```
/**********************************************************
 * EECS2011: Fundamentals of Data Structures, Fall 2016
 * Assignment 1, a1sol.pdf
 * Student Name:   Khushal Patel
 * Student cse account:  york18
 * Student ID number:  214037618
 **********************************************************/
```

Question 1: ArraySqueeze

General Idea: In order to squeeze an array, I have created an Integer ArrayList ar. Now I add all the numbers in ints to this array without any repetition. The code below does this.

```
"for(int i : ints)
{
      if(!ar.contains(i) || ar.lastIndexOf(i) != ar.size() - 1)
            {
                  ar.add(i);
            }
}"
```

The condition in if statement checks if there's already an element in ar or not. And even if there's an element already in the array then as long as that element is the last element in the array, add it. For example, ints = {1,1,1, 4,5,1,1} would be {1,4,5,1}.

Now, to add -1s, I take size difference between ar and ints. The difference between these two arrays gives me how many -1's to add in ar. After adding -1s to ar, I transfer all the elements of ar to ints. This will result in array getting squeezed and getting output as desired.

Output of this program:

```
Let's squeeze arrays!

squeezing [ 3 , 7 , 7 , 7 , 4 , 5 , 5 , 2 , 0 , 8 , 8 , 8 , 8 , 5 ]:
[ 3 , 7 , 4 , 5 , 2 , 0 , 8 , 5 , -1 , -1 , -1 , -1 , -1 , -1 ]

squeezing [ 6 , 6 , 6 , 6 , 6 , 3 , 6 , 3 , 6 , 3 , 3 , 3 , 3 , 3 , 3 ]:
[ 6 , 3 , 6 , 3 , 6 , 3 , -1 , -1 , -1 , -1 , -1 , -1 , -1 , -1 , -1 ]

squeezing [ 4 , 4 , 4 , 4 , 4 ]:
[ 4 , -1 , -1 , -1 , -1 ]

squeezing [ 0 , 1 , 2 , 3 , 4 , 5 , 6 ]:
[ 0 , 1 , 2 , 3 , 4 , 5 , 6 ]


Additional tests done by the student or TA:
```

Question 2: ArrayLongestPlateau

General Idea: For this question. I am considering three cases:

1. If the given array ints has duplicate elements (For example, {1,2,5,5,4,6,6,7,8,})
2. If the given array ints has no duplicate elements (For example, {4,5,7,1,4,8}) and
3. A special case when length of ints is 1.

The code for 1st case:

```
"for(int i = 0; i < ints.length - 1; i++)
{
	if(ints[i] == ints[i + 1])
	{
		frequency++;
		index = i + 1;
		if((i == 0 || lastValue <= ints[i]) && (index == ints.length - 1
			|| ints[index] > ints[index + 1]) && frequency >
bestFrequency)
		{
			condition = true;
			if(lastIndex != 0)
			{
				start = lastIndex + 1;
			}
			bestFrequency = frequency;
			value = ints[i];
		}
		else
		{
			lastValue = ints[i];
			lastIndex = i;
			frequency = 1;
		}
}"
```

So, for the first case, I am using a for loop that traverses all the elements in ints and checks if any sub-array satisfies the plateau condition. I.e., if an element before the first element of sub-array is less than itself or if it's index is 0 and if the last element of the sub-array is either the last element of the array ints or the integer after that is less than itself.

Along with all this, it also keeps track of the frequency. I.e., how many times that integer has repeated itself (in continuity). For example, if ints is {4,4,4,4,3,5,6,6,2}. Then two possible plateaus are [4,4,4,4] and [6,6] but since sixes are only repeated twice and fours are repeated four times, it will not consider sixes and only consider fours. This is what the condition inside the if statement in the above code essentially does.

For the second case, we know that no numbers are repeated or even if they are repeated, they don't satisfy the plateau condition. This is exactly why I have "boolean condition;" in my code. For this case, all I need to do is find the greatest element in the array ints and note its index and value. Frequency will be 1 since we know that elements are not repeated.

For the last case, we have an array of length 1. So the longest plateau would be of course the single element. So, I note its value and index and frequency would be obviously 0 and 1 respectively.

Output for this program:

```
Let's find longest plateaus of arrays!

longest plateau of [ 4 , 1 , 1 , 6 , 6 , 6 , 6 , 1 , 1 ]:
[ value , start , len ] = [ 6 , 3 , 4 ]

longest plateau of [ 3 , 3 , 1 , 2 , 4 , 2 , 1 , 1 , 1 , 1 ]:
[ value , start , len ] = [ 3 , 0 , 2 ]

longest plateau of [ 3 , 3 , 1 , 2 , 4 , 0 , 1 , 1 , 1 , 1 ]:
[ value , start , len ] = [ 1 , 6 , 4 ]

longest plateau of [ 3 , 3 , 3 , 4 , 1 , 2 , 4 , 4 , 0 , 1 ]:
[ value , start , len ] = [ 4 , 6 , 2 ]

longest plateau of [ 7 , 7 , 7 , 7 , 9 , 8 , 2 , 5 , 5 , 5 , 0 , 1 ]:
[ value , start , len ] = [ 5 , 7 , 3 ]

longest plateau of [ 4 ]:
[ value , start , len ] = [ 4 , 0 , 1 ]

longest plateau of [ 4 , 4 , 4 , 5 , 5 , 5 , 6 , 6 ]:
[ value , start , len ] = [ 6 , 6 , 2 ]


Additional tests done by the student or TA:

longest plateau of [ 4 , 2 ]:
[ value , start , len ] = [ 4 , 0 , 1 ]

longest plateau of [ 2 , 4 ]:
[ value , start , len ] = [ 4 , 1 , 1 ]

longest plateau of [ 1 , 2 , 3 , 4 , 4 , 5 , 6 ]:
[ value , start , len ] = [ 6 , 6 , 1 ]

longest plateau of [ 1 , 6 , 3 , 8 , 8 , 4 , 5 , 5 , 5 , 1 , 9 , 6 , 7 , 7 ]:
[ value , start , len ] = [ 5 , 6 , 3 ]

longest plateau of [ 3 , 3 , 3 , 3 , 1 , 8 , 2 , 5 , 5 ]:
[ value , start , len ] = [ 3 , 0 , 4 ]

longest plateau of [ 3 , 3 , 4 , 5 , 5 , 6 , 1 ]:
```

```
[ value , start , len ] = [ 6 , 5 , 1 ]
```

I have added few test cases to consider different possibilities.

Question 3: Window

General Idea:

(1) Boolean encloses(Window w) method:

```
public boolean encloses(Window w)
{
	if(this.left <= w.left && this.right >=  w.right && this.bottom
	<= w.bottom && this.top >= w.top
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

To know if a window win encloses window w, no part of w should be outside win. And if two window objects have same fields than they both encloses each other. So, for win to enclose w, win's left should be less than or equal to w's left and it's right should be greater than or equal to w's right. Same goes for bottom and top.

(2) Boolean overlaps(Window w) method:

```
public boolean overlaps(Window w)
{
	if(((w.right > this.left && w.left < this.right) && (w.bottom <
	this.top && w.top > this.bottom)) || this.encloses(w))
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

For two window objects to overlap, some of their boundaries must intersect. Just touching boundaries is not enough. Pair of windows that encloses also overlaps (since they have common interior point).

My main method runs some test cases:

```
public static void main(String[] args) throws InvalidWindowException
    {
      Window w1 = new Window(2, 3, 1, 5);
      Window w2 = new Window(1, 3, 3, 4);
      Window w3 = new Window(1, 3, 3, 5);
      Window w4 = new Window(4, 6, 1, 4);
      Window w5 = new Window(2, 3, 1, 4);
      Window w6 = new Window(2.5, 3, 2, 3);
      Window w7 = new Window(2, 3, 1, 5);
      Window w8 = new Window(2, 3, 0, 1);
      Window[] windows = {w1,w2,w3,w4,w5,w6,w7};

      System.out.println("Testing overlap and overlapCount method" ); //it
checks both the overlaps and overlapCount works since overlapCount uses
overlaps method
      System.out.println("Total Count: " + Window.overlapCount(windows));
      String Test1 = (Window.overlapCount(windows) == 13)? "Passed" :
"Failed";
      System.out.println(Test1);

      System.out.println("Testing encloses and enclosureCount method:");
      System.out.println("Total Count: " + Window.enclosureCount(windows));
      String Test2 = (Window.enclosureCount(windows) == 8)? "Passed" :
"Failed";
      System.out.println(Test2);

      System.out.println("Testing overlaps method (special case): "); //to
check if two windows whose boundaries touch, overlaps
      if(w1.overlaps(w8) == true)
      {
            System.out.println("Failed");
      }
      else
      {
            System.out.println("Passed");
      }

      System.out.println("Testing encloses method (special case): "); // to
check if two objects of same field encloses one another
      if(w1.encloses(w7) && w7.encloses(w1))
      {
            System.out.println("Passed");
      }
      else
      {
            System.out.println("Failed");
      }
    }
```

Output:

```
Testing overlap and overlapCount method
Total Count: 13
```

```
Passed
Testing encloses and enclosureCount method:
Total Count: 8
Passed
Testing overlaps method (special case):
Passed
Testing encloses method (special case):
Passed
```