# Homework 2

*cas2324*

## Problem 1

I wrote it two ways, one using the `contains` method which is probably pretty inefficient and then a better way that only iterates through each list a maximum of once.

```
public static void printLots(List<Integer> L, List<Integer> P) {
    Iterator iter = L.iterator();
    // Get the first value in the list L, cast object to int
    int num = (Integer) iter.next();
    // Set up a counter to track where the iterator is
    int counter = 0;
    for (Integer i : P) {
        // Don't waste our time if index is out of bounds
        if (i >= L.size()) {
            continue;
        }
        // Iterate counter and iterator until we reach the index
        while (counter < i) {
            counter++;
            num = (Integer) iter.next();
        }
        // If we get here, counter is equal to i and in bounds so we can
        // just print it out
        System.out.println(num);
    }
}

// Another implementation, more inefficient though, especially if we're
// only trying to print out a few early values.
public static void printLots(List<Integer> L, List<Integer> P) {
    Iterator iter = L.iterator();
    int num = (Integer) iter.next();
    int counter = 0;
    while (iter.hasNext()) {
        if (P.contains(counter)) {
            System.out.println(num);
        }
        num = (Integer) iter.next();
        counter++;
    }
}
```

## Problem 2: Weiss 3.4

*Given two sorted lists, `L1` and `L2`, write a procedure to compute `L1` intersection `L2` using only the basic list operations.*

- Create new list to store the intersection
- Instantiate an iterator, `iter`, on the first list and grab its first value in a var `num`
- Iterate through second list for each value `i`
- While `num < i`, `num = item.next()`
- Outside that loop, check if `num == i` and if so, add to our new list

I wanted to test this so wrote up the method as well, seems to work well. I added a method to keep track of the last added intersection so as to not introduce duplicate integers into the new list.

```
public static List<Integer> intersection(List<Integer> L1, List<Integer> L2) {
    // Create new list to store the intersection
    List<Integer> L3 = new ArrayList<>();

    // If either list is empty, just return the empty list
    if (L1.size() == 0 || L2.size() == 0) {
        return L3;
    }

    // Instantiate an iterator for the first list and grab its first value
    Iterator iter1 = L1.iterator();
    Integer num = (Integer) iter1.next();

    // Go through each value of the second list.  We could also have
    // used a while loop with iter2.hasNext() and incremented that way
    // but the advanced for loop is a bit easier.
    for (Integer i : L2) {
        // Compare our current value from L1 with our value from L2 and
        // if the L1 value is smaller, since the lists are sorted, just
        // go to the next value until that condition is not true.
        // Also make sure we're not at the end of the list by checking
        // hasNext().
        while (num < i && iter1.hasNext()) {
            num = (Integer) iter1.next();
        }
        // Now we know `num` is either equal to or greater than the
        // element in L2.  If it's equal, add to our intersection
        // list, otherwise, there's no match and we should go on
        // to the next value.
        Integer lastAdded;
        if (num == i) {
            if (L3.size() == 0) {
                L3.add(num);
            } else {
                // Since we don't want duplicates (via HW FAQ), check
                // to make sure this value doesn't equal the most
                // recently added one.
                lastAdded = L3.get(L3.size() - 1);
                if (num != lastAdded) {
                    L3.add(num);
                }
            }
        }
    }
    return L3;
}
```

# Problem 3: Weiss 3.24

*Write routines to implement two stacks using only one array. Your stack routines should not declare an overflow unless every slot in the array is used.*

```
import java.util.EmptyStackException;

/**
 * Created by alexscott on 10/6/16.
 */
public class TwoStack<AnyType> {
    // Default size for our internal array
    private static final int DEFAULT_CAPACITY = 10;

    // Internal array for storing the stacks
    private Object[] arr;

    // Variables to track the next open position in the
    // two stacks.
    private int leftStackPos = 0;
    private int rightStackPos;

    // Default constructor
    public TwoStack() {
        // Create the array
        arr = new Object[DEFAULT_CAPACITY];

        // Assign first open position of right stack
        rightStackPos = arr.length - 1;
    }

    // Constructor for setting capacity
    public TwoStack(int capacity) {
        // Create the array
        arr = new Object[capacity];

        // Assign first open position of right stack
        rightStackPos = arr.length - 1;
    }

    // Push item onto the top of the left stack
    public void push(AnyType el) {
        if (leftStackPos > rightStackPos) {
            throw new RuntimeException("TwoStack is full!");
        }
        arr[leftStackPos++] = el;
    }

    // Push item onto the top of the right stack
    public void pushRight(AnyType el) {
        if (rightStackPos < leftStackPos) {
            throw new RuntimeException("TwoStack is full!");
        }
        arr[rightStackPos--] = el;
    }

    // Return the top item on the left stack, and decrement our
```

```
    // position
    public AnyType pop() {
        if (leftStackPos == 0) {
            throw new EmptyStackException();
        }
        return((AnyType) arr[--leftStackPos]);
    }

    // Return the top item on the right stack, and increment our
    // position (since this stack is reversed)
    public AnyType popRight() {
        if (rightStackPos == arr.length - 1) {
            throw new EmptyStackException();
        }
        return((AnyType) arr[++rightStackPos]);
    }

    // Return the top item on the left stack
    public AnyType peek() {
        if (leftStackPos == 0) {
            throw new EmptyStackException();
        }
        return((AnyType) arr[leftStackPos - 1]);
    }

    // Return the top item on the right stack
    public AnyType peekRight() {
        if (rightStackPos == arr.length - 1) {
            throw new EmptyStackException();
        }
        return((AnyType) arr[rightStackPos + 1]);
    }
}
```

# Problem 4

## 4a.

- Input train: `[5, 9, 6, 7, 2, 8, 1, 3, 4]`
- Expected output train: `[9, 8, 7, 6, 5, 4, 3, 2, 1]`

Sequence of steps:

1. Put `4` into the first holding track
2. Put `3` into the first holding track
3. Put `1` onto the back of the output track
4. Put `8` into the second holding track
5. Put `2` onto the back of the output track
6. Put `7` into the second holding track
7. Put `6` into the second holding track
8. Put `9` into the third holding track
9. Move `3` onto the back of the output track
10. Move `4` onto the back of the output track

11. Put `5` onto the back of the output track
12. Move `6` onto the back of the output track
13. Move `7` onto the back of the output track
14. Move `8` onto the back of the output track
15. Move `9` onto the back of the output track

## 4b.

`[1, 9, 8, 7, 6, 5, 4, 3, 2]`

Even though it's very close to what we want on the output track, we can't solve this rearrangement given the constraints.

Since the output track has to be loaded as a queue (so lowest number first), we have to ensure that `1` goes first. Since it's at the very end, we'll have to put all the other cars into the holding tracks.

But since each holding track is a LIFO stack, we need to make sure we never put a car with a larger number on top of a car with smaller number.

So in this case, we'll do:

- Put `2` in the first holding track
- Put `3` in the second holding track
  - Put `4` in the third holding track
  - Put `5` ...hmmm...we're already stuck.