

Homework 2, Written

1.

```
/**
 * Goal: Print the elements in L that are in index positions specified by P.
 * Problem: Need to somehow "get" the int at position i in list L.
 * I use an enhanced for loop to traverse L and an iterator to traverse P.
 * The big-O running time should be O(N).
 */
public static void printLots(List<Integer> L, List<Integer> P) {

    /* Makes sure that both lists have at least one element. */
    if (L.isEmpty() || P.isEmpty()) {
        System.out.println("One of the lists is empty!");
        return;
    }

    /* Makes sure that P's elements stay within L's bounds. */
    for (int element : P) {
        if (element >= L.size()) {
            System.out.println("ERROR. Make sure that each " +
                "integer in P is less than the # of elements in L.");
            System.out.println("Try again!");
            return;
        }
        if (element < 0) {
            System.out.println("ERROR. Please make sure that P " +
                "contains no negative integers!");
            return;
        }
    }

    /**
     * I use an iterator to iterate through P.
     * counter keeps track of the index in L for the enhanced for loop.
     * This approach assumes that P will have no repeated elements.
     * (A TA on Piazza said that this should be safe to assume.)
     */
    Iterator<Integer> itrP = P.iterator();
    int position = itrP.next();
    int counter = -1;

    for (int element : L) {
        counter++;
    }
}
```

```

        if (counter == position) {
            System.out.println(element);
            if (itrP.hasNext()) {
                position = itrP.next();
            }
        }
    }
}

```

2. Weiss 3.4

Basic list operations used:

- add(element e)
- get(index i)
- size()
- Iterator's next()
- Iterator's hasNext()

PSEUDOCODE:

```

/* The big-O running time should be O(N). */
public <anyType extends Comparable<anyType>> void findIntersection
(List<anyType> L1, List<anyType> L2) {
    Iterator<anyType> itr1 = L1.iterator();
    Iterator<anyType> itr2 = L2.iterator();
    List<anyType> L3 = new ArrayList<>();

    /* If one of the lists is empty, there's no intersection! */
    if (L1.isEmpty() || L2.isEmpty()) {
        System.out.println("Intersection: " + L3);
        return;    // or return L3!
    }

    anyType elementL1 = itr1.next();
    anyType elementL2 = itr2.next();

    /**
     * I use this while loop to keep track of two iterators and to
     * compare elements from both lists. I use the Comparable interface's
     * compareTo method to see which element is bigger. If list one's
     * element is greater, I advance the 2nd iterator (if there's room).
     * If list two's element is greater, I advance the 1st iterator
     * (if there's room).
     * If both elements are "equal", then go to the inner if-else.
     * If L3 is empty, add elementL1. Or, if L3's last element
     * isn't equal to elementL1, then add elementL1. After that, I

```

```

    * advance the iterators that return true for hasNext().
    */
while (itr1.hasNext() || itr2.hasNext()) {
    if (elementL1.compareTo(elementL2) == 1) {
        if (itr2.hasNext()) {
            elementL2 = itr2.next();
        } else {
            break;
        }
    } else if (elementL1.compareTo(elementL2) == -1) {
        if (itr1.hasNext()) {
            elementL1 = itr1.next();
        } else {
            break;
        }
    } else {
        if (L3.size() == 0) {
            L3.add(elementL1);
        } else if (L3.get(L3.size() - 1).compareTo(elementL1) != 0) {
            L3.add(elementL1);
        }
        if (itr1.hasNext()) {
            elementL1 = itr1.next();
        }
        if (itr2.hasNext()) {
            elementL2 = itr2.next();
        }
    }
}

/**
 * Last pair to compare!
 * If the two elements are equal, then go to the inner if-else.
 * If L3 is empty, add elementL1. Or, if L3's last element
 * isn't equal to elementL1, then add elementL1.
 */
if (elementL1.compareTo(elementL2) == 0) {
    if (L3.size() == 0) {
        L3.add(elementL1);
    } else if (L3.get(L3.size() - 1).compareTo(elementL1) != 0) {
        L3.add(elementL1);
    }
}

System.out.println("Intersection: " + L3);    // or return L3!
}

```

3. Weiss 3.24

PSEUDOCODE:

```
/**
 * Stack one starts at the beginning of the array and goes toward the end.
 * Stack two starts at the end of the array and goes toward the start.
 */
// The first two instance variables would normally be initialized in the constructor.
private anyType[] wholeArray = (anyType[]) new Object[n];
private int topOfStack2 = n;
private int topOfStack1 = -1;

/**
 * Adds an element x to the "top" of stack one.
 */
void push1(anyType x) {
    topOfStack1++;
    if (topOfStack1 >= topOfStack2) {
        topOfStack1--;
        System.out.println("Stack overflow!");
        return;
    }
    wholeArray[topOfStack1] = x;
}

/**
 * Adds an element x to the "top" of stack two.
 */
void push2(anyType x) {
    topOfStack2--;

    if (topOfStack2 <= topOfStack1) {
        topOfStack2++;
        System.out.println("Stack overflow!");
        return;
    }
    wholeArray[topOfStack2] = x;
}

/**
 * @return the element at the "top" of stack one.
 * Also decrements topOfStack1 after.
 * @return null if stack one is empty.
 */
```

```

anyType pop1() {
    if (topOfStack1 == -1) {
        System.out.println("Stack one is empty.");
        return null;
    } else {
        return wholeArray[topOfStack1--];
    }
}

/**
 * @return the element at the "top" of stack two.
 * Also increments topOfStack2 after.
 * @return null if stack two is empty.
 */
anyType pop2() {
    if (topOfStack2 == wholeArray.length) {
        System.out.println("Stack two is empty.");
        return null;
    } else {
        return wholeArray[topOfStack2++];
    }
}

/**
 * @return the element at the "top" of stack one.
 * @return null if stack one is empty.
 * Leaves topOfStack1 unchanged.
 */
anyType peek1() {
    if (topOfStack1 == -1) {
        System.out.println("Stack one is empty.");
        return null;
    } else {
        return wholeArray[topOfStack1];
    }
}

/**
 * @return the element at the "top" of stack two.
 * @return null if stack two is empty.
 * Leaves topOfStack2 unchanged.
 */
anyType peek2() {
    if (topOfStack2 == wholeArray.length) {
        System.out.println("Stack two is empty.");
        return null;
    }
}

```

```

    } else {
        return wholeArray[topOfStack2];
    }
}

int size1() {
    return topOfStack1 + 1;
}

int size2() {
    return wholeArray.length - topOfStack2;
}

boolean isEmpty1() {
    if (topOfStack1 == -1) {
        return true;
    }
    return false;
}

boolean isEmpty2() {
    if (topOfStack2 == wholeArray.length) {
        return true;
    }
    return false;
}

```

4.a.

Two queues: input, output

Three stacks: S1, S2, S3

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

For this problem's queues, enqueue is addFirst and dequeue is removeLast!

Also, this solution only needs two holding tracks!

Sequence of steps for the solution:

1. S1.push(input.dequeue()) // push 4 to the top of S1. S1 should be [4].
2. S1.push(input.dequeue()) // push 3 to the top of S1. S1 should be [3, 4].
3. output.enqueue(input.dequeue()) // moves 1 from input to output; output should be [1].
4. S2.push(input.dequeue()) // pushes 8 to the top of S2. S2 should be [8].
5. output.enqueue(input.dequeue()) // moves 2 from input to output; output should be [2, 1].
6. output.enqueue(S1.pop()) // pops 3 from S1; output should be [3, 2, 1].
7. output.enqueue(S1.pop()) // pops 4 from S1; output should be [4, 3, 2, 1].
8. S2.push(input.dequeue()) // pushes 7 to the top of S2. S2 should be [7, 8].

```

9. S2.push(input.dequeue())           // pushes 6 to the top of S2. S2 should be [6, 7, 8].
10. S1.push(input.dequeue())          // pushes 9 to the top of S1. S1 should be [9].
11. output.enqueue(input.dequeue())   // moves 5 from input to output; output is [5, 4, 3, 2, 1].
12. output.enqueue(S2.pop())          // pops 6 from S2; output should be [6, 5, 4, 3, 2, 1].
13. output.enqueue(S2.pop())          // pops 7 from S2; output should be [7, 6, 5, 4, 3, 2, 1].
14. output.enqueue(S2.pop())          // pops 8 from S2; output should be [8, 7, 6, 5, 4, 3, 2, 1].
15. output.enqueue(S1.pop())          // pops 9 from S1; output should be [9, 8, 7, 6, 5, 4, 3, 2, 1].

```

In Java code:

(Note: The Queue interface's enqueue adds at the end, and its dequeue removes and returns the first element. The interface is different from this problem's queue, which essentially adds elements to the front of the list and removes and returns elements at the end of the list.)

```

import java.util.LinkedList;
import java.util.Queue;

public class written4 {
    public static void main(String[] args) {
        Queue<Integer> input = new LinkedList<>();
        Queue<Integer> output = new LinkedList<>();
        MyStack<Integer> S1 = new MyStack<>();    // I only need two holding tracks for this.
        MyStack<Integer> S2 = new MyStack<>();

        input.add(4);           // In Java, the Queue interface's add() is addLast,
        input.add(3);           // while the Queue interface's remove() is removeFirst!
        input.add(1);
        input.add(8);
        input.add(2);
        input.add(7);
        input.add(6);
        input.add(9);
        input.add(5);

        System.out.println(input);    // The left side is the front of the input track here!
                                     // Due to Java's Queue using addLast and removeFirst,
                                     // input's printout will be [4, 3, 1, 8, 2, 7, 6, 9, 5].

        S1.push(input.remove());
        S1.push(input.remove());
        output.add(input.remove());    // These are steps 1 through 15, using the
        S2.push(input.remove());        // Queue interface's add(x) and remove() methods.
        output.add(input.remove());
        output.add(S1.pop());
        output.add(S1.pop());
        S2.push(input.remove());
        S2.push(input.remove());
        S1.push(input.remove());
    }
}

```

```

        output.add(input.remove());
        output.add(S2.pop());
        output.add(S2.pop());
        output.add(S2.pop());
        output.add(S1.pop());

        System.out.println(output);    // output's printout will be [1, 2, 3, 4, 5, 6, 7, 8, 9].
    }                                  // The left side is the front of the output track here!
}

```

4.b.

An example for a train of length 9 that cannot be rearranged in increasing order using 3 holding tracks:

```

[1, 9, 8, 7, 6, 5, 4, 3, 2]           // The right side is the front of the input track here!

```