate sufficient memory for each month's list of events. With lists being displayed in chronological order and that we know the lengths of lists, we can simply run a for loop or whatever other method is most preferable to display events.

Problem 4: Improving the efficiency of an algorithm

4-1)

The worst-case running time of this algorithm is O(m^2 + mn^2). Imagine if everything in list1 is also in list2. The first for loop iterates “m” times and will use getItem for list1 “m” times. Nested under every “m”, the algorithm iterates through list2 “n” times and uses getItem “n” times, and then has to use addItem and add the node at the end of the linked list which requires “n” iterations for O(mn^2).

4-2)

```
public static LLList intersect1(LLList list1, LLList list2) {
    LLList inters = new LLList();

    ListIterator iter1 = list1.iterator();
    if (list1 == null || list2 == null) {
        throw new IllegalArgumentException();
    }
   while (iter1.hasNext()) {
        Object one = iter1.next();
        ListIterator iter2 = list2.iterator();
        while (iter2.hasNext()) {
            Object two = iter2.next();
            if (one.equals(two)) {
                inters.addItem(one, 0);
            }
        }
    }
    return inters;
}
```

4-3)

In this new algorithm, we implement two new iterators associated with the LLList class so that we don’t have to iterate using for loops and then have to use getItem for each for loop iteration. Using the iterators and given that “m” is list1 and “n” is list2, our new worst-case running time would be O(mn) because we essentially have a nested iterator inside an iterator. The “addItem” method will always be O(1) because we add a node to the beginning of the linked list and that is the best case always.