Practice Problems

For each problem, produce a test suite using each strategy, separately:

a) specification-based testing (partitions and boundaries)

b) structure-based testing exclusively (MC/DC for the cases with multiple conditions, branch + loop coverage for the rest)

c) property based testing.

1. `sum`

```java
/**
 * This method takes two integers and returns their sum.
 * The numbers have to be between 1 and 99 (both inclusive).
 *
 * @param n and m integers
 * @return their sum
 */
public int sum(int n, int m) {
    if (n < 1 || n > 99 || m < 1 || m > 99) {
        throw new IllegalArgumentException();
    }
    return n + m;
}
```

```
Specification:
Individual partitions for n and m are:
- <1
- >99
- in [1,99] interval (inclusive on both ends)
Boundary:
- 0 and 1
- 99 and 100
Combined tests:
- Valid+Non-valid:(50,-50), (-50,50), (50,150),(150,50)
- Valid+Boundary: (50,0),(50,1),(50,99),(50,100),(0,50),(1,50),(99,50),(100,50)
Structure: - with MC/DC we get 5:
- (50,50)
- (-50,50)
- (250,50)
- (50,-50)
- (50,150)
Property:
- generate n to be invalid (intervals <1 or >99) and m valid ([1,99] interval)
- generate n valid ([1,99] interval) and m to be invalid (intervals <1 or >99)
- generate n and m to be valid ([1,99] interval)
```

2. `lastIndexOf`

```java
public static final int INDEX_NOT_FOUND = -1;
/**
 * Finds the last index of the given value in the array starting at the given
 * index.
 *
 * <p>
 * This method returns {@link #INDEX_NOT_FOUND} ({@code -1}) for a {@code null}
 * input array.
 *
 * <p>
 * A negative startIndex will return {@link #INDEX_NOT_FOUND} ({@code -1}). A
 * startIndex larger than the array length will search from the end of the
 * array.
 *
 * @param array
 *            the array to traverse for looking for the object, may be
 *            {@code null}
 * @param valueToFind
 *            the value to find
 * @param startIndex
 *            the start index to traverse backwards from
 * @return the last index of the value within the array,
 *         {@link #INDEX_NOT_FOUND} ({@code -1}) if not found or {@code null}
 *         array input
 */
public static int lastIndexOf(final int[] array, final int valueToFind,
                                                        int startIndex) {
    if (array == null) {
        return INDEX_NOT_FOUND;
    }
    if (startIndex < 0) {
        return INDEX_NOT_FOUND;
    } else if (startIndex >= array.length) {
        startIndex = array.length - 1;
    }
    for (int i = startIndex; i >= 0; i--) {
        if (valueToFind == array[i]) {
            return i;
        }
    }
    return INDEX_NOT_FOUND;
}
```

```
Start/ individual partition
array:
- null
- empty
- single
- multiple
startIndex (and in relation to array)
- negative
- positive less than the size of array
- positive equal to the size of the array (boundary)
- positive greater than the size of array
relation between all 3 params
- value in the array once before startIndex
- value in the array once after startIndex
- value once at startIndex (boundary)
- value multiple times before startIndex
- value multiple time both before and after startIndex
Combined partitions:
- array null: (null, 1, 2) -> INDEX_NOT_FOUND
- empty array: ([], 0, 1) -> INDEX_NOT_FOUND
- negative index: ([0, 1, 2], 2, -1) -> INDEX_NOT_FOUND
- index bigger than array: ([0, 1, 2], 1, 5) -> 1
- length one array with element: ([1], 1, 0) -> 0
- length one array without element: ([1], 2, 0) -> INDEX_NOT_FOUND
- array with element: ([0, 1, 2], 1, 2) -> 1
- array with element many times: ([0, 1, 1, 2], 1, 2) -> 2
- array without element: ([0, 1, 2], 3, 2) -> INDEX_NOT_FOUND
- array with element, start index == 0: ([0, 1, 2], 0, 0) -> 0
- array without element, start index == 0: ([0, 1, 2], 3, 0) -> INDEX_NOT_FOUND
- element at the index: ([0, 1, 1, 2], 1, 1) -> 1
Structure tests:
- null array
- negative startIndex
- array.length = startIndex
- empty array (skip loop)
- ([0,1], 1, 1) -> 1 (loop once)
- ([1,0], 1, 1) -> 0 (loop multiple times)
Property testing:
- generate an array of numbers from a specific range (of size k)
- generate the valueToFind from a range outside the array range
(this ensures that it is unique)
- generate two indexes between 0 and k: one to insert valueToFind
at and one to represent the startIndex
- insert valueToFind in the array, run lastIndexOf and check that
it returns the expected index
```

3. zigzag

```java
public static final int INDEX_NOT_FOUND = -1;
/**
 * This method receives a string s and a number of rows numRows,
 * and writes it down in a zigzag pattern. For example,
 * for s="PAYPALISHIRING", and numRows=4, the function returns
 * P  I  N
 * A LS IG
 * YA HR
 * P  I
 */
public String zigzag(String s, int numRows) {
    // some pre-condition check
    if(s.length() < 1 || s.length() > 1000)
        throw new IllegalArgumentException("1 <= s.length <= 1000");
    if(numRows < 1 || numRows > 1000)
        throw new IllegalArgumentException("1 <= numRows <= 1000");

    // early return: if the number of rows is 1, then, we return the same string
    if (numRows == 1) return s;

    // We create a list of strings, based on the number of rows we need
    List<StringBuilder> rows = new ArrayList<>();
    for (int i = 0; i < Math.min(numRows, s.length()); i++)
        rows.add(new StringBuilder());

    int curRow = 0;
    boolean goingDown = false;

    // We visit character by character, and we put it in the list of strings.
    // We change directions whenever we reach the top or the bottom of the list.
    for (char c : s.toCharArray()) {
        // add the letter
        rows.get(curRow).append(c);

        // are we at the top or the bottom of the list?
        boolean topOrBottom = curRow == 0 || curRow == numRows - 1;

        // add spaces if we are 'zagging'
        if(!goingDown && !topOrBottom) {
            for(int i = 0; i < rows.size(); i++) {
                if(i!=curRow)
                    rows.get(i).append(" ");
            }
        }

```

```
46          // invert the direction in case we reached the top or the bottom
47          if (topOrBottom) goingDown = !goingDown;
48
49          // go to the next current row
50          curRow += goingDown ? 1 : -1;
51      }
52
53      // we return the final string by simply combining all
54      // the stringbuilders into a single string
55      return rows
56              .stream()
57              .map(x->x.toString().trim())
58              .collect(Collectors.joining("\n"))
59              .trim();
60  }
```

```
Start/ separate partitions:
s:
- null (note that this will crash the program)
- empty
- single
- multiple (less, equal, more than 1000 characters)
numRows:
- negative, 0, 1
- 999, 1000, 1001
relation between s and numRows
- s.length < numRows
- s.length == numRows
- s.length > numRows
Combined partitions:
- empty string -> IAE
- s with 1001 characters -> IAE
- numRow == 0 -> IAE
- numRow == 1001 -> IAE
- numRow == 1 -> s (the entire string on one row)
- s is 'abc', numRow is 1000 -> 'a\nb\nc'
- ('abc', 4) -> 'a\nb\nc'
- single row with one and 2 character strings
- multiple rows with the same string
Structural testing:
- 5 tests for the preconditions
- 3-5 tests for the for loops (note that neither can be skipped)
Property testing:
- generate width of top row (say between 2 and 5)
- generate numRows (also between 2 and 5)
- based on these construct both input and expected output (where each row is one letter)
```

4. `isItSummer`

```java
/**
 * This method predicts whether it is summer.
 * If at least 75% of the temperature values provided are 20 degrees
 * or above, it is summer. Otherwise, it is not summer.
 *
 * @param temperatures The list of temperature values
 * @return the probability of it being summer
 */
public static boolean isItSummer(List<Double> temperatures) {
    int count20OrAbove = 0;

    for (Double temp : temperatures) {
        if (temp >= 20) {
            count20OrAbove++;
        }
    }

    return count20OrAbove >= temperatures.size() * 0.75f;
}
```

Specification: null, empty, more than 75%, less than 75%, exactly 75% as on-point
Structure: 3 for the loop (empty list, 1 element, 2+ elements), 2 for the inner-loop branch, and 2 for the return branch (can be further pruned)
Property: generate 2 lists, one less than 20 and the other greater than 20 with different contents (less than 75, more than 25)