Data Structures HW 2
Nap2152

1.

```
public static void printLots(Collection<Integer> L,
Collection<Integer> P) {
	int i = 0;
	for (Integer iter: L) {
		if (P.contains(i)) {
			System.out.println(iter);
		}
		i++;
	}
}
```

2.

```
list union(list L1, list L2) {
	list result
	int posL1 = 0
	int posL2 = 0
	while ( posL1 < L1.size() && posL2 < L2.size() ) {
		if ( L1.get(posL1) > L2.get(posL2) )
			posL2 ++
		else if ( L1.get(posL1) < L2.get(posL2) )
			posL1 ++
		else {
			result.add( result.size(), L1.get(posL1) )
			posL1 ++
			posL2 ++
		}
	}
	return result
}
```

3.

```
public class twoStackArray<AnyType> {
	int top1
	int top2
	int totalSize
	int sizeUsed
	AnyType[] array
	public twoStackArray(int arraySize) {
		array = new AnyType[arraySize]
		totalSize = arraySize
		top1 = 0
		top2 = totalSize – 1
		sizeUsed = 0
	}
```

```
public void push1(AnyType val) {
        if( sizeUsed == totalSize) {
                print(“There is an overflow, all elements in the list have been filled”)
        } else {
                array[top1] = val
                top1 ++
                sizeUsed ++
        }
}
public void push2(AnyType val) {
        if( sizeUsed == totalSize ) {
                print(“There is an overflow, all elements in the list have been filled”)
        } else {
                array[top2] = val
                top2 --
                sizeUsed ++
        }
}
public AnyType pop1() {
        if ( top1 != 0 ) {
                top1 --
                sizeUsed --
                return array[top1+1]
        } else {
                print(“Stack 1 is empty”)
                return null
        }
}
public AnyType pop2() {
        if ( top2 != (totalSize -1) ) {
                top2 ++
                sizeUsed --
                return array[top2-1]
        } else {
                print(“Stack 2 is empty”)
                return null
        }
}
public boolean isEmpty() {
        return (top1 == 0 && top2 == totalSize - 1)
}
public int size() {
        return sizeUsed
}
public AnyType getTop1() {
        if (top1 != 0)
```

```
                return Array[top1]
        else
                return null
    }
    public AnyType getTop2() {
        if (top2 != totalSize – 1)
                return Array[top2]
        else
                return null
    }
}
```

4.

(a)

- Move 4 to S_1
- Move 3 to S_1
- Move 1 to output track
- Move 8 to S_2
- Move 2 to output track
- Move 3 to output track
- Move 4 to output track
- Move 7 to S_2
- Move 6 to S_2
- Move 9 to S_1
- Move 5 to output track
- Move 6 to output track
- Move 7 to output track
- Move 8 to output track
- Move 9 to output track

(b)

A train of length 9 that cannot be rearranged in increasing order with three holding tracks is [1][9][8][7][6][5][4][3][2]

This is because the first car that goes into the output track is the back car in the original train. When putting cars on the holding tracks a higher number car cannot be placed on top of a lower number car. In order for this specific train to work there would need to be 8 holding cells, one to hold each of the cars until the first one.