Problem 1 (10 pts):

You are given a list, L , and a another list, P, containing integers sorted in ascending order. The operation printLots(L,P) will print the elements in L that are in positions specified by P. For instance, if P=1,3,4,6, the elements in positions 1,3,4, and 6 in L are printed. Write the procedure printLots(L,P). The code you provide should be the java method itself (not pseudocode), the containing class is not necessary. You may use only the public Collection (Links to an external site.) methods that are inherited by lists L and P. You may not use methods that are only in List. Anything that the Collection methods return is fair game, so you might think about how to use iterators.

```java
public static <AnyType> void printLots(List<AnyType> L, List<Integer> P){
		//will print the elements in L that are in positions specified by P

		Iterator<AnyType> itr = L.iterator();
		int index=0;
		while(itr.hasNext()){
		//ensures you do not search an empty list. (hasNext must be true).
			AnyType element=itr.next();
			for (Integer e: P){
				if (index==e){
				//i.e. if the position referred to in P is equal to the index of
				//the current element in L
					System.out.println(element);
				}
			}
			index++;
		}
	}
```

Problem 2 (10 pts):

Weiss 3.4 - provide java-like pseudocode for this problem: Given two sorted lists, L_1 and L_2, write a procedure to compute $L_1 \cap L_2$ using only the basic list operations.

- Create an empty list to add our intersecting values to:

```
List<AnyType> L3 = new ArrayList<AnyType>( );
```

- Create 3 iterators to iterate through L_1 and L_2. Keep the return type of the Iterator as the generic AnyType to be as flexible as possible.

```
Iterator<AnyType> iterator1 = L1.iterator( );
Iterator<AnyType> iterator2 = L2.iterator( );
```

- Set up a while clause to stop the function from running when the lists run out of elements, using the hasNext( ) method.

```
While (iterator1.hasNext( ) && iterator2.hasNext( )){
```

- Assign variables for the current element we are looking at in L_1 and L_2.

```
        element1=iterator1.next( );
        element2=iterator2.next( );
```

- Compare these variables using the compareTo method. If compareTo returns a 0, that means element 1 is equal to element2.

```
        If (element1.compareTo(element2)==0) {
```

- Now that we know we have found a shared element, we should then set the condition that L_3 must *not* already contain the element (in order to avoid adding a duplicate), using an enhanced for loop.

```
                For (AnyType x : L3) {

                        If(x.compareTo(element1) != 0) {
```

- With this final condition set, we add the element (either element 1 or 2 – in this case, element 1) to the new list L_3.

```
                                L3.add(x)
                        }
                }
        }
}
```

Problem 3 (10 pts):

Weiss 3.24 - provide java-like pseudocode for this problem: 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.

The most space-efficient solution is to put the two stacks at opposite ends of the array. Both stacks grow inwards – stack1 to the right, stack2 to the left. An overflow is declared when there is no longer any space between the top of stack1 (top1) and the top of stack2 (top2). The class will take the array size as input, and then assign that to the class variable "size". The class signature could be:

```
Public class (int n) {
    size = n;
    AnyType[] array = (AnyType[]) new Object[n];
}
```

PUSH METHODS – The methods push1(int x) and push2(int x), will allow us to add elements to our respective stacks. Both methods push1 and push2 take the integer x, and first check that there is space to place it at the top of the stack (with top2-1>top1). If not, stack overflow is declared and the program exits. Otherwise, the position of top increments or decrements by 1 respectively, and we add x to the top of the stack, using the value of top as the array's index.

```
Push1(int x) {

    If (top2-1)>top1 {
        top1++;
        array[top1]=x;
    }
    else {
        System.out.println("Stack overflow!");
        System.exit( )
    }
}

Push2(int x) {

    If (top2-1)>top1 {
        top2--;
        array[top2]=x;
    }
    else {
        System.out.println("Stack overflow!");
        System.exit( )
    }
}
```

POP METHODS – The methods pop1 and pop2 will allow us to pop elements from each stack. You cannot pop an empty stack, so first and foremost, the pop methods check that the top variable is greater than the beginning and end of the stack respectively. (If it is, the program quits). If the stack isn't empty, then the pop method assigns the value at top to a variable x, increments and decrements top respectively, and returns x. The reason why we do it in this order (rather than returning x and then incrementing/decrementing top) is that the return statement completes the execution of the if statement, meaning the change to top would be lost.

```
int pop1( ) {

        If (top1>0) {
                int x=array[top1];
                top1--;
                return x;
        }
        else {
                System.out.println("The stack is empty.");
                System.exit( );
        }

int pop2( ) {

        If (size>top2) {
                //n.b. that size is the end of the array. Top2 cannot still be there or else the stack must be
                empty.
                int x=array[top2];
                top1++;
                return x;
        }
        else {
                System.out.println("The stack is empty.");
                System.exit( );
        }
```

Problem 4 (10 pts):

(a)

Start: 596728134

1. 4 → S1
2. 3 → S1
3. 1 → S1
4. 1 → output track
5. 8 → S2
6. 2 → output track
7. 3 → output track
8. 4 → output track
9. 7 → S2
10. 6 → S2
11. 9 → S1
12. 5 → output track
13. 6 → output track
14. 7 → output track
15. 8 → output track
16. 9 → output track

End: 987654321

(b)

Start: 918237654

1. 4 → S1
2. 5 → S2
3. 6 → S3
4. 7 → breaks

7 is larger than 4, 5 and 6, and cannot stack on top of them. The program breaks.