Partition

Implement `partition`, which takes in an `IntList lst` and an integer `k`, and *destructively* partitions `lst` into `k IntLists` such that each list has the following properties: Firstly, It is the **same** length as the other lists. If this is not possible, i.e. `lst` cannot be equally partitioned, then the later lists should be **one** element smaller. For example, partitioning an `IntList` of length 25 with `k = 3` would result in partitioned lists of lengths 9, 8, and 8. Secondly, its ordering is consistent with the ordering of `lst`, i.e. items in earlier in `lst` must **precede** items that are later.

These lists should be put in an array of length `k`, and this array should be returned. For instance, if `lst` contains the elements 5, 4, 3, 2, 1, and `k = 2`, then a **possible** partition (note that there are many possible partitions), is putting elements 5, 3, 2 at index 0, and elements 4, 1 at index 1.

You may assume you have the access to the method `reverse`, which destructively reverses the ordering of a given `IntList` and returns a pointer to the reversed `IntList`. You may not create any `IntList` instances. You may not need all the lines.

Hint: You may find the % operator helpful.

```
1  public static IntList[] partition(IntList lst, int k) {
2      IntList[] array = new IntList[k];
3      int index = 0;
4      IntList L = ____________________________________
5      while (L != null) {
6
7          ____________________________________________
8
9          ____________________________________________
10
11         ____________________________________________
12
13         ____________________________________________
14
15         ____________________________________________
16
17         ____________________________________________
18
19         ____________________________________________
20     }
21     return array;
22 }
```