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 == 1){
          System.out.println(first.val);
       }

       printOddsRecursive(first.next);

}
```

1-2)

```
public static void printOddsIterative(IntNode first) {

       if(first == null){
          return;
       }

       if(first.val % 2 == 1){
          System.out.println(first.val);
       }

       first = first.next;
}
```

Problem 2: Improving the efficiency of an algorithm

2-1)

It is O(m^2 * n^2) in the worst case.

- **O(m) for the first outer loop**
- **O(m) for getItem called on list1. getItem calls getNode which traverses the entire list in the worst case**
- **O(n) for the inner loop**
- **O(n) again for the getItem method call on list2**

I am omitting the call for addItem even though it traverses through the list formed by the intersection of the two lists. We may assume it has a size of the minimum of those since it is an intersection. However, if we enter the conditional block, we are breaking out of the inner loop early and that would stop it from becoming the worst case. The worst case happens when the conditional never evaluates to true

2-2)

```
public static LLList intersect(LLList list1, LLList list2) {

      LLList inters = new LLList();
      ListIterator iterator = list1.iterator();
      ListIterator iterator2 = list2.iterator();

      while(iterator.hasNext()){

         Object item1 = iterator.next();

         while(iterator2.hasNext()){

             if(item1.equals(iterator2.next())){
                 inters.addItem(item1, inters.length());
                 break;
             }
         }
      }

      return inters;
}
```

2-3)

It is big O(m*n) in the worst case. The efficiency is a square root of what we previously had. The outer loop traverses the list m times and the inner list n times. Again we exit early if we found a match from the inner loop so the path where the conditional evaluates to true and addItem is called is not in the worst case. The worst case is when the two lists are disjoint and the conditional never evaluates to true

Problem 3: Initializing a doubly linked list

```
public static DNode initNexts(DNode last) {

      if(last == null || last.prev == null){
         return last;
      }

      while(last.prev != null){
         last.prev.next = last;
         last = last.prev;
      }

      return last;
}
```

Problem 4: Using a queue to search a stack

```
static <T> boolean findItemInStack(Stack<T> stack, T item){

   boolean itemFound = false;

   Queue<T> queue; //this needs to be initialized

   while(!stack.isEmpty()){

       T itemFromStack = stack.pop();
       queue.insert(itemFromStack);

       if(itemFromStack.equals(item)){
           itemFound = true;
           break;
       }
   }

   while(!queue.isEmpty()){
       stack.push(queue.remove());
   }

   return itemFound;
}
```