Problem Set 3: Part I

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

1-1)

```
public static void printOddsRecursive(IntNode first) {
    if (first == null){
        return;
    }
    if (first.val % 2 != 0){
        System.out.println(first.val);
    }
    printOddsRecursive(first.next);
}
```

1-2)

```
public static void printOddsIterative(IntNode first){
    while (first != null){
        if (first.val % 2 != 0){
            System.out.println(first.val);
        }
        first = first.next;
    }
}
```

Problem 2: Improving the efficiency of an algorithm

2-1)

In the worst case (when there are no matches or the matches are at the very end of list2), there are m(m+1) / 2 operations involved in the calls to list1.getItem() and m * (n(n+1) / 2) operations in the calls to list2.getItem(). Simplifying and combining these equations, there are $m^2 / 2 + m/2 + n^2 * m / 2 + mn / 2$ operations. The terms with the largest exponents are m^2 and n^2, so this part of the algorithm has time complexity of $O(m^2 + n^2)$.

In the worst case (where every element in n and m is a match) there are (m*n -1) * (m*n / 2) operations in the addItem method. This simplifies to $((m^2 * n^2)/ 2) - (m*n / 2)$. The terms with the largest exponents are $m^2 * n^2$, so the time complexity of this part of the algorithm is $O(m^2 * n^2)$. Since $m^2 * n^2 > m^2 + n^2$ as m and n grow large, the overall time complexity is $O(m^2 * n^2)$.

2-2)

```
public static LLList intersect(LLList list1, LLList list2) {
    ListIterator list1Iterator = list1.iterator();
    LLList inters = new LLList();

    while (list1Iterator.hasNext()){
        Object item1 = list1Iterator.next();
        ListIterator list2Iterator = list2.iterator();
        while (list2Iterator.hasNext()){
            Object item2 = list2Iterator.next();
            if (item1.equals(item2)){
                inters.addItem(item2, 0);
            }
        }
    }
}
```

2-3)

Using an iterator for list1 and list2 reduces the time complexity of retrieving the next item to O(1), although list1Iterator is called m times and list2Iterator is called m * n times. Inserting the intersections at position 0 of inters instead of at the end of inters reduces that operation to O(1), although in the worst case inters.addItem can be called m*n times. Therefore, the time complexity of the new algorithm is O(n*m).

Problem 3: Initializing a doubly linked list

```
public static void initNexts(DNode last) {
    if (last == null){
        return;
    }
    Dnode trav = last;
    Dnode prevNode = last.prev;
    while (prevNode != null){
        prevNode.next = trav;
        trav = prevNode;
        prevNode = trav.prev;
    }
    return trav;
}
```

Problem 4: Using a queue to search a stack

Pseudocode:

```
Given Stack S, Queue Q, and Item I:

found = false
While not S isEmpty:
    pop nextItem from S
    if nextItem equals I:
        found = true
    insert nextItem in Q

While not Q isEmpty:
    remove nextItem from Q
    push nextItem to S

(Items are now back in Q but in reverse order)

While not S isEmpty:
    pop nextItem from S
    insert nextItem in Q

While not Q isEmpty:
    remove nextItem from Q
    push nextItem to S
```